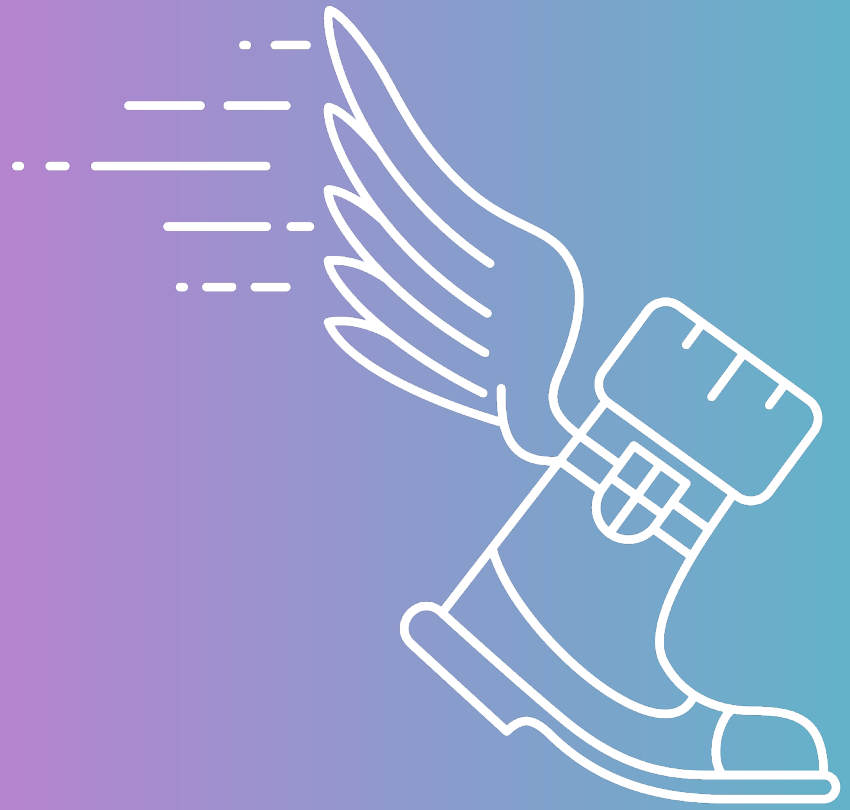
Имя мне скорость



Денис Аникин
https://xfenix.ru

Денис Аникин

Что я делаю

— работаю в Райфе

— lead разработки в 3+ командах

— community lead в Python Community

— fullstack: typescript, python, devops

— выступаю на конференциях и митапах

https://xfenix.ru

А зачем, собственно?

Давайте поговорим о том, зачем

— Текущая ситуация: бенчмарков мало
— Techempower ближе всех, но там пока немного актуальных сочетаний (например, fastapi + gunicorn + uvicorn)
— Ещё один набор «цифр» для сравнения
— Хочется понять что можно взять для действительно быстрой работы

Как я пришел к этой проблеме?

— Я большой поклонник идеи «взять самое быстрое и показать максимальную производительность сразу»
— Мне не нравится позиция «для нашего сервиса/юзеров достаточно и …»
— Мне не нравится позиция «ну у нас всего-то 1 rps, подождут»
— Мне не нравится позиция «бизнесу важнее фичей напилить, чем заморачиваться по производительности»
— Почему? Потому, что сделать хорошо сразу — не так дорого
— Почему? Потому, что хочется делать качественные сервисы для любых пользователей


Москва

Да и просто по приколу

Методика

Что делаем

— Собираем тестовое приложение на docker-compose
— Отправляем на него нагрузку
— Измеряем
— Повторяем N раз с разными docker compose файлами

Почему compose?

Самый «дешёвый» способ проверить работу в контейнерах

И на compose действительно у кого-то работает продакшн

Наш сетап

Что за железо использовалось на сервере?

— Сервер: недорогой VPS за стоковую цену в 959 рублей
— Тип процессора: Intel(R) Xeon(R) Silver 4214 CPU @ 2.20GHz
— Количество выделенных ядер: 4
— Оперативная память: 8 гигабайт
— Виртуализация: KVM
— Канал: 100 мегабит
— ОС: Ubuntu 22.04.3 LTS

Откуда шла нагрузка?

Macbook Apple M1 Pro

Что по софту?

— В основном: python 3.12, где нельзя, там 3.11
— Образ на базе Debian (с alpine слишком много проблем)
— Если можно сделать эффективную сборку без сайд-эффектов, мы делаем (поставить uvloop, например и/или не запускаем на голом uvicorn)
— Несколько рестовых ручек. Одна с базой, другая без
— Используем реальную базу — redis
— Используем валидацию, pydantic
— Пишем код с аннотациями
— Оптимизируем, что можем
— Nginx как реверс прокси, но конфигурацию не оптимизируем (мне немного лениво)

Как борюсь с субъективностью

— Железо одинаковое
— Нагрузка одинаковая
— docker-compose максимально похожие
— Nginx, ubuntu, postgres, redis — идентичны

Почему без руру?

Почему не с postgres?

БЕСТСЕЛЛЕР ГОДА

ЛУЧАНА ЛИТТИЦЦЕТТО

по кочану

Проблемы базы с нагрузкой

— Упирается в диск на 1 машине. А мы то хотим не базу и не диск бенчмаркать (!)


Ну так взял бы в кластере

Проблемы базы с нагрузкой

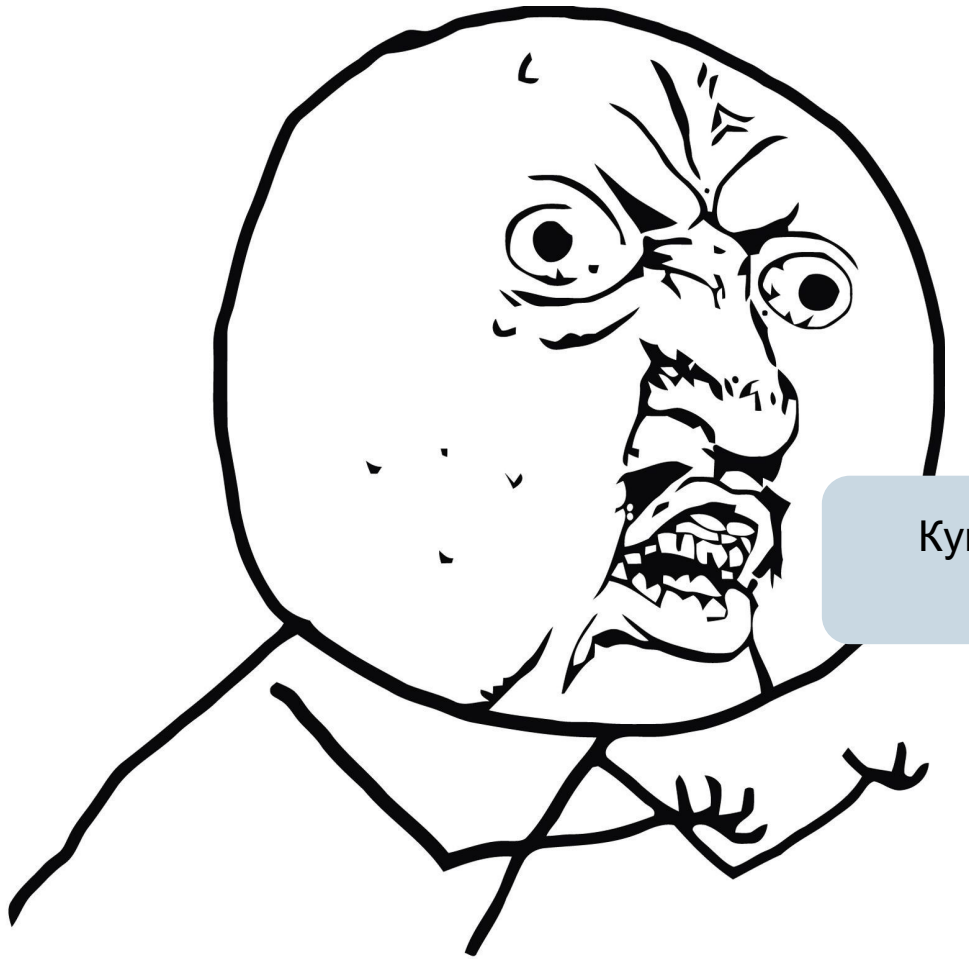
— Упирается в диск на 1 машине. А мы то хотим не базу и не диск бенчмаркать (!)

