Python over Go

Kharitonskiy Vitaliy
Prom.ua

Images are important

- Eye candy
- Click-bait
- Small Details
- Text in images
- Marketing value
- Seeing is believing

Images are not cheap

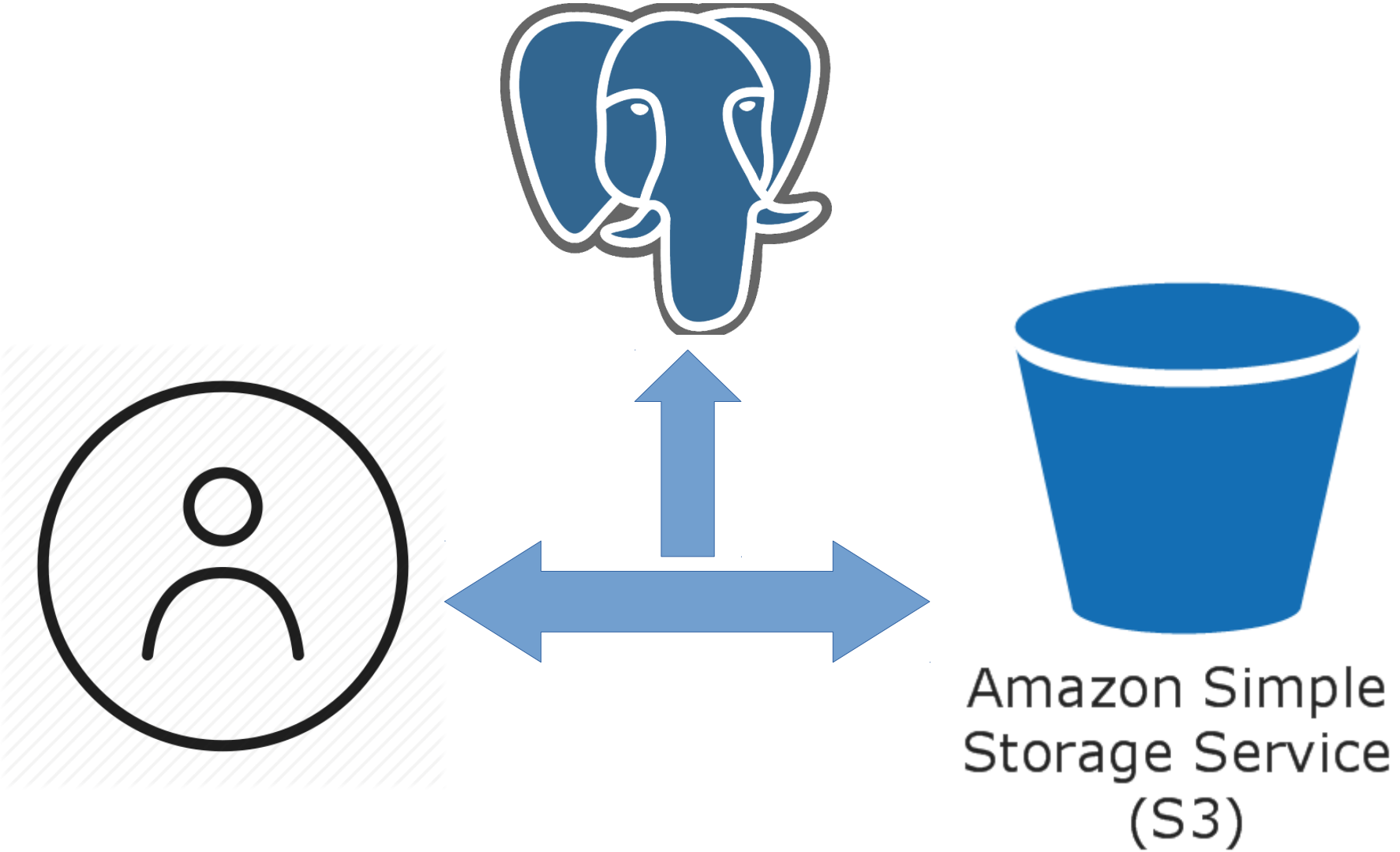
- Thousands of new images each day
- Storage/traffic worth $$$
- Largest bandwidth consumer per page viewed, important for mobile site/app.
- Image processing is hard.


TransloadIt
Uploadcare
BLITLINE
IMAGE PROCESSING IN THE CLOUD
<cloudimage.io/>
PIXPIE
imgIX
ReSRC.it
Kraken.io
IMAGERESIZER
Cloudinary
thumbr.io
RESIZOR
WURFL
Sirv

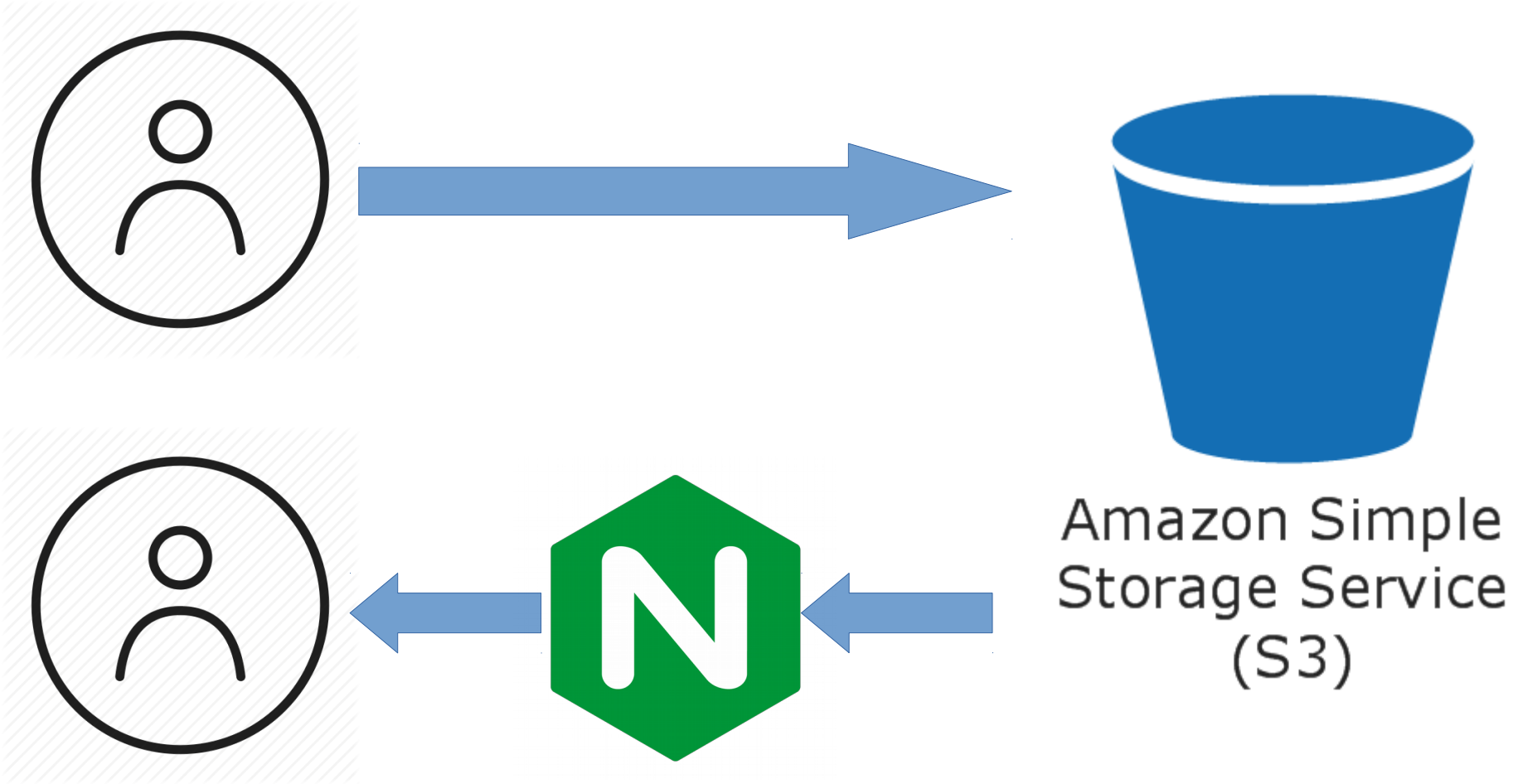
Some numbers

- 6k images per second served at top load
- 120 Tb in Amazon
- 60 Tb in local cache
- 29 Riak nodes
- 6 nginx cache nodes
- 40 python processes across ~30 nodes for resize & watermark

