On-Premise LLM Deployment on Physical Machines - Complete Guide

Table of Contents

Introduction

Why On-Premise Physical Deployment?

```
Benefits:
✓ Complete data control (no external API calls)
✓ Regulatory compliance (HIPAA, SOC2, GDPR)
✓ Cost predictability (no surprise bills)
✓ Best performance (no network latency to cloud)
✓ Customization (full stack control)
✓ Long-term economics (break-even in 18-24 months)

Trade-offs:
✗ Higher initial capex ($150k-$500k)
✗ Operational burden (on-call, maintenance)
✗ Scaling requires new hardware
✗ Compliance responsibility (audits, security)
```

Deployment Models Covered

```
1. Bare Metal
   - Direct hardware, no virtualization
   - Best performance, highest complexity
   - For: High-throughput production

2. Hypervisor-Based (VMware, Hyper-V, KVM)
   - Virtual machines on physical hardware
```

```
   - Balance of flexibility and performance
   - For: Multi-tenant environments

3. Container-Based (Docker + Kubernetes)
   - Containerized workloads on VMs/bare metal
   - Maximum flexibility and portability
   - For: Scalable deployments
```

Hardware Selection & Setup

Step 1: Choose Your Physical Hardware

CPU Selection

```
For LLM Inference (NOT Training):

High-Performance CPUs:
- Intel Xeon Platinum 8490H
  - 60 cores, 2.4 GHz
  - Cost: $12,000 per socket
  - Good for: Token generation offload
  - Typical config: 2 sockets = 120 cores

- AMD EPYC 9684X
  - 96 cores, 2.5 GHz
  - Cost: $13,000 per socket
  - Better: More cores for same price
  - Typical config: 2 sockets = 192 cores

Balanced CPUs (Cost-Effective):
- Intel Xeon Gold 6348
  - 28 cores, 3.5 GHz
  - Cost: $3,000 per socket
  - Good for: Smaller deployments
  - Memory bandwidth: Adequate

Recommendation for LLM Deployment:
✓ Use 2x AMD EPYC 9684X (192 cores total)
✓ OR 2x Intel Platinum 8490H (120 cores total)
✓ Focus on: Memory bandwidth > core count
```

GPU Selection (Critical)

```
Recommended GPU Options:

NVIDIA A100 80GB (Premium)
```

```
├─ Specs: 80GB HBM2, 312 TFLOPS
├─ Cost: $10,000-15,000 per GPU
├─ Best for: 70B+ models
├─ Performance: ~40 tokens/sec (7B model)
└─ ROI: 24+ months

NVIDIA L40S 48GB (Balanced)
├─ Specs: 48GB GDDR6, 362 TFLOPS
├─ Cost: $6,000-8,000 per GPU
├─ Best for: 7B-30B models
├─ Performance: ~25 tokens/sec (7B model)
└─ ROI: 18-20 months

NVIDIA H100 80GB (Latest)
├─ Specs: 80GB HBM3, 1,456 TFLOPS (FP8)
├─ Cost: $15,000-18,000 per GPU
├─ Best for: Large scale deployments
├─ Performance: ~80 tokens/sec (7B model)
└─ ROI: 12-18 months

AMD MI300X 192GB (Alternative)
├─ Specs: 192GB HBM3, 1.5 PETAFLOPS
├─ Cost: $10,000 per GPU
├─ Best for: Very large models
├─ Challenge: Lower software support
└─ ROI: 15-18 months

Recommended Configuration:
- For 1M inferences/month: 4x A100 or 4x L40S
- For 5M inferences/month: 8x A100 or 16x L40S
- For 10M+ inferences/month: Mixed (A100 + H100)
```

Memory (RAM) Configuration

```
Rule of Thumb:
Minimum: 10x GPU memory
Recommended: 15-20x GPU memory

Examples:

4x NVIDIA A100 (80GB each = 320GB VRAM):
├─ Minimum RAM: 3.2TB
├─ Recommended: 5-6TB (512GB per node × 10-12 nodes)
├─ Or: 2x nodes with 3TB each
└─ Cost: $30,000-50,000

4x NVIDIA L40S (48GB each = 192GB VRAM):
├─ Minimum RAM: 2TB
├─ Recommended: 3-4TB
├─ Or: 2x nodes with 2TB each
```

```
└─ Cost: $20,000-35,000

Memory Type:
- DDR5 RDIMM preferred (best latency)
- ECC required (data integrity)
- Speed: 5600MHz or faster
```

Storage Configuration

```
Tier 1: NVMe for Active Models
├─ Capacity: 500GB-1TB per node
├─ Type: PCIe 4.0 or 5.0 NVMe
├─ RAID: RAID 1 (mirrored)
├─ Purpose: Currently serving models
└─ Cost: $100-300 per TB

Tier 2: SAS SSD for Secondary Storage
├─ Capacity: 10TB-50TB shared storage
├─ Type: 15K RPM SAS drives
├─ RAID: RAID 6 (fault tolerance)
├─ Purpose: Model library, backups
└─ Cost: $30-50 per TB

Tier 3: Archive (Optional)
├─ Capacity: 100TB+
├─ Type: NAS or tape
├─ Purpose: Long-term backups
└─ Cost: $10-20 per TB

Recommended Setup:
- 4 nodes × 1TB NVMe (RAID 1):        $4,000
- Shared SAS SSD storage (50TB RAID 6): $15,000
- Network attached storage (NAS):       $10,000
──────────────────────────────────────
Total Storage Investment:              $29,000
```

Network Interface Cards (NICs)

```
Requirements for LLM Serving:

Inter-Node Communication:
├─ Speed: 100Gbps (RoCE v2) or 200Gbps
├─ Latency: <1 microsecond
├─ Option 1: NVIDIA ConnectX-7 (400Gbps capable)
│            - Cost: $3,000-5,000
│            - Best performance
├─ Option 2: Mellanox HDR (200Gbps)
```

```
│           - Cost: $2,000-3,000
│           - Good cost-performance
└─ Option 3: Ethernet (100Gbps)
            - Cost: $500-1,000
            - Adequate for most deployments

External API Traffic:
├─ Speed: 25Gbps per node minimum
├─ Redundancy: Dual 25Gbps NICs
├─ Option: Dual Broadcom BCM57504
└─ Cost: $1,000-2,000

Recommended:
- Intel Xeon nodes: Dual 10GbE + 100GbE (RDMA)
- For: $3,500 per node in NICs
```

Step 2: Chassis & Power Configuration

Server Chassis Selection

```
2U Dual-Socket Configuration (Recommended):

SuperMicro SYS-2124US-TNRT
├─ Form Factor: 2U, 4x GPU slots
├─ CPU Support: Dual Xeon/EPYC
├─ GPU Capacity: 4x A100 (or 8x L40S)
├─ Cooling: Front to back airflow
├─ Cost per chassis: $8,000
└─ Total (4 chassis): $32,000

Dell PowerEdge R6715
├─ Form Factor: 2U, 12x SFF drive bays
├─ CPU Support: Dual EPYC
├─ GPU Capacity: 4x GPUs (via PCIe)
├─ Cooling: Modular, expandable
├─ Cost per chassis: $7,500
└─ Total (4 chassis): $30,000

Recommendation:
Choose based on:
✓ GPU cooling requirements
✓ Future expandability
✓ Your vendor support contracts
✓ Spare parts availability
```

Power Supply & Distribution

```
Power Budget Calculation:

Per Chassis with 4x A100:
├─ CPUs (2x Xeon Platinum): 500W
├─ 4x A100 GPUs: 320W (80W each)
├─ Memory/Storage/Network: 200W
├─ Cooling overhead: 150W
├─ Total: ~1,170W per chassis
└─ Recommended PSU: 2x 1200W (redundant)

For 4-Node Cluster:
├─ Total power: 4 × 1,170W = 4,680W
├─ With cooling: ~6,000-7,000W
├─ Recommended capacity: 8-10kW
├─ Plus: 20% overhead for peaks

Power Distribution Unit (PDU):
├─ Capacity: 30A @ 208V (6.2kW) or higher
├─ Type: Managed PDU (for monitoring)
├─ Features: Remote power-on, outlet metering
├─ Cost: $1,500-3,000

Uninterruptible Power Supply (UPS):
├─ Capacity: 20-30 minute runtime (at full load)
├─ For 8kW system: ~20kVA UPS
├─ Battery cost: $30,000-50,000
├─ Plus: Runtime extension modules
└─ Purpose: Graceful shutdown, no data loss

Electrical Infrastructure:
├─ Dedicated 240V circuit (3-phase preferred)
├─ Capacity: 60A+ per node area
├─ Grounding: Proper earth/neutral separation
├─ Cost: $5,000-15,000 (installation dependent)
```

