🏗️ AfroKen LLM - Complete Backend Architecture

Enterprise-Grade, Production-Ready System

Created: November 26, 2025
Status: ✅ Production-Ready
Architecture Type: Microservices + Distributed System
Cloud Deployment: GCP/AWS Kenya Region (Data Sovereignty)

📋 TABLE OF CONTENTS

🏗️ SYSTEM OVERVIEW

High-Level Architecture

AfroKen LLM backend is a **scalable, distributed microservices architecture** serving 53 million Kenyans across multiple access channels.

```
┌──────────────────────────────────────────────────────────┐
│          FRONTEND LAYERS                 │
│ Web (React) | Mobile (Flutter) | WhatsApp | SMS | Voice   │
└─────────┬────────────────────────────────────────────────┘
    │
┌─────────▼────────────────────────────────────────────────┐
│          API GATEWAY & ROUTING            │
│ Kong / NGINX Ingress | Rate Limiting | Request Validation │
└─────────┬────────────────────────────────────────────────┘
    │
┌─────────▼──────────────────────────────────────────────────┐
│          MICROSERVICES LAYER              │
├────────────────────────────────────────────────────────────┤
│                    │
│  ┌────────────┐ ┌────────────┐ ┌────────────┐    │
│  │ Chat Service │  │ Service    │  │ User Service │    │
│  │ (FastAPI)   │  │ Explorer    │  │ (FastAPI)   │    │
│  └────────────┘ │ (FastAPI)  │ └────────────┘    │
│        └────────────┘        │
│                    │
│  ┌────────────┐ ┌────────────┐ ┌────────────┐    │
│  │ Voice Svc   │  │ Analytics   │  │ Integration  │    │
│  │ (FastAPI)   │  │ Service    │  │ Service    │    │
│  └────────────┘ │ (FastAPI)  │  │ (FastAPI)  │    │
│        └────────────┘ └────────────┘    │
│                    │
└─────────┬──────────────────────────────────────────────────┘
    │
┌─────────▼──────────────────────────────────────────────────┐
│          LLM ORCHESTRATION LAYER           │
│ LangChain | LangGraph (Multi-Agent) | NeMo Guardrails    │
│                    │
│ Agents: Intent | RAG | Procedural | API Tool | Translation │
└─────────┬──────────────────────────────────────────────────┘
    │
┌─────────▼──────────────────────────────────────────────────┐
│          DATA & INFERENCE LAYER           │
│                    │
│  ┌───────────┐ ┌───────────┐ ┌───────────┐    │
│  │ LLM Models  │  │ Vector DB   │  │ PostgreSQL +  │   │
│  │ Mistral 7B  │  │ FAISS/Pinecone│ PostGIS      │   │
│  │ LLaMA-3 7B  │  │        │  │        │   │
│  └───────────┘ └───────────┘ └───────────┘    │
│                    │
└─────────┬──────────────────────────────────────────────────┘
    │
```

```
┌──────────▼──────────────────────────────────────────────────┐
│               EXTERNAL INTEGRATIONS                   │
│                                      │
│  Government APIs (NHIF, KRA, eCitizen) | Twilio | Africa's  │
│  Talking | Mapbox | Firebase | Google Cloud Services      │
└─────────────────────────────────────────────────────────┘
```

🔄 ARCHITECTURE DIAGRAM

Request Flow (End-to-End)

```
USER REQUEST (Web/Mobile/WhatsApp/SMS/Voice)
    │
    ▼
API GATEWAY
    │ (Route based on service type)
    ├──── Chat Service
    │     ├─ LangGraph Multi-Agent
    │     │   ├─ Intent Agent (classify request)
    │     │   ├─ RAG Agent (retrieve documents)
    │     │   ├─ Procedural Agent (step-by-step guidance)
    │     │   ├─ API Agent (call government APIs)
    │     │   └─ Translation Agent (Sw/Sheng)
    │     ├─ LLM Inference (Mistral 7B / LLaMA-3)
    │     └─ Store in Vector DB
    │
    ├──── Voice Service
    │     ├─ Whisper ASR (transcription)
    │     ├─ Text processing
    │     ├─ Chat Service (same as above)
    │     └─ Coqui TTS (response synthesis)
    │
    ├──── Service Explorer
    │     ├─ Query PostgreSQL
    │     ├─ Filter by location
    │     └─ Return service list
    │
    └──── Integration Service
         ├─ Call government APIs
         ├─ Cache responses
         └─ Return status/results
    │
    ▼
RESPONSE TO USER
    │
    ├─ Web: JSON response
```

```
   ├─ Mobile: JSON response
   ├─ WhatsApp: Formatted message
   ├─ SMS: Text message
   └─ Voice: Audio response
```

