Temporal Training Session 12: Observability, Monitoring, and Deployment Considerations

Table of Contents

Welcome and Recap

Welcome to Day 12 of our Temporal training. Over the past eleven days, we've explored the fundamentals and advanced patterns of Temporal, from basic workflows to comprehensive testing strategies. Today, we focus on the critical aspects of **observability, monitoring, and deployment**—the bridge between development and production.

In production environments, Temporal applications must be observable, monitorable, and deployable at scale. We'll explore how to instrument your Temporal applications for comprehensive monitoring, set up effective alerting, and deploy them reliably in production environments.

Why Observability Matters in Temporal

Observability in Temporal is not just about monitoring—it's about understanding the **health, performance, and behavior** of your distributed workflow system. Unlike traditional applications, Temporal workflows have unique characteristics that make observability particularly important:

Unique Observability Challenges in Temporal

1. **Long-Running Processes:** Workflows can run for days, weeks, or months, requiring visibility into their progress and state.

2. **Distributed State:** Workflow state is distributed across multiple services, making it essential to track state transitions and consistency.

3. **Event-Driven Architecture:** The event-sourced nature of Temporal requires understanding the event flow and history.

4. **Failure Recovery:** When failures occur, you need visibility into the recovery process and state reconstruction.

5. **Performance at Scale:** As workflows scale, you need to monitor resource usage, throughput, and latency.

The Three Pillars of Observability

```
┌─────────────────┐  ┌─────────────────┐  ┌─────────────────┐
│    Metrics      │  │     Logs        │  │     Traces      │
│                 │  │                 │  │                 │
│ • Performance   │  │ • Event History │  │ • Request Flow  │
│ • Throughput    │  │ • Error Details │  │ • Dependencies  │
│ • Resource Use  │  │ • State Changes │  │ • Latency       │
└─────────────────┘  └─────────────────┘  └─────────────────┘
```

Temporal Observability Stack

Temporal provides a comprehensive observability stack that integrates with industry-standard tools:

Core Observability Components

Component	Purpose	Tools
Metrics	Quantitative measurements	Prometheus, StatsD
Logs	Event records and debugging	Structured logging, ELK stack
Traces	Request flow and dependencies	OpenTelemetry, Jaeger
Alerts	Proactive notifications	AlertManager, PagerDuty

Temporal Server Metrics

Temporal Server exposes extensive metrics that provide insights into:

- **Workflow Execution:** Success rates, completion times, failure rates
- **Activity Execution:** Performance, retry rates, timeout frequencies
- **Task Queue Performance:** Queue depths, processing rates, worker utilization
- **System Health:** Memory usage, CPU utilization, network I/O

Metrics Collection with Prometheus

Prometheus is the recommended metrics collection system for Temporal, providing powerful querying capabilities and integration with the broader observability ecosystem.

Setting Up Prometheus for Temporal

```
# prometheus.yml
global:
  scrape_interval: 15s
```

```yaml
  evaluation_interval: 15s

rule_files:
  - "temporal_rules.yml"

scrape_configs:
  - job_name: 'temporal'
    static_configs:
      - targets: ['localhost:9090']
    metrics_path: '/metrics'
    scrape_interval: 5s

  - job_name: 'temporal-workers'
    static_configs:
      - targets: ['worker1:9090', 'worker2:9090']
    metrics_path: '/metrics'
    scrape_interval: 10s
```

Key Temporal Metrics to Monitor

```
# Workflow execution rate
rate(temporal_workflow_execution_started_total[5m])

# Workflow completion rate
rate(temporal_workflow_execution_completed_total[5m])

# Workflow failure rate
rate(temporal_workflow_execution_failed_total[5m])

# Activity execution rate
rate(temporal_activity_execution_started_total[5m])

# Task queue depth
temporal_task_queue_depth

# Worker utilization
temporal_worker_task_slots_available
```

