Optimización del Evaluator en el sistema de Log Detector

Avance Final

INTRODUCCIÓN A LA COMPUTACIÓN PARALELA

Vicente Coopman
2022/08/04

Planificación

Avance 1: 26 de Mayo, 2022

1. Estudio sobre el nivel de rendimiento actual.

2. Estudio sobre herramientas y métodos para mejorar el rendimiento.

Avance 2: 21 de Julio, 2022

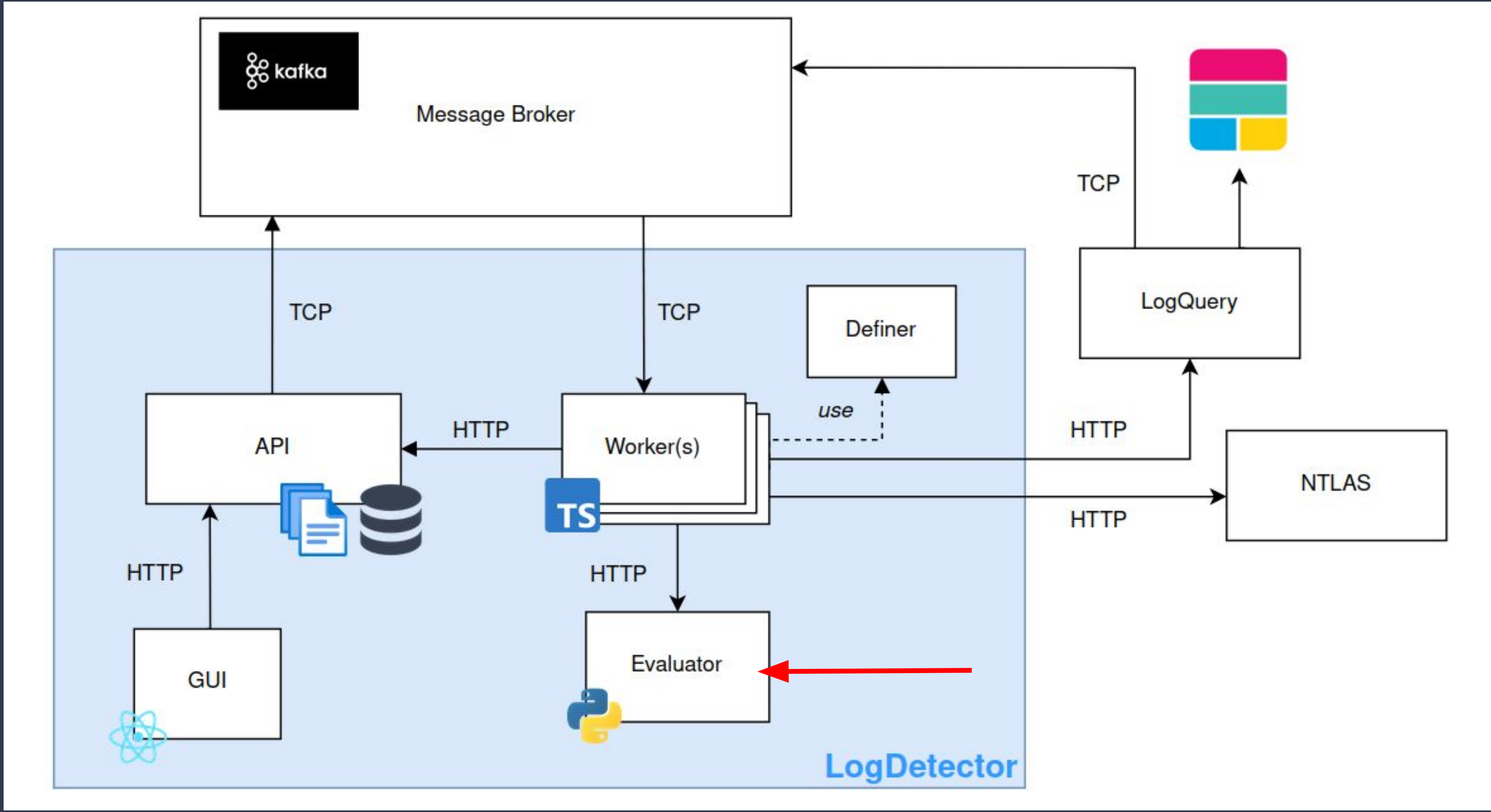
1. Implementación de herramientas y/o métodos seleccionados.

2. Comparación de rendimientos post implementación.

Planificación

Presentación e informe final: 3 de Agosto 2022

1. Mejoras a la implementación anterior.
2. Resultados e informe final.


kafka
Message Broker
TCP
TCP
TCP
Definer
LogQuery
use
HTTP
HTTP
API
Worker(s)
NTLAS
HTTP
HTTP
HTTP
Evaluator
GUI
LogDetector

Web Servers & Python Web apps 101

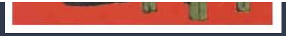
Motivación

- Conocimiento útil en la vida laboral, o con fines recreativos.
- Ejemplo de un caso real de donde se ocupa el paralelismo.
- Ayuda para entender el proyecto.

Motivación

DISCLAIMER:

I AM NOT AN EXPERT



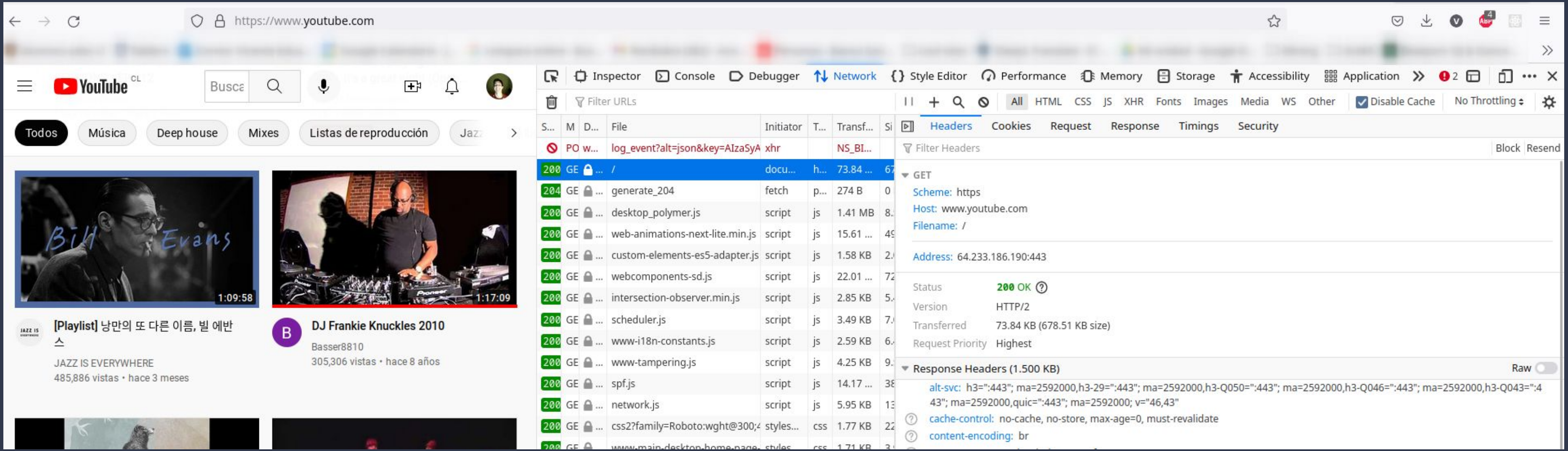
Ejemplo

Imaginemos que accedemos a un servicio web hecho en Python, como por ejemplo Youtube.

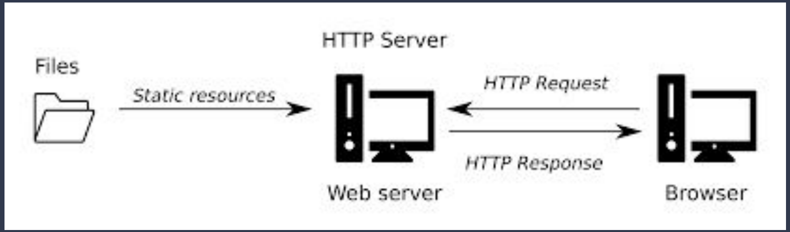
¿Cómo se vería esto del lado computín?

Ejemplo - Acceder a Youtube

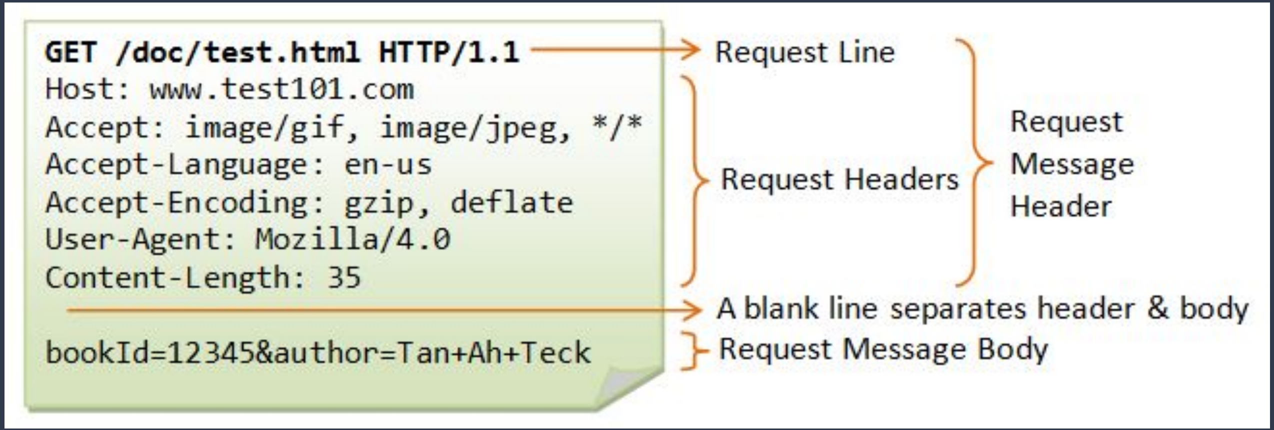
- Nuestro browser envía una request HTTP al web server de Youtube y recibe una respuesta con el contenido solicitado.



Web Server



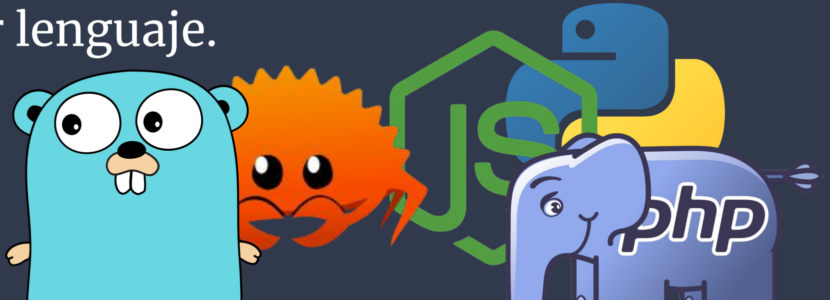
- Software que ofrece servicios/contenido mediante el protocolo Hypertext Transfer Protocol (HTTP).
- Se encarga de “servir”. Sirve recursos a usuarios a través de internet.



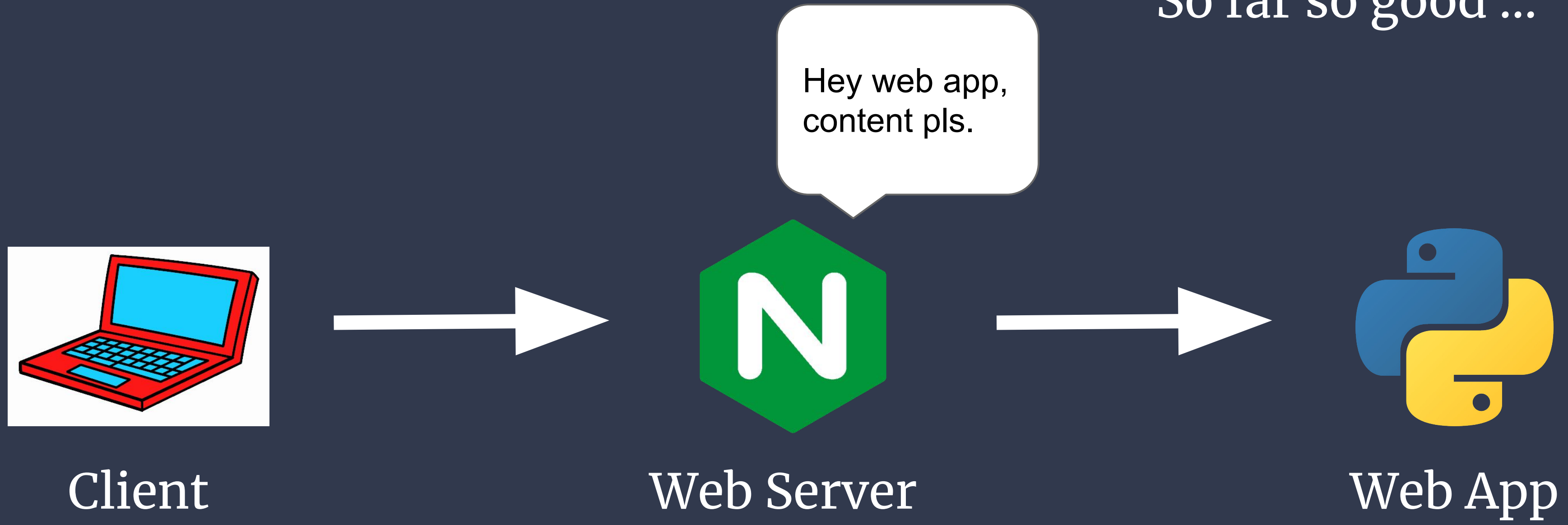
What about dynamic content?

- ¿Cómo genera Youtube el feed específicamente para cada uno de nosotros?
  → Mediante aplicaciones que dinámicamente generan contenido estático.

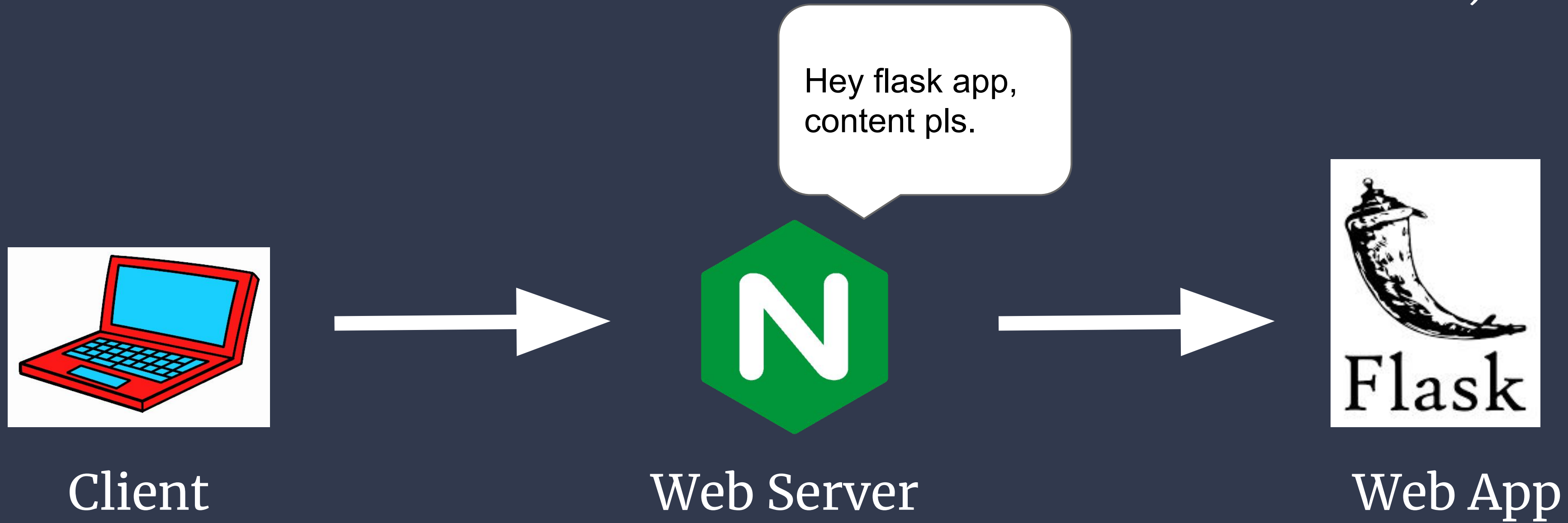
  Note: Estas aplicaciones no tienen porqué ser necesariamente Python.
  Pueden ser implementadas en cualquier lenguaje.



