AI Microservices Platform – Architecture & Implementation Guide

This document details an AWS-based microservices architecture for an AI-powered MVP. It covers system design, development steps, and best practices for each feature, focusing on containerized services (excludes serverless). The platform is built with Python/FastAPI microservices, leveraging LangChain, LangGraph, LlamaIndex and modern ML tools (e.g. HuggingFace, Whisper) for AI functionalities.

System Architecture Overview

We adopt a **microservices approach** – small, independently deployable services communicating via APIsdocs.aws.amazon.commedium.com. Each service handles a specific AI feature (e.g. "ReplyAssistant Service" or "SentimentAnalysis Service"). This aligns with the 12-Factor App and Domain-Driven Design principlesmedium.comharryk4y.medium.com. On AWS, services run in containers (Amazon ECS/EKS), ensuring scalability and isolationharryk4y.medium.com. We front them with an API Gateway or FastAPI-based router to route requests to the appropriate servicemedium.comharryk4y.medium.com. Services are stateless (no local data) for easy scaling; any needed state (logs, user data) is stored in external databases (RDS, DynamoDB, S3)harryk4y.medium.com. Inter-service calls use HTTP (via `httpx` or `requests`) and JSON contractsmedium.com.

Logging and observability are centralized: each service uses a standard logger (e.g. Loguru) writing to CloudWatch or ELK. AWS best practices like auto-scaling, load balancing (ALB) and service discovery (Cloud Map/App Mesh) ensure resilienceharryk4y.medium.comharryk4y.medium.com. We design API contracts with Pydantic models to enforce schemas. Container images are managed via ECR, deployed on ECS/EKS with Fargate for serverless-like ease. In sum, this microservices architecture yields modular, scalable services that can be developed and deployed independentlydocs.aws.amazon.commedium.com.

Feature: AI Reply Assistant

Purpose: Provide an intelligent conversational assistant (e.g. email/chat reply). It takes a user query (or email content) and generates a helpful, context-aware reply.

Microservices:

- **ReplyService** – FastAPI service with endpoint `POST /reply`.
- *Optional:* **PersonaService** (for tone/personality), **LoggingService** for conversation logs.

Dependencies: We use LangChain for prompt/chain management and optionally LangGraph if multi-step reasoning/approval is neededibm.com. The core model can be OpenAI's GPT-3.5/GPT-4 (via API) or a self-hosted LLM (e.g. Llama-2, Mistral). If user data or history is needed, we can integrate LlamaIndex to retrieve relevant context from documents or logsdocs.llamaindex.ai.

Tools/Libraries: Python, FastAPI, Pydantic, LangChain, LangGraph (for orchestrating multi-turn), LlamaIndex (for knowledge retrieval), OpenAI API or HuggingFace transformers (e.g. LLAMA-based models).

Models:

- Base LLM (GPT-4 or open models).
- *Optional:* Retrieval component (embedding model e.g. OpenAI's text-embedding-ada-002 or HuggingFace "all-MiniLM"), LlamaIndex for RAG.

Development Steps:

API & Data Model: Define `ReplyRequest` and `ReplyResponse`. For example:

json
CopyEdit
```
// ReplyRequest
{
  "user_input": "string",      // user message or email text
  "history": ["string", ...]   // (optional) chat history
}
```
json
CopyEdit
```
// ReplyResponse
{
  "reply_text": "string"
}
```

1.

Service Logic: In FastAPI, load the LLM client (e.g. `ChatOpenAI` from LangChain). Define `/reply` POST that takes `ReplyRequest`. Use LangChain to create a chat chain. Example prompt template:

css
CopyEdit
```
You are a helpful assistant. Given the conversation, provide a
professional reply:
User: {user_input}
Assistant:
```

2.
3. **Context Augmentation:** If using RAG: on each request, use LlamaIndex to query knowledge base with `user_input` (or conversation history) and prepend relevant facts to the prompt. This enhances factuality[docs.llamaindex.ai](docs.llamaindex.ai).

Implement Endpoint: Use FastAPI to call the LLM. For example, following [19†L139-L147]:

python
CopyEdit
```
from fastapi import FastAPI
from pydantic import BaseModel
from langchain.llms import OpenAI
```

```
app = FastAPI()
class ReplyRequest(BaseModel):
    user_input: str
    history: list[str] = []
llm = OpenAI(model="gpt-4")
@app.post("/reply")
def reply(request: ReplyRequest):
    prompt = build_prompt(request.user_input, request.history)
    res = llm(prompt)
    return {"reply_text": res}
```

4.
5. **Testing:** Simulate example queries. Ensure proper formatting.