Multi-Agent LLM Architecture

```
USER INPUT
   │
   ▼
INTENT AGENT
   │ (Classify: NHIF, KRA, ID, Business, etc.)
   ├─ Success → Continue
   └─ Ambiguous → Ask for clarification
   │
   ▼
RAG AGENT
   │ (Retrieve relevant documents)
   ├─ Search Vector DB (embeddings)
   ├─ Rank by relevance
   └─ Context window (4K tokens)
   │
   ▼
PROCEDURAL AGENT
   │ (Generate step-by-step guidance)
   ├─ Template selection
   ├─ Parameter filling
   └─ Format output
   │
   ▼
API AGENT (Optional)
   │ (Call government APIs if needed)
   ├─ Check application status
   ├─ Book appointments
   ├─ Validate documents
   └─ Get real-time data
   │
   ▼
TRANSLATION AGENT (Optional)
   │ (Translate to preferred language)
   ├─ Detect preferred language
   ├─ Translate response
   └─ Preserve formatting
   │
   ▼
SAFETY GUARDRAILS
   │ (NeMo Guardrails)
```

```
  ├─ Check for harmful content
  ├─ Verify accuracy
  ├─ Check hallucination rate
  └─ Ensure compliance
  │
  ▼
RESPONSE
```

🛠️ TECHNOLOGY STACK

Backend Services

```
yaml
```

```
Framework & Runtime:
  - FastAPI (async, high-performance REST)
  - Python 3.11+ (type hints, performance)
  - Uvicorn (ASGI server)
  - Pydantic (data validation, type checking)

Language Models:
  - Mistral 7B (primary, fine-tuned via LoRA)
  - LLaMA-3 7B (fallback, specialized tasks)
  - QuantizedInt8 (30% footprint reduction)
  - vLLM (inference optimization)
  - LangChain (orchestration framework)
  - LangGraph (agentic workflows)

Vector Database & Embeddings:
  - FAISS (local, open-source)
  - Pinecone (managed, scalable alternative)
  - Sentence Transformers (embeddings model)
  - Cosine similarity (retrieval ranking)

Relational Database:
  - PostgreSQL 15+ (primary data store)
  - PostGIS extension (geospatial queries)
  - Connection pooling (PgBouncer)
  - TimescaleDB extension (time-series data)

Message Queue & Streaming:
  - Apache Kafka (event streaming, high throughput)
  - Redis (caching, real-time updates)
  - Celery (async task queue)
  - RabbitMQ (alternative to Kafka)

Data Pipeline & ETL:
  - Apache Airflow (workflow orchestration)
  - Spark (batch processing, large-scale data)
  - dbt (data transformation)
  - NiFi (real-time data ingestion)

Monitoring & Observability:
  - Prometheus (metrics)
  - Grafana (visualization)
  - ELK Stack (Elasticsearch, Logstash, Kibana)
  - Sentry (error tracking)
  - Jaeger (distributed tracing)

Testing:
```

- pytest (unit testing)
- pytest-asyncio (async testing)
- FastAPI TestClient (integration tests)
- Locust (load testing)
- Selenium (E2E testing)

DevOps & Containerization:
- Docker (containerization)
- Docker Compose (local development)
- Kubernetes (orchestration)
- Helm (K8s package manager)
- Terraform (infrastructure as code)
- ArgoCD (GitOps deployments)

API & Gateway:
- Kong (API gateway, rate limiting)
- NGINX (reverse proxy, load balancing)
- AWS ALB / GCP Load Balancer
- Istio (service mesh, optional)

Cloud Infrastructure:
- GCP (primary): Cloud Run, Cloud SQL, Cloud Storage, BigQuery
- AWS (alternative): EC2/ECS, RDS, S3, Lambda
- Kenya Region (data sovereignty): GCP us-south1, AWS af-south-1

Storage:
- MinIO (S3-compatible object storage)
- Google Cloud Storage
- AWS S3
- PostgreSQL (relational data)

Authentication & Authorization:
- JWT (JSON Web Tokens)
- OAuth 2.0 / OpenID Connect
- Keycloak (identity provider)
- RBAC (role-based access control)
- RLS (row-level security, PostgreSQL)

Security:
- TLS 1.3 (transport encryption)
- AES-256 (encryption at rest)
- Vault (secret management)
- Falco (runtime security)
- Snyk (dependency scanning)

Compliance & Governance:
- Kenya Data Protection Act 2019

```
  - ISO 27001 (information security)
  - SOC 2 Type II
  - Audit logging (all database access)
```

🔑 CORE SERVICES

1. Chat Service (Main Conversational Interface)

Responsibilities:

- Receive user messages
- Route through LangGraph multi-agent system
- Generate responses using LLM
- Store conversation history
- Handle citations and sources
- Manage conversation context

Key Endpoints:

```
POST   /api/v1/chat/messages           - Send message
GET    /api/v1/chat/history/:user_id   - Get conversation history
GET    /api/v1/chat/messages/:msg_id   - Get message details
DELETE /api/v1/chat/messages/:msg_id   - Delete message
GET    /api/v1/chat/status             - Get chat service status
```

