Agentic AI Server Project Report: Integrated MCP Server and CrewAI

Prepared by Grok 3

June 6, 2025

1 Project Overview

The Agentic AI Server project aims to develop an offline, multimodal AI system for processing structured (CSV, JSON), unstructured (PDF, images, videos, audio), and semi-structured (JSON, XML) data. The system is deployed on an Ubuntu 24.04 server with an NVIDIA A100 GPU (40GB/80GB), leveraging Hadoop for storage, Kafka for streaming, PostGIS for geospatial data, and an integrated **MCP server and CrewAI** framework for agentic orchestration. The original roadmap specified **LLaMA 3.8B** as the central model with CrewAI orchestration. However, to meet requirements for direct multimodal processing and lightweight deployment, this report adopts **Qwen2-VL-2B** for native image, video, and text processing and **Whisper-Tiny** for audio transcription, integrated with an MCP server for standardized tool access and CrewAI for agentic workflows using ReAct and Tree of Thought (ToT) prompt techniques.

This report incorporates the provided pipeline architecture, updates the roadmap to integrate MCP server and CrewAI, and provides a detailed implementation plan, addressing deployment considerations, challenges, and integration points.

2 Data Pipeline Architecture

The pipeline processes diverse data types stored in Hadoop, streamed via Kafka, managed in PostGIS, and analyzed using an agentic AI system with Qwen2-VL-2B and Whisper-Tiny, orchestrated by an integrated MCP server and CrewAI. It is divided into Backend, Middleware, and Frontend layers, with detailed steps for each.

2.1 Backend Layer

The backend layer handles data storage, ingestion, streaming, preprocessing, and geospatial storage.

2.1.1 Step 1.1: Data Storage in Hadoop

Purpose: Store structured (CSV, JSON), unstructured (PDF, images, videos, audio), and semi-structured (JSON, XML) data in Hadoop Distributed File System (HDFS).

Details:

- *Structured Data*: CSV and JSON files are stored in columnar formats (Parquet, ORC) to optimize query performance and reduce storage overhead. For example, a CSV with sales data (columns: order_id, date, amount) is converted to Parquet using Apache Spark.
- *Unstructured Data*: PDFs, images (JPEG, PNG), videos (MP4, AVI), and audio (WAV, MP3) are stored as raw binary files in HDFS. Metadata (e.g., file name, size, creation date) is stored in Apache Hive tables.
- *Semi-Structured Data*: JSON and XML files are stored in HDFS with schema-on-read capabilities via Hive or Spark SQL.
- *Hadoop Ecosystem*: HDFS provides distributed storage, Hive manages metadata, and HBase supports real-time key-value access for semi-structured data.

Tools:

- HDFS: Distributed file storage.
- Apache Sqoop: Ingests structured data from relational databases (e.g., MySQL, Oracle).
- Apache Flume: Streams unstructured data (e.g., log files, media) into HDFS.
- Apache Hive: Manages metadata and enables SQL-like queries.
- Apache HBase: Handles semi-structured data for low-latency access.

Process:

1. *Structured Data Ingestion*:
   - Use Sqoop to import data from RDBMS (e.g., `sqoop import -connect jdbc:mysql://loca` `-table sales -target-dir /hdfs/sales`).
   - Convert CSV/JSON to Parquet using Spark (`spark.read.csv("hdfs://sales").write.pa`
   - Store in HDFS with partitioning (e.g., by date or region).
2. *Unstructured Data Ingestion*:
   - Configure Flume to stream media files to HDFS (e.g., `flume-ng agent -conf flume.conf -name media_agent`).
   - Store files in HDFS directories (e.g., `/hdfs/images/`, `/hdfs/videos/`).
   - Extract metadata (e.g., image resolution, video duration) using ExifTool and store in Hive (`CREATE TABLE media_metadata (file_path STRING, resolution STRING)`).
3. *Semi-Structured Data*:
   - Ingest JSON/XML via Flume or custom scripts.
   - Use Hive to define schemas (e.g., `CREATE EXTERNAL TABLE json_data (id INT, data MAP<STRING, STRING>) STORED AS JSONFILE LOCATION ’/hdfs/json’`).
4. *Metadata Management*:

- Create Hive tables for all data types (e.g., `CREATE TABLE files (path STRING, type STRING, metadata MAP<STRING, STRING>)`).
- Enable querying via HiveQL (e.g., `SELECT * FROM files WHERE type='pdf'`).

Challenges:

- Large-scale ingestion may cause bottlenecks; use Sqoops parallel tasks.
- Unstructured data requires significant storage; implement compression (e.g., Snappy).
- Ensure metadata consistency across Hive and HDFS.

