Intelligent Data Analyst Agent Architecture (MCP-Based)

Objective

To design an intelligent, modular, data-agnostic agent capable of processing diverse traffic-related data using a dynamic suite of up to 100 MCP (Model Context Protocol) tools. The system supports high-throughput workloads, tool chaining, visualization, and secure API-based integration into larger architectures.

🧱 Core Components

1. Agent API Layer

- **Tech**: Python (FastAPI/Flask), Docker
- **Purpose**: Exposes REST endpoints (e.g., `/analyze`) to accept input data + context
- **Responsibilities**:
- Parse & validate request payloads
- Forward to Router for tool assignment
- Manage response formatting, chaining, and error handling
- Emit job/run events for visualization dashboards

---

2. Router Module

- **Modes**:
- **Rule-Based Routing**: Based on input types, keywords, metadata
- **ML-Based Routing**: Intent classification using ML/LLM
- **Hybrid Routing**: Fast rules + fallback to model-driven dispatch
- **Responsibilities**:
- Decide best-fit MCP tool or tool chain
- Send routing metadata to Dispatcher
- Generate trace identifiers and step metadata for visualization

---

3. Tool Dispatcher

- **Function**: Orchestrates tool invocations
- **Protocols Supported**:
- **REST** (default)
- **gRPC** (for low-latency/high-throughput)
- **Kafka/RabbitMQ** (for async workloads)

• **Responsibilities**:
• Handles retries, timeouts
• Resolves tool endpoint from Tool Registry
• Loads authentication headers + payloads
• Publishes step progress and results for visualization and tracing

---

4. Tool Chaining Manager

• **Purpose**: Executes tool pipelines
• **Approach**:
• DAG-based chaining (e.g., Anomaly Detection -> Clustering)
• Agent-guided dynamic chaining
• **Responsibilities**:
• Manage data hand-off between tools
• Track intermediate results and state
• Report stage transitions to the visualization subsystem

---

5. Tool Registry

• **Storage**: Local JSON file or database
• **Fields**:
• Tool name, task type, supported data types
• Endpoint URL & communication protocol
• Version, metadata, health status
• Visualization metadata: category, icon, color code, owner

---

6. MCP Tool Interface

• **Standardized Schema**:

```
{
  "input": { ... },
  "context": { ... }
}
```

• **Returns**:

```
{
  "status": "success",
  "output": { ... },
  "meta": { ... }
}
```

- **Implementation**:
- Python class wrapper for tool logic
- Exposed via REST/gRPC/Kafka listener
- Emits structured logs, spans, and step metrics for visualization

---

7. Visualization & Monitoring Layer

- **Purpose**: Provide full transparency into the running processes and tool interactions.
- **Components**:
- **Run Service / Jobs API**: Tracks all runs, jobs, steps, and statuses; exposes REST + WebSocket endpoints (e.g., `/v1/runs`, `/ws/runs/{id}`).
- **Dashboard UI**: Shows tool catalog, live runs, DAG visualizer (tool chaining), job progress, and metrics.
- **Tracing**: OpenTelemetry + Jaeger/Tempo for distributed traces.
- **Logs**: ELK/OpenSearch for structured logs (linked to runs).
- **Metrics**: Prometheus + Grafana for latency, throughput, error rate.
- **Lineage**: OpenLineage/Marquez integration for dataset–tool–output provenance.
- **User View**:
- Tool catalog with capabilities, schema, and status.
- Real-time run status, progress bars, ETA, per-step logs, and trace links.
- DAG view showing current pipeline execution flow.

---

8. Communication & Throughput Management

- **Options**:
- REST (development, small-scale)
- gRPC (binary RPCs, high throughput)
- Kafka (buffered, async tasks)
- **Concurrency**:
- Python asyncio / Celery for parallel calls
- K8s for container auto-scaling
- KEDA for queue-based scaling

---

9. Security Layer

- **Auth**:
- JWT-based access control
- HMAC signing for internal tool calls
- **Transport**:
- TLS encryption for REST/gRPC
- Kafka: TLS + SASL
- **Audit Logging**:
- Request, tool, user, timestamp, result status
- Integrated into visualization UI for admin access

10. Observability & Governance

- **Tracing**: OpenTelemetry spans per request and tool
- **Metrics**: Prometheus collectors for latency, throughput, queue lag
- **Logging**: Structured, tenant-aware JSON logs
- **SLOs & Alerts**: Alertmanager for anomalies and health checks
- **Governance**:
- Versioning, tool lifecycle tracking
- Canary releases and shadow runs
- UI displays deprecation notices and tool change logs

🗃️Data & Task Support

Input Types:

- Tabular (CSV, Excel, SQL result)
- Text (incident reports, logs)
- JSON/XML (API or IoT device input)
- Images (traffic cams)
- Geo (GeoJSON, GPS points)

Supported Tasks:

- Anomaly Detection
- Incident Detection
- Time-Series Forecasting
- Descriptive Stats & Comparison
- Classification / Regression
- Clustering & Feature Engineering
- Geospatial Mapping & Analysis

Deployment Notes

- Containerized with Docker for each tool and core module
- Use Docker Compose or Kubernetes for orchestration
- Includes visualization stack (Grafana, Jaeger, ELK) and UI dashboard
- Designed to plug into larger systems as a callable API service
- Future-proofed for more advanced ML planning agents (e.g., LLM planner)

Next Steps

1. Scaffold base API + agent logic

2. Implement router (rule-based first)
3. Add 3–5 MCP tools with REST endpoints
4. Integrate Run Service + WebSocket for real-time progress
5. Add UI layer for visualization (DAG, runs, logs)
6. Package with Docker Compose for local testing
7. Extend with gRPC + Kafka for async cases