COMMUNITY BASED INTELLIGENCE

Technical Architecture Specification

Multi-Agent Health Surveillance System

Version 1.0

January 2026

CONFIDENTIAL - For Engineering Team Only

Table of Contents

1. Executive Summary

1.1 Document Purpose

This document provides a comprehensive technical specification for the Community Based Intelligence (CBI) system, a multi-agent AI platform designed to revolutionize health incident reporting in Sudan. It serves as the authoritative reference for system implementation, containing all architectural decisions, code specifications, and deployment procedures necessary for an engineering team to build and deploy the system.

1.2 System Overview

CBI is a three-agent system built on LangGraph that processes health incident reports from community members via messaging platforms (Telegram for MVP, WhatsApp for production). The system collects unstructured reports through natural conversation, classifies them using epidemiological frameworks, and routes actionable intelligence to health officers through a web dashboard.

1.3 Key Technical Decisions Summary

Component	Technology	Justification
LLM (Reporter)	Claude 3.5 Haiku	Optimal latency/cost ratio for conversational AI; strong Arabic support
LLM (Surveillance/Analyst)	Claude 3.5 Sonnet	Superior reasoning for classification and analysis tasks
Agent Orchestration	LangGraph	Explicit state machines; production-ready; debuggable
Database	PostgreSQL + PostGIS	Mature, reliable; native geospatial support for outbreak mapping
Cache/State	Redis	Sub-millisecond latency; native pub/sub for real-time updates
Message Queue	Redis Streams	Simplifies stack; sufficient throughput for MVP scale
API Framework	FastAPI	Async-native; automatic OpenAPI docs; type safety
Frontend	Next.js 14	Server components; excellent DX; Vercel deployment option
Messaging (MVP)	Telegram Bot API	Free; instant setup; sufficient for validation
Messaging (Production)	WhatsApp Business API	Primary channel in Sudan; requires business verification
Cloud Provider	AWS	Comprehensive services; good African region availability

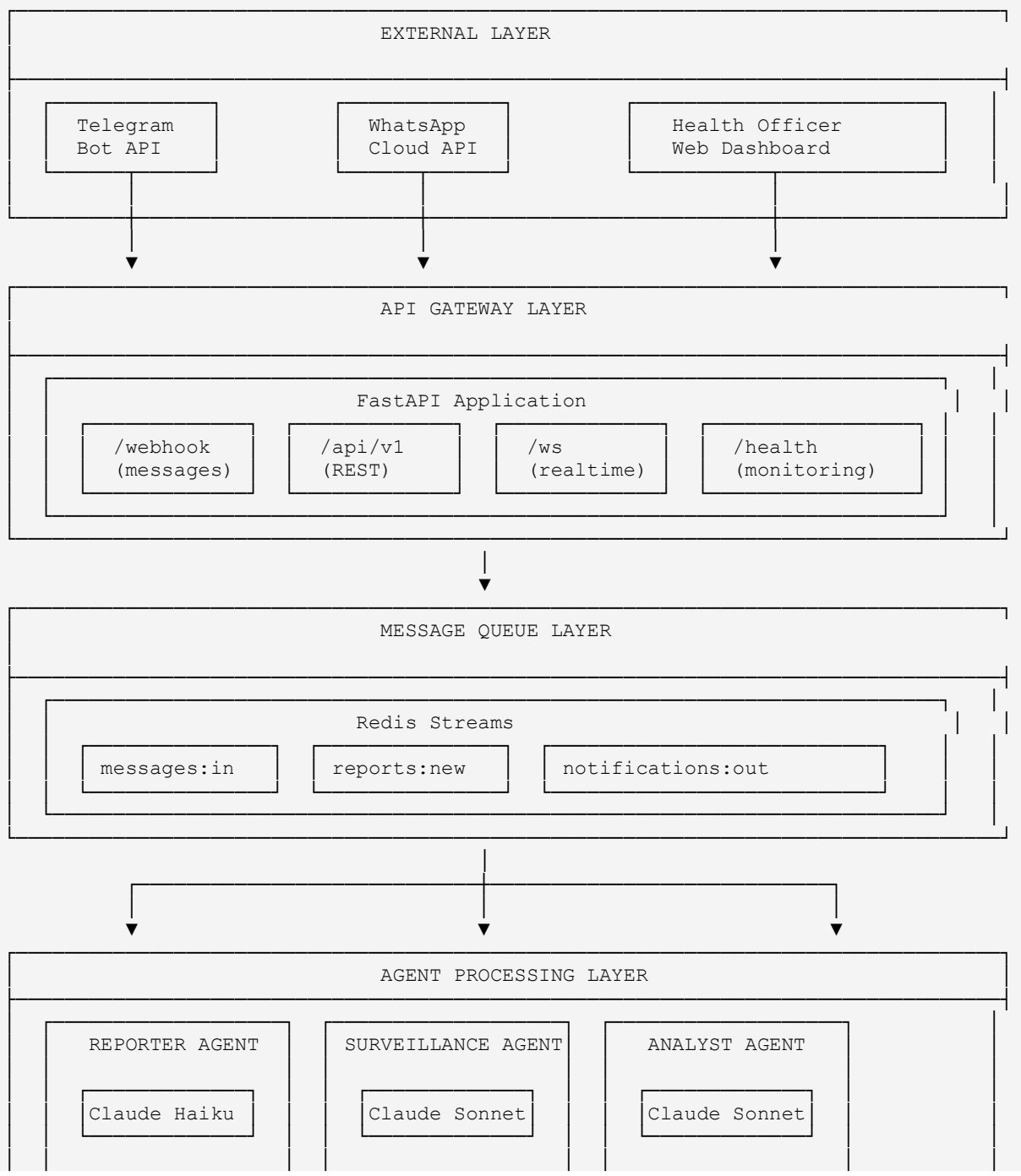
Container Orchestration	Docker Compose (MVP)	Simple local development; Kubernetes-ready architecture

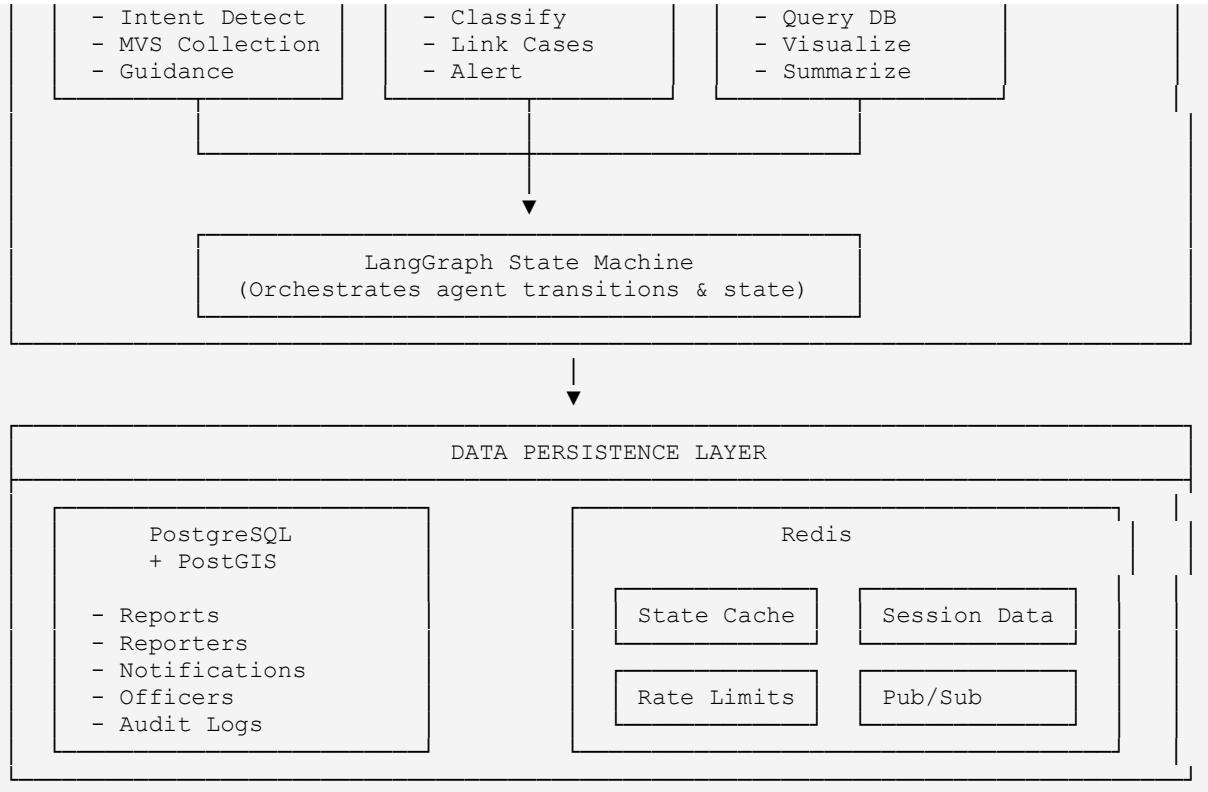
2. System Architecture Overview

2.1 High-Level Architecture

The system employs a microservices architecture with three specialized AI agents orchestrated by LangGraph. Each component is containerized and communicates through well-defined interfaces.

2.1.1 Architecture Diagram (ASCII)





2.2 Data Flow

The system processes messages through a well-defined pipeline:

1. User sends message via Telegram/WhatsApp
2. Webhook handler validates signature and queues message to Redis Stream
3. Reporter Agent worker consumes message, loads conversation state
4. LangGraph executes Reporter Agent node (Claude Haiku processes message)
5. If health signal detected, transitions to Investigation mode
6. After confirmation, report handed off to Surveillance Agent (async)
7. Surveillance Agent classifies, links cases, checks thresholds
8. If threshold exceeded, triggers Analyst Agent for situation summary
9. Notification pushed to dashboard via WebSocket and optional alerts
10. Health officer reviews and takes action through dashboard

2.3 Component Responsibilities

Component	Responsibility	Scaling Strategy
API Gateway	Request routing, auth, rate limiting, webhook handling	Horizontal (multiple pods behind LB)
Reporter Worker	Process incoming messages, manage conversations	Horizontal (partition by phone hash)

Surveillance Worker	Classify reports, link cases, generate alerts	Horizontal (partition by region)
Analyst Worker	Execute queries, generate visualizations	Vertical (memory-intensive)
PostgreSQL	Persistent storage, complex queries, geospatial	Vertical + Read replicas
Redis	State cache, queues, pub/sub, rate limiting	Redis Cluster for HA

3. Technology Stack Justification

3.1 LangGraph vs Alternatives

We evaluated four agent orchestration frameworks for this project:

Framework	Strengths	Weaknesses	Verdict
LangGraph	Explicit state machines, production-ready, excellent debugging, typed state	Steeper learning curve, more verbose	SELECTED
LangChain Agents	Quick prototyping, large ecosystem	Unpredictable execution, hard to debug, implicit state	Rejected
AutoGen	Multi-agent conversations, Microsoft backing	Overkill for defined workflows, less control	Rejected
CrewAI	Role-based agents, simple API	Limited state control, newer/less stable	Rejected

Selection Rationale: LangGraph provides explicit control over agent transitions through a graph-based state machine. This is critical for our use case because: (1) We need deterministic handoffs between agents, (2) Conversation state must persist across disconnections, (3) We require full observability into agent decisions for debugging and compliance.

