ChatGPT

Data Engineering Challenge: Self-Assembling Multi-Agent System with a Knowledge Graph

This challenge defines a complex multi-agent data engineering system orchestrated by a central knowledge graph. In our scenario (e.g. a smart manufacturing pipeline or supply-chain automation), **autonomous agents** (sensors, analysts, planners, etc.) coordinate indirectly via a shared RDF knowledge graph. Agents do not chat directly; instead they read and write structured triples, updating the graph state. The backend is an RDF store (using **RDFLib** in Python or an enterprise triplestore like **GraphDB** [1] [2]). The knowledge graph serves as the single source of truth and a trigger mechanism: new triples (or changed triples) cause agents to spawn or change roles. All reasoning steps are reflected as graph updates (no free-text dialogue between agents).

A **knowledge graph** here means a knowledge base using a graph-structured data model [3]. In practice we will define an ontology of entities like *Agent*, *Task*, *Machine*, *Sensor*, *Status*, etc. Each agent has a URI/node in the graph and edges representing roles, tasks, and data. For example, agents might add triples like `<MachineX> ex:hasStatus "Overheat"`, or `<Task42> ex:assignedTo Agent17`. A *Multi-Agent System* (MAS) is "a computerized system composed of multiple interacting intelligent agents" [4]; our MAS will be **self-organizing** via the knowledge graph. For instance, if a machine sensor reports an anomaly triple, the graph may trigger a maintenance agent to instantiate.

Below we break the design into **6 Work Packages (WP)**. Each WP has clear objectives, data I/O formats, agent-graph interactions, sample test cases, expected outcomes, and hooks for extension. The final deliverable will be a runnable Python codebase (using `rdflib`, `SPARQLWrapper`, etc.) with install instructions (a `requirements.txt` and a startup script). We include **annotated Python snippets**, sample data descriptions, and architectural diagrams. Test scripts verify that agents correctly read/write triples and handle faults. The result should feel like a polished, world-class engineering design ready for executive review.

Work Package 1: Knowledge Graph & Ontology Setup

Objectives: Define the core ontology and initialize the knowledge graph. Establish namespaces (e.g. `ex = Namespace("http://example.com#")`) and key classes (`ex.Agent`, `ex.Task`, `ex.Machine`, `ex.Sensor`, `ex.Status`, etc.). Populate the graph with sample domain data (machines, sensors, baseline tasks, etc.) in RDF (Turtle or JSON-LD). Provide scripts to load this data into the graph.

- **Input:** Sample RDF files (e.g. `ontology.ttl` and `initial_data.ttl`) describing entities. For instance, a TTL file listing machines and sensors:

```
@prefix ex: <http://example.com#> .
ex:MachineA a ex:Machine .
```

```
ex:Sensor1 a ex:TemperatureSensor; ex:attachedTo ex:MachineA .
ex:Task1 a ex:MaintenanceTask; ex:status ex:Pending .
```

- **Output:** An RDF graph (in-memory via RDFLib or loaded into GraphDB) containing the ontology and initial triples. We verify this by querying the graph.

- **Agents & Graph Links:** At this stage, we can simulate a *GraphInitializerAgent* whose job is to load RDF files. For example, in Python with RDFLib:

```python
from rdflib import Graph, Namespace, URIRef, Literal
graph = Graph()
ex = Namespace("http://example.com#")
graph.bind('ex', ex)
# Load data from Turtle file
graph.parse("initial_data.ttl", format="turtle")
# Example: add a static triple
graph.add((ex.MachineA, ex.hasStatus, Literal("Nominal")))
```

  Each triple represents a factual state. In the graph we might also define an `ex:hasRole` property to link agent URIs to roles, e.g. after WP2.

- **Test Cases:** Unit tests check that after loading:

- Query the graph for all `rdf:type ex:Machine` and ensure expected machine URIs appear.

- Query for `ex:status` on tasks and verify initial statuses (e.g. `ex:Pending` exists).
  A simple SPARQL or RDFLib query suffices:

```python
result = graph.query("SELECT ?m WHERE { ?m rdf:type ex:Machine }")
assert URIRef("http://example.com#MachineA") in [row[0] for row in result]
```

- **Expected Outcomes:** The graph now contains a skeleton of the domain knowledge (classes and instances). Agents can subsequently query this graph. For example, an AnalyticsAgent can list all `ex:Machine` nodes to monitor.