Architecture Diagram:

mermaid
CopyEdit

```
graph LR
  subgraph API Gateway
    A[Client] -->|POST /reply| B(ReplyService)
  end
  B --> C[LangChain Prompt Chain]
  C --> D[LLM (e.g. GPT-4)]
  C --> E[LlamaIndex (Optional KB Search)]
  D --> F[(Reply)]
  E --> C
```

Input/Output Schemas (JSON):

- *Input:* `{"user_input": "How do I reset my password?", "history": ["Previous conversation turns..."]}`
- *Output:* `{"reply_text": "You can reset your password by clicking..."}`

Example Prompt Template:

vbnet
CopyEdit

```
System: You are a helpful email assistant.
User: Hello, I forgot my account password. Can you help me reset it?
Assistant:
```

Functionality: Receives user text, optionally consults knowledge base, invokes LLM, and returns a reply.

Feature: Auto-Tagging

Purpose: Automatically generate relevant tags (keywords or categories) for a piece of text (e.g. a support ticket or document). Tags help with organization, search, and routing.

Microservices:

- **TaggingService** – FastAPI service with `POST /auto-tag`.

Tools/Models: Options include fine-tuned text classification models or LLM-based tagging. For an LLM approach, use GPT-4 or similar with a prompt like "List relevant tags for this text." Alternatively, use HuggingFace's multi-label classifiers (e.g., using `pipeline("text-classification", model="multi-label-classifier")`). Libraries: FastAPI, Transformers, scikit-learn (for preprocessing), SpaCy (for NLP preprocessing).

Development Steps:

1. **Dataset:** Prepare labeled data mapping text to tags for fine-tuning (optional). For MVP, use zero-shot LLM or rule-based tag list.

2. **Model Selection:** E.g. use Hugging Face Zero-Shot Classification pipeline to predict tags from a predefined label set, or prompt an LLM.

API Design: Define `TagRequest` and `TagResponse`. Example:

json
CopyEdit
```
{ "text": "string" }  ->  { "tags": ["tag1", "tag2", ...] }
```

3.

Implementation: In FastAPI, create endpoint that takes input text. If using HuggingFace:

python
CopyEdit
```
from transformers import pipeline
classifier = pipeline("text-classification",
model="joeddav/distilbert-base-uncased-go-emotions-student")
@app.post("/auto-tag")
def auto_tag(req: TagRequest):
    result = classifier(req.text)
    tags = extract_top_tags(result)
    return {"tags": tags}
```

4. Or if using LLM: send prompt like `f"Extract relevant keywords or topics from the following text: {req.text}"`.

5. **Output Format:** Return a JSON list of tags (strings).

Architecture Diagram:

mermaid
CopyEdit
```
flowchart LR
    A[Client Text Input] --> B(TaggingService)
```

```
B --> C[LLM / Classifier Model]
C --> D[(tags list)]
```

Schemas:

- *Input:* `{"text": "Our new AI system will analyze customer feedback."}`
- *Output:* `{"tags": ["AI", "customer feedback", "analysis"]}`

Prompt Example (LLM):

vbnet
CopyEdit
```
System: You extract key topics or tags.
User: The document discusses training a neural network on image data.
Assistant: ["neural network", "image data", "machine learning"]
```

Feature: Sentiment Analysis

Purpose: Determine the sentiment (e.g. positive, neutral, negative) of a given text message or content.

Microservices:

- **SentimentService** – FastAPI service with `POST /sentiment`.

Tools/Models: Use a pretrained sentiment analysis model from HuggingFace (e.g. `distilbert-base-uncased-finetuned-sst-2-english`) via the Transformers pipelinemedium.com. Pydantic for input validation.

Development Steps:

API & Schema: Define `SentimentRequest` (with `text`) and `SentimentResponse`. Example schema:

json
CopyEdit
```
{ "text": "string" }
=>
{ "label": "positive/negative/neutral", "score": float }
```

1.

Model Pipeline: Initialize HF pipeline:

python
CopyEdit
```
from transformers import pipeline
sentiment_pipe = pipeline("sentiment-analysis")
```

2.

Endpoint Implementation: Wrap the pipeline as in [19†L139-L148]:

python
CopyEdit
```
@app.post("/sentiment")
def analyze(request: SentimentRequest):
    result = sentiment_pipe(request.text)[0]
    return {"label": result["label"], "score": result["score"]}
```

3.
4. **Return Format:** JSON with sentiment label and confidence.

