Project - High Level Design

On

Multi-Agent Manufacturing System

Course Name: Agentic AI

Institution Name: Medicaps University – Datagami Skill Based Course

Student Name(s) & Enrolment Number(s):

Sr no	Student Name	Enrolment Number
01	Om Tiwari	EN22CS301669
02	Paridhi Shirwalkar	EN22CS301684
03	Nitesh Chourasiya	EN22CS301660
04	Mradul Jain	EN22CS301616

Group Name: Group 01D6

Project Number: AAI-12

Industry Mentor Name:

University Mentor Name: Nishant Shrivastav

Academic Year: 2025-2026

Table of Contents

1. Introduction.
   1.1. Scope of the document.
   1.2. Intended Audience
   1.3. System overview.
2. System Design.
   2.1. Application Design
   2.2. Process Flow.
   2.3. Information Flow.
   2.4. Components Design
   2.5. Key Design Considerations
   2.6. API Catalogue.
3. Data Design.
   3.1. Data Model
   3.2. Data Access Mechanism
   3.3. Data Retention Policies
   3.4. Data Migration
4. Interfaces
5. State and Session Management
6. Caching
7. Non-Functional Requirements
   7.1. Security Aspects
   7.2. Performance Aspects
8. References

1. Introduction

1.1 Scope of the Document

This document defines the low-level design for a Multi-Agent Manufacturing System that automates supplier sourcing and generates a structured supplier comparison report. The system uses a collaborative, role-based agent architecture with a strict handoff protocol to ensure predictable outputs.

The scope includes:

- System workflow and execution lifecycle
- Component-level architecture and responsibilities
- Data design and schemas for inter-agent handoffs
- Interfaces and APIs for UI and internal modules
- State/session handling, caching, and persistence
- Non-functional requirements (performance, security, reliability)
- Error handling, retry logic, and validation strategy

Out of scope:

- Production-grade deployment and autoscaling setup
- Paid integrations like enterprise vendor databases
- Advanced RAG pipelines (optional future enhancement)

1.2 Intended Audience

This document is intended for:

- Developers implementing the system (Python + CrewAI)
- Reviewers evaluating the architecture (LLD review)
- Testers validating data integrity and workflow states
- Maintainers extending agents/tools or adding new workflows

1.3 System Overview

The system is composed of two specialized agents orchestrated through a central controller:

- **Researcher Agent:** collects supplier information based on a user query and produces a raw dataset.
- **Writer Agent:** transforms the raw dataset into clean, structured supplier records and generates a formatted comparison report.

- **Orchestrator:** manages the end-to-end flow, validation, retries, state persistence, and artifact creation.
- **UI Layer (Streamlit):** provides input capture, model/API key handling, execution trigger, and output visualization/download.

Key design principles:

- Strict JSON-based handoff artifacts between agents
- Schema validation at each stage to prevent garbage-in → garbage-out
- Session-based runs with deterministic storage paths
- Defensive orchestration with controlled retries and fallbacks