Custom Application Metrics

```python
# metrics.py
from prometheus_client import Counter, Histogram, Gauge
from temporalio import activity, workflow

# Define custom metrics
workflow_execution_duration = Histogram(
    'custom_workflow_execution_duration_seconds',
    'Duration of workflow execution',
    ['workflow_type', 'status']
```

```
)

activity_execution_duration = Histogram(
    'custom_activity_execution_duration_seconds',
    'Duration of activity execution',
    ['activity_type', 'status']
)

business_events_total = Counter(
    'custom_business_events_total',
    'Total number of business events processed',
    ['event_type', 'status']
)

active_orders_gauge = Gauge(
    'custom_active_orders',
    'Number of active orders being processed'
)

# Instrument workflows
@workflow.defn
class InstrumentedWorkflow:
    @workflow.run
    async def run(self, input_data: dict) -> dict:
        start_time = time.time()

        try:
            # Workflow logic here
            result = await self.process_order(input_data)

            # Record success metrics
            workflow_execution_duration.labels(
                workflow_type="order_processing",
                status="success"
            ).observe(time.time() - start_time)

            return result

        except Exception as e:
            # Record failure metrics
            workflow_execution_duration.labels(
                workflow_type="order_processing",
                status="failure"
            ).observe(time.time() - start_time)
            raise

# Instrument activities
@activity.defn
async def instrumented_activity(data: dict) -> dict:
    start_time = time.time()

    try:
        # Activity logic here
        result = process_data(data)
```

```
        # Record success metrics
        activity_execution_duration.labels(
            activity_type="data_processing",
            status="success"
        ).observe(time.time() - start_time)

        return result

    except Exception as e:
        # Record failure metrics
        activity_execution_duration.labels(
            activity_type="data_processing",
            status="failure"
        ).observe(time.time() - start_time)
        raise
```

Prometheus Alerting Rules

```
# temporal_rules.yml
groups:
  - name: temporal_alerts
    rules:
      - alert: HighWorkflowFailureRate
        expr: rate(temporal_workflow_execution_failed_total[5m]) > 0.1
        for: 2m
        labels:
          severity: warning
        annotations:
          summary: "High workflow failure rate detected"
          description: "Workflow failure rate is {{ $value }} failures per second"

      - alert: HighTaskQueueDepth
        expr: temporal_task_queue_depth > 1000
        for: 5m
        labels:
          severity: warning
        annotations:
          summary: "High task queue depth detected"
          description: "Task queue depth is {{ $value }} tasks"

      - alert: WorkerUtilizationHigh
        expr: (temporal_worker_task_slots_total - temporal_worker_task_slots_available) / temporal_worker_task_slots_total > 0.9
        for: 5m
        labels:
          severity: warning
        annotations:
          summary: "High worker utilization detected"
```

```
          description: "Worker utilization is {{ $value |
humanizePercentage }}"

      - alert: TemporalServerDown
        expr: up{job="temporal"} == 0
        for: 1m
        labels:
          severity: critical
        annotations:
          summary: "Temporal server is down"
          description: "Temporal server has been down for more than 1
minute"
```

Visualization with Grafana

Grafana provides powerful visualization capabilities for Temporal metrics, enabling you to create comprehensive dashboards for monitoring your workflow system.

Temporal Dashboard Configuration

```
{
  "dashboard": {
    "title": "Temporal Workflow Monitoring",
    "panels": [
      {
        "title": "Workflow Execution Rate",
        "type": "graph",
        "targets": [
          {
            "expr": "rate(temporal_workflow_execution_started_total[5m])",
            "legendFormat": "{{workflow_type}}"
          }
        ]
      },
      {
        "title": "Workflow Success Rate",
        "type": "graph",
        "targets": [
          {
            "expr": "rate(temporal_workflow_execution_completed_total[5m])
/ rate(temporal_workflow_execution_started_total[5m])",
            "legendFormat": "Success Rate"
          }
        ]
      },
      {
        "title": "Task Queue Depth",
        "type": "graph",
        "targets": [
          {
```

```
            "expr": "temporal_task_queue_depth",
            "legendFormat": "{{task_queue}}"
          }
        ]
      },
      {
        "title": "Worker Utilization",
        "type": "gauge",
        "targets": [
          {
            "expr": "(temporal_worker_task_slots_total - 
temporal_worker_task_slots_available) / temporal_worker_task_slots_total",
            "legendFormat": "Utilization"
          }
        ]
      }
    ]
  }
}
```