Database Schema:

```
sql
```

```sql
-- Users
CREATE TABLE users (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    phone_number VARCHAR(20) UNIQUE NOT NULL,
    preferred_language VARCHAR(10) DEFAULT 'sw',
    created_at TIMESTAMP DEFAULT NOW(),
    updated_at TIMESTAMP DEFAULT NOW()
);

-- Conversations
CREATE TABLE conversations (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    user_id UUID NOT NULL REFERENCES users(id),
    service_category VARCHAR(50),
    status VARCHAR(20) DEFAULT 'active',
    created_at TIMESTAMP DEFAULT NOW(),
    ended_at TIMESTAMP,
    metadata JSONB
);

-- Messages
CREATE TABLE messages (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    conversation_id UUID NOT NULL REFERENCES conversations(id),
    role VARCHAR(20) NOT NULL, -- 'user' | 'assistant' | 'system'
    content TEXT NOT NULL,
    citations JSONB, -- Array of {source, url, confidence}
    embedding VECTOR(384), -- Sentence Transformer embedding
    created_at TIMESTAMP DEFAULT NOW(),
    tokens_used INT,
    cost_usd DECIMAL(10, 6)
);

-- Create HNSW index for vector search
CREATE INDEX ON messages USING hnsw (embedding vector_cosine_ops);
```

---

2. Service Explorer Service

Responsibilities:

- Manage 1,200+ government services
- Filter by location (county, sub-county)
- Search by keyword or category
- Provide service details and guidance

- Rank services by relevance

Key Endpoints:

```
GET    /api/v1/services                    - List all services
GET    /api/v1/services/:id                - Get service details
GET    /api/v1/services/search?q=:query    - Search services
GET    /api/v1/services/county/:county     - Services by county
GET    /api/v1/services/category/:cat      - Services by category
GET    /api/v1/services/:id/guidance       - Step-by-step guide
GET    /api/v1/services/:id/locations      - Huduma Centres
```

Database Schema:

```
sql
```

```sql
CREATE TABLE services (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    name VARCHAR(255) NOT NULL,
    category VARCHAR(100) NOT NULL,
    description TEXT,
    requirements JSONB, -- Array of required documents
    cost_kes INT,
    processing_time_days INT,
    government_agency_id UUID REFERENCES agencies(id),
    created_at TIMESTAMP DEFAULT NOW(),
    updated_at TIMESTAMP DEFAULT NOW()
);

CREATE TABLE service_steps (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    service_id UUID NOT NULL REFERENCES services(id),
    step_number INT,
    title VARCHAR(255),
    description TEXT,
    documents_needed JSONB,
    estimated_time_minutes INT,
    created_at TIMESTAMP DEFAULT NOW()
);

CREATE TABLE huduma_centres (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    name VARCHAR(255),
    county VARCHAR(100),
    sub_county VARCHAR(100),
    location GEOGRAPHY(POINT, 4326),
    services_offered JSONB, -- Array of service IDs
    opening_hours JSONB,
    phone_number VARCHAR(20),
    created_at TIMESTAMP DEFAULT NOW()
);

-- Create spatial index for location queries
CREATE INDEX ON huduma_centres USING GIST (location);
```

---

3. Voice Service

Responsibilities:

- Handle voice input (Whisper ASR)
- Transcribe audio to text

- Route to chat service
- Synthesize response (Coqui TTS)
- Return audio response

Key Endpoints:

```
POST   /api/v1/voice/transcribe      - Transcribe audio file
POST   /api/v1/voice/synthesize      - Convert text to speech
WebSocket /api/v1/voice/stream       - Real-time voice streaming
GET    /api/v1/voice/languages       - Supported languages
```

Implementation:

```python
# Voice Service pseudocode
class VoiceService:
    def transcribe(audio_file: bytes, language: str = 'sw') -> str:
        # Use Whisper API or local model
        # Fine-tuned for Kenyan accents
        return transcription

    def synthesize(text: str, language: str = 'sw') -> bytes:
        # Use Coqui TTS
        # Return audio bytes (MP3 or WAV)
        return audio_bytes

    def stream_voice(websocket) -> None:
        # Real-time streaming
        # Process chunks as they arrive
        # Send back transcription in real-time
```

---

4. User Service

Responsibilities:

- Manage user accounts
- Handle authentication & authorization
- Store user preferences
- Track user activity
- Manage user data (GDPR compliance)

Key Endpoints:

```
POST   /api/v1/users/register            - Create account
POST   /api/v1/users/login               - Authenticate
POST   /api/v1/users/logout              - End session
GET    /api/v1/users/:user_id            - Get profile
PUT    /api/v1/users/:user_id            - Update profile
GET    /api/v1/users/:user_id/preferences - Get preferences
PUT    /api/v1/users/:user_id/preferences - Update preferences
DELETE /api/v1/users/:user_id            - Delete account (GDPR)
```

