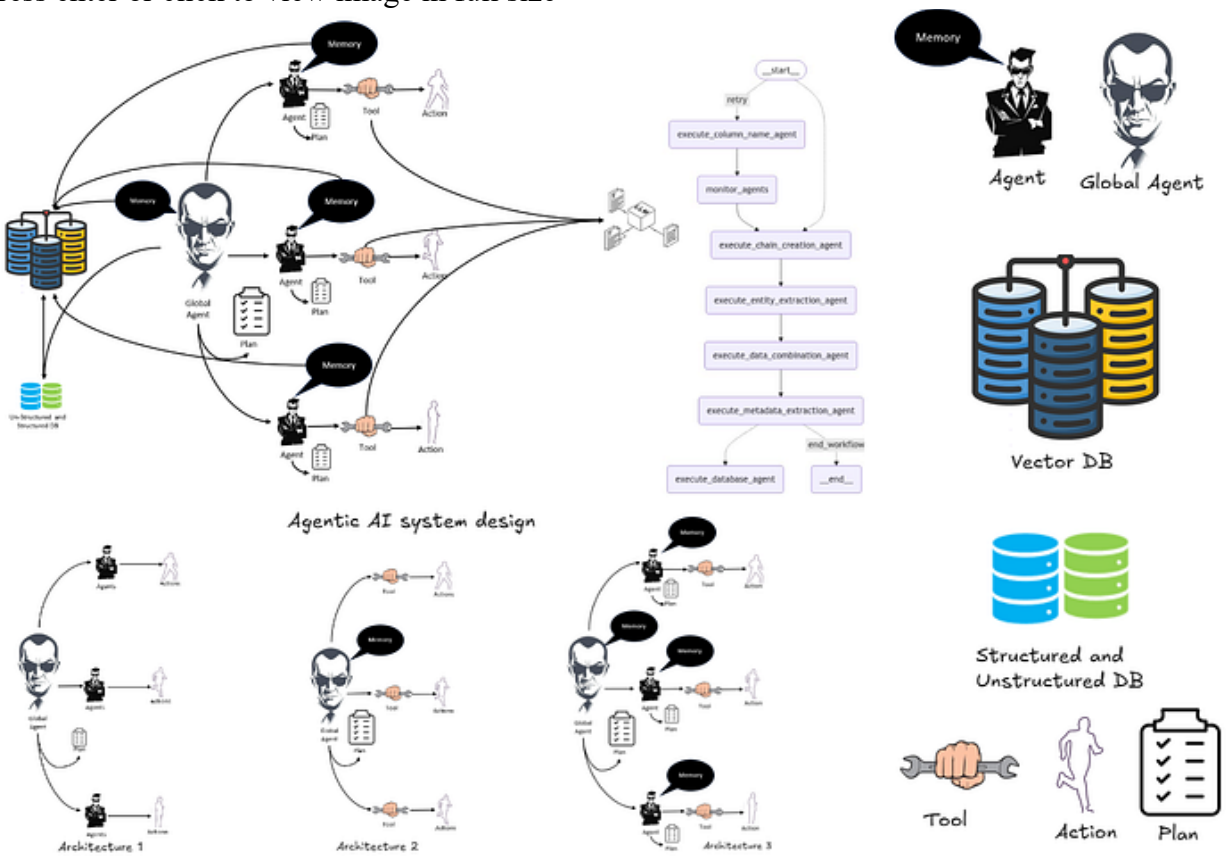
From Class based Agents to langchain Agents

As Artificial Intelligence (AI) systems evolve, the concept of **Agentic AI** — where AI systems are composed of modular, task-specific agents working collaboratively — has become a cornerstone of scalable and adaptable AI solutions. This blog explores the design considerations behind Agentic AI, examining how agents, tools, memory, state, and planning come together to create intelligent workflows. We'll compare three implementations — **CODE1**, **CODE2**, and **CODE5** — to shed light on the practicalities and complexities involved.

Press enter or click to view image in full size



Architecture 1, **Architecture 2**, and **Architecture 3**, mapped to **CODE1**, **CODE2**, and **CODE5** respectively.

The code can be found Here

Objective

The main objective of the provided code is to automate and streamline the extraction, transformation, and integration of structured data from unstructured textual reviews into a usable format for further analysis. Specifically, this workflow addresses the problem of entity extraction from customer reviews, associating extracted entities with existing structured datasets, and combining the two for comprehensive data analysis.

The problem being solved involves three key challenges:

1. **Unstructured Data Handling**: Customer reviews often contain valuable information buried in natural language text. Extracting structured data like customer names and purchase dates from such reviews is critical for enriching existing datasets.
2. **Data Integration**: After extracting relevant information, integrating it with structured datasets (e.g., customer transaction records) ensures a unified, comprehensive view of data for analysis.

3. **Automated Metadata Management**: The solution not only extracts and integrates data but also generates and stores metadata for both structured and unstructured datasets. This facilitates a deeper understanding of the data, such as its statistical properties and potential quality issues.

Tasks

- **ColumnNameAgent**: Parses column name and description pairs into a structured dictionary.
- **ChainCreationAgent**: Sets up a chain using LangChain and a local LLM to perform Named Entity Recognition (NER) on review texts.
- **EntityExtractionAgent**: Applies the chain to extract entities from unstructured text in a dataset.
- **DataCombinationAgent**: Combines the extracted entities with an existing dataset to create an enriched dataset.
- **DatabaseAgent**: Stores the merged data into an SQLite database for further access.
- **MetadataExtractionAgent**: Extracts metadata, such as column details and data statistics, for analysis.

By leveraging advanced language models, database management, and structured workflows, this code enables businesses to automate and

optimize their data processing pipelines, reducing manual effort and ensuring consistent, accurate data integration for decision-making.

The Role of Global Agents in Agentic AI

Global Agents are the central orchestrators in Agentic AI systems, ensuring smooth execution of workflows, coordinating agents or tools, and adapting dynamically to the state of the system. Below, we explore their role in **CODE1**, **CODE2**, and **CODE5**.

