# 140509\_26.md

## README

1. Intelligent Task Routing System

**Summary**: Develop an AI-driven system to intelligently route tasks to appropriate AI agents or human workers based on task complexity, urgency, and resource availability.

**Problem Statement**: Efficient task allocation in hybrid human-AI environments is critical for optimizing productivity but is challenging due to varying task complexities and resource constraints. Your task is to create a system that classifies tasks, evaluates agent/worker capabilities, and optimizes routing decisions to minimize completion time and maximize accuracy. The system should adapt to dynamic workloads, provide real-time monitoring, and support enterprise integration.

**Steps**: - Design task classification algorithms to assess complexity and urgency. - Implement capability matching for AI agents and human workers. - Create optimization algorithms for task routing. - Build real-time monitoring and feedback loops. - Develop integration with enterprise task management systems. - Include visualization of routing decisions and performance metrics.

**Suggested Data Requirements**: - Task datasets with attributes (e.g., complexity, urgency, type). - Agent/worker profiles (e.g., skills, availability, performance history). - Historical routing data for optimization benchmarks. - Performance metrics (e.g., completion time, accuracy).

**Themes**: Agentic AI, Optimization

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 Intelligent Task Routing System aims to optimize task allocation in hybrid AI-human environments, reducing task completion time by 30% and improving assignment accuracy by 25%. Goals include supporting diverse enterprise tasks (e.g., customer support, data annotation), ensuring seamless integration with existing systems, and providing transparent analytics for process efficiency, enabling organizations to scale operations effectively.

### Target Audience and Stakeholders

* **Primary Users**: Operations managers, team leads, IT administrators.
* **Stakeholders**: Employees (for task execution), AI developers (for agent integration), executives (for performance insights).
* **Personas**:
  + A customer support manager routing tickets to agents or chatbots.
  + An IT administrator optimizing data labeling tasks for AI training.

### Key Features and Functionality

* **Task Classification**: Categorize tasks by complexity and urgency using AI.
* **Capability Matching**: Match tasks to AI agents or humans based on skills and availability.
* **Routing Optimization**: Assign tasks to minimize completion time and costs.
* **Monitoring**: Track task status, agent performance, and bottlenecks in real-time.
* **Integration**: Connect with enterprise systems (e.g., Jira, Zendesk) via APIs.
* **Visualization**: Display task flows and performance metrics interactively.

### Business Requirements

* Support integration with 5+ enterprise systems (e.g., Jira, ServiceNow, Salesforce).
* Freemium model: Basic routing free, premium for advanced optimization and analytics.
* Export routing plans and metrics as JSON/CSV for reporting.

### Success Metrics

* **Efficiency**: >30% reduction in average task completion time.
* **Accuracy**: >95% task assignment accuracy (correct agent/worker).
* **Adoption**: 500+ tasks routed daily per deployment.
* **User Satisfaction**: NPS >75.

### Assumptions, Risks, and Dependencies

* **Assumptions**: Access to task metadata and agent/worker profiles.
* **Risks**: Misclassification of task complexity; mitigate with robust AI models and feedback loops.
* **Dependencies**: Datasets (e.g., OR scheduling benchmarks), libraries (BERT, PuLP), messaging systems (Kafka).

### Out of Scope

* Developing new AI agents for task execution.
* Multi-language task processing initially.