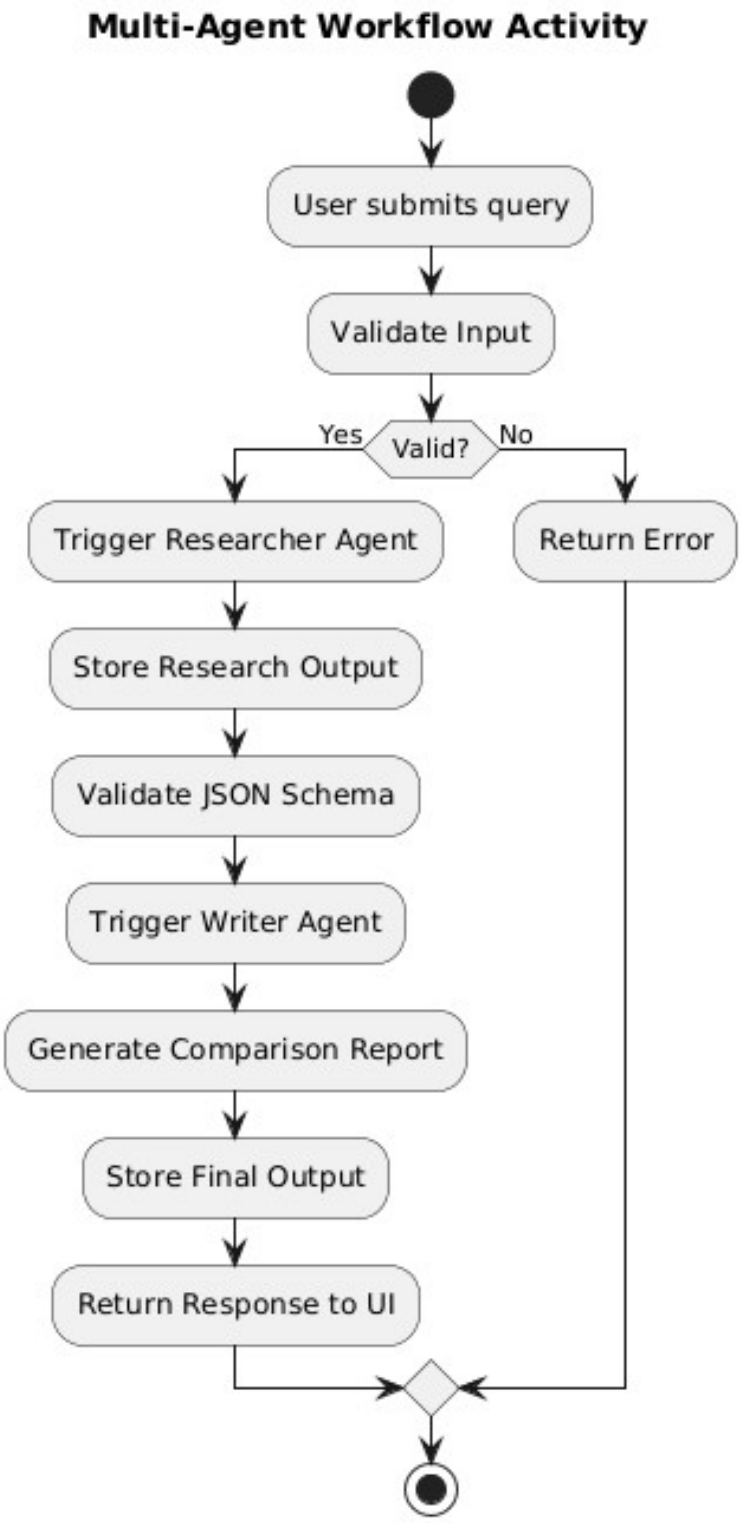
2. System Design

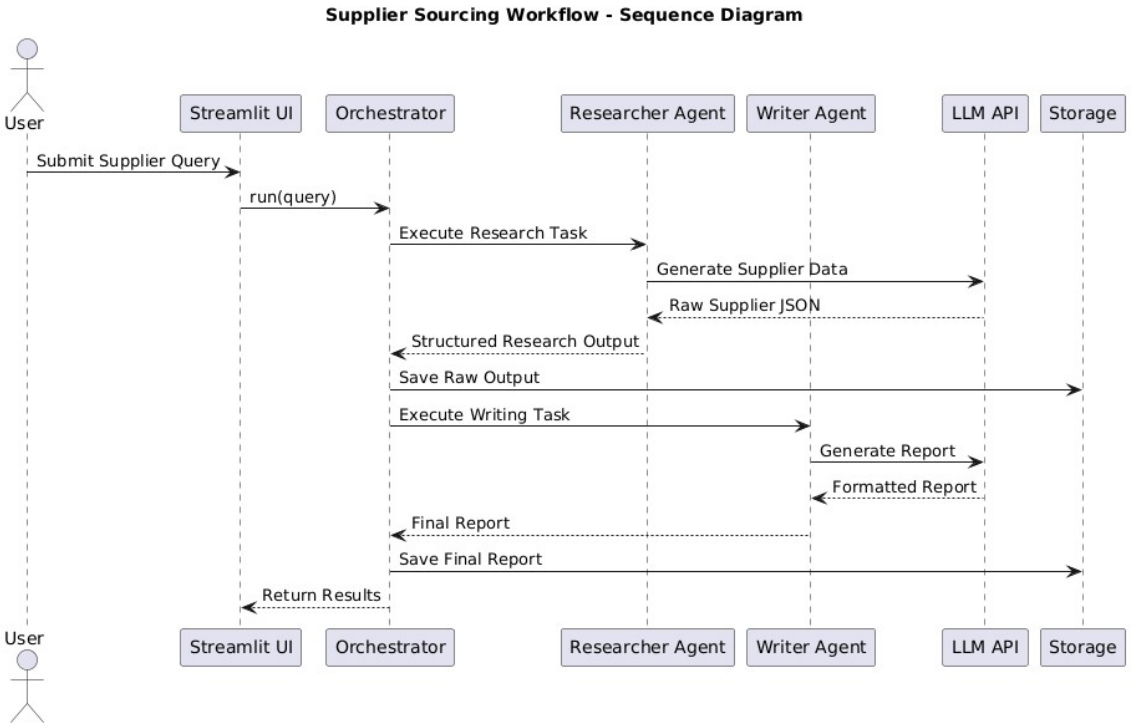
2.1 Application Design

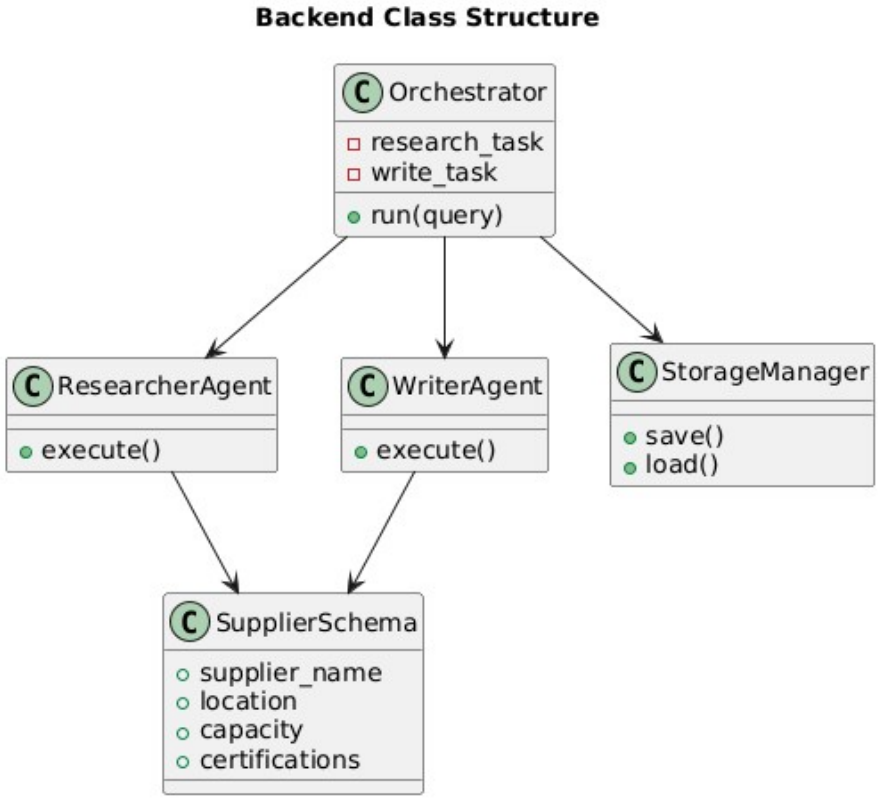
Architecture style: Layered + Orchestrated multi-agent pipeline.

Layers

1. Presentation Layer (Streamlit UI)
    - Input form (manufacturing query)
    - Model selection + API key handling
    - Progress logging and output rendering
2. Application Layer (Orchestrator + Workflow Engine)
    - Creates a unique run/session
    - Executes Research crew then Writer crew
    - Validates outputs, stores artifacts, marks workflow state
3. Domain Layer (Agent Logic + Tasks)
    - Agent definitions and role prompts
    - Task templates and output format constraints
4. Data Layer (Storage + Schemas)
    - JSON schemas for handoff
    - File-based persistence and caching


Multi-Agent Workflow Activity
User submits query
Validate Input
Valid?
Yes
No
Trigger Researcher Agent
Return Error
Store Research Output
Validate JSON Schema
Trigger Writer Agent
Generate Comparison Report
Store Final Output
Return Response to UI


Supplier Sourcing Workflow - Sequence Diagram
User
Streamlit UI
Orchestrator
Researcher Agent
Writer Agent
LLM API
Storage
Submit Supplier Query
run(query)
Execute Research Task
Generate Supplier Data
Raw Supplier JSON
Structured Research Output
Save Raw Output
Execute Writing Task
Generate Report
Formatted Report
Final Report
Save Final Report
Return Results
User
Streamlit UI
Orchestrator
Researcher Agent
Writer Agent
LLM API
Storage


