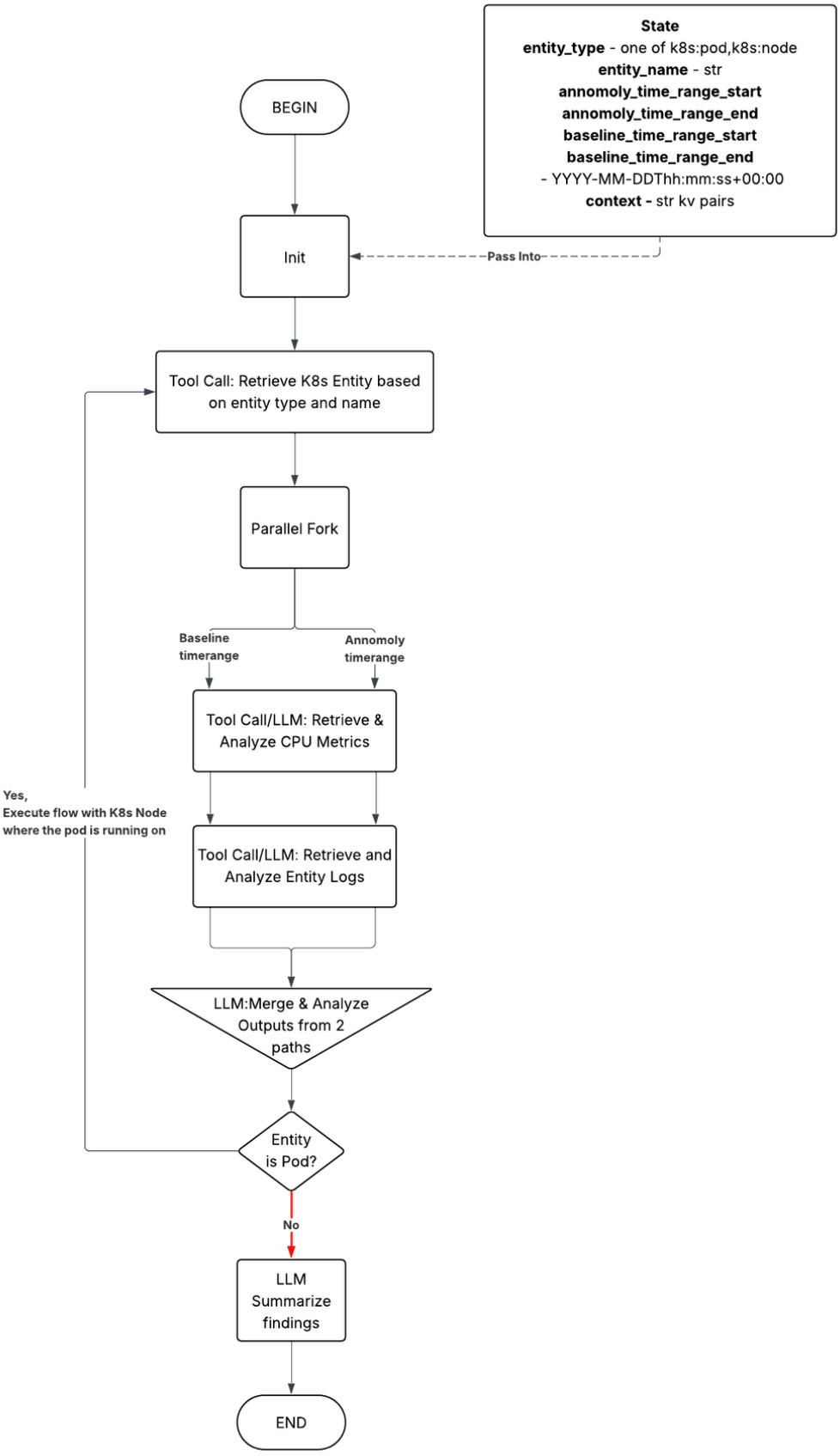
Demo K8s Workflow for Agentic Framework Evaluation

Summary

This sample workflow aims to distill our existing k8s ADT flow so that it can be used for the framework evaluation. This workflow starts with a known entity like a Kubernetes pod and traverse to the residing node for finding the root cause.

Workflow Diagram



Some of the capabilities to test against:

- Parallel Chain
- Loop
- Condition/Decision
- Tool Call
- Summarization

Mock Tool Calls

These calls simulate a pod (`frontend-6d8f4f79f7-kxzpl`) that's crash loop back-offing due to CPU pressure from the pod and the node (`node-1`).