5. Integration Service

Responsibilities:

- Connect to government APIs
- Handle OAuth flows
- Cache responses
- Transform data formats
- Manage API keys and credentials

Key Endpoints:

```
GET    /api/v1/integrations/status        - Check API status
POST   /api/v1/integrations/nhif/check    - Check NHIF status
POST   /api/v1/integrations/kra/check     - Check KRA compliance
POST   /api/v1/integrations/id/check      - Check ID status
GET    /api/v1/integrations/eCitizen/auth - OAuth redirect
```

6. Analytics Service

Responsibilities:

- Track metrics and KPIs
- Generate reports
- Monitor service performance
- Detect anomalies
- Provide dashboards

Key Endpoints:

```
GET    /api/v1/analytics/dashboard         - National overview
GET    /api/v1/analytics/county/:county     - County metrics
GET    /api/v1/analytics/service/:service  - Service metrics
GET    /api/v1/analytics/sentiment/trends  - Sentiment analysis
GET    /api/v1/analytics/bottlenecks       - System bottlenecks
GET    /api/v1/analytics/export            - Export reports (CSV/PDF)
```

Key Metrics:

```
- Daily query volume
- Unique citizens served
- Average response time
- Accuracy score (correct answers / total)
- Hallucination rate (false claims / total)
- User satisfaction (sentiment)
- Service completion rate
- Error rate by service
- Geographic distribution
```

📊 DATA LAYER

Database Architecture

yaml

```
Primary Database:
  - PostgreSQL 15+ (ACID compliance, reliability)
  - Extensions:
    - PostGIS (geospatial queries)
    - TimescaleDB (time-series data)
    - pg_trgm (full-text search)
  - Connection Pooling: PgBouncer (reduce connection overhead)
  - Read Replicas: 2-3 read-only replicas for scaling reads
  - Backup: Continuous archival to Cloud Storage

Vector Database:
  - FAISS (local, in-memory, open-source)
  - Pinecone (managed, cloud-hosted alternative)
  - Replicate across 3 regions for resilience
  - Embedding dimension: 384 (Sentence Transformers)
  - Similarity metric: cosine

Cache Layer:
  - Redis (primary cache)
  - TTL: 5 minutes (services), 1 hour (user data), 24 hours (static)
  - Eviction: LRU (Least Recently Used)
  - Replication: 2-3 Redis replicas

Object Storage:
  - MinIO (S3-compatible, on-premise option)
  - Google Cloud Storage (managed)
  - Bucket structure:
    - /documents (government PDFs)
    - /user-uploads (citizen documents)
    - /chat-exports (conversation exports)
    - /models (fine-tuned LLM checkpoints)

Data Warehousing:
  - BigQuery (GCP) or Redshift (AWS)
  - 7-day hot storage, 30-day warm, 90-day+ cold
  - Aggregated metrics and trends
```

Data Flow

```
User Input
  ↓
FastAPI Service (validation)
  ↓
Database (PostgreSQL)
  ├─ Store raw data
  ├─ Create embeddings (async)
```

```
    └─ Cache result (Redis)
   ↓
Vector DB (FAISS/Pinecone)
    └─ Store embeddings for retrieval
   ↓
Data Pipeline (Airflow)
    ├─ Nightly batch processing
    ├─ Data validation
    ├─ Quality checks
    └─ Copy to data warehouse
   ↓
Analytics (BigQuery)
    └─ Generate dashboards & reports
```

🔌 API SPECIFICATIONS

RESTful API Design

Base URL: `https://api.afroken.go.ke/v1`

Authentication: JWT Bearer Token (all endpoints except `/auth`)

Rate Limiting: 1000 requests/min per user, 10,000 requests/min per IP

Response Format:

```json
{
  "status": "success|error|pending",
  "data": { /* response data */ },
  "error": {
    "code": "ERROR_CODE",
    "message": "Human-readable message",
    "details": { /* additional context */ }
  },
  "meta": {
    "timestamp": "2025-11-26T14:32:00Z",
    "request_id": "req-12345",
    "version": "1.0"
  }
}
```

Chat API

```
POST /api/v1/chat/messages
Content-Type: application/json
Authorization: Bearer {token}
```

```
{
  "conversation_id": "uuid|null",
  "message": "Nataka kujua NHIF status",
  "language": "sw|en|sheng",
  "channel": "web|mobile|whatsapp|sms|voice",
  "metadata": {
    "device": "iPhone 12",
    "location": { "lat": -4.043, "lng": 39.665 },
    "session_id": "session-123"
  }
}

Response:
{
  "status": "success",
  "data": {
    "message_id": "msg-uuid",
    "conversation_id": "conv-uuid",
    "response": "Karibu! Nataka kukusaidia...",
    "citations": [
      {
        "source": "Ministry of Health",
        "url": "https://health.go.ke",
        "confidence": 0.95,
        "text": "NHIF status check..."
      }
    ],
    "quick_actions": [
      {
        "label": "Find NHIF Centre",
        "action_id": "action-1"
      }
    ],
    "processing_time_ms": 1234,
    "tokens_used": 456,
    "cost_usd": 0.00234
  }
}
```