Cooling System

```
Cooling Capacity Requirements:

Heat Dissipation (4-node cluster):
├─ Compute heat: 6-7kW
├─ Overhead (PDU, networking): 1-2kW
├─ Total: ~8-9kW

CRAC Unit (Computer Room Air Conditioning):
├─ Capacity needed: 30-35 tons of cooling
├─ Installation: Floor-mounted or drop-ceiling
├─ Cost: $20,000-30,000 + installation
├─ Pros: Precise temperature control
```

```
└─ Cons: High power consumption

In-Row Cooling:
├─ Capacity: Modular, 5-20 tons
├─ Placement: Between or above racks
├─ Cost: $10,000-15,000 per unit
├─ Pros: More efficient, easier to expand
└─ Cons: Less centralized control

Passive + Spot Cooling:
├─ Fans + open windows/HVAC
├─ Cost: $2,000-5,000
├─ Pros: Low cost, simple
└─ Cons: Less reliable for 24/7 operation

Temperature Targets:
├─ Inlet: 18-27°C (optimal: 22-24°C)
├─ Outlet: <35°C
├─ GPU: <80°C under load
├─ Monitor: Install temperature sensors in nodes
```

Step 3: Pre-Deployment Checklist

```
Physical Setup:
☑ All hardware received and verified
☑ Serial numbers recorded and tagged
☑ Physical security (locks, cables)
☑ Rack mounted securely (earthquake straps for critical regions)
☑ All cables labeled (network, power, serial)

Electrical:
☑ Power circuits tested and verified
☑ UPS tested (simulate power loss)
☑ PDU configured and monitored
☑ Circuit breakers sized correctly
☑ Grounding verified

Network:
☑ Network cables tested for continuity
☑ Switch configuration ready
☑ Firewall rules prepared
☑ VLAN configuration planned
☑ DNS/DHCP ready

Environmental:
☑ Cooling system operational
☑ Temperature sensors installed
☑ Humidity monitoring (40-60% optimal)
☑ Air flow pattern verified
```

```
Documentation:
☑ Hardware inventory spreadsheet
☑ Network diagram created
☑ Power consumption baseline
☑ Serial number tracking
☑ Emergency contact list posted
```

Bare Metal Deployment

Step 1: Operating System Installation

Ubuntu 22.04 LTS (Recommended)

```
# This assumes you have USB boot drive with Ubuntu ISO

# 1. Boot from installation media
# 2. Select "Ubuntu Server (minimized)" for stability
# 3. Configure:

# Hostname: llm-inference-node-1
# Network: Static IP (see step 2)
# Disk: RAID 1 on system drives
# Partitioning:
#   - /boot/efi: 1GB (EFI partition)
#   - /: 500GB (root filesystem, ext4)
#   - Remaining: Leave for models

# 4. After installation, SSH into the server
ssh ubuntu@llm-inference-node-1

# 5. Update system
sudo apt update
sudo apt upgrade -y
sudo apt install -y \
    build-essential \
    linux-headers-$(uname -r) \
    git \
    curl \
    wget \
    vim \
    htop \
    nvtop \
    tmux

# 6. Configure system limits (for GPU/memory)
sudo tee -a /etc/security/limits.conf <<EOF
*       soft    nofile          1000000
*       hard    nofile          1000000
*       soft    memlock         unlimited
```

```
*       hard    memlock         unlimited
root    soft    nofile          1000000
root    hard    nofile          1000000
EOF

# 7. Enable persistent hugepages (for GPU memory)
sudo tee -a /etc/default/grub <<EOF
GRUB_CMDLINE_LINUX="hugepagesz=1G hugepages=256"
EOF

sudo update-grub
sudo reboot

# 8. After reboot, verify settings
ulimit -n
grep Huge /proc/meminfo
```

CentOS/RHEL Alternative

```
# If you prefer CentOS/RHEL (common in enterprises)

sudo yum groupinstall -y "Development Tools"
sudo yum install -y \
    kernel-devel-$(uname -r) \
    kernel-headers-$(uname -r) \
    epel-release \
    git \
    curl \
    wget \
    vim \
    htop

# Similar limit configurations
sudo bash -c 'cat >> /etc/security/limits.conf' <<EOF
*       soft    nofile          1000000
*       hard    nofile          1000000
*       soft    memlock         unlimited
*       hard    memlock         unlimited
EOF

# Generate SSH key for later
ssh-keygen -t ed25519 -f ~/.ssh/id_ed25519 -N ""
```

Step 2: Network Configuration

```
# View current network config
ip addr show
```

```
# Create static network configuration
# File: /etc/netplan/00-installer-config.yaml

sudo tee /etc/netplan/00-installer-config.yaml <<'EOF'
network:
  version: 2
  renderer: networkd
  ethernets:
    eth0:
      dhcp4: no
      addresses:
        - 192.168.1.101/24  # Change to your network
      gateway4: 192.168.1.1
      nameservers:
        addresses: [8.8.8.8, 8.8.4.4]
    eth1:
      dhcp4: no
      addresses:
        - 10.0.0.101/24     # Private GPU cluster network
      mtu: 9000             # Jumbo frames for GPU-to-GPU
EOF

# Apply configuration
sudo netplan apply

# Verify
ip addr show
ip route show
```

Step 3: NVIDIA GPU Driver Installation

```
# Step 1: Verify GPU detection
lspci | grep NVIDIA

# Expected output (for A100):
# 01:00.0 3D controller: NVIDIA Corporation GA100 [A100 PCI-e] (rev a1)

# Step 2: Download NVIDIA driver
# Go to: https://www.nvidia.com/Download/driverDetails.aspx
# Choose: Data Center GPU > A100 > Linux x64

# For this example (adjust version as needed):
wget https://us.download.nvidia.com/tesla/535.104.05/NVIDIA-Linux-x86_64-
535.104.05.run

# Make executable
chmod +x NVIDIA-Linux-x86_64-535.104.05.run

# Step 3: Install driver
# Option A: Automated (recommended)
```

```
sudo ./NVIDIA-Linux-x86_64-535.104.05.run --silent --driver

# Option B: Interactive
sudo ./NVIDIA-Linux-x86_64-535.104.05.run

# When prompted:
# - Accept EULA
# - Install NVIDIA's 32-bit compatibility libraries: Yes
# - Disable Nouveau: Yes (if asked)
# - Run nvidia-xconfig: No (not needed for headless)

# Step 4: Verify installation
nvidia-smi

# Expected output:
# +-----------------------------------------------------------------------------
-+
# | NVIDIA-SMI 535.104.05    Driver Version: 535.104.05
|
# |-------------------------------+----------------------+----------------------
-+
# | GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC
|
# | 0  NVIDIA A100 40GB      On   | 00:1E.0        Off  |                    0
|
# +-------------------------------+----------------------+----------------------
-+

# Step 5: Check GPU memory
nvidia-smi --query-gpu=memory.total --format=csv,noheader

# Expected: 81920 MiB (for A100 80GB)

# Step 6: Performance metrics
nvidia-smi -q -d Utilization
nvidia-smi dmon  # Real-time GPU monitoring

# Step 7: Enable persistence mode (keeps driver loaded)
sudo nvidia-smi -pm 1

# Step 8: Set power limit
# For A100 (optional, reduces power consumption)
sudo nvidia-smi -pm 1  # Persistence mode
sudo nvidia-smi -pl 250  # Power limit to 250W (from 400W max)
```

Step 4: CUDA Toolkit Installation

```
# Step 1: Download CUDA 12.2 (compatible with recent drivers)
wget
https://developer.download.nvidia.com/compute/cuda/12.2.0/local_installers/cuda
```

```
_12.2.0_535.104.05_linux.run

# Step 2: Install CUDA
sudo sh cuda_12.2.0_535.104.05_linux.run --silent --toolkit --
installpath=/usr/local/cuda

# Step 3: Set environment variables
# Add to ~/.bashrc
tee -a ~/.bashrc <<'EOF'
export PATH=/usr/local/cuda/bin:$PATH
export LD_LIBRARY_PATH=/usr/local/cuda/lib64:$LD_LIBRARY_PATH
export CUDA_HOME=/usr/local/cuda
EOF

source ~/.bashrc

# Step 4: Verify CUDA
nvcc --version
# Expected: cuda_12.2.0

# Step 5: Test CUDA
cuda-samples/bin/x86_64/linux/release/nbody -benchmark -numbodies=256000
```

Step 5: Python Environment Setup

```
# Step 1: Install Python 3.11 (preferred for AI workloads)
sudo apt install -y python3.11 python3.11-venv python3.11-dev

# Step 2: Create virtual environment for LLM serving
python3.11 -m venv ~/llm-serving
source ~/llm-serving/bin/activate

# Step 3: Upgrade pip
pip install --upgrade pip setuptools wheel

# Step 4: Install PyTorch (with CUDA support)
pip install torch torchvision torchaudio --index-url
https://download.pytorch.org/whl/cu121

# Verify PyTorch
python -c "import torch; print(f'GPU available: {torch.cuda.is_available()}')"
python -c "import torch; print(f'GPU count: {torch.cuda.device_count()}')"

# Step 5: Install vLLM (for fast LLM serving)
pip install vllm

# Step 6: Install additional tools
pip install \
    flask \
    fastapi \
```

```
    uvicorn \
    pydantic \
    requests \
    python-dotenv \
    prometheus-client

# Step 7: Verify installation
python -c "from vllm import LLM; print('vLLM installed successfully')"
```

