Optimización del Evaluator en el Sistema de Log Detector

Introducción a la Computación Paralela (2022-1)

Vicente Coopman
2022-08-04

1 Idea del proyecto

La idea del proyecto consiste en optimizar el rendimiento del “Evaluador” (También referido como “Evaluator”) dentro del sistema de Log Detector. El sistema de Log Detector se conforma por un conjunto de microservicios, que trabajan juntos para ofrecer al usuario la funcionalidad de reconocer patrones en archivos de registro (También referidos como “Logs”). Dentro de éste sistema el Evaluador cumple un rol crítico, que de fallar o tener un pobre rendimiento, afectaría el funcionamiento de todo el sistema.

El Evaluador es un servicio web desarrollado en Python 3 mediante el web framework Flask [1]. Como principal objetivo se pretenden optimizar las rutas “/eval” y “/eval-launch” del servicio. Ambas rutas, en su ejecución, utilizan el módulo estándar *subprocess* para ejecutar código Python dinámicamente generado por el usuario. El Evaluador como tal, es deployado mediante contenedores Docker.

Para realizar la optimización se pretende primeramente estudiar las herramientas y métodos disponibles. Posteriormente, realizar una medición del desempeño actual del servicio. Implementar las herramientas/métodos seleccionados y comparar con el rendimiento inicial. A partir del resultado de la comparación, evaluar si es necesario mejorar la implementación realizada o investigar otros métodos.

2 Estudio de Herramientas y Métodos disponibles para la optimización

De los posibles métodos/herramientas para realizar la optimización, se seleccionan los siguientes ordenados de menor a mayor complejidad de implementar:

1. Pypy.

PyPy [2] es un intérprete y compilador JIT para el lenguaje Python, que se enfoca en la velocidad y eficiencia, y es 100% compatible con el intérprete original CPython. Pypy puede llegar a presentar mejoras de rendimiento de hasta un 80% para cierto tipo de procesos.

2. Escalamiento horizontal con Docker.

El escalamiento horizontal significa escalar el rendimiento de la aplicación mediante la replicación de la misma. En este caso, se considera algo relativamente sencillo de realizar, ya que el Evaluador por diseño es una aplicación stateless. Esto permite escalar el servicio, sin preocuparse de tener que sincronizar los datos entre todas las instancias del mismo. Lograr esto con Docker no presenta mayor dificultad.

3. Selección de un web framework más enfocado en el rendimiento.

Actualmente existe una gran oferta de web frameworks para el lenguaje Python, cada uno con sus ventajas y desventajas. En específico, el framework actualmente utilizado por el Evaluador es Flask, el cual se enfoca en ser multipropósito y minimalista. Es posible que el uso de otro framework más enfocado en el rendimiento pueda ofrecer un mejor desempeño en las rutas a optimizar.

4. Tuning en la cantidad de Workers de Gunicorn.

Durante los últimos años Python se ha vuelto más y más popular dentro del desarrollo web. Esto como anteriormente se mencionó en el punto 3, genera una gran oferta de frameworks para el desarrollo web. Debido a esto dentro de los estándares de Python se a adoptado la inferfaz Web Server Gateway Interface (WSGI) [3], la cual define los estándares de comunicación entre el servidor web y el servidor de aplicación (En este caso el servidor de aplicación es el Evaluador). Gunicorn [4] es un servidor WSGI que se encarga de conectar ambos servidor web y servidor de aplicación. Gunicorn, además de conectar los servidores, provee otras ventajas como la replicación del mismo mediante *workers.* Estos workers, son cada uno una réplica de Gunicorn manejados por un master. Está replicación funciona similar a un escalamiento horizontal. La documentación oficial recomienda utilizar 2 * CPUs + 1 workers.

5. Implementar paralelismo con Cython.

Cython [5] es un lenguaje de programación **compilado** similar a Python, que se enfoca facilitar la escritura de módulos wrappers a C o C++. Cython soporta por defecto el uso de OpenMP, el cual podría ser utilizado para paralelizar el loop principal de la ruta “/eval” en la aplicación del Evaluador.

