DECOMPOSITION OF THE MAIN THREAD IN NODE.JS TO INCREASE THROUGHPUT

HELLO!

I am Nikolay Matvienko

JS Developer at Grid Dynamics

You can find me at twitter.com/**matvi3nko**

github.com/**matvi3nko**

NODE.JS IN ENTERPRISE

- IS WIDELY USED
- HIGHLOAD ON BLACK FRIDAY
- COMPETITION IN THE BUSINESS
  - GROWTH OF PERFORMANCE REQUIREMENTS
  - GROWTH OF FUNCTIONALITY/FEATURES

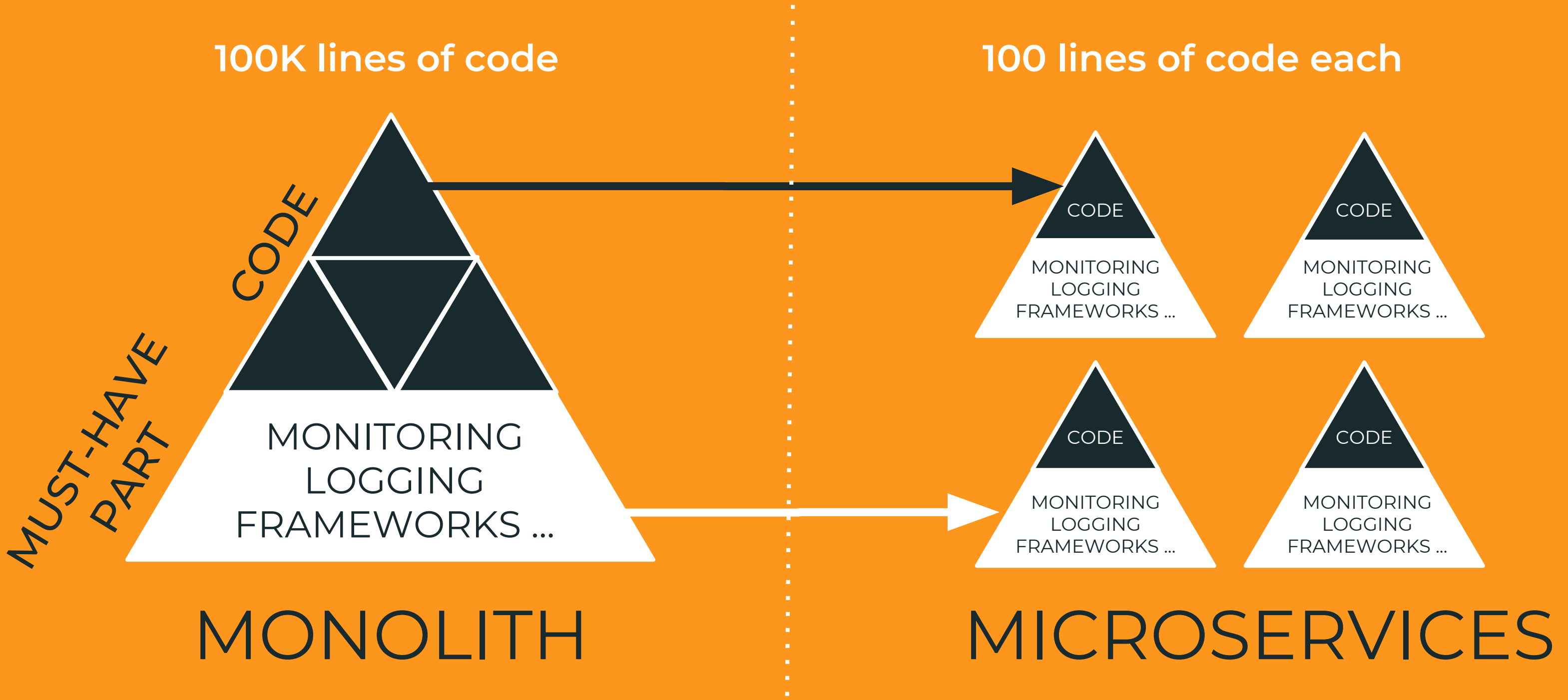
“WHAT IS FAST TODAY WILL BE SLOW TOMORROW AS DEMANDS CAN ONLY GROW”