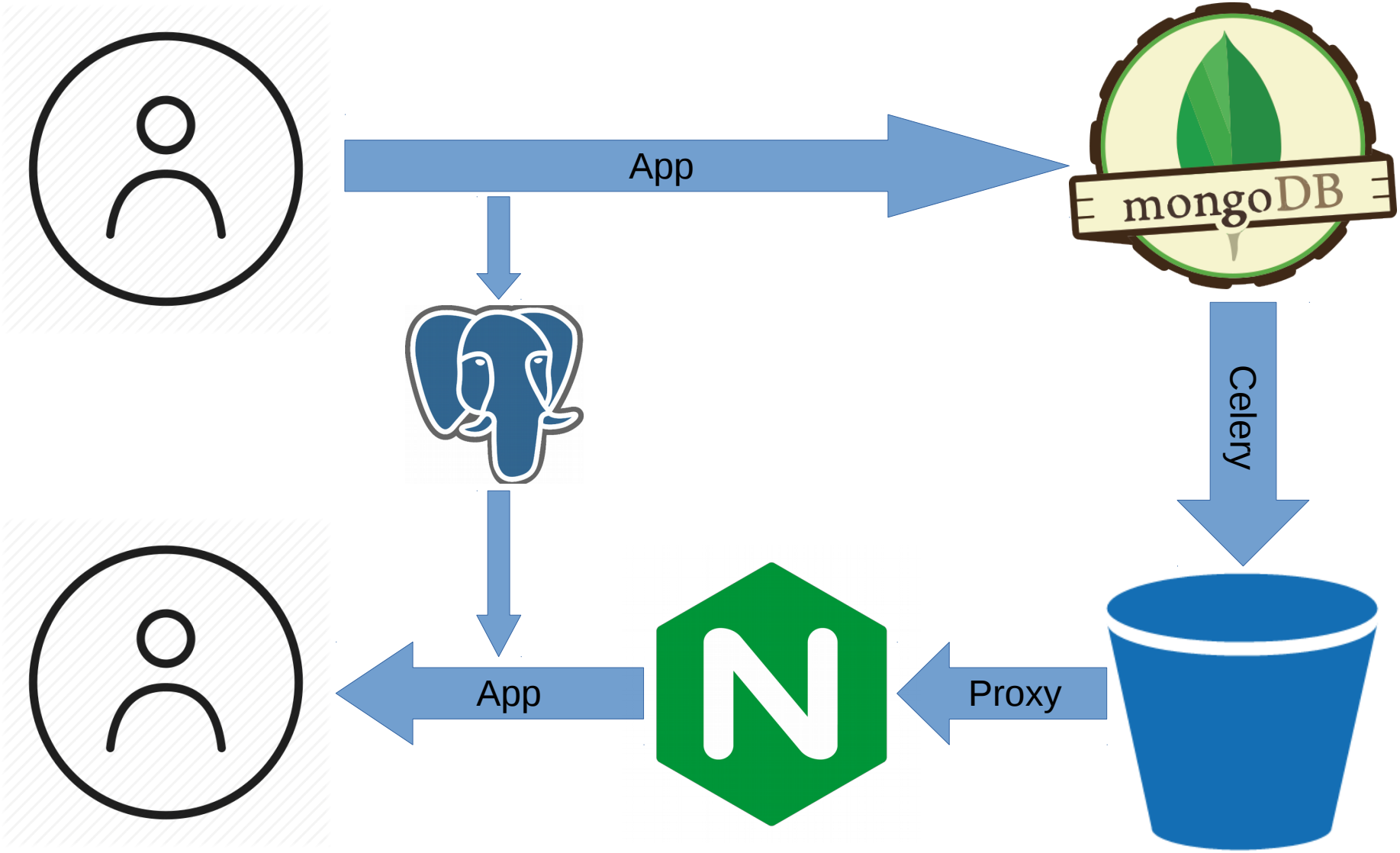
Evolution of content delivery@Prom



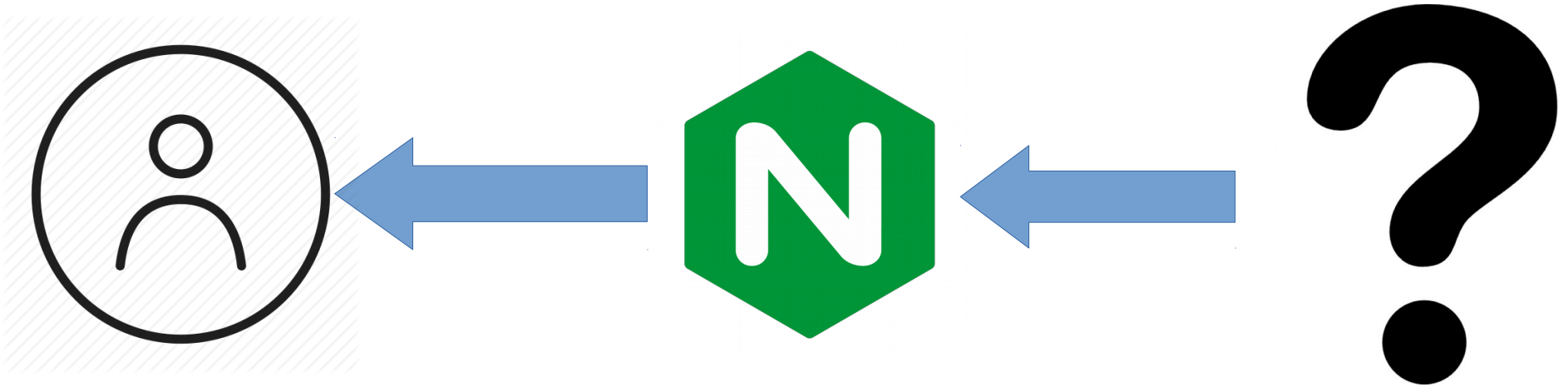
Evolution of content delivery@Prom



Evolution of content delivery@Prom



Evolution of content delivery@Prom

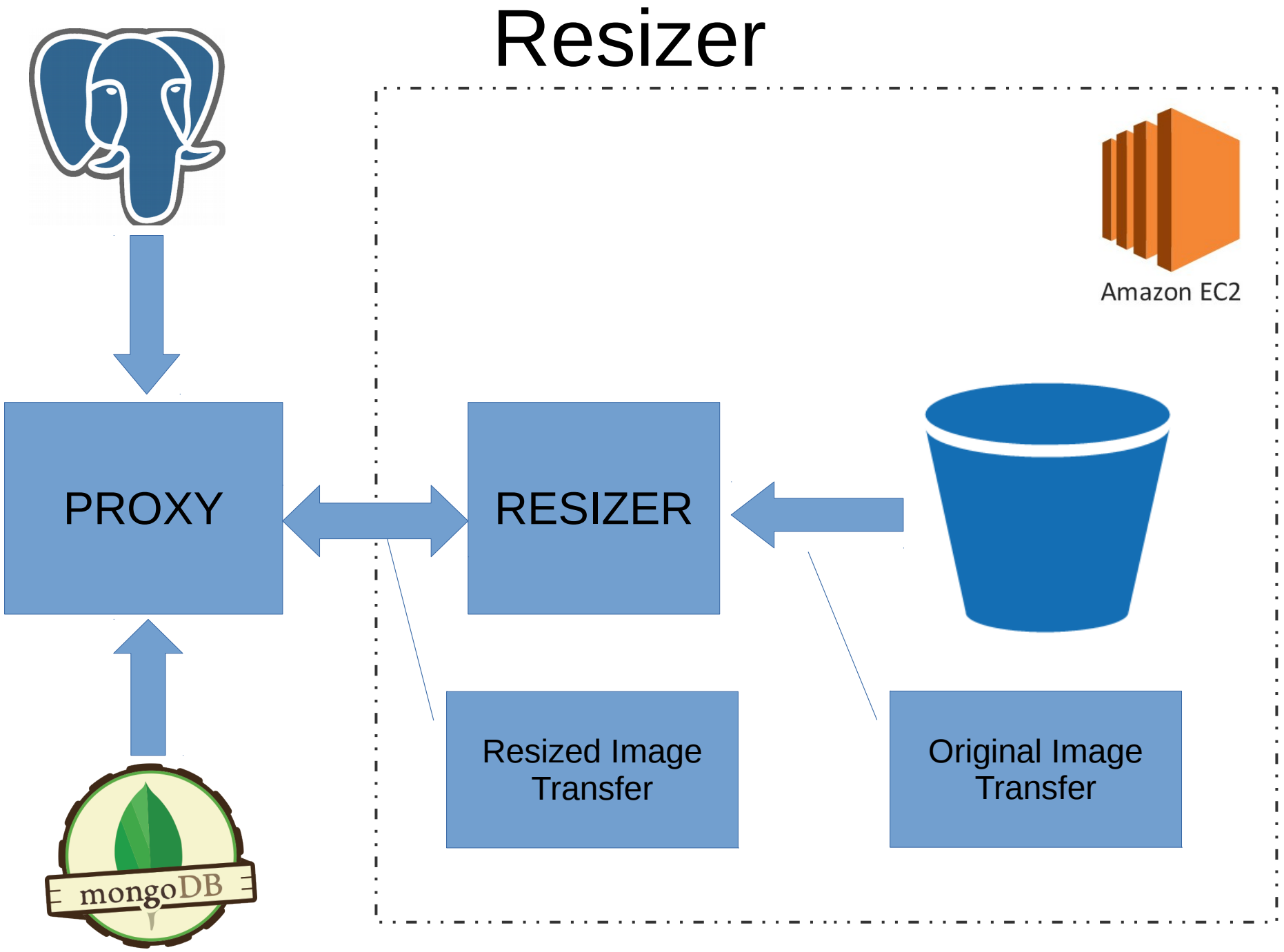


Application Requirements

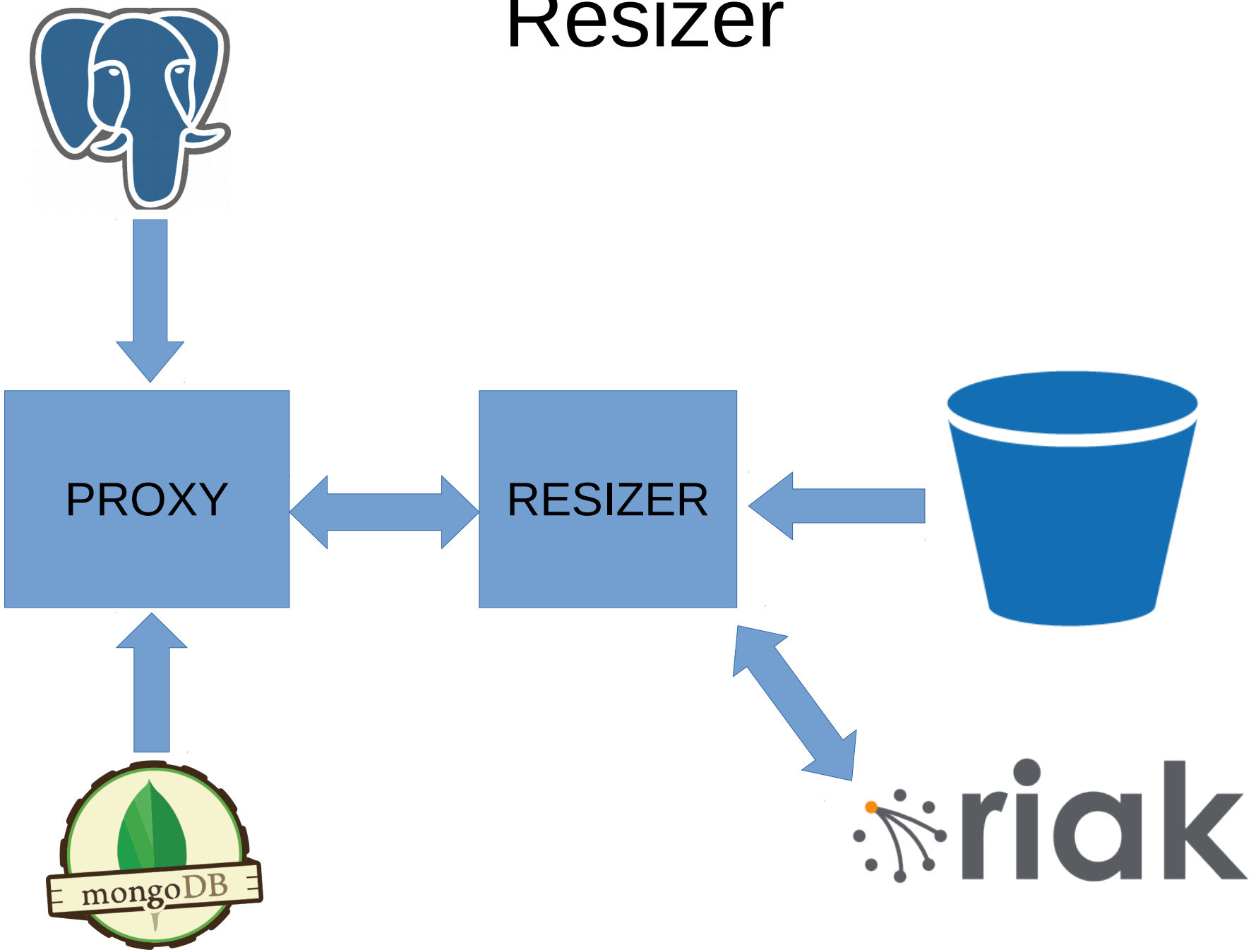
- Multiple data sources
- Unreliable local storage/frequent outages
- 80% IO / 20% CPU
- Reliable operations/deployment
- Shares logic with main app
- High load

Why Golang?

- So hype, much fast, wow
- NOT Python
- Golang IS the new Python
- Because D’oh


Resizer
Amazon EC2
PROXY
RESIZER
Resized Image Transfer
Original Image Transfer
mongoDB

Resizer



The Good

- Compiles to one binary, good for pre-container deployment
- HTTP is fast, Proxy part is stable
- Static compilation: no tests, zero downtime after deploy
- Gophers

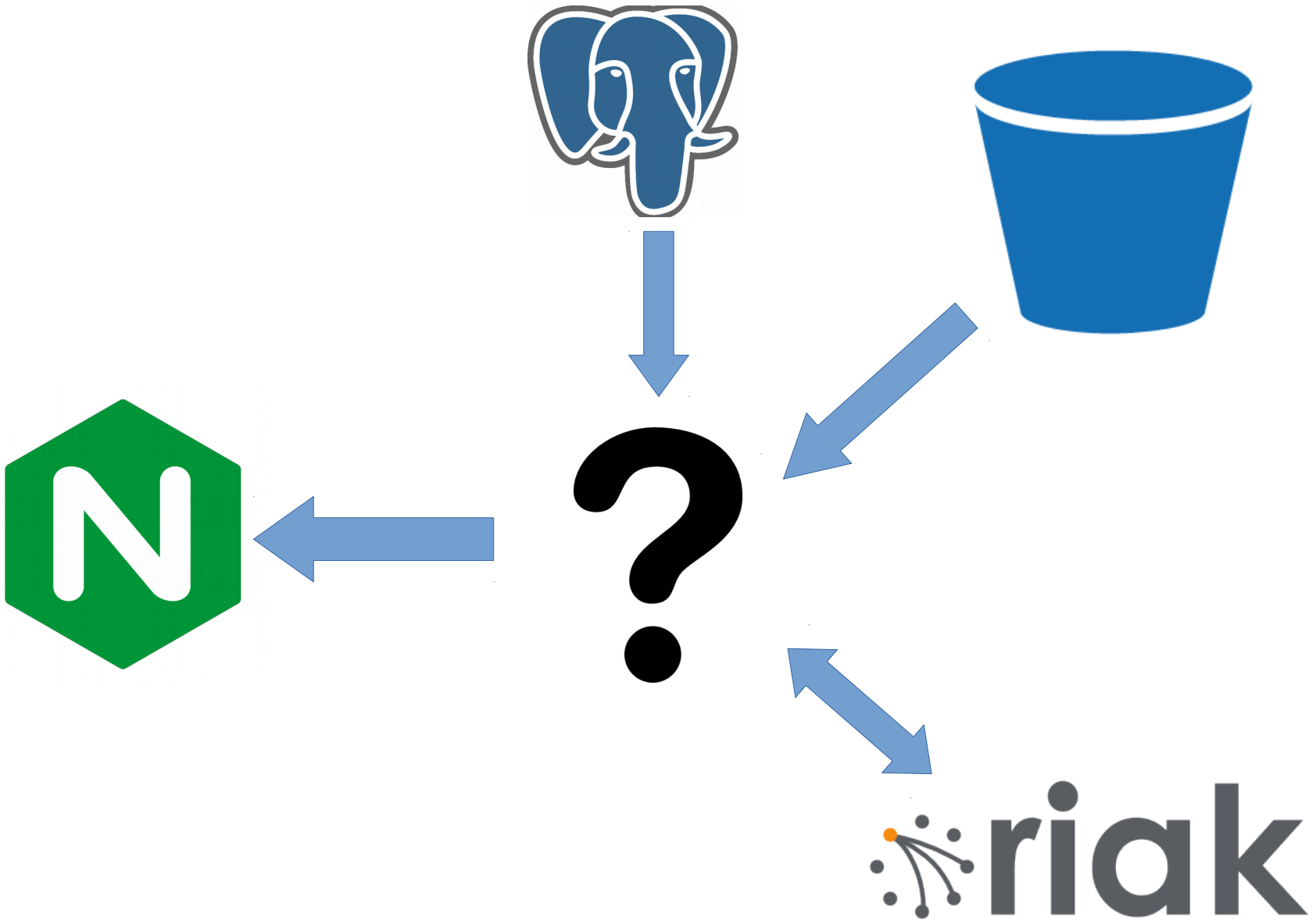
The Bad

- ImageMagick is not Pillow
- ImageMagick is slow
- Memory leaks in ImageMagick wrapper
- Weird concurrency/thread behavior
- Connection leaks everywhere
- Poor library/wrapper support(S3)
- DIY everywhere

The Ugly

- return null, err
- if err == null
- if err == ‘Not found’
- import “hell”
- Global Library Vars
- go func(){go func(){ go func yourself!!

Simplified



Microbenchmark(GO)

```go
package main
import (
  "github.com/gographics/imagick"
  "io/ioutil"
)

func main() {
  blob, _ := ioutil.ReadFile("yuuuge.jpg")
  for i:=0; i<100; i++{
        mw := imagick.NewMagickWand()
        mw.ReadImageBlob(blob)
        mw.ResizeImage(200, 200, imagick.FILTER_LANCZOS2, 0.5)
        mw.Destroy()
  }
}
```

Microbenchmark(Python)

```python
from PIL import Image
from io import BytesIO

data = BytesIO(open('yuuuge.jpg', 'rb').read())

for i in range(100):
    with Image.open(data) as image:
        image.resize((200, 200)).save('result_resize.jpg', 'JPEG')
```