— В кластере мы упираемся в коннекшены

Сделал бы больше

Проблемы базы с нагрузкой

— Упирается в диск на 1 машине. А мы то хотим не базу и не диск бенчмаркать (!)
— В кластере мы упираемся в коннекшены
— С коннекшенами мы упираемся в оперативку/пулинг



Купил бы кластер пожирнее

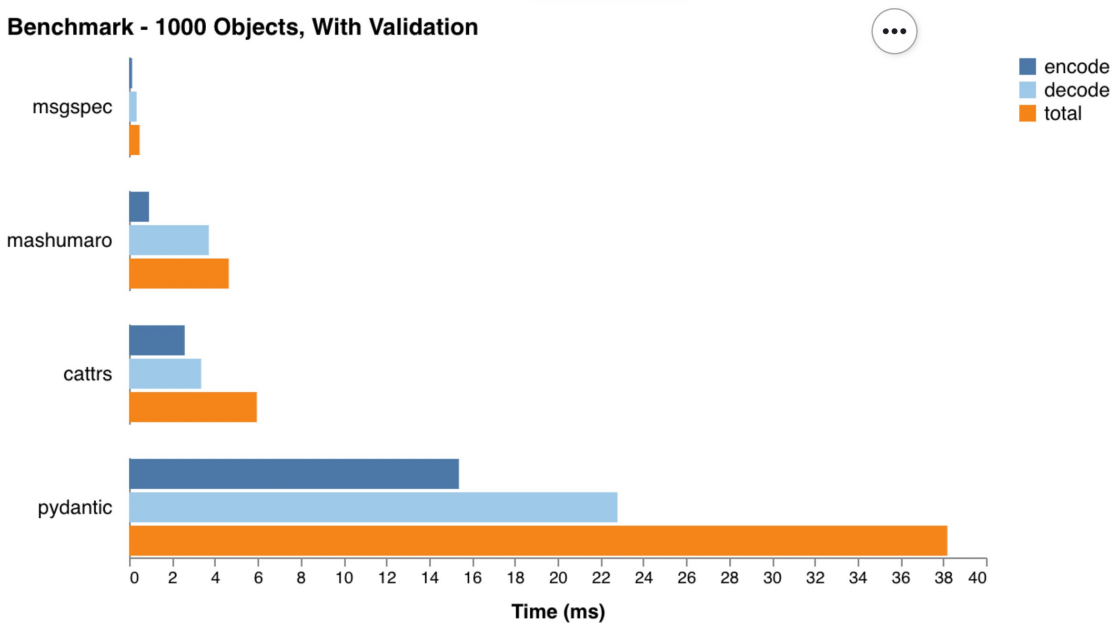
Проблемы базы с нагрузкой

— Упирается в диск на 1 машине. А мы то хотим не базу и не диск бенчмаркать (!)
— В кластере мы упираемся в коннекшены
— С коннекшенами мы упираемся в оперативку
— Я плохо админю сеть и постгрю (а мы не их бенчмаркаем!)
— А ещё вопрос в драйвере/orm

С redis мы лишены многих этих недостатков

Спорно: пробую местами заменить pydantic 2 на msgspec

msgspec



	import (µs)	create (µs)	equality (µs)	order (µs)
msgspec	9.92	0.09	0.02	0.03
standard classes	6.86	0.45	0.13	0.29
dataclasses	489.07	0.47	0.27	0.30
attrs	428.38	0.42	0.29	2.15
pydantic	371.52	4.84	10.56	N/A

Нагрузку шлём k6

The best developer experience for load testing

Open source and SaaS for engineering teams

OPEN SOURCE >_

GRAFANA CLOUD K6 >_

Что мы измеряем?

Важные числа

Я ставил перед собой задачу получить +- близкие к реальности цифры по этим категориям:

— RPS
— Latency

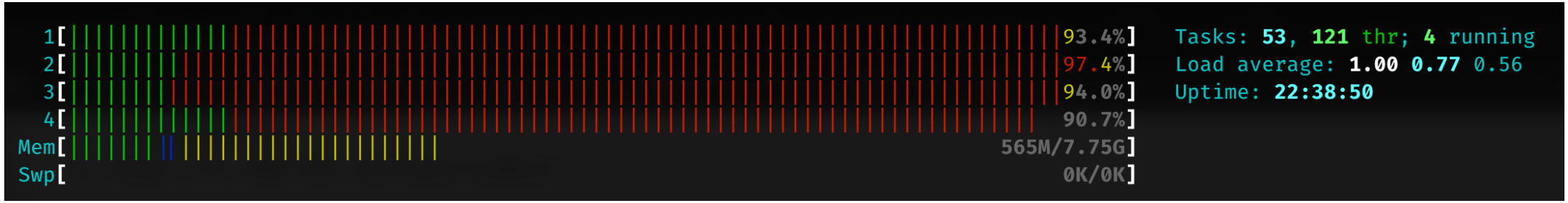
Мы сравниваем не абсолютные числа, а друг относительно друга в одинаковых условиях!

AioHTTP для начала

Результаты

```
data_received..................: 13 MB  190 kB/s
data_sent......................: 3.1 MB 45 kB/s
http_req_blocked...............: avg=555.89µs min=0s      med=3µs      max=672.36ms p(90)=9µs    p(95)=14µs
http_req_connecting............: avg=545µs    min=0s      med=0s       max=672.29ms p(90)=0s     p(95)=0s
http_req_duration..............: avg=650.66ms min=14.76ms med=449.66ms max=2.82s    p(90)=1.59s p(95)=1.85s
  { expected_response:true }...: avg=650.66ms min=14.76ms med=449.66ms max=2.82s    p(90)=1.59s p(95)=1.85s
http_req_failed................: 0.00%  ✓ 0          ✗ 34065
http_req_receiving.............: avg=55.04µs  min=4µs     med=24µs     max=15.9ms   p(90)=72µs   p(95)=120µs
http_req_sending...............: avg=54.27µs  min=2µs     med=10µs     max=12.23ms  p(90)=41µs   p(95)=125µs
http_req_tls_handshaking.......: avg=0s       min=0s      med=0s       max=0s       p(90)=0s     p(95)=0s
http_req_waiting...............: avg=650.56ms min=14.73ms med=449.58ms max=2.82s    p(90)=1.59s p(95)=1.85s
http_reqs......................: 34065  484.821487/s
iteration_duration.............: avg=852.92ms min=214.9ms med=654.54ms max=3.04s    p(90)=1.79s p(95)=2.06s
iterations.....................: 34065  484.821487/s
vus............................: 115    min=5        max=999
vus_max........................: 1000   min=1000     max=1000
```

