

Enterprise Multi-Agent System - Architecture Baseline

Project Overview

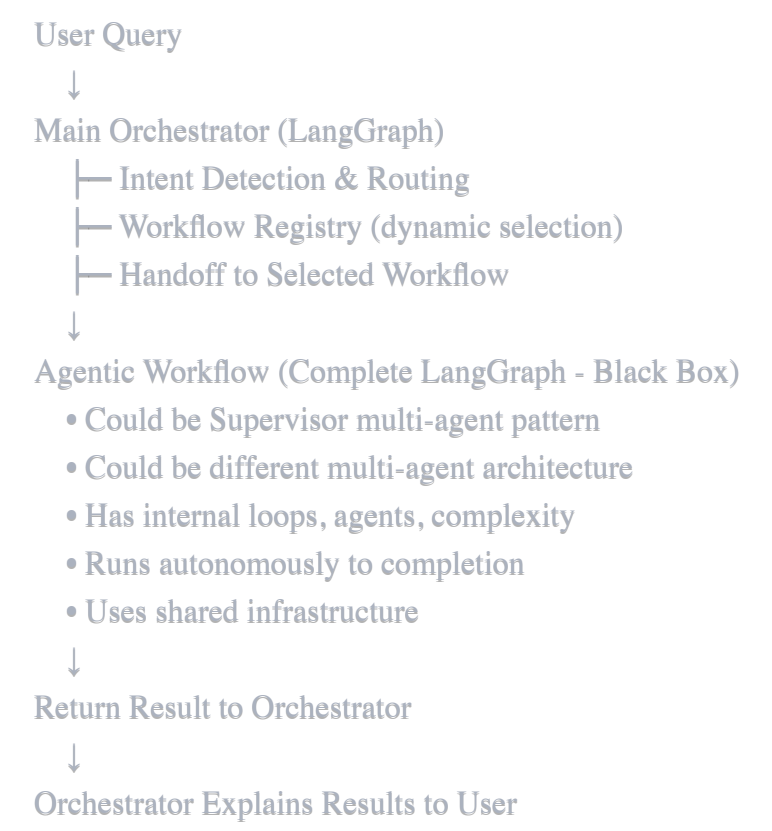
Building a scalable, modular agentic AI system where specialized multi-agent workflows can be dynamically composed and reused across various enterprise data engineering tasks.

System Architecture

Core Concept

A **Main Orchestrator Agent** that dynamically routes user queries to specialized **Agentic Workflows**, where each workflow is itself a complete multi-agent system.

Key Architectural Pattern



Critical Understanding: What is a "Workflow"

A workflow is **NOT** a simple function or chain.

A workflow is a **complete, compiled LangGraph** that:

- Contains multiple agents (could be supervisor pattern or other architectures)
- Has its own internal state management
- Performs complex, iterative operations

- May loop multiple times based on internal logic
- Runs to completion independently
- Returns structured result to orchestrator

Key Characteristic: Each workflow is a self-contained multi-agent system that can be developed, tested, and deployed independently.

Shared Infrastructure

Core Services (Used by ALL workflows and orchestrator)

2. State Management

- Categorized registry of available workflows
- Categories: SCATTERIN (research/analysis), GATHERIN (data collection), TRANSFORM (conversion), SYSTEM (integrations), VALIDATION (quality checks)
- Dynamic workflow discovery and selection
- Hot-reload capability for development iteration

3. State Management

- DynamoDB: Session data, conversation history
- Postgres: Structured metadata, workflow execution logs
- Multi-backend coordination

4. Artifact Manager

- S3: Large artifacts (parsed files, generated code, reports)
- Postgres: Artifact metadata and indexing
- Used for context persistence between workflow iterations

5. Session Manager

- User session tracking
- Conversation history management
- Checkpoint/resume capability

6. Context Handlers

- Manages context flow between iterations
 - Determines what gets saved, loaded, passed forward
 - Critical for iterative workflows
-

External Connectors & Tools

- **Database Connectors:** Abstract database access layer for multiple database types
 - **Compute Executors:** Serverless and containerized compute orchestration
 - **Integration Adapters:** Connectors for external enterprise systems
 - **Knowledge Bases:** RAG systems, document stores, vector databases
-

Key Design Principles

1. Dynamic Subgraph Injection

Workflows are pre-compiled LangGraphs stored in registry. Orchestrator dynamically selects and invokes appropriate workflow based on intent.

