Production Deployment & Scaling Design for Sentiment Analysis Service

(FastAPI + local Hugging Face models, beginner-friendly)

1) Overview

Objective: Run the sentiment analysis project API in production with cheap costs and easy operations, even when there are a lot of people during the day and a few at night.

Strategy:

- After finishing the project, we package it into a docker image
- Then a small VM can be used with multiple Uvicorn/Gunicorn workers to make use of the parallel requests to use all CPU cores: we can say that all servers even the cheap ones have multi core CPU's so to make use of all cores we use a process manger like Gunicorn to launch multiple worker processes each serving requests in parallel.
- Now for heavy hours "peak hours" we can use an identical VM with the same process that can be shut down at night or when the load is lighter os the project deployment can be cost effective.
- Support request batching via the existing texts: [...] input to improve sending one text at a time to the model
- Optionally we can introduce a small GPU VM later if latency or volume demands increase:
  Why: GPUs run the same model faster (more parallelism).

2) Assumptions

- **Traffic (~300k summaries/day):** Means we expect a lot of requests generally, but not all at once. The number helps size the system and estimate cost.
- **Peak window (~10 daytime hours → ~6–8 requests/second at busiest):** Most traffic arrives during the day. At the peak moments we estimate almost 6–8 inquiries every second. This tells us when to install the second VM and what capacity we require during business hours.
- **Input size (300–600 tokens per summary "the chunk"):** token is a chunk of text (part of a word). More tokens = more computing time. The code already handles huge inputs, so runtime stays predictable.

- **Latency target (p95 ≈ 0.5–1.5 seconds):** 95% of requests should finish within 0.5–1.5s. This is good enough for backoffice tools. If p95 goes higher, we need to introduce (more workers/VM, or add a small GPU VM).
- **Validation:** These are starting points. A short load test will confirm real numbers, and we'll adjust worker count/VM size accordingly.

3) Deployment Plan

1. Containerization: We are going to package the FastAPI project into a Docker image.

- We include requirements.txt, project files, and downloader.py.
- During build (or container start), we run downloader.py so /models is ready.
- We then run the container locally and test it on port 8000.

---

2. Single VM Baseline (Night Time): We are going to start with one Linux VM (example: 4 vCPUs, 8–16 GB RAM).

- We install Docker and run our container there.
- We expose port 8000 internally.
- We connect it to a cloud load balancer, so public traffic comes in safely (HTTPS + health checks).

---

3. Daytime Capacity (Extra VM): During the day when traffic is higher, we are going to add a second VM.

- Both VMs run the same container.
- Both are registered behind the load balancer.
- The load balancer checks /health to make sure both are alive.

This doubles our serving power during busy hours.

---

4. Multi-Worker Server: Inside each VM, we are going to run Gunicorn + Uvicorn workers so we can use all CPU cores.

- One worker = one CPU core.
- On a 4-core VM we run 4 workers.

Command we use:

gunicorn api:app -k uvicorn.workers.UvicornWorker -w 4 -b 0.0.0.0:8000 --timeout 60

This way, each VM handles more requests in parallel.

---

5. Monitoring & Alerts: We are going to keep an eye on 4 simple things:

- p95 latency (how long it takes for 95% of requests → must stay <1.5s)
- Error rate (should stay low, under 2–5%)
- CPU usage (%)
- Memory usage (%)

Alerts are set for:

- Latency above target for several minutes
- Rising error rate
- VM health check failure

---

6. Day/Night Scheduling: We are going to run 2 VMs during the day and scale down to 1 VM at night.

- Morning → Start VM #2 and attach it to the load balancer.
- Evening → Drain traffic from VM #2 and stop it.

Later, we can automate this with a simple scheduler if manual switching becomes boring.

---

4) Scaling Strategy

Vertical scaling (inside one VM)

- Increase the number of Gunicorn workers to use all CPU cores.
- Consider --threads 1–2 per worker if I/O becomes the limiter.
- Watch RAM: three models load into memory; budget several GB.

Horizontal scaling (across VMs)

- Nighttime: one VM serves all traffic.
- Daytime: two VMs share traffic via the load balancer.
- The second VM can use a spot/preemptible instance to reduce cost; the primary VM remains on-demand.

Optional GPU path

- If latency or peak throughput requires acceleration, we can add one small GPU VM (e.g., T4/L4) and run inference there.
- The FastAPI process can still run on CPU; inference requests route to the GPU instance internally.

---

5) Cost Measures

- One VM at night, two VMs during peak hours.
- Right-sized VM type (start with 4 vCPU, 8–16 GB RAM; adjust from measurements).
- Spot/preemptible for the daytime VM to cut cost.
- Batch requests via texts: [...] when upstream can aggregate a few items per call.

Action	When	Why it saves cost
Run **1 VM at night**, **2 VMs at peak**	Night vs. daytime	We are not paying for idle capacity at night
Right-size VMs (start with **4 vCPU, 8–16 GB RAM**)	Initial launch; revisit monthly	Avoid overpaying for unused CPU/RAM
Use **Spot/Preemptible** for daytime VM: Amazon EC2 Spot Instances let you take advantage of unused EC2 capacity in the AWS cloud and are available at up to a 90% **(amazon, n.d.)**	Daytime peaks	cheaper than on-demand for extra capacity
Batch requests via `texts: [...]`	Whenever clients can group items (8–32)	Fewer requests = less overhead per result; higher throughput per VM

---

6) Security

- API key in headers (per tenant if required).
- TLS termination at the load balancer (or Nginx).
- Basic rate limiting at the LB or Nginx.
- Input validation: strict JSON schema and maximum text length to protect CPU and memory.
- Data handling: avoid storing raw summaries unless necessary; if stored, apply PII controls and retention (for example, 30 days).

---

7) Minimal Runbook

1. Build image: docker build -t sentiment-api:latest .
2. Populate models during image build or container start (/models).
3. Start app:
4. gunicorn api:app -k uvicorn.workers.UvicornWorker -w 4 -b 0.0.0.0:8000 --timeout 60
5. Place the VM behind a cloud load balancer; health-check /health.
6. Daytime: start VM #2 and register in the load balancer.
7. Night: drain and stop VM #2.
8. Monitor p95 latency, error rate, CPU %, memory.
9. Blue/green: deploy new image to VM #2, verify, switch traffic, recycle VM #1.

---

8) Reasoning Behind Decisions

- The plan mirrors the existing FastAPI code and local model usage to reduce complexity.
- Multi-worker Gunicorn/Uvicorn provides parallelism without new frameworks.
- Two small VMs with a load balancer handle the day/night pattern and enable safe rollouts.

- Batching via the current JSON interface improves throughput without architectural changes.
- A small GPU node stays optional until measurements justify it.

9) Architecture drawing