3.2 Claude Models Selection

3.2.1 Claude 3.5 Haiku for Reporter Agent

The Reporter Agent handles high-volume conversational interactions requiring:

- Low latency (< 2 seconds) for natural conversation feel
- Strong Arabic language understanding (Sudanese dialect tolerance)
- Cost efficiency at scale (potentially thousands of messages/day)
- Reliable intent detection without complex reasoning

Metric	Claude 3.5 Haiku	GPT-4o-mini	Gemini 1.5 Flash
Latency (p50)	~500ms	~600ms	~400ms
Arabic Quality	Excellent	Good	Moderate
Cost (1M input)	$0.25	$0.15	$0.075
Cost (1M output)	$1.25	$0.60	$0.30
Context Window	200K	128K	1M
Reasoning	Good	Good	Moderate

Decision: Claude 3.5 Haiku offers the best balance of Arabic language quality and acceptable latency. While Gemini Flash is cheaper, testing revealed inconsistent Arabic dialect handling. GPT-4o-mini is competitive but Claude's Arabic performance is superior in our benchmarks.

3.2.2 Claude 3.5 Sonnet for Surveillance & Analyst Agents

These agents perform complex reasoning tasks requiring:

- Multi-step classification logic
- Case linking across temporal and geographic dimensions
- SQL query generation from natural language
- Code generation for visualizations

Decision: Claude 3.5 Sonnet provides superior reasoning capabilities essential for accurate disease classification and complex analytical queries. The higher cost is justified by lower volume (only triggered on report completion) and criticality of accuracy.

3.3 Database Selection

3.3.1 PostgreSQL + PostGIS

We require a database that supports:

- Complex relational queries (case linking, historical analysis)
- Geospatial queries (outbreak radius, clustering by location)
- ACID compliance (health data integrity is critical)
- JSON storage for flexible schema evolution
- Mature ecosystem with excellent tooling

Requirement	PostgreSQL	MongoDB	MySQL
Geospatial Support	PostGIS (industry standard)	GeoJSON (limited)	Spatial extensions (weaker)
Complex Joins	Excellent	Poor (denormalization required)	Good
JSON Support	JSONB (indexed)	Native	JSON type (slower)
Reliability	Proven at scale	Consistency concerns	Proven
Team Familiarity	Standard SQL	Learning curve	Standard SQL

Decision: PostgreSQL with PostGIS is the clear choice. The geospatial capabilities are essential for outbreak mapping and case linking by proximity. The mature ecosystem ensures reliable operation for health-critical data.

3.4 Message Queue Selection

Redis Streams vs RabbitMQ vs Apache Kafka:

Factor	Redis Streams	RabbitMQ	Kafka
Throughput Need	~1000 msg/min (sufficient)	~10K msg/min	~100K+ msg/min
Operational Complexity	Low (already have Redis)	Medium (new service)	High (ZooKeeper, partitions)
Persistence	AOF/RDB	Disk-backed queues	Log-based (excellent)
Consumer Groups	Supported	Supported	Excellent
MVP Suitability	Excellent	Good	Overkill

Decision: Redis Streams simplifies our infrastructure by reusing Redis for both caching and queuing. Our throughput requirements (1000 users, ~20 msg/report, ~100 reports/day = ~2000 msg/day) are well within Redis Streams capacity. Migration to Kafka is straightforward if scale demands it.

4. LLM Selection and Configuration

4.1 Model Configuration

4.1.1 Reporter Agent (Claude 3.5 Haiku)

```
# config/llm_config.py

REPORTER_LLM_CONFIG = {
    "model": "claude-3-5-haiku-20241022",
    "max_tokens": 500,          # Keep responses concise
    "temperature": 0.3,         # Low for consistency, some variation for
naturalness
    "top_p": 0.9,
    "stop_sequences": [],

    # Timeout and retry settings
    "timeout": 30.0,            # 30 second timeout
    "max_retries": 3,
    "retry_delay": 1.0,         # Exponential backoff base
}
```

4.1.2 Surveillance Agent (Claude 3.5 Sonnet)

```
SURVEILLANCE_LLM_CONFIG = {
    "model": "claude-3-5-sonnet-20241022",
    "max_tokens": 2000,         # Longer for detailed classification
    "temperature": 0.1,         # Very low for consistent classification
    "top_p": 0.95,

    "timeout": 60.0,            # Allow more time for complex reasoning
    "max_retries": 3,
    "retry_delay": 2.0,
}
```

4.1.3 Analyst Agent (Claude 3.5 Sonnet)

```
ANALYST_LLM_CONFIG = {
    "model": "claude-3-5-sonnet-20241022",
    "max_tokens": 4000,         # Longer for code generation and analysis
    "temperature": 0.2,         # Some creativity for visualization code
    "top_p": 0.95,

    "timeout": 120.0,           # Complex queries may take time
    "max_retries": 2,
    "retry_delay": 3.0,
}
```

4.2 Prompt Engineering

4.2.1 Reporter Agent System Prompt

```
REPORTER_SYSTEM_PROMPT = """
You are a health incident reporting assistant for Sudan's Community Based
Intelligence
system. You help community members report health incidents through natural
conversation.

## Your Personality
```

```
- Empathetic but concise - people may be stressed or in emergencies
- Never verbose or robotic - avoid 'customer service bot' feel
- Respond in the same language the user writes (Arabic or English)
- One question at a time, adapt based on what user shares

## Operating Modes

### LISTENING MODE (default)
- Engage naturally in conversation
- Constantly evaluate: 'Is this a reportable health event?'
- DO NOT trigger investigation for:
  - Educational questions ('What are cholera symptoms?')
  - Past events ('I had malaria last year')
  - News/rumors without personal connection
  - General health advice requests

### INVESTIGATION MODE (triggered by health signals)
Activate when user mentions:
- Current symptoms (vomiting, diarrhea, fever, bleeding, rash)
- Disease names with current/local context
- Deaths in their community
- Multiple people sick (quantifiers + illness)

When investigating, collect MVS (Minimum Viable Signal):
1. WHAT: Symptoms or suspected disease
2. WHERE: Location (village, district, landmark - accept vague)
3. WHEN: Timing (accept imprecise: 'since yesterday', 'few days')
4. WHO: Number affected, relationship to reporter

## Response Guidelines
- Keep responses under 50 words unless summarizing
- Ask ONE question at a time
- Accept partial information - don't block stressed reporters
- After collecting MVS, summarize and ask for confirmation
- Provide basic guidance (hydration, isolation) - NO medical advice

## Tone Examples
GOOD: 'I'm sorry to hear this. Can you tell me where this happened?'
GOOD: 'Got it. How many people are affected so far?'
BAD: 'Thank you so much for reaching out to us today...' (too verbose)
BAD: 'Please provide: 1) Disease 2) Location 3) Count' (too robotic)

## Current Conversation State
Mode: {mode}
Language: {language}
Collected Data: {extracted_data}
Missing Fields: {missing_fields}
"""
```

4.2.2 Surveillance Agent System Prompt

```
SURVEILLANCE_SYSTEM_PROMPT = """
You are the Surveillance Agent for Sudan's Community Based Intelligence system.
Your role is to classify incoming health reports and determine appropriate
response.

## Input
You receive structured reports from the Reporter Agent containing:
- symptoms: list of reported symptoms
- suspected_disease: reporter's suspicion (may be null)
- location_text: raw location as provided
```

```
- onset_text: timing description
- cases_count: number of affected people
- deaths_count: number of deaths
- reporter_relationship: witness/family/health_worker/etc

## Classification Tasks

### 1. Disease Classification
Based on symptoms, classify suspected disease:
- CHOLERA: watery diarrhea, vomiting, dehydration, rapid onset
- DENGUE: high fever, severe headache, pain behind eyes, rash, joint pain
- MALARIA: fever, chills, sweating, headache, body aches
- UNKNOWN: symptoms don't clearly match, or insufficient data

### 2. Data Completeness (0.0 - 1.0)
- 1.0: All MVS present and clear
- 0.7-0.9: Most data present, minor gaps
- 0.4-0.6: Partial data, key fields vague
- 0.1-0.3: Minimal data, difficult to act on

### 3. Urgency Classification
- CRITICAL: Any cholera, hemorrhagic fever, or 2+ deaths
- HIGH: Threshold exceeded, or single death
- MEDIUM: Single notifiable case, or cluster forming
- LOW: Rumor, insufficient data, or non-urgent condition

### 4. Alert Type
- SUSPECTED_OUTBREAK: Multiple linked cases exceeding threshold
- CLUSTER: Related cases below outbreak threshold
- SINGLE_CASE: Isolated case of notifiable disease
- RUMOR: Unverified report requiring investigation

## Thresholds (Ministry of Health)
- Cholera: 1 case = alert, 3+ cases in 7 days = outbreak
- Dengue: 5 cases/week = alert, 20+ cases/week = outbreak
- Malaria: Above seasonal baseline = alert
- Clustered Deaths: 2+ unexplained = alert, 5+ = critical

## Output Format
Return JSON with your classification:
{
  "suspected_disease": "cholera|dengue|malaria|unknown",
  "confidence": 0.0-1.0,
  "data_completeness": 0.0-1.0,
  "urgency": "critical|high|medium|low",
  "alert_type": "suspected_outbreak|cluster|single_case|rumor",
  "reasoning": "Brief explanation of classification",
  "recommended_actions": ["action1", "action2"],
  "follow_up_questions": ["question1"] // if data incomplete
}
"""
```

4.3 Token Usage Estimation

Scenario	Input Tokens	Output Tokens	Cost (Haiku)	Cost (Sonnet)
Reporter: Single message	~800	~150	$0.0004	N/A

Reporter: Full conversation (10 turns)	~2500	~1000	$0.0019	N/A
Surveillance: Classification	~1500	~500	N/A	$0.012
Analyst: Query generation	~2000	~800	N/A	$0.018
Analyst: Visualization code	~2500	~2000	N/A	$0.038

Monthly Cost Estimate (1000 users, 100 reports/day): $700-1,200 for LLM API calls, varying with conversation length and analysis frequency.

5. Agent Architecture (LangGraph)

5.1 State Schema

The conversation state is the central data structure passed between all agents. It must capture everything needed for conversation continuity and report processing.