Results
4712x3328 JPG resize to 200x200
100 passes

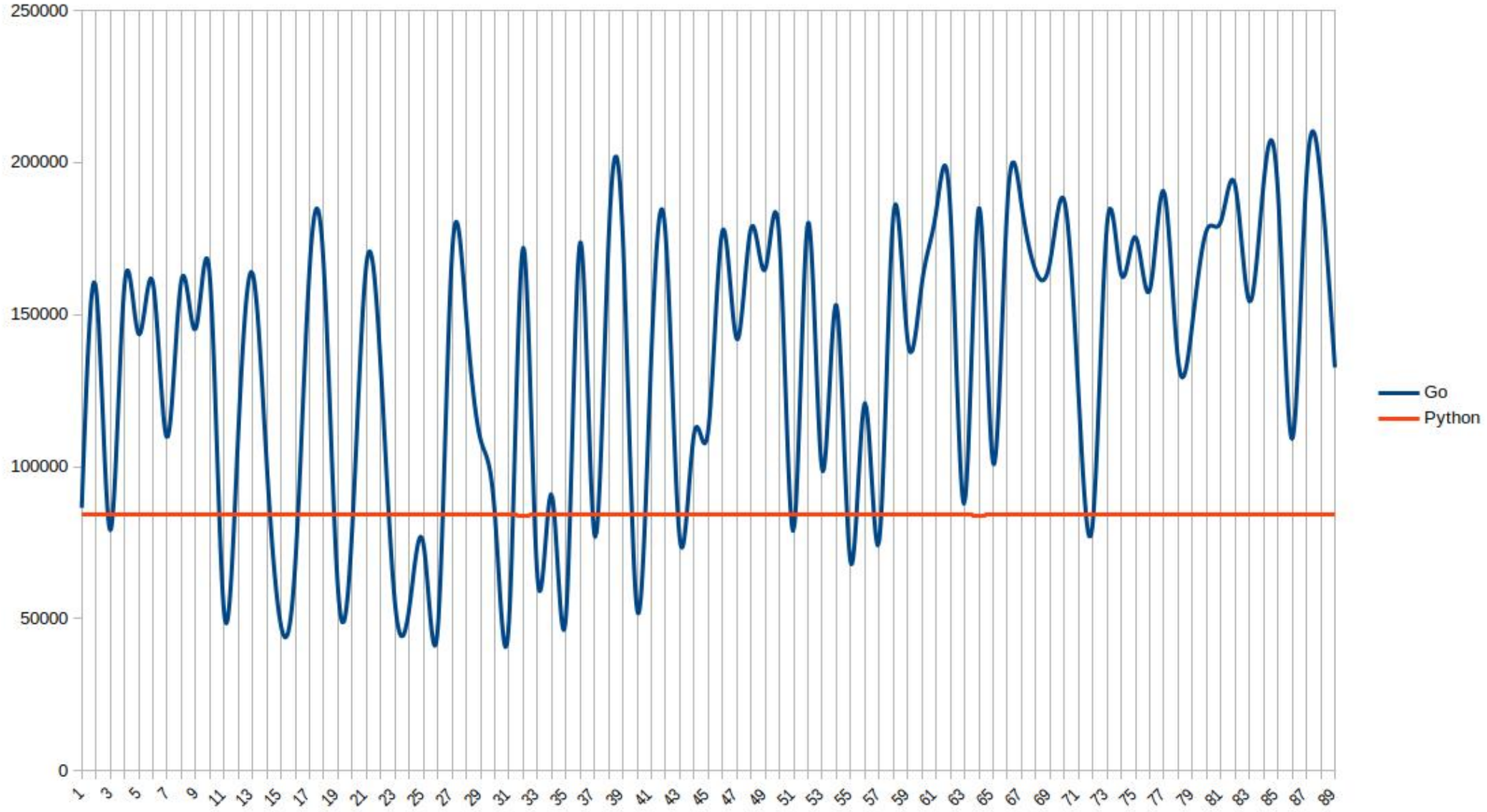
PYTHON

28s total
280 ms per pass
3.5 resizes per second

GOLANG

30s total
300 ms per pass
3 resizes per second

Results


250000
200000
150000
100000
50000
0
1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41 43 45 47 49 51 53 55 57 59 61 63 65 67 69 71 73 75 77 79 81 83 85 87 89
Go
Python

Why Python?

- AsyncIO for content proxy
- Pillow means compatibility with App
- Zero bus factor
- No more Mr. return err

The Good

- PIL is fast
- PIL does not leak memory
- Simple, understandable concurrency behaviour
- Great libraries for everything(aiobotocore, aiopg, motor)
- Readable code

The Bad

- Deployment is more complex(solved with containers)
- No static compilation(solved with tests/integration tests)
- CPU blocks IO(solved with more python processes)

The Ugly

The Real Ugly

- Request retry policy in botocore is configured via json file inside package folder

  https://github.com/boto/botocore/blob/develop/botocore/data/_retry.json

The Real Ugly

- AsyncIO.sleep until loop is empty in case of delayed tasks inside handler

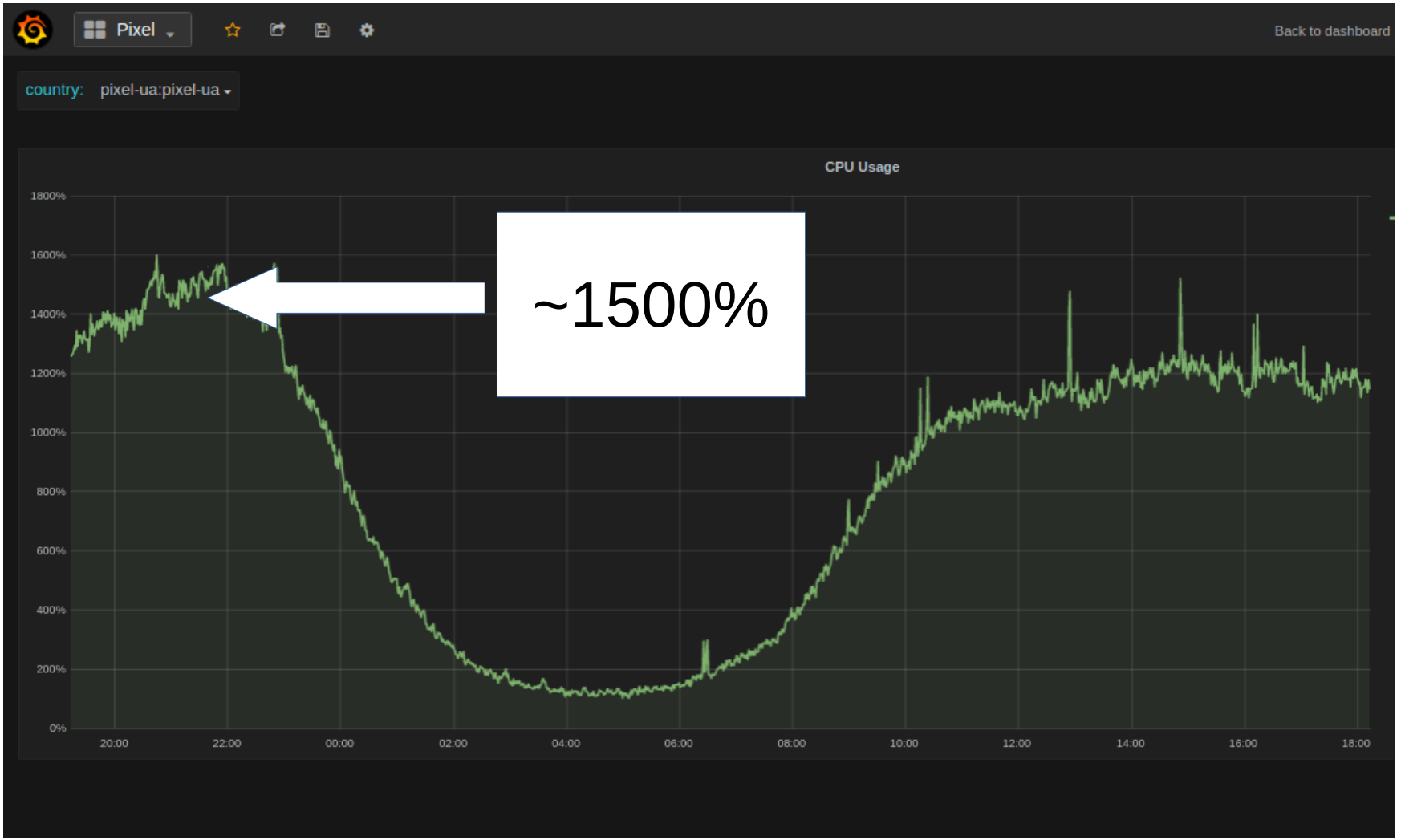
```
async def test_empty_image_fill(cli):
    resp = await cli.get('/7_w200_h200_test_empty.jpg')
    body = await resp.read()
    assert len(body) != 0
    assert resp.headers.get('X-Image-Source') == 'Resized original: local'
    # Wait until thumbnail is transferred to local cache
    await asyncio.sleep(0.4)
```

CPU(BEFORE)

5000% = 50 cores = 8~9 servers at max load

CPU(AFTER)

1500% = 15 cores = 2~3 servers at max load



Memory(BEFORE)

Memory(AFTER)

How we test

- pytest
- pytest-aiohttp
- GitLab CI
- Vagga - https://github.com/tailhook/vagga
- PostgreSQL
- MongoDB
- MinioGo
- ZERO Mocks

How we deploy

- GitLab CI
- Vagga - https://github.com/tailhook/vagga
- Lithos - https://github.com/tailhook/lithos
- Verwalter - https://github.com/tailhook/verwalter

How we measure

- Cantal - https://github.com/tailhook/cantal
- Carbon
- Graphite
- Graphana
- Logstash
- Kibana

What is measured

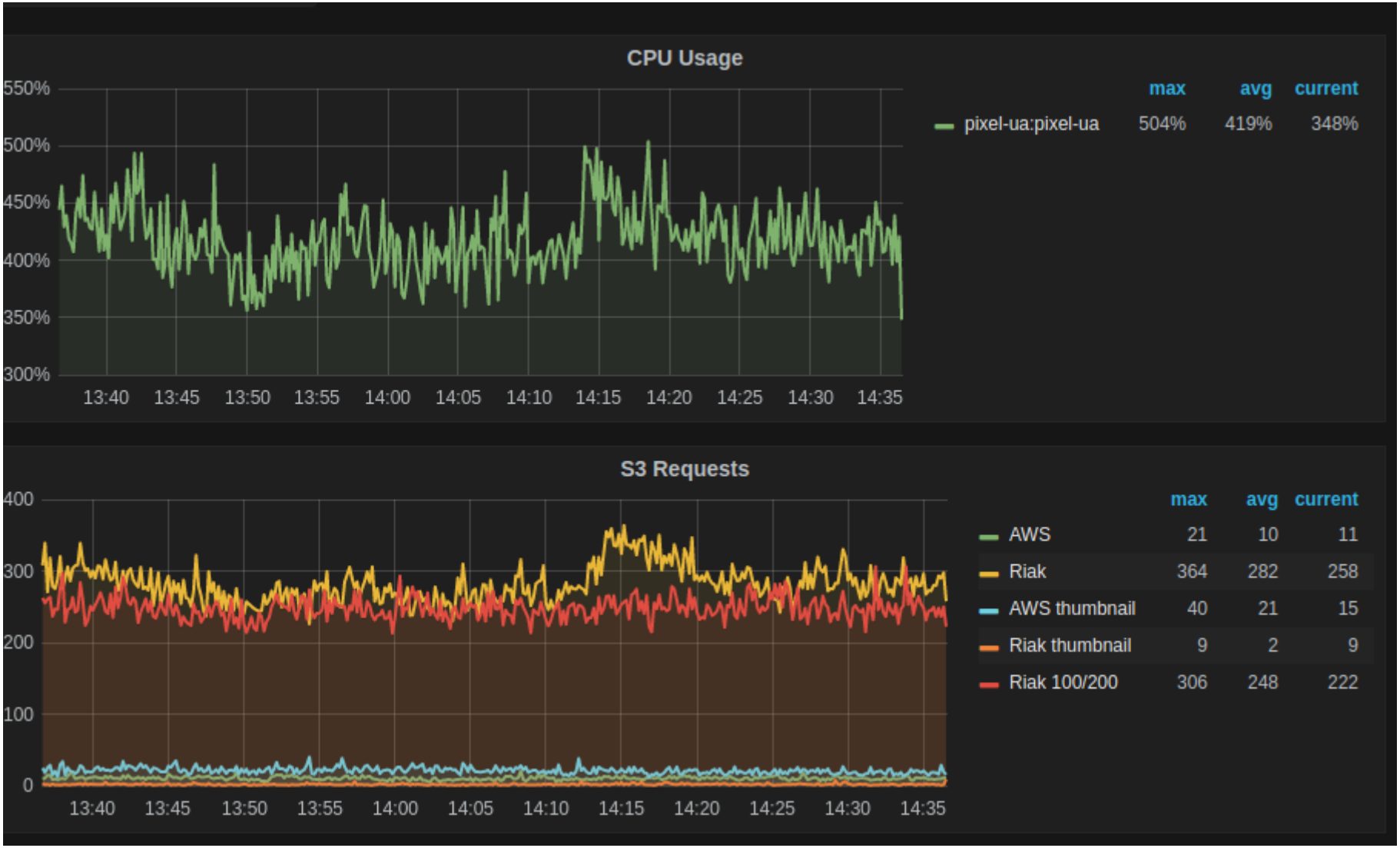
- CPU
- Memory
- S3 Request time/count
- Riak Request time/count
- DB Request time/count
- Number of resizes
- Number of watermarks
- Number of hot/cold cache responses

