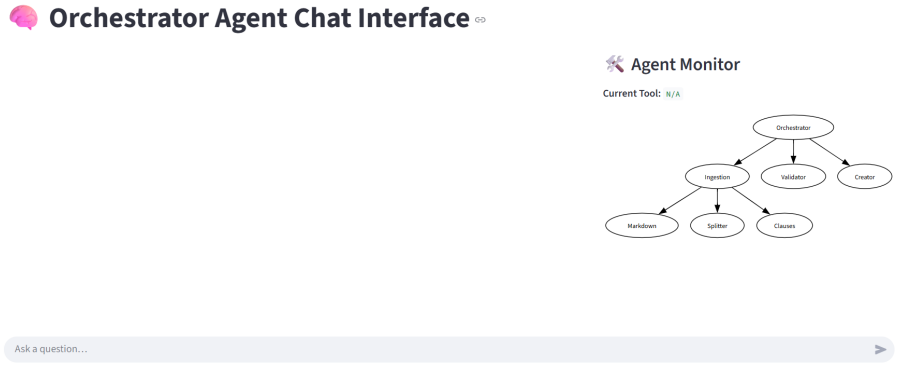
Voila AWS Hackathon: Solution Summary

Table of Contents

Introduction

This document details the architecture and solutions developed for the Voila AWS Hackathon project. The system is built on a modular, agentic approach, where each agent encapsulates a specific responsibility and exposes tools that can be orchestrated by other agents. The solution leverages Amazon Knowledge Base for vector search, the Strands agentic framework for orchestration, and a short memory object for stateful, multi-step workflows.

The UI interface is made in streamlit, allowing users to interact with the agents and tools seamlessly. Also it make possible to host the solution on AWS, in a serverless manner or in a EC2 instance.



Project Architecture

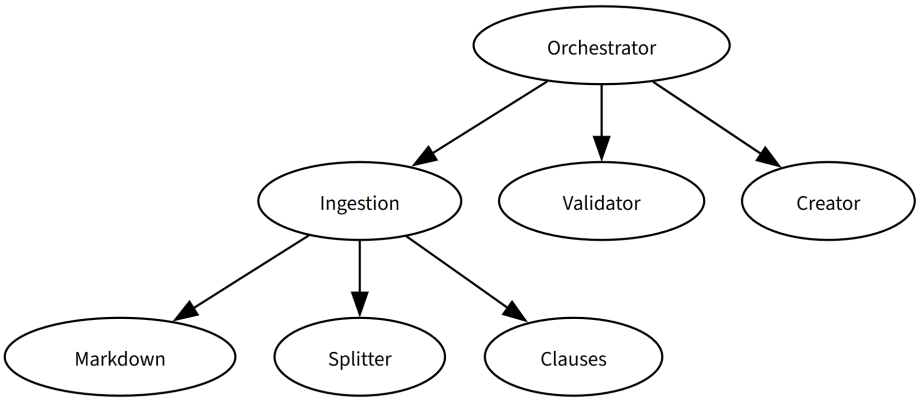
The project automates the ingestion, processing, and validation of policy documents using a multi-agent system. The workflow includes:

1. Ingesting documents from AWS S3.

2. Converting and normalizing document formats (PDF → Markdown).
3. Splitting documents into logical sections.
4. Extracting and ranking relevant clauses.
5. Retrieving information using a vector database (Amazon Knowledge Base).
6. Validating clauses against context and information retrieved from the vector database.
7. Generating final responses to user queries.

All those steps are handled by different agents, and many agents are exposed as tools for use by others, enabling composability and extensibility.

Agents and Tools



Orchestrator Agent

- **Purpose:** Coordinates the entire workflow and manages agent execution.
- **Key Tools:**
  - `custom_retrieve`: for vector search in Amazon Knowledge Base, and retrieve a Document List with the relevant documents to the user query.
  - `check_status`: checks if a document is already processed.
  - `ingestion_agent`: orchestrates the ingestion pipeline for new documents.
  - `validate_agent`: validates extracted clauses against context.
  - `create_answer`: generates the final response to user queries, based on validated clauses and user input.
- **How it works:** Retrieves relevant documents from a vector database (Amazon Knowledge Base), checks processing status, triggers ingestion, validation, and answer creation.

- **Interdependency:** Central controller, calls all other agents as tools to orchestrate the workflow and create a cohesive user experience.

