The First Time It Finally Clicked

Arsh Imtiaz

2025-12-29

{{< image src="/images/ai-integration-header.svg" alt="AI Integration" caption="Local model, real system, clean output." >}}

Ollama + APIs: My First Real AI Integration

I have used AI tools before. But this post is about the *first time I actually integrated AI* into an app and felt the whole thing click.
Not just "ask a model a question." I mean an API that *uses* a model and does something useful with it.

The two pieces that made it real for me: - **Ollama** for local models
- **APIs** to make the AI actually useful inside a system

This is not a step-by-step tutorial. It is a learning story, what worked, what broke, and how it finally came together.

Why Ollama?

I wanted something local, fast, and easy to swap models without rebuilding my entire app.

Ollama gave me: - Local inference without a monthly bill
- Simple model switching
- A clean REST API I could hit from anything

Bonus: If you are already building tooling or demos, having the model local means you control latency, limits, and privacy.

I know there are solid commercial options too: AWS Bedrock, Azure OpenAI, OpenAI's APIs, Anthropic, and more. Those are great when you need scale, managed infra, or compliance. I picked Ollama for this project because I wanted to learn the integration flow without burning credits, and I wanted full control over the models and the data path. Also because my GPU could actually run it without sounding like a jet engine (for once).

{{< admonition type="note" title="Local vs Cloud" >}} Local is perfect for learning and fast iteration. Cloud is better when you need scale, uptime, or enterprise compliance. {{< /admonition >}}

{{< image src="/images/local-vs-cloud.svg" alt="Local vs cloud AI tradeoffs" caption="Local keeps you fast and private. Cloud scales when you need it." >}}

The mental model that helped me

I stopped thinking "chatbot."
I started thinking **"AI as a function in a system."**

Your API takes inputs, runs logic, calls Ollama, and returns something useful.

That is the entire loop.

Here is a plain example that made it click for me:

- Input: raw log snippet + a short user question
- API: adds context (service name, environment, expected format)
- Output: a clean JSON response I can actually use in an app

For example, instead of asking a vague question, I pass a tiny contract like this:

```
Role: SOC analyst
Task: Summarize this alert and recommend next steps.
Constraints: No guessing. Use only the provided log.
Output: JSON with keys: summary, severity, next_steps
```

Now I can plug the output into a UI, ticket, or report without manual cleanup.

{{< image src="/images/ollama-api-flow.svg" alt="Ollama + API flow" caption="Input in, logic in the middle, model on call, useful output out." >}}

Minimal setup I used

- Ollama running locally
- A tiny Python API
- `requests` to call the model endpoint
- JSON response back to the client

Start Ollama

I run it as a systemd service instead of launching it manually.

```
sudo systemctl enable --now ollama
```

Pull a model:

```
ollama pull llama3
```

Run a quick test:

```
ollama run llama3
```

The API call (this is the moment it clicked)

Ollama exposes a simple API. You just POST a prompt.

```
curl http://localhost:11434/api/generate -d '{
  "model": "llama3",
  "prompt": "Explain NAT like I am debugging a reverse shell."
}'
```

It returns a stream of JSON. If you want a single response, add:

```
"stream": false
```

My tiny Python API (trimmed down)

```python
from flask import Flask, request, jsonify
import requests

app = Flask(__name__)

@app.post("/ask")
def ask():
    payload = request.get_json(silent=True) or {}
    prompt = payload.get("prompt")
    if not prompt:
        return jsonify({"error": "Missing prompt"}), 400

    response = requests.post(
        "http://localhost:11434/api/generate",
        json={
            "model": "llama3",
            "prompt": prompt,
            "stream": False
        },
        timeout=60
    )
```

```
    data = response.json()
    return jsonify({"answer": data.get("response", "")})

if __name__ == "__main__":
    app.run(host="0.0.0.0", port=3000, debug=False)
```

Now you have a real API endpoint that uses AI.
No magic. Just a normal service that happens to be powered by a model.