В процессе теста

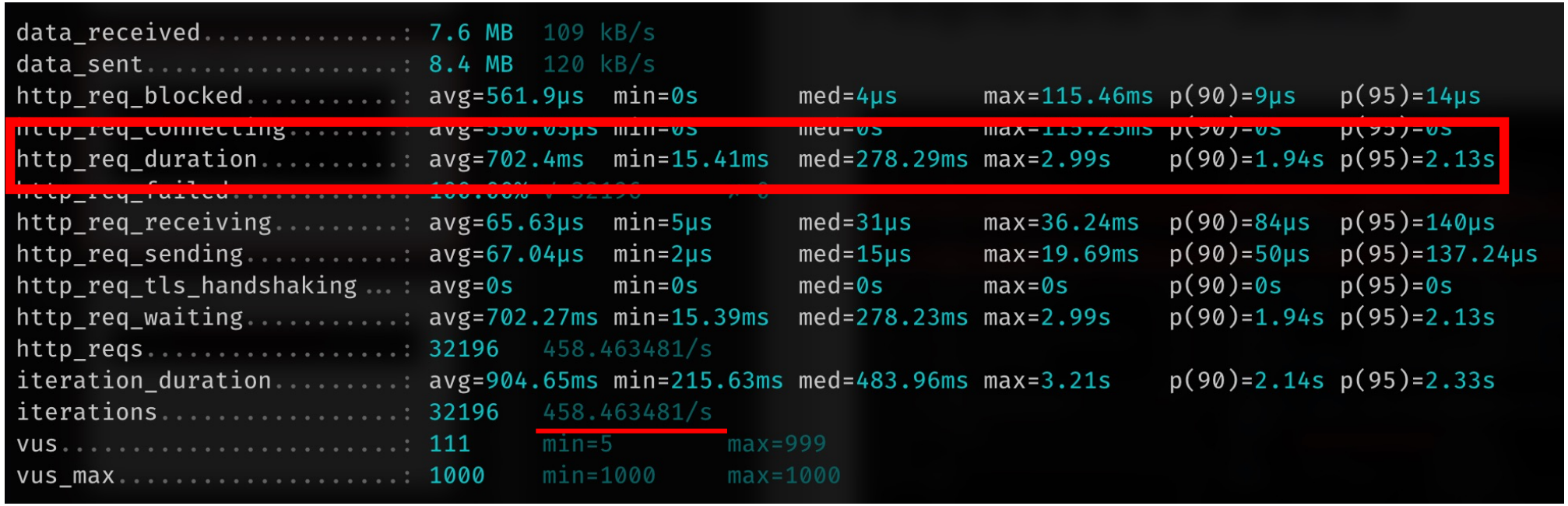


Результаты кратко

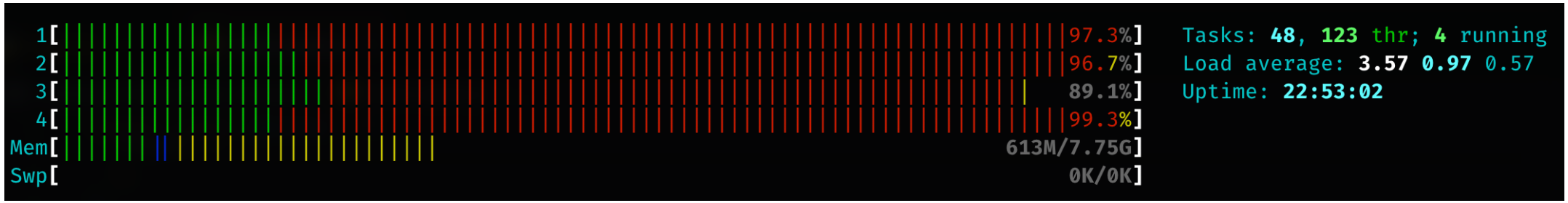
— 0.5к RPS

— 1.5s latency p90

Результаты — запись



В процессе теста



Результаты кратко

— 0.5к RPS

— 1.9s latency p90

FastAPI + gunicorn + uvicorn

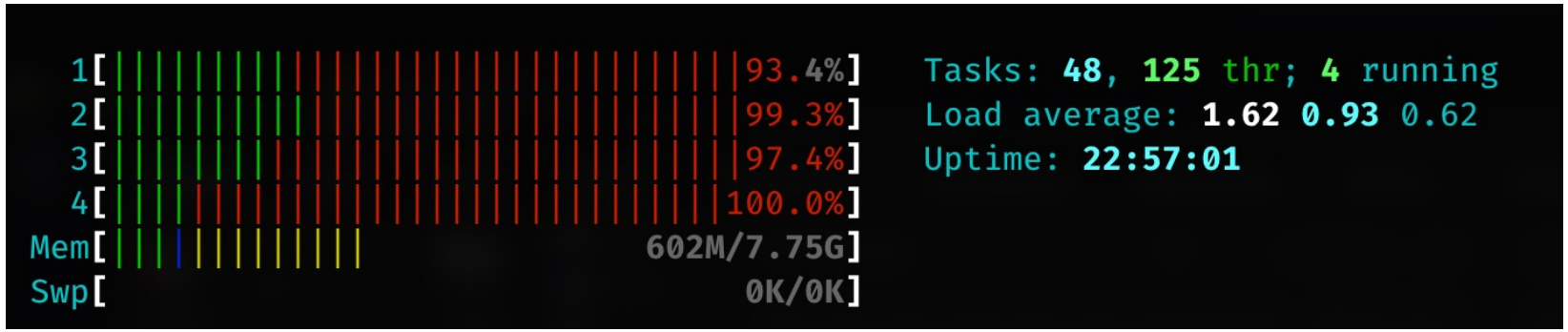
Пару слов о связке

— Есть связка fastapi + uvicorn
— Есть связка fastapi + gunicorn + uvicorn
— Различие лежит в подходе к скейлингу. В gunicorn связке скейлит gunicorn, в варианте только с uvicorn — мы можем использовать внешние «мощности» вроде k8s
— Однако, бывает, что gunicorn + uvicorn используется и в k8s
— Есть ли разница в том кто поднимает процессы?
— Измерить бы хотелось всё, но мы ограничены в ресурсах (включая время)

Результаты

```
data_received..................: 10 MB  148 kB/s
data_sent......................: 2.9 MB 41 kB/s
http_req_blocked...............: avg=583.1µs  min=0s       med=5µs      max=91.83ms p(90)=10µs   p(95)=18µs
http_req_connecting............: avg=570.91µs min=0s       med=0s       max=91.73ms p(90)=0s     p(95)=0s
http_req_duration..............: avg=741.25ms min=15.57ms  med=108.16ms max=2.99s   p(90)=2.07s  p(95)=2.27s
  { expected_response:true }...: avg=741.25ms min=15.57ms  med=108.16ms max=2.99s   p(90)=2.07s  p(95)=2.27s
http_req_failed................: 0.00%  ✓ 0          ✗ 31005
http_req_receiving.............: avg=60.85µs  min=4µs      med=32µs     max=14.96ms p(90)=84µs   p(95)=136µs
http_req_sending...............: avg=44.36µs  min=2µs      med=14µs     max=12.45ms p(90)=42µs   p(95)=119µs
http_req_tls_handshaking.......: avg=0s       min=0s       med=0s       max=0s      p(90)=0s     p(95)=0s
http_req_waiting...............: avg=741.14ms min=15.36ms  med=108.1ms  max=2.99s   p(90)=2.07s  p(95)=2.27s
http_reqs......................: 31005  440.274209/s
iteration_duration.............: avg=943.05ms min=215.74ms med=310.18ms max=3.23s   p(90)=2.27s  p(95)=2.48s
iterations.....................: 31005  440.274209/s
vus............................: 111    min=5        max=999
vus_max........................: 1000   min=1000     max=1000
```

В процессе теста



Результаты кратко

— ~0.5к RPS
— 2s latency p90

На запись результаты схожие (смысла смотреть их нет)

Litestar + gunicorn + uvicorn

Что такое litestar?

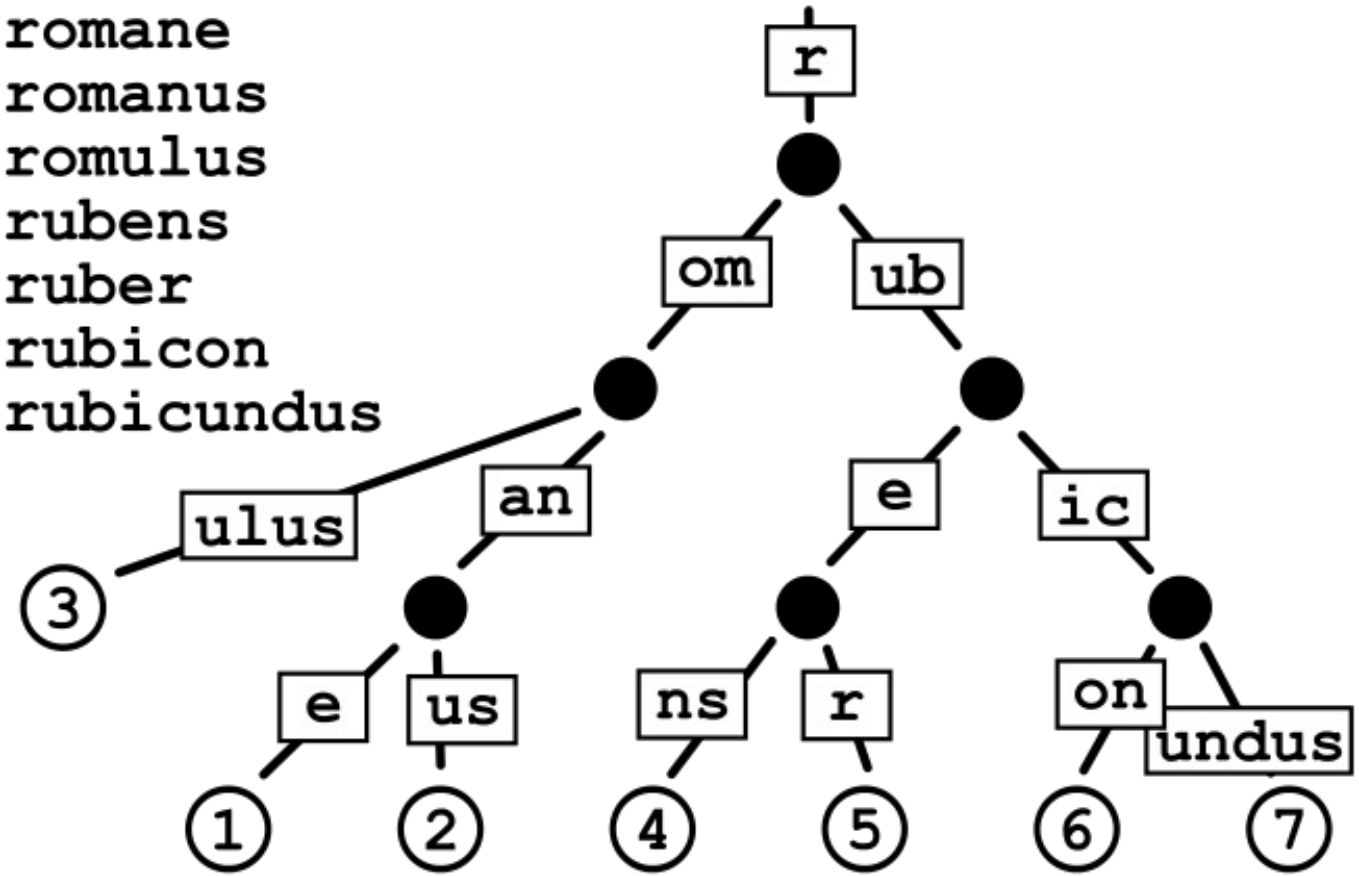
— Для меня — «сверхновая» из мира фреймворков
— Потенциальная замена fastapi
— Хорошие практики
— Классная архитектура
— Большое количество функций
— При этом удобство, близкое к fastapi

Пару фишек, что зашли

— Явные DTO (меньше магии как к FastAPI)
— Отличная интеграция с SqlAlchemy и вывод из моделей DTOшек и даже polyfactory на их основе
— Интеграция с prometheus и opentelemetry
— Channels — по сути, мини-фреймворк аля propan, встроенный для EDA (!!!)