Ingestion Agent

- **Purpose:** Orchestrates the ingestion pipeline for new documents.
- **Key Tools:**
  - `check_status`: checks if a document is already processed.
  - `pdf_to_md_agent`: downloads a document from S3 and converts PDF documents to Markdown format.
  - `splitter_agent`: splits Markdown documents into logical sections, can split by titles or sliding window.
  - `clauses_agent`: extracts and ranks clauses from document sections.
- **How it works:** Checks if a document is already processed, converts PDF to Markdown, splits into sections, and extracts clauses and saves them to memory.
- **Interdependency:** Uses Markdown, Splitter, and Clauses agents as tools.

Markdown Agent

- **Purpose:** Handles PDF-to-Markdown conversion and S3 integration.
- **Key Tools:**
  - `download_pdf_from_s3`: downloads a PDF document from S3.
  - `convert_pdf_save_md`: converts the downloaded PDF to Markdown format and saves it.
- **How it works:** Downloads PDFs from S3, converts to Markdown, and saves locally for further processing.
- **Interdependency:** Used by Ingestion Agent.

Splitter Agent

- **Purpose:** Splits Markdown documents into logical sections.
- **Key Tools:**
  - `split_sections_by_title`: splits sections based on titles.
  - `split_sections_by_sliding_window`: splits sections using a sliding window approach.
  - `count_words_and_titles`: counts words and titles in sections.
- **How it works:** Splits documents by headings or sliding window depending on the best approach determined by the content, saves sections as JSON for downstream clause extraction.
- **Interdependency:** Used by Ingestion Agent.

Clauses Agent

- **Purpose:** Extracts and ranks clauses from document sections.
- **Key Tool:**

  - `clauses_agent`: The agent itself is the `tool` used for clause extraction.
- **How it works:** Uses a Bedrock LLM to analyze each section and generate structured clause objects (with area and relevance). Saves top clauses to JSON and memory.
- **Interdependency:** Used as a tool by the Ingestion Agent.

Validator Agent

- **Purpose:** Validates extracted clauses against context.
- **Key Tools:**
  - `compare`: compares clauses against context.
- **How it works:** Uses LLM to check clause validity, retrieve additional context from Amazon Knowledge Base to compare against clauses.
- **Interdependency:** Used by Orchestrator.

Creator Agent

- **Purpose:** Generates the final response to user queries.
- **Key Tool:**
  - `create_response`: uses memory to access validated clauses and user input.
- **How it works:** Retrieves validated clauses and user question from memory, then uses LLM to compose a final answer.
- **Interdependency:** Consumes output from Validator Agent used by Orchestrator.

Tool Interdependencies

- The workflow is designed to be fully agentic controlled.
- Many agents are exposed as tools and are called by other agents, enabling a composable, agentic workflow.
- The Orchestrator Agent is the main entry point, chaining Ingestion, Validation, and Creation steps.
- The Ingestion Agent chains Markdown, Splitter, and Clauses agents as tools.

Vector Database and Knowledge Base

- **Amazon Knowledge Base:** Used to create a vector database of document information, enabling semantic search and retrieval of relevant content for validation and answer generation.

Agentic Framework and Memory

- **Strands Agentic Framework:** All agents and tools are built using the Strands framework, which provides the `@tool` decorator and agent orchestration primitives.
- **Bedrock LLMs:** Used for clause extraction, validation, and answer generation.
- **Short Memory Object:** A lightweight, global memory (`memory` from `AgentsMemory.py`) is used to store and retrieve state (e.g., current agent, tool, top clauses, validated clauses, user input) across the agent workflow. This enables stateful, multi-step processing and allows agents to share context efficiently.

Frameworks and Libraries

- **Python 3.13:** Main programming language.
- **strands-agents, strands-agents-tools:** For agent orchestration and tool exposure.
- **boto3:** For AWS S3 and Bedrock integration.
- **docling:** For document conversion and parsing.
- **pypdf, pypdf2:** For PDF parsing and text extraction.
- **tqdm, graphviz, python-dotenv:** Utilities for progress, visualization, and environment management.
- **Standard libraries:** `json`, `os`, `re`, etc.
- **All dependencies:** See `requirements.txt` for the full list.

Conclusion

The Voila AWS Hackathon project demonstrates a robust, modular approach to document processing using a multi-agent system. Each agent and tool is designed for a specific task, ensuring scalability and maintainability. The use of Amazon Knowledge Base for vector search, the Strands agentic framework for orchestration, and a short memory object for stateful flow organization enables advanced, context-aware document analysis and response generation. This architecture is extensible and can be adapted to support additional document types, validation rules, and integration points as needed. The system is accessible via a Streamlit UI, allowing users to interact with the agents and tools seamlessly. The UI has a graphical interface that allows users to watch the agents working, and which tools are being used at each step, providing transparency and ease of use.