- **Extensibility Hooks:** New classes (e.g. `ex.ConveyorBelt`, `ex.HealthMetric`) or predicates can be added to the ontology files. The loader agent can be extended to fetch data from external APIs or CSVs. We encourage providing a Python data loader that can accept different file paths via config.

Figure 1: Example domain ontology schema (public-domain "Aba CM Database Schema"). This stylized schema illustrates how a real-world system's ontology might look – entities like *Agent*, *Task*, *Resource* and relations among them. In our graph, similar relationships (e.g. which agent handles which task, or a sensor attached to a machine) will be represented by RDF triples. The work in WP1 is to define and load such a schema into an RDFLib/GraphDB graph.

Work Package 2: Agent Framework & Data Ingestion

Objectives: Build the Python agent framework. Define base classes for agents and implement core data ingestion agents. Each agent runs as a module or class with access to the shared graph. Agents should be able to query the graph state and perform updates. We also create initial "worker" agents (e.g. **SensorAgent**, **DataProcessorAgent**, **MaintenanceAgent**, **ReportingAgent**) that read graph triples and write new triples.

- **Input:** The initialized graph from WP1. Agent configuration (could be a JSON/YAML file) specifying agent types and identities. For example:

```
{ "agents": [
     {"id": "Agent1", "role": "SensorReader", "reads": "Sensor1"},
     {"id": "Agent2", "role": "MaintenancePlanner"},
     ...
  ] }
```

- **Output:** Python classes/modules for each agent, and registration of agents in the graph. For example, after instantiating, each agent writes its presence:

```
graph.add((ex.Agent1, ex.hasRole, ex.SensorAgent))
```

Agents may also add an `ex:status` triple (e.g. `ex:Agent1 ex:status ex:Idle`).

- **Agents & Graph:** Agents operate indirectly via the graph. For example, a `SensorAgent` could poll a sensor (simulated) and then do:

```
reading = read_sensor(ex.Sensor1)  # e.g. returns 78.5
graph.add((ex.Sensor1, ex.latestReading, Literal(reading)))
# Also add timestamp and link to agent
graph.add((ex.Sensor1, ex.lastUpdatedBy, ex.Agent1))
```

An `DataProcessorAgent` might query:

```
q = graph.query("SELECT ?s ?val WHERE { ?s ex:latestReading ?val }")
for sensor, val in q:
    if float(val) > threshold:
        graph.add((sensor, ex:anomalyDetected, Literal(True)))
```

This pattern ensures no direct agent-to-agent messaging: all results go into triples.

- **Test Cases:** Each agent has unit tests. For example:

- **SensorAgent Test:** Simulate a sensor update. After running `SensorAgent.run()`, the graph should contain a triple `(Sensor1, ex:latestReading, <value>)`.
- **MaintenanceAgent Test:** Given a triple `(MachineA, ex:needsService, ex:True)`, after `MaintenanceAgent.run()`, expect a triple `(TaskNew, ex:assignedTo, ex:MaintenanceAgent1)` indicating a new maintenance task was created and linked to the agent.
- **Expected Outcomes:** Agents correctly attach to the graph and perform their logic. The graph evolves with new triples (e.g. sensor readings, new tasks). The agent framework should be modular: one can add a new agent class without altering others.
- **Extensibility Hooks:** Use an **AgentFactory** pattern: a Python factory that can create agents by role from config. New agent types (e.g. `OptimizerAgent`) can be registered by subclassing a base `Agent` class. The factory reads the graph to decide which agents to launch. For example, the orchestrator might read `graph.objects(None, ex:needsAgent)` to spawn agents.

Example base class snippet:

```
class BaseAgent:
    def __init__(self, graph, agent_uri):
        self.graph = graph
        self.agent = agent_uri  # URIRef for this agent
    def run(self):
        raise NotImplementedError
# Example subclass
class SensorAgent(BaseAgent):
    def run(self):
        reading = self.read_sensor()
        self.graph.add((self.agent, ex:producedReading, Literal(reading)))
```

This sets the stage for advanced workflows in WP3.

Work Package 3: Workflow Orchestration via Knowledge Graph

Objectives: Implement the core coordination logic: agents should be instantiated or triggered based on graph state. For example, an **OrchestratorAgent** continuously queries the graph for conditions (like `ex:Pending` tasks or flagged events) and then delegates work by creating task triples. Agents **self-assemble**: new agents (or tasks) appear when needed.

