ChatGPT

System Design Feedback and Recommendations

The existing system already embodies a modular agentic design with a **shared knowledge graph (KG) as memory and coordination layer**, which aligns well with best practices in multi-agent architectures [1] [2]. For robustness and scalability, we recommend making the architecture **more dynamic and resilient**. In particular:

- **Modular, Service-Oriented Agents:** Ensure each agent (e.g. `VertexEmailAgent`, `EngagementManagerAgent`) operates as an independent service specialized for its domain [1]. This aligns with the ideal that *"each agent specializes in particular tasks or domains"* and can be added or replaced without breaking the system [1].
- **Shared Knowledge Graph as Central Hub:** Use the KG not just for static data, but as a *"living" memory* that agents continually **query and update** [2]. As Gillespie notes, in an agentic KG system *"agents store facts as nodes and edges, query and infer via graph operations, and then persist new knowledge back into the graph"*, forming a tight feedback loop [2]. Designing the KG with this loop in mind is critical: tasks, agent states, and results should all be first-class graph entities.
- **Dynamic Agent Lifecycle Management:** Introduce an **Agent Factory or orchestrator** that monitors the KG and **dynamically spawns or terminates agents** based on KG state. For example, a special `AgentRequested` node could trigger the factory to instantiate a new agent instance (and record an `AgentCreated` event in the KG). Agent status and lifecycles (Created, Active, Completed, Error) should be tracked as properties or linked event nodes. This follows a pattern where the KG *"grows and self-organizes through each agent's actions"* [2].
- **Task Delegation via KG:** Model work items as `Task` nodes in the KG with properties like `status`, `priority`, and relationships like `assignedTo → Agent`. Agents should pick up tasks by querying for unassigned `Task` nodes they are capable of handling, then update the task node (e.g. set status to *"in progress"* and link to themselves) when accepting it. Upon completion, agents write results back into the graph (e.g. `Task.status = "done"`, attach output data). This **graph-native coordination** ensures all agents read and write a single source of truth [2] [1].
- **Error Handling and Resilience:** Implement robust transaction and consistency safeguards in the KG (e.g. use ACID transactions, SHACL constraints). If an agent fails, it should update its node to an error state or emit a termination event. Monitoring agents (or the Agent Factory) can detect stalled tasks or zombies and reassign as needed. These patterns reflect the need for *"modular, scalable, and resilient"* system design [1].
- **Scalability:** To handle enterprise-scale data, consider a **distributed graph database** (e.g. clustered Neo4j, Amazon Neptune) and optimize with indexing or caching. Graphiti's real-time indexing (vector/BM25) is one approach [3]. Partition the KG if needed to balance load.
- **Feedback Loop Automation:** Embrace the *"query→reason→act→update"* cycle [4]. For example, after an agent updates the KG, other agents or a central coordinator should be able to immediately pick up any newly available tasks or insights. This continuous loop (e.g. via event triggers on the KG) makes the system self-improving [4].

In summary, we suggest treating the KG as the active **orchestration layer** of the system. Agents should be as stateless as possible beyond the KG, and all coordination (task assignment, state changes, role negotiation) should be encoded in the KG. This approach has been shown to yield *"highly modular, scalable, and resilient"* architectures [1].

Code Examples

Below are illustrative code snippets (in Python-like pseudocode) showing how the above patterns can be implemented. These are **examples**, not production-ready libraries.

