Azure Services Implementation Guide

Medical Document Chatbot - Complete Azure Stack

Project: AI Medical Document Chatbot

Region: India (Central India - Pune / South India - Chennai)

Date: October 14, 2025

Table of Contents

Azure Services Overview

Complete Azure Services List

```
┌─────────────────────────────────────────────────────────┐
│                  AZURE SERVICES STACK                   │
└─────────────────────────────────────────────────────────┘

1. AI & ML SERVICES
   ├─ Azure OpenAI Service (GPT-4)
   ├─ Azure AI Foundry (Med42-Llama3, BioGPT, BiomedCLIP)
   ├─ Azure Document Intelligence (Form Recognizer)
   ├─ Azure AI Search (Vector Database for RAG)
   └─ Azure Machine Learning (Optional - for custom models)

2. COMPUTE SERVICES
   ├─ Azure App Service (Web API hosting)
   ├─ Azure Functions (Serverless background tasks)
   └─ Azure Container Instances (Agent microservices)

3. STORAGE SERVICES
   ├─ Azure Blob Storage (Document storage)
   ├─ Azure SQL Database (Structured data - users, drugs)
   ├─ Azure Cosmos DB (Chat history, NoSQL)
   └─ Azure Redis Cache (Fast data caching)

4. NETWORKING & SECURITY
   ├─ Azure API Management (API Gateway)
   ├─ Azure Application Gateway (Load balancer)
   ├─ Azure Virtual Network (Private networking)
   ├─ Azure Key Vault (Secrets management)
   ├─ Azure AD B2C (User authentication)
   └─ Azure Web Application Firewall (WAF)

5. MONITORING & DEVOPS
   ├─ Azure Application Insights (Monitoring)
   ├─ Azure Monitor (Alerts & dashboards)
   ├─ Azure Log Analytics (Centralized logging)
   ├─ Azure DevOps / GitHub Actions (CI/CD)
   └─ Azure Container Registry (Docker images)

6. ADDITIONAL SERVICES
   ├─ Azure Communication Services (SMS/Email)
   ├─ Azure CDN (Content delivery)
   └─ Azure Backup (Automated backups)
```

Detailed Service Configuration

1. Azure OpenAI Service (Orchestrator Agent)

Purpose: Powers the main orchestrator that routes queries to specialized agents

Configuration:

```
Service: Azure OpenAI
Model: GPT-4 (gpt-4)
Region: East US / Sweden Central (GPT-4 availability)
Deployment: Standard
Pricing Tier: Standard S0

Model Settings:
  - Model Version: gpt-4-turbo (latest)
  - Tokens per Minute (TPM): 80,000
  - Requests per Minute (RPM): 800
  - Max Tokens: 8,192 context window
  - Temperature: 0.3 (for consistent routing)
```

How We Use It:

```
# Example: Orchestrator Agent Implementation

from openai import AzureOpenAI

client = AzureOpenAI(
    api_key="<your-key-from-key-vault>",
    api_version="2024-08-01-preview",
    azure_endpoint="https://<your-resource>.openai.azure.com/"
)

def orchestrate_query(user_message, user_context):
    """
    Main orchestrator that decides which agent(s) to call
    """

    system_prompt = """
    You are a medical assistant orchestrator. Analyze the user's
    query and determine which specialized agents to call:

    - DocumentAgent: For prescription OCR and extraction
    - MedicalQAAgent: For health questions, symptoms, diet
    - DrugInfoAgent: For medication information, interactions
    - ImageAgent: For medical images, lab reports

    Respond with JSON: {"agents": ["AgentName1", "AgentName2"], "reasoning": "..."}
    """

    response = client.chat.completions.create(
        model="gpt-4",
        messages=[
            {"role": "system", "content": system_prompt},
            {"role": "user", "content": f"User query: {user_message}\nContext: {user_context}"}
        ],
        temperature=0.3,
        max_tokens=500
    )

    return response.choices[0].message.content
```

Cost:

- Input: $10 per 1M tokens
- Output: $30 per 1M tokens
- Estimated: $50-100/day for 1000 active users

2. Azure AI Foundry (Specialized AI Agents)

Purpose: Deploy specialized medical AI models for different tasks

Models to Deploy:

Model 1: m42-health-llama3-med42 (Medical Q&A Agent)

```
Service: Azure AI Foundry
Model: m42-health-llama3-med42-70b
Deployment Type: Managed Online Endpoint
Instance Type: Standard_NC24ads_A100_v4 (GPU)
Instances: 2 (for high availability)
Auto-scaling: 2-5 instances based on load

Configuration:
  - Temperature: 0.5
  - Max Tokens: 1024
  - Top P: 0.9
```

Model 2: microsoft-biogpt-large (Drug Info Agent)

```
Service: Azure AI Foundry
Model: microsoft-biogpt-large
Deployment Type: Managed Online Endpoint
Instance Type: Standard_NC12s_v3
Instances: 1-3 (auto-scale)

Configuration:
  - Temperature: 0.2 (factual)
  - Max Tokens: 800
  - Top P: 0.85
```

Model 3: BiomedCLIP-PubMedBERT (Image Agent)

```
Service: Azure AI Foundry
Model: BiomedCLIP-PubMedBERT-base
Deployment Type: Managed Online Endpoint
Instance Type: Standard_D4s_v3
Instances: 1-2

Use Case: Medical image classification
```