Architecture Diagram:

mermaid
CopyEdit
```
flowchart TD
    InputText -->|HTTP POST| SentimentService
    SentimentService --> HFModel("HuggingFace\nSentiment Pipeline")
    HFModel --> Output("label+score JSON")
```

Schemas:

- *Input:* `{"text": "I love this product!"}`
- *Output:* `{"label": "POSITIVE", "score": 0.99}` (format as in [19†L155-L159]).

Feature: Spam Filter

Purpose: Detect and filter out spam or malicious messages (e.g. in emails, chat).

Microservices:

- **SpamFilterService** – FastAPI service with `POST /spam-check`.

Tools/Models: Use a binary text classifier. Possible models: a fine-tuned transformer on spam detection (or simple ML with TF-IDF + classifier). For MVP, a HuggingFace model (or OpenAI eval with spam prompts) works. Use `pipeline("text-classification", model="unitary/unbiased-toxic-bert" or a spam-specific model)` or train on SpamAssassin data.

Development Steps:

1. **Schema:** Input `{ "text": "..." }`, output `{ "is_spam": bool, "score": float }`.

Implementation: Similar to sentiment, but threshold the output. Example:

python

CopyEdit

```
classifier = pipeline("text-classification",
model="your-spam-model")
@app.post("/spam-check")
def check_spam(request: SpamRequest):
    res = classifier(request.text)[0]
    is_spam = (res["label"] == "LABEL_1")  # assume label_1=spam
    return {"is_spam": is_spam, "score": res["score"]}
```

2.
3. **Model Training (optional):** Fine-tune on a spam dataset if needed.
4. **Integration:** Downstream, the NotificationService or EmailService can use this output to drop or flag spam.

Diagram:

mermaid
CopyEdit

```
graph TD
   UserMsg -->|POST| SpamFilterService
   SpamFilterService --> ClassifierModel
   ClassifierModel --> SpamFilterDecision[(True/False + score)]
```

Schemas:

- *Input:* `{"text": "Congrats! You've won a prize. Click here!"}`
- *Output:* `{"is_spam": true, "score": 0.95}`

Feature: Summarization

Purpose: Generate a concise summary of input text or documents (e.g. summarizing articles or chat logs).

Microservices:

- **SummarizationService** – FastAPI `POST /summarize`.

Tools/Models: Use a pre-trained summarization model from HuggingFace (e.g. `t5-base`, `facebook/bart-large-cnn`) via pipeline[analyticsvidhya.com](analyticsvidhya.com). Alternatively, use an LLM with a summarization prompt. Use `transformers.pipeline("summarization")`.

Development Steps:

1. **Schema:** Input `{ "text": "..." }` (and optional parameters like length). Output `{ "summary": "..." }`.

Implement Pipeline:

python

CopyEdit
```
from transformers import pipeline
summarizer = pipeline("summarization")
@app.post("/summarize")
def summarize(req: SummarizationRequest):
    sum_text = summarizer(req.text,
max_length=100)[0]["summary_text"]
    return {"summary": sum_text}
```

2. The pipeline abstracts model loading per [37†L243-L246].

3. **API Testing:** Ensure it handles large text (may need truncation or chunking for very long inputs).

4. **Model Tuning:** Optionally fine-tune on domain data for improved accuracy.

Architecture Diagram:

mermaid
CopyEdit
```
flowchart LR
    C[Content Input] --> SummarizationService
    SummarizationService --> Summarizer("HF Summarization Model")
    Summarizer --> Result["summary text"]
```

Schemas:

- *Input:* `{"text": "The quick brown fox jumps over ... (long text)"}`
- *Output:* `{"summary": "The fox quickly jumps over..."}`

Prompt Template (if LLM-based):

bash
CopyEdit
```
Summarize the following text in a concise manner:
\"\"\"{text}\"\"\"
```

Feature: Voice-to-Text

Purpose: Transcribe spoken audio into text (speech recognition).

Microservices:

- **SpeechService** – FastAPI `POST /transcribe`.

Tools/Models: Use OpenAI's Whisper model (via HuggingFace Transformers)huggingface.co. Whisper is a powerful pre-trained ASR model. We wrap it in our service.

Development Steps:

Input Handling: Accept audio via multipart/form-data or URL. (In JSON schema, we use a pre-signed URL or Base64 string). Example input schema:

json
CopyEdit
```json
{ "audio_url": "string (HTTP URL to audio)" }
```

1. Alternatively, accept binary form.

2. **Preprocessing:** Download the audio, convert to the required format (e.g. WAV, 16kHz) using libraries like `pydub` or `soundfile`.