Time ranges

```
normal_range = (
    datetime.fromisoformat("2024-06-26T09:00:00+00:00"),
    datetime.fromisoformat("2024-06-26T10:00:00+00:00"),
)
high_range = (
    datetime.fromisoformat("2024-06-26T10:00:00+00:00"),
    datetime.fromisoformat("2024-06-26T10:30:00+00:00"),
)
```

```
import random
from datetime import datetime, timedelta
from typing import Any


def get_entities(entity_type: str, entity_name: str) -> dict[str, Any]:
    # Mock pod data
    mock_pod = {
        "frontend-6d8f4f79f7-kxzpl": {
            "node_name": "node-1",
            "pod_ip": "10.1.2.34",
            "host_ip": "192.168.1.10",
            "start_time": "2024-06-26T09:58:12Z",
            "labels": {"app": "frontend", "env": "prod", "tier": "web"},
            "annotations": {"prometheus.io/scrape": "true"},
        }
    }

    # Mock node data
    mock_node = {
        "node-1": {
            "internal_ip": "192.168.1.10",
            "capacity": {"cpu": "4", "memory": "16Gi"},
            "labels": {"topology.kubernetes.io/zone": "us-west1-a",
"kubernetes.io/role": "worker"},
            "taints": [],
            "conditions": [{"type": "Ready", "status": "True",
"last_heartbeat_time": "2024-06-26T10:00:01Z"}],
        }
    }

    # Dispatcher logic
```

```python
    if entity_type == "k8s:pod":
        if entity_name in mock_pod:
            return {"entity_type": "pod", "name": entity_name, "metadata": mock_pod[entity_name]}
        raise ValueError(f"Pod '{entity_name}' not found.")

    if entity_type == "k8s:node":
        if entity_name in mock_node:
            return {"entity_type": "node", "name": entity_name, "metadata": mock_node[entity_name]}
        raise ValueError(f"Node '{entity_name}' not found.")

    raise ValueError(f"Unsupported entity type: {entity_type}. Supported types are 'k8s:pod' and 'k8s:node'.")


def get_cpu_utilization(entity_type: str, entity_name: str, start: str, end: str) -> list[dict[str, Any]]:
    known_pods = ["frontend-6d8f4f79f7-kxzpl"]
    known_nodes = ["node-1"]

    # Parse start and end times
    try:
        start_dt = datetime.fromisoformat(start)
        end_dt = datetime.fromisoformat(end)
    except Exception as err:
        raise ValueError(f"Invalid time format for start '{start}' or end '{end}'. Expected ISO format.") from err

    # Validate entity
    if entity_type == "k8s:pod" and entity_name not in known_pods:
        raise ValueError(f"Unknown pod: {entity_name}")
    if entity_type == "k8s:node" and entity_name not in known_nodes:
        raise ValueError(f"Unknown node: {entity_name}")
    if entity_type not in ["k8s:pod", "k8s:node"]:
        raise ValueError(f"Unsupported entity_type '{entity_type}'")

    # Recognized mock time ranges
    normal_range = (
        datetime.fromisoformat("2024-06-26T09:00:00+00:00"),
        datetime.fromisoformat("2024-06-26T10:00:00+00:00"),
    )
    high_range = (
        datetime.fromisoformat("2024-06-26T10:00:00+00:00"),
        datetime.fromisoformat("2024-06-26T10:30:00+00:00"),
    )

    def generate_series(start: datetime, end: datetime, usage_range: tuple) -> list[dict[str, Any]]:
        points = []
        current = start
        while current <= end:
            usage = round(random.uniform(*usage_range), 2)
            points.append({"timestamp": current.isoformat(), "cpu_percent": usage})
            current += timedelta(minutes=5)
        return points

    if start_dt == normal_range[0] and end_dt == normal_range[1]:
```

```
        usage_range = (10.0, 30.0)
    elif start_dt == high_range[0] and end_dt == high_range[1]:
        usage_range = (80.0, 95.0)
    else:
        raise ValueError(f"No mock data available for time range '{start} to {end}'")

    return generate_series(start_dt, end_dt, usage_range)


def get_logs(entity_type: str, entity_name: str, start: str, end: str) -> list[dict[str, Any]]:
    known_pods = {
        "frontend-6d8f4f79f7-kxzpl": [
            # Baseline timerange: 2024-06-26T09:00:00+00:00 to 2024-06-26T10:00:00+00:00
            {
                "type": "Info",
                "reason": "Created",
                "message": "Created container frontend",
                "timestamp": "2024-06-26T09:02:00+00:00",
            },
            {
                "type": "Info",
                "reason": "Pulled",
                "message": "Successfully pulled image 'frontend:v1.2.3'",
                "timestamp": "2024-06-26T09:03:00+00:00",
            },
            {
                "type": "Info",