Key Dashboard Panels

1. **Workflow Performance Panel:**
    - Execution rate over time
    - Success/failure rates
    - Average execution duration
    - Top workflows by volume
2. **Activity Performance Panel:**
    - Activity execution rates
    - Retry rates and patterns
    - Timeout frequencies
    - Activity dependencies
3. **System Health Panel:**
    - Task queue depths
    - Worker utilization
    - Memory and CPU usage
    - Network I/O
4. **Business Metrics Panel:**
    - Custom business events
    - Order processing rates
    - Revenue impact metrics
    - SLA compliance

Advanced Logging and Tracing

Structured Logging in Temporal

```python
# logging_config.py
import logging
import json
from datetime import datetime
from temporalio import workflow, activity

class StructuredFormatter(logging.Formatter):
    def format(self, record):
        log_entry = {
            "timestamp": datetime.utcnow().isoformat(),
            "level": record.levelname,
            "logger": record.name,
            "message": record.getMessage(),
            "workflow_id": getattr(record, 'workflow_id', None),
            "activity_id": getattr(record, 'activity_id', None),
            "task_queue": getattr(record, 'task_queue', None)
        }

        if hasattr(record, 'workflow_type'):
            log_entry['workflow_type'] = record.workflow_type

        if hasattr(record, 'activity_type'):
            log_entry['activity_type'] = record.activity_type

        return json.dumps(log_entry)

# Configure logging
def setup_logging():
    logger = logging.getLogger()
    handler = logging.StreamHandler()
    handler.setFormatter(StructuredFormatter())
    logger.addHandler(handler)
    logger.setLevel(logging.INFO)

# Instrumented workflow with structured logging
@workflow.defn
class LoggedWorkflow:
    @workflow.run
    async def run(self, input_data: dict) -> dict:
        workflow.logger.info(
            "Workflow started",
            extra={
                "workflow_id": workflow.info().workflow_id,
                "workflow_type": "order_processing",
                "input_data": input_data
            }
        )
```

```python
        try:
            # Process the order
            result = await self.process_order(input_data)

            workflow.logger.info(
                "Workflow completed successfully",
                extra={
                    "workflow_id": workflow.info().workflow_id,
                    "result": result
                }
            )

            return result

        except Exception as e:
            workflow.logger.error(
                "Workflow failed",
                extra={
                    "workflow_id": workflow.info().workflow_id,
                    "error": str(e),
                    "error_type": type(e).__name__
                }
            )
            raise

# Instrumented activity with structured logging
@activity.defn
async def logged_activity(data: dict) -> dict:
    activity.logger.info(
        "Activity started",
        extra={
            "activity_id": activity.info().activity_id,
            "activity_type": "payment_processing",
            "input_data": data
        }
    )

    try:
        # Process payment
        result = process_payment(data)

        activity.logger.info(
            "Activity completed successfully",
            extra={
                "activity_id": activity.info().activity_id,
                "result": result
            }
        )

        return result

    except Exception as e:
        activity.logger.error(
            "Activity failed",
```

```
                extra={
                    "activity_id": activity.info().activity_id,
                    "error": str(e),
                    "error_type": type(e).__name__
                }
            )
            raise
```

