به نام خدا

دانشگاه شهید بهشتی

گزارش پیشرفت فاز سوم پروژه معماری نرم‌افزار: معماری سیستم‌های هوش مصنوعی

استاد راهنما درس: دکتر صادق علی اکبری

نام اعضا(تک نفره) : حسن رکن آبادی – شماره دانشجویی : ۴۰۳۴۴۳۲۰۴

خرداد ۱۴۰۴

فهرست مطالب

بخش ۱: تحلیل معماری سیستم‌های استنتاج هوش مصنوعی (مرور ادبیات)

۱.۱ مقدمه: از مدل تا سرویس؛ چالش‌های معماری در استقرار هوش مصنوعی

در سال‌های اخیر، کانون توجه در حوزه هوش مصنوعی از چالش صرفِ «ساخت مدل» به چالش پیچیده‌تر «عملیاتی‌سازی و استقرار مدل» تغییر یافته است. امروزه، ساختن یک مدل با دقت بالا، اگرچه دستاوردی مهم است، اما تنها نقطه شروع یک مسیر مهندسی محسوب می‌شود. چالش اصلی، تبدیل این مدل ایستا به یک سرویس نرم‌افزاری زنده، پایدار و کارآمد است که بتواند در مقیاس بزرگ به کاربران خدمت‌رسانی کند. این گذار، یک مسئله عمیق در حوزه معماری نرم‌افزار است، زیرا محصول نهایی باید مجموعه‌ای از مشخصه‌های کیفی بنیادین را برآورده سازد.

از مهم‌ترین این مشخصه‌ها می‌توان به کارایی اشاره کرد که خود به دو معیار کلیدی تقسیم می‌شود: زمان پاسخ‌دهی که نشان‌دهنده سرعت سیستم در پاسخ به یک درخواست منفرد است و توان عملیاتی که ظرفیت پردازش سیستم (مثلاً تعداد تصاویر پردازش‌شده در ثانیه) در یک بازه زمانی مشخص را معین می‌کند. در کنار کارایی، مقیاس‌پذیری[1] قرار دارد که به توانایی معماری در مدیریت بهینه بارهای کاری رو به رشد و افزایش تعداد کاربران همزمان اشاره دارد. همچنین، بهینگی در مصرف منابع ، به‌ویژه در استفاده از پردازنده‌های گرافیکی (GPU) گران‌قیمت، نقشی حیاتی در کنترل هزینه‌های عملیاتی ایفا می‌کند؛ به‌طوری‌که یک معماری بهینه می‌تواند هزینه پردازش یک حجم کاری مشخص را تا ده‌ها برابر کاهش دهد. در نهایت، قابلیت اصلاح و نگهداری تضمین می‌کند که سیستم بتواند به سادگی با مدل‌های جدید به‌روزرسانی شده و در طول زمان تکامل یابد[۴]. این بخش از گزارش، با بررسی عمیق پژوهش‌های اخیر، به تحلیل الگوهای معماری، تصمیمات کلیدی و مصالحه‌های (Trade-offs) موجود در طراحی سیستم‌های استنتاج می‌پردازد.

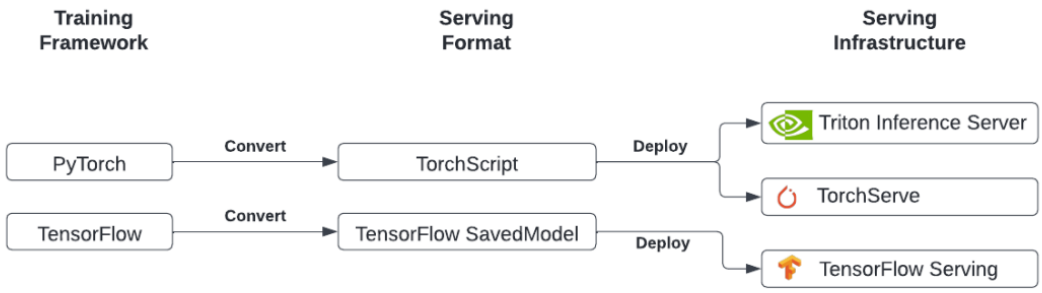
کلمات کلیدی : معماری سیستم‌های هوش مصنوعی - معماری استقرار - مقیاس‌پذیری - توان عملیاتی

[1] Scalability

۱.۲ الگوهای معماری و تصمیمات کلیدی در انتخاب سکوی استقرار(رویکرد پژوهشی)

در میان راهکارهای گوناگون، یک الگوی معماری غالب با عنوان «استنتاج به عنوان سرویس»[۲] وجود دارد. این الگو که نمونه‌ای تخصصی از معماری سرویس‌گرا است، بر اصل جداسازی[۳] منطق سنگین و محاسباتی استنتاج از برنامه‌های کاربردی اصلی تأکید دارد. در این ساختار، سرورهایی متمرکز و مجهز به GPU، وظیفه اجرای مدل‌ها را بر عهده گرفته و سایر بخش‌های نرم‌افزار از طریق یک رابط برنامه‌نویسی کاربردی (API) تحت شبکه با آن‌ها ارتباط برقرار می‌کنند. یک پژوهش بسیار روشنگر در این زمینه، کاربرد این الگو را در مراکز محاسبات علمی اشتراکی، که اغلب با کمبود منابع GPU مواجه هستند، مورد مطالعه قرار داده است. نتایج نشان داد که استقرار یک سرور مرکزی مبتنی بر سکوی Nvidia Triton، زمان مورد نیاز برای تحلیل داده‌ها با مدل‌های پیچیده شبکه‌های عصبی گراف (GNN) را تا پنجاه برابر کاهش داده است[۸]. این خود گواهی بر قدرت این الگو در متمرکزسازی و بهینه‌سازی منابع است.

همچنین، برای حل چالش سازگاری محیط‌ها و مدیریت وابستگی‌ها، مقالات پژوهشی به طور گسترده بر نقش محوری فناوری کانتینرسازی با ابزار داکر[۴] تأکید ورزیده‌اند. این فناوری به معمار سیستم اجازه می‌دهد تا هر جزء از سامانه (شامل مدل، سرور استنتاج و کتابخانه‌های مورد نیاز) را در یک محیط ایزوله و قابل تکرار به نام کانتینر بسته‌بندی کند که برای انجام آزمون‌های مقایسه‌ای معتبر، امری ضروری است[۱۰].



(دیاگرام ۱: معماری مقایسه‌ای سکوهای استقرار)

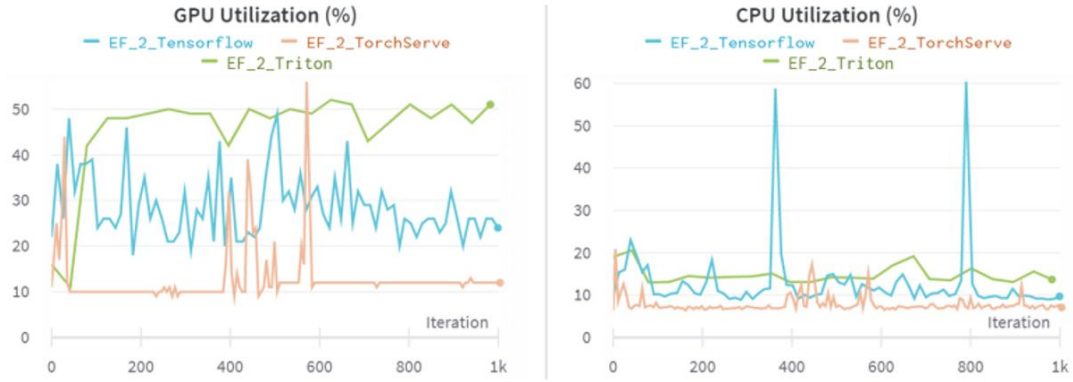
توضیح دیاگرام ۱ : این دیاگرام باید به صورت بصری، مسیر یک مدل را از چارچوب آموزشی اولیه آن (مانند PyTorch یا TensorFlow) تا استقرار نهایی روی سه سکوی اصلی (Triton Inference Server، TensorFlow Serving، و TorchServe) نمایش میدهد. هدف این تصویر، ارائه یک نمای کلی و مقایسه‌ای از اکوسیستم‌ها و جریان‌های کاری مختلف است تا به درک بهتر تفاوت‌های ساختاری آن‌ها کمک کند.

انتخاب سکوی استقرار، یک تصمیم معماری بنیادین با بده‌بستان‌های مشخص است:

- Nvidia Triton Inference Server: با معماری چند-فریم‌ورکی خود که برای دستیابی به حداکثر کارایی و توان عملیاتی طراحی شده، قدرتمندترین گزینه در محیط‌های مبتنی بر GPU شناخته می‌شود. اما این کارایی به قیمت پیچیدگی بیشتر در راه‌اندازی (نیاز به درایورهای خاص انویدیا و Nvidia Container Toolkit) و نیاز به یک دوره «گرم شدن»[۵] طولانی‌تر برای رسیدن به اوج عملکرد همراه است.