- **Input:** The evolving graph from WP2. Key patterns like `(TaskX, ex:status, ex:Pending)` or `(MachineY, ex:temperature, ?val)` indicate actions. For instance, a triple `(Machine1, ex:needsInspection, ex:True)` in the graph could trigger a **MaintenanceAgent**.
- **Output:** Graph updates and agent actions triggered in sequence. For example, upon detecting a pending task triple, the orchestrator might:

- Select an available agent (or create one) and
- Assign the task by adding `(TaskX, ex:assignedTo, AgentZ)` and `(AgentZ, ex:status, ex:Busy)`.
  Agents then act, updating `(TaskX, ex:status, ex:Done)` when finished.

- **Agents & Graph:** The OrchestratorAgent could run logic like:

```
pending = graph.query("SELECT ?t WHERE {?t rdf:type ex:Task . ?t ex:status ex:Pending}")
for task in pending:
    agent = find_free_agent(graph)
    graph.add((task[0], ex:assignedTo, agent))
    graph.set((task[0], ex:status, Literal("InProgress")))
    # Launch agent handler asynchronously (e.g. new thread/process)
```

  Meanwhile, other agents update results: e.g. `AnalyzerAgent` reads data triples and writes analytics:

```
if high_value_detected:
    graph.add((ex.Report1, ex:containsFinding, ex:HighRisk))
```

  All interactions remain in RDF form.

- **Test Cases:** Simulate complete workflow scenarios. Example tests:

- **Task Assignment:** Insert a triple `(Task42, ex:status, ex:Pending)` into a fresh graph. Run `OrchestratorAgent.run_once()`. Assert that `(Task42, ex:assignedTo, ?agent)` and `(Task42, ex:status, ex:InProgress)` now exist.
- **Data Pipeline Flow:** Create triples representing raw data. Run a series of agents (DataCollector -> DataProcessor -> ReportAgent) and verify the final graph contains expected results (e.g. `(AnalysisReport, ex:score, Literal(0.95))`).

- **Expected Outcomes:** The system processes workflows correctly end-to-end. A high-level trace: **create tasks → orchestrator spawns agents → agents update graph → new conditions spawn new agents**. For example, one test could feed in a triple `(Machine7, ex:overheated, ex:True)`, then expect the graph to show a new `ex:InspectionTask` assigned to a `MaintenanceAgent` with `(ex:InspectionTask, ex:status, ex:Done)` after processing.

- **Extensibility Hooks:** The orchestration logic should be data-driven. New triggers can be added by extending the query conditions in OrchestratorAgent. For example, loading SPARQL trigger rules from a config file. One could also plug in GraphDB's SPARQL endpoint via `SPARQLWrapper` if scaling beyond RDFLib. (GraphDB's TTYG "Talk-To-Your-Graph" API can be used here to let a small LLM assist in suggesting graph queries, but all outputs must be written back as triples.)

Figure 2: Conceptual knowledge-driven AI pipeline for multi-agent coordination. This figure (adapted from Cornelio *et al.*) shows how data, background knowledge, and reasoning interleave in cycles of analysis [5] [6]. In our system, a similar pipeline is realized via graph updates: for example, raw sensor triples feed into an inference agent, which augments the graph with *insights*, and a decision agent reads those insights to spawn tasks. The workflow is always reflected in the graph state.

Work Package 4: Dynamic Agent Instantiation & Role Delegation

Objectives: Support **self-assembling** behavior: agents must be able to spawn additional agent instances or change roles when the graph demands it. For instance, if processing backlog grows, the system should instantiate more `DataProcessorAgent`s. Roles may be delegated: a SupervisorAgent could reassign tasks to idle agents.

- **Input:** Graph state indicating dynamic conditions, such as `(ex:QueueLength, ex:count, Literal(100))`. Configuration thresholds (e.g. "spawn a new processor if pending tasks > 50").

- **Output:** New agent instances created and registered in graph, updated role assignments. Example graph triples after scaling:

  ```
  (ex:Agent17, ex:hasRole, ex:DataProcessor)
  (ex:Agent18, ex:hasRole, ex:DataProcessor)
  (ex:Agent17, ex:status, ex:Active)
  ```

  Agents write triples `(Agent17, ex:spawnedFrom, AgentFactory1)` or similar.

- **Agents & Graph:** Implement an **AgentFactory** or **SupervisorAgent** that monitors the graph and spawns processes. In Python, this could use `multiprocessing` or simply start new threads:

  ```
  pending_count = graph.query("SELECT (COUNT(?t) AS ?c) WHERE { ?t ex:status
  ex:Pending }")
  if int(pending_count.bindings[0]['c']) > threshold:
      new_agent = AgentFactory.create('DataProcessorAgent')
      graph.add((new_agent.uri, ex:hasRole, ex.DataProcessorAgent))
  ```

  Alternatively, agents could be containerized services launched via subprocesses. Each new agent registers itself by adding a triple like `(AgentX, ex:isInstanceOf, ex.DataProcessorAgent)`.