THE PROBLEM

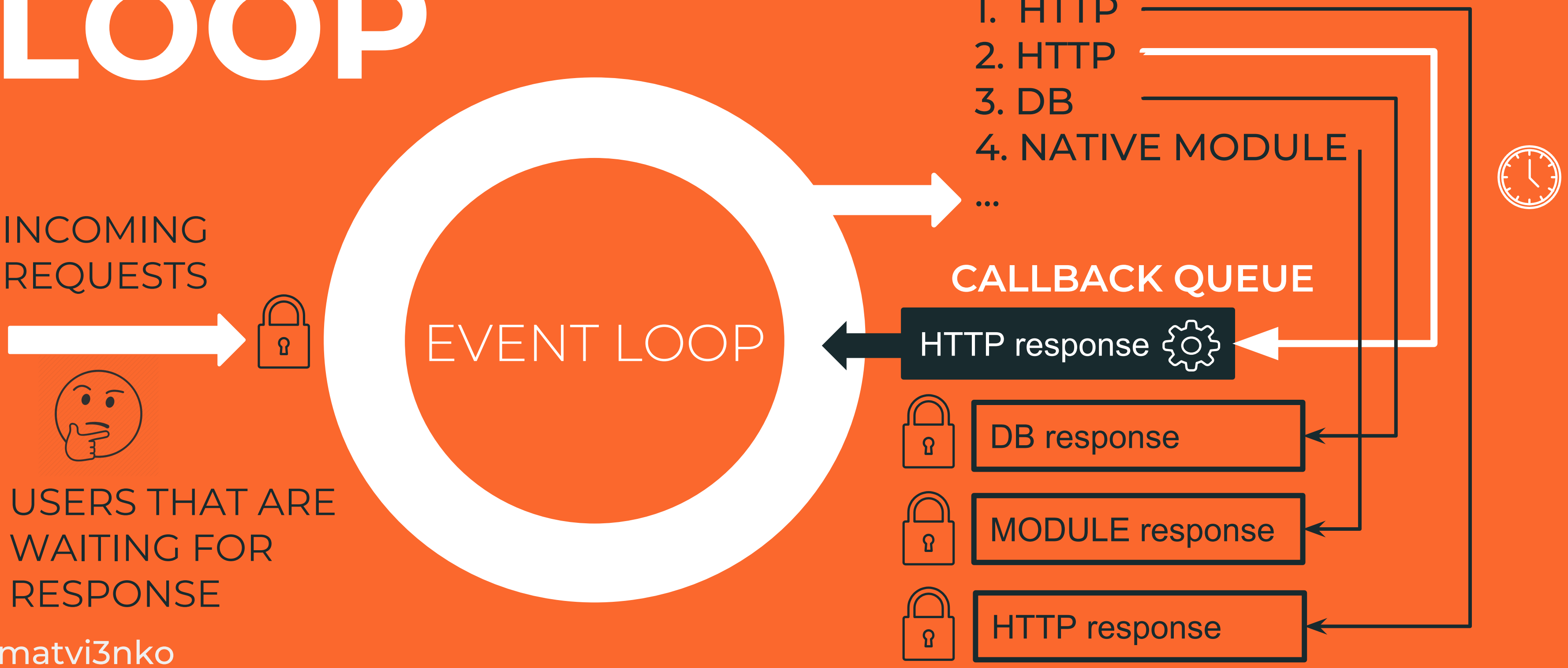
HEAVY LOAD OF

NODE.JS MAIN THREAD

MUST-HAVE PART

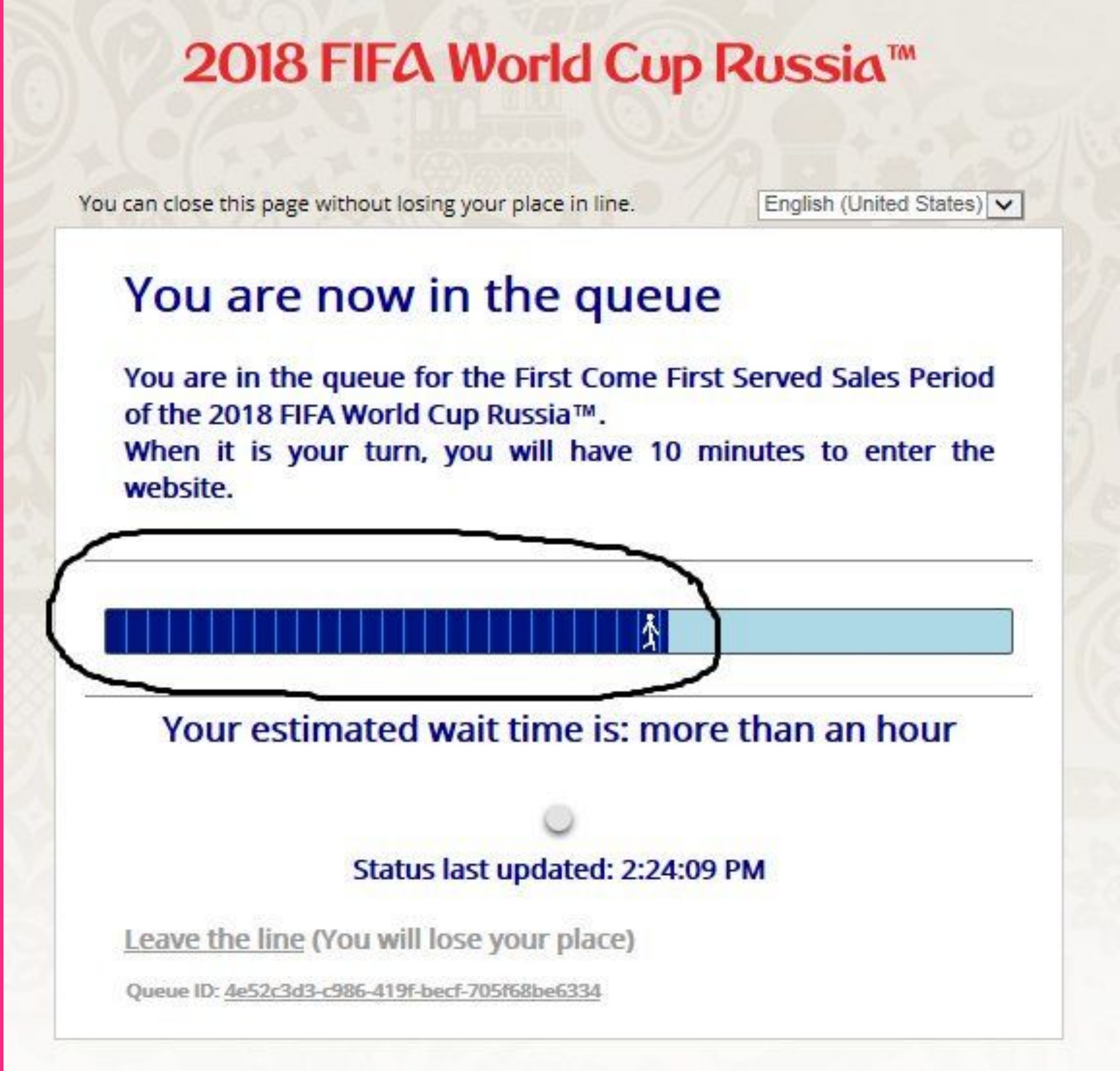


BLOCKED EVENT LOOP



@matvi3nko

IN THE QUEUE



2018 FIFA World Cup Russia™

You can close this page without losing your place in line.

English (United States)

You are now in the queue

You are in the queue for the First Come First Served Sales Period of the 2018 FIFA World Cup Russia™.
When it is your turn, you will have 10 minutes to enter the website.

Your estimated wait time is: more than an hour

Status last updated: 2:24:09 PM

Leave the line (You will lose your place)

Queue ID: 4e52c3d3-c986-419f-becf-705f68be6334

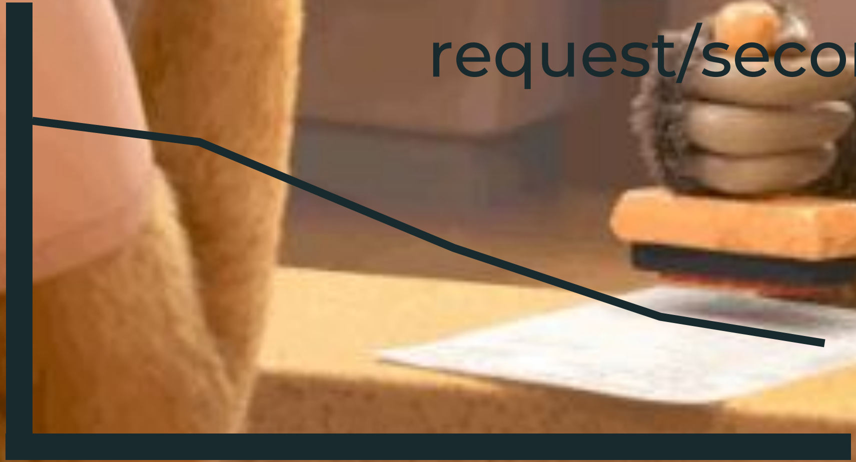
@matvi3nko

LATENCY ⬆

milliseconds

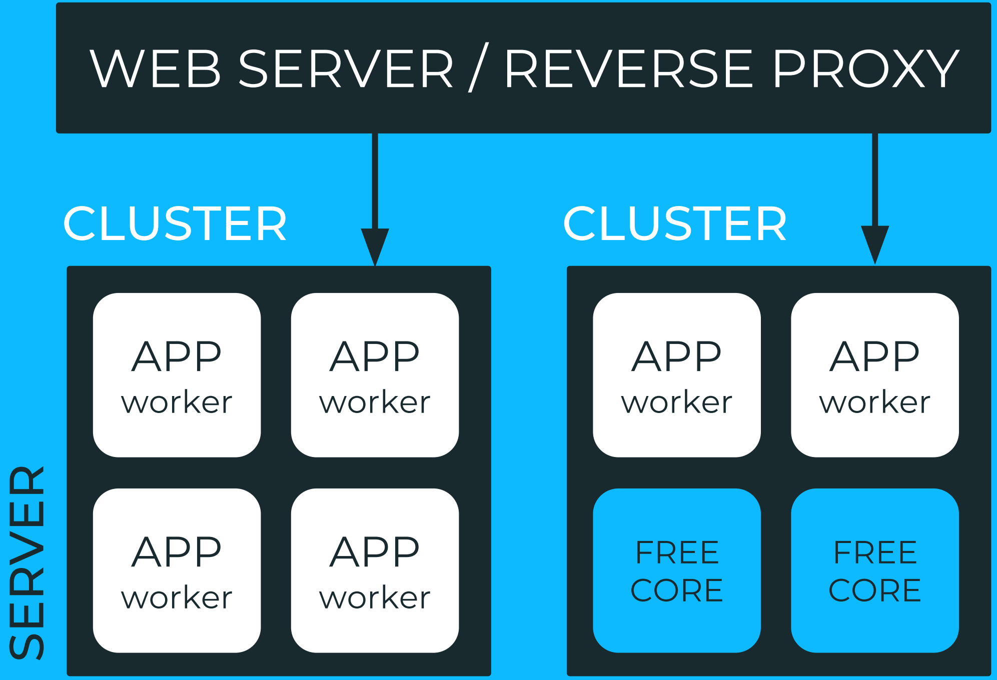
THROUGHPUT ⬇

request/second



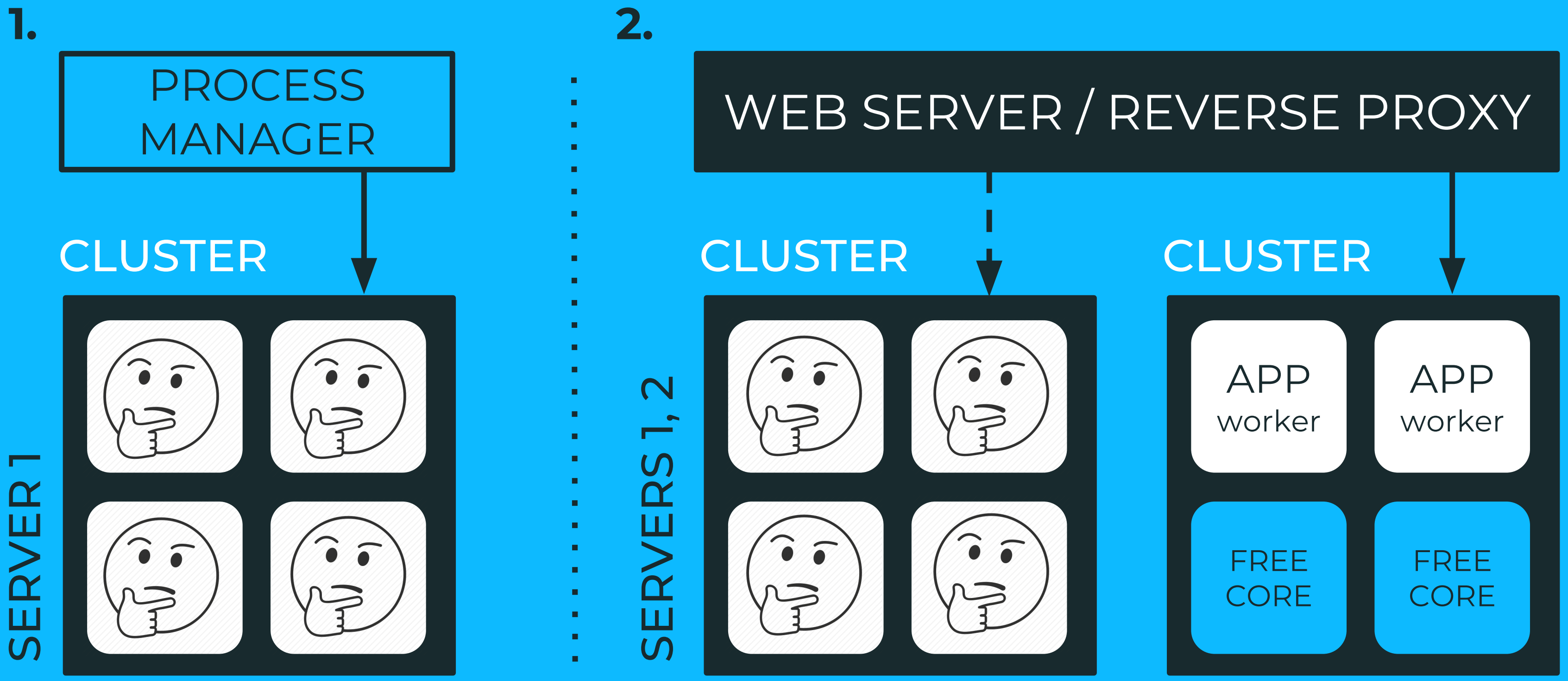
SCALING

1. **MULTIPLE PROCESSES**
   - CLUSTER module
   - PM2
2. **MULTIPLE SERVERS**
   WEB SERVER / REVERSE PROXY
   - PHUSION PASSENGER
   - NGINX

