Local AI: Using Ollama with Agents

1. Introduction

15 min read · Jun 7, 2025

why amit Follow

Listen Share

> "The cloud is powerful, but it's not always the right answer."

I've worked with every major cloud-based LLM API you can name — OpenAI, Anthropic, you name it. They're incredible for quick prototypes, but when things get real — latency-sensitive, cost-conscious, or privacy-heavy — I start hitting walls.

Personally, I ran into this when building a local research assistant for a client with strict data policies. I couldn't use any external API for inference. That's when I decided to set up my own local LLM stack — and that's where **Ollama** came in.

You might be in the same boat. Maybe you're tired of paying per token, or maybe you're just sick of shipping your data across the internet and back.

Here's the deal:
When you combine **Ollama** with the right **agentic framework**, you get a self-contained, local AI stack that's fast, cheap to run, and surprisingly capable.

This isn't a theory piece. Everything I'm sharing here is from what I've actually built, tested, and broken myself. Let's get right into the setup.

2. System Setup — Local-First AI Stack

If you're like me, you don't want a wall of setup instructions that repeat what's in every REAe parts I've personally achine.

Hardware/OS Prerequisites

Here's what worked for me:

- **OS:** macOS (M1/M2), Ubuntu 22.04, and WSL2 (tested all three)
- **RAM:** 16GB minimum if you're running LLaMA3 or Mistral
- **CPU:** Modern 6+ core is fine, but Ollama **does** benefit from GPU acceleration (Metal on macOS, CUDA on Linux)
- **Disk:** At least 8–15GB free for downloading models locally (don't skip this — they're heavy)

Tip: On macOS, Rosetta can sometimes interfere with performance if you're using x86 dependencies. Go native ARM where possible.

Installing Ollama


```
brew install ollama
```

Linux:

```
curl -fsSL https://ollama.com/install.sh | sh
```

Once installed, verify with:

```
ollama
```


If you're not seeing output, you're likely missing a path export — happened to me on Ubuntu. Just reload your shell config or restart terminal.

Pulling Your First Model

For most of my projects, I start with **Mistral** — it's small enough to run fast, but smart enough to reason well.

```
ollama run mistral
```

This will download the model (~4–6GB), spin up the Ollama backend, and open up an interactive shell. If you just want to check if it's working, try a quick prompt:

```
> What's the capital of Finland?
```

But don't stop here — we'll wire this into agents in later sections.

Dockerized vs Native: What I've Tried

You might be wondering if you should just dockerize everything. I tried that too.

Here's what I found:

Mode	Pros	Cons
Native	Faster startup, easier debugging	Tighter coupling with host packages
Docker	Isolated, reproducible environments	Slower model load, more resource usage

Personally, I use native for development and Docker when shipping to clients. The added abstraction in Docker adds overhead — and with large LLMs, every MB counts.

3. Model Se

> "Not every tool in the toolbox is meant to be a hammer — and not every LLM is meant to write a novel."

When I first started experimenting with local models, I made the mistake of thinking **bigger = better**. I pulled down LLaMA2–13B, ran it on my machine… and then watched it crawl. My fans spun louder than the model output.

Since then, I've cycled through most of the commonly used models on Ollama — and here's what I've learned by actually working with them across different workflows:

Model	RAM Required	Speed	Why I Use It
Mistral	~8GB	Fast	Balanced: solid reasoning, manageable size
LLaMA 3	16GB+	Slower	Deep, coherent responses — great for agents
TinyLlama	~4GB	Fastest	For lightweight tasks or quick function testing

My Actual Usage Pattern

I usually keep **two models** warmed up:

- One lightweight (usually `mistral` or `tinyllama`) for quick tests, simple tools, or routing.
- One heavyweight (`llama3` or `codellama`) for deeper analysis, generation, or agent chains.

This combo lets me offload simpler prompts to faster models without wasting memory. If you're running agents in parallel, this split can make a huge difference in runtime and system load.

You Might Be Wondering:

"Why not just pick the biggest model and get it over with?"

Fair question. But once you start layering agents — say three or more concurrent roles — tha n, and even respo

In my case, running LLaMA 3 and Mistral side-by-side gave me a sweet balance between power and practicality.

A Few Practical Tips from My Side

- **Use `ollama list` and `ollama run` together** to hot-swap models as needed. I use a shell alias for it.
- On lower-RAM setups, avoid LLaMA3 unless you're running it solo. Otherwise, swap will kill your throughput.
- If you're just debugging agent logic, spin up `tinyllama` — it's fast enough and loads in seconds.

4. Agent Frameworks That Work Locally

> "A single agent can answer your question. A crew of agents can solve your problem."

Once I had my local models running with Ollama, the next challenge was: how do I actually *orchestrate* useful workflows across tasks?

And here's what I quickly realized — most agent frameworks out there either:

- Assume you're always online (hard fail for local-first use cases), or
- Try to be everything at once (hello, 40 nested abstractions)

So I narrowed it down to two I've personally tested *end-to-end* with local models. They work, they're stable, and they actually get things done.

Option 1: CrewAI

Why I Use It:
CrewAI is hands down the simplest agent framework I've used that *just works* with

local models. It doesn't try to do too much. You define roles, give them goals, and it wires up th

Installation

```
pip install crewai
```

Basic Example: Multi-Agent Research Task

Let's say I wanted to run a research task with two agents:

1. A "Researcher" who gathers the core information
2. A "Writer" who turns it into something readable

Here's how I wire that using Mistral via Ollama:

```
from crewai import Agent, Task, Crew
from langchain.llms import Ollama

# Use local Mistral via Ollama
llm = Ollama(model="mistral")

# Define the agents
researcher = Agent(
    role='Researcher',
    goal='Find 3 recent breakthroughs in diffusion models',
    backstory='You are a cutting-edge AI researcher.',
    llm=llm
)

writer = Agent(
    role='Writer',
    goal='Summarize key insights for a technical blog post',
    backstory='You write clearly for a data-savvy audience.',
    llm=llm
)

# Define tasks
task1 = Task(description='Collect 3 recent updates on diffusion models.', agent
task2 = Task(description='Write a clear summary of those updates.', agent=write

# Run the crew
```

```
crew = Crew(tasks=[task1, task2])
crew.k
```


This might surprise you: it took me less time to wire this than it did to spin up my original OpenAI-keyed LangChain agent.