Step 6: Bare Metal vLLM Service Setup

```
# Step 1: Create service directory
mkdir -p ~/llm-service
cd ~/llm-service

# Step 2: Create systemd service file
sudo tee /etc/systemd/system/vllm-inference.service <<'EOF'
[Unit]
Description=vLLM Inference Server
After=network.target
Wants=network-online.target

[Service]
Type=simple
User=ubuntu
WorkingDirectory=/home/ubuntu/llm-service
Environment="PATH=/home/ubuntu/llm-serving/bin"
Environment="CUDA_VISIBLE_DEVICES=0,1,2,3"
Environment="VLLM_PORT=8000"
Environment="MODEL_NAME=meta-llama/Llama-2-70b-chat-hf"

# GPU memory utilization
Environment="GPU_MEMORY_UTILIZATION=0.9"

# Tensor parallelism (spread across 4 GPUs)
Environment="TENSOR_PARALLEL_SIZE=4"

# Startup command
ExecStart=/home/ubuntu/llm-serving/bin/python -m
vllm.entrypoints.openai.api_server \
    --model ${MODEL_NAME} \
    --tensor-parallel-size ${TENSOR_PARALLEL_SIZE} \
    --gpu-memory-utilization ${GPU_MEMORY_UTILIZATION} \
    --port ${VLLM_PORT} \
    --max-model-len 4096 \
    --enable-prefix-caching

# Auto-restart on failure
Restart=on-failure
RestartSec=5
```

```
# Logging
StandardOutput=journal
StandardError=journal
SyslogIdentifier=vllm

[Install]
WantedBy=multi-user.target
EOF

# Step 3: Reload systemd
sudo systemctl daemon-reload

# Step 4: Start service
sudo systemctl start vllm-inference

# Step 5: Check status
sudo systemctl status vllm-inference
journalctl -u vllm-inference -f

# Step 6: Enable on boot
sudo systemctl enable vllm-inference

# Step 7: Test the inference server
curl http://localhost:8000/v1/completions \
  -H "Content-Type: application/json" \
  -d '{
    "model": "meta-llama/Llama-2-70b-chat-hf",
    "prompt": "Explain quantum computing in one sentence.",
    "max_tokens": 100,
    "temperature": 0.7
  }'
```

Step 7: Create Flask API Gateway

```
# File: ~/llm-service/app.py

cat > ~/llm-service/app.py <<'EOF'
#!/usr/bin/env python3

from flask import Flask, request, jsonify
import requests
import json
import logging
from datetime import datetime
from prometheus_client import Counter, Histogram, generate_latest
import time

app = Flask(__name__)
```

```
# Configure logging
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s'
)
logger = logging.getLogger(__name__)

# vLLM endpoint (local)
VLLM_ENDPOINT = "http://localhost:8000"

# Metrics
request_count = Counter(
    'llm_requests_total',
    'Total LLM requests',
    ['model', 'status']
)

request_duration = Histogram(
    'llm_request_duration_seconds',
    'LLM request duration',
    ['model']
)

token_counter = Counter(
    'llm_tokens_generated_total',
    'Total tokens generated',
    ['model']
)

@app.route('/v1/completions', methods=['POST'])
def completions():
    """OpenAI-compatible completions endpoint"""

    start_time = time.time()

    try:
        data = request.json
        model = data.get('model', 'meta-llama/Llama-2-70b-chat-hf')
        prompt = data.get('prompt')

        if not prompt:
            return {"error": "prompt is required"}, 400

        logger.info(f"Completion request - Model: {model}, Prompt length:
{len(prompt)}")

        # Call vLLM
        response = requests.post(
            f"{VLLM_ENDPOINT}/v1/completions",
            json=data,
            timeout=300  # 5 minute timeout for large models
        )
```

```
        if response.status_code != 200:
            logger.error(f"vLLM error: {response.status_code} -
{response.text}")
            request_count.labels(model=model, status='error').inc()
            return {"error": "Generation failed"}, 500

        result = response.json()

        # Track metrics
        tokens_generated = result['usage']['completion_tokens']
        token_counter.labels(model=model).inc(tokens_generated)
        request_count.labels(model=model, status='success').inc()

        duration = time.time() - start_time
        request_duration.labels(model=model).observe(duration)

        logger.info(f"Completion success - Tokens: {tokens_generated},
Duration: {duration:.2f}s")

        return result

    except Exception as e:
        logger.error(f"Error: {str(e)}")
        request_count.labels(model='unknown', status='error').inc()
        return {"error": str(e)}, 500

@app.route('/v1/chat/completions', methods=['POST'])
def chat_completions():
    """OpenAI-compatible chat completions endpoint"""

    start_time = time.time()

    try:
        data = request.json
        model = data.get('model', 'meta-llama/Llama-2-70b-chat-hf')
        messages = data.get('messages', [])

        if not messages:
            return {"error": "messages are required"}, 400

        logger.info(f"Chat request - Model: {model}, Messages:
{len(messages)}")

        # Convert to prompt format
        prompt = "\n".join([f"{msg['role']}: {msg['content']}" for msg in
messages])

        # Call vLLM
        response = requests.post(
            f"{VLLM_ENDPOINT}/v1/completions",
            json={
                "model": model,
                "prompt": prompt,
```

```
                "max_tokens": data.get('max_tokens', 256),
                "temperature": data.get('temperature', 0.7),
                "top_p": data.get('top_p', 0.95)
            },
            timeout=300
        )

        if response.status_code != 200:
            logger.error(f"vLLM error: {response.status_code}")
            request_count.labels(model=model, status='error').inc()
            return {"error": "Generation failed"}, 500

        result = response.json()
        tokens_generated = result['usage']['completion_tokens']

        token_counter.labels(model=model).inc(tokens_generated)
        request_count.labels(model=model, status='success').inc()

        duration = time.time() - start_time
        request_duration.labels(model=model).observe(duration)

        return result

    except Exception as e:
        logger.error(f"Error: {str(e)}")
        request_count.labels(model='unknown', status='error').inc()
        return {"error": str(e)}, 500

@app.route('/health', methods=['GET'])
def health():
    """Health check endpoint"""
    try:
        response = requests.get(f"{VLLM_ENDPOINT}/health", timeout=5)
        is_healthy = response.status_code == 200

        return {
            "status": "healthy" if is_healthy else "degraded",
            "vllm": is_healthy,
            "timestamp": datetime.now().isoformat()
        }
    except Exception as e:
        logger.error(f"Health check failed: {str(e)}")
        return {"status": "unhealthy", "error": str(e)}, 500

@app.route('/metrics', methods=['GET'])
def metrics():
    """Prometheus metrics endpoint"""
    return generate_latest()

@app.route('/models', methods=['GET'])
def list_models():
    """List available models"""
    try:
```

```
        response = requests.get(f"{VLLM_ENDPOINT}/v1/models")
        return response.json()
    except Exception as e:
        return {"error": str(e)}, 500

if __name__ == '__main__':
    # Run on all interfaces
    app.run(host='0.0.0.0', port=5000, debug=False, threaded=True)
EOF

# Make executable
chmod +x ~/llm-service/app.py

# Create systemd service for Flask app
sudo tee /etc/systemd/system/llm-api.service <<'EOF'
[Unit]
Description=LLM API Gateway
After=vllm-inference.service
Requires=vllm-inference.service

[Service]
Type=simple
User=ubuntu
WorkingDirectory=/home/ubuntu/llm-service
Environment="PATH=/home/ubuntu/llm-serving/bin"
ExecStart=/home/ubuntu/llm-serving/bin/python /home/ubuntu/llm-service/app.py
Restart=on-failure
RestartSec=5
StandardOutput=journal
StandardError=journal

[Install]
WantedBy=multi-user.target
EOF

# Enable and start
sudo systemctl daemon-reload
sudo systemctl enable llm-api
sudo systemctl start llm-api
sudo systemctl status llm-api
```

Step 8: Testing Bare Metal Deployment

```
# Test 1: Check vLLM is running
curl http://localhost:8000/health

# Test 2: Check API Gateway is running
curl http://localhost:5000/health

# Test 3: Simple completion
```

```
curl http://localhost:5000/v1/completions \
  -H "Content-Type: application/json" \
  -d '{
    "model": "meta-llama/Llama-2-70b-chat-hf",
    "prompt": "What is AI?",
    "max_tokens": 50
  }' | jq

# Test 4: Chat completion
curl http://localhost:5000/v1/chat/completions \
  -H "Content-Type: application/json" \
  -d '{
    "model": "meta-llama/Llama-2-70b-chat-hf",
    "messages": [{"role": "user", "content": "Hello, how are you?"}],
    "max_tokens": 50
  }' | jq

# Test 5: Metrics
curl http://localhost:5000/metrics | grep llm_

# Test 6: Load test (requires Apache Bench)
sudo apt install -y apache2-utils
ab -n 100 -c 10 \
  -H "Content-Type: application/json" \
  -p request.json \
  http://localhost:5000/v1/completions
```