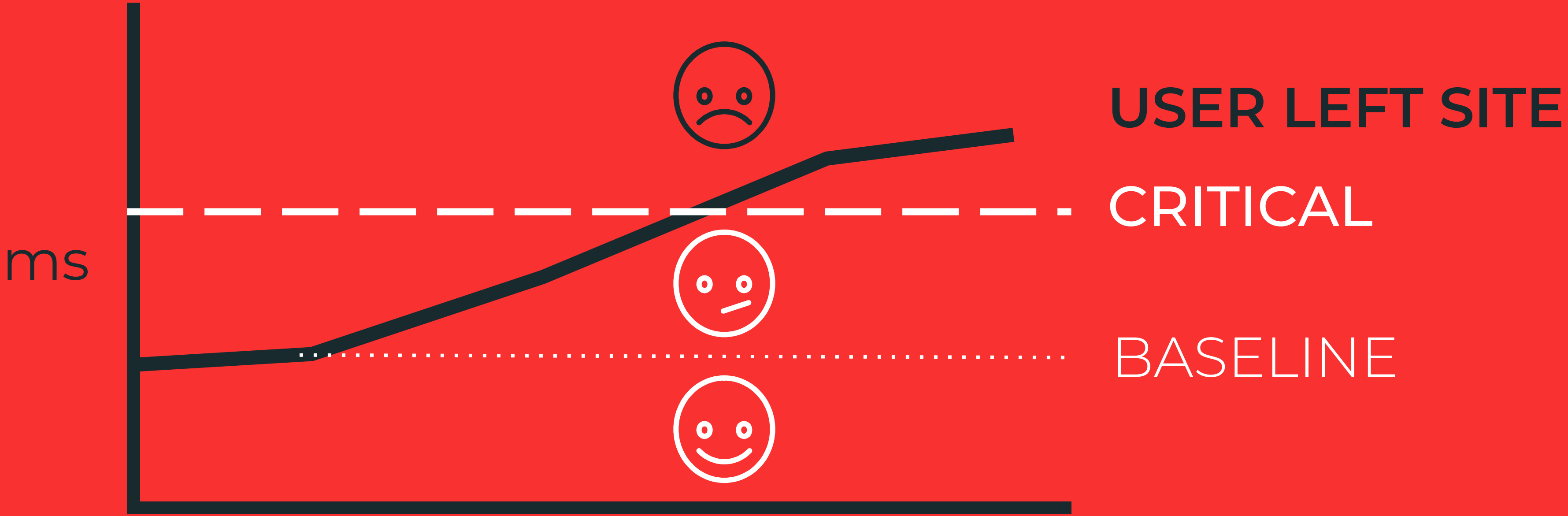


@matvi3nko

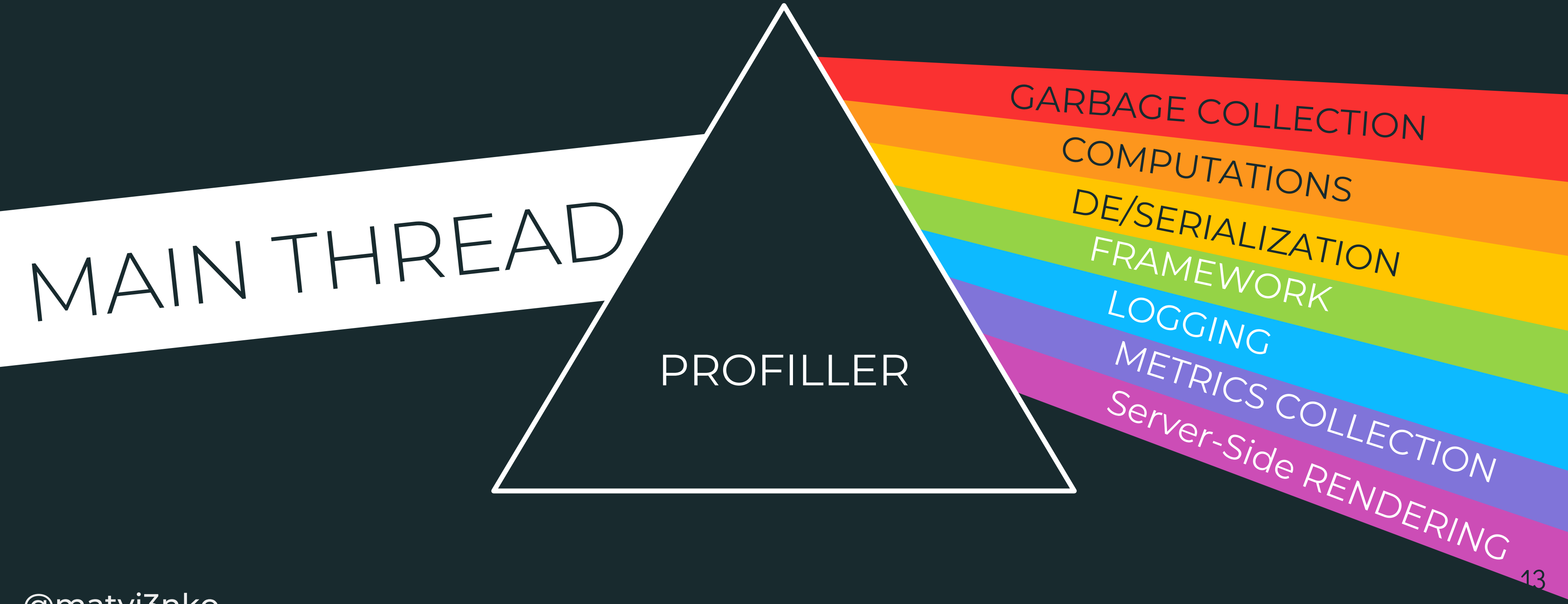
LOAD BALANCING



RESPONSE TIME

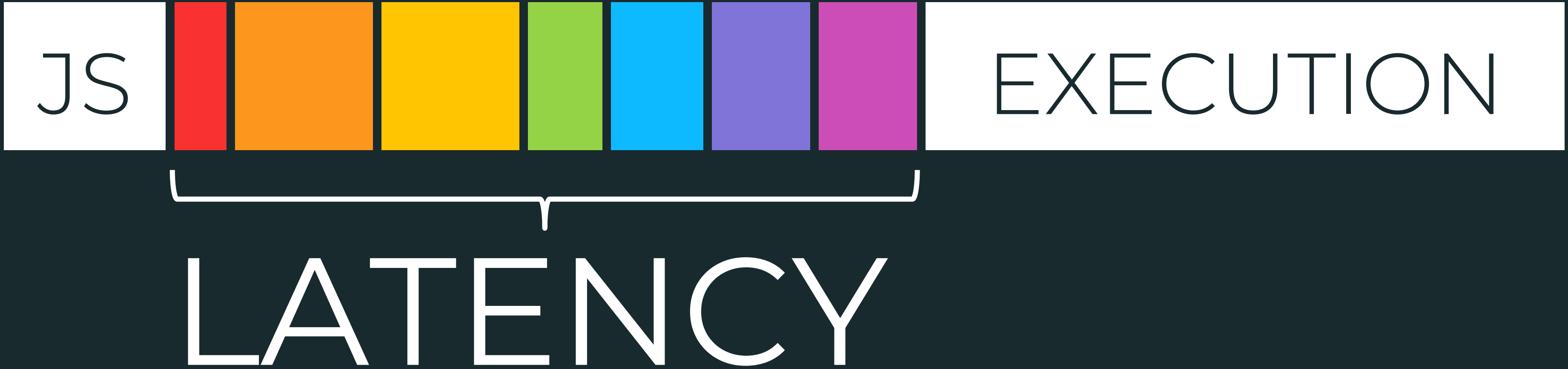


DISPERSION

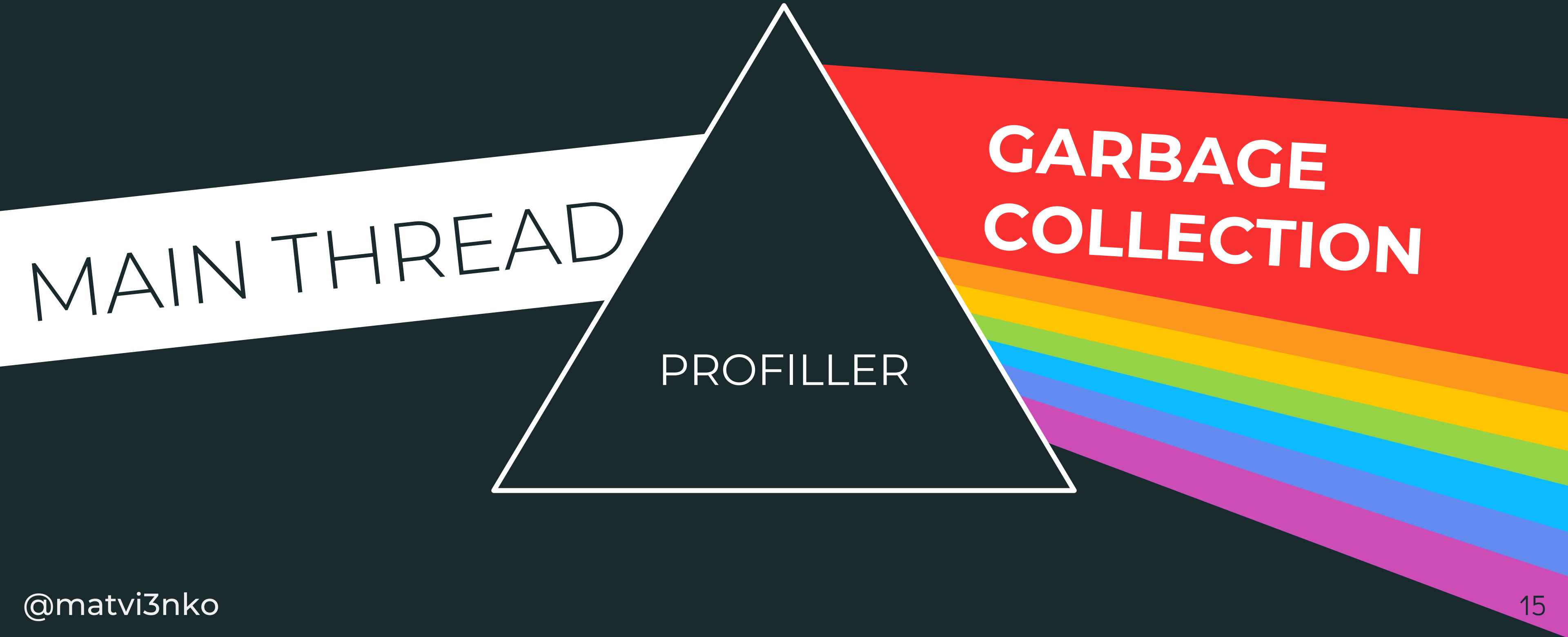

MAIN THREAD
PROFILLER
GARBAGE COLLECTION
COMPUTATIONS
DE/SERIALIZATION
FRAMEWORK
LOGGING
METRICS COLLECTION
Server-Side RENDERING

EACH REQUEST

USERS PAY FOR LOGGING, GC AND METRICS COLLECTIONS ... WITH THEIR TIME.