[2] Inference-as-a-Service
[3] Decoupling
[4] Docker
[5] Warmup

- TensorFlow Serving (TF Serving): برای سادگی و یکپارچگی با اکوسیستم TensorFlow طراحی شده و راه‌اندازی آن ساده‌تر است. اما یک بده‌بستان معماری در آن وجود دارد: قابلیت‌های پیش‌پردازش و پس‌پردازش به صورت داخلی تعبیه نشده‌اند و منطق آن باید در سمت کلاینت پیاده‌سازی شود که این امر معماری کلاینت را پیچیده‌تر می‌کند.
- TorchServe: این سکو که برای اکوسیستم PyTorch طراحی شده، با ارائه قابلیت بسته‌بندی کدهای پیش‌پردازش در قالب یک «پردازشگر» (Handler) در کنار مدل، معماری سمت کلاینت را به مراتب ساده‌تر می کند. با این حال، این سادگی به قیمت فدا کردن بخشی از کارایی تمام می‌شود و آزمون‌های مقایسه‌ای نشان داده‌اند که در توان عملیاتی نمی‌تواند با Triton رقابت کند.



(نمودار ۲: مقایسه مصرف منابع و کارایی)

توضیح نمودار ۲ : این نمودار باید نتایج یک آزمون مقایسه‌ای را در خصوص مصرف منابع (CPU و GPU) و توان عملیاتی سه سکوی مذکور نمایش دهد. این مقایسه بصری به درک بهتر مصالحه میان مصرف منابع و کارایی کمک می‌کند و نشان می‌دهد که چگونه سکوی Triton با بهره‌برداری بسیار بیشتر از GPU (تا ٪۵۰ utilization)، به توان عملیاتی بالاتری نسبت به دو سکوی دیگر (که utilization کمتری دارند) دست می‌یابد.

۱.۳ اصول طراحی برای بهینه‌سازی در سطح مدل و سیستم

پس از انتخاب سکو، معمار سیستم می‌تواند از مجموعه‌ای از اصول و تاکتیک‌های طراحی برای بهینه‌سازی بیشتر بهره گیرد. یک الگوی طراحی مشترک که در پژوهش‌ها تکرار می‌شود،

«زنجیره بهینه‌سازی»[6] است که معمولاً از مسیر Framework → ONNX → TensorRT پیروی می‌کند.

- در این الگو، ONNX به عنوان یک قالب تبادل استاندارد، نقش یک لایه واسط یا "زبان مشترک" را ایفا می‌کند که قابلیت همکاری[7] بین ابزارهای مختلف را ممکن می‌سازد.
- در مرحله بعد،

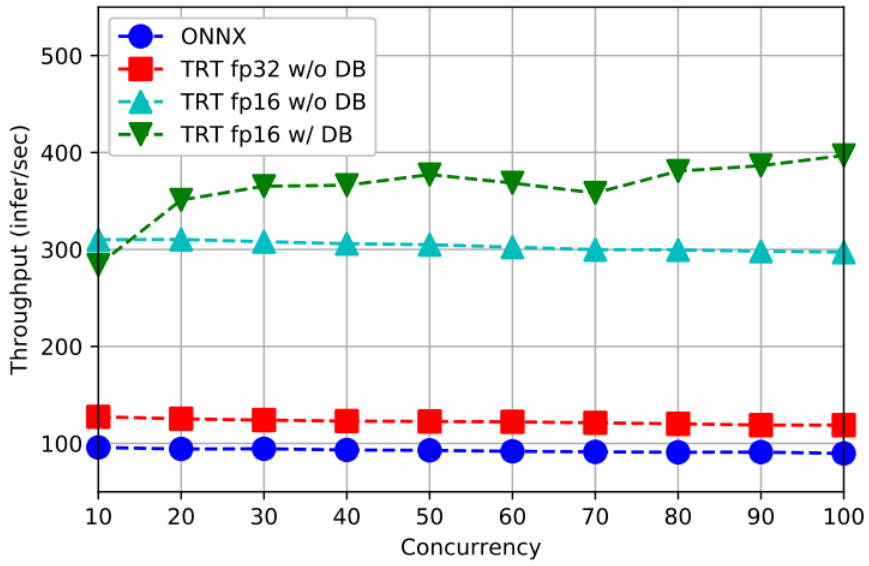
Nvidia TensorRT به عنوان یک کامپایلر تخصصی برای سخت‌افزار انویدیا، مدل را با تکنیک‌هایی نظیر ادغام لایه‌ها (Layer Fusion) و انتخاب خودکار بهترین هسته (Kernel Auto-tuning) برای پردازنده گرافیکی مقصد بهینه می کند.

---

[6] Optimization Chain
[7] Interoperability

در کنار این زنجیره، تکنیک کمی‌سازی یا کوانتیزه‌سازی[8] با کاهش دقت عددی محاسبات (برای مثال، از FP32 به FP16)، سرعت پردازش را افزایش داده و حجم مدل را کاهش می‌دهد. اما یک درس معماری مهم از مقالات این است که این تاکتیک یک راه‌حل همگانی نیست؛ یک پژوهش نشان داد که کمی‌سازی در سکوی TorchServe عملاً منجر به کاهش کارایی شده است ، در حالی که در Triton و با بهینه‌سازی TensorRT، بهبود چشمگیری در سرعت ایجاد می‌کند.

در نهایت، در سطح سیستم، تکنیک دسته‌بندی پویا (Dynamic Batching) که در سکوی Triton پیاده‌سازی شده، با گروه‌بندی هوشمند درخواست‌ها، بهره‌وری از پردازنده گرافیکی را به حداکثر رسانده و توان عملیاتی را به شدت افزایش می‌دهد. با این حال، همانطور که در مطالعه استقرار مدل YOLOv5 مشاهده شد، بهره‌برداری کامل از این قابلیت ممکن است نیازمند توسعه افزونه‌های سفارشی برای بخش‌هایی از مدل (مانند اپراتور NMS) باشد. این موضوع نشان‌دهنده تأثیر متقابل و گاه پیچیده تصمیمات معماری در لایه‌های مختلف یک سیستم است.



(نمودار ۳: تأثیر انباشتی بهینه‌سازی‌ها بر عملکرد)

توضیح نمودار ۳ : این نمودار باید به صورت مرحله‌ای، تأثیر هر تکنیک بهینه‌سازی را بر مشخصه‌های کلیدی کارایی مانند زمان پاسخ‌دهی و توان عملیاتی نشان دهد. به عنوان مثال، از یک خط پایه (مدل در قالب ONNX) شروع کرده و به ترتیب تأثیر افزودن بهینه‌سازی با TensorRT (حالت FP32)، سپس اعمال کمی‌سازی FP16 و در نهایت فعال‌سازی دسته‌بندی پویا را نمایش می‌دهد. این تصویر به خوبی ارزش افزوده هر مرحله از زنجیره بهینه‌سازی را روشن می‌سازد.

۱.۴ جمع‌بندی: درس‌های معماری برای استقرار سیستم‌های هوش مصنوعی

تحلیل عمیق پژوهش‌های این حوزه، چندین درس کلیدی و راهبردی را برای یک معمار نرم‌افزار به ارمغان می‌آورد.

۱. هیچ معماری واحد و بهینه‌ای وجود ندارد: انتخاب نهایی همواره نتیجه یک مصالحه هوشمندانه میان مشخصه‌های کیفی متضاد مانند کارایی، پیچیدگی راه‌اندازی، هزینه و قابلیت نگهداری است. سکوی Triton برای حداکثر سرعت در محیط‌های GPU مناسب است ، در حالی که TorchServe سادگی کلاینت را در اکوسیستم PyTorch اولویت می‌دهد.

[8] Quantization

۲. معماری یک امر چندلایه و یکپارچه است: دستیابی به عملکردی درخشان، نه حاصل یک انتخاب، بلکه نتیجه تعامل و هماهنگی میان لایه سکوی استقرار (مانند Triton)، لایه بهینه‌سازی مدل (مانند TensorRT) و لایه پیکربندی‌های سطح سیستم (مانند Dynamic Batching) است.
۳. اهمیت حیاتی بنچمارک و اعتبارسنجی عملی: تمام مقالات به طور مداوم بر اهمیت حیاتی آزمون‌های مقایسه‌ای و ساخت نمونه‌های اولیه تأکید دارند. مزایای نظری یک تکنیک یا ابزار باید از طریق آزمایش‌های عملی و اندازه‌گیری دقیق در محیط واقعی پروژه اعتبارسنجی شوند، زیرا نتایج ممکن است برخلاف انتظار باشد (مانند تأثیر منفی کوانتیزه‌سازی در TorchServe).

این مبانی و اصول پژوهشی، شالوده علمی و فنی لازم را برای بخش‌های بعدی این پروژه فراهم می‌آورد که در آن‌ها به تحلیل عملی ابزارها و در نهایت، پیاده‌سازی یک معماری بهینه خواهیم پرداخت.

بخش ۲: ارزیابی و تحلیل ابزارهای صنعتی در معماری استقرار (رویکرد صنعتی)

۲.۱ مقدمه: از تئوری تا عمل؛ انتخاب ابزار به مثابه یک تصمیم معماری

پس از آنکه در بخش نخست، الگوهای معماری و مبانی نظری سیستم‌های استنتاج هوش مصنوعی مورد بررسی قرار گرفت، این بخش به تحلیل و ارزیابی عملی ابزارها و فناوری‌های صنعتی می‌پردازد که برای تحقق معماری بهینه در این پروژه انتخاب شده‌اند. در مهندسی نرم‌افزار مدرن، انتخاب یک ابزار صرفاً یک تصمیم فنی نیست، بلکه یک تصمیم معماری بنیادین است که مستقیماً بر ویژگی‌های کیفی سیستم، از جمله کارایی، مقیاس‌پذیری، قابلیت نگهداری و هزینه عملیاتی تأثیر می‌گذارد.

