LLM Deployment Use Cases

Use Case 1: Customer Support Chatbot

Business Context

- **Users:** 10,000 daily active users
- **Peak Load:** 500 concurrent users
- **Availability Requirement:** 99.9% uptime
- **Response Time:** < 5 seconds
- **Budget:** $10,000/month

Architecture Decision

```
Traffic Pattern:
- 9 AM - 5 PM: Heavy traffic (400 concurrent)
- 5 PM - 9 AM: Light traffic (50 concurrent)
- Weekends: Very light (20 concurrent)
```

Chosen: AWS Bedrock + Auto-scaling + Caching

```
Why:
- Easy to scale (peak to light without manual intervention)
- Good cost for variable traffic (pay per token)
- Reliable enterprise-grade SLA
- Easy integration with CRM
```

Implementation

Step 1: Set up Bedrock

```
# Enable model access
aws bedrock request-model-access --model-id anthropic.claude-3-sonnet-20240229-v1:0
```

Step 2: Create Flask Application

```python
from flask import Flask, request, jsonify
import boto3
import json
from datetime import datetime
```

```python
app = Flask(__name__)
bedrock = boto3.client('bedrock-runtime', region_name='us-east-1')

SYSTEM_PROMPT = """You are a helpful customer support representative for 
TechCorp.
You have access to:
- Product documentation
- Pricing information
- Troubleshooting guides

Rules:
- Be friendly and professional
- If you don't know, offer to connect to human agent
- Don't make promises about refunds/discounts
- Keep responses under 200 words"""

@app.route('/support/chat', methods=['POST'])
def support_chat():
    data = request.json
    user_id = data.get('user_id')
    message = data.get('message')
    conversation_history = data.get('history', [])

    # Add current message to history
    conversation_history.append({
        "role": "user",
        "content": message
    })

    # Call Claude
    response = bedrock.invoke_model(
        modelId='anthropic.claude-3-sonnet-20240229-v1:0',
        contentType='application/json',
        accept='application/json',
        body=json.dumps({
            "anthropic_version": "bedrock-2023-06-01",
            "max_tokens": 1024,
            "system": SYSTEM_PROMPT,
            "messages": conversation_history
        })
    )

    result = json.loads(response['body'].read())
    assistant_response = result['content'][0]['text']

    # Add response to history
    conversation_history.append({
        "role": "assistant",
        "content": assistant_response
    })

    # Log interaction
    log_interaction(user_id, message, assistant_response, result['usage'])
```

```python
    # Check if escalation needed
    if should_escalate(assistant_response):
        return {
            "response": assistant_response,
            "escalate": True,
            "queue_position": get_queue_position(),
            "estimated_wait": "5-10 minutes"
        }

    return {
        "response": assistant_response,
        "escalate": False,
        "history": conversation_history[-10:]  # Keep last 10 messages
    }

def log_interaction(user_id, message, response, usage):
    """Log for analytics and auditing"""
    # CloudWatch
    cloudwatch = boto3.client('cloudwatch')
    cloudwatch.put_metric_data(
        Namespace='CustomerSupport',
        MetricData=[
            {
                'MetricName': 'TokensUsed',
                'Value': usage['input_tokens'] + usage['output_tokens'],
                'Unit': 'Count',
                'Dimensions': [
                    {'Name': 'UserId', 'Value': user_id}
                ]
            }
        ]
    )

    # Database (for compliance/analysis)
    db.insert_conversation({
        'user_id': user_id,
        'timestamp': datetime.now(),
        'user_message': message,
        'assistant_response': response,
        'tokens': usage['input_tokens'] + usage['output_tokens']
    })

def should_escalate(response):
    """Check if human escalation needed"""
    escalation_keywords = [
        "connect you to a specialist",
        "speak with a manager",
        "technical support team",
        "sales team"
    ]

    return any(keyword in response.lower() for keyword in escalation_keywords)
```

```
if __name__ == '__main__':
    app.run(port=5000)
```