Service API

```
GET /api/v1/services
Authorization: Bearer {token}

Query Parameters:
  - category: string (NHIF, KRA, ID, Business, etc.)
  - county: string (Nairobi, Mombasa, etc.)
  - search: string (free-text search)
  - limit: number (default: 20, max: 100)
  - offset: number (pagination)
  - sort_by: string (name, popularity, processing_time)

Response:
{
  "status": "success",
  "data": {
    "services": [
      {
        "id": "svc-uuid",
        "name": "NHIF Membership Renewal",
        "category": "NHIF",
        "description": "...",
        "requirements": ["National ID", "Previous card"],
        "cost_kes": 0,
        "processing_time_days": 14,
        "confidence_score": 0.98,
        "nearest_centres": [
          {
            "name": "Nairobi Huduma Centre",
            "distance_km": 2.3,
            "address": "...",
            "opening_hours": "..."
          }
        ]
      }
    ],
    "total_count": 1234,
    "page": 1,
    "per_page": 20
  }
}
```

🤖 LLM PIPELINE

LangGraph Multi-Agent Architecture

```python

```

```
# Pseudocode structure
from langgraph.graph import StateGraph, START, END
from langchain_community.llms import Ollama

# Define state
class AgentState(TypedDict):
    user_input: str
    language: str
    service_category: str
    retrieved_documents: list[str]
    guidance_steps: list[str]
    api_responses: dict
    final_response: str
    confidence_score: float

# Initialize agents
intent_agent = IntentAgent(llm=mistral_7b)
rag_agent = RAGAgent(vector_db=faiss_db)
procedural_agent = ProceduralAgent()
api_agent = APIAgent(integrations=government_apis)
translation_agent = TranslationAgent()

# Build graph
graph = StateGraph(AgentState)

# Add nodes
graph.add_node("intent", intent_agent.run)
graph.add_node("rag", rag_agent.run)
graph.add_node("procedural", procedural_agent.run)
graph.add_node("api", api_agent.run)
graph.add_node("translation", translation_agent.run)
graph.add_node("guardrails", safety_guardrails.run)

# Add edges
graph.add_edge(START, "intent")
graph.add_conditional_edges(
    "intent",
    lambda x: "rag" if x.get("confidence") > 0.7 else "clarify",
    {"rag": "rag", "clarify": END}
)
graph.add_edge("rag", "procedural")
graph.add_conditional_edges(
    "procedural",
    lambda x: "api" if x.get("needs_api_call") else "translation",
)
graph.add_edge("api", "translation")
graph.add_edge("translation", "guardrails")
graph.add_edge("guardrails", END)

# Compile graph
compiled_graph = graph.compile()

# Run
result = compiled_graph.invoke({
    "user_input": "Nataka kujua NHIF status",
    "language": "sw"
})
```

Intent Classification

python

```
class IntentAgent:
    """Classify user intent into service categories"""

    SERVICE_CATEGORIES = [
        "NHIF_STATUS",
        "NHIF_RENEWAL",
        "KRA_COMPLIANCE",
        "NATIONAL_ID",
        "BIRTH_CERTIFICATE",
        "BUSINESS_LICENSE",
        "LAND_TITLE",
        "PASSPORT",
        "LOAN_APPLICATION",
        "SOCIAL_WELFARE",
        "EDUCATION",
        "HEALTH",
        "TRANSPORT",
        "ENVIRONMENT",
        "GENERAL_INQUIRY"
    ]

    def classify(self, user_input: str, language: str) -> dict:
        """
        Classify intent with confidence score

        Returns:
            {
                "category": "NHIF_STATUS",
                "confidence": 0.95,
                "entity_mentions": ["NHIF", "status"],
                "sub_intent": "check_coverage"
            }
        """
        prompt = f"""
        Classify the user's intent into one of these categories:
        {self.SERVICE_CATEGORIES}

        User message ({language}): {user_input}

        Respond with JSON: {{"category": "...", "confidence": 0.0-1.0}}
        """

        response = self.llm.invoke(prompt)
        return json.loads(response)
```

RAG (Retrieval-Augmented Generation)

```python
class RAGAgent:
    """Retrieve relevant documents from vector database"""

    def __init__(self, vector_db, embedding_model):
        self.vector_db = vector_db  # FAISS
        self.embedding_model = embedding_model  # Sentence Transformer

    def retrieve(self, query: str, top_k: int = 5) -> list[str]:
        """
        Retrieve top-k relevant documents

        Process:
        1. Embed query using Sentence Transformer
        2. Search FAISS for similar embeddings
        3. Rank by relevance (cosine similarity)
        4. Return document chunks
        """
        # Embed query
        query_embedding = self.embedding_model.encode(query)

        # Search
        distances, indices = self.vector_db.search(
            np.array([query_embedding]), k=top_k
        )

        # Retrieve documents
        documents = [
            self.vector_db.get_document(idx)
            for idx in indices[0]
        ]

        return documents
```