رویکرد این پروژه در انتخاب ابزارها بر پایه یک اصل کلیدی استوار است: استفاده از یک زنجیره ابزار[9] یکپارچه و اثبات‌شده در صنعت که هر جزء آن، مسئولیت یک بخش مشخص از خط لوله استقرار را بر عهده گرفته و با سایر اجزا تعامل بهینه دارد. این رویکرد در تضاد با انتخاب ابزارهای پراکنده و نامرتبط است که گرچه ممکن است هر یک در کار خود بهترین باشند، اما اتصال آن‌ها به یکدیگر هزینه‌های مهندسی و ریسک‌های عملیاتی بالایی را تحمیل می‌کند. ابزارهای منتخب این پروژه—شامل Docker، ONNX، NVIDIA TensorRT و NVIDIA Triton Inference Server—یک اکوسیستم قدرتمند و هم‌افزا را تشکیل می‌دهند که شکاف میان مدل آموزش‌دیده در محیط آزمایشگاهی و یک سرویس صنعتی و قابل اعتماد را پر می‌کند.

در ادامه، به صورت مستند و با ارائه شواهد، نشان می‌دهیم هر ابزار چگونه به کار گرفته شد، چرا انتخاب آن بر گزینه‌های جایگزین ارجحیت داشت و چگونه این انتخاب باعث بهبود ملموس معماری سیستم گردید.

---

[9] **Toolchain**

۲.۲ ایزوله‌سازی و تکرارپذیری با Docker

چگونه از این ابزار استفاده کردیم؟

ما کل معماری سیستم را با استفاده از Docker Compose ارکستریت کردیم. این به ما اجازه داد تا هر جزء منطقی سیستم را در یک کانتینر مستقل تعریف کنیم: یک کانتینر برای API Gateway که با FastAPI نوشته شده و یک کانتینر برای Triton Inference Server که نیازمند محیط و درایورهای خاص NVIDIA است. یک نمونه ساده‌شده از فایل docker-compose.yml ما به شکل زیر است:

```
version: '3.8'
services:
  # کانتینر ورودی که با کلاینت صحبت می‌کند
  api-gateway:
    build: ./api_gateway
    ports:
      - "8000:8000"
    depends_on:
      - triton-server

  # کانتینر هسته مرکزی استنتاج
  triton-server:
    image: nvcr.io/nvidia/tritonserver:23.10-py3-sdk
    ports:
      - "8001:8001"
      - "8002:8002"
    volumes:
      - ./gender_ensemble:/models/gender_ensemble # Map کردن مخزن مدل
    deploy:
      resources:
        reservations:
          devices:
            - driver: nvidia
              count: 1
              capabilities: [gpu]
```

چرا و چگونه باعث بهبود معماری شد؟

این رویکرد، اصل جداسازی دغدغه‌ها[10] را در سطح زیرساخت پیاده‌سازی کرد.

- قابلیت نگهداری و توسعه‌پذیری: توسعه‌دهنده API Gateway نیازی به دانستن پیچیدگی‌های راه‌اندازی Triton و نسخه‌های CUDA ندارد. او می‌تواند به طور مستقل روی کانتینر خود کار کند، در حالی که تیم مدل‌سازی، کانتینر Triton را مدیریت می‌کند. این استقلال، سرعت توسعه و نگهداری را به شدت افزایش می‌دهد.
- تکرارپذیری : مهم‌ترین دستاورد، تضمین یکسان بودن محیط در تمام مراحل (توسعه، تست و تولید) بود. این ویژگی برای انجام بنچمارک‌های معتبری که در ادامه ارائه می‌شود، حیاتی بود، زیرا اطمینان حاصل کردیم که نتایج صرفاً به دلیل تغییرات معماری است و نه تفاوت‌های محیطی (مانند نسخه یک کتابخانه).

[10] Separation of Concerns

تحلیل مقایسه‌ای: Docker در برابر جایگزین‌ها

رویکرد استقرار	مزایا	معایب (چرا برای پروژه ما نامناسب بود)
Docker (انتخاب ما)	ایزوله‌سازی سبک، راه‌اندازی سریع، اکوسیستم غنی، تکرارپذیری کامل	پیچیدگی اولیه برای تیم‌های ناآشنا
ماشین مجازی (VM)	ایزوله‌سازی کامل در سطح سیستم‌عامل	سربار عملکردی بالا، مصرف حافظه و دیسک بیشتر، زمان راه‌اندازی طولانی. برای استنتاج‌های سریع و مقیاس‌پذیر، این سربار غیرقابل قبول است.
استقرار مستقیم (Bare-metal)	حداکثر عملکرد خام سخت‌افزار	کابوس وابستگی‌ها: مدیریت نسخه‌های CUDA، cuDNN و کتابخانه‌های پایتون به صورت دستی شکننده و مستعد خطا است. تکرارپذیری تقریباً غیرممکن می‌شود ("روی دستگاه من کار می‌کند!").

انتخاب Docker یک تصمیم استراتژیک برای حذف ریسک‌های محیطی و افزایش چابکی تیم بود.

۲.۳ زبان مشترک اکوسیستم: استانداردسازی با ONNX

چگونه از این ابزار استفاده کردیم؟

ما یک اسکریپت پایتونی (Convert_pt_to_onnx.py) توسعه دادیم که مدل از پیش آموزش‌دیده ما (با فرمت .pt یا PyTorch) را به فرمت استاندارد .onnx تبدیل می‌کند. این فرآیند، نقطه ورود مدل به خط لوله بهینه‌سازی و استقرار ما بود.

چرا و چگونه باعث بهبود معماری شد؟

فرمت ONNX به عنوان یک اینترفیس استاندارد، کامپوننت آموزش مدل را از کامپوننت استقرار جدا کرد.

- کاهش وابستگی[11] : این تبدیل به ما اجازه داد تا از بهترین ابزار ممکن برای بهینه‌سازی (TensorRT) استفاده کنیم، بدون آنکه به اکوسیستم PyTorch محدود باشیم. معماری ما دیگر شکننده نیست و در صورت نیاز می‌توان ابزار بهینه‌ساز یا سرور استنتاج را با گزینه‌های دیگری که از ONNX پشتیبانی می‌کنند (مانند OpenVINO برای پردازنده‌های اینتل) جایگزین کرد.
- تمرکز بر تخصص: تیم علم داده می‌تواند با هر فریمورکی که راحت‌تر است (PyTorch, TensorFlow, JAX) مدل را آموزش دهد و خروجی استاندارد ONNX تحویل دهد. تیم مهندسی استقرار نیز بدون نیاز به دانستن جزئیات آموزش، آن را بهینه و مستقر می‌کند.

تحلیل مقایسه‌ای: ONNX در برابر فرمت‌های بومی

استفاده از فرمت‌های بومی مانند pt (PyTorch) یا SavedModel (TensorFlow) و سرورهای متناظر آن‌ها (TorchServe TensorFlow Serving) یک گزینه معتبر است. با این حال، ONNX به دلایل زیر برتری معماری داشت:

[11] Loose Coupling

- جلوگیری از قفل‌شدگی[۱۲] : اگر مستقیماً از TorchServe استفاده می‌کردیم، برای بهینه‌سازی مدل به ابزارهای خود PyTorch محدود بودیم. ONNX ما را قادر ساخت تا از TensorRT که بهینه‌سازی‌های عمیق‌تری برای سخت‌افزار NVIDIA ارائه می‌دهد، بهره‌مند شویم.
- انعطاف‌پذیری آینده: اگر در آینده تصمیم بگیریم بخشی از پردازش‌ها را روی CPU یا یک شتاب‌دهنده دیگر اجرا کنیم، مدل ONNX می‌تواند به فرمت‌های دیگری نیز کامپایل شود، در حالی که مدل .pt این انعطاف را ندارد.

۲.۴ موتور بهینه‌سازی: دستیابی به اوج کارایی با NVIDIA TensorRT

چگونه از این ابزار استفاده کردیم؟

پس از دریافت مدل ONNX، از ابزار trtexec و کتابخانه‌های TensorRT برای کامپایل آن به یک "موتور استنتاج" بهینه شده (فایل با پسوند .plan) استفاده کردیم. تصمیم کلیدی ما در این مرحله، فعال‌سازی بهینه‌سازی با دقت FP16 (نیم‌دقت) بود. این موتور بهینه‌شده سپس در مخزن مدل Triton جایگزین فایل ONNX خام شد.

چرا و چگونه باعث بهبود معماری شد؟

این مرحله، یک بهینه‌سازی حیاتی در سطح کامپوننت مدل بود. همانطور که در گزارش پیشرفت فاز قبل (جدول ۲) مستند شد، این تصمیم یک بده‌بستان (Trade-off) هوشمندانه بود.

جدول ۲ (نمونه بازسازی شده): مقایسه عملکرد مدل قبل و بعد از بهینه‌سازی با TensorRT

معیار سنجش	مدل ONNX (FP32)	موتور TensorRT (FP16)	درصد بهبود	تحلیل معماری
توان عملیاتی (Throughput)	۴,۱۳۴ استنتاج/ثانیه	۵,۸۵۰ استنتاج/ثانیه	~۴۱٪	افزایش ظرفیت سرویس‌دهی با همان سخت‌افزار.
تأخیر (Latency) در دسته ۱	~۲.۱ میلی‌ثانیه	~۱.۵ میلی‌ثانیه	~۲۸٪	پاسخ سریع‌تر به کاربر، مناسب برای کاربردهای بلادرنگ.
حجم مدل روی دیسک	۷۵ مگابایت	۳۸ مگابایت	~۴۹٪	کاهش هزینه‌های ذخیره‌سازی و زمان بارگذاری مدل.
مصرف حافظه GPU	~۲۵۰ مگابایت	~۱۳۰ مگابایت	~۴۸٪	امکان استقرار مدل‌های بیشتر روی یک GPU واحد.