Distributed Tracing with OpenTelemetry

```
# tracing_config.py
from opentelemetry import trace
from opentelemetry.exporter.jaeger.thrift import JaegerExporter
from opentelemetry.sdk.trace import TracerProvider
from opentelemetry.sdk.trace.export import BatchSpanProcessor
from temporalio import workflow, activity

# Set up tracing
def setup_tracing():
    trace.set_tracer_provider(TracerProvider())
    jaeger_exporter = JaegerExporter(
        agent_host_name="localhost",
        agent_port=6831,
    )
    span_processor = BatchSpanProcessor(jaeger_exporter)
    trace.get_tracer_provider().add_span_processor(span_processor)

# Instrumented workflow with tracing
@workflow.defn
class TracedWorkflow:
    @workflow.run
    async def run(self, input_data: dict) -> dict:
        tracer = trace.get_tracer(__name__)

        with tracer.start_as_current_span("order_processing_workflow") as
span:
            span.set_attribute("workflow.id", workflow.info().workflow_id)
            span.set_attribute("workflow.type", "order_processing")
            span.set_attribute("input.customer_id",
input_data.get("customer_id"))

            # Process order
            result = await self.process_order(input_data)

            span.set_attribute("output.order_id", result.get("order_id"))
            span.set_attribute("output.status", result.get("status"))

            return result

# Instrumented activity with tracing
@activity.defn
```

```python
async def traced_activity(data: dict) -> dict:
    tracer = trace.get_tracer(__name__)

    with tracer.start_as_current_span("payment_processing_activity") as
span:
        span.set_attribute("activity.id", activity.info().activity_id)
        span.set_attribute("activity.type", "payment_processing")
        span.set_attribute("input.amount", data.get("amount"))

        # Process payment
        result = process_payment(data)

        span.set_attribute("output.transaction_id",
result.get("transaction_id"))
        span.set_attribute("output.status", result.get("status"))

        return result
```

Deployment Strategies

Docker Deployment

```dockerfile
# Dockerfile for Temporal Worker
FROM python:3.11-slim

WORKDIR /app

# Install dependencies
COPY requirements.txt .
RUN pip install -r requirements.txt

# Copy application code
COPY . .

# Expose metrics port
EXPOSE 9090

# Run worker
CMD ["python", "worker.py"]
```

```yaml
# docker-compose.yml
version: '3.8'

services:
  temporal:
    image: temporalio/auto-setup:1.22.0
    environment:
      - DB=postgresql
```

```
      - DB_PORT=5432
      - POSTGRES_USER=temporal
      - POSTGRES_PWD=temporal
      - POSTGRES_SEEDS=postgresql
    ports:
      - "7233:7233"
      - "9090:9090"
    depends_on:
      - postgresql

  postgresql:
    image: postgres:13
    environment:
      - POSTGRES_USER=temporal
      - POSTGRES_PWD=temporal
      - POSTGRES_DB=temporal
    volumes:
      - postgresql-data:/var/lib/postgresql/data

  worker:
    build: .
    environment:
      - TEMPORAL_HOST=temporal:7233
      - METRICS_PORT=9090
    ports:
      - "9091:9090"
    depends_on:
      - temporal

volumes:
  postgresql-data:
```

Kubernetes Deployment

```yaml
# temporal-deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: temporal-worker
spec:
  replicas: 3
  selector:
    matchLabels:
      app: temporal-worker
  template:
    metadata:
      labels:
        app: temporal-worker
    spec:
      containers:
      - name: worker
```

```yaml
        image: your-registry/temporal-worker:latest
        ports:
        - containerPort: 9090
        env:
        - name: TEMPORAL_HOST
          value: "temporal-frontend:7233"
        - name: TASK_QUEUE
          value: "production-queue"
        - name: METRICS_PORT
          value: "9090"
        resources:
          requests:
            memory: "256Mi"
            cpu: "250m"
          limits:
            memory: "512Mi"
            cpu: "500m"
        livenessProbe:
          httpGet:
            path: /health
            port: 9090
          initialDelaySeconds: 30
          periodSeconds: 10
        readinessProbe:
          httpGet:
            path: /ready
            port: 9090
          initialDelaySeconds: 5
          periodSeconds: 5

---
apiVersion: v1
kind: Service
metadata:
  name: temporal-worker-service
spec:
  selector:
    app: temporal-worker
  ports:
  - port: 9090
    targetPort: 9090
  type: ClusterIP

---
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: temporal-worker-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: temporal-worker
  minReplicas: 3
```

```yaml
  maxReplicas: 10
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 70
  - type: Resource
    resource:
      name: memory
      target:
        type: Utilization
        averageUtilization: 80
```

