Both requests and httpx are popular Python libraries for making HTTP requests, but they differ in design, features, and use cases. Below is a detailed comparison of the two, including their differences, strengths, and when to use each.

**Overview**

* **requests**: A widely-used, synchronous HTTP library that simplifies making HTTP requests in Python. It’s part of the Python ecosystem since 2011 and is known for its simplicity and extensive community support.
* **httpx**: A modern HTTP client that supports both synchronous and asynchronous requests. Introduced more recently, it builds on requests’s API while adding async capabilities and advanced features.

**Comparison**

| **Feature/Property** | **requests** | **httpx** |
| --- | --- | --- |
| **Release Date** | February 2011 | October 2019 |
| **Synchronous Support** | Yes | Yes |
| **Asynchronous Support** | No (requires third-party libs like aiohttp) | Yes (built-in with asyncio) |
| **API Similarity** | Native, simple, and intuitive | Similar to requests but extended |
| **HTTP Methods** | GET, POST, PUT, DELETE, etc. | GET, POST, PUT, DELETE, etc. (same) |
| **Timeouts** | Yes (via timeout parameter) | Yes (more granular: connect, read, write) |
| **SSL/TLS** | Yes (via verify and cert) | Yes (with advanced options) |
| **Proxy Support** | Yes | Yes (with more flexibility) |
| **Session Management** | Yes (via requests.Session) | Yes (via httpx.Client) |
| **Streaming** | Yes (via response.iter\_content) | Yes (via response.aiter\_bytes or sync) |
| **HTTP/2 Support** | No | Yes (optional, requires httpcore with HTTP/2) |
| **WebSocket Support** | No | Yes (via httpx.AsyncClient) |
| **Dependency Size** | Larger (includes urllib3, etc.) | Smaller (uses httpcore as base) |
| **Performance** | Good for synchronous tasks | Better for async/heavy workloads |
| **Community Support** | Extensive, mature ecosystem | Growing, but less mature |
| **Latest Version (as of 2025)** | ~2.32.0 (check PyPI) | ~0.27.0 (check PyPI) |

**Key Differences**

1. **Synchronous vs. Asynchronous**:
   * **requests**: Purely synchronous. All requests block the execution until a response is received. For async, you’d need to pair it with asyncio and another library like aiohttp.
   * **httpx**: Supports both synchronous and asynchronous modes. Use httpx for sync or httpx.AsyncClient for async, making it ideal for modern async applications (e.g., FastAPI, asyncio-based servers).
2. **API Design**:
   * **requests**: Offers a straightforward API (e.g., requests.get(url)), which is easy for beginners but lacks async support.
   * **httpx**: Mirrors requests’s API for sync usage (e.g., httpx.get(url)), but extends it with async methods (e.g., await httpx.AsyncClient().get(url)), providing a unified interface.
3. **Performance and Features**:
   * **requests**: Relies on urllib3, which is robust but not optimized for high-concurrency scenarios. No HTTP/2 or WebSocket support.
   * **httpx**: Built on httpcore, it supports HTTP/2 (with configuration) and WebSockets, offering better performance for concurrent requests and modern protocols.
4. **Timeout Granularity**:
   * **requests**: Uses a single timeout parameter for both connect and read timeouts.
   * **httpx**: Allows separate timeouts for connect, read, and write (e.g., httpx.Client(timeout=httpx.Timeout(connect=5, read=10))), providing finer control.
5. **Use Cases**:
   * **requests**: Best for simple, synchronous HTTP tasks (e.g., scripts, small applications) where async isn’t needed.
   * **httpx**: Ideal for async applications, high-performance APIs, or projects requiring HTTP/2, WebSockets, or concurrent requests (e.g., web scrapers, microservices).

**Example Code**

**Using requests (Synchronous)**

python

Copy

import requests

response = requests.get("https://api.example.com", timeout=5)

print(response.text)

**Using httpx (Synchronous)**

python

Copy

import httpx

with httpx.Client(timeout=httpx.Timeout(connect=5, read=10)) as client:

response = client.get("https://api.example.com")

print(response.text)

**Using httpx (Asynchronous)**

python

Copy

import httpx

import asyncio

async def fetch():

async with httpx.AsyncClient() as client:

response = await client.get("https://api.example.com")

print(response.text)

asyncio.run(fetch())

**Pros and Cons**

* **requests**:
  + **Pros**: Simple to use, widely adopted, extensive documentation, no async overhead.
  + **Cons**: No async support, no HTTP/2, less suited for high-concurrency needs.
* **httpx**:
  + **Pros**: Async support, HTTP/2, WebSocket, fine-tuned timeouts, modern design.
  + **Cons**: Less mature ecosystem, slightly steeper learning curve for async, additional dependencies.

**When to Use**

* Use **requests** if:
  + You’re building a simple script or application.
  + You don’t need async or HTTP/2.
  + You prefer a battle-tested library with broad compatibility.
* Use **httpx** if:
  + You’re developing an async application (e.g., with FastAPI or asyncio).
  + You need HTTP/2, WebSockets, or high-concurrency performance.
  + You want a modern, extensible HTTP client.

**Integration with Your MCP Server**

* Since your server.py uses requests for the Serper API, you could replace it with httpx for async support if your MCP server leverages asyncio. Example:

python

Copy

@mcp.tool()

async def search(query: str) -> dict:

api\_key = os.getenv("SERPER\_API\_KEY")

async with httpx.AsyncClient() as client:

response = await client.post("https://google.serper.dev/search", headers={"X-API-KEY": api\_key}, json={"q": query, "num": 5}, timeout=10)

data = response.json()

results = [{"title": item.get("title", "No title"), "link": item.get("link", "")} for item in data.get("organic", [])]

return {"query": query, "results": results}

* Adjust mcp.run to support async if needed (check MCP docs).