How We Use It:

```
# Example: Medical Q&A Agent

from azure.ai.ml import MLClient
from azure.ai.ml.entities import ManagedOnlineEndpoint

# Initialize client
ml_client = MLClient.from_config(credential=credential)

def query_medical_qa_agent(question, context_from_rag):
    """
    Call Med42-Llama3 for medical questions
    """

    endpoint_name = "med42-llama3-endpoint"

    payload = {
        "input_data": {
            "input_string": [
                f"Question: {question}\n\nContext: {context_from_rag}\n\nAnswer:"
            ],
            "parameters": {
                "temperature": 0.5,
                "max_new_tokens": 1024,
                "top_p": 0.9
            }
        }
    }

    response = ml_client.online_endpoints.invoke(
        endpoint_name=endpoint_name,
        request_file=payload
    )

    return response
```

Cost:

- Med42-Llama3: ~$3-5 per hour (GPU compute)
- BioGPT: ~$1-2 per hour
- Total: $100-150/day for moderate usage

---

3. Azure Document Intelligence (Document Agent)

Purpose: OCR and extraction from prescriptions and medical documents

Configuration:

```
Service: Azure Document Intelligence (Form Recognizer)
Resource: Multi-service AI Services
Region: Central India
Pricing Tier: S0 (Standard)

Features Enabled:
  - Prebuilt Read API (general OCR)
  - Custom Model Training (for Indian prescriptions)
  - Layout Analysis
  - Key-Value Pair Extraction
  - Table Extraction
  - Handwriting Recognition
```

Custom Model Training:

```
Training Dataset:
├─ 500+ Indian prescription samples
├─ Handwritten + Typed formats
├─ Multiple doctor handwriting styles
├─ Common Indian medicine names
└─ Regional language annotations (Hindi, Tamil, etc.)

Training Time: 2-4 hours
Model Accuracy Target: >85% on test set
Retraining Frequency: Quarterly
```

How We Use It:

```
# Example: Document Extraction

from azure.ai.formrecognizer import DocumentAnalysisClient
from azure.core.credentials import AzureKeyCredential

# Initialize client
endpoint = "https://<your-resource>.cognitiveservices.azure.com/"
credential = AzureKeyCredential("<key-from-keyvault>")
document_client = DocumentAnalysisClient(endpoint, credential)

def extract_prescription(image_url_or_bytes):
    """
    Extract structured data from prescription image
    """

    # Use custom trained model
    poller = document_client.begin_analyze_document_from_url(
        model_id="custom-prescription-model",  # Your trained model
        document_url=image_url_or_bytes
    )

    result = poller.result()

    # Parse extracted data
    prescription_data = {
        "medicines": [],
        "doctor_name": None,
        "date": None,
        "diagnosis": None
    }

    for document in result.documents:
        # Extract medicines
        if "medicines" in document.fields:
            for medicine in document.fields["medicines"].value:
                prescription_data["medicines"].append({
                    "name": medicine.value.get("name", {}).value,
                    "dosage": medicine.value.get("dosage", {}).value,
                    "frequency": medicine.value.get("frequency", {}).value,
                    "confidence": medicine.confidence
                })

        # Extract other fields
        if "doctor_name" in document.fields:
            prescription_data["doctor_name"] = document.fields["doctor_name"].value

        if "date" in document.fields:
            prescription_data["date"] = document.fields["date"].value

    return prescription_data

# Usage
result = extract_prescription("https://storage.blob.core.windows.net/prescriptions/rx001.jpg")
print(result)
```

Cost:

- Read API: $1.50 per 1,000 pages
- Custom Model Training: $40 per training hour
- Predictions: $10 per 1,000 pages with custom model
- Estimated: $10-20/day for 100 documents

4. Azure AI Search (RAG System - Vector Database)

Purpose: Store and search medical knowledge base using vector embeddings

Configuration:

```
Service: Azure AI Search (Cognitive Search)
Tier: Standard S1
Region: Central India
Replicas: 2 (high availability)
Partitions: 1
Storage: 25 GB

Index Settings:
  - Vectorization: Enabled
  - Vector Algorithm: HNSW (Hierarchical Navigable Small World)
  - Similarity Metric: Cosine
  - Vector Dimensions: 1536 (for text-embedding-ada-002)

Indexes Created:
  1. medical-knowledge-index (WHO, ICMR guidelines)
  2. drug-database-index (CDSCO drug info)
  3. user-documents-index (personal prescriptions)
```

Index Schema:

```
{
  "name": "medical-knowledge-index",
  "fields": [
    {
      "name": "id",
      "type": "Edm.String",
      "key": true,
      "searchable": false
    },
    {
      "name": "content",
      "type": "Edm.String",
      "searchable": true,
      "analyzer": "en.microsoft"
    },
    {
      "name": "content_vector",
      "type": "Collection(Edm.Single)",
      "searchable": true,
      "dimensions": 1536,
      "vectorSearchProfile": "medical-vector-profile"
    },
    {
      "name": "source",
      "type": "Edm.String",
      "filterable": true
    },
    {
      "name": "category",
      "type": "Edm.String",
      "filterable": true,
      "facetable": true
    },
    {
      "name": "last_updated",
      "type": "Edm.DateTimeOffset",
      "filterable": true,
      "sortable": true
    }
  ]
}
```

