Cloud Cost Tracking System Documentation

1. Architecture Overview

The Cloud Cost Tracking (CCT) system implements an event-driven architecture that provides real-time monitoring, anomaly detection, and automated optimization of cloud resources across multiple providers (AWS, Azure, and GCP).

System Components

Data Collection Layer

- **Cloud Providers**: AWS, Azure, and GCP generate resource logs containing cost data
- **Fluent Bit Log Processor**: Collects and normalizes logs from all providers every 5 seconds
- **VictoriaMetrics Time-Series DB**: Stores processed metrics for efficient querying and analysis

Monitoring and Alerting Layer

- **Alertmanager**: Checks metrics every 3 seconds for anomalies and threshold violations
- **Zenduty**: Receives incidents for human notification when critical issues are detected

Event Processing Layer

- **Apache Kafka Event Stream**: Central message bus for anomaly events and optimization triggers
- **Argo Workflows Automation Engine**: Orchestrates complex optimization workflows
- **Admission Controller**: Enforces policy compliance for all proposed optimizations

Security Layer

- **HashiCorp Vault**: Manages credentials securely for accessing cloud provider APIs

Analysis Layer

- **PostgreSQL Metadata Store**: Records optimization actions and results
- **Grafana Dashboards**: Visualizes cost trends and optimization outcomes

2. Major Design Decisions and Tradeoffs

Real-Time Monitoring vs. Resource Efficiency

Decision: Implement 5-second collection intervals and 3-second anomaly detection checks.

Tradeoff: This high-frequency monitoring creates higher infrastructure demands but provides near-immediate detection of cost anomalies. The system prioritizes early detection over reduced resource usage, recognizing that delayed cost visibility was a primary problem to solve.

Centralized Log Processing vs. Distributed Collection

Decision: Use Fluent Bit as a centralized log processor for multiple cloud providers.

Tradeoff: This approach creates a potential single point of failure but significantly simplifies the implementation with consistent processing rules across providers. The centralized approach was chosen to reduce complexity and improve maintainability.

Event-Driven Architecture vs. Direct Integration

Decision: Implement Kafka as the central event bus connecting detection and remediation.

Tradeoff: Adding Kafka increases system complexity but provides critical decoupling between components, allowing them to scale independently and continue functioning if one part experiences issues. This makes the system more resilient and easier to extend.

Automated Optimization vs. Manual Control

Decision: Combine automated workflows with policy enforcement and optional human intervention.

Tradeoff: This balanced approach trades some optimization speed for enhanced safety and governance. The system is designed to automatically address routine optimizations while escalating complex or high-impact changes for review.

Specialized Components vs. General-Purpose Solutions

Decision: Use purpose-built components for specific functions (VictoriaMetrics for time-series data, Argo for workflows).

Tradeoff: This increases the number of technologies to maintain but provides better performance and capabilities for each specific function. Each component was selected for its strengths in its particular domain.

3. Proof of Solution

Problem 1: Delayed Cost Increase Detection

Original Issue: There's a significant delay between cost increases occurring and teams discovering them.

Solution: The architecture implements:

- 5-second cost data collection from all cloud providers
- 3-second anomaly detection cycles from AlertManager
- Immediate notification pathways (Kafka for automation, Zenduty for humans)

This reduces detection time from days or weeks to mere seconds, enabling immediate response to cost anomalies.

Problem 2: Difficulty Identifying Cost Sources

Original Issue: Finding where cost increases come from is a "game of guesswork."

Solution: The system maintains comprehensive context throughout the pipeline:

- Resource-level attribution in collected logs
- Multi-dimensional querying capabilities in VictoriaMetrics
- Detailed visualization in Grafana with metadata from PostgreSQL

This eliminates guesswork by providing precise attribution of which services, regions, and resources are contributing to cost increases.

Problem 3: Unclear Optimization Path

Original Issue: The path to optimizing costs is not straightforward.

Solution: The architecture delivers:

- Predefined optimization workflows orchestrated by Argo
- Policy-governed actions via the AdmissionController
- Automated execution against cloud providers with appropriate credentials
- Result tracking to validate optimization outcomes

This creates clear, automated paths for addressing cost issues with appropriate governance.

