How to Run Karpathy's Autoresearch on Your Local GPU

March 12, 2026 · by Mark Bartlett

Download this guide as PDF

> **Quick Answer:** Autoresearch is Karpathy's 630-line framework that lets an AI coding agent (Claude Code, Codex, etc.) run autonomous ML experiments on a single GPU while you sleep. Each experiment takes 5 minutes. The agent edits train.py, runs training, checks if validation improved, keeps or reverts via git, and repeats forever. Karpathy ran 700 experiments over 2 days and found 20 real improvements (11% efficiency gain). Shopify's CEO ran 37 experiments overnight and got a 0.8B model that beat his 1.6B. Works on RTX 3090/4090 out of the box. For 12GB cards, scale down vocab size and sequence length. For 6GB cards, people have gotten it working at 1.7GB peak VRAM. Mac users need a community fork (autoresearch-mlx for Apple Silicon).

Andrej Karpathy released autoresearch on March 6, 2026, and it hit 29,000 stars in under a week. The idea is simple and a little unsettling: point an AI coding agent at a training script, go to sleep, wake up to a model that's better than what you could have tuned by hand.

630 lines of Python. Single GPU. No distributed training, no complex configs. An agent edits `train.py`, runs a 5-minute experiment, checks if validation loss improved, commits or reverts via git, and does it again. Forever, until you stop it.

Karpathy left it running for 2 days. The agent ran roughly 700 experiments and found 20 real improvements that stacked together for an 11% efficiency gain on code he already considered well-tuned. Shopify CEO Tobi Lutke ran it overnight, 37 experiments, and ended up with a 0.8B model that outperformed his previous 1.6B. Smaller model. Better results. Found while sleeping.

Here's how to set it up on your GPU.

What autoresearch actually does

Three files matter:

File	Who edits it	What it does
`prepare.py`		Data prep, tokenizer, dataloader, eval harness. Fixed constants.

File	Who edits it	What it does
	Nobody (read-only)	
`train.py`	The AI agent	Model architecture, optimizer, training loop. This is what gets experimented on.
`program.md`	You	Your research strategy in plain English. This is where the human intelligence goes.

The loop runs like this:

1. Agent reads `program.md` for your research strategy
2. Agent modifies `train.py` with an experimental idea
3. `git commit` (snapshot before running)
4. Training runs for exactly 5 minutes wall clock
5. Agent checks `val_bpb` (validation bits per byte, lower is better)
6. Improved? Keep the commit, branch advances. Didn't improve? `git reset` back.
7. Log results to `results.tsv` (with commit hash, val_bpb, VRAM usage, description)
8. Go to step 2. Never stop. "The human might be asleep."

That's it. About 12 experiments per hour, roughly 100 overnight. The git history becomes your experiment log, and `results.tsv` tracks every attempt including crashes and reverts.

Prerequisites

- NVIDIA GPU (CUDA 12.8, no AMD ROCm or Intel support yet)
- Python 3.10+
- `uv` package manager (Karpathy's preferred tool for dependency management)
- An AI coding agent: Claude Code, Codex, Cursor, OpenClaw, or similar. The system is agent-agnostic since `program.md` is just a markdown file any agent can read.

```
# Clone and set up
git clone https://github.com/karpathy/autoresearch.git
cd autoresearch
uv sync

# Prepare data (downloads and tokenizes)
```

```
uv run prepare.py

# Verify training works
uv run train.py
```

The first `prepare.py` run downloads the dataset to `~/.cache/autoresearch/`. After that, you're ready.

Setup by GPU tier

RTX 3090 / 4090 (24GB): runs as-is

No changes needed. Clone, sync, run. The default config fits comfortably in 24GB VRAM. Start your coding agent in the repo directory:

```
Hi have a look at program.md and let's kick off a new experiment!
let's do the setup first.
```

The agent will create a branch (`autoresearch/<tag>`), read all three files, verify the data cache exists, initialize `results.tsv`, and start experimenting.

RTX 3060 12GB / 3070 / 4060 Ti (12-16GB)

You need to scale down. Karpathy gives seven specific recommendations in the README:

1. Switch to the TinyStories dataset (GPT-4 generated short stories, lower entropy, reasonable results with smaller models)
2. Lower `vocab_size` from 8192 to 4096, 2048, or 1024. Even 256 works for byte-level tokenization.
3. Reduce `MAX_SEQ_LEN` in `prepare.py` to 256 and increase `DEVICE_BATCH_SIZE` in `train.py` to compensate
4. Reduce `EVAL_TOKENS` so validation runs faster
5. Drop `DEPTH` from 8 to 4
6. Change `WINDOW_PATTERN` to just `"L"` instead of `"SSSL"` (banded attention is VRAM-hungry on smaller cards)
7. Lower `TOTAL_BATCH_SIZE` (keep powers of 2, e.g. 2^14 = 16K)

You'll get smaller models with shorter context, but the experiment loop works the same. The improvements you find at small scale often transfer to larger models anyway. That's one of the key findings from Karpathy's own run.