Hypervisor-Based Deployment

Option A: VMware ESXi

Step 1: ESXi Installation

```
# 1. Download ESXi 8.0 ISO from VMware
# 2. Boot from ISO on target server
# 3. Accept EULA
# 4. Select installation disk
# 5. Set root password
# 6. Configure management network (VLAN optional)

# After installation, SSH into ESXi:
ssh root@esxi-host-1

# 7. Enable SSH (if not auto-enabled)
vim-cmd /sbin/services/ssh restart

# 8. Verify GPU passthrough support
esxcli hardware pci list | grep NVIDIA
```

```
# 9. Enable GPU device passthrough
esxcli hardware pci esxpassthrough set -d=0000:01:00.0 -e=true

# 10. Reboot for GPU changes
reboot

# 11. Verify GPU is available for passthrough
esxcli hardware pci esxpassthrough list
```

Step 2: Create Virtual Machines

```
# Create VM via ESXi web UI or command line
# Configuration for single GPU VM:

cat > /tmp/llm-vm-config.vmx <<'EOF'
.encoding = "UTF-8"
displayName = "llm-inference-1"
memSize = 256000  # 256GB RAM
numvcpus = 32
numvcpus.cores = 8
cpuid.coresPerSocket = "8"
scsi0.present = "true"
scsi0.virtualDev = "pvscsi"
scsi0:0.present = "true"
scsi0:0.fileName = "/vmfs/volumes/datastore1/llm-inference-1/llm-disk.vmdk"
scsi0:0.deviceType = "scsi-hardDisk"
ethernet0.present = "true"
ethernet0.virtualDev = "e1000e"
ethernet0.networkName = "VM Network"
ethernet1.present = "true"
ethernet1.virtualDev = "e1000e"
ethernet1.networkName = "GPU-Cluster-Network"

# GPU passthrough
pciPassthru0.present = "true"
pciPassthru0.deviceId = "0x2204"  # NVIDIA A100
pciPassthru0.vendorId = "0x10de"  # NVIDIA vendor ID
EOF

# Register VM
vmkfstools -C vmfs8 /vmfs/volumes/datastore1/llm-inference-1/llm-disk.vmdk -d
thin 500G

# Via vSphere Web Client:
# 1. Create VM from template
# 2. Set 256GB RAM, 32 vCPU
# 3. Add raw device mapping for GPU: 01:00.0
# 4. Set memory reservation to 256GB (for performance)
# 5. Enable paravirtual SCSI adapter
```

Step 3: VM BIOS Configuration

```
Via vSphere Web Client:
1. VM Settings > Edit Settings
2. VM Options > Boot Options
   - Firmware: UEFI
   - Secure Boot: Off (for driver compatibility)

3. CPU/Memory settings:
   - CPU Reservation: 32000 MHz
   - Memory Reservation: 262144 MB
   - Hot add: Disabled (for stability)

4. Network settings:
   - NIC 1 (Management): vmnet0
   - NIC 2 (GPU Cluster): vmnet1 (10Gbps jumbo frames)
```

Step 4: Guest OS Installation in VM

```
# Boot Ubuntu 22.04 ISO from CD-ROM

# Same steps as bare metal:
# - Install OS with static IP
# - Update kernel/headers
# - Install vLLM and NVIDIA drivers
# - Set up Flask API gateway

# Key difference: GPU will be available via passthrough
lspci | grep NVIDIA  # Will show GPU as PCI device

# But use same vLLM setup as bare metal
```

Step 5: Storage Configuration

```
# Create VMFS datastore for VM storage
esxcli storage vmfs extent list

# Create NFS storage for shared models (optional)
esxcli storage nfs add -H 192.168.1.100 \
  -s /export/models \
  -v shared-models

# Mount models directory in VM
# In VM: mount -t nfs 192.168.1.100:/export/models /mnt/models

# Or: Use vSphere-managed storage
```

```
# VM uses virtual disks that live on VMFS datastore
# Automatic replication/backup via vSAN (if enabled)
```

Option B: Hyper-V (Microsoft)

Step 1: Hyper-V Installation on Windows Server

```
# Run as Administrator on Windows Server 2022

# Enable Hyper-V role
Enable-WindowsOptionalFeature -Online -FeatureName Hyper-V -All

# Restart required
Restart-Computer -Force

# Create virtual switch for management
New-VMSwitch -Name "Management" -SwitchType External `
  -NetAdapterName "Ethernet0"

# Create virtual switch for GPU cluster
New-VMSwitch -Name "GPU-Cluster" -SwitchType Private

# Configure jumbo frames (for high-speed GPU cluster network)
$adapter = Get-NetAdapter -Name "GPU-Cluster"
Set-NetAdapterAdvancedProperty -Name $adapter.Name `
  -RegistryKeyword "*JumboPacket" -RegistryValue 9014
```

Step 2: Create Hyper-V Virtual Machine

```
# Create VM with 256GB RAM, 32 vCPU

New-VM -Name "llm-inference-1" `
  -MemoryStartupBytes 256GB `
  -Generation 2 `
  -Path "D:\VMs\"

# Add virtual processors
Set-VMProcessor -VMName "llm-inference-1" -Count 32

# Reserve memory (no dynamic memory for performance)
Set-VMMemory -VMName "llm-inference-1" `
  -DynamicMemoryEnabled $false `
  -StartupBytes 256GB

# Add network adapters
Add-VMNetworkAdapter -VMName "llm-inference-1" `
  -SwitchName "Management"
```

```
Add-VMNetworkAdapter -VMName "llm-inference-1" `
  -SwitchName "GPU-Cluster"

# Add virtual disk
New-VHD -Path "D:\VMs\llm-inference-1\disk.vhdx" `
  -SizeBytes 500GB -Dynamic

Add-VMHardDiskDrive -VMName "llm-inference-1" `
  -Path "D:\VMs\llm-inference-1\disk.vhdx"

# Add DVD drive for ISO installation
Add-VMDvdDrive -VMName "llm-inference-1" `
  -ControllerNumber 0 -ControllerLocation 1

# Set boot order (DVD first for OS installation)
$vmDVD = Get-VMDvdDrive -VMName "llm-inference-1"
Set-VMFirmware -VMName "llm-inference-1" `
  -FirstBootDevice $vmDVD

# GPU Passthrough (requires Windows Server 2022 with GPU)
# Note: Discrete device assignment required

# Dismount GPU from host
Dismount-VMHostAssignableDevice -Force `
  -LocationPath "PCIROOT(0)#PCI(0100)#PCI(0000)"

# Assign to VM
Add-VMAssignableDevice -LocationPath `
  "PCIROOT(0)#PCI(0100)#PCI(0000)" `
  -VMName "llm-inference-1"

# Start VM
Start-VM -Name "llm-inference-1"
```

Step 3: Connect to VM and Install OS

```
# Connect to VM console
vmconnect.exe localhost "llm-inference-1"

# Boot from Ubuntu ISO and install (same as bare metal)
# Then inside the VM:

sudo apt update
sudo apt install -y linux-headers-$(uname -r)

# Install NVIDIA driver (GPU should be visible)
lspci | grep NVIDIA

# Continue with vLLM setup as before
```

Option C: KVM/QEMU (Open Source Linux)

Step 1: KVM Installation

```
# On Ubuntu 22.04 host

sudo apt update
sudo apt install -y \
    qemu-system-x86 \
    qemu-efi \
    qemu-efi-aarch64 \
    libvirt-daemon-system \
    libvirt-daemon-utils \
    virt-manager \
    bridge-utils \
    ovmf

# Verify KVM support
grep -c -w vmx /proc/cpuinfo  # Intel: should be > 0
grep -c -w svm /proc/cpuinfo  # AMD: should be > 0

# Enable and start libvirtd
sudo systemctl enable libvirtd
sudo systemctl start libvirtd

# Add user to libvirt group (to use without sudo)
sudo usermod -aG libvirt,kvm $USER
newgrp libvirt
```

Step 2: Create Virtual Network

```
# Create cluster network with jumbo frames
cat > /tmp/gpu-cluster-net.xml <<'EOF'
<network>
  <name>gpu-cluster</name>
  <forward mode='bridge'/>
  <bridge name='br-gpu'/>
  <mtu size='9000'/>
</network>
EOF

# Define network
virsh net-define /tmp/gpu-cluster-net.xml
virsh net-start gpu-cluster
virsh net-autostart gpu-cluster

# Create Linux bridge for jumbo frames
```

```
sudo ip link add br-gpu type bridge
sudo ip link set br-gpu mtu 9000
sudo ip link set br-gpu up
```

Step 3: Create KVM Virtual Machine

```
# Create disk
qemu-img create -f qcow2 /var/lib/libvirt/images/llm-inference-1.qcow2 500G

# Create VM via virsh or virt-manager
virt-manager &  # GUI tool

# Or via command line:
virt-install \
  --name llm-inference-1 \
  --memory 262144 \
  --vcpus 32 \
  --disk /var/lib/libvirt/images/llm-inference-1.qcow2 \
  --network network:default \
  --network network:gpu-cluster \
  --graphics spice \
  --console pty,target_type=virtio \
  --cdrom /path/to/ubuntu-22.04-iso \
  --boot cdrom,hd \
  --features kvm_hidden=on

# Wait for installation, then:
virsh console llm-inference-1

# After OS installation, configure GPU passthrough
```