Step 3: Deploy to Lambda + API Gateway

```
# Package and deploy
zip -r lambda.zip .

aws lambda create-function \
  --function-name llm-support-chat \
  --runtime python3.11 \
  --role arn:aws:iam::ACCOUNT:role/lambda-bedrock \
  --handler lambda_function.lambda_handler \
  --zip-file fileb://lambda.zip \
  --timeout 30 \
  --memory-size 512

# Create API Gateway
aws apigateway create-rest-api --name support-chat-api
```

Step 4: Set up Caching

```
import redis

cache = redis.Redis(host='elasticache-endpoint', port=6379)

@app.route('/support/chat', methods=['POST'])
def support_chat():
    # Check cache for similar questions
    cache_key = f"faq:{hash_prompt(message)}"
    cached_response = cache.get(cache_key)

    if cached_response:
        return {"response": cached_response, "source": "cache"}

    # Not in cache, call Bedrock
    ...

    # Cache for 24 hours
    cache.setex(cache_key, 86400, assistant_response)
```

Costs

Monthly Cost Breakdown:

```
Bedrock API calls:
- 10k users × 20 messages/day × 30 days = 6M API calls
- 150 tokens input, 200 tokens output per message
- Input cost: (6M × 150 × $0.003) / 1000 = $2,700
- Output cost: (6M × 200 × $0.015) / 1000 = $18,000
- Total API cost: $20,700

Lambda:
- 6M invocations × $0.20 per 1M = $1.20
- Compute: Negligible

ElastiCache (1GB):
- $27/month

API Gateway:
- $3.50 per million API calls
- 6M calls × $3.50 = $21 (after first 1B free)

Total: ~$20,750/month
```

Cost Optimization:

```
Option 1: Use smaller model (Llama 2 instead of Claude)
- Reduce cost by 50%

Option 2: More aggressive caching
- FAQ questions (40% of traffic) can be cached
- Reduce API calls by 40% = $8,300 savings

Option 3: Schedule-based scaling
- Use provisioned capacity during peak hours
- Save 30% with commitment

After optimization: ~$10,000/month (within budget)
```

Monitoring

Key Metrics:

```
- Chat latency (P95): <3 seconds
- Error rate: <0.5%
- Daily active users: 10,000
- Messages per day: 200,000
- Cache hit rate: >40%
- Cost per conversation: <$0.10
```

Sample Dashboard Query (CloudWatch):

```
# Average response time
cloudwatch.get_metric_statistics(
    Namespace='CustomerSupport',
    MetricName='ChatLatency',
    StartTime=datetime.now() - timedelta(hours=24),
    EndTime=datetime.now(),
    Period=3600,
    Statistics=['Average', 'Maximum']
)
```

Use Case 2: Content Generation Pipeline

Business Context

- **Requirement:** Generate 5,000 product descriptions/day
- **Timeline:** Complete by midnight daily
- **Quality:** 90%+ acceptable (reviewed by editors)
- **Budget:** $2,000/month
- **Customization:** Based on product category & style

Architecture Decision

Chosen: Batch Processing with SQS + Lambda

```
Why:
- Non-real-time = batch API discount (50%)
- Schedule-based (off-peak execution)
- Cost-effective for high volume
- Can pause/resume based on budget
```

Implementation

Step 1: Upload Product Data to S3

```
import pandas as pd
import boto3

# Product data
products = [
    {"id": "PROD001", "name": "Laptop", "category": "Electronics", "style":
"technical"},
    {"id": "PROD002", "name": "Coffee Maker", "category": "Appliances",
"style": "casual"},
```

```
    ...
]

df = pd.DataFrame(products)

s3 = boto3.client('s3')
s3.put_object(
    Bucket='product-data',
    Key='products-to-generate.csv',
    Body=df.to_csv(index=False)
)
```

Step 2: Create SQS Queue

```
# Create queue
aws sqs create-queue --queue-name content-generation --attributes
VisibilityTimeout=900

# Get queue URL
aws sqs get-queue-url --queue-name content-generation
```

Step 3: Enqueue Tasks

```
import csv

sqs = boto3.client('sqs')
s3 = boto3.client('s3')

# Read products from S3
obj = s3.get_object(Bucket='product-data', Key='products-to-generate.csv')
products = pd.read_csv(obj['Body'])

# Send each to SQS
for idx, product in products.iterrows():
    sqs.send_message(
        QueueUrl=QUEUE_URL,
        MessageBody=json.dumps({
            'product_id': product['id'],
            'product_name': product['name'],
            'category': product['category'],
            'style': product['style']
        })
    )

print(f"Enqueued {len(products)} tasks")
```