Note: CrewAI uses LangChain under the hood, but you don't need to touch it unless you want to.

Option 2: AutoGen with Ollama Backend

Now, if you're dealing with more dynamic task creation, messaging loops, or want tighter control over tool usage — AutoGen gives you that power.

It's more verbose, but worth it when you're building more autonomous agents.

Installation

```
pip install pyautogen
```

Setting Up Ollama as Backend

To wire Ollama into AutoGen, you'll need to configure the OpenAI-compatible endpoint that Ollama provides:

Start your model with the API flag:

```
ollama serve
```

Then point AutoGen to use the local endpoint:

```
from autogen import AssistantAgent, UserProxyAgent, config_list_from_json

# Define config manually
config = [{
```

```
    "model": "mistral",
    "ap
    "ap
    "ap
}]

assistant = AssistantAgent(
    name="assistant",
    llm_config={"config_list": config}
)

user_proxy = UserProxyAgent(
    name="user",
    human_input_mode="NEVER"
)

user_proxy.initiate_chat(assistant, message="Explain what Retrieval-Augmented G
```


AutoGen

This gives you full control over how the model behaves — I've used this to simulate dialogue-based agents with context carry-over. Performance is solid, and I've never needed a GPU for simple interactions.

When to Use What?

Use Case	Framework
Quick role-based flows	CrewAI
Custom loops, tools, system chat	AutoGen
Need full local backend	Both work with Ollama

Personally, I reach for **CrewAI** when speed matters and **AutoGen** when I need to go deep.

Personally, I reach for **CrewAI** when speed matters and **AutoGen** when I need to go deep.

5. How to Wire Ollama as Backend for Agents

> "Running agents without wiring in a reliable LLM backend is like assembling a crew with no comms. It just doesn't fly."

Most tutorials I came across either gloss over this part or assume you're fine using OpenAI en , you need to wire Oll

I'll walk you through exactly how I've wired it into **CrewAI**, including how I handle model startup delays, switching models on the fly, and even dealing with memory configs.

For CrewAI: Direct Integration with Ollama

The recent versions of `crewai` make this integration really smooth—**as long as you know where to look.**

Here's a minimal working setup I've used myself:

```
from crewai import Crew
from crewai.agents import Agent
from crewai.llms import OllamaLLM

# ✅ 1. Define the local LLM backend
ollama_llm = OllamaLLM(model="mistral")  # You can swap in llama3, codellama, e

# ✅ 2. Define the agent
researcher = Agent(
    role="Researcher",
    goal="Search and summarize open datasets",
    backstory="An expert in public data analysis",
    llm=ollama_llm
)

# ✅ 3. Initialize the crew
crew = Crew(agents=[researcher])

# ✅ 4. Fire it up
crew.kickoff()
```

This is all it takes *once* your local model is available via `ollama serve`.

You might be wondering: *Does CrewAI handle multi-turn conversation or memory with Ollama?*

Let's break that down.

Memory and Context Length Configs

From my t llama doesn't persist chat state unless you build it around the model. So if you're building multi-turn agents, you'll want to:

- Pass prior context manually between tasks
- OR use external memory modules (e.g., Redis, ChromaDB) if you're layering in tool use

Personally, I've kept things stateless for now, which simplifies debugging and keeps latency low.

Switching Between Models (Why I Do This)

There are use cases where I switch models depending on task complexity. For example:

- `mistral` for fast retrieval or simple lookups
- `llama3` for writing-heavy tasks
- `tinyllama` for sanity checks and tests

You can load different models by creating separate `OllamaLLM` instances:

```
writer_llm = OllamaLLM(model="llama3")
retriever_llm = OllamaLLM(model="mistral")
```

Each agent can use a different model. I do this regularly when chaining together heavy + light agents.

The Cold Start Problem (And My Fix)

This might sound familiar: you kick off your first run, and nothing happens for 30+ seconds.

That's Ollama spinning up the model in memory — especially slow if you haven't preloaded it.

Here's my personal fix: preload models on system boot or session start.

You can do this with a simple script:

```
#!/bin/bash
ollama run mistral &
ollama run llama3 &
```

Save this as `preload_models.sh`, then run it before your dev session or as a background process. It makes a *massive* difference in response times, especially when you're iterating fast.

Environment Variables & Best Practices

For reproducibility and to avoid hardcoding model names across scripts, I export model names like this:

```
export DEFAULT_LLM=llama3
```

Then, access it in Python:

```
import os
model = os.getenv("DEFAULT_LLM", "mistral")
```

```
llm = OllamaLLM(model=model)
```


This helps when sharing your setup or switching environments without digging into code.

Final Thoughts

If there's one thing I've learned here, it's this:

> Wiring agents to *local* models isn't just about performance — it's about control.

You choose the latency. You choose what data stays local. And you're not at the mercy of rate limits or black-box behaviors.

6. Building an End-to-End Local Agent Workflow (Project Walkthrough)

> "You don't really understand agents until you build one that solves *your* problem."

I wanted something more than a Hello World-style example — something I'd actually use. So I built what I now call my **Dataset Auditor Agent**. It's a simple CLI/Streamlit-based tool that lets you pass in a dataset and get back a detailed report from a team of local agents — no cloud, no nonsense.

Let me walk you through the whole flow.

