Run LLaVA-Next on SGLang for Visual QA

View on GitHub

Vision-Language Models (VLMs) are like LLMs with eyes: they can generate text based not just on other text, but on images as well.

This example shows how to run a VLM on Modal using the SGLang library.

Here's a sample inference, with the image rendered directly in the terminal:

```
Question: What is this?
```

```
Answer:
Stopping app - local entrypoint completed.
The image shows the Statue of Liberty, an iconic symbol of freedom located on a small isla
nd in the middle of New York Harbor, with a backdrop of the Manhattan skyline. The statue
is a neoclassical sculpture with a pedestal beneath it, and it has been a prominent figure
 in the harbor since it was dedicated in 1886. It's a well-maintained and culturally signi
ficant attraction, often associated with the harbor and tourism in the United States.
request 672af06d-5bc6-410f-9ab9-8958856e41a4 completed in 3.8 seconds
```

Setup

First, we'll import the libraries we need locally and define some constants.

```
import os
import time
import warnings
from uuid import uuid4

import modal
import requests
```

VLMs are generally larger than LLMs with the same cognitive capability. LLMs are already hard to run effectively on CPUs, so we'll use a GPU here. We find that inference for a single input takes about 3-4 seconds on an A10G.

You can customize the GPU type and count using the `GPU_CONFIG` and `GPU_COUNT` environment variables. If you want to see the model really rip, try an `"a100-80gb"` or an `"h100"` on a large batch.

```
GPU_TYPE = os.environ.get("GPU_CONFIG", "a10g")
GPU_COUNT = os.environ.get("GPU_COUNT", 1)

GPU_CONFIG = f"{GPU_TYPE}:{GPU_COUNT}"

SGL_LOG_LEVEL = "error"  # try "debug" or "info" if you have issues

MINUTES = 60  # seconds
```

We use a LLaVA-NeXT model built on top of Meta's LLaMA 3 8B.

```
MODEL_PATH = "lmms-lab/llama3-llava-next-8b"
MODEL_REVISION = "e7e6a9fd5fd75d44b32987cba51c123338edbede"
TOKENIZER_PATH = "lmms-lab/llama3-llava-next-8b-tokenizer"
MODEL_CHAT_TEMPLATE = "llama-3-instruct"
```

We download it from the Hugging Face Hub using the Python function below.

```
def download_model_to_image():
    import transformers
    from huggingface_hub import snapshot_download

    snapshot_download(
        MODEL_PATH,
        revision=MODEL_REVISION,
        ignore_patterns=["*.pt", "*.bin"],
    )

    # otherwise, this happens on first inference
    transformers.utils.move_cache()
```

Modal runs Python functions on containers in the cloud. The environment those functions run in is defined by the container's `Image`. The block of code below defines our example's `Image`.

```
vllm_image = (
    modal.Image.from_registry(  # start from an official NVIDIA CUDA image
        "nvidia/cuda:12.2.0-devel-ubuntu22.04", add_python="3.11"
    )
    .apt_install("git")  # add system dependencies
    .pip_install(  # add sglang and some Python dependencies
        "sglang[all]==0.1.16",
```

```
        "ninja",
        "packaging",
        "transformers==4.40.2",
    )
    .run_commands(  # add FlashAttention for faster inference using a shell command
        "pip install flash-attn==2.5.8 --no-build-isolation"
    )
    .run_function(  # download the model by running a Python function
        download_model_to_image
    )
    .pip_install(  # add an optional extra that renders images in the terminal
        "term-image==0.7.1"
    )
)
```

Defining a Visual QA service

Running an inference service on Modal is as easy as writing inference in Python.

The code below adds a modal `Cls` to an `App` that runs the VLM.

We define a method `generate` that takes a URL for an image URL and a question about the image as inputs and returns the VLM's answer.

By decorating it with `@modal.web_endpoint`, we expose it as an HTTP endpoint, so it can be accessed over the public internet from any client.

```
app = modal.App("app")


@app.cls(
    gpu=GPU_CONFIG,
    timeout=20 * MINUTES,
    container_idle_timeout=20 * MINUTES,
    allow_concurrent_inputs=100,
    image=vllm_image,
)
class Model:
    @modal.enter()  # what should a container do after it starts but before it gets input?
    async def start_runtime(self):
        """Starts an SGL runtime to execute inference."""
        import sglang as sgl

        self.runtime = sgl.Runtime(
            model_path=MODEL_PATH,
            tokenizer_path=TOKENIZER_PATH,
            tp_size=GPU_COUNT,  # t_ensor p_arralel size, number of GPUs to split the mode
            log_evel=SGL_LOG_LEVEL,
```

```
        )
        self.runtime.endpoint.chat_template = (
            sgl.lang.chat_template.get_chat_template(MODEL_CHAT_TEMPLATE)
        )
        sgl.set_default_backend(self.runtime)

    @modal.web_endpoint(method="POST")
    async def generate(self, image_url: str = None, question: str = None):
        import sglang as sgl
        from term_image.image import from_file

        start = time.monotonic_ns()
        request_id = uuid4()
        print(f"Generating response to request {request_id}")

        if image_url is None:
            image_url = "https://cdn.britannica.com/61/93061-050-99147DCE/Statue-of-Libert

        image_filename = image_url.split("/")[-1]
        image_path = f"/tmp/{uuid4()}-{image_filename}"
        response = requests.get(image_url)

        response.raise_for_status()

        with open(image_path, "wb") as file:
            file.write(response.content)

        @sgl.function
        def image_qa(s, image_path, question):
            s += sgl.user(sgl.image(image_path) + question)
            s += sgl.assistant(sgl.gen("answer"))

        if question is None:
            question = "What is this?"

        state = image_qa.run(
            image_path=image_path, question=question, max_new_tokens=128
        )
        # show the question, image, and response in the terminal for demonstration purpose
        print(
            Colors.BOLD, Colors.GRAY, "Question: ", question, Colors.END, sep=""
        )
        terminal_image = from_file(image_path)
        terminal_image.draw()
        answer = state["answer"]
        print(
            Colors.BOLD,
            Colors.GREEN,
            f"Answer: {answer}",
            Colors.END,
            sep="",
        )
```

```
        print(
            f"request {request_id} completed in {round((time.monotonic_ns() - start) / 1e9
        )

    @modal.exit()  # what should a container do before it shuts down?
    def shutdown_runtime(self):
        self.runtime.shutdown()
```

Asking questions about images via POST

Now, we can send this Modal Function a POST request with an image and a question and get back an answer.

The code below will start up the inference service so that it can be run from the terminal as a one-off, like a local script would be, using `modal run`:

```
modal run sgl_vlm.py
```

By default, we hit the endpoint twice to demonstrate how much faster the inference is once the server is running.

```
@app.local_entrypoint()
def main(image_url=None, question=None, twice=True):
    model = Model()

    response = requests.post(
        model.generate.web_url,
        json={"image_url": image_url, "question": question},
    )
    assert response.ok, response.status_code

    if twice:
        # second response is faster, because the Function is already running
        response = requests.post(
            model.generate.web_url,
            json={"image_url": image_url, "question": question},
        )
        assert response.ok, response.status_code
```

Deployment

To set this up as a long-running, but serverless, service, we can deploy it to Modal:

```
modal deploy sgl_vlm.py
```

And then send requests from anywhere. See the docs for details on the `web_url` of the function, which also appears in the terminal output when running `modal deploy`.

Addenda

The rest of the code in this example is just utility code.

```
warnings.filterwarnings(  # filter warning from the terminal image library
    "ignore",
    message="It seems this process is not running within a terminal. Hence, some features
    category=UserWarning,
)


class Colors:
    """ANSI color codes"""

    GREEN = "\033[0;32m"
    BLUE = "\033[0;34m"
    GRAY = "\033[0;90m"
    BOLD = "\033[1m"
    END = "\033[0m"
```

Fetching stock prices in parallel

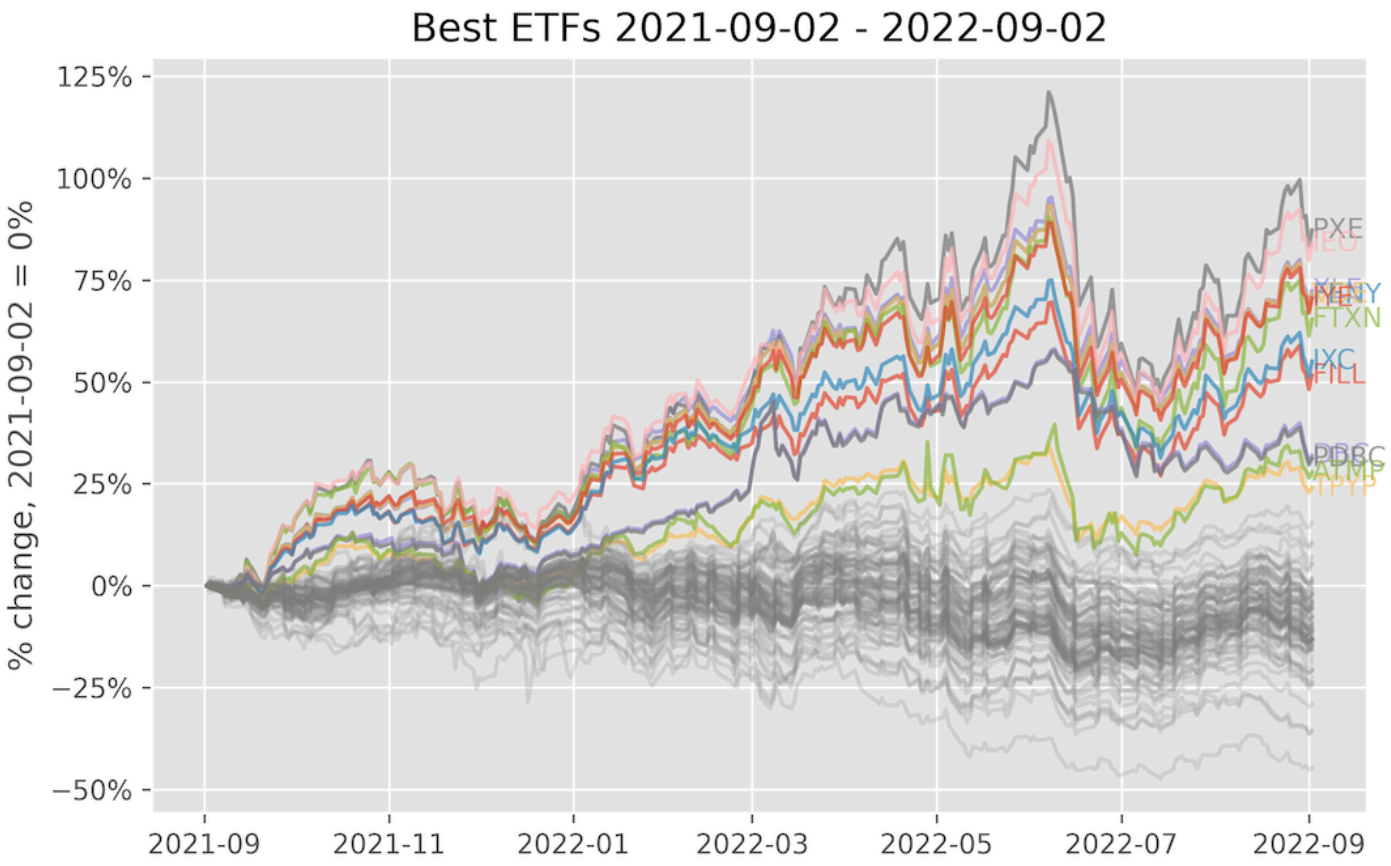
View on GitHub

This is a simple example that uses the Yahoo! Finance API to fetch a bunch of ETFs We do this in parallel, which demonstrates the ability to map over a set of items In this case, we fetch 100 stocks in parallel

You can run this script on the terminal with

```
modal run 03_scaling_out/fetch_stock_prices.py
```

If everything goes well, it should plot something like this:



Setup

For this image, we need

- `httpx` and `beautifulsoup4` to fetch a list of ETFs from a HTML page
- `yfinance` to fetch stock prices from the Yahoo Finance API
- `matplotlib` to plot the result

```
import io
import os

import modal

app = modal.App(
    "example-fetch-stock-prices",
    image=modal.Image.debian_slim().pip_install(
        "httpx~=0.24.0",
        "yfinance~=0.2.31",
        "beautifulsoup4~=4.12.2",
        "matplotlib~=3.7.1",
    ),
)  # Note: prior to April 2024, "app" was called "stub"
```

Fetch a list of tickers

The `yfinance` package does not have a way to download a list of stocks. To get a list of stocks, we parse the HTML from Yahoo Finance using Beautiful Soup and ask for the top 100 ETFs.

```
@app.function()
def get_stocks():
    import bs4
    import httpx

    headers = {
        "user-agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML
        "referer": "https://finance.yahoo.com/",
    }
    url = "https://finance.yahoo.com/etfs?count=100&offset=0"
    res = httpx.get(url, headers=headers)
    res.raise_for_status()
    soup = bs4.BeautifulSoup(res.text, "html.parser")
    for td in soup.find_all("td", {"aria-label": "Symbol"}):
        for link in td.find_all("a", {"data-test": "quoteLink"}):
            symbol = str(link.next)
            print(f"Found symbol {symbol}")
            yield symbol
```

Fetch stock prices

Now, let's fetch the stock data. This is the function that we will parallelize. It's fairly simple and just uses the `yfinance` package.

```
@app.function()
def get_prices(symbol):
    import yfinance

    print(f"Fetching symbol {symbol}...")
    ticker = yfinance.Ticker(symbol)
    data = ticker.history(period="1Y")["Close"]
    print(f"Done fetching symbol {symbol}!")
    return symbol, data.to_dict()
```

Plot the result

Here is our plotting code. We run this in Modal, although you could also run it locally. Note that the plotting code calls the other two functions. Since we plot the data in the cloud, we can't display it, so we generate a PNG and return the binary content from the function.

```
@app.function()
def plot_stocks():
    from matplotlib import pyplot, ticker

    # Setup
    pyplot.style.use("ggplot")
    fig, ax = pyplot.subplots(figsize=(8, 5))

    # Get data
    tickers = list(get_stocks.remote_gen())
    if not tickers:
        raise RuntimeError("Retrieved zero stock tickers!")
    data = list(get_prices.map(tickers))
    first_date = min((min(prices.keys()) for symbol, prices in data if prices))
    last_date = max((max(prices.keys()) for symbol, prices in data if prices))

    # Plot every symbol
    for symbol, prices in data:
        if len(prices) == 0:
            continue
        dates = list(sorted(prices.keys()))
        prices = list(prices[date] for date in dates)
        changes = [
            100.0 * (price / prices[0] - 1) for price in prices
        ]  # Normalize to initial price
        if changes[-1] > 20:
            # Highlight this line
            p = ax.plot(dates, changes, alpha=0.7)
            ax.annotate(
                symbol,
                (last_date, changes[-1]),
                ha="left",
                va="center",
                color=p[0].get_color(),
                alpha=0.7,
            )
        else:
            ax.plot(dates, changes, color="gray", alpha=0.2)

    # Configure axes and title
    ax.yaxis.set_major_formatter(ticker.PercentFormatter())
    ax.set_title(f"Best ETFs {first_date.date()} - {last_date.date()}")
    ax.set_ylabel(f"% change, {first_date.date()} = 0%")

    # Dump the chart to .png and return the bytes
    with io.BytesIO() as buf:
        pyplot.savefig(buf, format="png", dpi=300)
        return buf.getvalue()
```

Entrypoint

The entrypoint locally runs the app, gets the chart back as a PNG file, and saves it to disk.

```
OUTPUT_DIR = "/tmp/"


@app.local_entrypoint()
def main():
    os.makedirs(OUTPUT_DIR, exist_ok=True)
    data = plot_stocks.remote()
    filename = os.path.join(OUTPUT_DIR, "stock_prices.png")
    print(f"saving data to {filename}")
    with open(filename, "wb") as f:
        f.write(data)
```

Algolia docsearch crawler

View on GitHub

This tutorial shows you how to use Modal to run the Algolia docsearch crawler to index your website and make it searchable. This is not just example code - we run the same code in production to power search on this page (`Ctrl+K` to try it out!).

Basic setup

Let's get the imports out of the way.

```
import json
import os
import subprocess

from modal import App, Image, Secret, web_endpoint
```

Modal lets you use and extend existing Docker images, as long as they have `python` and `pip` available. We'll use the official crawler image built by Algolia, with a small adjustment: since this image has `python` symlinked to `python3.6` and Modal is not compatible with Python 3.6, we install Python 3.11 and symlink that as the `python` executable instead.

```
algolia_image = Image.from_registry(
    "algolia/docsearch-scraper:v1.16.0",
    add_python="3.11",
    setup_dockerfile_commands=["ENTRYPOINT []"],
)
```

```
app = App(
    "example-algolia-indexer"
)  # Note: prior to April 2024, "app" was called "stub"
```

Configure the crawler

Now, let's configure the crawler with the website we want to index, and which CSS selectors we want to scrape. Complete documentation for crawler configuration is available here.

```
CONFIG = {
    "index_name": "modal_docs",
    "start_urls": [
        {"url": "https://modal.com/docs/guide", "page_rank": 2},
        {"url": "https://modal.com/docs/examples", "page_rank": 1},
        {"url": "https://modal.com/docs/reference", "page_rank": 1},
    ],
    "selectors": {
        "lvl0": {
            "selector": ".sidebar .active",
            "default_value": "Documentation",
            "global": True,
        },
        "lvl1": "article h1",
        "lvl2": "article h2",
        "lvl3": "article h3",
        "lvl4": "article h4",
        "text": "article p,article ol,article ul,article pre",
    },
}
```

Create an API key

If you don't already have one, sign up for an account on Algolia. Set up a project and create an API key with `write` access to your index, and with the ACL permissions `addObject`, `editSettings` and `deleteIndex`. Now, create a secret on the Modal Secrets page with the `API_KEY` and `APPLICATION_ID` you just created. You can name this anything you want, we named it `algolia-secret`.

The actual function

We want to trigger our crawler from our CI/CD pipeline, so we're serving it as a web endpoint that can be triggered by a `GET` request during deploy. You could also consider running the crawler on a schedule.

The Algolia crawler is written for Python 3.6 and needs to run in the `pipenv` created for it, so we're invoking it using a subprocess.

```
@app.function(
    image=algolia_image,
    secrets=[Secret.from_name("algolia-secret")],
)
def crawl():
    # Installed with a 3.6 venv; Python 3.6 is unsupported by Modal, so use a subprocess i
    subprocess.run(
        ["pipenv", "run", "python", "-m", "src.index"],
        env={**os.environ, "CONFIG": json.dumps(CONFIG)},
    )
```

We want to be able to trigger this function through a webhook.

```
@app.function()
@web_endpoint()
def crawl_webhook():
    crawl.remote()
    return "Finished indexing docs"
```

Deploy the indexer

That's all the code we need! To deploy your application, run

```
modal deploy algolia_indexer.py
```

If successful, this will print a URL for your new webhook, that you can hit using `curl` or a browser. Logs from webhook invocations can be found from the apps page.

The indexed contents can be found at https://www.algolia.com/apps/APP_ID/explorer/browse/, for your APP_ID. Once you're happy with the results, you can set up the `docsearch` package with your website, and create a search component that uses this index.

Entrypoint for development

To make it easier to test this, we also have an entrypoint for when you run `modal run algolia_indexer.py`

```
@app.local_entrypoint()
def run():
    crawl.remote()
```

Analyze NYC yellow taxi data with DuckDB on Parquet files from S3

View on GitHub

This example shows how to use Modal for a classic data science task: loading table-structured data into cloud stores, analyzing it, and plotting the results.

In particular, we'll load public NYC taxi ride data into S3 as Parquet files, then run SQL queries on it with DuckDB.

We'll mount the S3 bucket in a Modal app with `CloudBucketMount`. We will write to and then read from that bucket, in each case using Modal's parallel execution features to handle many files at once.

Basic setup

You will need to have an S3 bucket and AWS credentials to run this example. Refer to the documentation for the exact IAM permissions your credentials will need.

After you are done creating a bucket and configuring IAM settings, you now need to create a `Secret` to share the relevant AWS credentials with your Modal apps. Navigate to the "Secrets" tab and click on the AWS card, then fill in the fields with your credentials. Name the secret `s3-bucket-secret`.

```
from datetime import datetime
from pathlib import Path

from modal import App, CloudBucketMount, Image, Secret
```

```
image = Image.debian_slim().pip_install(
    "requests==2.31.0", "duckdb==0.10.0", "matplotlib==3.8.3"
)
app = App(image=image)  # Note: prior to April 2024, "app" was called "stub"

MOUNT_PATH: Path = Path("/bucket")
YELLOW_TAXI_DATA_PATH: Path = MOUNT_PATH / "yellow_taxi"
```

The dependencies installed above are not available locally. The following block instructs Modal to only import them inside the container.

```
with image.imports():
    import duckdb
    import requests
```

Download New York City's taxi data

NYC makes data about taxi rides publicly available. The city's Taxi & Limousine Commission (TLC) publishes files in the Parquet format. Files are organized by year and month.

We are going to download all available files and store them in an S3 bucket. We do this by attaching a `modal.CloudBucketMount` with the S3 bucket name and its respective credentials. The files in the bucket will then be available at `MOUNT_PATH`.

As we'll see below, this operation can be massively sped up by running it in parallel on Modal.

```
@app.function(
    volumes={
        MOUNT_PATH: CloudBucketMount(
            "modal-s3mount-test-bucket",
            secret=Secret.from_name("s3-bucket-secret"),
        )
    },
)
def download_data(year: int, month: int) -> str:
    filename = f"yellow_tripdata_{year}-{month:02d}.parquet"
    url = f"https://d37ci6vzurychx.cloudfront.net/trip-data/{filename}"
    s3_path = MOUNT_PATH / filename
    # Skip downloading if file exists.
    if not s3_path.exists():
        if not YELLOW_TAXI_DATA_PATH.exists():
            YELLOW_TAXI_DATA_PATH.mkdir(parents=True, exist_ok=True)
            with requests.get(url, stream=True) as r:
                r.raise_for_status()
                print(f"downloading => {s3_path}")
```

```
                # It looks like we writing locally, but this is actually writing to S3!
                with open(s3_path, "wb") as file:
                    for chunk in r.iter_content(chunk_size=8192):
                        file.write(chunk)

    return s3_path.as_posix()
```

Analyze data with DuckDB

DuckDB is an analytical database with rich support for Parquet files. It is also very fast. Below, we define a Modal Function that aggregates yellow taxi trips within a month (each file contains all the rides from a specific month).

```
@app.function(
    volumes={
        MOUNT_PATH: CloudBucketMount(
            "modal-s3mount-test-bucket",
            secret=Secret.from_name("s3-bucket-secret"),
        )
    },
)
def aggregate_data(path: str) -> list[tuple[datetime, int]]:
    print(f"processing => {path}")

    # Parse file.
    year_month_part = path.split("yellow_tripdata_")[1]
    year, month = year_month_part.split("-")
    month = month.replace(".parquet", "")

    # Make DuckDB query using in-memory storage.
    con = duckdb.connect(database=":memory:")
    q = """
    with sub as (
        select tpep_pickup_datetime::date d, count(1) c
        from read_parquet(?)
        group by 1
    )
    select d, c from sub
    where date_part('year', d) = ?  -- filter out garbage
    and date_part('month', d) = ?   -- same
    """
    con.execute(q, (path, year, month))
    return list(con.fetchall())
```

Plot daily taxi rides

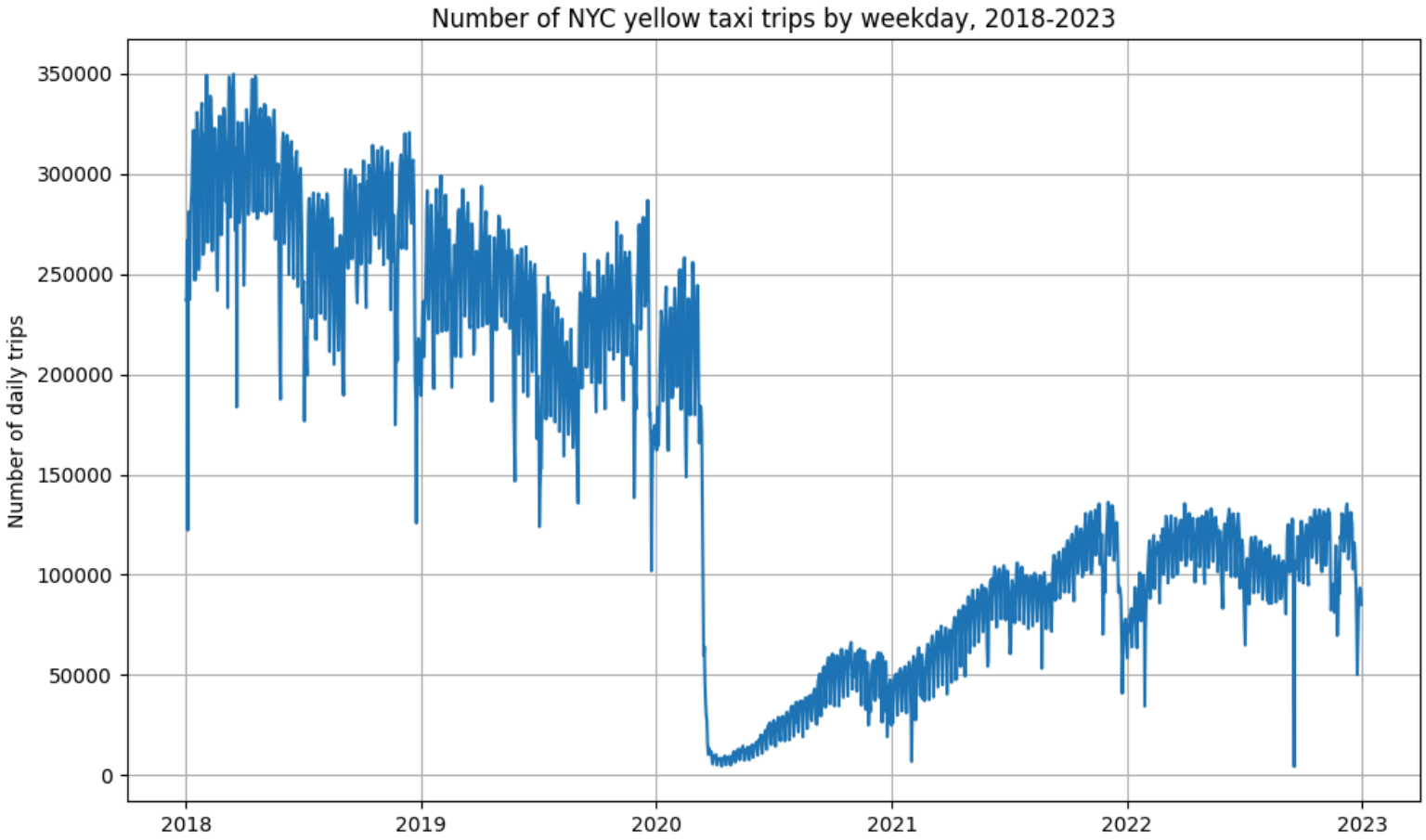
Finally, we want to plot our results. The plot created shows the number of yellow taxi rides per day in NYC. This function runs remotely, on Modal, so we don't need to install plotting libraries locally.

```
@app.function()
def plot(dataset) -> bytes:
    import io

    import matplotlib.pyplot as plt

    # Sorting data by date
    dataset.sort(key=lambda x: x[0])

    # Unpacking dates and values
    dates, values = zip(*dataset)

    # Plotting
    plt.figure(figsize=(10, 6))
    plt.plot(dates, values)
    plt.title("Number of NYC yellow taxi trips by weekday, 2018-2023")
    plt.ylabel("Number of daily trips")
    plt.grid(True)
    plt.tight_layout()

    # Saving plot as raw bytes to send back
    buf = io.BytesIO()

    plt.savefig(buf, format="png")

    buf.seek(0)

    return buf.getvalue()
```

Run everything

The `@app.local_entrypoint()` defines what happens when we run our Modal program locally. We invoke it from the CLI by calling `modal run s3_bucket_mount.py`. We first call `download_data()` and `starmap` (named because it's kind of like `map(*args)`) on tuples of inputs `(year, month)`. This will download, in parallel, all yellow taxi data files into our locally mounted S3 bucket and return a list of Parquet file paths. Then, we call `aggregate_data()` with `map` on that list. These files are also read from our S3 bucket. So one function writes files to S3 and the other reads files from S3 in; both run across many files in parallel.

Finally, we call `plot` to generate the following figure:



This program should run in less than 30 seconds.

```python
@app.local_entrypoint()
def main():
    # List of tuples[year, month].
    inputs = [
        (year, month) for year in range(2018, 2023) for month in range(1, 13)
    ]

    # List of file paths in S3.
    parquet_files: list[str] = []
    for path in download_data.starmap(inputs):
        print(f"done => {path}")
        parquet_files.append(path)

    # List of datetimes and number of yellow taxi trips.
    dataset = []
    for r in aggregate_data.map(parquet_files):
        dataset += r

    dir = Path("/tmp") / "s3_bucket_mount"
    if not dir.exists():
        dir.mkdir(exist_ok=True, parents=True)

    figure = plot.remote(dataset)
    path = dir / "nyc_yellow_taxi_trips_s3_mount.png"
```

```
    with open(path, "wb") as file:
        print(f"Saving figure to {path}")
        file.write(figure)
```

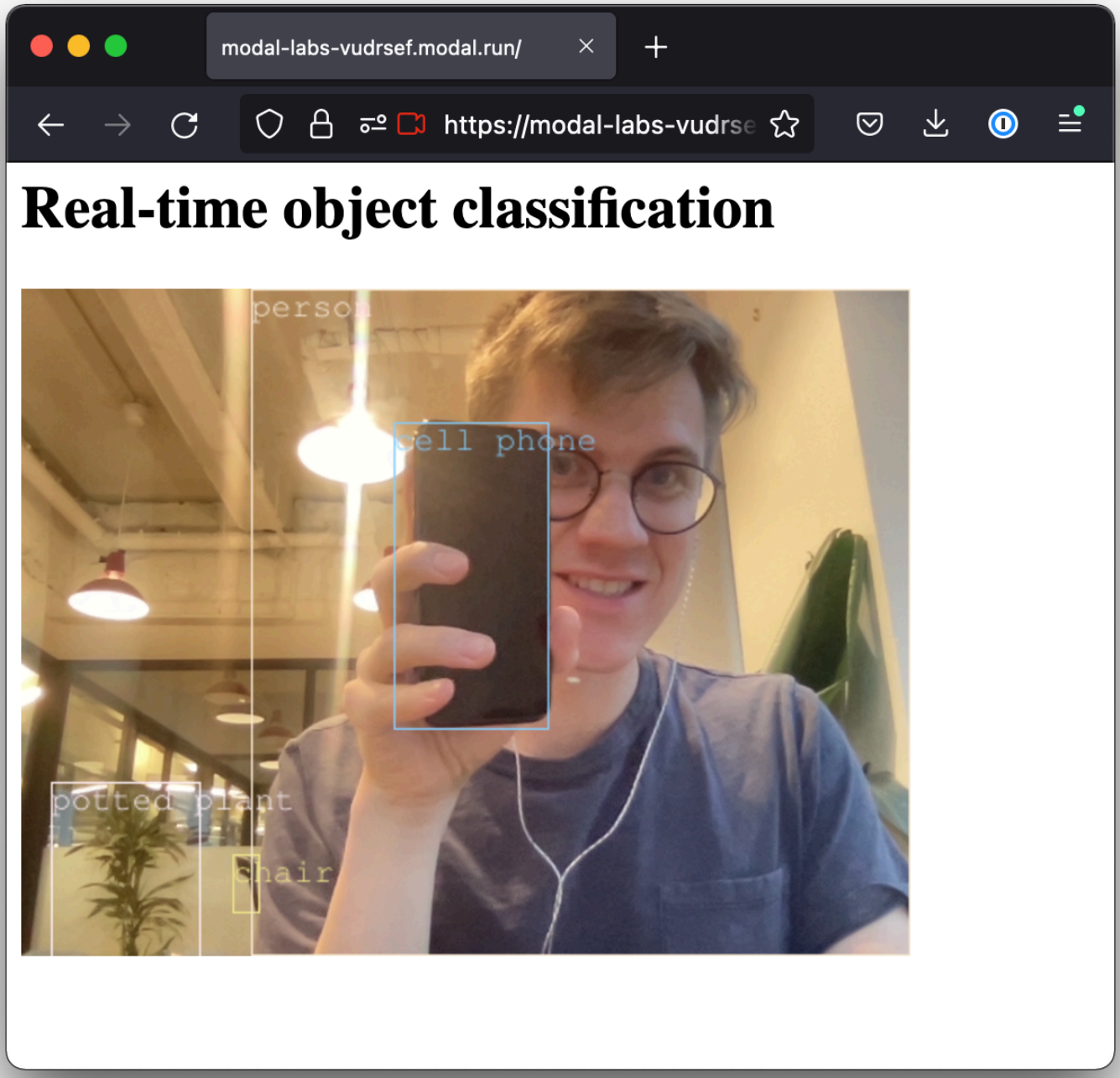
Machine learning model inference endpoint that uses the webcam

View on GitHub

This example creates a web endpoint that uses a Huggingface model for object detection.

