项目2：LLM 推理服务可观测性方案 — 实施指导书

> **定位：** 项目1的深度延伸。证明你不只会部署，还能保障服务质量。
> **前置条件：** 项目1已完成（K8s + vLLM + Prometheus + Grafana 已部署）
> **预计工期：** 4 周（第3-6周，与项目1部分并行）
> **核心交付物：** 专业级 Grafana Dashboard + 自定义 Exporter + 告警规则 + 最佳实践文档

时间安排总览

周次	天数	目标	核心产出
第3周	Day 1-7	基础搭建 + GPU 指标采集	Prometheus + DCGM 完整对接
第4周	Day 8-14	自定义 vLLM Exporter 开发	Python Exporter + 推理层指标采集
第5周	Day 15-21	Dashboard 打磨 + 告警配置	三层 Grafana 面板 + 5+ 告警规则
第6周	Day 22-28	OpenTelemetry + 文档 + 上线	最佳实践文档 + GitHub 仓库上线

第3周：基础搭建 + GPU 指标深度采集（Day 1-7）

> 如果你已经在项目1中完成了 Prometheus + Grafana + DCGM 的部署，这周主要是**深化理解和补齐缺失的配置**。

Day 1-2：确认现有监控栈状态

Step 1：检查所有监控组件是否正常运行

```
# 查看 monitoring 命名空间所有 Pod
kubectl -n monitoring get pod

# 期望结果：所有 Pod 状态为 Running
# - prometheus-xxx                    2/2 Running
# - monitoring-grafana-xxx            3/3 Running
```

```
# - alertmanager-xxx                 2/2 Running
# - dcgm-exporter-xxx                1/1 Running
# - monitoring-kube-state-metrics    1/1 Running
# - monitoring-prometheus-node-exporter  1/1 Running
# - monitoring-kube-prometheus-operator  1/1 Running
```

Step 2：确认 Prometheus 采集目标正常

```
# 端口转发 Prometheus
kubectl port-forward svc/monitoring-kube-prometheus-prometh
eus -n monitoring 9090:9090 --address 0.0.0.0 &

# 查看所有 targets 状态
curl -s localhost:9090/api/v1/targets | python3 -c "
import sys,json
data=json.load(sys.stdin)
for t in data['data']['activeTargets']:
    print(f\"{t['scrapePool']:60s} -> {t['health']}\")
"
```

所有 target 应该是 `up` 状态。如果有 `down` 的，需要先排查修复。

Step 3：确认 DCGM 和 vLLM 指标已接入

```
# 验证 GPU 指标
curl -s localhost:9090/api/v1/query?query=DCGM_FI_DEV_GPU_U
TIL | python3 -c "
import sys,json; data=json.load(sys.stdin)
print('GPU指标:', '✅ 有数据' if data['data']['result'] else
'❌ 无数据')
"

# 验证 vLLM 指标
curl -s localhost:9090/api/v1/query?query=vllm:num_requests
_running | python3 -c "
import sys,json; data=json.load(sys.stdin)
print('vLLM指标:', '✅ 有数据' if data['data']['result'] els
```

```
e '❌ 无数据')
"
```

Day 3-4：创建项目仓库结构

Step 4：初始化项目2仓库

```
mkdir -p ~/llm-observability-stack
cd ~/llm-observability-stack

# 创建完整目录结构
mkdir -p docs
mkdir -p exporters/vllm_exporter
mkdir -p prometheus
mkdir -p grafana/dashboards
mkdir -p grafana/provisioning
mkdir -p alertmanager
mkdir -p otel
mkdir -p k8s
mkdir -p scripts

# 初始化 Git
git init
```

Step 5：梳理需要采集的完整指标清单

创建指标清单文档，这是整个项目的规划蓝图：

```
cat > docs/metrics-catalog.md << 'EOF'
# LLM 推理服务监控指标目录

## 第一层：GPU 硬件层（DCGM Exporter 采集）

| 指标名 | 含义 | 单位 | 告警阈值建议 |
|--------|------|------|-------------|
| DCGM_FI_DEV_GPU_UTIL | GPU 计算核心利用率 | % | >95% 持续5分
钟 |
| DCGM_FI_DEV_FB_USED | 已使用显存 | MiB | >90% 总显存 |
| DCGM_FI_DEV_FB_FREE | 可用显存 | MiB | <2048MiB |
```

```
| DCGM_FI_DEV_GPU_TEMP | GPU 温度 | ℃ | >80℃ |
| DCGM_FI_DEV_POWER_USAGE | GPU 功耗 | W | >140W (A10) |
| DCGM_FI_DEV_SM_CLOCK | SM 时钟频率 | MHz | 降频告警 |
| DCGM_FI_DEV_MEM_CLOCK | 显存时钟频率 | MHz | 降频告警 |
| DCGM_FI_DEV_PCIE_TX_THROUGHPUT | PCIe 发送带宽 | bytes/s | 信息采集 |
| DCGM_FI_DEV_PCIE_RX_THROUGHPUT | PCIe 接收带宽 | bytes/s | 信息采集 |
| DCGM_FI_DEV_ECC_SBE_VOL_TOTAL | ECC 单比特错误 | count | >0 |
| DCGM_FI_DEV_ECC_DBE_VOL_TOTAL | ECC 双比特错误 | count | >0（严重） |

## 第二层：推理服务层（vLLM 原生 + 自定义 Exporter 采集）

### vLLM 原生暴露的指标

| 指标名 | 含义 | 类型 |
|---|---|---|
| vllm:e2e_request_latency_seconds | 端到端请求延迟 | Histogram |
| vllm:num_requests_running | 正在处理的请求数 | Gauge |
| vllm:num_requests_waiting | 等待中的请求数 | Gauge |
| vllm:num_preemptions_total | 抢占次数 | Counter |
| vllm:gpu_cache_usage_perc | GPU KV Cache 使用率 | Gauge |
| vllm:cpu_cache_usage_perc | CPU KV Cache 使用率 | Gauge |
| vllm:prompt_tokens_total | 输入 Token 总数 | Counter |
| vllm:generation_tokens_total | 生成 Token 总数 | Counter |
| vllm:time_to_first_token_seconds | 首 Token 延迟（TTFT） | Histogram |
| vllm:time_per_output_token_seconds | 每个输出 Token 耗时 | Histogram |

### 自定义 Exporter 补充采集的指标

| 指标名 | 含义 | 来源 |
|---|---|---|
| vllm_model_loaded | 模型是否加载完成 | /health 端点 |
```

```
| vllm_available_models | 可用模型数量 | /v1/models 端点 |
| vllm_request_success_total | 成功请求计数 | /v1/chat/comple
tions |
| vllm_request_error_total | 失败请求计数（按错误类型） | /v1/c
hat/completions |
| vllm_input_tokens_per_request | 每请求输入 Token 数分布 | /
v1/chat/completions |
| vllm_output_tokens_per_request | 每请求输出 Token 数分布 |
/v1/chat/completions |

## 第三层：业务层（自定义 Exporter + Prometheus 规则计算）

| 指标/规则 | 含义 | 计算方式 |
|-----------|------|---------|
| QPS | 每秒请求数 | rate(vllm:e2e_request_latency_seconds_c
ount[1m]) |
| 请求成功率 | 成功/总数 | 1 - (error_total / total_requests)
|
| P50/P95/P99 延迟 | 延迟分位数 | histogram_quantile(0.99,
...) |
| Token 吞吐量 | 每秒生成 Token 数 | rate(vllm:generation_tok
ens_total[1m]) |
| KV Cache 耗尽风险 | Cache 使用率趋势 | predict_linear(gpu_ca
che_usage[10m], 300) |
EOF
```