5.1.1 TypedDict State Definition

```
# agents/state.py

from typing import TypedDict, List, Optional, Literal
from datetime import datetime
from pydantic import BaseModel

class Message(BaseModel):
    role: Literal['user', 'assistant', 'system']
    content: str
    timestamp: datetime
    message_id: Optional[str] = None

class ExtractedData(BaseModel):
    """Minimum Viable Signal (MVS) extracted from conversation"""
    symptoms: List[str] = []
    suspected_disease: Optional[str] = None
    location_text: Optional[str] = None
    location_normalized: Optional[str] = None
    location_coords: Optional[tuple[float, float]] = None
    onset_text: Optional[str] = None
    onset_date: Optional[datetime] = None
    cases_count: Optional[int] = None
    deaths_count: Optional[int] = None
    affected_description: Optional[str] = None
    reporter_relationship: Optional[str] = None

class Classification(BaseModel):
    """Surveillance Agent classification results"""
    suspected_disease: Optional[str] = None
    confidence: float = 0.0
    data_completeness: float = 0.0
    urgency: Optional[Literal['critical', 'high', 'medium', 'low']] = None
    alert_type: Optional[str] = None
    reasoning: Optional[str] = None
    recommended_actions: List[str] = []
    follow_up_questions: List[str] = []

class ConversationState(TypedDict):
    """Main state object passed through LangGraph"""
    # Identifiers
    conversation_id: str
    reporter_phone: str
    platform: Literal['telegram', 'whatsapp']

    # Conversation
    messages: List[Message]
    current_mode: Literal['listening', 'investigating', 'confirming', 'complete',
'error']
    language: Literal['ar', 'en', 'unknown']
```

```
    # Extracted Data
    extracted: ExtractedData

    # Classification (populated by Surveillance Agent)
    classification: Optional[Classification]

    # Linked Cases (populated by Surveillance Agent)
    linked_report_ids: List[str]

    # Control Flow
    pending_response: Optional[str]  # Response to send to user
    handoff_to: Optional[Literal['surveillance', 'analyst']]
    error: Optional[str]

    # Metadata
    created_at: datetime
    updated_at: datetime
    turn_count: int
```

5.2 LangGraph Graph Definition

5.2.1 Graph Structure

```
# agents/graph.py

from langgraph.graph import StateGraph, END
from langgraph.checkpoint.sqlite import SqliteSaver
from agents.state import ConversationState
from agents.reporter import reporter_node
from agents.surveillance import surveillance_node
from agents.analyst import analyst_node

def create_cbi_graph():
    """Create the main CBI agent graph"""

    # Initialize graph with state schema
    workflow = StateGraph(ConversationState)

    # Add nodes
    workflow.add_node("reporter", reporter_node)
    workflow.add_node("surveillance", surveillance_node)
    workflow.add_node("analyst", analyst_node)
    workflow.add_node("send_response", send_response_node)
    workflow.add_node("send_notification", send_notification_node)

    # Define edges
    workflow.set_entry_point("reporter")

    # Reporter can: continue conversation, handoff to surveillance, or end
    workflow.add_conditional_edges(
        "reporter",
        route_after_reporter,
        {
            "respond": "send_response",
            "handoff_surveillance": "surveillance",
            "end": END
        }
    )
```

```
    # After sending response, end this turn
    workflow.add_edge("send_response", END)

    # Surveillance can: trigger analyst, send notification, or end
    workflow.add_conditional_edges(
        "surveillance",
        route_after_surveillance,
        {
            "analyst": "analyst",
            "notify": "send_notification",
            "end": END
        }
    )

    # Analyst always leads to notification
    workflow.add_edge("analyst", "send_notification")
    workflow.add_edge("send_notification", END)

    # Compile with checkpointing
    memory = SqliteSaver.from_conn_string(':memory:')  # Use Redis in production
    return workflow.compile(checkpointer=memory)


def route_after_reporter(state: ConversationState) -> str:
    """Determine next step after Reporter Agent processes message"""
    if state.get('error'):
        return 'end'
    if state.get('handoff_to') == 'surveillance':
        return 'handoff_surveillance'
    if state.get('pending_response'):
        return 'respond'
    return 'end'


def route_after_surveillance(state: ConversationState) -> str:
    """Determine next step after Surveillance Agent classifies report"""
    classification = state.get('classification')
    if not classification:
        return 'end'

    # Trigger analyst for situation summary if threshold exceeded
    if classification.urgency in ['critical', 'high']:
        return 'analyst'

    # Send notification for medium urgency
    if classification.urgency == 'medium':
        return 'notify'

    return 'end'
```

5.3 Reporter Agent Implementation

5.3.1 Core Logic

```
# agents/reporter.py

import anthropic
from datetime import datetime
from agents.state import ConversationState, Message, ExtractedData
```

```
from agents.prompts import REPORTER_SYSTEM_PROMPT
from config.llm_config import REPORTER_LLM_CONFIG
from utils.language import detect_language

client = anthropic.Anthropic()

async def reporter_node(state: ConversationState) -> ConversationState:
    """
    Reporter Agent: Handles conversation with community member.
    Detects health signals and collects MVS data.
    """

    # Get the latest user message
    user_message = state['messages'][-1]

    # Detect language if unknown
    if state['language'] == 'unknown':
        state['language'] = detect_language(user_message.content)

    # Build conversation history for context
    conversation_history = [
        {'role': msg.role, 'content': msg.content}
        for msg in state['messages']
    ]

    # Format system prompt with current state
    system_prompt = REPORTER_SYSTEM_PROMPT.format(
        mode=state['current_mode'],
        language=state['language'],
        extracted_data=state['extracted'].model_dump_json(),
        missing_fields=get_missing_fields(state['extracted'])
    )

    # Call Claude Haiku
    response = client.messages.create(
        model=REPORTER_LLM_CONFIG['model'],
        max_tokens=REPORTER_LLM_CONFIG['max_tokens'],
        temperature=REPORTER_LLM_CONFIG['temperature'],
        system=system_prompt,
        messages=conversation_history
    )

    assistant_message = response.content[0].text

    # Parse LLM response for state updates
    updated_state = parse_reporter_response(
        assistant_message,
        state,
        response.stop_reason
    )

    # Add assistant message to history
    updated_state['messages'].append(Message(
        role='assistant',
        content=assistant_message,
        timestamp=datetime.utcnow()
    ))

    updated_state['pending_response'] = assistant_message
    updated_state['updated_at'] = datetime.utcnow()
```

```
    updated_state['turn_count'] += 1

    return updated_state


def get_missing_fields(extracted: ExtractedData) -> list[str]:
    """Identify which MVS fields are still missing"""
    missing = []
    if not extracted.symptoms and not extracted.suspected_disease:
        missing.append('symptoms/disease')
    if not extracted.location_text:
        missing.append('location')
    if not extracted.onset_text and not extracted.onset_date:
        missing.append('timing')
    if extracted.cases_count is None:
        missing.append('case count')
    return missing


def parse_reporter_response(
    response: str,
    state: ConversationState,
    stop_reason: str
) -> ConversationState:
    """
    Parse the LLM response to extract:
    - Mode transitions
    - Extracted data updates
    - Handoff signals
    """
    # Implementation uses structured output or regex parsing
    # to extract data from conversational response
    # ...
    return state
```

5.4 Surveillance Agent Implementation

5.4.1 Classification Logic

```
# agents/surveillance.py

import anthropic
import json
from datetime import datetime, timedelta
from agents.state import ConversationState, Classification
from agents.prompts import SURVEILLANCE_SYSTEM_PROMPT
from config.llm_config import SURVEILLANCE_LLM_CONFIG
from db.queries import find_related_cases, get_case_count_for_area

client = anthropic.Anthropic()

# Disease thresholds (from Ministry of Health)
THRESHOLDS = {
    'cholera': {'alert': 1, 'outbreak': 3, 'window_days': 7},
    'dengue': {'alert': 5, 'outbreak': 20, 'window_days': 7},
    'malaria': {'alert': 'baseline', 'outbreak': 'significant_deviation'},
    'clustered_deaths': {'alert': 2, 'outbreak': 5, 'window_days': 7}
}

async def surveillance_node(state: ConversationState) -> ConversationState:
```

```
    """
    Surveillance Agent: Classifies reports and monitors thresholds.
    """

    # Build classification request
    report_data = {
        'symptoms': state['extracted'].symptoms,
        'suspected_disease': state['extracted'].suspected_disease,
        'location_text': state['extracted'].location_text,
        'onset_text': state['extracted'].onset_text,
        'cases_count': state['extracted'].cases_count,
        'deaths_count': state['extracted'].deaths_count,
        'reporter_relationship': state['extracted'].reporter_relationship,
    }

    # Get historical context
    related_cases = await find_related_cases(
        location=state['extracted'].location_normalized,
        symptoms=state['extracted'].symptoms,
        window_days=7
    )

    # Call Claude Sonnet for classification
    response = client.messages.create(
        model=SURVEILLANCE_LLM_CONFIG['model'],
        max_tokens=SURVEILLANCE_LLM_CONFIG['max_tokens'],
        temperature=SURVEILLANCE_LLM_CONFIG['temperature'],
        system=SURVEILLANCE_SYSTEM_PROMPT,
        messages=[{
            'role': 'user',
            'content': f'''
                Classify this health report:

                Report Data: {json.dumps(report_data)}

                Related Cases in Area (last 7 days): {json.dumps(related_cases)}

                Return your classification as JSON.
            '''
        }]
    )

    # Parse classification
    classification_json = extract_json(response.content[0].text)
    classification = Classification(**classification_json)

    # Check thresholds
    classification = await check_thresholds(classification, state, related_cases)

    # Link cases
    linked_ids = [case['id'] for case in related_cases]

    return {
        **state,
        'classification': classification,
        'linked_report_ids': linked_ids,
        'updated_at': datetime.utcnow()
    }
```

```
async def check_thresholds(
    classification: Classification,
    state: ConversationState,
    related_cases: list
) -> Classification:
    """Apply MoH thresholds to upgrade urgency if needed"""

    disease = classification.suspected_disease
    if not disease or disease not in THRESHOLDS:
        return classification

    threshold = THRESHOLDS[disease]
    total_cases = len(related_cases) + (state['extracted'].cases_count or 1)

    # Check outbreak threshold
    if isinstance(threshold['outbreak'], int) and total_cases >=
threshold['outbreak']:
        classification.urgency = 'critical'
        classification.alert_type = 'suspected_outbreak'
        classification.reasoning += f' THRESHOLD EXCEEDED: {total_cases} cases in
{threshold["window_days"]} days.'

    # Check alert threshold
    elif isinstance(threshold['alert'], int) and total_cases >= threshold['alert']:
        if classification.urgency not in ['critical']:
            classification.urgency = 'high'

    return classification
```