3 Desarrollo

3.1 Tooling Inicial

Para llevar a cabo las mediciones tanto de tiempo como de uso de recursos se implementan los scripts: *stress.py* y *docker_stats_fetcher.sh*.

El primero, *stress.py*, envía peticiones HTTP paralelamente utilizando hebras a las rutas del Evaluador. Cada una de estas peticiones tiene un contenido variable, el cual se obtiene aleatoriamente desde *payloads.py*. Este script ofrece escenario de pruebas ya definidos en *settings_scenarios.py*. El tiempo transcurrido de la ejecución del escenario, además de otras métricas son escritos en un archivo .csv bajo el directorio *results.*

docker_stats_fetcher.sh por su parte se encarga de obtener los datos relacionados al uso del recursos del container en el host. Para lograr esto, se consulta cada ~10 ms al daemon de docker (mediante *docker stats <nombre container>*) cuantos recursos ocupa el container. Dentro de estos recursos se encuentran CPU Usage, Mem Usage, Network I/O y Disk I/O.

3.2 Pypy

La adición de Pypy a la aplicación del proyecto es sencilla (Que sea sencilla es una de las principales preocupaciones de la gente de Pypy). Primeramente descargar la versión deseada desde https://www.pypy.org/download.html, luego modificar el Dockerfile de la aplicación del Evaluador para ejecutar la misma con Pypy.

```
1  FROM python:3.8-buster
2
3  WORKDIR /app
4
5  COPY . .
6  COPY requirements.txt requirements.txt
7  RUN ./pypy3.8-v7.3.9-linux64/bin/pypy -m ensurepip
8  RUN ./pypy3.8-v7.3.9-linux64/bin/pypy -mpip install -r requirements.txt
9
10 EXPOSE 5000
11 CMD [ "./pypy3.8-v7.3.9-linux64/bin/pypy", "-mgunicorn", "app:app", "-b
0.0.0.0:5000", "--workers", "9", "--access-logfile", "/dev/stdout" ]
```

Figura 1: Dockerfile Evaluator usando Pypy-v7.3.9 compatible con Python 3.8

3.3 Escalamiento Horizontal con Docker

Para la implementación del escalamiento horizontal, Docker ofrece herramientas como *docker-compose* [6]. Esta permite definir la cantidad de réplicas que se desea de un servicio específico y distribuir la carga entre las réplicas en un estilo Round-Robin.

Sin embargo, una importante desventaja de utilizar esta funcionalidad por defecto de Docker, es que los containers solo quedarán visibles desde dentro de la red interna de Docker; lo que lleva a tener que ejecutar todos los escenarios de prueba desde dentro de un container de "soporte". El Dockerfile de este container se encuentra bajo la carpeta "evaluator" con el nombre "Dockerfile_support".

3.4 Selección de un web framework más enfocado en el rendimiento

Dentro de la búsqueda realizada para encontrar un web framework más ad hoc (Ver ranking de velocidad para parsear data en formato JSON en la figura 2.), se seleccionan los frameworks: FastAPI [7], Falcon [8], japronto [9] y el framework actual, Flask. Estos web frameworks son puestos a prueba bajo las mismas condiciones.

Para realizar las pruebas, se reescribe la aplicación del Evaluador para cada uno de los frameworks (La aplicación escrita en Flask ya estaba, por lo tanto, sólo se escribe para FastAPI, Falcon y japronto). Se incluyen además los correspondientes Dockerfiles para poder correr cada una de las aplicaciones en container de Docker. Finalmente para facilitar la manipulación de estas aplicaciones/containers se utiliza un docker compose file.