1. AgentFactory for Dynamic Instantiation

```
class AgentFactory:
    """Monitors the KG and spawns agent instances for new agent nodes."""
    def __init__(self, kg_client):
        self.kg = kg_client             # Interface to the knowledge graph
        self.active_agents = {}         # Map: agent_node_id -> agent_instance

    def check_and_spawn(self):
        # Query KG for any Agent nodes with status 'Requested'
        query = """
            MATCH (a:Agent {status: 'Requested'}) RETURN a.agent_id AS id,
a.type AS agent_type
        """
        results = self.kg.execute(query)
        for record in results:
            agent_id = record["id"]
            agent_type = record["agent_type"]
            if agent_id not in self.active_agents:
                # Dynamically instantiate the appropriate agent class
                agent = self._instantiate_agent(agent_type, agent_id)
                self.active_agents[agent_id] = agent
                # Update KG to mark agent as 'Active'
                update = f"""
                    MATCH (a:Agent {{agent_id: '{agent_id}'}})
                    SET a.status = 'Active', a.startTime = datetime()
                """
                self.kg.execute(update)

    def _instantiate_agent(self, agent_type, agent_id):

# Map agent_type string to actual class (could use a registry or factory dict)
        agent_class = {
            "VertexEmailAgent": VertexEmailAgent,
            "EngagementManagerAgent": EngagementManagerAgent,
            # add other agent classes here
        }.get(agent_type)
        if not agent_class:
            raise ValueError(f"Unknown agent type: {agent_type}")
        # Create the agent instance (passing its ID and KG client)
        return agent_class(agent_id=agent_id, kg_client=self.kg)
```

This `AgentFactory` periodically scans the KG for new `Agent` nodes with a status of `"Requested"` (or `"Created"`), and spawns the corresponding agent objects. It then updates the node to `"Active"`. In practice, this loop could be driven by events or a scheduler. Each spawned agent would run in its own thread/process and register itself in `self.active_agents`.

2. Task Delegation and State Updates

```
class TaskCoordinator:
    """Coordinates tasks between agents using the KG as the task board."""
    def __init__(self, kg_client):
        self.kg = kg_client

    def assign_tasks(self):
        # Find all unassigned tasks
        query = """
            MATCH (t:Task {status: 'Pending'}) RETURN t.task_id AS id,
t.requiredSkill AS skill
        """
        tasks = self.kg.execute(query)
        for record in tasks:
            task_id = record["id"]
            skill = record["skill"]
            # Find an available agent with matching capability
            agent_query = f"""
                MATCH (a:Agent)
                WHERE a.status = 'Active' AND '{skill}' IN a.capabilities
                  AND NOT EXISTS((a)-[:ASSIGNED]->(:Task {{task_id:
'{task_id}'}}))
                RETURN a.agent_id AS agent_id LIMIT 1
            """
            agent_rec = self.kg.execute(agent_query)
            if agent_rec:
                agent_id = agent_rec[0]["agent_id"]
                # Assign the task in the KG
                assign_query = f"""
                    MATCH (t:Task {{task_id: '{task_id}'}}), (a:Agent
{{agent_id: '{agent_id}'}})
                    CREATE (a)-[:ASSIGNED {{assignedAt: datetime()}}]->(t)
                    SET t.status = 'InProgress', t.assignedTo = '{agent_id}'
                """
                self.kg.execute(assign_query)
```

Here, `TaskCoordinator` looks for all `Task` nodes with `status: 'Pending'`, finds an active agent who can perform the task, and creates a relationship `(:Agent)-[:ASSIGNED]->(:Task)` in the graph. It also updates task properties. Agents themselves would run a loop similar to this, pulling tasks from the KG, or alternatively a central coordinator could do it. This **graph-based delegation** encodes task assignments as explicit graph edges.