Integration Point: Data stored in HDFS is accessible to Kafka for streaming and Spark for preprocessing.

2.1.2 Step 1.2: Data Streaming with Kafka

Purpose: Stream data from HDFS to downstream systems for real-time processing.

Details:

- Kafka serves as the messaging system for high-throughput, real-time data streams.
- Kafka Connect integrates with HDFS to extract data, with topics defined for each data type (e.g., `csv_data`, `pdf_data`, `geospatial_data`).
- Producers serialize structured data (Avro/JSON) and unstructured data (binary with metadata headers).
- Consumers (e.g., Spark Streaming, Flink) process streams for preprocessing or routing to PostGIS/AI.

Tools:

- Apache Kafka: Messaging system.
- Kafka Connect: Integrates HDFS with Kafka.
- Kafka Streams: Processes streams for transformations.

Process:

1. *Configure Kafka Connect*:
   - Use HDFS Connector (`connect-hdfs-sink.properties` with `topics=csv_data,pdf_data`)
   - Serialize structured data with Avro schemas (e.g., `{"type":"record","name":"Sales","fi`
   - Serialize unstructured data as binary with JSON metadata headers.
2. *Create Topics*:
   - Define topics for each data type (`kafka-topics.sh -create -topic csv_data -partitions 10 -replication-factor 3`).
3. *Producers*:
   - Write a producer to push HDFS data to Kafka (e.g., Python with `confluent_kafka`).
   - Example: `producer.produce('csv_data', key='order_id', value=avro_serialized_d`

4. *Consumers*:
    - Deploy consumers to read from topics (`spark.readStream.format("kafka").option("subs` `"csv_data")`).
    - Route data to PostGIS or preprocessing pipelines.

Challenges:

- Ensure schema compatibility using Schema Registry for Avro data.
- Handle backpressure with Kafkas partitioning and consumer groups.
- Monitor lag using Kafkas metrics to prevent bottlenecks.

Integration Point: Kafka streams data to PostGIS for geospatial storage and Spark for AI preprocessing.

2.1.3 Step 1.3: Geospatial Data Storage in PostGIS

Purpose: Store and query geospatial data (e.g., coordinates from JSON, image metadata) using PostGIS.

Details:

- PostGIS, an extension of PostgreSQL, supports geospatial data types (geometry, geography) and spatial queries (e.g., `ST_DWithin` for proximity).
- Geospatial data is extracted from Kafka topics (e.g., `geospatial_data`) and stored in PostGIS tables.
- Spatial indexes (GIST) optimize query performance.

Tools:

- PostgreSQL with PostGIS: Geospatial database.
- Kafka Consumer: Reads geospatial data from Kafka.

Process:

1. *Consumer Setup*:
    - Write a Kafka consumer (e.g., Python with `confluent_kafka`) to read from `geospatial_data`.
    - Parse JSON data (e.g., `{"lat": 40.7128, "lon": -74.0060}`).
2. *Data Storage*:
    - Create PostGIS tables (e.g., `CREATE TABLE locations (id SERIAL PRIMARY KEY, geom GEOMETRY(POINT, 4326))`).
    - Insert data using SQL (e.g., `INSERT INTO locations (geom) VALUES (ST_SetSRID(ST_Ma` `40.7128), 4326))`).
3. *Indexing*:
    - Create GIST indexes (`CREATE INDEX locations_geom_idx ON locations USING GIST (geom)`).

4. *Querying*:
    - Support spatial queries (e.g., `SELECT * FROM locations WHERE ST_DWithin(geom, ST_MakePoint(-74.0060, 40.7128)::geography, 10000)` for 10km radius).

Challenges:

- Parsing geospatial data from unstructured sources requires robust preprocessing.
- Large datasets may slow queries; use partitioning or sharding.
- Ensure SRID (e.g., 4326 for WGS84) consistency.

Integration Point: PostGIS data is queried by AI agents via MCP tools for geospatial analysis.

2.1.4 Step 1.4: Data Preprocessing for AI

Purpose: Transform and enrich data for AI processing.

Details:

- *Structured Data*: Clean (remove nulls, duplicates), normalize (scale numerical fields), and convert to tensors or embeddings.
- *Unstructured Data*: Extract text from PDFs (pdfplumber), process images/videos (OpenCV, FFmpeg), transcribe audio (Whisper-Tiny).
- *Semi-Structured Data*: Parse JSON/XML, extract fields, and convert to structured formats or embeddings.
- Processed data is stored in HDFS or cached in Redis for low-latency access.

