epsiclaw: OpenClaw Stripped to 515 Lines of Python (The Karpathy Treatment)

March 30, 2026 · by Mark Bartlett

Download this guide as PDF

> **Quick Answer:** epsiclaw is a 515-line Python reimplementation of OpenClaw's core agent loop -- 6 files, one dependency (httpx), MIT licensed. It's not a product replacement. It's an answer to the question 'what does a personal AI assistant actually do under the hood?' Clone it, read it in an afternoon, and you'll understand agent architecture better than any tutorial could teach you.

📚 **More on this topic:** OpenClaw Setup Guide · Best Models for OpenClaw · OpenClaw vs Cursor · OpenClaw on Low-VRAM GPUs

OpenClaw has 335,000 GitHub stars and roughly 400,000-500,000 lines of TypeScript. It surpassed React's 10-year star record in 60 days. Most people using it have no idea how it actually works underneath. The codebase is too large to read, and the docs describe what it does, not how.

Dor Ringel, an engineer at JFrog, decided to fix that. He took the same approach Karpathy used with nanoGPT, micrograd, and autoresearch: strip the system down to its algorithmic core, throw away everything that isn't the core idea, and publish what's left. The result is epsiclaw – epsilon (ε) + claw – 515 lines of Python across 6 files with a single dependency. You can read the entire thing in an afternoon and understand exactly what a personal AI assistant does.

What's in the 515 lines

Every file has a clear job. No abstractions hiding behind abstractions.

File	Lines	What it does
`agent.py`	146	The ReAct loop: receive message, call LLM, execute tools, loop until the LLM responds with text
`tools.py`	155	Tool registry with decorator-based auto-schema generation. 7 built-in tools
`memory.py`	65	Builds the system prompt from three markdown files. Persists session history as JSONL

File	Lines	What it does
`cron.py`	63	JSON-file job scheduler. Fires synthetic messages when jobs are due
`channel.py`	53	Telegram long-polling. Sends and receives messages
`llm.py`	33	One function: POST to any OpenAI-compatible chat completions endpoint

One dependency: httpx. That's it. No LangChain, no vector database, no embeddings library.

The agent loop in `agent.py` is the heart. It runs a ReAct cycle capped at 10 iterations per turn: build context from memory files, call the LLM, check if the response contains tool calls. If yes, execute the tool and feed the result back. If no, send the text response to the user. That's the entire agent architecture.

The 7 built-in tools: `get_current_time`, `web_search` (via Tavily API), `memory_read`, `memory_write`, `cron_add`, `cron_list`, `cron_remove`. The tool registry uses a Python decorator that auto-generates OpenAI function-calling schemas from type hints and docstrings. Adding a new tool is one decorated function.

Memory is three markdown files: `SOUL.md` (the agent's persona), `USER.md` (static info about the user), and `MEMORY.md` (dynamic memory the LLM reads and writes at runtime). Session history is capped at 50 messages per chat. No embeddings, no RAG, no vector search. Just files the LLM can read and overwrite.

What got thrown away

Here's what full OpenClaw has that epsiclaw doesn't:

Feature	OpenClaw	epsiclaw
Lines of code	~400K-500K TypeScript	515 Python
Channel integrations	50+ (Slack, Discord, web, etc.)	1 (Telegram)
Plugin marketplace	Yes	No
Memory system	Vector DB with embeddings	3 markdown files
Tool sandboxing	Isolated execution	Runs with your permissions
Multi-tenant auth	Yes	Single user
Provider failover	Multiple providers with retry	1 endpoint

Feature	OpenClaw	epsiclaw
Config files	53+	4 environment variables

Ringel's distinction is worth quoting: "The algorithm is always small. The systems around it are large because they solve real problems – auth, sandboxing, multi-tenancy – but those problems are distinct from the algorithm itself."

That's the point. epsiclaw isn't a lightweight OpenClaw alternative for production use. NanoClaw (~3,900 lines) and NanoBot (~4,000 lines) fill that role. epsiclaw exists to answer a question: what is the core algorithm behind a personal AI assistant? The answer turns out to be a ReAct loop, a tool registry, and a file-based memory system. Everything else is infrastructure.

The Karpathy pattern

Ringel explicitly positions epsiclaw in a lineage of minimal reimplementations:

Project	Question it answers	Lines
micrograd	What is backpropagation?	94
nanoGPT	What is GPT training?	~300
minbpe	What is tokenization?	~300
llm.c	What is LLM training in C?	~1,000
autoresearch	What is autonomous ML research?	630
epsiclaw	**What is a personal AI assistant?**	**515**

The pattern is always the same. Take a system that's grown complex through real-world requirements, strip it to the algorithm, and publish readable code. People learn more from 500 lines they can hold in their head than from 500,000 lines they can't navigate.

Karpathy's autoresearch, published earlier this month, followed this exactly – 630 lines of Python that let an AI agent run autonomous ML experiments on a single GPU. It found 20 improvements over hand-tuned code and produced an 11% training speedup. The code is readable enough that the community forked it for CPU-only machines, Apple Silicon, and distributed setups within days.

epsiclaw does the same thing for agent architecture. If autoresearch answers "what is autonomous ML research?", epsiclaw answers "what is a personal AI assistant?"

Running it with a local model

epsiclaw connects to any OpenAI-compatible API, which means it works with local inference servers out of the box.

```
git clone https://github.com/dorringel/epsiclaw.git
cd epsiclaw
pip install httpx
cp .env.example .env
```

Edit `.env` to point at your local server:

```
LLM_API_URL=http://localhost:11434/v1/chat/completions  # Ollama
LLM_MODEL=qwen3.5:9b
TELEGRAM_BOT_TOKEN=your_token_from_botfather
```

Then run:

```
python agent.py
```

The model needs to support function/tool calling through the OpenAI API format. For local models, your best options:

Model	VRAM needed	Why it works
Qwen 3.5 9B Q4_K_M	~6GB	Good tool-calling support, fits on 8GB GPUs
Qwen 3.5 27B Q4_K_M	~17GB	Stronger reasoning for complex agent tasks
DeepSeek R1 32B Q4_K_M	~19GB	Strong reasoning, fits on 24GB

You'll also need a Telegram bot token from @BotFather (free, takes 2 minutes). Web search is optional – set a Tavily API key if you want it, or skip it.

Who this is for

Read epsiclaw if you want to understand how agent systems work. The entire architecture fits in your head. You can trace every step from user message to LLM call to tool execution to response. Try doing that with OpenClaw's codebase.

Fork epsiclaw if you want a minimal base to build your own agent. Adding a new tool is one decorated function. Swapping Telegram for another channel is rewriting 53 lines. The code is MIT licensed and the structure invites modification.

Skip epsiclaw if you need a production-ready assistant. No sandboxing, no auth, no multi-channel support, no error recovery beyond basic retry. For that, use OpenClaw or one of the lightweight alternatives.

The repo is 5 days old and has 34 stars. It'll grow, but the value isn't in the community around it – it's in the 515 lines of code that show you exactly what the most-starred project on GitHub is doing under the hood.

Related guides

- OpenClaw Setup Guide
- Best Models for OpenClaw
- Best OpenClaw Alternatives
- OpenClaw on Low-VRAM GPUs
- Qwen 3.5 9B Setup Guide

Get notified when we publish new guides.

Subscribe — free, no spam