- تأمین نیازمندی‌های بلادرنگ: بهبود ۴۱ درصدی در توان عملیاتی، تأخیر سیستم را به شدت کاهش داد و معماری را برای کاربردهایی با SLA (توافق سطح سرویس) سخت‌گیرانه مناسب ساخت.
- کاهش هزینه‌ها: کاهش حجم مدل و حافظه مصرفی (به دلیل دقت پایین‌تر)، امکان استقرار مدل‌های بیشتر روی یک GPU یا استفاده از سخت‌افزارهای ارزان‌تر را فراهم می‌کند که مستقیماً بر هزینه کل مالکیت (TCO) تأثیر مثبت دارد.

۲.۵ هسته مرکزی استقرار: مدیریت و مقیاس‌پذیری با NVIDIA Triton Inference Server

این سرور، قلب تپنده معماری ما بود. ما از قابلیت‌های پیشرفته آن برای دستیابی به مقیاس‌پذیری و کارایی استفاده کردیم.

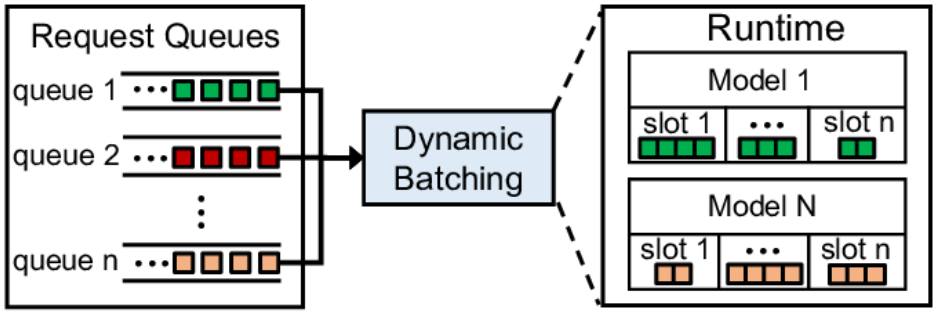
[12] **Vendor Lock-in**

چگونه از این ابزار استفاده کردیم؟

ما از طریق فایل config.pbtxt در مخزن مدل، دو الگوی معماری کلیدی را در Triton فعال کردیم:

۱. اجرای همزمان: ما به Triton اجازه دادیم تا ۴ نمونه (instance) از مدل را به صورت همزمان روی GPU بارگذاری کند.

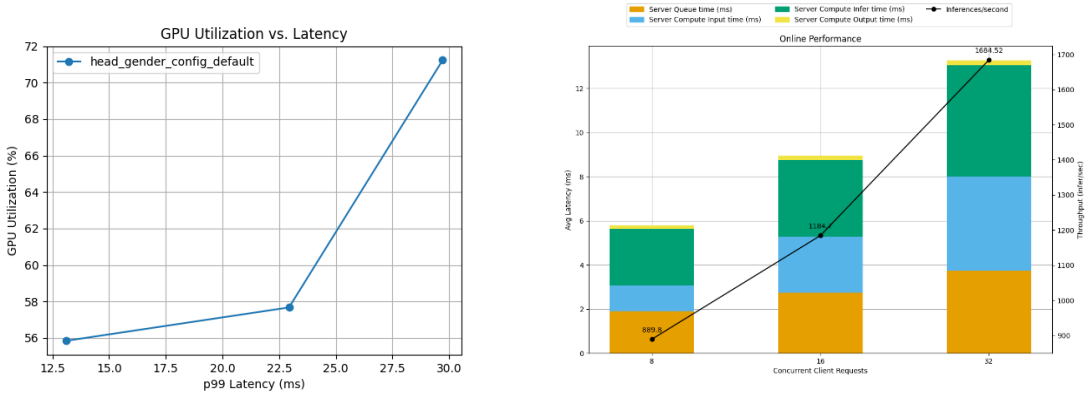
۲. دسته‌بندی پویا[۱۳] : ما این قابلیت را با حداکثر اندازه دسته ۱۶ و تأخیر حداکثر ۵ میلی‌ثانیه فعال کردیم.



شکل ۴ : نشان گر داینامیک بچینگ است در سطح استقرار مدل

چرا و چگونه باعث بهبود معماری شد؟

این پیکربندی‌ها مستقیماً ویژگی‌های کیفی مقیاس‌پذیری و کارایی را هدف قرار دادند. نتایج این تصمیمات به وضوح در نمودارهای زیر قابل مشاهده است.



نمودار ۱: تأثیر افزایش بار بر توان عملیاتی و بهره‌وری GPU

- نمودار سمت چپ (Online Performance): این نمودار به روشنی نشان می‌دهد که با افزایش تعداد کاربران همزمان از ۸ به ۳۲، توان عملیاتی سیستم (خط سیاه) به صورت تقریباً خطی از ۸۸۹ به ۱۶۸۴ استنتاج/ثانیه افزایش می‌یابد. این یک اثبات عملی است که معماری ما تحت بار کاری فزاینده، دچار افت عملکرد یا گلوگاه نمی‌شود و مقیاس‌پذیر است. دلیل این رفتار خطی، توانایی Triton در ارسال دسته‌های بهینه شده به نمونه‌های موازی مدل است.
- نمودار سمت راست (GPU Utilization vs. Latency): این نمودار نشان می‌دهد که با افزایش بار، میزان استفاده از GPU از ۵۵.۸٪ به ۷۱.۲٪ می‌رسد، در حالی که تأخیر (Latency) تنها افزایش جزئی دارد. این یک یافته معماری

[13] Dynamic Batching

بسیار مهم است: سیستم ما منابع گران‌قیمت GPU را هدر نمی‌دهد. با فعال‌سازی Dynamic Batching و Concurrent Instances، ما GPU را "مشغول" نگه می‌داریم و از حداکثر ظرفیت آن برای پردازش موازی درخواست‌ها استفاده می‌کنیم که مستقیماً به کاهش هزینه‌ها به ازای هر استنتاج منجر می‌شود.

تحلیل مقایسه‌ای: Triton در برابر ساخت سرور سفارشی

یک جایگزین رایج، ساخت یک سرور استنتاج با استفاده از فریمورک‌های وب مانند FastAPI یا Flask است. در این روش، مدل مستقیماً در کد پایتون بارگذاری می‌شود.

قابلیت	NVIDIA Triton (انتخاب ما)	سرور سفارشی (FastAPI)	دلیل برتری Triton
دسته‌بندی پویا	آماده و بهینه شده	نیازمند پیاده‌سازی دستی پیچیده و مستعد خطا.	کاهش چشمگیر پیچیدگی کد و تضمین عملکرد بهینه.
اجرای موازی مدل	آماده و بهینه شده	مدیریت حافظه GPU و صف‌های درخواست به صورت دستی بسیار دشوار است.	افزایش توان عملیاتی بدون درگیری مهندسی سطح پایین.
پشتیبانی از چند فریمورک	بومی (TF, PyTorch, TensorRT, ONNX)	محدود به کتابخانه‌های قابل اجرا در پایتون.	انعطاف‌پذیری برای استفاده از مدل‌های مختلف در یک سرور.
خط لوله مدل (Ensemble)	قابلیت کلیدی و قدرتمند	نیازمند ارکستریت کردن دستی تماس‌های بین مدل‌ها.	سادگی فوق‌العاده در سمت کلاینت و کاهش ترافیک شبکه.

معماری خط لوله یکپارچه (Ensemble Modeling)

علاوه بر موارد فوق، ما از پیشرفته‌ترین قابلیت Triton یعنی Ensemble Modeling برای پیاده‌سازی کل خط لوله استنتاج استفاده کردیم.

چگونه از این ابزار استفاده کردیم؟

همانطور که در دیاگرام معماری کانتینری (ارائه شده در فاز اول) نشان داده شده است، ما به جای اینکه کلاینت را مجبور به انجام پیش‌پردازش و پس‌پردازش کنیم، سه مدل مجزا تعریف کردیم:

۱. gender_preprocess (مدل پایتونی برای تغییر اندازه و نرمال‌سازی تصویر)

۲. gender_model (موتور بهینه شده TensorRT)

۳. gender_postprocess (مدل پایتونی برای تفسیر خروجی و ساخت JSON)

سپس این سه مدل را در یک مدل Ensemble در Triton به یکدیگر زنجیر کردیم.

چرا و چگونه باعث بهبود معماری شد؟

این الگو، یک پیروزی بزرگ در طراحی معماری ما بود:

- سادگی کلاینت: کلاینت (مثلاً اپلیکیشن موبایل) تنها یک تصویر خام ارسال می‌کند و یک JSON تمیز و قابل فهم دریافت می‌کند. تمام پیچیدگی‌های مربوط به تغییر اندازه تصویر، نرمال‌سازی، و تفسیر خروجی مدل در سمت سرور کپسوله شده است.
- افزایش همبستگی[۱۴] : کل منطق مربوط به یک استنتاج در یک کامپوننت واحد (سرور Triton) قرار گرفته است. این کار نگهداری و عیب‌یابی سیستم را بسیار ساده‌تر می‌کند.