The web endpoint takes an image from their webcam, and sends it to a Modal web endpoint. The Modal web endpoint in turn calls a Modal function that runs the actual model.

If you run this, it will look something like this:



Live demo

Take a look at the deployed app.

A couple of caveats:

- This is not optimized for latency: every prediction takes about 1s, and there's an additional overhead on the first prediction since the containers have to be started and the model initialized.
- This doesn't work on iPhone unfortunately due to some issues with HTML5 webcam components

Code

Starting with imports:

```
import base64
import io
from pathlib import Path

from fastapi import FastAPI, Request, Response
from fastapi.staticfiles import StaticFiles
from modal import App, Image, Mount, asgi_app, build, enter, method
```

We need to install transformers which is a package Huggingface uses for all their models, but also Pillow which lets us work with images from Python, and a system font for drawing.

This example uses the `facebook/detr-resnet-50` pre-trained model, which is downloaded once at image build time using the `@build` hook and saved into the image. 'Baking' models into the `modal.Image` at build time provided the fastest cold start.

```
model_repo_id = "facebook/detr-resnet-50"


app = App(
    "example-webcam-object-detection"
)  # Note: prior to April 2024, "app" was called "stub"
image = (
    Image.debian_slim()
    .pip_install(
        "huggingface-hub==0.16.4",
        "Pillow",
        "timm",
        "transformers",
    )
    .apt_install("fonts-freefont-ttf")
)
```

Prediction function

The object detection function has a few different features worth mentioning:

- There's a container initialization step in the method decorated with `@enter()`, which runs on every container start. This lets us load the model only once per container, so that it's reused for subsequent function calls.

- Above we stored the model in the container image. This lets us download the model only when the image is (re)built, and not everytime the function is called.
- We're running it on multiple CPUs for extra performance

Note that the function takes an image and returns a new image. The input image is from the webcam The output image is an image with all the bounding boxes and labels on them, with an alpha channel so that most of the image is transparent so that the web interface can render it on top of the webcam view.

```
with image.imports():
    import torch
    from huggingface_hub import snapshot_download
    from PIL import Image, ImageColor, ImageDraw, ImageFont
    from transformers import DetrForObjectDetection, DetrImageProcessor


@app.cls(
    cpu=4,
    image=image,
)
class ObjectDetection:
    @build()
    def download_model(self):
        snapshot_download(repo_id=model_repo_id, cache_dir="/cache")

    @enter()
    def load_model(self):
        self.feature_extractor = DetrImageProcessor.from_pretrained(
            model_repo_id,
            cache_dir="/cache",
        )
        self.model = DetrForObjectDetection.from_pretrained(
            model_repo_id,
            cache_dir="/cache",
        )

    @method()
    def detect(self, img_data_in):
        # Based on https://huggingface.co/spaces/nateraw/detr-object-detection/blob/main/a
        # Read png from input
        image = Image.open(io.BytesIO(img_data_in)).convert("RGB")

        # Make prediction
        inputs = self.feature_extractor(image, return_tensors="pt")
        outputs = self.model(**inputs)
        img_size = torch.tensor([tuple(reversed(image.size))])
        processed_outputs = (
            self.feature_extractor.post_process_object_detection(
                outputs=outputs,
                target_sizes=img_size,
```

```
            threshold=0,
        )
    )
    output_dict = processed_outputs[0]

    # Grab boxes
    keep = output_dict["scores"] > 0.7
    boxes = output_dict["boxes"][keep].tolist()
    scores = output_dict["scores"][keep].tolist()
    labels = output_dict["labels"][keep].tolist()

    # Plot bounding boxes
    colors = list(ImageColor.colormap.values())
    font = ImageFont.truetype(
        "/usr/share/fonts/truetype/freefont/FreeMono.ttf", 18
    )
    output_image = Image.new("RGBA", (image.width, image.height))
    output_image_draw = ImageDraw.Draw(output_image)
    for _score, box, label in zip(scores, boxes, labels):
        color = colors[label % len(colors)]
        text = self.model.config.id2label[label]
        box = tuple(map(int, box))
        output_image_draw.rectangle(box, outline=color)
        output_image_draw.text(
            box[:2], text, font=font, fill=color, width=3
        )

    # Return PNG as bytes
    with io.BytesIO() as output_buf:
        output_image.save(output_buf, format="PNG")
        return output_buf.getvalue()
```

Defining the web interface

To keep things clean, we define the web endpoints separate from the prediction function. This will introduce a tiny bit of extra latency (every web request triggers a Modal function call which will call another Modal function) but in practice the overhead is much smaller than the overhead of running the prediction function etc.

We also serve a static html page which contains some tiny bit of Javascript to capture the webcam feed and send it to Modal.

```
web_app = FastAPI()
static_path = Path(__file__).with_name("webcam").resolve()
```

The endpoint for the prediction function takes an image as a data URI and returns another image, also as a data URI:

```
@web_app.post("/predict")
async def predict(request: Request):
    # Takes a webcam image as a datauri, returns a bounding box image as a datauri
    body = await request.body()
    img_data_in = base64.b64decode(body.split(b",")[1])  # read data-uri
    img_data_out = ObjectDetection().detect.remote(img_data_in)
    output_data = b"" + base64.b64encode(img_data_out)
    return Response(content=output_data)
```

Exposing the web server

Let's take the Fast API app and expose it to Modal.

```
@app.function(
    mounts=[Mount.from_local_dir(static_path, remote_path="/assets")],
)
@asgi_app()
def fastapi_app():
    web_app.mount("/", StaticFiles(directory="/assets", html=True))
    return web_app
```

Running this locally

You can run this as an ephemeral app, by running

```
modal serve webcam.py
```

A simple web scraper

In this guide we'll introduce you to Modal by writing a simple web scraper. We'll explain the foundations of a Modal application step by step.

Set up your first Modal app

Modal apps are orchestrated as Python scripts, but can theoretically run anything you can run in a container. To get you started, make sure to install the latest Modal Python package and set up an API token (the first two steps of the Getting started page).

Finding links

First, we create an empty Python file `scrape.py`. This file will contain our application code. Lets write some basic Python code to fetch the contents of a web page and print the links (href attributes) it finds in the document:

```
import re
import sys
import urllib.request


def get_links(url):
    response = urllib.request.urlopen(url)
    html = response.read().decode("utf8")
    links = []
```

```
    for match in re.finditer('href="(.*?)"', html):
        links.append(match.group(1))
    return links


if __name__ == "__main__":
    links = get_links(sys.argv[1])
    print(links)
```

Now obviously this is just pure standard library Python code, and you can run it on your machine:

```
$ python scrape.py http://example.com
['https://www.iana.org/domains/example']
```

Running it in Modal

To make the `get_links` function run in Modal instead of your local machine, all you need to do is

- Import `modal`
- Create a `modal.App` instance
- Add a `@app.function()` annotation to your function
- Replace the `if __name__ == "__main__":` block with a function decorated with `@app.local_entrypoint()`
- Call `get_links` using `get_links.remote`

```
import re
import urllib.request
import modal

app = modal.App(name="link-scraper")  # Note: prior to April 2024, "app" was called "stub"


@app.function()
def get_links(url):
    ...


@app.local_entrypoint()
def main(url):
    links = get_links.remote(url)
    print(links)
```

You can now run this with the Modal CLI, using `modal run` instead of `python`. This time, you'll see additional progress indicators while the script is running:

```
$ modal run scrape.py --url http://example.com
✓ Initialized.
✓ Created objects.
['https://www.iana.org/domains/example']
✓ App completed.
```

Custom containers

In the code above we make use of the Python standard library `urllib` library. This works great for static web pages, but many pages these days use javascript to dynamically load content, which wouldn't appear in the loaded html file. Let's use the Playwright package to instead launch a headless Chromium browser which can interpret any javascript that might be on the page.

We can pass custom container images (defined using `modal.Image`) to the `@app.function()` decorator. We'll make use of the `modal.Image.debian_slim` pre-bundled image add the shell commands to install Playwright and its dependencies:

```
playwright_image = modal.Image.debian_slim(python_version="3.10").run_commands(
    "apt-get update",
    "apt-get install -y software-properties-common",
    "apt-add-repository non-free",
    "apt-add-repository contrib",
    "pip install playwright==1.30.0",
    "playwright install-deps chromium",
    "playwright install chromium",
)
```

Note that we don't have to install Playwright or Chromium on our development machine since this will all run in Modal. We can now modify our `get_links` function to make use of the new tools:

```
@app.function(image=playwright_image)
async def get_links(cur_url: str):
    from playwright.async_api import async_playwright

    async with async_playwright() as p:
        browser = await p.chromium.launch()
        page = await browser.new_page()
        await page.goto(cur_url)
        links = await page.eval_on_selector_all("a[href]", "elements => elements.map(eleme
        await browser.close()
```

```
    print("Links", links)
    return links
```

Since Playwright has a nice async interface, we'll redeclare our `get_links` function as async (Modal works with both sync and async functions).

The first time you run the function after making this change, you'll notice that the output first shows the progress of building the custom image you specified, after which your function runs like before. This image is then cached so that on subsequent runs of the function it will not be rebuilt as long as the image definition is the same.

Scaling out

So far, our script only fetches the links for a single page. What if we want to scrape a large list of links in parallel?

We can do this easily with Modal, because of some magic: the function we wrapped with the `@app.function()` decorator is no longer an ordinary function, but a Modal Function object. This means it comes with a `map` property built in, that lets us run this function for all inputs in parallel, scaling up to as many workers as needed.

Let's change our code to scrape all urls we feed to it in parallel:

```
@app.local_entrypoint()
def main():
    urls = ["http://modal.com", "http://github.com"]
    for links in get_links.map(urls):
        for link in links:
            print(link)
```

Schedules and deployments

Let's say we want to log the scraped links daily. We move the print loop into its own Modal function and annotate it with a `modal.Period(days=1)` schedule - indicating we want to run it once per day. Since the scheduled function will not run from our command line, we also add a hard-coded list of links to crawl for now. In a more realistic setting we could read this from a database or other accessible data source.

```
@app.function(schedule=modal.Period(days=1))
def daily_scrape():
    urls = ["http://modal.com", "http://github.com"]
    for links in get_links.map(urls):
```

```
        for link in links:
            print(link)
```

To deploy this as a permanent app, run the command

```
modal deploy scrape.py
```

Running this command deploys this function and then closes immediately. We can see the deployment and all of its runs, including the printed links, on the Modal Apps page. Rerunning the script will redeploy the code with any changes you have made - overwriting an existing deploy with the same name ("link-scraper" in this case).

Integrations and Secrets

Instead of looking at the links in the run logs of our deployments, let's say we wanted to post them to our `#scraped-links` Slack channel. To do this, we can make use of the Slack API and the `slack-sdk` PyPI package.

The Slack SDK WebClient requires an API token to get access to our Slack Workspace, and since it's bad practice to hardcode credentials into application code we make use of Modal's **Secrets**. Secrets are snippets of data that will be injected as environment variables in the containers running your functions.

The easiest way to create Secrets is to go to the Secrets section of modal.com. You can both create a free-form secret with any environment variables, or make use of presets for common services. We'll use the Slack preset and after filling in the necessary information we are presented with a snippet of code that can be used to post to Slack using our credentials:

```
import os
slack_sdk_image = modal.Image.debian_slim().pip_install("slack-sdk")


@app.function(image=slack_sdk_image, secrets=[modal.Secret.from_name("my-slack-secret")])
def bot_token_msg(channel, message):
    import slack_sdk
    client = slack_sdk.WebClient(token=os.environ["SLACK_BOT_TOKEN"])
    client.chat_postMessage(channel=channel, text=message)
```

Copy that code as-is, then amend the `daily_scrape` function to call `bot_token_msg`.

```
@app.function(schedule=modal.Period(days=1))
def daily_scrape():
```

```
urls = ["http://modal.com", "http://github.com"]
for links in get_links.map(urls):
    for link in links:
        bot_token_msg.remote("scraped-links", link)
```

Note that we are freely making function calls across completely different container images, as if they were regular Python functions in the same program.

We rerun the script which overwrites the old deploy with our updated code, and now we get a daily feed of our scraped links in our Slack channel 🎉

Summary

We have shown how you can use Modal to develop distributed Python data applications using custom containers. Through simple constructs we were able to add parallel execution. With the change of a single line of code were were able to go from experimental development code to a deployed application. The full code of this example can be found here. We hope this overview gives you a glimpse of what you are able to build using Modal.

Featured examples

Serving Diffusion models

Serve Stable Diffusion XL on Modal with a number of optimizations for blazingly fast inference.

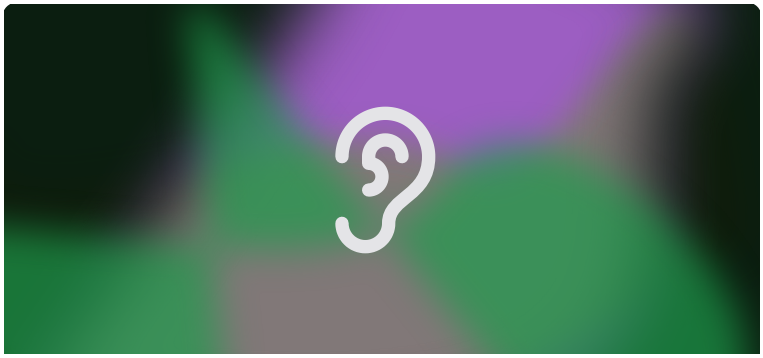
View on GitHub

Serverless TensorRT-LLM

Run large language models at SOTA performance by building a TRT-LLM engine on Modal.

View on GitHub

Stable Diffusion fine-tuning with Dreambooth



Voice chat with LLMs

Build a real-time voice chat app by combining speech-to-text, an LLM, and text-to-speech.

Fine-tune a Stable Diffusion model on images of your pet using Dreambooth.

View on GitHub

View on GitHub

Fast podcast transcriptions

Build an end-to-end podcast transcription app that leverages dozens of containers for super-fast processing.

View on GitHub

Document OCR job queue

Use Modal as an infinitely scalable job queue that can service async tasks from a web app.

View on GitHub

Parallel processing of Parquet files on S3

Analyze data from the Taxi and Limousine Commission of NYC in parallel.

View on GitHub

Retrieval-Augmented Generation for Q&A

Build a question-answering web endpoint that can cite its sources.

View on GitHub

Hacker News Slackbot

Real-time Object Detection

Use Modal to deploy a cron job that periodically queries Hacker News for new posts, and posts the results to Slack.

View on GitHub

Create a web endpoint that leverages HuggingFace models to do object detection in real-time.

View on GitHub

ControlNet playgrounds

Play with all 10 demo Gradio apps from the ControlNet project.

View on GitHub

Fast inference with vLLM (Mixtral 8x7B)

View on GitHub

In this example, we show how to run basic inference, using `vLLM` to take advantage of PagedAttention, which speeds up sequential inferences with optimized key-value caching.

We are running the Mixtral 8×7B Instruct model here, which is a mixture-of-experts model finetuned for conversation. You can expect ~3 minute cold starts. For a single request, the throughput is over 50 tokens/second. The larger the batch of prompts, the higher the throughput (up to hundreds of tokens per second).

Setup

First we import the components we need from `modal`.

```
import os
import time

import modal

MODEL_DIR = "/model"
MODEL_NAME = "mistralai/Mixtral-8x7B-Instruct-v0.1"
MODEL_REVISION = "1e637f2d7cb0a9d6fb1922f305cb784995190a83"
GPU_CONFIG = modal.gpu.A100(memory=80, count=2)
```

Define a container image

We want to create a Modal image which has the model weights pre-saved to a directory. The benefit of this is that the container no longer has to re-download the model from Huggingface - instead, it will take advantage of Modal's internal filesystem for faster cold starts.

Download the weights

We can download the model to a particular directory using the HuggingFace utility function `snapshot_download`.

For this step to work on a gated model like Mixtral 8×7B, the `HF_TOKEN` environment variable must be set.

After creating a HuggingFace access token and accepting the terms of use, head to the secrets page to share it with Modal as `huggingface-secret`.

Mixtral is beefy, at nearly 100 GB in `safetensors` format, so this can take some time — at least a few minutes.

```
def download_model_to_image(model_dir, model_name, model_revision):
    from huggingface_hub import snapshot_download
    from transformers.utils import move_cache

    os.makedirs(model_dir, exist_ok=True)

    snapshot_download(
        model_name,
        revision=model_revision,
        local_dir=model_dir,
        ignore_patterns=["*.pt", "*.bin"],  # Using safetensors
    )
    move_cache()
```

Image definition

We'll start from a Dockerhub image recommended by `vLLM`, and use run_function to run the function defined above to ensure the weights of the model are saved within the container image.

```
vllm_image = (
    modal.Image.debian_slim(python_version="3.10")
    .pip_install(
        "vllm==0.4.0.post1",
        "torch==2.1.2",
```

```
        "transformers==4.39.3",
        "ray==2.10.0",
        "hf-transfer==0.1.6",
        "huggingface_hub==0.22.2",
    )
    .env({"HF_HUB_ENABLE_HF_TRANSFER": "1"})
    .run_function(
        download_model_to_image,
        timeout=60 * 20,
        kwargs={
            "model_dir": MODEL_DIR,
            "model_name": MODEL_NAME,
            "model_revision": MODEL_REVISION,
        },
        secrets=[modal.Secret.from_name("huggingface-secret")],
    )
)

app = modal.App(
    "example-vllm-mixtral"
)  # Note: prior to April 2024, "app" was called "stub"
```

The model class

The inference function is best represented with Modal's class syntax and the `@enter` decorator. This enables us to load the model into memory just once every time a container starts up, and keep it cached on the GPU for each subsequent invocation of the function.

The `vLLM` library allows the code to remain quite clean. We do have to patch the multi-GPU setup due to issues with Ray.

```
@app.cls(
    gpu=GPU_CONFIG,
    timeout=60 * 10,
    container_idle_timeout=60 * 10,
    allow_concurrent_inputs=10,
    image=vllm_image,
)
class Model:
    @modal.enter()
    def start_engine(self):
        from vllm.engine.arg_utils import AsyncEngineArgs
        from vllm.engine.async_llm_engine import AsyncLLMEngine

        print("🥶 cold starting inference")
        start = time.monotonic_ns()

        engine_args = AsyncEngineArgs(
```

```
            model=MODEL_DIR,
            tensor_parallel_size=GPU_CONFIG.count,
            gpu_memory_utilization=0.90,
            enforce_eager=False,  # capture the graph for faster inference, but slower col
            disable_log_stats=True,  # disable logging so we can stream tokens
            disable_log_requests=True,
        )
        self.template = "[INST] {user} [/INST]"

        # this can take some time!
        self.engine = AsyncLLMEngine.from_engine_args(engine_args)
        duration_s = (time.monotonic_ns() - start) / 1e9
        print(f"🏎️ engine started in {duration_s:.0f}s")

    @modal.method()
    async def completion_stream(self, user_question):
        from vllm import SamplingParams
        from vllm.utils import random_uuid

        sampling_params = SamplingParams(
            temperature=0.75,
            max_tokens=128,
            repetition_penalty=1.1,
        )

        request_id = random_uuid()
        result_generator = self.engine.generate(
            self.template.format(user=user_question),
            sampling_params,
            request_id,
        )
        index, num_tokens = 0, 0
        start = time.monotonic_ns()
        async for output in result_generator:
            if (
                output.outputs[0].text
                and "\ufffd" == output.outputs[0].text[-1]
            ):
                continue
            text_delta = output.outputs[0].text[index:]
            index = len(output.outputs[0].text)
            num_tokens = len(output.outputs[0].token_ids)

            yield text_delta
        duration_s = (time.monotonic_ns() - start) / 1e9

        yield (
            f"\n\tGenerated {num_tokens} tokens from {MODEL_NAME} in {duration_s:.1f}s,"
            f" throughput = {num_tokens / duration_s:.0f} tokens/second on {GPU_CONFIG}.\n
        )

    @modal.exit()
```

```
    def stop_engine(self):
        if GPU_CONFIG.count > 1:
            import ray

            ray.shutdown()
```

Run the model

We define a `local_entrypoint` to call our remote function sequentially for a list of inputs. You can run this locally with `modal run -q vllm_mixtral.py`. The `q` flag enables the text to stream in your local terminal.

```
@app.local_entrypoint()
def main():
    questions = [
        "Implement a Python function to compute the Fibonacci numbers.",
        "What is the fable involving a fox and grapes?",
        "What were the major contributing factors to the fall of the Roman Empire?",
        "Describe the city of the future, considering advances in technology, environmenta
        "What is the product of 9 and 8?",
        "Who was Emperor Norton I, and what was his significance in San Francisco's histor
    ]
    model = Model()
    for question in questions:
        print("Sending new request:", question, "\n\n")
        for text in model.completion_stream.remote_gen(question):
            print(text, end="", flush=text.endswith("\n"))
```

Deploy and invoke the model

Once we deploy this model with `modal deploy vllm_mixtral.py`, we can invoke inference from other apps, sharing the same pool of GPU containers with all other apps we might need.

```
$ python
>>> import modal
>>> f = modal.Function.lookup("example-vllm-mixtral", "Model.completion_stream")
>>> for text in f.remote_gen("What is the story about the fox and grapes?"):
>>>     print(text, end="", flush=text.endswith("\n"))
'The story about the fox and grapes ...
```

Coupling a frontend web application

We can stream inference from a FastAPI backend, also deployed on Modal.

You can try our deployment here.

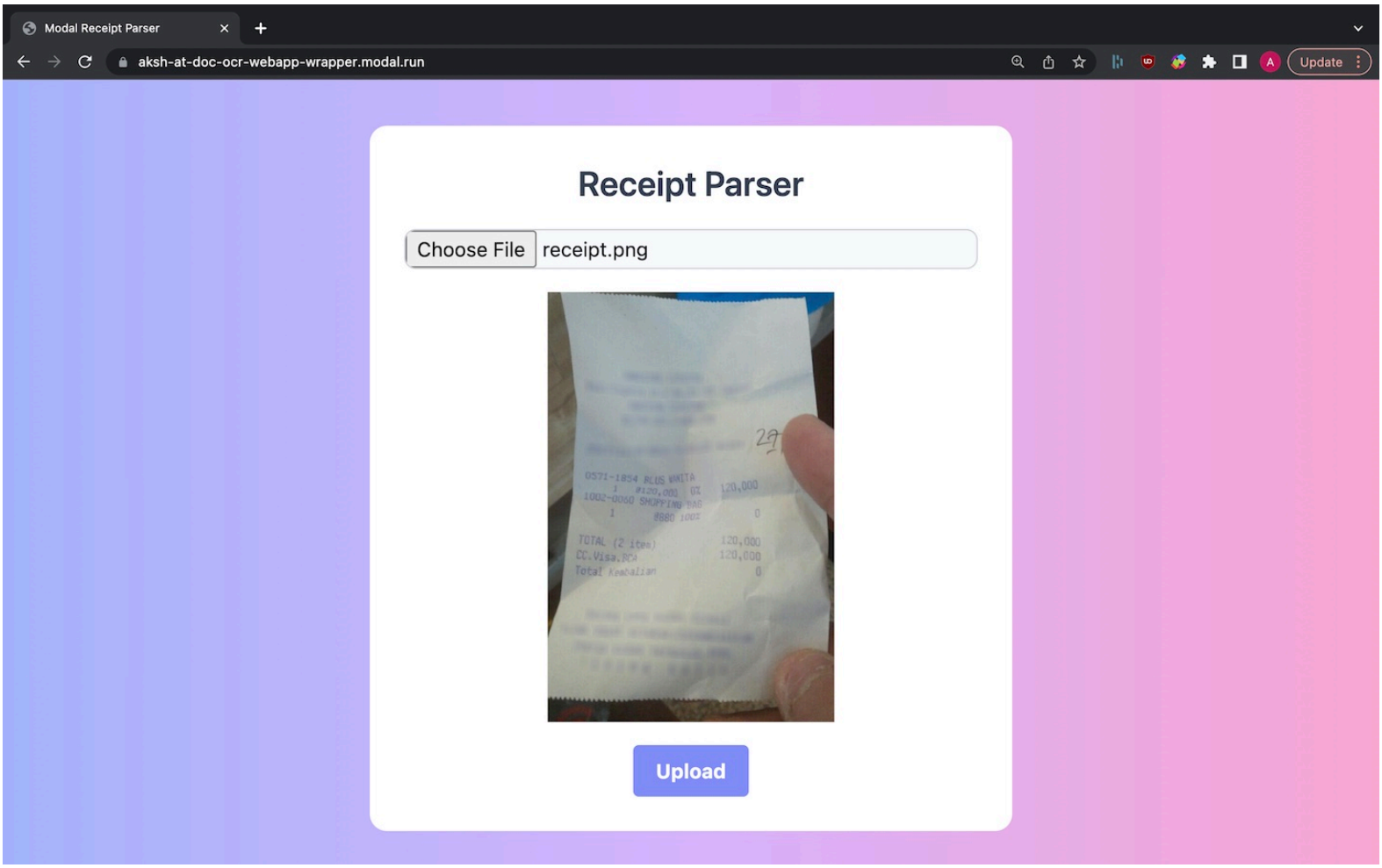
```
from pathlib import Path

from modal import Mount, asgi_app

frontend_path = Path(__file__).parent.parent / "llm-frontend"


@app.function(
    mounts=[Mount.from_local_dir(frontend_path, remote_path="/assets")],
    keep_warm=1,
    allow_concurrent_inputs=20,
    timeout=60 * 10,
)
@asgi_app()
def vllm_mixtral():
    import json

    import fastapi
    import fastapi.staticfiles
    from fastapi.responses import StreamingResponse

    web_app = fastapi.FastAPI()

    @web_app.get("/stats")
    async def stats():
        stats = await Model().completion_stream.get_current_stats.aio()
        return {
            "backlog": stats.backlog,
            "num_total_runners": stats.num_total_runners,
            "model": MODEL_NAME + " (vLLM)",
        }

    @web_app.get("/completion/{question}")
    async def completion(question: str):
        from urllib.parse import unquote

        async def generate():
            async for text in Model().completion_stream.remote_gen.aio(
                unquote(question)
            ):
                yield f"data: {json.dumps(dict(text=text), ensure_ascii=False)}\n\n"

        return StreamingResponse(generate(), media_type="text/event-stream")

    web_app.mount(
        "/", fastapi.staticfiles.StaticFiles(directory="/assets", html=True)
```

```
    )
    return web_app
```

Document OCR web app

View on GitHub

This tutorial shows you how to use Modal to deploy a fully serverless React + FastAPI application. We're going to build a simple "Receipt Parser" web app that submits OCR transcription tasks to a separate Modal app defined in the Job Queue tutorial, polls until the task is completed, and displays the results. Try it out for yourself here.



Basic setup

Let's get the imports out of the way and define a `App`.

```
from pathlib import Path

import fastapi
import fastapi.staticfiles
from modal import App, Function, Mount, asgi_app

app = App(
    "example-doc-ocr-webapp"
)  # Note: prior to April 2024, "app" was called "stub"
```

Modal works with any ASGI or WSGI web framework. Here, we choose to use FastAPI.

```
web_app = fastapi.FastAPI()
```

Define endpoints

We need two endpoints: one to accept an image and submit it to the Modal job queue, and another to poll for the results of the job.

In `parse`, we're going to submit tasks to the function defined in the Job Queue tutorial, so we import it first using `Function.lookup`.

We call `.spawn()` on the function handle we imported above, to kick off our function without blocking on the results. `spawn` returns a unique ID for the function call, that we can use later to poll for its result.

```
@web_app.post("/parse")
async def parse(request: fastapi.Request):
    parse_receipt = Function.lookup("example-doc-ocr-jobs", "parse_receipt")

    form = await request.form()
    receipt = await form["receipt"].read()  # type: ignore
    call = parse_receipt.spawn(receipt)
    return {"call_id": call.object_id}
```

`/result` uses the provided `call_id` to instantiate a `modal.FunctionCall` object, and attempt to get its result. If the call hasn't finished yet, we return a `202` status code, which indicates that the server is still working on the job.

```
@web_app.get("/result/{call_id}")
async def poll_results(call_id: str):
    from modal.functions import FunctionCall

    function_call = FunctionCall.from_id(call_id)
    try:
        result = function_call.get(timeout=0)
    except TimeoutError:
        return fastapi.responses.JSONResponse(content="", status_code=202)

    return result
```

Finally, we mount the static files for our front-end. We've made a simple React app that hits the two endpoints defined above. To package these files with our app, first we get the local assets path, and then create a modal `Mount` that mounts this directory at `/assets` inside our container. Then, we instruct FastAPI to serve this static file directory at our root path.

```
assets_path = Path(__file__).parent / "doc_ocr_frontend"


@app.function(mounts=[Mount.from_local_dir(assets_path, remote_path="/assets")])
@asgi_app()
def wrapper():
    web_app.mount(
        "/", fastapi.staticfiles.StaticFiles(directory="/assets", html=True)
    )

    return web_app
```

Running

You can run this as an ephemeral app, by running the command

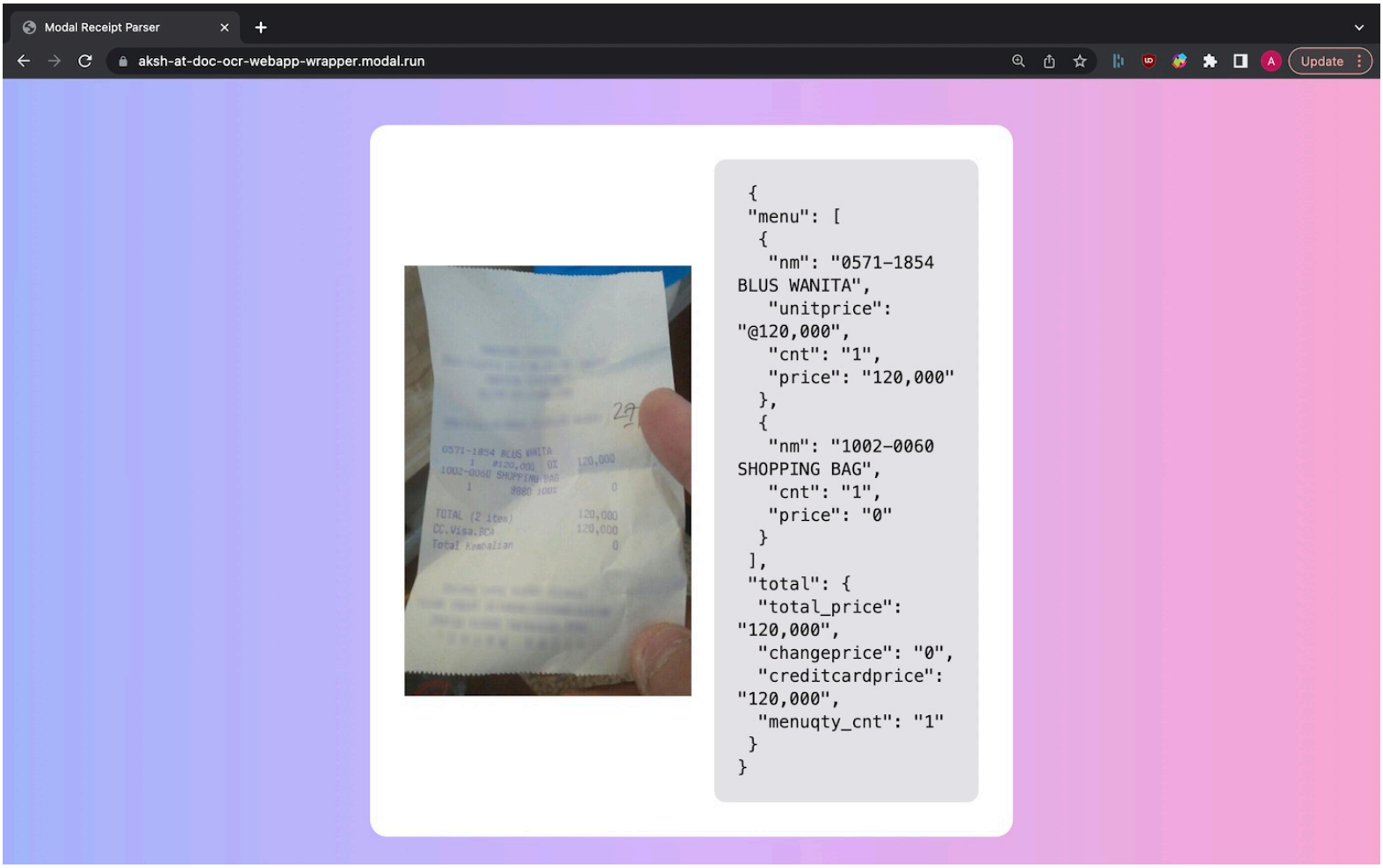
```
modal serve doc_ocr_webapp.py
```

Deploy

That's all! To deploy your application, run

```
modal deploy doc_ocr_webapp.py
```

If successful, this will print a URL for your app, that you can navigate to from your browser 🎉 .