## FRD (Functional Requirements Document)

### System Modules and Requirements

1. **Task Classification Module (FR-001)**:
   * **Input**: Task description (e.g., “resolve customer complaint”).
   * **Functionality**: Use BERT to classify tasks by complexity (low/medium/high) and urgency (e.g., SLA deadlines).
   * **Output**: JSON with task attributes (e.g., {“id”: 1, “complexity”: “high”, “urgency”: “1h”}).
   * **Validation**: Achieve >90% classification accuracy against labeled datasets.
2. **Capability Matching Module (FR-002)**:
   * **Input**: Task attributes, agent/worker profiles.
   * **Functionality**: Match tasks to agents/workers using cosine similarity on skill embeddings; consider availability.
   * **Output**: List of candidate agents/workers (e.g., {“agent\_id”: “nlp\_bot”, “score”: 0.95}).
   * **Validation**: Ensure top matches align with task requirements.
3. **Routing Optimization Module (FR-003)**:
   * **Input**: Task list, candidate matches, constraints (e.g., deadlines).
   * **Functionality**: Use PuLP to optimize assignments, minimizing completion time and cost.
   * **Output**: Assignment plan (e.g., {“task\_id”: 1, “agent\_id”: “nlp\_bot”, “start\_time”: “2025-08-27T08:00”}).
   * **Validation**: Verify assignments meet constraints (e.g., SLA compliance).
4. **Integration Module (FR-004)**:
   * **Input**: Assignment plan, enterprise API specs.
   * **Functionality**: Send tasks to systems (e.g., POST to Jira /issue) via REST APIs.
   * **Output**: Task execution confirmation (e.g., API response code).
   * **Validation**: Check response codes (e.g., 200 OK).
5. **Monitoring Module (FR-005)**:
   * **Input**: Task assignments, execution logs.
   * **Functionality**: Stream metrics (e.g., completion time, errors) via Kafka; alert on delays.
   * **Output**: Real-time dashboard with KPIs.
   * **Validation**: Cross-check metrics with logs for consistency.
6. **Visualization Module (FR-006)**:
   * **Input**: Assignment plan, performance metrics.
   * **Functionality**: Render task flows and agent loads using vis.js.
   * **Output**: Interactive HTML/JS visualization.
   * **Validation**: Ensure visualization matches assignment data.

### Interfaces and Integrations

* **UI**: Web dashboard (React) for task input, routing review, and monitoring.
* **API**: RESTful endpoints (e.g., POST /route, GET /metrics) with JSON payloads.
* **Data Flow**: Input task -> Classify -> Match -> Optimize -> Integrate -> Monitor -> Visualize.
* **Integrations**: BERT for classification, PuLP for optimization, Kafka for streaming, vis.js for visualization, Jira/Zendesk APIs.

### Error Handling and Validation

* **Invalid Task**: Prompt clarification for ambiguous inputs.
* **Agent Unavailability**: Reassign to next-best candidate; log issue.
* **Tests**: Unit tests for classification (90% coverage), E2E tests for routing pipeline.

## NFRD (Non-Functional Requirements Document)

### Performance Requirements

* **Latency**: <100ms for task routing decisions; <1min for 100-task batch.
* **Throughput**: 1,000 tasks/hour on standard hardware (16GB RAM, 4 vCPUs).

### Scalability and Availability

* **Scalability**: Kubernetes for scaling routing services; auto-scale based on task volume.
* **Availability**: 99.9% uptime; redundant Kafka brokers.

### Security and Privacy

* **Data Privacy**: Encrypt task data (AES-256); anonymize sensitive fields (e.g., customer info).
* **Authentication**: OAuth2 for API access; role-based access for dashboard.
* **Compliance**: GDPR for task data, audit logs for routing decisions.

### Reliability and Maintainability

* **Error Rate**: <1% routing errors.
* **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., screen reader support).
* **Documentation**: Swagger API docs, user guides with routing examples.

### Environmental Constraints

* **Deployment**: Cloud-agnostic (AWS, GCP, Azure) or on-prem with Docker.
* **Cost**: Optimize for <0.01 USD per task routing.

## AD (Architecture Diagram)

+---------------------+  
| User Interface | (React: Task Input, Routing Dashboard, Metrics Viewer)  
+---------------------+  
 |  
 v  
+---------------------+  
| API Gateway | (FastAPI: Endpoints for Routing, Monitoring)  
+---------------------+  
 / | \  
 v v v  
+---------------------+ +---------------------+ +---------------------+  
| Task Classifier | | Capability Matcher | | Routing Optimizer |  
| (BERT) | | (SentenceTransformers)| (PuLP) |  
+---------------------+ +---------------------+ +---------------------+  
 | ^  
 v |  
+---------------------+ |  
| Integration Layer | <-------+  
| (REST APIs) |  
+---------------------+ |  
 | |  
 v |  
+---------------------+ |  
| Monitoring/Visualization | <---+  
| (Kafka, vis.js, Grafana)|  
+---------------------+