2. Black Box Handoff

Main orchestrator doesn't know internal workings of workflows. Clean separation:

- Orchestrator: Intent → Route → Explain

- Workflow: Execute → Return Result

3. Artifact-Centric Iteration

Complex workflows use artifact storage to maintain context across iterations:

- Save intermediate state after each iteration
- Load accumulated context in subsequent iterations
- Continue processing with full historical context
- Exit loop based on completion criteria
- Enables workflows to handle tasks that span multiple execution cycles

4. Modular Development

Multiple teams can develop different workflows independently:

- Each workflow is self-contained
- Shared infrastructure (artifact store, session manager) is consistent
- Simple registration process to add new workflows
- No changes to orchestrator needed

5. Category-Based Organization

Workflows organized by functional purpose:

- **SCATTERIN**: Exploratory, iterative analysis and research tasks
 - **GATHERIN**: Data collection, extraction, aggregation
 - **TRANSFORM**: Conversion, migration, restructuring
 - **SYSTEM**: Database connectors, API integrations, external system interfaces
 - **VALIDATION**: Quality checks, validation rules, compliance verification
-

Architecture Goals

Reusability

1. **Graph Reusability**: Compiled workflow graphs can be invoked from multiple contexts
2. **Code Module Reusability**: Shared infrastructure components used across all workflows
3. **Pattern Reusability**: Common patterns (iteration, artifact storage, state management) abstracted into reusable components
4. **Agent Reusability**: Individual agents can be composed into different workflow architectures

Modularity

1. Enable independent development of workflows by different teams
2. Isolate workflow complexity from orchestrator
3. Support multiple simultaneous workflow executions
4. Allow workflow versioning and A/B testing

Abstraction

1. Abstract away complex state management across distributed systems
2. Hide artifact storage complexity from workflow developers
3. Provide clean interfaces for workflow registration and discovery
4. Separate intent detection from workflow execution

Scalability

1. Support hot-pluggable workflow architecture
2. Handle complex, iterative analysis patterns
3. Coordinate state across multiple backend systems
4. Scale horizontally as workflow count increases

Developer Experience

1. Simple workflow registration (config-based, minimal boilerplate)
2. Isolated development and testing environments
3. Shared infrastructure handles cross-cutting concerns
4. Clear patterns and base classes for common tasks
5. Fast iteration cycle (develop → test → deploy)

Technology Stack

- **Framework:** LangChain + LangGraph for agentic workflow orchestration
- **LLM:** Claude (Anthropic) for agent reasoning
- **State Storage:** NoSQL (sessions, events) + SQL (structured metadata)
- **Artifact Storage:** Object storage for large files and intermediate results
- **Compute:** Serverless functions + container orchestration
- **Language:** Python 3.10+

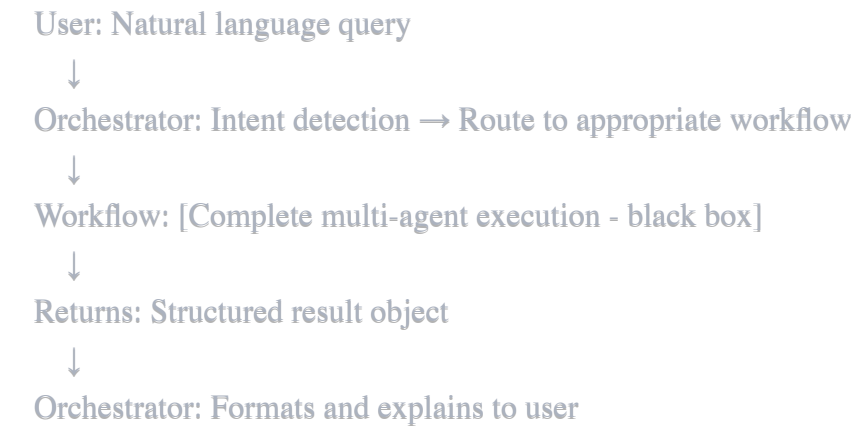
Critical Patterns

Pattern 1: Iterative Analysis with Artifact Storage



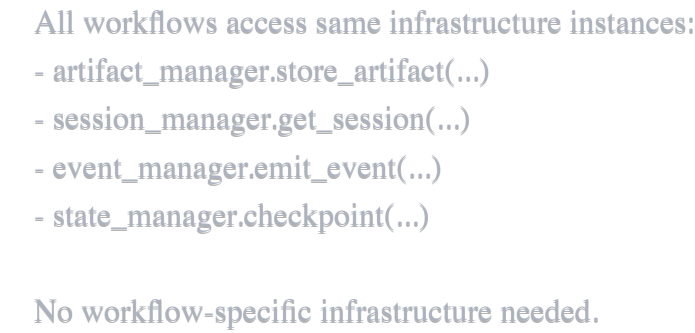
Purpose: Enable workflows to handle tasks requiring multiple passes over data, building context incrementally.

