Compute Considerations

GPU Cloud Rental Costs & Scaling Experiments for LLM Fine-Tuning

What a Mac Studio Gives You

A Mac Studio M4 Max (128 GB) or M3 Ultra (192 GB) costs $4,000–$8,000 upfront and runs 24/7 for free after that. Its main advantage for LLM work is **unified memory** — the GPU and CPU share the same RAM pool, so a 70B model fits in memory without any special setup.

For fine-tuning specifically, Apple Silicon is slower than an NVIDIA H100 but has no per-hour cost.

Cloud GPU Pricing (Feb 2026)

GPU	Provider	$/hour	VRAM	Notes
A100 40 GB	Vast.ai	~$0.80–1.00	40 GB	Cheapest option, older hardware
A100 80 GB	RunPod	~$1.50–1.80	80 GB	Enough for 70B 4-bit
H100 80 GB	Vast.ai	~$1.87	80 GB	Marketplace rate
H100 80 GB	RunPod	~$1.99	80 GB	Community cloud
H100 80 GB	Lambda	$2.99	80 GB	More reliable, reserved
H100 80 GB	AWS / GCP	~$3–4	80 GB	Enterprise, on-demand

Day-Equivalent Cost Comparison

For a sleeping LLM, you don't need the GPU running 24/7 — only during sleep cycles:

Usage Pattern	Hours/day on GPU	Cost/day (H100 @ $2)	Cost/month
Sleep only (10 cycles × 10 min)	~1.5 hrs	$3	~$90
Sleep only (heavy, 20 cycles)	~3 hrs	$6	~$180
Always on (inference + training)	24 hrs	$48	~$1,440

The Smart Approach

You don't need to rent a GPU all day. The architecture already separates wake and sleep:

1. Wake (chat) — run inference locally on your MacBook Air, or use a cheap API
2. Sleep (training) — spin up a cloud GPU, upload conversation logs, run LoRA fine-tuning, download fused weights,

shut down

At ~$3/day for training-only use, that's ~$90/month — and you'd be training a 70B model that would be dramatically better at memory formation than the 3B you're running now.

Break-Even: Rent vs. Buy

	Mac Studio M4 Max 128 GB	Cloud H100 (sleep-only)
Upfront	~$5,000	$0
Monthly	$0 (electricity ~$10)	~$90
Break-even	—	~55 months
Model size	Up to 70B 4-bit	Up to 70B 4-bit
Training speed	Slower (memory bandwidth)	~3–5× faster
Privacy	Full	Data leaves your machine
Availability	Always	Subject to provider uptime

If you're experimenting for a few months, renting is cheaper. If this becomes a long-term project, the Mac Studio pays for itself in under 5 months and you keep full privacy and control.

How It Works

The workflow between your local machine and a rented GPU:

Step	Your MacBook	Rented GPU (e.g. RunPod)
1	You have fused weights from last sleep cycle (~4–14 GB for 70B 4-bit)	Empty machine with an H100 GPU
2	Upload weights (scp, rsync, or S3) →	Weights now on their disk
3	Upload conversation logs →	Training data on their disk
4	SSH in, run your code →	LoRA fine-tune runs on H100 Your weights, your code
5	← Download fused weights	New weights with memories
6	Done	Machine wiped / returned

Cloud API vs. Rented GPU

	Cloud API (OpenAI, etc.)	Rented GPU (Lambda, RunPod, etc.)
Whose model?	Theirs	Yours
Whose weights?	Theirs (you never see them)	Yours (you upload/download them)

	Cloud API (OpenAI, etc.)	Rented GPU (Lambda, RunPod, etc.)
Who runs training?	Their pipeline, their params	Your code, your params
What's on the machine?	Their infrastructure	A Linux box with a GPU — that's it
After you're done?	Model lives in their cloud	You take your weights home

It's no different from plugging an external GPU into your MacBook, conceptually. The weights live on your machine. You temporarily copy them to faster hardware for the training step, then bring them back.

For This Project Specifically

You'd change exactly one thing — the `train_lora()` method in `mlx_backend.py` would need to either:

Option A: Run training remotely via SSH

```
# Instead of local mlx_lm.lora call:
subprocess.run(["ssh", "gpu-box", "python", "-m", "mlx_lm.lora", ...])
```

Option B: Run the whole sleep cycle on the remote machine
Sync data/conversations/ up → Run sleep cycle remotely → Sync models/current/ back down

Everything else — the wake phase, conversation logging, curation — stays local. The GPU is only borrowed for the 5–10 minutes of actual gradient computation.

Running the Scaling Experiment with Offsite Compute

The main complication: MLX only runs on Apple Silicon. A rented NVIDIA GPU needs a PyTorch backend.