3. Self-Assembly Test Scenario

Below is an example of how one might write an integration test to simulate self-assembly and edge cases. In reality, a test framework (e.g. `pytest`) would be used.

```
def test_dynamic_agent_creation_and_task_flow():
    # Initialize an in-memory KG and factory/coordinator
    kg = KnowledgeGraph(in_memory=True)
    factory = AgentFactory(kg_client=kg)
    coordinator = TaskCoordinator(kg_client=kg)

    # 1. Simulate a new agent request in the KG
    kg.execute("""
        CREATE (a:Agent {agent_id: 'A1', type: 'VertexEmailAgent', status:
'Requested'})
    """)
    # The factory should detect and spawn it
    factory.check_and_spawn()
    assert 'A1' in factory.active_agents

    # 2. Simulate a new task requiring email skill
    kg.execute("""
        CREATE (t:Task {task_id: 'T1', requiredSkill: 'email', status:
'Pending'})
    """)
    # Coordinator assigns the task
    coordinator.assign_tasks()
    # Verify that the task is assigned to agent A1
    status = kg.query("MATCH (t:Task {task_id: 'T1'}) RETURN t.status AS s")[0]
['s']
    assignee = kg.query("MATCH (:Agent {agent_id: 'A1'})-[:ASSIGNED]->(t:Task)
RETURN t.task_id AS id")[0]['id']
    assert status == 'InProgress'
    assert assignee == 'T1'

    # 3. Agent completes the task (simulate by the test)
    kg.execute("""
        MATCH (t:Task {task_id: 'T1'})
        SET t.status = 'Done', t.completionTime = datetime()
    """)
    # Ensure final state is recorded
    final_status = kg.query("MATCH (t:Task {task_id: 'T1'}) RETURN t.status AS
s")[0]['s']
    assert final_status == 'Done'
```

This test sequence creates a new agent and a new task in the KG, runs the factory and coordinator, and checks that the agent is spawned and the task is properly assigned and completed. Additional tests should simulate error conditions (e.g., no agent available, agent throws exception, duplicate tasks) to ensure robustness.

Ontology Design Patterns for Agents and Coordination

To effectively use the KG for agent coordination, we should extend its ontology with concepts and relationships for agents, tasks, and lifecycle events. Here are some suggested design patterns:

- **Agent and AgentType Classes:** Define an `:Agent` class (subclass of `prov:Agent` or similar) with properties like `:agentId`, `:agentType`, `:status`, `:createdAt`, `:terminatedAt`. Also define `:AgentType` or `:Capability` classes to enumerate roles (e.g. EmailAgent, ChatAgent, ManagerAgent) or skills. For example:

  ```
  :Agent rdf:type owl:Class .
  :agentId rdf:type owl:DatatypeProperty; rdfs:domain :Agent; rdfs:range
  xsd:string .
  :agentType rdf:type owl:ObjectProperty; rdfs:domain :Agent;
  rdfs:range :AgentType .
  :status rdf:type owl:DatatypeProperty; rdfs:domain :Agent; rdfs:range
  xsd:string .
  ```

- **Task Class and States:** Introduce a `:Task` class with properties `:taskId`, `:status`, `:assignedTo`, `:priority`, etc. Relate agents and tasks via object properties:

  ```
  :Task rdf:type owl:Class .
  :assignedTo rdf:type owl:ObjectProperty; rdfs:domain :Task;
  rdfs:range :Agent .
  :hasStatus rdf:type owl:DatatypeProperty; rdfs:domain :Task; rdfs:range
  xsd:string .
  ```

For example, `(:Task {taskId="T1", status="Pending"})-[:assignedTo]->(:Agent {agentId="A1"})`.

- **Lifecycle Event Patterns:** Use an `:Event` or reuse PROV-O's `prov:Activity` to model events like creation, assignment, completion. E.g., a `:TaskAssignedEvent` with properties `:timestamp`, linking an `Agent` and `Task`. This provides an audit trail. For instance:

```
:TaskAssignedEvent rdf:type owl:Class .
:triggeredAt rdf:type owl:DatatypeProperty; rdfs:domain :TaskAssignedEvent;
rdfs:range xsd:dateTime .
:involvesAgent rdf:type owl:ObjectProperty; rdfs:domain :TaskAssignedEvent;
rdfs:range :Agent .
:involvesTask rdf:type owl:ObjectProperty; rdfs:domain :TaskAssignedEvent;
rdfs:range :Task .
```

  By creating event nodes (or edge annotations), the KG can record *when* agents are created, when tasks are started/finished, etc. This follows provenance patterns (e.g. PROV-O) and enables tracing and rollback.