So far so good ...

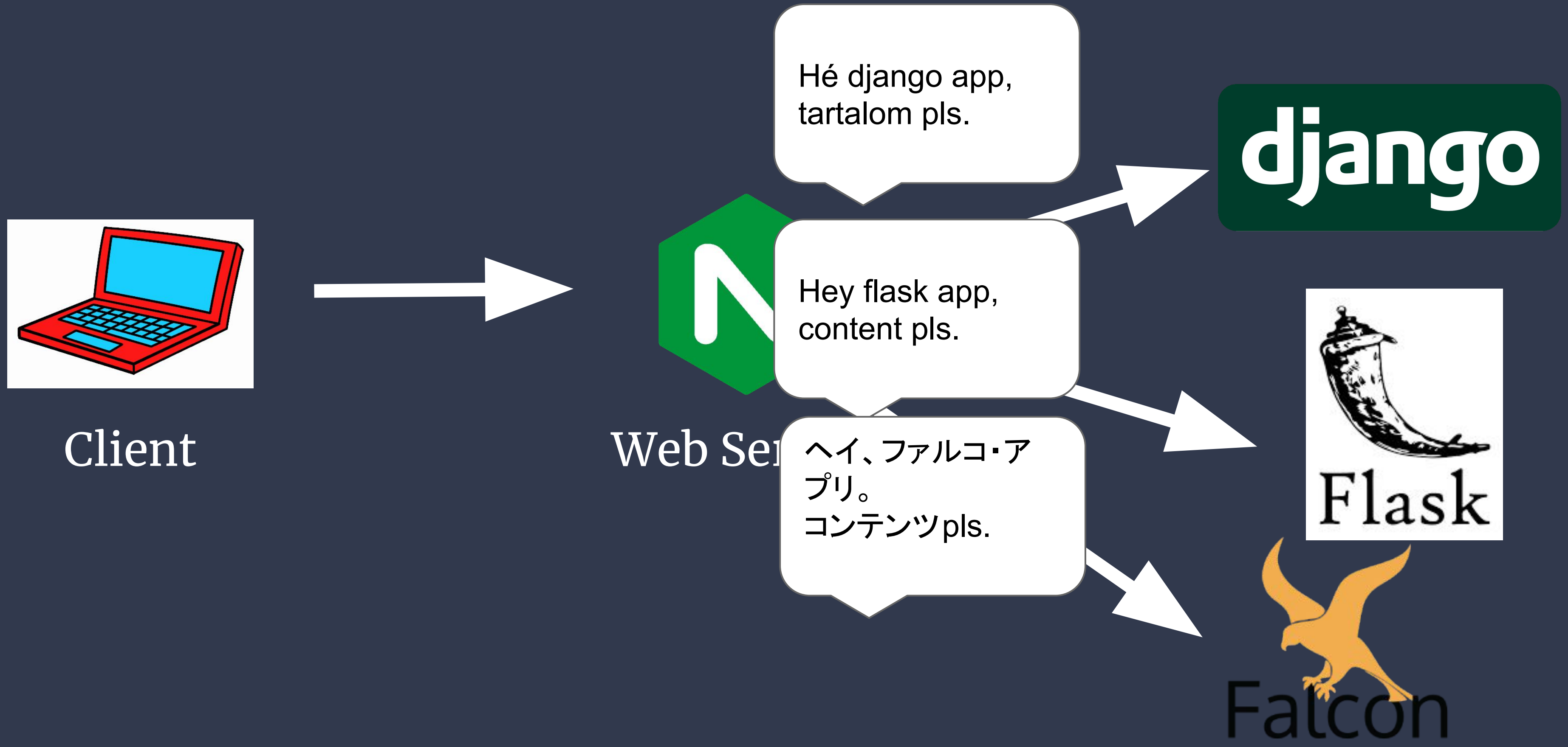


Still OK :)

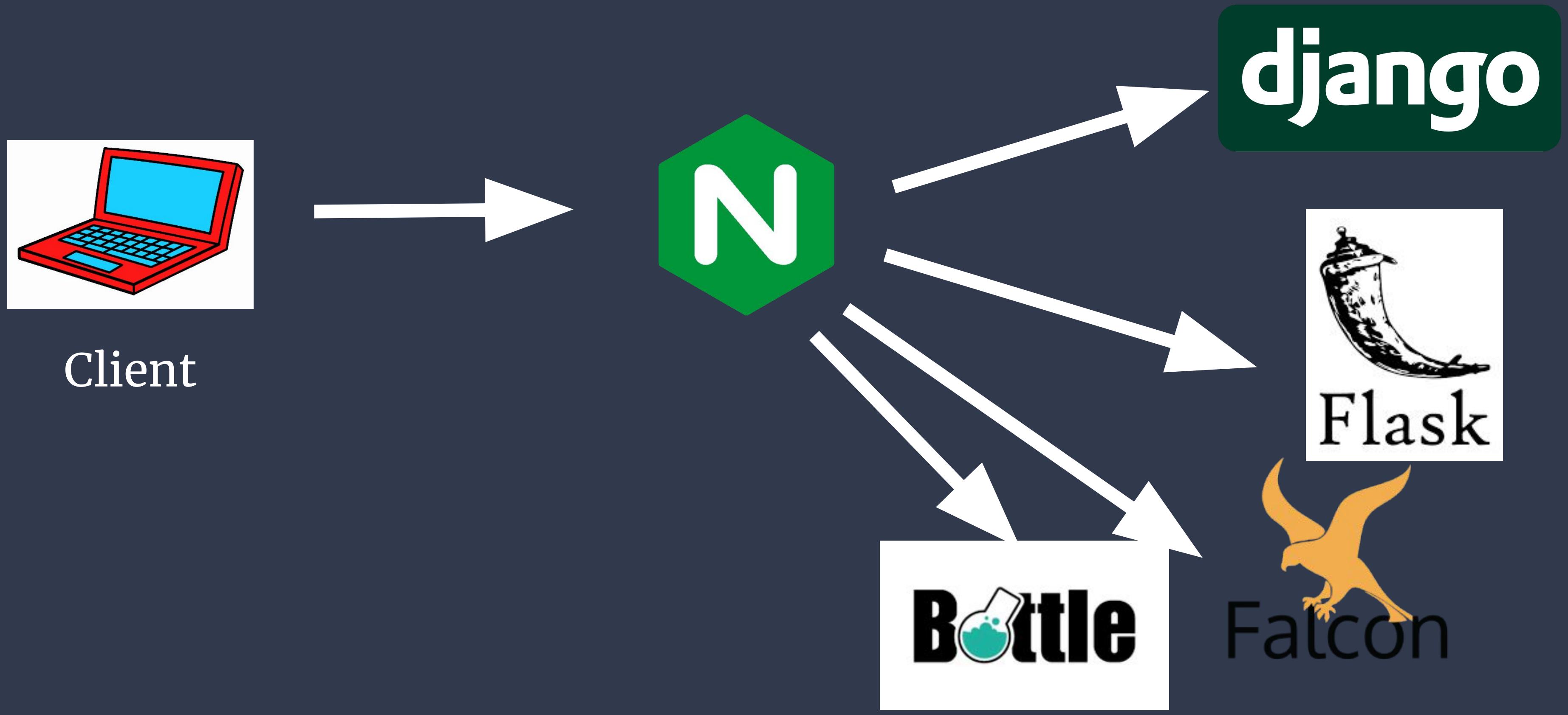


Double work, but still OK ...

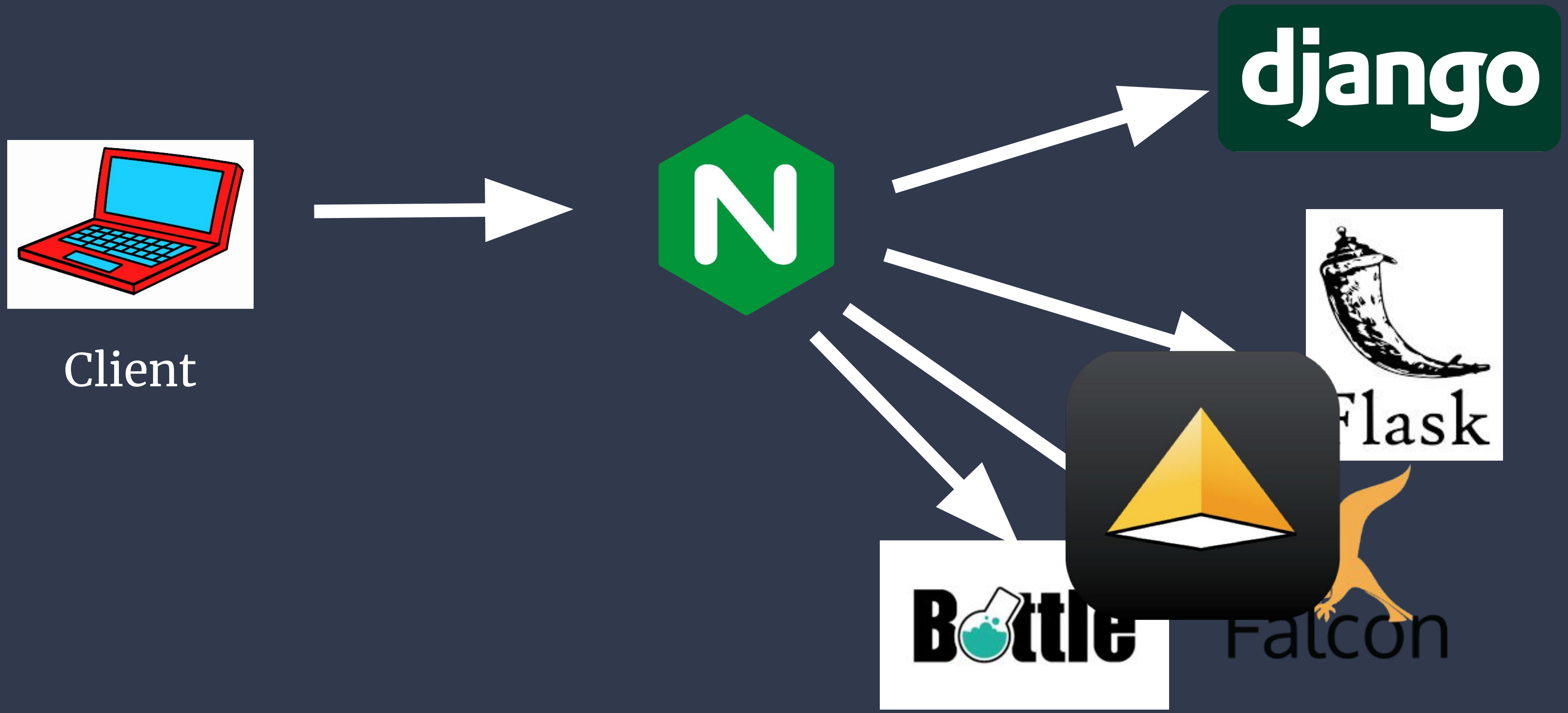
Triple work, getting tired...