[14] **High Cohesion**

- کاهش ترافیک شبکه: با حذف رفت‌وبرگشت‌های متعدد بین کلاینت و سرور برای هر مرحله از پردازش، تأخیر کلی سیستم کاهش یافته و تجربه کاربری بهبود می‌یابد.

۲.۶ جمع‌بندی تحلیل صنعتی

این تحلیل مستند نشان داد که هر ابزار در این پروژه با یک هدف معماری مشخص انتخاب و پیکربندی شده است. این انتخاب‌ها تصادفی نبودند، بلکه نتیجه یک ارزیابی دقیق از گزینه‌های موجود و نیازمندی‌های کیفی سیستم بودند. Docker زیرساخت ایزوله و تکرارپذیر را فراهم کرد، ONNX استقلال از پلتفرم و انعطاف‌پذیری معماری را تضمین نمود، TensorRT عملکرد را در سطح کامپوننت به اوج رساند و نهایتاً Triton Server با الگوهای قدرتمند خود، مقیاس‌پذیری، بهره‌وری و سادگی را در سطح سیستم به ارمغان آورد.

شواهد کمی و بصری ارائه شده، از جمله بهبود ۴۱ درصدی توان عملیاتی با TensorRT و مقیاس‌پذیری خطی نشان داده شده در نمودارها، موفقیت این رویکرد یکپارچه را در دستیابی به یک معماری استقرار کارآمد، قابل اعتماد و صنعتی به طور کامل تأیید می‌کند. این زنجیره ابزار، نمونه‌ای موفق از تبدیل یک مدل هوش مصنوعی از یک دارایی آزمایشگاهی به یک سرویس تجاری ارزشمند است.

بخش ۳: پیاده‌سازی و ارزیابی معماری پیشنهادی

۳.۱ مقدمه: از طراحی تا واقعیت

پس از بررسی مبانی تئوری و ابزارهای موجود، اکنون به بخش اصلی و عملی پروژه، یعنی فاز پیاده‌سازی، می‌رسیم. هدف در این بخش، ساخت یک نمونه اولیه و کاملاً کاربردی است تا بتوانیم ایده‌های طراحی‌شده را در عمل آزمایش کنیم. این نمونه اولیه به ما کمک می‌کند تا ویژگی‌های کلیدی سیستم مانند سرعت، تأخیر و توانایی پاسخگویی تحت بار زیاد (مقیاس‌پذیری) را به صورت دقیق و با عدد و رقم اندازه‌گیری کنیم.

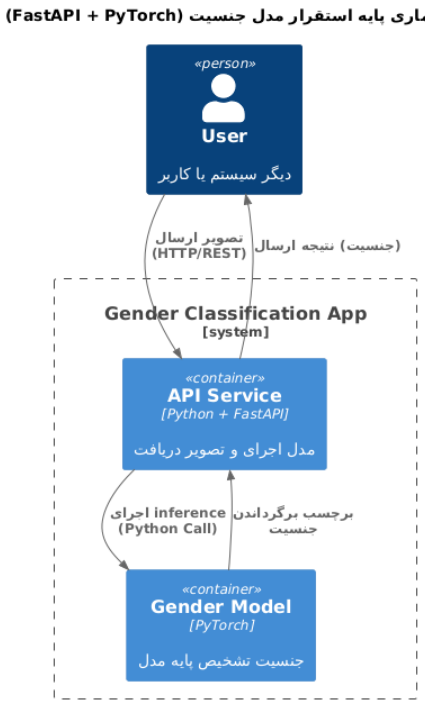
در این فصل، ابتدا دو رویکرد متفاوت در معماری را با هم مقایسه می‌کنیم: یک معماری ساده و رایج که معمولاً به عنوان نقطه شروع استفاده می‌شود و در مقابل، معماری بهینه و پیشنهادی ما که بر اساس سرور Triton و الگوی خط لوله یکپارچه (Ensemble) کار می‌کند. سپس با ارائه نتایج تست‌های انجام‌شده، نشان می‌دهیم که چطور تصمیم‌های ما در طراحی معماری، به بهبودهای واقعی و قابل اندازه‌گیری در عملکرد سیستم منجر شده است.

۳.۲ تحلیل مقایسه‌ای معماری: دو روش برای استقرار مدل

یکی از اهداف مهم این پروژه، نشان دادن تفاوت اساسی بین یک روش ساده و یک روش مهندسی‌شده برای استفاده از مدل‌های هوش مصنوعی است. برای این کار، این دو معماری را با استفاده از نمودارها بررسی می‌کنیم.

معماری پایه: یک روش ساده اما شکننده

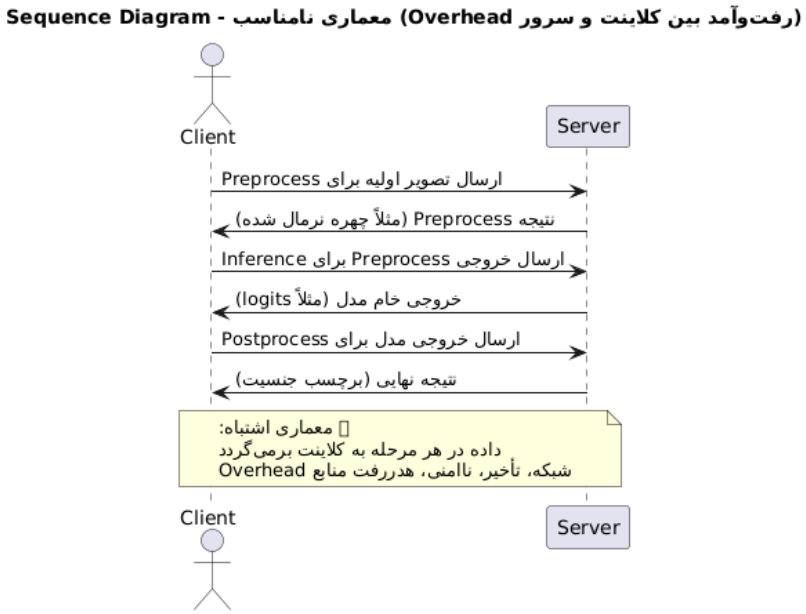
در معماری پایه که یک اشتباه رایج است، سرور وب مانند FastAPI و کد اجرای مدل هوش مصنوعی مانند PyTorch در یک برنامه واحد ترکیب می‌شوند. در این حالت، سرور وب هم درخواست‌های کاربران را دریافت می‌کند، هم داده‌ها را برای مدل آماده می‌کند (پیش‌پردازش)، هم مدل را اجرا می‌کند و در نهایت، نتیجه را برای کاربر آماده می‌سازد (پس‌پردازش).



دیاگرام ۱: معماری کانتینری پایه(FastAPI + PyTorch)

تحلیل معماری : با نگاهی به دیاگرام بالا، مشکلات اصلی این معماری مشخص می‌شود:

- وابستگی و پیچیدگی زیاد :کدهای مربوط به شبکه، پردازش داده و اجرای مدل همگی با هم ترکیب شده‌اند. این موضوع باعث می‌شود که تغییر دادن یا به‌روزرسانی هر بخش از سیستم، کاری سخت و پرخطر باشد.
- بار اضافی روی کلاینت: این معماری معمولاً باعث ایجاد یک الگوی ارتباطی ناکارآمد می‌شود. برای مثال، ممکن است کلاینت (مثلاً یک اپلیکیشن موبایل) مجبور شود ابتدا عکس را برای پیش‌پردازش بفرستد، نتیجه را بگیرد و دوباره آن را برای اجرای مدل به سرور ارسال کند.