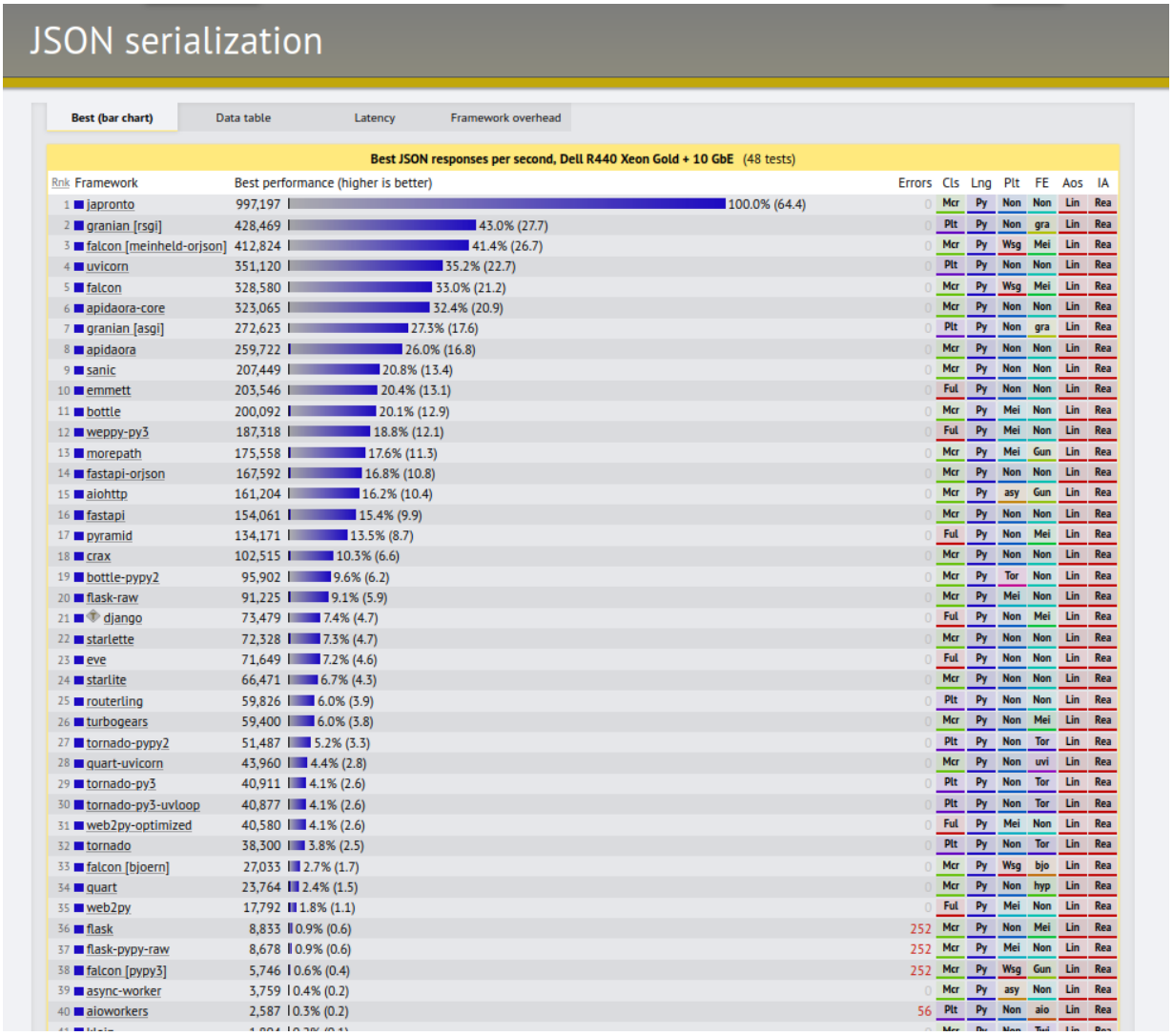

Figura 2: Benchmark web frameworks Python 3 según velocidad para analizar data en formato JSON.

3.5 Tuning en la cantidad de Workers de Gunicorn

La documentaciones oficial de Gunicorn recomienda como punto de base utilizar 2 * CPUs + 1 workers. Esto nace de la suposición de que en cualquier momento, un core va a estar leyendo o escribiendo en un socket, mientras el otro va a estar procesando la request. El extra es para el master, que se encarga de monitorear a los workers.