Еще radix роутинг

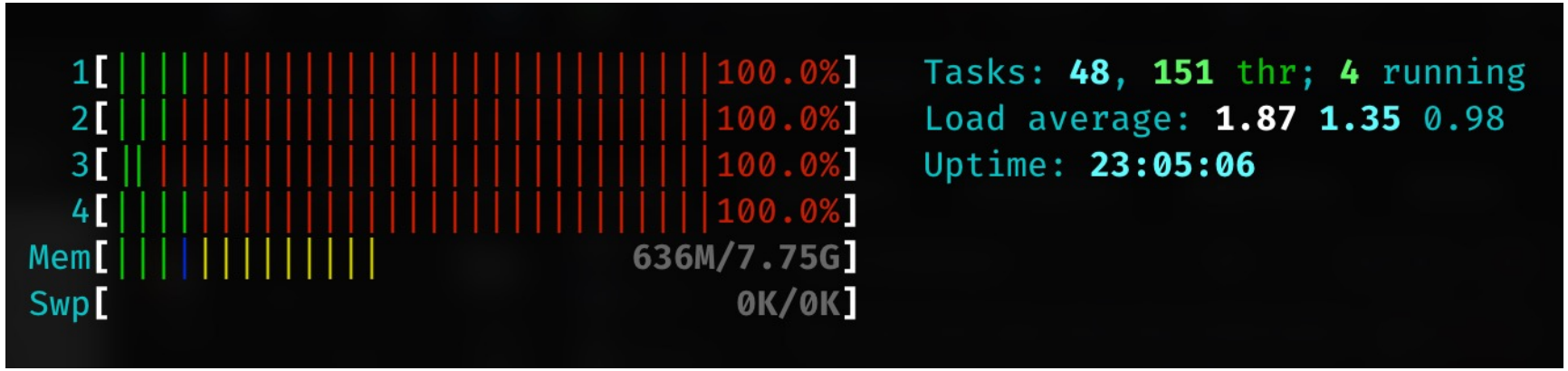
1 romane
2 romanus
3 romulus
4 rubens
5 ruber
6 rubicon
7 rubicundus



Результаты

```
data_received..................: 12 MB  165 kB/s
data_sent......................: 3.2 MB 45 kB/s
http_req_blocked...............: avg=580.98µs min=0s       med=3µs      max=893.99ms p(90)=9µs    p(95)=14µs
http_req_connecting............: avg=571.63µs min=0s       med=0s       max=893.93ms p(90)=0s     p(95)=0s
http_req_duration..............: avg=644.98ms min=14.8ms   med=45.42ms  max=4.39s    p(90)=1.83s  p(95)=2.09s
  { expected_response:true }...: avg=644.98ms min=14.8ms   med=45.42ms  max=4.39s    p(90)=1.83s  p(95)=2.09s
http_req_failed................: 0.00%  ✓ 0          ✗ 34380
http_req_receiving.............: avg=52.97µs  min=4µs      med=26µs     max=19.75ms  p(90)=70µs   p(95)=112µs
http_req_sending...............: avg=42.2µs   min=2µs      med=11µs     max=12.94ms  p(90)=39µs   p(95)=97.04µs
http_req_tls_handshaking.......: avg=0s       min=0s       med=0s       max=0s       p(90)=0s     p(95)=0s
http_req_waiting...............: avg=644.88ms min=14.64ms  med=45.33ms  max=4.39s    p(90)=1.83s  p(95)=2.09s
http_reqs......................: 34380  489.495683/s
iteration_duration.............: avg=846.74ms min=215.05ms med=247.32ms max=4.6s     p(90)=2.03s  p(95)=2.29s
iterations.....................: 34380  489.495683/s
vus............................: 110    min=5        max=1000
vus_max........................: 1000   min=1000     max=1000
```

Что там по загрузке



Результаты кратко

— ~0.5к RPS

— 1.83s latency p90

Интересно, что мы с techempower разошлись

Что же следующее?

Загадочный следующий блок

FastAPI + Granian

Что за granian?

EMMETT

The web framework for inventors


I DON'T NEED IT
memegenerator.net

А ещё…


Rust
Я всё переписал на rust’e,
почему мне до сих пор больно
писать софт?

Кстати, так, наверное, правильнее?


[redacted]
Я всё переписал на [redacted],
почему мне до сих пор больно
писать софт?

Чуть больше о granian

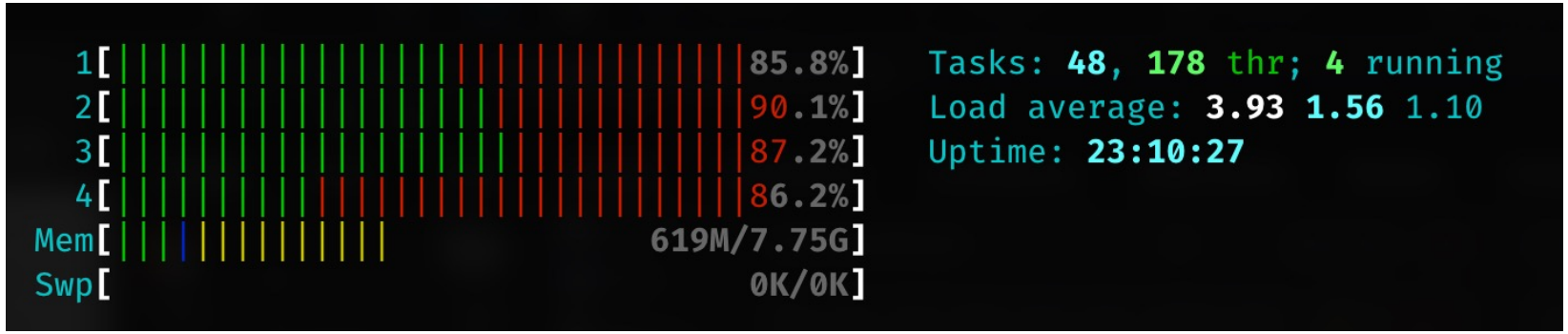
— Призван заменить gunicorn + uvicorn
— Поддерживает ASGI, WSGI, RSGI (!)
— Использовать просто: **granian --interface asgi main:app**
— Супервизор
— Амбициозно?
— BISTRO (мы же хотим быстро)

А теперь вернемся к связке fastapi + granian

Результаты

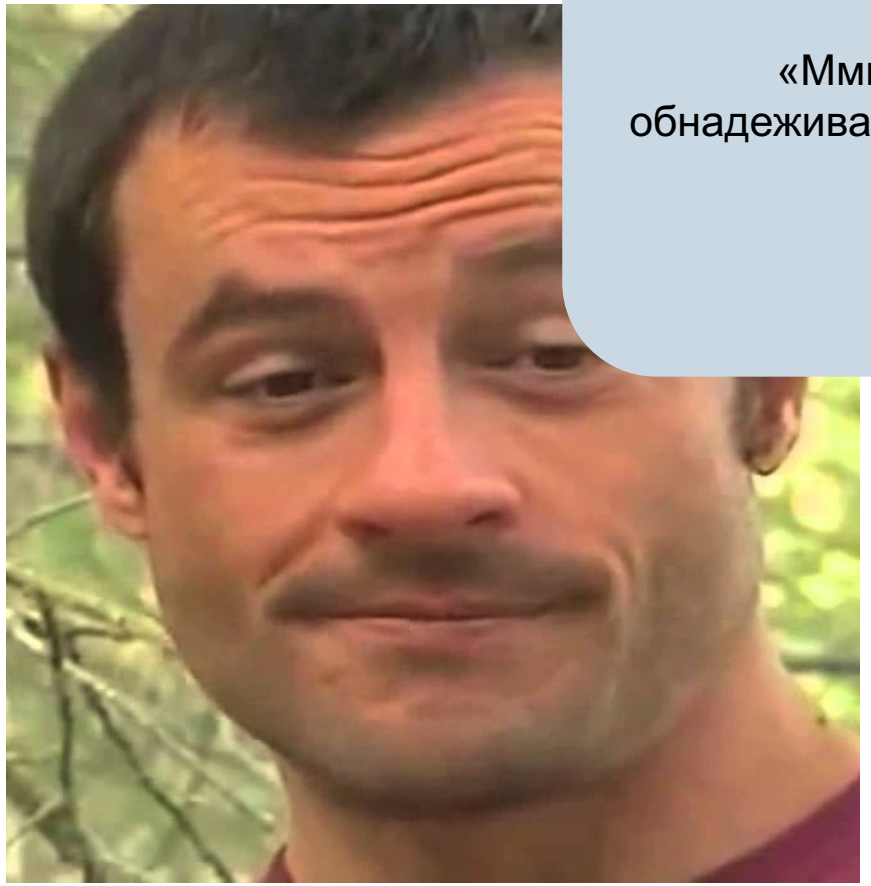
```
data_received..................: 29 MB  418 kB/s
data_sent......................: 8.0 MB 114 kB/s
http_req_blocked...............: avg=256.92µs min=0s       med=2µs     max=811.02ms p(90)=7µs      p(95)=9µs
http_req_connecting............: avg=251.64µs min=0s       med=0s      max=810.95ms p(90)=0s       p(95)=0s
http_req_duration..............: avg=130.32ms min=14.86ms  med=32.33ms max=1.95s    p(90)=411.52ms p(95)=696.24ms
  { expected_response:true } ... : avg=130.32ms min=14.86ms  med=32.33ms max=1.95s    p(90)=411.52ms p(95)=696.24ms
http_req_failed................: 0.00%  ✓ 0          ✗ 87062
http_req_receiving.............: avg=39.27µs  min=4µs      med=19µs    max=24.68ms  p(90)=49µs     p(95)=80µs
http_req_sending...............: avg=28.33µs  min=2µs      med=7µs     max=23.59ms  p(90)=23µs     p(95)=50µs
http_req_tls_handshaking.......: avg=0s       min=0s       med=0s      max=0s       p(90)=0s       p(95)=0s
http_req_waiting...............: avg=130.26ms min=14.84ms  med=32.25ms max=1.95s    p(90)=411.37ms p(95)=696.08ms
http_reqs......................: 87062  1241.222484/s
iteration_duration.............: avg=331.49ms min=214.93ms med=234.4ms max=2.15s    p(90)=613.24ms p(95)=896.55ms
iterations.....................: 87062  1241.222484/s
vus............................: 113    min=5        max=999
vus_max........................: 1000   min=1000     max=1000
```

В процессе теста



Результаты кратко

— ~1.2к RPS
— 0.4s latency p90



«Ммм, это очень обнадеживающие результаты!»