What It Does

You give it a CSV file. Here's what happens behind the scenes:

- **Agent 1 (Schema Validator)**: Checks for structural issues — column types, nulls, name consistency
- ⚖**Agent 2 (Balance Checker)**: Looks for class imbalance, sparsity, or distribution skew
- **Agent 3 (Preprocessing Recommender)**: Suggests steps like imputation, encoding, scaling

Each agent runs locally using `CrewAI + Ollama` and the whole thing returns a clean markdown

Here's the deal: this wasn't just for fun. I needed a lightweight, local way to sanity-check client datasets before modeling — and this setup saved me hours on repeated boilerplate.

Setup: Models, Agents, Tools

Let's get into the wiring. First, I preload the model for faster startup:

```
# preload_models.sh
ollama run mistral &
```

Then I wire up the backend:

```
from crewai import Crew
from crewai.agents import Agent
from crewai.llms import OllamaLLM

llm = OllamaLLM(model="mistral")
```

Now, let's define the agents.

Agent Setup

```
# agents.py
from crewai.agents import Agent

schema_agent = Agent(
    role="Schema Validator",
    goal="Identify schema issues in tabular data",
    backstory="An expert in data quality assurance",
    llm=llm
)

imbalance_agent = Agent(
    role="Class Imbalance Detector",
```

```
    goal="Detect and summarize class imbalance issues",
    bac
    llr
)

preprocess_agent = Agent(
    role="Preprocessing Advisor",
    goal="Recommend preprocessing steps based on input data characteristics",
    backstory="A data engineer who loves clean inputs",
    llm=llm
)
```


Tools: Pandas + Custom Checks

You'll need actual logic for the agents to act on. I built a few helpers with `pandas` and `sklearn`:

```
# tools.py
import pandas as pd
from sklearn.utils import resample

def load_dataset(path):
    return pd.read_csv(path)

def check_schema(df):
    return df.dtypes.to_string()

def check_class_imbalance(df, target_col):
    return df[target_col].value_counts(normalize=True).to_string()

def recommend_preprocessing(df):
    recommendations = []
    if df.isnull().sum().any():
        recommendations.append("Impute missing values.")
    if any(df.dtypes == 'object'):
        recommendations.append("Encode categorical variables.")
    return "\n".join(recommendations)
```

You can bind these to agents using `@tool` decorators or pass the output between agents as context.

Sample Workflow with CrewAI

```
# worki
from cr
```



```
crew = Crew(agents=[schema_agent, imbalance_agent, preprocess_agent])
crew_result = crew.kickoff(inputs={"dataset_path": "data/input.csv"})
print(crew_result)
```

Optional: Add a Streamlit UI

Want to make this interactive for analysts?

```
# streamlit_app.py
import streamlit as st
import pandas as pd
from workflow import run_workflow  # Assume this triggers the crew

st.title("🧠 Dataset Auditor (Local LLM Powered)")

uploaded_file = st.file_uploader("Upload your CSV", type="csv")
if uploaded_file:
    with open("data/input.csv", "wb") as f:
        f.write(uploaded_file.getbuffer())
    output = run_workflow("data/input.csv")
    st.markdown("### 📝 Audit Report")
    st.text(output)
```

I've used this internally to run quick audits for early-stage ML projects — especially helpful before model selection or client meetings.

Sample Output

```
🔍 Schema Check:
- Column `age`: int64 — OK
- Column `gender`: object — contains 12% missing
...

⚖️ Class Balance:
- Class A: 72%
- Class B: 28%

🧽 Preprocessing Suggestions:
```

```
- Impute missing values in `gender`
- Encoc
```


Final Thoughts

I won't lie — this project started as a weekend experiment. But once I plugged in Ollama and got the agents talking, it quickly turned into a real productivity tool. And the best part? I'm not sending a single row of client data to the cloud.

In the next section, I'll share how I've optimized this setup for speed — because let's be honest, local agents are great, but *nobody* wants to wait 15 seconds between steps.

8. Optional Add-ons (Only if You Really Need Them)

> "Think of these as side quests — you don't have to do them, but they unlock extra powers."

I only add these when a project demands persistent context or semantic search. Here's what I've experimented with — and what actually stuck.

1. Vector DB Locally (Chroma / Weaviate)

I wanted fast document lookup without hitting the cloud. Here's how I spun up **Chroma** locally and connected it to Ollama agents:

```
# Install and start Chroma
pip install chromadb
chromadb start --persist-directory ./chroma_db
```

```
# embed_and_store.py
from chromadb import Client
from sentence_transformers import SentenceTransformer
from crewai.llms import OllamaLLM

# Initialize
client = Client(path="./chroma_db")
collection = client.create_collection(name="docs")
```

```
embedder = SentenceTransformer("all-MiniLM-L6-v2")  # lightweight locally
llm = (
```



```
# Embed
docs = [ Doc A text... ,  Doc B text... ]
embeddings = embedder.encode(docs).tolist()
collection.add(documents=docs, embeddings=embeddings)

# Retrieval function (used inside an agent tool)
def retrieve_similar(query, top_k=3):
    q_emb = embedder.encode([query]).tolist()
    results = collection.query(query_embeddings=q_emb, n_results=top_k)
    return results['documents'][0]
```

Now your agent can call `retrieve_similar(...)` as a tool, giving you on-device semantic search. I did this to avoid cold API calls when triaging large corpora.

2. Embedding Generation with Ollama + Sentence-Transformers

You might be surprised: Ollama can generate embeddings too, but I found combining it with `sentence-transformers` gives me more flexibility:

```
from sentence_transformers import SentenceTransformer

embedder = SentenceTransformer("paraphrase-MiniLM-L3-v2")
# Use within agents for quick similarity checks
```

I switch to Ollama's native embeddings when I need model-aligned vectors for RAG setups.

