Leveraging LLMs for Agentic Workflow Orchestration with Apache Beam YAML Pipelines

BEAM SUMMIT
NYC 2025

Charles Adetiloye

Lead ML Engineer

About the Presenter

Charles Adetiloye is a Cofounder and Lead AI Engineer at MavenCode. He has well over 18 years of experience building large-scale distributed applications with extensive experience working and consulting with several companies implementing production grade GenAI / Agentic AI / ML platforms.

About MavenCode

MavenCode is an Artificial Intelligence Solutions Company with HQ in Dallas, Texas and a remote delivery workforce across multiple time zones. We do training, product development and consulting services with specializations in:

- Provisioning Scalable AI and ML Infrastructure - OnPrem and In the Cloud
- Development & Production Operationalization of ML platforms - OnPrem and In the Cloud
- Streaming Data Analytics and Edge IoT Model Deployment for Federated Learning
- Building out Agentic AI, LLM and ML pipelines at scale.

Today's Agenda

1. Introduction to agentic workflow orchestration
2. Overview of Apache Beam YAML pipelines
3. Leveraging LLMs for pipeline automation
4. Designing modular orchestration agents
5. Future directions, and Q&A

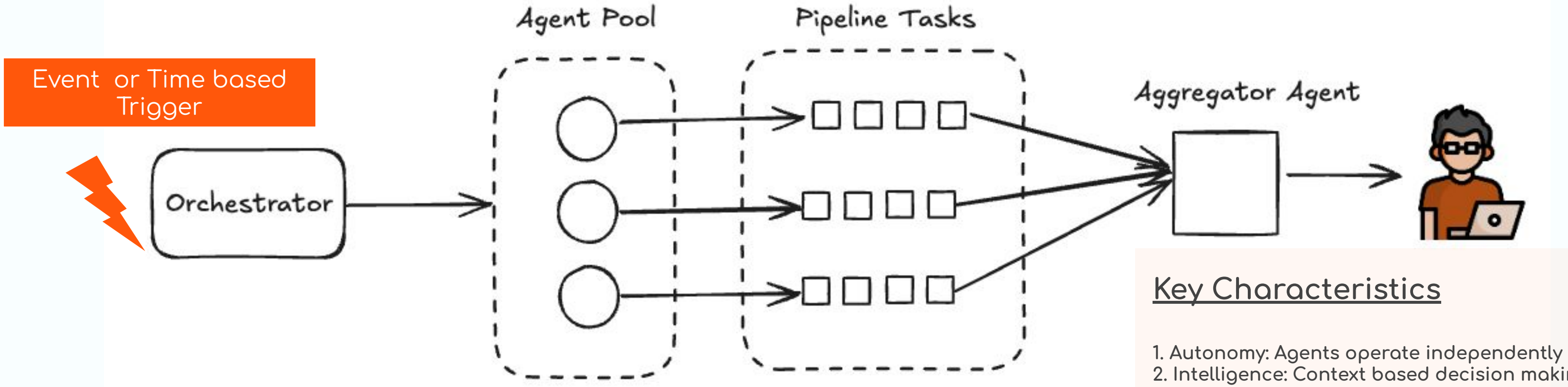
1. Introduction to Agentic Workflow Orchestration

What is Agentic Workflow Orchestration all About?

Agentic workflow orchestration leverages autonomous AI agents to dynamically coordinate, decide, and adapt across complex processes, enabling robust, scalable, intelligent, and resilient operations under varying conditions.

