Modal.com Üzerinde Llama.cpp (GGUF) ile FastAPI Endpoint Dağıtımı – Teknik Rapor

Özet

Bu rapor, verilen app_modal.py kodunu inceleyerek Modal.com üzerinde GGUF formatlı bir LLM'in llama-cpp-python kütüphanesi ile nasıl servis edildiğini, kaynak (CPU/RAM) rezervasyonlarını, kuyruk/çağrı modelini, yaşam döngüsü (lifecycle) akışını ve ortaya çıkan web endpoint'ini detaylandırır. Sonuçta, Modal üzerinde FastAPI tabanlı bir /generate uç noktası oluşturulmuştur.

Modal.com Nedir?

Modal; Python fonksiyonlarını konteyner içinde çalıştırmaya, kaynak ayırmaya, kalıcı hacimler (Volume) kullanmaya ve bu fonksiyonları web endpoint'lerine dönüştürmeye yarayan sunucusuz (serverless) bir çalışma platformudur. Otomatik ölçeklenme ve sınıf tabanlı yaşam döngüsü kancaları (enter/exit) gibi özellikler sağlar.

Kodun Yaptıkları – Genel Bakış

1) Bir Modal uygulaması tanımlanır: app = modal.App("link-cloud-llamacpp").

2) Çalışma imajı oluşturulur: Image.debian_slim() + build-essential + FastAPI + pydantic + llama-cpp-python (0.3.5).

3) GGUF model dosyalarını barındırmak için bir Volume bağlanır: Volume.from_name("gguf-models").

4) Kaynak parametreleri çevre değişkenlerinden okunur (CPU_CORES, MEM_MB, LLM_THREADS, LLM_CTX, LLM_N_BATCH).

5) LlamaWorker sınıfı @app.cls ile tanımlanır; cpu=CPU_CORES ve memory=MEM_MB ayırır; /models altına Volume mount edilir. @modal.enter ile model tek seferde yüklenir (container başlarken). @modal.method ile generate(message) uzaktan çağrılabilir bir Modal Function'a dönüştürülür.

6) FastAPI uygulaması tanımlanır ve @modal.asgi_app ile bir Modal Function olarak web'e açılır. /generate endpoint'i POST gövdesindeki message alanını alır, bir LlamaWorker örneği üzerinden generate.remote(message) çağırıp sonucu JSON olarak döndürür.

Kaynak Kullanımı ve Rezervasyonlar

• LlamaWorker (@app.cls): cpu=CPU_CORES, memory=MEM_MB. Varsayılanlar kodda CPU_CORES=8, MEM_MB=16384 (MiB), yani 8 vCPU ve ~16 GiB RAM rezerve edilir. Bu sınıf içindeki llama.cpp modeli ayrıca n_threads=THREADS (varsayılan 8), n_ctx=CTX (varsayılan 8192) ve n_batch=N_BATCH (varsayılan 256) ile yapılandırılır.

• FastAPI ASGI Function (@app.function): cpu=0.25, memory=512. Bu, sadece HTTP isteklerini almak ve LlamaWorker.method() çağrısını tetiklemek için hafif bir kap; asıl ağır iş LlamaWorker'da yapılır.

• Toplam: Her aktif LlamaWorker konteyneri başına ~8 vCPU / 16 GiB ayrılır. Modal gerekli olduğunda ek kopyalar (replica) başlatabilir; ölçekleme politikaları ayrıca konfigüre edilebilir.

Kuyruklama ve Eşzamanlılık Modeli

generate_endpoint içinde worker.generate.remote(message) çağrısı, Modal'ın yönetimli bir kuyruklama ve uzaktan çağrı (RPC) katmanını kullanır. İstekler ASGI fonksiyonuna gelir, burada LlamaWorker.generate fonksiyonuna yönlendirilir. Modal, aynı sınıfın birden fazla konteynerini gerektiğinde ayağa kaldırabilir (autoscaling). Model yükü @modal.enter ile konteyner başına bir kez yapılır, böylece ardışık çağrılar sıcak konteynerlerde daha hızlıdır.