Step 4: Lambda Worker Function

```python
import json
import boto3
from datetime import datetime

bedrock = boto3.client('bedrock-runtime')
s3 = boto3.client('s3')
sqs = boto3.client('sqs')

def lambda_handler(event, context):
    """
    Process single SQS message (product)
    Generate description via Bedrock
    Save result to S3
    """

    for record in event['Records']:
        try:
            # Parse message
            body = json.loads(record['Body'])
            product_id = body['product_id']
            product_name = body['product_name']
            category = body['category']
            style = body['style']

            # Generate description
            description = generate_description(
                product_name,
                category,
                style
            )

            # Save to S3
            save_description(product_id, description)

            # Delete from queue (success)
            sqs.delete_message(
                QueueUrl=QUEUE_URL,
                ReceiptHandle=record['receiptHandle']
            )

        except Exception as e:
            print(f"Error processing {product_id}: {e}")
            # Message will reappear in queue after visibility timeout

def generate_description(product_name, category, style):
    """Generate product description"""

    system_prompt = f"""Generate a {style} product description for an e-
commerce site.

Category: {category}
```

```
Product: {product_name}

Requirements:
- 50-100 words
- Highlight key features
- Include call-to-action
- No fake specifications
- Match the {style} tone"""

    response = bedrock.invoke_model(
        modelId='anthropic.claude-3-haiku-20240307-v1:0',  # Cheaper Haiku
model
        contentType='application/json',
        accept='application/json',
        body=json.dumps({
            "anthropic_version": "bedrock-2023-06-01",
            "max_tokens": 256,
            "system": system_prompt,
            "messages": [{
                "role": "user",
                "content": f"Generate description for {product_name}"
            }]
        })
    )

    result = json.loads(response['body'].read())
    return result['content'][0]['text']

def save_description(product_id, description):
    """Save to S3 for review"""
    timestamp = datetime.now().strftime("%Y%m%d-%H%M%S")

    s3.put_object(
        Bucket='generated-descriptions',
        Key=f'pending-review/{product_id}-{timestamp}.txt',
        Body=description,
        Metadata={'product_id': product_id}
    )
```

Step 5: Schedule Daily Execution

```
# Create EventBridge rule to run at 8 PM daily
aws events put-rule \
  --name daily-content-generation \
  --schedule-expression "cron(0 20 * * ? *)"  # 8 PM UTC daily

# Create Lambda target
aws events put-targets \
  --rule daily-content-generation \
  --targets
"Id"="1","Arn"="arn:aws:lambda:...","RoleArn"="arn:aws:iam::...:role/..."
```

Costs

Monthly Cost:

```
Claude 3 Haiku (cheaper model):
- 5,000 descriptions × (50 input + 100 output) tokens
- Input: 5k × 50 × $0.00080 / 1000 = $0.20
- Output: 5k × 100 × $0.0024 / 1000 = $1.20
- Total API cost: ~$4,500/month (only 5k requests!)

Lambda:
- 5,000 invocations × $0.20 per 1M = $0.001

SQS:
- 5,000 messages × $0.40 per 1M = $0.002

S3:
- Storage: ~500 KB = $0.01

Total: ~$4,500/month (way over budget!)

Solution: Use even cheaper model or schedule less frequently
```

Optimization:

```
Option 1: Use Llama 2 7B (self-hosted)
- Cost: $10/month (single T4 GPU)
- Quality: 95% acceptable (vs 99%)
- Setup time: 2 hours

Option 2: Generate 2,500/day instead of 5,000
- Cost: $2,250/month (within budget)
- Trade-off: Longer delivery cycle

Option 3: Use cache for similar products
- Same category products → similar descriptions
- Reduce API calls by 60%
- Cost: $1,800/month
```

Use Case 3: Code Generation Service

Business Context

- **Use Case:** Internal developer tool
- **Users:** 200 developers

- **Frequency:** 50,000 code generation requests/month
- **Real-time:** Required (< 2 seconds)
- **Models:** Code-specific LLMs (Codex)
- **Budget:** $5,000/month