Measurement Rule of Thumb

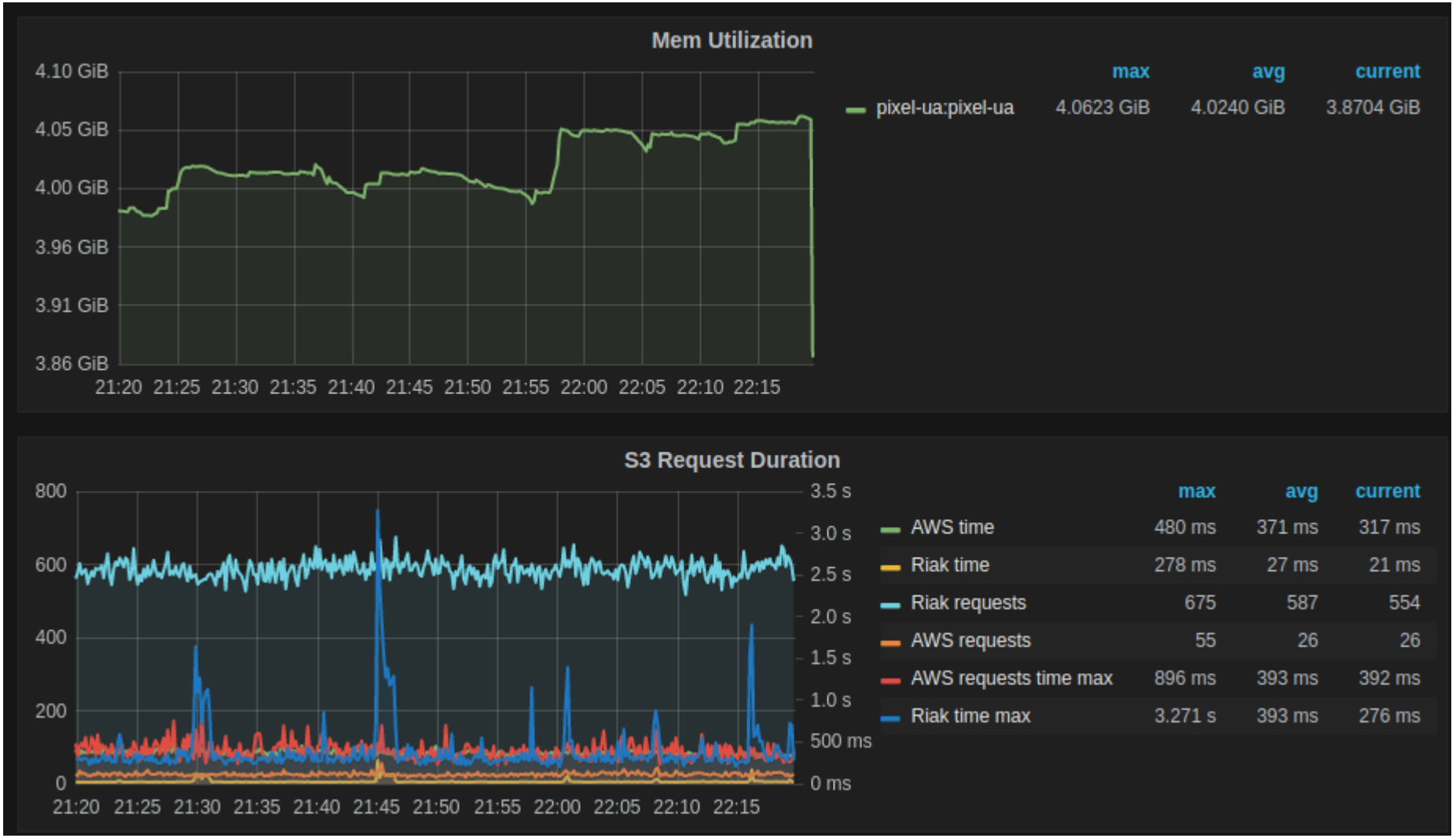
What I do not see on the dashboard is probably not happening right now. Same applies to errors.

Confucius

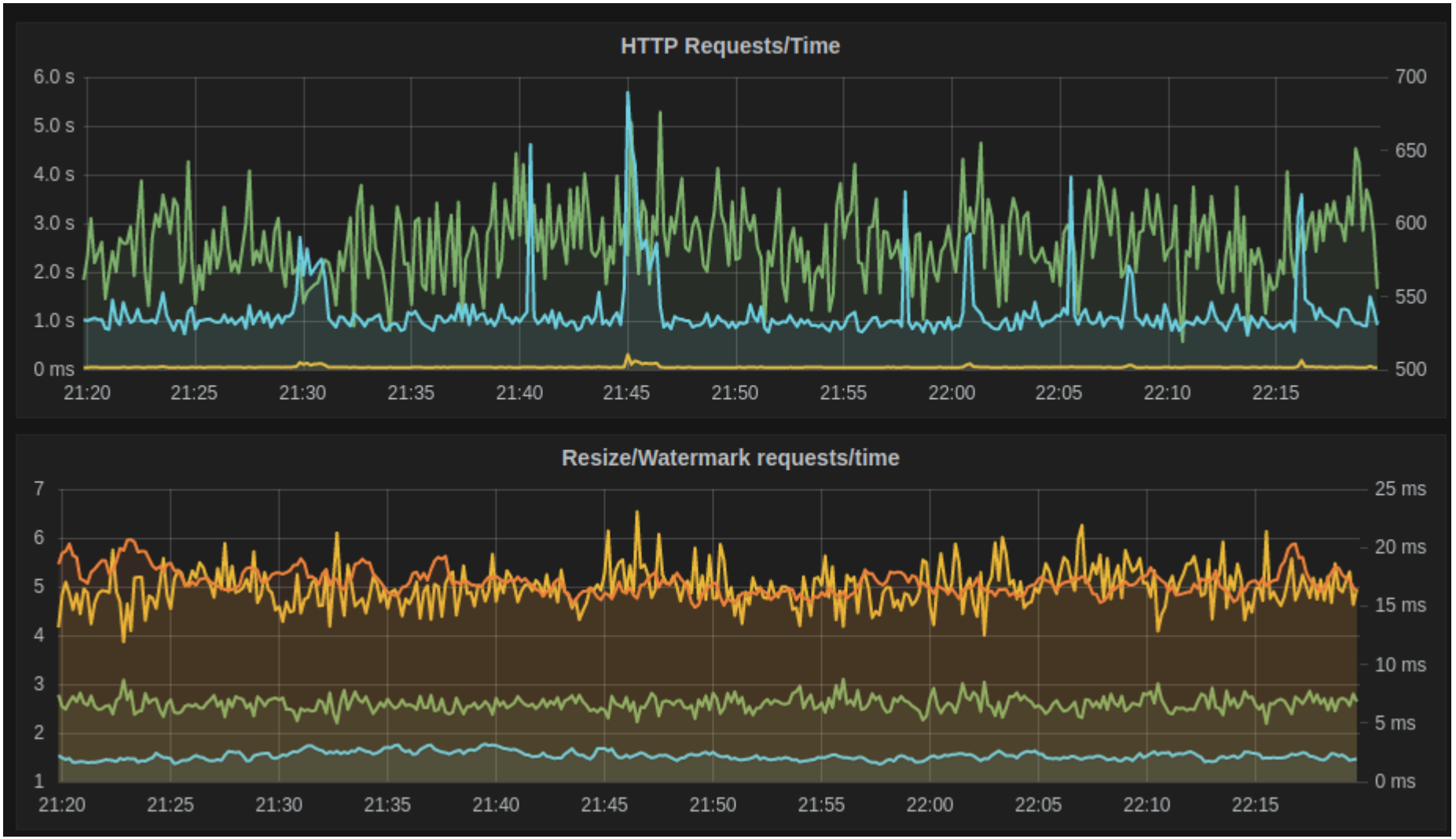
Dashboard


CPU Usage
550%
500%
450%
400%
350%
300%
13:40
13:45
13:50
13:55
14:00
14:05
14:10
14:15
14:20
14:25
14:30
14:35
max
avg
current
pixel-ua:pixel-ua
504%
419%
348%
S3 Requests
400
300
200
100
0
13:40
13:45
13:50
13:55
14:00
14:05
14:10
14:15
14:20
14:25
14:30
14:35
max
avg
current
AWS
21
10
11
Riak
364
282
258
AWS thumbnail
40
21
15
Riak thumbnail
9
2
9
Riak 100/200
306
248
222