Role	CODE1	CODE2	CODE5
Orchestration	Static sequence with explicit logic for agent execution.	Dynamic tool selection via LangChain's reasoning.	Hybrid approach: Static workflows for agents and dynamic execution for tools.
Monitoring	Basic tracking of active agents and state updates.	Implicit monitoring via LangChain's reasoning and memory.	Combines explicit monitoring for structured workflows with implicit monitoring for dynamic tools.
Error Handling	Basic retry logic for failed agents.	Robust retries through LangChain's persistent state and dynamic reasoning.	Combines explicit retries for structured workflows with LangChain's memory-driven error handling.
Adaptability	Limited to predefined workflows.	Fully dynamic and adaptive to context.	Partially adaptable: Blends predictable workflows with flexible LangChain tools.

Global Agents in **CODE1** focus on static, predictable workflows, while those in **CODE2** embrace dynamic adaptability. **CODE5** offers the best of both worlds, orchestrating structured agents while leveraging the flexibility of LangChain tools

Orchestration

The primary role of a Global Agent is to manage the sequence of task execution.

- **CODE1**: The Global Agent explicitly orchestrates a predefined workflow by sequentially invoking agents like `ColumnNameAgent`, `ChainCreationAgent`, and others. Each agent is hardcoded to perform specific tasks in a linear fashion.
- **CODE2**: Instead of explicit orchestration, the Global Agent leverages LangChain's reasoning-driven dynamic execution. Tools are invoked adaptively based on inputs, observations, and the custom prompt (`ZERO_SHOT_REACT_DESCRIPTION`).

- **CODE5**: Combines explicit orchestration for structured workflows with dynamic LangChain tools. The Global Agent coordinates both explicit agents and tools for hybrid workflow execution.

Monitoring

Monitoring the progress and state of the workflow is essential for reliability and debugging.

- **CODE1**: The Global Agent performs basic monitoring by tracking which agent is active and updating the state explicitly after each task.
- **CODE2**: Monitoring is less explicit, as LangChain's agent inherently handles observation and reasoning between tasks. The Global Agent primarily ensures the dynamic flow of tool execution.
- **CODE5**: Integrates both explicit and implicit monitoring. Structured agents track progress in explicit workflows, while LangChain tools provide reasoning and feedback in adaptive workflows.

Error Handling

Global Agents play a crucial role in handling errors and ensuring task completion.

- **CODE1**: Basic error handling relies on retry logic embedded in the workflow. If an agent fails, the Global Agent may attempt to re-execute the task up to a specified limit.
- **CODE2**: LangChain's memory and reasoning framework simplify error handling. Persistent state allows the Global Agent to retry tools seamlessly without requiring explicit logic.
- **CODE5**: Combines structured retry logic for explicit agents with LangChain's robust error-handling for tools. This hybrid approach ensures resilience across static and dynamic workflows.

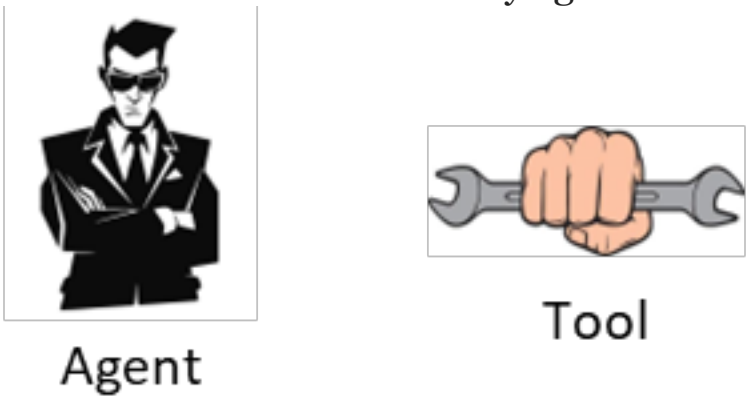
Adaptability

A Global Agent determines the next step dynamically or based on pre-defined rules.

- **CODE1**: Limited adaptability, as the workflow is hardcoded and follows a static sequence.
- **CODE2**: Fully adaptable through LangChain's dynamic reasoning. The Global Agent selects tools based on context and custom prompts.
- **CODE5**: Partially adaptable. Structured workflows provide predictable behavior, while LangChain tools add flexibility to handle varied inputs.

The Role of Agents and Tools in Agentic AI Systems

At the heart of Agentic AI systems are **agents** and **tools**, which modularize functionality to address specific tasks. Agents are independent entities capable of decision-making, whereas tools serve as functional utilities invoked by agents to execute well-defined tasks.



Agent and tool

Agents and Tools Across the Codes

CODE1: Features **6 class-based agents**:

- `ColumnNameAgent`, `ChainCreationAgent`, `EntityExtractionAgent`, `DataCombinationAgent`, `DatabaseAgent`, and `MetadataExtractionAgent`.
- These agents work in a predefined, sequential workflow.

CODE2: Introduces a **LangChain-based approach**:

- **5 tools** replace explicit agents: `ColumnNameExtraction`, `ChainCreation`, `EntityExtraction`, `DataCombination`, and `MetadataExtraction`.
- These tools function dynamically under the guidance of LangChain's `ZERO_SHOT_REACT_DESCRIPTION` agent.

CODE5: Combines **6 explicit agents** and **5 tools**:

- Explicit agents handle workflow orchestration (`execute_column_name_agent`, etc.).
- Tools are invoked dynamically for task execution, blending structured and adaptive approaches.

Aspect	CODE1	CODE2	CODE5
Agents	6	0	6
Tools	0	5	5

Here is the number of **agents** and **tools** in **CODE1**, **CODE2**, and **CODE5**, adesign choice

The choice between agents and tools often depends on the desired balance between modularity and adaptability.