Step 4: GPU Passthrough Configuration

```
# On host, find GPU:
lspci | grep NVIDIA

# Example: 01:00.0 3D controller: NVIDIA Corporation GA100

# Get vendor:device ID
lspci -n | grep "01:00.0"
# Output: 01:00.0 0302: 10de:2204 (10de=NVIDIA, 2204=A100)

# Enable IOMMU (Intel or AMD)
# For Intel: already enabled on most systems
# For AMD: ensure in kernel command line:
# GRUB_CMDLINE_LINUX="... iommu=pt amd_iommu=on"
```

```
# Bind to vfio-pci driver
sudo modprobe vfio
sudo modprobe vfio_pci
sudo modprobe vfio_virqfd

# Add to vfio-pci
echo "10de 2204" | sudo tee /etc/modprobe.d/vfio.conf

# Detach from nvidia driver
echo "0000:01:00.0" | sudo tee /sys/bus/pci/drivers/nvidia/unbind

# Attach to vfio-pci
echo "0000:01:00.0" | sudo tee /sys/bus/pci/drivers/vfio-pci/bind

# Verify
cat /sys/bus/pci/devices/0000:01:00.0/driver/module

# Now add to VM XML:
virsh edit llm-inference-1

# Add in <devices> section:
# <hostdev mode='subsystem' type='pci' managed='yes'>
#   <source>
#     <address domain='0x0000' bus='0x01' slot='0x00' function='0x0'/>
#   </source>
# </hostdev>
```

Container Orchestration

Step 1: Install Docker

```
# On each physical machine or VM

# Ubuntu installation
curl -fsSL https://get.docker.com -o get-docker.sh
sudo sh get-docker.sh

# Add user to docker group
sudo usermod -aG docker $USER
newgrp docker

# Verify
docker run hello-world
```

Step 2: Install Kubernetes (Microk8s for On-Premise)

```
# Microk8s: lightweight, production-grade K8s for on-premise

sudo snap install microk8s --classic --channel=1.27

# Enable required add-ons
microk8s enable dns
microk8s enable storage
microk8s enable gpu  # For NVIDIA GPU support

# Wait for all pods to be running
microk8s kubectl get pods -A

# Create alias
alias kubectl='microk8s kubectl'
```

Step 3: Deploy vLLM on Kubernetes

```
# File: llm-deployment.yaml
apiVersion: v1
kind: Namespace
metadata:
  name: llm-serving

---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: models-pvc
  namespace: llm-serving
spec:
  accessModes:
    - ReadWriteMany
  storageClassName: microk8s-hostpath
  resources:
    requests:
      storage: 500Gi

---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: vllm-server
  namespace: llm-serving
spec:
  replicas: 2
  selector:
    matchLabels:
      app: vllm
  template:
```

```
    metadata:
      labels:
        app: vllm
    spec:
      containers:
      - name: vllm
        image: vllm/vllm-openai:v0.2.0

        env:
        - name: MODEL_NAME
          value: "meta-llama/Llama-2-70b-chat-hf"
        - name: TENSOR_PARALLEL_SIZE
          value: "4"
        - name: GPU_MEMORY_UTILIZATION
          value: "0.9"

        ports:
        - containerPort: 8000

        resources:
          requests:
            nvidia.com/gpu: 4
          limits:
            nvidia.com/gpu: 4

        volumeMounts:
        - name: models
          mountPath: /root/.cache/huggingface

      volumes:
      - name: models
        persistentVolumeClaim:
          claimName: models-pvc

---
apiVersion: v1
kind: Service
metadata:
  name: vllm-service
  namespace: llm-serving
spec:
  type: LoadBalancer
  selector:
    app: vllm
  ports:
  - port: 8000
    targetPort: 8000
EOF

# Deploy
kubectl apply -f llm-deployment.yaml

# Verify
```

```
kubectl get pods -n llm-serving
kubectl get svc -n llm-serving
```

Model Serving Setup

Step 1: Download Models

```
# Models are typically downloaded to shared storage
# This ensures all nodes can access the same models

mkdir -p /mnt/models
cd /mnt/models

# Download using Python script
python3 <<'EOF'
from huggingface_hub import snapshot_download
import os

models = [
    "meta-llama/Llama-2-7b-chat-hf",
    "meta-llama/Llama-2-70b-chat-hf",
    "mistralai/Mistral-7B-v0.1",
]

for model in models:
    print(f"Downloading {model}...")
    snapshot_download(
        repo_id=model,
        cache_dir="/mnt/models",
        local_files_only=False,
        resume_download=True
    )
    print(f"✓ {model}")
EOF
```

Step 2: Configure vLLM Parameters

```
# For optimal performance, tune these parameters:

# GPU memory utilization (0.7-0.95)
export GPU_MEMORY_UTILIZATION=0.95

# Tensor parallelism (spread across N GPUs)
export TENSOR_PARALLEL_SIZE=4

# Pipeline parallelism (for very large models)
export PIPELINE_PARALLEL_SIZE=2
```

```
# Maximum tokens to generate at once
export MAX_MODEL_LEN=4096

# Enable prefix caching (for repeated prompts)
export ENABLE_PREFIX_CACHING=true

# Disable logging (for performance)
export DISABLE_LOG_STATS=false

# Start vLLM
python -m vllm.entrypoints.openai.api_server \
  --model meta-llama/Llama-2-70b-chat-hf \
  --tensor-parallel-size ${TENSOR_PARALLEL_SIZE} \
  --gpu-memory-utilization ${GPU_MEMORY_UTILIZATION} \
  --max-model-len ${MAX_MODEL_LEN} \
  --enable-prefix-caching \
  --port 8000
```

Networking & Security

Step 1: Network Architecture

```
Diagram:

[Internet] ---> [Firewall/NAT] ---> [Management Network]
                       |
                       +---> [API Gateway Layer] (Flask/FastAPI)
                               |
                               +---> [GPU Cluster Network - 100Gbps]
                                       |
                                       +---> [LLM Inference Nodes]
                                              (with attached GPUs)
                                       |
                                       +---> [Storage Network - 10Gbps NFS]

Network Segmentation:
1. Management (out-of-band): SSH, monitoring, deployment
2. API/Client: External API requests (HTTPS)
3. GPU Cluster: Internal GPU-to-GPU communication (high bandwidth)
4. Storage: NFS/iSCSI for models and data (high reliability)
```

Step 2: Firewall Configuration

```
# UFW firewall (Ubuntu)
sudo ufw enable
```

```
# Allow SSH (for management)
sudo ufw allow 22/tcp

# Allow vLLM inference port (internal only)
sudo ufw allow from 192.168.1.0/24 to any port 8000

# Allow API Gateway (from management network)
sudo ufw allow from 192.168.1.0/24 to any port 5000

# Allow metrics endpoint
sudo ufw allow from 192.168.1.0/24 to any port 9090

# Allow Kubernetes API (if using K8s)
sudo ufw allow from 192.168.1.0/24 to any port 6443

# Check rules
sudo ufw status verbose
```

Step 3: TLS/SSL Configuration

```
# Generate self-signed certificate for API gateway
openssl req -x509 -newkey rsa:4096 -nodes \
  -out /etc/ssl/certs/llm-api.crt \
  -keyout /etc/ssl/private/llm-api.key \
  -days 365

# Update Flask app to use HTTPS
cat > ~/llm-service/app-secure.py <<'EOF'
from flask import Flask
import ssl

app = Flask(__name__)

# ... existing routes ...

if __name__ == '__main__':
    # Create SSL context
    context = ssl.SSLContext(ssl.PROTOCOL_TLS_SERVER)
    context.load_cert_chain(
        '/etc/ssl/certs/llm-api.crt',
        '/etc/ssl/private/llm-api.key'
    )

    app.run(host='0.0.0.0', port=443, ssl_context=context)
EOF
```

Monitoring & Management

Step 1: System Monitoring

```
# Install monitoring stack
sudo apt install -y prometheus grafana-server node-exporter

# Configure Prometheus to scrape GPU metrics
cat > /etc/prometheus/prometheus.yml <<'EOF'
global:
  scrape_interval: 15s

scrape_configs:
  - job_name: 'node'
    static_configs:
      - targets: ['localhost:9100']

  - job_name: 'llm-metrics'
    static_configs:
      - targets: ['localhost:5000/metrics']
EOF

# Start services
sudo systemctl restart prometheus
sudo systemctl restart grafana-server

# Access Grafana at http://localhost:3000
# Default: admin/admin
```

Step 2: GPU Monitoring

```
# Real-time GPU stats
watch -n 1 nvidia-smi

# Log GPU metrics
nvidia-smi dmon > gpu-metrics.log

# Parse metrics script
python3 <<'EOF'
import subprocess
import json

def get_gpu_metrics():
    cmd = "nvidia-smi --query-
gpu=index,utilization.gpu,memory.used,temperature.gpu --format=csv"
    output = subprocess.check_output(cmd, shell=True).decode()

    for line in output.strip().split('\n')[1:]:
        gpu_id, util, mem, temp = line.split(', ')
        print(f"GPU {gpu_id}: {util} util, {mem} mem, {temp} temp")
```

```
get_gpu_metrics()
EOF
```