Helm Chart Deployment

```yaml
# values.yaml
temporal:
  server:
    replicaCount: 3
    resources:
      requests:
        memory: "1Gi"
        cpu: "500m"
      limits:
        memory: "2Gi"
        cpu: "1000m"

  worker:
    replicaCount: 5
    resources:
      requests:
        memory: "512Mi"
        cpu: "250m"
      limits:
        memory: "1Gi"
        cpu: "500m"

    autoscaling:
      enabled: true
      minReplicas: 3
      maxReplicas: 15
      targetCPUUtilizationPercentage: 70
      targetMemoryUtilizationPercentage: 80

monitoring:
  prometheus:
    enabled: true
    retention: "15d"
```

```yaml
  grafana:
    enabled: true
    adminPassword: "admin"

  alertmanager:
    enabled: true
    config:
      global:
        slack_api_url:
"https://hooks.slack.com/services/YOUR/SLACK/WEBHOOK"
      route:
        group_by: ['alertname']
        group_wait: 10s
        group_interval: 10s
        repeat_interval: 1h
        receiver: 'slack-notifications'
      receivers:
      - name: 'slack-notifications'
        slack_configs:
        - channel: '#temporal-alerts'
          title: '{{ template "slack.title" . }}'
          text: '{{ template "slack.text" . }}'
```

High Availability and Disaster Recovery

Multi-Region Deployment

```yaml
# multi-region-deployment.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: temporal-config
data:
  temporal.yaml: |
    global:
      membership:
        maxJoinDuration: 30s
      pprof:
        port: 7936
      tls:
        enabled: false

    persistence:
      defaultStore: postgresql
      visibilityStore: postgresql
      numHistoryShards: 4
      datastores:
        postgresql:
          sql:
            user: temporal
```

```
            password: temporal
            pluginName: "postgres"
            databaseName: temporal
            connectAddr: "postgresql:5432"
            connectProtocol: "tcp"
            maxConns: 20
            maxIdleConns: 20
            maxConnLifetime: "1h"

    clusterMetadata:
      enableGlobalNamespace: true
      failoverVersionIncrement: 10
      masterClusterName: "primary"
      currentClusterName: "primary"
      clusterInformation:
        primary:
          enabled: true
          initialFailoverVersion: 1
          rpcName: "frontend"
          rpcAddress: "127.0.0.1:7233"
        secondary:
          enabled: true
          initialFailoverVersion: 2
          rpcName: "frontend"
          rpcAddress: "127.0.0.1:7234"

    dcv2:
      membership:
        maxJoinDuration: 30s
      tls:
        enabled: false
      frontend:
        rpc:
          grpcPort: 7233
          membershipPort: 6933
          pprofPort: 7936
      matching:
        rpc:
          grpcPort: 7235
          membershipPort: 6935
          pprofPort: 7937
      history:
        rpc:
          grpcPort: 7234
          membershipPort: 6934
          pprofPort: 7938
      worker:
        rpc:
          grpcPort: 7239
          membershipPort: 6939
          pprofPort: 7941
```