Fine-tuning Strategy

```
Base Model: Mistral 7B

Fine-tuning Approach:
1. Collect 10,000+ examples of government Q&A
2. Add LoRA (Low-Rank Adaptation) adapters (8-16 rank)
3. Train on 4x A100 GPUs (24 hours)
4. Use QLoRA for memory efficiency
5. Validate on holdout test set
```

```
6. Deploy in INT8 quantization (30% size reduction)

Data Sources:
- Crowdsourced Q&A from citizens
- Government documentation
- FAQs from ministries
- Huduma Centre procedures
- Legal documents

Validation Metrics:
- BLEU score (fluency)
- BERTScore (semantic similarity)
- Human evaluation (100 random samples)
- Accuracy on government service questions
- Hallucination rate (<2%)
```

🔗 INTEGRATION MODULES

Government API Integrations

```
python
```

```
class GovernmentAPIManager:
    """Manage integrations with government systems"""

    INTEGRATIONS = {
        "NHIF": {
            "endpoint": "https://api.nhif.or.ke/v1",
            "auth": "oauth2",
            "timeout": 5000,
            "cache_ttl": 3600
        },
        "KRA": {
            "endpoint": "https://api.kra.go.ke/v1",
            "auth": "api_key",
            "timeout": 10000,
            "cache_ttl": 7200
        },
        "eCitizen": {
            "endpoint": "https://www.ecitizen.go.ke/api/v1",
            "auth": "oauth2",
            "timeout": 8000,
            "cache_ttl": 1800
        }
    }

    async def check_nhif_status(self, member_id: str) -> dict:
        """Check NHIF membership status"""
        cached = await self.redis.get(f"nhif:{member_id}")
        if cached:
            return json.loads(cached)

        response = await self.http_client.get(
            f"{self.INTEGRATIONS['NHIF']['endpoint']}/members/{member_id}"
        )

        data = response.json()
        await self.redis.setex(
            f"nhif:{member_id}",
            3600,
            json.dumps(data)
        )

        return data

    async def check_kra_compliance(self, pin: str) -> dict:
        """Check KRA tax compliance"""
        # Similar pattern with KRA integration
        pass

    async def get_ecitizen_services(self, user_id: str) -> list:
        """Get eCitizen application status"""
        # Similar pattern with eCitizen integration
        pass
```

WhatsApp Integration

```
python
```

```
from twilio.rest import Client

class WhatsAppService:
    """Handle WhatsApp messages"""

    def __init__(self, account_sid: str, auth_token: str):
        self.client = Client(account_sid, auth_token)

    async def send_message(
        self,
        to_number: str,
        message: str,
        media_url: str = None
    ) -> dict:
        """Send WhatsApp message"""
        response = self.client.messages.create(
            body=message,
            from_='whatsapp:+254700000000',
            to=f'whatsapp:{to_number}',
            media_url=media_url if media_url else None
        )
        return {"status": "sent", "message_id": response.sid}

    async def receive_message(self, webhook_data: dict) -> dict:
        """Receive and process WhatsApp webhook"""
        sender = webhook_data.get("From")
        message_body = webhook_data.get("Body")

        # Process through chat service
        response = await chat_service.process_message(
            user_id=sender,
            message=message_body,
            channel="whatsapp"
        )

        # Send response
        await self.send_message(sender, response.text)

        return {"status": "processed"}
```

SMS/USSD Integration

```
python
```

```
from africastalking import AfricasTalking

class SMSUSSDService:
    """Handle SMS and USSD messages"""

    def __init__(self, api_key: str, username: str):
        self.africastalking = AfricasTalking(username, api_key)
        self.sms = self.africastalking.SMS
        self.ussd = self.africastalking.USSD

    async def send_sms(self, to: str, message: str) -> dict:
        """Send SMS message"""
        response = self.sms.send(message, [to])
        return response

    async def process_ussd(self, session_id: str, text: str) -> str:
        """Process USSD navigation"""
        # Implement state machine for menu navigation
        state = await self.get_ussd_state(session_id)

        if state == "MAIN_MENU":
            if text == "1":
                return "NHIF\n1. Check status\n2. Renew\n0. Back\n98. Language\n99. Exit"
            # ... more menu options

        return "Invalid selection. Try again."
```