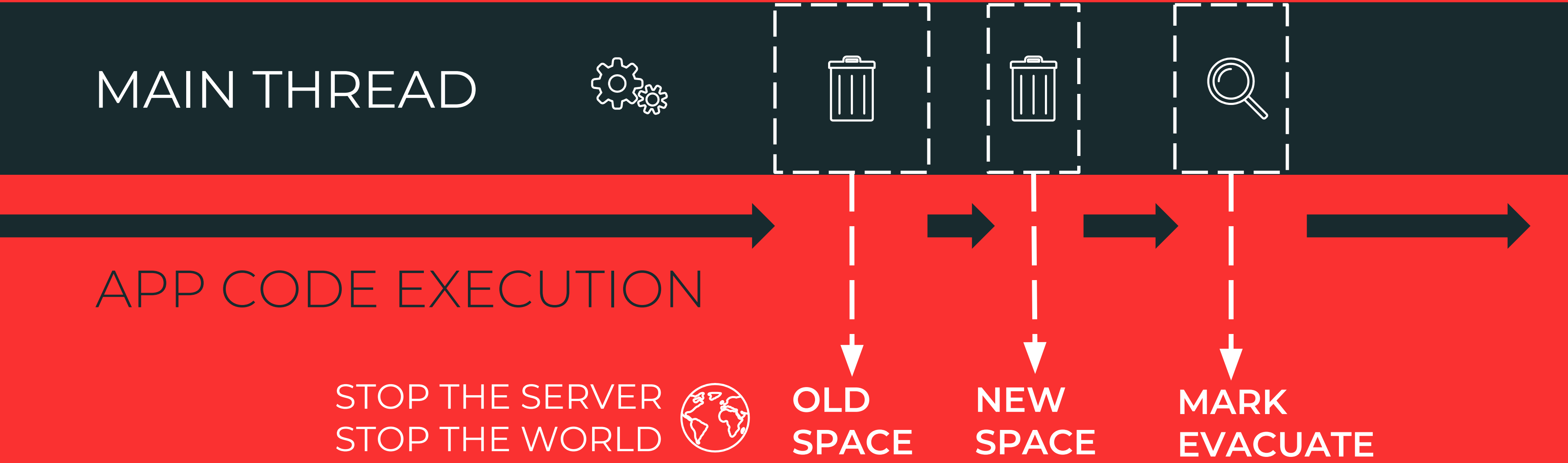
@matvi3nko


MAIN THREAD
PROFILLER
GARBAGE COLLECTION

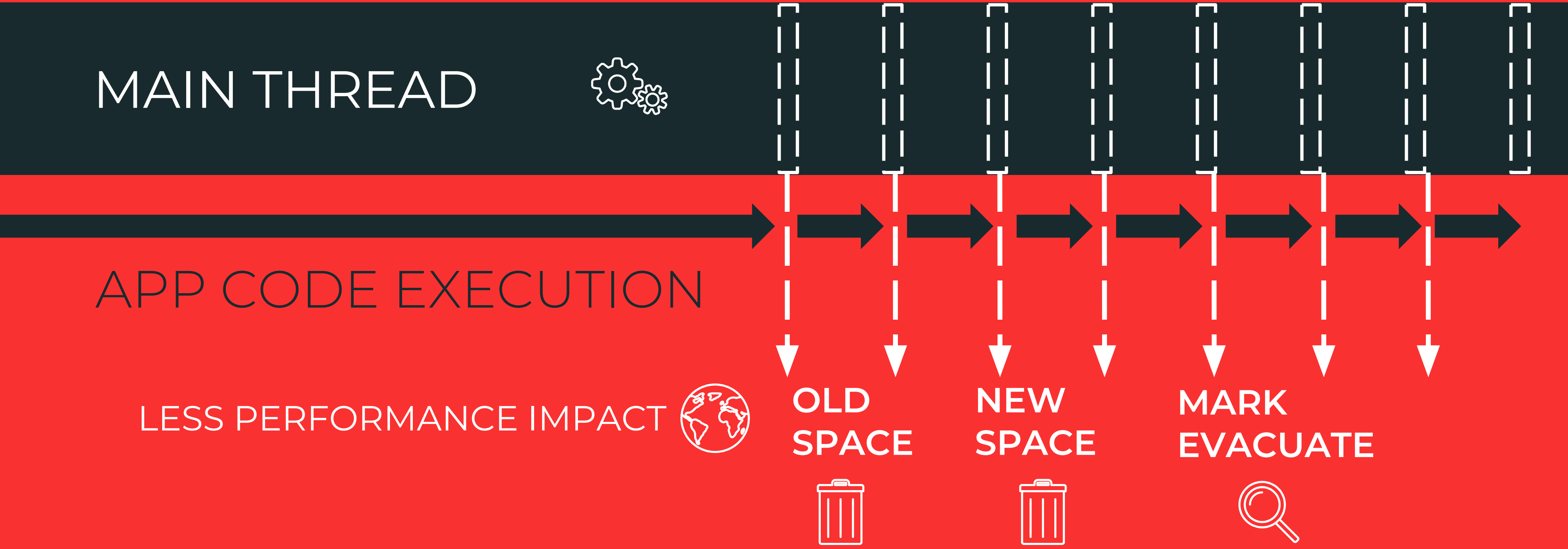
“THE WORLD IS MINE”

© GARBAGE COLLECTOR
1959

GARBAGE COLLECTION

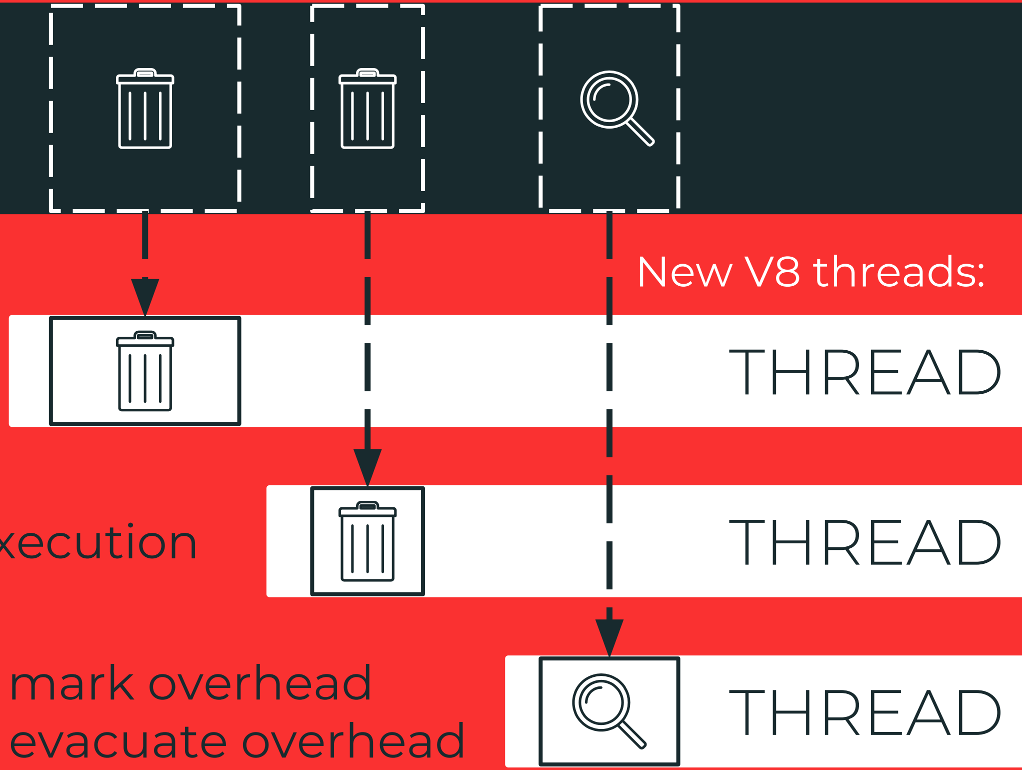


INCREMENTAL COLLECTION



GC DECOMPOSITION

MAIN THREAD
APP CODE EXECUTION



ALGORITHMS of ORINOCO GC:

