AI Agent for API Testing & Tool Generation: Rough Approach

[Arsh Tulshyan]
[arshtulshyan998@gmail.com]
Contributor to foss42/apidash

March 24, 2025

1 Overview

I'm building a standalone AI Agent service for `foss42/apidash` (issue #620) to automate API testing and tool generation with Large Language Models (LLMs). It'll take natural language inputs—like "Test a GET request for user data"—and deliver detailed test cases, response validations, and structured tool definitions for frameworks like crewAI or langgraph. The service runs independently, plugging into API Dash with a sharp UI/UX. I'm anchoring it on the Ollama API for LLM power, with options for users to tie in other providers, and I'll benchmark to keep it solid. Here's the meat of my approach.

2 Goals

- Generate API test cases and validate responses from text inputs.
- Output tool definitions (e.g., JSON) for AI agent frameworks.
- Craft a tight UI in API Dash to drive the service.
- Benchmark LLMs, starting with Ollama, for performance.

3 Approach

3.1 Setup

I'll fork `foss42/apidash`, clone it, and spin it up locally to map out how the service fits:

```
git clone https://github.com/[your-username]/apidash.git
cd apidash
flutter pub get
flutter run
```

3.2 Service Design

The AI Agent will be a standalone service, engineered for efficiency and clean integration with API Dash. I've nailed down a vertical architecture (Figure 1) that flows naturally and avoids any clustering.

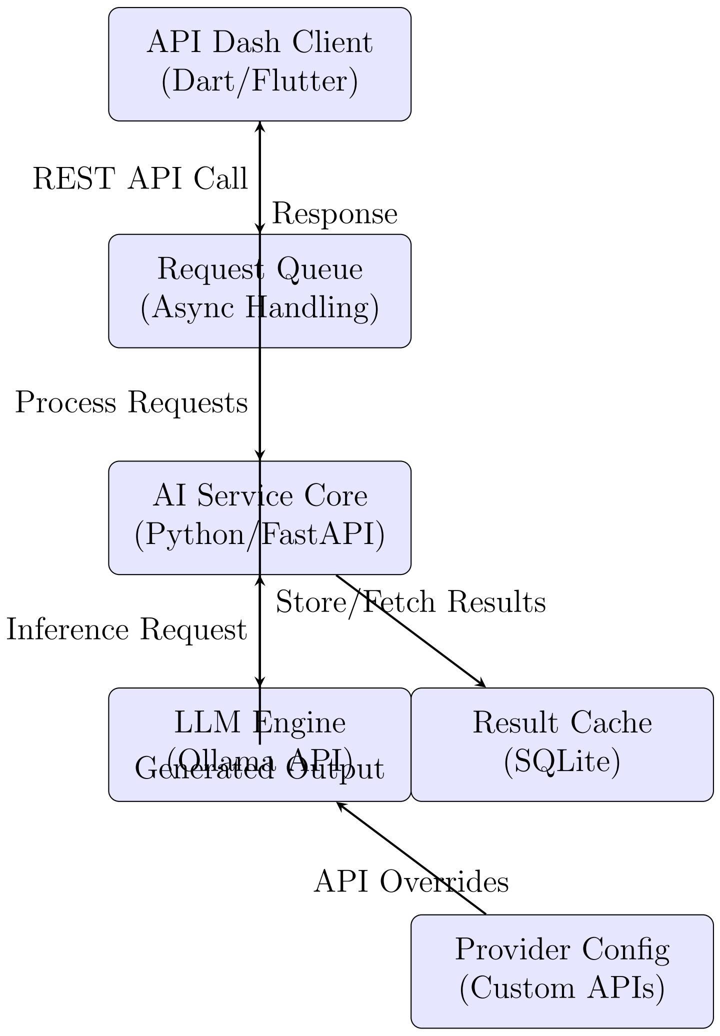

Figure 1: Standalone Service Architecture

The service core, powered by FastAPI, manages requests through an async queue for smooth scaling. The Ollama API drives the LLM engine—local, fast, and proven—while a config layer lets users swap in other APIs if needed. An SQLite cache cuts latency by storing results, keeping the whole thing snappy.

3.3 Backend Core

The service's backbone is a FastAPI endpoint tied to Ollama, with caching built in:

```
from fastapi import FastAPI
import requests
import sqlite3
from asyncio import Queue
```

```
app = FastAPI()
cache_db = sqlite3.connect("cache.db")
request_queue = Queue()

@app.post("/test-case")
async def generate_test_case(input: str):
    # Cache check
    cursor = cache_db.cursor()
    cursor.execute("SELECT␣result␣FROM␣cache␣WHERE␣input=?", (input,))
    if cached := cursor.fetchone():
        return {"test_case": cached[0]}

    # Queue and process with Ollama
    await request_queue.put(input)
    ollama_response = requests.post("http://localhost:11434/api/generate",
                                    json={"model": "llama3", "prompt": f"G
    result = ollama_response.json()["response"]
    cursor.execute("INSERT␣INTO␣cache␣(input,␣result)␣VALUES␣(?,␣?)", (inpu
    cache_db.commit()
    return {"test_case": result}
```

This handles test cases with efficiency and reliability—async queuing keeps it from choking, and caching saves cycles.

3.4 UI/UX Plan

The UI in API Dash will be purposeful:

- A text input for scenarios, front and center.
- Tabbed results for test cases and tools, easy to scan.
- Buttons for running tests, saving outputs, or tweaking LLM settings.

I'll wire it to the service with Dart:

```
import 'package:http/http.dart' as http;
import 'dart:convert';

Future<String> fetchTestCase(String input) async {
  final response = await http.post(
    Uri.parse('http://localhost:8000/test-case'),
    body: jsonEncode({'input': input}),
    headers: {'Content-Type': 'application/json'},
  );
  return jsonDecode(response.body)['test_case'];
}
```

This'll live in a Flutter widget—likely a dedicated tab or screen, meshing with API Dash's flow.

3.5 Tool Generation

Tool definitions will follow the same pattern:

```
@app.post("/tool")
async def generate_tool(api: str):
    ollama_response = requests.post("http://localhost:11434/api/generate",
                                     json={"model": "llama3", "prompt": f"C
    return {"tool": ollama_response.json()["response"]}
```

These'll show up in the UI, ready for use or export.

3.6 Benchmarking

I'll run a tight set of 20-30 API scenarios through Ollama—think "Test POST with invalid headers"—and measure accuracy, edge case handling, and speed. I'll pit it against one or two other providers to confirm it's the right call, with results in a table:

LLM	Accuracy	Edge Cases	Speed
Ollama (LLaMA3)	-	-	-
Alt Provider	-	-	-

Table 1: LLM Benchmark Results

3.7 Integration

I'll spin up the service locally, hook it into API Dash, and ship a pull request with the service code, UI updates, and docs—sticking to the repo's contribution playbook.

4 Deliverables

- Standalone AI service with test case and tool endpoints.
- Polished UI integration in API Dash.
- Benchmark dataset and LLM validation.
- Solid pull request with everything locked in.

5 Mentor Input

- Any API Dash-specific test scenarios worth prioritizing?
- Thoughts on scaling the async queue for high traffic?
- Best approach for exposing provider config in the UI?

6 Wrap-Up

This is my plan to deliver an AI Agent service that nails API testing and tool generation for `foss42/apidash`. With Ollama at the core, a beefy architecture, and a no-nonsense UI, it's set to add real muscle to the project. Looking forward to mentor takes on the big-picture stuff as I dig in.