Load Model: Use the Whisper pipeline or model. Example:

python
CopyEdit
```python
from transformers import pipeline
asr = pipeline("automatic-speech-recognition",
model="openai/whisper-large-v2")
@app.post("/transcribe")
def transcribe(req: TranscriptionRequest):
    audio = download_and_load_audio(req.audio_url)
    text = asr(audio)["text"]
    return {"transcript": text}
```

3. Whisper requires a `WhisperProcessor` for preprocessing, but the pipeline simplifies it.

4. **Output:** Return the transcription text.

Architecture Diagram:

mermaid
CopyEdit
```mermaid
flowchart TD
    AudioFile -->|POST| SpeechService
    SpeechService --> WhisperModel("Whisper ASR Model")
    WhisperModel --> Transcription["text output"]
```

Schemas:

- *Input:* `{"audio_url": "https://.../audio.wav"}`
- *Output:* `{"transcript": "Hello, how are you?"}`

Remark: Whisper handles multilingual ASR too (it can identify and transcribe languages)huggingface.co. For translation, see **Multilingual AI**.

Feature: Multilingual AI

Purpose: Provide language translation and/or multi-language processing (e.g. user queries in various languages). Could include on-the-fly translation of input/output or support for multi-language chat.

Microservices:

- **TranslationService** – FastAPI `POST /translate`.
- *Optional:* **LanguageDetectionService** or integrated.

Tools/Models: Use HuggingFace translation models (e.g. Facebook M2M100) or GPT with translation prompts. HF's `pipeline("translation_xx_to_yy")` can also be used. If using LLM, prefix user input with "Translate into {target_lang}: ...".

Development Steps:

1. **Schema:** `{ "text": "...", "target_lang": "fr" }` → `{ "translation": "..." }`.

Model Pipeline: Use HF:

python
CopyEdit
```
from transformers import pipeline
translator = pipeline("translation", model="facebook/m2m100_418M")
@app.post("/translate")
def translate(req: TranslateRequest):
    out = translator(req.text, src_lang="en",
tgt_lang=req.target_lang)[0]
    return {"translation": out['translation_text']}
```

2. (Adjust `src_lang` based on detected or assumed language.)
3. **Language Detection:** Optionally detect language with a library (e.g. LangDetect) or let translation model infer source.
4. **Integration:** Useful for the Reply Assistant (allow multilingual queries) or translating content for search.

Diagram:

mermaid
CopyEdit
```
flowchart TD
    QueryText --> TranslationService
    TranslationService --> Model("Translation Model")
    Model --> TranslatedText
```

Schemas:

- *Input:* `{"text": "Hello world", "target_lang": "fr"}`

- *Output:* `{"translation": "Bonjour le monde"}`

Note: Transformer-based translation is sequence-to-sequence similar to summarizationhuggingface.co.

Feature: AI Writing Assistant

Purpose: Help users write or improve text (e.g. grammar correction, style enhancement, suggestion for content).

Microservices:

- **WritingAssistantService** – FastAPI `POST /improve-text`.

Tools/Models: Use an LLM (GPT-4/3.5) or dedicated grammar model (e.g. LanguageTool or [Grammarly API]). For generative improvement, use LLM with prompts like “Improve the following email.”

Development Steps:

1. **Schema:** `{ "text": "...", "task": "grammar/style/formatting" }` → `{ "improved_text": "..." }`.

Prompt Strategy: A prompt template might be:

css
CopyEdit
```
The user has written a paragraph. Improve its clarity and style:
"{text}"
```

2.

API Implementation: Similar to ReplyService, but static instruction. Example:

python
CopyEdit
```
llm = ChatOpenAI(model="gpt-4")
@app.post("/improve-text")
def improve(req: WritingRequest):
    prompt = f"Improve the following text for grammar and
style:\n\"{req.text}\""
    res = llm(prompt)
    return {"improved_text": res}
```

3.
4. **Alternatives:** Use a classification-based approach for grammar (LangTool), or a fine-tuned transformer for grammar correction (e.g. T5 fine-tuned on correction tasks).

Diagram:

mermaid
CopyEdit

```
flowchart LR
    UserDraft --> WritingService
    WritingService --> LLM("Text Generation Model")
    LLM --> ImprovedText
```

Schemas:

- *Input:* `{"text": "He go to school yesterday.", "task": "grammar"}`
- *Output:* `{"improved_text": "He went to school yesterday."}`

Prompt Example:

kotlin
CopyEdit

```
User: Please correct the grammar and improve the clarity of this sentence:
"She do not want to go to movie tonight."
Assistant: ...
```

Feature: AI Site Search

Purpose: Enable semantic search over website or document content. Given a query, return relevant documents/snippets.