Depolama: Volume Kullanımı

gguf_vol = modal.Volume.from_name("gguf-models") satırı ile kalıcı bir Volume bağlanır ve LlamaWorker sınıfında volumes={"/models": gguf_vol} şeklinde mount edilir. MODEL_PATH "/models/models/model_q4_0.gguf" olarak tanımlıdır, yani model dosyası Volume içinde beklenir. Volume, model ağırlıklarını bir kez yükleyip birden çok konteyner arasında paylaşmaya uygundur.

Model Sunumu: llama-cpp-python Neden?

llama-cpp-python, C tabanlı llama.cpp motorunun Python bağlayıcısıdır. GGUF formatlı modelleri doğrudan, GPU gerektirmeden (yalnızca CPU ile) veya uygun derleme/kurulumla GPU hızlandırmalı olarak çalıştırabilir. Bu kodda, Modal imajına 'llama-cpp-python==0.3.5' kurulmuş ve Llama(...) ile aşağıdaki temel parametreler kullanılmıştır:

• n_threads: CPU paralelliğini kontrol eder (varsayılan 8).

• n_ctx: Bağlam penceresi; prompt+cevap toplam token kapasitesini etkiler (varsayılan 8192).

• n_batch: Token üretimde (veya prompt işleme) iç parti büyüklüğü; throughput'u etkiler (varsayılan 256).

• model_path: GGUF dosya yolu.

Bu yaklaşımın avantajı, GGUF'nin hafif ve hızlı yüklenebilir olması ve llama.cpp'nin düşük bellek ayak izi ile CPU tabanlı servis için uygunluğudur.

İstek Akışı (Request Flow)

1) İstemci POST /generate ile {"message": "..."} gönderir.

2) ASGI fonksiyonu isteği alır ve body'yi GenReq ile doğrular.

3) LlamaWorker.generate.remote(body.message) çağrılır.

4) LlamaWorker konteynerinde, @modal.enter ile önceden yüklenmiş Llama nesnesi kullanılarak _run() çağrılır. Sistem prompt'u + kullanıcı mesajı ile inference yapılır.

5) Üretilen metin FastAPI üzerinden JSON {"response": "..."} olarak döner.

Güvenlik ve Bakım Notları

• Volume içeriğine doğru model dosyasının yüklendiğinden emin olun. Aksi halde FileNotFoundError fırlatılır.

• ASGI uç noktasına rate limit, auth ve logging eklemeniz önerilir.

• Model parametreleri (n_ctx, n_threads, n_batch) aşırı büyütülürse bellek kullanımı artar. Modal'da memory hataları konteyner yeniden başlatmalarına yol açabilir.

• Üretim ortamında timeout, geri alma (retry) ve gözlemlenebilirlik (metrics, tracing) eklemek iyi bir pratiktir.

Sonuç

Bu kurulumla Modal üzerinde, GGUF formatlı bir LLM'i llama-cpp-python aracılığıyla çalıştıran ve FastAPI ile HTTP üzerinden erişilebilen bir /generate endpoint'i oluşturulmuştur. LlamaWorker konteyneri başına ~8 vCPU ve ~16 GiB RAM rezerve edilmiştir; ASGI katmanı hafif tutulmuştur (0.25 vCPU / 512 MiB). Kuyruklama ve autoscaling Modal tarafından yönetilir; model yükü konteyner başına bir kez yapılır.

Kaynakça

• Modal – Image referansı ve container imajları: https://modal.com/docs/reference/modal.Image

• Modal – Images rehberi: https://modal.com/docs/guide/images

• Modal – Volumes rehberi: https://modal.com/docs/guide/volumes

• Modal – Volume referansı: https://modal.com/docs/reference/modal.Volume

• Modal – Kaynak ayırma (CPU/RAM): https://modal.com/docs/guide/resources

• Modal – Lifecycle (enter/exit) ve @app.cls: https://modal.com/docs/guide/lifecycle-functions

• Modal – @modal.method referansı: https://modal.com/docs/reference/modal.method

• Modal – ASGI uygulamaları (@modal.asgi_app): https://modal.com/docs/reference/modal.asgi_app