Day 5-7：深化 DCGM 指标采集和理解

Step 6：确认 DCGM Exporter 采集的完整指标列表

```
# 直接查看 DCGM 暴露了哪些指标
kubectl port-forward daemonset/dcgm-exporter -n monitoring
9400:9400 &
curl -s localhost:9400/metrics | grep "^DCGM" | awk '{print
$1}' | cut -d'{' -f1 | sort -u
```

把输出的指标列表和上面的 metrics-catalog.md 对比，确认哪些指标已经在采集，哪些还需要补充。

Step 7：在 Prometheus 中验证指标查询

逐一在 Prometheus UI（http://<IP>:9090）中测试以下 PromQL 查询，确保每个都有数据：

```
# GPU 利用率（当前值）
DCGM_FI_DEV_GPU_UTIL

# GPU 利用率（5分钟平均）
avg_over_time(DCGM_FI_DEV_GPU_UTIL[5m])

# 显存使用率百分比
DCGM_FI_DEV_FB_USED / (DCGM_FI_DEV_FB_USED + DCGM_FI_DEV_FB
_FREE) * 100

# vLLM QPS（每秒请求数）
rate(vllm:e2e_request_latency_seconds_count[1m])

# vLLM P99 延迟
histogram_quantile(0.99, rate(vllm:e2e_request_latency_seco
nds_bucket[5m]))

# vLLM 首 Token 延迟 P50
histogram_quantile(0.5, rate(vllm:time_to_first_token_secon
ds_bucket[5m]))

# Token 生成吞吐量（每秒）
rate(vllm:generation_tokens_total[1m])

# KV Cache 使用率
vllm:gpu_cache_usage_perc
```

> **重要提示：** vLLM 不同版本暴露的指标名可能不同。有些版本用冒号分隔（`vllm:xxx`），有些用下划线（`vllm_xxx`）。先到 vLLM Pod 的 /metrics 端点看实际暴露了什么指标名，然后调整查询。

```
# 查看 vLLM 实际暴露的指标名
kubectl port-forward svc/llm-prod-svc -n llm-serving 8000:8
```

```
000 --address 0.0.0.0 &
curl -s localhost:8000/metrics | grep "^vllm" | awk '{print
$1}' | cut -d'{' -f1 | sort -u
```

第4周：自定义 vLLM Exporter 开发（Day 8-14）

> **目标：** 编写一个 Python 自定义 Prometheus Exporter，补充采集 vLLM 原生不暴露的业务层指标。

Day 8-9：理解 Prometheus Exporter 原理

Exporter 是什么：

Exporter 就是一个 HTTP 服务，暴露一个 `/metrics` 端点，按 Prometheus 格式返回指标数据。Prometheus 定时来拉取这个端点的数据。

```
Prometheus  --每15秒拉取-->  Exporter(:9101/metrics)  --调用-
->  vLLM(:8000/v1/models)
                                                          --调
用-->  vLLM(:8000/health)
                                                          --调
用-->  vLLM(:8000/metrics)
```