Custom Prompts and 'ZERO_SHOT_REACT_DESCRIPTION'

Custom prompts are essential for reasoning, enabling agents to follow the **"Thought → Action → Action Input → Observation → Final Answer"** framework. The `ZERO_SHOT_REACT_DESCRIPTION` approach in

LangChain leverages this framework to dynamically decide which tool to invoke, based on input context and observations. This allows the system to adapt flexibly to diverse scenarios without retraining.

Aspect	CODE1	CODE2	CODE5
Usage of ZERO_SHOT_REACT_DESCRIPTION	Not used. Relies on predefined workflows and static sequencing of agents.	Fully utilized. Central to dynamically selecting tools based on reasoning and inputs.	Partially used. Combines explicit agent workflows with dynamic tool selection via ZERO_SHOT_REACT_DESCRIPTION.
Purpose	Static task execution with explicit logic for agent invocation.	Dynamic tool invocation based on reasoning, adapting to varied inputs without predefining workflows.	Enhances flexibility by enabling dynamic tool selection while maintaining structured workflows.
Decision-Making	Hardcoded decision-making for agent execution.	Relies on LangChain's reasoning framework to decide the next tool dynamically.	Combines predefined agent transitions with dynamic decisions for tool execution.
Integration	Not applicable.	Integrated via LangChain's `initialize_agent` with custom prompts guiding the workflow.	Integrated for tools, alongside explicit agents in the hybrid workflow.
Dynamicity	None. Sequential, static workflows.	Fully dynamic and adaptable to varied scenarios.	Hybrid. Explicit workflows are predictable, while tools leverage dynamic reasoning.
Custom Prompt Dependency	No custom prompt used.	Relies heavily on the "**Thought → Action → Action Input → Observation → Final Answer**" prompt.	Uses custom prompts for tool reasoning, combined with explicit workflow logic.

Comparison of the usage of **“ZERO_SHOT_REACT_DESCRIPTION”** in **CODE1**, **CODE2**, and **CODE5**

- **CODE1**: Does not use `ZERO_SHOT_REACT_DESCRIPTION` and relies on static workflows.
- **CODE2**: Fully leverages `ZERO_SHOT_REACT_DESCRIPTION` for dynamic and reasoning-driven tool selection.
- **CODE5**: Combines the best of both worlds, using `ZERO_SHOT_REACT_DESCRIPTION` for tools while maintaining explicit agent-based workflows.

State Management

Agents and tools rely on a shared state to track progress, intermediate results, and workflow context. In structured systems like **CODE1**, the

state is explicitly defined (e.g., `GraphState`) and manually updated. Dynamic systems like **CODE2** and **CODE5** use LangChain's `StatefulMemory` to handle state implicitly during tool execution.

Memory Class

A dedicated memory class (e.g., `StatefulMemory`) provides a persistent and centralized way to store, update, and retrieve state variables. While **CODE1** lacks this, **CODE2** and **CODE5** use `StatefulMemory` for dynamic updates, ensuring state is consistently maintained across retries and workflows.

State Variables

Explicit variables in structured systems track predefined workflows (e.g., `column_names`, `chain`, `metadata` in **CODE1**). Dynamic systems allow tools to update variables based on observations, adapting to changing inputs.

Dynamicity of State

Structured agents follow fixed state transitions, while tools in dynamic systems interact with the state adaptively, as seen in **CODE2** and **CODE5**.

Persistence

Memory ensures state is persisted beyond tool execution, enabling retries and error recovery.

Error Handling

Dynamic state systems inherently manage errors better, as persistent memory and retries maintain workflow integrity, while static systems require explicit error-handling logic.

Aspect	CODE1	CODE2	CODE5
State Management	- Uses `GraphState`, a predefined `TypedDict`. - State updates are handled explicitly by agents.	- Utilizes LangChain's `StatefulMemory` for dynamic state management. - State updates are implicit during tool execution.	- Combines `GraphState` for structured state tracking and `StatefulMemory` for dynamic updates.
Memory Class	- No dedicated memory class. - State updates are handled manually.	- Uses LangChain's `StatefulMemory`, enabling automated state persistence and updates.	- Combines `StatefulMemory` for dynamic state updates with explicit manual state management in `GraphState`.
State Variables	- Explicitly defined in `GraphState`. - Examples: `question`, `chain`, `merged_data`, etc.	- Stored as key-value pairs in `StatefulMemory`. - Examples: `chain`, `metadata`, `merged_data`.	- Split between `GraphState` (structured variables) and `StatefulMemory` (dynamic variables).
Dynamicity of State	- Static and limited to predefined workflows. - Explicit logic drives state transitions.	- Fully dynamic state updates based on tool outputs. - Adaptive to varying inputs and workflows.	- Hybrid approach: - `GraphState` provides structure, while `StatefulMemory` supports adaptability.
Persistence	- No persistence beyond the workflow. - State is passed manually between agents.	- Persistent across tasks within `StatefulMemory`, ensuring seamless retries and error handling.	- Combines manual persistence for structured workflows with `StatefulMemory` for automated persistence.
Error Handling	- Basic retries and static error handling. - Errors are agent-specific.	- Robust error handling through memory-based retries. - State consistency aids recovery.	- Combines explicit error handling (for structured tasks) with dynamic retries in `StatefulMemory`.

Actions and Planning in Agentic AI

In Agentic AI, **Actions** and **Planning Mechanisms** are the fundamental elements that drive task execution and optimize workflows, making the system efficient, adaptive, and resilient. Let's explore these in more detail with examples from the codes.

Actions: The Core of Agent Behavior

What Are Actions?

- Actions are the atomic units of an agent's behavior. They represent specific operations, such as invoking a tool, running a method, or executing a predefined task.
- Actions bridge decision-making (reasoning by agents) and execution (invoking tools or methods).

Key Characteristics:

1. **Task-Specific**: Actions are designed to handle specific tasks like extracting entities, combining data, or creating chains.
2. **Dynamic Execution**: In systems like CODE2 and CODE5, actions are dynamically chosen based on the system's reasoning or state.
3. **Tool Invocation**: Actions often involve invoking tools, such as the `ChainCreation` tool, which constructs a Named Entity Recognition (NER) pipeline in CODE2.