1. Introduction to Agentic Workflow Orchestration

Core Architecture

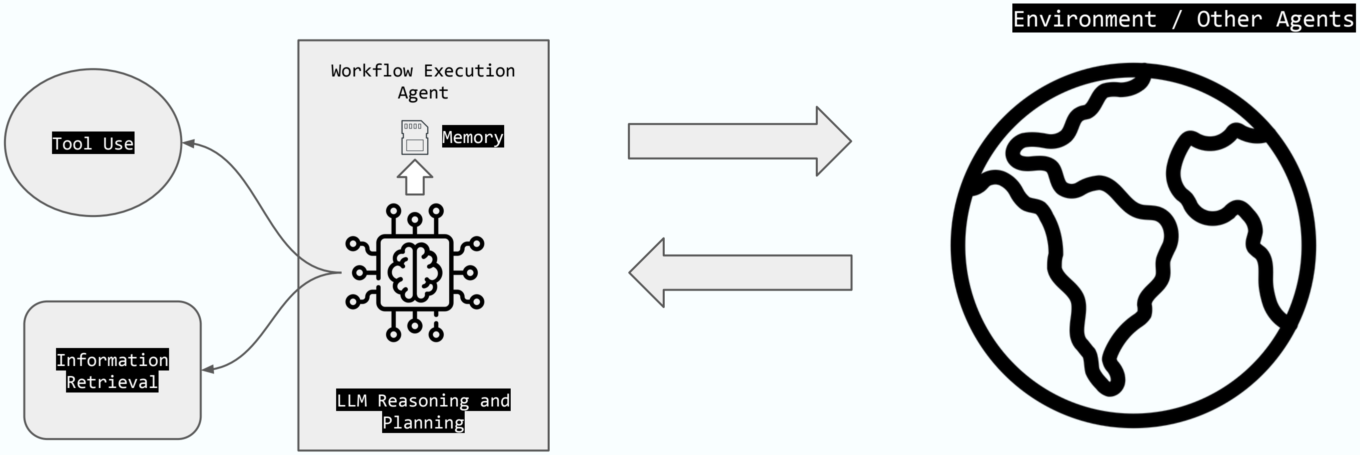


Key Characteristics

1. Autonomy: Agents operate independently
2. Intelligence: Context based decision making
3. Collaboration: Seamless coordination between multiple agents
4. Adaptability: Dynamic response to changing conditions & requirements

1. Introduction to Agentic Workflow Orchestration

Agentic Reasoning + Task Execution



Key Objectives

1. Autonomous Decision-making aligned with Workflow goals
2. Coordinated multi-step task sequencing + resource allocation
3. Continuous monitoring with dynamic error recovery mechanism

1. Introduction to Agentic Workflow Orchestration

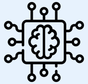
Agentic Execution of Apache Beam Workflow

1

Prompt Input: Was there any system failure
in the observed pumps in
the last 12 hrs

2

MoE RouterAgent:
Expert Selector



3

Expert Agent Pool

- **ETL Agent** – Data Pipeline
- **Analytics Agent** – Aggregation
- **ML Agent** – Inference / Prediction
- **Stream Agent** – Realtime
- **Batch Agent** – Bulk Processing
- **Quality Agent** – Validation

4

Beam YAML Pipeline Jobs Execution

- **ETL_Pipeline.yaml** – Params: {input, output, args}
- **stream_proc.yaml** – Params: Pub/Sub, Windowing
- **ML_feature.yaml** – Params: Feature Eng, Transform
- **validation.yaml** – Params: input, x, y
- **ML_Inference.yaml** – Params: Model,,Batch,

- User submits natural language prompt describing data processing task or requirement
- MoE Router analyzes prompt and selects appropriate expert agents from pool
- Selected expert agents execute corresponding Apache Beam YAML pipeline jobs automatically
- System completes data processing workflow without manual pipeline configuration or coding

2. Overview of Apache Beam YAML

Apache Beam YAML Pipelines

Apache Beam YAML is a declarative syntax for describing Apache Beam pipelines using YAML files. You can use Beam YAML to author and run a Beam pipeline without writing any code. This approach makes data processing pipelines more accessible by eliminating the need to write code in traditional Beam SDK languages.