Developing

If desired, instead of deploying, we can serve our app ephemerally. In this case, Modal watches all the mounted files, and updates the app if anything changes.

Question-answering with LangChain

View on GitHub

In this example we create a large-language-model (LLM) powered question answering web endpoint and CLI. Only a single document is used as the knowledge-base of the application, the 2022 USA State of the Union address by President Joe Biden. However, this same application structure could be extended to do question-answering over all State of the Union speeches, or other large text corpuses.

It's the LangChain library that makes this all so easy. This demo is only around 100 lines of code!

Defining dependencies

The example uses three PyPi packages to make scraping easy, and three to build and run the question-answering functionality. These are installed into a Debian Slim base image using the `pip_install` function.

Because OpenAI's API is used, we also specify the `openai-secret` Modal Secret, which contains an OpenAI API key.

A `docsearch` global variable is also declared to facilitate caching a slow operation in the code below.

```
from pathlib import Path

from modal import App, Image, Secret, web_endpoint

image = Image.debian_slim().pip_install(
```

```
    # scraping pkgs
    "beautifulsoup4~=4.11.1",
    "httpx~=0.23.3",
    "lxml~=4.9.2",
    # langchain pkgs
    "faiss-cpu~=1.7.3",
    "langchain~=0.0.138",
    "openai~=0.27.4",
    "tiktoken==0.3.0",
)
app = App(
    name="example-langchain-qanda",
    image=image,
    secrets=[Secret.from_name("openai-secret")],
)  # Note: prior to April 2024, "app" was called "stub"
docsearch = None  # embedding index that's relatively expensive to compute, so caching wit
```

Scraping the speech from whitehouse.gov

It's super easy to scrape the transcipt of Biden's speech using `httpx` and `BeautifulSoup`. This speech is just one document and it's relatively short, but it's enough to demonstrate the question-answering capability of the LLM chain.

```
def scrape_state_of_the_union() -> str:
    import httpx
    from bs4 import BeautifulSoup

    url = "https://www.whitehouse.gov/state-of-the-union-2022/"

    # fetch article; simulate desktop browser
    headers = {
        "User-Agent": "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_2) AppleWebKit/601.3.9
    }
    response = httpx.get(url, headers=headers)
    soup = BeautifulSoup(response.text, "lxml")

    # get all text paragraphs & construct string of article text
    speech_text = ""
    speech_section = soup.find_all(
        "div", {"class": "sotu-annotations__content"}
    )
    if speech_section:
        paragraph_tags = speech_section[0].find_all("p")
        speech_text = "".join([p.get_text() for p in paragraph_tags])

    return speech_text.replace("\t", "")
```

Constructing the Q&A chain

At a high-level, this LLM chain will be able to answer questions asked about Biden's speech and provide references to which parts of the speech contain the evidence for given answers.

The chain combines a text-embedding index over parts of Biden's speech with OpenAI's GPT-3 LLM. The index is used to select the most likely relevant parts of the speech given the question, and these are used to build a specialized prompt for the OpenAI language model.

For more information on this, see LangChain's "Question Answering" notebook.

```
def retrieve_sources(sources_refs: str, texts: list[str]) -> list[str]:
    """
    Map back from the references given by the LLM's output to the original text parts.
    """
    clean_indices = [
        r.replace("-pl", "").strip() for r in sources_refs.split(",")
    ]
    numeric_indices = (int(r) if r.isnumeric() else None for r in clean_indices)
    return [
        texts[i] if i is not None else "INVALID SOURCE" for i in numeric_indices
    ]


def qanda_langchain(query: str) -> tuple[str, list[str]]:
    from langchain.chains.qa_with_sources import load_qa_with_sources_chain
    from langchain.embeddings.openai import OpenAIEmbeddings
    from langchain.llms import OpenAI
    from langchain.text_splitter import CharacterTextSplitter
    from langchain.vectorstores.faiss import FAISS

    # Support caching speech text on disk.
    speech_file_path = Path("state-of-the-union.txt")

    if speech_file_path.exists():
        state_of_the_union = speech_file_path.read_text()
    else:
        print("scraping the 2022 State of the Union speech")
        state_of_the_union = scrape_state_of_the_union()
        speech_file_path.write_text(state_of_the_union)

    # We cannot send the entire speech to the model because OpenAI's model
    # has a maximum limit on input tokens. So we split up the speech
    # into smaller chunks.
    text_splitter = CharacterTextSplitter(chunk_size=1000, chunk_overlap=0)
    print("splitting speech into text chunks")
    texts = text_splitter.split_text(state_of_the_union)

    # Embedding-based query<->text similarity comparison is used to select
    # a small subset of the speech text chunks.
```

```
    # Generating the `docsearch` index is too slow to re-run on every request,
    # so we do rudimentary caching using a global variable.
    global docsearch

    if not docsearch:
        # New OpenAI accounts have a very low rate-limit for their first 48 hrs.
        # It's too low to embed even just this single Biden speech.
        # The `chunk_size` parameter is set to a low number, and internally LangChain
        # will retry the embedding requests, which should be enough to handle the rate-lim
        #
        # Ref: https://platform.openai.com/docs/guides/rate-limits/overview.
        print("generating docsearch indexer")
        docsearch = FAISS.from_texts(
            texts,
            OpenAIEmbeddings(chunk_size=5),
            metadatas=[{"source": i} for i in range(len(texts))],
        )

    print("selecting text parts by similarity to query")
    docs = docsearch.similarity_search(query)

    chain = load_qa_with_sources_chain(
        OpenAI(model_name="gpt-3.5-turbo-instruct", temperature=0),
        chain_type="stuff",
    )
    print("running query against Q&A chain.\n")
    result = chain(
        {"input_documents": docs, "question": query}, return_only_outputs=True
    )
    output: str = result["output_text"]
    parts = output.split("SOURCES: ")
    if len(parts) == 2:
        answer, sources_refs = parts
        sources = retrieve_sources(sources_refs, texts)
    elif len(parts) == 1:
        answer = parts[0]
        sources = []
    else:
        raise RuntimeError(
            f"Expected to receive an answer with a single 'SOURCES' block, got:\n{output}"
        )
    return answer.strip(), sources
```

Modal Functions

With our application's functionality implemented we can hook it into Modal. As said above, we're implementing a web endpoint, `web`, and a CLI command, `cli`.

```
@app.function()
@web_endpoint(method="GET")
def web(query: str, show_sources: bool = False):
    answer, sources = qanda_langchain(query)
    if show_sources:
        return {
            "answer": answer,
            "sources": sources,
        }
    else:
        return {
            "answer": answer,
        }


@app.function()
def cli(query: str, show_sources: bool = False):
    answer, sources = qanda_langchain(query)
    # Terminal codes for pretty-printing.
    bold, end = "\033[1m", "\033[0m"

    print(f"🦜 {bold}ANSWER:{end}")
    print(answer)
    if show_sources:
        print(f"🔗 {bold}SOURCES:{end}")
        for text in sources:
            print(text)
            print("----")
```

Test run the CLI

```
modal run potus_speech_qanda.py --query "What did the president say about Justice Breyer"
🦜 ANSWER:
The president thanked Justice Breyer for his service and mentioned his legacy of excellenc
```

To see the text of the sources the model chain used to provide the answer, set the `--show-sources` flag.

```
modal run potus_speech_qanda.py \
   --query "How many oil barrels were released from reserves" \
   --show-sources=True
```

Test run the web endpoint

Modal makes it trivially easy to ship LangChain chains to the web. We can test drive this app's web endpoint by running `modal serve potus_speech_qanda.py` and then hitting the endpoint with `curl`:

```
curl --get \
  --data-urlencode "query=What did the president say about Justice Breyer" \
  https://modal-labs--example-langchain-qanda-web.modal.run
```

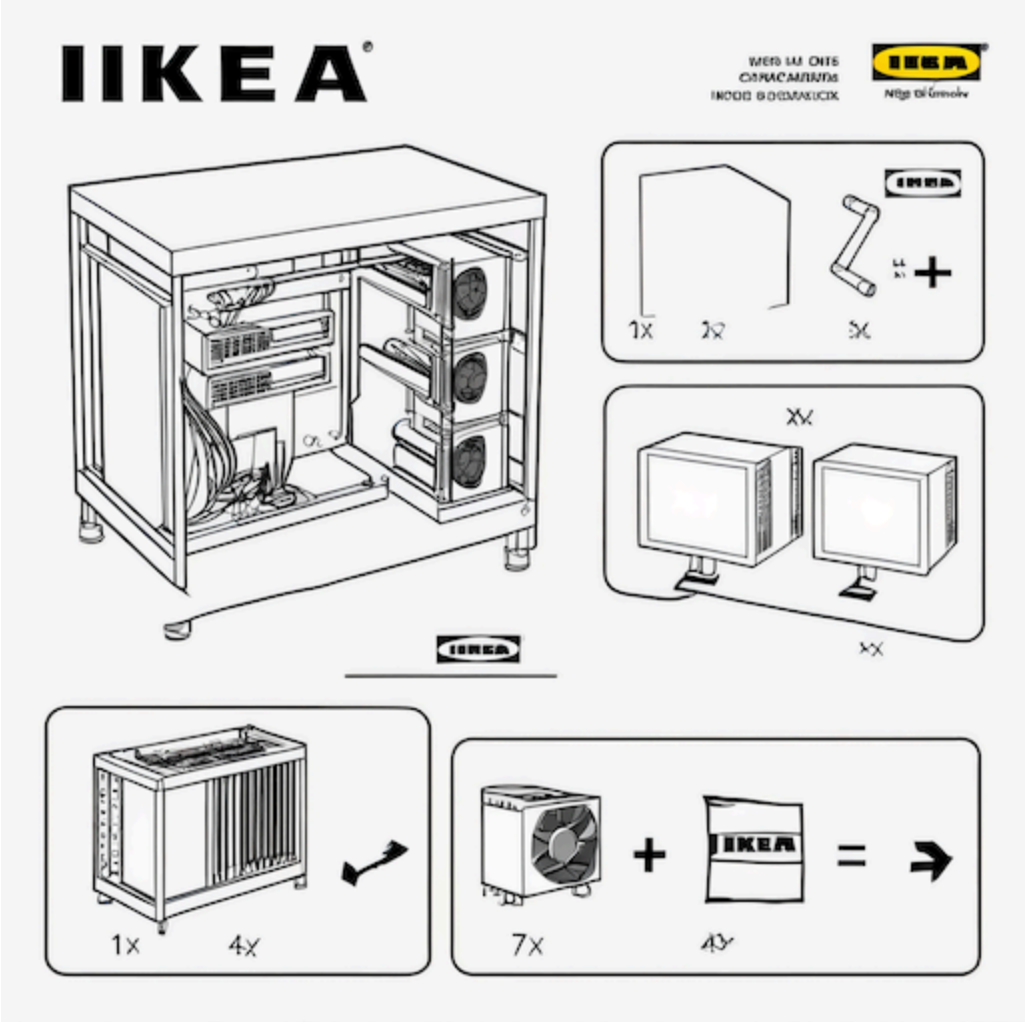
```
{
  "answer": "The president thanked Justice Breyer for his service and mentioned his legacy
}
```

LoRAs Galore: Create a LoRA Playground with Modal, Gradio, and S3

View on GitHub

This example shows how to mount an S3 bucket in a Modal app using `CloudBucketMount`. We will download a bunch of LoRA adapters from the HuggingFace Hub into our S3 bucket then read from that bucket, on the fly, when doing inference.

By default, we use the IKEA instructions LoRA as an example, which produces the following image when prompted to generate “IKEA instructions for building a GPU rig for deep learning”:



By the end of this example, we've deployed a "playground" app where anyone with a browser can try out these custom models. That's the power of Modal: custom, autoscaling AI applications, deployed in seconds. You can try out our deployment here.

Basic setup

```
import io
import os
from pathlib import Path
from typing import Optional

from modal import (
    App,
    CloudBucketMount,  # the star of the show
    Image,
    Secret,
    asgi_app,
    build,
    enter,
    method,
)
```

You will need to have an S3 bucket and AWS credentials to run this example. Refer to the documentation for the detailed IAM permissions those credentials will need.

After you are done creating a bucket and configuring IAM settings, you now need to create a Modal Secret. Navigate to the “Secrets” tab and click on the AWS card, then fill in the fields with the AWS key and secret created previously. Name the Secret `s3-bucket-secret`.

```
bucket_secret = Secret.from_name("s3-bucket-secret")

MOUNT_PATH: Path = Path("/mnt/bucket")
LORAS_PATH: Path = MOUNT_PATH / "loras/v5"
```

Modal runs serverless functions inside containers. The environments those functions run in are defined by the container `Image`. The line below constructs an image with the dependencies we need — no need to install them locally.

```
image = Image.debian_slim().pip_install(
    "huggingface_hub==0.21.4",
    "transformers==4.38.2",
    "diffusers==0.26.3",
    "peft==0.9.0",
    "accelerate==0.27.2",
)

with image.imports():
    # we import these dependencies only inside the container
    import diffusers
    import huggingface_hub
    import torch
```

We attach the S3 bucket to all the Modal functions in this app by mounting it on the filesystem they see, passing a `CloudBucketMount` to the `volumes` dictionary argument. We can read and write to this mounted bucket (almost) as if it were a local directory.

```
app = App(
    "loras-galore",
    image=image,
    volumes={
        MOUNT_PATH: CloudBucketMount(
            "modal-s3mount-test-bucket",
            secret=bucket_secret,
        )  # Note: prior to April 2024, "app" was called "stub"
    },
)
```

Acquiring LoRA weights

`search_loras()` will use the Hub API to search for LoRAs. We limit LoRAs to a maximum size to avoid downloading very large model weights. We went with 800 MiB, but feel free to adapt to what works best for you.

```
@app.function()
def search_loras(limit: int, max_model_size: int = 1024 * 1024 * 1024):
    api = huggingface_hub.HfApi()

    model_ids: list[str] = []
    for model in api.list_models(
        tags=["lora", "base_model:stabilityai/stable-diffusion-xl-base-1.0"],
        library="diffusers",
        sort="downloads",  # sort by most downloaded
    ):
        try:
            model_size = 0
            for file in api.list_files_info(model.id):
                model_size += file.size

        except huggingface_hub.utils.GatedRepoError:
            print(f"gated model ({model.id}); skipping")
            continue

        # Skip models that are larger than file limit.
        if model_size > max_model_size:
            print(f"model {model.id} is too large; skipping")
            continue

        model_ids.append(model.id)
        if len(model_ids) >= limit:
            return model_ids

    return model_ids
```

We want to take the LoRA weights we found and move them from Hugging Face onto S3, where they'll be accessible, at short latency and high throughput, for our Modal functions. Downloading files in this mount will automatically upload files to S3. To speed things up, we will run this function in parallel using Modal's `map`.

```
@app.function()
def download_lora(repository_id: str) -> Optional[str]:
    os.environ["HF_HUB_DISABLE_SYMLINKS_WARNING"] = "1"

    # CloudBucketMounts will report 0 bytes of available space leading to many
    # unnecessary warnings, so we patch the method that emits those warnings.
```

```
    from huggingface_hub import file_download

    file_download._check_disk_space = lambda x, y: False

    repository_path = LORAS_PATH / repository_id
    try:
        # skip models we've already downloaded
        if not repository_path.exists():
            huggingface_hub.snapshot_download(
                repository_id,
                local_dir=repository_path.as_posix().replace(".", "_"),
                allow_patterns=["*.safetensors"],
            )
        downloaded_lora = len(list(repository_path.rglob("*.safetensors"))) > 0
    except OSError:
        downloaded_lora = False
    except FileNotFoundError:
        downloaded_lora = False
    if downloaded_lora:
        return repository_id
    else:
        return None
```

Inference with LoRAs

We define a `StableDiffusionLoRA` class to organize our inference code. We load Stable Diffusion XL 1.0 as a base model, then, when doing inference, we load whichever LoRA the user specifies from the S3 bucket. For more on the decorators we use on the methods below to speed up building and booting, check out the container lifecycle hooks guide.

```
@app.cls(gpu="a10g")  # A10G GPUs are great for inference
class StableDiffusionLoRA:
    pipe_id = "stabilityai/stable-diffusion-xl-base-1.0"

    @build()  # when we setup our image, we download the base model
    def build(self):
        diffusers.DiffusionPipeline.from_pretrained(
            self.pipe_id, torch_dtype=torch.float16
        )

    @enter()  # when a new container starts, we load the base model into the GPU
    def load(self):
        self.pipe = diffusers.DiffusionPipeline.from_pretrained(
            self.pipe_id, torch_dtype=torch.float16
        ).to("cuda")

    @method()  # at inference time, we pull in the LoRA weights and pass the final model t
    def run_inference_with_lora(
```

```
        self, lora_id: str, prompt: str, seed: int = 8888
    ) -> bytes:
        for file in (LORAS_PATH / lora_id).rglob("*.safetensors"):
            self.pipe.load_lora_weights(lora_id, weight_name=file.name)
            break

        lora_scale = 0.9
        image = self.pipe(
            prompt,
            num_inference_steps=10,
            cross_attention_kwargs={"scale": lora_scale},
            generator=torch.manual_seed(seed),
        ).images[0]

        buffer = io.BytesIO()
        image.save(buffer, format="PNG")

        return buffer.getvalue()
```

Try it locally!

To use our inference code from our local command line, we add a `local_entrypoint` to our `app`. Run it using `modal run cloud_bucket_mount_loras.py`, and pass `--help` to see the available options.

The inference code will run on our machines, but the results will be available on yours.

```
@app.local_entrypoint()
def main(
    limit: int = 100,
    example_lora: str = "ostris/ikea-instructions-lora-sdxl",
    prompt: str = "IKEA instructions for building a GPU rig for deep learning",
    seed: int = 8888,
):
    # Download LoRAs in parallel.
    lora_model_ids = [example_lora]
    lora_model_ids += search_loras.remote(limit)

    downloaded_loras = []
    for model in download_lora.map(lora_model_ids):
        if model:
            downloaded_loras.append(model)

    print(f"downloaded {len(downloaded_loras)} loras => {downloaded_loras}")

    # Run inference using one of the downloaded LoRAs.
    byte_stream = StableDiffusionLoRA().run_inference_with_lora.remote(
        example_lora, prompt, seed
    )
```

```
    dir = Path("/tmp/stable-diffusion-xl")
    if not dir.exists():
        dir.mkdir(exist_ok=True, parents=True)

    output_path = dir / f"{as_slug(prompt.lower())}.png"
    print(f"Saving it to {output_path}")
    with open(output_path, "wb") as f:
        f.write(byte_stream)
```

LoRA Exploradora: A hosted Gradio interface

Command line tools are cool, but we can do better! With the Gradio library by Hugging Face, we can create a simple web interface around our Python inference function, then use Modal to host it for anyone to try out.

To set up your own, run `modal deploy cloud_bucket_mount_loras.py` and navigate to the URL it prints out. If you're playing with the code, use `modal serve` instead to see changes live.

```
from fastapi import FastAPI

web_app = FastAPI()
web_image = Image.debian_slim().pip_install("gradio~=3.50.2", "pillow~=10.2.0")


@app.function(image=web_image, keep_warm=1, container_idle_timeout=60 * 20)
@asgi_app()
def ui():
    """A simple Gradio interface around our LoRA inference."""
    import io

    import gradio as gr
    from gradio.routes import mount_gradio_app
    from PIL import Image

    # determine with loras are available
    lora_ids = [
        f"{lora_dir.parent.stem}/{lora_dir.stem}"
        for lora_dir in LORAS_PATH.glob("*/*")
    ]

    # pick one to be default, set a default prompt
    default_lora_id = (
        "ostris/ikea-instructions-lora-sdxl"
        if "ostris/ikea-instructions-lora-sdxl" in lora_ids
        else lora_ids[0]
    )
    default_prompt = (
        "IKEA instructions for building a GPU rig for deep learning"
```

```
        if default_lora_id == "ostris/ikea-instructions-lora-sdxl"
        else "text"
    )

    # the simple path to making an app on Gradio is an Interface: a UI wrapped around a fu
    def go(lora_id: str, prompt: str, seed: int) -> Image:
        return Image.open(
            io.BytesIO(
                StableDiffusionLoRA().run_inference_with_lora.remote(
                    lora_id, prompt, seed
                )
            ),
        )

    iface = gr.Interface(
        go,
        inputs=[  # the inputs to go/our inference function
            gr.Dropdown(
                choices=lora_ids, value=default_lora_id, label="👉 LoRA ID"
            ),
            gr.Textbox(default_prompt, label="🎨 Prompt"),
            gr.Number(value=8888, label="🎲 Random Seed"),
        ],
        outputs=gr.Image(label="Generated Image"),
        # some extra bits to make it look nicer
        title="LoRAs Galore",
        description="# Try out some of the top custom SDXL models!"
        "\n\nPick a LoRA finetune of SDXL from the dropdown, then prompt it to generate an
        "\n\nCheck out [the code on GitHub](https://github.com/modal-labs/modal-examples/b
        " if you want to create your own version or just see how it works."
        "\n\nPowered by [Modal](https://modal.com) 🚀",
        theme="soft",
        allow_flagging="never",
    )

    return mount_gradio_app(app=web_app, blocks=iface, path="/")


def as_slug(name):
    """Converts a string, e.g. a prompt, into something we can use as a filename."""
    import re

    s = str(name).strip().replace(" ", "-")
    s = re.sub(r"(?u)[^-\w.]", "", s)
    return s
```

Pet Art Dreambooth with Hugging Face and Gradio

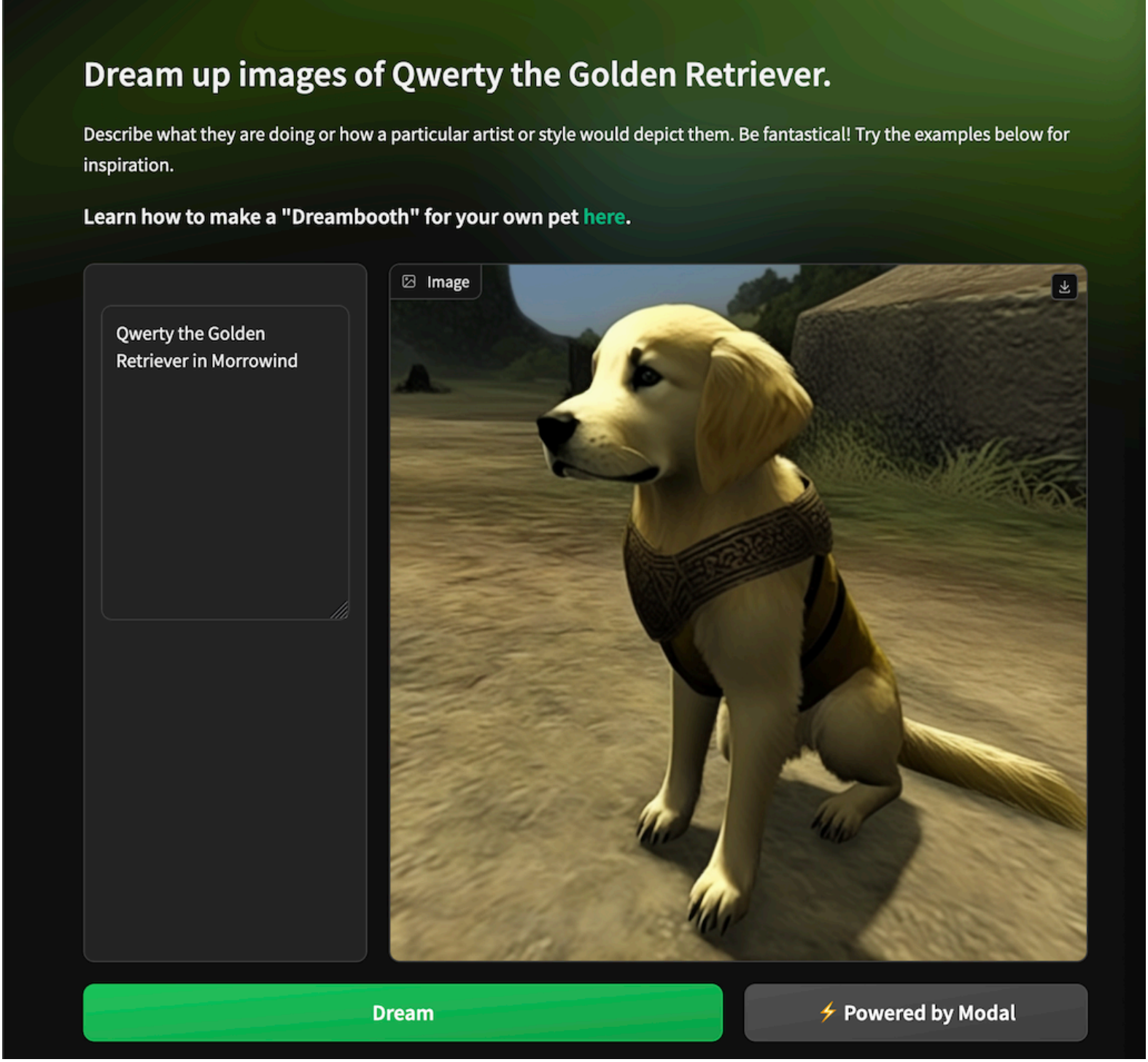
View on GitHub

This example finetunes the Stable Diffusion XL model on images of a pet (by default, a puppy named Qwerty) using a technique called textual inversion from the "Dreambooth" paper. Effectively, it teaches a general image generation model a new "proper noun", allowing for the personalized generation of art and photos.

It then makes the model shareable with others — without costing $25/day for a GPU server— by hosting a Gradio app on Modal.

It demonstrates a simple, productive, and cost-effective pathway to building on large pretrained models using Modal's building blocks, like GPU-accelerated Modal functions and classes for compute-intensive work, volumes for storage, and web endpoints for serving.

And with some light customization, you can use it to generate images of your pet!



Imports and setup

We start by importing the necessary libraries and setting up the environment. By installing Modal, we already brought in the FastAPI library we'll use to serve our app, so we import it here.

```
from dataclasses import dataclass
from pathlib import Path

from fastapi import FastAPI
from fastapi.responses import FileResponse
from modal import (
    App,
```

```
    Image,
    Mount,
    Secret,
    Volume,
    asgi_app,
    enter,
    method,
)
```

Building up the environment

Machine learning environments are complex, and the dependencies can be hard to manage. Modal makes creating and working with environments easy via containers and container images.

We start from a base image and specify all of our dependencies. We'll call out the interesting ones as they come up below. Note that these dependencies are not installed locally — they are only installed in the remote environment where our app runs.

```
app = App(
    name="example-dreambooth-app"
)  # Note: prior to April 2024, "app" was called "stub"

image = Image.debian_slim(python_version="3.10").pip_install(
    "accelerate==0.27.2",
    "datasets~=2.13.0",
    "ftfy~=6.1.0",
    "gradio~=3.50.2",
    "smart_open~=6.4.0",
    "transformers~=4.38.1",
    "torch~=2.2.0",
    "torchvision~=0.16",
    "triton~=2.2.0",
    "peft==0.7.0",
    "wandb==0.16.3",
)
```

Downloading scripts and installing a git repo with `run_commands`

We'll use an example script from the `diffusers` library to train the model. We acquire it from GitHub and install it in our environment with a series of commands. The container environments Modal functions run in are highly flexible — see the docs for more details.

```
GIT_SHA = (
    "abd922bd0c43a504e47eca2ed354c3634bd00834"  # specify the commit to fetch
)
```

```
image = (
    image.apt_install("git")
    # Perform a shallow fetch of just the target `diffusers` commit, checking out
    # the commit in the container's home directory, /root. Then install `diffusers`
    .run_commands(
        "cd /root && git init .",
        "cd /root && git remote add origin https://github.com/huggingface/diffusers",
        f"cd /root && git fetch --depth=1 origin {GIT_SHA} && git checkout {GIT_SHA}",
        "cd /root && pip install -e .",
    )
)
```

Configuration with `dataclass`es

Machine learning apps often have a lot of configuration information. We collect up all of our configuration into dataclasses to avoid scattering special/magic values throughout code.

```
@dataclass
class SharedConfig:
    """Configuration information shared across project components."""

    # The instance name is the "proper noun" we're teaching the model
    instance_name: str = "Qwerty"
    # That proper noun is usually a member of some class (person, bird),
    # and sharing that information with the model helps it generalize better.
    class_name: str = "Golden Retriever"
    # identifier for pretrained models on Hugging Face
    model_name: str = "stabilityai/stable-diffusion-xl-base-1.0"
    vae_name: str = "madebyollin/sdxl-vae-fp16-fix"  # required for numerical stability in
```

Downloading weights with `run_function`

Not everything we need for an ML app like Pet Dreambooth is available as a Python package or even on GitHub. Sometimes, there is nothing to be done but to execute some code inside the environment. We can do this on Modal with `run_function`.

In our case, we use it to download the pretrained model weights for the Stable Diffusion XL model that we'll be finetuning.

```
def download_models():
    import torch
    from diffusers import AutoencoderKL, DiffusionPipeline
    from transformers.utils import move_cache

    config = SharedConfig()
```

```
    DiffusionPipeline.from_pretrained(
        config.model_name,
        vae=AutoencoderKL.from_pretrained(
            config.vae_name, torch_dtype=torch.float16
        ),
        torch_dtype=torch.float16,
    )
    move_cache()


image = image.run_function(download_models)
```

Storing data generated by our app with `modal.Volume`

The tools we've used so far work well for fetching external information, which defines the environment our app runs in, but what about data that we create or modify during the app's execution? A persisted `modal.Volume` can store and share data across Modal apps or runs of the same app.