Tools:

- Apache Spark: Distributed preprocessing.
- pdfplumber: Text extraction from PDFs.
- OpenCV/FFmpeg: Image and video processing.
- Whisper-Tiny: Audio transcription.
- SentenceTransformers: Text embeddings.

Process:

1. *Structured Data*:
    - Clean CSV/JSON with Spark (`spark.read.parquet("hdfs://sales_parquet").dropDupl`
    - Normalize numerical fields (e.g., MinMaxScaler in Spark ML).
    - Generate embeddings for categorical data (e.g., SentenceTransformers).
2. *Unstructured Data*:
    - Extract PDF text (`pdfplumber.open("hdfs://pdfs/doc.pdf").pages`).
    - Process images with OpenCV (e.g., resize, extract features).

    - Extract video frames with FFmpeg (`ffmpeg -i video.mp4 -vf fps=1 frame%d.jpg`).
    - Transcribe audio with Whisper-Tiny (`whisper transcribe audio.wav`).

3. *Semi-Structured Data*:
    - Parse JSON/XML with Spark (`spark.read.json("hdfs://json_data")`).
    - Extract fields and convert to embeddings.
4. *Storage*:
    - Store in HDFS (`spark.write.parquet("hdfs://processed_data")`).
    - Cache in Redis (`redis.set("key", tensor_data)`).

Challenges:

- Processing large unstructured data is computationally expensive; use distributed Spark clusters.
- Embedding generation requires GPU acceleration.
- Ensure data lineage for traceability.

Integration Point: Processed data is accessed by AI agents via HDFS or Redis, with MCP tools facilitating preprocessing.

2.2 Middleware Layer

The middleware layer orchestrates AI tasks, manages agentic workflows, and integrates backend data with frontend interfaces using an integrated MCP server and CrewAI framework.

2.2.1 Step 2.1: Agentic AI with Qwen2-VL-2B and Whisper-Tiny

Purpose: Use Qwen2-VL-2B as the central brain for multi-agent systems, handling image, video, and text processing, with Whisper-Tiny for audio transcription, orchestrated by MCP and CrewAI.

Details:

- *Model Choice*: The original roadmap used LLaMA 3.8B, but Qwen2-VL-2B is adopted for native image and video processing, and Whisper-Tiny for audio, due to their lightweight nature (4-6GB and 100MB GPU memory) and direct multimodal capabilities, fitting the A100 GPU.
- *MCP Server*: Exposes standardized tools for preprocessing (e.g., PDF extraction, audio transcription) and data access (e.g., PostGIS queries).
- *CrewAI*: Orchestrates multiple agents (e.g., data extraction, geospatial analysis) using ReAct (reason, act) and ToT (explore multiple reasoning paths) prompts.

Tools:

- Qwen2-VL-2B: Multimodal LLM for image, video, and text (Hugging Face).
- Whisper-Tiny: Audio transcription (Hugging Face).

- MCP: Tool exposure framework.
- CrewAI: Agent orchestration framework.

Process:

1. *Agent Definition*:
   - Define agents (e.g., DataExtractorAgent, GeospatialAgent) with roles and goals.
   - Example: DataExtractorAgent processes PDFs; GeospatialAgent queries PostGIS.
2. *Task Assignment*:
   - CrewAI assigns tasks based on data type (e.g., image tasks to Qwen2-VL-2B, audio to Whisper-Tiny).
   - ReAct prompts: "Reason about the PDF content, extract key points, act by summarizing."
   - ToT prompts: "Explore multiple geospatial query strategies, evaluate accuracy, select best."
3. *MCP Tool Integration*:
   - MCP tools preprocess data (e.g., extract text from PDFs, transcribe audio) and feed to Qwen2-VL-2B.
   - Example: `process_pdf` tool extracts text for CrewAI tasks.
4. *Execution*:
   - Agents call Qwen2-VL-2B for image/video/text tasks and Whisper-Tiny for audio.
   - CrewAI manages inter-agent communication and task dependencies.
5. *Fine-Tuning*:
   - Fine-tune Qwen2-VL-2B for domain-specific tasks (e.g., geospatial reasoning) using a small dataset.

Challenges:

- Qwen2-VL-2Bs reasoning is less robust than larger models; use ToT for enhanced reasoning.
- Prompt engineering must balance depth and response time.
- Agent coordination requires clear task definitions.

Integration Point: Agents interact with backend data stores (HDFS, PostGIS) via MCP tools and frontend APIs.

2.2.2 Step 2.2: Tools and Functions

Purpose: Provide agents with MCP tools for data processing and external system interactions.