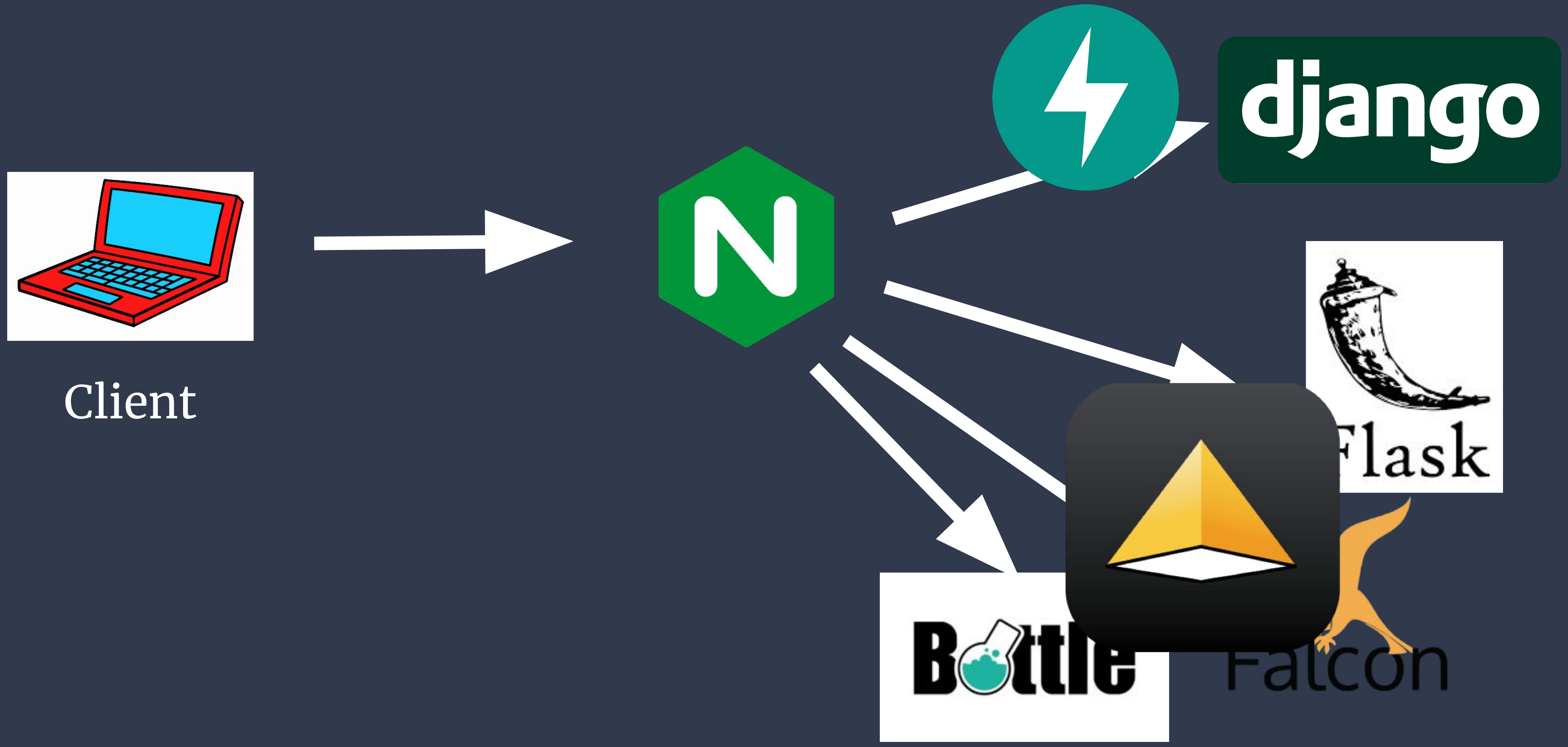
Cuádruple work … bro…



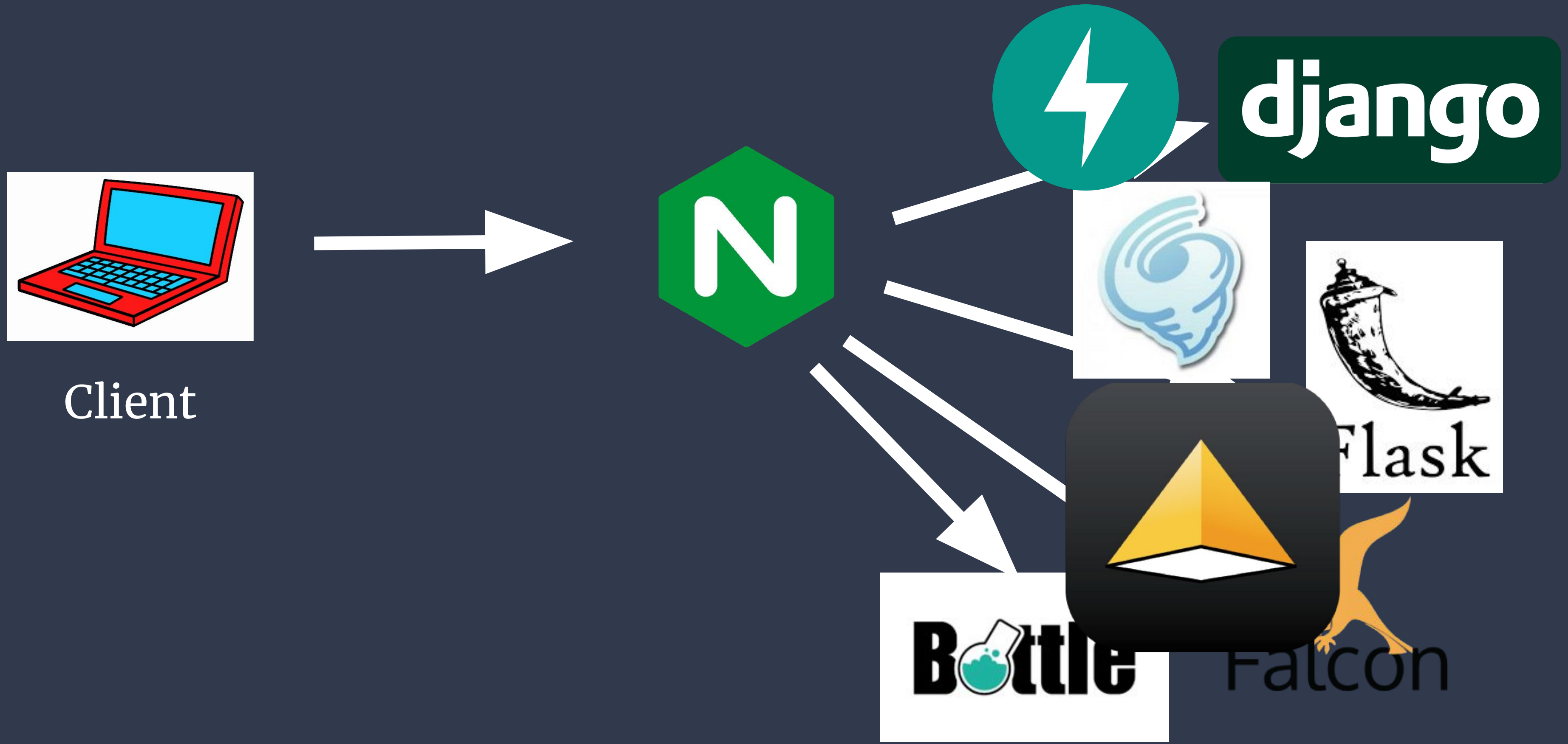
Cuádruple work ... bro...



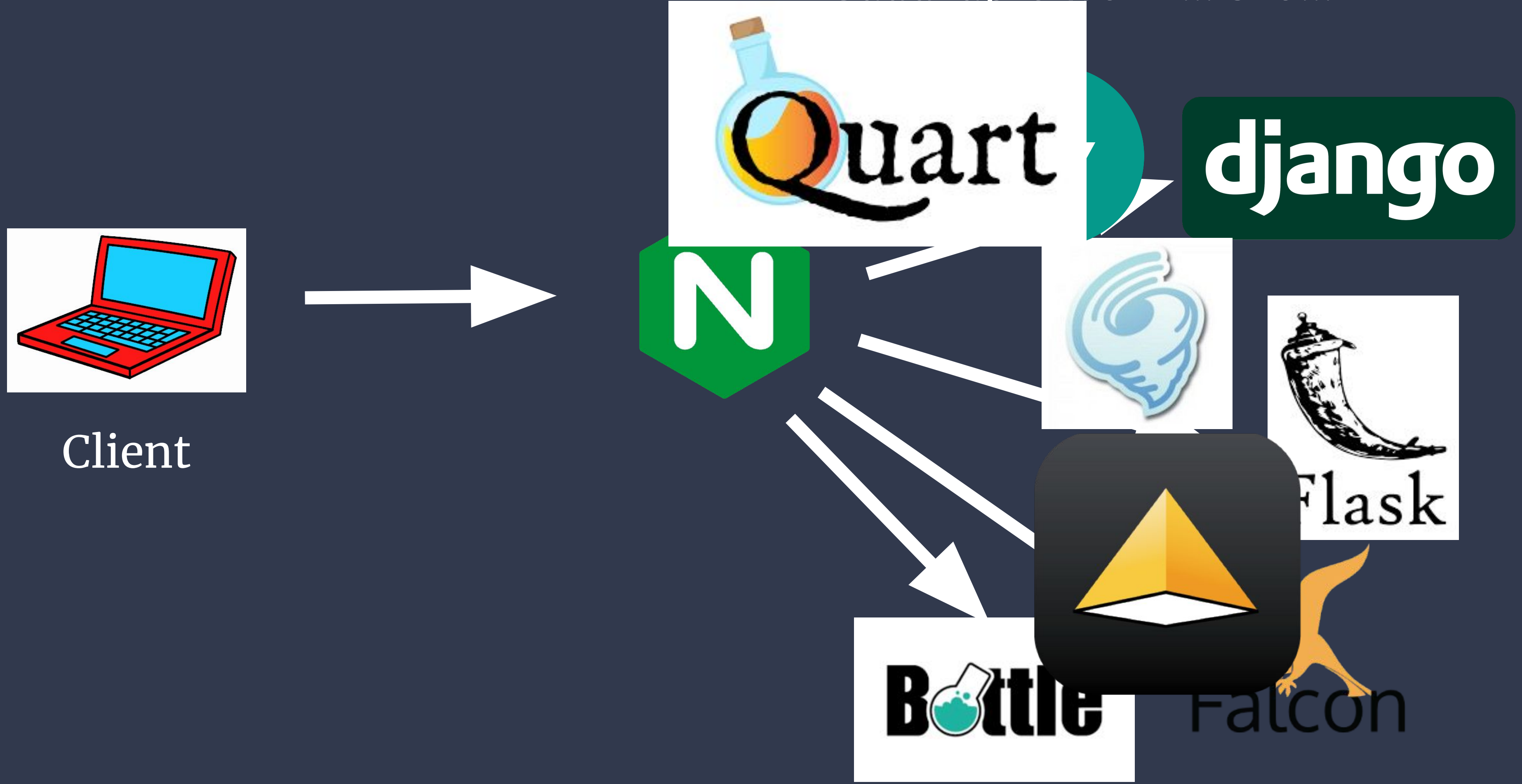
Cuádruple work ... bro...



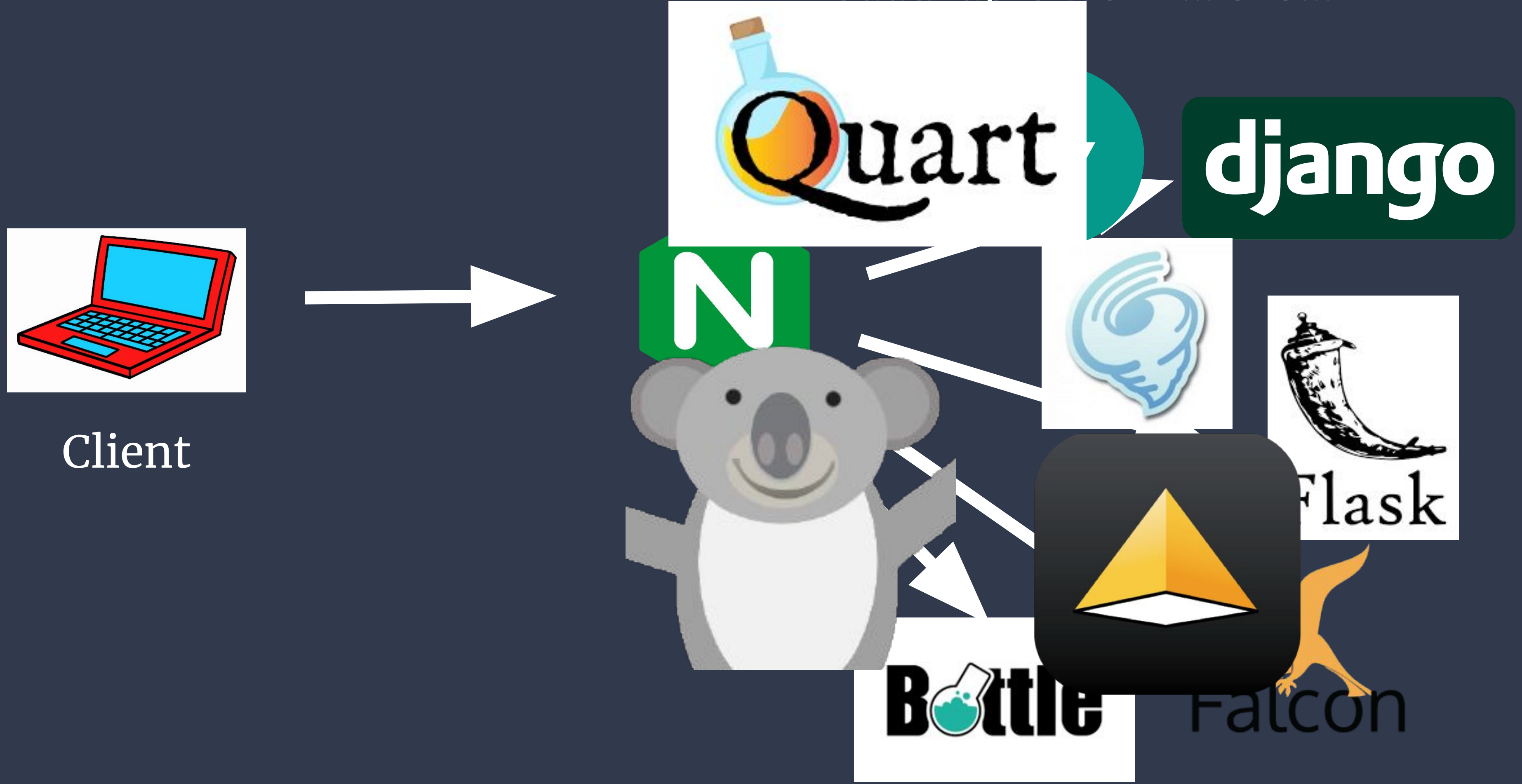
Cuádruple work ... bro...



Cuádruple work ... bro...



Cuádruple work ... bro...



Web Server Gateway Interface (WSGI)

- Interfaz que implementa tanto el web server como la web app para estandarizar la comunicación.

Abstract

This document specifies a proposed standard interface between web servers and Python web applications or frameworks, to promote web application portability across a variety of web servers.

Rationale and Goals

Python currently boasts a wide variety of web application frameworks, such as Zope, Quixote, Webware, SkunkWeb, PSO, and Twisted Web – to name just a few [1]. This wide variety of choices can be a problem for new Python users, because generally speaking, their choice of web framework will limit their choice of usable web servers, and vice versa.

https://peps.python.org/pep-0333/

Web Serv ay Interface (WSGI)

- Interfaz que im ment pp para estandarizar la comunic

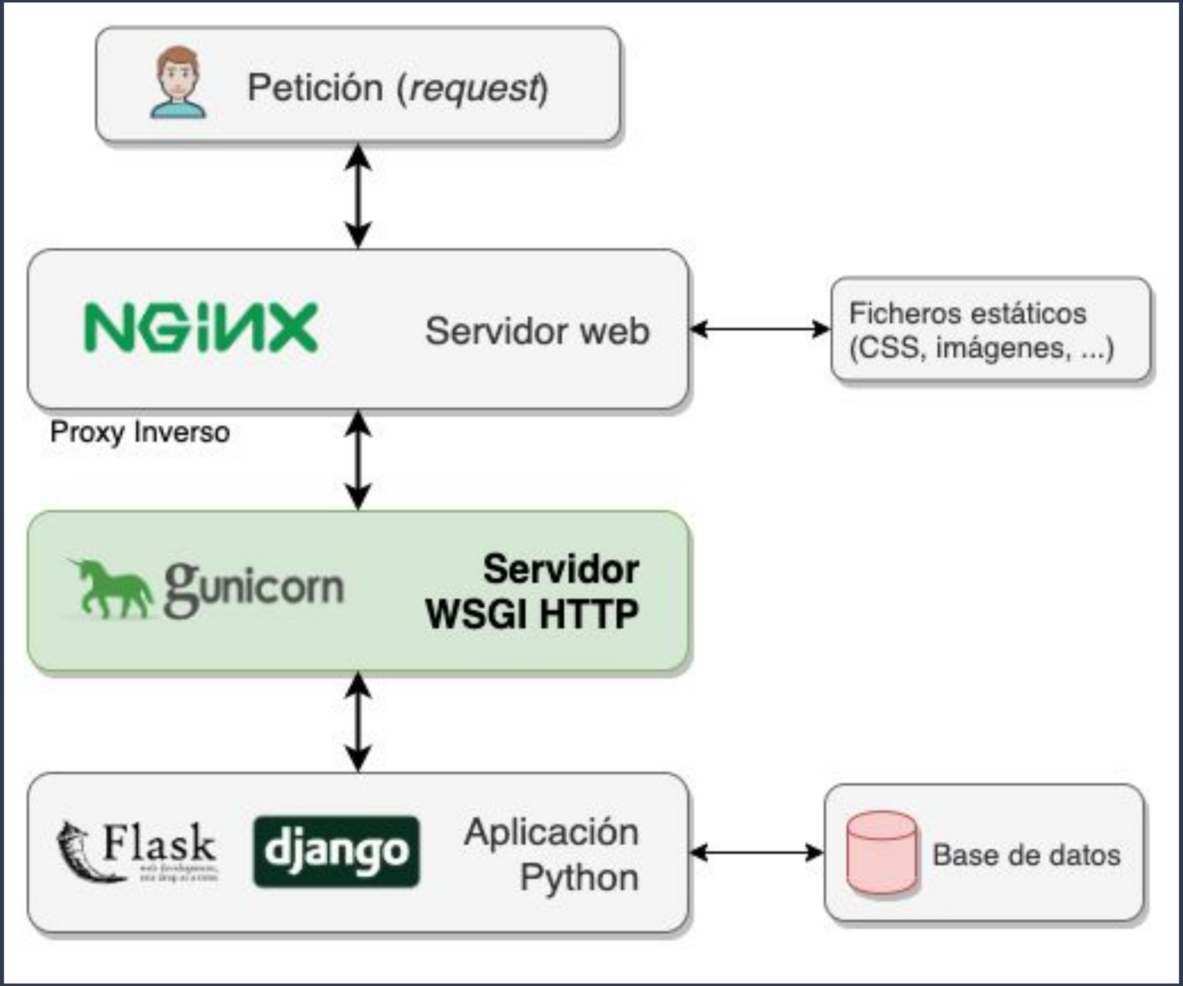
Abstract

This document specifies appl or frameworks, to promo catior ity across a variety of s.