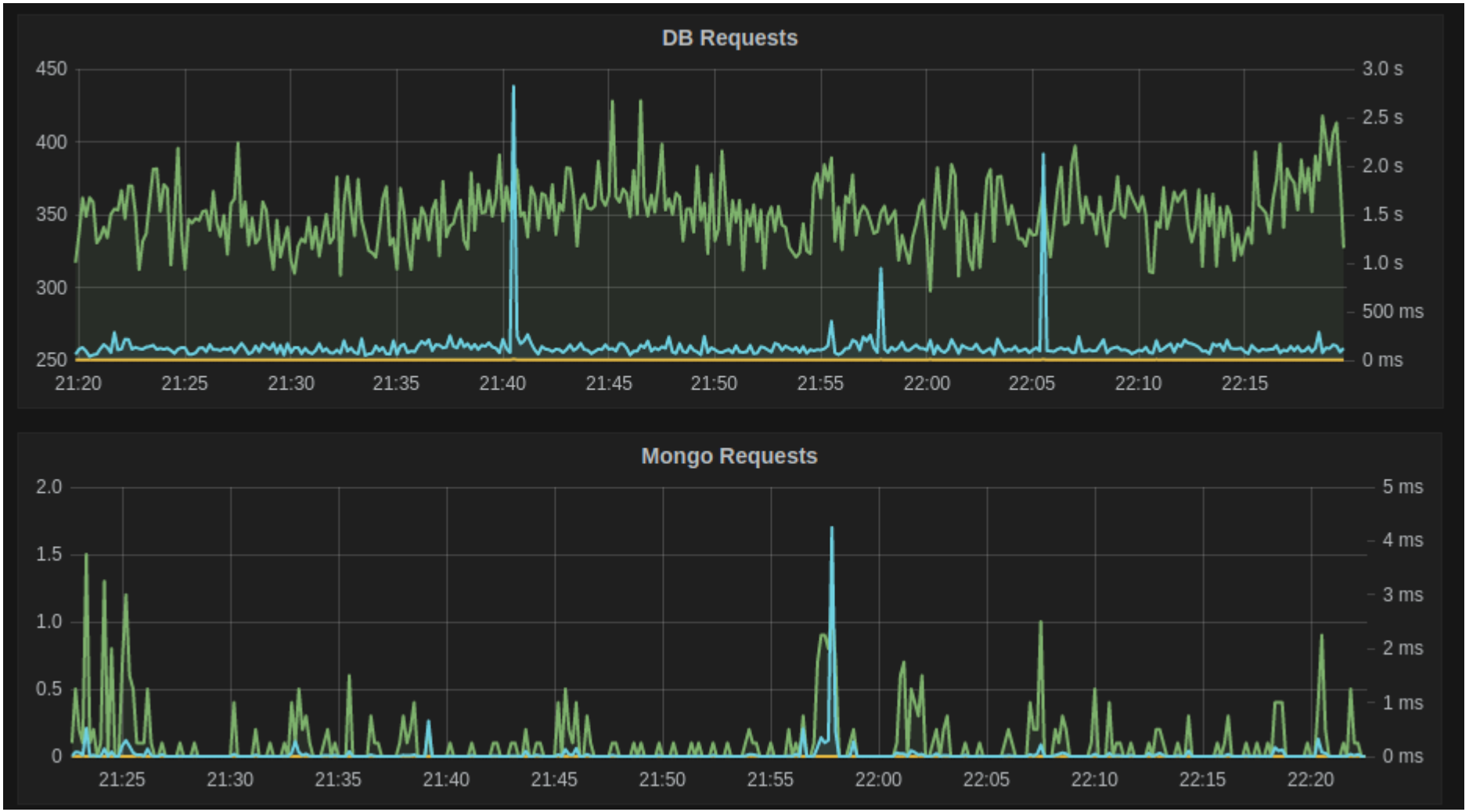
Dashboard


Mem Utilization
4.10 GiB
4.05 GiB
4.00 GiB
3.96 GiB
3.91 GiB
3.86 GiB
21:20 21:25 21:30 21:35 21:40 21:45 21:50 21:55 22:00 22:05 22:10 22:15
max avg current
pixel-ua:pixel-ua 4.0623 GiB 4.0240 GiB 3.8704 GiB
S3 Request Duration
800
600
400
200
0
3.5 s
3.0 s
2.5 s
2.0 s
1.5 s
1.0 s
500 ms
0 ms
21:20 21:25 21:30 21:35 21:40 21:45 21:50 21:55 22:00 22:05 22:10 22:15
max avg current
AWS time 480 ms 371 ms 317 ms
Riak time 278 ms 27 ms 21 ms
Riak requests 675 587 554
AWS requests 55 26 26
AWS requests time max 896 ms 393 ms 392 ms
Riak time max 3.271 s 393 ms 276 ms

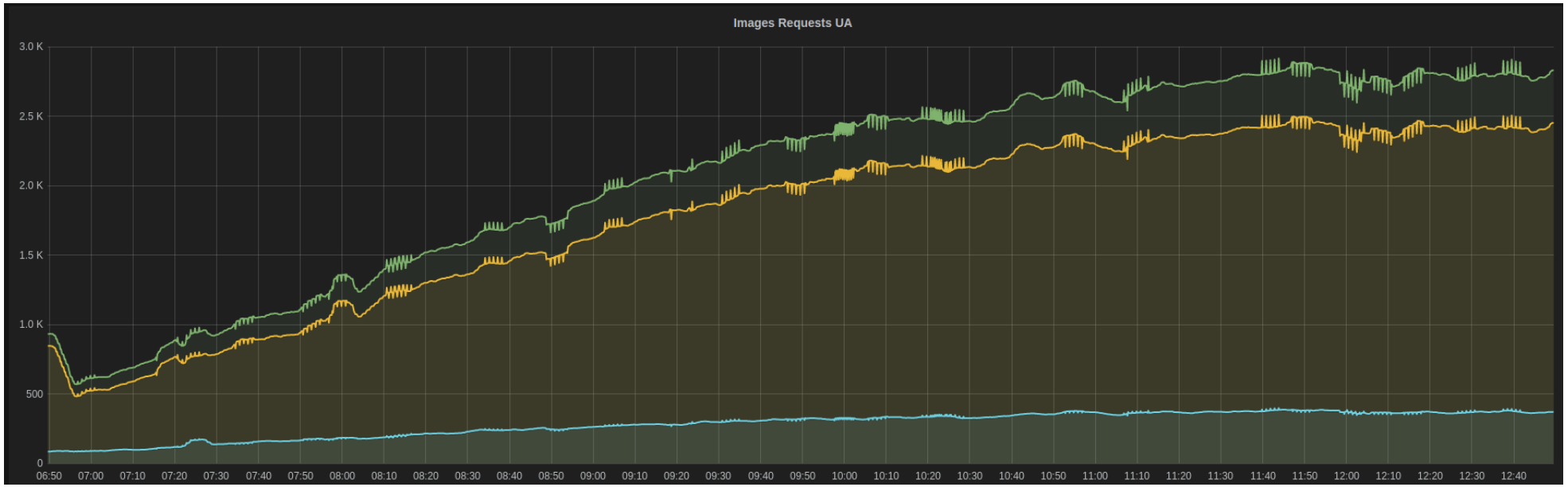
Dashboard


HTTP Requests/Time
6.0 s
5.0 s
4.0 s
3.0 s
2.0 s
1.0 s
0 ms
700
650
600
550
500
21:20
21:25
21:30
21:35
21:40
21:45
21:50
21:55
22:00
22:05
22:10
22:15
Resize/Watermark requests/time
7
6
5
4
3
2
1
25 ms
20 ms
15 ms
10 ms
5 ms
0 ms
21:20
21:25
21:30
21:35
21:40
21:45
21:50
21:55
22:00
22:05
22:10
22:15

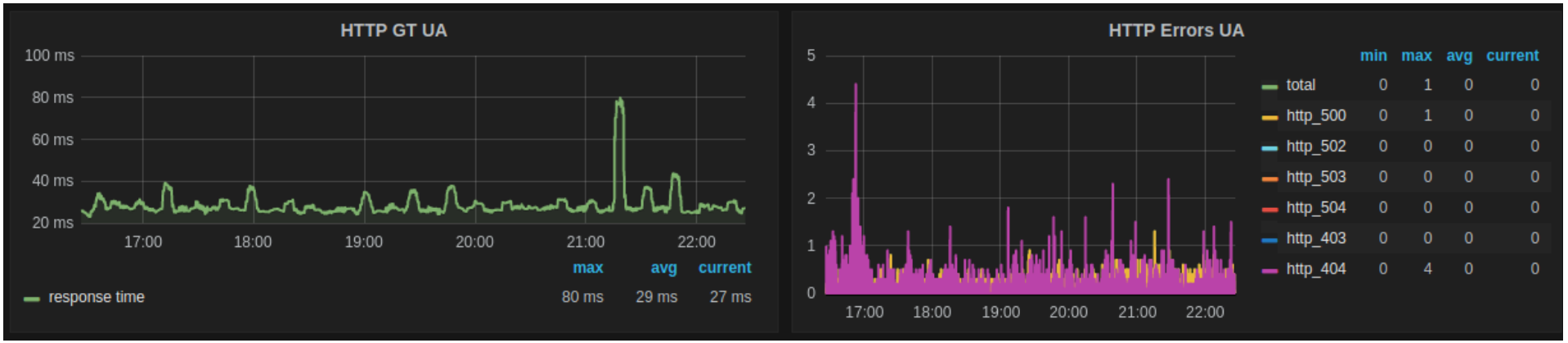
Dashboard


DB Requests
450
400
350
300
250
3.0 s
2.5 s
2.0 s
1.5 s
1.0 s
500 ms
0 ms
21:20
21:25
21:30
21:35
21:40
21:45
21:50
21:55
22:00
22:05
22:10
22:15
Mongo Requests
2.0
1.5
1.0
0.5
0
5 ms
4 ms
3 ms
2 ms
1 ms
0 ms
21:25
21:30
21:35
21:40
21:45
21:50
21:55
22:00
22:05
22:10
22:15
22:20

