140509_25.md

README

25. Multi-Agent Coordination Platform

Summary: Develop a platform that orchestrates multiple specialized AI agents to collaboratively solve complex tasks through task decomposition, communication, and conflict resolution.

Problem Statement: Complex tasks often require coordination among multiple specialized AI agents, but managing their interactions, resolving conflicts, and ensuring efficient execution is challenging. Your task is to create a platform that decomposes tasks, assigns them to specialized agents (e.g., for NLP, vision, reasoning), coordinates their communication, resolves conflicts, and monitors performance. The system should support dynamic agent scaling and provide insights into task execution efficiency.

Steps: - Design task decomposition and assignment algorithms. - Implement inter-agent communication protocols. - Create conflict resolution mechanisms for competing agent outputs. - Build a scheduling system for agent execution. - Develop monitoring and analytics for task performance. - Include visualization of agent interactions and task progress.

Suggested Data Requirements: - Synthetic or real-world multi-task datasets (e.g., task graphs, workflows). - Agent capability profiles (e.g., skills, performance metrics). - Communication logs and conflict resolution benchmarks. - Performance metrics for task completion (e.g., time, accuracy).

Themes: Agentic AI, Multi-Agent Systems

The steps and data requirements outlined above are intended solely as reference points to assist you in conceptualizing your solution.

PRD (Product Requirements Document)

Product Vision and Goals

The Multi-Agent Coordination Platform aims to streamline complex task execution by orchestrating specialized AI agents, reducing task completion time by 50% and improving outcome accuracy by 25%. Goals include supporting diverse tasks (e.g., research, planning, analysis), enabling seamless agent collaboration, and providing transparent analytics for enterprise and research use cases, such as automated R&D pipelines or customer service workflows.

Target Audience and Stakeholders

- **Primary Users**: AI developers, enterprise automation teams, researchers.
- **Stakeholders**: Agent developers (for capability integration), business analysts (for task outcomes), IT teams (for deployment).
- **Personas**:
  - An AI researcher coordinating agents for scientific literature analysis.
  - An enterprise automation lead optimizing a customer support pipeline.

Key Features and Functionality

- **Task Decomposition**: Break complex tasks into sub-tasks for agent assignment.
- **Agent Registry**: Maintain profiles of agent capabilities (e.g., NLP, image processing).
- **Communication**: Enable inter-agent messaging via a pub-sub model.
- **Conflict Resolution**: Resolve competing outputs (e.g., via voting, prioritization).
- **Scheduling**: Dynamically assign tasks to agents based on availability and skills.
- **Monitoring/Analytics**: Track task progress, agent performance, and bottlenecks.
- **Visualization**: Display task graphs and agent interactions interactively.

Business Requirements

- Integration with agent frameworks (e.g., AutoGen, LangChain).
- Freemium model: Basic coordination free, premium for advanced analytics and scaling.
- Export task plans and analytics as JSON/CSV for integration with tools like Jira.

Success Metrics

- **Efficiency**: >50% reduction in task completion time compared to manual coordination.
- **Accuracy**: >90% task outcome alignment with ground truth.
- **Adoption**: >20 agent types integrated per deployment.
- **User Satisfaction**: NPS >80.

Assumptions, Risks, and Dependencies

- **Assumptions**: Access to pre-trained agent models (e.g., Hugging Face, Llama).
- **Risks**: Agent miscommunication leading to errors; mitigate with robust conflict resolution.
- **Dependencies**: Synthetic task datasets, messaging systems (RabbitMQ), monitoring tools (Prometheus).

Out of Scope

- Developing new AI agents from scratch.
- Real-time human-agent collaboration.

FRD (Functional Requirements Document)