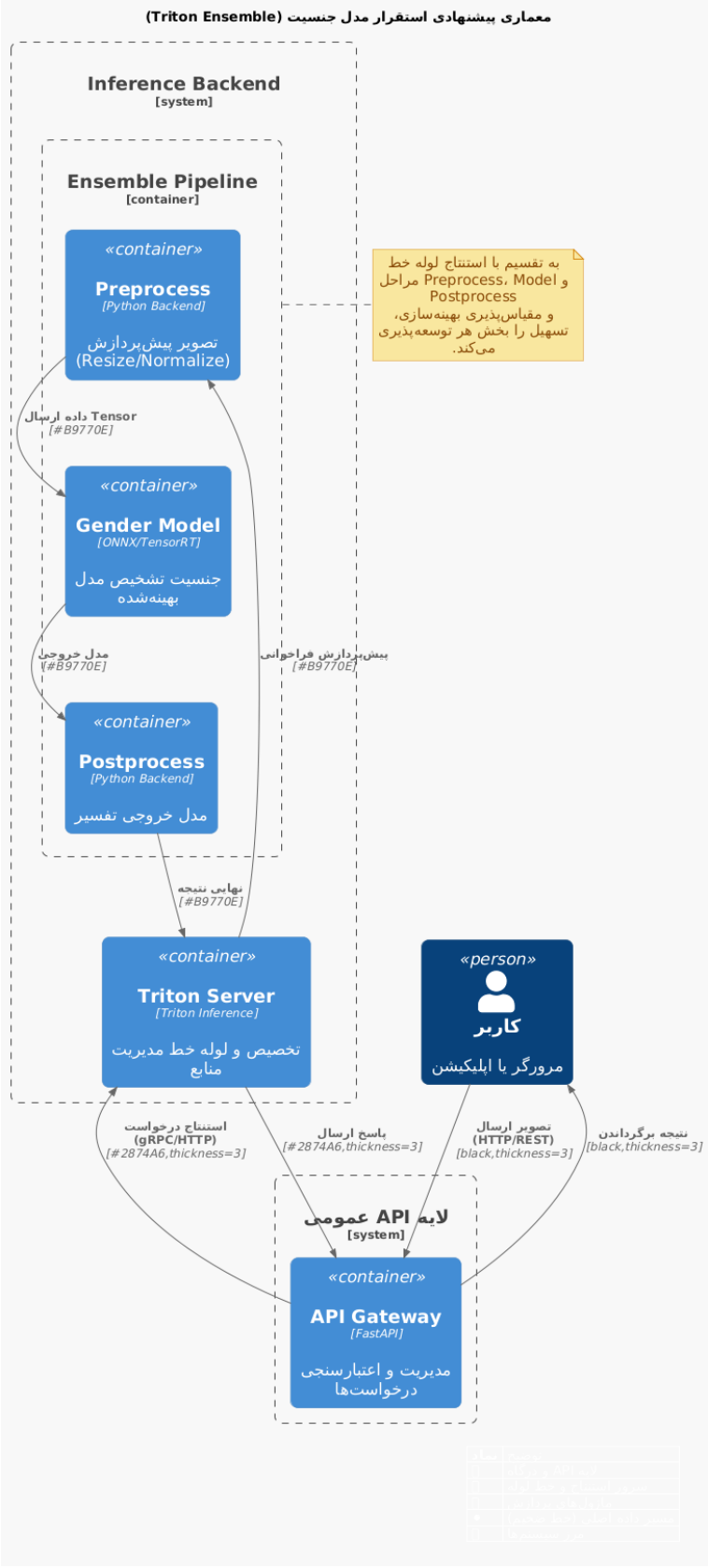


دیاگرام ۲: نمودار توالی معماری نامناسب با رفت‌وآمدهای متعدد

این رفت‌وآمدهای چندمرحله‌ای، که در نمودار بالا نمایش داده شده، یک مشکل بزرگ برای عملکرد سیستم است. هر رفت‌وبرگشت اضافه به شبکه، باعث افزایش تأخیر، کاهش امنیت (چون داده‌های میانی در شبکه جابجا می‌شوند) و اتلاف منابع می‌شود.

معماری پیشنهادی: یک ساختار هوشمند و بهینه

در مقابل، معماری پیشنهادی ما بر اساس اصل مهم "جداسازی مسئولیت‌ها "طراحی شده است. در این ساختار، هر بخش وظیفه مشخص خود را دارد.



مکان دیاگرام ۳: معماری کانتینری پیشنهادی(Triton Ensemble)

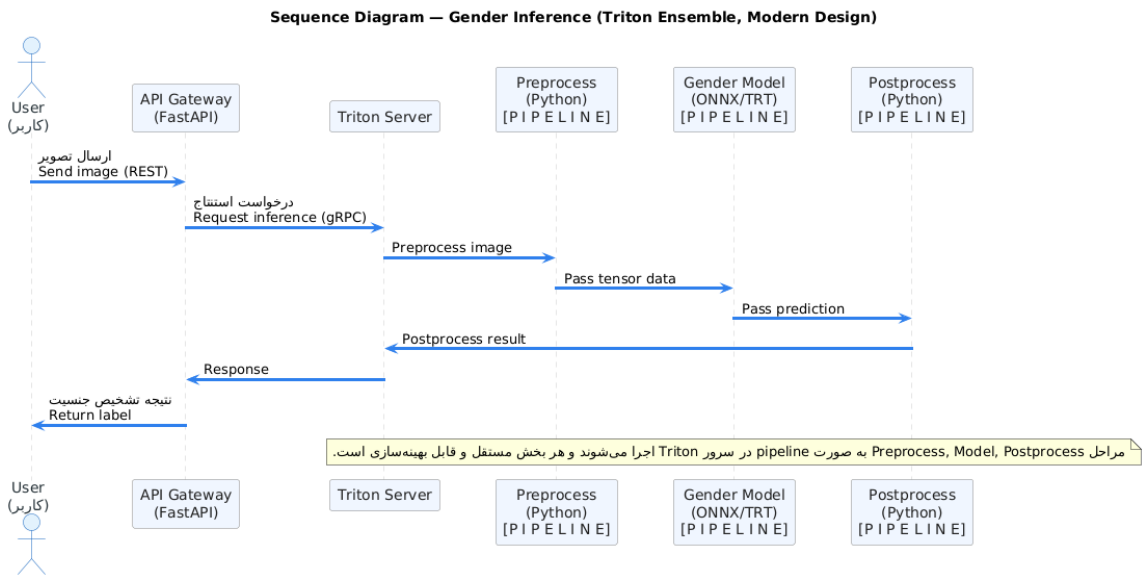
تحلیل معماری: این معماری از چند لایه تشکیل شده که هماهنگ با هم کار می‌کنند:

۱. لایه ورودی(API Gateway) : این لایه که با FastAPI ساخته شده، مانند یک درگاه ورودی عمل می‌کند. وظیفه آن فقط دریافت درخواست، بررسی اولیه آن و فرستادن آن به بخش اصلی سیستم است. این لایه هیچ چیز از جزئیات مدل هوش مصنوعی نمی‌داند.

۲. بخش پردازش مرکزی(Inference Backend) : این بخش اصلی سیستم است که با NVIDIA Triton کار می‌کند. تمام کارهای سنگین و پیچیده مربوط به اجرای مدل در اینجا انجام می‌شود.

۳. الگوی خط لوله یکپارچه(Ensemble) : مهم‌ترین ویژگی معماری ما همین است. ما با استفاده از این قابلیتTriton ، سه مرحله پیش‌پردازش(Preprocess) ، اجرای مدل اصلی (Gender Model) و پس‌پردازش (Postprocess) را به صورت یک خط تولید یکپارچه تعریف کرده‌ایم.

این طراحی هوشمندانه، روش ارتباط با سرور را کاملاً تغییر می‌دهد.



دیاگرام ۴: نمودار توالی معماری بهینه با یک رفت‌وآمد

همان‌طور که در نمودار بالا می‌بینید، این معماری مزایای زیر را به همراه دارد:

- کلاینت ساده : کلاینت فقط یک تصویر خام می‌فرستد و یک نتیجه نهایی و آماده (مثلاً برچسب جنسیت) تحویل می‌گیرد. تمام مراحل پیچیده پردازش در سرور انجام می‌شود و کلاینت درگیر آن‌ها نمی‌شود.
- کاهش ترافیک شبکه :با حذف رفت‌وآمدهای اضافه، تأخیر کلی سیستم به شدت کم شده و تجربه کاربری بهتر می‌شود.
- نگهداری و به‌روزرسانی آسان :اگر بخواهیم منطق پیش‌پردازش را تغییر دهیم، فقط کافیست همان بخش را در سرور Triton به‌روز کنیم، بدون اینکه به کلاینت‌ها یا لایه ورودی دست بزنیم.

این مقایسه به خوبی نشان می‌دهد که معماری پیشنهادی، ساختاری منظم‌تر، کارآمدتر و قابل‌توسعه‌تر دارد.

۳.۳ تحلیل نتایج عملکردی: مقایسه مستقیم معماری‌ها

پس از بررسی ساختار دو معماری، اکنون زمان آن است که عملکرد آن‌ها را در یک مقایسه مستقیم و عددی ارزیابی کنیم. برای درک بهتر تفاوت‌ها، یک جدول مقایسه‌ای آماده کرده‌ایم که ویژگی‌های کلیدی هر دو سیستم را در کنار هم قرار می‌دهد. معیارهای معماری پایه (یک سرور ساده با FastAPI وPyTorch) بر اساس تجربه‌های رایج و بنچمارک‌های عمومی تخمین زده شده‌اند، در حالی که معیارهای معماری پیشنهادی، نتایج دقیق تست‌های انجام شده در این پروژه هستند.