What Needs to Change

Your Codebase (works as-is)	Cloud GPU (needs new backend)
src/backend/mlx_backend.py	src/backend/torch_backend.py
mlx_lm.generate	transformers.generate
mlx_lm.lora (CLI)	peft LoRA + trl trainer
mlx fuse	merge_and_unload()

Everything else stays identical: orchestrator.py, curator.py, trainer.py, validator.py, replay.py, etc. The architecture was already designed with a single backend abstraction. You write one new file — `torch_backend.py` — that implements the same interface as `mlx_backend.py`, and the entire system works on NVIDIA hardware.

Step by Step

1. Rent a GPU

RunPod or Vast.ai, cheapest option that fits:

Model	Min VRAM	GPU to Rent	~Cost/hr
Llama 3.1 8B 4-bit	6 GB	A100 40 GB	~$0.80
Llama 3.1 70B 4-bit	40 GB	A100 80 GB	~$1.50
Llama 3.1 70B 16-bit	140 GB	2×H100	~$4.00

For the scaling experiment, an A100 80 GB at ~$1.50/hr running a 70B 4-bit model is the sweet spot.

2. Write torch_backend.py

Same interface as your MLX backend, using HuggingFace + PEFT:

```
# src/backend/torch_backend.py
from transformers import AutoModelForCausalLM, AutoTokenizer, BitsAndBytesConfig
from peft import LoraConfig, get_peft_model, PeftModel
from trl import SFTTrainer, SFTConfig

class TorchBackend:
    def load(self):
        quantization_config = BitsAndBytesConfig(load_in_4bit=True)
        self.model = AutoModelForCausalLM.from_pretrained(
            self.model_path, quantization_config=quantization_config)
        self.tokenizer = AutoTokenizer.from_pretrained(self.model_path)

    def generate(self, messages, max_tokens):
        inputs = self.tokenizer.apply_chat_template(messages, return_tensors="pt")
        outputs = self.model.generate(inputs, max_new_tokens=max_tokens)
        return self.tokenizer.decode(outputs[0])

    def train_lora(self, train_file, adapter_path, config):
        lora_config = LoraConfig(
            r=config["rank"], lora_alpha=config["alpha"],
            target_modules=["q_proj","v_proj","k_proj","o_proj"],
            lora_dropout=0.05)
        trainer = SFTTrainer(
            model=self.model, train_dataset=dataset,
            peft_config=lora_config,
            args=SFTConfig(
                num_train_epochs=config["epochs"],
                learning_rate=config["learning_rate"],
                per_device_train_batch_size=1,
                output_dir=adapter_path))
        trainer.train()

    def fuse_adapter(self, adapter_path, output_path):
        model = PeftModel.from_pretrained(self.model, adapter_path)
        merged = model.merge_and_unload()
        merged.save_pretrained(output_path)
```

3. Add a config switch

```
# config.yaml
backend: "torch"  # or "mlx"
model:
  path: "meta-llama/Llama-3.1-70B-Instruct"
```

4. The experiment script

```
# On the rented GPU:

# Setup (~5 min)
git clone your-repo
pip install torch transformers peft trl bitsandbytes
cd j

# Upload your existing conversation data
scp -r data/conversations/ gpu-box:~/j/data/conversations/

# Run the experiment
python -m src.main --backend torch --model meta-llama/Llama-3.1-70B-Instruct
```

Chat, sleep, test recall — same workflow as on your MacBook, just with a 70B model.

5. The scaling experiment itself

Run the same conversations on multiple model sizes and measure memory formation:

Model	Size	Sleep Cycles	Facts Recalled	Cost
Llama 3.2 3B	4-bit	1–2	baseline (current data)	free (local)
Llama 3.1 8B	4-bit	1–2	?	~$1–2
Llama 3.1 70B	4-bit	1–2	?	~$3–5

Total experiment cost: under $10 for a few hours of GPU time.

What You'd Measure

The paper's hypothesis to test: **larger models form memories more reliably** with the same training protocol.

Concrete metrics per model size:

- Facts retained after 1 sleep cycle (out of N told)
- Facts retained after 2 sleep cycles
- Hallucination rate (wrong details invented)
- Benchmark score preservation (catastrophic forgetting resistance)
- Whether the learning rate sweet spot widens with scale (3B had a razor-thin window of 1e-4)

Practical Timeline

Task	Time	Cost
Write torch_backend.py	~2 hours (your time)	$0
Rent GPU + setup	30 min	~$1
Run 8B experiment	1–2 hours	~$2
Run 70B experiment	2–3 hours	~$5
Total	One afternoon	~$8

That's an **$8 experiment** that directly tests the core scaling hypothesis of the paper. Hard to beat that cost-to-insight ratio.