А сможете сейчас угадать фреймворк?

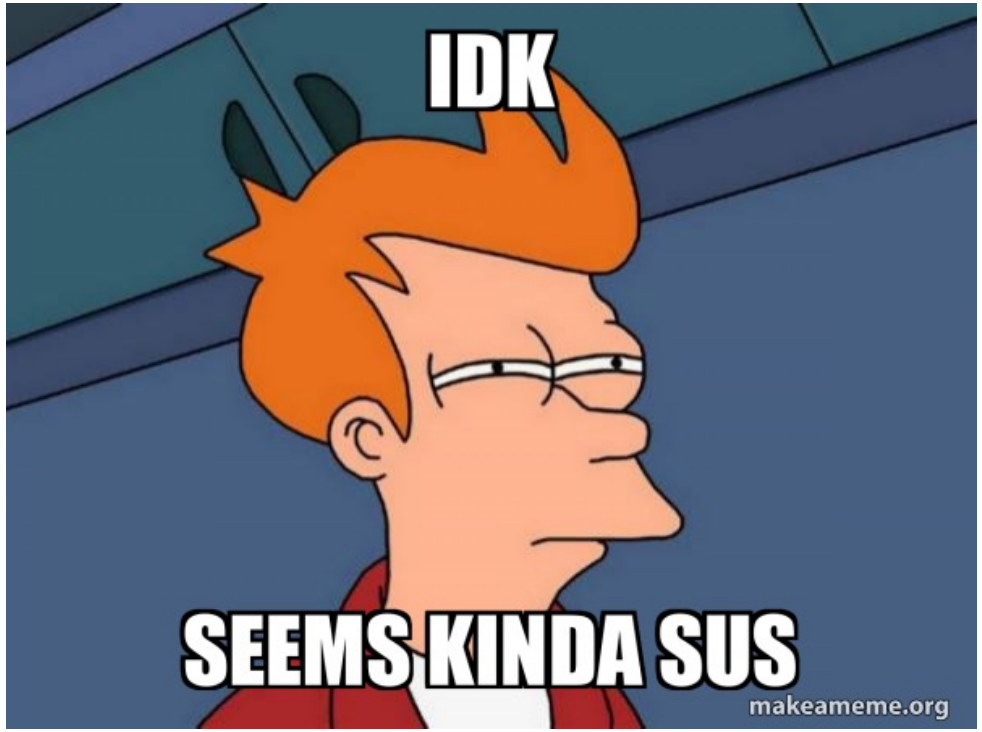
Robyn


Documentation
Releases
Community
Discord
⌘K
Meet Robyn
A Fast, Innovator Friendly, and Community Driven Python Web Framework
Robyn merges Python's async capabilities with a Rust runtime for reliable, scalable web solutions.
Experience quick project scaffolding, enjoyable usage, and robust plugin support.
Get started
Learn more

. Subsequently, the router gets populated.

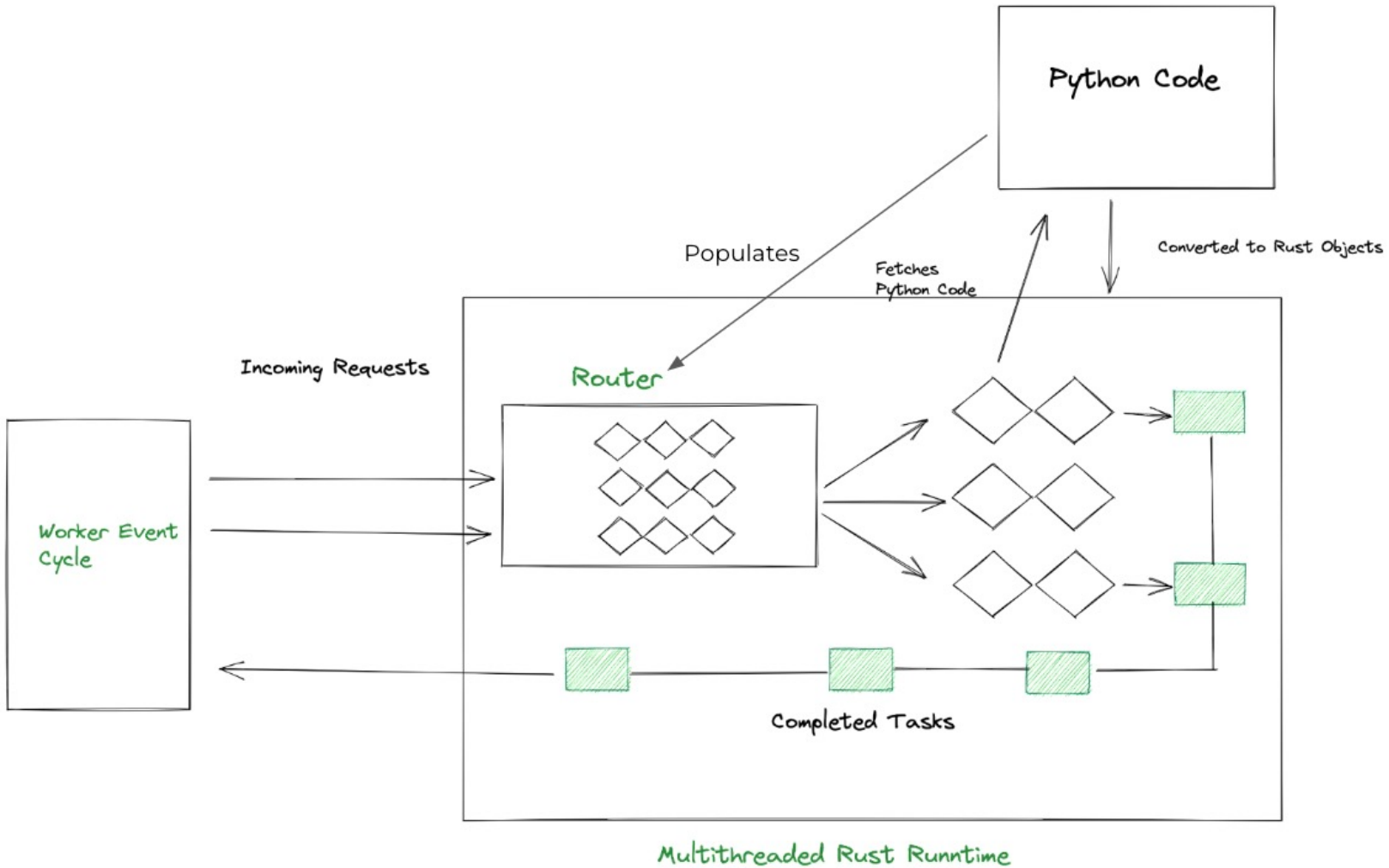
Python code is dynamically transformed into Rust objects.

architecture allows incoming requests to be matched against this router, and relevant Rust objects are queued for execution in the thread pool based on their types.


IDK
SEEMS KINDA SUS
makeameme.org


What did it cost?
IDK

The Architecture



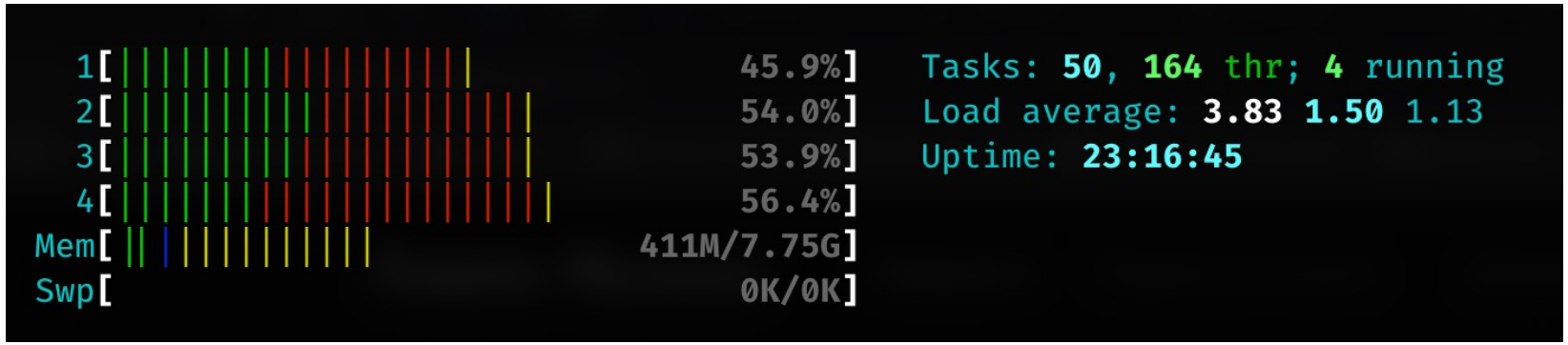
Вот так выглядит код на нём (приблизительно)

```
from robyn import Robyn

app = Robyn(__file__)

@app.get("/")
async def get_really_important(request):
    return "Hello, world!"

app.start(port=8080)
```