Step 8：编写自定义 vLLM Exporter

```
cat > ~/llm-observability-stack/exporters/vllm_exporter/exp
orter.py << 'PYEOF'
"""
vLLM 自定义 Prometheus Exporter
采集 vLLM 推理服务的业务层指标，补充原生 /metrics 不暴露的信息
"""
import time
import logging
import argparse
import requests
from prometheus_client import (
    start_http_server,
    Gauge,
    Counter,
```

```
    Histogram,
    Info,
)

logging.basicConfig(level=logging.INFO, format='%(asctime)s
%(levelname)s %(message)s')
logger = logging.getLogger(__name__)


class VLLMExporter:
    """vLLM 自定义指标采集器"""

    def __init__(self, vllm_url: str):
        self.vllm_url = vllm_url.rstrip('/')

        # ===== 服务健康指标 =====
        self.model_loaded = Gauge(
            'vllm_model_loaded',
            '模型是否加载完成（1=已加载，0=未加载）'
        )
        self.service_up = Gauge(
            'vllm_service_up',
            'vLLM 服务是否可达（1=可达，0=不可达）'
        )
        self.available_models = Gauge(
            'vllm_available_models_total',
            '当前可用的模型数量'
        )

        # ===== 模型信息 =====
        self.model_info = Info(
            'vllm_model',
            '当前加载的模型信息'
        )

        # ===== 健康检查延迟 =====
        self.health_check_latency = Histogram(
            'vllm_health_check_latency_seconds',
```

```
            '健康检查端点响应延迟',
            buckets=[0.001, 0.005, 0.01, 0.025, 0.05, 0.1,
0.25, 0.5, 1.0]
        )

        # ===== 探测请求指标 =====
        self.probe_request_success = Counter(
            'vllm_probe_request_success_total',
            '探测请求成功次数'
        )
        self.probe_request_error = Counter(
            'vllm_probe_request_error_total',
            '探测请求失败次数',
            ['error_type']
        )

        # ===== 模型配置指标 =====
        self.max_model_len = Gauge(
            'vllm_max_model_len',
            '模型最大序列长度'
        )

        # ===== 采集状态 =====
        self.last_scrape_duration = Gauge(
            'vllm_exporter_scrape_duration_seconds',
            '上一次指标采集耗时'
        )
        self.scrape_errors = Counter(
            'vllm_exporter_scrape_errors_total',
            '指标采集错误次数',
            ['endpoint']
        )

    def collect_health(self):
        """采集健康检查指标"""
        try:
            start = time.time()
            resp = requests.get(f"{self.vllm_url}/health",
```

```
timeout=5)
            latency = time.time() - start

            self.health_check_latency.observe(latency)

            if resp.status_code == 200:
                self.model_loaded.set(1)
                self.service_up.set(1)
                self.probe_request_success.inc()
            else:
                self.model_loaded.set(0)
                self.service_up.set(1)  # 服务可达但模型未就绪
                logger.warning(f"Health check returned {res
p.status_code}")

        except requests.exceptions.ConnectionError:
            self.service_up.set(0)
            self.model_loaded.set(0)
            self.probe_request_error.labels(error_type='con
nection_error').inc()
            self.scrape_errors.labels(endpoint='health').in
c()
            logger.error("Cannot connect to vLLM service")

        except requests.exceptions.Timeout:
            self.service_up.set(1)  # 可达但超时
            self.model_loaded.set(0)
            self.probe_request_error.labels(error_type='tim
eout').inc()
            self.scrape_errors.labels(endpoint='health').in
c()
            logger.error("Health check timed out")

    def collect_models(self):
        """采集模型信息指标"""
        try:
            resp = requests.get(f"{self.vllm_url}/v1/model
s", timeout=5)
```

```
            if resp.status_code == 200:
                data = resp.json()
                models = data.get('data', [])
                self.available_models.set(len(models))

                if models:
                    model = models[0]
                    self.model_info.info({
                        'id': model.get('id', 'unknown'),
                        'owned_by': model.get('owned_by',
'unknown'),
                        'root': model.get('root', 'unknow
n'),
                    })
                    # 采集 max_model_len
                    max_len = model.get('max_model_len', 0)
                    if max_len:
                        self.max_model_len.set(max_len)

                self.probe_request_success.inc()
            else:
                self.scrape_errors.labels(endpoint='model
s').inc()
                logger.warning(f"/v1/models returned {resp.
status_code}")

        except Exception as e:
            self.scrape_errors.labels(endpoint='models').in
c()
            logger.error(f"Failed to collect model info:
{e}")

    def collect_all(self):
        """执行一次完整的指标采集"""
        start = time.time()

        self.collect_health()
        self.collect_models()
```

```
        duration = time.time() - start
        self.last_scrape_duration.set(duration)
        logger.info(f"Scrape completed in {duration:.3f}s")

    def run(self, port: int = 9101, interval: int = 15):
        """启动 Exporter HTTP 服务并定时采集"""
        start_http_server(port)
        logger.info(f"vLLM Exporter started on port {por
t}")
        logger.info(f"Scraping {self.vllm_url} every {inter
val}s")

        while True:
            try:
                self.collect_all()
            except Exception as e:
                logger.error(f"Unexpected error during coll
ection: {e}")
            time.sleep(interval)


if __name__ == '__main__':
    parser = argparse.ArgumentParser(description='vLLM Cust
om Prometheus Exporter')
    parser.add_argument('--vllm-url', default='http://local
host:8000',
                        help='vLLM 服务地址 (default: htt
p://localhost:8000)')
    parser.add_argument('--port', type=int, default=9101,
                        help='Exporter 监听端口 (default: 91
01)')
    parser.add_argument('--interval', type=int, default=15,
                        help='采集间隔秒数 (default: 15)')
    args = parser.parse_args()

    exporter = VLLMExporter(args.vllm_url)
```

```
    exporter.run(port=args.port, interval=args.interval)
PYEOF
```

Step 9：编写 Exporter 的依赖和 Dockerfile

```
# requirements.txt
cat > ~/llm-observability-stack/exporters/vllm_exporter/req
uirements.txt << 'EOF'
prometheus_client==0.20.0
requests==2.31.0
EOF

# Dockerfile
cat > ~/llm-observability-stack/exporters/vllm_exporter/Doc
kerfile << 'EOF'
FROM python:3.11-slim

WORKDIR /app

COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

COPY exporter.py .

EXPOSE 9101

ENTRYPOINT ["python", "exporter.py"]
CMD ["--vllm-url", "http://llm-prod-svc.llm-serving.svc.clu
ster.local:8000", "--port", "9101"]
EOF
```

Day 10-11：在 K8s 中部署自定义 Exporter

Step 10：构建并推送 Exporter 镜像

```
cd ~/llm-observability-stack/exporters/vllm_exporter

# 构建镜像
docker build -t vllm-exporter:v1 .
```

```
# 如果用 K3s 的 containerd
# nerdctl -n k8s.io build -t vllm-exporter:v1 .

# 推送到你的阿里云镜像仓库（替换为你的实际地址）
docker tag vllm-exporter:v1 <你的镜像仓库>/vllm-exporter:v1
docker push <你的镜像仓库>/vllm-exporter:v1
```

> **简化方案（适合学习）：** 如果不想折腾镜像构建，可以直接在节点上用 Python 运行 Exporter，然后用 hostNetwork Pod 或 NodePort Service 暴露给 Prometheus。