Example from CODE2:

- **Thought**: "I need to create an NER chain for the column names."
- **Action**: Invoking the `ChainCreation` tool.
- **Action Input**: A list of column names with descriptions, such as `{"CustomerName": "<Name of customer>", "PurchaseDate": "<Date of purchase>"}`.
- **Observation**: A successfully created NER chain ready for use.

This action-based modularity allows flexibility, as agents can focus on "deciding what to do" while tools perform the actual tasks.

Planning Mechanisms: Optimizing Workflows

What Is Planning in Agentic AI? Planning refers to the process of sequencing and organizing actions to ensure efficient task execution. Effective planning ensures that agents execute the right actions in the right order, handle errors gracefully, and adapt to changing scenarios.

Key Goals of Planning:

Sequencing for Efficiency:

- Planning ensures actions are executed in a logical sequence to minimize redundant computations.
- Example: In CODE1, the `DataCombinationAgent` logically follows the `EntityExtractionAgent` because it depends on the extracted data.

Error Recovery:

- A well-planned system anticipates potential errors and includes retries or alternative paths.
- Example: In CODE5, retries are incorporated both for explicit agents (via manual logic) and tools (via LangChain memory).

Blending Structured and Adaptive Planning:

- Structured Planning: Predefined workflows, as seen in CODE1, where the sequence of agents is hardcoded.
- Adaptive Planning: Dynamic decision-making, as seen in CODE2 and CODE5, where actions are chosen based on reasoning.
- Hybrid Planning: CODE5 exemplifies this by using structured workflows for explicit agents and adaptive planning for tools.

Example of Planning in CODE5

Structured Workflow:

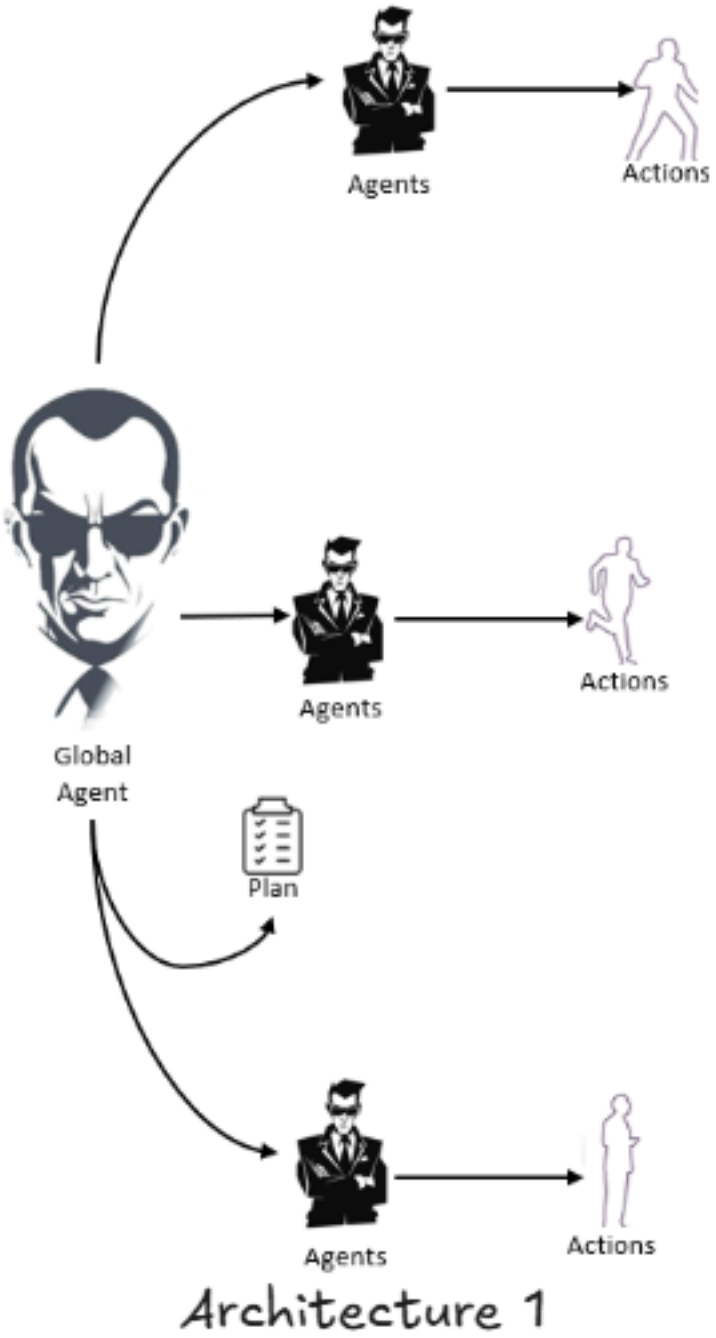
- The `Global Agent` invokes explicit agents (`execute_column_name_agent`, `execute_chain_creation_agent`, etc.) in a fixed sequence.
- This ensures predictable execution for tasks with dependencies.

Dynamic Planning:

- For tools like `ChainCreation`, planning dynamically selects the appropriate tool based on state and reasoning.
- Example: If the system encounters unexpected data, it might dynamically adjust to invoke `MetadataExtraction` for additional context.