## HLD (High Level Design)

* **Components**:
  + **Frontend**: React with Redux for state management, vis.js for task flow visualization.
  + **Backend**: FastAPI for APIs, Celery for async task processing.
  + **AI/ML**: BERT for task classification, Sentence Transformers for capability matching, PuLP for optimization.
  + **Integration**: REST APIs for enterprise systems (e.g., Jira, Zendesk).
  + **Monitoring/Visualization**: Kafka for streaming metrics, Grafana for dashboards, vis.js for task flows.
* **Design Patterns**:
  + **Pipeline**: Sequential flow (classify -> match -> optimize -> integrate -> monitor).
  + **Strategy**: Adaptive classification based on task type.
  + **Observer**: Real-time updates via Kafka streams.
* **Data Management**:
  + **Sources**: OR scheduling benchmarks, synthetic task datasets (e.g., tasks with complexity/urgency).
  + **Storage**: MongoDB for task and agent profiles, Redis for caching assignments.
* **Security Design**:
  + JWT for API authentication.
  + AES-256 encryption for task data.
  + Role-based access (e.g., admin for monitoring, user for task input).
* **High-Level Flow**:
  1. Receive task input.
  2. Classify task complexity and urgency.
  3. Match task to agents/workers.
  4. Optimize routing assignments.
  5. Integrate with enterprise systems.
  6. Monitor and visualize performance.

## LLD (Low Level Design)

* **Task Classification**:
  + Preprocess: task\_tokens = tokenizer(task\_desc, return\_tensors="pt").
  + Classify: outputs = bert\_model(\*\*task\_tokens); complexity = softmax(outputs.logits)[0].
  + Output: {"id": task\_id, "complexity": "high", "urgency": "1h"}.
* **Capability Matching**:
  + Embed: task\_emb = sentence\_transformer.encode(task\_desc); agent\_emb = sentence\_transformer.encode(agent\_skills).
  + Match: scores = [cosine\_sim(task\_emb, a\_emb) for a in agents]; select top-3.
* **Routing Optimization**:
  + Model: model = pulp.LpProblem("Routing", pulp.LpMinimize); model += sum(task\_duration \* assignment\_var).
  + Constraints: model += (agent\_load <= max\_load); model += (task\_deadline <= sla).
  + Solve: model.solve().
* **Integration**:
  + API Call: response = requests.post("https://api.jira.com/issue", headers={"Authorization": "Bearer {token}"}, json=task\_data).
  + Validate: if response.status\_code != 200: retry\_with\_backoff(max\_attempts=3).
* **Monitoring**:
  + Stream: kafka\_producer.send("metrics\_topic", {"task\_id": id, "completion\_time": time\_taken}).
  + Alerts: if time\_taken > sla: send\_alert().
* **Visualization**:
  + Render: viz\_data = {"nodes": tasks, "edges": assignments}; vis\_js.Network(container, viz\_data).

## Pseudocode

class TaskRouter:  
 def \_\_init\_\_(self):  
 self.classifier = BertModel.from\_pretrained("bert-base-uncased")  
 self.matcher = SentenceTransformer("all-MiniLM-L6-v2")  
 self.optimizer = pulp  
 self.kafka = KafkaProducer(brokers="localhost:9092")  
 self.db = MongoDBClient(uri="mongodb://localhost:27017")  
 self.viz = VisJS()  
  
 def classify\_task(self, task\_desc):  
 tokens = tokenizer(task\_desc, return\_tensors="pt")  
 outputs = self.classifier(\*\*tokens)  
 complexity = softmax(outputs.logits)[0]  
 urgency = parse\_deadline(task\_desc) # Custom parsing logic  
 return {"id": task\_id, "complexity": complexity.argmax(), "urgency": urgency}  
  
 def match\_capabilities(self, task):  
 task\_emb = self.matcher.encode(task["desc"])  
 agents = self.db.agents.find()  
 scores = [cosine\_sim(task\_emb, self.matcher.encode(a["skills"])) for a in agents]  
 candidates = sorted(zip(agents, scores), key=lambda x: x[1], reverse=True)[:3]  
 return [{"agent\_id": a["id"], "score": s} for a, s in candidates]  
  
 def optimize\_routing(self, tasks, candidates):  
 model = self.optimizer.LpProblem("Routing", self.optimizer.LpMinimize)  
 assignments = [(t["id"], c["agent\_id"]) for t in tasks for c in candidates]  
 vars = self.optimizer.LpVariable.dicts("assign", assignments, cat="Binary")  
 model += sum(t["duration"] \* vars[(t["id"], c["agent\_id"])] for t in tasks for c in candidates)  
 for t in tasks:  
 model += sum(vars[(t["id"], c["agent\_id"])] for c in candidates) == 1  
 for a in agents:  
 model += sum(vars[(t["id"], a["id"])] for t in tasks) <= a["max\_load"]  
 model.solve()  
 return [{"task\_id": t, "agent\_id": a} for (t, a), v in vars.items() if v.value() == 1]  
  