How We Use It:

```
# Example: RAG Implementation with Azure AI Search

from azure.search.documents import SearchClient
from azure.search.documents.models import VectorizedQuery
from openai import AzureOpenAI

# Initialize clients
search_client = SearchClient(
    endpoint="https://<search-service>.search.windows.net",
    index_name="medical-knowledge-index",
    credential=AzureKeyCredential("<key>")
)

openai_client = AzureOpenAI(...)

def get_embedding(text):
    """Convert text to vector embedding"""
    response = openai_client.embeddings.create(
        model="text-embedding-ada-002",
        input=text
    )
    return response.data[0].embedding

def retrieve_relevant_context(query, top_k=5):
    """
    RAG Retrieval: Find relevant medical information
    """

    # Convert query to vector
    query_vector = get_embedding(query)

    # Vector search in Azure AI Search
    vector_query = VectorizedQuery(
        vector=query_vector,
        k_nearest_neighbors=top_k,
        fields="content_vector"
    )

    results = search_client.search(
        search_text=query,  # Hybrid search (vector + keyword)
        vector_queries=[vector_query],
        select=["content", "source", "category"],
        top=top_k
    )

    # Format results
    context_documents = []
    for result in results:
        context_documents.append({
            "content": result["content"],
            "source": result["source"],
            "score": result["@search.score"]
        })

    return context_documents

# Usage in Medical Q&A Agent
def answer_medical_question(question):
    # Step 1: Retrieve relevant context
    context = retrieve_relevant_context(question, top_k=5)

    # Step 2: Format context for LLM
    context_text = "\n\n".join([
        f"Source: {doc['source']}\n{doc['content']}"
        for doc in context
    ])

    # Step 3: Generate answer with context
    response = openai_client.chat.completions.create(
        model="gpt-4",
        messages=[
            {"role": "system", "content": "You are a medical information assistant. Use the provided context to answer accurately."},
            {"role": "user", "content": f"Context:\n{context_text}\n\nQuestion: {question}"}
        ]
    )

    return response.choices[0].message.content, context
```

Indexing Pipeline:

```
# Batch indexing medical documents

import PyPDF2
import os

def index_medical_documents(folder_path):
    """
    Index all medical PDFs into Azure AI Search
    """

    documents = []

    for filename in os.listdir(folder_path):
        if filename.endswith('.pdf'):
            # Extract text from PDF
            with open(os.path.join(folder_path, filename), 'rb') as file:
                pdf_reader = PyPDF2.PdfReader(file)
                text = ""
                for page in pdf_reader.pages:
                    text += page.extract_text()

            # Chunk text (every 500 words)
            chunks = chunk_text(text, chunk_size=500)

            # Create embeddings and documents
            for i, chunk in enumerate(chunks):
                embedding = get_embedding(chunk)

                documents.append({
                    "id": f"{filename}_{i}",
                    "content": chunk,
                    "content_vector": embedding,
                    "source": filename,
                    "category": "medical_guideline",
                    "last_updated": datetime.now().isoformat()
                })

    # Batch upload to Azure AI Search
    search_client.upload_documents(documents=documents)
    print(f"Indexed {len(documents)} document chunks")

# Run once during setup
index_medical_documents("/path/to/WHO_ICMR_guidelines/")
```

Cost:

- S1 Tier: $250/month (fixed)
- Embedding API: $0.10 per 1M tokens
- Total: ~$300/month

5. Azure Storage Services

5A. Azure Blob Storage (Document Storage)

Purpose: Store uploaded prescription images and medical documents

```
Service: Azure Blob Storage
Account Type: StorageV2 (general purpose v2)
Replication: LRS (Locally Redundant Storage)
Region: Central India
Performance Tier: Standard
Access Tier: Hot (for frequently accessed docs)

Containers:
  - prescription-uploads (user documents)
  - extracted-data (JSON results)
  - medical-images (X-rays, scans)

Security:
  - Private access only
  - Shared Access Signatures (SAS) for temporary access
  - Encryption at rest (Microsoft-managed keys)
  - Soft delete enabled (7-day retention)
```

How We Use It:

```
from azure.storage.blob import BlobServiceClient, ContainerClient

# Initialize
connection_string = "<from-key-vault>"
blob_service_client = BlobServiceClient.from_connection_string(connection_string)

def upload_prescription(user_id, file_bytes, filename):
    """Upload user's prescription to blob storage"""

    container_name = "prescription-uploads"
    blob_name = f"{user_id}/{datetime.now().strftime('%Y%m%d_%H%M%S')}_{filename}"

    # Get blob client
    blob_client = blob_service_client.get_blob_client(
        container=container_name,
        blob=blob_name
    )

    # Upload with metadata
    blob_client.upload_blob(
        file_bytes,
        metadata={
            "user_id": user_id,
            "upload_date": datetime.now().isoformat(),
            "content_type": "image/jpeg"
        },
        overwrite=False
    )

    return blob_client.url

def get_prescription_url(blob_name, expiry_hours=1):
    """Generate temporary SAS URL for secure access"""

    from azure.storage.blob import generate_blob_sas, BlobSasPermissions
    from datetime import timedelta

    sas_token = generate_blob_sas(
        account_name="<storage-account>",
        container_name="prescription-uploads",
        blob_name=blob_name,
        account_key="<key>",
        permission=BlobSasPermissions(read=True),
        expiry=datetime.utcnow() + timedelta(hours=expiry_hours)
    )

    return f"https://<storage-account>.blob.core.windows.net/prescription-uploads/{blob_name}?{sas_token}"
```

Cost: ~$20-50/month for 1TB storage

5B. Azure SQL Database (Structured Data)

Purpose: Store user profiles, prescriptions, drug database

```
Service: Azure SQL Database
Tier: Standard S2 (50 DTUs)
Region: Central India
Max Size: 250 GB
Backup: Automated daily, 7-day retention
Geo-Replication: Disabled (MVP), enable for prod

Databases:
  - medicalchatbot_users
  - medicalchatbot_drugs
```