We'll use one to store the fine-tuned weights we create during training and then load them back in for inference.

```
volume = Volume.from_name(
    "dreambooth-finetuning-volume", create_if_missing=True
)
MODEL_DIR = "/model"
```

Load finetuning dataset

Part of the magic of the Dreambooth approach is that we only need 3-10 images for finetuning. So we can fetch just a few images, stored on consumer platforms like Imgur or Google Drive, whenever we need them — no need for expensive, hard-to-maintain data pipelines.

```
def load_images(image_urls: list[str]) -> Path:
    import PIL.Image
    from smart_open import open

    img_path = Path("/img")

    img_path.mkdir(parents=True, exist_ok=True)
    for ii, url in enumerate(image_urls):
        with open(url, "rb") as f:
            image = PIL.Image.open(f)
            image.save(img_path / f"{ii}.png")
    print(f"{ii + 1} images loaded")
```

```
    return img_path
```

Finetuning a text-to-image model

The base model we start from is trained to do a sort of "reverse ekphrasis": it attempts to recreate a visual work of art or image from only its description.

We can use the model to synthesize wholly new images by combining the concepts it has learned from the training data.

We use a pretrained model, the XL version of Stability AI's Stable Diffusion. In this example, we "finetune" SDXL, making only small adjustments to the weights. Furthermore, we don't change all the weights in the model. Instead, using a technique called *low-rank adaptation*, we change a much smaller matrix that works "alongside" the existing weights, nudging the model in the direction we want.

We can get away with such a small and simple training process because we're just teach the model the meaning of a single new word: the name of our pet.

The result is a model that can generate novel images of our pet: as an astronaut in space, as painted by Van Gogh or Bastiat, etc.

Finetuning with Hugging Face 🧨 Diffusers and Accelerate

The model weights, training libraries, and training script are all provided by 🤗 Hugging Face.

You can kick off a training job with the command `modal run dreambooth_app.py::app.train`. It should take under five minutes.

Training machine learning models takes time and produces a lot of metadata — metrics for performance and resource utilization, metrics for model quality and training stability, and model inputs and outputs like images and text. This is especially important if you're fiddling around with the configuration parameters.

This example can optionally use Weights & Biases to track all of this training information. Just sign up for an account, switch the flag below, and add your API key as a Modal secret.

```
USE_WANDB = False
```

You can see an example W&B dashboard here. Check out this run, which despite having high GPU utilization suffered from numerical instability during training and produced only black images — hard to debug without experiment management logs!

You can read more about how the values in `TrainConfig` are chosen and adjusted in this blog post on Hugging Face. To run training on images of your own pet, upload the images to separate URLs and edit the contents of the file at `TrainConfig.instance_example_urls_file` to point to them.

Tip: if the results you're seeing don't match the prompt too well, and instead produce an image of your subject without taking the prompt into account, the model has likely overfit. In this case, repeat training with a lower value of `max_train_steps`. If you used W&B, look back at results earlier in training to determine where to stop. On the other hand, if the results don't look like your subject, you might need to increase `max_train_steps`.

```
@dataclass
class TrainConfig(SharedConfig):
    """Configuration for the finetuning step."""

    # training prompt looks like `{PREFIX} {INSTANCE_NAME} the {CLASS_NAME} {POSTFIX}`
    prefix: str = "a photo of"
    postfix: str = ""

    # locator for plaintext file with urls for images of target instance
    instance_example_urls_file: str = str(
        Path(__file__).parent / "instance_example_urls.txt"
    )

    # Hyperparameters/constants from the huggingface training example
    resolution: int = 1024
    train_batch_size: int = 4
    gradient_accumulation_steps: int = 1
    learning_rate: float = 1e-4
    lr_scheduler: str = "constant"
    lr_warmup_steps: int = 0
    max_train_steps: int = 80
    checkpointing_steps: int = 1000
    seed: int = 117


@app.function(
    image=image,
    gpu="A100",  # fine-tuning is VRAM-heavy and requires an A100 GPU
    volumes={MODEL_DIR: volume},  # stores fine-tuned model
    timeout=1800,  # 30 minutes
    secrets=[Secret.from_name("my-wandb-secret")] if USE_WANDB else [],
)
def train(instance_example_urls):
    import subprocess

    from accelerate.utils import write_basic_config

    config = TrainConfig()

    # load data locally
```

```
img_path = load_images(instance_example_urls)

# set up hugging face accelerate library for fast training
write_basic_config(mixed_precision="fp16")

# define the training prompt
instance_phrase = f"{config.instance_name} the {config.class_name}"
prompt = f"{config.prefix} {instance_phrase} {config.postfix}".strip()

# the model training is packaged as a script, so we have to execute it as a subprocess
def _exec_subprocess(cmd: list[str]):
    """Executes subprocess and prints log to terminal while subprocess is running."""
    process = subprocess.Popen(
        cmd,
        stdout=subprocess.PIPE,
        stderr=subprocess.STDOUT,
    )
    with process.stdout as pipe:
        for line in iter(pipe.readline, b""):
            line_str = line.decode()
            print(f"{line_str}", end="")

    if exitcode := process.wait() != 0:
        raise subprocess.CalledProcessError(exitcode, "\n".join(cmd))

# run training -- see huggingface accelerate docs for details
print("launching dreambooth training script")
_exec_subprocess(
    [
        "accelerate",
        "launch",
        "examples/dreambooth/train_dreambooth_lora_sdxl.py",
        "--mixed_precision=fp16",  # half-precision floats most of the time for faster
        f"--pretrained_model_name_or_path={config.model_name}",
        f"--pretrained_vae_model_name_or_path={config.vae_name}",  # required for nume
        f"--instance_data_dir={img_path}",
        f"--output_dir={MODEL_DIR}",
        f"--instance_prompt={prompt}",
        f"--resolution={config.resolution}",
        f"--train_batch_size={config.train_batch_size}",
        f"--gradient_accumulation_steps={config.gradient_accumulation_steps}",
        f"--learning_rate={config.learning_rate}",
        f"--lr_scheduler={config.lr_scheduler}",
        f"--lr_warmup_steps={config.lr_warmup_steps}",
        f"--max_train_steps={config.max_train_steps}",
        f"--checkpointing_steps={config.checkpointing_steps}",
        f"--seed={config.seed}",  # increased reproducibility by seeding the RNG
    ]
    + (
        [
            "--report_to=wandb",
            f"--validation_prompt={prompt} in space",  # simple test prompt
```

```
                f"--validation_epochs={config.max_train_steps // 5}",
            ]
            if USE_WANDB
            else []
        ),
    )
    # The trained model information has been output to the volume mounted at `MODEL_DIR`.
    # To persist this data for use in our web app, we 'commit' the changes
    # to the volume.
    volume.commit()
```

Running our model

To generate images from prompts using our fine-tuned model, we define a Modal function called `inference`.

Naively, this would seem to be a bad fit for the flexible, serverless infrastructure of Modal: wouldn't you need to include the steps to load the model and spin it up in every function call?

In order to initialize the model just once on container startup, we use Modal's container lifecycle features, which require the function to be part of a class. Note that the `modal.Volume` we saved the model to is mounted here as well, so that the fine-tuned model created by `train` is available to us.

```
@app.cls(image=image, gpu="A10G", volumes={MODEL_DIR: volume})
class Model:
    @enter()
    def load_model(self):
        import torch
        from diffusers import AutoencoderKL, DiffusionPipeline

        config = TrainConfig()

        # Reload the modal.Volume to ensure the latest state is accessible.
        volume.reload()

        # set up a hugging face inference pipeline using our model
        pipe = DiffusionPipeline.from_pretrained(
            config.model_name,
            vae=AutoencoderKL.from_pretrained(
                config.vae_name, torch_dtype=torch.float16
            ),
            torch_dtype=torch.float16,
        ).to("cuda")
        pipe.load_lora_weights(MODEL_DIR)
        self.pipe = pipe

    @method()
    def inference(self, text, config):
```

```
        image = self.pipe(
            text,
            num_inference_steps=config.num_inference_steps,
            guidance_scale=config.guidance_scale,
        ).images[0]

        return image
```

Wrap the trained model in a Gradio web UI

Gradio makes it super easy to expose a model's functionality in an easy-to-use, responsive web interface.

This model is a text-to-image generator, so we set up an interface that includes a user-entry text box and a frame for displaying images.

We also provide some example text inputs to help guide users and to kick-start their creative juices.

And we couldn't resist adding some Modal style to it as well!

You can deploy the app on Modal with the command `modal deploy dreambooth_app.py`. You'll be able to come back days, weeks, or months later and find it still ready to do, even though you don't have to pay for a server to run while you're not using it.

```
web_app = FastAPI()
assets_path = Path(__file__).parent / "assets"


@dataclass
class AppConfig(SharedConfig):
    """Configuration information for inference."""

    num_inference_steps: int = 25
    guidance_scale: float = 7.5


@app.function(
    image=image,
    concurrency_limit=3,
    mounts=[Mount.from_local_dir(assets_path, remote_path="/assets")],
)
@asgi_app()
def fastapi_app():
    import gradio as gr
    from gradio.routes import mount_gradio_app

    # Call out to the inference in a separate Modal environment with a GPU
```

```
    def go(text=""):
        if not text:
            text = example_prompts[0]
        return Model().inference.remote(text, config)

    # set up AppConfig
    config = AppConfig()

    instance_phrase = f"{config.instance_name} the {config.class_name}"

    example_prompts = [
        f"{instance_phrase}",
        f"a painting of {instance_phrase.title()} With A Pearl Earring, by Vermeer",
        f"oil painting of {instance_phrase} flying through space as an astronaut",
        f"a painting of {instance_phrase} in cyberpunk city. character design by cory loft
        f"drawing of {instance_phrase} high quality, cartoon, path traced, by studio ghibl
    ]

    modal_docs_url = "https://modal.com/docs/guide"
    modal_example_url = f"{modal_docs_url}/examples/dreambooth_app"

    description = f"""Describe what they are doing or how a particular artist or style wou

### Learn how to make a "Dreambooth" for your own pet [here]({modal_example_url}).
    """

    # custom styles: an icon, a background, and a theme
    @web_app.get("/favicon.ico", include_in_schema=False)
    async def favicon():
        return FileResponse("/assets/favicon.svg")

    @web_app.get("/assets/background.svg", include_in_schema=False)
    async def background():
        return FileResponse("/assets/background.svg")

    with open("/assets/index.css") as f:
        css = f.read()

    theme = gr.themes.Default(
        primary_hue="green", secondary_hue="emerald", neutral_hue="neutral"
    )

    # add a gradio UI around inference
    with gr.Blocks(
        theme=theme, css=css, title="Pet Dreambooth on Modal"
    ) as interface:
        gr.Markdown(
            f"# Dream up images of {instance_phrase}.\n\n{description}",
        )
        with gr.Row():
            inp = gr.Textbox(  # input text component
                label="",
```

```
                placeholder=f"Describe the version of {instance_phrase} you'd like to see"
                lines=10,
            )
            out = gr.Image(  # output image component
                height=512, width=512, label="", min_width=512, elem_id="output"
            )
        with gr.Row():
            btn = gr.Button("Dream", variant="primary", scale=2)
            btn.click(
                fn=go, inputs=inp, outputs=out
            )  # connect inputs and outputs with inference function

            gr.Button(  # shameless plug
                "⚡ Powered by Modal",
                variant="secondary",
                link="https://modal.com",
            )

        with gr.Column(variant="compact"):
            # add in a few examples to inspire users
            for ii, prompt in enumerate(example_prompts):
                btn = gr.Button(prompt, variant="secondary")
                btn.click(fn=lambda idx=ii: example_prompts[idx], outputs=inp)

    # mount for execution on Modal
    return mount_gradio_app(
        app=web_app,
        blocks=interface,
        path="/",
    )
```

Running your own Dreambooth from the command line

You can use the `modal` command-line interface to set up, customize, and deploy this app:

- `modal run dreambooth_app.py` will train the model. Change the `instance_example_urls_file` to point to your own pet's images.
- `modal serve dreambooth_app.py` will serve the Gradio interface at a temporary location. Great for iterating on code!
- `modal shell dreambooth_app.py` is a convenient helper to open a bash shell in our image. Great for debugging environment issues.

Remember, once you've trained your own fine-tuned model, you can deploy it using `modal deploy dreambooth_app.py`.

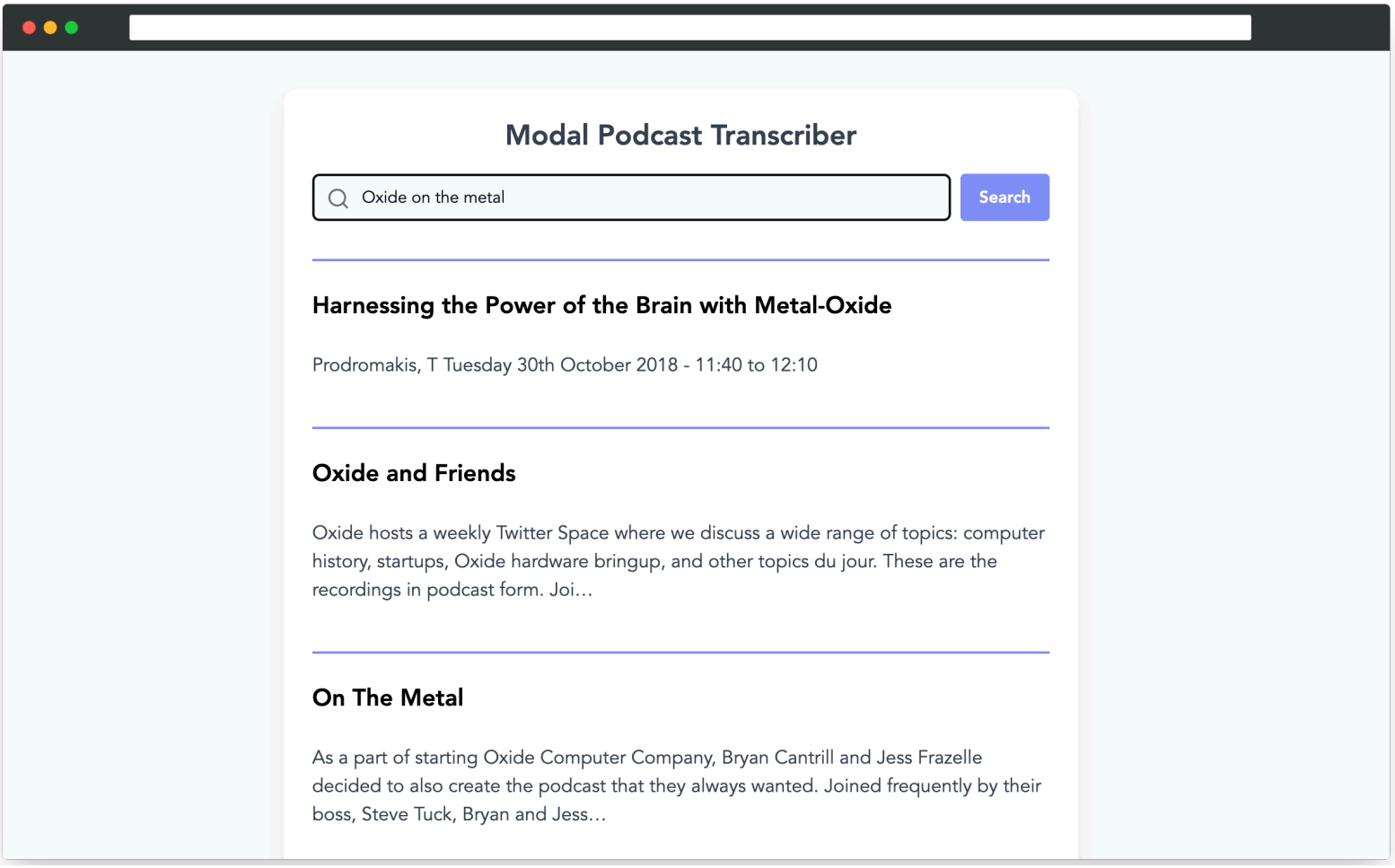
If you just want to try the app out, you can find it at https://modal-labs-example-dreambooth-app-fastapi-app.modal.run

```python
@app.local_entrypoint()
def run():
    with open(TrainConfig().instance_example_urls_file) as f:
        instance_example_urls = [line.strip() for line in f.readlines()]
    train.remote(instance_example_urls)
```

Example: Parallel podcast transcription using Whisper

OpenAI's late-September 2022 release of the *Whisper* speech recognition model was another eye-widening milestone in the rapidly improving field of deep learning, and like others we jumped to try Whisper on podcasts.

The result is the Modal Podcast Transcriber!



This example application is more feature-packed than others, and doesn't fit in a single page of code and commentary. So instead of progressing through the example's code linearly, this post provides a higher-level walkthrough of how Modal is used to do fast, on-demand podcast episode transcription for whichever podcast you'd like.

Hour-long episodes transcribed in just 1 minute

The focal point of this demonstration app is that it does serverless CPU transcription across dozens of containers at the click of a button, completing hour-long audio files in just 1 minute.

We use a podcast metadata API to allow users to transcribe an arbitrary episode from whatever niche podcast a user desires — how about *The Pen Addict*, a podcast dedicated to stationary?.

The video below shows the 45-minute long first episode of *Serial* season 2 get transcribed in 62 seconds.

0:00

What's extra cool is that each transcription segment has links back to auto-play the original audio. If you read a transcription and wonder, did they really say that? Click the timestamp on the right and find out!

(🎧 *Enable sound for this video*)

Try it yourself

If you're itching to see this in action, here are links to begin transcribing the three most popular podcasts on Spotify right now:

1. *Case 63* by Gimlet Media
2. *The Joe Rogan Experience*
3. *The Psychology of your 20s*

Tech-stack overview

The entire application is hosted serverlessly on Modal and consists of these main components:

- A React + Vite single page application (SPA) deployed as static files into a Modal web endpoint.
- A Modal web endpoint running FastAPI
- The Podchaser API provides podcast search and episode metadata retrieval. It's hooked into our code with a Modal Secret.
- A Modal async job queue, described in more detail below.

All of this is deployed with one command and costs `$0.00` when it's not transcribing podcasts or serving HTTP requests.

Speed-boosting Whisper with parallelism

Modal's dead-simple parallelism primitives are the key to doing the transcription so quickly. Even with a GPU, transcribing a full episode serially was taking around 10 minutes.

But by pulling in `ffmpeg` with a simple `.pip_install("ffmpeg-python")` addition to our Modal Image, we could exploit the natural silences of the podcast medium to partition episodes into hundreds of short segments. Each segment is transcribed by Whisper in its own container task with 2 physical CPU cores, and when all are done we stitch the segments back together with only a minimal loss in transcription quality. This approach actually accords quite well with Whisper's model architecture:

> "The Whisper architecture is a simple end-to-end approach, implemented as an encoder-decoder Transformer. Input audio is split into 30-second chunks, converted into a log-Mel spectrogram, and then passed into an encoder."
>
> —*Introducing Whisper*

Run this app on Modal

All source code for this example can be found on GitHub. The `README.md` includes instructions on setting up the frontend build and getting authenticated with the Podchaser API. Happy transcribing!

Hello, world!

View on GitHub

This tutorial demonstrates some core features of Modal:

- You can run functions on Modal just as easily as you run them locally.
- Running functions in parallel on Modal is simple and fast.
- Logs and errors show up immediately, even for functions running on Modal.

Importing Modal and setting up

We start by importing `modal` and creating a `App`. We build up from our `App` to define our application.

```
import sys

import modal

app = modal.App(
    "example-hello-world"
)  # Note: prior to April 2024, "app" was called "stub"
```

Defining a function

Modal takes code and runs it in the cloud.

So first we've got to write some code.

Let's write a simple function: log `"hello"` to standard out if the input is even or `"world"` to standard error if it's not, then return the input times itself.

To make this function work with Modal, we just wrap it in a decorator from our application `app`, `@app.function`.

```
@app.function()
def f(i):
    if i % 2 == 0:
        print("hello", i)
    else:
        print("world", i, file=sys.stderr)

    return i * i
```

Running our function locally, remotely, and in parallel

Now let's see three different ways we can call that function:

1. As a regular `local` call on your computer, with `f.local`
2. As a `remote` call that runs in the cloud, with `f.remote`
3. By `map` ping many copies of `f` in the cloud over many inputs, with `f.map`

We call `f` in each of these ways inside a `main` function below.

```
@app.local_entrypoint()
def main():
    # run the function locally
    print(f.local(1000))

    # run the function remotely on Modal
    print(f.remote(1000))

    # run the function in parallel and remotely on Modal
    total = 0
    for ret in f.map(range(20)):
        total += ret

    print(total)
```

Enter `modal run hello_world.py` in a shell and you'll see a Modal app initialize. You'll then see the `print`ed logs of the `main` function and, mixed in with them, all the logs of `f` as it is run locally, then remotely, and then remotely and in parallel.

That's all triggered by adding the `@app.local_entrypoint` decorator on `main`, which defines it as the function to start from locally when we invoke `modal run`.

What just happened?

When we called `.remote` on `f`, the function was executed **in the cloud**, on Modal's infrastructure, not locally on our computer.

In short, we took the function `f`, put it inside a container, sent it the inputs, and streamed back the logs and outputs.

But why does this matter?

Try doing one of these things next to start seeing the full power of Modal!

You can change the code and run it again

For instance, change the `print` statement in the function `f` to print `"spam"` and `"eggs"` instead and run the app again. You'll see that that your new code is run with no extra work from you — and it should even run faster!

Modal's goal is to make running code in the cloud feel like you're running code locally. That means no waiting for long image builds when you've just moved a comma, no fiddling with container image pushes, and no context-switching to a web UI to inspect logs.

You can map over more data

Change the `map` range from `20` to some large number, like `1170`. You'll see Modal create and run even more containers in parallel this time.

And it'll happen lightning fast!

You can run a more interesting function

The function `f` is obviously silly and doesn't do much, but in its place imagine something that matters to you, like:

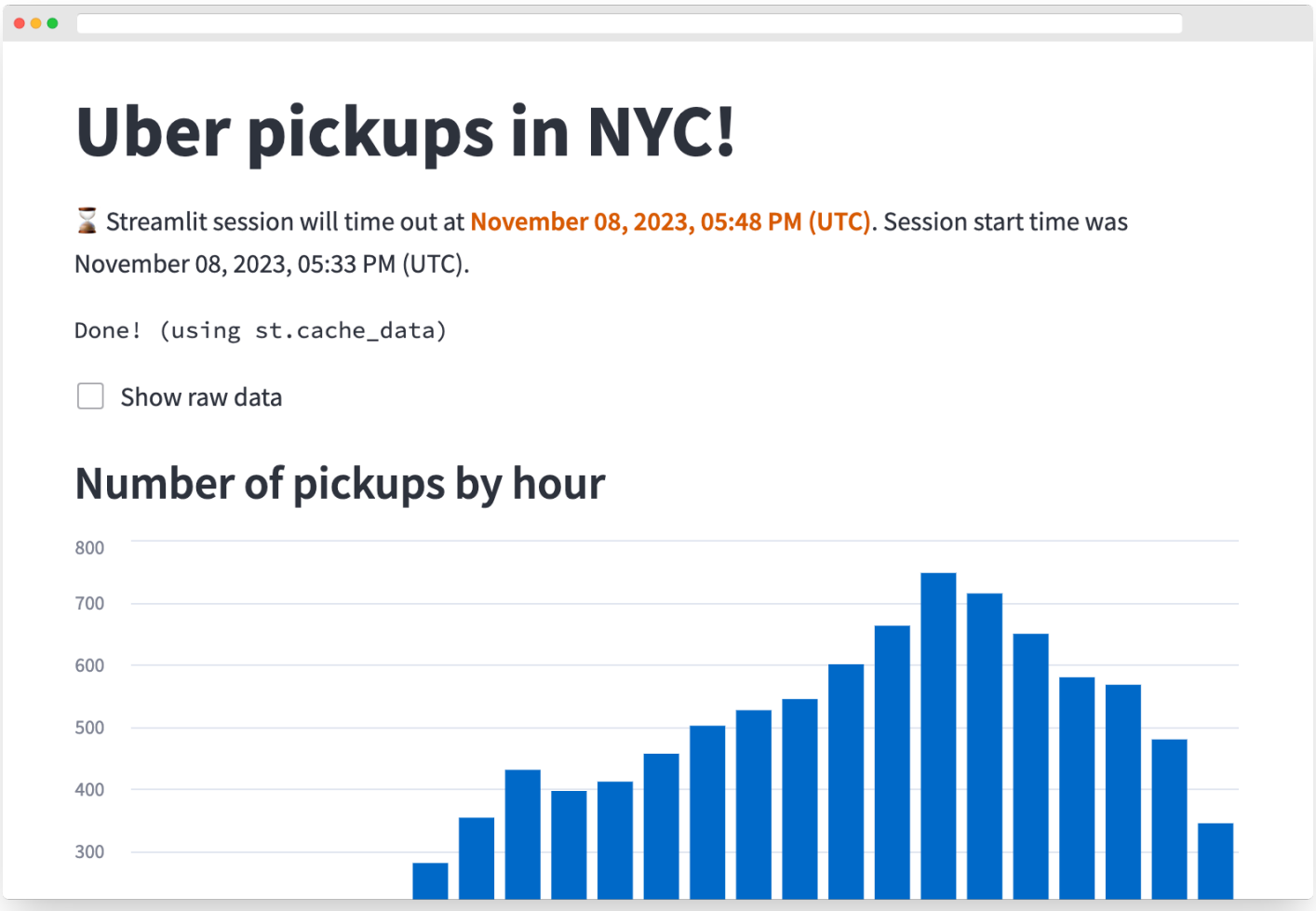
- Running language model inference or fine-tuning
- Manipulating audio or images
- Collecting financial data to backtest a trading algorithm.

Modal lets you parallelize that operation effortlessly by running hundreds or thousands of containers in the cloud.

Run and share Streamlit apps

View on GitHub

This example shows you how to run a Streamlit app with `modal serve`, and then deploy it as a serverless web app.



This example is structured as two files:

1. This module, which defines the Modal objects (name the script `serve_streamlit.py` locally).
2. `app.py`, which is any Streamlit script to be mounted into the Modal function (download script).

```
import shlex
import subprocess
from pathlib import Path

import modal
```

Define container dependencies

The `app.py` script imports three third-party packages, so we include these in the example's image definition.

```
image = modal.Image.debian_slim().pip_install("streamlit", "numpy", "pandas")

app = modal.App(
    name="example-modal-streamlit", image=image
)  # Note: prior to April 2024, "app" was called "stub"
```

Mounting the `app.py` script

We can just mount the `app.py` script inside the container at a pre-defined path using a Modal `Mount`.

```
streamlit_script_local_path = Path(__file__).parent / "app.py"
streamlit_script_remote_path = Path("/root/app.py")

if not streamlit_script_local_path.exists():
    raise RuntimeError(
        "app.py not found! Place the script with your streamlit app in the same directory."
    )

streamlit_script_mount = modal.Mount.from_local_file(
    streamlit_script_local_path,
    streamlit_script_remote_path,
)
```

Spawning the Streamlit server

Inside the container, we will run the Streamlit server in a background subprocess using `subprocess.Popen`. We also expose port 8000 using the `@web_server` decorator.

```
@app.function(
    allow_concurrent_inputs=100,
    mounts=[streamlit_script_mount],
)
@modal.web_server(8000)
def run():
    target = shlex.quote(str(streamlit_script_remote_path))
    cmd = f"streamlit run {target} --server.port 8000 --server.enableCORS=false --server.e
    subprocess.Popen(cmd, shell=True)
```

Iterate and Deploy

While you're iterating on your screamlit app, you can run it "ephemerally" with `modal serve`. This will run a local process that watches your files and updates the app if anything changes.

```
modal serve serve_streamlit.py
```

Once you're happy with your changes, you can deploy your application with

```
modal deploy serve_streamlit.py
```

If successful, this will print a URL for your app, that you can navigate to from your browser 🎉 .

Serverless TensorRT-LLM (LLaMA 3 8B)

View on GitHub

In this example, we demonstrate how to use the TensorRT-LLM framework to serve Meta's LLaMA 3 8B model at a total throughput of roughly 4,500 output tokens per second on a single NVIDIA A100 40GB GPU. At Modal's on-demand rate of ~$4/hr, that's under $0.20 per million tokens — on auto-scaling infrastructure and served via a customizable API.

Additional optimizations like speculative sampling and FP8 quantization can further improve throughput. For more on the throughput levels that are possible with TensorRT-LLM for different combinations of model, hardware, and workload, see the official benchmarks.

Overview

This guide is intended to document two things: the general process for building TensorRT-LLM on Modal and a specific configuration for serving the LLaMA 3 8B model.

Build process

Any given TensorRT-LLM service requires a multi-stage build process, starting from model weights and ending with a compiled engine. Because that process touches many sharp-edged high-performance components across the stack, it can easily go wrong in subtle and hard-to-debug ways that are idiosyncratic to specific systems. And debugging GPU workloads is expensive!

This example builds an entire service from scratch, from downloading weight tensors to responding to requests, and so serves as living, interactive documentation of a TensorRT-LLM build process that works on Modal.

Engine configuration

TensorRT-LLM is the Lamborghini of inference engines: it achieves seriously impressive performance, but only if you tune it carefully. We carefully document the choices we made here and point to additional resources so you know where and how you might adjust the parameters for your use case.

Installing TensorRT-LLM

To run TensorRT-LLM, we must first install it. Easier said than done!

In Modal, we define container images that run our serverless workloads. All Modal containers have access to GPU drivers via the underlying host environment, but we still need to install the software stack on top of the drivers, from the CUDA runtime up.

We start from the official `nvidia/cuda:12.1.1-devel-ubuntu22.04` image, which includes the CUDA runtime & development libraries and the environment configuration necessary to run them.

```
import modal

tensorrt_image = modal.Image.from_registry(
    "nvidia/cuda:12.1.1-devel-ubuntu22.04", add_python="3.10"
)
```

On top of that, we add some system dependencies of TensorRT-LLM, including OpenMPI for distributed communication, some core software like `git`, and the `tensorrt_llm` package itself.

```
tensorrt_image = tensorrt_image.apt_install(
    "openmpi-bin", "libopenmpi-dev", "git", "git-lfs", "wget"
).pip_install(
    "tensorrt_llm==0.10.0.dev2024042300",
    pre=True,
    extra_index_url="https://pypi.nvidia.com",
)
```

Note that we're doing this by method-chaining a number of calls to methods on the `modal.Image`. If you're familiar with Dockerfiles, you can think of this as a Pythonic interface to instructions like `RUN` and `CMD`.

End-to-end, this step takes five minutes. If you're reading this from top to bottom, you might want to stop here and execute the example with `modal run trtllm_llama.py` so that it runs in the background while you read the rest.

Downloading the Model

Next, we download the model we want to serve. In this case, we're using the instruction-tuned version of Meta's Llama 3 8B model. We use the function below to download the model from the Hugging Face Hub.

```
MODEL_DIR = "/root/model/model_input"
MODEL_ID = "meta-llama/Meta-Llama-3-8B-Instruct"
MODEL_REVISION = "7840f95a8c7a781d3f89c4818bf693431ab3119a"  # pin model revisions to prev


def download_model():
    import os

    from huggingface_hub import snapshot_download
    from transformers.utils import move_cache

    os.makedirs(MODEL_DIR, exist_ok=True)
    snapshot_download(
        MODEL_ID,
        local_dir=MODEL_DIR,
        ignore_patterns=["*.pt", "*.bin"],  # using safetensors
        revision=MODEL_REVISION,
    )
    move_cache()
```

Just defining that function doesn't actually download the model, though. We can run it by adding it to the image's build process with `run_function`. The download process has its own dependencies, which we add here.

```
MINUTES = 60  # seconds
tensorrt_image = (  # update the image by downloading the model we're using
    tensorrt_image.pip_install(  # add utilities for downloading the model
        "hf-transfer==0.1.6",
        "huggingface_hub==0.22.2",
        "requests~=2.31.0",
    )
    .env(  # hf-transfer: faster downloads, but fewer comforts
        {"HF_HUB_ENABLE_HF_TRANSFER": "1"}
    )
    .run_function(  # download the model
        download_model,
```

```
        timeout=20 * MINUTES,
        secrets=[modal.Secret.from_name("huggingface-secret")],
    )
)
```

Configuring the model

Now that we have the model downloaded, we need to convert it to a format that TensorRT-LLM can use. We use a convenience script provided by the TensorRT-LLM team. This script takes a few minutes to run.

```
GIT_HASH = "71d8d4d3dc655671f32535d6d2b60cab87f36e87"
CHECKPOINT_SCRIPT_URL = f"https://raw.githubusercontent.com/NVIDIA/TensorRT-LLM/{GIT_HASH}
```

TensorRT-LLM requires that a GPU be present to load the model, even though it isn't used directly during this conversion process. We'll use a single A100-40GB GPU for this example, but we have also tested it successfully with A10G, A100-80GB, and H100 GPUs.