GTX 1660 Ti / 6GB cards

Someone in GitHub issue #87 got it running at 1.7GB peak VRAM on a GTX 1660 Ti. The trick: FlashAttention-3 doesn't support Turing GPUs, so you need to replace it with PyTorch's `scaled_dot_product_attention`. Their config:

- Sequence length: 2048 → 512
- Layers: down to 4
- Embedding dim: 256
- Device batch size: 128 → 8
- Disable `torch.compile()` to avoid compilation memory overhead

```
TORCH_DYNAMO_DISABLE=1 PYTORCH_CUDA_ALLOC_CONF=expandable_segments:True uv run train.py
```

50-100 experiments overnight on a 6GB card. Not bad.

Mac (Apple Silicon)

The main repo doesn't support MPS. Karpathy acknowledged this is doable but said he doesn't want to maintain it. Community forks fill the gap:

- autoresearch-mlx — native Apple Silicon MLX port, no PyTorch required
- autoresearch-macos — MPS support via PyTorch

If you're on a Mac with 16GB+ unified memory, the MLX fork is the better bet. MLX is purpose-built for Apple Silicon and avoids the MPS compatibility issues that plague PyTorch on Mac.

Writing a good program.md

This is where your judgment matters. The agent is the hands, but `program.md` is the brain. Karpathy's default `program.md` is 7,039 bytes of detailed instructions covering:

- What the agent can and can't modify (only `train.py`, no new packages)

- How to interpret results (lower val_bpb = better)
- When to keep or discard (improvement threshold)
- Simplicity preference: "A 0.001 val_bpb improvement that adds 20 lines of hacky code? Probably not worth it. A 0.001 val_bpb improvement from deleting code? Definitely keep."

You can customize `program.md` to focus the agent's experiments. Want it to explore learning rate schedules? Say so. Want it to try architectural changes only? Write that. The agent follows the strategy you lay out and executes it mechanically. It won't have the intuition to know which experiments are worth trying. That part is still on you.

What the agent actually finds

The kinds of improvements autoresearch discovers are specific and technical:

- Adding a scaler to parameterless QKnorm to sharpen attention distributions
- Applying regularization to value embeddings
- Widening banded attention windows
- Correcting AdamW beta parameters
- Tuning weight decay schedules and initialization schemes

These aren't random walks through hyperparameter space. The agent reads the code, forms hypotheses (guided by your `program.md`), and makes targeted edits. Some of them are things Karpathy said he'd missed after two decades of manual work.

The failure rate is high. In Lutke's run, 26 of 35 non-baseline experiments failed or showed no improvement. But the 7 that succeeded were enough to get a 0.8B model that beat his 1.6B, a 19% improvement in validation scores.

This matches the broader pattern: most experiments fail. The value is in running enough of them cheaply enough that the wins accumulate. A human researcher might try 5 ideas in a day. Autoresearch tries 100.

Honest limitations

NVIDIA only. No AMD ROCm, no Intel Arc, no CPU. FlashAttention-3 and CUDA 12.8 wheels lock this to NVIDIA for now. Mac users need community forks.

Results are hardware-specific. Your 5-minute experiments on an RTX 3060 aren't comparable to someone else's on an H100. Faster GPUs explore more of the model's capacity per experiment because the wall clock is fixed.

This optimizes training, not inference. Autoresearch makes your training run more efficient. It won't make your model respond faster when you're chatting with it. Different problem.

Most experiments fail. Karpathy's 700 experiments yielded 20 keepers (2.9% hit rate). Lutke's yielded 7 out of 35 (20%). This is normal for ML research. The agent just makes failure cheap enough that the wins accumulate.

You need a coding agent, and it needs to be good. Autoresearch doesn't include one. You bring your own: Claude Code, Codex, Cursor, OpenClaw, Aider. The agent is writing real modifications to a training loop, which is harder than autocompleting a React component. Smaller local coding models might struggle here. A frontier model (Claude, GPT-4.x) will do better.

Autoresearch doesn't remove the need to think. It shifts the bottleneck from "running experiments" to "designing good research strategies." A bad `program.md` produces 100 useless experiments just as easily as a good one produces 20 improvements.

Getting started tonight

If you have an NVIDIA GPU with 12GB+ VRAM:

```
git clone https://github.com/karpathy/autoresearch.git
cd autoresearch
uv sync
uv run prepare.py
```

Open your coding agent in the repo. If you're on a 12GB card, modify the constants first (see the scaling section above). Then:

```
Read program.md and kick off a new experiment.
Set the tag to "overnight-1". Do setup first.
```

Go to sleep. Check `results.tsv` in the morning. The git log shows every successful improvement as a separate commit, with the experiment description in the commit message.

For a guide on choosing the right GPU for local AI work, or if you want to understand VRAM requirements before committing your card to an overnight run, those guides have the specifics.

Related reading

- Best local coding models — which models are good enough to be the coding agent driving autoresearch
- VRAM requirements for local LLMs — make sure your overnight autoresearch run doesn't compete for VRAM with a running Ollama instance
- GPU buying guide for local AI — if autoresearch convinces you to upgrade
- Replace GitHub Copilot with local LLMs — the coding agent side of the equation

Get notified when we publish new guides.

Subscribe — free, no spam