Step 11：用 K8s 部署 Exporter（简化方案）

```
cat > ~/llm-observability-stack/k8s/vllm-exporter-deploymen
t.yaml << 'EOF'
apiVersion: apps/v1
kind: Deployment
metadata:
  name: vllm-exporter
  namespace: monitoring
  labels:
    app: vllm-exporter
spec:
  replicas: 1
  selector:
    matchLabels:
      app: vllm-exporter
  template:
    metadata:
      labels:
        app: vllm-exporter
    spec:
      containers:
        - name: vllm-exporter
          image: python:3.11-slim
          command:
            - /bin/bash
            - -c
```

```
            - |
              pip install prometheus_client requests && \
              python /app/exporter.py \
                --vllm-url http://llm-prod-svc.llm-serving.
svc.cluster.local:8000 \
                --port 9101 \
                --interval 15
          ports:
            - containerPort: 9101
              name: metrics
          volumeMounts:
            - name: exporter-code
              mountPath: /app
          resources:
            requests:
              cpu: 100m
              memory: 128Mi
            limits:
              cpu: 200m
              memory: 256Mi
          livenessProbe:
            httpGet:
              path: /metrics
              port: 9101
            initialDelaySeconds: 30
            periodSeconds: 30
      volumes:
        - name: exporter-code
          configMap:
            name: vllm-exporter-code
---
apiVersion: v1
kind: Service
metadata:
  name: vllm-exporter
  namespace: monitoring
  labels:
    app: vllm-exporter
```

```
spec:
  selector:
    app: vllm-exporter
  ports:
    - name: metrics
      port: 9101
      targetPort: 9101
---
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  name: vllm-exporter
  namespace: monitoring
  labels:
    release: monitoring
spec:
  selector:
    matchLabels:
      app: vllm-exporter
  endpoints:
    - port: metrics
      interval: 15s
      path: /metrics
EOF
```

Step 12：用 ConfigMap 挂载 Exporter 代码

```
# 把 exporter.py 创建为 ConfigMap
kubectl -n monitoring create configmap vllm-exporter-code \
  --from-file=exporter.py=~/llm-observability-stack/exporte
rs/vllm_exporter/exporter.py

# 部署
kubectl apply -f ~/llm-observability-stack/k8s/vllm-exporte
r-deployment.yaml

# 确认 Pod 运行
kubectl -n monitoring get pod -l app=vllm-exporter
```

```
# 验证指标
kubectl port-forward svc/vllm-exporter -n monitoring 9101:9
101 &
curl -s localhost:9101/metrics | head -30
```

Day 12-14：验证和调试

Step 13：确认 Prometheus 采集到自定义指标

```
# 在 Prometheus UI 中查询
curl -s localhost:9090/api/v1/query?query=vllm_service_up
curl -s localhost:9090/api/v1/query?query=vllm_model_loaded
curl -s localhost:9090/api/v1/query?query=vllm_available_mo
dels_total
curl -s localhost:9090/api/v1/query?query=vllm_health_check
_latency_seconds_count
```

Step 14：用压测验证指标变化

```
# 终端1：跑压测
python3 ~/k8s-llm-inference-platform/scripts/benchmark.py -
-concurrency 10 --requests 100

# 终端2：观察自定义指标
watch -n 5 "curl -s localhost:9101/metrics | grep vllm_"
```

第5周：Grafana Dashboard 打磨 + 告警配置（Day 15-21）

> **目标：** 创建三个专业级 Dashboard，配置至少 5 条告警规则。Dashboard 截图是简历上最直观的展示。

Day 15-17：创建三层 Grafana Dashboard

Dashboard 设计原则：

- 每个 Dashboard 聚焦一个层面，不要把所有指标堆在一起

- 从上到下按重要程度排列：最上面放最核心的指标
- 使用合适的可视化类型：Gauge 用于阈值、Time Series 用于趋势、Stat 用于当前值
- 配色统一，绿色=正常，黄色=警告，红色=危险

Step 15：Dashboard 1 — GPU 总览面板（gpu-overview）

在 Grafana 中新建 Dashboard，添加以下面板：

```
Row 1: 核心状态一览（Stat 面板）
┌──────────────┬──────────────┬──────────────┬──────────────┐
│ GPU 利用率    │  显存使用率    │  GPU 温度     │  当前功耗     │
│  Stat/Gauge  │  Stat/Gauge  │  Stat/Gauge  │  Stat/Gauge  │
│ DCGM_FI_DEV  │ FB_USED/     │ DCGM_FI_DEV  │ DCGM_FI_DEV  │
│ _GPU_UTIL    │ (USED+FREE)  │ _GPU_TEMP    │ _POWER_USAGE │
└──────────────┴──────────────┴──────────────┴──────────────┘

Row 2: GPU 利用率趋势（Time Series 面板）
┌──────────────────────────────────────────────────────────┐
│ GPU Utilization Over Time                                │
│ Query: DCGM_FI_DEV_GPU_UTIL                              │
│ 展示过去 1 小时的 GPU 利用率曲线                            │
└──────────────────────────────────────────────────────────┘

Row 3: 显存使用趋势 + 温度趋势（并排）
┌──────────────────────────┬──────────────────────────┐
│ Memory Usage (MiB)       │ Temperature & Power
```

```
│
│ DCGM_FI_DEV_FB_USED         │ DCGM_FI_DEV_GPU_TEMP
│
│ DCGM_FI_DEV_FB_FREE         │ DCGM_FI_DEV_POWER_USAGE
│
└─────────────────────────────┴──────────────────────────────
┘

Row 4: 硬件详情
┌─────────────────────────────┬──────────────────────────────
┐
│ SM Clock Frequency           │ PCIe Throughput
│
│ DCGM_FI_DEV_SM_CLOCK        │ DCGM_FI_DEV_PCIE_TX/RX
│
└─────────────────────────────┴──────────────────────────────
┘
```

各面板具体 PromQL：

```
# GPU 利用率 (Gauge 面板，阈值：绿<70，黄70-90，红>90)
DCGM_FI_DEV_GPU_UTIL

# 显存使用率百分比 (Gauge 面板)
DCGM_FI_DEV_FB_USED / (DCGM_FI_DEV_FB_USED + DCGM_FI_DEV_FB
_FREE) * 100

# GPU 温度 (Gauge 面板，阈值：绿<70，黄70-80，红>80)
DCGM_FI_DEV_GPU_TEMP

# 功耗 (Stat 面板，单位 W)
DCGM_FI_DEV_POWER_USAGE
```