Backend Class Structure
Orchestrator
research_task
write_task
run(query)
ResearcherAgent
execute()
WriterAgent
execute()
StorageManager
save()
load()
SupplierSchema
supplier_name
location
capacity
certifications

2.2 Process Flow

Primary Flow (Happy Path)

1. User submits supplier sourcing query via UI
2. Orchestrator creates run_id and initializes workflow state
3. Researcher Agent runs and returns Raw Supplier Dataset (JSON)
4. Orchestrator validates raw JSON (schema + minimal quality checks)
5. Writer Agent consumes raw dataset and returns:
    - Structured Supplier Records (JSON)
    - Supplier Comparison Report (Markdown/HTML/Text)
6. Orchestrator stores outputs, finalizes state, returns result to UI
7. UI renders report and provides downloadable artifacts

Failure Handling (High-Level)

- If Researcher fails → retry with backoff (configurable)
- If schema invalid → request agent correction (limited attempts) or fail run with detailed error artifact
- If Writer fails → retry; if still fails, store raw research artifact for debugging and return partial output

2.3 Information Flow

Input → Output data movement

- UI sends: {query, api_key, model}
- Orchestrator sends to research crew: {query, run_id}
- Researcher returns: raw_suppliers.json (untrusted)
- Orchestrator validates + stores: raw_suppliers_validated.json
- Writer consumes validated raw: generates:
    - structured_suppliers.json
    - report.md or report.txt
- Orchestrator returns a summary object to UI:
    - run_id
    - artifact paths
    - report preview snippet

- status + timestamps

2.4 Components Design

2.4.1 Streamlit Frontend

Responsibilities:

- Collect query text (supplier sourcing request)
- Allow selection of model (Gemini, etc.)
- Execute orchestrator run
- Render results: report, tables, JSON preview
- Provide download buttons for artifacts

Key UI states:

- Idle
- Running
- Completed
- Failed (show error + debug artifact link)

2.4.2 Orchestrator (Core Controller)

Responsibilities:

- Create run/session ID
- Prepare storage directories
- Execute agent crews sequentially
- Validate and normalize outputs
- Save artifacts and workflow metadata
- Return structured results to UI

Important orchestration decisions:

- Sequential is preferred for deterministic handoff correctness
- Strict schema validation occurs between agents
- Adds guardrails: max retries, max tokens, timeouts (where supported)

2.4.3 Researcher Agent

Responsibilities:

- Understand the sourcing request
- Identify supplier candidates

- Collect key supplier attributes (name, location, products, certifications, lead time, MOQ, pricing ranges if possible, contact channels)
- Produce a raw dataset that is complete enough for synthesis

Output format:

- JSON list of supplier objects
- Includes source references (links or notes)
- Includes confidence/quality notes per supplier if applicable

2.4.4 Writer Agent

Responsibilities:

- Clean and standardize supplier data
- Fill missing fields with "unknown" or null rather than hallucinating
- Rank suppliers based on scoring logic (configurable)
- Produce:
  - structured normalized JSON
  - human-readable comparison report with findings, risks, and gaps

Report sections:

- Executive summary
- Comparison table
- Ranking explanation
- Risks/gaps (missing certifications, unclear lead times, weak contact info)
- Next steps and recommended outreach questions

2.4.5 Storage Layer

Responsibilities:

- Write artifacts to disk under run-specific folder
- Maintain workflow state file
- Optional caching for identical queries

Folder example:

```
artifacts/
 run_20260219_101530/
  input.json
  state.json
```

raw_suppliers.json

raw_suppliers_validated.json

structured_suppliers.json

report.md

logs.jsonl

2.5 Key Design Considerations

2.5.1 Deterministic Outputs

- Force agents to return JSON in strict schema
- Validate using Pydantic/JSONSchema
- Normalize fields (string trimming, list coercion, default values)

2.5.2 Preventing Hallucinated Data

- Writer must not fabricate missing supplier details
- Use explicit rules:
  - If unknown: set "unknown"
  - If not found: set null
  - Add missing_fields list per supplier

2.5.3 Observability

- Store run logs with timestamps
- Capture agent prompts and responses (optional, sanitized)
- Maintain state.json transitions for audit

2.5.4 Extensibility

- Add new agents later (e.g., Negotiator Agent, Compliance Agent)
- Add tools (web search, vector DB, document ingestion)
- Add new workflows (RFQ generation, supplier onboarding checklist)