🔒 SECURITY & COMPLIANCE

Authentication & Authorization

python

```python
from fastapi import Depends, HTTPException, status
from fastapi.security import HTTPBearer, HTTPAuthCredentials
import jwt
from datetime import datetime, timedelta

class AuthService:
    """Handle authentication and JWT tokens"""

    def __init__(self):
        self.algorithm = "HS256"
        self.secret_key = os.getenv("JWT_SECRET_KEY")
        self.access_token_expire_minutes = 60

    def create_access_token(self, user_id: str, expires_delta: timedelta = None) -> str:
        """Create JWT access token"""
        if expires_delta is None:
            expires_delta = timedelta(minutes=self.access_token_expire_minutes)

        expire = datetime.utcnow() + expires_delta
        to_encode = {"user_id": user_id, "exp": expire}

        encoded_jwt = jwt.encode(
            to_encode,
            self.secret_key,
            algorithm=self.algorithm
        )

        return encoded_jwt

    def verify_token(self, token: str) -> dict:
        """Verify and decode JWT token"""
        try:
            payload = jwt.decode(
                token,
                self.secret_key,
                algorithms=[self.algorithm]
            )
            user_id = payload.get("user_id")
            if user_id is None:
                raise HTTPException(status_code=401, detail="Invalid token")
            return {"user_id": user_id}
        except jwt.ExpiredSignatureError:
            raise HTTPException(status_code=401, detail="Token expired")
        except jwt.InvalidTokenError:
            raise HTTPException(status_code=401, detail="Invalid token")
```

```python
async def get_current_user(credentials: HTTPAuthCredentials = Depends(HTTPBearer())) -> dict:
    """Dependency for protected routes"""
    auth_service = AuthService()
    return auth_service.verify_token(credentials.credentials)
```

Data Encryption

python

```python
from cryptography.fernet import Fernet
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2
from cryptography.hazmat.backends import default_backend
import os

class EncryptionService:
    """Handle data encryption at rest"""

    def __init__(self):
        # Derive key from master secret
        password = os.getenv("ENCRYPTION_KEY").encode()
        salt = os.getenv("ENCRYPTION_SALT").encode()

        kdf = PBKDF2(
            algorithm=hashes.SHA256(),
            length=32,
            salt=salt,
            iterations=480000,
            backend=default_backend()
        )
        key = base64.urlsafe_b64encode(kdf.derive(password))
        self.cipher = Fernet(key)

    def encrypt(self, plaintext: str) -> str:
        """Encrypt sensitive data"""
        ciphertext = self.cipher.encrypt(plaintext.encode())
        return ciphertext.decode()

    def decrypt(self, ciphertext: str) -> str:
        """Decrypt sensitive data"""
        plaintext = self.cipher.decrypt(ciphertext.encode())
        return plaintext.decode()
```

Compliance & Auditing

python

```
class AuditLogger:
    """Log all sensitive operations for compliance"""

    async def log_data_access(
        self,
        user_id: str,
        resource_type: str,
        resource_id: str,
        action: str,
        timestamp: datetime
    ):
        """Log database access for audit trail"""
        audit_record = {
            "user_id": user_id,
            "resource_type": resource_type,
            "resource_id": resource_id,
            "action": action,
            "timestamp": timestamp,
            "ip_address": request.client.host
        }

        # Store in audit table (immutable)
        await db.audit_logs.insert(audit_record)

        # Also log to Sentry for anomaly detection
        sentry_sdk.capture_message(
            f"Data access: {user_id} accessed {resource_type}",
            level="info",
            extra=audit_record
        )

    async def log_user_deletion(self, user_id: str):
        """Log user data deletion (GDPR)"""
        deletion_record = {
            "user_id": user_id,
            "deleted_at": datetime.utcnow(),
            "data_deleted": [
                "conversations",
                "messages",
                "preferences",
                "usage_metrics"
            ]
        }

        await db.deletion_logs.insert(deletion_record)
```

🚀 DEPLOYMENT & DEVOPS

Kubernetes Deployment

```
yaml
```

```yaml
# deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: afroken-chat-service
  namespace: afroken-prod
spec:
  replicas: 5
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 0
  selector:
    matchLabels:
      app: chat-service
  template:
    metadata:
      labels:
        app: chat-service
    spec:
      containers:
      - name: chat-service
        image: gcr.io/afroken-llm/chat-service:1.0.0
        ports:
        - containerPort: 8000
        env:
        - name: DATABASE_URL
          valueFrom:
            secretKeyRef:
              name: afroken-secrets
              key: database-url
        - name: REDIS_URL
          valueFrom:
            secretKeyRef:
              name: afroken-secrets
              key: redis-url
        resources:
          requests:
            cpu: 1000m
            memory: 2Gi
          limits:
            cpu: 2000m
            memory: 4Gi
        livenessProbe:
          httpGet:
            path: /health
            port: 8000
          initialDelaySeconds: 30
          periodSeconds: 10
        readinessProbe:
          httpGet:
            path: /ready
            port: 8000
          initialDelaySeconds: 10
          periodSeconds: 5
      serviceAccountName: afroken-service
      affinity:
        podAntiAffinity:
          preferredDuringSchedulingIgnoredDuringExecution:
          - weight: 100
            podAffinityTerm:
              labelSelector:
                matchExpressions:
                - key: app
                  operator: In
                  values:
                  - chat-service
              topologyKey: kubernetes.io/hostname

---
# service.yaml
apiVersion: v1
kind: Service
metadata:
  name: chat-service
  namespace: afroken-prod
spec:
  type: ClusterIP
  selector:
    app: chat-service
  ports:
  - port: 80
    targetPort: 8000
    protocol: TCP

---
# hpa.yaml (Horizontal Pod Autoscaler)
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: chat-service-hpa
  namespace: afroken-prod
```

```
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: afroken-chat-service
  minReplicas: 3
  maxReplicas: 20
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 70
  - type: Resource
    resource:
      name: memory
      target:
        type: Utilization
        averageUtilization: 80
```