1. PARALLEL MARK-SWEEP
2. PARALLEL SCAVENGER

X2 execution

3. PARALLEL MARK-EVACUATE

-30% mark overhead
-70% evacuate overhead

@matvi3nko

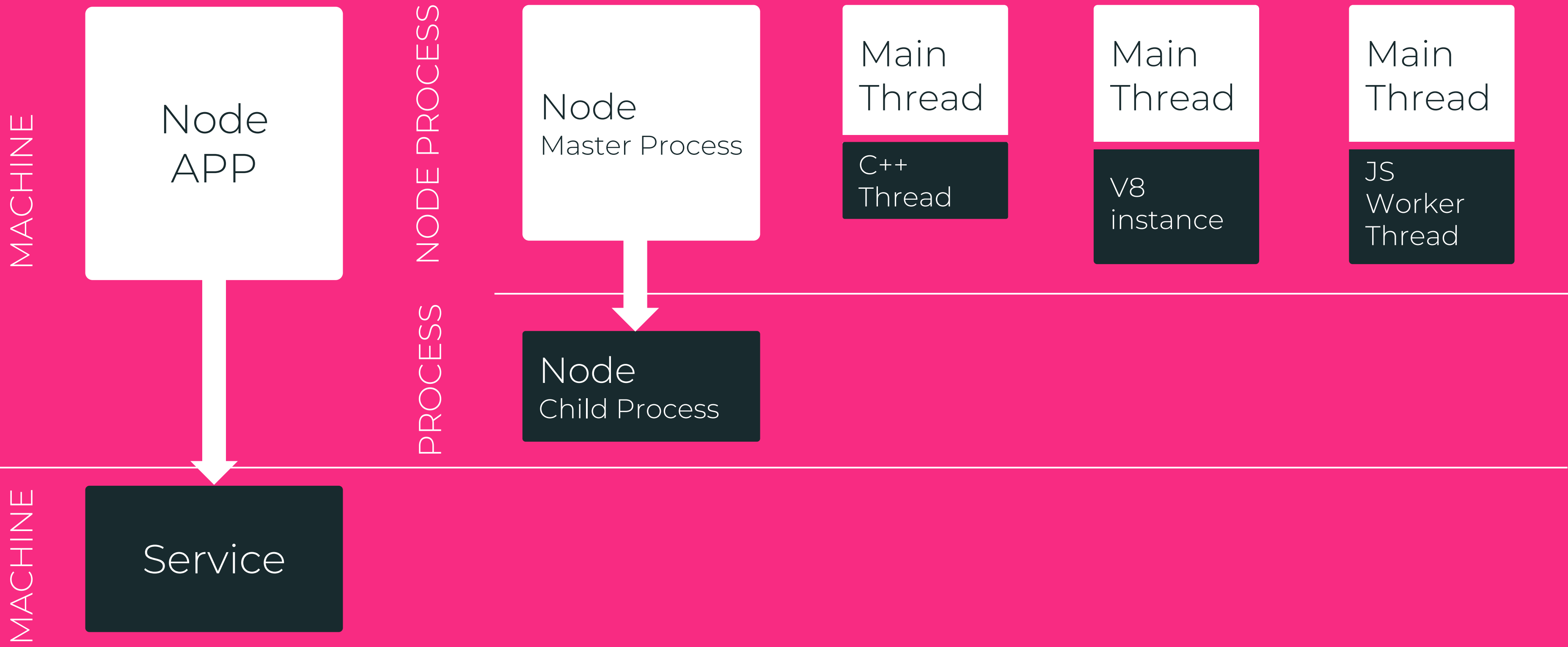
… 50 years later

“THE WORLD 🌍 IS MINE NOW”

© JS COMPUTING OPERATION IN NODE.JS
2009

@matvi3nko

PARALLELIZATION



WEBWORKER THREADS / NAPA.JS / ALIOS

MULTY THREADING

IN NODE.JS

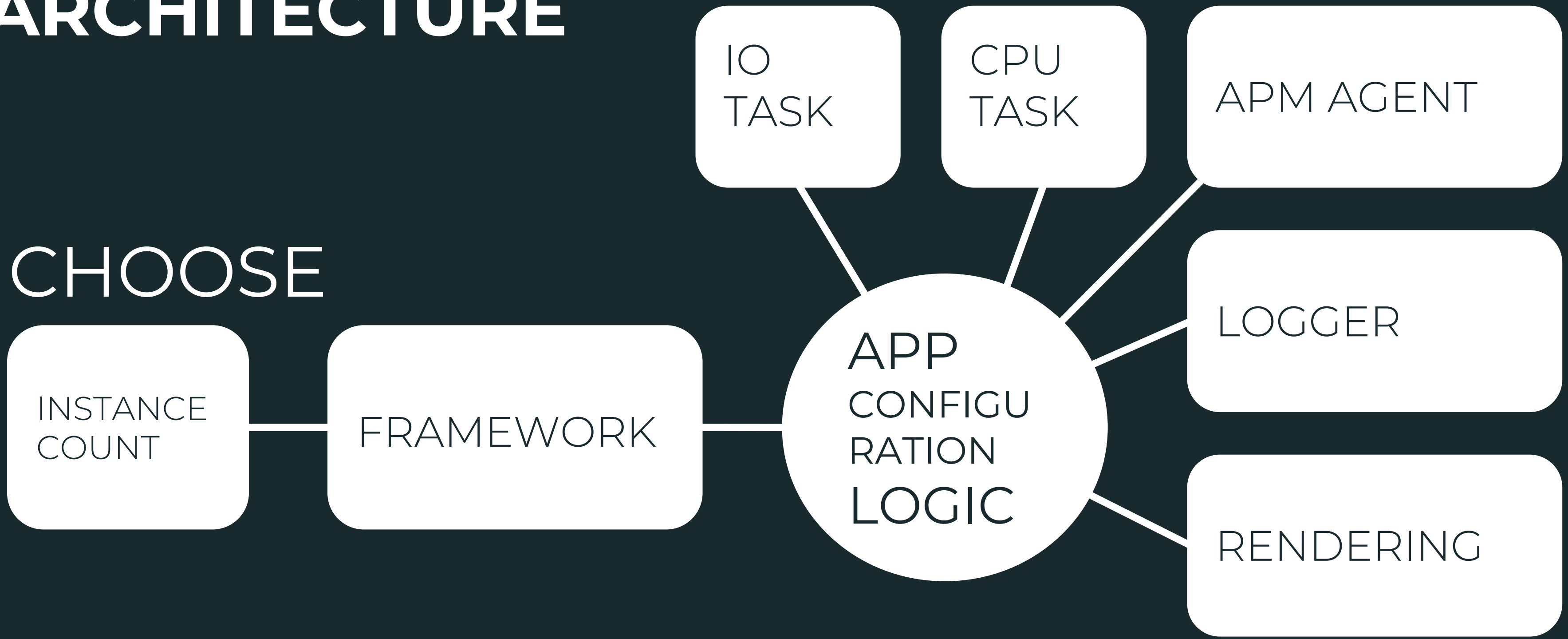
@matvi3nko

WORKER THREADS

- RESOURCES ARE LIMITED
  - IoT, modules
- TASK IS DIFFICULT TO SEPARATE FROM THE APPLICATION CONTEXT
- PARSING, HASHING, HTML MINIFICATION, ...