Details:

- *Tools*: Include PDF extraction (pdfplumber), geospatial queries (PostGIS), embeddings (SentenceTransformers), image/video processing (OpenCV), and audio transcription (Whisper-Tiny).
- *MCP Server*: Exposes tools via a standardized protocol, accessible to CrewAI agents.
- *Functions*: Custom APIs for HDFS/PostGIS access, preprocessing scripts, and external service calls (if needed).

Tools:

- MCP: Tool exposure (e.g., `process_pdf`, `query_postgis`).
- pdfplumber: PDF text extraction.
- OpenCV: Image/video processing.
- Whisper-Tiny: Audio transcription.
- SentenceTransformers: Embeddings.

Process:

1. *MCP Tool Definition*:
   - Define tools in MCP (e.g., `@tool def process_pdf(file_path): ...`).
   - Tools handle preprocessing (e.g., PDF to text, audio transcription).
2. *Tool Integration*:
   - CrewAI agents access MCP tools via `MCPServerAdapter`.
   - Example: `process_pdf` extracts text for Qwen2-VL-2B.
3. *Function Execution*:
   - Tools fetch data from HDFS/PostGIS (e.g., `query_postgis("SELECT ST_AsText(geom) FROM locations")`).
   - Process results (e.g., generate embeddings).
4. *Result Feedback*:
   - Tool outputs are returned to Qwen2-VL-2B for reasoning or task completion.

Challenges:

- Tool reliability requires robust error handling.
- Ensure tools are compatible with Qwen2-VL-2Bs input formats.
- Scalability of tool execution in distributed environments.

Integration Point: MCP tools connect backend data stores to CrewAI agents and frontend APIs.

2.2.3 Step 2.3: Data Store Integration

Purpose: Provide agents with access to processed data, embeddings, and metadata.

Details:

- *Vector Database (Milvus)*: Stores embeddings for text, images, and audio.
- *Redis*: Caches frequently accessed data (e.g., embeddings, query results).
- *Hive/PostgreSQL*: Stores metadata for structured and geospatial data.

Tools:

- Milvus: Vector database.
- Redis: In-memory cache.
- PostgreSQL/Hive: Metadata storage.

Process:

1. *Embedding Storage*:
    - Generate embeddings with SentenceTransformers (`model.encode("text")`).
    - Store in Milvus (`milvus.insert(collection_name="embeddings", data=embeddings)`).
2. *Caching*:
    - Cache results in Redis (`redis.setex("query_result", 3600, result)`).
3. *Metadata Querying*:
    - Query Hive/PostgreSQL for metadata (`SELECT metadata FROM files WHERE type='image'`).
4. *Agent Access*:
    - Agents query Milvus for similar items (`milvus.search(collection_name="embeddings", query_embedding)`).
    - Fetch cached data from Redis.

Challenges:

- Embedding storage requires optimized indexing in Milvus.
- Redis cache eviction policies must balance memory and freshness.
- Ensure metadata consistency across Hive and PostGIS.

Integration Point: Data stores provide fast access to CrewAI agents via MCP tools.

2.3 Frontend Layer

The frontend layer enables user interaction, query submission, and visualization of AI outputs.

2.3.1 Step 3.1: User Interface

Purpose: Provide a web-based interface for users to interact with the AI system.

Details:

- Built using React with Tailwind CSS for a responsive UI.
- Features include a chat interface, data upload forms, and visualization panels.
- Supports geospatial visualization (e.g., interactive maps) and data tables.

Tools:

- React: Frontend framework.
- Tailwind CSS: Styling framework.
- Mapbox: Geospatial visualization.

Process:

1. *UI Components*:
    - Chat interface for queries (e.g., "Summarize this PDF").
    - File upload form for PDFs, CSVs, etc.
    - Visualization panel for maps and charts.
2. *Query Submission*:
    - Users input queries via text or file upload.
    - Queries sent to middleware via REST APIs (`fetch("/api/query", { method: "POST", body: query })`).
3. *Result Display*:
    - Render results as tables, charts, or Mapbox maps.
    - Example: Display locations on a map with markers.

Challenges:

- Ensure UI responsiveness for large datasets (e.g., lazy loading).
- Secure file uploads to prevent malicious content.
- Optimize Mapbox for large geospatial datasets.

Integration Point: UI communicates with middleware APIs to fetch AI results.

2.3.2 Step 3.2: Visualization and Reporting

Purpose: Generate visualizations and reports from AI outputs.

Details:

- Geospatial data visualized on interactive maps using Mapbox.
- Structured data displayed as charts (e.g., bar, line) using Chart.js.

- Reports generated as PDFs using wkhtmltopdf.

Tools:

- Mapbox: Geospatial visualization.
- Chart.js: Data visualization.
- wkhtmltopdf: PDF generation.