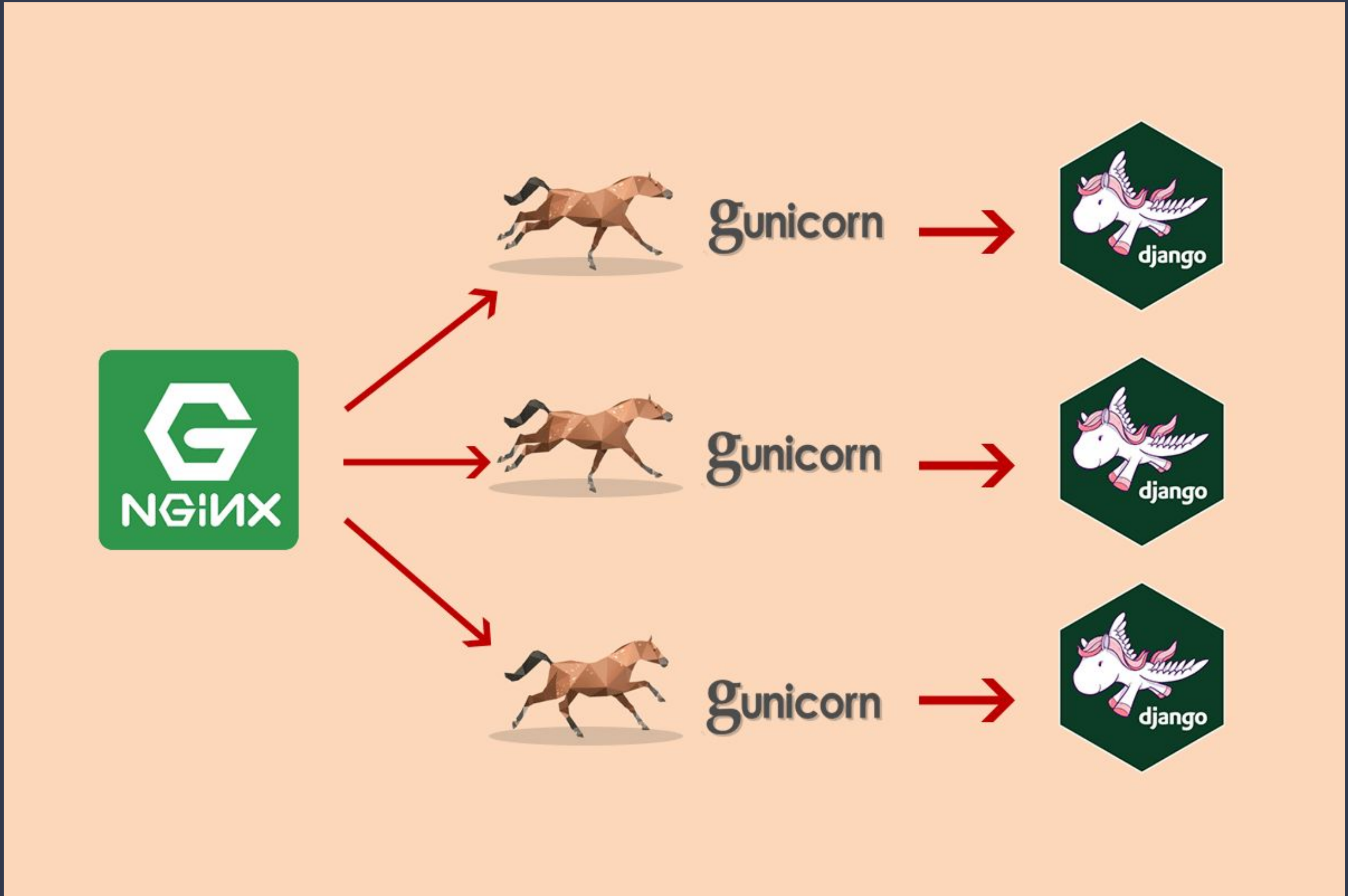
Rationale an oals

Python currently boas variety of web tion frameworks, Zope, Quixote, are, SkunkWeb, PSO, and Twisted Web – just a few [1]. de variety of ch n be a problem fo Python users, because generally speaking, eir choice of web work will limit t oice of usable web rs, and vice versa.

https://peps.python.org/pep-0333/


Petición (*request*)
NGINX
Servidor web
Ficheros estáticos
(CSS, imágenes, ...)
Proxy Inverso
gunicorn
Servidor
WSGI HTTP
Flask
django
Aplicación
Python
Base de datos

OK, nice.
But where is
my parallelism?


NGINX
gunicorn
gunicorn
gunicorn
django
django
django

Server Model

Gunicorn is based on the pre-fork worker model. This means that there is a central master process that manages a set of worker processes. The master never knows anything about individual clients. All requests and responses are handled completely by worker processes.

https://docs.gunicorn.org/en/latest/design.html

How Many Workers?

DO NOT scale the number of workers to the number of clients you expect to have. Gunicorn should only need 4-12 worker processes to handle hundreds or thousands of requests per second.

Gunicorn relies on the operating system to provide all of the load balancing when handling requests. Generally we recommend `(2 x $num_cores) + 1` as the number of workers to start off with. While not overly scientific, the formula is based on the assumption that for a given core, one worker will be reading or writing from the socket while the other worker is processing a request.

Obviously, your particular hardware and application are going to affect the optimal number of workers. Our recommendation is to start with the above guess and tune using TTIN and TTOU signals while the application is under load.

Always remember, there is such a thing as too many workers. After a point your worker processes will start thrashing system resources decreasing the throughput of the entire system.

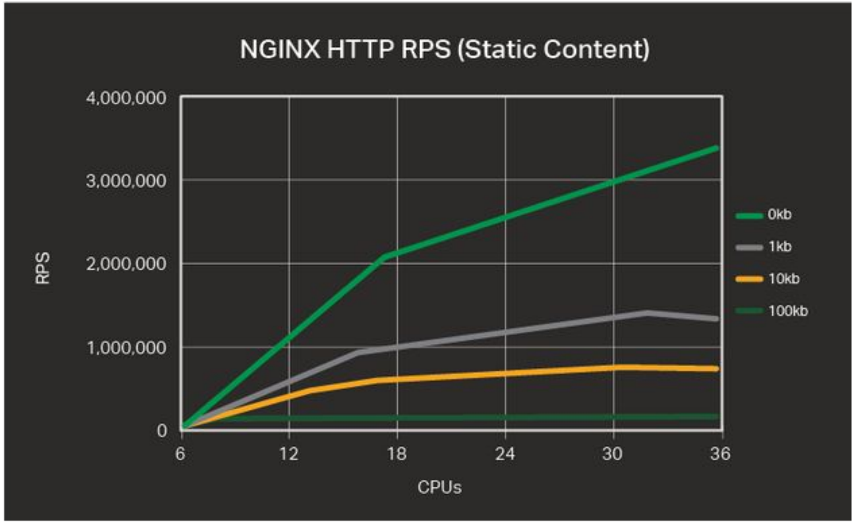
https://docs.gunicorn.org/en/latest/design.html

What about NGINX?

RPS for HTTP Requests

The table and graph below show the number of HTTP requests for varying numbers of CPUs and varying request sizes, in kilobytes (KB).

CPUs	0 KB	1 KB	10 KB	100 KB
1	145,551	74,091	54,684	33,125
2	249,293	131,466	102,069	62,554
4	543,061	261,269	207,848	88,691
8	1,048,421	524,745	392,151	91,640
16	2,001,846	972,382	663,921	91,623
32	3,019,182	1,316,362	774,567	91,640
36	3,298,511	1,309,358	764,744	91,655



https://www.nginx.com/blog/testing-the-performance-of-nginx-and-nginx-plus-web-servers/

What about NGINX?

https://www.nginx.com/blog/inside-nginx-how-we-designed-for-performance-scale/

What a
NGINX

Why Is Architecture Important?

The fundamental basis of any Unix application is the thread or process. (From the Linux OS perspective, threads and processes are mostly identical; the major difference is the degree to which they share memory.) A thread or process is a self-contained set of instructions that the operating system can schedule to run on a CPU core. Most complex applications run multiple threads or processes in parallel for two reasons:

- They can use more compute cores at the same time.
- Threads and processes make it very easy to do operations in parallel (for example, to handle multiple connections at the same time).

Processes and threads consume resources. They each use memory and other OS resources, and they need to be swapped on and off the cores (an operation called a *context switch*). Most modern servers can handle hundreds of small, active threads or processes simultaneously, but performance degrades seriously once memory is exhausted or when high I/O load causes a large volume of context switches.

The common way to design network applications is to assign a thread or process to each connection. This architecture is simple and easy to implement, but it does not scale when the application needs to handle thousands of simultaneous connections.

https://www.nginx.com/blog/inside-nginx-how-we-designed-for-performance-scale/

How Does NGINX Work?

NGINX uses a predictable process model that is tuned to the available hardware resources:

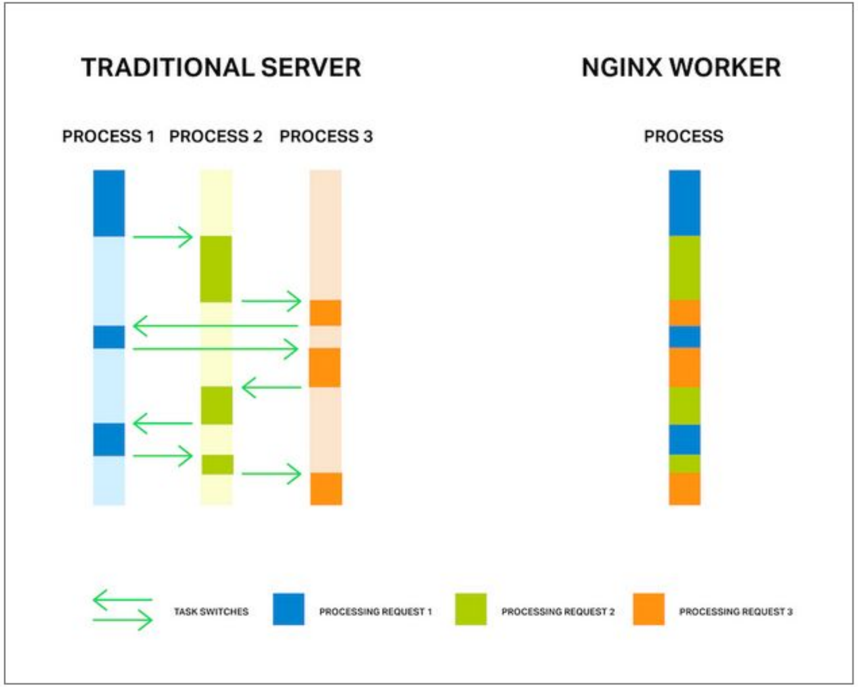
- The *master* process performs the privileged operations such as reading configuration and binding to ports, and then creates a small number of child processes (the next three types).
- The *cache loader* process runs at startup to load the disk-based cache into memory, and then exits. It is scheduled conservatively, so its resource demands are low.
- The *cache manager* process runs periodically and prunes entries from the disk caches to keep them within the configured sizes.
- The *worker* processes do all of the work! They handle network connections, read and write content to disk, and communicate with upstream servers.

The NGINX configuration recommended in most cases – running one worker process per CPU core – makes the most efficient use of hardware resources. You configure it by setting the `auto` parameter on the `worker_processes` directive:

```
worker_processes auto;
```

When an NGINX server is active, only the worker processes are busy. Each worker process handles multiple connections in a nonblocking fashion, reducing the number of context switches.

Each worker process is single-threaded and runs independently, grabbing new connections and processing them. The processes can communicate using shared memory for shared cache data, session persistence data, and other shared resources.



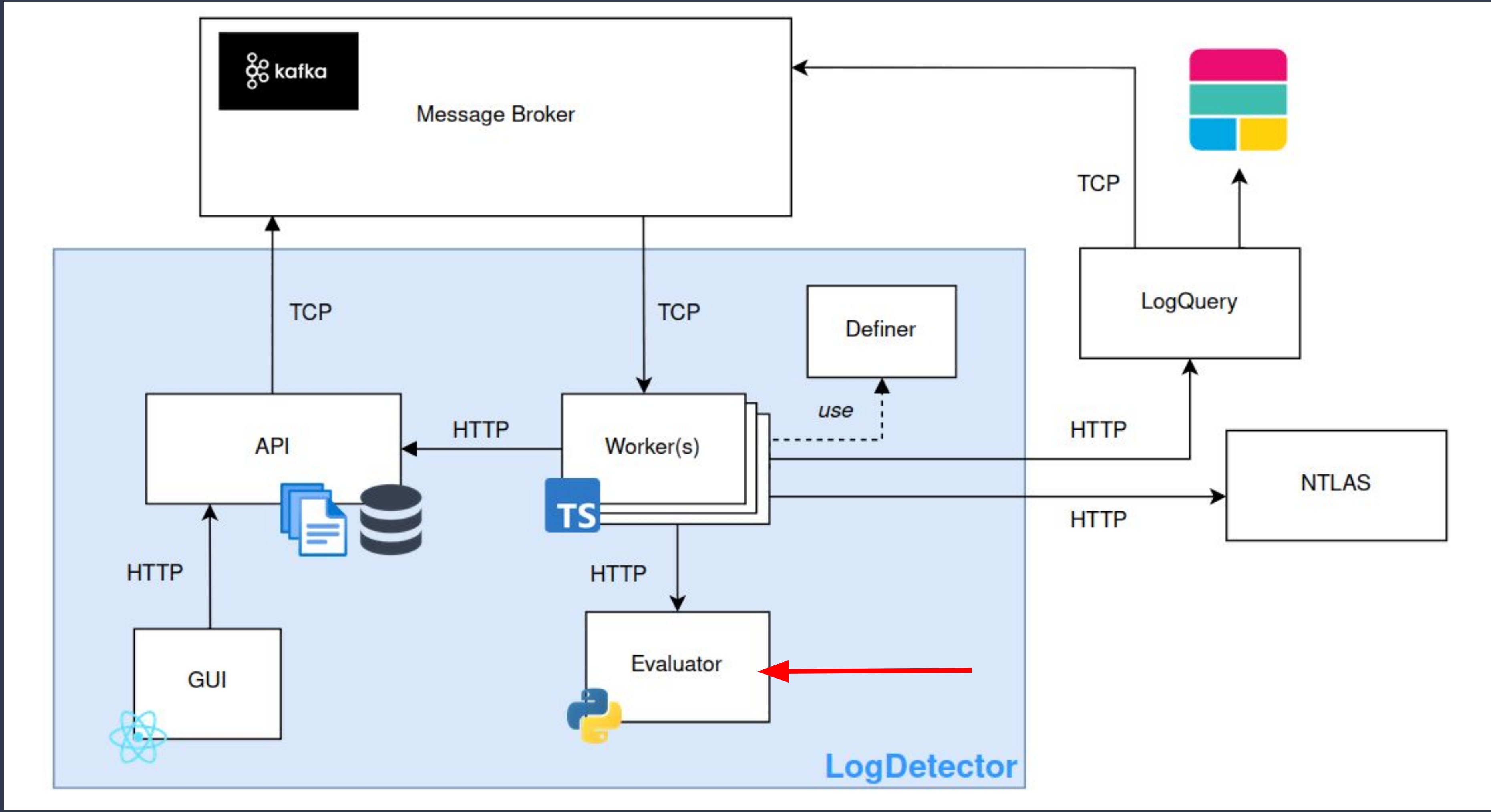
Each process consumes additional memory, and each switch between them consumes CPU cycles and trashes L-caches