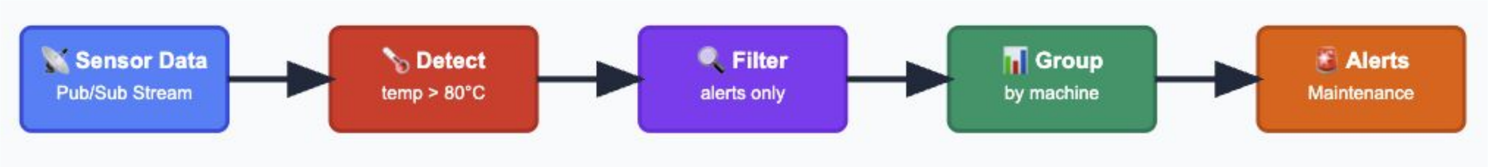
2. Overview of Apache Beam YAML

Key Benefits of Beam YAML pipeline

- **No-code Development:** Create sophisticated data pipelines using only YAML configuration without programming
- **Cloud-native ready:** Deploy instantly to Kubernetes / Google Dataflow with automatic scaling and management
- **Easy maintenance:** Update pipeline logic by editing text files instead of recompiling code
- **Self-documenting code:** YAML declarative nature makes AI-generated pipelines naturally explainable and auditable
- **Template-based generation:** GenAI can easily modify reusable YAML templates for different data scenarios

2. Overview of Apache Beam YAML

Real-time Anomaly Detection and Preemptive Notification



- Read: Stream log data from Manufacturing Plant
- Detect Machines that are overheating
- Filter and keep the identified machines
- Send alert notifications by Machines

2. Overview of Apache Beam YAML

1

```yaml
# Real-time sensor monitoring with anomaly detection and maintenance alerts

pipeline:
  type: chain

  source:
    type: ReadFromPubSub
    config:
      topic: projects/[redacted]/topics/sensor-data
      format: JSON

  transforms:
    # Detect sensor anomalies
    - type: MapToFields
      config:
        language: python
        fields:
          machine_id: machine_id
          temperature: temperature
          pressure: pressure
          vibration: vibration
          # Simple anomaly flags
          temp_alert: "temperature > 80"
          pressure_alert: "pressure > 150"
          vibration_alert: "vibration > 10"
          alert_level: |
            'CRITICAL' if temperature > 90 or pressure > 160
            else 'WARNING' if temperature > 80 or pressure > 140
            else 'NORMAL'

    # Only keep problematic readings
    - type: Filter
      config:
        language: python
        keep: "temp_alert or pressure_alert or vibration_alert"

    # Group by machine and count issues
    - type: Sql
      config:
        query: |
          SELECT
            machine_id,
            COUNT(*) as alert_count,
            MAX(temperature) as max_temp,
            MAX(pressure) as max_pressure,
            COUNT(CASE WHEN alert_level = 'CRITICAL' THEN 1 END) as critical_count
          FROM PCOLLECTION
          GROUP BY machine_id
```

2

```yaml
    # Add maintenance recommendations
    - type: MapToFields
      config:
        language: python
        fields:
          machine_id: machine_id
          alert_count: alert_count
          max_temp: max_temp
          max_pressure: max_pressure
          critical_count: critical_count
          # Simple maintenance logic
          action_needed: |
            'SHUTDOWN' if critical_count > 2
            else 'MAINTENANCE' if alert_count > 5
            else 'MONITOR'

  sink:
    type: WriteToPubSub
    config:
      topic: projects[redacted]/topics/maintenance-alerts
      format: JSON

options:
  streaming: true
  runner: DataflowRunner
```

2. Overview of Apache Beam YAML

1

```
# Real-time sensor monitoring with anomaly detection and maintenance alerts

pipeline:
  type: chain

  source:
    type: ReadFromPubSub
    config:
      topic: projects/[redacted]/topics/sensor-data
      format: JSON

  transforms:
    # Detect sensor anomalies
    - type: MapToFields
      config:
        language: python
        fields:
          machine_id: machine_id
          temperature: temperatur
          pressure: pressure
          vibration: vibration
          # Simple anomaly flags
          temp_alert: "temperatur
          pressure_alert: "pressu
          vibration_alert: "vibra
          alert_level: |
            'CRITICAL' if tempera
            else 'WARNING' if tem
            else 'NORMAL'

    # Only keep problematic readi
    - type: Filter
      config:
        language: python
        keep: "temp_alert or pres

    # Group by machine and count iss
    - type: Sql
      config:
        query: |
          SELECT
            machine_id,
            COUNT(*) as alert_count,
            MAX(temperature) as max_temp,
            MAX(pressure) as max_pressure,
            COUNT(CASE WHEN alert_level = 'CRITICAL' THEN 1 END) as critical_count
          FROM PCOLLECTION
          GROUP BY machine_id
```

