ETL Automation Agent Overview

Tasks:

1. **Data Extraction**: Collect data from various sources using the **Data Extractor Agent**.
2. **Data Transformation**: Map extracted data to the required schema using the **Schema Mapper Agent**.
3. **Data Validation**: Validate the transformed data for quality and consistency using the **Data Validator Agent**.
4. **Data Loading**: Load the validated data to the target storage.

Frameworks and Tools:

- **Agents**: crewAI, LangChain for orchestrating task-based agents.
- **Data Indexing**: llamaindex for schema mapping.
- **APIs**: Expose input/output endpoints with Flask/FastAPI.
- **Validation**: groq for rule-based validation.
- **Dataflow Management**: stategraph for process orchestration.
- **Graphs**: langgraph for schema and transformation visualization.

```
from fastapi import FastAPI
from pydantic import BaseModel
from typing import Dict, Any
from langchain.chains import TransformChain
from langchain.tools import BaseTool
from langgraph.stategraph import StateGraph
from crewAI import AgentBuilder
from llamaindex import SimpleKeywordTableIndex
from groq import DataQualityChecker


# Initialize FastAPI application
app = FastAPI()
```

```
# Define input and output schemas
class InputSchema(BaseModel):
data_source: str
parameters: Dict[str, Any]


class OutputSchema(BaseModel):
status: str
message: str
transformed_data: Dict[str, Any]


# Define ETL tasks using state graphs
class ETLProcess:
def __init__(self):
self.state_graph = StateGraph()
self.build_agents()


def build_agents(self):
# Data Extraction Agent
@AgentBuilder.register_agent(name="DataExtractor")
def data_extractor(data_source: str, parameters: Dict[str, Any]):
# Placeholder: Replace with actual data extraction logic
print("Extracting data from", data_source)
return {"raw_data": "Sample extracted data"}


# Schema Mapping Agent
@AgentBuilder.register_agent(name="SchemaMapper")
def schema_mapper(raw_data: Any):
# Placeholder: Replace with actual schema mapping logic
print("Mapping schema for", raw_data)
return {"mapped_data": "Sample mapped data"}


# Data Validation Agent
@AgentBuilder.register_agent(name="DataValidator")
def data_validator(mapped_data: Any):
# Using groq for data quality checks
checker = DataQualityChecker()
if checker.validate(mapped_data):
print("Validation passed for", mapped_data)
return {"validated_data": mapped_data}
```

```
else:
raise ValueError("Validation failed")


# Integrate agents into the state graph
self.state_graph.add_nodes([
("extract", data_extractor),
("map", schema_mapper),
("validate", data_validator)
])
self.state_graph.add_edges([
("extract", "map"),
("map", "validate")
])


def run(self, input_data: InputSchema):
try:
output = self.state_graph.run({"data_source": input_data.data_source, "parameters":
input_data.parameters})
return output["validate"]
except Exception as e:
return {"status": "failure", "message": str(e), "transformed_data": None}


etl_process = ETLProcess()


@app.post("/etl-process", response_model=OutputSchema)
def run_etl_process(input_data: InputSchema):
result = etl_process.run(input_data)
if result.get("status") == "failure":
return result


return OutputSchema(
status="success",
message="ETL process completed successfully",
transformed_data=result.get("validated_data")
)


# Example to expose agents for custom tasks
@app.get("/agents")
```

```
def list_agents():
return {“available_agents”: AgentBuilder.list_agents()}


# Additional hooks for customization if needed
@app.post(“/add-agent”)
def add_custom_agent(name: str, agent_logic: str):
exec(agent_logic) # Dynamically add new agents — secure implementations recommended
return {“status”: “Agent added successfully”, “name”: name}
```

Key Features

1. **Extensibility**: Additional tasks can be added to the agent with modular tools.
2. **API Exposure**: Each ETL task is exposed as an API endpoint (`/extract`, `/transform`, `/validate`).
3. **Automation**: The ETL flow can be triggered via the `/etl` endpoint.