Error Handling in Planning:

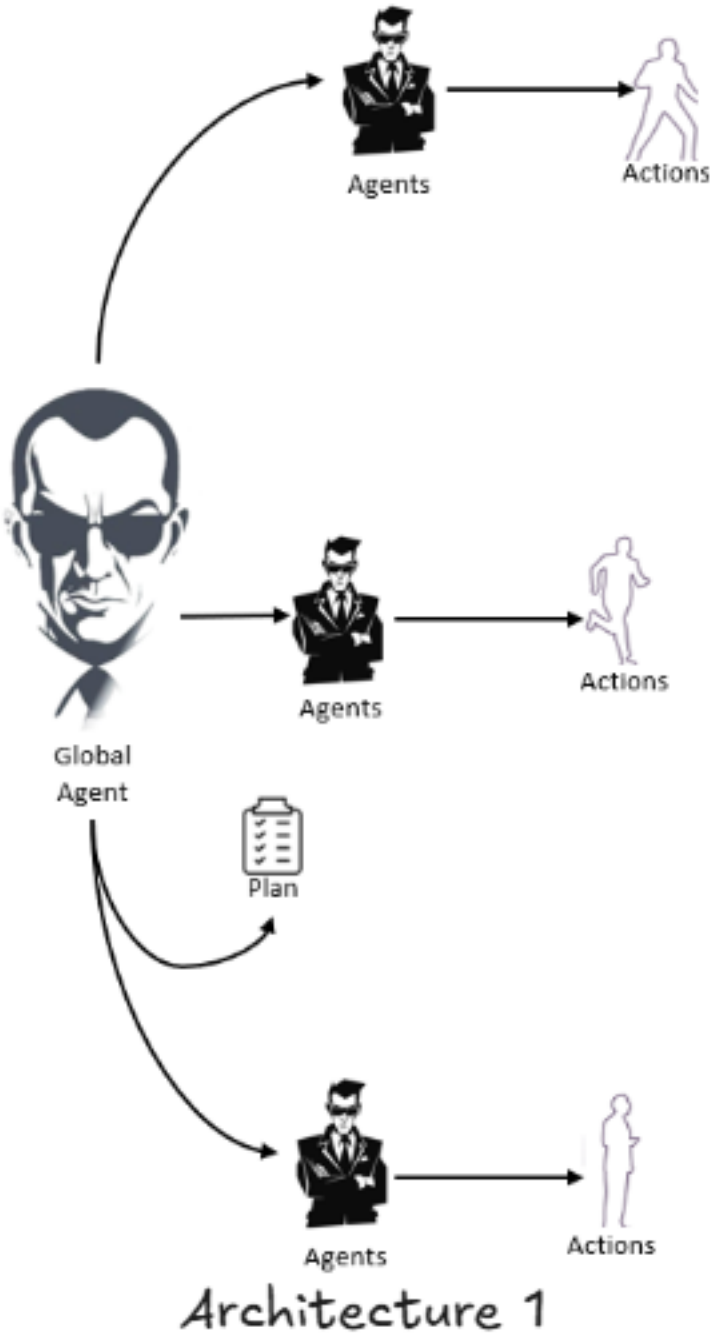
- If a task like `EntityExtraction` fails, the system retries using LangChain's memory or predefined retry logic.



Comparing CODE1, CODE2, and CODE5

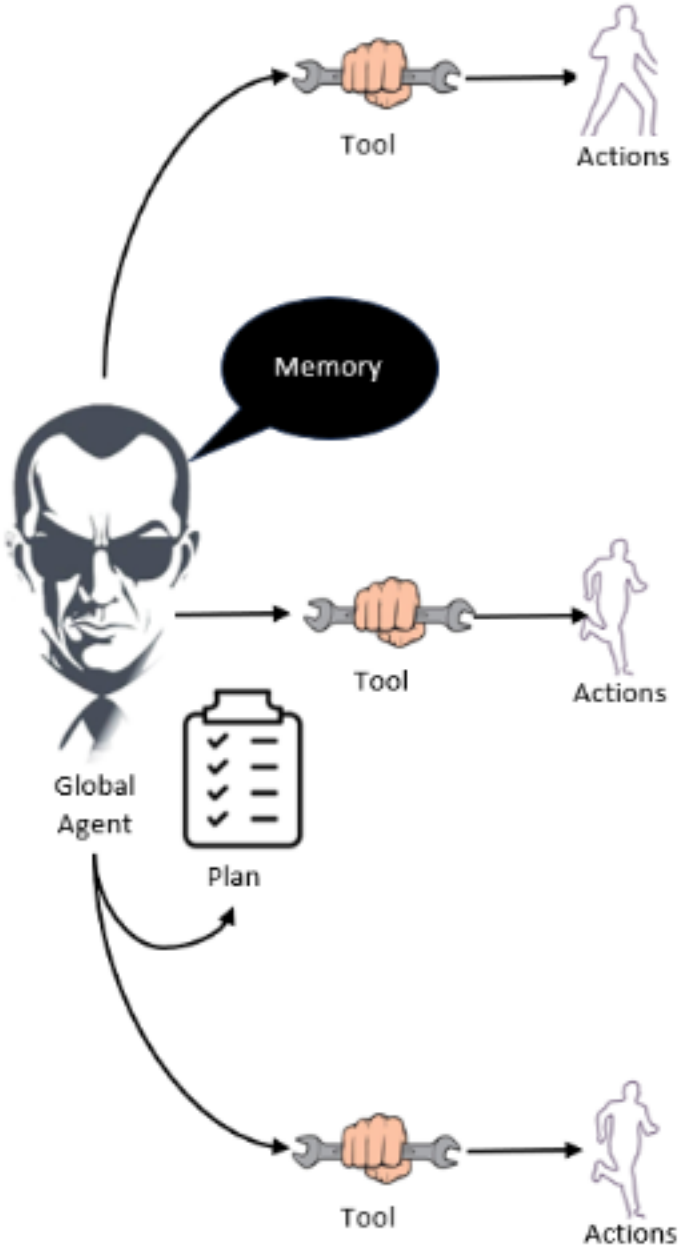
CODE1: A Structured Approach

- **Design**: Relies on **class-based agents** and a predefined workflow orchestrated by a **Global Agent**.
- **Strengths**:
- Transparent and easy to debug.
- Suitable for static workflows with predictable task sequences.
- **Limitations**:
- Lacks dynamic adaptability.
- No integration with LangChain for reasoning-based tool selection.