Schema:

```sql
-- Users table
CREATE TABLE Users (
    user_id UNIQUEIDENTIFIER PRIMARY KEY DEFAULT NEWID(),
    email NVARCHAR(255) UNIQUE NOT NULL,
    phone NVARCHAR(15),
    name NVARCHAR(100),
    date_of_birth DATE,
    created_at DATETIME DEFAULT GETDATE(),
    last_login DATETIME,
    language_preference NVARCHAR(10) DEFAULT 'en'
);

-- Prescriptions table
CREATE TABLE Prescriptions (
    prescription_id UNIQUEIDENTIFIER PRIMARY KEY DEFAULT NEWID(),
    user_id UNIQUEIDENTIFIER FOREIGN KEY REFERENCES Users(user_id),
    document_blob_url NVARCHAR(500),
    upload_date DATETIME DEFAULT GETDATE(),
    doctor_name NVARCHAR(100),
    prescription_date DATE,
    diagnosis NVARCHAR(500),
    extracted_data NVARCHAR(MAX),  -- JSON
    ocr_confidence DECIMAL(3,2)
);

-- Medicines table (from prescription)
CREATE TABLE PrescriptionMedicines (
    medicine_id UNIQUEIDENTIFIER PRIMARY KEY DEFAULT NEWID(),
    prescription_id UNIQUEIDENTIFIER FOREIGN KEY REFERENCES Prescriptions(prescription_id),
    medicine_name NVARCHAR(200),
    generic_name NVARCHAR(200),
    dosage NVARCHAR(50),
    frequency NVARCHAR(50),
    duration NVARCHAR(50),
    instructions NVARCHAR(500)
);

-- Drug Information Database
CREATE TABLE DrugDatabase (
    drug_id INT PRIMARY KEY IDENTITY(1,1),
    generic_name NVARCHAR(200) NOT NULL,
    brand_names NVARCHAR(MAX),  -- JSON array
    category NVARCHAR(100),
    uses NVARCHAR(MAX),
    dosage_adult NVARCHAR(200),
    dosage_pediatric NVARCHAR(200),
    side_effects_common NVARCHAR(MAX),  -- JSON
    side_effects_serious NVARCHAR(MAX),  -- JSON
    drug_interactions NVARCHAR(MAX),  -- JSON
    food_interactions NVARCHAR(MAX),
    contraindications NVARCHAR(MAX),
    pregnancy_category NVARCHAR(10),
    source NVARCHAR(50) DEFAULT 'CDSCO',
    last_updated DATETIME DEFAULT GETDATE()
);

-- Indexes for performance
CREATE INDEX IX_User_Email ON Users(email);
CREATE INDEX IX_Prescription_UserId ON Prescriptions(user_id);
CREATE INDEX IX_Drug_GenericName ON DrugDatabase(generic_name);
CREATE FULLTEXT INDEX ON DrugDatabase(generic_name, brand_names);
```

How We Use It:

```
import pyodbc

# Connection
conn_str = (
    "Driver={ODBC Driver 18 for SQL Server};"
    "Server=tcp:<server>.database.windows.net,1433;"
    "Database=medicalchatbot_users;"
    "Uid=<username>;"
    "Pwd=<password-from-keyvault>;"
    "Encrypt=yes;"
    "TrustServerCertificate=no;"
)

def save_prescription(user_id, extracted_data, blob_url):
    """Save extracted prescription to database"""

    conn = pyodbc.connect(conn_str)
    cursor = conn.cursor()

    # Insert prescription
    cursor.execute("""
        INSERT INTO Prescriptions
        (user_id, document_blob_url, extracted_data, doctor_name, prescription_date, ocr_confidence)
        VALUES (?, ?, ?, ?, ?, ?)
    """, (
        user_id,
        blob_url,
        json.dumps(extracted_data),
        extracted_data.get('doctor_name'),
        extracted_data.get('date'),
        extracted_data.get('confidence', 0.0)
    ))

    prescription_id = cursor.execute("SELECT @@IDENTITY").fetchone()[0]

    # Insert medicines
    for medicine in extracted_data.get('medicines', []):
        cursor.execute("""
            INSERT INTO PrescriptionMedicines
            (prescription_id, medicine_name, dosage, frequency, duration)
            VALUES (?, ?, ?, ?, ?)
        """, (
            prescription_id,
            medicine['name'],
            medicine['dosage'],
            medicine['frequency'],
            medicine.get('duration')
        ))

    conn.commit()
    conn.close()

    return prescription_id

def get_drug_info(medicine_name):
    """Query drug database"""

    conn = pyodbc.connect(conn_str)
    cursor = conn.cursor()

    cursor.execute("""
        SELECT * FROM DrugDatabase
        WHERE generic_name LIKE ? OR brand_names LIKE ?
    """, (f"%{medicine_name}%", f"%{medicine_name}%"))
    result = cursor.fetchone()
    conn.close()

    return result
```

Cost: ~$150-200/month (S2 tier)

5C. Azure Cosmos DB (Chat History - NoSQL)

Purpose: Store chat conversations with low-latency access

```
Service: Azure Cosmos DB
API: NoSQL (Core SQL API)
Consistency: Session (default)
Region: Central India
Throughput: Autoscale (400-4000 RU/s)
Indexing: Automatic

Containers:
  - conversations (partition key: /user_id)
  - messages (partition key: /conversation_id)
```