• Modal – Web endpoint’leri genel: https://modal.com/docs/guide/webhooks

• llama-cpp-python API referansı ve parametreler: https://llama-cpp-python.readthedocs.io/en/latest/api-reference/

• GGUF biçimi üzerine açıklamalar: https://apxml.com/courses/practical-llm-quantization/chapter-5-quantization-formats-tooling/gguf-format

• GGUF (genel, bilgilendirici yazı): https://medium.com/@charles.vissol/gguf-in-details-8a9953ac7883

Performans Testi – Locust (100 Kullanıcı, 1 Saniye Aralıklarla)

Test Senaryosu:

- Locust ile 100 sanal kullanıcı, her biri 1-3 saniye bekleme süresi ile `/generate` endpoint'ine POST isteği göndermiştir.

- Başlangıçta kullanıcı sayısı 0’dan 100’e lineer olarak artmıştır.

- Her istek gövdesinde `{"message": "link cloud avantajları nelerdir"}` yer almıştır.

Ölçülen Sonuçlar:

- Ortalama Yanıt Süresi: ~6452 ms

- Minimum: 3774 ms, Maksimum: 19232 ms

- 50. yüzdelik (Median): 5700 ms, 95. yüzdelik: 11000-12000 ms arası

- Toplam İstek: 370

- Hata Oranı: %0 (hiç başarısız istek yok)

- RPS (Requests per Second): Maksimum ~9 RPS seviyesine ulaşılmıştır.

Grafik Yorumları:

- Toplam İstek/Saniye grafiğinde RPS, kullanıcı sayısı arttıkça lineer olarak artmış ve ~9 civarında dengelenmiştir.

- Yanıt süreleri, kullanıcı sayısı arttığında ilk başta yükselmiş, ardından kısmen stabil olmuştur.

- Kuyruk etkisi gözlemlenmiştir; 95. yüzdelikte yüksek tail latency (~19 sn) mevcuttur.

Performans Testi – Lokal Script (50 İstek, 1 Saniye Aralıklarla)

Test Senaryosu:

- Lokal Python script ile `50-questions.jsonl` dosyasından alınan 50 farklı soru, 1 saniye aralıklarla `/generate` endpoint'ine gönderilmiştir.

- Her istek JSON formatında gönderilmiş ve model yanıtları ekrana yazdırılmıştır.

Ölçülen Sonuçlar:

- Ortalama Yanıt Süresi (son 4 istekte görülen değerler örnek): ~4-6 sn aralığında

- Minimum gözlemlenen: ~3362 ms, Maksimum gözlemlenen: ~5911 ms (son isteklerde)

- Toplam Süre: 271.26 saniye (50 istek)

- Hata Oranı: %0

- Yanıt içerikleri tutarlı ve formatlı JSON olarak gelmiştir.

Karşılaştırma ve Değerlendirme

1) **Kullanıcı Yükü**: Locust testinde aynı anda 100 kullanıcı çalışırken, lokal testte tek thread ile ardışık 50 istek atılmıştır.

2) **Yanıt Süresi**: 100 kullanıcı yükünde ortalama yanıt süresi (~6.45 sn) lokal testin ortalama süresinden biraz yüksektir; ayrıca Locust testinde tail latency (~19 sn) görülmüştür.

3) **RPS**: Locust testinde eşzamanlı kullanıcılar sayesinde ~9 RPS seviyesine ulaşılmıştır; lokal testte eşzamanlılık olmadığından RPS düşüktür.

4) **Kaynak Kullanımı Etkisi**: Modal konteyneri tek LlamaWorker ile çalıştığından, 100 kullanıcı yükünde istekler sıraya girmiş ve bazı isteklerde yüksek gecikme yaşanmıştır.

5) **Tutarlılık**: Her iki testte de hata oranı %0'dır ve tüm yanıtlar modelin beklenen formatında dönmüştür.

Sonuç: Tek konteynerli (8 vCPU / 16 GiB RAM) yapı, 50 ardışık isteği düşük gecikmeyle işleyebilirken; yüksek eşzamanlı yükte (100 kullanıcı) kuyruklama sebebiyle gecikmeler artmıştır. Ölçeklenebilirlik için Modal üzerinde ek replica başlatılması (autoscaling) önerilir.