Architecture

Chosen: SageMaker Endpoints (Codex via self-hosted)

Implementation

```python
# API endpoint for code generation
@app.route('/generate-code', methods=['POST'])
def generate_code():
    data = request.json
    language = data.get('language')  # python, javascript, etc
    prompt = data.get('prompt')

    # Route to appropriate model based on language
    response = sagemaker_runtime.invoke_endpoint(
        EndpointName='codex-endpoint',
        ContentType='application/json',
        Body=json.dumps({
            'inputs': f"{language}\n# {prompt}",
            'parameters': {
                'max_new_tokens': 256,
                'temperature': 0.1,  # Low temp for consistent output
                'top_p': 0.95
            }
        })
    )

    result = json.loads(response['Body'].read())
    generated_code = result[0]['generated_text']

    # Save for training feedback loop
    log_code_generation(prompt, generated_code, language)

    return {
        "code": generated_code,
        "language": language
    }
```

Use Case 4: Semantic Search

Business Context

- **Data:** 1 million documents

- **Query:** Find most relevant 10 documents
- **Frequency:** 100,000 searches/month
- **Real-time:** < 1 second required
- **Setup:** One-time embedding generation

Implementation

```python
from vertexai.language_models import TextEmbeddingModel
import numpy as np

# One-time: Generate embeddings for all documents
def generate_embeddings_batch():
    model = TextEmbeddingModel.from_pretrained("textembedding-gecko@001")

    documents = load_documents_from_db()

    for doc in documents:
        embedding = model.get_embeddings([doc['text']])[0].values

        # Store in vector database
        pinecone.upsert([
            (doc['id'], embedding, {"title": doc['title']})
        ])

# At query time: Fast search
@app.route('/search', methods=['POST'])
def search():
    data = request.json
    query = data.get('query')

    # Get query embedding
    model = TextEmbeddingModel.from_pretrained("textembedding-gecko@001")
    query_embedding = model.get_embeddings([query])[0].values

    # Find similar documents
    results = pinecone.query(
        vector=query_embedding,
        top_k=10,
        include_metadata=True
    )

    return [
        {
            "document_id": match['id'],
            "title": match['metadata']['title'],
            "similarity": match['score']
        }
        for match in results['matches']
    ]
```

Use Case 5: On-Premise LLM Deployment (VMware Aria)

Business Context

- **Deployment:** On-premise data center (regulated industry)
- **Use Case:** Document classification and analysis
- **Users:** 500 internal employees
- **Infrastructure:** VMware vSphere + Aria Automation
- **Requirements:** Data sovereignty, compliance (HIPAA/SOC2)
- **Budget:** $30,000 capital + $5,000/month operational
- **Data Security:** All data stays within datacenter

Why On-Premise?

```
Regulatory Requirements:
✓ Healthcare data must stay on-premise
✓ No cloud compliance (HIPAA requires this)
✓ Audit trails needed
✓ Cannot share data with third parties

Cost Considerations:
- Lower per-request costs at scale
- One-time infrastructure investment
- Full control over resources
- No usage surprises

Challenges:
✗ Maintenance responsibility
✗ Scaling manually
✗ Version management
✗ Support burden
```

Architecture Decision

Chosen: VMware Aria Automation + Kubernetes (on vSphere) + vLLM

```
Why:
- Native integration with existing VMware infrastructure
- Kubernetes for orchestration and scaling
- vLLM for efficient model serving (20x faster than transformers)
- Private, secure, no external API calls
- Cost-effective for steady-state workloads
```

Implementation

Step 1: Prepare VMware Aria Environment

```
# 1. Create Kubernetes cluster on vSphere
# Using Tanzu Kubernetes Grid (TKG) or similar

# Prerequisites:
# - vSphere 7.0+
# - 4+ nodes with GPU support (NVIDIA T4 or A100)
# - 500GB+ storage for model weights
# - 100Gbps network fabric

# Create cluster via Aria
aria_cli cluster create \
  --name llm-inference \
  --nodes 4 \
  --worker-vcpu 16 \
  --worker-memory 128 \
  --storage-class vmware-storage
```