Process:

1. *Geospatial Visualization*:
   - Fetch PostGIS data via API (`fetch("/api/locations")`).
   - Render on Mapbox map (`mapboxgl.Map.addLayer({ type: "circle", source: locations })`).
2. *Data Visualization*:
   - Generate charts with Chart.js (e.g., `{ type: "bar", data: { labels: dates, datasets: [{ data: amounts, backgroundColor: "#3b82f6" }] }}`).
3. *Report Generation*:
   - Create HTML report with results and visualizations.
   - Convert to PDF (`wkhtmltopdf report.html report.pdf`).
   - Serve PDF for download (`<a href="/reports/report.pdf">Download</a>`).

Challenges:

- Large datasets may slow visualization; use pagination or sampling.
- Ensure cross-browser compatibility for Mapbox and Chart.js.
- PDF generation must handle complex layouts.

Integration Point: Visualizations pull data from middleware APIs and PostGIS.

3 Project Roadmap with MCP and CrewAI Integration

The project roadmap is updated to integrate the MCP server and CrewAI, replacing the original CrewAI-only approach. The MCP server provides standardized tools for preprocessing and data access, while CrewAI orchestrates agentic tasks.

1. **Requirement Analysis and Planning (Phase 1)**:
   - Define data formats: CSV, JSON, PDF, images, videos, audio, XML.
   - Identify hardware: NVIDIA A100 (40GB/80GB), Ubuntu 24.04.
   - Select models: Qwen2-VL-2B (images, video, text), Whisper-Tiny (audio).
   - Choose frameworks: Integrated MCP server and CrewAI.

    - *Integration*: Plan MCP tools for preprocessing (e.g., PDF extraction, audio transcription) and CrewAI for agentic workflows.

2. **Environment Setup (Phase 2)**:
    - Install NVIDIA drivers, CUDA, and dependencies.
    - Pre-download model weights and libraries for offline use.
    - *Integration*: Configure MCP server and CrewAI dependencies.
3. **Preprocessing Pipeline Development (Phase 3)**:
    - Develop MCP tools for data preprocessing (e.g., PDF text extraction, audio transcription).
    - *Integration*: Expose tools via MCP server for CrewAI access.
4. **Agentic Workflow Implementation (Phase 4)**:
    - Build CrewAI agents for task orchestration (e.g., summarization, geospatial analysis).
    - *Integration*: Use `MCPServerAdapter` to integrate MCP tools with CrewAI agents.
5. **API Development and Testing (Phase 5)**:
    - Expose the integrated pipeline via FastAPI.
    - Test with sample datasets across all formats.
    - *Integration*: Ensure API calls trigger MCP tools and CrewAI tasks.
6. **Optimization and Deployment (Phase 6)**:
    - Optimize inference with vLLM.
    - Deploy the offline server and validate performance.
    - *Integration*: Optimize MCP server and CrewAI for low latency and resource efficiency.

4 Implementation Plan

4.1 Environment Setup

- **Hardware**: NVIDIA A100 (40GB/80GB).
- **OS**: Ubuntu 24.04.
- **Python**: 3.11+ (required for CrewAI).
- **Models**:
    - Qwen2-VL-2B (`Qwen/Qwen2-VL-2B-Instruct`, 4-6GB GPU memory).
    - Whisper-Tiny (`openai/whisper-tiny`, 100MB).
- **Dependencies** (pre-downloaded for offline use):

  - torch (CUDA 12.1).
  - transformers, openai-whisper, pdfplumber, pandas, opencv-python, moviepy.
  - crewai, crewai-tools, mcp.
  - vllm (optional for optimized inference).
  - confluent_kafka, pyspark, psycopg2, sentence-transformers, milvus, redis.

Listing 1: Install NVIDIA Drivers and CUDA

```
sudo apt update
sudo apt install nvidia-driver-<version> nvidia-utils-<version>
   cuda
```

Listing 2: Pre-download Dependencies (on a connected machine)

```
pip download torch --index-url https://download.pytorch.org/whl/
   cu121 -d /path/to/offline-packages
pip download transformers openai-whisper pdfplumber pandas opencv
   -python moviepy crewai crewai-tools mcp vllm confluent_kafka
   pyspark psycopg2 sentence-transformers milvus redis -d /path/
   to/offline-packages
```

Listing 3: Install Dependencies Offline

```
pip install --no-index --find-links=/path/to/offline-packages
   torch transformers openai-whisper pdfplumber pandas opencv-
   python moviepy crewai crewai-tools mcp vllm confluent_kafka
   pyspark psycopg2 sentence-transformers milvus redis
```

- **Model Weights**: Pre-download Qwen2-VL-2B and Whisper-Tiny from Hugging Face, store in /path/to/models.