Data Model:

```
// Conversation document
{
  "id": "conv_123abc",
  "user_id": "user_456",
  "title": "Questions about Metformin",
  "created_at": "2025-10-14T10:30:00Z",
  "last_message_at": "2025-10-14T10:45:00Z",
  "message_count": 8,
  "status": "active"
}

// Message document
{
  "id": "msg_789def",
  "conversation_id": "conv_123abc",
  "role": "user",  // or "assistant"
  "content": "What are the side effects of Metformin?",
  "timestamp": "2025-10-14T10:30:15Z",
  "metadata": {
    "tokens": 156,
    "latency_ms": 1240,
    "agent_used": "DrugInfoAgent",
    "sources": ["CDSCO_DB", "WHO_Guidelines"]
  }
}
```

How We Use It:

```
from azure.cosmos import CosmosClient, PartitionKey

# Initialize
cosmos_client = CosmosClient(
    url="https://<account>.documents.azure.com:443/",
    credential="<key-from-keyvault>"
)
database = cosmos_client.get_database_client("medicalchatbot")
messages_container = database.get_container_client("messages")

def save_chat_message(conversation_id, user_id, role, content, metadata=None):
    """Save a chat message to Cosmos DB"""

    message = {
        "id": str(uuid.uuid4()),
        "conversation_id": conversation_id,
        "user_id": user_id,
        "role": role,
        "content": content,
        "timestamp": datetime.utcnow().isoformat(),
        "metadata": metadata or {}
    }

    messages_container.create_item(body=message)
    return message

def get_conversation_history(conversation_id, limit=50):
    """Retrieve chat history for context"""

    query = f"""
        SELECT * FROM messages m
        WHERE m.conversation_id = @conversation_id
        ORDER BY m.timestamp ASC
        OFFSET 0 LIMIT {limit}
    """

    items = list(messages_container.query_items(
        query=query,
        parameters=[{"name": "@conversation_id", "value": conversation_id}],
        enable_cross_partition_query=False
    ))

    return items
```

Cost: ~$25-50/month (autoscale 400-4000 RU/s)

6. Azure App Service (Web API Hosting)

Purpose: Host the main backend API

```
Service: Azure App Service (Web App)
OS: Linux
Runtime: Python 3.11 or Node.js 20
Tier: Standard S1 (1 core, 1.75 GB RAM)
Instances: 2 (auto-scale to 5 based on CPU >70%)
Region: Central India

Features:
  - Always On: Enabled
  - HTTPS Only: Yes
  - Application Insights: Integrated
  - Deployment Slots: staging, production
```

API Structure:

```
/api/v1/
├── /auth
│   ├── POST /signup
│   ├── POST /login
│   └── POST /logout
├── /documents
│   ├── POST /upload
│   ├── GET /{document_id}
│   └── DELETE /{document_id}
├── /chat
│   ├── POST /message
│   ├── GET /history/{conversation_id}
│   └── WS /stream
├── /medicines
│   ├── GET /{medicine_name}
│   └── GET /search
└── /profile
    ├── GET /
    └── PUT /
```

Deployment:

```
# app.yaml for Python Flask/FastAPI

runtime: python311
env: standard
instance_class: F2

automatic_scaling:
  target_cpu_utilization: 0.65
  min_instances: 2
  max_instances: 5

env_variables:
  AZURE_OPENAI_ENDPOINT: "https://..."
  AZURE_STORAGE_CONNECTION: "<from-keyvault>"
  SQL_CONNECTION_STRING: "<from-keyvault>"
```

Cost: ~$75/month (S1 tier × 2 instances)

---

7. Azure Functions (Background Tasks)

Purpose: Serverless functions for async processing

```
Service: Azure Functions
Plan: Consumption (serverless)
Runtime: Python 3.11
Region: Central India

Functions:
  - process-document-async (triggered by blob upload)
  - send-notifications (timer trigger - daily)
  - cleanup-old-data (timer trigger - weekly)
  - generate-reports (HTTP trigger)
```

Example Function:

```
# Function: process-document-async
# Triggered when prescription uploaded to blob storage

import azure.functions as func
import logging

app = func.FunctionApp()

@app.blob_trigger(
    arg_name="myblob",
    path="prescription-uploads/{user_id}/{filename}",
    connection="AzureWebJobsStorage"
)
def process_prescription_trigger(myblob: func.InputStream):
    """
    Automatically process prescription when uploaded
    """
    logging.info(f"Processing blob: {myblob.name}")

    # Get blob bytes
    file_bytes = myblob.read()

    # Call Document Intelligence
    extracted_data = extract_prescription(file_bytes)

    # Save to SQL database
    save_prescription(user_id, extracted_data, myblob.uri)

    # Send notification to user
    send_notification(user_id, "Prescription processed successfully!")

    logging.info("Processing complete")
```

Cost: ~$10-20/month (consumption plan, first 1M executions free)

---

8. Azure Key Vault (Secrets Management)

Purpose: Securely store API keys, connection strings, passwords

```
Service: Azure Key Vault
Tier: Standard
Region: Central India
Purge Protection: Enabled
Soft Delete: Enabled (90 days)
Access Policy: RBAC-based

Secrets Stored:
  - openai-api-key
  - document-intelligence-key
  - storage-connection-string
  - sql-connection-string
  - cosmos-connection-string
  - jwt-secret-key
  - sendgrid-api-key
```