- **Roles and Capabilities:** If agents have specific skills, model these either as classes or via properties. For example, a `:hasCapability` property linking an `Agent` to a `Capability` node (e.g. “SendEmail”, “AnalyzeSentiment”). This makes it easy to query “which agent can handle this task?” by matching capabilities.
- **Coordination Patterns:** Use edges to represent communication or dependencies. E.g., `(:Agent)-[:REQUESTED_DATA]->(:Task)` for an agent asking for input, or `(:Agent)-[:INVITES]->(:Agent)` if one agent delegates to another. These richer relations let the KG capture workflow structure.
- **Time and Versioning:** For dynamic behaviors, consider temporal patterns. You might include timestamp properties (`t_valid`, `t_invalid`) on relationships (as in Graphiti) to indicate when a piece of information was true, allowing historical queries or conflict resolution [5].
- **Pattern Example (Turtle snippet):**

```
@prefix : <http://example.org/agent-kg#> .
:VertexEmailAgent rdf:type :AgentType .
:A1 rdf:type :Agent; :agentType :VertexEmailAgent; :status "Created" .
:T1 rdf:type :Task; :status "Pending"; :priority "High" .
:Assignment1
rdf:type :TaskAssignedEvent; :involvesAgent :A1; :involvesTask :T1; :triggeredAt
"2025-05-31T03:00:00Z"^^xsd:dateTime .
:T1 :assignedTo :A1 .
```

  This schema ensures all agent and task states are queryable and auditable in the graph.

By formalizing these ontology patterns, the KG can answer questions like “which agents are available for email tasks?”, “what is the current status of task T42?”, or “give me the event log of agent A3”. This makes coordination explicit and machine-interpretable.

Comparative Framework Analysis

Several emerging frameworks illustrate different approaches to multi-agent, graph-backed AI:

- **AutoGen (Microsoft)** – AutoGen provides a **conversational multi-agent infrastructure** where agents talk in structured dialogues. Agents in AutoGen are *“customizable, conversable, and can operate in various modes”* (LLM, human, tools) [6]. It excels at dynamic collaboration: agents can **role-play and reassign tasks** on the fly [7]. AutoGen likens agents to teammates, enabling **self-improving workflows** through iterative feedback (e.g. one agent critiques another’s output) [8] [7]. However, AutoGen’s focus is on the conversation layer and agent cooperation protocols; it doesn’t prescribe a central knowledge graph. Instead, memory is often implicit or handled via prompt context. In practice, AutoGen could be integrated with a KG for persistent memory, but out-of-the-box it emphasizes free-form agent conversation over structured graph coordination.

- **LangGraph** – LangGraph explicitly models multi-agent workflows as directed graphs. Each node is a logical step or agent, and edges define control flow [9]. It provides **deterministic execution paths** (“decision trees”) that ensure consistency and traceability [9]. LangGraph is *“designed for structured, graph-based execution flows”* [10], making it well-suited to complex, hierarchical pipelines (e.g. customer support routing) [11]. It integrates with LangChain for memory and tools, but the emphasis is on composition of agents/tasks rather than on adaptive self-assembly. In LangGraph, the multi-agent design (agents and transitions) is usually pre-defined, though one could imagine dynamically modifying the graph at runtime. Compared to our system’s KG-centric approach, LangGraph provides strong control flow semantics but lacks an inherent persistent KG memory. Its benefit is **precise orchestration**, while its limitation (for agentic KG) is that it treats the workflow as mostly static.

- **Graphiti (Zep AI)** – Graphiti focuses on the **knowledge graph memory layer** rather than the agent conversation layer. It provides a real-time, **temporally-aware graph engine** for agentic applications [12]. Graphiti continuously ingests new data (chats, JSON, text) and updates the graph without batch re-computation [12], using a bi-temporal model to handle evolving facts [5]. In an agentic system, Graphiti can serve as the self-organizing memory: *“an ever-present source of context for agents, continuously available and updated”* [12]. This matches the *“living graph”* vision [4]. While Graphiti itself does not manage multiple agent dialogs, it perfectly complements them by maintaining the up-to-date state. For example, an AutoGen or LangGraph agent could use Graphiti as its knowledge base. Graphiti’s strength is **dynamic, conflict-resolved memory**; its coordination role is indirect (it surfaces information, but doesn’t decide who acts next).