Результаты

```
data_received..................: 27 MB  388 kB/s
data_sent......................: 7.4 MB 106 kB/s
http_req_blocked...............: avg=539.39µs min=0s       med=2µs      max=1.31s   p(90)=6µs      p(95)=9µs
http_req_connecting............: avg=534.05µs min=0s       med=0s       max=1.31s   p(90)=0s       p(95)=0s
http_req_duration..............: avg=155.03ms min=14.84ms  med=23.89ms  max=28.92s  p(90)=292.48ms p(95)=559.42ms
  { expected_response:true }...: avg=155.03ms min=14.84ms  med=23.89ms  max=28.92s  p(90)=292.48ms p(95)=559.42ms
http_req_failed................: 0.00%  ✓ 0          ✗ 80812
http_req_receiving.............: avg=41.66µs  min=4µs      med=19µs     max=17.91ms p(90)=50µs     p(95)=85.44µs
http_req_sending...............: avg=24.72µs  min=2µs      med=8µs      max=16.83ms p(90)=23µs     p(95)=46µs
http_req_tls_handshaking.......: avg=0s       min=0s       med=0s       max=0s      p(90)=0s       p(95)=0s
http_req_waiting...............: avg=154.96ms min=14.82ms  med=23.83ms  max=28.92s  p(90)=292.43ms p(95)=559.38ms
http_reqs......................: 80812  1150.890877/s
iteration_duration.............: avg=356.73ms min=215.08ms med=225.21ms max=29.12s  p(90)=493.88ms p(95)=764.58ms
iterations.....................: 80812  1150.890877/s
vus............................: 112    min=5        max=999
vus_max........................: 1000   min=1000     max=1000
```

В процессе теста — приятно



Результаты кратко

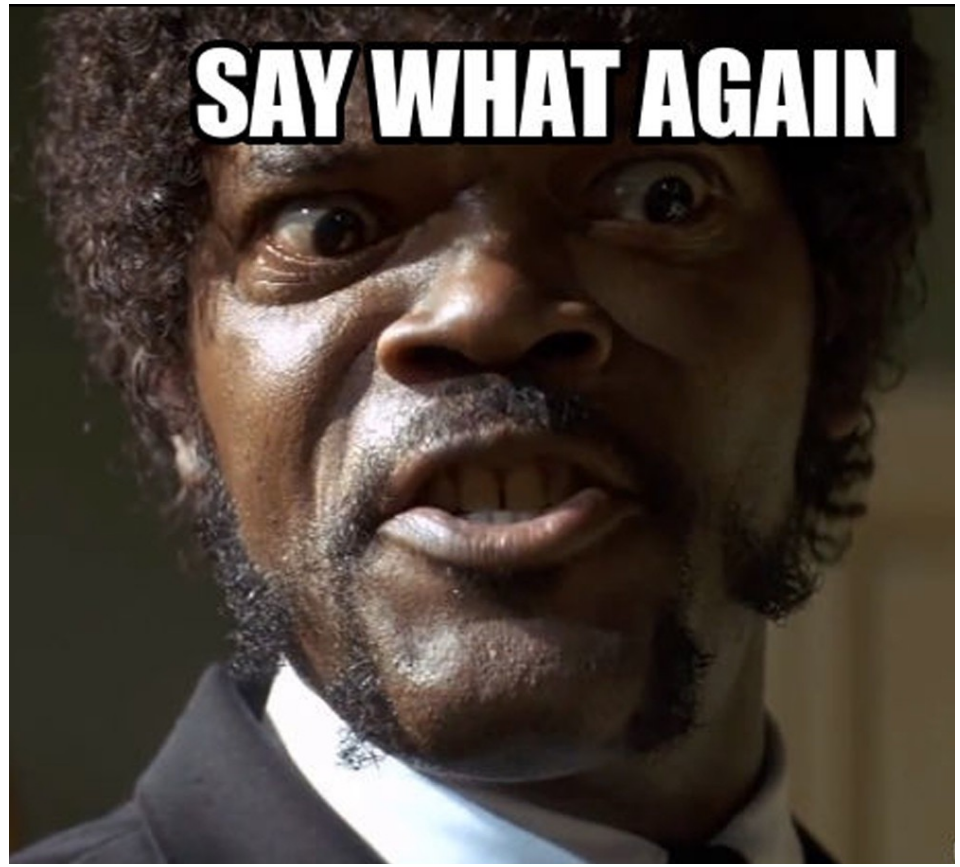
— 1.1к RPS
— 0.3s latency p90

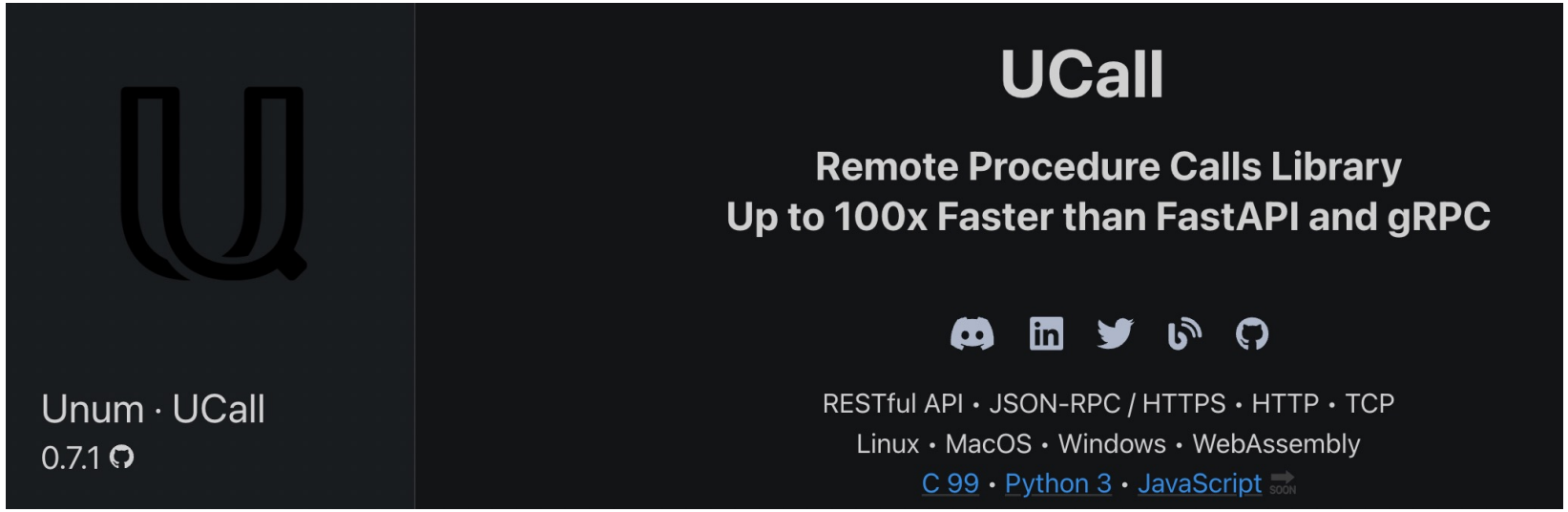
Спорю, что тут никто, кроме одного человека, не догадается…

Внезапно: ucall

Что такое ucall?