2

```
# Add maintenance recommendations
- type: MapToFields
```

```
config:
  topic: projects[redacted]/topics/maintenance-alerts
  format: JSON

options:
  streaming: true
  runner: DataflowRunner
```

Local Dev:

```
python -m apache_beam.yaml.main \
--yaml_pipeline_file=digital_stream_pipeline.yaml \ --runner=DirectRunner
```

Dataflow Runner:

```
gcloud dataflow yaml run digital-stream-process-monitoring \
--yaml-pipeline-file=digital_stream_pipeline.yaml \ --region=us-central1 \
--max-workers=10 \ --enable-streaming-engine
```

3. Leverage LLM for Beam Pipeline Orchestration

Prompt Driven Pipeline Generation
Define transforms or trigger pipelines in natural language on the fly.

RAG Enhanced Context Retrieval
Fetch relevant pipeline templates, docs, or metrics to ground LLM decisions

Adaptive Branching Logic
Real-time rerouting of elements when anomalies or new conditions are detected

Schema & Code Synthesis
Auto-generate YAML/SDK snippets for connectors, transforms, and I/O.

Human-in-the-Loop Verification
Insert review checkpoints, leveraging RAG to surface past best practices

3. Leverage LLM for Beam Pipeline Orchestration

LLM Enriched Pipeline

Beam YAML Pipeline
- Process Alerts
  {machine_id, temperature..}

→

Pubsub Topic
- Process Alerts
  JSON stream

→

Data Enrichment
- create incident reports
  Add equipment context

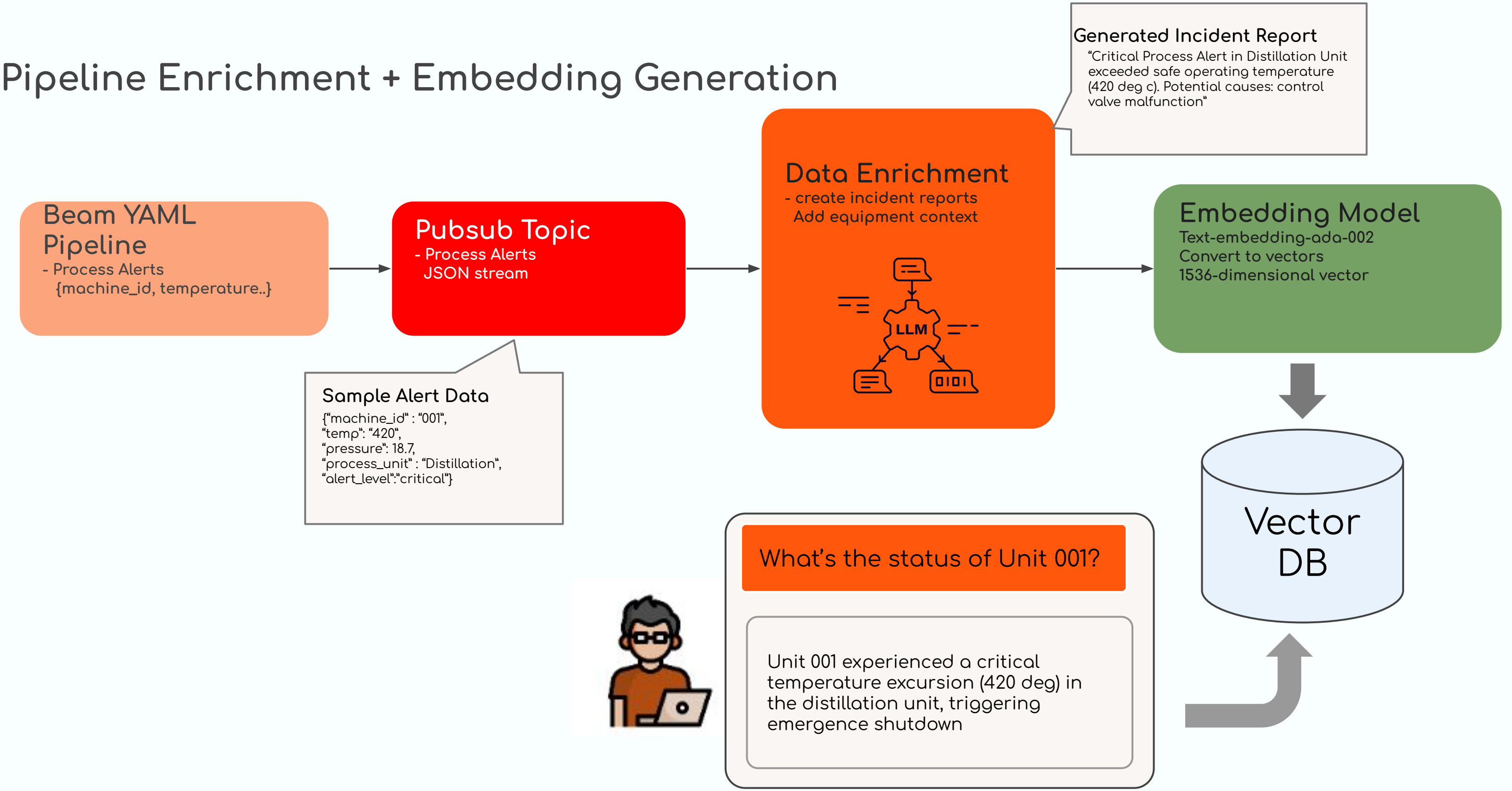
Sample Alert Data

{"machine_id" : "001",
"temp": "420",
"pressure": 18.7,
"process_unit" : "Distillation",
"alert_level":"critical"}

```
import apache_beam as beam
import requests
import json
import logging
import os

# --- Configuration for your self-hosted vLLM endpoint ---
# The internal URL for your vLLM server.
VLLM_ENDPOINT = "http://your-vllm-host.internal:8000/v1/chat/completions"

def enrich_with_data(element: beam.Row) -> beam.Row:
  """
  Calls a self-hosted, unsecured vLLM endpoint to enrich data.