Para realizar las mediciones con cantidad de workers variable, se utilizan las señales TTIN y TTOU, las cuales significan añadir y quitar un worker respectivamente.

```
def reduce_worker_count(container_name):
 subprocess.call(['docker', 'exec', '-it', container_name, 'kill', '-TTOU', '1'])

def increase_worker_count(container_name):
 subprocess.call(['docker', 'exec', '-it', container_name, 'kill', '-TTIN', '1'])
```

Figura 3: Funciones utilizadas por stress.py para aumentar o disminuir cantidad de workers de un proceso Gunicorn ubicado dentro de un container de Docker.

3.6 Implementar paralelismo con Cython

Esta opción no se llegó a implementar, ya que el análisis previo sobre las características de los loops que se pretendía optimizar concluyó en que el overhead de manejar las hebras sería mayor al beneficio obtenido. La cantidad de ciclos era muy baja (Aproximadamente 5 - 10 ciclos).

4 Resultados

4.1 Pypy

Los resultados obtenidos por el Evaluador utilizando Pypy no fueron para nada los esperados.

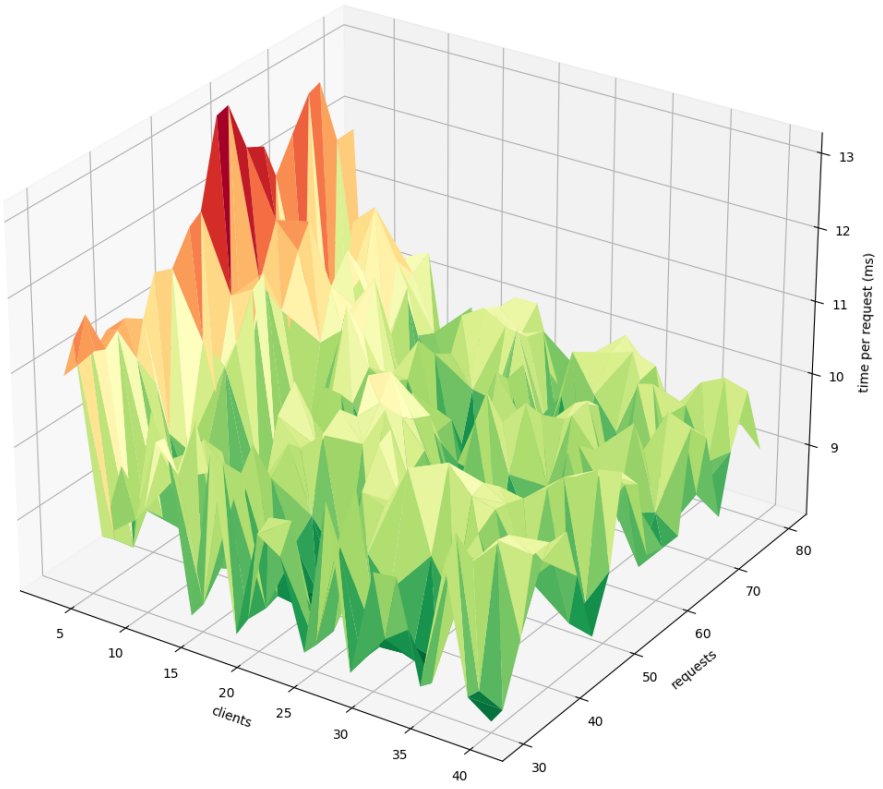

Figura 4: Resultados Evaluador utilizando Pypy v7.3.9. El eje "clientes" representa la cantidad de clientes concurrentes. El Eje "requests" representa cuántas request envía cada cliente concurrente (Esto se arregló para futuros experimentos, pero ya que el experimento con Pypy no se volvió a realizar la gráfica es de una versión desactualizada). El eje "Time per Request" el tiempo promedio por request.

El KPI definido "Time per Request", que significa cuando se demora la aplicación del Evaluador en responder a una HTTP request, se mantuvo muy similar a los resultados obtenidos sin el uso de Pypy, solo se mejoró en 0.2 ms.