5.5 Case Linking Algorithm

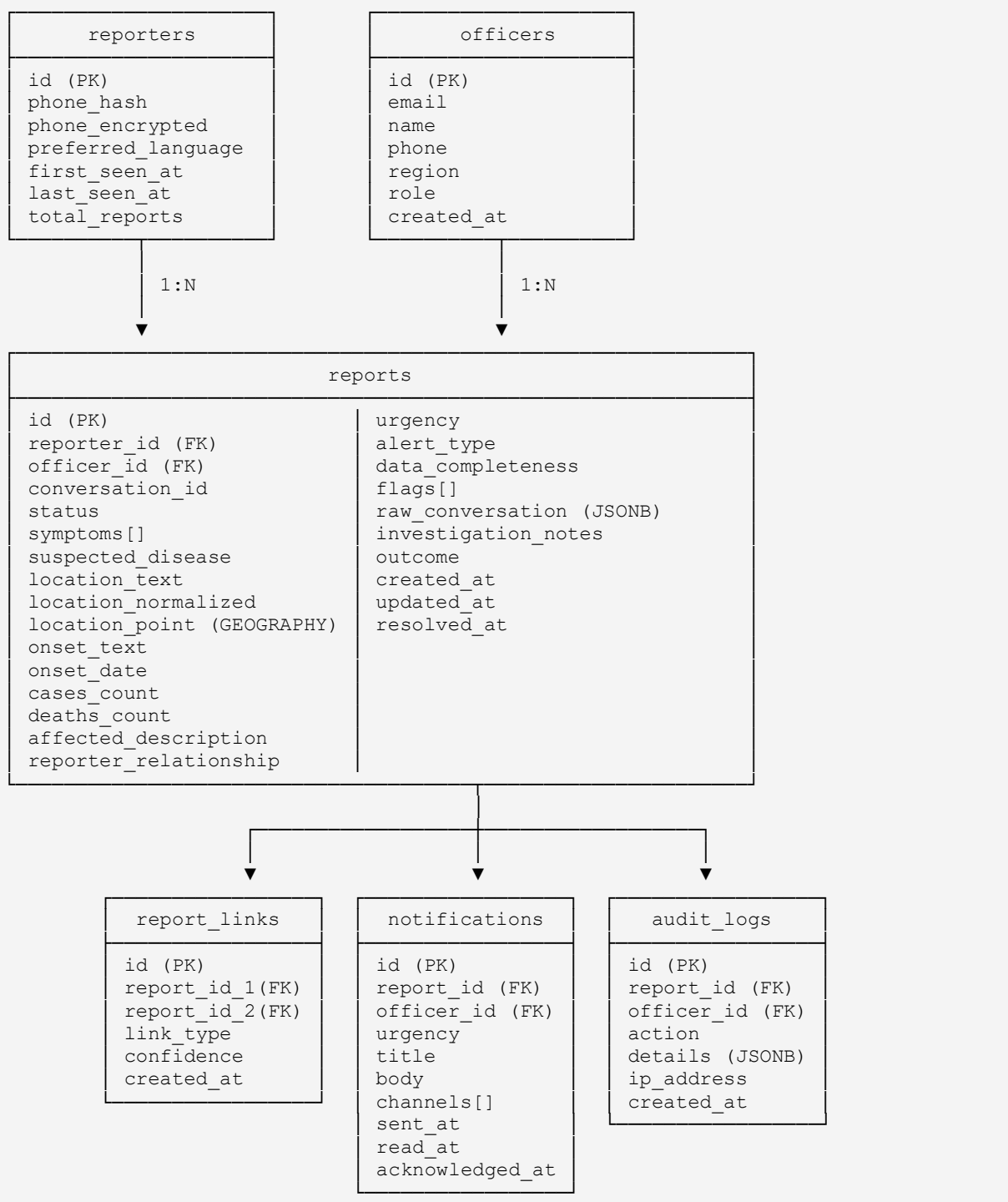
Cases are linked based on multiple proximity dimensions:

```
# db/queries.py

from datetime import datetime, timedelta
from sqlalchemy import select, and_, or_, func
from geoalchemy2 import Geography
from db.models import Report

async def find_related_cases(
    location: str,
    symptoms: list[str],
    window_days: int = 7,
    radius_km: float = 10.0,
    session = None
) -> list[dict]:
    """
    Find potentially related cases based on:
    1. Geographic proximity (within radius_km)
    2. Temporal proximity (within window_days)
    3. Symptom similarity (any overlap)
    """

    cutoff_date = datetime.utcnow() - timedelta(days=window_days)

    # Build query with PostGIS
    query = select(Report).where(
        and_(
            Report.created_at >= cutoff_date,
```

```
            Report.status.in_(['open', 'investigating']),
            or_(
                # Geographic match: within radius
                func.ST_DWithin(
                    Report.location_point,
                    func.ST_MakePoint(location.lng, location.lat),
                    radius_km * 1000  # Convert to meters
                ),
                # Location text match (fuzzy)
                Report.location_normalized.ilike(f'%{location}%')
            )
        )
    )

    results = await session.execute(query)
    cases = results.scalars().all()

    # Score by symptom overlap
    scored_cases = []
    for case in cases:
        symptom_overlap = len(set(symptoms) & set(case.symptoms or []))
        scored_cases.append({
            'id': str(case.id),
            'symptoms': case.symptoms,
            'suspected_disease': case.suspected_disease,
            'cases_count': case.cases_count,
            'created_at': case.created_at.isoformat(),
            'symptom_overlap_score': symptom_overlap / max(len(symptoms), 1)
        })

    # Sort by relevance
    scored_cases.sort(key=lambda x: x['symptom_overlap_score'], reverse=True)

    return scored_cases
```

6. Database Design

6.1 Entity Relationship Diagram



6.2 Complete Schema DDL

```
-- migrations/001_initial_schema.sql
```

```
-- Enable extensions
CREATE EXTENSION IF NOT EXISTS "uuid-ossp";
CREATE EXTENSION IF NOT EXISTS "postgis";
CREATE EXTENSION IF NOT EXISTS "pgcrypto";

-- Enum types
CREATE TYPE report_status AS ENUM ('open', 'investigating', 'resolved',
'false_alarm');
CREATE TYPE urgency_level AS ENUM ('critical', 'high', 'medium', 'low');
CREATE TYPE alert_type AS ENUM ('suspected_outbreak', 'cluster', 'single_case',
'rumor');
CREATE TYPE disease_type AS ENUM ('cholera', 'dengue', 'malaria', 'unknown');
CREATE TYPE reporter_rel AS ENUM ('witness', 'family', 'health_worker',
'community_leader', 'unknown');
CREATE TYPE link_type AS ENUM ('geographic', 'temporal', 'symptom', 'manual');

-- Reporters table (minimal PII)
CREATE TABLE reporters (
    id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
    phone_hash VARCHAR(64) UNIQUE NOT NULL,
    phone_encrypted BYTEA,  -- AES-256 encrypted
    preferred_language VARCHAR(2) DEFAULT 'ar',
    first_seen_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),
    last_seen_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),
    total_reports INTEGER DEFAULT 0,
    is_blocked BOOLEAN DEFAULT FALSE,
    block_reason TEXT
);

-- Officers table
CREATE TABLE officers (
    id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
    email VARCHAR(255) UNIQUE NOT NULL,
    password_hash VARCHAR(255) NOT NULL,
    name VARCHAR(255) NOT NULL,
    phone VARCHAR(20),
    region VARCHAR(100),
    role VARCHAR(50) DEFAULT 'officer',
    is_active BOOLEAN DEFAULT TRUE,
    last_login_at TIMESTAMP WITH TIME ZONE,
    created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),
    updated_at TIMESTAMP WITH TIME ZONE DEFAULT NOW()
);

-- Main reports table
CREATE TABLE reports (
    id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
    reporter_id UUID REFERENCES reporters(id) ON DELETE SET NULL,
    officer_id UUID REFERENCES officers(id) ON DELETE SET NULL,
    conversation_id VARCHAR(100) NOT NULL,
    platform VARCHAR(20) DEFAULT 'telegram',

    -- Status
    status report_status DEFAULT 'open',

    -- MVS Data
    symptoms TEXT[],
    suspected_disease disease_type,
    location_text TEXT,
    location_normalized VARCHAR(200),
```

```
    location_point GEOGRAPHY(POINT, 4326),
    onset_text TEXT,
    onset_date DATE,
    cases_count INTEGER,
    deaths_count INTEGER DEFAULT 0,
    affected_description TEXT,
    reporter_relationship reporter_rel DEFAULT 'unknown',

    -- Classification
    data_completeness FLOAT,
    urgency urgency_level,
    alert_type alert_type,
    classification_reasoning TEXT,

    -- Investigation
    investigation_notes TEXT,
    outcome TEXT,
    resolved_at TIMESTAMP WITH TIME ZONE,

    -- Metadata
    flags TEXT[],
    raw_conversation JSONB,
    created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),
    updated_at TIMESTAMP WITH TIME ZONE DEFAULT NOW()
);

-- Report links for case clustering
CREATE TABLE report_links (
    id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
    report_id_1 UUID REFERENCES reports(id) ON DELETE CASCADE,
    report_id_2 UUID REFERENCES reports(id) ON DELETE CASCADE,
    link_type link_type NOT NULL,
    confidence FLOAT DEFAULT 1.0,
    created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),
    UNIQUE(report_id_1, report_id_2)
);

-- Notifications
CREATE TABLE notifications (
    id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
    report_id UUID REFERENCES reports(id) ON DELETE CASCADE,
    officer_id UUID REFERENCES officers(id) ON DELETE CASCADE,
    urgency urgency_level NOT NULL,
    title TEXT NOT NULL,
    body TEXT NOT NULL,
    channels TEXT[] DEFAULT ARRAY['dashboard'],
    sent_at TIMESTAMP WITH TIME ZONE,
    read_at TIMESTAMP WITH TIME ZONE,
    acknowledged_at TIMESTAMP WITH TIME ZONE,
    created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW()
);

-- Audit logs
CREATE TABLE audit_logs (
    id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
    report_id UUID REFERENCES reports(id) ON DELETE SET NULL,
    officer_id UUID REFERENCES officers(id) ON DELETE SET NULL,
    action VARCHAR(100) NOT NULL,
    details JSONB,
    ip_address INET,
    user_agent TEXT,
    created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW()
```

```
);

-- Indexes for performance
CREATE INDEX idx_reports_location ON reports USING GIST(location_point);
CREATE INDEX idx_reports_created ON reports(created_at DESC);
CREATE INDEX idx_reports_status ON reports(status);
CREATE INDEX idx_reports_disease ON reports(suspected_disease);
CREATE INDEX idx_reports_urgency ON reports(urgency);
CREATE INDEX idx_reports_officer ON reports(officer_id);
CREATE INDEX idx_reports_conversation ON reports(conversation_id);
CREATE INDEX idx_reporters_phone ON reporters(phone_hash);
CREATE INDEX idx_notifications_officer ON notifications(officer_id, read_at);
CREATE INDEX idx_audit_report ON audit_logs(report_id);

-- Trigger for updated_at
CREATE OR REPLACE FUNCTION update_updated_at()
RETURNS TRIGGER AS $$
BEGIN
    NEW.updated_at = NOW();
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER reports_updated_at
    BEFORE UPDATE ON reports
    FOR EACH ROW EXECUTE FUNCTION update_updated_at();
```