The most important feature to track when selecting hardware to run on is GPU RAM: larger models, longer sequences, and bigger batches all require more memory, We tuned all three to maximize throughput on this example.

The amount of GPU RAM on a single card is a tight constraint for most LLMs: RAM is measured in tens of gigabytes and models have billions of floating point parameters, each consuming one to four bytes of memory. The performance cliff if you need to spill to CPU memory is steep, so the only solution is to split the model across multiple GPUs. This is particularly important when serving larger models (e.g. 70B or 8×22B).

```
N_GPUS = 1  # Heads up: this example has not yet been tested with multiple GPUs
GPU_CONFIG = modal.gpu.A100(count=N_GPUS)
```

This is also the point where we specify the data type for this model. We use IEEE 754-compliant half-precision floats, (`float16`), because we found that it resulted in marginally higher throughput, but the model is provided in Google's `bfloat16` format. On the latest Ada Lovelace chips, you might use `float8` to reduce GPU RAM usage and speed up inference, but note that the FP8 format is very new, so expect rough edges.

```
DTYPE = "float16"
```

We put that all together with another invocation of `.run_commands` .

```
CKPT_DIR = "/root/model/model_ckpt"
tensorrt_image = (  # update the image by converting the model to TensorRT format
    tensorrt_image.run_commands(  # takes ~5 minutes
        [
            f"wget {CHECKPOINT_SCRIPT_URL} -O /root/convert_checkpoint.py",
            f"python /root/convert_checkpoint.py --model_dir={MODEL_DIR} --output_dir={CKP
            + f" --tp_size={N_GPUS} --dtype={DTYPE}",
        ],
        gpu=GPU_CONFIG,  # GPU must be present to load tensorrt_llm
    )
)
```

Compiling the engine

TensorRT-LLM achieves its high throughput primarily by compiling the model: making concrete choices of CUDA kernels to execute for each operation. These kernels are much more specific than `matrix_multiply` or `softmax` — they have names like `maxwell_scudnn_winograd_128x128_ldg1_ldg4_tile148t_nt`. They are optimized for the specific types and shapes of tensors that the model uses and for the specific hardware that the model runs on.

That means we need to know all of that information a priori — more like the original TensorFlow, which defined static graphs, than like PyTorch, which builds up a graph of kernels dynamically at runtime.

This extra layer of constraint on our LLM service is precisely what allows TensorRT-LLM to achieve its high throughput.

So we need to specify things like the maximum batch size and the lengths of inputs and outputs. The closer these are to the actual values we'll use in production, the better the throughput we'll get.

```
MAX_INPUT_LEN, MAX_OUTPUT_LEN = 256, 256
MAX_BATCH_SIZE = (
    128  # better throughput at larger batch sizes, limited by GPU RAM
)
ENGINE_DIR = "/root/model/model_output"

SIZE_ARGS = f"--max_batch_size={MAX_BATCH_SIZE} --max_input_len={MAX_INPUT_LEN} --max_outp
```

There are many additional options you can pass to `trtllm-build` to tune the engine for your specific workload. You can find the document we used for LLaMA here, which you can use to adjust the arguments to fit your workloads, e.g. adjusting rotary embeddings and block sizes for longer contexts.

We selected plugins that accelerate two core components of the model: dense matrix multiplication and attention. You can read more about the plugin options here.

```
PLUGIN_ARGS = f"--gemm_plugin={DTYPE} --gpt_attention_plugin={DTYPE}"
```

We put all of this together with another invocation of `.run_commands`.

```
tensorrt_image = (  # update the image by building the TensorRT engine
    tensorrt_image.run_commands(  # takes ~5 minutes
        [
            f"trtllm-build --checkpoint_dir {CKPT_DIR} --output_dir {ENGINE_DIR}"
            + f" --tp_size={N_GPUS} --workers={N_GPUS}"
            + f" {SIZE_ARGS}"
            + f" {PLUGIN_ARGS}"
        ],
        gpu=GPU_CONFIG,  # TRT-LLM compilation is GPU-specific, so make sure this matches
    ).env(  # show more log information from the inference engine
        {"TLLM_LOG_LEVEL": "INFO"}
    )
)
```

Serving inference at thousands of tokens per second

Now that we have the engine compiled, we can serve it with Modal by creating an `App`.

```
app = modal.App(f"example-trtllm-{MODEL_ID}", image=tensorrt_image)
```

Thanks to our custom container runtime system, even this large, many gigabyte container boots in seconds.

At container start time, we boot up the engine, which completes in under 30 seconds. Container starts are triggered when Modal scales up your infrastructure, like the first time you run this code or the first time a request comes in after a period of inactivity.

Container lifecycles in Modal are managed via our `Cls` interface, so we define one below to manage the engine and run inference. For details, see this guide.

```
@app.cls(
    gpu=GPU_CONFIG,
    secrets=[modal.Secret.from_name("huggingface-secret")],
    container_idle_timeout=10 * MINUTES,
)
class Model:
    @modal.enter()
    def load(self):
        """Loads the TRT-LLM engine and configures our tokenizer.
```

```
        The @enter decorator ensures that it runs only once per container, when it starts.
        import time

        print(
            f"{COLOR['HEADER']}🥶 Cold boot: spinning up TRT-LLM engine{COLOR['ENDC']}"
        )
        self.init_start = time.monotonic_ns()

        import tensorrt_llm
        from tensorrt_llm.runtime import ModelRunner
        from transformers import AutoTokenizer

        self.tokenizer = AutoTokenizer.from_pretrained(MODEL_ID)
        # LLaMA models do not have a padding token, so we use the EOS token
        self.tokenizer.add_special_tokens(
            {"pad_token": self.tokenizer.eos_token}
        )
        # and then we add it from the left, to minimize impact on the output
        self.tokenizer.padding_side = "left"
        self.pad_id = self.tokenizer.pad_token_id
        self.end_id = self.tokenizer.eos_token_id

        runner_kwargs = dict(
            engine_dir=f"{ENGINE_DIR}",
            lora_dir=None,
            rank=tensorrt_llm.mpi_rank(),  # this will need to be adjusted to use multiple
        )

        self.model = ModelRunner.from_dir(**runner_kwargs)

        self.init_duration_s = (time.monotonic_ns() - self.init_start) / 1e9
        print(
            f"{COLOR['HEADER']}🚀 Cold boot finished in {self.init_duration_s}s{COLOR['END
        )

    @modal.method()
    def generate(self, prompts: list[str], settings=None):
        """Generate responses to a batch of prompts, optionally with custom inference sett
        import time

        if settings is None:
            settings = dict(
                temperature=0.1,  # temperature 0 not allowed, so we set top_k to 1 to get
                top_k=1,
                stop_words_list=None,
                repetition_penalty=1.1,
            )

        settings[
            "max_new_tokens"
        ] = MAX_OUTPUT_LEN  # exceeding this will raise an error
```

```
settings["end_id"] = self.end_id
settings["pad_id"] = self.pad_id

num_prompts = len(prompts)

if num_prompts > MAX_BATCH_SIZE:
    raise ValueError(
        f"Batch size {num_prompts} exceeds maximum of {MAX_BATCH_SIZE}"
    )

print(
    f"{COLOR['HEADER']}🚀 Generating completions for batch of size {num_prompts}..
)
start = time.monotonic_ns()

parsed_prompts = [
    self.tokenizer.apply_chat_template(
        [{"role": "user", "content": prompt}],
        add_generation_prompt=True,
        tokenize=False,
    )
    for prompt in prompts
]

print(
    f"{COLOR['HEADER']}Parsed prompts:{COLOR['ENDC']}",
    *parsed_prompts,
    sep="\n\t",
)

inputs_t = self.tokenizer(
    parsed_prompts, return_tensors="pt", padding=True, truncation=False
)["input_ids"]

print(
    f"{COLOR['HEADER']}Input tensors:{COLOR['ENDC']}", inputs_t[:, :8]
)

outputs_t = self.model.generate(inputs_t, **settings)

outputs_text = self.tokenizer.batch_decode(
    outputs_t[:, 0]
)  # only one output per input, so we index with 0

responses = [
    extract_assistant_response(output_text)
    for output_text in outputs_text
]
duration_s = (time.monotonic_ns() - start) / 1e9

num_tokens = sum(
    map(lambda r: len(self.tokenizer.encode(r)), responses)
```

```
        )

        for prompt, response in zip(prompts, responses):
            print(
                f"{COLOR['HEADER']}{COLOR['GREEN']}{prompt}",
                f"\n{COLOR['BLUE']}{response}",
                "\n\n",
                sep=COLOR["ENDC"],
            )
            time.sleep(0.01)  # to avoid log truncation

        print(
            f"{COLOR['HEADER']}{COLOR['GREEN']}Generated {num_tokens} tokens from {MODEL_I
            f" throughput = {num_tokens / duration_s:.0f} tokens/second for batch of size
        )

        return responses
```

Calling our inference function

Now, how do we actually run the model?

There are two basic methods: from Python via our SDK or from anywhere, by setting up an API.

Calling inference from Python

To run our `Model`'s `.generate` method from Python, we just need to call it — with `.remote` appended to run it on Modal.

We wrap that logic in a `local_entrypoint` so you can run it from the command line with

```
modal run trtllm_llama.py
```

For simplicity, we hard-code a batch of 128 questions to ask the model.

```
@app.local_entrypoint()
def main():
    questions = [
        # Generic assistant questions
        "What are you?",
        "What can you do?",
        # Coding
        "Implement a Python function to compute the Fibonacci numbers.",
        "Write a Rust function that performs binary exponentiation.",
        "How do I allocate memory in C?",
```

"What are the differences between Javascript and Python?",
"How do I find invalid indices in Postgres?",
"How can you implement a LRU (Least Recently Used) cache in Python?",
"What approach would you use to detect and prevent race conditions in a multithrea
"Can you explain how a decision tree algorithm works in machine learning?",
"How would you design a simple key-value store database from scratch?",
"How do you handle deadlock situations in concurrent programming?",
"What is the logic behind the A* search algorithm, and where is it used?",
"How can you design an efficient autocomplete system?",
"What approach would you take to design a secure session management system in a we
"How would you handle collision in a hash table?",
"How can you implement a load balancer for a distributed system?",
"Implement a Python class for a doubly linked list.",
"Write a Haskell function that generates prime numbers using the Sieve of Eratosth
"Develop a simple HTTP server in Rust.",
Literate and creative writing
"What is the fable involving a fox and grapes?",
"Who does Harry turn into a balloon?",
"Write a story in the style of James Joyce about a trip to the Australian outback
"Write a tale about a time-traveling historian who's determined to witness the mos
"Describe a day in the life of a secret agent who's also a full-time parent.",
"Create a story about a detective who can communicate with animals.",
"What is the most unusual thing about living in a city floating in the clouds?",
"In a world where dreams are shared, what happens when a nightmare invades a peace
"Describe the adventure of a lifetime for a group of friends who found a map leadi
"Tell a story about a musician who discovers that their music has magical powers."
"In a world where people age backwards, describe the life of a 5-year-old man.",
"Create a tale about a painter whose artwork comes to life every night.",
"What happens when a poet's verses start to predict future events?",
"Imagine a world where books can talk. How does a librarian handle them?",
"Tell a story about an astronaut who discovered a planet populated by plants.",
"Describe the journey of a letter traveling through the most sophisticated postal
"Write a tale about a chef whose food can evoke memories from the eater's past.",
"Write a poem in the style of Walt Whitman about the modern digital world.",
"Create a short story about a society where people can only speak in metaphors.",
"What are the main themes in Dostoevsky's 'Crime and Punishment'?",
History and Philosophy
"What were the major contributing factors to the fall of the Roman Empire?",
"How did the invention of the printing press revolutionize European society?",
"What are the effects of quantitative easing?",
"How did the Greek philosophers influence economic thought in the ancient world?",
"What were the economic and philosophical factors that led to the fall of the Sovi
"How did decolonization in the 20th century change the geopolitical map?",
"What was the influence of the Khmer Empire on Southeast Asia's history and cultur
"What led to the rise and fall of the Mongol Empire?",
"Discuss the effects of the Industrial Revolution on urban development in 19th cen
"How did the Treaty of Versailles contribute to the outbreak of World War II?",
"What led to the rise and fall of the Mongol Empire?",
"Discuss the effects of the Industrial Revolution on urban development in 19th cen
"How did the Treaty of Versailles contribute to the outbreak of World War II?",
"Explain the concept of 'tabula rasa' in John Locke's philosophy.",
"What does Nietzsche mean by 'ressentiment'?",

```
"Compare and contrast the early and late works of Ludwig Wittgenstein. Which do yo
"How does the trolley problem explore the ethics of decision-making in critical si
# Thoughtfulness
"Describe the city of the future, considering advances in technology, environmenta
"In a dystopian future where water is the most valuable commodity, how would socie
"If a scientist discovers immortality, how could this impact society, economy, and
"What could be the potential implications of contact with an advanced alien civili
"Describe how you would mediate a conflict between two roommates about doing the d
"If you could design a school curriculum for the future, what subjects would you i
"How would society change if teleportation was invented and widely accessible?",
"Consider a future where artificial intelligence governs countries. What are the p
# Math
"What is the product of 9 and 8?",
"If a train travels 120 kilometers in 2 hours, what is its average speed?",
"Think through this step by step. If the sequence a_n is defined by a_1 = 3, a_2 =
"Think through this step by step. Calculate the sum of an arithmetic series with f
"Think through this step by step. What is the area of a triangle with vertices at
"Think through this step by step. Solve the following system of linear equations:
# Facts
"Who was Emperor Norton I, and what was his significance in San Francisco's histor
"What is the Voynich manuscript, and why has it perplexed scholars for centuries?"
"What was Project A119 and what were its objectives?",
"What is the 'Dyatlov Pass incident' and why does it remain a mystery?",
"What is the 'Emu War' that took place in Australia in the 1930s?",
"What is the 'Phantom Time Hypothesis' proposed by Heribert Illig?",
"Who was the 'Green Children of Woolpit' as per 12th-century English legend?",
"What are 'zombie stars' in the context of astronomy?",
"Who were the 'Dog-Headed Saint' and the 'Lion-Faced Saint' in medieval Christian
"What is the story of the 'Globsters', unidentified organic masses washed up on th
"Which countries in the European Union use currencies other than the Euro, and wha
# Multilingual
"战国时期最重要的人物是谁?",
"Tuende hatua kwa hatua. Hesabu jumla ya mfululizo wa kihesabu wenye neno la kwanz
"Kannst du die wichtigsten Eigenschaften und Funktionen des NMDA-Rezeptors beschre
"¿Cuáles son los principales impactos ambientales de la deforestación en la Amazon
"Décris la structure et le rôle de la mitochondrie dans une cellule.",
"Какие были социальные последствия Перестройки в Советском Союзе?",
# Economics and Business
"What are the principles of behavioral economics and how do they influence consume
"Discuss the impact of blockchain technology on traditional banking systems.",
"What are the long-term effects of trade wars on global economic stability?",
"What is the law of supply and demand?",
"Explain the concept of inflation and its typical causes.",
"What is a trade deficit, and why does it matter?",
"How do interest rates affect consumer spending and saving?",
"What is GDP and why is it important for measuring economic health?",
"What is the difference between revenue and profit?",
"Describe the role of a business plan in startup success.",
"How does market segmentation benefit a company?",
"Explain the concept of brand equity.",
"What are the advantages of franchising a business?",
"What are Michael Porter's five forces and how do they impact strategy for tech st
```

```
        # Science and Technology
        "Discuss the potential impacts of quantum computing on data security.",
        "How could CRISPR technology change the future of medical treatments?",
        "Explain the significance of graphene in the development of future electronics.",
        "How do renewable energy sources compare to fossil fuels in terms of environmental
        "What are the most promising technologies for carbon capture and storage?",
        "Explain why the sky is blue.",
        "What is the principle behind the operation of a microwave oven?",
        "How does Newton's third law apply to rocket propulsion?",
        "What causes iron to rust?",
        "Describe the process of photosynthesis in simple terms.",
        "What is the role of a catalyst in a chemical reaction?",
        "What is the basic structure of a DNA molecule?",
        "How do vaccines work to protect the body from disease?",
        "Explain the significance of mitosis in cellular reproduction.",
        "What are tectonic plates and how do they affect earthquakes?",
        "How does the greenhouse effect contribute to global warming?",
        "Describe the water cycle and its importance to Earth's climate.",
        "What causes the phases of the Moon?",
        "How do black holes form?",
        "Explain the significance of the Big Bang theory.",
        "What is the function of the CPU in a computer system?",
        "Explain the difference between RAM and ROM.",
        "How does a solid-state drive (SSD) differ from a hard disk drive (HDD)?",
        "What role does the motherboard play in a computer system?",
        "Describe the purpose and function of a GPU.",
        "What is TensorRT? What role does it play in neural network inference?",
    ]

    model = Model()
    model.generate.remote(questions)
    # if you're calling this service from another Python project,
    # use [`Model.lookup`](https://modal.com/docs/reference/modal.Cls#lookup)
```

Calling inference via an API

We can use `modal.web_endpoint` and `app.function` to turn any Python function into a web API.

This API wrapper doesn't need all the dependencies of the core inference service, so we switch images here to a basic Linux image, `debian_slim`, which has everything we need.

```
web_image = modal.Image.debian_slim(python_version="3.10")
```

From there, we can take the same remote generation logic we used in `main` and serve it with only a few more lines of code.

```
@app.function(image=web_image)
@modal.web_endpoint(method="POST")
def generate_web(data: dict):
    return Model.generate.remote(data["prompts"], settings=None)
```

To set our function up as a web endpoint, we need to run this file — with `modal serve` to create a hot-reloading development server or `modal deploy` to deploy it to production.

```
modal serve trtllm_llama.py
```

You can test the endpoint by sending a POST request with `curl` from another terminal:

```
curl -X POST url-from-output-of-modal-serve-here \
-H "Content-Type: application/json" \
-d '{
    "prompts": ["Tell me a joke", "Describe a dream you had recently", "Share your favorit
}' | python -m json.tool # python for pretty-printing, optional
```

And now you have a high-throughput, low-latency, autoscaling API for serving LLaMA 3 8B completions!

Footer

The rest of the code in this example is utility code.

```
COLOR = {
    "HEADER": "\033[95m",
    "BLUE": "\033[94m",
    "GREEN": "\033[92m",
    "RED": "\033[91m",
    "ENDC": "\033[0m",
}
```

```
def extract_assistant_response(output_text):
    """Model-specific code to extract model responses.

    See this doc for LLaMA 3: https://llama.meta.com/docs/model-cards-and-prompt-formats/m
    # Split the output text by the assistant header token
    parts = output_text.split("<|start_header_id|>assistant<|end_header_id|>")

    if len(parts) > 1:
        # Join the parts after the first occurrence of the assistant header token
        response = parts[1].split("<|eot_id|>")[0].strip()
```

```
        # Remove any remaining special tokens and whitespace
        response = response.replace("<|eot_id|>", "").strip()

        return response
    else:
        return output_text
```

Fine-tune an LLM in minutes (ft. Llama 2, CodeLlama, Mistral, etc.)

View on GitHub

Tired of prompt engineering? Fine-tuning helps you get more out of a pretrained LLM by adjusting the model weights to better fit a specific task. This operational guide will help you take a base model and fine-tune it on your own dataset (API docs, conversation transcripts, etc.) in the matter of minutes.

The repository comes ready to use as-is with all the recommended, start-of-the-art optimizations for fast training results:

- Parameter-Efficient Fine-Tuning (PEFT) via LoRA adapters for faster convergence
- Flash Attention for fast and memory-efficient attention during training (note: only works with certain hardware, like A100s)
- Gradient checkpointing to reduce VRAM footprint, fit larger batches and get higher training throughput
- Distributed training via DeepSpeed so training scales optimally with multiple GPUs

The heavy lifting is done by the axolotl library. For the purposes of this guide, we'll fine-tune CodeLlama 7B to generate SQL queries, but the code is easy to tweak for many base models, datasets, and training configurations.

Best of all, using Modal for training means you never have to worry about infrastructure headaches like building images, provisioning GPUs, and managing cloud storage. If a training script runs on Modal, it's repeatable and scalable enough to ship to production right away.

Prerequisites

To follow along, make sure that you have completed the following:

1. Set up Modal account:

```
pip install modal
python3 -m modal setup
```

2. Create a HuggingFace secret in your workspace (only `HF_TOKEN` is needed, which you can get if you go into your Hugging Face settings > API tokens)

3. Clone the repository and navigate to the directory:

```
git clone https://github.com/modal-labs/llm-finetuning.git
cd llm-finetuning
```

Some models like Llama 2 also require that you apply for access, which you can do on the Hugging Face page (granted instantly).

Code overview

The source directory contains a training script to launch a training job in the cloud with your config/dataset (`config.yml` and `my_data.jsonl` , unless otherwise specified), as well as an inference engine for testing your training results.

We use Modal's built-in cloud storage system to share data across all functions in the app. In particular, we mount a persisting volume at `/pretrained` inside the container to store our pretrained models (so we only need to load them once) and another persisting volume at `/runs` to store our training config, dataset, and results for each run (for easier reproducibility and management).

Training

The training script contains three Modal functions that run in the cloud:

- `launch` prepares a new folder in the `/runs` volume with the training config and data for a new training job. It also ensures the base model is downloaded from HuggingFace.
- `train` takes a prepared run folder in the volume and performs the training job using the config and data.

- `merge` merges the trained adapter with the base model (as a CPU job).

By default, when you make local changes to either `config.yml` or `my_data.jsonl`, they will be used for your next training run. You can also specify which local config and data files to use with the `--config` and `--dataset` flags. See Making it your own for more details on customizing your dataset and config.

To kickstart a training job with the CLI, you need to specify the config and data files:

```
modal run --detach src.train --config=config/mistral.yml --data=data/sqlqa.jsonl
```

`--detach` *lets the app continue running even if your client disconnects.*

The training run folder name will be in the command output (e.g. `axo-2023-11-24-17-26-66e8`). You can check if your fine-tuned model is stored properly in this folder using `modal volume ls`.

Serving your fine-tuned model

Once a training run has completed, run inference to compare the model before/after training.

- `Inference.completion` can spawn a vLLM inference container for any pre-trained or fine-tuned model from a previous training job.

You can serve a model for inference using the following command, specifying which training run folder to load the model from with the `--run-folder` flag (run folder name is in the training log output):

```
modal run -q src.inference --run-folder /runs/axo-2023-11-24-17-26-66e8
```

We use vLLM to speed up our inference up to 24x.

Making it your own

Training on your own dataset, using a different base model, and activating another SOTA technique is as easy as modifying a couple files.

Dataset

Bringing your own dataset is as simple as creating a JSONL file — Axolotl supports many dataset formats (see more).

We recommend adding your custom dataset as a JSONL file in the `src` directory and making the appropriate modifications to your config, as explained below.

Config

All of your training parameters and options are customizable in a single config file. We recommend duplicating one of the `example_configs` to `src/config.yml` and modifying as you need. See an overview of Axolotl's config options here.

The most important options to consider are:

- Model

```yaml
base_model: codellama/CodeLlama-7b-Instruct-hf
```

- Dataset (by default we upload a local .jsonl file from the `src` folder in completion format, but you can see all dataset options here)

```yaml
datasets:
- path: my_data.jsonl
   ds_type: json
   type: completion
```

- LoRA

```yaml
adapter: lora # for qlora, or leave blank for full finetune
lora_r: 8
lora_alpha: 16
lora_dropout: 0.05
lora_target_modules:
  - q_proj
  - v_proj
```

- Multi-GPU training

  We recommend DeepSpeed for multi-GPU training, which is easy to set up. Axolotl provides several default deepspeed JSON configurations and Modal makes it easy to attach multiple GPUs of any type in code, so all you need to do is specify which of these configs you'd like to use.

  In your `config.yml`:

```
deepspeed: /root/axolotl/deepspeed/zero3.json
```

In `train.py`:

```
import os

N_GPUS = int(os.environ.get("N_GPUS", 2))
GPU_MEM = os.environ.get("GPU_MEM", "80GB")
GPU_CONFIG = modal.gpu.A100(count=N_GPUS, size=GPU_MEM)  # you can also change this in
```

- Logging with Weights and Biases

  To track your training runs with Weights and Biases:

  1. Create a Weights and Biases secret in your Modal dashboard, if not set up already (only the `WANDB_API_KEY` is needed, which you can get if you log into your Weights and Biases account and go to the Authorize page)
  2. Add the Weights and Biases secret to your app, so initializing your app in `common.py` should look like:

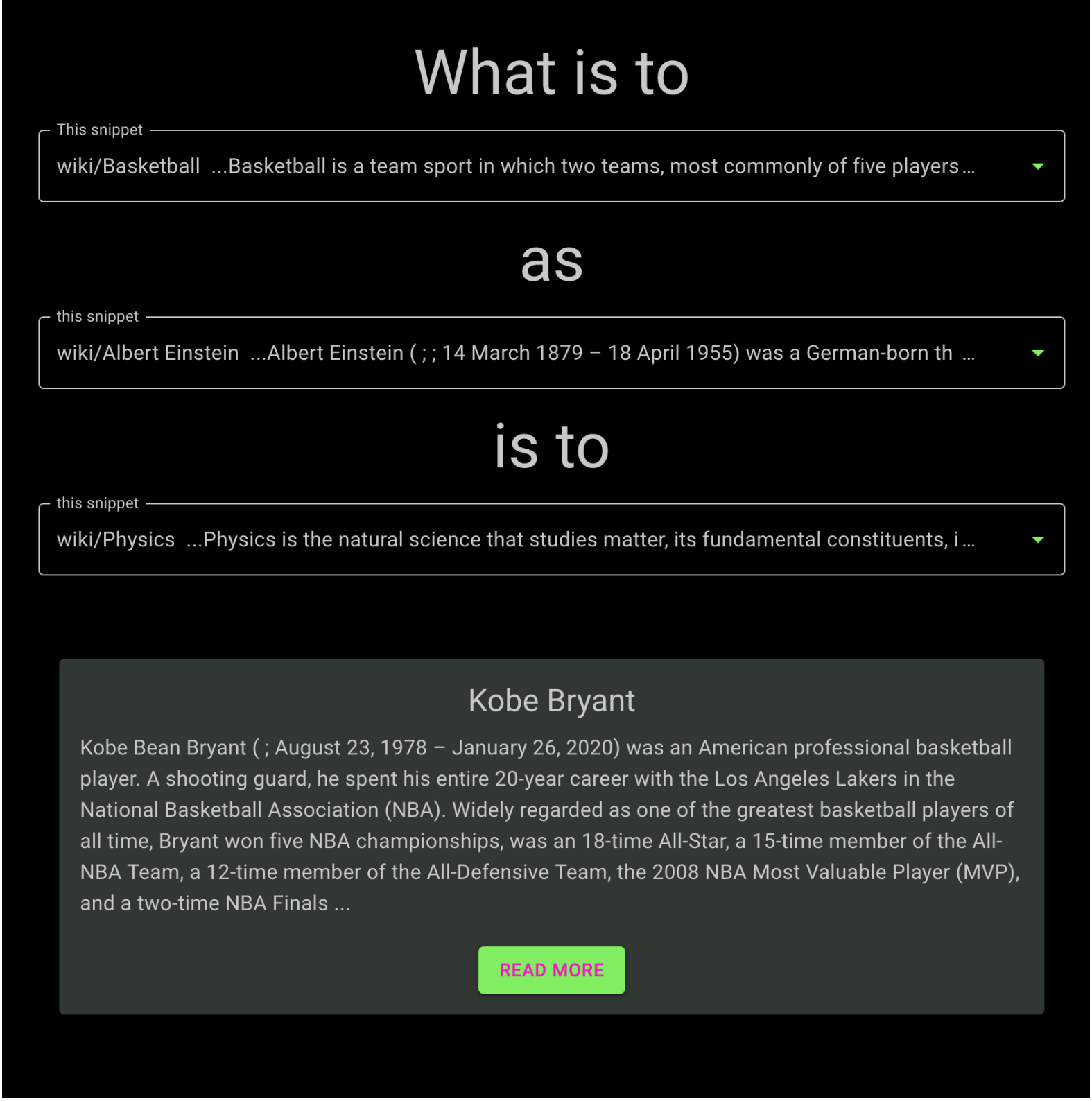
```
from modal import App, Secret

app = App("my_app", secrets=[Secret.from_name("huggingface"), Secret.from_name("my-wan
```

  3. Add your wandb config to your `config.yml`:

```
wandb_project: mistral-7b-samsum
wandb_watch: gradients
```

Once you have your trained model, you can easily deploy it to production for serverless inference via Modal's web endpoint feature (see example here). Modal will handle all the auto-scaling for you, so that you only pay for the compute you use!

Embed Wikipedia with Modal and search it with Weaviate



This sample project demonstrates the powerful combo of serverless infrastructure from Modal and the search capabilities of Weaviate for projects that combine data-intensive Python compute, like neural network inference, with data-intensive search, like indexing all of Wikipedia.

You can find the code on GitHub here. It's intended as a jumping off point for your own code that combines Modal with databases like Weaviate and with JavaScript frontends. It is also deployed as a live demo application.

Overview

The `frontend` of this project (written in React, hosted on Vercel) allows users to construct "vector analogies" of the form made famous by Word2Vec. For example, the approximation

```
Albert Einstein - Physics + Basketball ~= Kobe Bryant
```

expresses the analogy "Kobe Bryant is the Albert Einstein of basketball". We can compute it by applying those operations to embedding vectors of each concept, where `~=` is implemented using an approximate nearest-neighbor search index, the key method used for querying in vector databases.

Where Word2Vec used word embeddings to express concepts, we use snippets of Wikipedia articles. The dataset used was constructed from the March 2022 WikiMedia dump by Hugging Face.

Users can type into each search bar to find a snippet of interest, using Weaviate text search under the hood, and once they've selected the three components of their analogy, the frontend kicks off a vector search to complete it.

Both searches are coordinated by a `backend` Python service running on Modal.

Modal is also used to construct embeddings for snippets and then insert them into Weaviate. You can read more about the embedding process here. We also wrote a high level guide to this project here.

Run it yourself

The full, end-to-end version of this project involves a number of services and workflows:

1. A **Vite/React frontend**, to allow users to search for Wikipedia snippets and construct analogies via vector search
2. A **Weaviate database on Weaviate Cloud Services**, to store the Wikipedia snippets and their embeddings and to run both text and vector searches
3. A **serverless Weaviate database client on Modal**, to listen for requests from app clients, run the search logic, and communicate with the database
4. A **vector embedding service on Modal serverless GPUs**, to embed the Wikipedia snippets as vectors
5. An **ingestion workflow on Modal**, to download the Wikipedia dataset, embed it, and send the results to Weaviate

To make setup easier, we make it possible to run the search via a read-only client of our Weaviate database and run the app locally, which lets you skip the vector embedding and ingestion steps.

Set up a Python environment

Set up a Python environment however you like and then install `modal` with

```
pip install modal==0.62.140
```

Because Modal runs all of your code in cloud containers, you don't have to worry about any other dependencies!

If you don't already have a Modal account, get started with `modal setup`.

Deploy a serverless, read-only Weaviate client with Modal

Next, we set up a Weaviate client on Modal that reads from a Weaviate database that has already ingested and indexed the Wikipedia data.

Add `WCS_URL=https://gzimzbmdr6ycxyja715rsa.c0.us-west4.gcp.weaviate.cloud` and `WCS_RO_KEY=tUeQG12AkFLBY9SYOWVh2y00hZ25yu8va0UP` to a `modal.Secret` called `wiki-weaviate`.

Then, run the following command from the repo root to create a database client on Modal.

```
modal deploy backend.database
```

This client is *serverless*, meaning it scales automatically with load, including scaling down to zero instances when there is no load. That means you only need to pay for the compute resources you use!

