Chapter 9: Run, Monitor & Maintain (7%)

Every great agent is like a river — once it begins to flow, it never moves in a straight line. It bends, adjusts, corrects itself, and keeps moving forward. But this flow is not accidental. It requires measurement, intervention, and continuous care.

Monitoring and maintenance ensure that an agent stays trustworthy, accurate, and ready for real-world demands.

This chapter focuses on deploying operational monitoring, tracking performance, identifying degradation early, and setting up automated improvement loops — exactly the type of real-world readiness the NVIDIA exam expects.

9.1 Observability & Metrics

Once an agent is deployed, observability becomes your primary tool for understanding how it behaves in production.

Exam expectations focus on tracking real-time metrics that indicate reliability, responsiveness, and efficiency.

Key Reliability Metrics

- Latency (response time per request)
- Throughput (requests processed per second)
- Accuracy & output stability
- Cost per request (critical for GPU workloads)
- Error rate & anomaly spikes

Prometheus and Grafana are widely used to track and visualize these metrics in long-running agentic systems.

Example: Collecting Metrics with Prometheus (Python)

Pre-Requisite:

Ensure the `prometheus_client` is installed using the command:

```
pip install prometheus_client
```

Code Implementation:

```
from prometheus_client import Counter, Histogram, start_http_server
import time

REQUEST_COUNT = Counter('agent_requests_total', 'Total requests processed')
RESPONSE_LATENCY = Histogram('agent_response_latency_seconds',
                             'Response latency in seconds')

start_http_server(8000)

def agent_respond(query):
    start = time.time()
    time.sleep(0.2)  # Simulated inference time
    RESPONSE_LATENCY.observe(time.time() - start)
    REQUEST_COUNT.inc()
    return "Response generated"

while True:
    agent_respond("What is AI?")
```

Goto URL:

`http://localhost:8000/`

Sample Logs Generated by Prometheus:

```
# TYPE agent_requests_total counter
agent_requests_total 72.0
# HELP agent_requests_created Total requests processed
# TYPE agent_requests_created gauge
agent_requests_created 1.7632280142349288e+09
# HELP agent_response_latency_seconds Response latency in seconds
# TYPE agent_response_latency_seconds histogram
agent_response_latency_seconds_bucket{le="0.005"} 0.0
agent_response_latency_seconds_bucket{le="0.01"} 0.0
agent_response_latency_seconds_bucket{le="0.025"} 0.0
agent_response_latency_seconds_bucket{le="0.05"} 0.0
agent_response_latency_seconds_bucket{le="0.075"} 0.0
agent_response_latency_seconds_bucket{le="0.1"} 0.0
agent_response_latency_seconds_bucket{le="0.25"} 72.0
agent_response_latency_seconds_bucket{le="0.5"} 72.0
agent_response_latency_seconds_bucket{le="0.75"} 72.0
agent_response_latency_seconds_bucket{le="1.0"} 72.0
agent_response_latency_seconds_bucket{le="2.5"} 72.0
agent_response_latency_seconds_bucket{le="5.0"} 72.0
agent_response_latency_seconds_bucket{le="7.5"} 72.0
agent_response_latency_seconds_bucket{le="10.0"} 72.0
agent_response_latency_seconds_bucket{le="+Inf"} 72.0
agent_response_latency_seconds_count 72.0
agent_response_latency_seconds_sum 14.711804151535034
# HELP agent_response_latency_seconds_created Response latency in seconds
# TYPE agent_response_latency_seconds_created gauge
agent_response_latency_seconds_created 1.7632280142349398e+09
```

This forms the foundation for monitoring dashboards used in exam scenarios.

9.2 Monitoring Models & Agents

Monitoring extends beyond infrastructure — you must also track model and agent behavior over time.

The NVIDIA exam focuses heavily on understanding drift, version degradation, and state tracking.

Types of Drift

- **Data Drift** — Input data distribution changes
- **Model Drift** — Model behavior deviates from baseline
- **Feature Importance Drift** — Certain features become predictive

LangGraph's Checkpointer helps persist state so you can compare baseline vs current executions.

Code Implementation for Drift Tracking:

```
BASELINE_FILE = "drift_baseline.json"

def save_baseline(vector):
    """Persist the baseline vector to a local JSON file."""
    with open(BASELINE_FILE, "w") as f:
        json.dump({"baseline": vector}, f)

def load_baseline():
    """Load baseline vector if it exists."""
    if not os.path.exists(BASELINE_FILE):
        return None
    with open(BASELINE_FILE, "r") as f:
        data = json.load(f)
        return data.get("baseline")
```

```
def detect_drift(baseline, current, threshold=0.15, metric="feature_importance_drift"):
    """
    Simple L1 drift detection.
    baseline/current are vectors like [0.1, 0.5, 0.4].
    Returns a dictionary with drift detection results in standardized format.
    """
    if baseline is None:
        print("No baseline found. Initializing with current vector.")
        save_baseline(current)
        return {
            "timestamp": datetime.now(timezone.utc).isoformat().replace("+00:00", "Z"),
            "metric": metric,
            "baseline_distribution": current,
            "current_distribution": current,
            "drift_detected": False
        }

    diff = sum(abs(b - c) for b, c in zip(baseline, current))
    drift_detected = diff > threshold

    if drift_detected:
        print(f"Drift detected! change={diff:.3f} > threshold={threshold}")
    else:
        print(f"Model stable. change={diff:.3f}")

    return {
        "timestamp": datetime.now(timezone.utc).isoformat().replace("+00:00", "Z"),
        "metric": metric,
        "baseline_distribution": baseline,
        "current_distribution": current,
        "drift_detected": drift_detected
    }
```

```
# ----------------------------
# Example Usage
# ----------------------------

# Baseline feature vector (e.g., feature importance or embedding stats)
initial_vector = [0.1, 0.5, 0.4]
save_baseline(initial_vector)

# New vector from current model behavior
current_vector = [0.2, 0.7, 0.1]

baseline = load_baseline()
result = detect_drift(baseline, current_vector)

# Output in standardized JSON format
print("\nDrift Detection Result:")
print(json.dumps(result, indent=2))
```