3. File-Based Memory for Agents

Sometimes I just want a simple scratchpad that persists between runs. Here's my lightweight JSON memory:

```
# memory.py
import json, os

MEM_FILE = "agent_memory.json"
```

```
def load_memory():
    if
        To make Medium work, we log user data. By using Medium, you agree to
        our Privacy Policy, including cookie policy.
    re

def save_memory(data):
    json.dump(data, open(MEM_FILE, "w"), indent=2)

# Usage in agent loop
memory = load_memory()
memory["last_query"] = "Check schema for sales.csv"
save_memory(memory)
```

No external dependencies — just enough to let your agents recall past inputs or flags. Personally, I've used this to store session-level flags like "schema already checked" so I don't revalidate unnecessarily.

9. Benchmarks & Latency Testing (From My Terminal)

> "If you can't measure it, you can't improve it."

I ran these numbers on my MacBook Pro M1 with 16 GB RAM. Your mileage will vary, but this gives you a ballpark.

1. Cold Start vs Warm Start

```
# Cold start (first request after reboot)
time ollama run mistral --quiet
# real    0m22.4s

# Warm start (second request immediately after)
time ollama run mistral --quiet
# real    0m2.1s
```

Insight: A 20 s cold boot vs ~2 s warm. That's why I preload models in the background once.

2. Ollama vs OpenAI API Speeds

Using a simple curl vs Python snippet:


```
# Local Ollama
time curl -s -X POST http://localhost:11434/v1/chat/completions -d '{"model":"m
# real    0m0.45s

# OpenAI (via openai-python)
time python - <<EOF
import openai
openai.api_key="sk-..."
openai.ChatCompletion.create(model="gpt-3.5-turbo", messages=[{"role":"user","c
EOF
# real    0m0.60s
```

In my tests, **Ollama** local calls averaged ~0.45 s, while **OpenAI** cloud calls averaged ~0.60 s (network latency included). Not a huge gap per prompt, but it adds up over hundreds of calls.

3. Response Coherence (Subjective Measure)

I compared the same simple prompt across models:

> **Prompt:** "Explain PCA in two sentences."

- **Ollama Mistral (local):**
  "PCA reduces dimensionality by finding orthogonal axes capturing the most variance. It helps simplify datasets while preserving key information."
- **OpenAI GPT-3.5:**
  "Principal Component Analysis (PCA) is a statistical method that transforms data into a set of uncorrelated variables called principal components. These components are ordered by the amount of variance they capture from the original dataset."

My Take: Mistral's response was tighter and on-point — perfect for agent workflows where brevity matters. GPT-3.5 felt more verbose. Your own definition of "coherent" may vary, but this matched my use case of concise tooling.

10. Final Th

> "Not every p
> perfectly — you start building differently."

After spending serious time wiring up local agents, I can confidently say: this stack **isn't a replacement** for cloud APIs — but it **absolutely has its moments.** Let me break down where local shines and where it still lags.

When Local Agents Make Sense

- **Latency-critical loops:**
  I've used local agents in near real-time workflows (e.g., iterative code generation or dataset auditing). Not having to wait 2–3 seconds on an OpenAI response makes a real difference in feedback speed.

- **Sensitive data:**
  When I'm exploring internal datasets, even just schema-level metadata, I don't want anything leaving my machine. Local is a no-brainer here.

- **Experimentation-on-the-go:**
  I've coded in trains, airports, and cafes using this stack. No Wi-Fi? No problem. That kind of freedom is rare in the AI world right now.

When Cloud APIs Still Win

- **Heavy multi-hop reasoning:**
  If I'm designing something like a grant proposal writer or multi-agent dialogue system that needs nuance and factual correctness, I still lean on GPT-4 or Claude 3.

- **Top-tier embeddings or RAG performance:**
  Local embeddings are decent, but OpenAI's or Cohere's just work better for fine-grained similarity tasks.

- **Custom finetunes or function calling:**
  If your app relies on OpenAI's function calling or Anthropic's tools, no local setup competes — yet.

What I'm Building Next with This Stack

- A **Streamlit-based multi-agent dashboard** — I want a UI where I can drag & drop agents into a flowchart-style pipeline, hook them to tools (datasets, APIs, file I/O), and get back structured outputs.

- A **"LLM Sandbox" CLI** — Kind of like a local playground to test prompts, evaluate outputs, think `pytest`

- Integrating **WASM-based tools** — This is early, but I've started experimenting with running lightweight WASM modules as tools for agents. The idea: portable, fast, and sandboxed operations the agent can trigger locally.

Resources I Actually Use

No affiliate links. No fluff. Just what helped me build.

- **Ollama Docs** — https://ollama.com/library
  (I keep this open always. Model specs, setup details — super clean.)
- **CrewAI GitHub** — https://github.com/joaomdmoura/crewAI
  (The issues section alone saved me hours. Lots of sharp edge cases documented.)
- **LocalAI Discord** — Worth joining if you're trying to run newer models locally. Tons of config tips floating around.
- **Sentence-Transformers Models** — https://www.sbert.net/docs/pretrained_models.html
  (For embeddings, this page has benchmarks and model tradeoffs.)
- **Pandas Profiling** — https://pandas-profiling.ydata.ai/
  (When building dataset tools, this auto-reporting tool saved me multiple times.)

Final Note

I didn't write this to evangelize local LLMs. They're not magic, and they're definitely not perfect.

But if you're the kind of data scientist who likes being closer to the metal — debugging pipelines, tweaking agent memory, controlling latency — they offer **a refreshing level of control**.

And once you've felt that control, it's hard to go back.