                "reason": "Started",
                "message": "Started container frontend",
                "timestamp": "2024-06-26T09:05:00+00:00",
            },
            {
                "type": "Info",
                "reason": "Started",
                "message": "Started container frontend",
                "timestamp": "2024-06-26T09:55:00+00:00",
            },
            # High timerange: 2024-06-26T10:00:00+00:00 to 2024-06-26T10:30:00+00:00
            {
                "type": "Warning",
                "reason": "Unhealthy",
                "message": "Liveness probe failed: HTTP probe failed with status code 500",
                "timestamp": "2024-06-26T10:19:30+00:00",
            },
            {
                "type": "Warning",
                "reason": "BackOff",
                "message": "Back-off restarting failed container",
                "timestamp": "2024-06-26T10:20:00+00:00",
            },
        ]
    }

    known_nodes = {
```

```
        "node-1": [
            # Baseline timerange: 2024-06-26T09:00:00+00:00 to 2024-06-26T10:00:00+00:00
            {
                "type": "Normal",
                "reason": "KubeletReady",
                "message": "kubelet is posting ready status",
                "timestamp": "2024-06-26T09:10:00+00:00",
            },
            {
                "type": "Normal",
                "reason": "NodeHasSufficientMemory",
                "message": "Node has sufficient memory available",
                "timestamp": "2024-06-26T09:20:00+00:00",
            },
            {
                "type": "Normal",
                "reason": "NodeHasNoDiskPressure",
                "message": "Node has no disk pressure",
                "timestamp": "2024-06-26T09:30:00+00:00",
            },
            # High timerange: 2024-06-26T10:00:00+00:00 to 2024-06-26T10:30:00+00:00
            {
                "type": "Normal",
                "reason": "KubeletReady",
                "message": "kubelet is posting ready status",
                "timestamp": "2024-06-26T10:10:00+00:00",
            },
            {
                "type": "Warning",
                "reason": "CPUPressure",
                "message": "Node is under CPU pressure",
                "timestamp": "2024-06-26T10:15:00+00:00",
            },
        ]
    }

    try:
        start_dt = datetime.fromisoformat(start)
        end_dt = datetime.fromisoformat(end)
    except Exception as err:
        raise ValueError(f"Invalid time format for start '{start}' or end '{end}'. Expected ISO format.") from err

    def filter_events(events, start_dt, end_dt):
        filtered = []
        for event in events:
            event_ts = datetime.fromisoformat(event["timestamp"])
            if start_dt <= event_ts <= end_dt:
                filtered.append(event)
        return filtered

    if entity_type == "k8s:pod":
        if entity_name not in known_pods:
            raise ValueError(f"Unknown pod: {entity_name}")
        events = filter_events(known_pods[entity_name], start_dt, end_dt)
    elif entity_type == "k8s:node":
        if entity_name not in known_nodes:
```

```
            raise ValueError(f"Unknown node: {entity_name}")
        events = filter_events(known_nodes[entity_name], start_dt, end_dt)
    else:
        raise ValueError(f"Unsupported entity_type '{entity_type}'")

    return events


if __name__ == "__main__":
    # Example usage
    print(get_entities("k8s:pod", "frontend-6d8f4f79f7-kxzpl"))
    print(get_entities("k8s:node", "node-1"))
    print(
        get_cpu_utilization(
            "k8s:pod", "frontend-6d8f4f79f7-kxzpl", "2024-06-26T09:00:00+00:00", "2024-06-26T10:00:00+00:00"
        )
    )
    print(
        get_cpu_utilization(
            "k8s:pod", "frontend-6d8f4f79f7-kxzpl", "2024-06-26T10:00:00+00:00", "2024-06-26T10:30:00+00:00"
        )
    )
    print(get_cpu_utilization("k8s:node", "node-1", "2024-06-26T09:00:00+00:00", "2024-06-26T10:00:00+00:00"))
    print(get_cpu_utilization("k8s:node", "node-1", "2024-06-26T10:00:00+00:00", "2024-06-26T10:30:00+00:00"))
    print(get_logs("k8s:pod", "frontend-6d8f4f79f7-kxzpl", "2024-06-26T09:00:00+00:00", "2024-06-26T10:00:00+00:00"))
    print(get_logs("k8s:pod", "frontend-6d8f4f79f7-kxzpl", "2024-06-26T10:00:00+00:00", "2024-06-26T10:30:00+00:00"))
    print(get_logs("k8s:node", "node-1", "2024-06-26T09:00:00+00:00", "2024-06-26T10:00:00+00:00"))
    print(get_logs("k8s:node", "node-1", "2024-06-26T10:00:00+00:00", "2024-06-26T10:30:00+00:00"))

```