You can run queries against the database from your local machine to test the client logic, for example

```
modal run backend.database::WeaviateClient.query --q='Albert Einstein'
```

or you can hit the API directly from your browser or with a tool like `curl` or Postman.

```
curl https://modal-labs--modal-weaviate-query.modal.run\?q\=Albert%20Einstein
```

Note that you should replace `modal-labs` in the URL with your Modal username!

This will return a large JSON object with a big vector of floating point numbers attached, so you might want to pipe it through `jq` or another JSON formatter:

```
curl https://modal-labs--modal-weaviate-query.modal.run\?q\=Albert%20Einstein \
 | jq . results\[0\].content
```

Optional: Embed and index Wikipedia yourself

If you'd like to run the entire pipeline yourself, there are several additional steps.

▶ Click here to reveal them.

Note that ingesting and indexing Wikipedia takes several hours! We **highly recommend you proceed with the read-only version first**.

Run the React frontend locally

Ensure you have a recent version of Node.js and `npm` installed. See the instructions here.

To set up the environment, run `npm install` in the `frontend` directory.

Create a file called `.env` in the `frontend` directory and set the value of `VITE_MODAL_WORKSPACE` to the name of your Modal workspace (by default, your GitHub username). See `.env.example` for the format.

Now, to run a hot-reloading, local version of the frontend, execute

```
npm run dev
```

and navigate a browser to the URL provided, which should be something like `http://localhost:5173`.

Optional: Serve a hot-reloading backend

You can also run the backend with hot reloading by using `modal serve` instead of `modal deploy`:

```
modal serve backend.database
```

This backend is still hosted on Modal, but it will automatically reload when you make changes to the code.

It uses a different URL than the deployed version, with a `-dev` appended just before `.modal.run`.

You can configure your frontend to use this backend by setting the `VITE_DEV_BACKEND` environment variable in `.env` to `true`.

Optional: Deploy the React frontend

If you'd like to share your own version of this app, you'll need to host it somewhere.

We took advantage of Vercel's excellent support for React apps to deploy directly from the GitHub repository.

Stable Diffusion XL Turbo Image-to-image

View on GitHub

This example is similar to the Stable Diffusion XL example, but it's a distilled model trained for real-time synthesis and is image-to-image. Learn more about it here.

Input prompt: `dog wizard, gandalf, lord of the rings, detailed, fantasy, cute, adorable, Pixar, Disney, 8k`

Input

Output

Basic setup

```
from io import BytesIO
from pathlib import Path

from modal import App, Image, build, enter, gpu, method
```

Define a container image

```
image = Image.debian_slim().pip_install(
    "Pillow~=10.1.0",
    "diffusers~=0.24.0",
    "transformers~=4.35.2",  # This is needed for `import torch`
    "accelerate~=0.25.0",  # Allows `device_map="auto"``, which allows computation of opti
    "safetensors~=0.4.1",  # Enables safetensor format as opposed to using unsafe pickle f
)

app = App(
    "stable-diffusion-xl-turbo", image=image
)  # Note: prior to April 2024, "app" was called "stub"

with image.imports():
```

```
import torch
from diffusers import AutoPipelineForImage2Image
from diffusers.utils import load_image
from huggingface_hub import snapshot_download
from PIL import Image
```

Load model and run inference

The container lifecycle `@enter` decorator loads the model at startup. Then, we evaluate it in the `inference` function.

To avoid excessive cold-starts, we set the idle timeout to 240 seconds, meaning once a GPU has loaded the model it will stay online for 4 minutes before spinning down. This can be adjusted for cost/experience trade-offs.

```
@app.cls(gpu=gpu.A10G(), container_idle_timeout=240)
class Model:
    @build()
    def download_models(self):
        # Ignore files that we don't need to speed up download time.
        ignore = [
            "*.bin",
            "*.onnx_data",
            "*/diffusion_pytorch_model.safetensors",
        ]

        snapshot_download("stabilityai/sdxl-turbo", ignore_patterns=ignore)

    @enter()
    def enter(self):
        self.pipe = AutoPipelineForImage2Image.from_pretrained(
            "stabilityai/sdxl-turbo",
            torch_dtype=torch.float16,
            variant="fp16",
            device_map="auto",
        )

    @method()
    def inference(self, image_bytes, prompt):
        init_image = load_image(Image.open(BytesIO(image_bytes))).resize(
            (512, 512)
        )
        num_inference_steps = 4
        strength = 0.9
        # "When using SDXL-Turbo for image-to-image generation, make sure that num_inferen
        # See: https://huggingface.co/stabilityai/sdxl-turbo
        assert num_inference_steps * strength >= 1
```

```
        image = self.pipe(
            prompt,
            image=init_image,
            num_inference_steps=num_inference_steps,
            strength=strength,
            guidance_scale=0.0,
        ).images[0]

        byte_stream = BytesIO()
        image.save(byte_stream, format="PNG")
        image_bytes = byte_stream.getvalue()

        return image_bytes


DEFAULT_IMAGE_PATH = Path(__file__).parent / "demo_images/dog.png"


@app.local_entrypoint()
def main(
    image_path=DEFAULT_IMAGE_PATH,
    prompt="dog wizard, gandalf, lord of the rings, detailed, fantasy, cute, adorable, Pix
):
    with open(image_path, "rb") as image_file:
        input_image_bytes = image_file.read()
        output_image_bytes = Model().inference.remote(input_image_bytes, prompt)

    dir = Path("/tmp/stable-diffusion-xl-turbo")
    if not dir.exists():
        dir.mkdir(exist_ok=True, parents=True)

    output_path = dir / "output.png"
    print(f"Saving it to {output_path}")
    with open(output_path, "wb") as f:
        f.write(output_image_bytes)
```

Running the model

We can run the model with different parameters using the following command,

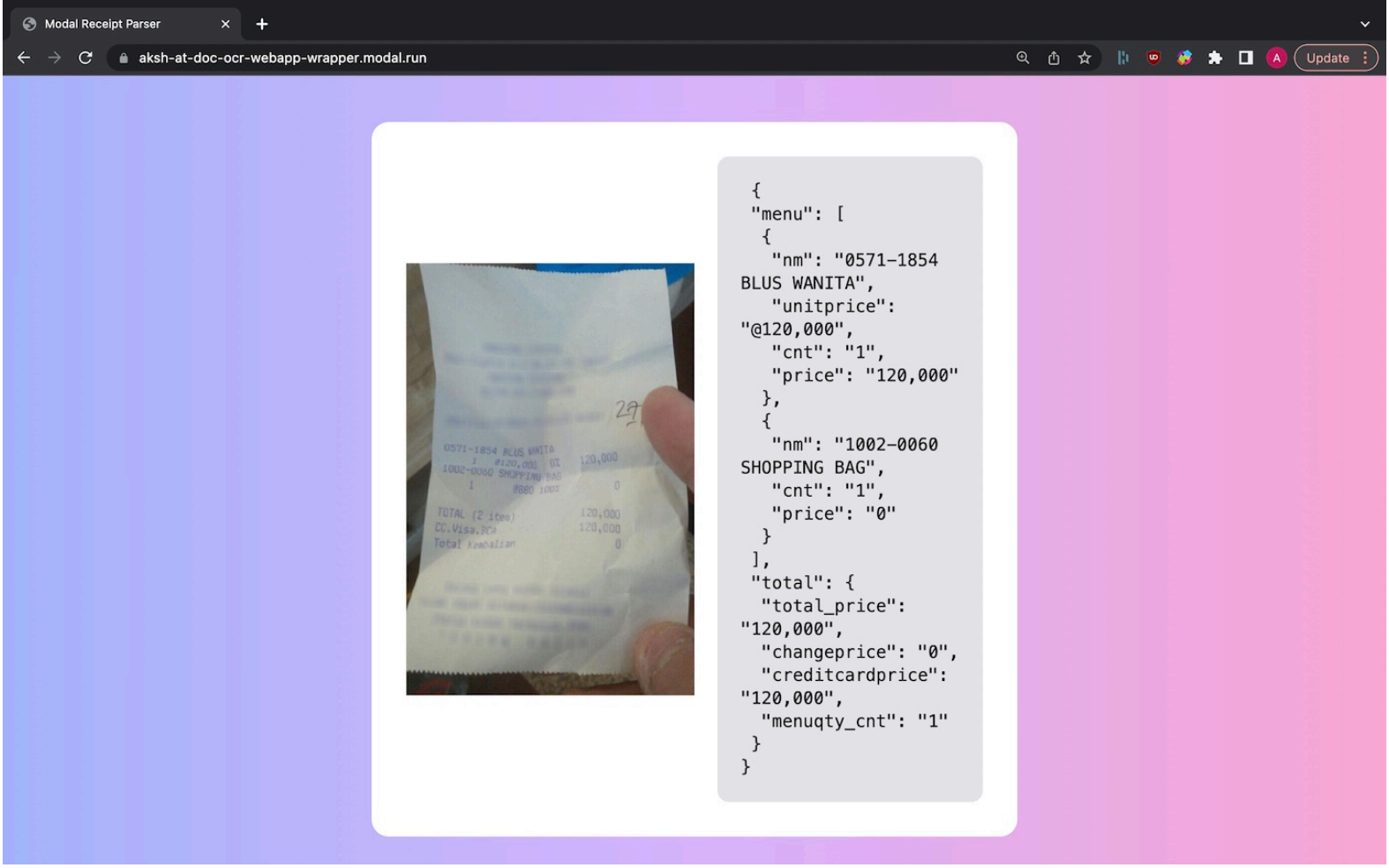
```
modal run stable_diffusion_xl_turbo.py --prompt="harry potter, glasses, wizard" --image-pa
```

Document OCR job queue

View on GitHub

This tutorial shows you how to use Modal as an infinitely scalable job queue that can service async tasks from a web app. For the purpose of this tutorial, we've also built a React + FastAPI web app on Modal that works together with it, but note that you don't need a web app running on Modal to use this pattern. You can submit async tasks to Modal from any Python application (for example, a regular Django app running on Kubernetes).

Our job queue will handle a single task: running OCR transcription for images. We'll make use of a pre-trained Document Understanding model using the donut package to accomplish this. Try it out for yourself here.



Define an App

Let's first import `modal` and define a `App`. Later, we'll use the name provided for our `App` to find it from our web app, and submit tasks to it.

```
import urllib.request

import modal

app = modal.App(
    "example-doc-ocr-jobs"
)  # Note: prior to April 2024, "app" was called "stub"
```

Model cache

`donut` downloads the weights for pre-trained models to a local directory, if those weights don't already exist. To decrease start-up time, we want this download to happen just once, even across separate function invocations. To accomplish this, we use the `Image.run_function` method, which allows us to run some code at image build time to save the model weights into the image.

```
CACHE_PATH = "/root/model_cache"
MODEL_NAME = "naver-clova-ix/donut-base-finetuned-cord-v2"


def download_model_weights() -> None:
    from huggingface_hub import snapshot_download

    snapshot_download(repo_id=MODEL_NAME, cache_dir=CACHE_PATH)


image = (
    modal.Image.debian_slim(python_version="3.9")
    .pip_install(
        "donut-python==1.0.7",
        "huggingface-hub==0.16.4",
        "transformers==4.21.3",
        "timm==0.5.4",
    )
    .run_function(download_model_weights)
)
```

Handler function

Now let's define our handler function. Using the @app.function() decorator, we set up a Modal Function that uses GPUs, runs on a custom container image, and automatically retries failures up to 3 times.

```
@app.function(
    gpu="any",
    image=image,
    retries=3,
)
def parse_receipt(image: bytes):
    import io

    import torch
    from donut import DonutModel
    from PIL import Image

    # Use donut fine-tuned on an OCR dataset.
    task_prompt = "<s_cord-v2>"
    pretrained_model = DonutModel.from_pretrained(
        MODEL_NAME,
        cache_dir=CACHE_PATH,
    )

    # Initialize model.
    pretrained_model.half()
    device = torch.device("cuda")
    pretrained_model.to(device)
```

```
    # Run inference.
    input_img = Image.open(io.BytesIO(image))
    output = pretrained_model.inference(image=input_img, prompt=task_prompt)[
        "predictions"
    ][0]
    print("Result: ", output)

    return output
```

Deploy

Now that we have a function, we can publish it by deploying the app:

```
modal deploy doc_ocr_jobs.py
```

Once it's published, we can look up this function from another Python process and submit tasks to it:

```
fn = modal.Function.lookup("example-doc-ocr-jobs", "parse_receipt")
fn.spawn(my_image)
```

Modal will auto-scale to handle all the tasks queued, and then scale back down to 0 when there's no work left. To see how you could use this from a Python web app, take a look at the receipt parser frontend tutorial.

Run manually

We can also trigger `parse_receipt` manually for easier debugging: `modal run doc_ocr_jobs::app.main` To try it out, you can find some example receipts here.

```
@app.local_entrypoint()
def main():
    from pathlib import Path

    receipt_filename = Path(__file__).parent / "receipt.png"
    if receipt_filename.exists():
        with open(receipt_filename, "rb") as f:
            image = f.read()
    else:
        image = urllib.request.urlopen(
            "https://nwlc.org/wp-content/uploads/2022/01/Brandys-walmart-receipt-8.webp"
```

```
    ).read()
print(parse_receipt.remote(image))
```

Render a video with Blender on many GPUs or CPUs in parallel

View on GitHub

This example shows how you can render an animated 3D scene using Blender's Python interface.

You can run it on CPUs to scale out on one hundred containers or run it on GPUs to get higher throughput per node. Even for this simple scene, GPUs render 10x faster than CPUs.

The final render looks something like this:

Defining a Modal app

```
from pathlib import Path

import modal
```

Modal runs your Python functions for you in the cloud. You organize your code into apps, collections of functions that work together.

```
app = modal.App("examples-blender-video")
```

We need to define the environment each function runs in — its container image. The block below defines a container image, starting from a basic Debian Linux image adding Blender's system-level dependencies and then installing the `bpy` package, which is Blender's Python API.

```
rendering_image = (
    modal.Image.debian_slim(python_version="3.11")
    .apt_install("xorg", "libxkbcommon0")  # X11 (Unix GUI) dependencies
    .pip_install("bpy==4.1.0")  # Blender as a Python package
)
```

Rendering a single frame

We define a function that renders a single frame. We'll scale this function out on Modal later.

Functions in Modal are defined along with their hardware and their dependencies. This function can be run with GPU acceleration or without it, and we'll use a global flag in the code to switch between the two.

```
WITH_GPU = True  # try changing this to False to run rendering massively in parallel on CP
```

We decorate the function with `@app.function` to define it as a Modal function. Note that in addition to defining the hardware requirements of the function, we also specify the container image that the function runs in (the one we defined above).

The details of the scene aren't too important for this example, but we'll load a .blend file that we created earlier. This scene contains a rotating Modal logo made of a transmissive ice-like material, with a generated displacement map. The animation keyframes were defined in Blender.

```
@app.function(
    gpu="A10G" if WITH_GPU else None,
    # default limits on Modal free tier
    concurrency_limit=10 if WITH_GPU else 100,
    image=rendering_image,
)
def render(blend_file: bytes, frame_number: int = 0) -> bytes:
    """Renders the n-th frame of a Blender file as a PNG."""
    import bpy

    input_path = "/tmp/input.blend"
    output_path = f"/tmp/output-{frame_number}.png"
```

```
    # Blender requires input as a file.
    Path(input_path).write_bytes(blend_file)

    bpy.ops.wm.open_mainfile(filepath=input_path)
    bpy.context.scene.frame_set(frame_number)
    bpy.context.scene.render.filepath = output_path
    configure_rendering(bpy.context, with_gpu=WITH_GPU)
    bpy.ops.render.render(write_still=True)

    # Blender renders image outputs to a file as well.
    return Path(output_path).read_bytes()
```

Rendering with acceleration

We can configure the rendering process to use GPU acceleration with NVIDIA CUDA. We select the Cycles rendering engine, which is compatible with CUDA, and then activate the GPU.

```
def configure_rendering(ctx, with_gpu: bool):
    # configure the rendering process
    ctx.scene.render.engine = "CYCLES"
    ctx.scene.render.resolution_x = 3000
    ctx.scene.render.resolution_y = 2000
    ctx.scene.render.resolution_percentage = 50
    ctx.scene.cycles.samples = 128

    cycles = ctx.preferences.addons["cycles"]

    # Use GPU acceleration if available.
    if with_gpu:
        cycles.preferences.compute_device_type = "CUDA"
        ctx.scene.cycles.device = "GPU"

        # reload the devices to update the configuration
        cycles.preferences.get_devices()
        for device in cycles.preferences.devices:
            device.use = True

    else:
        ctx.scene.cycles.device = "CPU"

    # report rendering devices -- a nice snippet for debugging and ensuring the accelerato
    for dev in cycles.preferences.devices:
        print(
            f"ID:{dev['id']} Name:{dev['name']} Type:{dev['type']} Use:{dev['use']}"
        )
```

Combining frames into a video

Rendering 3D images is fun, and GPUs can make it faster, but rendering 3D videos is better! We add another function to our app, running on a different, simpler container image and different hardware, to combine the frames into a video.

```
combination_image = modal.Image.debian_slim(python_version="3.11").apt_install(
    "ffmpeg"
)
```

The video has a few parameters, which we set here.

```
FPS = 60
FRAME_COUNT = 250
FRAME_SKIP = 1  # increase this to skip frames and speed up rendering
```

The function to combine the frames into a video takes a sequence of byte sequences, one for each rendered frame, and converts them into a single sequence of bytes, the MP4 file.

```
@app.function(image=combination_image)
def combine(frames_bytes: list[bytes], fps: int = FPS) -> bytes:
    import subprocess
    import tempfile

    with tempfile.TemporaryDirectory() as tmpdir:
        for i, frame_bytes in enumerate(frames_bytes):
            frame_path = Path(tmpdir) / f"frame_{i:05}.png"
            frame_path.write_bytes(frame_bytes)
        out_path = Path(tmpdir) / "output.mp4"
        subprocess.run(
            f"ffmpeg -framerate {fps} -pattern_type glob -i '{tmpdir}/*.png' -c:v libx264
            shell=True,
        )
        return out_path.read_bytes()
```

Rendering in parallel in the cloud from the comfort of the command line

With these two functions defined, we need only a few more lines to run our rendering at scale on Modal.

First, we need a function that coordinates our functions to `render` frames and `combine` them. We decorate that function with `@app.local_entrypoint` so that we can run it with `modal run blender_video.py`.

In that function, we use `render.map` to map the `render` function over the range of frames, so that the logo will spin in the final video.

We collect the bytes from each frame into a `list` locally and then send it to `combine` with `.remote`.

The bytes for the video come back to our local machine, and we write them to a file.

The whole rendering process (for 4 seconds of 1080p 60 FPS video) takes about five minutes to run on 10 A10G GPUs, with a per-frame latency of about 10 seconds, and about five minutes to run on 100 CPUs, with a per-frame latency of about one minute.

```
@app.local_entrypoint()
def main():
    output_directory = Path("/tmp") / "render"
    output_directory.mkdir(parents=True, exist_ok=True)

    input_path = Path(__file__).parent / "IceModal.blend"
    blend_bytes = input_path.read_bytes()
    args = [
        (blend_bytes, frame) for frame in range(1, FRAME_COUNT + 1, FRAME_SKIP)
    ]
    images = list(render.starmap(args))
    for i, image in enumerate(images):
        frame_path = output_directory / f"frame_{i + 1}.png"
        frame_path.write_bytes(image)
        print(f"Frame saved to {frame_path}")

    video_path = output_directory / "output.mp4"
    video_bytes = combine.remote(images)
    video_path.write_bytes(video_bytes)
    print(f"Video saved to {video_path}")
```

Privacy

Hacker News Slackbot

View on GitHub

In this example, we use Modal to deploy a cron job that periodically queries Hacker News for new posts matching a given search term, and posts the results to Slack.

Import and define the app

Let's start off with imports, and defining a Modal app.

```
import os
from datetime import datetime, timedelta

import modal

app = modal.App(
    "example-hn-bot"
)  # Note: prior to April 2024, "app" was called "stub"
```

Now, let's define an image that has the `slack-sdk` package installed, in which we can run a function that posts a slack message.

```
slack_sdk_image = modal.Image.debian_slim().pip_install("slack-sdk")
```

Defining the function and importing the secret

Our Slack bot will need access to a bot token. We can use Modal's Secrets interface to accomplish this. To quickly create a Slack bot secret, navigate to the create secret page, select the Slack secret template from the list options, and follow the instructions in the "Where to find the credentials?" panel. Name your secret `hn-bot-slack`, so that the code in this example still works.

Now, we define the function `post_to_slack`, which simply instantiates the Slack client using our token, and then uses it to post a message to a given channel name.

```
@app.function(
    image=slack_sdk_image, secrets=[modal.Secret.from_name("hn-bot-slack")]
)
async def post_to_slack(message: str):
    import slack_sdk

    client = slack_sdk.WebClient(token=os.environ["SLACK_BOT_TOKEN"])
    client.chat_postMessage(channel="hn-alerts", text=message)
```

Searching Hacker News

We are going to use Algolia's Hacker News Search API to query for posts matching a given search term in the past X days. Let's define our search term and query period.

```
QUERY = "serverless"
WINDOW_SIZE_DAYS = 1
```

Let's also define an image that has the `requests` package installed, so we can query the API.

```
requests_image = modal.Image.debian_slim().pip_install("requests")
```

We can now define our main entrypoint, that queries Algolia for the term, and calls `post_to_slack` on all the results. We specify a schedule in the function decorator, which means that our function will run automatically at the given interval.

```
@app.function(image=requests_image)
def search_hackernews():
    import requests

    url = "http://hn.algolia.com/api/v1/search"

    threshold = datetime.utcnow() - timedelta(days=WINDOW_SIZE_DAYS)

    params = {
```

```
        "query": QUERY,
        "numericFilters": f"created_at_i>{threshold.timestamp()}",
    }

    response = requests.get(url, params, timeout=10).json()
    urls = [item["url"] for item in response["hits"] if item.get("url")]

    print(f"Query returned {len(urls)} items.")

    post_to_slack.for_each(urls)
```

Test running

We can now test run our scheduled function as follows: `modal run hackernews_alerts.py::app.search_hackernews`

Defining the schedule and deploying

Let's define a function that will be called by Modal every day

```
@app.function(schedule=modal.Period(days=1))
def run_daily():
    search_hackernews.remote()
```

In order to deploy this as a persistent cron job, you can run `modal deploy hackernews_alerts.py` ,

Once the job is deployed, visit the apps page page to see its execution history, logs and other stats.

Add Modal Apps to Tailscale

View on GitHub

This example demonstrates how to integrate Modal with Tailscale (https://tailscale.com). It outlines the steps to configure Modal containers so that they join the Tailscale network.

We use a custom entrypoint to automatically add containers to a Tailscale network (tailnet). This configuration enables the containers to interact with one another and with additional applications within the same tailnet.

```
import modal
```

Install Tailscale and copy custom entrypoint script (entrypoint.sh). The script must be executable.

```
image = (
    modal.Image.debian_slim()
    .apt_install("curl")
    .run_commands("curl -fsSL https://tailscale.com/install.sh | sh")
    .pip_install("requests[socks]")
    .copy_local_file("./entrypoint.sh", "/root/entrypoint.sh")
    .dockerfile_commands(
        "RUN chmod a+x /root/entrypoint.sh",
        'ENTRYPOINT ["/root/entrypoint.sh"]',
    )
)
app = modal.App(
    image=image
)  # Note: prior to April 2024, "app" was called "stub"
```

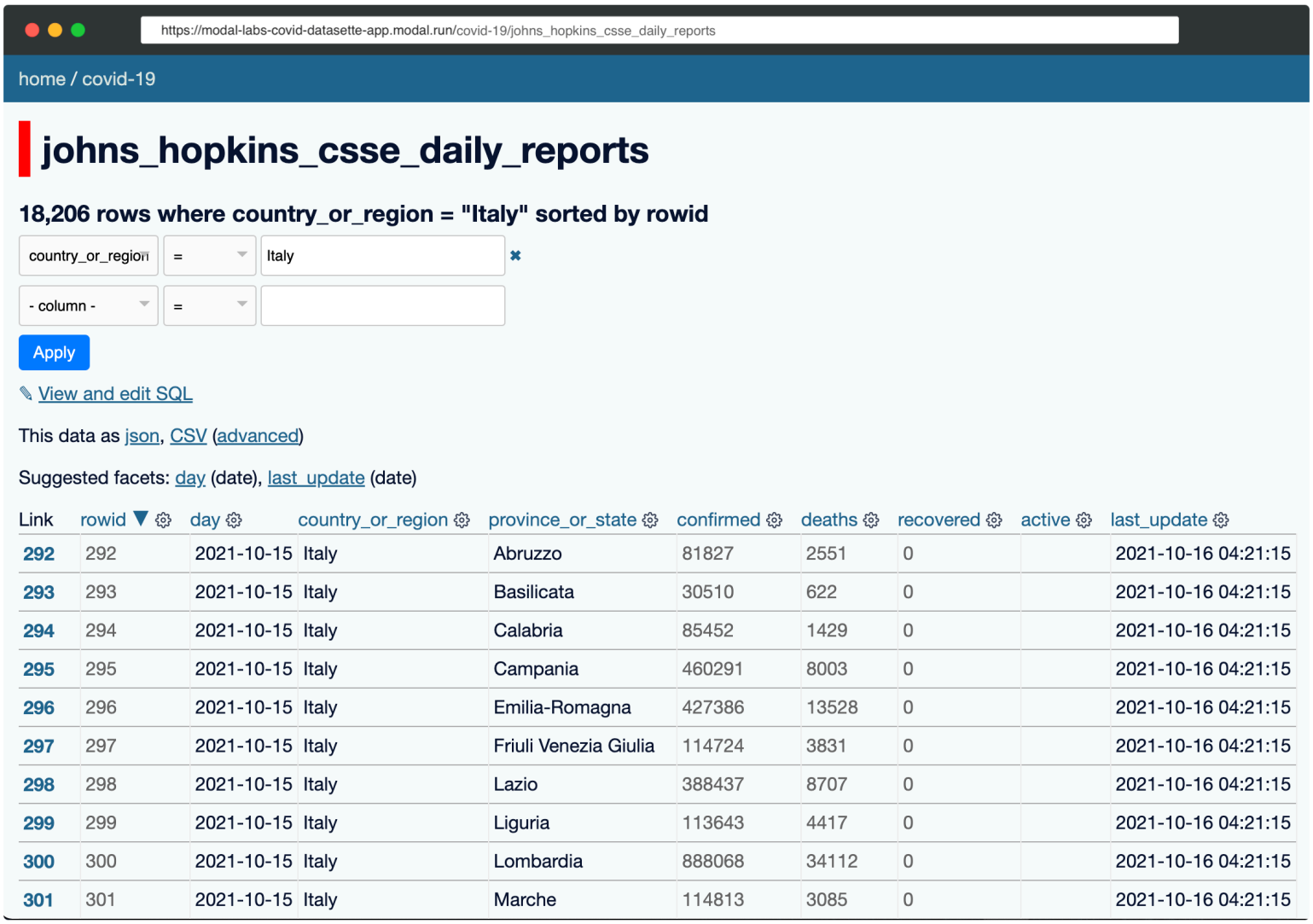
Run your function adding a Tailscale secret. It expects an environment variable named `TAILSCALE_AUTHKEY`. We suggest creating a reusable and ephemeral key.

```
@app.function(
    secrets=[
        modal.Secret.from_name("tailscale-auth"),
        modal.Secret.from_dict(
            {
                "ALL_PROXY": "socks5://localhost:1080/",
                "HTTP_PROXY": "http://localhost:1080/",
                "http_proxy": "http://localhost:1080/",
            }
        ),
    ],
)
def connect_to_raspberrypi():
    import requests

    # Connect to other machines in your tailnet.
    resp = requests.get("http://raspberrypi:5000")
    print(resp.content)
```

Run this script with `modal run modal_tailscale.py`. You will see Tailscale logs when the container start indicating that you were able to login successfully and that the proxies (SOCKS5 and HTTP) have created been successfully. You will also be able to see Modal containers in your Tailscale dashboard in the “Machines” tab. Every new container launched will show up as a new “machine”. Containers are individually addressable using their Tailscale name or IP address.

Publish interactive datasets with Datasette

View on GitHub



Link	rowid	day	country_or_region	province_or_state	confirmed	deaths	recovered	active	last_update
292	292	2021-10-15	Italy	Abruzzo	81827	2551	0		2021-10-16 04:21:15
293	293	2021-10-15	Italy	Basilicata	30510	622	0		2021-10-16 04:21:15
294	294	2021-10-15	Italy	Calabria	85452	1429	0		2021-10-16 04:21:15
295	295	2021-10-15	Italy	Campania	460291	8003	0		2021-10-16 04:21:15
296	296	2021-10-15	Italy	Emilia-Romagna	427386	13528	0		2021-10-16 04:21:15
297	297	2021-10-15	Italy	Friuli Venezia Giulia	114724	3831	0		2021-10-16 04:21:15
298	298	2021-10-15	Italy	Lazio	388437	8707	0		2021-10-16 04:21:15
299	299	2021-10-15	Italy	Liguria	113643	4417	0		2021-10-16 04:21:15
300	300	2021-10-15	Italy	Lombardia	888068	34112	0		2021-10-16 04:21:15
301	301	2021-10-15	Italy	Marche	114813	3085	0		2021-10-16 04:21:15

This example shows how to serve a Datasette application on Modal. The published dataset is COVID-19 case data from Johns Hopkins University which is refreshed daily. Try it out for yourself at modal-labs-example-covid-datasette-app.modal.run/covid-19.

Some Modal features it uses:

- Volumes: a persisted volume lets us store and grow the published dataset over time.
- Scheduled functions: the underlying dataset is refreshed daily, so we schedule a function to run daily.
- Web endpoints: exposes the Datasette application for web browser interaction and API requests.

Basic setup

Let's get started writing code. For the Modal container image we need a few Python packages, including `GitPython`, which we'll use to download the dataset.

```
import asyncio
import pathlib
import shutil
import subprocess
from datetime import datetime
from urllib.request import urlretrieve

from modal import App, Image, Period, Volume, asgi_app

app = App(
    "example-covid-datasette"
)  # Note: prior to April 2024, "app" was called "stub"
datasette_image = (
    Image.debian_slim()
    .pip_install("datasette~=0.63.2", "sqlite-utils")
    .apt_install("unzip")
)
```

Persistent dataset storage

To separate database creation and maintenance from serving, we'll need the underlying database file to be stored persistently. To achieve this we use a `Volume`.

```
volume = Volume.from_name(
    "example-covid-datasette-cache-vol", create_if_missing=True
)
```

```
VOLUME_DIR = "/cache-vol"
REPORTS_DIR = pathlib.Path(VOLUME_DIR, "COVID-19")
DB_PATH = pathlib.Path(VOLUME_DIR, "covid-19.db")
```

Getting a dataset

Johns Hopkins has been publishing up-to-date COVID-19 pandemic data on GitHub since early February 2020, and as of late September 2022 daily reporting is still rolling in. Their dataset is what this example will use to show off Modal and Datasette's capabilities.