- **Test Cases:** Test the auto-scaling logic. For example:

- **Scale-Up Test:** Insert 100 pending-task triples into the graph. Run `SupervisorAgent.run_once()`. Verify that at least `⌊100/threshold⌋` new `DataProcessorAgent`s appear in the graph (each with a `ex:hasRole` triple).

- **Role Change Test:** Simulate that an agent's service is needed in another area: e.g. `(Agent5, ex:requestRoleChange, ex:DataValidator)`. After running a RoleManagerAgent, assert that `(Agent5, ex:hasRole, ex:DataValidator)` and possibly `(Agent5, ex:hadRole, ex:OldRole)` have been added.

- **Expected Outcomes:** The system adapts to workload. Graph examples:

- When workload spikes, new agent URIs appear with appropriate role triples.
- Tasks are delegated to these new agents, reflected by `(Task42, ex:assignedTo, Agent17)`.
- Over time, if tasks drop, agents may retire or switch to idle status, e.g. `(Agent17, ex:status, ex:Idle)`.

- **Extensibility Hooks:** Policies for instantiation are data-driven. One can extend the graph schema to include *capabilities* (e.g. `(AgentY, ex:canHandle, ex:ImageData)`) and let the factory respect those. New factories can be added for different agent families. We also plan hooks for integrating a monitoring dashboard: agents could push metrics (e.g. message rates) back into the graph for visualization.

Work Package 5: Fault Tolerance and Monitoring

Objectives: Ensure the system is robust. Agents may fail or produce errors; the system should detect these via the graph and recover. We include a **MonitorAgent** or similar that checks for anomalies (e.g. tasks stuck in `InProgress` too long) and reassigns them. Faulty agents should be restartable or replaceable.

- **Input:** Scenarios where agents crash or behave incorrectly. For example, an agent fails to complete `(Task99, ex:status, ex:Done)` or an exception is logged. The graph could be annotated with a `ex:failed` flag: `(Agent5, ex:status, ex:Failed)`.

- **Output:** Graph-based alerts and recovery actions. For example:

```
(Task99, ex:status, ex:Failed)
(Alert1, ex:alertsTo, ex:MonitorAgent)
(Agent5, ex:needsRestart, True)
(Task99, ex:assignedTo, ex:Agent6)
```

  A *MonitorAgent* might add `(Task99, ex:status, ex:Failed)` when it notices no update over time, then create a new task or reassign.

- **Agents & Graph:** Build a `WatchdogAgent` that periodically scans the graph. It can query:

```
stuck_tasks = graph.query("SELECT ?t WHERE { ?t ex:status ex:InProgress . ?t ex:lastUpdated ?time . FILTER(now()-?time > 3600) }")
for task in stuck_tasks:
```

```
    graph.add((task, ex:status, ex:Failed))
    # Optionally re-create task
    new_task = create_task_for(task)
    graph.add((new_task, ex:assignedTo, find_free_agent(graph)))
```

We also use standard logging: Python's `logging` module writes to files, and critical events (errors) are written back to the graph as triples (`ex:LogEntry`).

- **Test Cases:** Emulate failures. For instance:

- **Agent Crash:** Simulate raising an exception inside an agent run (or forcibly terminate a thread). After giving time for the monitor, assert that the graph shows a new agent with `(ex:status, ex:Idle)` and that orphaned tasks were reassigned.

- **Data Consistency:** Corrupt a triple on purpose (e.g. delete an intermediate step) and verify that the monitor detects a gap (maybe by expecting a certain pattern) and flags an alert triple `(ex:Alert123, ex:severity, ex:High)`.

- **Expected Outcomes:** Faults are reflected and handled in the knowledge graph. We should see recovery steps in the graph: e.g. tasks re-queued, agents reinitialized. The system should log its own health as triples (e.g. `(AgentStatus, ex:agentCount, Literal(7))`).

- **Extensibility Hooks:** The monitoring logic can be extended with custom anomaly detectors. One could plug in a Prometheus exporter or build a small dashboard that queries the graph (e.g. count of `ex:Failed` tasks over time). Alerts could be published by adding triples to an `ex:AlertQueue` node. We can also plan a "Game-over" test: if >90% tasks fail, the system resets to a known good state (roll back the graph or reload baseline data).