https://www.nginx.com/blog/inside-nginx-how-we-designed-for-performance-scale/

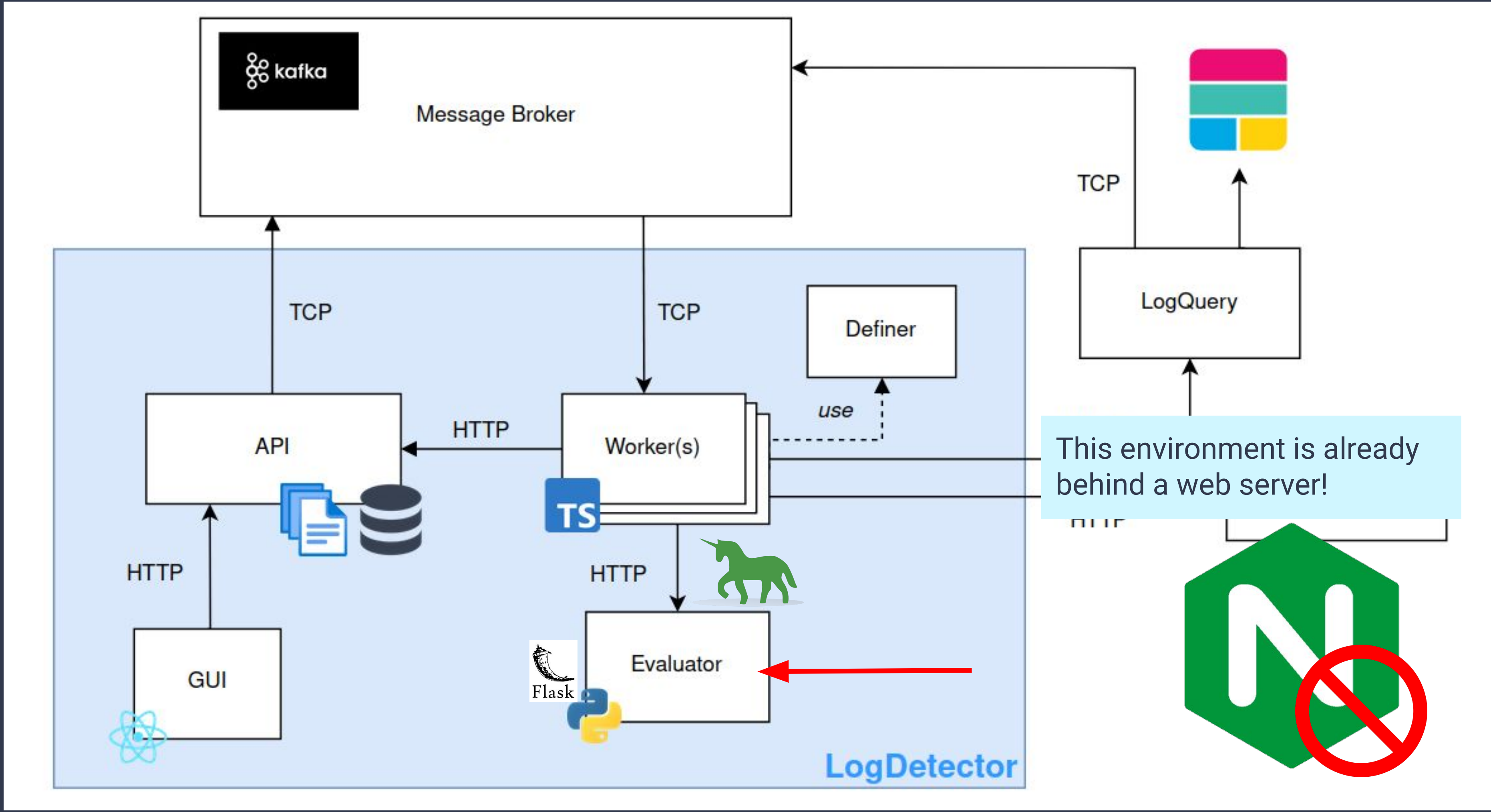
https://www.nginx.com/blog/thread-pools-boost-performance-9x/

Web Servers & Python Web apps 101

Now back to the project ...



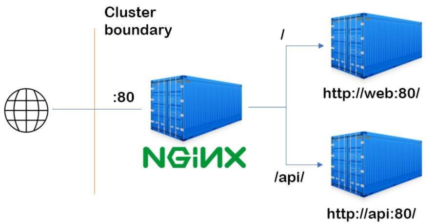
kafka
Message Broker
TCP
TCP
TCP
Definer
LogQuery
use
API
HTTP
Worker(s)
HTTP
NTLAS
TS
HTTP
HTTP
HTTP
GUI
Evaluator
LogDetector


kafka
Message Broker
TCP
TCP
TCP
Definer
LogQuery
use
HTTP
API
Worker(s)
TS
HTTP
HTTP
GUI
Evaluator
Flask
LogDetector
This environment is already behind a web server!

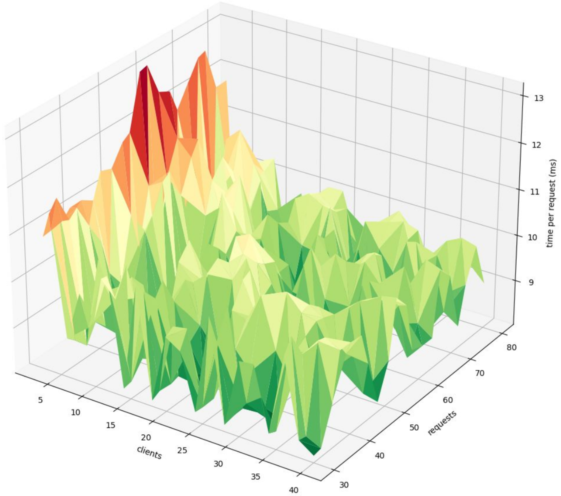
1. Mejoras a la implementación anterior

- La implementación anterior era buena, sin embargo el método no implementado no entrega resultados satisfactorios.

Total Logs: 459311
Elapsed: 71.6 min



- Processed 400 points.
- MIN: (32, 37, 8.12 ms)
- MAX: (3, 58, 13.164 ms)
- AVG: 9.505484999999995 ms

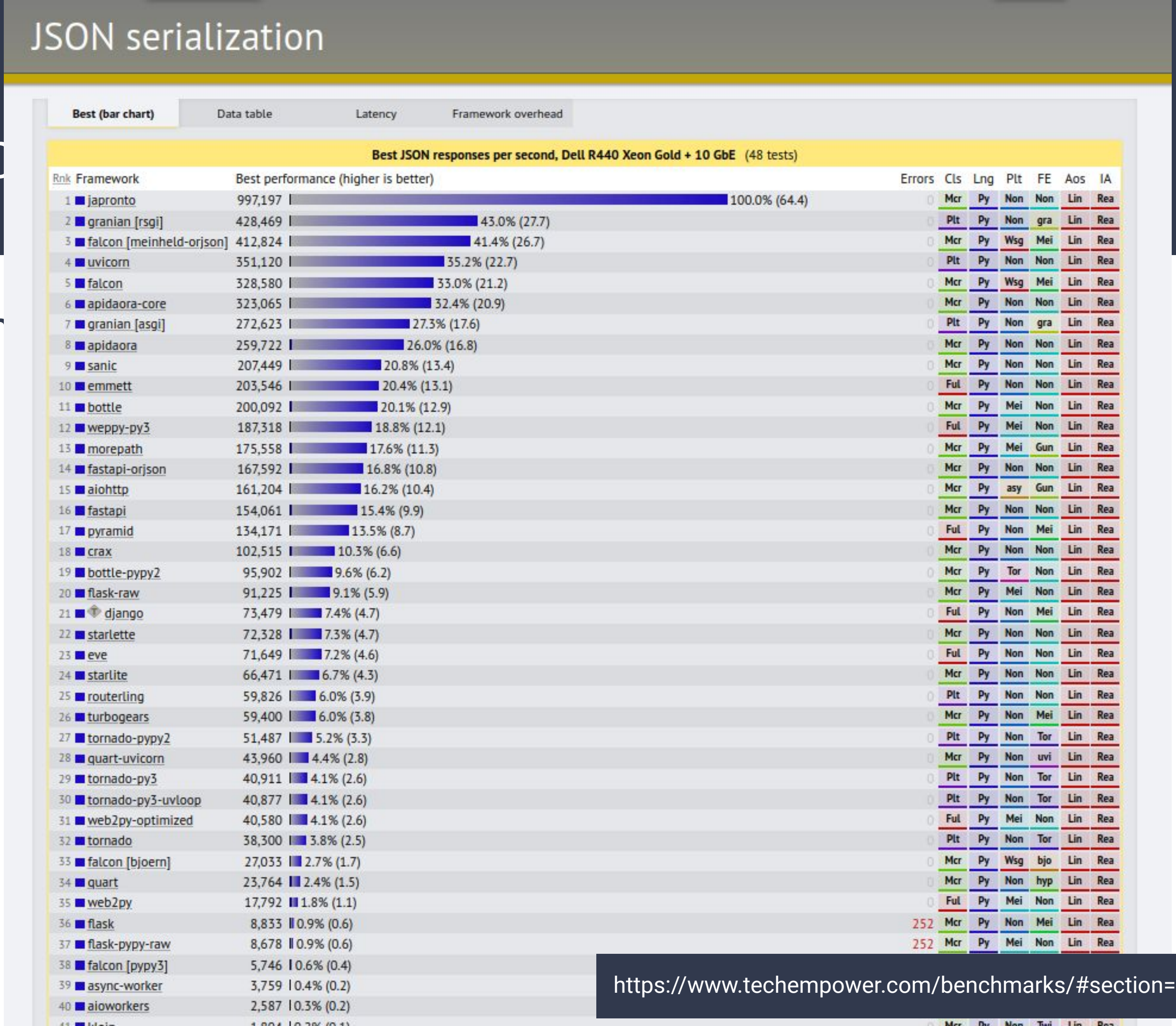


1. Mejoras a la implementación anterior

- Intentar con otros métodos:
    - Probar con Python web frameworks más rápidos.

1. Mejo

- Inter
  -



JSON serialization

Best (bar chart) | Data table | Latency | Framework overhead

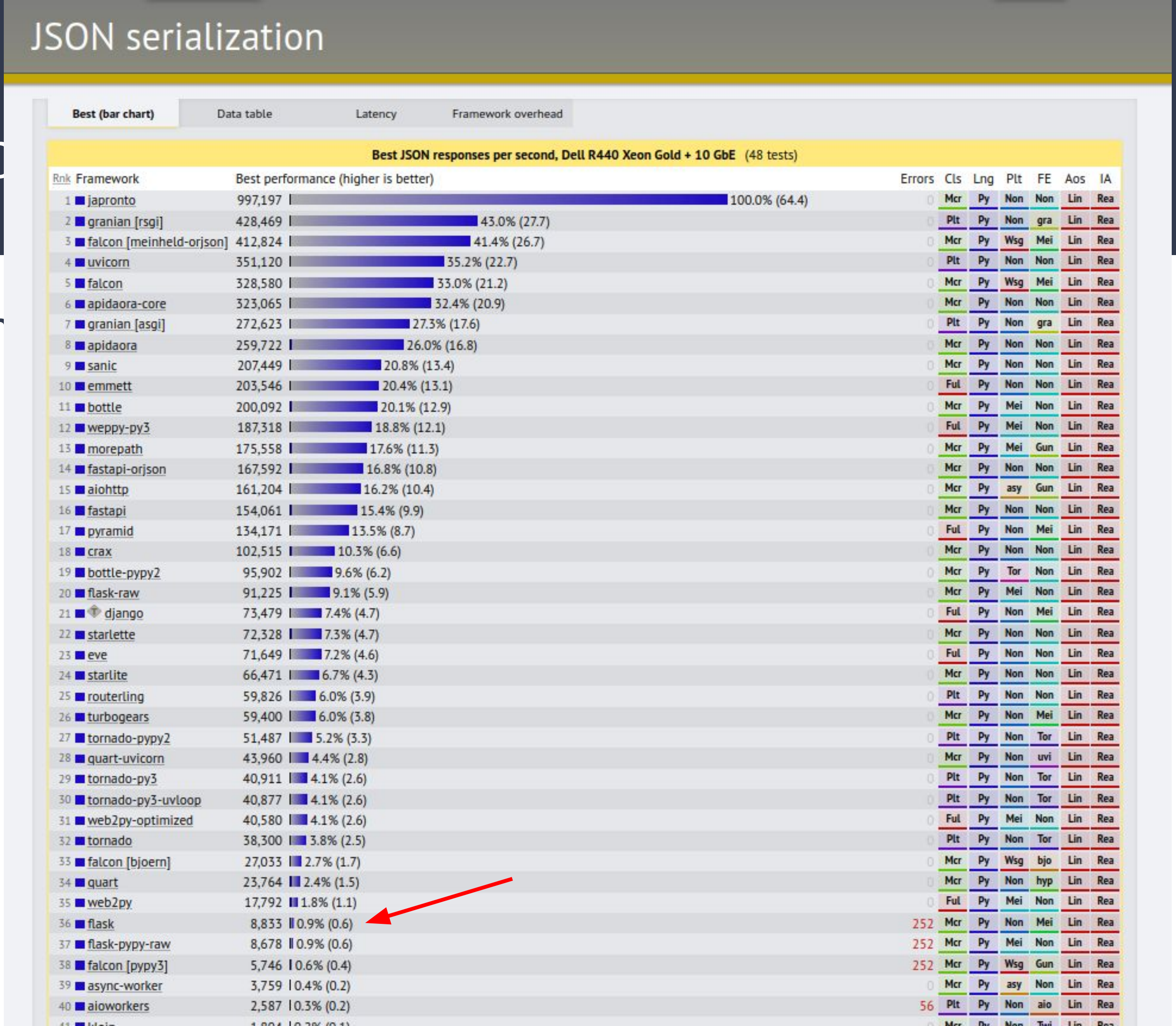
Best JSON responses per second, Dell R440 Xeon Gold + 10 GbE (48 tests)