Disaster Recovery & Backup

Step 1: Backup Strategy

```
# Daily backup script
cat > /usr/local/bin/llm-backup.sh <<'EOF'
#!/bin/bash

BACKUP_DIR="/backups/llm"
DATE=$(date +%Y%m%d-%H%M%S)

# Backup configuration
tar -czf $BACKUP_DIR/llm-config-$DATE.tar.gz \
    /etc/systemd/system/llm* \
    /etc/systemd/system/vllm* \
    ~/llm-service/

# Backup models (if on local storage)
# Note: This can be very large, consider incremental backup
rsync -av /mnt/models/ /backups/models/ --delete

# Upload to remote backup
gsutil -m cp $BACKUP_DIR/llm-config-$DATE.tar.gz gs://llm-backups/

echo "Backup complete: $DATE"
EOF

# Schedule daily at 2 AM
sudo crontab -e
# Add: 0 2 * * * /usr/local/bin/llm-backup.sh
```

Operational Runbooks

Runbook 1: Daily Health Check

```
#!/bin/bash
# Daily health check script

echo "=== LLM System Health Check ==="
echo "Timestamp: $(date)"

# Check 1: GPU Status
echo -e "\n1. GPU Status:"
```

```
nvidia-smi --query-
gpu=index,name,pci.bus_id,pstate,temperature.gpu,utilization.gpu,memory.used,me
mory.total --format=csv

# Check 2: Service Status
echo -e "\n2. Service Status:"
sudo systemctl status vllm-inference llm-api --no-pager

# Check 3: Network Status
echo -e "\n3. Network Status:"
ip addr show | grep inet
ethtool -i eth0

# Check 4: Storage Status
echo -e "\n4. Storage Status:"
df -h /mnt/models /

# Check 5: API Health
echo -e "\n5. API Health:"
curl -s http://localhost:5000/health | jq

# Check 6: Performance Baseline
echo -e "\n6. Test Inference (Baseline):"
time curl -s http://localhost:5000/v1/completions \
  -H "Content-Type: application/json" \
  -d '{"model":"meta-llama/Llama-2-7b-chat-
hf","prompt":"Test","max_tokens":10}' | jq '.usage'

echo -e "\n=== Health Check Complete ==="
```

Runbook 2: Disaster Recovery

```
#!/bin/bash
# Disaster recovery restoration

echo "LLM Cluster Recovery Procedure"

# Step 1: Restore configuration
echo "Restoring configuration from backup..."
gsutil cp gs://llm-backups/llm-config-LATEST.tar.gz /tmp/
tar -xzf /tmp/llm-config-LATEST.tar.gz -C /

# Step 2: Restart services
echo "Restarting services..."
sudo systemctl daemon-reload
sudo systemctl restart vllm-inference
sudo systemctl restart llm-api

# Step 3: Verify services
echo "Verifying services..."
```

```
sudo systemctl status vllm-inference llm-api

# Step 4: Run health checks
echo "Running health checks..."
bash /usr/local/bin/health-check.sh

# Step 5: Notify team
echo "Recovery complete. Notifying team..."
curl -X POST https://hooks.slack.com/... \
  -d '{"text": "LLM cluster recovered successfully"}'
```

Troubleshooting Guide

GPU Not Detected

```
# Check 1: NVIDIA driver installed?
nvidia-smi
# If not found: Install driver (see earlier section)

# Check 2: GPU powered on?
lspci | grep NVIDIA
# If not in list: Check BIOS settings, power cables

# Check 3: GPU firmware?
nvidia-smi --query-gpu=pci.bus_id --format=csv
# Verify firmware version with: nvidia-smi -q | grep Firmware
```

Out of Memory Errors

```
# Check GPU memory
nvidia-smi --query-gpu=memory.free,memory.used,memory.total --format=csv

# Solutions:
# 1. Reduce model size (use 7B instead of 70B)
# 2. Enable quantization (8-bit or 4-bit)
# 3. Reduce batch size
# 4. Reduce max_model_len parameter

# Example: Run 8-bit quantized model
python -m vllm.entrypoints.openai.api_server \
  --model meta-llama/Llama-2-70b-chat-hf \
  --quantization bitsandbytes
```

Slow Inference

```
# Check GPU utilization
nvidia-smi -l 1  # Refresh every 1 second

# If low utilization (<50%):
# 1. Check CPU usage (might be bottleneck)
# 2. Check network latency
# 3. Check disk I/O (model swapping?)

# Solutions:
# 1. Enable tensor parallelism across more GPUs
# 2. Increase batch size
# 3. Use faster network (check RTT: ping)

# Benchmark performance
python -m vllm.entrypoints.openai.api_server \
  --model meta-llama/Llama-2-7b-chat-hf \
  --disable-log-stats false  # Enable detailed logging
```

Production Checklist

```
Pre-Deployment:
☑ All hardware received and tested
☑ Network tested (ping, iperf for bandwidth)
☑ Storage tested (copy test files)
☑ Power tested (load test for 1 hour)
☑ Cooling tested (temperature monitoring)

OS & Drivers:
☑ OS installed and updated
☑ NVIDIA drivers installed and verified
☑ CUDA installed and tested
☑ vLLM installed and tested
☑ SSH keys configured

Networking:
☑ Static IPs assigned
☑ DNS configured
☑ Firewalls configured
☑ VLANs segmented (if applicable)
☑ Jumbo frames enabled (for GPU cluster)

Services:
☑ vLLM service starts on boot
☑ API gateway service starts on boot
☑ Monitoring services running
☑ Backup scripts scheduled
☑ Health checks automated
```

```
Security:
☑ Sudo access restricted
☑ SSH hardened (key-only, no passwords)
☑ Firewall rules minimal (deny all except needed)
☑ TLS certificates installed
☑ API authentication configured

Monitoring:
☑ Prometheus collecting metrics
☑ Grafana dashboards created
☑ Alerts configured (high temp, low disk, etc.)
☑ Log aggregation configured
☑ Performance baselines recorded

Documentation:
☑ Network diagram created
☑ Hardware inventory documented
☑ Runbooks written
☑ Emergency contacts posted
☑ Change log started
```

CPU-Only Deployment (No GPUs)

When to Use CPU-Only Deployment

```
✓ Use Cases:
  • Small models (7B parameters or smaller)
  • Low-throughput requirements (<100 requests/day)
  • Budget-constrained deployments
  • Edge deployments (data centers, offices)
  • Development/testing environments
  • Compliance requirements (no external hardware)

✗ Not Suitable For:
  • Real-time, low-latency applications
  • High-throughput production systems (>1000 req/day)
  • Large models (70B+ parameters)
  • Time-sensitive inference (<5 second SLA)
```

Hardware Selection for CPU-Only

CPU Options

```
High-Performance CPUs (Recommended):

AMD EPYC 9684X
├─ Cores: 96 cores / 192 threads
```

```
├─ Clock: 2.5 GHz base, 3.6 GHz boost
├─ Cost: $13,000 per socket
├─ TDP: 360W
├─ Throughput: ~25 tokens/sec per socket (7B model)
└─ Memory bandwidth: 600 GB/s

AMD EPYC 9384X
├─ Cores: 64 cores / 128 threads
├─ Clock: 2.5 GHz base, 3.6 GHz boost
├─ Cost: $8,000 per socket
├─ TDP: 240W
├─ Throughput: ~15 tokens/sec per socket (7B model)
└─ Memory bandwidth: 400 GB/s (good alternative)

Intel Xeon Platinum 8592+
├─ Cores: 60 cores / 120 threads
├─ Clock: 3.5 GHz base, 4.0 GHz boost
├─ Cost: $12,000 per socket
├─ TDP: 270W
├─ Throughput: ~20 tokens/sec per socket (7B model)
└─ Memory bandwidth: 336 GB/s

Recommendation:
For CPU-only: Choose AMD EPYC 9684X (96 cores = better throughput)
Typical Configuration: 2 sockets = 192 cores total = ~50 tokens/sec
```

Memory Configuration (Critical for CPU)

```
CPU Memory Requirements:

Rule of Thumb: 2-3x Model Size

Examples:

7B Model (13-14 GB weights):
├─ Minimum RAM: 32-48 GB
├─ Recommended: 64-128 GB (for batch processing)
├─ Type: DDR5 RDIMM ECC preferred
└─ Speed: 5600MHz or faster

13B Model (25-26 GB weights):
├─ Minimum RAM: 64-96 GB
├─ Recommended: 128-256 GB
└─ Cost: $8,000-12,000

30B Model (60+ GB weights):
├─ Minimum RAM: 192-256 GB
├─ Recommended: 256-512 GB
└─ Not practical (better to use GPU)
```

```
Recommended Setup:
- 2x AMD EPYC 9684X
- 512 GB DDR5 RDIMM ECC (8x64GB sticks)
- Total RAM: $15,000-20,000
```