System Modules and Requirements

1. **Task Decomposition Module (FR-001)**:
    - **Input**: Complex task (e.g., â€œAnalyze market trends and predict salesâ€).
    - **Functionality**: Use LLM to decompose into sub-tasks (e.g., â€œsearch trends,â€ â€œrun forecastâ€) via prompt engineering.
    - **Output**: JSON task graph with dependencies (e.g., {â€œtask_idâ€: 1, â€œdescriptionâ€: â€œsearch trendsâ€, â€œdepends_onâ€: []}).
    - **Validation**: Ensure sub-tasks cover full task scope; validate via user feedback.
2. **Agent Registry Module (FR-002)**:
    - **Input**: Agent profiles (e.g., {â€œidâ€: â€œnlp_agentâ€, â€œskillsâ€: [â€œtext_analysisâ€], â€œendpointâ€: â€œhttp://â€¦â€}).
    - **Functionality**: Store and query agent capabilities in MongoDB; match tasks to agents using cosine similarity.
    - **Output**: List of suitable agents per sub-task.
    - **Validation**: Verify agent availability and skill relevance.
3. **Communication Module (FR-003)**:
    - **Input**: Sub-task assignments, agent endpoints.
    - **Functionality**: Enable pub-sub messaging via RabbitMQ; broadcast task updates and receive outputs.
    - **Output**: Message logs (e.g., {â€œtask_idâ€: 1, â€œagentâ€: â€œnlp_agentâ€, â€œoutputâ€: â€œtrend dataâ€}).
    - **Validation**: Ensure message delivery within 1s.
4. **Conflict Resolution Module (FR-004)**:
    - **Input**: Multiple agent outputs for a task (e.g., conflicting predictions).
    - **Functionality**: Resolve via majority voting or LLM-based prioritization (prompt: â€œSelect best output: {outputs} â€).
    - **Output**: Single resolved output with confidence score.
    - **Validation**: Check resolution aligns with task goals (>90% accuracy).
5. **Scheduling Module (FR-005)**:
    - **Input**: Task graph, agent availability.
    - **Functionality**: Use OR-Tools to schedule tasks, optimizing for load balancing and deadlines.
    - **Output**: Execution plan (e.g., {â€œtask_idâ€: 1, â€œagentâ€: â€œnlp_agentâ€, â€œstart_timeâ€: â€œ2025-08-27T08:00â€}).
    - **Validation**: Ensure no deadlocks or resource over-allocation.
6. **Monitoring/Analytics Module (FR-006)**:
    - **Input**: Task execution logs, agent performance metrics.
    - **Functionality**: Track KPIs (e.g., completion time, error rate) via Prometheus; generate analytics with ELK Stack.
    - **Output**: Dashboard with task progress and agent efficiency.
    - **Validation**: Cross-check metrics with logs for consistency.
7. **Visualization Module (FR-007)**:
    - **Input**: Task graph, execution logs.
    - **Functionality**: Render task dependencies and agent interactions using vis.js.
    - **Output**: Interactive HTML/JS visualization.
    - **Validation**: Ensure graph matches task structure.

Interfaces and Integrations

- **UI**: Web dashboard (React) for task input, agent management, and analytics.
- **API**: RESTful endpoints (e.g., POST /coordinate, GET /analytics) with JSON payloads.
- **Data Flow**: Input task -> Decompose -> Assign agents -> Communicate -> Resolve conflicts -> Schedule -> Monitor -> Visualize.
- **Integrations**: LangChain for decomposition, RabbitMQ for messaging, Prometheus for monitoring, vis.js for visualization.

Error Handling and Validation

- **Invalid Task**: Prompt user for clarification if decomposition fails.
- **Agent Failure**: Reassign task to backup agent; log failure.
- **Tests**: Unit tests for decomposition (90% coverage), E2E tests for full coordination pipeline.

NFRD (Non-Functional Requirements Document)

Performance Requirements

- **Latency**: <1s for inter-agent messaging; <5min for task coordination (10 sub-tasks).
- **Throughput**: 100 concurrent tasks across 50 agents.

Scalability and Availability

- **Scalability**: Kubernetes for agent scaling; auto-scale based on task load.
- **Availability**: 99.9% uptime; redundant RabbitMQ brokers.

Security and Privacy

- **Data Privacy**: Encrypt task data and messages (AES-256); delete after completion.
- **Authentication**: OAuth2 for API access; role-based access for dashboard (e.g., admin, user).
- **Compliance**: GDPR for task data, audit logs for agent actions.

Reliability and Maintainability

- **Error Rate**: <1% task coordination failures.
- **Code Quality**: Modular design, 85% test coverage, CI/CD with GitHub Actions.
- **Monitoring**: Prometheus for latency and error tracking, Grafana for dashboards.

Usability and Accessibility

- **UI/UX**: Responsive React dashboard, WCAG 2.1 AA compliance (e.g., keyboard navigation).
- **Documentation**: Swagger API docs, user guides with example task flows.