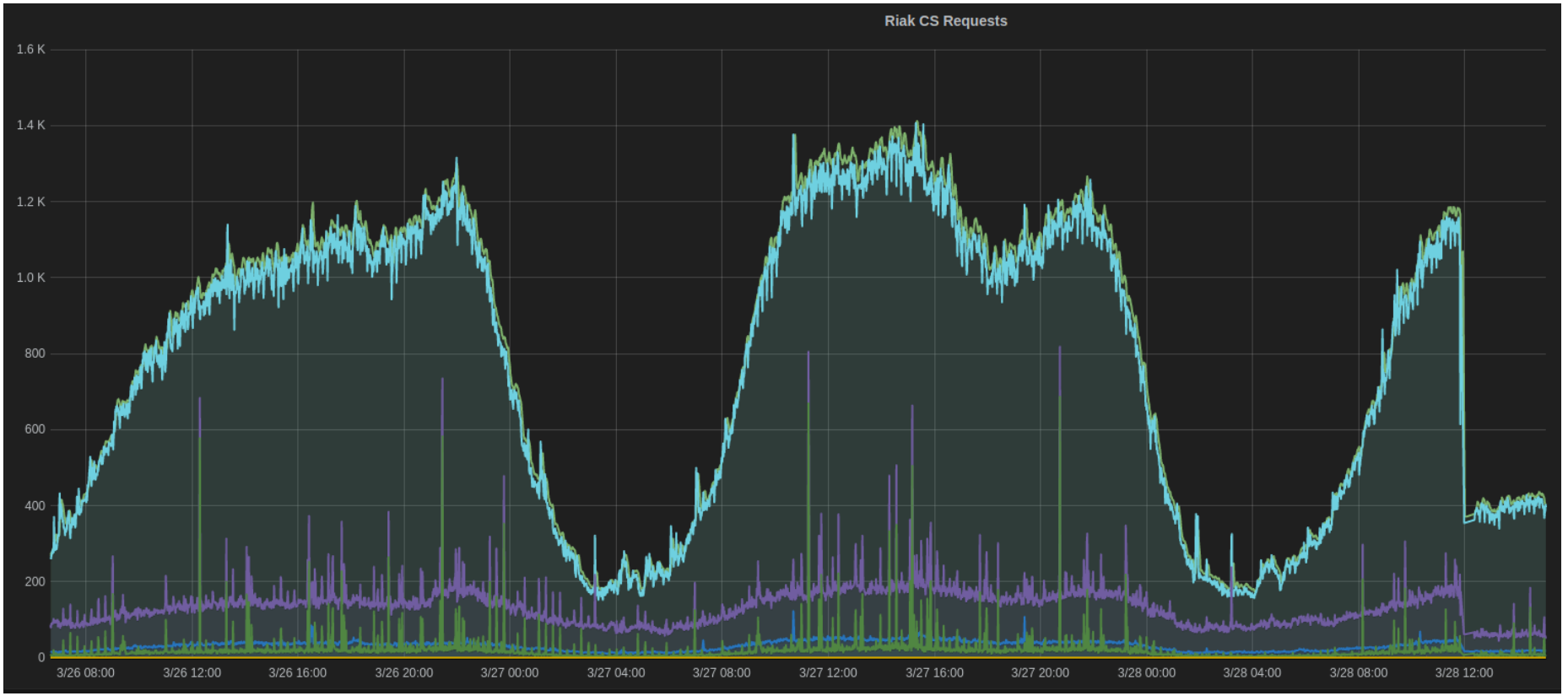
Dashboard


Images Requests UA
3.0 K
2.5 K
2.0 K
1.5 K
1.0 K
500
0
06:50
07:00
07:10
07:20
07:30
07:40
07:50
08:00
08:10
08:20
08:30
08:40
08:50
09:00
09:10
09:20
09:30
09:40
09:50
10:00
10:10
10:20
10:30
10:40
10:50
11:00
11:10
11:20
11:30
11:40
11:50
12:00
12:10
12:20
12:30
12:40

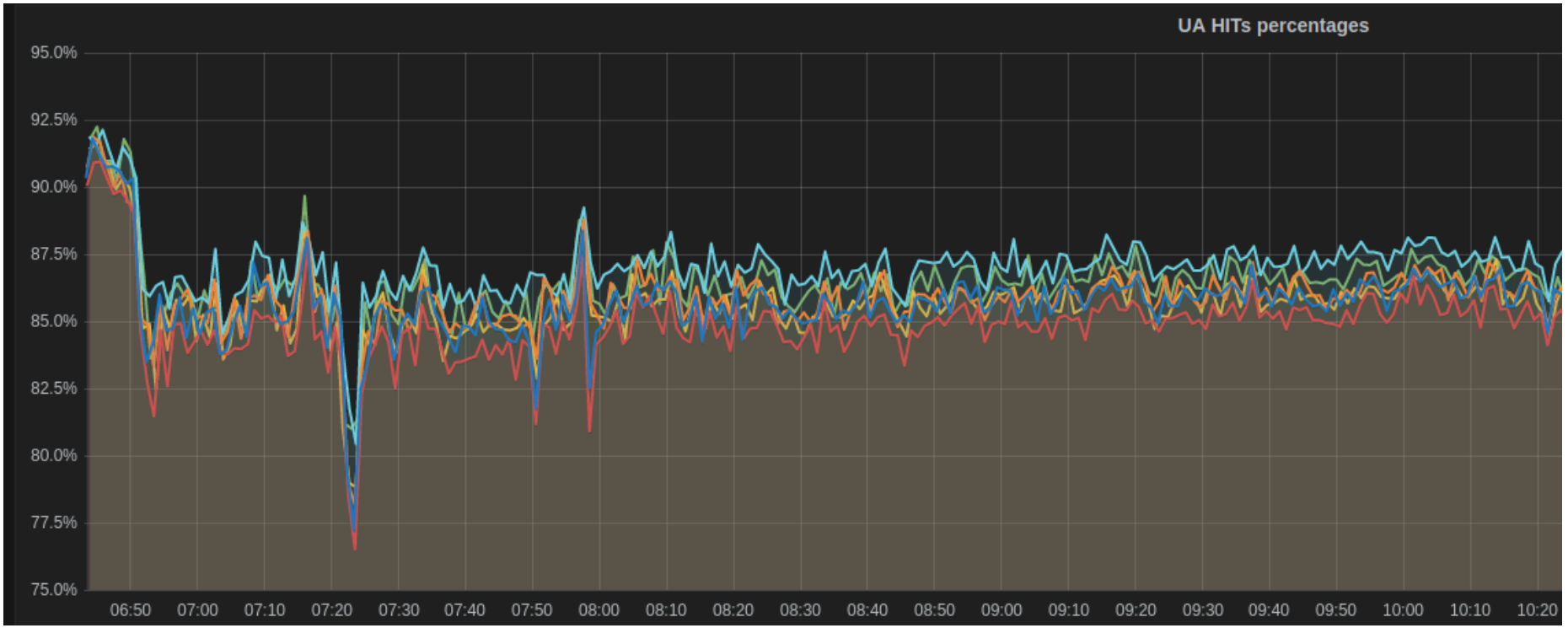
Dashboard


HTTP GT UA
100 ms
80 ms
60 ms
40 ms
20 ms
17:00
18:00
19:00
20:00
21:00
22:00
max
avg
current
response time
80 ms
29 ms
27 ms
HTTP Errors UA
5
4
3
2
1
0
17:00
18:00
19:00
20:00
21:00
22:00
min
max
avg
current
total 0 1 0 0
http_500 0 1 0 0
http_502 0 0 0 0
http_503 0 0 0 0
http_504 0 0 0 0
http_403 0 0 0 0
http_404 0 4 0 0