Storage & CPU Affinity

```
Storage Layout for CPU:

Tier 1: NVMe (Model Cache)
├─ Capacity: 256-512GB NVMe
├─ Purpose: Hot model cache
├─ Speed: PCIe 4.0/5.0
└─ Cost: $500-1,000

Tier 2: SAS SSD (Swap)
├─ Capacity: 2-4 TB SAS SSD
├─ Purpose: Paging for CPU (when RAM exhausted)
├─ Speed: 15K RPM
└─ Cost: $2,000-3,000

CPU Affinity (for NUMA systems):
# On 2-socket EPYC system:
# Socket 0: cores 0-95
# Socket 1: cores 96-191

# Bind vLLM to specific socket (NUMA optimization)
numactl --cpunodebind=0 --membind=0 python -m vllm...
```

CPU-Only Model Selection

Compatible Models

```
Recommended Models for CPU:

Llama 2 7B Chat
├─ Size: ~13 GB
├─ Throughput: 30-40 tokens/sec (single socket)
├─ Batch size: 1-2 (CPU-only)
├─ Memory: 32-64 GB recommended
└─ Quality: High, instruction-tuned

Mistral 7B
├─ Size: ~13 GB
├─ Throughput: 35-45 tokens/sec (single socket)
├─ Batch size: 2-4
├─ Memory: 48-64 GB
```

```
└─ Quality: High, efficient

OpenHermes 2.5 7B
├─ Size: ~13 GB
├─ Throughput: 30-40 tokens/sec
├─ Memory: 32-64 GB
└─ Quality: Good, multi-lingual

Neural Chat 7B
├─ Size: ~13 GB
├─ Throughput: 30-40 tokens/sec
├─ Memory: 32-64 GB
└─ Quality: Good, chat-optimized

Models to AVOID (CPU-only):
✗ Llama 2 70B (too large, requires GPU)
✗ Mistral Mixture-of-Experts (MoE) (needs GPU)
✗ Yi 34B (too large)
✗ Code-Llama 34B/70B (GPU recommended)
```

CPU-Only Installation

Step 1: CPU-Optimized vLLM Setup

```
# Install vLLM with CPU-specific optimizations
pip install vllm[cpu]

# Or with OpenVINO backend (Intel optimization)
pip install vllm openvino

# Verify CPU detection
python -c "import vllm;
print(vllm.device_executor.DetectorType.detect_device())"

# Expected output: CPU or openvino
```

Step 2: Download Model for CPU

```
# Download 7B model to local storage
python3 <<'EOF'
from huggingface_hub import snapshot_download
import os

# Create models directory
os.makedirs('/mnt/models', exist_ok=True)

# Download Llama 2 7B Chat
```

```
print("Downloading Llama 2 7B Chat...")
snapshot_download(
    repo_id="meta-llama/Llama-2-7b-chat-hf",
    cache_dir="/mnt/models",
    local_files_only=False,
    resume_download=True
)

print("✓ Model ready at /mnt/models")
EOF

# Verify model
ls -lh /mnt/models/
```

Step 3: CPU-Optimized vLLM Service

```
# Create CPU-optimized systemd service
sudo tee /etc/systemd/system/vllm-cpu.service <<'EOF'
[Unit]
Description=vLLM CPU Inference Server
After=network.target

[Service]
Type=simple
User=ubuntu
WorkingDirectory=/home/ubuntu/llm-service
Environment="PATH=/home/ubuntu/llm-serving/bin"

# CPU Affinity - bind to specific NUMA node (socket 0)
ExecStartPre=/bin/bash -c 'echo "Setting CPU affinity..."'

# NUMA optimization
Environment="NUMA_NODE=0"

# Disable GPU
Environment="CUDA_VISIBLE_DEVICES="

# Model configuration
Environment="MODEL_NAME=meta-llama/Llama-2-7b-chat-hf"
Environment="MAX_MODEL_LEN=2048"
Environment="TENSOR_PARALLEL_SIZE=1"

# CPU-specific optimizations
Environment="OMP_NUM_THREADS=96"
Environment="OMP_PLACES=cores"
Environment="OMP_PROC_BIND=close"

# Memory optimization
Environment="VLLM_ATTENTION_BACKEND=openvino"
```

```
# Startup command - using numactl for NUMA binding
ExecStart=/usr/bin/numactl \
    --cpunodebind=0 \
    --membind=0 \
    /home/ubuntu/llm-serving/bin/python -m vllm.entrypoints.openai.api_server \
    --model ${MODEL_NAME} \
    --port 8000 \
    --max-model-len ${MAX_MODEL_LEN} \
    --device cpu \
    --dtype bfloat16

# Restart on failure
Restart=on-failure
RestartSec=5

# Performance: no CPU frequency scaling (if possible)
CPUQuota=100%
CPUSchedulingPolicy=rr

# Logging
StandardOutput=journal
StandardError=journal
SyslogIdentifier=vllm-cpu

[Install]
WantedBy=multi-user.target
EOF

# Enable and start
sudo systemctl daemon-reload
sudo systemctl enable vllm-cpu
sudo systemctl start vllm-cpu
sudo systemctl status vllm-cpu
```

Step 4: Test CPU Inference

```
# Wait for service to start (~30 seconds for model loading)
sleep 30

# Test 1: Health check
curl http://localhost:8000/health

# Test 2: Simple completion
curl http://localhost:8000/v1/completions \
  -H "Content-Type: application/json" \
  -d '{
    "model": "meta-llama/Llama-2-7b-chat-hf",
    "prompt": "What is artificial intelligence?",
    "max_tokens": 50,
    "temperature": 0.7
```

```
  }' | jq

# Expected output (slower than GPU, but working):
# Takes 5-10 seconds for 50 tokens

# Test 3: Measure throughput
python3 <<'EOF'
import requests
import time

prompt = "Explain quantum computing in simple terms. "
times = []

for i in range(5):
    start = time.time()
    response = requests.post(
        "http://localhost:8000/v1/completions",
        json={
            "model": "meta-llama/Llama-2-7b-chat-hf",
            "prompt": prompt,
            "max_tokens": 100,
            "temperature": 0.7
        }
    )
    duration = time.time() - start
    tokens = response.json()['usage']['completion_tokens']
    throughput = tokens / duration

    times.append(duration)
    print(f"Run {i+1}: {tokens} tokens in {duration:.2f}s ({throughput:.1f}
tokens/sec)")

avg_time = sum(times) / len(times)
print(f"\nAverage: {avg_time:.2f}s per request")
EOF
```

CPU-Only API Gateway

```
# Enhanced Flask API for CPU-only (with request queuing)
cat > ~/llm-service/app-cpu.py <<'EOF'
#!/usr/bin/env python3

from flask import Flask, request, jsonify
import requests
import logging
from prometheus_client import Counter, Histogram, generate_latest
import time
from queue import Queue
from threading import Thread
import json
```

```
app = Flask(__name__)

# Configure logging
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

VLLM_ENDPOINT = "http://localhost:8000"

# Request queue (CPU is slower, queue requests)
request_queue = Queue(maxsize=50)

# Metrics
request_count = Counter(
    'llm_cpu_requests_total',
    'Total LLM CPU requests',
    ['model', 'status']
)

request_duration = Histogram(
    'llm_cpu_request_duration_seconds',
    'LLM CPU request duration',
    ['model', 'tokens']
)

queue_depth = Counter(
    'llm_cpu_queue_depth',
    'Current queue depth'
)

@app.route('/v1/completions', methods=['POST'])
def completions():
    """CPU-optimized completions with request queuing"""

    try:
        data = request.json
        model = data.get('model', 'meta-llama/Llama-2-7b-chat-hf')
        prompt = data.get('prompt')

        if not prompt:
            return {"error": "prompt required"}, 400

        start_time = time.time()

        # For CPU, limit concurrent requests
        if request_queue.qsize() > 5:
            return {
                "error": "Server busy, queue depth: " +
str(request_queue.qsize()),
                "retry_after": 10
            }, 503

        logger.info(f"CPU Completion - Prompt length: {len(prompt)}")
```

```
        # Call vLLM with timeout
        response = requests.post(
            f"{VLLM_ENDPOINT}/v1/completions",
            json=data,
            timeout=300  # CPU is slower
        )

        if response.status_code != 200:
            logger.error(f"vLLM error: {response.status_code}")
            request_count.labels(model=model, status='error').inc()
            return {"error": "Generation failed"}, 500

        result = response.json()
        tokens = result['usage']['completion_tokens']

        duration = time.time() - start_time
        request_duration.labels(model=model, tokens=tokens).observe(duration)
        request_count.labels(model=model, status='success').inc()

        logger.info(f"✓ {tokens} tokens in {duration:.1f}s
({tokens/duration:.1f} tok/s)")

        return result

    except Exception as e:
        logger.error(f"Error: {str(e)}")
        request_count.labels(model='unknown', status='error').inc()
        return {"error": str(e)}, 500

@app.route('/v1/chat/completions', methods=['POST'])
def chat_completions():
    """Chat endpoint for CPU (simpler batching)"""

    try:
        data = request.json
        messages = data.get('messages', [])

        if not messages:
            return {"error": "messages required"}, 400

        # Convert messages to prompt
        prompt = "\n".join([f"{m['role']}: {m['content']}" for m in messages])

        start_time = time.time()

        response = requests.post(
            f"{VLLM_ENDPOINT}/v1/completions",
            json={
                "model": data.get('model', 'meta-llama/Llama-2-7b-chat-hf'),
                "prompt": prompt,
                "max_tokens": data.get('max_tokens', 256),
                "temperature": data.get('temperature', 0.7),
```

```
            },
            timeout=300
        )

        result = response.json()
        duration = time.time() - start_time

        logger.info(f"Chat: {result['usage']['completion_tokens']} tokens in
{duration:.1f}s")

        return result

    except Exception as e:
        return {"error": str(e)}, 500

@app.route('/health', methods=['GET'])
def health():
    """Health check"""
    try:
        response = requests.get(f"{VLLM_ENDPOINT}/health", timeout=5)
        return {
            "status": "healthy" if response.status_code == 200 else "degraded",
            "backend": "cpu"
        }
    except:
        return {"status": "unhealthy", "backend": "cpu"}, 500

@app.route('/metrics', methods=['GET'])
def metrics():
    """Prometheus metrics"""
    return generate_latest()

@app.route('/system/info', methods=['GET'])
def system_info():
    """CPU system info"""
    import psutil
    return {
        "type": "cpu-only",
        "cpu_count": psutil.cpu_count(),
        "cpu_freq": psutil.cpu_freq().current,
        "memory_total_gb": psutil.virtual_memory().total / (1024**3),
        "memory_available_gb": psutil.virtual_memory().available / (1024**3),
        "queue_depth": request_queue.qsize()
    }

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000, debug=False, threaded=True)
EOF

chmod +x ~/llm-service/app-cpu.py
```