Step 16：Dashboard 2 — 推理性能面板（inference-perf）

```
Row 1: 核心性能指标一览
┌──────────────────┬──────────────────┬──────────────────┬──────────────
┐
```

```
| 当前 QPS     |  P99 延迟     |  TTFT(P50)   |  Token 吞吐
|
|  Stat        |  Stat        |  Stat        |  Stat
|
└──────────────┴──────────────┴──────────────┴──────────────
┘

Row 2: QPS 和并发趋势
┌──────────────────────────────────────────────────────────────
────┐
| Requests Per Second (QPS)
|
| rate(vllm:e2e_request_latency_seconds_count[1m])
|
└──────────────────────────────────────────────────────────────
────┘

Row 3: 延迟分布
┌──────────────────────────────────────────────────────────────
────┐
| Request Latency Percentiles (P50 / P95 / P99)
|
| histogram_quantile(0.5, rate(vllm:e2e_request_latency_
|
| seconds_bucket[5m]))
|
| histogram_quantile(0.95, ...)
|
| histogram_quantile(0.99, ...)
|
└──────────────────────────────────────────────────────────────
────┘

Row 4: TTFT + Token 生成速度
┌──────────────────────────────┬──────────────────────────────
─┐
| Time To First Token (TTFT)  | Token Generation Rate
|
```

```
| histogram_quantile(0.5,     | rate(vllm:generation_
|
| rate(vllm:time_to_first_    | tokens_total[1m])
|
| token_seconds_bucket[5m]))  |
|
└─────────────────────────────┴──────────────────────────
┘

Row 5: 队列和缓存
┌─────────────────────────────┬──────────────────────────
┐
| Running / Waiting Requests  | KV Cache Usage
|
| vllm:num_requests_running   | vllm:gpu_cache_usage_perc
|
| vllm:num_requests_waiting   |
|
└─────────────────────────────┴──────────────────────────
┘
```

各面板具体 PromQL：

```
# QPS
rate(vllm:e2e_request_latency_seconds_count[1m])

# P50 / P95 / P99 延迟（秒转毫秒，在 Grafana 中设置 Unit 为 ms）
histogram_quantile(0.5, rate(vllm:e2e_request_latency_secon
ds_bucket[5m]))
histogram_quantile(0.95, rate(vllm:e2e_request_latency_seco
nds_bucket[5m]))
histogram_quantile(0.99, rate(vllm:e2e_request_latency_seco
nds_bucket[5m]))

# TTFT P50
histogram_quantile(0.5, rate(vllm:time_to_first_token_secon
ds_bucket[5m]))
```

```
# Token 生成速率（tokens/sec）
rate(vllm:generation_tokens_total[1m])

# 正在处理 vs 等待中
vllm:num_requests_running
vllm:num_requests_waiting

# KV Cache 使用率
vllm:gpu_cache_usage_perc
```

Step 17：Dashboard 3 — 请求分析面板（request-analysis）

```
Row 1: 服务健康度
┌──────────────┬──────────────┬──────────────┬──────────────┐
│ 服务状态      │  模型状态     │  成功率       │  Exporter 延迟│
│ vllm_service │ vllm_model   │ 计算规则      │ vllm_health  │
│ _up          │ _loaded      │              │ _check_lat   │
└──────────────┴──────────────┴──────────────┴──────────────┘

Row 2: 请求成功/失败趋势
┌──────────────────────────────────────────────────────────┐
│ Request Success vs Error Rate                            │
│ rate(vllm_probe_request_success_total[5m])               │
│ rate(vllm_probe_request_error_total[5m])                 │
└──────────────────────────────────────────────────────────┘

Row 3: 抢占和缓存压力
┌──────────────────────────┬──────────────────────────┐
```

```
┐
│ Preemption Events            │ Cache Pressure
│
│ rate(vllm:num_preemptions_   │ vllm:gpu_cache_usage_perc
│
│ total[5m])                   │ vllm:cpu_cache_usage_perc
│
└──────────────────────────────┘──────────────────────────────
┘

Row 4: Token 统计
┌──────────────────────────────┐──────────────────────────────
┐
│ Input Tokens Rate            │ Output Tokens Rate
│
│ rate(vllm:prompt_tokens_     │ rate(vllm:generation_
│
│ total[1m])                   │ tokens_total[1m])
│
└──────────────────────────────┘──────────────────────────────
┘
```

Step 18：导出 Dashboard JSON 到仓库

每个 Dashboard 做好后，通过 Grafana UI 导出：

1. 打开 Dashboard → 右上角齿轮图标（Dashboard Settings）
2. 左侧 **JSON Model** → 复制完整 JSON
3. 保存到仓库：

```
# 分别保存三个 Dashboard
# grafana/dashboards/gpu-overview.json
# grafana/dashboards/inference-perf.json
# grafana/dashboards/request-analysis.json
```