How We Use It:

```
from azure.identity import DefaultAzureCredential
from azure.keyvault.secrets import SecretClient

# Initialize
credential = DefaultAzureCredential()
vault_url = "https://<keyvault-name>.vault.azure.net/"
secret_client = SecretClient(vault_url=vault_url, credential=credential)

def get_secret(secret_name):
    """Retrieve secret from Key Vault"""
    try:
        secret = secret_client.get_secret(secret_name)
        return secret.value
    except Exception as e:
        logging.error(f"Failed to retrieve secret {secret_name}: {e}")
        raise

# Usage in application
openai_key = get_secret("openai-api-key")
sql_connection = get_secret("sql-connection-string")
```

Security Best Practices:

- Use Managed Identity for Azure services (no keys needed)
- Rotate secrets every 90 days
- Enable audit logging
- Set access policies per service

Cost: $0.03 per 10,000 operations ($5/month)

---

9. Azure API Management (API Gateway)

Purpose: Central API gateway for rate limiting, authentication, monitoring

```
Service: Azure API Management
Tier: Developer (for MVP), Standard (for production)
Region: Central India
Virtual Network: Integrated

Features:
  - Rate Limiting: 100 requests/minute per user
  - API Key Authentication
  - JWT Token Validation (Azure AD B2C)
  - Request/Response Logging
  - API Versioning (/api/v1/, /api/v2/)
  - Caching (Redis)
```

API Policy Configuration:

```
<!-- Rate Limiting Policy -->
<policies>
    <inbound>
        <!-- Validate JWT token -->
        <validate-jwt header-name="Authorization" failed-validation-httpcode="401">
            <openid-config url="https://<tenant>.b2clogin.com/<tenant>.onmicrosoft.com/v2.0/.well-known/openid-configuration" />
        </validate-jwt>

        <!-- Rate limit per user -->
        <rate-limit-by-key calls="100" renewal-period="60"
                           counter-key="@(context.Request.Headers.GetValueOrDefault("Authorization",""))" />

        <!-- Set backend URL -->
        <set-backend-service base-url="https://<app-service>.azurewebsites.net" />
    </inbound>

    <backend>
        <forward-request />
    </backend>

    <outbound>
        <!-- Add CORS headers -->
        <cors>
            <allowed-origins>
                <origin>https://yourdomain.com</origin>
            </allowed-origins>
            <allowed-methods>
                <method>GET</method>
                <method>POST</method>
            </allowed-methods>
        </cors>
    </outbound>
</policies>
```

Cost: Developer tier ~$50/month, Standard ~$600/month

---

10. Azure AD B2C (User Authentication)

Purpose: User sign-up, sign-in, password management

```
Service: Azure Active Directory B2C
Tenant: medicalchatbot.onmicrosoft.com
Region: Global
User Flows:
  - Sign up and sign in (Email + Password)
  - Sign in (Phone + OTP)
  - Password reset
  - Profile editing

Authentication Methods:
  - Email + Password
  - Phone + OTP (via SMS)
  - Social logins: Google, Facebook (Phase 2)

Token Configuration:
  - Access Token: JWT, 1 hour expiry
  - Refresh Token: 90 days expiry
  - Claims: user_id, email, name, phone
```

Integration Example:

```
# Backend: Validate JWT token from Azure AD B2C

from jose import jwt, JWTError
import requests

def verify_token(token):
    """Verify JWT token from Azure AD B2C"""

    # Get public keys from Azure AD B2C
    jwks_url = "https://<tenant>.b2clogin.com/<tenant>.onmicrosoft.com/discovery/v2.0/keys?p=B2C_1_signupsignin"
    jwks = requests.get(jwks_url).json()

    try:
        # Decode and verify token
        payload = jwt.decode(
            token,
            jwks,
            algorithms=["RS256"],
            audience="<your-app-id>",
            issuer="https://<tenant>.b2clogin.com/<tenant-id>/v2.0/"
        )
        return payload
    except JWTError:
        raise Exception("Invalid token")

# Usage in API endpoint
from fastapi import Depends, HTTPException
from fastapi.security import HTTPBearer

security = HTTPBearer()

def get_current_user(token: str = Depends(security)):
    try:
        payload = verify_token(token.credentials)
        return payload
    except:
        raise HTTPException(status_code=401, detail="Invalid authentication")

@app.get("/api/v1/profile")
def get_profile(user = Depends(get_current_user)):
    user_id = user["sub"]
    # Fetch user profile from database
    return {"user_id": user_id, "email": user["email"]}
```

Cost: First 50,000 users free, then $0.00325 per user/month

11. Azure Application Insights (Monitoring)

Purpose: Real-time monitoring, logging, performance tracking

```
Service: Application Insights
Type: Workspace-based
Region: Central India
Sampling: Adaptive (to reduce costs)
Retention: 90 days

Metrics Tracked:
  - Request rate, duration, failures
  - Dependency calls (SQL, Cosmos, APIs)
  - Exceptions and errors
  - Custom events (document_uploaded, ocr_completed)
  - User analytics (DAU, MAU, retention)
```

Custom Telemetry:

```
from opencensus.ext.azure.log_exporter import AzureLogHandler
from opencensus.ext.azure import metrics_exporter
import logging

# Configure logging
logger = logging.getLogger(__name__)
logger.addHandler(AzureLogHandler(
    connection_string='InstrumentationKey=<key-from-keyvault>'
))

# Track custom events
def track_document_upload(user_id, document_type, ocr_confidence):
    """Log custom event"""
    logger.info(
        "Document uploaded",
        extra={
            'custom_dimensions': {
                'user_id': user_id,
                'document_type': document_type,
                'ocr_confidence': ocr_confidence,
                'event_name': 'document_uploaded'
            }
        }
    )

# Track performance
from opencensus.trace import tracer as tracer_module

tracer = tracer_module.Tracer()

def process_document_with_tracing(document_id):
    with tracer.span(name='process_document') as span:
        span.add_attribute('document_id', document_id)

        # OCR step
        with tracer.span(name='ocr_extraction'):
            result = extract_prescription(document_id)

        # Save step
        with tracer.span(name='save_to_database'):
            save_prescription(result)

        return result
```