Pattern 2: Workflow Handoff



Purpose: Clean separation between routing logic and workflow execution. Orchestrator remains simple while workflows handle complexity.

Pattern 3: Shared Infrastructure Access



Purpose: Consistency across workflows, reduced boilerplate, centralized management of cross-cutting concerns.

Core Architectural Concepts

Graph Reusability

Principle: Compiled LangGraph workflows are reusable components.

- Workflows compiled once, invoked many times
- Same workflow graph can be used in different orchestration contexts
- Graphs are stateless - all state passed via input/output
- Enables composition: workflows can invoke other workflows
- Facilitates testing: graphs testable in isolation

Code Module Abstraction

Principle: Complex functionality abstracted into shared modules.

Infrastructure Modules:

- State management abstraction (multi-backend coordination)
- Artifact storage abstraction (lifecycle management)
- Session management abstraction (user context tracking)
- Event management abstraction (observability)

Workflow Modules:

- Base workflow classes (standardized interface)
- Common agent patterns (supervisor, loop, router)
- State transformation utilities
- Result formatting helpers

Benefits:

- Reduced code duplication
- Consistent behavior across workflows
- Easier maintenance and updates
- Lower learning curve for new developers

Component Isolation

Principle: Components are loosely coupled and independently deployable.

Boundaries:

- Orchestrator ↔ Workflows (clean handoff interface)
- Workflows ↔ Infrastructure (dependency injection)
- Agents ↔ Tools (tool abstraction layer)

Characteristics:

- Workflows don't know about each other
- Orchestrator doesn't know workflow internals
- Infrastructure services are swappable
- Failure isolation prevents cascades

Interface Standardization

Principle: All components implement well-defined contracts.

Key Interfaces:

- BaseWorkflow: All workflows implement this
- ArtifactStore: Standard storage operations
- StateManager: Standard state operations
- SessionManager: Standard session operations

Contract Elements:

- Input schema definition
 - Output schema definition
 - Metadata specification
 - Error handling protocols
-

Implementation Approach

Registry-Based Workflow Management

- Workflows registered via configuration files
- Auto-discovery at system startup
- Each workflow implements standardized interface (BaseWorkflow)
- Orchestrator dynamically selects from registry based on intent
- Support for workflow priority and fallback chains

State Transformation Layer

- Orchestrator state schema ≠ Workflow state schemas
- Transformation happens at handoff boundary
- Workflows define input preparation and output formatting
- Clean contracts between components

Graph Composition Architecture

- Workflows are compiled LangGraph objects
- Main orchestrator invokes workflows as callable functions
- Workflows run as black boxes from orchestrator perspective
- Results flow back through standardized interface
- Support for nested subgraph composition

Event-Driven Observability

- Workflows emit events during execution
- Centralized event collection and logging
- Real-time monitoring across all workflow executions
- Debug and audit trails for compliance

Abstraction Layers

Layer 1: Infrastructure Abstraction

- Database access (abstract over specific DB types)
- Storage access (abstract over specific storage backends)
- Compute orchestration (abstract over execution environments)

Layer 2: State Management Abstraction




- Session lifecycle (creation, retrieval, cleanup)
- Artifact lifecycle (store, retrieve, version, cleanup)
- Checkpoint management (save, restore, time-travel)

Layer 3: Workflow Abstraction





- Base workflow interface
 - Standard input/output contracts
 - Metadata specification
 - Common agent patterns
-

Architectural Success Criteria





Modularity

-  Workflows can be developed in complete isolation
-  Adding new workflow requires only config change, no orchestrator modification
-  Workflows can be versioned and deployed independently





Reusability

-  Compiled graphs can be reused across different contexts
-  Shared infrastructure components are framework-agnostic
-  Common patterns (iteration, artifact storage) are abstracted into base classes
-  Individual agents can be composed into multiple workflows





Scalability

-  System handles multiple concurrent workflow executions
-  State persists reliably across sessions and restarts
-  Horizontal scaling possible as workflow count grows
-  Resource isolation prevents workflow failures from cascading

Abstraction Quality

-  Infrastructure complexity hidden from workflow developers
-  Clean separation between routing and execution logic
-  Minimal boilerplate for new workflows
-  Clear contracts and interfaces between components

Observability

-  Complete audit trail of all operations
-  Real-time monitoring of workflow executions
-  Debug capability with state inspection and time-travel
-  Performance metrics and bottleneck identification