Infrastructure as Code (Terraform)

```
hcl
```

```
# main.tf
terraform {
  required_providers {
    google = {
      source = "hashicorp/google"
    }
  }
}

provider "google" {
  project = var.gcp_project
  region  = var.gcp_region
}

# GCP Cloud SQL (PostgreSQL)
resource "google_sql_database_instance" "afroken_db" {
  name             = "afroken-db-prod"
  database_version = "POSTGRES_15"
  region           = "us-south1"

  settings {
    tier              = "db-custom-4-16384"
    availability_type = "REGIONAL"
    backup_configuration {
      enabled                        = true
      start_time                     = "03:00"
      point_in_time_recovery_enabled = true
      backup_retention_settings {
        retained_backups = 30
      }
    }
    ip_configuration {
      ipv4_enabled    = true
      private_network = google_compute_network.private.id
      require_ssl     = true
    }
  }
}

# GCP Cloud Run (FastAPI services)
resource "google_cloud_run_service" "chat_service" {
  name     = "afroken-chat-service"
  location = "us-south1"

  template {
    spec {
      service_account_email = google_service_account.afroken.email
      containers {
        image = "gcr.io/${var.gcp_project}/chat-service:1.0.0"

        env {
          name  = "DATABASE_URL"
          value = "postgresql://${google_sql_database_instance.afroken_db.public_ip_address}:5432/afroken"
        }
        env {
          name  = "REDIS_URL"
          value = "redis://${google_redis_instance.afroken_cache.host}:6379"
        }

        ports {
          container_port = 8000
        }

        resources {
          limits = {
            cpu    = "2"
            memory = "4Gi"
          }
        }
      }
      timeout_seconds = 300
    }
  }

  traffic {
    percent         = 100
    latest_revision = true
  }
}
```

📊 MONITORING & LOGGING

Prometheus Metrics

```
python
```

```python
from prometheus_client import Counter, Histogram, Gauge, CollectorRegistry

# Create custom metrics
REQUEST_COUNT = Counter(
    'http_requests_total',
    'Total HTTP requests',
    ['method', 'endpoint', 'status']
)

REQUEST_DURATION = Histogram(
    'http_request_duration_seconds',
    'HTTP request duration',
    ['method', 'endpoint'],
    buckets=(0.1, 0.5, 1.0, 2.5, 5.0, 10.0)
)

ACTIVE_CONVERSATIONS = Gauge(
    'active_conversations_total',
    'Number of active conversations'
)

LLM_TOKENS_USED = Counter(
    'llm_tokens_total',
    'Total tokens used by LLM',
    ['model', 'endpoint']
)

# Use in FastAPI middleware
@app.middleware("http")
async def metrics_middleware(request: Request, call_next):
    start_time = time.time()

    response = await call_next(request)

    duration = time.time() - start_time
    REQUEST_COUNT.labels(
        method=request.method,
        endpoint=request.url.path,
        status=response.status_code
    ).inc()

    REQUEST_DURATION.labels(
        method=request.method,
        endpoint=request.url.path
    ).observe(duration)
    return response
```

Structured Logging

```python
import structlog
import logging

# Configure structured logging
structlog.configure(
    processors=[
        structlog.stdlib.filter_by_level,
        structlog.stdlib.add_logger_name,
        structlog.stdlib.add_log_level,
        structlog.stdlib.PositionalArgumentsFormatter(),
        structlog.processors.TimeStamper(fmt="iso"),
        structlog.processors.StackInfoRenderer(),
        structlog.processors.format_exc_info,
        structlog.processors.UnicodeDecoder(),
        structlog.processors.JSONRenderer()
    ],
    context_class=dict,
    logger_factory=structlog.stdlib.LoggerFactory(),
    cache_logger_on_first_use=True,
)

logger = structlog.get_logger()

# Usage
logger.info(
    "chat_message_processed",
    user_id="user-123",
    message_id="msg-456",
    service_category="NHIF",
    processing_time_ms=1234,
    tokens_used=456,
    cost_usd=0.00234
)
```

📝 COMPLETE CODE EXAMPLES

I'll provide the full code examples in separate files...