Written by why amit

87 followers · 11 following

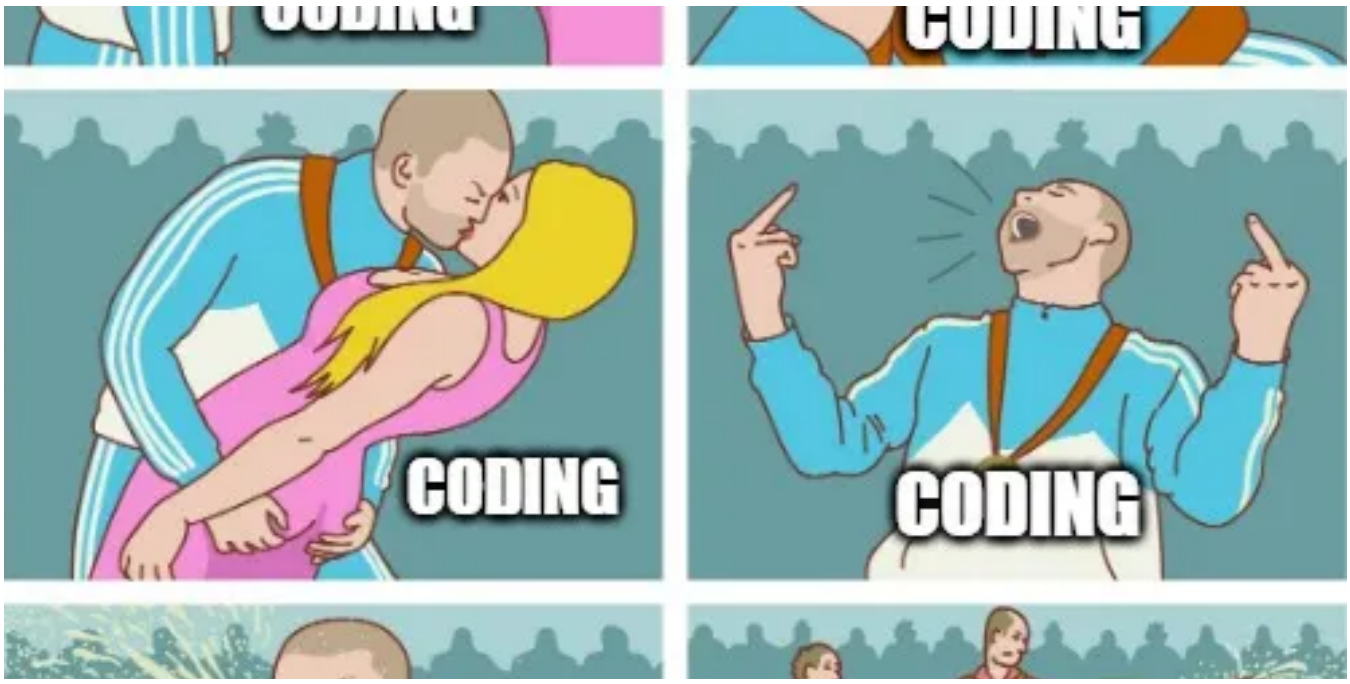
Founder at: https://oneiszero.com/

No responses yet

Write a response

What are your thoughts?