Environmental Constraints

- **Deployment**: Cloud-agnostic (AWS, GCP, Azure) or on-prem with Docker.
- **Cost**: Optimize for <0.05 USD per task coordination.

AD (Architecture Diagram)

+â€”â€”â€”â€”â€”â€”â€”+ | User Interface | (React: Task Input, Agent Management, Analytics Dashboard) +â€”â€”â€”â€”â€”â€”â€”+ | v +â€”â€”â€”â€”â€”â€”â€”+ | API Gateway | (FastAPI: Endpoints for Coordination, Analytics) +â€”â€”â€”â€”â€”â€”â€”+ / |
v v v +â€”â€”â€”â€”â€”â€”â€”+ +â€”â€”â€”â€”â€”â€”â€”+ +â€”â€”â€”â€”â€”â€”â€”+ | Task Decomposer | | Agent Communicator | | Conflict Resolver | | (LangChain LLM) | | (RabbitMQ) | | (LLM, Voting) | +â€”â€”â€”â€”â€”â€”â€”+ +â€”â€”â€”â€”â€”â€”â€”+ +â€”â€”â€”â€”â€”â€”â€”+ | ^ v | +â€”â€”â€”â€”â€”â€”â€”+ | | Scheduler | <â€”â€”-+ | (OR-Tools) | +â€”â€”â€”â€”â€”â€”â€”+ | | | v | +â€”â€”â€”â€”â€”â€”â€”+ | | Monitoring/Visualization | <â€”+ | (Prometheus, vis.js) | +â€”â€”â€”â€”â€”â€”â€”+ text## HLD (High Level Design)

- **Components**:
  - **Frontend**: React with Redux for state management, vis.js for task graph visualization.
  - **Backend**: FastAPI for APIs, Celery for async task coordination.
  - **AI Layer**: LangChain for task decomposition and conflict resolution (e.g., Llama-3-8B).
  - **Communication**: RabbitMQ for pub-sub messaging between agents.
  - **Scheduling**: OR-Tools for task scheduling and load balancing.
  - **Monitoring/Visualization**: Prometheus for metrics, ELK Stack for analytics, vis.js for task graphs.
- **Design Patterns**:
  - **Agent Pattern**: LangChain for orchestrating agent workflows.
  - **Pub-Sub**: RabbitMQ for asynchronous agent communication.
  - **Scheduler**: Constraint-based task assignment with OR-Tools.
- **Data Management**:
  - **Sources**: Synthetic task datasets (e.g., task graphs with dependencies), agent capability profiles.
  - **Storage**: MongoDB for agent registry and task logs, Redis for session state.
- **Security Design**:
  - JWT for API authentication.
  - AES-256 encryption for task data and messages.
  - Role-based access for agent management.
- **High-Level Flow**:
  1. User submits complex task.
  2. Decompose task into sub-tasks.
  3. Assign sub-tasks to agents based on capabilities.
  4. Facilitate agent communication and resolve conflicts.
  5. Schedule execution and monitor progress.
  6. Visualize task graph and analytics.

LLD (Low Level Design)

- **Task Decomposition**:
  - Prompt: `"Decompose task into sub-tasks: {task}"`.
  - Parse: `tasks = llm.generate_json(prompt)` (e.g., [{â€œidâ€: 1, â€œdescâ€: â€œsearch trendsâ€, â€œdepends_onâ€: []}]).
  - Graph: `task_graph = nx.DiGraph(); task_graph.add_edges_from([(t["id"], t["depends_on"]) for t in tasks])`.
- **Agent Registry**:
  - Store: `mongo_db.agents.insert_one({"id": "nlp_agent", "skills": ["text_analysis"], "endpoint": "http://..."})`.
  - Match: `similarity = cosine_sim(task_embedding, agent_skills_embedding)`; select top-3 agents.
- **Communication**:
  - Publish: `rabbitmq.publish("task_queue", {"task_id": 1, "data": task_data})`.
  - Subscribe: `rabbitmq.consume("agent_output", callback=process_output)`.
- **Conflict Resolution**:
  - Voting: `best_output = max(outputs, key=lambda x: sum(x["score"] for x in agent_votes))`.
  - LLM: `prompt = "Select best output: {outputs}\nCriteria: {task_goals}"; resolved = llm.generate_json(prompt)`.
- **Scheduling**:
  - Model: `model = cp_model.CpModel(); model.AddConstraint(agent_load < max_load)`.
  - Solve: `solver = cp_model.CpSolver(); solver.Solve(model)`.
- **Monitoring**:
  - Metrics: `prometheus.Counter("task_completion", ...).inc()`.
  - Analytics: `elk.index({"task_id": id, "latency": time_taken})`.
- **Visualization**:
  - Render: `viz_data = {"nodes": tasks, "edges": dependencies}; vis_js.Network(container, viz_data)`.