4. Implementation Code Example

```
import asyncio
import random
from datetime import datetime

# Simulated cloud cost data
class CloudCostData:
    def __init__(self, provider, service, cost):
        self.provider = provider
        self.service = service
        self.cost = cost
        self.timestamp = datetime.now()

# Component simulations
async def fluent_bit_collector():
    """Simulate Fluent Bit collecting logs every 5 seconds"""
    providers = ['AWS', 'Azure', 'GCP']
    services = {'AWS': ['EC2', 'S3', 'RDS'],
                'Azure': ['VM', 'Storage', 'SQL'],
                'GCP': ['Compute', 'Storage', 'BigQuery']}

    while True:
        provider = random.choice(providers)
        service = random.choice(services[provider])

        # Introduce occasional cost spikes
        baseline = 300
        if random.random() > 0.7:
            cost = baseline * random.uniform(1.5, 2.0)  # Anomalous cost
        else:
            cost = baseline * random.uniform(0.8, 1.2)  # Normal cost

        cost_data = CloudCostData(provider, service, round(cost, 2))
        print(f"📊 [Fluent Bit] Collecting logs from {provider} {service}")

        # Send to VictoriaMetrics
        await store_metrics(cost_data)

        # Collection interval: 5 seconds
```

```
        await asyncio.sleep(5)

async def store_metrics(cost_data):
    """Simulate VictoriaMetrics storing metrics"""
    print(f"📊 [VictoriaMetrics] Stored {cost_data.provider} {cost_data.service} cost:
${cost_data.cost}")

    # Send to AlertManager for checking
    await check_for_anomalies(cost_data)

async def check_for_anomalies(cost_data):
    """Simulate AlertManager checking for anomalies every 3 seconds"""
    # Anomaly detection interval
    await asyncio.sleep(3)

    # Simple threshold-based anomaly detection
    baseline = 300
    if cost_data.cost > baseline * 1.5:
        deviation = ((cost_data.cost - baseline) / baseline) * 100
        print(f"🚨 [AlertManager] Cost anomaly detected! {cost_data.provider}
{cost_data.service} ${cost_data.cost} ({deviation:.1f}% above baseline)")

        # Critical anomalies create incidents
        if deviation > 50:
            print(f"🔔 [Zenduty] Created incident for {cost_data.provider}
{cost_data.service} cost spike")

        # All anomalies produce Kafka events
        event = {
            'provider': cost_data.provider,
            'service': cost_data.service,
            'deviation': deviation
        }
        await produce_kafka_event(event)

async def produce_kafka_event(event):
    """Simulate Kafka producing an optimization event"""
    print(f"🔁 [Kafka] Producing event: {{provider: \"{event['provider']}\", service:
\"{event['service']}\", deviation: {event['deviation']:.1f}%}}")
```

```
    # Event triggers workflow
    await trigger_optimization_workflow(event)

async def trigger_optimization_workflow(event):
    """Simulate Argo Workflows orchestrating optimization"""
    print(f"⚙️ [Argo Workflows] Starting {event['service'].lower()}-optimization
workflow")

    # Get credentials from Vault
    print(f"🔑 [HashiCorp Vault] Retrieved credentials for {event['provider']}")

    # Check policy compliance
    policy_compliant = await check_policy(event)

    if policy_compliant:
        # Execute optimization with current cost estimation
        current_cost = 300 * (1 + event['deviation']/100)
        optimized_cost = current_cost * 0.8  # 20% reduction

        print(f"📊 [Argo Workflows] Rightsizing complete: ${current_cost:.2f} →
${optimized_cost:.2f} (20% reduction)")

        # Store results
        await store_results(event, current_cost, optimized_cost)
    else:
        print("❌ [AdmissionController] Policy validation failed")

async def check_policy(event):
    """Simulate AdmissionController policy check"""
    # Simple policy: don't optimize critical services during business hours
    critical_services = ['RDS', 'SQL', 'BigQuery']
    current_hour = datetime.now().hour

    if event['service'] in critical_services and 9 <= current_hour <= 17:
        print("🔒 [AdmissionController] Policy validation failed - Cannot optimize
critical service during business hours")
        return False

    print("🔒 [AdmissionController] Policy validation passed")
    return True
```

```
async def store_results(event, before_cost, after_cost):
    """Simulate storing results in PostgreSQL"""
    print(f"💾 [PostgreSQL] Stored optimization result: {event['provider']}
{event['service']} saved ${before_cost - after_cost:.2f}")

async def main():
    """Run the simulation"""
    print("🚀 Starting Cloud Cost Tracking System")
    # Create and run the collection task
    collector_task = asyncio.create_task(fluent_bit_collector())

    # Let it run for a while then cancel
    await asyncio.sleep(60)
    collector_task.cancel()

# Run the simulation
if __name__ == "__main__":
    asyncio.run(main())
```

5. Known Gaps and Limitations

1. Initial Training Period

Gap: The anomaly detection system requires historical data to establish accurate baselines. During the initial deployment period, there will be limited baseline data.

Why It's Safe to Ignore: This is a temporary limitation that resolves itself as the system collects more data. During the initial period, the system can use static thresholds until sufficient data is available for more sophisticated anomaly detection.

2. False Positives

Gap: Any anomaly detection system will generate some false positives, particularly early in deployment.

Why It's Safe to Ignore: The architecture is designed to handle this through:

- Human review of critical incidents via Zenduty
- Policy controls in AdmissionController to prevent risky automated actions

- Continuous improvement of detection algorithms as more data is collected

3. Limited End-User Input

Gap: The current design has limited mechanisms for incorporating end-user feedback into optimization decisions.

Why It's Safe to Ignore: The primary goal was to address the immediate problems of detection delay and attribution confusion. User feedback integration can be added in future iterations without compromising the core functionality.

4. Dependency on External Systems

Gap: The architecture relies on several third-party components (Kafka, Vault, Argo, etc.) which introduce external dependencies.

Why It's Safe to Ignore: These are industry-standard components with proven reliability and active maintenance. The modular design also allows for component replacement if needed, without changing the overall architecture.

6. Conclusion

This cloud cost tracking system addresses the core problems identified in the Atlan challenge by providing:

1. **Real-time visibility** into cloud costs across providers
2. **Precise attribution** of cost increases to specific resources
3. **Automated optimization paths** with appropriate governance

The architecture makes deliberate tradeoffs that prioritize early detection and automated response over resource efficiency, recognizing that the cost of delayed detection far outweighs the infrastructure cost of the monitoring system itself.

While some limitations exist, they are either temporary in nature or manageable within the operational context, making them acceptable compromises for the significant benefits the system delivers.