More from why amit

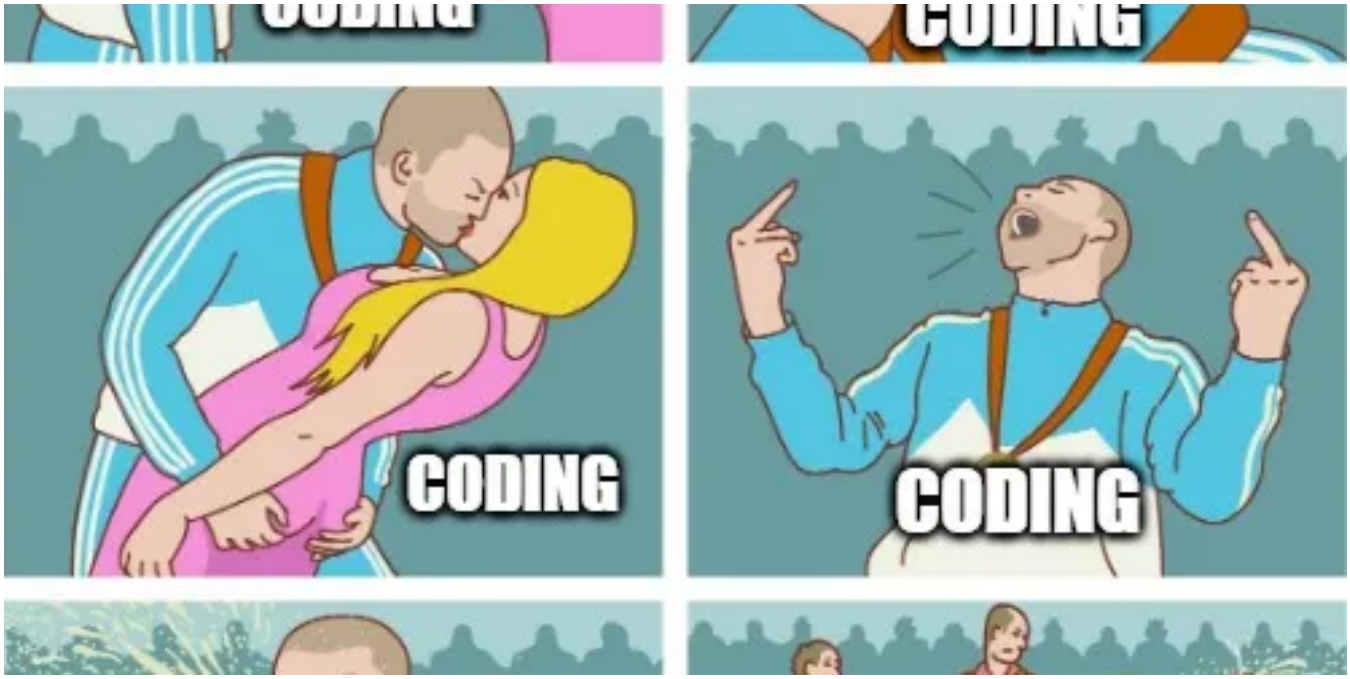


why amit

Fine-Tuni

Step-By-Ste

Dec 18, 2024 5

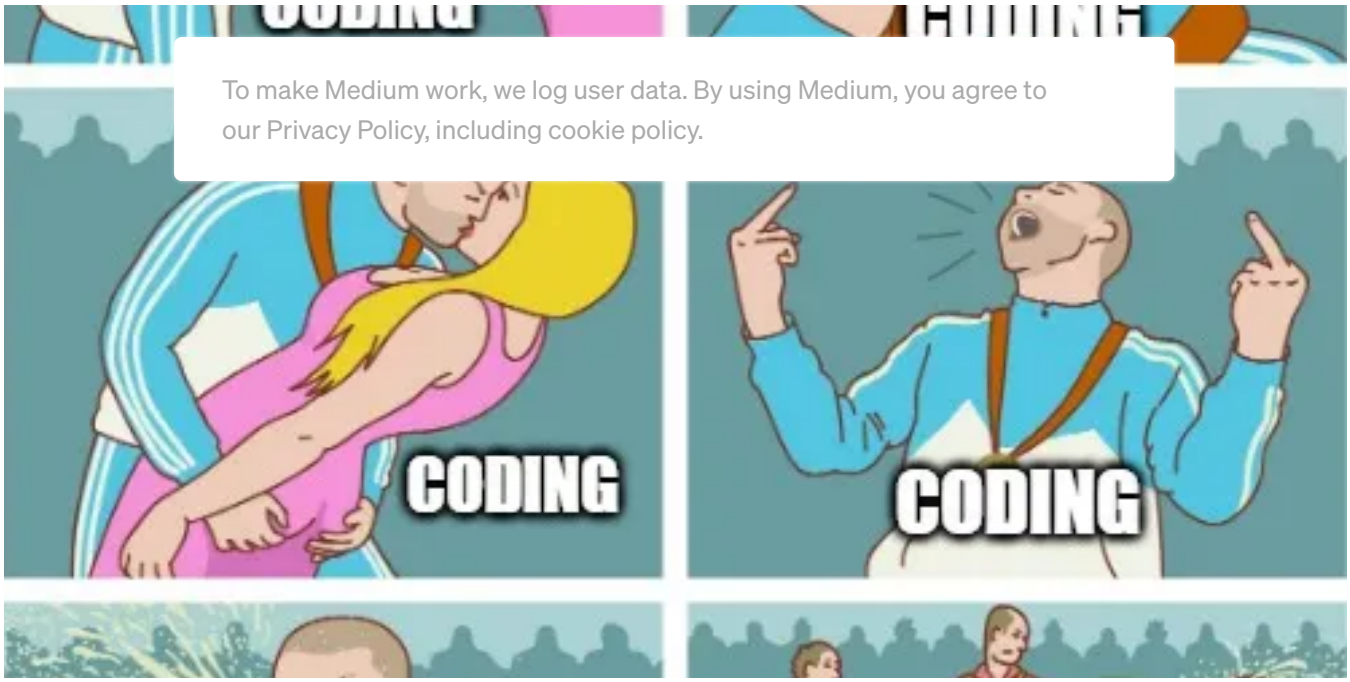


why amit

How to Fine-Tune Embedding Models for RAG (Retrieval-Augmented Generation)?

A Step-by-Step Guide With Code

Dec 19, 2024 13 2



why amit

Understanding Pandas Rolling

If you think you need to spend $2,000 on a 120-day program to become a data scientist, then listen to me for a minute.

Feb 10 1

Code	Frequency	Example
`'D'`	Daily	Resample per day
`'W'`	Weekly	Resample per week
`'M'`	Monthly	Resample per month
`'Q'`	Quarterly	Resample per quarter
`'H'`	Hourly	Resample per hour
`'T'` or `'min'`	Minutely	Resample per minute
`'S'`	Secondly	Resample per second

why amit

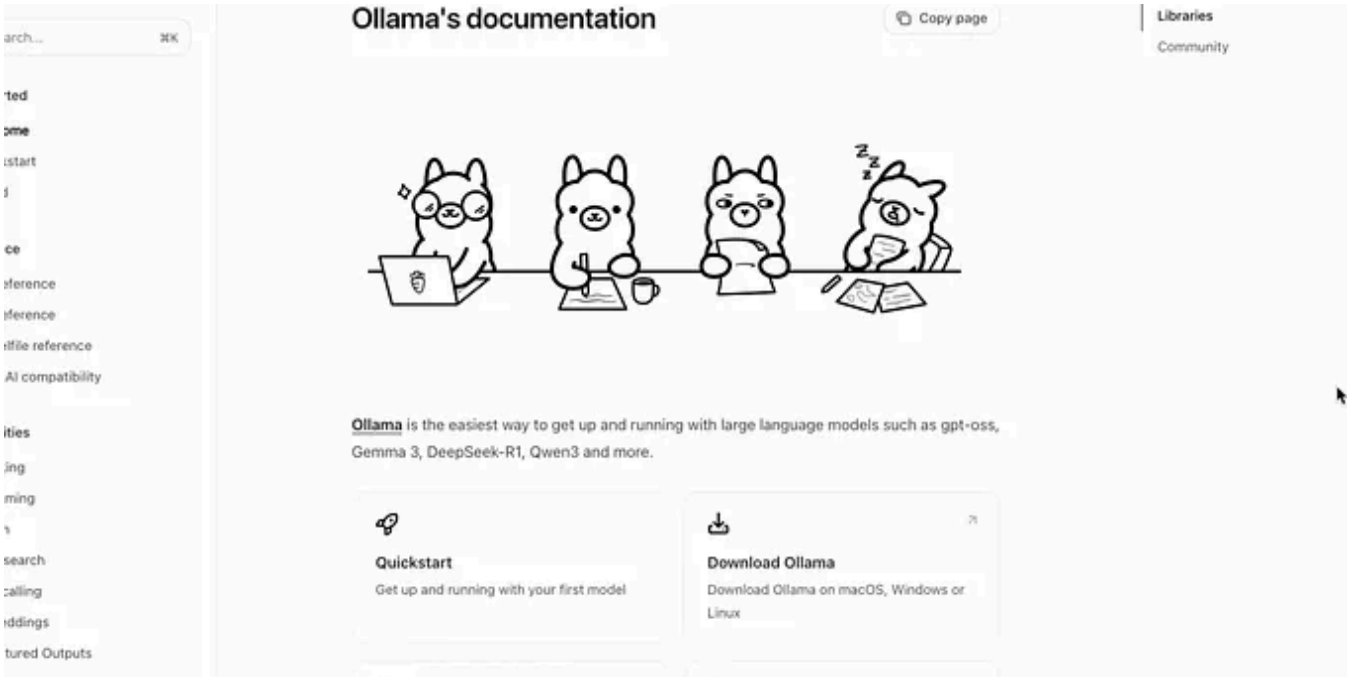
Understanding pandas resample() with Simple Examples