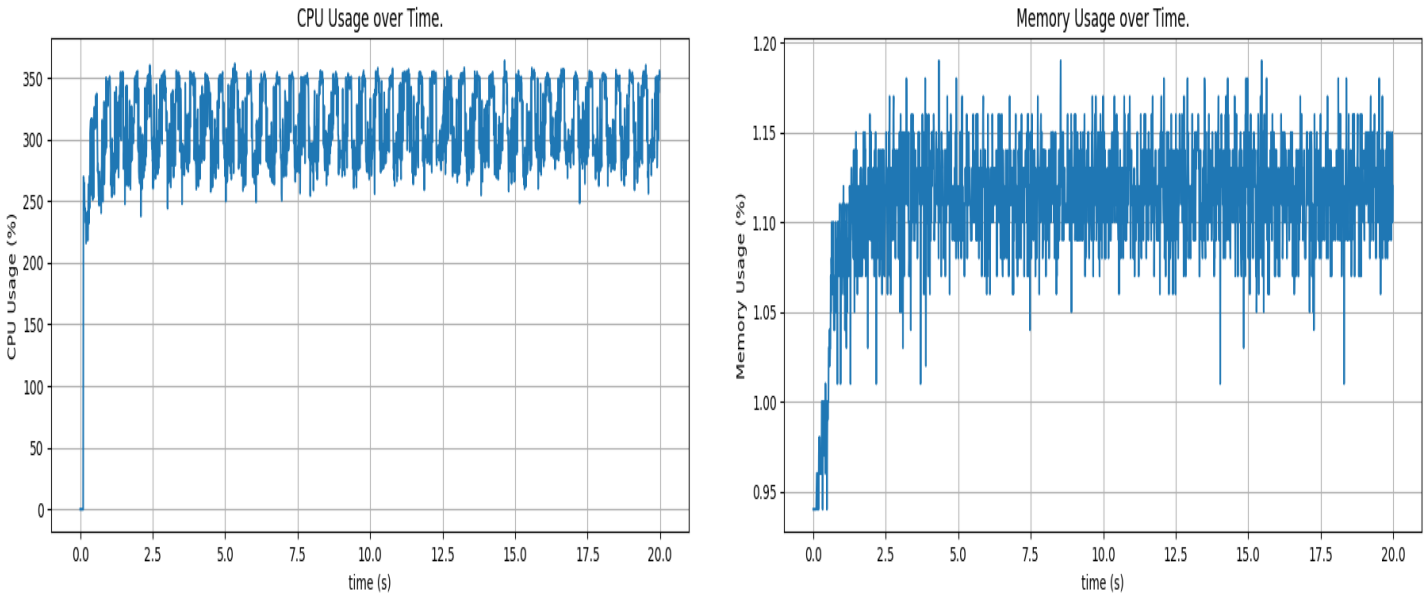

Figura 5: Uso de CPU y Memoria durante el test de Pypy.

Este experimento reveló que el servicio del Evaluador presentaba un bottleneck en el consumo de CPU. Cómo se puede ver en la gráfica de la izquierda, de los 4 cores del host, se está usando un ~360%. Mientras la memoría ronda ~1%.

4.2 Escalamiento Horizontal con Docker

El escalamiento horizontal tampoco demostró ser eficiente. Principalmente por el hecho de que las réplicas se encuentran en la misma máquina, por lo tanto, comparten los recursos de CPU.