4.2 MCP Server Implementation

The MCP server exposes tools for preprocessing and data access, ensuring portability and reusability.

Listing 4: MCP Server Script (mcp$_s$*erver.py*)

```python
from mcp import tool, StdioServer
import pdfplumber
import pandas as pd
import whisper
import cv2
from PIL import Image
import psycopg2
import json
from sentence_transformers import SentenceTransformer
from transformers import Qwen2VLForConditionalGeneration,
   AutoProcessor

model = Qwen2VLForConditionalGeneration.from_pretrained("/path/to
   /models/Qwen2-VL-2B-Instruct", device_map="cuda")
```

```python
processor = AutoProcessor.from_pretrained("/path/to/models/Qwen2-VL-2B-Instruct")
whisper_model = whisper.load_model("tiny")
embedding_model = SentenceTransformer('all-MiniLM-L6-v2')

@tool
def process_pdf(file_path: str) -> str:
    """Extract text from a PDF file."""
    try:
        with pdfplumber.open(file_path) as pdf:
            return "".join(page.extract_text() for page in pdf.pages)
    except Exception as e:
        return f"Error␣extracting␣PDF:␣{str(e)}"

@tool
def process_csv(file_path: str) -> str:
    """Summarize CSV data."""
    try:
        df = pd.read_csv(file_path)
        return f"CSV␣with␣{len(df)}␣rows.␣Columns:␣{',␣'.join(df.columns)}."
    except Exception as e:
        return f"Error␣processing␣CSV:␣{str(e)}"

@tool
def process_json(file_path: str) -> str:
    """Parse JSON data."""
    try:
        with open(file_path) as f:
            data = json.load(f)
        return f"JSON␣data:␣{json.dumps(data,␣indent=2)}"
    except Exception as e:
        return f"Error␣processing␣JSON:␣{str(e)}"

@tool
def process_xml(file_path: str) -> str:
    """Parse XML data."""
    try:
        import xml.etree.ElementTree as ET
        tree = ET.parse(file_path)
        root = tree.getroot()
        return ET.tostring(root, encoding='unicode')
    except Exception as e:
        return f"Error␣processing␣XML:␣{str(e)}"

@tool
def process_audio(file_path: str) -> str:
    """Transcribe audio."""
    try:
        return whisper_model.transcribe(file_path)["text"]
    except Exception as e:
        return f"Error␣transcribing␣audio:␣{str(e)}"

@tool
def process_video(file_path: str) -> str:
    """Analyze a video frame."""
    try:
        cap = cv2.VideoCapture(file_path)
        ret, frame = cap.read()
        cap.release()
        if ret:
            cv2.imwrite("temp_frame.jpg", frame)
            inputs = processor(text="Describe␣this␣frame.",
               images=[Image.open("temp_frame.jpg")],
               return_tensors="pt").to("cuda")
            outputs = model.generate(**inputs)
            return processor.decode(outputs[0])
        return "Failed␣to␣process␣video."
    except Exception as e:
        return f"Error␣processing␣video:␣{str(e)}"

@tool
def query_postgis(query: str) -> str:
    """Query geospatial data from PostGIS."""
    try:
        conn = psycopg2.connect(dbname="geodb", user="user",
           password="pass", host="localhost")
        cur = conn.cursor()
        cur.execute(query)
        result = cur.fetchall()
        conn.close()
        return str(result)
    except Exception as e:
        return f"Error␣querying␣PostGIS:␣{str(e)}"

@tool
def generate_embedding(text: str) -> list:
    """Generate text embeddings."""
    try:
        return embedding_model.encode(text).tolist()
    except Exception as e:
        return f"Error␣generating␣embedding:␣{str(e)}"

server = StdioServer()
server.run()
```

Listing 5: Run MCP Server

```
python mcp_server.py
```

4.3 CrewAI with MCP Integration

CrewAI orchestrates agentic tasks, using MCP tools via `MCPServerAdapter`.