 def integrate(self, assignments, api\_specs):  
 results = []  
 for a in assignments:  
 response = requests.post(api\_specs[a["agent\_id"]]["url"], json=a["task\_data"])  
 results.append({"task\_id": a["task\_id"], "status": response.status\_code})  
 if response.status\_code != 200:  
 self.retry\_with\_backoff(a, api\_specs)  
 return results  
  
 def monitor(self, assignments):  
 metrics = [{"task\_id": a["task\_id"], "time": time.time() - a["start"]} for a in assignments]  
 for m in metrics:  
 self.kafka.send("metrics\_topic", m)  
 if m["time"] > m["sla"]:  
 self.send\_alert(m)  
 return metrics  
  
 def visualize(self, assignments):  
 viz\_data = {"nodes": [{"id": a["task\_id"], "label": a["task\_desc"]} for a in assignments],  
 "edges": [{"from": a["task\_id"], "to": a["agent\_id"]} for a in assignments]}  
 return self.viz.Network(viz\_data).to\_html()  
  
 def route\_task(self, task\_desc):  
 task = self.classify\_task(task\_desc)  
 candidates = self.match\_capabilities(task)  
 assignments = self.optimize\_routing([task], candidates)  
 results = self.integrate(assignments, api\_specs)  
 metrics = self.monitor(assignments)  
 viz = self.visualize(assignments)  
 return {"task": task, "assignments": assignments, "results": results, "metrics": metrics, "viz": viz}

```

### Summaries for Remaining Files (140509\_20.md to 140509\_25.md, 140509\_27.md to 140509\_29.md)

To ensure completeness while avoiding redundancy, here are summaries of the remaining Markdown files, each following the same structure as **140509\_26.md** (README, PRD, FRD, NFRD, AD, HLD, LLD, pseudocode). I can provide the full content for any specific file upon request.

* **140509\_20.md** (Knowledge Graph Enhanced Q&A System):
  + **Purpose**: Combines knowledge graphs and generative AI for multi-hop Q&A with reasoning and confidence scoring.
  + **Key Details**: Uses spaCy/REBEL for KG construction, Neo4j for Cypher queries, Hugging Face LLMs for responses, vis.js for visualization. Targets <3s query latency, 99.5% uptime, F1-score >0.85 on HotpotQA.
  + **Pseudocode**: KGQASystem class with build\_kg (NER/relation extraction) and process\_query (Cypher + LLM).
* **140509\_21.md** (Model Quantization and Fine-tuning Platform):
  + **Purpose**: Automates quantization (INT8/INT4) and fine-tuning (LoRA/QLoRA) for edge LLMs.
  + **Key Details**: Uses PyTorch for quantization, Hugging Face for models, OR-Tools for Pareto ranking. Targets 4x-8x size reduction, <30min quantization, <50ms/token inference.
  + **Pseudocode**: QuantFinePlatform class with quantize, fine\_tune, benchmark.
* **140509\_22.md** (Multi-Step Research Assistant Agent):
  + **Purpose**: Automates multi-step research with web/doc retrieval and report synthesis.
  + **Key Details**: Uses LangChain for task decomposition, SerpAPI/PyPDF2 for retrieval, FEVER for fact-checking. Targets <8min cycle, >90% fact accuracy.
  + **Pseudocode**: ResearchAgent class with decompose, retrieve, reason, synthesize.
* **140509\_23.md** (Autonomous Data Analysis Agent):
  + **Purpose**: Automates EDA with profiling, statistical testing, and visualization.
  + **Key Details**: Uses Pandas for profiling, SciPy for stats, Plotly for visuals, Hugging Face LLM for narratives. Targets <5min for 100k rows, >90% insight relevance.
  + **Pseudocode**: DataAnalysisAgent class with profile, detect\_patterns, test\_stats, visualize.
* **140509\_24.md** (Business Process Automation Agent):
  + **Purpose**: Automates workflows via process mining, rule generation, and optimization.
  + **Key Details**: Uses PM4Py for mining, OR-Tools for optimization, Kafka for monitoring, bpmn-js for visualization. Targets >20% cycle time reduction, 99% uptime.
  + **Pseudocode**: ProcessAutomationAgent class with mine\_process, generate\_rules, optimize, monitor.