Rnk	Framework	Best performance (higher is better)		Errors	Cls	Lng	Plt	FE	Aos	IA
1	japronto	997,197	100.0% (64.4)	0	Mcr	Py	Non	Non	Lin	Rea
2	granian [rsgi]	428,469	43.0% (27.7)	0	Plt	Py	Non	gra	Lin	Rea
3	falcon [meinheld-orjson]	412,824	41.4% (26.7)	0	Mcr	Py	Wsg	Mei	Lin	Rea
4	uvicorn	351,120	35.2% (22.7)	0	Plt	Py	Non	Non	Lin	Rea
5	falcon	328,580	33.0% (21.2)	0	Mcr	Py	Wsg	Mei	Lin	Rea
6	apidaora-core	323,065	32.4% (20.9)	0	Mcr	Py	Non	Non	Lin	Rea
7	granian [asgi]	272,623	27.3% (17.6)	0	Plt	Py	Non	gra	Lin	Rea
8	apidaora	259,722	26.0% (16.8)	0	Mcr	Py	Non	Non	Lin	Rea
9	sanic	207,449	20.8% (13.4)	0	Mcr	Py	Non	Non	Lin	Rea
10	emmett	203,546	20.4% (13.1)	0	Ful	Py	Non	Non	Lin	Rea
11	bottle	200,092	20.1% (12.9)	0	Mcr	Py	Mei	Non	Lin	Rea
12	weppy-py3	187,318	18.8% (12.1)	0	Ful	Py	Mei	Non	Lin	Rea
13	morepath	175,558	17.6% (11.3)	0	Mcr	Py	Mei	Gun	Lin	Rea
14	fastapi-orjson	167,592	16.8% (10.8)	0	Mcr	Py	Non	Non	Lin	Rea
15	aiohttp	161,204	16.2% (10.4)	0	Mcr	Py	asy	Gun	Lin	Rea
16	fastapi	154,061	15.4% (9.9)	0	Mcr	Py	Non	Non	Lin	Rea
17	pyramid	134,171	13.5% (8.7)	0	Ful	Py	Non	Mei	Lin	Rea
18	crax	102,515	10.3% (6.6)	0	Mcr	Py	Non	Non	Lin	Rea
19	bottle-pypy2	95,902	9.6% (6.2)	0	Mcr	Py	Tor	Non	Lin	Rea
20	flask-raw	91,225	9.1% (5.9)	0	Mcr	Py	Mei	Non	Lin	Rea
21	django	73,479	7.4% (4.7)	0	Ful	Py	Non	Mei	Lin	Rea
22	starlette	72,328	7.3% (4.7)	0	Mcr	Py	Non	Non	Lin	Rea
23	eve	71,649	7.2% (4.6)	0	Ful	Py	Non	Non	Lin	Rea
24	starlite	66,471	6.7% (4.3)	0	Mcr	Py	Non	Non	Lin	Rea
25	routerling	59,826	6.0% (3.9)	0	Plt	Py	Non	Non	Lin	Rea
26	turbogears	59,400	6.0% (3.8)	0	Mcr	Py	Non	Mei	Lin	Rea
27	tornado-pypy2	51,487	5.2% (3.3)	0	Plt	Py	Non	Tor	Lin	Rea
28	quart-uvicorn	43,960	4.4% (2.8)	0	Mcr	Py	Non	uvi	Lin	Rea
29	tornado-py3	40,911	4.1% (2.6)	0	Plt	Py	Non	Tor	Lin	Rea
30	tornado-py3-uvloop	40,877	4.1% (2.6)	0	Plt	Py	Non	Tor	Lin	Rea
31	web2py-optimized	40,580	4.1% (2.6)	0	Ful	Py	Mei	Non	Lin	Rea
32	tornado	38,300	3.8% (2.5)	0	Plt	Py	Non	Tor	Lin	Rea
33	falcon [bjoern]	27,033	2.7% (1.7)	0	Mcr	Py	Wsg	bjo	Lin	Rea
34	quart	23,764	2.4% (1.5)	0	Mcr	Py	Non	hyp	Lin	Rea
35	web2py	17,792	1.8% (1.1)	0	Ful	Py	Mei	Non	Lin	Rea
36	flask	8,833	0.9% (0.6)	252	Mcr	Py	Non	Mei	Lin	Rea
37	flask-pypy-raw	8,678	0.9% (0.6)	252	Mcr	Py	Mei	Non	Lin	Rea
38	falcon [pypy3]	5,746	0.6% (0.4)							
39	async-worker	3,759	0.4% (0.2)							
40	aioworkers	2,587	0.3% (0.2)							

https://www.techempower.com/benchmarks/#section=data-r21&test=json&l=zijzen-6bj

1. Mejo

- Inter
  -



JSON serialization

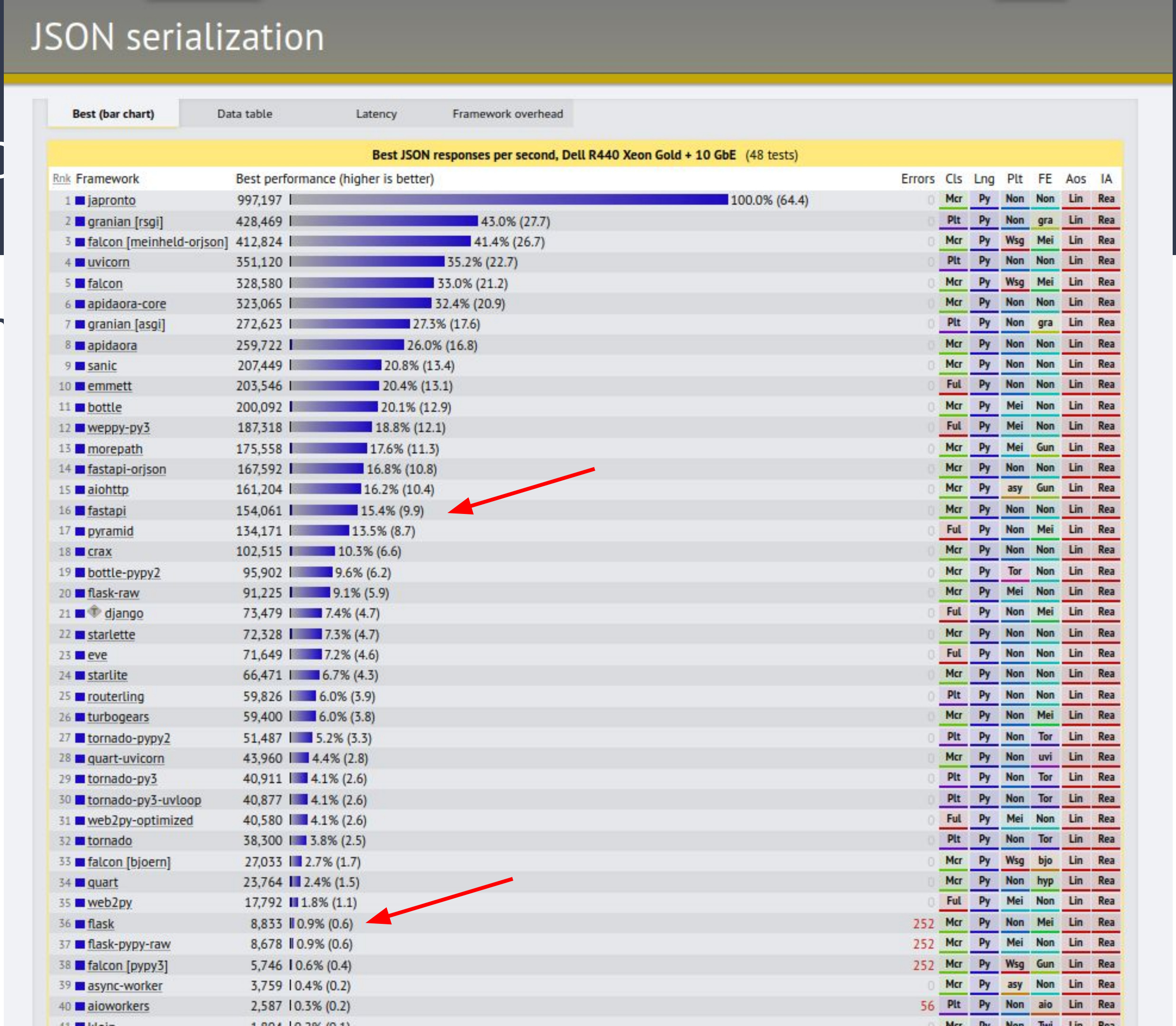
Best (bar chart) | Data table | Latency | Framework overhead

Best JSON responses per second, Dell R440 Xeon Gold + 10 GbE (48 tests)

Rnk	Framework	Best performance (higher is better)		Errors	Cls	Lng	Plt	FE	Aos	IA
1	japronto	997,197	100.0% (64.4)	0	Mcr	Py	Non	Non	Lin	Rea
2	granian [rsgi]	428,469	43.0% (27.7)	0	Plt	Py	Non	gra	Lin	Rea
3	falcon [meinheld-orjson]	412,824	41.4% (26.7)	0	Mcr	Py	Wsg	Mei	Lin	Rea
4	uvicorn	351,120	35.2% (22.7)	0	Plt	Py	Non	Non	Lin	Rea
5	falcon	328,580	33.0% (21.2)	0	Mcr	Py	Wsg	Mei	Lin	Rea
6	apidaora-core	323,065	32.4% (20.9)	0	Mcr	Py	Non	Non	Lin	Rea
7	granian [asgi]	272,623	27.3% (17.6)	0	Plt	Py	Non	gra	Lin	Rea
8	apidaora	259,722	26.0% (16.8)	0	Mcr	Py	Non	Non	Lin	Rea
9	sanic	207,449	20.8% (13.4)	0	Mcr	Py	Non	Non	Lin	Rea
10	emmett	203,546	20.4% (13.1)	0	Ful	Py	Non	Non	Lin	Rea
11	bottle	200,092	20.1% (12.9)	0	Mcr	Py	Mei	Non	Lin	Rea
12	weppy-py3	187,318	18.8% (12.1)	0	Ful	Py	Mei	Non	Lin	Rea
13	morepath	175,558	17.6% (11.3)	0	Mcr	Py	Mei	Gun	Lin	Rea
14	fastapi-orjson	167,592	16.8% (10.8)	0	Mcr	Py	Non	Non	Lin	Rea
15	aiohttp	161,204	16.2% (10.4)	0	Mcr	Py	asy	Gun	Lin	Rea
16	fastapi	154,061	15.4% (9.9)	0	Mcr	Py	Non	Non	Lin	Rea
17	pyramid	134,171	13.5% (8.7)	0	Ful	Py	Non	Mei	Lin	Rea
18	crax	102,515	10.3% (6.6)	0	Mcr	Py	Non	Non	Lin	Rea
19	bottle-pypy2	95,902	9.6% (6.2)	0	Mcr	Py	Tor	Non	Lin	Rea
20	flask-raw	91,225	9.1% (5.9)	0	Mcr	Py	Mei	Non	Lin	Rea
21	django	73,479	7.4% (4.7)	0	Ful	Py	Non	Mei	Lin	Rea
22	starlette	72,328	7.3% (4.7)	0	Mcr	Py	Non	Non	Lin	Rea
23	eve	71,649	7.2% (4.6)	0	Ful	Py	Non	Non	Lin	Rea
24	starlite	66,471	6.7% (4.3)	0	Mcr	Py	Non	Non	Lin	Rea
25	routerling	59,826	6.0% (3.9)	0	Plt	Py	Non	Non	Lin	Rea
26	turbogears	59,400	6.0% (3.8)	0	Mcr	Py	Non	Mei	Lin	Rea
27	tornado-pypy2	51,487	5.2% (3.3)	0	Plt	Py	Non	Tor	Lin	Rea
28	quart-uvicorn	43,960	4.4% (2.8)	0	Mcr	Py	Non	uvi	Lin	Rea
29	tornado-py3	40,911	4.1% (2.6)	0	Plt	Py	Non	Tor	Lin	Rea
30	tornado-py3-uvloop	40,877	4.1% (2.6)	0	Plt	Py	Non	Tor	Lin	Rea
31	web2py-optimized	40,580	4.1% (2.6)	0	Ful	Py	Mei	Non	Lin	Rea
32	tornado	38,300	3.8% (2.5)	0	Plt	Py	Non	Tor	Lin	Rea
33	falcon [bjoern]	27,033	2.7% (1.7)	0	Mcr	Py	Wsg	bjo	Lin	Rea
34	quart	23,764	2.4% (1.5)	0	Mcr	Py	Non	hyp	Lin	Rea
35	web2py	17,792	1.8% (1.1)	0	Ful	Py	Mei	Non	Lin	Rea
36	flask	8,833	0.9% (0.6)	252	Mcr	Py	Non	Mei	Lin	Rea
37	flask-pypy-raw	8,678	0.9% (0.6)	252	Mcr	Py	Mei	Non	Lin	Rea
38	falcon [pypy3]	5,746	0.6% (0.4)	252	Mcr	Py	Wsg	Gun	Lin	Rea
39	async-worker	3,759	0.4% (0.2)	0	Mcr	Py	asy	Non	Lin	Rea
40	aioworkers	2,587	0.3% (0.2)	56	Plt	Py	Non	aio	Lin	Rea

1. Mejo

- Inter
  -



JSON serialization

Best (bar chart) | Data table | Latency | Framework overhead

Best JSON responses per second, Dell R440 Xeon Gold + 10 GbE (48 tests)

Rnk	Framework	Best performance (higher is better)		Errors	Cls	Lng	Plt	FE	Aos	IA
1	japronto	997,197	100.0% (64.4)	0	Mcr	Py	Non	Non	Lin	Rea
2	granian [rsgi]	428,469	43.0% (27.7)	0	Plt	Py	Non	gra	Lin	Rea
3	falcon [meinheld-orjson]	412,824	41.4% (26.7)	0	Mcr	Py	Wsg	Mei	Lin	Rea
4	uvicorn	351,120	35.2% (22.7)	0	Plt	Py	Non	Non	Lin	Rea
5	falcon	328,580	33.0% (21.2)	0	Mcr	Py	Wsg	Mei	Lin	Rea
6	apidaora-core	323,065	32.4% (20.9)	0	Mcr	Py	Non	Non	Lin	Rea
7	granian [asgi]	272,623	27.3% (17.6)	0	Plt	Py	Non	gra	Lin	Rea
8	apidaora	259,722	26.0% (16.8)	0	Mcr	Py	Non	Non	Lin	Rea
9	sanic	207,449	20.8% (13.4)	0	Mcr	Py	Non	Non	Lin	Rea
10	emmett	203,546	20.4% (13.1)	0	Ful	Py	Non	Non	Lin	Rea
11	bottle	200,092	20.1% (12.9)	0	Mcr	Py	Mei	Non	Lin	Rea
12	weppy-py3	187,318	18.8% (12.1)	0	Ful	Py	Mei	Non	Lin	Rea
13	morepath	175,558	17.6% (11.3)	0	Mcr	Py	Mei	Gun	Lin	Rea
14	fastapi-orjson	167,592	16.8% (10.8)	0	Mcr	Py	Non	Non	Lin	Rea
15	aiohttp	161,204	16.2% (10.4)	0	Mcr	Py	asy	Gun	Lin	Rea
16	fastapi	154,061	15.4% (9.9)	0	Mcr	Py	Non	Non	Lin	Rea
17	pyramid	134,171	13.5% (8.7)	0	Ful	Py	Non	Mei	Lin	Rea
18	crax	102,515	10.3% (6.6)	0	Mcr	Py	Non	Non	Lin	Rea
19	bottle-pypy2	95,902	9.6% (6.2)	0	Mcr	Py	Tor	Non	Lin	Rea
20	flask-raw	91,225	9.1% (5.9)	0	Mcr	Py	Mei	Non	Lin	Rea
21	django	73,479	7.4% (4.7)	0	Ful	Py	Non	Mei	Lin	Rea
22	starlette	72,328	7.3% (4.7)	0	Mcr	Py	Non	Non	Lin	Rea
23	eve	71,649	7.2% (4.6)	0	Ful	Py	Non	Non	Lin	Rea
24	starlite	66,471	6.7% (4.3)	0	Mcr	Py	Non	Non	Lin	Rea
25	routerling	59,826	6.0% (3.9)	0	Plt	Py	Non	Non	Lin	Rea
26	turbogears	59,400	6.0% (3.8)	0	Mcr	Py	Non	Mei	Lin	Rea
27	tornado-pypy2	51,487	5.2% (3.3)	0	Plt	Py	Non	Tor	Lin	Rea
28	quart-uvicorn	43,960	4.4% (2.8)	0	Mcr	Py	Non	uvi	Lin	Rea
29	tornado-py3	40,911	4.1% (2.6)	0	Plt	Py	Non	Tor	Lin	Rea
30	tornado-py3-uvloop	40,877	4.1% (2.6)	0	Plt	Py	Non	Tor	Lin	Rea
31	web2py-optimized	40,580	4.1% (2.6)	0	Ful	Py	Mei	Non	Lin	Rea
32	tornado	38,300	3.8% (2.5)	0	Plt	Py	Non	Tor	Lin	Rea
33	falcon [bjoern]	27,033	2.7% (1.7)	0	Mcr	Py	Wsg	bjo	Lin	Rea
34	quart	23,764	2.4% (1.5)	0	Mcr	Py	Non	hyp	Lin	Rea
35	web2py	17,792	1.8% (1.1)	0	Ful	Py	Mei	Non	Lin	Rea
36	flask	8,833	0.9% (0.6)	252	Mcr	Py	Non	Mei	Lin	Rea
37	flask-pypy-raw	8,678	0.9% (0.6)	252	Mcr	Py	Mei	Non	Lin	Rea
38	falcon [pypy3]	5,746	0.6% (0.4)	252	Mcr	Py	Wsg	Gun	Lin	Rea
39	async-worker	3,759	0.4% (0.2)	0	Mcr	Py	asy	Non	Lin	Rea
40	aioworkers	2,587	0.3% (0.2)	56	Plt	Py	Non	aio	Lin	Rea