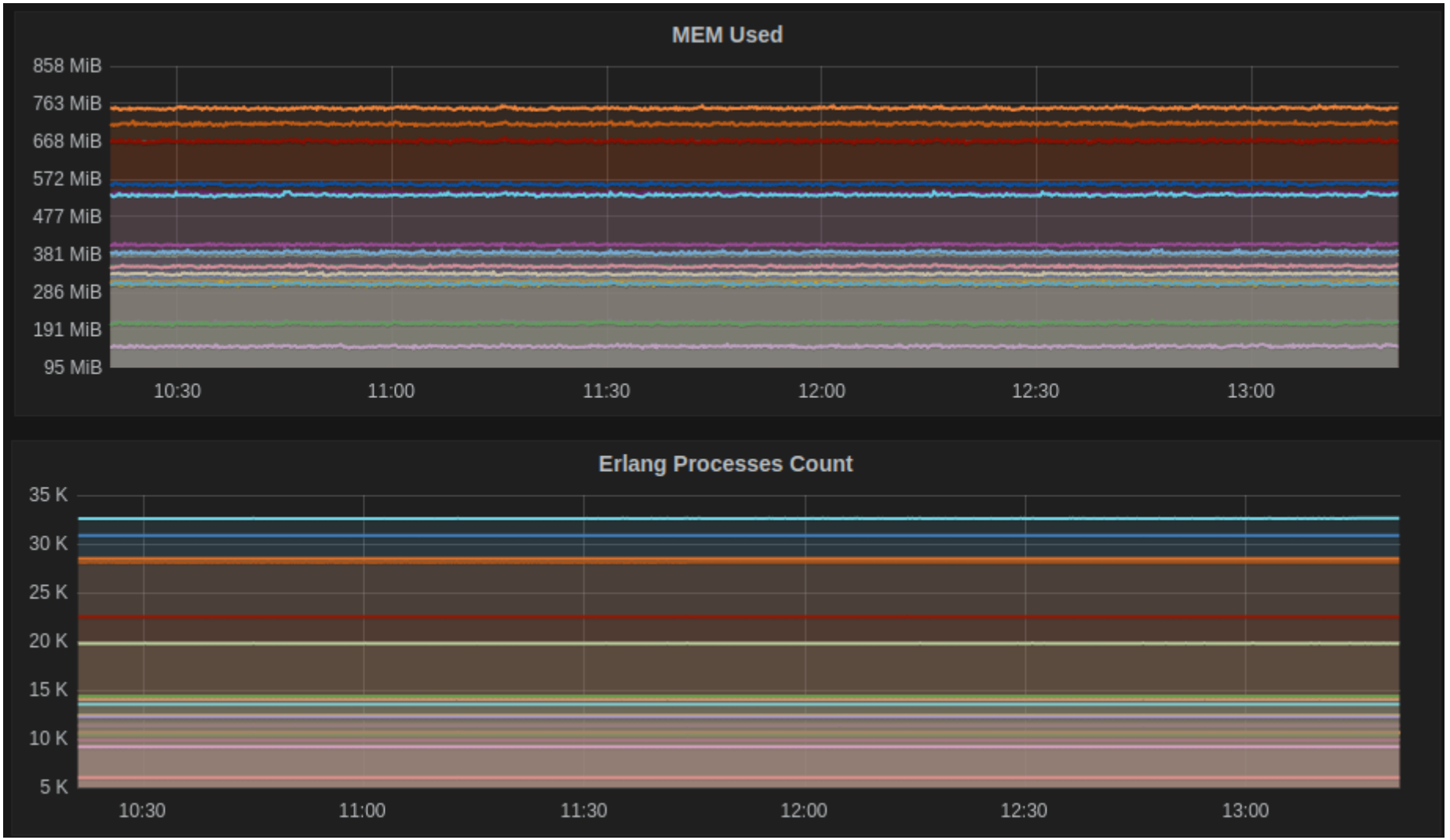
Dashboard


Riak CS Requests
1.6 K
1.4 K
1.2 K
1.0 K
800
600
400
200
0
3/26 08:00
3/26 12:00
3/26 16:00
3/26 20:00
3/27 00:00
3/27 04:00
3/27 08:00
3/27 12:00
3/27 16:00
3/27 20:00
3/28 00:00
3/28 04:00
3/28 08:00
3/28 12:00

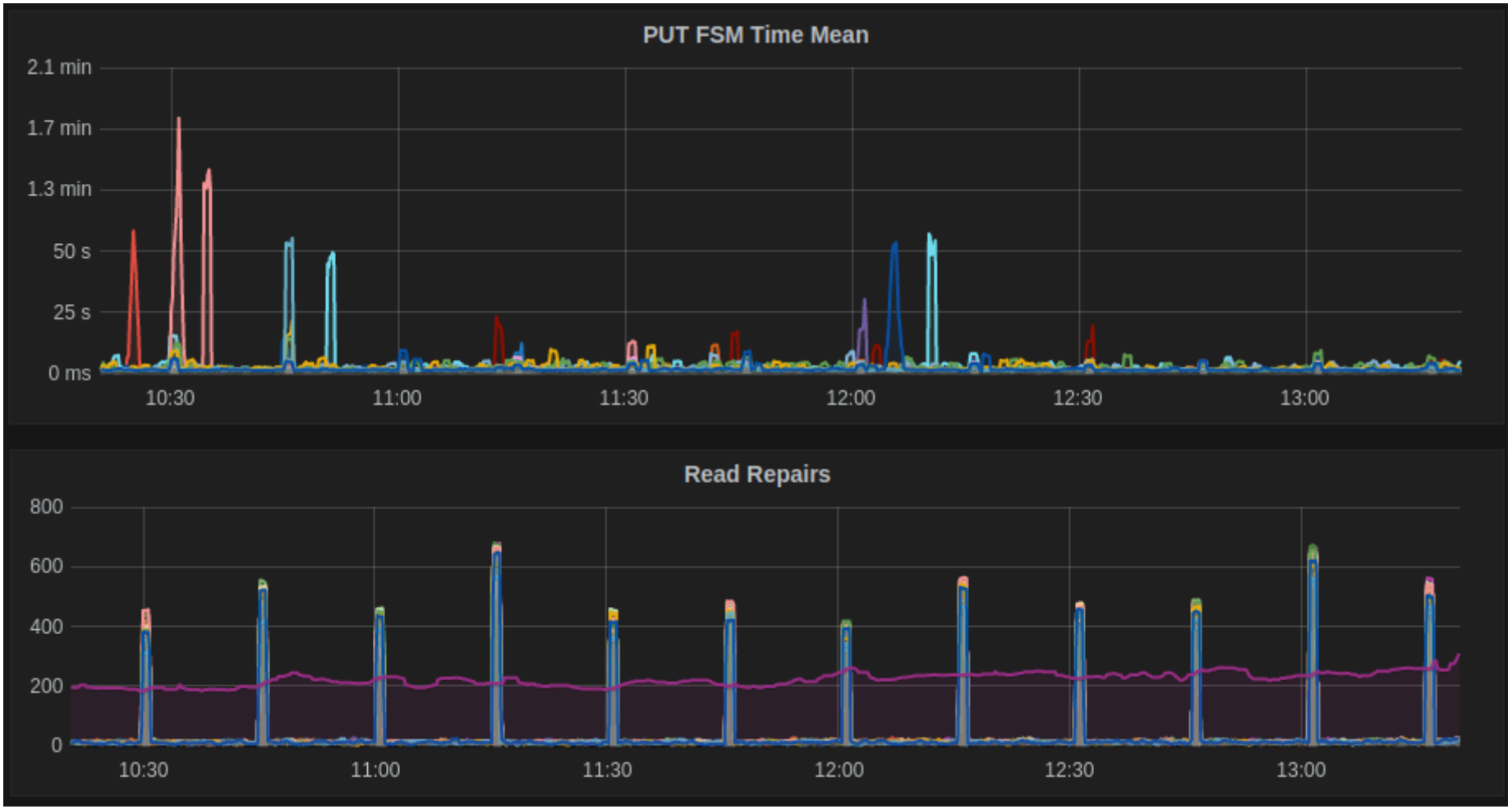
Dashboard


UA HITs percentages
95.0%
92.5%
90.0%
87.5%
85.0%
82.5%
80.0%
77.5%
75.0%
06:50
07:00
07:10
07:20
07:30
07:40
07:50
08:00
08:10
08:20
08:30
08:40
08:50
09:00
09:10
09:20
09:30
09:40
09:50
10:00
10:10
10:20

Dashboard


MEM Used
858 MiB
763 MiB
668 MiB
572 MiB
477 MiB
381 MiB
286 MiB
191 MiB
95 MiB
10:30
11:00
11:30
12:00
12:30
13:00
Erlang Processes Count
35 K
30 K
25 K
20 K
15 K
10 K
5 K
10:30
11:00
11:30
12:00
12:30
13:00

Dashboard


PUT FSM Time Mean
2.1 min
1.7 min
1.3 min
50 s
25 s
0 ms
10:30
11:00
11:30
12:00
12:30
13:00
Read Repairs
800
600
400
200
0
10:30
11:00
11:30
12:00
12:30
13:00

Lessons learned

- Optimize the path for most popular size
- Read the code
- Go safe(aiopg vs asyncpg)
- Retries are evil
- Timeout every IO
- Inspect every IO operation for avg/max time
- DB connection for each request, use PgBouncer, use semaphore

Tricks

- Do not touch the image
- Calculate format with mimetype, not with PIL
- Use Image.draft when possible
- Use Pillow SIMD with AXV2 compilation for speed
- Calculate source image quality before compressing the destination
- Nginx consistent hashing

Pillow SIMD

- drop-in replacement for Pillow
- follows Pillow versions very closely
- can be compiled with AVX or AVX2
- very fast for resize, repack

https://github.com/uploadcare/pillow-simd

Microbenchmark

```python
from PIL import Image
from io import BytesIO

data = BytesIO(open('yuuuge.jpg', 'rb').read())

for i in range(100):
    with Image.open(data) as image:
        image.resize((200, 200)).save('result_resize.jpg', 'JPEG')
```

Pillow SIMD vs Pillow
4712x3328 JPG resize to 200x200
100 passes

SIMD

18s total
180 ms per pass
5.6 resizes per second

VANILLA

28s total
280 ms per pass
3.5 resizes per second

Image.draft

- configure image reader to read resized thumbnail
- very fast as no Python code is involved
- keeps aspect ratio
- result size can be close to desired

Microbenchmark

```python
from PIL import Image
from io import BytesIO

data = BytesIO(open('yuuuge.jpg', 'rb').read())

for i in range(100):
    with Image.open(data) as image:
        image.draft('RGB', (200, 200))
        image.save('result_draft.jpg', 'JPEG')
```

Draft vs Resize
4712x3328 JPG resize to 200x200
100 passes

DRAFT

12s total
12 ms per pass
83 resizes per second

RESIZE

28s total
280 ms per pass
3.5 resizes per second

Image quality

- You have a 60 kb JPEG
- You open it with Pillow
- You save it to JPEG with quality=100
- What size will the result file have ?

IT DEPENDS!

IT DEPENDS!

- Quality is not quality, it is a compression factor
- Given a JPEG with Q=75, compress it with Q=100, you get a bigger image
- Calculate quality with magic tables:

  https://github.com/vharitonsky/imagequality
- Compress image with quality=’keep’ if it is already compressed (Q~=60-75)

Dyakuyu!

Embarrassing questions, embarrassing comments, libvips recommendations, etc.

Dyakuyu!

Embarrassing questions, embarrassing comments, libvips recommendations, etc.