Backup and Recovery Strategy

```
# backup_strategy.py
import boto3
import json
from datetime import datetime, timedelta
from temporalio.client import Client

class TemporalBackupStrategy:
    def __init__(self, temporal_client: Client, s3_bucket: str):
        self.client = temporal_client
        self.s3_bucket = s3_bucket
        self.s3_client = boto3.client('s3')

    async def backup_workflow_histories(self, workflow_ids: list):
        """Backup workflow histories to S3."""
        backup_data = {
            "timestamp": datetime.utcnow().isoformat(),
            "workflows": []
        }

        for workflow_id in workflow_ids:
            try:
                handle = self.client.get_workflow_handle(workflow_id)
                history = await handle.fetch_history()

                backup_data["workflows"].append({
                    "workflow_id": workflow_id,
                    "history": history
                })

            except Exception as e:
                print(f"Failed to backup workflow {workflow_id}: {e}")

        # Upload to S3
        backup_key =
f"backups/workflow_histories_{datetime.utcnow().strftime('%Y%m%d_%H%M%S')}.
json"
        self.s3_client.put_object(
            Bucket=self.s3_bucket,
            Key=backup_key,
            Body=json.dumps(backup_data, indent=2)
        )

        return backup_key

    async def restore_workflow_histories(self, backup_key: str):
        """Restore workflow histories from S3 backup."""
        response = self.s3_client.get_object(
            Bucket=self.s3_bucket,
            Key=backup_key
        )
```

```
        backup_data = json.loads(response['Body'].read())

        for workflow_info in backup_data["workflows"]:
            try:
                # Restore workflow with history
                await self.client.start_workflow(
                    workflow_type="RestoredWorkflow",
                    id=workflow_info["workflow_id"],
                    task_queue="restore-queue",
                    workflow_id_reuse_policy="AllowDuplicate"
                )

            except Exception as e:
                print(f"Failed to restore workflow
{workflow_info['workflow_id']}: {e}")

    def cleanup_old_backups(self, retention_days: int = 30):
        """Clean up backups older than retention period."""
        cutoff_date = datetime.utcnow() - timedelta(days=retention_days)

        response = self.s3_client.list_objects_v2(
            Bucket=self.s3_bucket,
            Prefix="backups/"
        )

        for obj in response.get('Contents', []):
            if obj['LastModified'].replace(tzinfo=None) < cutoff_date:
                self.s3_client.delete_object(
                    Bucket=self.s3_bucket,
                    Key=obj['Key']
                )
                print(f"Deleted old backup: {obj['Key']}")
```

Production Deployment Considerations

Security Considerations

```
# security-config.yaml
apiVersion: v1
kind: Secret
metadata:
  name: temporal-secrets
type: Opaque
data:
  # Base64 encoded secrets
  temporal-cert: <base64-encoded-cert>
  temporal-key: <base64-encoded-key>
  db-password: <base64-encoded-password>
```

```yaml
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: temporal-security-config
data:
  security.yaml: |
    tls:
      enabled: true
      certFile: /etc/temporal/certs/temporal.crt
      keyFile: /etc/temporal/certs/temporal.key
      caFile: /etc/temporal/certs/ca.crt
      requireClientAuth: true

    authorization:
      enabled: true
      jwt:
        enabled: true
        keyFile: /etc/temporal/jwt/public.key
        algorithm: RS256
```

Resource Planning

```yaml
# resource-planning.yaml
apiVersion: v1
kind: ResourceQuota
metadata:
  name: temporal-resource-quota
spec:
  hard:
    requests.cpu: "10"
    requests.memory: 20Gi
    limits.cpu: "20"
    limits.memory: 40Gi
    persistentvolumeclaims: "10"

---
apiVersion: v1
kind: LimitRange
metadata:
  name: temporal-limit-range
spec:
  limits:
  - default:
      cpu: "500m"
      memory: "1Gi"
    defaultRequest:
      cpu: "250m"
      memory: "512Mi"
    type: Container
```

Monitoring and Alerting

```yaml
# monitoring-config.yaml
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  name: temporal-monitor
spec:
  selector:
    matchLabels:
      app: temporal
  endpoints:
  - port: metrics
    interval: 15s

---
apiVersion: monitoring.coreos.com/v1
kind: PrometheusRule
metadata:
  name: temporal-alerts
spec:
  groups:
  - name: temporal
    rules:
    - alert: TemporalServerDown
      expr: up{job="temporal"} == 0
      for: 1m
      labels:
        severity: critical
      annotations:
        summary: "Temporal server is down"
        description: "Temporal server has been down for more than 1 minute"

    - alert: HighWorkflowFailureRate
      expr: rate(temporal_workflow_execution_failed_total[5m]) > 0.1
      for: 2m
      labels:
        severity: warning
      annotations:
        summary: "High workflow failure rate"
        description: "Workflow failure rate is {{ $value }} failures per second"
```