If you think you need to spend $2,000 on a 120-day program to become a data scientist, then listen to me for a minute.

Feb 13 4

See all from why amit

Recommended from Medium

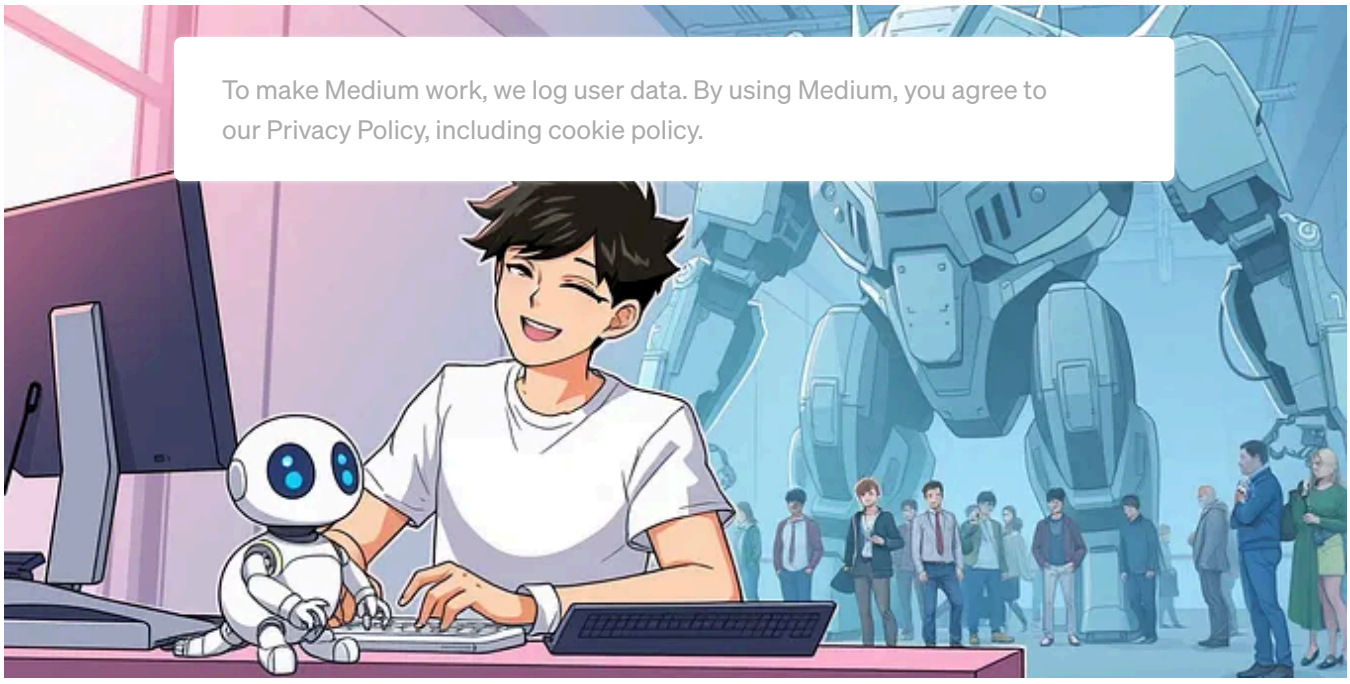


In Coding Nexus by Minervee

Build Your First AI Agent From Scratch with LangGraph, Ollama, and Qwen3

A Free Guide with Complete Execution

Oct 30 1



Rodrigo Estrada

🎯 Build a LOCAL AI Coding Assistant: Qwen3 + Ollama + Continue.dev

I wanted to share my journey using different AI coding assistants—from GitHub Copilot to Cursor and now Windsurf—and how I finally...

May 16 98 5

Rohan Dutt

Summarize Massive Documents Locally Using Langchain + Ollama (With Python Code)

Here Everyth... Master This Once and Fo...

