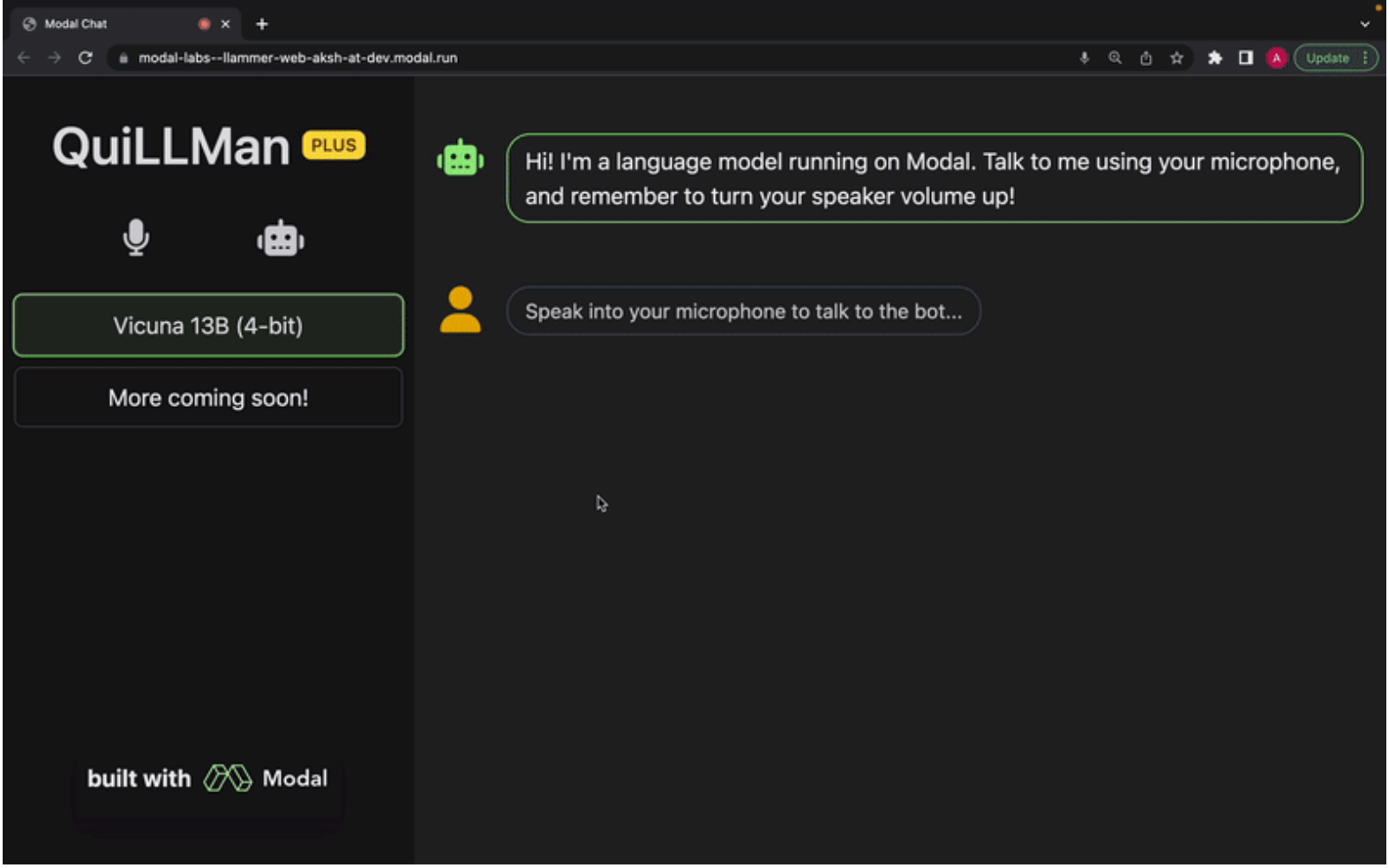
QuiLLMan: Voice Chat with LLMs

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.