1. Mejo

- Inter
    -

JSON serialization

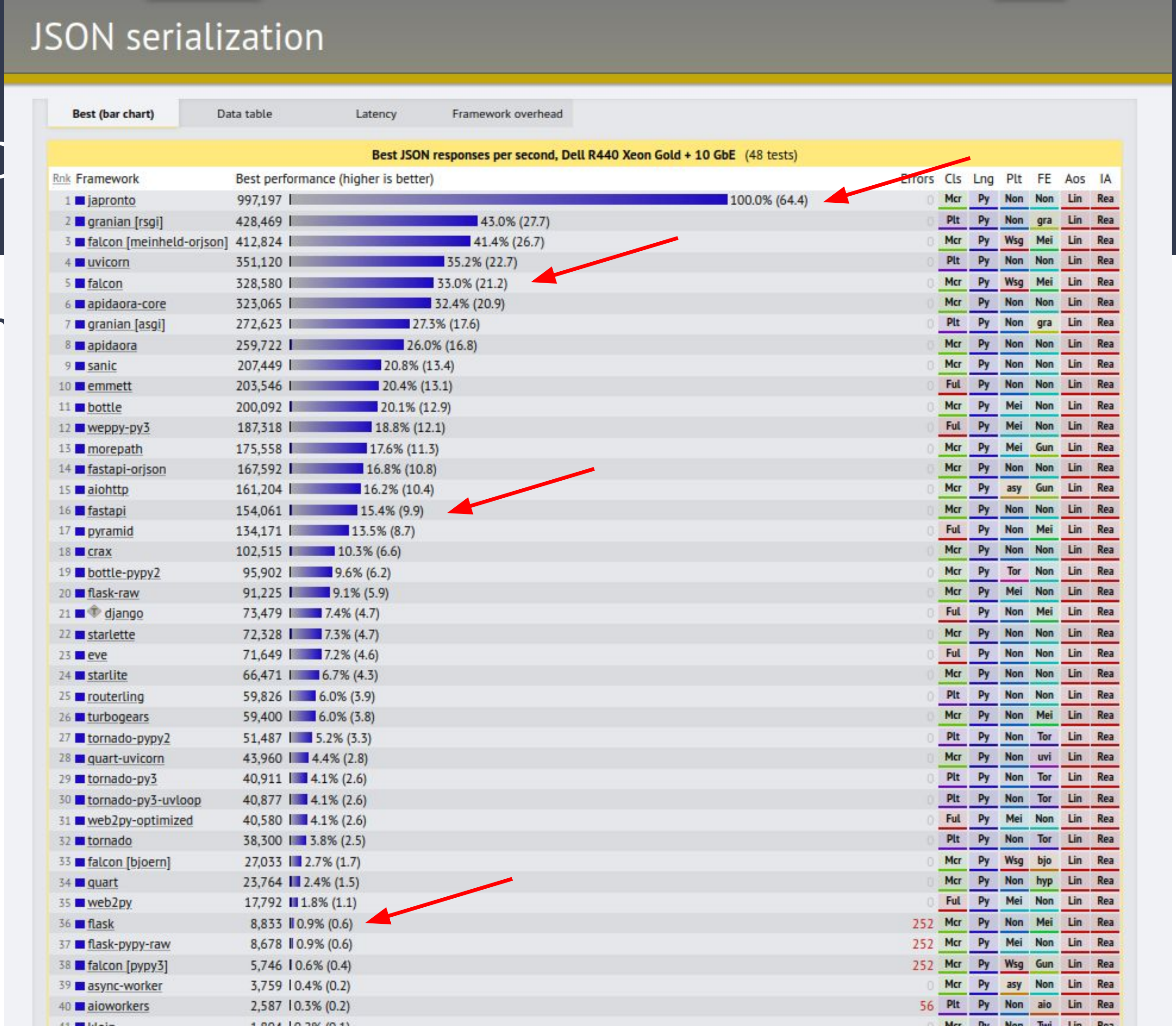
Best (bar chart) | Data table | Latency | Framework overhead

Best JSON responses per second, Dell R440 Xeon Gold + 10 GbE (48 tests)

Rnk	Framework	Best performance (higher is better)		Errors	Cls	Lng	Plt	FE	Aos	IA
1	japronto	997,197	100.0% (64.4)	0	Mcr	Py	Non	Non	Lin	Rea
2	granian [rsgi]	428,469	43.0% (27.7)	0	Plt	Py	Non	gra	Lin	Rea
3	falcon [meinheld-orjson]	412,824	41.4% (26.7)	0	Mcr	Py	Wsg	Mei	Lin	Rea
4	uvicorn	351,120	35.2% (22.7)	0	Plt	Py	Non	Non	Lin	Rea
5	falcon	328,580	33.0% (21.2)	0	Mcr	Py	Wsg	Mei	Lin	Rea
6	apidaora-core	323,065	32.4% (20.9)	0	Mcr	Py	Non	Non	Lin	Rea
7	granian [asgi]	272,623	27.3% (17.6)	0	Plt	Py	Non	gra	Lin	Rea
8	apidaora	259,722	26.0% (16.8)	0	Mcr	Py	Non	Non	Lin	Rea
9	sanic	207,449	20.8% (13.4)	0	Mcr	Py	Non	Non	Lin	Rea
10	emmett	203,546	20.4% (13.1)	0	Ful	Py	Non	Non	Lin	Rea
11	bottle	200,092	20.1% (12.9)	0	Mcr	Py	Mei	Non	Lin	Rea
12	weppy-py3	187,318	18.8% (12.1)	0	Ful	Py	Mei	Non	Lin	Rea
13	morepath	175,558	17.6% (11.3)	0	Mcr	Py	Mei	Gun	Lin	Rea
14	fastapi-orjson	167,592	16.8% (10.8)	0	Mcr	Py	Non	Non	Lin	Rea
15	aiohttp	161,204	16.2% (10.4)	0	Mcr	Py	asy	Gun	Lin	Rea
16	fastapi	154,061	15.4% (9.9)	0	Mcr	Py	Non	Non	Lin	Rea
17	pyramid	134,171	13.5% (8.7)	0	Ful	Py	Non	Mei	Lin	Rea
18	crax	102,515	10.3% (6.6)	0	Mcr	Py	Non	Non	Lin	Rea
19	bottle-pypy2	95,902	9.6% (6.2)	0	Mcr	Py	Tor	Non	Lin	Rea
20	flask-raw	91,225	9.1% (5.9)	0	Mcr	Py	Mei	Non	Lin	Rea
21	django	73,479	7.4% (4.7)	0	Ful	Py	Non	Mei	Lin	Rea
22	starlette	72,328	7.3% (4.7)	0	Mcr	Py	Non	Non	Lin	Rea
23	eve	71,649	7.2% (4.6)	0	Ful	Py	Non	Non	Lin	Rea
24	starlite	66,471	6.7% (4.3)	0	Mcr	Py	Non	Non	Lin	Rea
25	routerling	59,826	6.0% (3.9)	0	Plt	Py	Non	Non	Lin	Rea
26	turbogears	59,400	6.0% (3.8)	0	Mcr	Py	Non	Mei	Lin	Rea
27	tornado-pypy2	51,487	5.2% (3.3)	0	Plt	Py	Non	Tor	Lin	Rea
28	quart-uvicorn	43,960	4.4% (2.8)	0	Mcr	Py	Non	uvi	Lin	Rea
29	tornado-py3	40,911	4.1% (2.6)	0	Plt	Py	Non	Tor	Lin	Rea
30	tornado-py3-uvloop	40,877	4.1% (2.6)	0	Plt	Py	Non	Tor	Lin	Rea
31	web2py-optimized	40,580	4.1% (2.6)	0	Ful	Py	Mei	Non	Lin	Rea
32	tornado	38,300	3.8% (2.5)	0	Plt	Py	Non	Tor	Lin	Rea
33	falcon [bjoern]	27,033	2.7% (1.7)	0	Mcr	Py	Wsg	bjo	Lin	Rea
34	quart	23,764	2.4% (1.5)	0	Mcr	Py	Non	hyp	Lin	Rea
35	web2py	17,792	1.8% (1.1)	0	Ful	Py	Mei	Non	Lin	Rea
36	flask	8,833	0.9% (0.6)	252	Mcr	Py	Non	Mei	Lin	Rea
37	flask-pypy-raw	8,678	0.9% (0.6)	252	Mcr	Py	Mei	Non	Lin	Rea
38	falcon [pypy3]	5,746	0.6% (0.4)	252	Mcr	Py	Wsg	Gun	Lin	Rea
39	async-worker	3,759	0.4% (0.2)	0	Mcr	Py	asy	Non	Lin	Rea
40	aioworkers	2,587	0.3% (0.2)	56	Plt	Py	Non	aio	Lin	Rea

1. Mejo

- Inter
  -



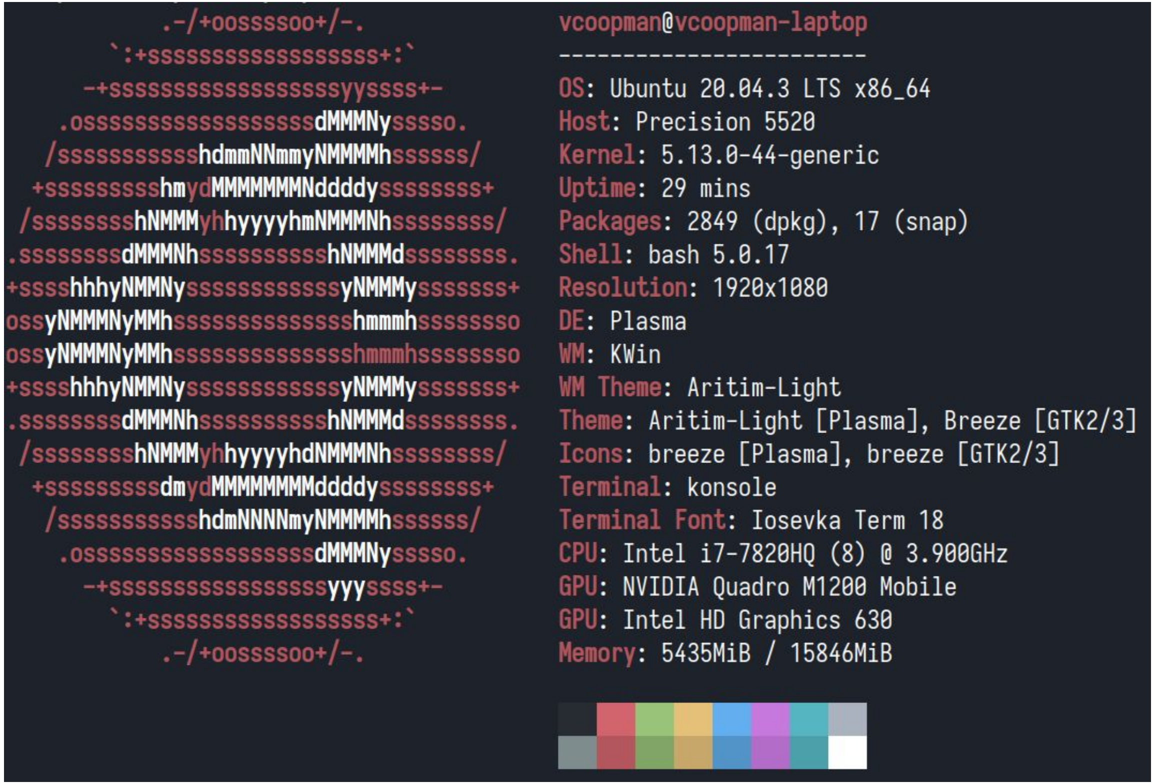
JSON serialization

Best (bar chart) | Data table | Latency | Framework overhead

Best JSON responses per second, Dell R440 Xeon Gold + 10 GbE (48 tests)

Rnk	Framework	Best performance (higher is better)		Errors	Cls	Lng	Plt	FE	Aos	IA
1	japronto	997,197	100.0% (64.4)	0	Mcr	Py	Non	Non	Lin	Rea
2	granian [rsgi]	428,469	43.0% (27.7)	0	Plt	Py	Non	gra	Lin	Rea
3	falcon [meinheld-orjson]	412,824	41.4% (26.7)	0	Mcr	Py	Wsg	Mei	Lin	Rea
4	uvicorn	351,120	35.2% (22.7)	0	Plt	Py	Non	Non	Lin	Rea
5	falcon	328,580	33.0% (21.2)	0	Mcr	Py	Wsg	Mei	Lin	Rea
6	apidaora-core	323,065	32.4% (20.9)	0	Mcr	Py	Non	Non	Lin	Rea
7	granian [asgi]	272,623	27.3% (17.6)	0	Plt	Py	Non	gra	Lin	Rea
8	apidaora	259,722	26.0% (16.8)	0	Mcr	Py	Non	Non	Lin	Rea
9	sanic	207,449	20.8% (13.4)	0	Mcr	Py	Non	Non	Lin	Rea
10	emmett	203,546	20.4% (13.1)	0	Ful	Py	Non	Non	Lin	Rea
11	bottle	200,092	20.1% (12.9)	0	Mcr	Py	Mei	Non	Lin	Rea
12	weppy-py3	187,318	18.8% (12.1)	0	Ful	Py	Mei	Non	Lin	Rea
13	morepath	175,558	17.6% (11.3)	0	Mcr	Py	Mei	Gun	Lin	Rea
14	fastapi-orjson	167,592	16.8% (10.8)	0	Mcr	Py	Non	Non	Lin	Rea
15	aiohttp	161,204	16.2% (10.4)	0	Mcr	Py	asy	Gun	Lin	Rea
16	fastapi	154,061	15.4% (9.9)	0	Mcr	Py	Non	Non	Lin	Rea
17	pyramid	134,171	13.5% (8.7)	0	Ful	Py	Non	Mei	Lin	Rea
18	crax	102,515	10.3% (6.6)	0	Mcr	Py	Non	Non	Lin	Rea
19	bottle-pypy2	95,902	9.6% (6.2)	0	Mcr	Py	Tor	Non	Lin	Rea
20	flask-raw	91,225	9.1% (5.9)	0	Mcr	Py	Mei	Non	Lin	Rea
21	django	73,479	7.4% (4.7)	0	Ful	Py	Non	Mei	Lin	Rea
22	starlette	72,328	7.3% (4.7)	0	Mcr	Py	Non	Non	Lin	Rea
23	eve	71,649	7.2% (4.6)	0	Ful	Py	Non	Non	Lin	Rea
24	starlite	66,471	6.7% (4.3)	0	Mcr	Py	Non	Non	Lin	Rea
25	routerling	59,826	6.0% (3.9)	0	Plt	Py	Non	Non	Lin	Rea
26	turbogears	59,400	6.0% (3.8)	0	Mcr	Py	Non	Mei	Lin	Rea
27	tornado-pypy2	51,487	5.2% (3.3)	0	Plt	Py	Non	Tor	Lin	Rea
28	quart-uvicorn	43,960	4.4% (2.8)	0	Mcr	Py	Non	uvi	Lin	Rea
29	tornado-py3	40,911	4.1% (2.6)	0	Plt	Py	Non	Tor	Lin	Rea
30	tornado-py3-uvloop	40,877	4.1% (2.6)	0	Plt	Py	Non	Tor	Lin	Rea
31	web2py-optimized	40,580	4.1% (2.6)	0	Ful	Py	Mei	Non	Lin	Rea
32	tornado	38,300	3.8% (2.5)	0	Plt	Py	Non	Tor	Lin	Rea
33	falcon [bjoern]	27,033	2.7% (1.7)	0	Mcr	Py	Wsg	bjo	Lin	Rea
34	quart	23,764	2.4% (1.5)	0	Mcr	Py	Non	hyp	Lin	Rea
35	web2py	17,792	1.8% (1.1)	0	Ful	Py	Mei	Non	Lin	Rea
36	flask	8,833	0.9% (0.6)	252	Mcr	Py	Non	Mei	Lin	Rea
37	flask-pypy-raw	8,678	0.9% (0.6)	252	Mcr	Py	Mei	Non	Lin	Rea
38	falcon [pypy3]	5,746	0.6% (0.4)	252	Mcr	Py	Wsg	Gun	Lin	Rea
39	async-worker	3,759	0.4% (0.2)	0	Mcr	Py	asy	Non	Lin	Rea
40	aioworkers	2,587	0.3% (0.2)	56	Plt	Py	Non	aio	Lin	Rea

1. Mejoras a la implementación anterior



1.1 Probar con Python web frameworks más rápidos.

- Test consiste en:
  - Cantidad de peticiones fija: 1000 HTTP request.
  - Variar cantidad de clientes concurrentes.
  - Aplicar test a los 4 frameworks seleccionados (Hosted in Docker).
  - 1 WSGI worker.