CODE2: Dynamic Flexibility with LangChain

- **Design**: Uses LangChain's tools and memory to enable a dynamic, reasoning-driven approach.
- **Strengths**:
- Highly flexible, capable of selecting tools dynamically based on inputs and state.
- Employs **custom prompts** for reasoning, leveraging the `ZERO_SHOT_REACT_DESCRIPTION` agent type.
- **Limitations**:
- Heavily dependent on LangChain.
- Slightly abstract, making debugging more complex compared to CODE1.

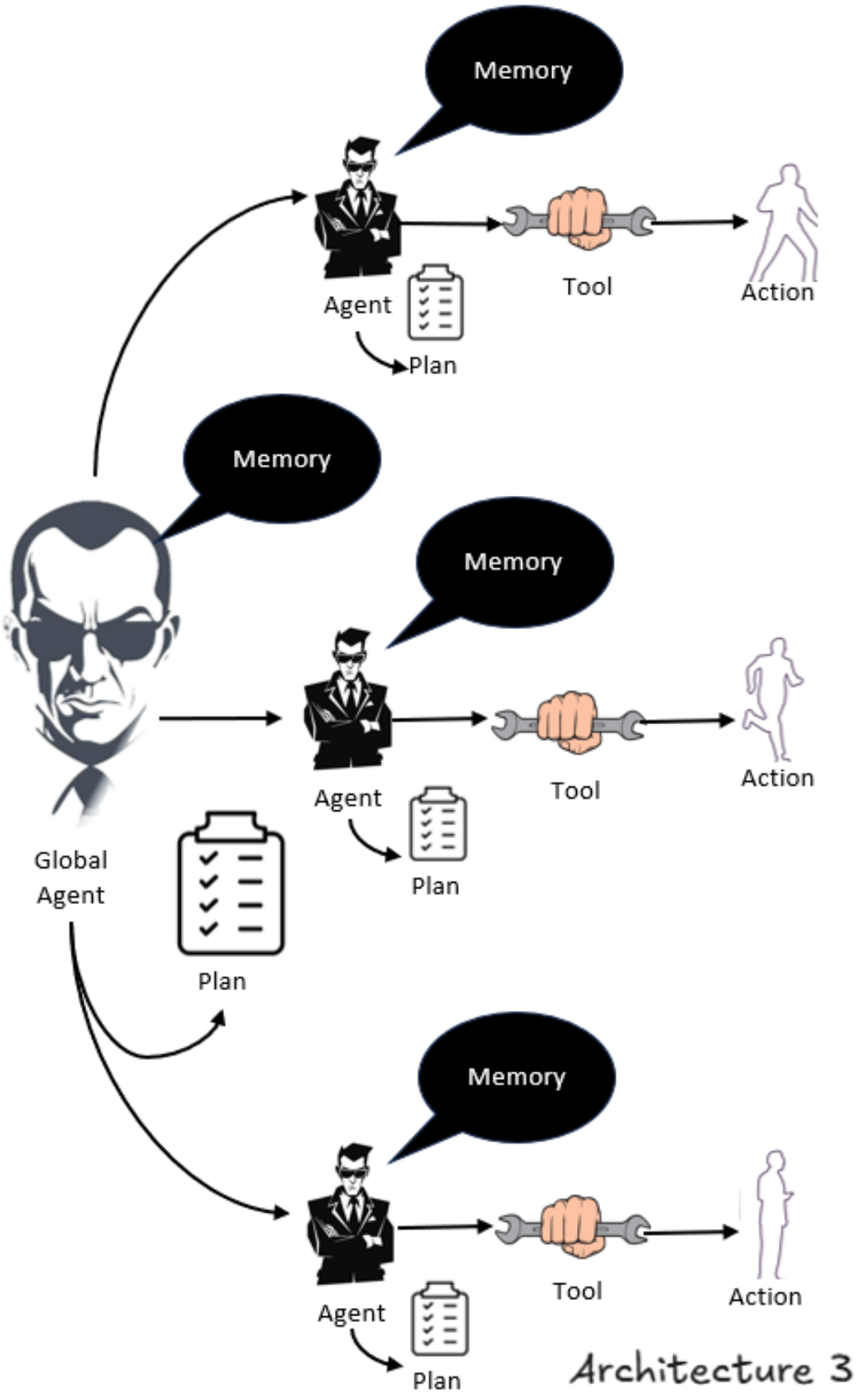


Architecture 2

CODE5: Hybrid Design for Scalability

- **Design**: Combines the structured approach of CODE1 with the dynamic flexibility of CODE2.

- **Strengths**:
- Explicit agents ensure structured workflows, while LangChain tools offer adaptability.
- Suitable for scalable systems requiring both transparency and flexibility.
- **Limitations**:
- Complexity in blending static and dynamic components.


Memory
Agent
Plan
Tool
Action
Memory
Global
Agent
Plan
Memory
Agent
Plan
Tool
Action
Memory
Agent
Plan
Tool
Action
Architecture 3

CODE1:

- Implements all tasks as explicit class-based agents, executed sequentially via a predefined workflow.
- Agents directly update the shared `GraphState`.

CODE2:

- All tasks are converted into **LangChain tools**, invoked dynamically using the `ZERO_SHOT_REACT_DESCRIPTION` reasoning framework.
- Relies on memory for maintaining state and seamless tool invocation.

CODE5:

- Uses explicit agents for workflow orchestration while relying on LangChain tools for task execution.
- Combines structured workflows from CODE1 and dynamic adaptability from CODE2.

This table highlights the evolution from the static, structured approach of **CODE1** to the fully dynamic, tool-centric implementation in **CODE2**, and finally, the hybrid model in **CODE5**.