Summary: AutoGen emphasizes **flexible multi-agent dialogue** (agents as teammates with dynamic role assignment) [8] [7]. LangGraph emphasizes **structured, graph-driven workflows** with clear execution order [9]. Graphiti provides a **live knowledge graph memory** that agents share and update [12] [4]. Our proposed system aims to combine these strengths: agents that are modular and conversational (à la AutoGen), workflows that can form and adapt in a graph-like manner (à la LangGraph), and a KG memory that records all facts and events in real time (à la Graphiti/agentic KG). In other words, we want the **self-assembly** and adaptability of AutoGen with the structural clarity of LangGraph and the continuous memory feedback loop of Graphiti [4] [9] [8].

Architecture and Agent Design (Illustrative Diagrams)

Multi-agent AI systems typically follow a sense-think-act cycle, with **shared memory** at the center [13] [2]. For example, each agent may include perception (`Sense`), reasoning/planning (`Reason`/`Plan`), coordination (`Coordinate` via the KG), and action (`Act`) components [13]. The diagram below sketches a single-agent architecture (borrowed from industry practices) showing perception, cognitive layers, and actuators.

Figure: Example agent architecture – an agent connects to external tools, maintains short-term/long-term memory, and has a planning (reasoning) module [13] *.*

At runtime, multiple such agents operate in parallel, linked by the shared knowledge graph (not shown). An agent's **design guide** would emphasize: - **Memory:** Include a module for **short-term context** (recent conversation or sensory inputs) and **long-term memory** (facts and histories stored in the KG). - **Tools:** Provide agents with specialized tools or APIs (e.g., email sender, code executor, database query) that they can call as actions. - **Planning/Reasoning:** Agents should support advanced reasoning (LLM chain-of-thought, reflection, or planning algorithms) to decide next actions. - **Coordination:** Each agent should have a connector to the KG: after acting, it writes results to the KG; before planning, it queries the KG for relevant facts. - **Lifecycle Hooks:** Agents emit events to the KG when they start, pause, or finish. These help the system keep the graph *"self-organizing through each agent's actions"* [2] .

Such a design (tools + memory + planning) matches modern recommendations for agent architectures [13] and is illustrated conceptually above.

Knowledge Graph Schema (Agent Lifecycle & Interaction)

Below is a **conceptual schema** (in RDF/Turtle) for agent lifecycle and interaction. It demonstrates how agents, tasks, and events can be modeled:

```
@prefix : <http://example.org/agentkg#> .
@prefix prov: <http://www.w3.org/ns/prov#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

:Agent rdf:type owl:Class .
:Task rdf:type owl:Class .
:AgentType rdf:type owl:Class .

:agentId rdf:type owl:DatatypeProperty; rdfs:domain :Agent; rdfs:range
xsd:string .
:agentType rdf:type owl:ObjectProperty; rdfs:domain :Agent;
```

```
rdfs:range :AgentType .
:status rdf:type owl:DatatypeProperty; rdfs:domain [ :Agent :Task ]; rdfs:range
xsd:string .

:assignedTo rdf:type owl:ObjectProperty; rdfs:domain :Task; rdfs:range :Agent .
:priority rdf:type owl:DatatypeProperty; rdfs:domain :Task; rdfs:range
xsd:string .

:TaskAssignedEvent rdf:type owl:Class .
:triggeredAt rdf:type owl:DatatypeProperty; rdfs:domain :TaskAssignedEvent;
rdfs:range xsd:dateTime .
:involvesAgent rdf:type owl:ObjectProperty; rdfs:domain :TaskAssignedEvent;
rdfs:range :Agent .
:involvesTask rdf:type owl:ObjectProperty; rdfs:domain :TaskAssignedEvent;
rdfs:range :Task .
```