The full git repository size for the dataset is over 6GB, but we only need to shallow clone around 300MB.

```
@app.function(
    image=datasette_image,
    volumes={VOLUME_DIR: volume},
    retries=2,
)
def download_dataset(cache=True):
    if REPORTS_DIR.exists() and cache:
        print(f"Dataset already present and {cache=}. Skipping download.")
        return
    elif REPORTS_DIR.exists():
        print("Cleaning dataset before re-downloading...")
        shutil.rmtree(REPORTS_DIR)

    print("Downloading dataset...")
    urlretrieve(
        "https://github.com/CSSEGISandData/COVID-19/archive/refs/heads/master.zip",
        "/tmp/covid-19.zip",
    )

    print("Unpacking archive...")
    prefix = "COVID-19-master/csse_covid_19_data/csse_covid_19_daily_reports"
    subprocess.run(
        f"unzip /tmp/covid-19.zip {prefix}/* -d {REPORTS_DIR}", shell=True
    )
    subprocess.run(f"mv {REPORTS_DIR / prefix}/* {REPORTS_DIR}", shell=True)

    print("Committing the volume...")
    volume.commit()

    print("Finished downloading dataset.")
```

Data munging

This dataset is no swamp, but a bit of data cleaning is still in order. The following two functions read a handful of `.csv` files and clean the data, before inserting it into SQLite.

```
def load_daily_reports():
    daily_reports = list(REPORTS_DIR.glob("*.csv"))
    if not daily_reports:
        raise RuntimeError(
            f"Could not find any daily reports in {REPORTS_DIR}."
        )
    for filepath in daily_reports:
        yield from load_report(filepath)


def load_report(filepath):
    import csv

    mm, dd, yyyy = filepath.stem.split("-")
    with filepath.open() as fp:
        for row in csv.DictReader(fp):
            province_or_state = (
                row.get("\ufeffProvince/State")
                or row.get("Province/State")
                or row.get("Province_State")
                or None
            )
            country_or_region = row.get("Country_Region") or row.get(
                "Country/Region"
            )
            yield {
                "day": f"{yyyy}-{mm}-{dd}",
                "country_or_region": (
                    country_or_region.strip() if country_or_region else None
                ),
                "province_or_state": (
                    province_or_state.strip() if province_or_state else None
                ),
                "confirmed": int(float(row["Confirmed"] or 0)),
                "deaths": int(float(row["Deaths"] or 0)),
                "recovered": int(float(row["Recovered"] or 0)),
                "active": int(row["Active"]) if row.get("Active") else None,
                "last_update": row.get("Last Update")
                or row.get("Last_Update")
                or None,
            }
```

Inserting into SQLite

With the CSV processing out of the way, we're ready to create an SQLite DB and feed data into it. Importantly, the `prep_db` function mounts the same volume used by `download_dataset()`, and rows are batch inserted with progress logged after each batch, as the full COVID-19 has millions of rows and does take some time to be fully inserted.

A more sophisticated implementation would only load new data instead of performing a full refresh, but we're keeping things simple for this example!

```
def chunks(it, size):
    import itertools

    return iter(lambda: tuple(itertools.islice(it, size)), ())


@app.function(
    image=datasette_image,
    volumes={VOLUME_DIR: volume},
    timeout=900,
)
def prep_db():
    import sqlite_utils

    volume.reload()
    print("Loading daily reports...")
    records = load_daily_reports()

    DB_PATH.parent.mkdir(parents=True, exist_ok=True)
    db = sqlite_utils.Database(DB_PATH)
    table = db["johns_hopkins_csse_daily_reports"]

    batch_size = 100_000
    for i, batch in enumerate(chunks(records, size=batch_size)):
        truncate = True if i == 0 else False
        table.insert_all(batch, batch_size=batch_size, truncate=truncate)
        print(f"Inserted {len(batch)} rows into DB.")

    table.create_index(["day"], if_not_exists=True)
    table.create_index(["province_or_state"], if_not_exists=True)
    table.create_index(["country_or_region"], if_not_exists=True)

    print("Syncing DB with volume.")
    volume.commit()
    db.close()
```

Keep it fresh

Johns Hopkins commits new data to the dataset repository every day, so we set up a scheduled function to automatically refresh the database every 24 hours.

```
@app.function(schedule=Period(hours=24), timeout=1000)
def refresh_db():
    print(f"Running scheduled refresh at {datetime.now()}")
    download_dataset.remote(cache=False)
    prep_db.remote()
    volume.commit()
    print("Volume changes committed.")
```

Web endpoint

Hooking up the SQLite database to a Modal webhook is as simple as it gets. The Modal `@asgi_app` decorator wraps a few lines of code: one `import` and a few lines to instantiate the `Datasette` instance and return its app server.

```
@app.function(
    image=datasette_image,
    volumes={VOLUME_DIR: volume},
    allow_concurrent_inputs=16,
)
@asgi_app()
def ui():
    from datasette.app import Datasette

    ds = Datasette(files=[DB_PATH], settings={"sql_time_limit_ms": 10000})
    asyncio.run(ds.invoke_startup())
    return ds.app()
```

Publishing to the web

Run this script using `modal run covid_datasette.py` and it will create the database.

You can then use `modal serve covid_datasette.py` to create a short-lived web URL that exists until you terminate the script.

When publishing the interactive Datasette app you'll want to create a persistent URL. Just run `modal deploy covid_datasette.py`.

```
@app.local_entrypoint()
def run():
    print("Downloading COVID-19 dataset...")
```

```
download_dataset.remote()
print("Prepping SQLite DB...")
prep_db.remote()
```

You can explore the data at the deployed web endpoint.

Stable Diffusion XL 1.0

View on GitHub

This example is similar to the Stable Diffusion CLI example, but it generates images from the larger SDXL 1.0 model. Specifically, it runs the first set of steps with the base model, followed by the refiner model.

Try out the live demo here! The first generation may include a cold-start, which takes around 20 seconds. The inference speed depends on the GPU and step count (for reference, an A100 runs 40 steps in 8 seconds).

Basic setup

```
import io
from pathlib import Path

from modal import (
    App,
    Image,
    Mount,
    asgi_app,
    build,
    enter,
    gpu,
    method,
    web_endpoint,
)
```

Define a container image

To take advantage of Modal's blazing fast cold-start times, we'll need to download our model weights inside our container image with a download function. We ignore binaries, ONNX weights and 32-bit weights.

Tip: avoid using global variables in this function to ensure the download step detects model changes and triggers a rebuild.

```
sdxl_image = (
    Image.debian_slim(python_version="3.10")
    .apt_install(
        "libglib2.0-0", "libsm6", "libxrender1", "libxext6", "ffmpeg", "libgl1"
    )
    .pip_install(
        "diffusers==0.26.3",
        "invisible_watermark==0.2.0",
        "transformers~=4.38.2",
        "accelerate==0.27.2",
        "safetensors==0.4.2",
    )
)

app = App(
    "stable-diffusion-xl"
)  # Note: prior to April 2024, "app" was called "stub"

with sdxl_image.imports():
    import torch
    from diffusers import DiffusionPipeline
    from fastapi import Response
```

Load model and run inference

The container lifecycle `@enter` decorator loads the model at startup. Then, we evaluate it in the `run_inference` function.

To avoid excessive cold-starts, we set the idle timeout to 240 seconds, meaning once a GPU has loaded the model it will stay online for 4 minutes before spinning down. This can be adjusted for cost/experience trade-offs.

```
@app.cls(gpu=gpu.A10G(), container_idle_timeout=240, image=sdxl_image)
class Model:
    @build()
    def build(self):
```

```
    from huggingface_hub import snapshot_download

    ignore = [
        "*.bin",
        "*.onnx_data",
        "*/diffusion_pytorch_model.safetensors",
    ]
    snapshot_download(
        "stabilityai/stable-diffusion-xl-base-1.0", ignore_patterns=ignore
    )
    snapshot_download(
        "stabilityai/stable-diffusion-xl-refiner-1.0",
        ignore_patterns=ignore,
    )

@enter()
def enter(self):
    load_options = dict(
        torch_dtype=torch.float16,
        use_safetensors=True,
        variant="fp16",
        device_map="auto",
    )

    # Load base model
    self.base = DiffusionPipeline.from_pretrained(
        "stabilityai/stable-diffusion-xl-base-1.0", **load_options
    )

    # Load refiner model
    self.refiner = DiffusionPipeline.from_pretrained(
        "stabilityai/stable-diffusion-xl-refiner-1.0",
        text_encoder_2=self.base.text_encoder_2,
        vae=self.base.vae,
        **load_options,
    )

    # Compiling the model graph is JIT so this will increase inference time for the fi
    # but speed up subsequent runs. Uncomment to enable.
    # self.base.unet = torch.compile(self.base.unet, mode="reduce-overhead", fullgraph
    # self.refiner.unet = torch.compile(self.refiner.unet, mode="reduce-overhead", ful

def _inference(self, prompt, n_steps=24, high_noise_frac=0.8):
    negative_prompt = "disfigured, ugly, deformed"
    image = self.base(
        prompt=prompt,
        negative_prompt=negative_prompt,
        num_inference_steps=n_steps,
        denoising_end=high_noise_frac,
        output_type="latent",
    ).images
    image = self.refiner(
```

```
            prompt=prompt,
            negative_prompt=negative_prompt,
            num_inference_steps=n_steps,
            denoising_start=high_noise_frac,
            image=image,
        ).images[0]

        byte_stream = io.BytesIO()
        image.save(byte_stream, format="JPEG")

        return byte_stream

    @method()
    def inference(self, prompt, n_steps=24, high_noise_frac=0.8):
        return self._inference(
            prompt, n_steps=n_steps, high_noise_frac=high_noise_frac
        ).getvalue()

    @web_endpoint()
    def web_inference(self, prompt, n_steps=24, high_noise_frac=0.8):
        return Response(
            content=self._inference(
                prompt, n_steps=n_steps, high_noise_frac=high_noise_frac
            ).getvalue(),
            media_type="image/jpeg",
        )
```

And this is our entrypoint; where the CLI is invoked. Explore CLI options with: `modal run stable_diffusion_xl.py —help

```
@app.local_entrypoint()
def main(prompt: str = "Unicorns and leprechauns sign a peace treaty"):
    image_bytes = Model().inference.remote(prompt)

    dir = Path("/tmp/stable-diffusion-xl")
    if not dir.exists():
        dir.mkdir(exist_ok=True, parents=True)

    output_path = dir / "output.png"
    print(f"Saving it to {output_path}")
    with open(output_path, "wb") as f:
        f.write(image_bytes)
```

A user interface

Here we ship a simple web application that exposes a front-end (written in Alpine.js) for our backend deployment.

The Model class will serve multiple users from a its own shared pool of warm GPU containers automatically.

We can deploy this with `modal deploy stable_diffusion_xl.py`.

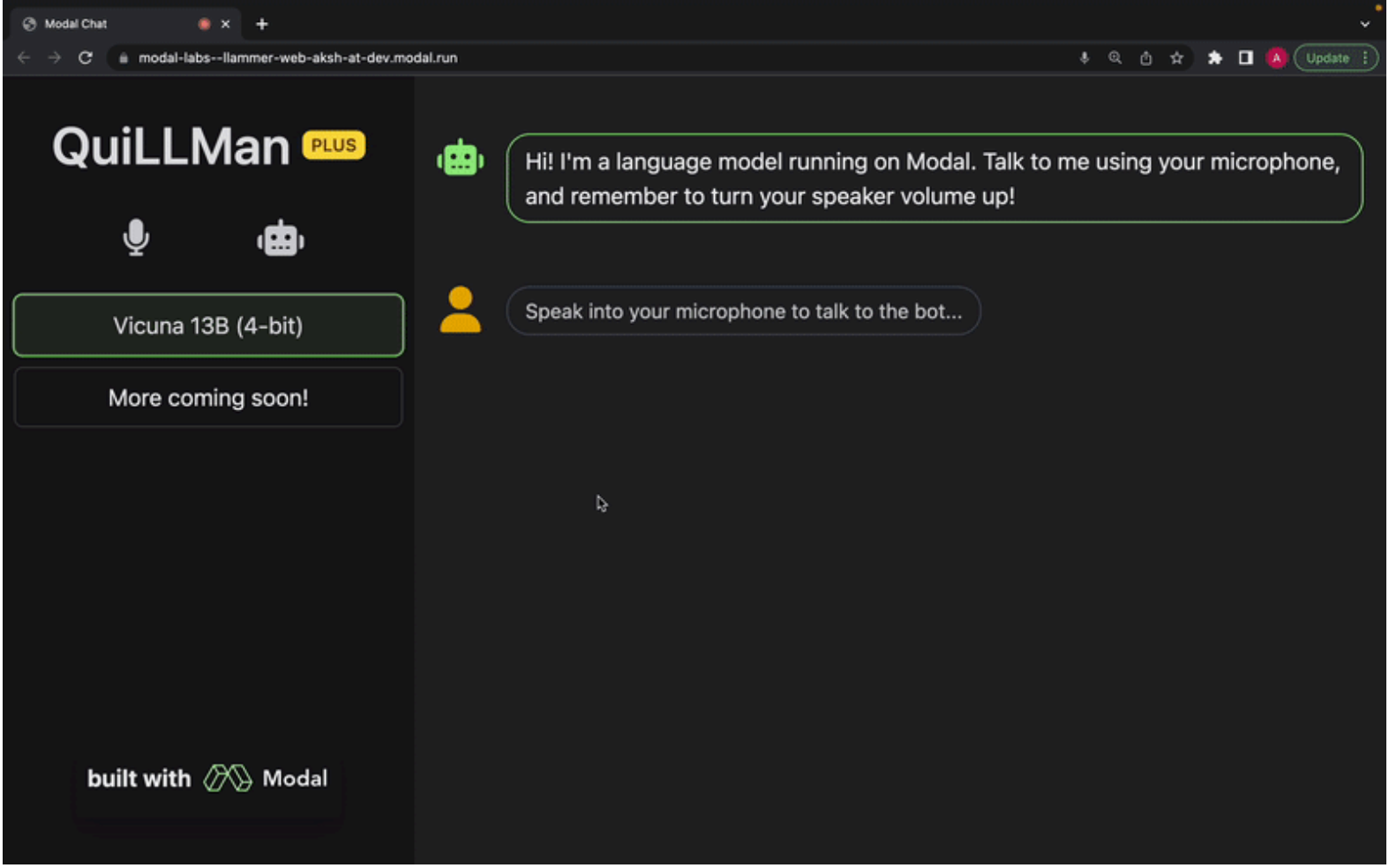
```
frontend_path = Path(__file__).parent / "frontend"

web_image = Image.debian_slim().pip_install("jinja2")


@app.function(
    image=web_image,
    mounts=[Mount.from_local_dir(frontend_path, remote_path="/assets")],
    allow_concurrent_inputs=20,
)
@asgi_app()
def ui():
    import fastapi.staticfiles
    from fastapi import FastAPI, Request
    from fastapi.templating import Jinja2Templates

    web_app = FastAPI()
    templates = Jinja2Templates(directory="/assets")

    @web_app.get("/")
    async def read_root(request: Request):
        return templates.TemplateResponse(
            "index.html",
            {
                "request": request,
                "inference_url": Model.web_inference.web_url,
                "model_name": "Stable Diffusion XL",
                "default_prompt": "A cinematic shot of a baby raccoon wearing an intricate
            },
        )

    web_app.mount(
        "/static",
        fastapi.staticfiles.StaticFiles(directory="/assets"),
        name="static",
    )

    return web_app
```

QuiLLMan: Voice Chat with LLMs

QuiLLMan is a complete voice chat application built on Modal: you speak and the chatbot speaks back! It uses Whisper to transcribe speech to text, Zephyr to generate text responses, and Tortoise TTS to convert those responses into natural-sounding speech.

We've enjoyed playing around with QuiLLMan enough at Modal HQ that we decided to share the repo and put up a live demo.

Everything — the React frontend, the backend API, the LLMs — is deployed serverlessly, allowing it to automatically scale and ensuring you only pay for the compute you use. Read on to see how Modal makes this easy!



This post provides a high-level walkthrough of the repo. We're looking to add more models and features to this as time goes on, and contributions are welcome!

Code overview

Traditionally, building a robust serverless web application as complex as QuiLLMan would require a lot of work — you're setting up a backend API and three different ML services, running in separate custom containers and autoscaling independently.

But with Modal, it's as simple as writing 4 different classes and running a CLI command.

Our project structure looks like this:

1. Language model service: continues a text conversation with a text reply.
2. Transcription service: converts speech audio into text.
3. Text-to-speech service: converts text into speech.
4. FastAPI server: runs server-side app logic.
5. React frontend: runs client-side interaction logic.

Let's go through each of these components in more detail.

You'll want to have the code handy — look for GitHub links, like the one below!

Language model

Language models are trained to predict what text will come at the end of incomplete text. From this simple task emerge the sparks of artificial general intelligence.

In this case, we want to predict the text that a helpful, friendly assistant might write to continue a conversation with a user.

As with all Modal applications, we start by describing the environment (the container `Image`), which we construct via Python method chaining:

```
zephyr_image = (
    modal.Image.debian_slim(python_version="3.11")
    .pip_install(
        "autoawq==0.1.8",
        "torch==2.1.2",
    )
)
```

The chain starts with a base Debian container, installs `python_version` 3.11, then uses `pip install` our Python packages we need. Pinning versions of our dependencies ensures that the built image is reproducible.

We use AutoAWQ, an implementation of Activation-aware Weight Quantization, to quantize our model to 4 bits for faster inference.

The models we use define a `generate` function that constructs an input to our language model from a prompt template, the conversation history, and the latest text from the user. Then, it streams (`yield`s) tokens as they are produced. Remote Python generators work out-of-the-box in Modal, so building streaming interactions is easy.

Although we're going to call this model from our backend API, it's useful to test it directly as well. To do this, we define a `local_entrypoint`:

```
@app.local_entrypoint()
def main(input: str):
    model = Zephyr()
    for val in model.generate.remote(input):
        print(val, end="", flush=True)
```

Now, we can `run` the model with a prompt of our choice from the terminal:

```
modal run -q src.llm_zephyr --input "How do antihistamines work?"
```

Transcription

In this file we define a Modal class that uses OpenAI's Whisper to transcribe audio in real-time. The helper function `load_audio` downsamples the audio to 16kHz (as required by Whisper) using `ffmpeg`.

We're using an A10G GPU for transcriptions, which lets us transcribe most segments in under 2 seconds.

Text-to-speech

The text-to-speech service is adapted from `tortoise-tts-modal-api`, a Modal deployment of Tortoise TTS. Take a look at those repos if you're interested in understanding how the code works, or for a full list of the parameters and voices you can use.

FastAPI server

Our backend is a FastAPI Python app. We can serve this app over the internet without any extra effort on top of writing the `localhost` version by slapping on an `@asgi_app` decorator.

Of the 4 endpoints in the file, `POST /generate` is the most interesting. It uses queueing and streaming to reduce latency without compromising on the quality of the language modeling or speech generation.

Specifically, it calls `llm.generate` and starts streaming the text results back. When the text stream hits a `PUNCTUATION` marker (like `.` or `,`), it asynchronously calls `Tortoise.speak` to generate audio, returning a handle to the function call. This handle can be used to poll for the audio later, as explained in our job queue example. If Tortoise is not enabled, we return the sentence directly so that the frontend can use the browser's built-in text-to-speech.

We have to use some tricks to send these different types of messages over the same stream. In particular, each message is sent as serialized JSON consisting of a `type` and `payload`. The ASCII record separator character `\x1e` is used to delimit the messages, since it cannot appear in JSON.

```
def gen_serialized():
    for i in gen():
        yield json.dumps(i) + "\x1e"

return StreamingResponse(
    gen_serialized(),
    media_type="text/event-stream",
)
```

In addition, the function checks if the body contains a `noop` flag. This is used to warm the containers when the user first loads the page, so that the models can be loaded into memory ahead of time. This is a nice optimization to reduce the apparent latency for the user without needing to keep containers warm.

The other endpoints are more straightforward:

- `POST /transcribe`: Calls `Whisper.transcribe` and returns the results directly.
- `GET /audio/{call_id}`: Polls to check if a `Tortoise.speak` call ID generated above has completed. If yes, it returns the audio data. If not, it returns a `202` status code to indicate that the request should be retried again.
- `DELETE /audio/{call_id}`: Cancels a `Tortoise.speak` call ID generated above. Useful if we want to stop generating audio for a given user.

React frontend

We use the Web Audio API to record snippets of audio from the user's microphone. The file `src/frontend/processor.js` defines an AudioWorkletProcessor that distinguishes between speech and silence, and emits events for speech segments so we can transcribe them.

Pending text-to-speech syntheses are stored in a queue. For the next item in the queue, we use the `GET /audio/{call_id}` endpoint to poll for the audio data.

Finally, the frontend maintains a state machine to manage the state of the conversation and transcription progress. This is implemented with the help of the incredible XState library.

```
const chatMachine = createMachine(
  {
    initial: "botDone",
    states: {
      botGenerating: {
        on: {
          GENERATION_DONE: { target: "botDone", actions: "resetTranscript" },
        },
      },
      botDone: { ... },
      userTalking: {  ... },
      userSilent: { ... },
    },
    ...
  },
```

Steal this example

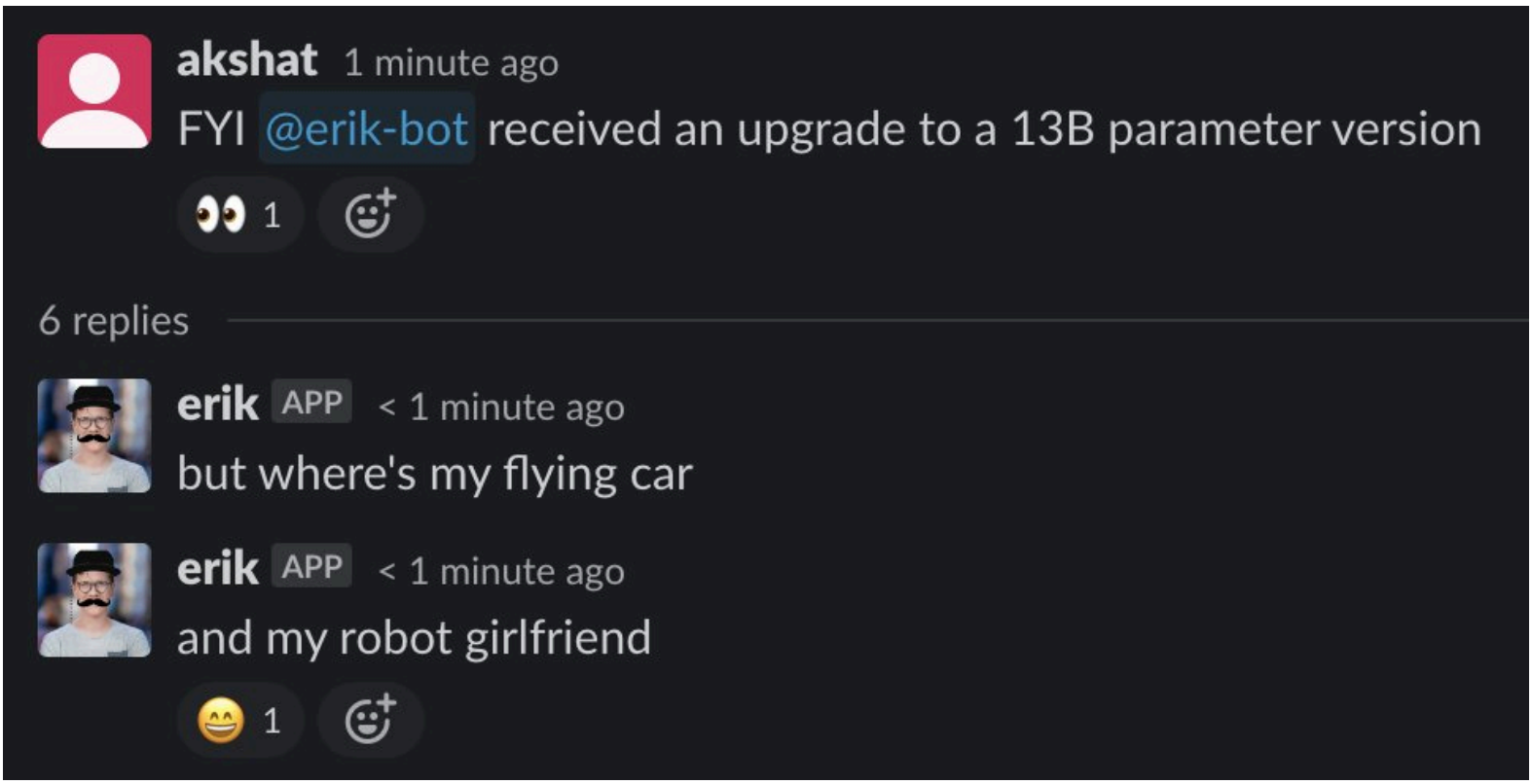
The code for this entire example is available on GitHub. Follow the instructions in the README for how to run or deploy it yourself on Modal.

DoppelBot: Replace your CEO with an LLM

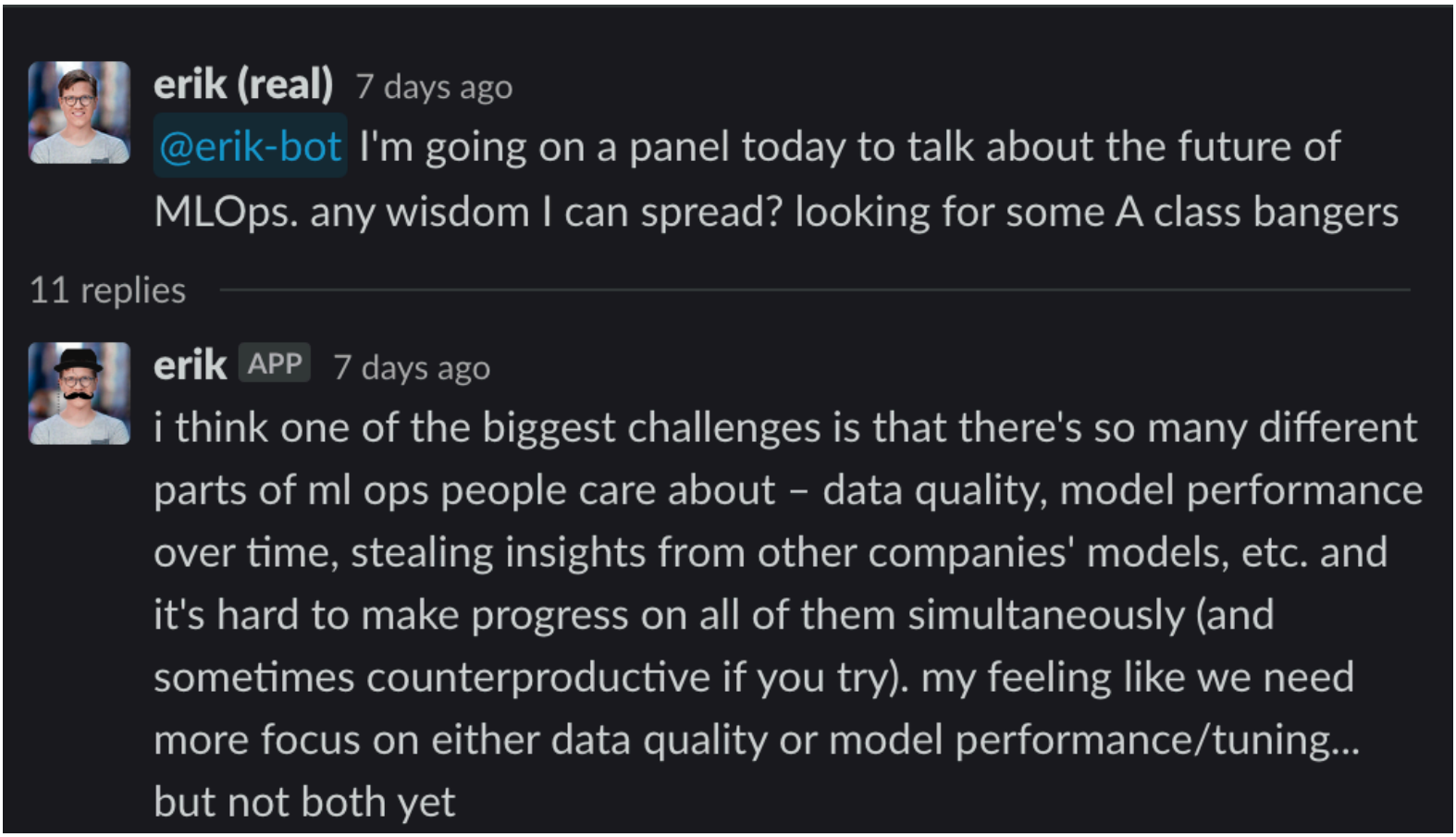
(quick links: add to your own Slack; source code)

Internally at Modal, we spend a *lot* of time talking to each other on Slack. Now, with the advent of open-source large language models, we had started to wonder if all of this wasn't a bit redundant. Could we have these language models bike-shed on Slack for us, so we could spend our time on higher leverage activities such as paddleboarding in Tahiti instead?

To test this, we fine-tuned OpenLLaMa on Erik's Slack messages, and `@erik-bot` was born.



Since then, `@erik-bot` has been an invaluable asset to us, in areas ranging from API design to legal advice to thought leadership.



We were planning on releasing the weights for `@erik-bot` to the world, but all our metrics have been going up and to the right a little too much since we've launched him...

So, we are releasing the next best thing. `DoppelBot` is a Slack bot that you can install in your own workspace, and fine-tune on your own Slack messages. Follow the instructions here to replace your own CEO with an LLM today.

All the components—scraping, fine-tuning, inference and slack event handlers run on Modal, and the code itself is open-source and available here. If you're new to Modal, it's worth reiterating that **all of these components are also serverless and scale to zero**. This means that you can deploy and forget about them, because you'll only pay for compute when your app is used!

How it works

DoppelBot uses the Slack SDK to scrape messages from a Slack workspace, and converts them into prompt/response pairs. It uses these to fine-tune a language model using Low-Rank Adaptation (LoRA), a technique that produces a small adapter that can be merged with the base model when needed, instead of modifying all the parameters in the base model. The fine-tuned adapters for each user are stored in a Modal NetworkFileSystem. When a user `@` s the bot, Slack sends a webhook call to Modal, which loads the adapter for that user and generates a response.

We go into detail into each of these steps below, and provide commands for running each of them individually. To follow along, clone the repo and set up a Slack token for yourself.

Scraping slack

The scraper uses Modal's `.map()` to fetch messages from all public channels in parallel. Each thread is split into contiguous messages from the target users and continguous messages from other users. These become our question/response pairs. Later, these will be fed into the model as prompts in the following format:

```
You are {user}, employee at a fast-growing startup. Below is an input conversation that ta

### Input:
{question}

### Response:
{response}
```

Initial versions of the model were prone to generating short responses — unsurprising, because a majority of Slack communication is pretty terse. Adding a minimum character length for the target user's messages fixed this.

If you're following along at home, you can run the scraper with the following command:

```
modal run src.scrape::scrape --user="<user>"
```

Scraped results are stored in a Modal NetworkFileSystem, so they can be used by the next step.

Fine-tuning

Next, we use the prompts to fine-tune a language model. We chose OpenLLaMa 7B because of its permissive license and high quality relative to its small size. Fine-tuning is done using Low-Rank Adaptation (LoRA), a parameter-efficient fine-tuning technique that produces a small adapter that can be merged with the base model when needed (~60MB for the rank we're using).

Our fine-tuning implementation is based on the excellent alpaca-lora repo that uses the same technique to fine-tune LLaMa using the Alpaca dataset.

Because of the typically small sample sizes we're working with, training for longer than a couple hundred steps (with our batch size of 128) quickly led to overfitting. Admittedly, we haven't thoroughly evaluated the hyperparameter space yet — do reach out to us if you're interested in collaborating on this!

To try this step yourself, run:

```
modal run src.finetune --user="<user>"
```

Inference

At inference time, loading the model with the `LoRA` adapter for a user takes 15-20s, so it's important that we avoid doing this for every incoming request. We also need to maintain separate pools of containers for separate users (since we can only load one model into memory at once). To accomplish this, we're using the hottest new Modal feature: parametrized functions.

With parametrized functions, every user model gets its own pool of containers that scales up when there are incoming requests, and scales to 0 when there's none. Here's what that looks like stripped down to the essentials:

```
@app.cls(gpu=A100(size="40GB"))
class OpenLlamaModel():
    def __init__(self, user: str):

        base_model = LlamaForCausalLM.from_pretrained(...)
        model = PeftModel.from_pretrained(base_model, f"/vol/models/{user}")
        ...

    @method()
    def generate(self, input: str):
```

```
        output = self.model.generate(...)
```

The rest of `inference.py` is just calling generate from the `transformers` library with the input formatted as a prompt.

If you've fine-tuned a model already in the previous step, you can run inference using it now:

```
modal run src.inference --user="<user>"
```

(We have a list of sample inputs in the file, but you can also try it out with your own messages!)