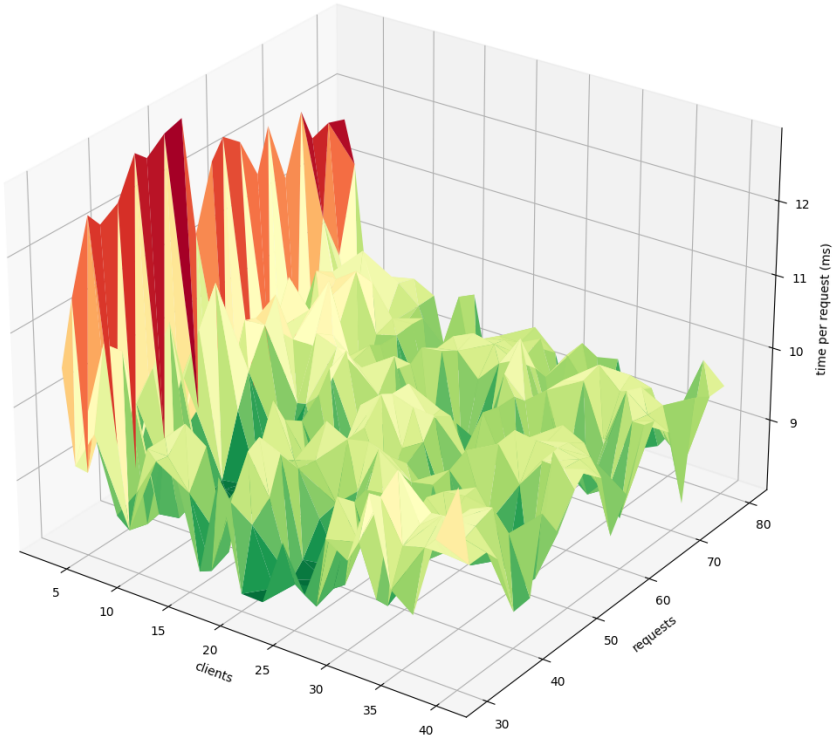

Figura 6: Resultados del test con 2 réplicas del Evaluador.

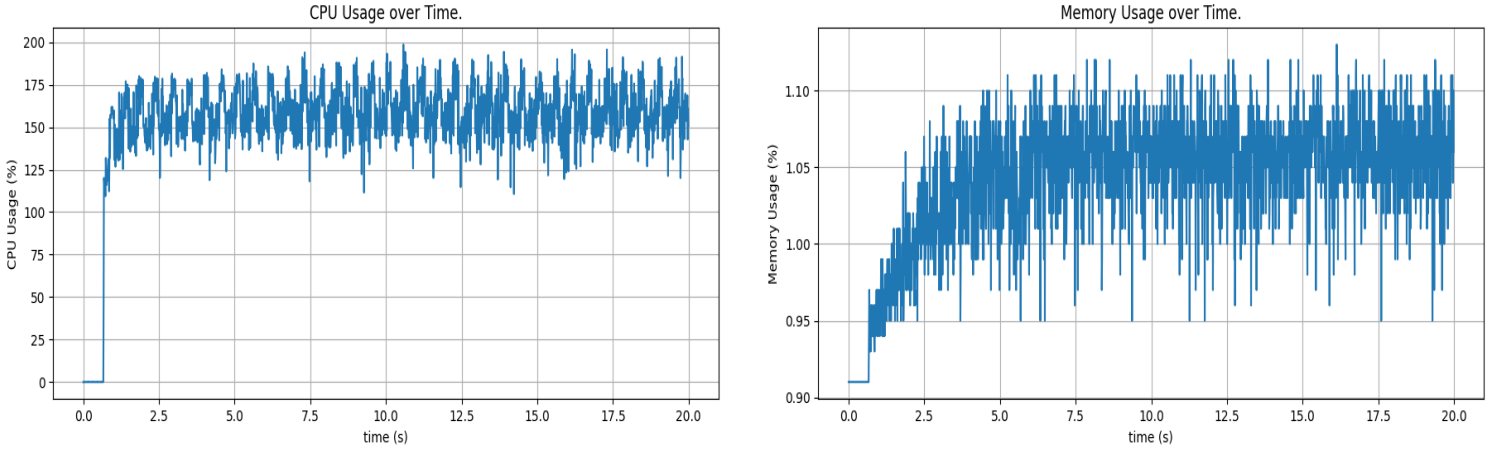

Figura 7: Uso de CPU y Memoria durante el test de Escalamiento horizontal con Docker. Interesante notar que el container solo llega a utilizar >~200%, ya que el otro porciento se debe dividir entre la 2da réplica y el OS.