* **140509\_25.md** (Multi-Agent Coordination Platform):
  + **Purpose**: Orchestrates specialized AI agents for complex tasks.
  + **Key Details**: Uses LangChain for decomposition, RabbitMQ for messaging, Prometheus for monitoring, vis.js for visualization. Targets <1s message latency, 99.9% uptime.
  + **Pseudocode**: MultiAgentPlatform class with decompose\_task, assign\_agents, coordinate.
* **140509\_27.md** (Mixture of Experts Model Implementation):
  + **Purpose**: Implements MoE architecture for efficient LLM inference.
  + **Key Details**: Uses PyTorch for MoE layers, Fairseq for training, GLUE/MNLI datasets. Targets >50% FLOPs reduction, stable convergence.
  + **Pseudocode**: MoEModel class with forward (gating + experts), train\_moe.
* **140509\_28.md** (Explainable AI Dashboard for Complex Models):
  + **Purpose**: Provides interpretable insights for black-box models.
  + **Key Details**: Uses SHAP/LIME for explanations, Plotly for visuals, DiCE for counterfactuals. Targets <1min explanation, WCAG 2.1 AA compliance.
  + **Pseudocode**: XAI\_Dashboard class with explain\_model (SHAP + visuals).
* **140509\_29.md** (Custom Domain-Specific Model Architecture):
  + **Purpose**: Framework for neural architecture search (NAS) to build domain-specific models.
  + **Key Details**: Uses NASLib for search, Optuna for hyperparameter tuning, CIFAR-10/MedMNIST datasets. Targets <24hr search, reproducible architectures.
  + **Pseudocode**: CustomArch class with nas\_search (evolve population).

### Instructions to Create the ZIP File Locally

Since you’ve indicated issues with the ZIP file being corrupt or not extractable, and previous responses provided conceptual ZIP content rather than a binary file due to text-based limitations, here’s how to create 140509\_20\_to\_29.zip locally with **140509\_26.md** and other files:

1. **Save Markdown Files**:
   * Copy the content of **140509\_26.md** above into a file named 140509\_26.md using a text editor (e.g., VS Code, Notepad).
   * For other files (140509\_20.md to 140509\_25.md, 140509\_27.md to 140509\_29.md), use previously provided content (e.g., **140509\_22.md**, **140509\_23.md**, **140509\_24.md**, **140509\_25.md** from earlier responses) or request full content for specific files. I can generate them immediately if needed.
   * Save all files in a folder (e.g., task\_routing\_files).
2. **Create the ZIP**:
   * **Windows**: Right-click the folder -> “Send to” -> “Compressed (zipped) folder” to create 140509\_20\_to\_29.zip.
   * **Mac/Linux**: Run zip -r 140509\_20\_to\_29.zip task\_routing\_files/ in the terminal from the folder’s parent directory.
   * This creates a ZIP containing all .md files.
3. **Verify**:
   * Extract the ZIP using a tool like 7-Zip, WinRAR, or unzip to confirm all files are present and readable.
   * Open **140509\_26.md** to verify it includes all sections (up to pseudocode) without cutoff.
4. **Python Script to Automate ZIP Creation** (optional):

* import zipfile  
  import os  
    
  folder = "task\_routing\_files"  
  with zipfile.ZipFile("140509\_20\_to\_29.zip", "w", zipfile.ZIP\_DEFLATED) as zipf:  
   for file in os.listdir(folder):  
   if file.endswith(".md"):  
   zipf.write(os.path.join(folder, file), file)
  + Save **140509\_26.md** (and others) in task\_routing\_files, run the script, and check the generated ZIP.

1. **Alternative Delivery**:
   * If you need a downloadable binary ZIP, I can guide you to host the files on a platform like GitHub or a file-sharing service (e.g., Google Drive).
   * Alternatively, I can provide a JSON structure with all file contents, which you can convert to a ZIP programmatically:
   * {  
      "files": [  
      {"name": "140509\_26.md", "content": "..."},  
      // Add other files  
      ]  
     }