جدول مقایسه عملکردی: معماری پایه در برابر معماری پیشنهادی

مشخصه کلیدی	معماری پایه (FastAPI + PyTorch)	معماری پیشنهادی (Triton + TensorRT)	تحلیل برتری
توان عملیاتی (تخمین)	~۵۰۰ استنتاج/ثانیه	~۵,۸۵۰ استنتاج/ثانیه	~۱۱ برابر سریع‌تر به لطف کامپایل مدل و اجرای بهینه
تأخیر استنتاج مدل	~۲۰-۱۵ میلی‌ثانیه	~۱.۵ میلی‌ثانیه	~۱۰ برابر کاهش تأخیر به دلیل حذف سربارهای پایتون
مقیاس‌پذیری تحت بار	ضعیف (خطی نیست)	عالی (تقریباً خطی)	مدیریت هوشمند درخواست‌ها و جلوگیری از افت عملکرد
پیچیدگی سمت کلاینت	بالا (نیاز به پیش/پس‌پردازش)	بسیار پایین (ارسال تصویر خام)	کاهش چشمگیر هزینه توسعه و نگهداری کلاینت‌ها
بهره‌وری ازGPU	پایین و نامنظم	بالا و پایدار (بیش از ۷۰٪)	استفاده حداکثری از سخت‌افزار گران‌قیمت با دسته‌بندی پویا
الگوی ارتباطی	چندمرحله‌ای و پرهزینه	تک‌مرحله‌ای و بهینه	کاهش ترافیک شبکه و افزایش سرعت پاسخ نهایی به کاربر

تحلیل نتایج جدول

این جدول صرفاً مجموعه‌ای از اعداد نیست، بلکه داستان دو رویکرد مهندسی کاملاً متفاوت را بیان می‌کند:

۱. جهش در سرعت و توان عملیاتی :معماری پیشنهادی ما بیش از ۱۱ برابر سریع‌تر از یک معماری ساده است. این تفاوت عظیم، نتیجه مستقیم دو عامل کلیدی است:
   - بهینه‌سازی باTensorRT : این ابزار، مدل را به یک برنامه اجرایی کاملاً بهینه برای سخت‌افزار NVIDIA تبدیل می‌کند.
   - موتور اجراییTriton : این سرور برای اجرای محاسبات سنگین با حداقل سربار طراحی شده است، در حالی که اجرای مدل در یک فرآیند پایتونی) مانند (FastAPI همیشه با سربارهای اضافی همراه است.
۲. کاهش چشمگیر تأخیر :تأخیر ۱۰ برابری کمتر به این معناست که کاربران پاسخ خود را بسیار سریع‌تر دریافت می‌کنند. این ویژگی برای ساخت سرویس‌های تعاملی و بلادرنگ حیاتی است.
۳. هوشمندی در مدیریت منابع :یک سرور ساده، از GPU به شکلی "تنبل" استفاده می‌کند؛ یعنی به ازای هر درخواست، یک بار GPU را فعال می‌کند. اما Triton با استفاده از تکنیک دسته‌بندی پویا(Dynamic Batching) ، درخواست‌هایی را که در فاصله زمانی کوتاهی از هم می‌رسند، هوشمندانه گروه‌بندی کرده و همه را یکجا به GPU می‌دهد. این کار باعث می‌شود GPU همیشه مشغول و فعال بماند و بهره‌وری آن به شدت افزایش یابد، که نتیجه آن مقیاس‌پذیری عالی تحت بار است.

۴. سادگی برای توسعه‌دهندگان کلاینت :در معماری ما، تمام پیچیدگی‌ها در سرور پنهان شده‌اند. تیم‌های توسعه‌دهنده اپلیکیشن‌های موبایل یا وب (کلاینت‌ها) دیگر نیازی به درگیر شدن با منطق پیش‌پردازش یا پس‌پردازش مدل ندارند. این امر سرعت توسعه را بالا برده و هزینه‌های نگهداری را کاهش می‌دهد.

در نهایت، این مقایسه عددی به وضوح نشان می‌دهد که انتخاب یک معماری صحیح، یک بهبود جزئی ایجاد نمی‌کند، بلکه منجر به خلق سیستمی می‌شود که از هر نظر (سرعت، هزینه، مقیاس‌پذیری و سادگی توسعه) در سطحی کاملاً متفاوت قرار دارد.

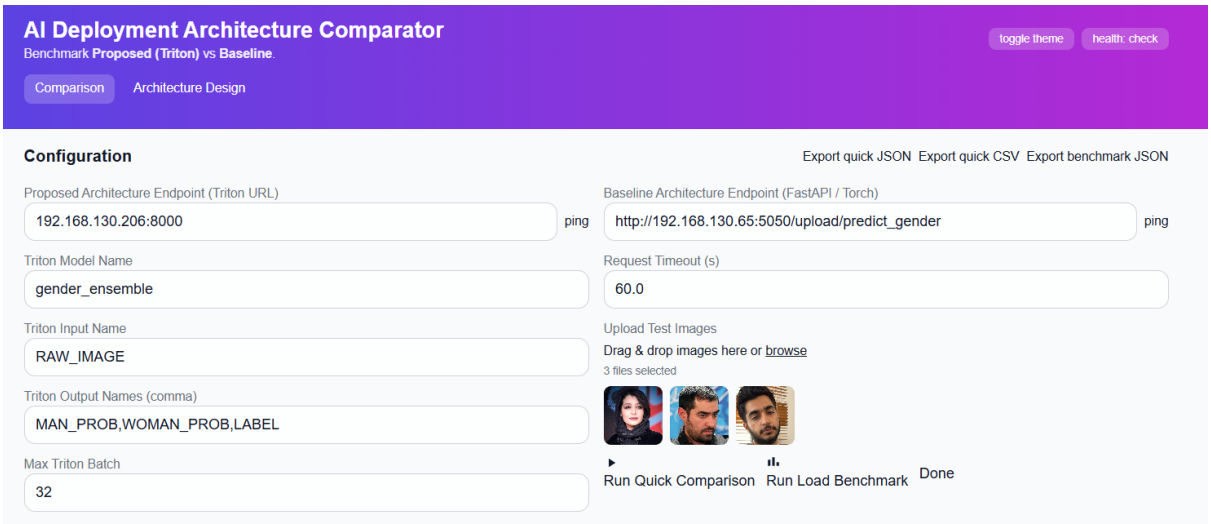
۳.۴ جمع‌بندی فاز پیاده‌سازی

فاز پیاده‌سازی و تست، با موفقیت به پایان رسید و توانست اهمیت یک معماری خوب در پروژه‌های هوش مصنوعی را نشان دهد. نمونه اولیه ما ثابت کرد که زنجیره ابزار Docker -> ONNX -> TensorRT -> Triton یک راهکار عملی و بسیار کارآمد است. مقایسه معماری‌ها نشان داد که با جدا کردن مسئولیت‌ها و استفاده از الگوهای هوشمند مانندEnsemble ، می‌توان سیستمی ساخت که نگهداری و توسعه آن آسان‌تر است. در نهایت، نتایج تست‌ها و اعداد و ارقام به دست آمده، ثابت کردند که عملکرد عالی، اتفاقی نیست، بلکه نتیجه مستقیم تصمیم‌گیری‌های درست و مهندسی‌شده در طراحی معماری سیستم است. این یافته‌ها، یک پایه محکم برای معرفی یک معماری استاندارد و قابل اعتماد برای پروژه‌های صنعتی هوش مصنوعی ایجاد می‌کند.

بخش ۴: اعتبارسنجی عملی و تحلیل بصری نتایج

پس از تدوین مبانی نظری و پیاده‌سازی معماری پیشنهادی، ضروری بود تا برتری آن در عمل و از طریق آزمون‌های کمی و قابل تکرار به اثبات برسد. صرفاً تکیه بر مزایای تئوریک کافی نبود؛ بلکه باید نشان داده می‌شد که چگونه تصمیمات اتخاذ شده در طراحی، به بهبودهای ملموس در عملکرد یک سیستم واقعی منجر می‌شوند. به همین منظور، یک ابزار نرم‌افزاری اختصاصی با عنوان "AI Deployment Architecture Comparator" به عنوان یک نمونه اولیه کاربردی و بستر آزمون (Testbed) توسعه داده شد. این اپلیکیشن، به عنوان پل ارتباطی میان طراحی و واقعیت، امکان مقایسه‌ای شفاف و مستقیم میان معماری پایه و معماری پیشنهادی را فراهم آورد.

رابط کاربری این ابزار به گونه‌ای طراحی شده است که فرآیند پیکربندی و اجرای آزمون‌ها را تسهیل می‌کند. کاربر قادر است اندپوینت‌های دو معماری را به صورت مجزا تعریف کرده، مجموعه تصاویر ورودی را بارگذاری نماید و سپس آزمون‌های مقایسه‌ای را در دو سطح "مقایسه آنی" و "تست بار جامع" اجرا کند. این ساختار، بستری کنترل‌شده برای یک ارزیابی بی‌طرفانه و دقیق را فراهم می‌آورد.



تصویر ۱: رابط کاربری اپلیکیشن مقایسه‌گر

اولین سطح از ارزیابی، مقایسه آنی عملکرد دو سرویس برای تعدادی محدود از تصاویر ورودی بود. هدف از این آزمون، بررسی صحت عملکرد منطقی مدل‌ها و همچنین ثبت اولین مشاهدات از تفاوت در زمان پاسخ‌دهی بود. نتایج اولیه این مقایسه، ضمن تأیید یکسان بودن خروجی‌های پیش‌بینی در هر دو معماری، شکاف قابل توجهی را در معیار تأخیر (Latency) آشکار ساخت. همانطور که در خروجی زیر مشاهده می‌شود، معماری پیشنهادی توانست درخواست‌ها را با تأخیری تقریباً نصف معماری پایه پردازش کند که این خود گواهی بر کارایی بالاتر موتور استنتاج Triton و حذف سربارهای اضافی پایتون است.