Listing 6: CrewAI with MCP Integration (crewai$_{m}cp.py$)

```python
from crewai import Agent, Task, Crew
from crewai_tools import MCPServerAdapter
from mcp import StdioServerParameters
from transformers import Qwen2VLForConditionalGeneration, AutoProcessor
import os

model = Qwen2VLForConditionalGeneration.from_pretrained("/path/to/models/Qwen2-VL-2B-Instruct", device_map="cuda")
processor = AutoProcessor.from_pretrained("/path/to/models/Qwen2-VL-2B-Instruct")

server_params = StdioServerParameters(
    command="python3",
    args=["mcp_server.py"],
    env={"UV_PYTHON": "3.11", **os.environ}
)

with MCPServerAdapter([server_params]) as aggregated_tools:
    data_extraction_agent = Agent(
        role="Data␣Extractor",
        goal="Extract␣and␣summarize␣text␣from␣unstructured␣and␣semi-structured␣data",
        backstory="Expert␣in␣processing␣PDFs,␣images,␣audio,␣JSON,␣and␣XML␣with␣high␣accuracy",
        llm=model,
        tools=aggregated_tools,
        verbose=True
    )
    geospatial_agent = Agent(
        role="Geospatial␣Analyst",
        goal="Analyze␣and␣query␣geospatial␣data␣for␣insights",
        backstory="Specialist␣in␣geospatial␣data␣processing␣and␣PostGIS␣queries",
        llm=model,
        tools=aggregated_tools,
        verbose=True
    )

    extract_task = Task(
        description="Extract␣text␣from␣PDFs,␣JSON,␣XML

System:␣XML,␣and␣audio␣in␣HDFS,␣generate␣embeddings,␣and␣cache␣summaries␣in␣Redis",
        expected_output="Summary␣of␣extracted␣data",
        agent=data_extraction_agent,
        prompt="Use␣ReAct:␣Reason␣about␣the␣content␣of␣PDFs,␣JSON
```

```
            ,␣and␣audio,␣extract␣key␣information,␣generate␣embeddings,␣cache␣in␣Redis,␣and␣summarize."
    )
    geospatial_task = Task(
        description="Query␣PostGIS␣for␣locations␣within␣10km␣of␣a␣given␣point,␣evaluate␣results",
        expected_output="Geospatial␣analysis␣report",
        agent=geospatial_agent,
        prompt="Use␣ToT:␣Explore␣multiple␣query␣strategies,␣evaluate␣accuracy,␣select␣optimal␣query."
    )

    crew = Crew(agents=[data_extraction_agent, geospatial_agent], tasks=[extract_task, geospatial_task], verbose=2)
    result = crew.kickoff()
    print(result)
```

4.4 API Layer

A FastAPI endpoint exposes the integrated pipeline for external access.

Listing 7: FastAPI Endpoint (api.py)

```
from fastapi import FastAPI
from crewai import Agent, Task, Crew
from crewai_tools import MCPServerAdapter
from mcp import StdioServerParameters
from transformers import Qwen2VLForConditionalGeneration, AutoProcessor
import os

app = FastAPI()
model = Qwen2VLForConditionalGeneration.from_pretrained("/path/to/models/Qwen2-VL-2B-Instruct", device_map="cuda")
processor = AutoProcessor.from_pretrained("/path/to/models/Qwen2-VL-2B-Instruct")

@app.post("/process")
async def process_data(file_path: str, file_type: str):
    server_params = StdioServerParameters(command="python3", args=["mcp_server.py"], env={"UV_PYTHON": "3.11", **os.environ})
    with MCPServerAdapter([server_params]) as aggregated_tools:
        analyzer = Agent(
            role="Data␣Analyzer",
            goal="Process␣and␣analyze␣various␣data␣formats",
            backstory="Expert␣in␣handling␣multimodal␣data",
            llm=model,
            tools=aggregated_tools,
            verbose=True
        )
        task = Task(
            description=f"Analyze␣{file_path}␣as␣{file_type}",
            expected_output="Summary␣of␣input",
            agent=analyzer
        )
        crew = Crew(agents=[analyzer], tasks=[task], verbose=2)
        return {"response": str(crew.kickoff())}
```

Listing 8: Run FastAPI Server

```
uvicorn api:app --host 127.0.0.1 --port 8000
```

4.5 Deployment Considerations

- **Scalability**:
  - Deploy Kafka, Spark, and CrewAI/MCP on Kubernetes for horizontal scaling.
  - Use Sparks dynamic allocation for preprocessing large datasets.
  - Scale PostGIS with read replicas for high query loads.
- **Security**:
  - Enable Kerberos for Hadoop security.
  - Use SSL/TLS for Kafka and PostGIS connections.
  - Implement OAuth2 for frontend API authentication.
  - Bind MCP server to `localhost` for offline security.
- **Monitoring**:
  - Monitor pipeline with Prometheus (Kafka lag, Spark job metrics, MCP/CrewAI performance).
  - Visualize metrics with Grafana dashboards.
  - Log errors with ELK Stack (Elasticsearch, Logstash, Kibana).
- **Error Handling**:
  - Implement Kafka dead-letter queues for failed messages.
  - Use retry policies in Spark and CrewAI for transient failures.
  - Handle Qwen2-VL-2B errors (e.g., token limits) with fallback prompts.
- **Performance Optimization**:
  - Optimize Spark jobs with caching and broadcast joins.
  - Use Redis to reduce repeated embedding generation.
  - Partition PostGIS tables by spatial regions.
  - Use vLLM for Qwen2-VL-2B inference.