Pseudocode

```python class MultiAgentPlatform: def **init**(self): self.llm = LangChainLLM(â€œmeta-llama/Llama-3-8bâ€) self.registry =
```

MongoDBClient(uri=â€œmongodb://localhost:27017â€) self.mq = RabbitMQClient(brokers=â€œlocalhost:5672â€) self.scheduler = ORTools() self.monitor = PrometheusClient() self.viz = VisJS() self.state = RedisClient()

```
def decompose_task(self, task):
    prompt = f"Decompose task into sub-tasks: {task}"
    tasks = self.llm.generate_json(prompt)
    task_graph = nx.DiGraph()
    for t in tasks:
        task_graph.add_node(t["id"], desc=t["desc"])
        task_graph.add_edges_from([(t["id"], dep) for dep in t["depends_on"]])
    return tasks, task_graph

def assign_agents(self, tasks):
    assignments = []
    for task in tasks:
        task_emb = self.llm.embed(task["desc"])
        agents = self.registry.find({"skills": {"$in": task["skills"]}})
        scores = [cosine_sim(task_emb, a["skills_emb"]) for a in agents]
        assignments.append({"task": task, "agent": agents[scores.index(max(scores))]})
    return assignments

def communicate(self, assignments):
    outputs = []
    for a in assignments:
        self.mq.publish("task_queue", {"task_id": a["task"]["id"], "data": a["task"]["desc"]})
        output = self.mq.consume(f"output_{a['agent']['id']}")
        outputs.append({"task_id": a["task"]["id"], "output": output, "agent": a["agent"]["id"]})
    return outputs

def resolve_conflicts(self, outputs):
    grouped = groupby(outputs, key=lambda x: x["task_id"])
    resolved = []
    for task_id, task_outputs in grouped:
        if len(task_outputs) > 1:
            prompt = f"Select best output for task {task_id}: {task_outputs}"
            best = self.llm.generate_json(prompt)
            resolved.append({"task_id": task_id, "output": best, "confidence": best["score"]})
        else:
            resolved.append(task_outputs[0])
    return resolved

def schedule(self, tasks, assignments):
    model = self.scheduler.CpModel()
    for a in assignments:
        model.AddConstraint(a["agent"]["load"] < a["agent"]["max_load"])
    model.Minimize(sum(t["duration"] for t in tasks))
    solver = self.scheduler.CpSolver()
    solver.Solve(model)
    return [{"task_id": a["task"]["id"], "agent": a["agent"]["id"], "start": solver.Value(a["start"])} for a in assignments]

def monitor(self, task_id, outputs):
    metrics = {"task_id": task_id, "latency": time.time() - start_time, "errors": len([o for o in outputs if o["status"] == "error"])}
    self.monitor.Counter("task_metrics").labels(task_id).inc()
    self.monitor.push(metrics)
    return metrics

def visualize(self, task_graph):
    viz_data = {"nodes": [{"id": n, "label": task_graph.nodes[n]["desc"]} for n in task_graph.nodes],
                "edges": [{"from": e[0], "to": e[1]} for e in task_graph.edges]}
    return self.viz.Network(viz_data).to_html()

def coordinate(self, task):
    tasks, task_graph = self.decompose_task(task)
    assignments = self.assign_agents(tasks)
    schedule = self.schedule(tasks, assignments)
    outputs = self.communicate(assignments)
    resolved = self.resolve_conflicts(outputs)
    metrics = self.monitor(task["id"], resolved)
    viz = self.visualize(task_graph)
    return {"tasks": tasks, "schedule": schedule, "outputs": resolved, "metrics": metrics, "viz": viz}
```