Step 2: Set up Kubernetes for LLM Serving

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

---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: model-storage
  namespace: llm-serving
spec:
  accessModes:
    - ReadWriteMany
  storageClassName: vmware-storage
  resources:
    requests:
      storage: 500Gi

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

```yaml
  template:
    metadata:
      labels:
        app: vllm
    spec:
      # Request GPU for this pod
      nodeSelector:
        accelerator: nvidia-gpu

      containers:
      - name: vllm
        image: vllm/vllm-openai:latest

        # GPU resources
        resources:
          requests:
            nvidia.com/gpu: 1
          limits:
            nvidia.com/gpu: 1

        # Model parameters
        env:
        - name: MODEL_NAME
          value: "meta-llama/Llama-2-7b-chat"
        - name: TENSOR_PARALLEL_SIZE
          value: "1"
        - name: GPU_MEMORY_UTILIZATION
          value: "0.9"

        ports:
        - containerPort: 8000
          name: http

        # Mount model storage
        volumeMounts:
        - name: model-storage
          mountPath: /models

        # Health check
        livenessProbe:
          httpGet:
            path: /health
            port: 8000
          initialDelaySeconds: 120
          periodSeconds: 30

        # Startup probe for model loading
        startupProbe:
          httpGet:
            path: /health
            port: 8000
          failureThreshold: 120
          periodSeconds: 10
```

```yaml
      volumes:
      - name: model-storage
        persistentVolumeClaim:
          claimName: model-storage

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

---
# Horizontal Pod Autoscaler
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: vllm-hpa
  namespace: llm-serving
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: vllm-server
  minReplicas: 2
  maxReplicas: 4
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 70
```

Step 3: Deploy via Aria Automation

```bash
# Apply the deployment
kubectl apply -f llm-deployment.yaml

# Verify deployment
kubectl get pods -n llm-serving
```

```
# Get service endpoint
kubectl get svc vllm-service -n llm-serving
```

Step 4: Create Flask API Gateway

```
from flask import Flask, request, jsonify
import requests
import json
from datetime import datetime
import logging

app = Flask(__name__)

# Internal vLLM cluster endpoint (no internet access needed)
VLLM_ENDPOINT = "http://vllm-service.llm-serving.svc.cluster.local:8000"

# Configure logging (audit trail for compliance)
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',
    handlers=[
        logging.FileHandler('/var/log/llm-api/audit.log'),
        logging.StreamHandler()
    ]
)
logger = logging.getLogger(__name__)

@app.route('/api/v1/classify-document', methods=['POST'])
def classify_document():
    """
    Internal API for document classification
    All data stays on-premise
    """

    try:
        data = request.json
        document_id = data.get('document_id')
        document_text = data.get('text')
        user_id = data.get('user_id')

        # Audit log
        logger.info(f"Classification request - User: {user_id}, Doc:
{document_id}")

        # Call vLLM (local, private)
        response = requests.post(
            f"{VLLM_ENDPOINT}/v1/completions",
            json={
                "model": "meta-llama/Llama-2-7b-chat",
                "prompt": f"""Classify this document into one of these
categories:
```

```
                   - CLINICAL_NOTES
                   - BILLING
                   - AUTHORIZATION
                   - TEST_RESULTS
                   - ADMINISTRATIVE

Document:
{document_text}

Classification: """,
                   "max_tokens": 50,
                   "temperature": 0.1,  # Low temp for consistent output
                   "top_p": 0.95
               },
               timeout=30
           )

           result = response.json()
           classification = result['choices'][0]['text'].strip()

           # Store result in on-premise database
           store_classification(
               document_id,
               classification,
               user_id,
               datetime.now()
           )

           logger.info(f"Classification complete - Doc: {document_id}, Class:
{classification}")

           return {
               "document_id": document_id,
               "classification": classification,
               "confidence": 0.95,
               "timestamp": datetime.now().isoformat()
           }

       except Exception as e:
           logger.error(f"Classification error: {str(e)}")
           return {"error": "Classification failed"}, 500

@app.route('/api/v1/health', methods=['GET'])
def health():
    """Health check endpoint for monitoring"""
    try:
        # Check vLLM is responsive
        response = requests.get(
            f"{VLLM_ENDPOINT}/health",
            timeout=5
        )

        return {
```

```
            "status": "healthy",
            "vllm": response.status_code == 200,
            "database": check_database_connection(),
            "timestamp": datetime.now().isoformat()
        }
    except Exception as e:
        logger.error(f"Health check failed: {str(e)}")
        return {"status": "unhealthy", "error": str(e)}, 500

def store_classification(document_id, classification, user_id, timestamp):
    """Store in on-premise database"""
    # Example: PostgreSQL on-premise
    import psycopg2

    try:
        conn = psycopg2.connect(
            host="db.internal",
            database="llm_classifications",
            user="api_user",
            password="secure_password"  # Use secrets management
        )

        cursor = conn.cursor()
        cursor.execute("""
            INSERT INTO classifications
            (document_id, classification, user_id, timestamp)
            VALUES (%s, %s, %s, %s)
        """, (document_id, classification, user_id, timestamp))

        conn.commit()
        cursor.close()
        conn.close()

    except Exception as e:
        logger.error(f"Database error: {str(e)}")
        raise

if __name__ == '__main__':
    # Run on internal network only
    app.run(host='0.0.0.0', port=5000, ssl_context='adhoc')
```