6.3 Geospatial Query Examples

```
-- Find all reports within 10km of a point
SELECT * FROM reports
WHERE ST_DWithin(
    location_point,
    ST_MakePoint(32.5599, 15.5007)::geography,  -- Khartoum coordinates
    10000  -- 10km in meters
)
AND created_at > NOW() - INTERVAL '7 days'
ORDER BY created_at DESC;

-- Find clusters (reports within 5km of each other)
SELECT
    r1.id as report_1,
    r2.id as report_2,
    ST_Distance(r1.location_point, r2.location_point) as distance_meters
FROM reports r1
JOIN reports r2 ON r1.id < r2.id
WHERE ST_DWithin(r1.location_point, r2.location_point, 5000)
AND r1.suspected_disease = r2.suspected_disease
AND r1.created_at > NOW() - INTERVAL '7 days'
AND r2.created_at > NOW() - INTERVAL '7 days';

-- Aggregate cases by district (requires district polygons table)
SELECT
    d.name as district,
    COUNT(r.id) as case_count,
    SUM(r.cases_count) as total_affected
FROM districts d
LEFT JOIN reports r ON ST_Within(r.location_point::geometry, d.boundary)
WHERE r.created_at > NOW() - INTERVAL '30 days'
GROUP BY d.name
ORDER BY case_count DESC;
```

7. API Specification

7.1 API Overview

Endpoint	Method	Purpose	Auth
/webhook/telegram	POST	Receive Telegram messages	Signature
/webhook/whatsapp	POST	Receive WhatsApp messages	Signature
/api/v1/reports	GET	List reports (paginated)	JWT
/api/v1/reports/{id}	GET	Get report details	JWT
/api/v1/reports/{id}	PATCH	Update report (assign, resolve)	JWT
/api/v1/reports/{id}/notes	POST	Add investigation note	JWT
/api/v1/notifications	GET	List notifications	JWT
/api/v1/notifications/{id}/read	POST	Mark as read	JWT
/api/v1/analytics/query	POST	Natural language query	JWT
/api/v1/analytics/visualize	POST	Generate visualization	JWT
/ws	WebSocket	Real-time updates	JWT
/health	GET	Health check	None

7.2 FastAPI Implementation

7.2.1 Main Application

```
# api/main.py

from fastapi import FastAPI, Request
from fastapi.middleware.cors import CORSMiddleware
from contextlib import asynccontextmanager
import logging

from api.routes import webhook, reports, notifications, analytics, auth
from api.middleware import RateLimitMiddleware, LoggingMiddleware
from db.session import init_db, close_db
from config import settings

logger = logging.getLogger(__name__)

@asynccontextmanager
async def lifespan(app: FastAPI):
    """Startup and shutdown events"""
    # Startup
    await init_db()
    logger.info('Database initialized')
```

```
    yield
    # Shutdown
    await close_db()
    logger.info('Database connections closed')

app = FastAPI(
    title="CBI API",
    description="Community Based Intelligence Health Surveillance API",
    version="1.0.0",
    lifespan=lifespan
)

# CORS
app.add_middleware(
    CORSMiddleware,
    allow_origins=settings.CORS_ORIGINS,
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)

# Custom middleware
app.add_middleware(RateLimitMiddleware)
app.add_middleware(LoggingMiddleware)

# Routes
app.include_router(webhook.router, prefix='/webhook', tags=['Webhooks'])
app.include_router(auth.router, prefix='/api/v1/auth', tags=['Authentication'])
app.include_router(reports.router, prefix='/api/v1/reports', tags=['Reports'])
app.include_router(notifications.router, prefix='/api/v1/notifications',
tags=['Notifications'])
app.include_router(analytics.router, prefix='/api/v1/analytics',
tags=['Analytics'])

@app.get('/health')
async def health_check():
    return {'status': 'healthy', 'version': '1.0.0'}
```

7.2.2 Webhook Handler

```
# api/routes/webhook.py

from fastapi import APIRouter, Request, HTTPException, BackgroundTasks
import hmac
import hashlib
from config import settings
from services.message_queue import queue_incoming_message

router = APIRouter()

@router.post('/telegram')
async def telegram_webhook(
    request: Request,
    background_tasks: BackgroundTasks
):
    """Handle incoming Telegram messages"""
    data = await request.json()

    # Validate it's a message (not edit, callback, etc)
    if 'message' not in data:
        return {'ok': True}
```

```
    message = data['message']

    # Extract relevant fields
    incoming = {
        'platform': 'telegram',
        'message_id': str(message['message_id']),
        'chat_id': str(message['chat']['id']),
        'from_id': str(message['from']['id']),
        'text': message.get('text', ''),
        'timestamp': message['date']
    }

    # Queue for async processing
    background_tasks.add_task(queue_incoming_message, incoming)

    return {'ok': True}


@router.post('/whatsapp')
async def whatsapp_webhook(
    request: Request,
    background_tasks: BackgroundTasks
):
    """Handle incoming WhatsApp messages"""

    # Verify signature
    signature = request.headers.get('X-Hub-Signature-256', '')
    body = await request.body()

    expected = 'sha256=' + hmac.new(
        settings.WHATSAPP_APP_SECRET.encode(),
        body,
        hashlib.sha256
    ).hexdigest()

    if not hmac.compare_digest(signature, expected):
        raise HTTPException(status_code=403, detail='Invalid signature')

    data = await request.json()

    # Process entries
    for entry in data.get('entry', []):
        for change in entry.get('changes', []):
            value = change.get('value', {})
            for message in value.get('messages', []):
                incoming = {
                    'platform': 'whatsapp',
                    'message_id': message['id'],
                    'from_id': message['from'],
                    'text': message.get('text', {}).get('body', ''),
                    'timestamp': int(message['timestamp'])
                }
                background_tasks.add_task(queue_incoming_message, incoming)

    return {'status': 'ok'}


@router.get('/whatsapp')
async def whatsapp_verify(
```

```
    request: Request
):
    """WhatsApp webhook verification"""
    params = request.query_params

    if params.get('hub.mode') == 'subscribe':
        if params.get('hub.verify_token') == settings.WHATSAPP_VERIFY_TOKEN:
            return int(params.get('hub.challenge', 0))

    raise HTTPException(status_code=403)
```

7.2.3 Reports API

```
# api/routes/reports.py

from fastapi import APIRouter, Depends, Query, HTTPException
from typing import Optional, List
from uuid import UUID
from datetime import datetime

from api.deps import get_current_officer, get_db
from api.schemas import ReportResponse, ReportUpdate, ReportListResponse
from db.queries import get_reports, get_report_by_id, update_report

router = APIRouter()

@router.get('', response_model=ReportListResponse)
async def list_reports(
    status: Optional[str] = None,
    urgency: Optional[str] = None,
    disease: Optional[str] = None,
    region: Optional[str] = None,
    from_date: Optional[datetime] = None,
    to_date: Optional[datetime] = None,
    page: int = Query(1, ge=1),
    per_page: int = Query(20, ge=1, le=100),
    db = Depends(get_db),
    officer = Depends(get_current_officer)
):
    """List reports with filtering and pagination"""
    reports, total = await get_reports(
        db=db,
        status=status,
        urgency=urgency,
        disease=disease,
        region=region,
        from_date=from_date,
        to_date=to_date,
        offset=(page - 1) * per_page,
        limit=per_page
    )

    return {
        'items': reports,
        'total': total,
        'page': page,
        'per_page': per_page,
        'pages': (total + per_page - 1) // per_page
    }
```

```
@router.get('/{report_id}', response_model=ReportResponse)
async def get_report(
    report_id: UUID,
    db = Depends(get_db),
    officer = Depends(get_current_officer)
):
    """Get single report with full details"""
    report = await get_report_by_id(db, report_id)
    if not report:
        raise HTTPException(status_code=404, detail='Report not found')
    return report


@router.patch('/{report_id}', response_model=ReportResponse)
async def update_report_endpoint(
    report_id: UUID,
    update: ReportUpdate,
    db = Depends(get_db),
    officer = Depends(get_current_officer)
):
    """Update report (assign, change status, add notes)"""
    report = await update_report(
        db=db,
        report_id=report_id,
        officer_id=officer.id,
        **update.model_dump(exclude_unset=True)
    )
    if not report:
        raise HTTPException(status_code=404, detail='Report not found')
    return report
```

8. Messaging Gateway Abstraction

8.1 Platform Abstraction Layer

To support both Telegram (MVP) and WhatsApp (production) with minimal code changes, we implement a messaging gateway abstraction.

```
# services/messaging/base.py

from abc import ABC, abstractmethod
from dataclasses import dataclass
from typing import Optional

@dataclass
class IncomingMessage:
    """Normalized incoming message"""
    platform: str
    message_id: str
    chat_id: str
    from_id: str
    text: str
    timestamp: int
    reply_to_id: Optional[str] = None

@dataclass
class OutgoingMessage:
    """Message to send"""
    chat_id: str
    text: str
    reply_to_id: Optional[str] = None

class MessagingGateway(ABC):
    """Abstract base for messaging platforms"""

    @abstractmethod
    async def send_message(self, message: OutgoingMessage) -> str:
        """Send message, return message ID"""
        pass

    @abstractmethod
    async def send_template(
        self,
        chat_id: str,
        template_name: str,
        params: dict
    ) -> str:
        """Send template message (required for WhatsApp proactive)"""
        pass

    @abstractmethod
    def parse_webhook(self, data: dict) -> list[IncomingMessage]:
        """Parse webhook payload into normalized messages"""
        pass
```

8.1.1 Telegram Implementation

```
# services/messaging/telegram.py

import httpx
```

```
from config import settings
from services.messaging.base import MessagingGateway, IncomingMessage,
OutgoingMessage

class TelegramGateway(MessagingGateway):
    def __init__(self):
        self.token = settings.TELEGRAM_BOT_TOKEN
        self.base_url = f'https://api.telegram.org/bot{self.token}'
        self.client = httpx.AsyncClient(timeout=30.0)

    async def send_message(self, message: OutgoingMessage) -> str:
        response = await self.client.post(
            f'{self.base_url}/sendMessage',
            json={
                'chat_id': message.chat_id,
                'text': message.text,
                'parse_mode': 'HTML',
                'reply_to_message_id': message.reply_to_id
            }
        )
        data = response.json()
        return str(data['result']['message_id'])

    async def send_template(
        self,
        chat_id: str,
        template_name: str,
        params: dict
    ) -> str:
        # Telegram doesn't have templates - send formatted message
        templates = {
            'report_confirmation': 'تم استلام بلاغك. رقم المرجع: {reference_id}',
            'status_update': 'تحديث: {status}',
        }
        text = templates.get(template_name, '').format(**params)
        return await self.send_message(OutgoingMessage(chat_id=chat_id, text=text))

    def parse_webhook(self, data: dict) -> list[IncomingMessage]:
        messages = []
        if 'message' in data and 'text' in data['message']:
            msg = data['message']
            messages.append(IncomingMessage(
                platform='telegram',
                message_id=str(msg['message_id']),
                chat_id=str(msg['chat']['id']),
                from_id=str(msg['from']['id']),
                text=msg['text'],
                timestamp=msg['date'],
                reply_to_id=str(msg.get('reply_to_message', {}).get('message_id'))
            ))
        return messages
```