Practical: Setting Up Monitoring

Step 1: Deploy Monitoring Stack

```
# Deploy Prometheus and Grafana
helm repo add prometheus-community https://prometheus-
```

```
community.github.io/helm-charts
helm repo update

helm install prometheus prometheus-community/kube-prometheus-stack \
  --namespace monitoring \
  --create-namespace \
  --values monitoring-values.yaml
```

Step 2: Configure Temporal Metrics

```python
# worker_with_metrics.py
import asyncio
from temporalio.client import Client
from temporalio.worker import Worker
from prometheus_client import start_http_server, Counter, Histogram
from your_workflows import OrderProcessingWorkflow
from your_activities import validate_order, process_payment

# Define metrics
workflow_executions = Counter(
    'temporal_workflow_executions_total',
    'Total workflow executions',
    ['workflow_type', 'status']
)

workflow_duration = Histogram(
    'temporal_workflow_duration_seconds',
    'Workflow execution duration',
    ['workflow_type']
)

activity_executions = Counter(
    'temporal_activity_executions_total',
    'Total activity executions',
    ['activity_type', 'status']
)

async def main():
    # Start metrics server
    start_http_server(9090)

    # Connect to Temporal
    client = await Client.connect("localhost:7233")

    # Start worker
    worker = Worker(
        client,
        task_queue="monitored-queue",
        workflows=[OrderProcessingWorkflow],
        activities=[validate_order, process_payment]
    )
```

```
    await worker.run()

if __name__ == "__main__":
    asyncio.run(main())
```

Step 3: Create Grafana Dashboard

```
{
  "dashboard": {
    "title": "Temporal Production Monitoring",
    "panels": [
      {
        "title": "Workflow Execution Overview",
        "type": "row",
        "panels": [
          {
            "title": "Execution Rate",
            "type": "graph",
            "targets": [
              {
                "expr":
"rate(temporal_workflow_execution_started_total[5m])",
                "legendFormat": "{{workflow_type}}"
              }
            ]
          },
          {
            "title": "Success Rate",
            "type": "graph",
            "targets": [
              {
                "expr":
"rate(temporal_workflow_execution_completed_total[5m]) /
rate(temporal_workflow_execution_started_total[5m])",
                "legendFormat": "Success Rate"
              }
            ]
          }
        ]
      },
      {
        "title": "System Health",
        "type": "row",
        "panels": [
          {
            "title": "Task Queue Depth",
            "type": "graph",
            "targets": [
              {
                "expr": "temporal_task_queue_depth",
```

```
                "legendFormat": "{{task_queue}}"
              }
            ]
          },
          {
            "title": "Worker Utilization",
            "type": "gauge",
            "targets": [
              {
                "expr": "(temporal_worker_task_slots_total - temporal_worker_task_slots_available) / temporal_worker_task_slots_total",
                "legendFormat": "Utilization"
              }
            ]
          }
        ]
      }
    ]
  }
}
```

Step 4: Set Up Alerting

```yaml
# alertmanager-config.yaml
global:
  slack_api_url: 'https://hooks.slack.com/services/YOUR/SLACK/WEBHOOK'

route:
  group_by: ['alertname']
  group_wait: 10s
  group_interval: 10s
  repeat_interval: 1h
  receiver: 'slack-notifications'

receivers:
- name: 'slack-notifications'
  slack_configs:
  - channel: '#temporal-alerts'
    title: '{{ template "slack.title" . }}'
    text: '{{ template "slack.text" . }}'
    actions:
    - type: button
      text: 'View in Grafana'
      url: '{{ template "slack.grafana" . }}'

templates:
- '/etc/alertmanager/template/*.tmpl'
```

This comprehensive approach to observability, monitoring, and deployment ensures that your Temporal applications are production-ready, maintainable, and reliable at scale. The combination of metrics, logging, tracing, and proper deployment strategies provides the foundation for successful production operations.