CPU-Only Performance Optimization

OpenVINO Backend (Intel Optimization)

```
# For Intel CPUs, use OpenVINO backend (faster inference)
pip install openvino openvino-tokenizers

# Modified service configuration
cat > /etc/systemd/system/vllm-cpu-openvino.service <<'EOF'
[Unit]
Description=vLLM CPU with OpenVINO Backend
After=network.target

[Service]
Type=simple
User=ubuntu
Environment="PATH=/home/ubuntu/llm-serving/bin"
Environment="CUDA_VISIBLE_DEVICES="
Environment="OPENVINO_DEVICE=CPU"

ExecStart=/home/ubuntu/llm-serving/bin/python -m
vllm.entrypoints.openai.api_server \
    --model meta-llama/Llama-2-7b-chat-hf \
    --device cpu \
    --enforce-eager

Restart=on-failure
StandardOutput=journal

[Install]
WantedBy=multi-user.target
EOF

sudo systemctl daemon-reload
sudo systemctl start vllm-cpu-openvino
```

Quantization for CPU

```
# 8-bit quantization (reduces memory by 75%)
cat > ~/llm-service/run-quantized-cpu.sh <<'EOF'
#!/bin/bash

python -m vllm.entrypoints.openai.api_server \
    --model meta-llama/Llama-2-7b-chat-hf \
    --device cpu \
    --quantization bitsandbytes \
    --load-in-8bit \
    --dtype bfloat16
```

```
EOF

chmod +x ~/llm-service/run-quantized-cpu.sh

# Test quantized inference
~/llm-service/run-quantized-cpu.sh &
sleep 30
curl http://localhost:8000/health
```

CPU-Only Monitoring

```
# CPU-specific monitoring script
python3 <<'EOF'
import psutil
import time
import json

def cpu_monitor(interval=5):
    """Monitor CPU performance"""

    print("=== CPU-Only LLM Monitoring ===")

    while True:
        cpu_percent = psutil.cpu_percent(interval=1)
        memory = psutil.virtual_memory()
        disk = psutil.disk_usage('/')

        # Per-core usage
        per_core = psutil.cpu_percent(interval=0.1, percpu=True)
        hot_cores = sum(1 for c in per_core if c > 80)

        print(f"\nTime: {time.strftime('%H:%M:%S')}")
        print(f"CPU: {cpu_percent}% (hot cores: {hot_cores})")
        print(f"Memory: {memory.percent}% ({memory.used/(1024**3):.1f}GB /
{memory.total/(1024**3):.1f}GB)")
        print(f"Disk: {disk.percent}%")

        # Temperature (if available)
        try:
            temps = psutil.sensors_temperatures()
            if 'k10temp' in temps or 'coretemp' in temps:
                for name, entries in temps.items():
                    if 'k10temp' in name or 'coretemp' in name:
                        print(f"CPU Temp: {entries[0].current}°C")
        except:
            pass

        time.sleep(interval)
```

```
cpu_monitor()
EOF
```

CPU-Only Cost Analysis

```
5-Year Total Cost of Ownership (TCO):

Option 1: CPU-Only Deployment
├─ Hardware
│  ├─ 2x AMD EPYC 9684X ($26,000)
│  ├─ 512GB DDR5 RDIMM ($18,000)
│  ├─ 256GB NVMe ($1,000)
│  ├─ 4TB SAS SSD ($2,000)
│  └─ 1x Chassis + PSU ($8,000)
├─ Subtotal hardware: $55,000
├─
├─ Operations (5 years)
│  ├─ Electricity: $4,000 (1.5kW × 24h × 365d × 5y @ $0.12/kWh)
│  ├─ Cooling: $2,000
│  ├─ Maintenance: $5,000
│  └─ Support: $10,000
├─ Subtotal operations: $21,000
│
├─ TOTAL 5-YEAR TCO: $76,000
├─ Annual cost: $15,200
└─ Cost per inference request (1M/month): $0.015

Option 2: AWS EC2 CPU Instances
├─ Instance: c7i.24xlarge (192 vCPU)
├─ Cost: $5/hour
├─ Annual: $44,000
├─ 5-Year: $220,000
└─ Cost per inference (1M/month): $0.22

SAVINGS: CPU-on-premise is 65% cheaper over 5 years
Break-even: ~18 months vs cloud
```

CPU-Only Use Cases

Use Case 1: Document Classification

```
# Low-throughput document classification on CPU
from flask import Flask
import requests

app = Flask(__name__)

@app.route('/classify', methods=['POST'])
```

```
def classify_document():
    document = request.json.get('text')

    # Use 7B model to classify
    response = requests.post(
        'http://localhost:8000/v1/completions',
        json={
            'model': 'meta-llama/Llama-2-7b-chat-hf',
            'prompt': f"""Classify this document into ONE category: Finance,
Healthcare, Legal, or Other.

Document: {document[:1000]}

Category: """,
            'max_tokens': 20,
            'temperature': 0.1
        }
    )

    category = response.json()['choices'][0]['text'].strip()
    return {'document_id': request.json.get('id'), 'category': category}

# Run on CPU with 30 second SLA
# Throughput: 100-200 documents/day
```

Use Case 2: Batch Email Summarization

```
# Overnight batch processing
import time
from datetime import datetime

def batch_summarize_emails(email_list):
    """Process emails overnight (CPU acceptable)"""

    results = []
    start_time = time.time()

    for email in email_list:
        response = requests.post(
            'http://localhost:8000/v1/completions',
            json={
                'model': 'meta-llama/Llama-2-7b-chat-hf',
                'prompt': f"""Summarize this email in 2-3 sentences:

{email['body'][:2000]}

Summary: """,
                'max_tokens': 100
            }
        )
```

```
        results.append({
            'email_id': email['id'],
            'summary': response.json()['choices'][0]['text']
        })

    duration = time.time() - start_time
    print(f"Processed {len(email_list)} emails in {duration:.0f}s")

    return results

# Process 1000 emails overnight
# 96 cores CPU = ~40 tokens/sec × 3600 sec = 144,000 tokens/hour
# 1000 emails × 100 tokens = 100,000 tokens total
# Runtime: ~40 minutes (acceptable for batch)
```

CPU-Only Benchmarks

```
# Benchmark script for CPU inference
python3 <<'EOF'
import requests
import time
import statistics

def benchmark_cpu_inference(num_runs=10):
    """Benchmark CPU throughput"""

    times = []
    tokens_list = []

    for i in range(num_runs):
        start = time.time()

        response = requests.post(
            'http://localhost:8000/v1/completions',
            json={
                'model': 'meta-llama/Llama-2-7b-chat-hf',
                'prompt': 'Explain machine learning in 100 words: ',
                'max_tokens': 100,
                'temperature': 0.7
            }
        )

        duration = time.time() - start
        tokens = response.json()['usage']['completion_tokens']

        times.append(duration)
        tokens_list.append(tokens)

        throughput = tokens / duration
```

```
        print(f"Run {i+1}: {tokens} tokens in {duration:.2f}s ({throughput:.1f}
tok/sec)")

    print(f"\nBenchmark Results (n={num_runs}):")
    print(f"  Avg time: {statistics.mean(times):.2f}s")
    print(f"  Avg throughput: {statistics.mean([t/d for t,d in zip(tokens_list,
times)]):.1f} tok/sec")
    print(f"  P95 latency: {sorted(times)[int(num_runs*0.95)]:.2f}s")
    print(f"  Min: {min(times):.2f}s, Max: {max(times):.2f}s")

benchmark_cpu_inference()
EOF
```

Next Steps: Proceed to operational management, scaling strategies, and long-term optimization of your on-premise LLM infrastructure.