DEMO APP

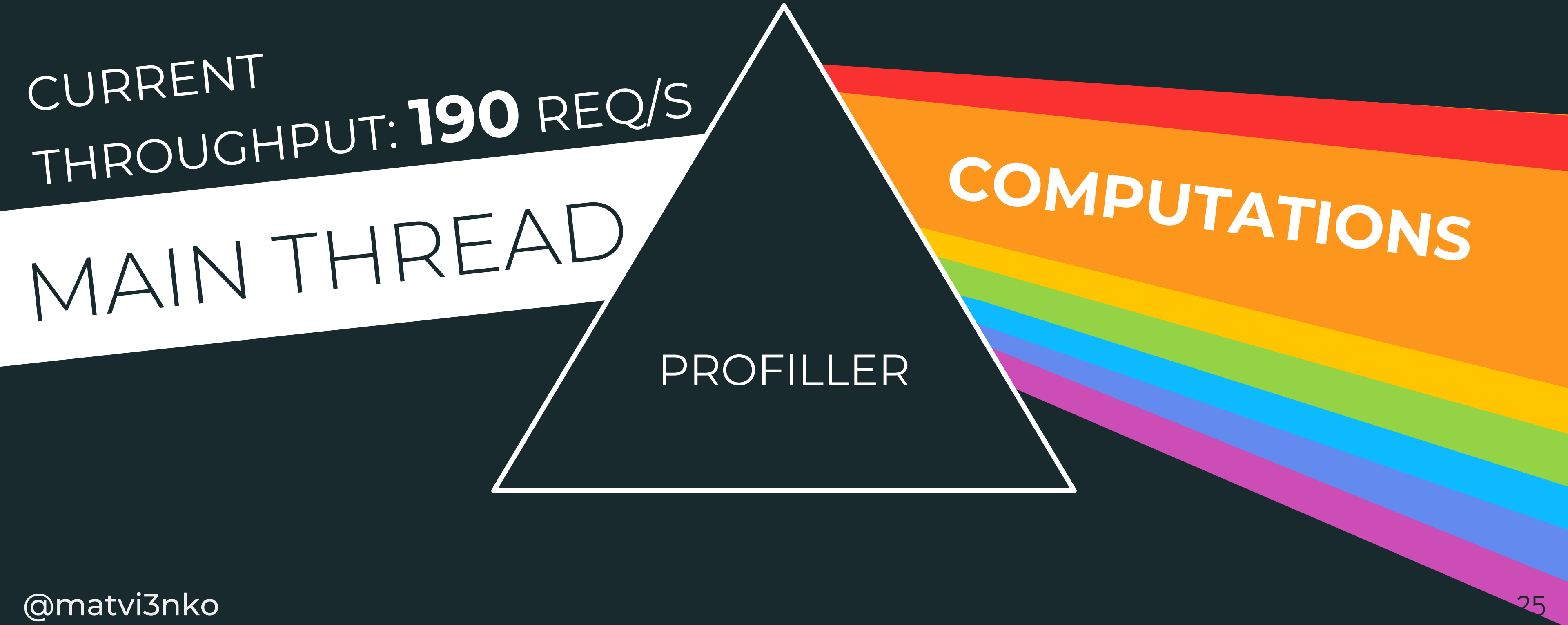
ARCHITECTURE



@matvi3nko

DECOMPOSING

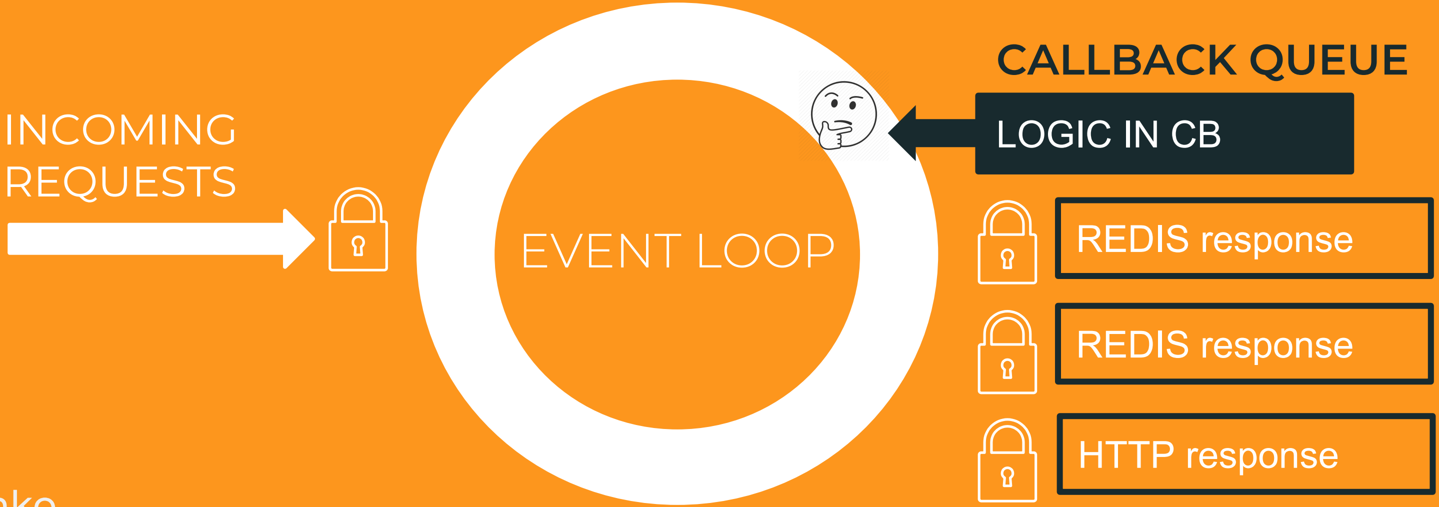
CURRENT THROUGHPUT: **190** REQ/S



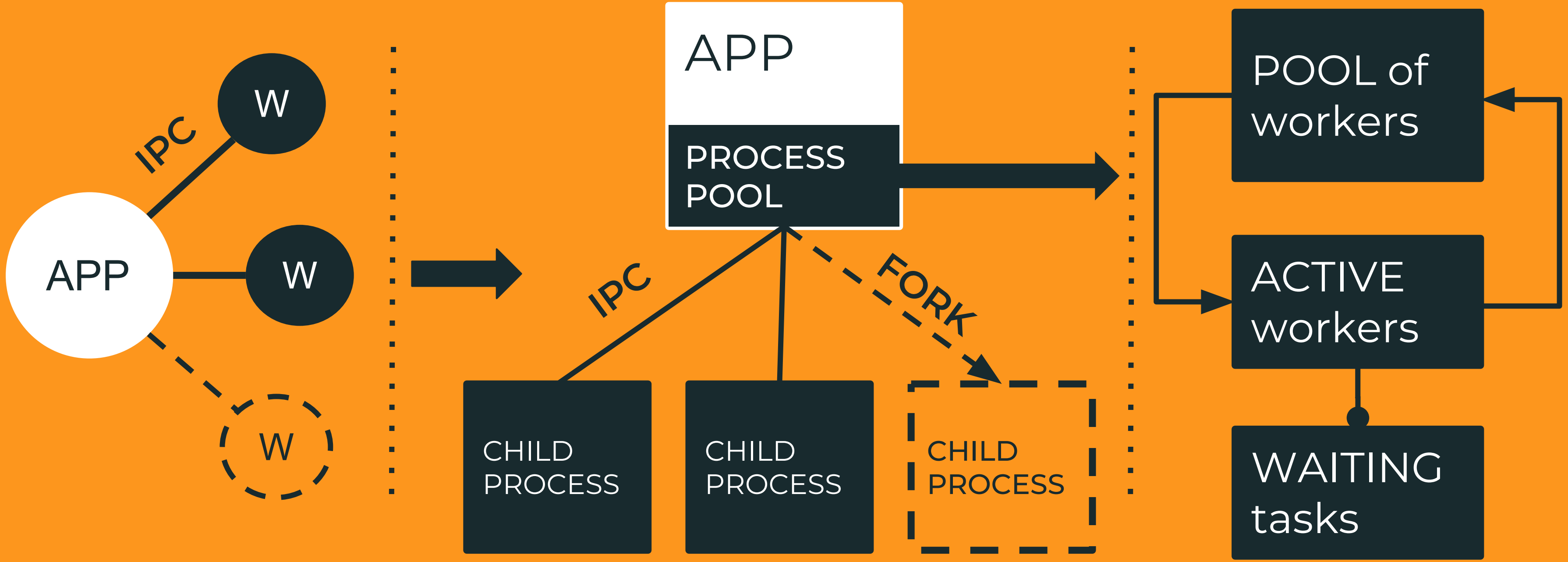
CPU-BOUND TASKS

MAIN THREAD
APP CODE EXECUTION

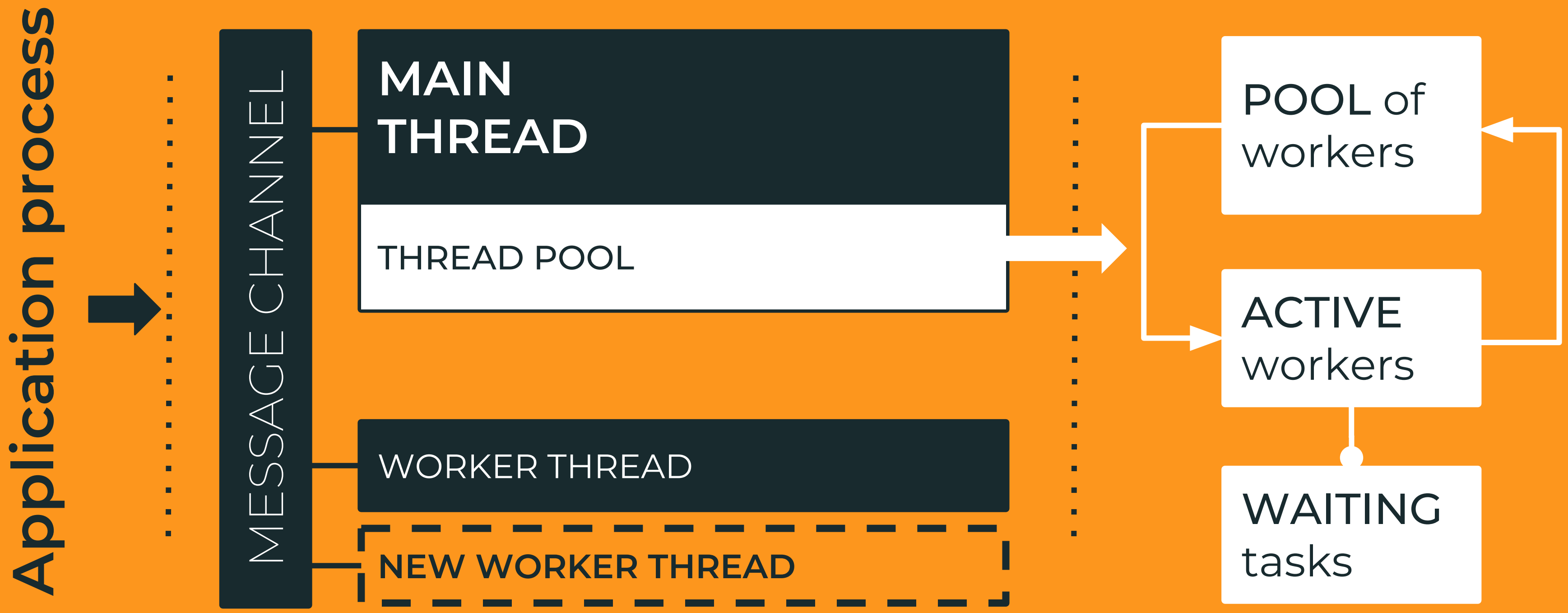
CPU-bound processing



PROCESS POOL



THREAD POOL

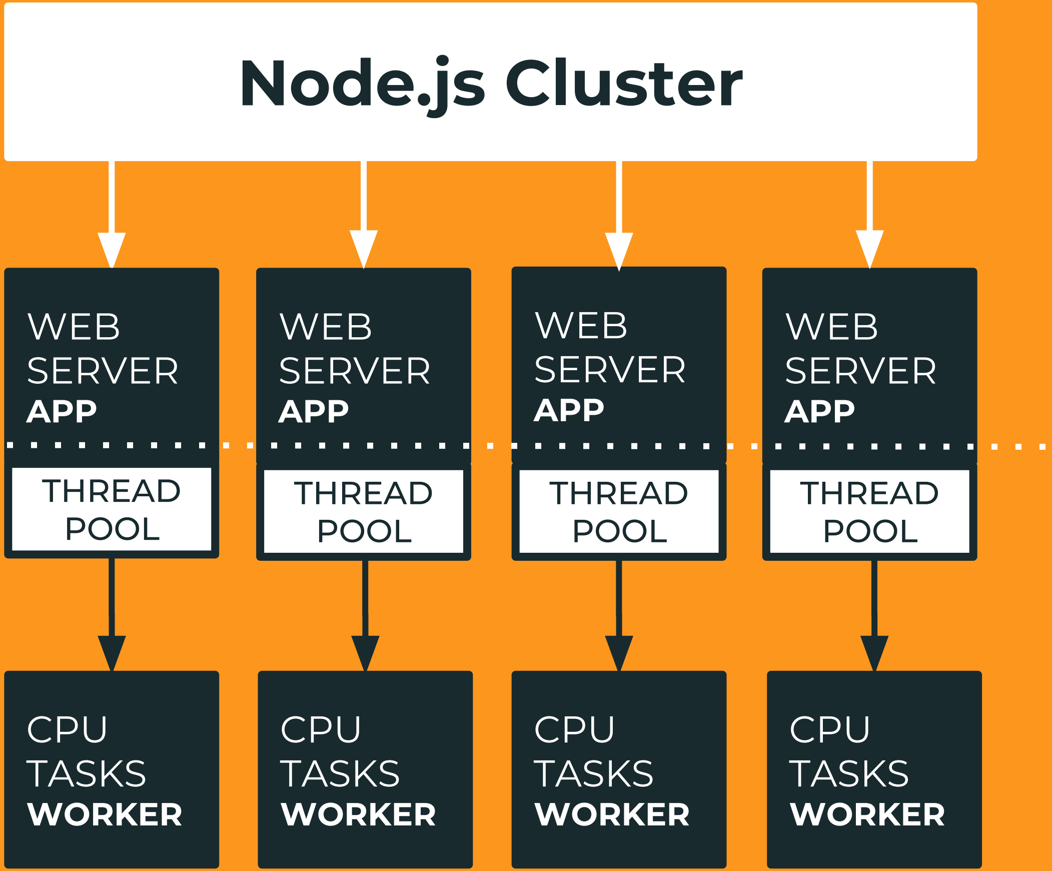


@matvi3nko

CLUSTER

IN-MAIN THREAD COMPUTING

PARALLEL COMPUTING



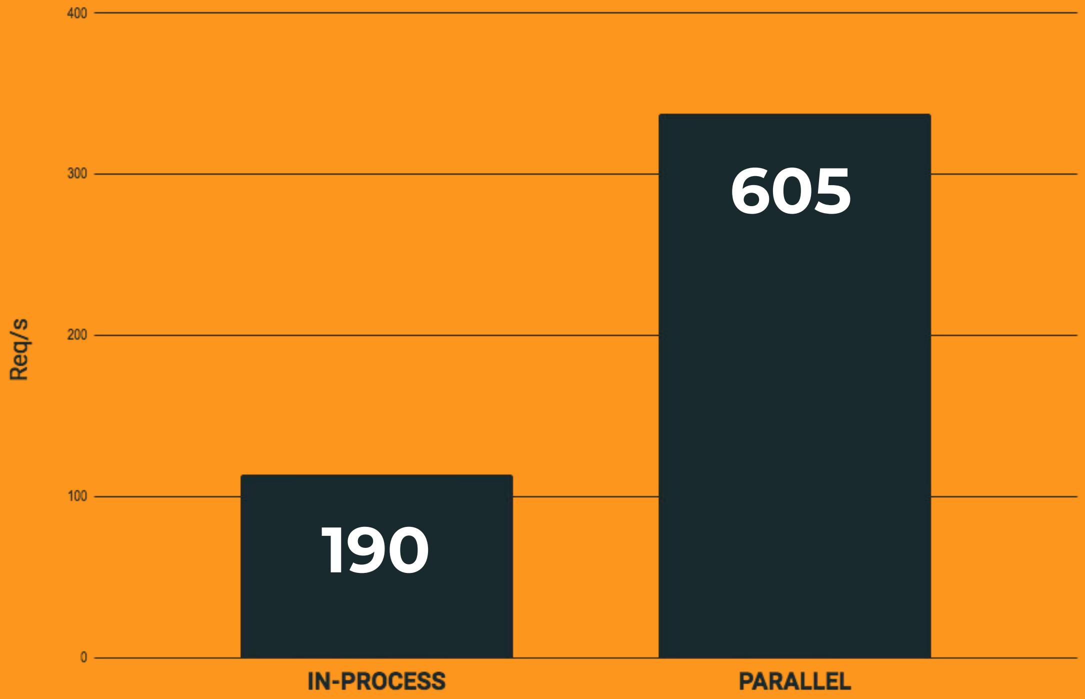