Microservices:

- **SearchService** – FastAPI `POST /search`.
- **IndexingService** – background or on-demand service that ingests site content into a vector index.

Tools/Models: Use LlamaIndex to build a retrieval index and query enginedocs.llamaindex.ai. Underlying vector store could be Chroma, Pinecone, or Weaviate. Use pre-computed embeddings (OpenAI or HF).

Development Steps:

1. **Data Ingestion:** Gather website content or document corpus. Preprocess (clean HTML, segment into passages). Store raw data (S3 or DB).

Indexing: Use LlamaIndex to create a Vector Index:

python
CopyEdit

```
from llama_index import SimpleWebPageReader, GPTVectorStoreIndex
docs = SimpleWebPageReader(urls=["https://example.com"]).load_data()
index = GPTVectorStoreIndex.from_documents(docs)
index.save_to_disk("index.json")
```

2.

Query API: The SearchService loads the index (or queries a live index). It takes a `query` and returns top results. Example:

python
CopyEdit
```
@app.post("/search")
def search(req: SearchRequest):
    results = index.query(req.query, top_k=req.k)
    return {"results": [res.text for res in results]}
```

3. This uses LlamaIndex's query engine for RAG.

4. **JSON Schema:** Input: `{ "query": "topic keywords", "top_k": 5 }`. Output: list of matches with source references.

Diagram:

mermaid
CopyEdit
```
graph LR
   Query --> SearchService
   SearchService -->|embeddings| VectorStore
   VectorStore --> SearchService
   SearchService --> [Result Snippets]
```

Schemas:

- *Input:* `{"query": "best microservices framework", "top_k": 3}`

Output:

json
CopyEdit
```
{"results": [
  {"doc_id": "doc1", "text": "Our platform uses FastAPI for
microservices...", "score": 0.95},
  {"doc_id": "doc2", "text": "We integrate LangChain for LLM
workflows...", "score": 0.90}
]}
```

-

Feature: Internal AI Copilot

Purpose: An agentic assistant for internal tasks (e.g. answering complex queries, automating workflows, providing contextual help across systems). It orchestrates other microservices (like search, summarization) and maintains conversation state.

Microservices:

- **CopilotService** – FastAPI `POST /copilot`. Acts as a LangGraph agent coordinator.

- Relies on other services (calls them as tools).

Tools/Models: Use LangGraph (with LangChain) to create a multi-step agent. This service can call other microservices (via HTTP) as tools/actions. LangGraph provides stateful orchestration and human-in-loop checksibm.comlangchain.com.

Development Steps:

1. **Define Agent Workflow:** Design tasks (e.g. "Collect user info via search, then summarize relevant docs, then draft email reply"). Using LangGraph or LangChain Agent API, script the steps.

2. **Tool Integration:** In LangChain, define tools that call our microservices (e.g. a SearchTool hitting `/search`, SummarizeTool hitting `/summarize`). Register these tools with the agent.

LangGraph Orchestration: Use LangGraph's stateful agent to maintain context across calls. Example (pseudocode):

python
CopyEdit
```
from langgraph import Agent
agent = Agent(tools=[SearchTool, SummarizeTool, TagTool],
model="gpt-4")
@app.post("/copilot")
def copilot(req: CopilotRequest):
    response = agent.run(query=req.input, history=req.history)
    return {"result": response}
```

3.
4. **Human-in-Loop (Optional):** LangGraph supports human review of each stepibm.com.

5. **Memory:** The Copilot can store conversation memory (LangGraph built-in memory) to personalize interactionslangchain.com.

Diagram:

mermaid
CopyEdit
```
flowchart TD
    UserQuery --> CopilotService
    CopilotService -->|Tool call| SearchService
    CopilotService -->|Tool call| SummarizationService
    CopilotService -->|Tool call| TaggingService
    CopilotService -->|Uses LLM| LLMModel
    LLMModel --> CopilotService
    CopilotService --> UserResponse
```

Schemas:

- *Input:* `{"input": "Summarize our Q3 sales report and draft an email to execs.", "history": []}`
- *Output:* `{"result": "Email draft or summary text"}`

Note: LangGraph excels at such multi-step agentic workflows, offering built-in statefulness and flexibility for complex tasksibm.comlangchain.com.