Alerts Configuration:

```
Alerts:
  - Name: High Error Rate
    Condition: Exceptions > 10 in 5 minutes
    Action: Email + SMS to on-call engineer

  - Name: Slow Response Time
    Condition: Avg response time > 5 seconds for 10 minutes
    Action: Email to team

  - Name: Low OCR Confidence
    Condition: OCR confidence < 75% for 20 documents
    Action: Email to medical team for review

  - Name: High AI Costs
    Condition: OpenAI costs > $100 in 1 hour
    Action: Email to product manager
```

Cost: First 5GB/month free, then $2.30/GB

12. Azure Redis Cache (Caching Layer)

Purpose: Cache frequent queries, reduce AI API calls, improve response time

```
Service: Azure Cache for Redis
Tier: Basic C1 (1 GB cache)
Region: Central India
Persistence: Disabled (MVP)
Clustering: Disabled (MVP)

Cache Strategy:
  - Drug information (TTL: 7 days)
  - Common medical queries (TTL: 24 hours)
  - User session data (TTL: 1 hour)
  - RAG search results (TTL: 6 hours)
```

How We Use It:

```
import redis
import json
import hashlib

# Initialize Redis
redis_client = redis.Redis(
    host='<cache-name>.redis.cache.windows.net',
    port=6380,
    password=get_secret('redis-password'),
    ssl=True
)

def get_cached_response(query):
    """Check if query response is cached"""
    cache_key = f"query:{hashlib.md5(query.encode()).hexdigest()}"

    cached = redis_client.get(cache_key)
    if cached:
        return json.loads(cached)
    return None

def cache_response(query, response, ttl_seconds=3600):
    """Cache query response"""
    cache_key = f"query:{hashlib.md5(query.encode()).hexdigest()}"
    redis_client.setex(
        cache_key,
        ttl_seconds,
        json.dumps(response)
    )

# Usage in chat endpoint
@app.post("/api/v1/chat/message")
def chat_message(query: str):
    # Check cache first
    cached = get_cached_response(query)
    if cached:
        return {"response": cached, "from_cache": True}

    # Generate new response
    response = generate_ai_response(query)

    # Cache for future
    cache_response(query, response, ttl_seconds=3600)

    return {"response": response, "from_cache": False}

# Cache drug information
def get_drug_info_cached(drug_name):
    cache_key = f"drug:{drug_name.lower()}"

    cached = redis_client.get(cache_key)
    if cached:
        return json.loads(cached)

    # Fetch from database
    drug_info = get_drug_info_from_db(drug_name)

    # Cache for 7 days
    redis_client.setex(cache_key, 7*24*3600, json.dumps(drug_info))

    return drug_info
```

Cost: ~$17/month (Basic C1)

Architecture Diagram

```
┌─────────────────────────────────────────────────────────────────────┐
│                        USER LAYER                                   │
│  Web Browser / Mobile App / Progressive Web App                     │
└─────────────────────────┬───────────────────────────────────────────┘
                          │ HTTPS
                          ▼
┌─────────────────────────────────────────────────────────────────────┐
│                   AZURE CDN (Content Delivery)                      │
│               Static assets, images, frontend                       │
└─────────────────────────┬───────────────────────────────────────────┘
                          │
                          ▼
┌─────────────────────────────────────────────────────────────────────┐
│             AZURE APPLICATION GATEWAY + WAF                         │
│             Load Balancer + DDoS Protection                         │
└─────────────────────────┬───────────────────────────────────────────┘
                          │
                          ▼
┌─────────────────────────────────────────────────────────────────────┐
│                  AZURE API MANAGEMENT                               │
│  • Rate Limiting  • Authentication  • API Gateway                   │
└─────────────────────────┬───────────────────────────────────────────┘
                          │
       ┌──────────────────┼──────────────────┐
       │                  │                  │
       ▼                  ▼                  ▼
┌──────────────┐  ┌──────────────┐  ┌──────────────┐
│  Azure AD    │  │   AZURE      │  │   AZURE      │
│  B2C         │  │   APP        │  │  FUNCTIONS   │
│              │  │   SERVICE    │  │              │
│ Auth/Login   │  │              │  │ Background   │
└──────────────┘  │ • Python API │  │ Processing   │
                 │ • REST/WS    │  └──────────────┘
                 └──────────────┘
                        │
       ┌────────────────┼────────────────┐
       │                │                │
       ▼                ▼                ▼
┌──────────────┐  ┌──────────────┐  ┌──────────────┐
│   AZURE      │  │   AZURE      │  │   AZURE      │
│  OPENAI      │  │ AI FOUNDRY   │  │  DOCUMENT    │
│              │  │              │  │ INTELLIGENCE │
│ GPT-4        │  │ Med42-Llama3 │  │              │
│ Orchestrator │  │ BioGPT       │  │ OCR Engine   │
│              │  │ BiomedCLIP   │  │              │
└──────────────┘  └──────────────┘  └──────────────┘
       │                │                │
       └────────────────┼────────────────┘
                        │
                        ▼
       ┌────────────────────────────────┐
       │    AZURE AI SEARCH             │
       │    (Vector Database for RAG)   │
       │    • Medical Knowledge         │
       │    • Drug Database             │
       └────────────────────────────────┘
                        │
       ┌────────────────┼────────────────┐
       │                │                │
       ▼                ▼                ▼
┌──────────────┐  ┌──────────────┐  ┌──────────────┐
│   AZURE      │  │   AZURE SQL  │  │   AZURE      │
│   BLOB       │  │   DATABASE   │  │  COSMOS DB   │
│   STORAGE    │  │              │  │              │
│              │  │ • Users      │  │ • Chat       │
│ Documents      │  │ • Drugs       │  │   History     │
│ Images         │  │ • Rx Data     │  │               │
└────────────────┘  └───────────────┘  └───────────────┘
        │                  │                  │
        └──────────────────┼──────────────────┘
                           │
                           ▼
        ┌──────────────────────────────────┐
        │   AZURE REDIS CACHE              │
        │   (Fast caching layer)           │
        └──────────────────────────────────┘

┌───────────────────────────────────────────────────────────────────────┐
│                    SECURITY & MANAGEMENT LAYER                        │
├───────────────────────────────────────────────────────────────────────┤
│ • Azure Key Vault (Secrets)                                           │
│ • Azure Monitor + Application Insights (Monitoring)                   │
│ • Azure Log Analytics (Centralized Logging)                           │
│ • Azure Security Center (Threat Protection)                           │
└───────────────────────────────────────────────────────────────────────┘
```