JAPRONTO

Flask

Falcon

proyecto : pipenv — Konsole <2>

```python
from flask import Flask, request

from eval_func import eval_func # local module

DEBUG = True
PORT = 5000

app = Flask(__name__)

@app.route("/eval", methods = ['POST'])
def evaluate():
    payload = request.json['payload']
    r = eval_func(payload)
    return r

if __name__ == '__main__':
    app.run(debug=DEBUG, host="0.0.0.0", port=PORT)
```

app_flask.py 11% 2:0

```python
from japronto import Application

from eval_func import eval_func # local module

DEBUG = True
PORT = 8000

def evaluate(request):
    payload = request.json['payload']
    r = eval_func(payload)
    return request.Response(json=r)


app = Application()
app.router.add_route("/eval", evaluate)

if __name__ == '__main__':
    app.run(host="0.0.0.0", port=PORT, debug=DEBUG)
```

NORMAL app_japronto.py unix | utf-8 | python 5% 1:1

```python
from fastapi import FastAPI

from eval_func import eval_func # local module

DEBUG = True
PORT = 5000

app = FastAPI()

@app.post("/eval")
def evaluate(body: Dict[Any, Any]):
    payload = body['payload']
    r = eval_func(payload)
    return r

if __name__ == "__main__":
    uvicorn.run(app, host="0.0.0.0", port=PORT)
```

app_fastapi.py 21% 4:0

```python
import falcon

from eval_func import eval_func # local module

DEBUG = True
PORT = 5000

class EvalResource:

    def on_post(self, req, resp):
        payload = req.media['payload']
        r = eval_func(payload)
        req.media['payload'] = r

app = falcon.App()
app.add_route('/eval', EvalResource())

if __name__ == '__main__':
    with make_server('', PORT, app) as httpd:
        print(f'Serving on port { PORT }')
```

app_falcon.py 4% 1:1

proyecto : pipenv — Konsole <2>

```dockerfile
FROM python:3.8-buster

WORKDIR /app

RUN pip install https://github.com/squeaky-pl/japronto/archive/master.zip

COPY . .

EXPOSE 8000
CMD [ "python3", "app_japronto.py" ]
```

Dockerfile_japronto 10% 1:1

```dockerfile
FROM python:3.8-buster

WORKDIR /app

RUN pip install flask gunicorn

COPY . .

EXPOSE 5000
CMD [ "gunicorn", "app_flask:app", "-b 0.0.0.0:5000", "--workers", "1", "--access-logfile", "/dev/stdout" ]
```

NORMAL Dockerfile_flask unix | utf-8 | dockerfile 80% 8:0

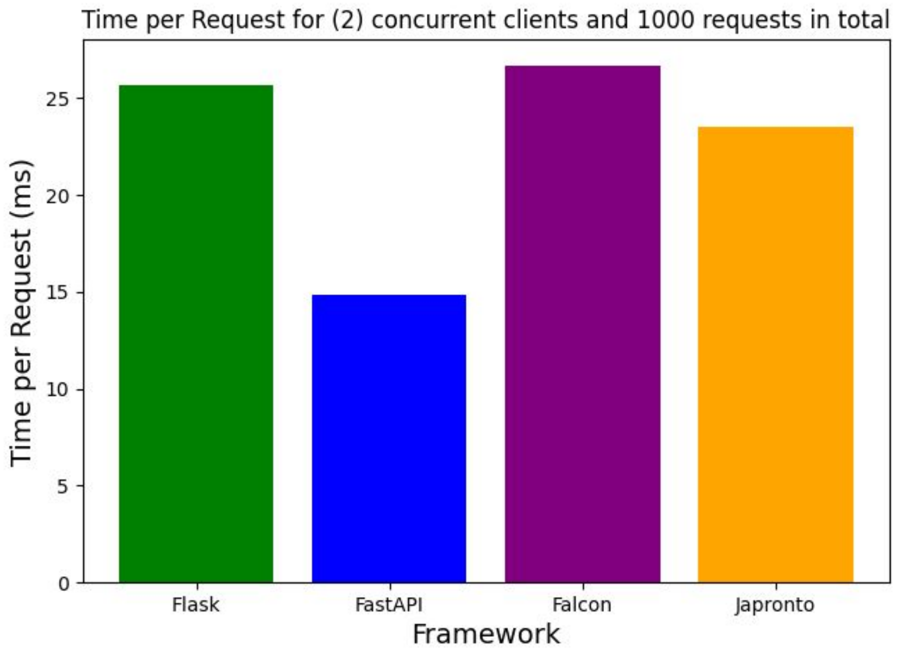
```dockerfile

WORKDIR /app

RUN pip install fastapi uvicorn gunicorn

COPY . .

EXPOSE 7000
CMD [ "gunicorn", "app_fastapi:app", "-b 0.0.0.0:7000", "--worker-class", "uvicorn.workers.UvicornWorker", "--workers", "1", "--access-logfile", "/dev/stdout" ]
```

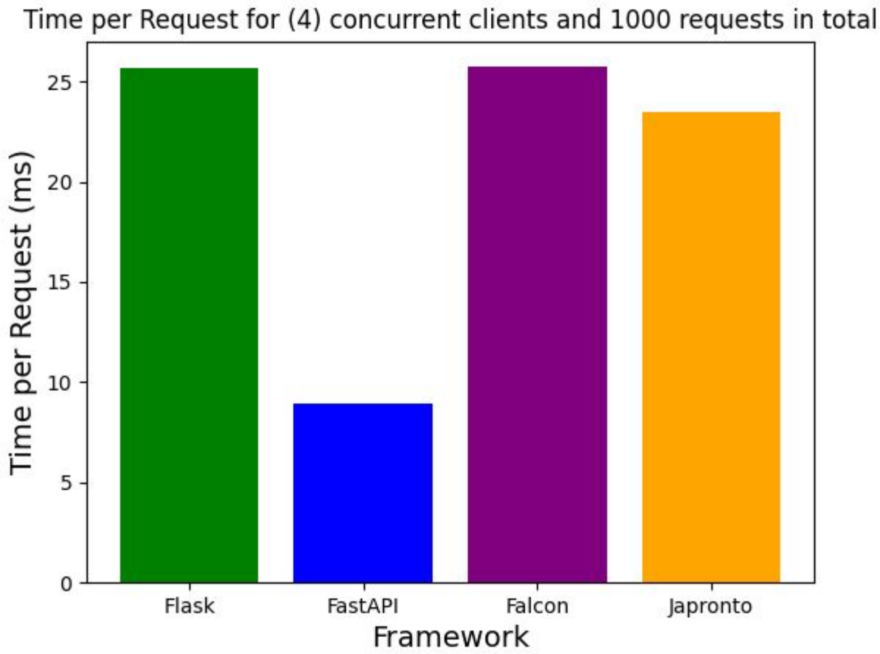
Dockerfile_fastapi 20% 2:0

```dockerfile
FROM python:3.8-buster

WORKDIR /app

RUN pip install falcon gunicorn

COPY . .

EXPOSE 6000
CMD [ "gunicorn", "app_falcon:app", "-b 0.0.0.0:6000", "--workers", "1", "--access-logfile", "/dev/stdout" ]
```

Dockerfile_falcon 10% 1:1

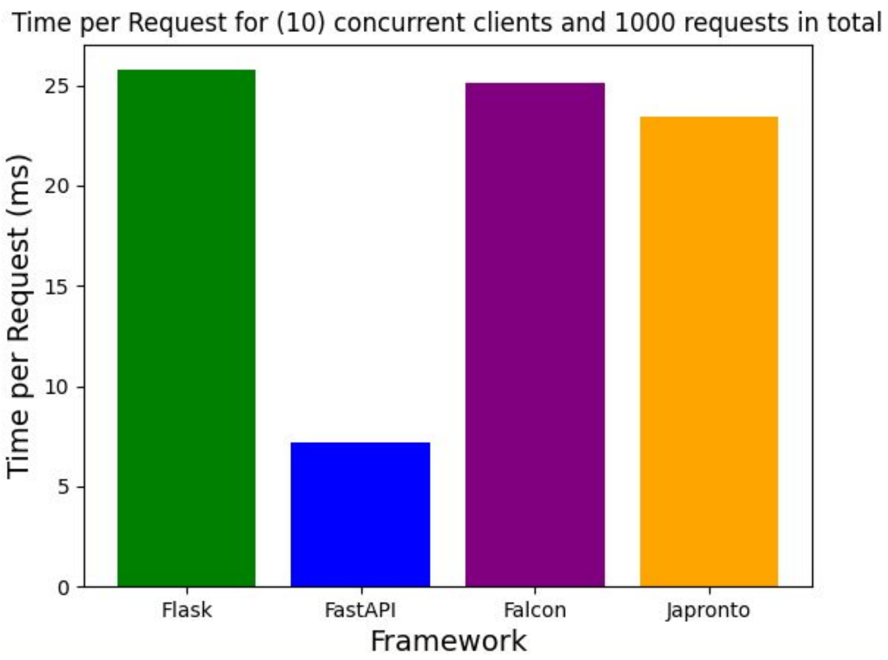
1.1 Probar con Python web frameworks más rápidos.

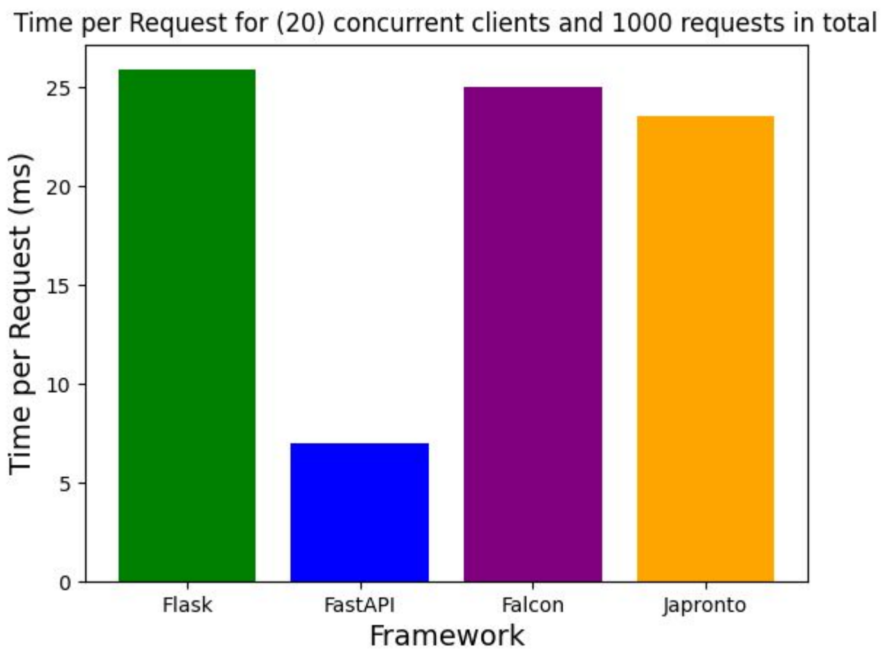




Less is better

1.1 Probar con Python web frameworks más rápidos.





Less is better

1. Mejoras a la implementación anterior

- Intentar con otros métodos:
    - Probar con Python web frameworks más rápidos.

Why this result?
→ Async gunicorn workers.

1. Mejoras a la implementación anterior

- Intentar con otros métodos:
  - Ajustar cantidad de gunicorn workers (FastAPI).

Gunicorn relies on the operating system to provide all of the load balancing when handling requests. Generally we recommend `(2 x $num_cores) + 1` as the number of workers to start off with. While not overly scientific, the formula is based on the assumption that for a given core, one worker will be reading or writing from the socket while the other worker is processing a request.

1.2 Ajustar cantidad de gunicorn workers (FastAPI).

- Test consiste en:
  - Cantidad de peticiones fija: 1000 HTTP request.
  - Variar cantidad de clientes concurrentes.
  - Variar cantidad de gunicorn workers.

tAPI).

FastAPI						
Requests 1000		Clients				
		2	4	5	10	20
Workers	14	15.148	8.589	8.302	7.056	7.818
	15	16.082	8.88	8.16	6.917	7.104
	16	15.198	8.792	7.97	7.059	7.117
	17	15.141	8.784	8.101	6.833	6.978
	18	15.374	9.224	8.311	7.259	7.253
	19	15.376	8.851	7.99	7.186	7.279
	20	15.621	9.935	8.323	7.83	8.547

Requests						
1000		2				
Workers	14	15.148				
	15	16.082	[illegible]	[illegible]	[illegible]	[illegible]
	16	15.198	8.792	7.97	7.059	7.117
	17	15.141	8.784	8.101	6.833	6.978
	18	15.374	9.224	8.311	7.259	7.253
	19	15.376	8.851	7.99	7.186	7.279
	20	15.621	9.935	8.323	7.83	8.547

Does Gunicorn suffer from the thundering herd problem?

The thundering herd problem occurs when many sleeping request handlers, which may be either threads or processes, wake up at the same time to handle a new request. Since only one handler will receive the request, the others will have been awakened for no reason, wasting CPU cycles. At this time, Gunicorn does not implement any IPC solution for coordinating between worker processes. You may experience high load due to this problem when using many workers or threads. However a work has been started to remove this issue.

Conclusiones

Conclusiones

- Escalar el servicio dentro de la misma máquina no lleva a mejores resultados (CPU Bottleneck).
- Utilizar workers asíncronos (FastAPI es una buena opción!).
- #Workers == CPUs * 2 + 1, parece ser una buena fórmula.


"That's all Folks!"