@matvi3nko

CPU TASKS PARALLELIZATION

RESULT

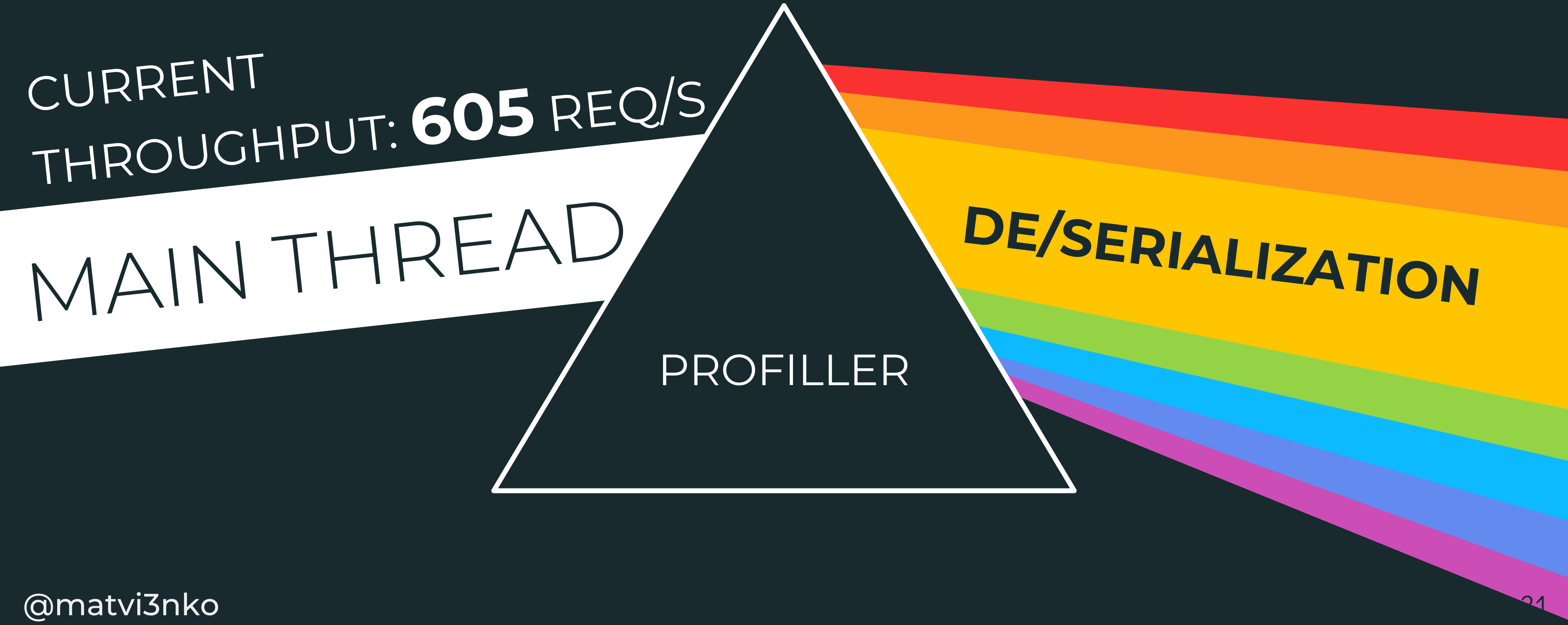
3X

MORE REQ/S
WITH
PARALLEL JS
COMPUTATION

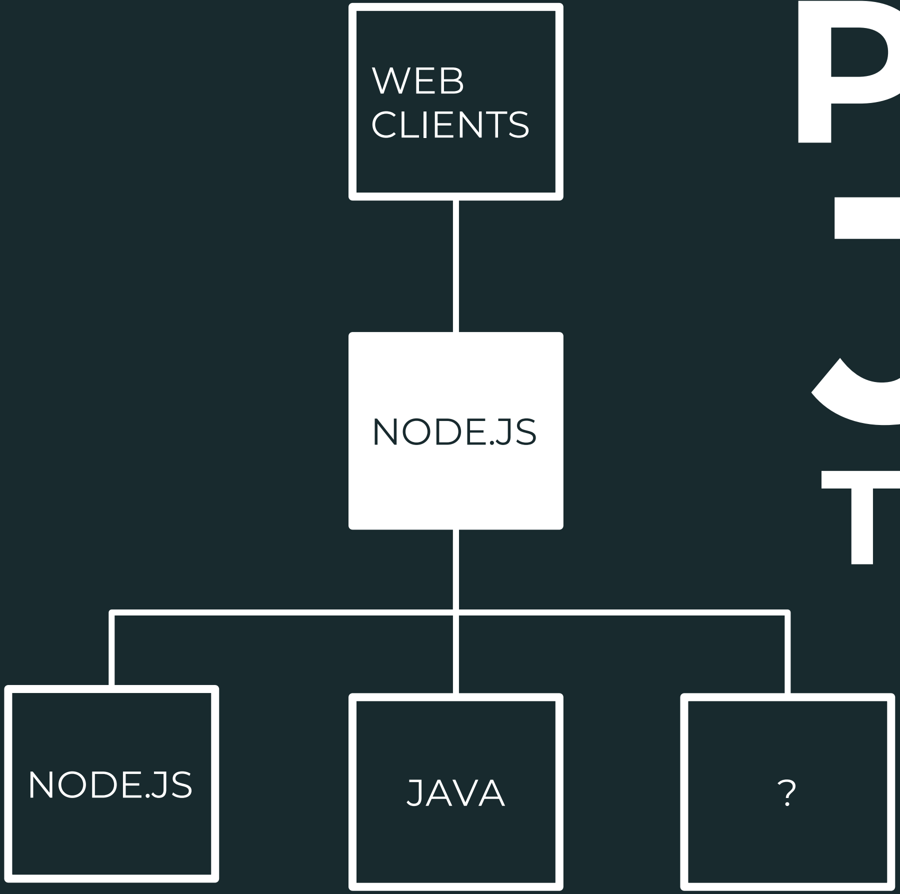


@matvi3nko

DE/SERIALIZATION



JSON STRINGIFY & PARSE



Protobuf

JSON

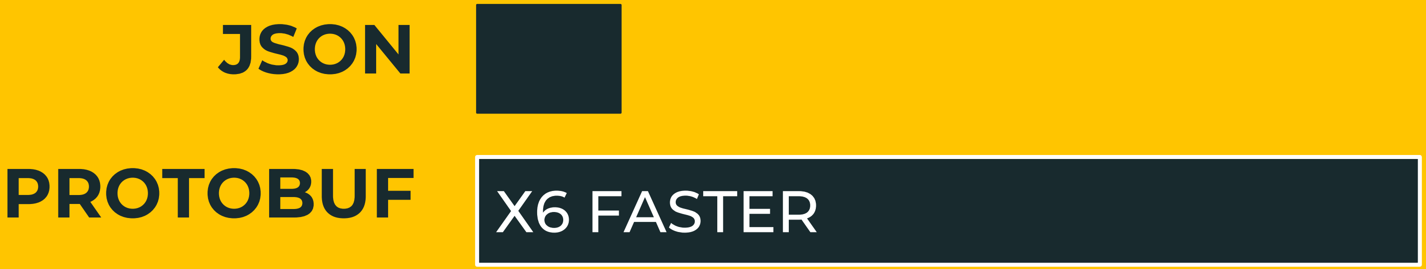
Thrift Avro

… and others

@matvi3nko

IN NOT NATIVE ENV

SERIALIZATION & PARSE



*Average values in not JavaScript environment.

JSON IN NATIVE ENV

+ OPTIMIZED BY V8 TEAM
- DEPENDS ON THE TYPE OF DATA

NUMBERS