  Uses the OpenAI-compatible API format without an API key.
  """
  # 1. Format the prompt for the model.
  prompt = (
    f"Analyze the following industrial machine data and provide a one-sentence summary "
    f"of its operational status. Data: "
    f"Machine ID: {element.machine_id}, "
    f"Temperature: {element.temperature}°C, "
    f"Pressure: {element.pressure} psi, "
    f"Vibration: {element.vibration_freq} Hz."
  )

  # 2. Prepare the request headers. No 'Authorization' is needed.
  headers = { "Content-Type": "application/json" }

  # 3. Prepare the OpenAI-compatible payload.
  # The 'model' name must match the model you loaded into the vLLM server.
  payload = {
    "model": "mistralai/Mistral-7B-Instruct-v0.2", # <-- IMPORTANT: Use your loaded model's name
    "messages": [
      {"role": "system", "content": "You are an expert AI for industrial machine monitoring."},
      {"role": "user", "content": prompt}
    ],
    "max_tokens": 150,
    "temperature": 0.5
  }

  try:
    # 4. Make the unauthenticated API call to your local vLLM server.
    response = requests.post(VLLM_ENDPOINT, headers=headers, json=payload, timeout=30)
    response.raise_for_status()

    # 5. Parse the OpenAI-compatible response.
    api_response = response.json()
    enriched_message = api_response['choices'][0]['message']['content'].strip()

  except requests.exceptions.RequestException as e:
    logging.error(f"vLLM API call failed for machine {element.machine_id}: {e}")
    enriched_message = "Error: Could not reach self-hosted SLM."
  except (KeyError, IndexError) as e:
    logging.error(f"Failed to parse vLLM API response for machine {element.machine_id}: {e}")
    enriched_message = "Error: Invalid API response format from self-hosted SLM."

  # 6. Return the enriched Beam Row.
  return beam.Row(**element._asdict(), enriched_message=enriched_message)
```