Work Package 6: Reporting, Testing & Deployment

Objectives: Finalize the challenge with end-to-end demonstration, reporting, and extensibility. Provide scripts to run the entire system, comprehensive tests, and documentation. Showcase the intelligent coordination via sample scenario outputs and monitoring dashboards.

- **Input:** The final graph after several cycles, logs of agent activities, and sample query results. For example, an output triple `(ReportFinal, ex:summary, "All tasks completed successfully")`.

- **Output:** A runnable codebase with:

- **Installation**: `requirements.txt` specifying `rdflib`, `SPARQLWrapper`, etc., plus instructions to spin up GraphDB (or use an RDFLib in-memory store).
- **Startup script**: e.g. `main.py` that initializes the graph, creates agents (via the factory), and enters a loop or scheduler. Example snippet:

```
if __name__ == "__main__":
    graph = setup_graph()  # from WP1
    orchestrator = OrchestratorAgent(graph, ex.Orchestrator)
    while True:
        orchestrator.run()
        # run other periodic agents
        time.sleep(1)
```

- **Sample data:** A `data/` folder with example RDF/Turtle files (e.g. `sensors.ttl`, `initial_tasks.ttl`) to feed the system.

- **Monitoring tools:** Scripts that export graph metrics, e.g. a SPARQL query that counts pending tasks every minute.

- **Test Cases:** Use a testing framework (e.g. `pytest`). We provide:

- **Unit tests** for each agent class (as in WP2–WP5).
- **Integration test:** A scenario script that loads `data/sample_scenario.ttl`, runs the system for a set number of steps, and then checks final graph state. For instance, after running, `assert ('http://example.com#TaskFinal', ex:status, ex:Done) in graph` and `(ex:FinalReport, ex:madeBy, ex:ReportingAgent)` exists.

- **Fault-tolerance test:** Automated tests that simulate failures (e.g. kill a thread) and verify the monitor agent recovers.

- **Expected Outcomes:** A user can clone the repository, run `pip install -r requirements.txt`, and start the system. The README provides clear setup instructions. After running with sample data, the system outputs a final graph (exportable as `final_state.ttl`) and logs. We expect:

- Graph updates tracing all agent actions.
- A final report triple or file summarizing the run (e.g. total tasks completed, agents used).

- Example visualization: a script can take the RDF graph and generate a PDF/PNG of the subgraph of completed tasks for demo purposes.

- **Extensibility Hooks:** The design is modular so new features can be added. For example, one might add a new agent (say, `PlannerAgent`) by writing a subclass and registering it. The ontology can be extended by editing the TTL schema. The orchestrator's logic can be made more data-driven by reading SPARQL rules from a file. The system could be containerized (each agent as a microservice) by publishing metrics via the graph to external tools.

Each Work Package above includes sample code, diagrams, and detailed steps. This design ensures that as data flows through the knowledge graph, the multi-agent system **self-organizes**: agents appear, tasks move from `Pending` to `Done`, and any issues are flagged and resolved. Together, they demonstrate an **enterprise-grade workflow** with end-to-end automation and robust coordination (as if a team of 20 Google engineers had built it).

References: We leverage RDFLib (a pure-Python RDF library [1]) or GraphDB (Ontotext's high-performance RDF store [2]) for the graph backend, and follow knowledge-graph best practices [3]. Agents follow MAS principles [4], but all communication is via RDF triples, not unstructured chat. The embedded figures illustrate the domain schema (Figure 1) and reasoning pipeline (Figure 2) for context. This artifact is intended as a polished, shareable blueprint ready for technical review.

---

[1] rdflib 7.1.4 — rdflib 7.1.4 documentation
https://rdflib.readthedocs.io/en/stable/

[2] GraphDB: Empowering Semantic Data Management and Knowledge Exploration | enRichMyData
https://www.linkedin.com/posts/enrichmydata_graphdb-empowering-semantic-data-management-activity-7283044876001177600-Xr1f

[3] Knowledge graph - Wikipedia
https://en.wikipedia.org/wiki/Knowledge_graph

[4] Multi-agent system - Wikipedia
https://en.wikipedia.org/wiki/Multi-agent_system

[5] [6] File:AI Descartes system overview (an AI scientist).webp - Wikimedia Commons
https://commons.wikimedia.org/wiki/File:AI_Descartes_system_overview_(an_AI_scientist).webp