Day 18-21：配置告警规则

Step 19：创建 Prometheus 告警规则

```
cat > ~/llm-observability-stack/prometheus/alert-rules.yml
<< 'EOF'
# Prometheus AlertManager 告警规则
# 覆盖 GPU / 推理服务 / 业务 三层故障场景

groups:
  # ===== GPU 硬件层告警 =====
  - name: gpu-alerts
    rules:
      # 规则1: GPU 温度过高
      - alert: GPUTemperatureHigh
        expr: DCGM_FI_DEV_GPU_TEMP > 80
        for: 5m
        labels:
          severity: warning
        annotations:
          summary: "GPU 温度过高"
          description: "GPU {{ $labels.gpu }} 温度 {{ $valu
e }}°C，已超过 80°C 阈值并持续 5 分钟。可能原因：散热不良、负载过
高。建议检查风扇和降低并发。"

      # 规则2: GPU 温度严重过高
      - alert: GPUTemperatureCritical
        expr: DCGM_FI_DEV_GPU_TEMP > 90
        for: 2m
        labels:
          severity: critical
        annotations:
          summary: "GPU 温度严重过高，有降频/宕机风险"
          description: "GPU {{ $labels.gpu }} 温度 {{ $valu
e }}°C，超过 90°C。GPU 可能触发热保护降频，严重影响推理性能。"

      # 规则3: 显存即将耗尽
      - alert: GPUMemoryAlmostFull
        expr: DCGM_FI_DEV_FB_USED / (DCGM_FI_DEV_FB_USED +
DCGM_FI_DEV_FB_FREE) * 100 > 95
        for: 5m
        labels:
```

```
          severity: warning
        annotations:
          summary: "GPU 显存使用率超过 95%"
          description: "GPU {{ $labels.gpu }} 显存使用 {{ $v
alue | printf \"%.1f\" }}%。可能导致 OOM 或 KV Cache 不足引发请
求抢占。"

      # 规则4: GPU ECC 错误
      - alert: GPUECCError
        expr: DCGM_FI_DEV_ECC_DBE_VOL_TOTAL > 0
        for: 1m
        labels:
          severity: critical
        annotations:
          summary: "GPU 双比特 ECC 错误"
          description: "GPU {{ $labels.gpu }} 检测到不可纠正
的 ECC 错误，硬件可能存在问题，计算结果可能不可靠。建议更换 GPU。"

  # ===== 推理服务层告警 =====
  - name: inference-alerts
    rules:
      # 规则5: vLLM 服务不可用
      - alert: VLLMServiceDown
        expr: vllm_service_up == 0
        for: 1m
        labels:
          severity: critical
        annotations:
          summary: "vLLM 推理服务不可达"
          description: "vLLM 服务在过去 1 分钟内无法连接。请检查
Pod 状态: kubectl -n llm-serving get pod"

      # 规则6: 请求延迟过高
      - alert: HighRequestLatency
        expr: histogram_quantile(0.99, rate(vllm:e2e_reques
t_latency_seconds_bucket[5m])) > 30
        for: 5m
        labels:
```

```
          severity: warning
        annotations:
          summary: "推理请求 P99 延迟超过 30 秒"
          description: "P99 延迟 {{ $value | printf \"%.1f
\" }}s。可能原因：并发过高、max_tokens 过大、GPU 性能不足。"

      # 规则7：请求队列堆积
      - alert: RequestQueueBacklog
        expr: vllm:num_requests_waiting > 10
        for: 3m
        labels:
          severity: warning
        annotations:
          summary: "请求队列堆积"
          description: "当前等待中的请求数 {{ $value }}，已持续
3 分钟。GPU 处理速度跟不上请求到达速度，考虑扩容或限流。"

      # 规则8：KV Cache 使用率过高
      - alert: KVCacheHigh
        expr: vllm:gpu_cache_usage_perc > 0.95
        for: 5m
        labels:
          severity: warning
        annotations:
          summary: "KV Cache 使用率超过 95%"
          description: "KV Cache 使用 {{ $value | printf
\"%.1f\" }}%。可能导致请求抢占和性能下降。考虑减小 max_model_len
或增加 GPU 显存。"

      # 规则9：频繁抢占
      - alert: HighPreemptionRate
        expr: rate(vllm:num_preemptions_total[5m]) > 0.1
        for: 5m
        labels:
          severity: warning
        annotations:
          summary: "KV Cache 抢占频繁"
          description: "过去 5 分钟抢占速率 {{ $value | print
```

```
f \"%.2f\" }}/s。抢占会导致已有请求被回退重算，严重影响用户体验。"

  # ===== 业务层告警 =====
  - name: business-alerts
    rules:
      # 规则10: Pod 反复重启
      - alert: PodRestartLooping
        expr: increase(kube_pod_container_status_restarts_t
otal{namespace="llm-serving"}[30m]) > 3
        for: 5m
        labels:
          severity: critical
        annotations:
          summary: "vLLM Pod 反复重启"
          description: "Pod {{ $labels.pod }} 在过去 30 分钟
内重启了 {{ $value }} 次。请检查: kubectl logs {{ $labels.pod
}} -n llm-serving --previous"
EOF
```

Step 20：将告警规则注入 Prometheus

```
# 方法1: 通过 PrometheusRule CRD（推荐，和 Prometheus Operator
配合）
cat > ~/llm-observability-stack/k8s/prometheus-rules.yaml <
< 'EOF'
apiVersion: monitoring.coreos.com/v1
kind: PrometheusRule
metadata:
  name: llm-alert-rules
  namespace: monitoring
  labels:
    release: monitoring    # 必须有这个标签
    app: kube-prometheus-stack
spec:
  groups:
    # 把上面 alert-rules.yml 的 groups 内容复制到这里
    # （此处省略，实际使用时需要完整复制）
EOF
```

```
kubectl apply -f ~/llm-observability-stack/k8s/prometheus-r
ules.yaml
```

Step 21：配置 Alertmanager 通知路由

```
cat > ~/llm-observability-stack/alertmanager/alertmanager.y
ml << 'EOF'
# Alertmanager 配置
# 告警通知路由

global:
  resolve_timeout: 5m

route:
  group_by: ['alertname', 'severity']
  group_wait: 30s          # 收到第一条告警后等 30 秒再发通知
（等待同类告警聚合）
  group_interval: 5m       # 同一组告警的通知间隔
  repeat_interval: 4h      # 同一条告警重复通知间隔
  receiver: 'default'

  routes:
    - match:
        severity: critical
      receiver: 'critical-alerts'
      continue: true

receivers:
  - name: 'default'
    # 可配置邮件、钉钉、飞书、Slack 等
    # webhook_configs:
    #   - url: 'http://your-webhook-url'
    #     send_resolved: true

  - name: 'critical-alerts'
    # 严重告警单独发送
    # webhook_configs:
```

```
    #   - url: 'http://your-critical-webhook'
    #     send_resolved: true

# 抑制规则：critical 告警触发时，抑制同组的 warning 告警
inhibit_rules:
  - source_match:
      severity: 'critical'
    target_match:
      severity: 'warning'
    equal: ['alertname']
EOF
```

Step 22：验证告警规则

```
# 在 Prometheus UI 中查看告警规则是否加载
curl -s localhost:9090/api/v1/rules | python3 -c "
import sys,json
data=json.load(sys.stdin)
for g in data['data']['groups']:
    if 'gpu' in g['name'] or 'inference' in g['name'] or 'b
usiness' in g['name']:
        print(f\"规则组: {g['name']}\")
        for r in g['rules']:
            print(f\"  {r['name']:30s} 状态: {r.get('healt
h','unknown')}\")
"

# 在 Prometheus UI → Alerts 页面查看所有告警规则状态
# http://<IP>:9090/alerts
```