{{< admonition type="tip" title="Shortcut" >}} For local experiments, go direct to the Ollama API and keep the app layer for when you need guardrails or multi-client access. {{< /admonition >}}

Why Structure Matters

{{< image src="/images/prompt-structure.svg" alt="Prompt structure for reliable outputs" caption="Clear roles, constraints, and formats turn raw model text into usable results." >}}

Here is a quick before/after that shows why structure matters:

Before (messy):

```
Explain this log and tell me what to do.
```

After (usable):

```
Role: Incident responder
Task: Summarize this log and list 3 next steps.
Constraints: Keep it under 60 words.
Output: JSON keys: summary, next_steps
```

Summary Example

Input data (from your system):

```
2025-12-29T08:41:12Z web01 nginx[4132]: 192.0.2.14 - - "POST /login HTTP/1.1" 401 612 "-" "M
2025-12-29T08:41:18Z web01 nginx[4132]: 192.0.2.14 - - "POST /login HTTP/1.1" 401 612 "-" "M
2025-12-29T08:41:24Z web01 nginx[4132]: 192.0.2.14 - - "POST /login HTTP/1.1" 401 612 "-" "M
2025-12-29T08:41:31Z web01 nginx[4132]: 192.0.2.14 - - "POST /login HTTP/1.1" 401 612 "-" "M
```

Structured prompt you send to Ollama:

```
Role: Security analyst
Task: Summarize the activity and recommend next steps.
Context: These are auth failures on a public web app.
Constraints: No guessing. Use only the log lines. Keep it concise.
Output: JSON with keys: summary, severity, indicators, next_steps
```

```
Logs:
<paste logs here>
```

Why this works: - The role sets the tone (security analyst, not “creative writer”). - The context removes ambiguity (auth failures on a public app). - Constraints reduce hallucination (only log lines). - The output contract makes the response machine-friendly.

Example response you can actually use:

```
{
  "summary": "Four failed login attempts from 192.0.2.14 to /login within 20 seconds.",
  "severity": "low",
  "indicators": ["192.0.2.14", "/login", "401"],
  "next_steps": [
    "Check if the IP repeats across other hosts or time windows.",
    "Review account lockout policy for repeated failures.",
    "Add the IP to a temporary watchlist if failures continue."
  ]
}
```

Now your API can return this JSON straight into a ticketing system, a dashboard, or a Slack alert without hand-editing.

{{< admonition type=“tip” title=“Related” >}} If you are into detection labs, my Wazuh post might be a fun next read. {{< /admonition >}}

What I learned the hard way

- **You need structure.** Raw model text is messy. I started wrapping prompts with output requirements and it got way better.
- **Latency matters.** Even local models can feel slow if you do not manage prompt size.
- **Security is still security.** This is still an API. Validate inputs, rate limit, log requests.

AI does not replace good engineering. It just adds a powerful function call in the middle.

{{< admonition type=“warning” title=“Do not skip this” >}} If your prompt is vague, your output will be vague. Structure is the cheapest reliability upgrade you can make. {{< /admonition >}}

Practical use cases I am building next

- Summarize logs into incident notes
- Auto-tag security alerts
- Generate draft incident reports
- Turn raw terminal output into explanations for non-technical teams

These are not “chatbot” tasks. They are *workflow* tasks.

Final thoughts

This was the first time I felt like AI was not a toy but a **real system component**. Ollama made it local and easy. The API made it usable.

If you are learning AI integration, my honest advice is: - Build something small
- Make it usable
- Then make it smarter

Because once AI becomes an API, you can wire it into anything.

{{< admonition type=“success” title=“Takeaway” >}} Treat the model like a dependency, not a magic box. The more deliberate your inputs, the more usable your outputs. {{< /admonition >}}

If you want to compare notes or want help debugging your integration, reach out. I am still learning too, and that is the fun part.