3. Leverage LLM for Beam Pipeline Orchestration

Pipeline Enrichment + Embedding Generation



3. Leverage LLM for Beam Pipeline Orchestration

1 Beam Pipeline YAML definition

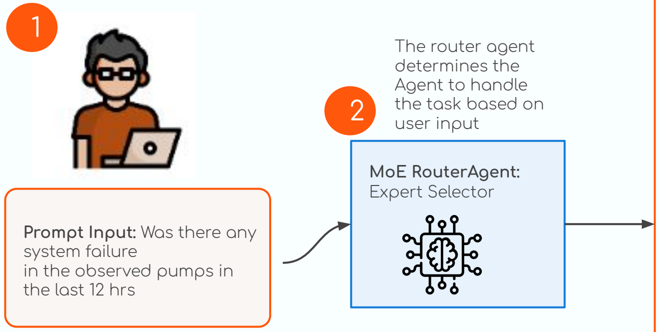
```yaml
pipeline:
  type: chain
  source:
    type: ReadFromPubSub
    config:
      topic: projects/xxx/topics/maintenance-alerts
      format: bytes
  transforms:
    - type: Map
      name: Parse JSON to Beam Row
      config:
        language: python
        callable: 'lambda x: beam.Row(**json.loads(x.decode(''utf-8'')))'
      imports:
        - import json
        - import apache_beam as beam
    - type: Map
      name: Enrich Data
      config:
        language: python
        callable: slm_enrichment.enrich_with_data
      imports:
        - import slm_enrichment
    - type: MLTransform
      name: Generate Embeddings
      config:
        write_artifact_location: 'gs://bucket/embedding-artifacts'
        transforms:
          - type: text_embedding
            config:
              model: sentence-transformers/all-MiniLM-L6-v2
              columns:
                - enriched_message
              embedding_column_name: embedding
    - type: Map
      name: Prepare for PGVector DB Insert
      config:
        language: python
        callable: |
          lambda x: beam.Row(
              machine_id=x.machine_id,
              temperature=x.temperature,
              pressure=x.pressure,
              vibration_freq=x.vibration_freq,
              enriched_message=x.enriched_message,
              embedding=str(list(x.embedding))
          )
      imports:
        - import apache_beam as beam
  sink:
    type: WriteToJdbc
    config:
      driver_class_name: org.postgresql.Driver
      jdbc_url: 'jdbc:postgresql://your-db-host:5432/your-db-name'
      username: your-db-user
      password: your-db-password
      statement: >-
        INSERT INTO machine_telemetry (machine_id, temperature, pressure,
        vibration_freq, enriched_message, embedding) VALUES (?, ?, ?, ?, ?, ?)
```