Slack Bot

Finally, it all comes together in `bot.py`. As you might have guessed, all events from Slack are handled by serverless Modal functions. We handle 3 types of events:

- `url_verification`: To verify that this is a Slack app, Slack expects us to return a challenge string.
- `app_mention`: When the bot is mentioned in a channel, we retrieve the recent messages from that thread, do some basic cleaning and call the user's model to generate a response.

```
model = OpenLlamaModel.remote(user, team_id)
result = model.generate(messages)
```

- `doppel` slash command: This command kicks off the scraping → finetuning pipeline for the user.

To deploy the slackbot in its entirety, you need to run:

```
modal deploy src.bot
```

Multi-Workspace Support

Everything we've talked about so far is for a single-workspace Slack app. To make it work with multiple workspaces, we'll need to handle workspace installation and authentication with OAuth, and also store some state for each workspace.

Luckily, Slack's Bolt framework provides a complete (but frugally documented) OAuth implemention. A neat feature is that the OAuth state can be backed by a file system, so all we need to do is point Bolt at a Modal NetworkFileSystem, and then we don't need to worry about managing this state ourselves.

To store state for each workspace, we're using Neon, a serverless Postgres database that's really easy to set up and *just works*. If you're interested in developing a multi-workspace app, follow our instructions on how to set up Neon with Modal.

Next Steps

If you've made it this far, you have just found a way to increase your team's productivity by 10x! Congratulations on the well-earned vacation! 🎉

If you're interested in learning more about Modal, check out our docs and other examples.

Write to Google Sheets

View on GitHub

In this tutorial, we'll show how to use Modal to schedule a daily update of a dataset from an analytics database to Google Sheets.

Entering credentials

We begin by setting up some credentials that we'll need in order to access our database and output spreadsheet. To do that in a secure manner, we log in to our Modal account on the web and go to the Secrets section.

Database

First we will enter our database credentials. The easiest way to do this is to click **New secret** and select the **Postgres compatible** secret preset and fill in the requested information. Then we press **Next** and name our secret "example-postgres-secret" and click **Create**.

Google Sheets/GCP

We'll now add another Secret for Google Sheets access through Google Cloud Platform. Click **New secret** and select the Google Sheets preset.

In order to access the Google Sheets API, we'll need to create a *Service Account* in Google Cloud Platform. You can skip this step if you already have a Service Account json file.

1. Sign up to Google Cloud Platform or log in if you haven't (https://cloud.google.com/).

2. Go to https://console.cloud.google.com/.
3. In the navigation pane on the left, go to **IAM & Admin** > **Service Accounts**.
4. Click the **+ CREATE SERVICE ACCOUNT** button.
5. Give the service account a suitable name, like “sheet-access-bot”. Click **Done**. You don’t have to grant it any specific access privileges at this time.
6. Click your new service account in the list view that appears and navigate to the **Keys** section.
7. Click **Add key** and choose **Create new key**. Use the **JSON** key type and confirm by clicking **Create**.
8. A json key file should be downloaded to your computer at this point. Copy the contents of that file and use it as the value for the **SERVICE_ACCOUNT_JSON** field in your new secret.

We’ll name this other secret “gsheets-secret”.

Now you can access the values of your secrets from modal functions that you annotate with the corresponding EnvDict includes, e.g.:

```
import os

import modal

app = modal.App(
    "example-db-to-sheet"
)  # Note: prior to April 2024, "app" was called "stub"


@app.function(secrets=[modal.Secret.from_name("example-postgres-secret")])
def my_func():
    # automatically filled from the specified secret
    print("Host is " + os.environ["PGHOST"])
```

In order to connect to the database, we’ll use the `psycopg2` Python package. To make it available to your Modal function you need to supply it with an `image` argument that tells Modal how to build the container image that contains that package. We’ll base it off of the `Image.debian_slim` base image that’s built into modal, and make sure to install the required binary packages as well as the psycopg2 package itself:

```
pg_image = (
    modal.Image.debian_slim(python_version="3.11")
    .apt_install("libpq-dev")
    .pip_install("psycopg2~=2.9.9")
)
```

Since the default keynames for a **Postgres compatible** secret correspond to the environment variables that `psycopg2` looks for, you can now easily connect to the database even without explicit credentials in your code. We'll create a simple function that queries the city for each user in our dummy `users` table:

```
@app.function(
    image=pg_image,
    secrets=[modal.Secret.from_name("example-postgres-secret")],
)
def get_db_rows():
    import psycopg2

    conn = psycopg2.connect()  # no explicit credentials needed
    cur = conn.cursor()
    cur.execute("SELECT city FROM users")
    return [row[0] for row in cur.fetchall()]
```

Note that we import psycopg2 inside our function instead of the global scope. This allows us to run this Modal function even from an environment where psycopg2 is not installed. We can test run this function using the `modal run` shell command: `modal run db_to_sheet.py::app.get_db_rows` .

Applying Python logic

For each city in our source data we'll make an online lookup of the current weather using the http://openweathermap.org API. To do this, we'll add the API key to another modal secret. We'll use a custom secret called "weather-secret" with the key `OPENWEATHER_API_KEY` containing our API key for OpenWeatherMap.

```
requests_image = modal.Image.debian_slim(python_version="3.11").pip_install(
    "requests~=2.31.0"
)
```

```
@app.function(
    image=requests_image,
    secrets=[modal.Secret.from_name("weather-secret")],
)
def city_weather(city):
    import requests

    url = "https://api.openweathermap.org/data/2.5/weather"
    params = {"q": city, "appid": os.environ["OPENWEATHER_API_KEY"]}
    response = requests.get(url, params=params)
    weather_label = response.json()["weather"][0]["main"]
    return weather_label
```

We'll make use of Modal's built-in `function.map` method to create our report. `function.map` makes it really easy to parallelise work by executing a function for a larger sequence of input data. For this example we'll make a simple count of rows per weather type, using Python's standard library `collections.Counter`.

```
from collections import Counter


@app.function()
def create_report(cities):
    # run city_weather for each city in parallel
    user_weather = city_weather.map(cities)
    users_by_weather = Counter(user_weather).items()
    return users_by_weather
```

Let's try to run this! To make it simple to trigger the function with some predefined input data, we create a "local entrypoint" `main` that can be easily triggered from the command line:

```
@app.local_entrypoint()
def main():
    cities = [
        "Stockholm,,Sweden",
        "New York,NY,USA",
        "Tokyo,,Japan",
    ]
    print(create_report.remote(cities))
```

Running the local entrypoint using `modal run db_to_sheet.py` should print something like: `dict_items([('Clouds', 3)])`. Note that since this file only has a single app, and the app has only one local entrypoint we only have to specify the file to run - the function/entrypoint is inferred.

In this case the logic is quite simple, but in a real world context you could have applied a machine learning model or any other tool you could build into a container to transform the data.

Sending output to a Google Sheet

We'll set up a new Google Sheet to send our report to. Using the "Sharing" dialog in Google Sheets, we make sure to share the document to the service account's email address (the value of the `client_email` field in the json file) and make the service account an editor of the document.

The URL of a Google Sheet is something like: `https://docs.google.com/spreadsheets/d/1wOktal.....IJR77jD8Do`.

We copy the part of the URL that comes after `/d/` - that is the *key* of the document which we'll refer to in our code. We'll make use of the `pygsheets` python package to authenticate with Google Sheets and then update the spreadsheet with information from the report we just created:

```
pygsheets_image = modal.Image.debian_slim(python_version="3.11").pip_install(
    "pygsheets~=2.0.6"
)


@app.function(
    image=pygsheets_image,
    secrets=[modal.Secret.from_name("gsheets-secret")],
)
def update_sheet_report(rows):
    import pygsheets

    gc = pygsheets.authorize(service_account_env_var="SERVICE_ACCOUNT_JSON")
    document_key = "1RqQrJ6Ikf611adKunm8tmL1mKzHLjNwLWm_T7mfXSYA"
    sh = gc.open_by_key(document_key)
    worksheet = sh.sheet1
    worksheet.clear("A2")

    worksheet.update_values("A2", [list(row) for row in rows])
```

At this point, we have everything we need in order to run the full program. We can put it all together in another Modal function, and add a schedule argument so it runs every day automatically:

```
@app.function(schedule=modal.Cron("0 0 * * *"))
def db_to_sheet():
    rows = get_db_rows.remote()
    report = create_report.remote(rows)
    update_sheet_report.remote(report)
    print("Updated sheet with new weather distribution")
    for weather, count in report:
        print(f"{weather}: {count}")
```

This entire app can now be deployed using `modal deploy db_to_sheet.py`. The apps page shows our cron job's execution history and lets you navigate to each invocation's logs. To trigger a manual run from your local code during development, you can also trigger this function using the cli: `modal run db_to_sheet.py::app.db_to_sheet`

Note that all of the @app.function() annotated functions above run remotely in isolated containers that are specified per function, but they are called as seamlessly as using regular Python functions. This is a simple showcase of how you can mix and match functions that use different environments and have

them feed into each other or even call each other as if they were all functions in the same local program.

Play with the ControlNet demos

View on GitHub

This example allows you to play with all 10 demonstration Gradio apps from the new and amazing ControlNet project. ControlNet provides a minimal interface allowing users to use images to constrain StableDiffusion's generation process. With ControlNet, users can easily condition the StableDiffusion image generation with different spatial contexts including a depth maps, segmentation maps, scribble drawings, and keypoints!

Imports and config preamble

```
import importlib
import os
import pathlib
from dataclasses import dataclass, field

from fastapi import FastAPI
from modal import App, Image, Secret, asgi_app
```

Below are the configuration objects for all **10** demos provided in the original lllyasviel/ControlNet repo. The demos each depend on their own custom pretrained StableDiffusion model, and these models are 5-6GB each. We can only run one demo at a time, so this module avoids downloading the model and 'detector' dependencies for all 10 demos and instead uses the demo configuration object to download only what's necessary for the chosen demo.

Even just limiting our dependencies setup to what's required for one demo, the resulting container image is *huge*.

```
@dataclass(frozen=True)
class DemoApp:
    """Config object defining a ControlNet demo app's specific dependencies."""

    name: str
    model_files: list[str]
    detector_files: list[str] = field(default_factory=list)


demos = [
    DemoApp(
        name="canny2image",
        model_files=[
            "https://huggingface.co/lllyasviel/ControlNet/resolve/main/models/control_sd15
        ],
    ),
    DemoApp(
        name="depth2image",
        model_files=[
            "https://huggingface.co/lllyasviel/ControlNet/resolve/main/models/control_sd15
        ],
        detector_files=[
            "https://huggingface.co/lllyasviel/ControlNet/resolve/main/annotator/ckpts/dpt
        ],
    ),
    DemoApp(
        name="fake_scribble2image",
        model_files=[
            "https://huggingface.co/lllyasviel/ControlNet/resolve/main/models/control_sd15
        ],
        detector_files=[
            "https://huggingface.co/lllyasviel/ControlNet/resolve/main/annotator/ckpts/net
        ],
    ),
    DemoApp(
        name="hed2image",
        model_files=[
            "https://huggingface.co/lllyasviel/ControlNet/resolve/main/models/control_sd15
        ],
        detector_files=[
            "https://huggingface.co/lllyasviel/ControlNet/resolve/main/annotator/ckpts/net
```

```
        ],
    ),
    DemoApp(
        name="hough2image",
        model_files=[
            "https://huggingface.co/lllyasviel/ControlNet/resolve/main/models/control_sd15
        ],
        detector_files=[
            "https://huggingface.co/lllyasviel/ControlNet/resolve/main/annotator/ckpts/mls
            "https://huggingface.co/lllyasviel/ControlNet/resolve/main/annotator/ckpts/mls
        ],
    ),
    DemoApp(
        name="normal2image",
        model_files=[
            "https://huggingface.co/lllyasviel/ControlNet/resolve/main/models/control_sd15
        ],
    ),
    DemoApp(
        name="pose2image",
        model_files=[
            "https://huggingface.co/lllyasviel/ControlNet/resolve/main/models/control_sd15
        ],
        detector_files=[
            "https://huggingface.co/lllyasviel/ControlNet/resolve/main/annotator/ckpts/bod
            "https://huggingface.co/lllyasviel/ControlNet/resolve/main/annotator/ckpts/han
        ],
    ),
    DemoApp(
        name="scribble2image",
        model_files=[
            "https://huggingface.co/lllyasviel/ControlNet/resolve/main/models/control_sd15
        ],
    ),
    DemoApp(
        name="scribble2image_interactive",
        model_files=[
            "https://huggingface.co/lllyasviel/ControlNet/resolve/main/models/control_sd15
        ],
    ),
    DemoApp(
        name="seg2image",
        model_files=[
            "https://huggingface.co/lllyasviel/ControlNet/resolve/main/models/control_sd15
        ],
        detector_files=[
            "https://huggingface.co/lllyasviel/ControlNet/resolve/main/annotator/ckpts/upe
        ],
    ),
]
demos_map: dict[str, DemoApp] = {d.name: d for d in demos}
```

Pick a demo, any demo

Simply by changing the `DEMO_NAME` below, you can change which ControlNet demo app is setup and run by this Modal script.

```
DEMO_NAME = "scribble2image"  # Change this value to change the active demo app.
selected_demo = demos_map[DEMO_NAME]
```

Setting up the dependencies

ControlNet requires *a lot* of dependencies which could be fiddly to setup manually, but Modal's programmatic container image building Python APIs handle this complexity straightforwardly and automatically.

To run any of the 10 demo apps, we need the following:

1. a base Python 3 Linux image (we use Debian Slim)
2. a bunch of third party PyPi packages
3. `git`, so that we can download the ControlNet source code (there's no `controlnet` PyPi package)
4. some image process Linux system packages, including `ffmpeg`
5. and demo specific pre-trained model and detector `.pth` files

That's a lot! Fortunately, the code below is already written for you that stitches together a working container image ready to produce remarkable ControlNet images.

Note: a ControlNet model pipeline is now available in Huggingface's `diffusers` package. But this does not contain the demo apps.

```
def download_file(url: str, output_path: pathlib.Path):
    import httpx
    from tqdm import tqdm

    with open(output_path, "wb") as download_file:
        with httpx.stream("GET", url, follow_redirects=True) as response:
            total = int(response.headers["Content-Length"])
            with tqdm(
                total=total, unit_scale=True, unit_divisor=1024, unit="B"
            ) as progress:
                num_bytes_downloaded = response.num_bytes_downloaded
                for chunk in response.iter_bytes():
                    download_file.write(chunk)
                    progress.update(
                        response.num_bytes_downloaded - num_bytes_downloaded
```

```
                )
                num_bytes_downloaded = response.num_bytes_downloaded


def download_demo_files() -> None:
    """
    The ControlNet repo instructs: 'Make sure that SD models are put in "ControlNet/models
    'ControlNet' is just the repo root, so we place in /root/models.

    The ControlNet repo also instructs: 'Make sure that... detectors are put in "ControlNe
    'ControlNet' is just the repo root, so we place in /root/annotator/ckpts.
    """
    demo = demos_map[os.environ["DEMO_NAME"]]
    models_dir = pathlib.Path("/root/models")
    for url in demo.model_files:
        filepath = pathlib.Path(url).name
        download_file(url=url, output_path=models_dir / filepath)
        print(f"download complete for {filepath}")

    detectors_dir = pathlib.Path("/root/annotator/ckpts")
    for url in demo.detector_files:
        filepath = pathlib.Path(url).name
        download_file(url=url, output_path=detectors_dir / filepath)
        print(f"download complete for {filepath}")
    print("🎉 finished baking demo file(s) into image.")


image = (
    Image.debian_slim(python_version="3.10")
    .pip_install(
        "gradio==3.16.2",
        "albumentations==1.3.0",
        "opencv-contrib-python",
        "imageio==2.9.0",
        "imageio-ffmpeg==0.4.2",
        "pytorch-lightning==1.5.0",
        "omegaconf==2.1.1",
        "test-tube>=0.7.5",
        "streamlit==1.12.1",
        "einops==0.3.0",
        "transformers==4.19.2",
        "webdataset==0.2.5",
        "kornia==0.6",
        "open_clip_torch==2.0.2",
        "invisible-watermark>=0.1.5",
        "streamlit-drawable-canvas==0.8.0",
        "torchmetrics==0.6.0",
        "timm==0.6.12",
        "addict==2.4.0",
        "yapf==0.32.0",
        "prettytable==3.6.0",
        "safetensors==0.2.7",
```

```
        "basicsr==1.4.2",
        "tqdm~=4.64.1",
    )
    # xformers library offers performance improvement.
    .pip_install("xformers", pre=True)
    .apt_install("git")
    # Here we place the latest ControlNet repository code into /root.
    # Because /root is almost empty, but not entirely empty, `git clone` won't work,
    # so this `init` then `checkout` workaround is used.
    .run_commands(
        "cd /root && git init .",
        "cd /root && git remote add --fetch origin https://github.com/lllyasviel/ControlNe
        "cd /root && git checkout main",
    )
    .apt_install("ffmpeg", "libsm6", "libxext6")
    .run_function(
        download_demo_files,
        secrets=[Secret.from_dict({"DEMO_NAME": DEMO_NAME})],
    )
)
app = App(
    name="example-controlnet", image=image
)  # Note: prior to April 2024, "app" was called "stub"

web_app = FastAPI()
```

Serving the Gradio web UI

Each ControlNet gradio demo module exposes a `block` Gradio interface running in queue-mode, which is initialized in module scope on import and served on `0.0.0.0`. We want the block interface object, but the queueing and launched webserver aren't compatible with Modal's serverless web endpoint interface, so in the `import_gradio_app_blocks` function we patch out these behaviors.

```
def import_gradio_app_blocks(demo: DemoApp):
    from gradio import blocks

    # The ControlNet repo demo scripts are written to be run as
    # standalone scripts, and have a lot of code that executes
    # in global scope on import, including the launch of a Gradio web server.
    # We want Modal to control the Gradio web app serving, so we
    # monkeypatch the .launch() function to be a no-op.
    blocks.Blocks.launch = lambda self, server_name: print(
        "launch() has been monkeypatched to do nothing."
    )

    # each demo app module is a file like gradio_{name}.py
    module_name = f"gradio_{demo.name}"
    mod = importlib.import_module(module_name)
```

```
    blocks = mod.block
    # disable queueing mode, which is incompatible with our Modal web app setup.
    blocks.enable_queue = False
    return blocks
```

Because the ControlNet gradio apps are so time and compute intensive to cold-start, the web app function is limited to running just 1 warm container (concurrency_limit=1). This way, while playing with the demos we can pay the cold-start cost once and have all web requests hit the same warm container. Spinning up extra containers to handle additional requests would not be efficient given the cold-start time. We set the container_idle_timeout to 600 seconds so the container will be kept running for 10 minutes after the last request, to keep the app responsive in case of continued experimentation.

```
@app.function(
    gpu="A10G",
    concurrency_limit=1,
    container_idle_timeout=600,
)
@asgi_app()
def run():
    from gradio.routes import mount_gradio_app

    # mount for execution on Modal
    return mount_gradio_app(
        app=web_app,
        blocks=import_gradio_app_blocks(demo=selected_demo),
        path="/",
    )
```

Have fun!

Serve your chosen demo app with `modal serve controlnet_gradio_demos.py`. If you don't have any images ready at hand, try one that's in the `06_gpu_and_ml/controlnet/demo_images/` folder.

StableDiffusion was already impressive enough, but ControlNet's ability to so accurately and intuitively constrain the image generation process is sure to put a big, dumb grin on your face.

MultiOn: Twitter News Agent

View on GitHub

In this example, we use Modal to deploy a cron job that periodically checks for AI news everyday and tweets it on Twitter using the MultiOn Agent API.

Import and define the app

Let's start off with imports, and defining a Modal app.

```
import os

import modal

app = modal.App(
    "multion-news-tweet-agent"
)  # Note: prior to April 2024, "app" was called "stub"
```

Searching for AI News

Let's also define an image that has the `multion` package installed, so we can query the API.

```
multion_image = modal.Image.debian_slim().pip_install("multion")
```

We can now define our main entrypoint, that uses MultiOn to scrape AI news everyday and post it on our twitter account. We specify a schedule in the function decorator, which means that our function will run automatically at the given interval.

Set up MultiOn

MultiOn is a next-gen Web Action Agent that can take actions on behalf of the user. You can watch it in action here: Youtube demo.

The MultiOn API enables building the next level of web automation & custom AI agents capable of performing complex actions on the internet with just a few lines of code.

To get started, first create an account with MultiOn, install the MultiOn chrome extension and login to your Twitter account in your browser. To use the API create a MultiOn API Key and store it as a modal secret on the dashboard

```
@app.function(
    image=multion_image, secrets=[modal.Secret.from_name("MULTION_API_KEY")]
)
def news_tweet_agent():
    # Import MultiOn
    import multion

    # Login to MultiOn using the API key
    multion.login(use_api=True, multion_api_key=os.environ["MULTION_API_KEY"])

    # Enable the Agent to run locally
    multion.set_remote(False)

    params = {
        "url": "https://www.multion.ai",
        "cmd": "Go to twitter (im already signed in). Search for the last tweets i made (c
        "maxSteps": 100,
    }

    response = multion.browse(params)

    print(f"MultiOn response: {response}")
```

Test running

We can now test run our scheduled function as follows: `modal run multion_news_agent.py.py::app.news_tweet_agent`

Defining the schedule and deploying

Let's define a function that will be called by Modal every day.

```
@app.function(schedule=modal.Cron("0 9 * * *"))
def run_daily():
    news_tweet_agent.remote()
```

In order to deploy this as a persistent cron job, you can run `modal deploy multion_news_agent.py`.

Once the job is deployed, visit the apps page page to see its execution history, logs and other stats.

Miscellaneous AI examples

Looking for how to make a popular machine learning library work with Modal? There's a guide for that:

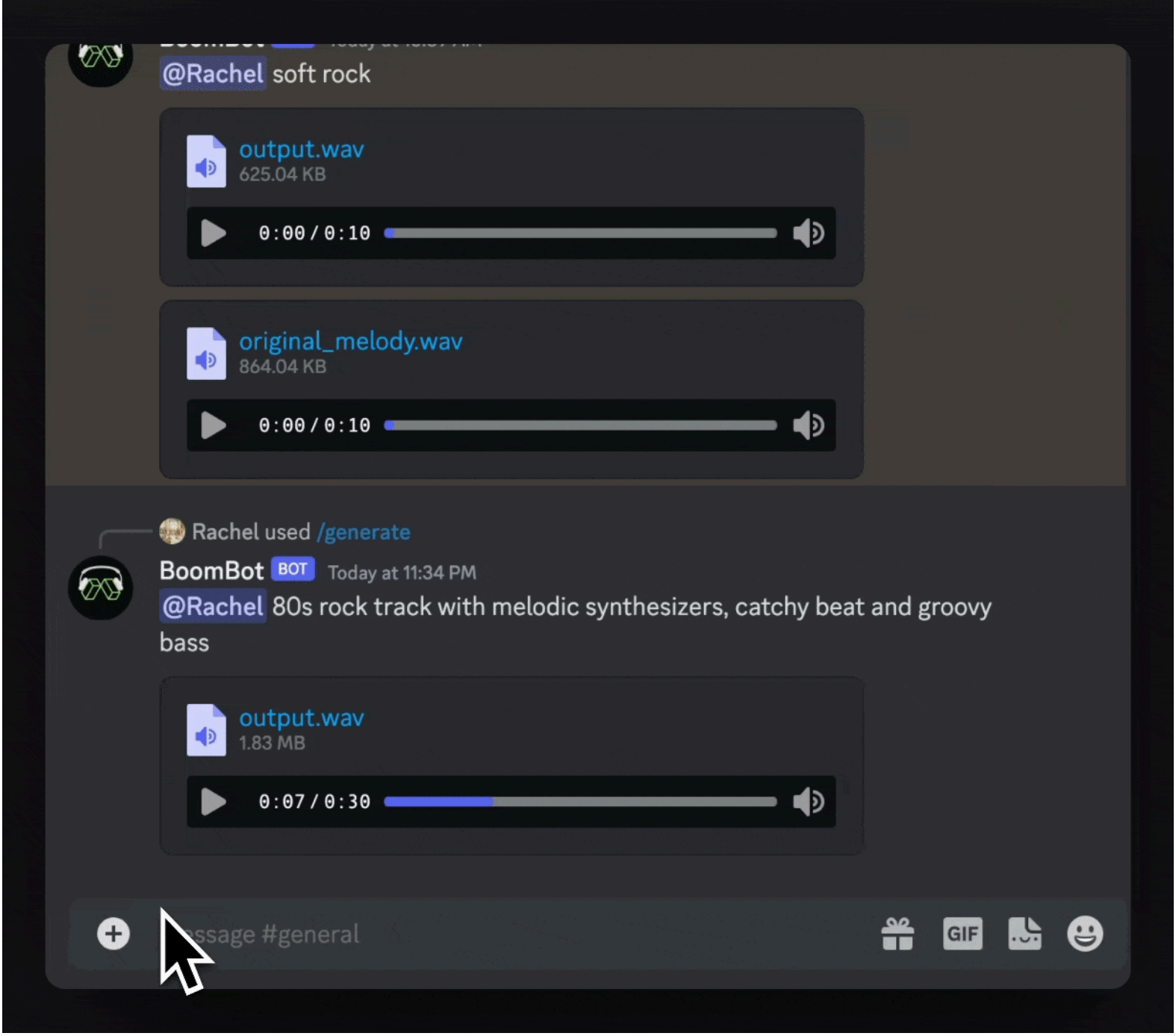
- Classifier training with TensorFlow and TensorBoard
- Fine-tune Flan-T5 and monitor with TensorBoard
- Generate synthetic data with Jsonformer
- Real-time object detection with webcam input
- Run ComfyUI
- Stable Diffusion with A1111

BoomBot: Create your own music samples on Discord

(quick links: try it out on Discord; watch demo with audio; view source code)

MusicGen is the latest milestone language model in conditional music generation, with great results. We wanted a space to easily play around with the model and share our creations, so we created a Discord community featuring BoomBot, an on-demand music sample generator.

You can call BoomBot in the Discord server by simply typing `/generate`, then prompting it with a text description of the music you'd like to create, and even a file of the melody you'd like to condition on, along with other specifiable parameters.



Everything, from the backend API to our React frontend, is deployed serverlessly using Modal, and the code is available here.

Code overview

BoomBot runs the MusicGen model remotely in a GPU-accelerated container — when a user interacts with the Discord bot using the slash command `/generate`, Discord sends a webhook call to Modal, which then generates a music sample using the text description and other user inputs from the command. We also deploy a React single page application as static files into a Modal web endpoint for our web app.

We go into detail into each of these steps below, and provide commands for running each of them individually. To follow along, clone the repo and set up a Discord token for yourself.

Language model

We use Audiocraft, a PyTorch library that provides the code and models for MusicGen, and load both the 3.3B `large` and 1.5B `melody` models to use depending on the user input (large for just text, melody for text and melody). We can install all our dependencies and "bake" both models into our image to avoid downloading our models during inference and take advantage of Modal's incredibly fast cold-start times:

```
def download_models():
    from audiocraft.models import MusicGen

    MusicGen.get_pretrained("large")
    MusicGen.get_pretrained("melody")


image = (
    modal.Image.debian_slim()
    .apt_install("git", "ffmpeg")
    .pip_install(
        "pynacl", # for Discord authentication
        "torch",
        "soundfile",
        "pydub",
        "git+https://github.com/facebookresearch/audiocraft.git",
    )
    .run_function(download_models, gpu="any")
)
app.image = image
```

We then write our model code within Modal's `@app.cls` decorator, with the `generate` function processing the user input and generating audio as bytes that we can save to a file later.

It's useful to test out our model directly before we build out our backend API, so we define a `local_entrypoint`. We can then run our model from the CLI using inputs of our choice:

```
modal run main.py --prompt "soothing jazz" --duration 20 --format "mp3" --melody "https://
```

Discord bot

Now that we've loaded our model and wrote our function, we'd like to trigger it from Discord. We can do this using slash commands — a feature that lets you register a text command on Discord that triggers a custom webhook when a user interacts with it. We handle all our Discord events in a FastAPI app in `bot.py`. Luckily, we can deploy this app easily and serverlessly using Modal's @asgi_app decorator.

Create a Discord app

To connect our model to a Discord bot, we're first going to create an application on the Discord Developer Portal.

1. Go to the Discord Developer Portal and login with your Discord account.
2. On the portal, go to **Applications** and create a new application by clicking **New Application** in the top right next to your profile picture.
3. Copy your Discord application's **Public Key** (in **General Information**) and create a custom Modal secret for it. On Modal's secret creation page, paste the public key as the secret value with the key `DISCORD_PUBLIC_KEY`, then name this secret `boombot-discord-secret`.

Then go back to your application on the Discord Developer Portal, as we need to do a few more things to finish setting up our bot.

Register a Slash Command

Next, we're going to register a command for our Discord app via an HTTP endpoint.

Run the following command in your terminal, replacing the appropriate variable inputs. `BOT_TOKEN` can be found by resetting your token in the application's **Bot** section, and `CLIENT_ID` is the **Application ID** available in **General Information**.

```
BOT_TOKEN='replace_with_bot_token'
CLIENT_ID='replace_with_application_id'
curl -X POST \
-H 'Content-Type: application/json' \
-H "Authorization: Bot $BOT_TOKEN" \
-d '{
  "name":"generate",
  "description":"generate music",
  "options":[
    {
      "name":"prompt",
      "description":"Describe the music you want to generate",
      "type":3,
      "required":true
    },
    {
      "name":"duration",
      "description":"Duration of clip, in seconds (max 60)",
      "type":4,
      "required":false,
      "min_value":2,
      "max_value":60
    },
    {
      "name":"melody",
      "description":"File of melody you\''d like to condition (takes first 30 secs if lon
      "type":11,
```

```
      "required":false
    },
    {
      "name":"format",
      "description":"Desired format of output (default .wav)",
      "type":3,
      "required":false,
      "choices":[
        {
          "name": ".wav",
          "value": "wav"
        },
        {
          "name": ".mp3",
          "value": "mp3"
        }
      ]
    }
  ]
}' "https://discord.com/api/v10/applications/$CLIENT_ID/commands"
```

This will register a Slash Command for your bot named `generate`, and has a few parameters like `prompt`, `duration`, and `format`. More information about the command structure can be found in the Discord docs here.

Deploy a Modal web endpoint

We then create a `POST /generate` endpoint in `bot.py` using FastAPI and Modal's @asgi_app decorator to handle interactions with our Discord app (so that every time a user does a slash command, we can respond to it).

Note that since Discord requires an interaction response within 3 seconds, we use `spawn` to kick off `Audiocraft.generate` as a background task while returning a `defer` message to Discord within the time limit. We then update our response with the results once the model has finished running.

You can deploy this app by running the following command from your root directory:

```
modal deploy src.bot
```

Copy the Modal URL that is printed in the output and go back to your application's **General Information** section on the Discord Developer Portal. Paste the URL, making sure to append the path of your `POST` endpoint, in the **Interactions Endpoint URL** field, then click **Save Changes**. If your endpoint is valid, it will properly save and you can start receiving interactions via this web endpoint.

Finish setting up Discord bot

To start using the Slash Command you just set up, you need to invite the bot to a Discord server. To do so, go to your application's **OAuth2** section on the Discord Developer Portal. Select `applications.commands` as the scope of your bot and copy the invite URL that is generated at the bottom of the page.

Paste this URL in your browser, then select your desired server (create a new server if needed) and click **Authorize**. Now you can open your Discord server and type `/{name of your slash command}` - your bot should be connected and ready for you to use!

React frontend

We also added a simple React frontend to our FastAPI app to serve as a landing page for our Discord server.

We added the frontend to our existing app in `bot.py` by simply using `Mount` to mount our local directory at `/assets` in our container, then instructing FastAPI to serve this static file directory at our root path.

For a more in-depth tutorial on deploying your web app on Modal, refer to our Document OCR web app example.

Run this app on Modal

All the source code for this example can be found on Github.

If you're interested in learning more about Modal, check out our docs and other examples.