4.3 Selección de un web framework más enfocado en el rendimiento

Esta prueba tuvo resultados exitosos. No los esperados, pero sí exitosos. La prueba se realizó con 1 worker para cada una de las aplicaciones, excepto para la aplicación con el framework "japroxy" (debido a que no se encuentra documentación al respecto, de hecho, pareciera que el framework "japroxy" tuviera internamente un manejador de workers, similar a lo que provee Gunicorn). **japroxy probablemente no sea WSGI compatible.**

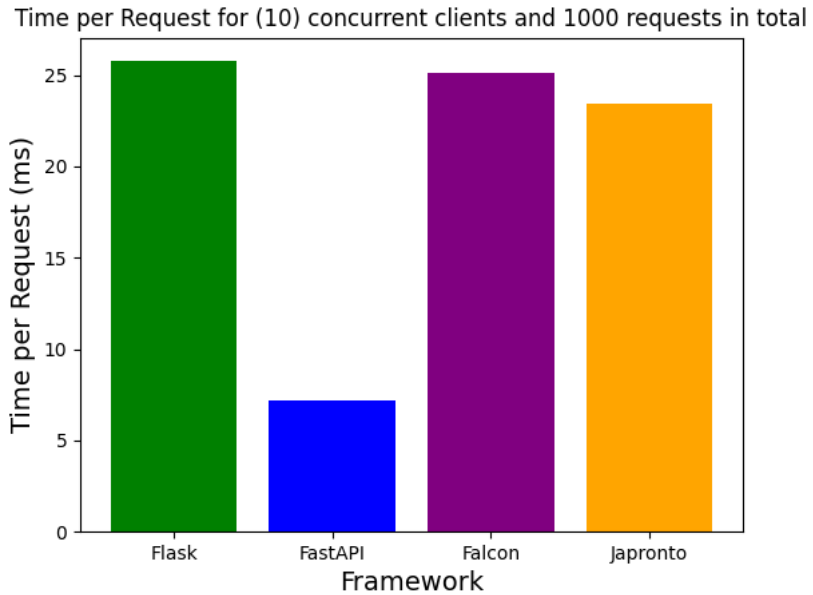

Figura 8: Comparación de Time per Request para 4 frameworks distintos a 10 clientes concurrentes donde en total a cada aplicación se le envían 1000 requests.

Figura 8. muestra cómo FastAPI es muchísimo más rápido que el resto de los competidores. Esto se debe probablemente a la naturaleza asíncrona de FastAPI, que a diferencia de los demás competidores, no queda en blocking mientras procesa una requests, sino que va recibiendo múltiples y cuando termina de procesar, notifica a Gunicorn para que busque la respuesta.

Disclaimer: Me parece que el framework Falco también es compatible con workers asíncronos, pero lamentablemente no tuve momento para re-testearlo.

4.4 Tuning en la cantidad de Workers de Gunicorn

Los resultados de esta prueba concuerdan con la fórmula propuesta por la documentación de Gunicorn. La prueba se realiza contra el Evaluador utilizando FastAPI. Donde el mejor resultado se obtiene en los 2 * CPUs + 1 workers.

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

Figura 9: Tabla de resultados obtenidos para distintas combinaciones de Gunicorn workers y clientes (usuarios web), para una cantidad fija de 1000 requests. Los mejores tiempos se destacan en verde.

Conclusiones

Si el proceso al cual se le quiere mejorar el rendimiento, ya utiliza todos los recursos del host, el escalamiento horizontal de por si no tendrá el efecto esperado, ya que requiere tener más recursos disponibles (Más o más rápidas CPUs).

El uso de frameworks y workers asíncronos mejoran notablemente el rendimiento del Evaluador. Frameworks como FastAPI puede llegar a duplicar el rendimiento.

Recursos

Link al repositorio: https://github.com/vcoopman/proyecto-comp-paralela
En él se encuentran las instrucciones para ejecutar el proyecto.

Referencias

[1] https://flask.palletsprojects.com/en/2.2.x/
[2] https://www.pypy.org/
[3] https://peps.python.org/pep-0333/
[4] https://gunicorn.org/
[5] https://cython.org/
[6] https://docs.docker.com/compose/
[7] https://fastapi.tiangolo.com/
[8] https://falcon.readthedocs.io/en/stable/
[9] https://github.com/squeaky-pl/japronto