第6周：OpenTelemetry + 文档 + 上线（Day 22-28）

Day 22-23：OpenTelemetry 接入（可选但加分）

> OpenTelemetry（OTel）主要用于**链路追踪**——跟踪一个请求从进入系统到返回的完整路径和每个阶段的耗时。对于 LLM 推理场景，可以追踪：接收请求 → Tokenize → 调度 → GPU 计算 → Detokenize → 返回。

Step 23：部署 OpenTelemetry Collector

```
cat > ~/llm-observability-stack/otel/otel-collector-config.
yaml << 'EOF'
apiVersion: v1
kind: ConfigMap
metadata:
  name: otel-collector-config
  namespace: monitoring
data:
  config.yaml: |
    receivers:
      otlp:
        protocols:
          grpc:
            endpoint: 0.0.0.0:4317
          http:
            endpoint: 0.0.0.0:4318
      prometheus:
        config:
          scrape_configs:
            - job_name: 'otel-collector'
              scrape_interval: 15s
              static_configs:
                - targets: ['localhost:8888']

    processors:
      batch:
        timeout: 5s
        send_batch_size: 100
      memory_limiter:
        check_interval: 1s
        limit_mib: 256

    exporters:
      prometheus:
        endpoint: "0.0.0.0:8889"
      logging:
```

```
        loglevel: info

    service:
      pipelines:
        metrics:
          receivers: [otlp, prometheus]
          processors: [memory_limiter, batch]
          exporters: [prometheus, logging]
        traces:
          receivers: [otlp]
          processors: [memory_limiter, batch]
          exporters: [logging]
EOF
```

> **实际建议：** 对于项目2的范围，OpenTelemetry 是加分项而非必须。如果时间紧张，可以跳过这一步，在简历里提到"规划了 OpenTelemetry 链路追踪方案"即可。重点精力放在 Dashboard 和告警规则的质量上。

Day 24-26：撰写最佳实践文档

Step 24：撰写「LLM 推理服务监控最佳实践」文档

这份文档是你的**个人品牌内容**，可以发到掘金/知乎引流。

```
cat > ~/llm-observability-stack/docs/best-practices.md <<
'EOF'
# LLM 推理服务监控最佳实践

> 基于 vLLM + NVIDIA A10 + Kubernetes 的生产级可观测性方案

## 一、为什么 LLM 推理服务需要专门的监控?

LLM 推理和传统 Web 服务有三个根本性差异，决定了不能沿用传统的监控方案:

1. **资源消耗模式不同:** 传统服务主要消耗 CPU 和内存，LLM 推理主要消耗 GPU 显存和算力
2. **延迟量级不同:** 传统 API 延迟在毫秒级，LLM 推理延迟在秒级到十秒级
3. **新的性能指标:** TTFT、Token 吞吐量、KV Cache 使用率等指标在
```

传统监控中不存在

二、三层监控架构

第一层：GPU 硬件监控
为什么需要： GPU 是最贵的资源，也是最容易成为瓶颈的环节。
采集方案： DCGM Exporter (DaemonSet) → Prometheus
核心指标： GPU 利用率、显存使用率、温度、功耗、ECC 错误

第二层：推理引擎监控
为什么需要： 理解推理引擎内部状态，识别性能瓶颈。
采集方案： vLLM 原生 /metrics + 自定义 Exporter
核心指标： QPS、延迟分位数、TTFT、KV Cache 使用率、抢占次数

第三层：业务质量监控
为什么需要： 最终用户只关心"能不能用、快不快、对不对"。
采集方案： 自定义 Exporter + Prometheus 告警规则
核心指标： 成功率、错误率、超时率、服务可用性

三、告警阈值设计

GPU 温度告警
- Warning: >80°C 持续 5 分钟
- Critical: >90°C 持续 2 分钟
- **原理：** NVIDIA GPU 在 83°C 开始降频（因GPU型号而异），90°C+ 可能触发热保护关机

显存使用率告警
- Warning: >95% 持续 5 分钟
- **原理：** 显存耗尽会导致 OOM，vLLM 会抢占已有请求的 KV Cache

请求延迟告警
- Warning: P99 >30s 持续 5 分钟
- **原理：** 结合业务 SLA 设定，30s 超时对大多数在线应用不可接受

KV Cache 告警
- Warning: >95% 持续 5 分钟
- **原理：** KV Cache 满了会触发抢占，已有请求被回退重算

```
## 四、常见故障模式与监控策略

| 故障模式 | 监控信号 | 根因 | 解决方案 |
|---------|---------|------|---------|
| GPU OOM | 显存使用率飙到100%，Pod OOMKilled | max_model_len 或并发过高 | 减小 max_model_len，降低 gpu-memory-utilization |
| KV Cache 抢占 | preemptions_total 飙升，延迟不稳定 | 并发请求过多，Cache 不够分 | 减少并发、缩短 max_tokens、增加 GPU |
| 模型加载失败 | model_loaded=0，Pod CrashLoopBackOff | 显存不够加载模型、模型文件损坏 | 检查显存、重新下载模型 |
| GPU 降频 | SM Clock 下降，GPU Temp >83°C | 散热不良 | 检查风扇、降低功耗限制 |
| 请求堆积 | waiting_requests 持续增长 | 到达速率 > 处理速率 | 扩容、限流、异步处理 |
| 单请求超时 | 个别请求延迟远高于平均 | 输入过长、生成内容过长 | 限制 max_tokens、设置请求超时 |

## 五、Grafana Dashboard 设计原则

1. **分层设计：** GPU / 推理引擎 / 业务 各一个 Dashboard
2. **黄金信号优先：** 每个 Dashboard 第一行放 延迟、吞吐量、错误率、饱和度
3. **阈值可视化：** Gauge 面板设置红黄绿阈值，一眼看出是否正常
4. **关联性：** 相关指标放在同一行，方便关联分析
5. **时间对齐：** 所有面板使用相同的时间范围，方便对比

## 六、生产环境建议

1. Prometheus 数据至少保留 15 天，关键指标保留 90 天
2. Grafana Dashboard 通过 JSON 文件版本化管理，存入 Git
3. 告警规则避免过于敏感（for 持续时间不要太短）
4. 建立 Runbook：每条告警对应一个处理手册
5. 定期进行故障演练，验证告警是否能准确触发
EOF
```