Step 5: Configure Aria for Monitoring and Auto-scaling

```
# File: aria-monitoring.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: vllm-monitoring
  namespace: llm-serving
data:
  prometheus-rules.yml: |
```

```yaml
    groups:
    - name: vllm
      interval: 30s
      rules:
      # Alert on high latency
      - alert: HighLLMLatency
        expr: rate(vllm_request_duration_seconds_sum[5m]) > 2
        for: 5m
        annotations:
          summary: "vLLM latency high"

      # Alert on GPU memory pressure
      - alert: HighGPUMemory
        expr: nvidia_smi_memory_used_mb / nvidia_smi_memory_total_mb > 0.9
        for: 2m
        annotations:
          summary: "GPU memory usage critical"

      # Alert on error rate
      - alert: HighErrorRate
        expr: rate(vllm_request_errors_total[5m]) > 0.05
        for: 5m
        annotations:
          summary: "vLLM error rate high"
```

Step 6: Set up Disaster Recovery

```bash
# Backup strategy for on-premise
# 1. Daily snapshots of model storage

snapshot_schedule=$(cat <<EOF
{
  "schedule": "0 2 * * *",  # 2 AM daily
  "retention_days": 30,
  "storage": "/backups/models"
}
EOF
)

# 2. Database backup
pg_dump -h db.internal -U api_user -d llm_classifications \
  > /backups/db/llm_db_$(date +%Y%m%d).sql

# 3. Cross-site replication (for HA)
# Set up replication to secondary VMware site
vmware-aria backup create \
  --name llm-cluster-backup \
  --type incremental \
  --target secondary-site
```

Costs

Capital Expenses (One-time):

```
Hardware:
- 4x Server nodes (16-core, 128GB RAM): $60,000
  (each: $15,000)
- 4x GPU cards (NVIDIA A100): $40,000
  (each: $10,000)
- Storage system (500TB SAS): $20,000
- Networking equipment: $15,000
---
Total Hardware: $135,000

Software:
- VMware vSphere Enterprise+: $10,000
- VMware Aria Automation: $5,000
- Kubernetes licensing: Included (open-source)
---
Total Software: $15,000

**Total Capital: $150,000** (one-time)
```

Operational Expenses (Monthly):

```
Personnel:
- Infrastructure engineer (40% time): $2,000
- System administrator (30% time): $1,500
- Backup/DR oversight: $500

Power & Cooling:
- 4 servers + 4 GPUs: ~15kW avg
- 15kW × 30 days × 24 hrs = 10,800 kWh
- @ $0.12/kWh = $1,296

Maintenance:
- Hardware support: $500
- Software licenses: $300

**Total Monthly: ~$6,096**

Total 3-year cost:
$150,000 (capital) + ($6,096 × 36 months) = $369,456

Per-inference cost (@ 1M inferences/month):
$6,096 / 1M = $0.006 per inference

Compare to cloud: $0.01+ per inference
Savings: 40-60% over 3 years
```