Final Azure Services - Consolidated Guide

Medical Document Chatbot - Complete Azure Stack

Project: AI Medical Document Chatbot

Region: India (Central India / South India)

Date: October 14, 2025

Table of Contents

Detailed Service Configuration (1–12)

Note: Items 1–8 are sourced from `azure_sercvice1.md`; items 9–12 are sourced from `azure_Service2.md`. This file consolidates all services into one place for planning and review.

1) Azure OpenAI Service (Orchestrator Agent)

- Purpose: Central LLM orchestrator to route queries to specialized agents.
- Key points: GPT-4 deployment, low temperature for routing, TPM/RPM limits.
- Source: `azure_sercvice1.md` (Section: 1. Azure OpenAI Service)

2) Azure AI Foundry (Specialized AI Agents)

- Purpose: Host Med42-Llama3, BioGPT, BiomedCLIP endpoints.
- Key points: Managed Online Endpoints, autoscaling, GPU SKUs.
- Source: `azure_sercvice1.md` (Section: 2. Azure AI Foundry)

3) Azure Document Intelligence (Document Agent)

- Purpose: OCR and structured data extraction for prescriptions.
- Key points: Custom model for Indian prescriptions, layout/kv/table extraction.
- Source: `azure_sercvice1.md` (Section: 3. Azure Document Intelligence)

4) Azure AI Search (RAG System - Vector DB)

- Purpose: Vector search over medical knowledge and drug data.
- Key points: HNSW, cosine similarity, embeddings, hybrid search.
- Source: `azure_sercvice1.md` (Section: 4. Azure AI Search)

5) Azure Storage Services (Blob Storage)

- Purpose: Store uploads, extracted JSON, and medical images.
- Key points: StorageV2, LRS, private containers, SAS for access.
- Source: `azure_sercvice1.md` (Section: 5A. Azure Blob Storage)

6) Azure SQL Database (Structured Data)

- Purpose: Users, prescriptions, medicines, drug database.
- Key points: S2 tier, schema for Users/Prescriptions/DrugDatabase.
- Source: `azure_sercvice1.md` (Section: 5B. Azure SQL Database)

7) Azure Cosmos DB (Chat History - NoSQL)

- Purpose: Low-latency chat storage for conversations and messages.
- Key points: NoSQL API, autoscale throughput, partitioning.
- Source: `azure_sercvice1.md` (Section: 5C. Azure Cosmos DB)

8) Azure App Service (Web API) and Azure Functions (Background Tasks)

- Purpose: Host backend API and async processing.
- Key points: Python 3.11, Always On, blob-triggered processing, timers.
- Source: `azure_sercvice1.md` (Sections: 6. App Service, 7. Functions)

9) Azure API Management (API Gateway)

- Purpose: Central gateway for auth, rate limiting, logging, and versioning.
- Key points: Developer/Standard tiers, JWT validation, caching, CORS.
- Source: `azure_Service2.md` (Section: 9. Azure API Management)

10) Azure AD B2C (User Authentication)

- Purpose: Sign-up/sign-in, password reset, phone OTP, social logins.
- Key points: JWT tokens, refresh tokens, claims; backend JWT verification.
- Source: `azure_Service2.md` (Section: 10. Azure AD B2C)

11) Azure Application Insights (Monitoring)

- Purpose: Telemetry, tracing, custom events, performance monitoring.
- Key points: Workspace-based, adaptive sampling, alerts for errors/latency.
- Source: `azure_Service2.md` (Section: 11. App Insights)

12) Azure Redis Cache (Caching Layer)

- Purpose: Cache frequent queries, RAG results, and session data.
- Key points: Basic C1, SSL, TTL strategies, hash-based keys.
- Source: `azure_Service2.md` (Section: 12. Redis Cache)

Notes

- This file is a consolidated index of all 12 Detailed Service Configuration sections across both source documents, enabling a single place to track and communicate the full Azure stack.
- For implementation YAML, code snippets, and cost details, refer to the original sections indicated above.