We Stopped Paying Per Token and Put the Models on Our Own Machine

One workstation, a stack of open models from Hugging Face, a few hundred lines of Python, and an ngrok tunnel - how Protovate builds and tests AI apps before a single API bill arrives.

32 GB GPU	**128 GB RAM**	**LOCAL AI LAB**

Prateek Sharma | Technical Lead, Protovate
protovate.com

Every AI prototype starts with the same question, and it is not a technical one. It is: what happens to the bill, and what happens to the data? You want to try six different models against a client's real content, run the same prompt two hundred times to see how stable it is, and let a mobile app hammer the endpoint all week during QA. Do that on a hosted API and you are metering every experiment. Do it with client data and you have a conversation to have with legal first.

So we did the obvious thing. We took the workstation sitting in the Protovate office, 32 GB of GPU memory and 128 GB of system RAM, and turned it into an AI lab. Open models downloaded straight from Hugging Face. Python wrappers around them. A tunnel to the outside world so our apps could call it from anywhere. The whole setup was built in evenings, and most of the code was written in conversation with Claude.

Why open weights, and why locally

The hosted APIs are excellent and we use them in production. But local models solve four problems that money cannot:

- **Nothing leaves the building.** Client documents, internal transcripts, unreleased product copy. It stays on a machine we own.
- **Experiments are free.** Once the model is on disk, running it a thousand times costs electricity. That changes how boldly you test.
- **No rate limits during QA.** A testing team can pound an endpoint for a week without anyone watching a dashboard nervously.
- **You learn what is actually happening.** Loading weights yourself, watching VRAM fill, tuning batch size. You understand the thing you are shipping.

The stack, end to end

Pick the model, not the hype

Hugging Face hosts hundreds of thousands of models and most of them are not for you. The filters that matter are size, licence and format. Check the parameter count against your VRAM before anything else, then read the licence properly, because "open" and "usable in a commercial client project" are not the same sentence.

Pull the weights once

The Hugging Face CLI caches everything locally, so a model downloads once and every script afterwards loads from disk. Point the cache at a drive with room to spare. These files are large, and you will collect more of them than you planned.

```
# one-time setup
pip install huggingface_hub transformers accelerate diffusers torch

# pull the weights into the local cache
hf download <org>/<model> --local-dir ./models/<model>
```

Wrap it in an API, not a notebook

This is the step that turns an experiment into something a mobile app can use. Load the model once at startup, keep it resident in GPU memory, and expose a plain HTTP endpoint. A notebook reloads the model every time you rerun a cell. A service loads it once and answers in milliseconds.

```
from fastapi import FastAPI
from pydantic import BaseModel
from transformers import pipeline

app = FastAPI()

# loaded ONCE at startup, stays in VRAM
generator = pipeline(
    "text-generation",
    model="./models/<model>",
    device_map="auto",
)

class Req(BaseModel):
    prompt: str
    max_new_tokens: int = 256

@app.post("/generate")
def generate(req: Req):
    out = generator(req.prompt, max_new_tokens=req.max_new_tokens)
    return {"text": out[0]["generated_text"]}
```

Open a tunnel and hand out the URL

An endpoint on `localhost` is useless to a tester holding a phone. One ngrok command gives you a public HTTPS address that forwards straight to the workstation, so an Android build, an iOS build and a designer's browser can all hit the same model at the same time.

```
ngrok http 8000
# → https://xxxx-xx-xx.ngrok-free.app  points at the box under the desk
```

Put a token check in front of it. A public tunnel is public.

Test like it is production

Same request shape as the hosted API you might swap in later, same error handling, same timeouts. When the app is written against a clean HTTP contract, moving from the workstation to a cloud endpoint later is a config change, not a rewrite.

Image generation is the same shape, with a different appetite

Text models want VRAM for weights. Image models want VRAM for weights and then a burst more during sampling, and they punish you for large batch sizes far faster. The pattern is identical though: load the pipeline once, expose a POST endpoint, return the image as base64 or write it to a static folder and return the URL. The second option keeps your JSON small and your logs readable.

What surprised us was how much the surrounding code mattered. Generation quality was fine out of the box. The work was everything else: queueing requests so two testers do not collide, capping resolution so nobody accidentally requests a poster, cleaning up files, and returning a useful error instead of a stack trace when the GPU runs out of room.

Where Claude fitted into this

Almost every line of Python in this setup was written in conversation with Claude, and the pattern that made it work is worth describing, because it is not "generate me an app".

Environment setup is where it saved the most time. CUDA versions, torch builds, driver mismatches, a dependency conflict that only shows up on the fourth import. This is the least interesting part of AI work and the part that eats whole evenings. Pasting the actual traceback and getting a specific next command beat searching through five-year-old forum threads every time.

Errors first, opinions second. The habit that made the difference was empirical: reproduce the failure, capture the real output, hand over the log rather than a description of the log. An out-of-memory error tells you exactly which allocation failed and at what size. Feed that in and you get a real fix. Describe it as "it crashes sometimes" and you get plausible guesses.

Complete files, not fragments. I ask for whole drop-in files rather than diffs, and for changes that are strictly additive to code that already works. On a service that takes two minutes to reload a model, a broken edit is an expensive mistake.

The actual lesson Using AI to build AI infrastructure is not about generating clever code. It is about compressing the setup, the dependency archaeology and the debugging, so the hours go into the product instead of the toolchain.

Five things we learned the hard way

1. Do the VRAM arithmetic before you download

A rough rule: parameters multiplied by bytes per parameter, plus overhead for activations and context. Full precision is four bytes per parameter, half precision is two, and quantized formats bring it down further. Guessing here costs you an 80 GB download and a crash.

2. Quantization is the difference between "runs" and "runs well"

A model that will not fit at full precision often fits comfortably quantized, with quality loss that is barely visible for most application work. This is the single highest-leverage lever on a fixed GPU budget.

3. Model load time is not inference time

Reading tens of gigabytes of weights off disk takes real seconds. Load at startup, never per request. If your first API call takes forty seconds and the rest take one, you have your answer about where the time went.

4. A tunnel URL is a moving target

Free ngrok URLs change on every restart, which breaks whatever the mobile build has hardcoded. Put the base URL in remote config or a settings screen so testers can update it without a new build. We learned this after shipping exactly one build with a dead URL baked in.

5. System RAM buys you patience

128 GB of RAM does not make inference faster, but it lets you offload layers when a model does not quite fit, keep several models cached, and preprocess large datasets without thrashing. It is the quiet half of the machine and it earns its place.

So when should you run local, and when should you pay the API?

Situation	Local workstation	Hosted API
Early prototyping and model comparison	Yes, unlimited runs	Meters every experiment

Situation	Local workstation	Hosted API
Sensitive or client-owned data	Yes, nothing leaves	Needs a data agreement
Heavy QA cycles	Yes, no rate limits	Costly and throttled
Best-in-class reasoning quality	Behind the frontier	Yes, clearly ahead
Production traffic at scale	One box, one failure point	Yes, built for it

We use both, deliberately. The workstation is where ideas get tested, where clients' data stays put, and where the team learns how these systems actually behave. The hosted APIs are where the finished product runs. Having both means the decision to spend is made after the evidence, not before it.

That is the real return on one machine under a desk. Not the saved API spend, though that is nice. It is that "let us just try it" stops being a budget conversation.

Let's Build What's Next

Have an AI, software, or product idea you want to validate or build? Schedule a consultation with the Protovate team and let's talk through what comes next.

SCHEDULE A CONSULTATION