Monitoring in Aria

Key Metrics for On-Premise:

```python
# Aria monitoring integration
import requests

def send_aria_metrics(metric_name, value, tags):
    """Send metrics to Aria Automation"""

    aria_endpoint = "http://aria-collector.internal/api/metrics"

    payload = {
        "metric": metric_name,
        "value": value,
        "tags": tags,
        "timestamp": datetime.now().isoformat()
    }

    requests.post(aria_endpoint, json=payload)

# Example usage
send_aria_metrics(
    "vllm_inference_latency_ms",
    125.5,
    {"service": "document-classification", "model": "llama2-7b"}
)

send_aria_metrics(
    "gpu_memory_utilization_percent",
    87.3,
    {"node": "k8s-worker-1", "gpu": "0"}
)
```

Monitoring Dashboard (Grafana):

```
Panels:
- vLLM active requests (real-time)
- GPU utilization per node
- Model inference latency (P50, P95, P99)
- Error rate and types
- Queue depth
- Power consumption
- Network bandwidth
- Storage usage
```

Security & Compliance

Data Sovereignty:

```
✓ All data stays on-premise
✓ No external API calls
✓ Audit logs for compliance
✓ Network isolation (air-gapped possible)
✓ Encryption at rest and in transit
```

Security Implementation:

```python
# File: security.py
import ssl
import hashlib

# Enable mTLS for service-to-service communication
ssl_context = ssl.create_default_context()
ssl_context.load_cert_chain(
    certfile="/secrets/api-server.crt",
    keyfile="/secrets/api-server.key"
)
ssl_context.load_verify_locations("/secrets/ca.crt")

# Audit logging (HIPAA requirement)
def audit_log(user_id, action, document_id, result):
    """Log all access for compliance"""
    log_entry = {
        "timestamp": datetime.now().isoformat(),
        "user_id": user_id,
        "action": action,
        "document_id": document_id,
        "result": result,
        "ip": request.remote_addr,
        "hash": hashlib.sha256(
            f"{user_id}{action}{document_id}".encode()
        ).hexdigest()
    }

    # Write to tamper-proof log
    with open("/secure/audit.log", "a") as f:
        f.write(json.dumps(log_entry) + "\n")
```

Operational Considerations

Pros:

- ☑ Full data control (HIPAA/regulated industries)
- ☑ Lower cost at scale (1M+ inferences/month)
- ☑ No external dependencies

- ☑ Full customization possible
- ☑ Predictable costs

Cons:

- ❌ Maintenance burden (patching, updates)
- ❌ Scaling requires physical hardware
- ❌ Initial capital investment large
- ❌ Need dedicated ops team
- ❌ Manual backup/DR management

When to Choose On-Premise:

```
✓ Regulated data (healthcare, finance, govt)
✓ High volume (>1M inferences/month)
✓ 3+ year ROI horizon
✓ Data sovereignty critical
✓ Existing datacenter infrastructure
✓ Compliance audits required

✗ Avoid if:
- Unpredictable volume (use cloud)
- No ops team (staffing cost)
- Need rapid scaling
- Budget constraints (high capex)
```

Quick Reference: Choosing Deployment for Your Use Case

Use Case	Recommended	Reason
Chatbot	AWS Bedrock	Real-time, easy scaling
Batch Processing	AWS Batch	Cost-effective for high volume
Code Generation	SageMaker	Specialized models
Content Generation	SageMaker + Lambda	Scheduled, high volume
Search	Vertex AI (embeddings)	Vector database integration
Internal Tools	Self-hosted EC2	Full control, cost-effective
Rapid Prototyping	Azure OpenAI	Quick setup, good models
Multi-region	GKE	Global distribution
On-Premise/Regulated	VMware Aria + K8s	Data sovereignty, compliance
Healthcare/Finance	VMware Aria + K8s	HIPAA/SOC2 compliance
High Volume (>1M/mo)	VMware Aria + K8s	Best cost per inference

---

Key Takeaway: Match the deployment pattern to your use case's requirements (real-time vs batch, scale, cost, data location, and compliance needs).