Task	CODE1 (Agent)	CODE2 (Tool)	CODE5 (Agent)
Column Name Parsing	`ColumnNameAgent`	`ColumnNameExtraction`	`execute_column_name_agent`
Chain Creation	`ChainCreationAgent`	`ChainCreation`	`execute_chain_creation_agent`
Entity Extraction	`EntityExtractionAgent`	`EntityExtraction`	`execute_entity_extraction_agent`
Data Combination	`DataCombinationAgent`	`DataCombination`	`execute_data_combination_agent`
Database Storage	`DatabaseAgent`	Handled via Pandas `.to_sql()` for `merged_data`	`execute_database_agent`
Metadata Extraction	`MetadataExtractionAgent`	`MetadataExtraction`	`execute_metadata_extraction_agent`

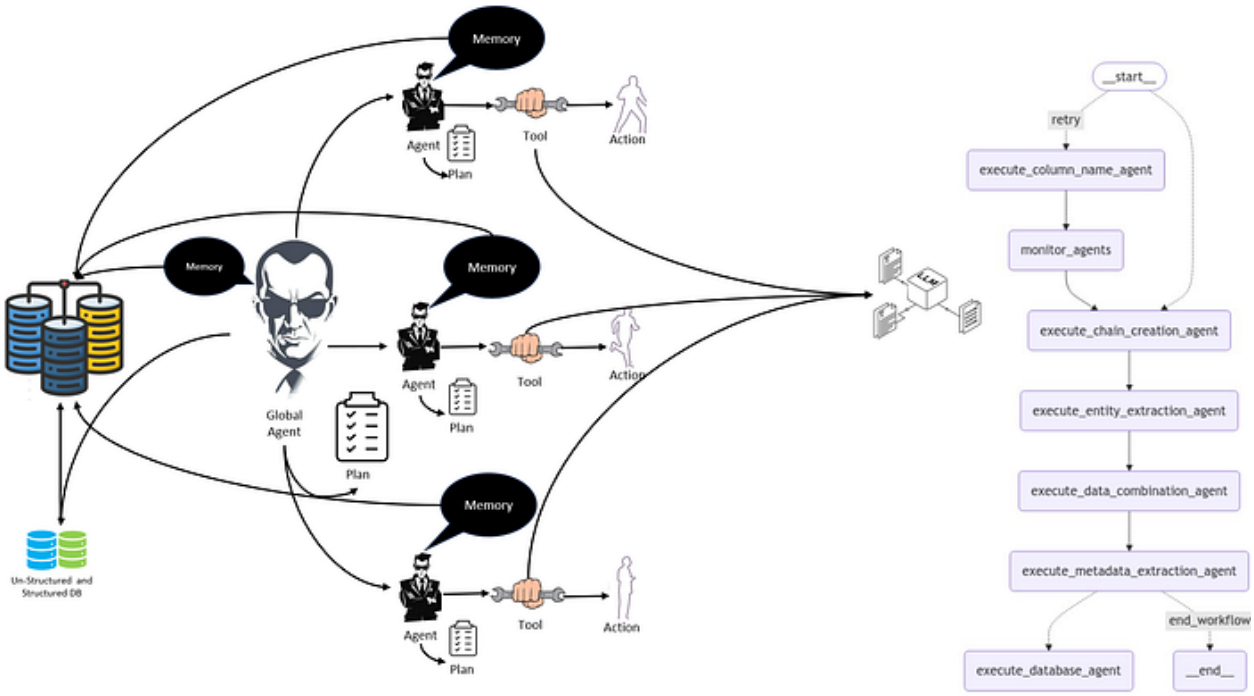
Converting Class-Based Agents to LangChain Tools

Class-based agents, as seen in structured systems like **CODE1**, are explicitly defined entities that perform specific tasks sequentially. They provide clear, modular logic for workflows, making them ideal for static systems with predictable task flows. However, this approach requires manual updates to state and explicit orchestration, limiting adaptability. Converting these agents into LangChain tools involves modularizing their functionality and leveraging LangChain's dynamic execution framework. Each task becomes a callable tool with a defined input-output schema, enabling seamless invocation during runtime. For example, a `ColumnNameAgent` can be converted into a `ColumnNameExtraction` tool that takes column descriptors as input and returns a structured dictionary.

LangChain tools add flexibility by integrating with **reasoning agents** like `ZERO_SHOT_REACT_DESCRIPTION`, which dynamically select and execute tools based on input context and state. These reasoning agents replace the need for hardcoded workflows, enabling dynamic planning and execution. Additionally, LangChain's memory (e.g., `StatefulMemory`) simplifies state management, ensuring persistence and consistency without explicit updates. The resulting system, as seen in **CODE2** and **CODE5**, balances modularity and adaptability. By transitioning from class-based agents to LangChain tools and reasoning agents, workflows become more scalable and robust, capable of handling varied and complex inputs with minimal manual intervention.

Roles of Structured, Unstructured, and Vector Databases in an Agentic AI System

In the context of the diagram and workflow, each type of database plays a distinct but interconnected role:



End to End implemenation with LLM

1. Structured Data

Definition:

- **Structured data** is highly organized and stored in tabular formats such as relational databases (e.g., PostgreSQL, MySQL). Examples include customer records, transaction logs, and inventory tables.

Role:

- **Query and Retrieval**: The **Global Agent** accesses structured databases to retrieve pre-organized data for tasks like matching, aggregation, or reporting.

- **Baseline Knowledge**: Acts as the foundational layer for deterministic queries like “Which customer purchased a specific product?” or “What is the total sales volume?”
- **Integration**: This data can feed directly into downstream tasks like merging with extracted entities or being used as reference data during LLM processing.

Example Use:

- A structured database might store customer purchase histories (`CustomerName`, `PurchaseDate`), which are retrieved and combined with extracted unstructured data for a comprehensive analysis.

2. Unstructured Data

Definition:

- **Unstructured data** includes textual, image, or audio data stored in document-based or NoSQL databases like Elasticsearch, MongoDB, or file repositories. Examples include customer reviews, emails, and social media posts.

Role:

- **Input for LLM**: Acts as raw input for the **LLM**, which processes the data for tasks like summarization, Named Entity Recognition (NER), or sentiment analysis.
- **Entity Extraction**: The extracted entities (e.g., names, dates, or sentiments) can be structured and merged with the structured database for enriched insights.
- **Search and Retrieval**: Provides full-text search capabilities to locate specific documents or phrases that are contextually relevant for the task.