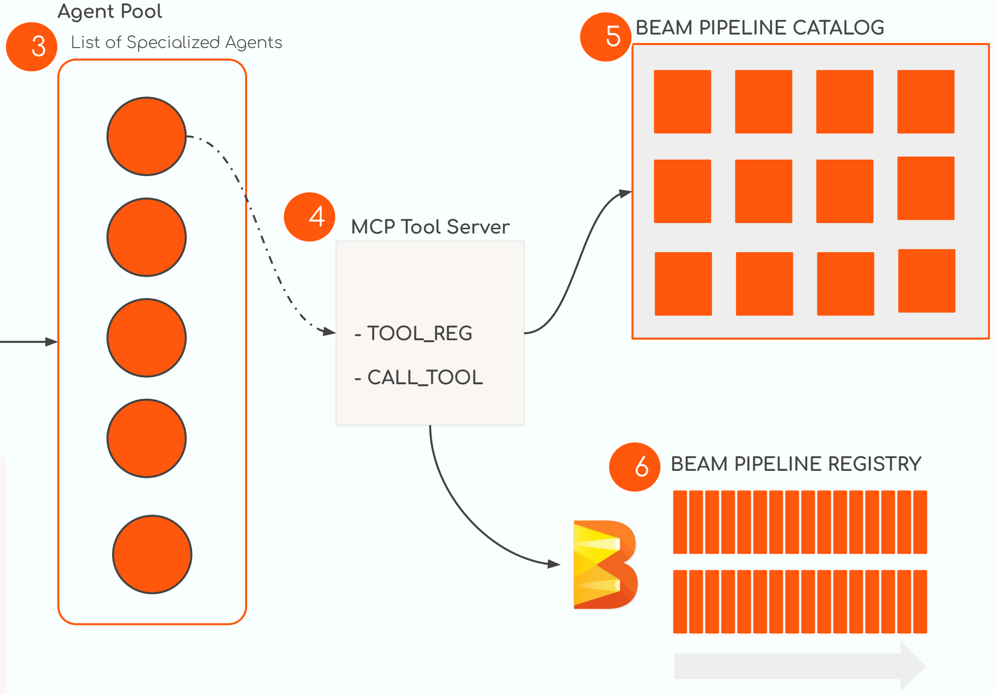
Enrichment Python Code

2

```python
import apache_beam as beam
import requests
import json
import logging
import os

# --- Configuration for your self-hosted vLLM endpoint ---
# The internal URL for your vLLM server.
VLLM_ENDPOINT = "http://your-vllm-host.internal:8000/v1/chat/completions"

def enrich_with_data(element: beam.Row) -> beam.Row:
    """
    Calls a self-hosted, unsecured vLLM endpoint to enrich data.
    Uses the OpenAI-compatible API format without an API key.
    """
    # 1. Format the prompt for the model.
    prompt = (
        f"Analyze the following industrial machine data and provide a one-sentence
        f"of its operational status. Data: "
        f"Machine ID: {element.machine_id}, "
        f"Temperature: {element.temperature}°C, "
        f"Pressure: {element.pressure} psi, "
        f"Vibration: {element.vibration_freq} Hz."
    )

    # 2. Prepare the request headers. No 'Authorization' is needed.
    headers = { "Content-Type": "application/json" }

    # 3. Prepare the OpenAI-compatible payload.
    # The 'model' name must match the model you loaded into the vLLM server.
    payload = {
        "model": "mistralai/Mistral-7B-Instruct-v0.2", # <-- IMPORTANT: Use your lo
        "messages": [
            {"role": "system", "content": "You are an expert AI for industrial mach
            {"role": "user", "content": prompt}
        ],
        "max_tokens": 150,
        "temperature": 0.5
    }

    try:
        # 4. Make the unauthenticated API call to your local vLLM server.
        response = requests.post(VLLM_ENDPOINT, headers=headers, json=payload, time
        response.raise_for_status()

        # 5. Parse the OpenAI-compatible response.
        api_response = response.json()
```

4. Designing Modular Orchestration Agents





Key Advantages

- **Separation of Concerns:** The LLM handles high-level reasoning, while Beam handles scalable, parallel data execution.
- **Modularity:** Each transform in the YAML file is a reusable, plug-and-play "tool" for the agent.
- **Declarative & Auditable:** YAML provides a human-readable and machine-parseable record of the exact workflow that was executed.