— RPC фреймворк…
— Быстрее FastAPI и gRPC
— На питоне

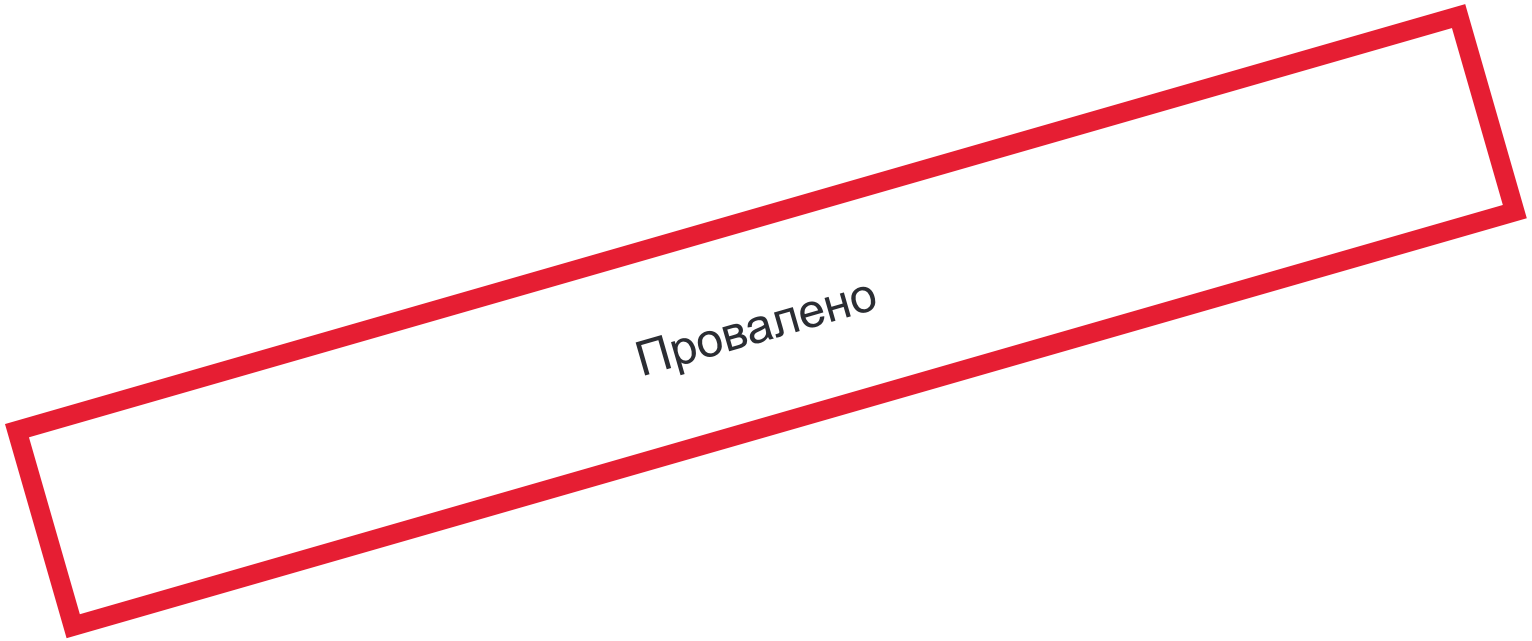

SAY WHAT AGAIN


Unum · UCall
0.7.1
UCall
Remote Procedure Calls Library
Up to 100x Faster than FastAPI and gRPC
RESTful API • JSON-RPC / HTTPS • HTTP • TCP
Linux • MacOS • Windows • WebAssembly
C 99 • Python 3 • JavaScript
SOON

Немного на кодовом

```
from ucall.posix import Server
# from ucall.uring import Server on 5.19+ (это про ядро линукса)


server = Server()


@server
def sum(a: int, b: int):
    return a + b


server.run()
```

Результаты



Пару слов в конце секции

— Не завел ни локально, ни на серваке, ни в контейнере (но я пытался)
— Все супер сырое и документации, считай, что нет
— Большой вопрос как масштабировать это решение (там есть кое-что про треды, но настройка недоступна)
— Всё таки это JSON RPC в первую очередь, REST сделать у меня не удалось
— Если вы хотите, исходники эксперимента есть в моем репозитории

Ну и зачем ты затащил…?

Setup	🔁	Server	Latency w 1 client	Throughput w 32 clients
Fast API over REST	❌	🐍	1'203 µs	3'184 rps
Fast API over WebSocket	✅	🐍	86 µs	11'356 rps [1]
gRPC [2]	✅	🐍	164 µs	9'849 rps
UCall with POSIX	❌	C	62 µs	79'000 rps
UCall with io_uring	✅	🐍	40 µs	210'000 rps
UCall with io_uring	✅	C	22 µs	231'000 rps


ТЫ ПОПАЛСЯ
НА КЛИКБЕЙТ

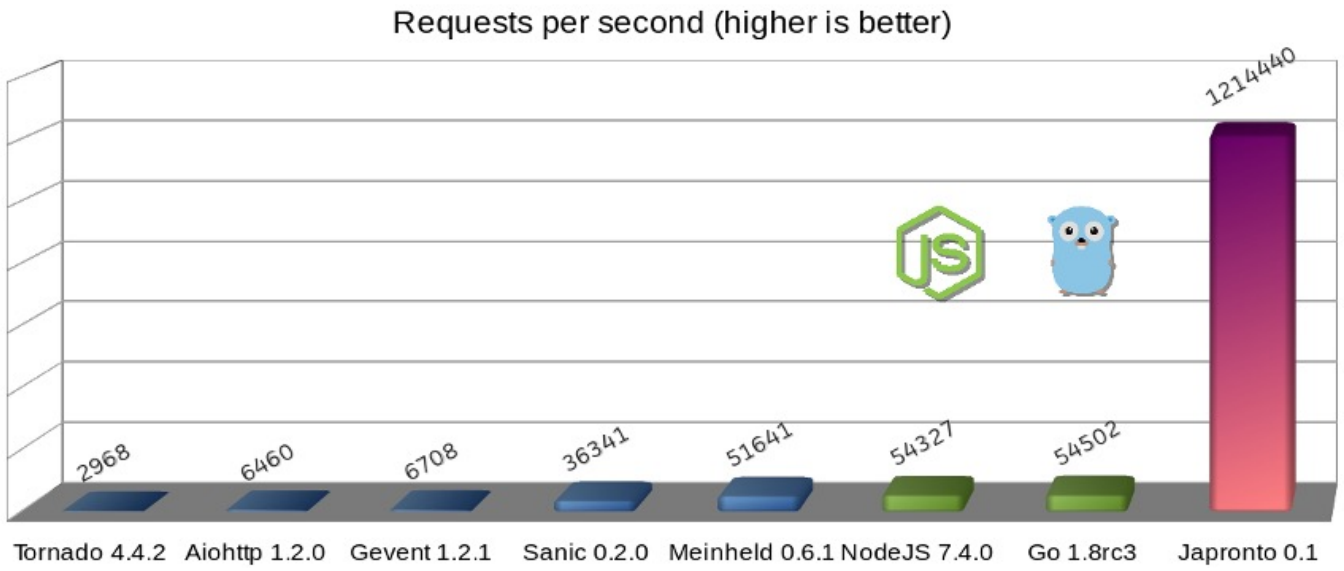

ПОТРАЧЕНО

Спорим, я вновь вас удивлю?

Э, то есть, больше, чем в предыдущей секции не выйдет, но…

Внезапно: socketify.py

Эпоха свершений: japronto, vibora



vibora


ВИДЕЛ ЭТОГО ДЕВЕЛОПЕРА
ПИШЕТ НА СИ, А
ГОВОРИТ, ЧТО НА ПИТОНЕ
imgflip.com


SOCKETIFY

Что даёт socketify.py?

— Вебсокеты
— HTTP/HTTPS
— Встроенный ASGI (внезапно) / WSGI

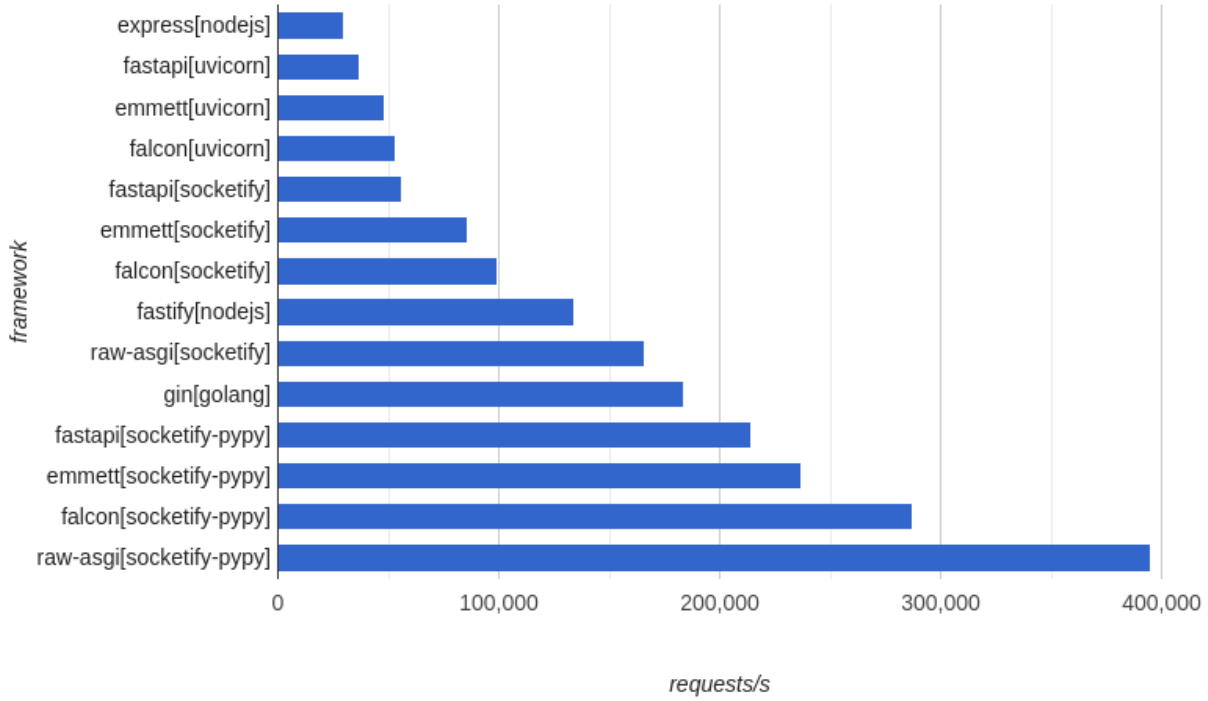
README.md

- WebSocket with pub/sub support
- Fast and reliable Http/Https
- Support for Windows, Linux and macOS Silicon & x64
- Support for `PyPy3` and `CPython`
- Dynamic URL Routing with Wildcard & Parameter support
- Sync and Async Function Support
- Really Simple API
- Fast and Encrypted TLS 1.3 quicker than most alternative servers can do even unencrypted, cleartext messaging
- Per-SNI HttpRouter Support
- Proxy Protocol v2
- Shared or Dedicated Compression Support
- Max Backpressure, Max Timeout, Max Payload and Idle Timeout Support
- Automatic Ping / Pong Support
- Per Socket Data
- `Middlewares`
- `Templates` Support (examples with `Mako` and `Jinja2`)
- `ASGI Server`
- `WSGI Server`
- `Plugins/Extensions`