8.1.2 WhatsApp Implementation

```
# services/messaging/whatsapp.py

import httpx
from config import settings
from services.messaging.base import MessagingGateway, IncomingMessage,
OutgoingMessage
```

```
class WhatsAppGateway(MessagingGateway):
    def __init__(self):
        self.phone_number_id = settings.WHATSAPP_PHONE_NUMBER_ID
        self.access_token = settings.WHATSAPP_ACCESS_TOKEN
        self.base_url = f'https://graph.facebook.com/v17.0/{self.phone_number_id}'
        self.client = httpx.AsyncClient(
            timeout=30.0,
            headers={'Authorization': f'Bearer {self.access_token}'}
        )

    async def send_message(self, message: OutgoingMessage) -> str:
        response = await self.client.post(
            f'{self.base_url}/messages',
            json={
                'messaging_product': 'whatsapp',
                'to': message.chat_id,
                'type': 'text',
                'text': {'body': message.text}
            }
        )
        data = response.json()
        return data['messages'][0]['id']

    async def send_template(
        self,
        chat_id: str,
        template_name: str,
        params: dict
    ) -> str:
        # WhatsApp requires pre-approved templates for proactive messages
        response = await self.client.post(
            f'{self.base_url}/messages',
            json={
                'messaging_product': 'whatsapp',
                'to': chat_id,
                'type': 'template',
                'template': {
                    'name': template_name,
                    'language': {'code': params.get('language', 'ar')},
                    'components': [{
                        'type': 'body',
                        'parameters': [
                            {'type': 'text', 'text': v}
                            for v in params.get('body_params', [])
                        ]
                    }]
                }
            }
        )
        data = response.json()
        return data['messages'][0]['id']

    def parse_webhook(self, data: dict) -> list[IncomingMessage]:
        messages = []
        for entry in data.get('entry', []):
            for change in entry.get('changes', []):
                value = change.get('value', {})
                for msg in value.get('messages', []):
                    if msg.get('type') == 'text':
                        messages.append(IncomingMessage(
                            platform='whatsapp',
                            message_id=msg['id'],
```

```
                            chat_id=msg['from'],
                            from_id=msg['from'],
                            text=msg['text']['body'],
                            timestamp=int(msg['timestamp']),
                        ))
        return messages
```

8.1.3 Gateway Factory

```
# services/messaging/factory.py

from services.messaging.base import MessagingGateway
from services.messaging.telegram import TelegramGateway
from services.messaging.whatsapp import WhatsAppGateway

_gateways = {}

def get_gateway(platform: str) -> MessagingGateway:
    """Get messaging gateway for platform"""
    if platform not in _gateways:
        if platform == 'telegram':
            _gateways[platform] = TelegramGateway()
        elif platform == 'whatsapp':
            _gateways[platform] = WhatsAppGateway()
        else:
            raise ValueError(f'Unknown platform: {platform}')
    return _gateways[platform]
```

9. State Management

9.1 Redis Schema

```
# Redis Key Patterns

# Conversation state (hot data)
# Key: conversation:{conversation_id}
# Value: JSON serialized ConversationState
# TTL: 24 hours (extended on each interaction)

# Session mapping (phone -> active conversation)
# Key: session:{platform}:{phone_hash}
# Value: conversation_id
# TTL: 1 hour

# Rate limiting
# Key: ratelimit:{platform}:{phone_hash}:{window}
# Value: count
# TTL: window duration

# Location cache (normalized locations)
# Key: location:{hash(location_text)}
# Value: JSON {normalized, coords}
# TTL: 7 days

# Active alerts (for dashboard real-time)
# Key: alerts:active
# Type: Sorted Set (score = timestamp)
# Value: notification_id
```

9.2 State Service Implementation

```
# services/state.py

import json
import hashlib
from datetime import datetime, timedelta
from typing import Optional
import redis.asyncio as redis

from agents.state import ConversationState
from config import settings

class StateService:
    def __init__(self):
        self.redis = redis.from_url(settings.REDIS_URL)
        self.state_ttl = 86400  # 24 hours
        self.session_ttl = 3600  # 1 hour

    def _phone_hash(self, phone: str) -> str:
        return hashlib.sha256(
            (phone + settings.PHONE_HASH_SALT).encode()
        ).hexdigest()[:16]

    async def get_or_create_conversation(
        self,
        platform: str,
        phone: str
```

```
    ) -> tuple[ConversationState, bool]:
        """Get existing conversation or create new one"""
        phone_hash = self._phone_hash(phone)
        session_key = f'session:{platform}:{phone_hash}'

        # Check for existing session
        conversation_id = await self.redis.get(session_key)

        if conversation_id:
            state = await self.get_state(conversation_id.decode())
            if state:
                return state, False

        # Create new conversation
        conversation_id =
f'{platform}_{phone_hash}_{int(datetime.utcnow().timestamp())}'
        state = ConversationState(
            conversation_id=conversation_id,
            reporter_phone=phone,
            platform=platform,
            messages=[],
            current_mode='listening',
            language='unknown',
            extracted=ExtractedData(),
            classification=None,
            linked_report_ids=[],
            pending_response=None,
            handoff_to=None,
            error=None,
            created_at=datetime.utcnow(),
            updated_at=datetime.utcnow(),
            turn_count=0
        )

        # Save state and session
        await self.save_state(state)
        await self.redis.setex(
            session_key,
            self.session_ttl,
            conversation_id
        )

        return state, True

    async def get_state(self, conversation_id: str) -> Optional[ConversationState]:
        key = f'conversation:{conversation_id}'
        data = await self.redis.get(key)
        if data:
            return ConversationState(**json.loads(data))
        return None

    async def save_state(self, state: ConversationState):
        key = f'conversation:{state["conversation_id"]}'
        await self.redis.setex(
            key,
            self.state_ttl,
            json.dumps(state, default=str)
        )

    async def delete_state(self, conversation_id: str):
        key = f'conversation:{conversation_id}'
```

```
        await self.redis.delete(key)
```

10. Security Architecture

10.1 Security Requirements

Category	Requirement	Implementation
Data Encryption	Encrypt PII at rest	AES-256 for phone numbers, PostgreSQL disk encryption
Data Encryption	Encrypt data in transit	TLS 1.3 for all connections
Authentication	Webhook verification	HMAC signature validation
Authentication	Dashboard auth	JWT with refresh tokens, bcrypt passwords
Authorization	Role-based access	Officer roles: admin, supervisor, officer
Privacy	Minimal PII collection	Only phone number for follow-up
Privacy	Phone number hashing	SHA-256 for lookups, encrypted for retrieval
Audit	Action logging	All data access logged with officer ID
Rate Limiting	Prevent abuse	10 msg/min per phone, 100 requests/min per IP

10.2 Phone Number Encryption

```
# services/crypto.py

import hashlib
import base64
from cryptography.fernet import Fernet
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC
from config import settings

class PhoneCrypto:
    """Handle phone number encryption and hashing"""

    def __init__(self):
        # Derive encryption key from master key
        kdf = PBKDF2HMAC(
            algorithm=hashes.SHA256(),
            length=32,
            salt=settings.ENCRYPTION_SALT.encode(),
            iterations=100000
        )
        key = base64.urlsafe_b64encode(
            kdf.derive(settings.ENCRYPTION_KEY.encode())
        )
        self.fernet = Fernet(key)

    def hash_phone(self, phone: str) -> str:
        """Create one-way hash for lookups"""
        return hashlib.sha256(
```

```
            (phone + settings.PHONE_HASH_SALT).encode()
        ).hexdigest()

    def encrypt_phone(self, phone: str) -> bytes:
        """Encrypt phone for storage"""
        return self.fernet.encrypt(phone.encode())

    def decrypt_phone(self, encrypted: bytes) -> str:
        """Decrypt phone when needed for follow-up"""
        return self.fernet.decrypt(encrypted).decode()
```

10.3 JWT Authentication

```
# services/auth.py

from datetime import datetime, timedelta
from typing import Optional
import jwt
from passlib.context import CryptContext
from fastapi import HTTPException, Depends
from fastapi.security import HTTPBearer
from config import settings

pwd_context = CryptContext(schemes=['bcrypt'], deprecated='auto')
security = HTTPBearer()

def create_access_token(officer_id: str, role: str) -> str:
    expires = datetime.utcnow() + timedelta(hours=settings.JWT_EXPIRE_HOURS)
    payload = {
        'sub': officer_id,
        'role': role,
        'exp': expires,
        'iat': datetime.utcnow()
    }
    return jwt.encode(payload, settings.JWT_SECRET, algorithm='HS256')

def verify_token(token: str) -> dict:
    try:
        payload = jwt.decode(
            token,
            settings.JWT_SECRET,
            algorithms=['HS256']
        )
        return payload
    except jwt.ExpiredSignatureError:
        raise HTTPException(status_code=401, detail='Token expired')
    except jwt.InvalidTokenError:
        raise HTTPException(status_code=401, detail='Invalid token')

async def get_current_officer(credentials = Depends(security)):
    payload = verify_token(credentials.credentials)
    # Load officer from database
    officer = await get_officer_by_id(payload['sub'])
    if not officer or not officer.is_active:
        raise HTTPException(status_code=401, detail='Officer not found')
    return officer
```

11. Deployment Architecture

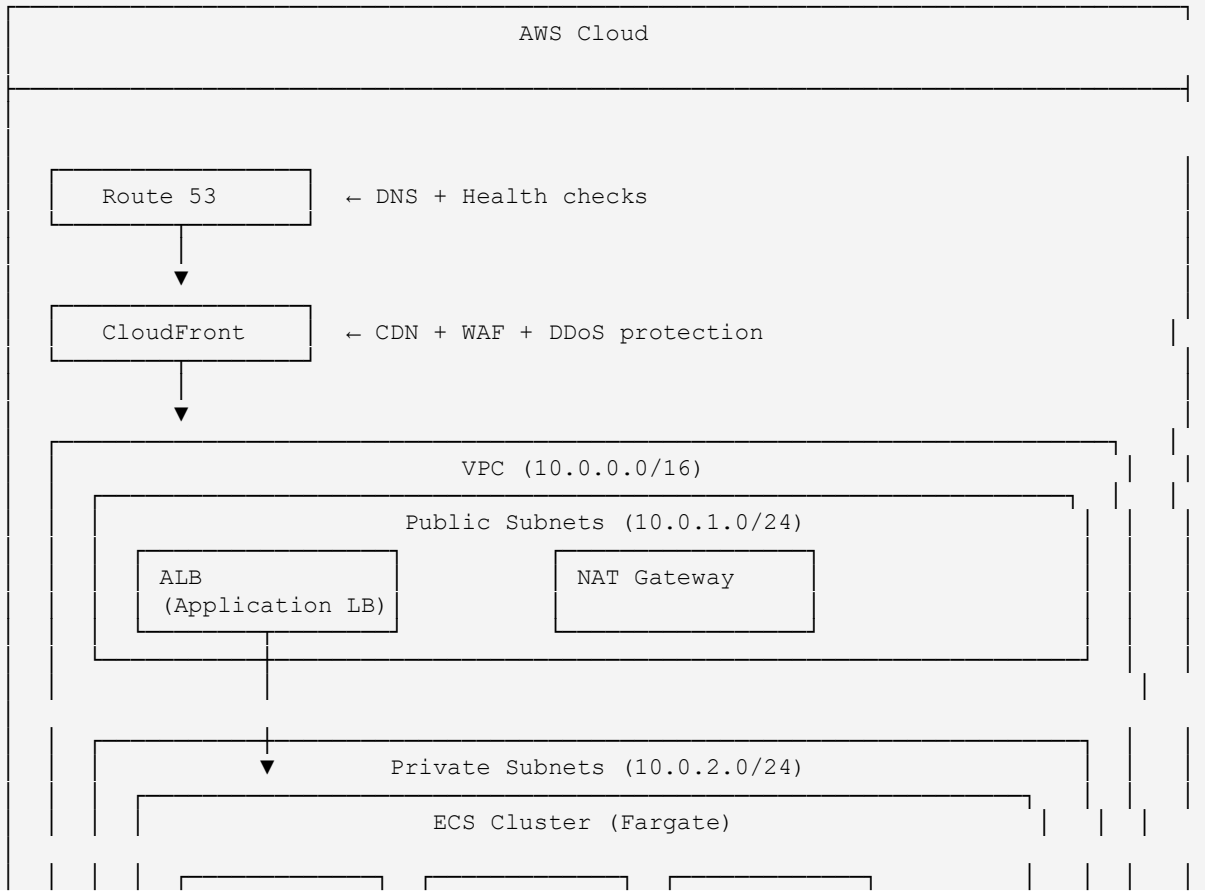
11.1 Local Development (Docker Compose)