Key points: `:Agent` and `:Task` are core classes, with properties for IDs and states. The `assignedTo` relationship links tasks to agents. `:TaskAssignedEvent` (a subclass of `prov:Activity`) records the time, agent, and task whenever a delegation happens. Similar event classes (`:AgentCreatedEvent`, `:TaskCompletedEvent`) can be added. This schema allows queries like:

- *"Which agent is assigned to task T123?"*
- *"List all tasks created by agent A1."*
- *"Show the timeline of events for task T123."*

By conforming to such a schema, the KG can enforce consistency (e.g. using SHACL) and support graph-based reasoning (e.g. finding all pending tasks for a given capability). Patterns from PROV-O and schema.org can be reused (e.g. `prov:Agent`, `schema:Action`) to align with standards.

Test Suite Improvements

To validate the **self-assembling** behaviors, the test suite should be expanded with:

- **Dynamic Scenario Tests:** Write integration tests where new agent and task nodes are added at runtime, and verify the system spawns agents and updates the KG correctly (as illustrated above).
- **Concurrency and Race Conditions:** Simulate multiple agents trying to claim the same task or update the same graph region, ensuring proper locking or idempotent updates. For example, use concurrent threads in tests to create potential conflicts.
- **Failure Injection:** Test how the system handles agent failures. For instance, force an agent to crash mid-task and check that the KG reflects an error state and that another agent can recover or retry the task.
- **Edge Cases:** Include tests for missing or invalid data (e.g. a task with no `requiredSkill` property) to ensure the system fails gracefully or logs clear errors.
- **Scalability Stress Tests:** With a mocked large KG (many nodes), test that agent-factory queries remain performant and that task delegation works under load.

- **Ontological Constraints:** Validate the KG against the ontology (e.g. using SHACL) as part of tests, ensuring no invalid triples are created by agents.

These improvements will ensure that as the system becomes more dynamic (self-spawning agents, on-the-fly task graphs), it remains **robust and predictable** under varied conditions.

References

- Arzye et al., **AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation** – describes flexible, role-playing agents [6] [8].
- Chalef (Neo4j), **Graphiti: Knowledge Graph Memory for an Agentic World** – introduces a dynamic KG engine for agent memory [12] [5].
- Arsanjani, **The Anatomy of Agentic AI** – details agent internal components (Sense, Plan, Act, etc.) and the need for shared memory [13] [14].
- Gillespie, **Agentic Knowledge Graphs: A Living, Learning AI Architecture** – defines the "agentic KG" pattern of continuous update and reasoning [2] [4].
- Shah, *"AutoGen vs. LangGraph vs. CrewAI: Who Wins?"* – compares frameworks; notes AutoGen's dynamic role assignment and LangGraph's structured workflows [8] [9].
- Hypermode blog, **"How knowledge graphs underpin AI-agent applications"** – emphasizes KG as communication medium and modularity [1].

---

[1] How knowledge graphs form a system of truth underpinning agentic apps – Hypermode
https://hypermode.com/blog/how-knowledge-graphs-underpin-ai-agent-applications

[2] [4] Agentic Knowledge Graphs: A Living, Learning AI Architecture
https://www.linkedin.com/pulse/agentic-knowledge-graphs-living-learning-ai-rick-gillespie-3g4ue

[3] [5] [12] Graphiti: Knowledge Graph Memory for an Agentic World - Graph Database & Analytics
https://neo4j.com/blog/developer/graphiti-knowledge-graph-memory/

[6] AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation - Microsoft Research
https://www.microsoft.com/en-us/research/publication/autogen-enabling-next-gen-llm-applications-via-multi-agent-conversation-framework/

[7] [8] [9] [10] [11] AutoGen vs. LangGraph vs. CrewAI:Who Wins? | by Khushbu Shah | ProjectPro | Medium
https://medium.com/projectpro/autogen-vs-langgraph-vs-crewai-who-wins-02e6cc7c5cb8

[13] [14] The Anatomy of Agentic AI. In this article we will elaborate on... | by Ali Arsanjani | Medium
https://dr-arsanjani.medium.com/the-anatomy-of-agentic-ai-0ae7d243d13c