Example Use:

- A database of customer reviews is fed into the **LLM**, which extracts structured entities like `CustomerName` and `Sentiment` for further processing.

3. Vector Databases

Definition:

- A **Vector Database** (e.g., Pinecone, Weaviate, or Milvus) stores vectorized representations (embeddings) of data, enabling similarity searches and semantic queries. Examples include embeddings of product descriptions, customer reviews, or FAQ documents.

Role:

- **Memory for Context Retrieval**: Works as an external memory, allowing agents to retrieve relevant context by performing similarity searches on vectorized data.
- **Semantic Matching**: Facilitates advanced queries like “Find reviews similar to this one” or “Retrieve similar transactions for this customer.”
- **Integration**: Often complements unstructured data by storing its embeddings, enabling quick lookups during reasoning tasks.

Example Use:

- Customer reviews are vectorized, stored in a vector database, and retrieved during LLM queries to provide contextual information for a specific customer or sentiment pattern.

Comparing Roles

TypeStoragePrimary RoleExampleStructured Data Relational Databases (SQL)Query precise, pre-organized information.Retrieve all purchases by a customer from `Customer` and `Orders` tables for analysis.**Unstructured Data**Document-based (NoSQL, Text)Provide raw inputs for LLM or full-text search capabilities.Analyze customer reviews stored in a NoSQL database to extract insights like sentiment or named entities.**Vector Data**Vectorized Embedding StorageSemantic retrieval and similarity searches.Retrieve reviews with similar sentiment or tone by performing a cosine similarity search on vector embeddings stored in the vector database.

Type	Storage	Primary Role	Example
Structured Data	Relational Databases (SQL)	Query precise, pre-organized information.	Retrieve all purchases by a customer from `Customer` and `Orders` tables for analysis.
Unstructured Data	Document-based (NoSQL, Text)	Provide raw inputs for LLM or full-text search capabilities.	Analyze customer reviews stored in a NoSQL database to extract insights like sentiment or named entities.
Vector Data	Vectorized Embedding Storage	Semantic retrieval and similarity searches.	Retrieve reviews with similar sentiment or tone by performing a cosine similarity search on vector embeddings stored in the vector database.

How They Work Together

Input:

- **Structured Data**: Provides the core reference or relational data.

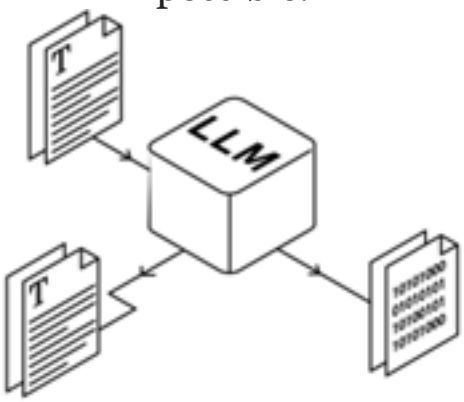
- **Unstructured Data**: Supplies rich, free-form content that complements structured data.

Processing:

- **Unstructured Data** is passed through the **LLM** for feature extraction or semantic understanding.
- Extracted entities are merged with structured data for enriched datasets.

Retrieval:

- **Vector Database** enables advanced retrieval by storing embeddings, making contextual and similarity-based searches possible.



Role of LLMs in Architectures 1, 2, and 3 (formerly CODE1, CODE2, CODE5)

Last but not the least, Large Language Models (LLMs) like L**lama** play a pivotal role in enabling reasoning, adaptability, and automation across Architectures 1, 2, and 3. Their integration enhances the ability to process natural language inputs, extract insights, and make decisions dynamically.

Architecture 1 (CODE1):

- The LLM is used explicitly within agents like `ChainCreationAgent` to set up a Named Entity Recognition (NER) pipeline.
- Agents rely on predefined prompts and workflows to invoke the LLM for extracting entities from unstructured data such as review texts.
- While the LLM performs well in task execution, the integration is rigid, with limited adaptability to changing contexts.

Architecture 2 (CODE2):

- The LLM is leveraged dynamically through LangChain tools like `ChainCreation` and `EntityExtraction`.
- Using `ZERO_SHOT_REACT_DESCRIPTION`, the LLM reasons about the task and decides which tool to invoke, adapting to various inputs.

- This approach maximizes flexibility and allows the system to handle unforeseen scenarios without predefined workflows.

Architecture 3 (CODE5):

- Combines explicit agents with tools powered by LLMs, achieving a hybrid model.
- LLMs are dynamically invoked by tools and explicitly integrated into agents for structured workflows, ensuring scalability and flexibility.

Across all architectures, LLMs empower the system to process natural language, extract insights, and dynamically adapt to real-world tasks.

Conclusion

Designing robust Agentic AI systems requires balancing structure, adaptability, and scalability to meet the demands of modern workflows. **CODE1**, **CODE2**, and **CODE5** highlight different approaches to implementing these systems, showcasing how agents, tools, memory, and planning mechanisms can be tailored to diverse scenarios.

CODE1 demonstrates the value of structured workflows using explicit agents. This approach is transparent, predictable, and easy to debug,

making it ideal for tasks with well-defined sequences. However, its lack of adaptability limits its application in dynamic environments.

CODE2, on the other hand, embraces flexibility and reasoning. By leveraging LangChain's tools and `ZERO_SHOT_REACT_DESCRIPTION`, it dynamically selects tools based on context, enabling the system to adapt to varying inputs and scenarios. This adaptability, while powerful, can make debugging more complex due to its abstracted logic.

CODE5 combines the strengths of both approaches, creating a hybrid system. It uses explicit agents for structured tasks and LangChain tools for dynamic adaptability, providing scalability and transparency in complex workflows.

As Agentic AI continues to evolve, integrating structured planning, dynamic reasoning, and efficient state management will remain key. Systems like **CODE5** offer a glimpse into how these principles can work together to create intelligent, scalable, and future-ready AI solutions.