2.6 API Catalogue

2.6.1 Frontend → Orchestrator (Internal Call)

Function: run(query: str) -> dict

Request:

{

"query": "Find aluminum die casting suppliers in India for automotive parts"
}
Response:
{
"run_id": "run_20260219_101530",
"status": "completed",
"artifacts": {
"raw": "artifacts/run_.../raw_suppliers.json",
"structured": "artifacts/run_.../structured_suppliers.json",
"report": "artifacts/run_.../report.md"
},
"summary": "Top suppliers ranked by certifications + lead time..."
}

2.6.2 Storage API (Internal)

- init_run() -> run_id
- save_json(run_id, filename, data)
- load_json(run_id, filename)
- save_text(run_id, filename, text)
- update_state(run_id, patch)

3. Data Design

3.1 Data Model

3.1.1 Raw Supplier (Researcher Output)

Core fields:

- supplier_name (string)
- country / city (string)
- capabilities (list of strings)
- industries_served (list)
- certifications (list)
- lead_time (string)

- moq (string/number)
- pricing_info (string, optional)
- contact (object: email/phone/website)
- sources (list: url or notes)

3.1.2 Structured Supplier (Writer Output)

Adds:

- normalized fields (standard formats)
- scoring fields (quality_score, cost_score, lead_time_score)
- rank (integer)
- gaps (list)
- confidence (low/medium/high)

3.2 Data Access Mechanism

- File-based storage (JSON + markdown) for simplicity and portability
- Each run is isolated by run_id
- Reads happen only within the run directory
- Optional caching layer:
  - Hash of query + model → stored response reuse

3.3 Data Retention Policies

- Default retention: keep artifacts locally for debugging/demo
- Optional cleanup job:
  - Delete runs older than N days
  - Keep only report.md + structured_suppliers.json

Sensitive data considerations:

- API keys should never be written to disk
- If storing prompts/responses, strip secrets and tokens

3.4 Data Migration

If schema changes:

- version your schema: schema_version field in each JSON artifact
- maintain migration scripts:
  - v1 → v2: rename fields, add defaults, convert types

- backward compatibility rules:
    - Writer can accept v1 raw schema and output v2 structured schema

4. Interfaces

4.1 User Interface

- Text area: sourcing query
- Inputs: API key, model name
- Buttons: Run, Reset, Download report/JSON
- Output panels:
    - Status (Running/Done/Failed)
    - Report preview
    - Supplier table

4.2 Internal Module Interfaces

- Agents module exposes: build_researcher(), build_writer()
- Tasks module exposes: research_task(agent), write_task(agent)
- Schemas module exposes: Pydantic models + validators
- Orchestrator exposes: run(query)

5. State and Session Management

5.1 Run Lifecycle

State machine example:

- CREATED
- RESEARCH_RUNNING
- RESEARCH_DONE
- WRITE_RUNNING
- COMPLETED
- FAILED

State file (state.json) fields:

- run_id
- status

- timestamps (created_at, updated_at)
- error (if failed)
- artifacts list

5.2 Session Isolation

- Each run gets a unique directory
- No shared mutable state between runs
- Thread safety: avoid global variables for run data

6. Caching

6.1 Caching Strategy

Goal: reduce cost and speed up repeat queries.

Cache key:

- hash(query + model + prompt_version)

Cache store:

- cache/index.json maps key → artifact path

Cache rules:

- Only cache if run completed successfully
- Invalidate cache if schema_version changes or prompt_version changes

7. Non-Functional Requirements

7.1 Security Aspects

- Never store API keys on disk
- .env included in .gitignore
- Validate and sanitize user input (length limits, safe characters)
- Avoid executing user input as code
- If adding web tools:
  - restrict allowed domains (optional)
  - log sources used

7.2 Performance Aspects

- Target: usable demo experience on a laptop

- Minimize LLM calls:
  - one research run + one writer run (baseline)
  - limit agent iterations
- Implement backoff retries for transient errors (429, network)
- Graceful degradation:
  - return partial results if writer fails but research succeeded

Reliability targets:

- Each run produces a final state.json even if failed
- Errors stored as error.json for debugging

8. References

- CrewAI documentation (agents, tasks, crews, orchestration concepts)
- LLM provider API documentation (Gemini/OpenAI depending on provider)
- Pydantic / JSON Schema validation references
- Streamlit documentation for UI patterns and session handling