Sample Output:

```
Drift detected! change=0.600 > threshold=0.15

Drift Detection Result:
{
  "timestamp": "2025-11-15T17:38:30.781510Z",
  "metric": "feature_importance_drift",
  "baseline_distribution": [
    0.1,
    0.5,
    0.4
  ],
  "current_distribution": [
    0.2,
    0.7,
    0.1
  ],
  "drift_detected": true
}
```

9.3 Automated Retraining Pipelines

As soon as drift or performance degradation is detected, retraining pipelines restore accuracy and reliability.

The NVIDIA exam expects you to understand:

- How drift triggers automation
- How new data re-aligns the model
- How validation gates ensure safety
- How redeployment closes the loop

Typical Retraining Loop

1. **Monitor** metrics continuously
2. **Trigger** retraining on drift
3. **Retrain** with updated datasets
4. **Validate** against baseline test suites
5. **Re-deploy** if quality improves

Trigger mechanisms include CI/CD pipelines, scheduled jobs, event-driven systems (like AWS Lambda), or LangGraph task nodes.

9.4 Logging & Alerting

Logs offer a structured snapshot of how the agent responds under varying conditions.
They are critical for debugging, root-cause analysis, traceability, and alerting.

Important Log Fields

- Timestamp
- Request ID
- Query summary
- Response latency
- Drift score
- Status (stable / alert)

Example Log Entry:

```
2025-11-04T09:42:11Z | agent_response | latency=0.29s | drift=0.18 | status=alert
2025-11-04T09:45:18Z | agent_response | latency=0.22s | drift=0.10 | status=stable
```

Log Entry Format

Each log entry contains:

- **Timestamp**: ISO 8601 format (UTC)
- **Event Type**: agent_response
- **Latency**: Response time in seconds
- **Drift**: Drift metric value (higher values indicate more drift)
- **Status**: Current status (alert or stable)

9.5 Before You Begin

You're not alone when you get errors, Join the Community with me & Fellow Learners to ask questions:

https://ncpaai.agenticailaunchpad.com/

(Exclusive for verified learners. This link always points to the latest NVIDIA Agentic AI Certification Lounge – updates and new access portals available anytime on this link)

(Alternatively Scan the QR code)

If you're new to Python or LangChain/LangGraph, don't worry! Access your free Python Foundation and LangChain & LangGraph video tutorials that accompany this book. Visit : https://resources.agenticailaunchpad.com or scan the QR code below book to get started.

9.6 Mini Lab – Build a Monitoring Hook for Your Agent

Modern agentic systems must be continuously observed, measured, and refined.

In this mini-lab, you will build a simple production-inspired monitoring hook that logs every inference event along with key signals such as latency and drift score.

Objective:

In this lab, you will:

- Capture agent activity using a lightweight monitoring hook
- Log every inference event in structured JSON format
- Track response latency
- Record a simple drift indicator
- Produce a JSONL log file that mimics a production observability pipeline

This is the same conceptual pipeline used in the NVIDIA Agentic AI exam:
understanding how to observe, measure, and respond to model behavior changes.

Code Implementation:

```
import json
import time

def monitor_hook(query, response_time, drift_score):
    log = {
        "timestamp": time.time(),
        "query": query,
        "latency": response_time,
        "drift_score": drift_score
    }
    with open("agent_logs.jsonl", "a") as f:
        f.write(json.dumps(log) + "\n")

# Example call
monitor_hook("Summarize report", 0.21, 0.12)
```

Sample Output:

```
{"timestamp": 1763227811.128711, "query": "Summarize report", "latency": 0.21, "drift_score": 0.12}
```

9.7 Case Study – Self-Optimizing Chatbot

A healthcare chatbot tracked user satisfaction, latency, and drift metrics.
Whenever performance dropped below a threshold, a retraining job triggered automatically.

Results Over Six Months

- 25% improvement in accuracy

- 15% reduction in latency
- Increased trust and reduced user-reported errors.

This illustrates how real-time monitoring combined with automated retraining forms a **self-improving system**.

Exam Tip: NVIDIA exam questions emphasize:

- detecting drift
- interpreting logs
- reliability metrics
- retraining triggers
- benchmark comparisons

Focus on thresholds and stabilization strategies.

9.8 Key Terms

Data Drift, Model Drift, Checkpointer, Metric, Log, Retraining Pipeline, Alert Threshold

9.9 Milestone Checklist

- I can interpret latency, throughput, and cost metrics.

- I can monitor agent behavior using Prometheus & Grafana.
- I can detect drift using LangGraph checkpointers.
- I can describe automated retraining pipelines.
- I understand logging structure and alerting strategies.
- I can evaluate production agents against baseline versions.

"Maintenance is the art of keeping momentum alive." – Nachiketh Murthy

9.10 CTA Zone

Ready to test your knowledge and take the learning to next level? Your Book Purchase already comes with sample questions on each chapter, now dive deeper and master this with additional 2 full 70-question practice test with Full Video course at https://upgrade.agenticailaunchpad.com

Scan the QR below to unlock your Practice Test with Full Video Course Pack at **₹2,999 / $49**