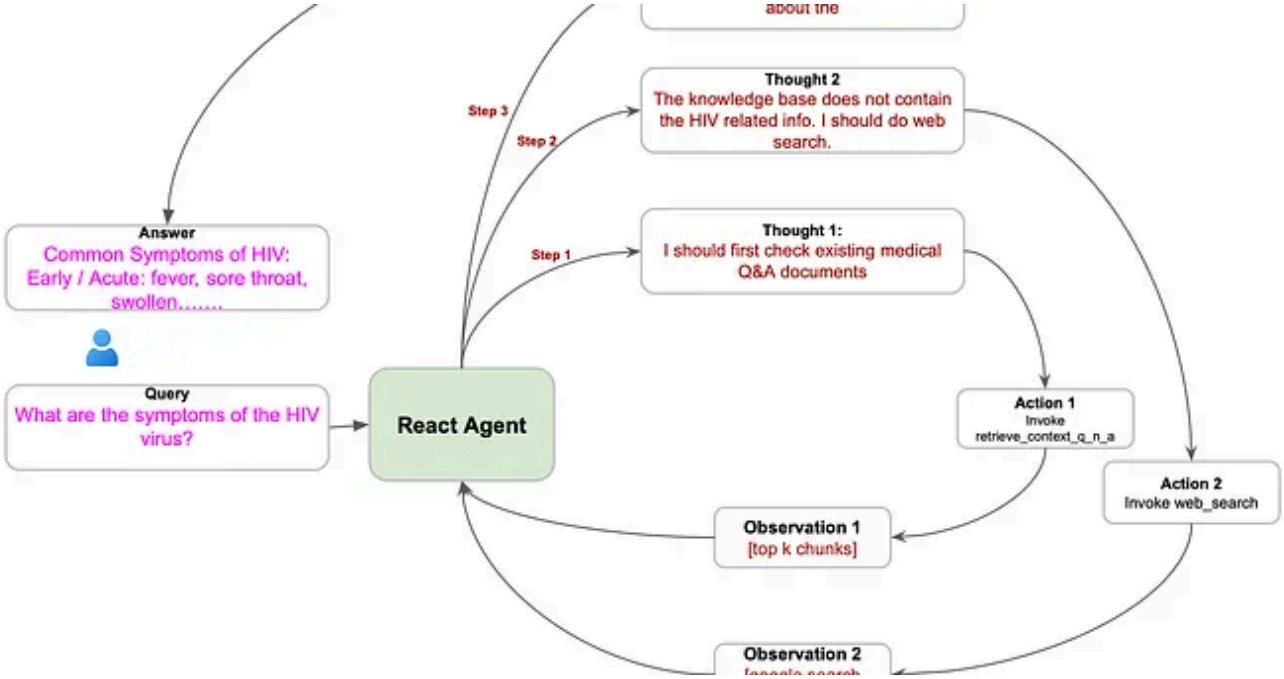
Jun 9

In Artificial Intelligence in Plain English by Jageen Shukla

How to Build Ollama-Powered AI Agents with ADK, Tool Calling, and MCP Integration

adk-ollama-mcp-tool

Jun 16 61

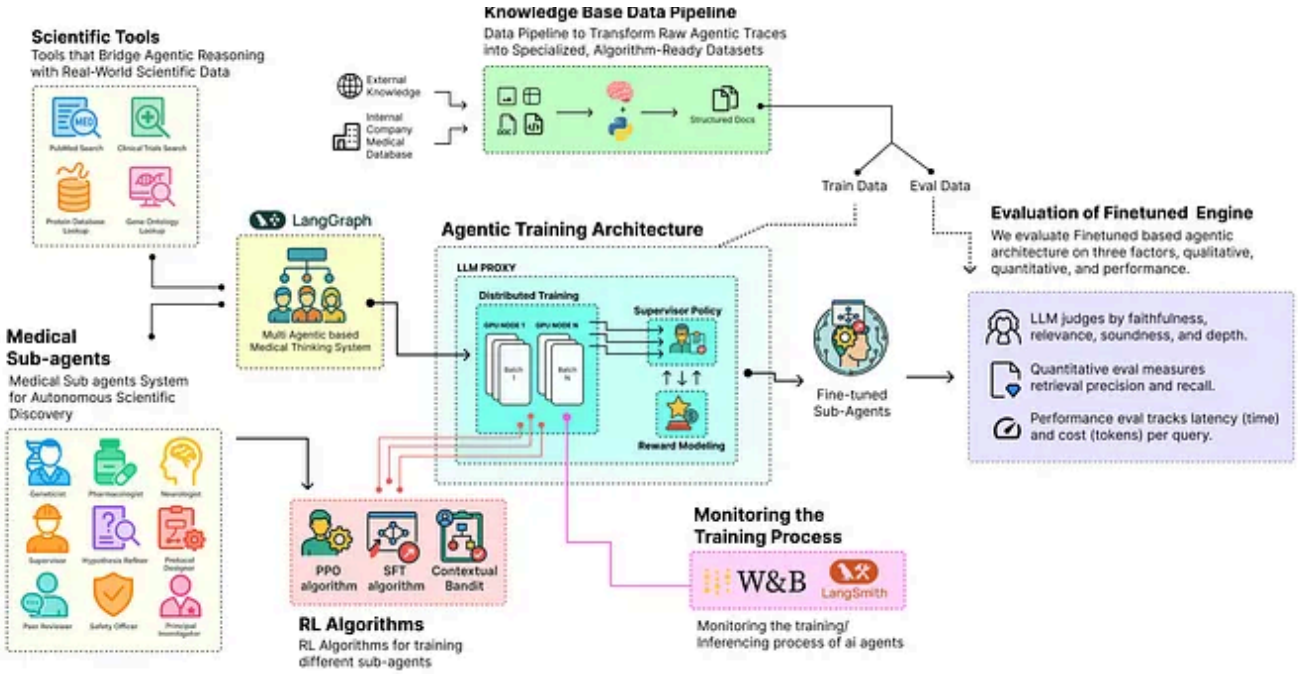


In Towards AI by Alpha Iterations

Agentic A

Practical Gu ded]

Oct 25 168 1



In Level Up Coding by Fareed Khan

Building a Training Architecture for Self-Improving AI Agents

RL Algorithms, Policy Modeling, Distributed Training and more.

3d ago 704 14

See more recommendations