4.6 Challenges and Mitigations

1. **Data Volume**:
   - *Challenge*: Large unstructured data (e.g., videos) can overwhelm HDFS and Spark.
   - *Mitigation*: Use compression (Snappy, Gzip) and distributed processing.
2. **Real-Time Processing**:
   - *Challenge*: Kafka streaming may face latency issues with high throughput.
   - *Mitigation*: Increase partitions and use consumer groups.
3. **AI Accuracy**:
   - *Challenge*: Qwen2-VL-2Bs reasoning may be suboptimal for complex tasks.
   - *Mitigation*: Fine-tune Qwen2-VL-2B; use ToT for robust reasoning.
4. **Frontend Scalability**:
   - *Challenge*: Rendering large Mapbox maps or Chart.js charts may slow the UI.
   - *Mitigation*: Implement lazy loading and client-side caching.

5 Integration Flow

1. *Data Ingestion*: Sqoop and Flume ingest data into HDFS, partitioned by type and date.
2. *Streaming*: Kafka Connect streams HDFS data to topics, serialized with Avro or binary formats.
3. *Geospatial Storage*: Kafka consumers parse and store geospatial data in PostGIS with GIST indexes.
4. *Preprocessing*: Spark preprocesses data; MCP tools handle PDF, JSON, XML, audio, and video processing.
5. *AI Processing*: CrewAI assigns tasks to agents powered by Qwen2-VL-2B and Whisper-Tiny, using MCP tools and ReAct/ToT prompts.
6. *Data Store Access*: Agents query Milvus, Redis, and PostGIS/Hive via MCP tools.
7. *Frontend Interaction*: Users submit queries via React UI, view results as visualizations or reports via APIs.

6 Diagram Explanation

- **Nodes**: Components (HDFS, Kafka, PostGIS, Spark, MCP Server, CrewAI, Qwen2-VL-2B, React UI) represented as points in a scatter chart.
- **Layers**:
  - Backend (Blue, #3b82f6): HDFS, Kafka, PostGIS, Spark.

  - Middleware (Green, #10b981): MCP Server, CrewAI, Qwen2-VL-2B, Whisper-Tiny, Milvus, Redis.
  - Frontend (Orange, #f59e0b): React UI, Mapbox, Chart.js.
- **Edges**: Arrows show data flow:
  - HDFS → Kafka → PostGIS/Spark.
  - Spark/PostGIS → MCP Tools → CrewAI.
  - CrewAI → Qwen2-VL-2B/Whisper-Tiny → Data Stores.
  - CrewAI → React UI → Visualizations.
  - Bidirectional arrow between React UI and CrewAI for query/result exchange.
- **Colors**: Blue (Backend), Green (Middleware), Orange (Frontend), dark gray (#1f2937) for arrows/text.
- **Layout**: Nodes arranged vertically by layer (x-axis) and by function (y-axis).

Interpretation:

- *Backend Flow*: Data starts in HDFS, streams through Kafka, and is stored in PostGIS or preprocessed by Spark.
- *Middleware Flow*: MCP tools preprocess data; CrewAI orchestrates tasks using Qwen2-VL-2B and Whisper-Tiny.
- *Frontend Flow*: React UI displays results as visualizations or reports.

7 Conclusion

The integrated **MCP server and CrewAI** framework is the optimal solution for the Agentic AI Server project, replacing the original CrewAI-only approach. The MCP server provides standardized tools for preprocessing PDFs, CSVs, JSON, XML, audio, and video, while CrewAI orchestrates complex agentic workflows using Qwen2-VL-2B for image, video, and text processing and Whisper-Tiny for audio transcription. This meets all requirements:

- **Direct Processing**: Qwen2-VL-2B handles images and video natively; Whisper-Tiny handles audio; MCP tools preprocess other formats.
- **Lightweight**: Fits on NVIDIA A100 with minimal overhead.
- **Offline Compatibility**: Fully offline on Ubuntu 24.04 with pre-downloaded dependencies.
- **Agentic Capabilities**: CrewAI enables autonomous, multi-agent tasks with ReAct/ToT prompts.

The implementation plan provides a clear path for environment setup, MCP tool development, CrewAI integration, and API deployment. Future enhancements could include fine-tuning Qwen2-VL-2B for domain-specific tasks or expanding MCP tools for advanced processing.