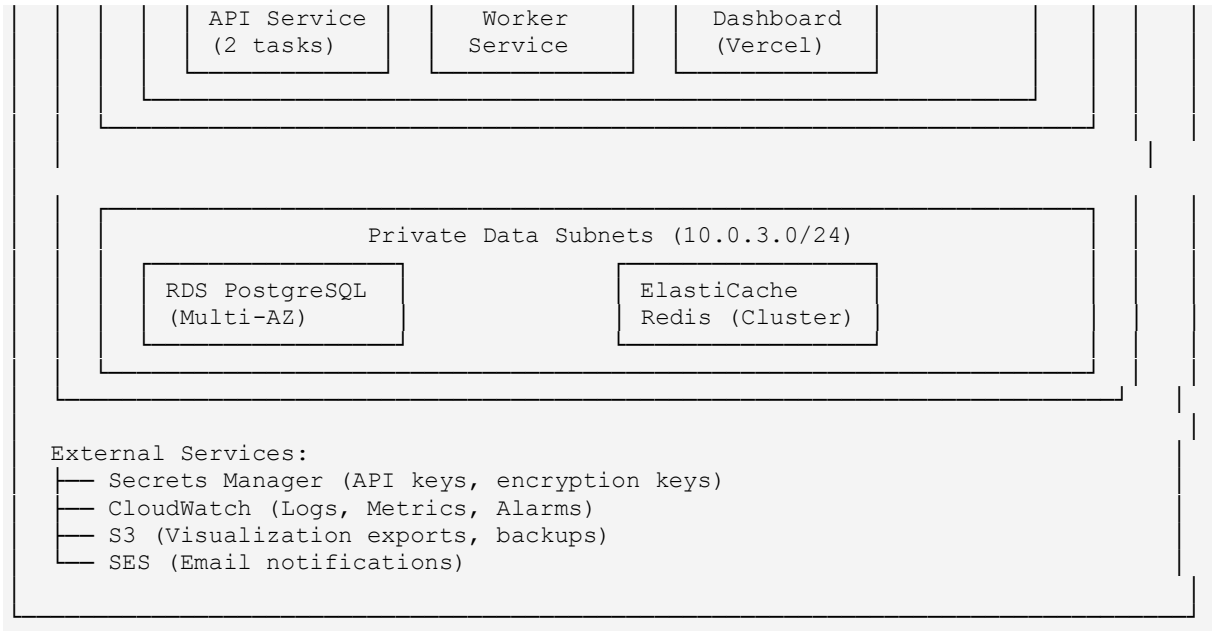
```
# docker-compose.yml

version: '3.8'

services:
  api:
    build:
      context: .
      dockerfile: Dockerfile
    ports:
      - '8000:8000'
    environment:
      - DATABASE_URL=postgresql://cbi:cbi_password@db:5432/cbi
      - REDIS_URL=redis://redis:6379
      - ANTHROPIC_API_KEY=${ANTHROPIC_API_KEY}
      - TELEGRAM_BOT_TOKEN=${TELEGRAM_BOT_TOKEN}
      - JWT_SECRET=${JWT_SECRET}
      - ENCRYPTION_KEY=${ENCRYPTION_KEY}
    depends_on:
      - db
      - redis
    volumes:
      - ./src:/app/src  # Hot reload
    command: uvicorn api.main:app --host 0.0.0.0 --port 8000 --reload

  worker:
    build:
      context: .
      dockerfile: Dockerfile
    environment:
      - DATABASE_URL=postgresql://cbi:cbi_password@db:5432/cbi
      - REDIS_URL=redis://redis:6379
      - ANTHROPIC_API_KEY=${ANTHROPIC_API_KEY}
    depends_on:
      - db
      - redis
    command: python -m workers.main

  dashboard:
    build:
      context: ./dashboard
      dockerfile: Dockerfile
    ports:
      - '3000:3000'
    environment:
      - NEXT_PUBLIC_API_URL=http://localhost:8000
    volumes:
      - ./dashboard/src:/app/src  # Hot reload

  db:
    image: postgis/postgis:15-3.3
    environment:
      - POSTGRES_DB=cbi
      - POSTGRES_USER=cbi
      - POSTGRES_PASSWORD=cbi_password
    volumes:
```

```
      - postgres_data:/var/lib/postgresql/data
      - ./migrations:/docker-entrypoint-initdb.d
    ports:
      - '5432:5432'

  redis:
    image: redis:7-alpine
    volumes:
      - redis_data:/data
    ports:
      - '6379:6379'

  ngrok:
    image: ngrok/ngrok:latest
    command: http api:8000 --domain=${NGROK_DOMAIN}
    environment:
      - NGROK_AUTHTOKEN=${NGROK_AUTHTOKEN}
    depends_on:
      - api

volumes:
  postgres_data:
  redis_data:
```

11.2 AWS Production Architecture

```
# AWS Architecture Overview
```





11.3 Terraform Configuration

```
# terraform/main.tf

terraform {
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 5.0"
    }
  }
  backend "s3" {
    bucket = "cbi-terraform-state"
    key    = "prod/terraform.tfstate"
    region = "eu-west-1"
  }
}

provider "aws" {
  region = var.aws_region
}

# VPC
module "vpc" {
  source = "terraform-aws-modules/vpc/aws"

  name = "cbi-vpc"
  cidr = "10.0.0.0/16"

  azs             = ["eu-west-1a", "eu-west-1b"]
  private_subnets = ["10.0.1.0/24", "10.0.2.0/24"]
  public_subnets  = ["10.0.101.0/24", "10.0.102.0/24"]

  enable_nat_gateway = true
  single_nat_gateway = true  # Cost savings for MVP
}
```

```
# RDS PostgreSQL with PostGIS
module "rds" {
  source = "terraform-aws-modules/rds/aws"

  identifier = "cbi-postgres"
  engine     = "postgres"
  engine_version = "15.4"
  family     = "postgres15"

  instance_class    = "db.t3.medium"
  allocated_storage = 20

  db_name  = "cbi"
  username = "cbi_admin"
  port     = 5432

  vpc_security_group_ids = [aws_security_group.rds.id]
  subnet_ids             = module.vpc.private_subnets

  backup_retention_period = 7
  deletion_protection     = true
}

# ElastiCache Redis
resource "aws_elasticache_cluster" "redis" {
  cluster_id           = "cbi-redis"
  engine               = "redis"
  node_type            = "cache.t3.micro"
  num_cache_nodes      = 1
  port                 = 6379
  security_group_ids   = [aws_security_group.redis.id]
  subnet_group_name    = aws_elasticache_subnet_group.main.name
}
```

12. Monitoring and Observability

12.1 Metrics Collection

Metric	Type	Labels	Alert Threshold
cbi_messages_received_total	Counter	platform, language	N/A
cbi_messages_processed_total	Counter	platform, status	N/A
cbi_llm_request_duration_seconds	Histogram	model, agent	p99 > 10s
cbi_llm_tokens_total	Counter	model, agent, direction	N/A
cbi_llm_errors_total	Counter	model, error_type	> 10/min
cbi_reports_created_total	Counter	urgency, disease	N/A
cbi_classification_duration_seconds	Histogram	disease	p95 > 30s
cbi_queue_depth	Gauge	queue_name	> 1000
cbi_active_conversations	Gauge	platform	N/A

12.2 Prometheus Configuration

```
# prometheus/prometheus.yml

global:
  scrape_interval: 15s
  evaluation_interval: 15s

alerting:
  alertmanagers:
    - static_configs:
        - targets:
          - alertmanager:9093

rule_files:
  - '/etc/prometheus/rules/*.yml'

scrape_configs:
  - job_name: 'cbi-api'
    static_configs:
      - targets: ['api:8000']
    metrics_path: '/metrics'

  - job_name: 'cbi-worker'
    static_configs:
      - targets: ['worker:8001']

  - job_name: 'redis'
    static_configs:
      - targets: ['redis-exporter:9121']

  - job_name: 'postgres'
    static_configs:
```

```
      - targets: ['postgres-exporter:9187']
```

12.3 Alert Rules

```
# prometheus/rules/cbi_alerts.yml

groups:
  - name: cbi_alerts
    rules:
      - alert: HighLLMLatency
        expr: histogram_quantile(0.99,
rate(cbi_llm_request_duration_seconds_bucket[5m])) > 10
        for: 5m
        labels:
          severity: warning
        annotations:
          summary: 'LLM latency is high'
          description: 'P99 latency is {{ $value }}s'

      - alert: HighLLMErrorRate
        expr: rate(cbi_llm_errors_total[5m]) > 0.1
        for: 5m
        labels:
          severity: critical
        annotations:
          summary: 'High LLM error rate'
          description: 'Error rate is {{ $value }}/s'

      - alert: QueueBacklog
        expr: cbi_queue_depth > 1000
        for: 10m
        labels:
          severity: warning
        annotations:
          summary: 'Message queue backlog'
          description: 'Queue depth is {{ $value }}'

      - alert: CriticalReportUnacknowledged
        expr: cbi_unacknowledged_critical_reports > 0
        for: 30m
        labels:
          severity: critical
        annotations:
          summary: 'Critical report not acknowledged'
          description: '{{ $value }} critical reports pending > 30min'
```