Day 27-28：GitHub 上线 + 最终检查

Step 25：整理仓库，提交代码

```
cd ~/llm-observability-stack

# 写 README
cat > README.md << 'EOF'
# 🔍 LLM Observability Stack

LLM 推理服务全链路可观测性方案：GPU 监控 + 推理性能分析 + 智能告警。

## 架构图

（贴你在项目1中画的监控相关的架构图）

## 核心特性

- 三层监控覆盖：GPU 硬件 → 推理引擎 → 业务质量
- 自定义 Prometheus Exporter 采集 vLLM 业务指标
- 3 个专业级 Grafana Dashboard（GPU 总览 / 推理性能 / 请求分析）
- 10 条告警规则覆盖常见故障场景
- OpenTelemetry 链路追踪（可选）
- 最佳实践文档

## 监控覆盖

| 层面 | 指标数量 | 采集方式 |
|------|---------|---------|
| GPU 硬件 | 11 项 | DCGM Exporter |
| 推理引擎 | 10 项 | vLLM 原生 /metrics |
| 业务质量 | 6 项 | 自定义 Exporter |

## 告警规则

| 场景 | 级别 | 阈值 |
|------|------|------|
| GPU 温度过高 | Warning / Critical | 80°C / 90°C |
| 显存耗尽 | Warning | >95% |
| 服务不可达 | Critical | down >1min |
| P99 延迟过高 | Warning | >30s |
| 请求队列堆积 | Warning | >10 |
```

```
| KV Cache 过高 | Warning | >95% |
| 频繁抢占 | Warning | >0.1/s |
| Pod 反复重启 | Critical | >3次/30min |

## 快速开始

（部署步骤）

## Dashboard 截图

（贴三个 Dashboard 的截图）

## 技术栈

- Prometheus + Grafana + Alertmanager
- DCGM Exporter (NVIDIA GPU 监控)
- 自定义 Python Prometheus Exporter
- OpenTelemetry Collector（可选）
- Kubernetes (K3s)

## 详细文档

- [监控指标目录](docs/metrics-catalog.md)
- [最佳实践文档](docs/best-practices.md)
- [架构设计](docs/architecture.md)
EOF
```

Step 26：提交并推送

```
cd ~/llm-observability-stack
git add -A
git commit -m "feat: LLM observability stack v1.0"
git remote add origin git@github.com:<你的用户名>/llm-observ
ability-stack.git
git push -u origin main
```

Step 27：发布最佳实践文档到技术社区

把 `docs/best-practices.md` 的内容发到：

- **掘金：** juejin.cn（国内前端/运维社区活跃）
- **知乎：** 作为技术文章发布
- **CSDN：** 国内搜索引擎收录好

文章标题建议：「从零搭建 LLM 推理服务监控体系：GPU + vLLM + Prometheus 全链路实践」

过关验收 Checklist

基础搭建

☐ Prometheus + Grafana + DCGM Exporter 全部正常运行

☐ 所有 ServiceMonitor 配置正确，targets 全部 up

自定义 Exporter

☐ vllm-exporter Pod 正常运行

☐ 自定义指标（vllm_service_up、vllm_model_loaded 等）可在 Prometheus 中查询

☐ Exporter 代码有注释、有错误处理

Grafana Dashboard

☐ Dashboard 1: GPU 总览（利用率、显存、温度、功耗）

☐ Dashboard 2: 推理性能（QPS、延迟分位数、TTFT、Token 吞吐、KV Cache）

☐ Dashboard 3: 请求分析（服务健康、成功/失败率、抢占、Token 统计）

☐ 面板配色统一，阈值清晰（红黄绿）

☐ 截图效果专业，可直接放简历

告警规则

☐ 至少 5 条告警规则（实际配了 10 条）

☐ 覆盖 GPU / 推理服务 / 业务三层

☐ 告警规则可在 Prometheus → Alerts 页面查看

文档与仓库

- [ ] README 专业完整，有架构图和 Dashboard 截图
- [ ] 监控指标目录文档完成
- [ ] 最佳实践文档完成并发布到技术社区
- [ ] GitHub 仓库上线，标签和描述齐全

简历素材

- [ ] 能用一句话描述项目价值
- [ ] 能说清楚三层监控架构
- [ ] 能回答"告警阈值怎么定的"
- [ ] 能回答"遇到 GPU OOM 怎么排查"
- [ ] Dashboard 截图已准备好放简历

面试问答准备

Q: 你的监控方案覆盖了哪些层面?

> 三层：GPU 硬件层（DCGM Exporter 采集利用率、显存、温度等 11 项指标）、推理引擎层（vLLM 原生暴露 QPS、延迟、TTFT、KV Cache 等 10 项指标）、业务质量层（自定义 Exporter 补充采集服务健康度、成功率等 6 项指标）。

Q: 为什么要写自定义 Exporter？vLLM 自带的 /metrics 不够吗?

> vLLM 的 /metrics 主要暴露引擎内部指标，但缺少业务层面的信息。比如服务是否可达（vllm_service_up）、模型是否加载完成（vllm_model_loaded）、健康检查延迟等。这些信息对告警和业务监控很关键，所以需要自定义 Exporter 补充。

Q: 告警阈值是怎么确定的?

> 基于 GPU 硬件规格和实际压测数据。比如 GPU 温度告警设 80°C 是因为 NVIDIA GPU 在 83°C 左右开始降频，需要提前预警。显存 95% 是因为留出 5% 缓冲防止 OOM。延迟 30s 阈值是结合业务 SLA 和压测基线数据确定的。

Q: 如果 GPU 利用率一直是 100%，说明什么?

> 不一定是问题。要结合其他指标看：如果队列没有堆积（waiting=0）、延迟在 SLA 内、温度正常，那 100% 反而说明 GPU 利用率很高、没有浪费。但如果队列在增长、延迟飙升，就说明 GPU 已经是瓶颈，需要扩容。

Q: KV Cache 使用率和显存使用率是什么关系？

> 显存被两部分占用：模型权重（固定不变）和 KV Cache（动态分配）。vLLM 的 gpu-memory-utilization 参数决定了预留给 KV Cache 的显存比例。KV Cache 使用率是指在这个预留空间中实际用了多少。显存使用率看的是整张卡的总用量。