🔎 Upcoming Features

- In-Memory Cache Tools
- Fetch like API powered by libuv
- Async file IO powered by libuv
- Full asyncio integration with libuv
- SSGI Server spec and support
- RSGI Server support
- Full Http3 support
- `HPy` integration to better support `CPython`, `PyPy` and `GraalPython`
- Hot Reloading

А что их бенчмарки показывают…



Результаты

Если вы думали, что всё пойдет хорошо, то вы молодцы, но ЗРЯ

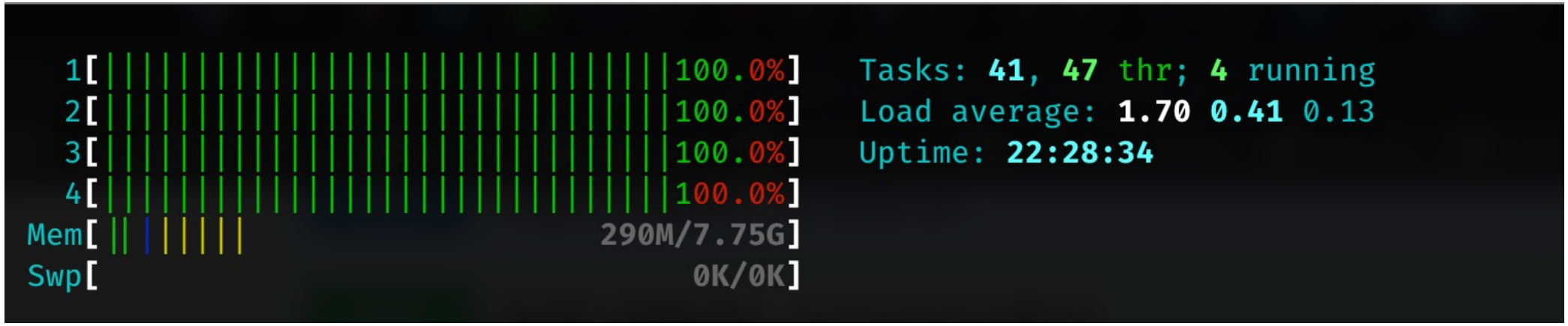

потрачено
потрачено
потрачено
ачено
чено
потра

В докере его не запустить никак сейчас

/usr/local/lib/python3.11/site-packages/socketify/libsocketify_linux_amd64.so: cannot open shared object file: No such file or directory. Additionally, ctypes.util.find_library() did not manage to locate a library called '/usr/local/lib/python3.11/site-packages/socketify/libsocketify_linux_amd64.so'

Но я решил, что шатнуть основы эксперимента и прогнал прям на сервере…

В процессе теста



Если честно, результаты такие, что слюнки текут

```
data_received..................: 38 MB  538 kB/s
data_sent......................: 11 MB  158 kB/s
http_req_blocked...............: avg=313.95µs min=0s      med=2µs     max=287.31ms p(90)=6µs     p(95)=9µs
http_req_connecting............: avg=307.81µs min=0s      med=0s      max=287.19ms p(90)=0s      p(95)=0s
http_req_duration..............: avg=35.32ms  min=14.14ms  med=23.12ms  max=3.72s    p(90)=36.55ms  p(95)=46.68ms
  { expected_response:true } ... : avg=35.32ms  min=14.14ms  med=23.12ms  max=3.72s    p(90)=36.55ms  p(95)=46.68ms
http_req_failed................: 0.00%  ✓ 0       ✗ 120421
http_req_receiving.............: avg=46.67µs  min=3µs     med=19µs    max=27.22ms  p(90)=49µs     p(95)=85µs
http_req_sending...............: avg=49.32µs  min=2µs     med=8µs     max=21.51ms  p(90)=29µs     p(95)=91µs
http_req_tls_handshaking.......: avg=0s       min=0s      med=0s      max=0s       p(90)=0s       p(95)=0s
http_req_waiting...............: avg=35.23ms  min=14.08ms  med=23.03ms  max=3.72s    p(90)=36.41ms  p(95)=46.55ms
http_reqs......................: 120421 1714.936389/s
iteration_duration.............: avg=237.52ms min=214.31ms med=225.19ms max=3.92s    p(90)=241.42ms p(95)=252.34ms
iterations.....................: 120421 1714.936389/s
vus............................: 109    min=5         max=999
vus_max........................: 1000   min=1000      max=1000
```

Результаты кратко

— 1.7к RPS
— 37ms latency p90

Ещё я хотел сделать: FastAPI + socketify.py asgi

Что?

Socketify.py ещё и asgi

```
Example:
    python3 -m socketify main:app -w 8 -p 8181
```

И:
Litestar + socketify.py asgi

Но решил, что не буду этого делать, т.к. условия эксперимента нарушились…

Что еще стоило бы протестировать? Blacksheep

Где и для чего подойдут какие фреймворки?

Для быстрых рестов

— Robyn? Поговорим о минусах
— Socketify? Уф, какая сложная идея

Для надежной и предсказуемой разработки

— Вы не поверите, всё ещё fastapi, но с granian!

— А я лично выбираю litestar с granian!

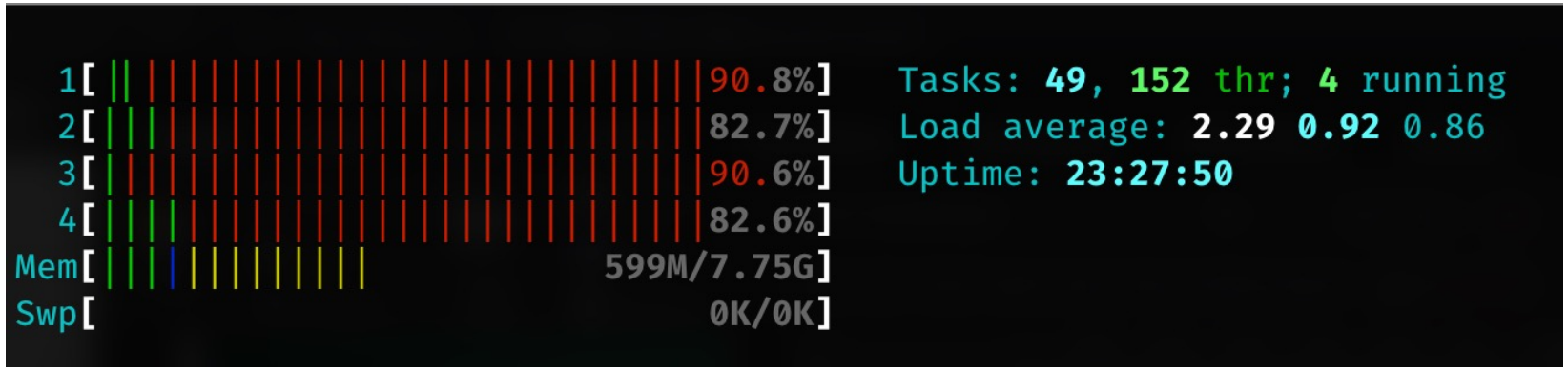
MsgSpec vs Pydantic?

Возьмем litestar и заменим pydantic на msgspec

Результаты

```
data_received..................: 12 MB  168 kB/s
data_sent......................: 3.2 MB 46 kB/s
http_req_blocked...............: avg=493.36µs min=0s      med=2µs     max=78.83ms p(90)=9µs    p(95)=14µs
http_req_connecting............: avg=484.64µs min=0s      med=0s      max=78.75ms p(90)=0s     p(95)=0s
http_req_duration..............: avg=627.93ms min=14.6ms  med=44.16ms max=2.74s   p(90)=1.78s p(95)=1.91s
  { expected_response:true } ...: avg=627.93ms min=14.6ms  med=44.16ms max=2.74s   p(90)=1.78s p(95)=1.91s
http_req_failed................: 0.00%  ✓ 0         ✗ 35115
http_req_receiving.............: avg=44.57µs  min=4µs     med=23µs    max=19.13ms p(90)=64µs   p(95)=103µs
http_req_sending...............: avg=35.55µs  min=2µs     med=9µs     max=15.09ms p(90)=37µs   p(95)=90µs
http_req_tls_handshaking.......: avg=0s       min=0s      med=0s      max=0s      p(90)=0s     p(95)=0s
http_req_waiting...............: avg=627.85ms min=14.56ms med=44.12ms max=2.74s   p(90)=1.78s p(95)=1.91s
http_reqs......................: 35115  498.28599/s
iteration_duration.............: avg=829.48ms min=214.84ms med=247.39ms max=2.96s   p(90)=1.98s p(95)=2.11s
iterations.....................: 35115  498.28599/s
vus............................: 107    min=5       max=999
vus_max........................: 1000   min=1000    max=1000
```

Что там по загрузке



Обнаружилась существенная корелляция

Я так и не смог это посчитать существенной разницей

Ссылки и исходный код

Ссылки/исходники

— https://github.com/xfenix/piterpy-imya-mne-skorost
— https://github.com/vibora-io/vibora
— https://github.com/squeaky-pl/japronto
— https://github.com/unum-cloud/ucall
— https://github.com/cirospaciari/socketify.py
— https://robyn.tech/
— https://jcristharif.com/msgspec/


iStock
by Getty Images

DGTL Raiffeisen

Спасибо! 🏃
Задавайте ответы!

Денис Аникин
https://xfenix.ru
https://github.com/xfenix/