12.4 Structured Logging

```
# config/logging.py

import logging
import json
from datetime import datetime

class JSONFormatter(logging.Formatter):
    def format(self, record):
        log_obj = {
            'timestamp': datetime.utcnow().isoformat(),
            'level': record.levelname,
            'logger': record.name,
```

```
            'message': record.getMessage(),
            'module': record.module,
            'function': record.funcName,
            'line': record.lineno
        }

        # Add extra fields
        if hasattr(record, 'conversation_id'):
            log_obj['conversation_id'] = record.conversation_id
        if hasattr(record, 'reporter_phone_hash'):
            log_obj['reporter_phone_hash'] = record.reporter_phone_hash
        if hasattr(record, 'agent'):
            log_obj['agent'] = record.agent

        # Sanitize - never log PII
        if hasattr(record, 'message_content'):
            log_obj['message_length'] = len(record.message_content)
            # Don't log actual content

        return json.dumps(log_obj)

def setup_logging():
    handler = logging.StreamHandler()
    handler.setFormatter(JSONFormatter())

    logging.root.handlers = [handler]
    logging.root.setLevel(logging.INFO)
```

13. Testing Strategy

13.1 Test Pyramid

Layer	Test Type	Count Target	Tools
Unit	Pure functions, utilities	~200	pytest
Integration	DB queries, API endpoints	~100	pytest, testcontainers
Agent	Agent nodes with mocked LLM	~50	pytest, unittest.mock
E2E	Full message flow simulation	~20	pytest, httpx

13.2 LLM Testing Approach

```
# tests/agents/test_reporter_intent.py

import pytest
from unittest.mock import AsyncMock, patch
from agents.reporter import reporter_node
from agents.state import ConversationState, Message

# Golden dataset for intent detection
INTENT_TEST_CASES = [
    # Should trigger investigation
    {
        'input': 'جارتي عندها إسهال شديد من أمس',  # My neighbor has severe
diarrhea since yesterday
        'expected_mode': 'investigating',
        'expected_symptoms': ['diarrhea'],
        'language': 'ar'
    },
    {
        'input': 'Three children in my village are vomiting',
        'expected_mode': 'investigating',
        'expected_symptoms': ['vomiting'],
        'language': 'en'
    },
    # Should NOT trigger investigation
    {
        'input': 'What are the symptoms of cholera?',
        'expected_mode': 'listening',
        'expected_symptoms': [],
        'language': 'en'
    },
    {
        'input': 'I had malaria last year',
        'expected_mode': 'listening',
        'expected_symptoms': [],
        'language': 'en'
    },
]

@pytest.mark.parametrize('test_case', INTENT_TEST_CASES)
async def test_intent_detection(test_case):
    """Test that intent detection correctly identifies reportable events"""
```

```
    initial_state = create_initial_state()
    initial_state['messages'].append(Message(
        role='user',
        content=test_case['input'],
        timestamp=datetime.utcnow()
    ))

    # Run agent (with real LLM or mocked response)
    result = await reporter_node(initial_state)

    assert result['current_mode'] == test_case['expected_mode']
    if test_case['expected_symptoms']:
        assert any(
            symptom in result['extracted'].symptoms
            for symptom in test_case['expected_symptoms']
        )
```

13.3 Integration Test Example

```
# tests/integration/test_full_flow.py

import pytest
from httpx import AsyncClient
from api.main import app

@pytest.mark.asyncio
async def test_telegram_report_flow():
    """Test complete flow from Telegram message to report creation"""

    async with AsyncClient(app=app, base_url='http://test') as client:
        # Simulate Telegram webhook - message 1
        response = await client.post('/webhook/telegram', json={
            'update_id': 123,
            'message': {
                'message_id': 1,
                'chat': {'id': 12345},
                'from': {'id': 12345},
                'date': 1704067200,
                'text': 'Hello'
            }
        })
        assert response.status_code == 200

        # Wait for processing
        await asyncio.sleep(2)

        # Message 2 - health report
        response = await client.post('/webhook/telegram', json={
            'update_id': 124,
            'message': {
                'message_id': 2,
                'chat': {'id': 12345},
                'from': {'id': 12345},
                'date': 1704067201,
                'text': 'My neighbor has severe diarrhea since yesterday. Three
people sick in Kassala.'
            }
        })
        assert response.status_code == 200
```

```
        # Wait for processing
        await asyncio.sleep(5)

        # Verify report was created
        # (would need to authenticate as officer)
        reports = await get_recent_reports()
        assert len(reports) >= 1
        assert reports[0].suspected_disease == 'cholera'
        assert reports[0].cases_count == 3
```

14. Cost Analysis

14.1 Monthly Cost Breakdown (1000 Users)

Component	Specification	Low Estimate	High Estimate
LLM API (Claude)	Haiku + Sonnet, ~100 reports/day	$500	$1,200
AWS ECS (Fargate)	2 vCPU, 4GB RAM × 2 services	$80	$120
AWS RDS (PostgreSQL)	db.t3.medium, 20GB, Multi-AZ	$60	$100
AWS ElastiCache (Redis)	cache.t3.micro	$15	$25
AWS ALB	Application Load Balancer	$20	$30
AWS Data Transfer	~50GB/month	$5	$10
AWS CloudWatch	Logs, metrics, alarms	$20	$40
AWS Secrets Manager	~10 secrets	$5	$10
Domain + SSL	Route 53 + ACM	$10	$15
Telegram API	Free tier	$0	$0
TOTAL		$715	$1,550

14.2 Scaling Cost Projections

User Scale	Reports/Day	LLM Cost	Infra Cost	Total/Month
1,000	100	$500-1,200	$215-350	$715-1,550
10,000	1,000	$3,000-7,000	$500-800	$3,500-7,800
100,000	10,000	$25,000-60,000	$2,000-4,000	$27,000-64,000

14.3 Cost Optimization Strategies

- Use Haiku for all initial messages, Sonnet only for classification
- Cache common responses (greetings, confirmations) - saves ~20% LLM costs
- Implement conversation length limits (max 20 turns)
- Batch Analyst queries during off-peak hours
- Use spot instances for non-critical workers (50% savings)
- Reserved instances for predictable workloads (30% savings)

15. Implementation Roadmap

15.1 Phase 1: Foundation (Weeks 1-2)

- Project setup: Repository, CI/CD, Docker configuration
- Database schema implementation and migrations
- Basic FastAPI application with health checks
- Redis connection and state management
- Telegram bot setup and webhook handling

15.2 Phase 2: Reporter Agent (Weeks 3-4)

- LangGraph state machine setup
- Reporter Agent implementation with Claude Haiku
- Intent detection and mode transitions
- MVS extraction logic
- Language detection (Arabic/English)
- Conversation state persistence

15.3 Phase 3: Surveillance Agent (Weeks 5-6)

- Classification logic with Claude Sonnet
- Threshold monitoring implementation
- Case linking algorithm (geographic + temporal + symptom)
- Notification generation
- Alert queue and delivery

15.4 Phase 4: Dashboard (Weeks 7-8)

- Next.js project setup
- Authentication (JWT)
- Report list and detail views
- Real-time notifications (WebSocket)
- Basic filtering and search

15.5 Phase 5: Analyst Agent (Weeks 9-10)

- Natural language query interface
- Visualization code generation
- Situation summary generation
- Dashboard integration

15.6 Phase 6: Testing & Hardening (Weeks 11-12)

- Comprehensive test suite
- Security audit and fixes
- Performance optimization
- Documentation completion
- Staging deployment

15.7 Phase 7: Production Deployment (Week 13)

- AWS infrastructure provisioning (Terraform)
- Production deployment
- Monitoring setup (Prometheus/Grafana)
- Alert configuration
- Pilot user onboarding

Appendix A: Environment Configuration

```
# .env.example

# Application
APP_ENV=development
DEBUG=true
LOG_LEVEL=INFO

# Database
DATABASE_URL=postgresql://cbi:password@localhost:5432/cbi

# Redis
REDIS_URL=redis://localhost:6379

# Anthropic
ANTHROPIC_API_KEY=sk-ant-...

# Telegram
TELEGRAM_BOT_TOKEN=123456:ABC-DEF...
TELEGRAM_WEBHOOK_URL=https://your-domain.com/webhook/telegram

# WhatsApp (for production)
WHATSAPP_PHONE_NUMBER_ID=
WHATSAPP_ACCESS_TOKEN=
WHATSAPP_APP_SECRET=
WHATSAPP_VERIFY_TOKEN=

# Security
JWT_SECRET=your-256-bit-secret
JWT_EXPIRE_HOURS=24
ENCRYPTION_KEY=your-32-byte-key
ENCRYPTION_SALT=your-salt
PHONE_HASH_SALT=your-phone-salt

# CORS
CORS_ORIGINS=["http://localhost:3000"]
```

Appendix B: Project Structure

```
cbi/
├── api/
│   ├── __init__.py
│   ├── main.py                 # FastAPI application
│   ├── deps.py                 # Dependency injection
│   ├── middleware.py           # Custom middleware
│   ├── routes/
│   │   ├── __init__.py
│   │   ├── webhook.py          # Telegram/WhatsApp webhooks
│   │   ├── reports.py          # Reports CRUD
│   │   ├── notifications.py    # Notifications
│   │   ├── analytics.py        # Analyst queries
│   │   └── auth.py             # Authentication
│   └── schemas/
│       ├── __init__.py
│       ├── reports.py          # Pydantic schemas
│       └── auth.py
│
├── agents/
│   ├── __init__.py
│   ├── state.py                # ConversationState definition
│   ├── graph.py                # LangGraph workflow
│   ├── reporter.py             # Reporter Agent
│   ├── surveillance.py         # Surveillance Agent
│   ├── analyst.py              # Analyst Agent
│   └── prompts.py              # System prompts
│
├── services/
│   ├── __init__.py
│   ├── state.py                # State management
│   ├── crypto.py               # Encryption utilities
│   ├── auth.py                 # JWT handling
│   ├── message_queue.py        # Redis Streams
│   └── messaging/
│       ├── __init__.py
│       ├── base.py             # Abstract gateway
│       ├── telegram.py         # Telegram implementation
│       ├── whatsapp.py         # WhatsApp implementation
│       └── factory.py          # Gateway factory
│
├── db/
│   ├── __init__.py
│   ├── session.py              # Database connection
│   ├── models.py               # SQLAlchemy models
│   └── queries.py              # Database queries
│
├── workers/
│   ├── __init__.py
│   └── main.py                 # Worker entry point
│
├── config/
│   ├── __init__.py
│   ├── settings.py             # Pydantic settings
│   ├── llm_config.py           # LLM configurations
│   └── logging.py              # Logging setup
│
├── migrations/
│   └── 001_initial_schema.sql
│
├── tests/
```

```
│   ├── unit/
│   ├── integration/
│   └── agents/
│
├── dashboard/                    # Next.js frontend
│   ├── src/
│   ├── package.json
│   └── Dockerfile
│
├── terraform/                    # Infrastructure as Code
│   ├── main.tf
│   ├── variables.tf
│   └── outputs.tf
│
├── docker-compose.yml
├── Dockerfile
├── pyproject.toml
├── requirements.txt
└── README.md
```

— End of Document —