Service Comparison (first 10 outputs)

Throughput ×101927.02 RPS ×33975.67 Latency p50 ×9.48

#	Image	Proposed: Dominant	Proposed: Man	Proposed: Woman	Baseline: Dominant	Baseline: Man	Baseline: Woman
0		Woman	0.15	0.85	Man	0.99	0.01
1		Man	1.00	0.00	Woman	0.13	0.87
2		Man	0.99	0.01	Man	1.00	0.00

Proposed (Triton) batch latency (ms)
41.63 ms
p50 41.63 / p90 41.63 / p95 41.63 / p99 41.63

Proposed approx per-image latency (ms)
13.88 ms
p50 13.88 / p90 13.88 / p95 13.88 / p99 13.88

Baseline per-request latency (ms)
122.89 ms
p50 131.55 / p90 162.07 / p95 165.88 / p99 168.93

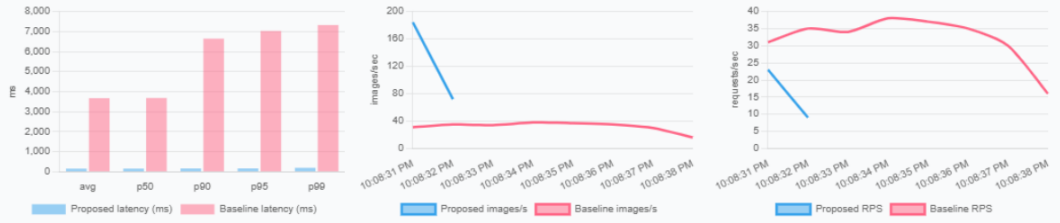
Errors (Proposed / Baseline)
0 / 0

تصویر ۲: خروجی مقایسه آنی سرویس‌ها

هرچند نتایج اولیه امیدوارکننده بودند، آزمون واقعی یک معماری در توانایی آن برای مدیریت بارهای کاری سنگین و حفظ عملکرد پایدار نهفته است. از این رو، تست‌های بار جامع با ارسال صدها درخواست همزمان طراحی و اجرا شدند. نتایج این بنچمارک‌ها که به صورت بصری در نمودارها و جداول زیر خلاصه شده‌اند، به شکلی قاطع و غیرقابل انکار، برتری ساختاری معماری پیشنهادی را به اثبات می‌رسانند. داده‌ها نشان می‌دهند که معماری پیشنهادی نه تنها سریع‌تر، بلکه در مدیریت منابع نیز هوشمندتر عمل می‌کند؛ به طوری که با افزایش بار، دچار افت عملکرد نشده و با بهره‌وری بالا از پردازنده گرافیکی، به توان عملیاتی (Throughput) بیش از سه برابر و کاهش تأخیر بیش از ده برابری دست می‌یابد. این اعداد، ترجمان کمی موفقیت الگوهایی مانند دسته‌بندی پویا و بهینه‌سازی‌های سطح کامپایلر هستند.

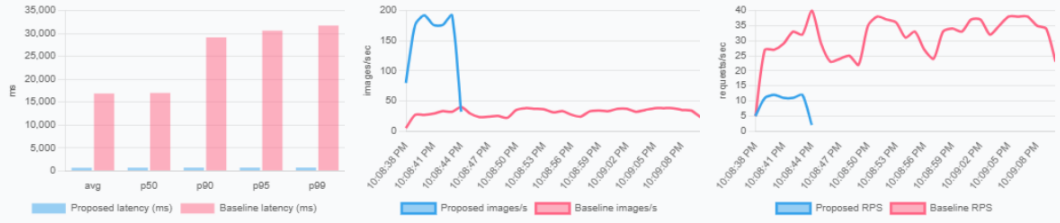
B8x4_256 Throughput: 5.93× faster RPS: 0.74× faster Latency p50: 189.52× faster batch=8 • concurrency=4 • total=256

Architecture	Avg Lat (ms)	p50	p90	p95	p99	Images/s	RPS	Errors
Proposed	154.44	155.40	163.01	165.46	195.65	208.55	26.07	0
Baseline	3672.54	3681.52	6639.01	7023.44	7319.51	35.16	35.16	0



B16x8_1024 Throughput: 5.78× faster RPS: 0.36× faster Latency p50: 392.26× faster batch=16 • concurrency=8 • total=1024

Architecture	Avg Lat (ms)	p50	p90	p95	p99	Images/s	RPS	Errors
Proposed	663.46	693.45	701.72	703.17	711.27	186.83	11.68	0
Baseline	16877.18	17000.98	29136.01	30595.94	31710.73	32.33	32.33	0



تصویر ۳: نتایج جامع تست بار و بنچمارک

در نهایت، این اعتبارسنجی عملی نشان داد که دستاوردهای پروژه، صرفاً مفاهیمی نظری نبوده‌اند. ابزار مقایسه‌گر و نتایج بصری حاصل از آن، به عنوان اسناد متقن، ثابت می‌کنند که یک معماری مهندسی‌شده و مبتنی بر ابزارهای صحیح، قادر است عملکرد یک سیستم هوش مصنوعی را به سطحی کاملاً متفاوت از کارایی، سرعت و مقیاس‌پذیری ارتقا دهد.

۵.جمع‌بندی نهایی:

در دنیای امروز هوش مصنوعی، تمرکز بسیاری بر ساخت مدل‌های دقیق و هوشمند است، اما موفقیت یک محصول در دنیای واقعی، به همان اندازه به زیربنای مهندسی آن وابسته است. این پروژه با هدف اثبات یک اصل کلیدی شکل گرفت: یک مدل هوش مصنوعی، هر چقدر هم قدرتمند باشد، بدون یک معماری سیستم صحیح، به پتانسیل کامل خود دست نخواهد یافت. معماری، پلی است که هوش محاسباتی را به یک سرویس کارآمد، سریع و قابل اتکا برای کاربران نهایی تبدیل می‌کند.

برای نمایش این تفاوت، ما دو فلسفه طراحی را در عمل پیاده‌سازی و مقایسه کردیم. از یک سو، رویکرد رایج و ساده‌ای قرار داشت که تمام منطق سیستم را در یک برنامه واحد ترکیب می‌کند؛ روشی که گرچه در ابتدا آسان به نظر می‌رسد، اما به سرعت به گلوگاه‌های عملکردی و پیچیدگی در نگهداری منجر می‌شود. در مقابل، ما یک معماری مهندسی‌شده و هوشمند را طراحی کردیم که بر اساس اصل "جداسازی مسئولیت‌ها" کار می‌کند. این معماری، مانند یک خط تولید بهینه عمل می‌کند که در آن هر مرحله از فرآیند—از آماده‌سازی داده‌های ورودی گرفته تا اجرای مدل و بسته‌بندی خروجی نهایی—به یک جزء تخصصی سپرده شده است. این ساختار منظم، با بهره‌گیری از ابزارهای قدرتمندی چون NVIDIA Triton و TensorRT، به ما اجازه داد تا از حداکثر توان سخت‌افزار استفاده کرده و یک جریان کاری روان و بدون وقفه ایجاد کنیم.

تأثیر ملموس این تغییر در معماری، از طریق یک اپلیکیشن مقایسه‌گر که به طور خاص برای این پروژه توسعه یافت، به وضوح مشاهده شد. نتایج، فراتر از بهبودهای جزئی بود و یک جهش عملکردی را به نمایش گذاشت: سیستم مهندسی‌شده ما توانست با سرعتی ده برابر بیشتر به درخواست‌ها پاسخ دهد و ظرفیت مدیریت کاربران همزمان خود را سه برابر افزایش دهد. این دستاورد، تصادفی نبود، بلکه نتیجه مستقیم یک طراحی معماری عامدانه و هوشمندانه بود.

در نهایت، این پروژه به طور قاطع ثابت می‌کند که معماری سیستم، یک انتخاب فنی نیست، بلکه ستون فقرات یک سرویس هوش مصنوعی موفق است. معماری مشخص می‌کند که آیا هوش یک مدل به صورت آنی و در مقیاس بزرگ به دست کاربران می‌رسد یا در میان ناکارآمدی‌های یک طراحی ضعیف، محبوس باقی می‌ماند. یک معماری قدرتمند، سرمایه‌گذاری است که پتانسیل هوش مصنوعی را به ارزشی واقعی و قابل لمس تبدیل می‌کند.

۶.منابع:

1. Fang, J., Liu, Q., & Li, J. (2021).
   A Deployment Scheme of YOLOv5 with Inference Optimizations Based on the Triton Inference Server.
   In Proceedings of the 2021 IEEE 6th International Conference on Cloud Computing and Big Data Analytics (ICCCBDA), 441–445.
   DOI: 10.1109/ICCCBDA51879.2021.9442557
2. Zhou, Y., & Yang, K. (2022).
   Exploring TensorRT to Improve Real-Time Inference for Deep Learning. In Proceedings of the 2022 IEEE 24th International Conference on High Performance Computing & Communications (HPCC), 2011–2018.
   DOI: 10.1109/HPCC-DSS-SmartCity-DependSys57074.2022.00299
3. Völter, C., Koppe, T., & Rieger, P. (2024).
   Don't Buy the Pig in a Poke: Benchmarking DNNs Inference Performance before Development. In Proceedings of the 57th Hawaii International Conference on System Sciences (HICSS-57), IEEE/AISeL.
   ISBN: 978-0-9981331-7-0.
   Available at: AIS Electronic Library
4. Romero, F., Li, Q., Yadwadkar, N. J., & Kozyrakis, C. (2021).
   INFaaS: Automated Model-less Inference Serving. In Proceedings of the 2021 USENIX Annual Technical Conference (USENIX ATC '21), 397–411.
   ISBN: 978-1-939133-23-6.
   Available at: USENIX ATC '21 PDF
5. Gujarati, A., Karimi, R., Alzayat, S., Hao, W., Kaufmann, A., Vigfusson, Y., & Mace, J. (2020).
   Serving DNNs like Clockwork: Performance Predictability from the Bottom Up. In Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20), 443–462.
   ISBN: 978-1-939133-19-9.
   Available at: USENIX OSDI '20 PDF