4. Designing Modular Orchestration Agents

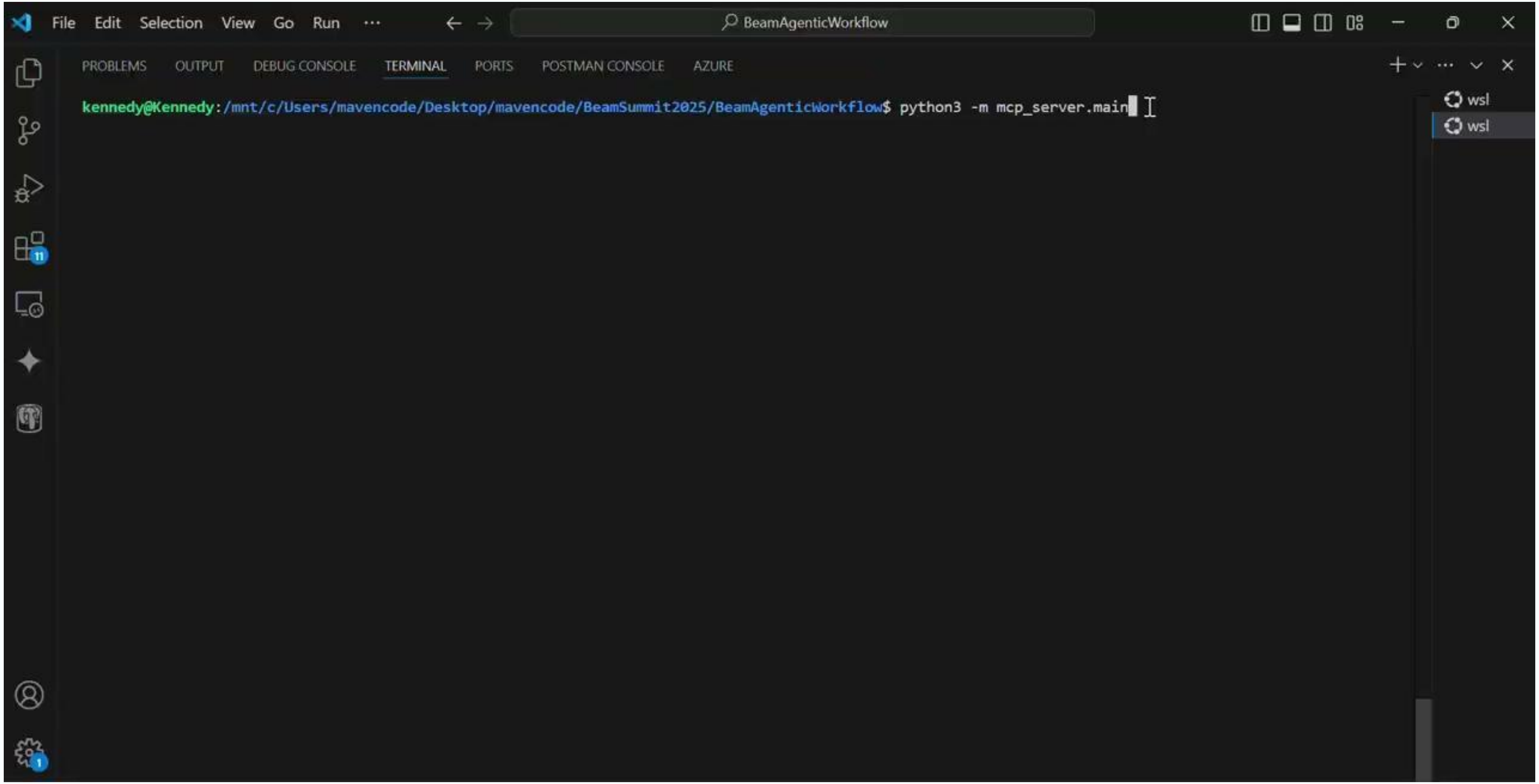
Pipeline Tools Catalog

Agent Tools for Apache Beam Pipelines

- **GET** `/tools` Get Tools
- **POST** `/stream-pipeline-tool` Stream Agent Tool
- **POST** `/etl-pipeline-tool` Etl Agent Tool
- **POST** `/analytics-pipeline-tool` Analytics Agent Tool
- **POST** `/batch-pipeline-tool` Batch Agent Tool
- **POST** `/ml-pipeline-tool` Ml Agent Tool
- **POST** `/quality-pipeline-tool` Quality Agent Tool
- **POST** `/pipeline-run-tool` Pipeline Run Tool

4. Designing Modular Orchestration Agents

4. Designing Modular Orchestration Agents



7. Future Directions

- **Dynamic YAML Synthesis from Natural Language:** Agent translates natural language goals into executable beam.yml files by selecting transforms from a catalog.

- **Adaptive Workflow Repair via YAML Modification:** Agent analyzes pipeline errors, programmatically edits the beam.yml to fix the issue, and automatically relaunches the job.

- **Declarative A/B Testing of Pipeline Logic:** Agent generates multiple beam.yml versions for A/B testing transforms, then launches another YAML pipeline to compare results.

- **Automated Discovery and Integration of New Transforms:** Agent automatically scans documentation for new transforms, adding them to its catalog for use in future YAML pipelines.

Charles Adetiloye

QUESTIONS?

https://www.linkedin.com/in/charlesadetiloye/

github.com/MavenCode