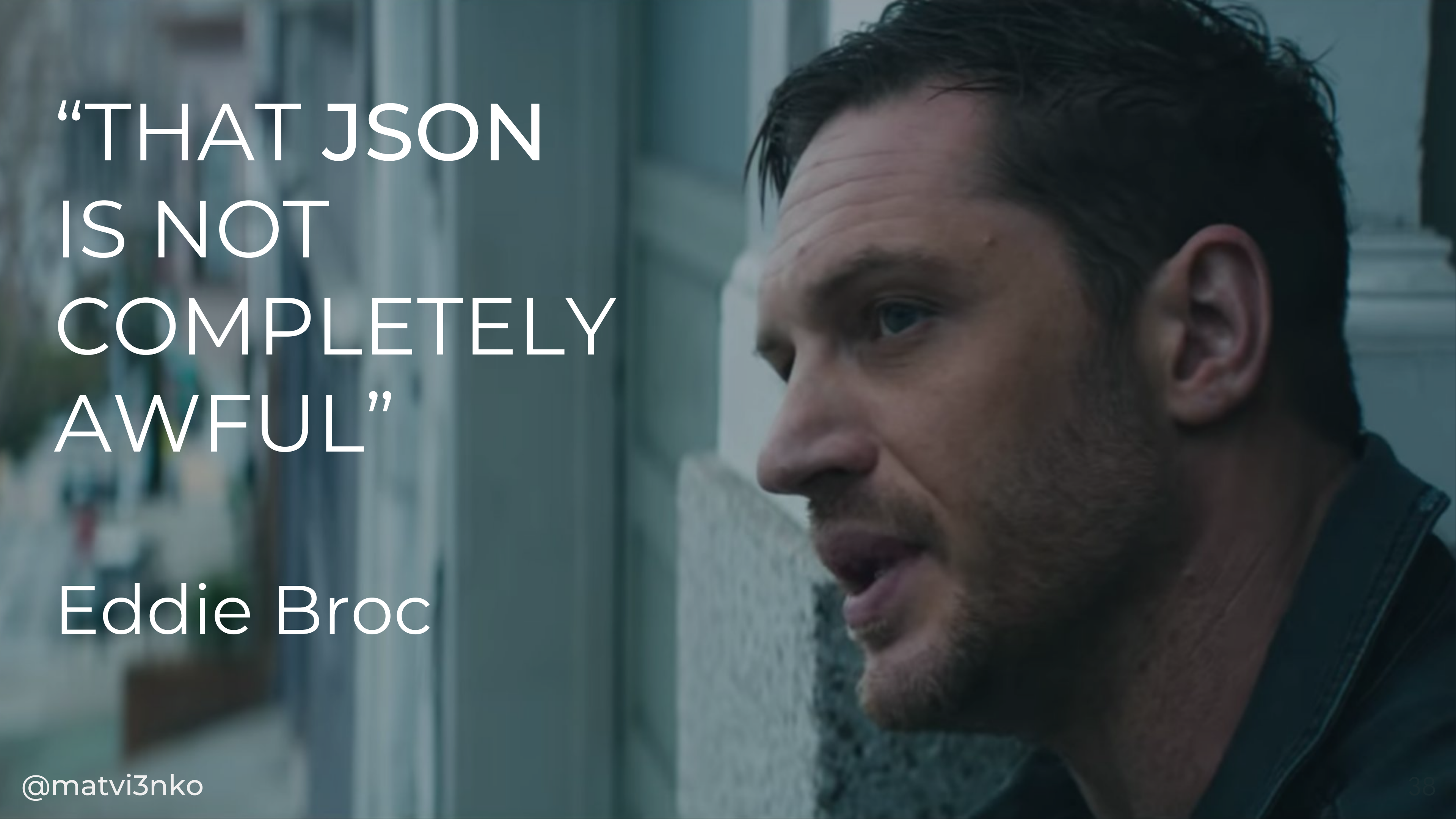
IN NODE.JS	SERIALIZATION	PARSE
JSON	■	■
PROTOBUF	■	X2 FASTER
JSON WITH SCHEMA	X2 FASTER	+30%

*Average values in JavaScript environment.
See libraries in resources.

STRINGS

IN NODE.JS	SERIALIZATION	PARSE
JSON	X2	
PROTOBUF		X2 FASTER
JSON WITH SCHEMA	X2.3	X3 FASTER

*Average values in JavaScript environment.
See libraries in resources.



“THAT JSON IS NOT COMPLETELY AWFUL”

Eddie Broc

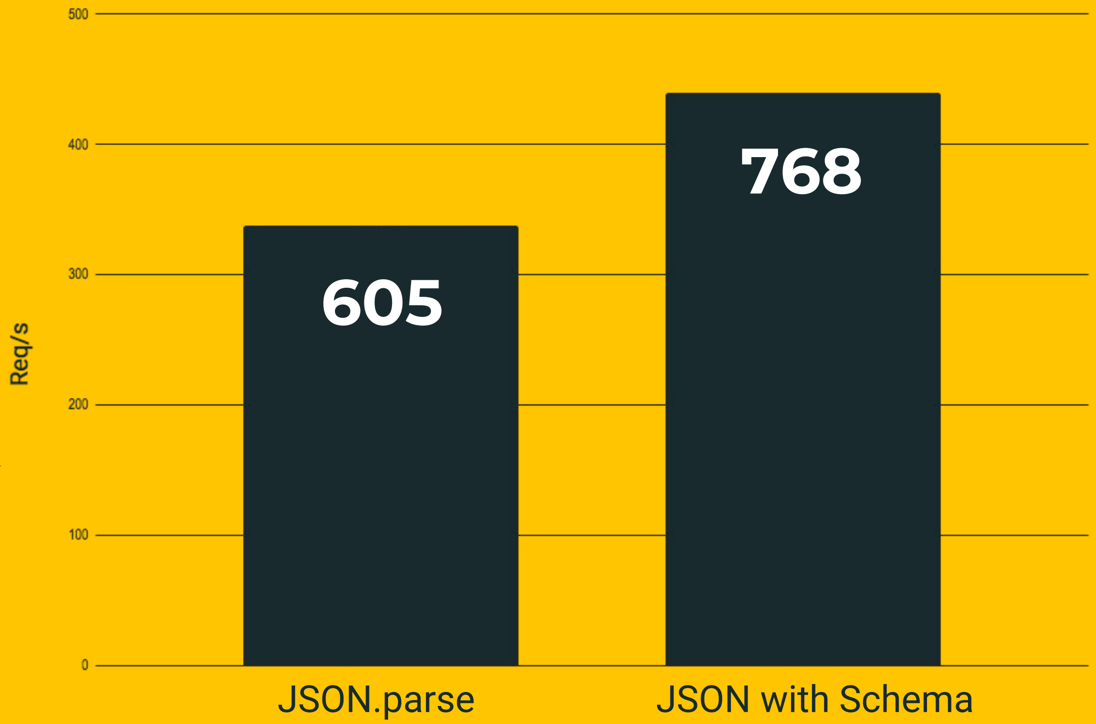
@matvi3nko

JSON.PARSE CHANGE RESULT

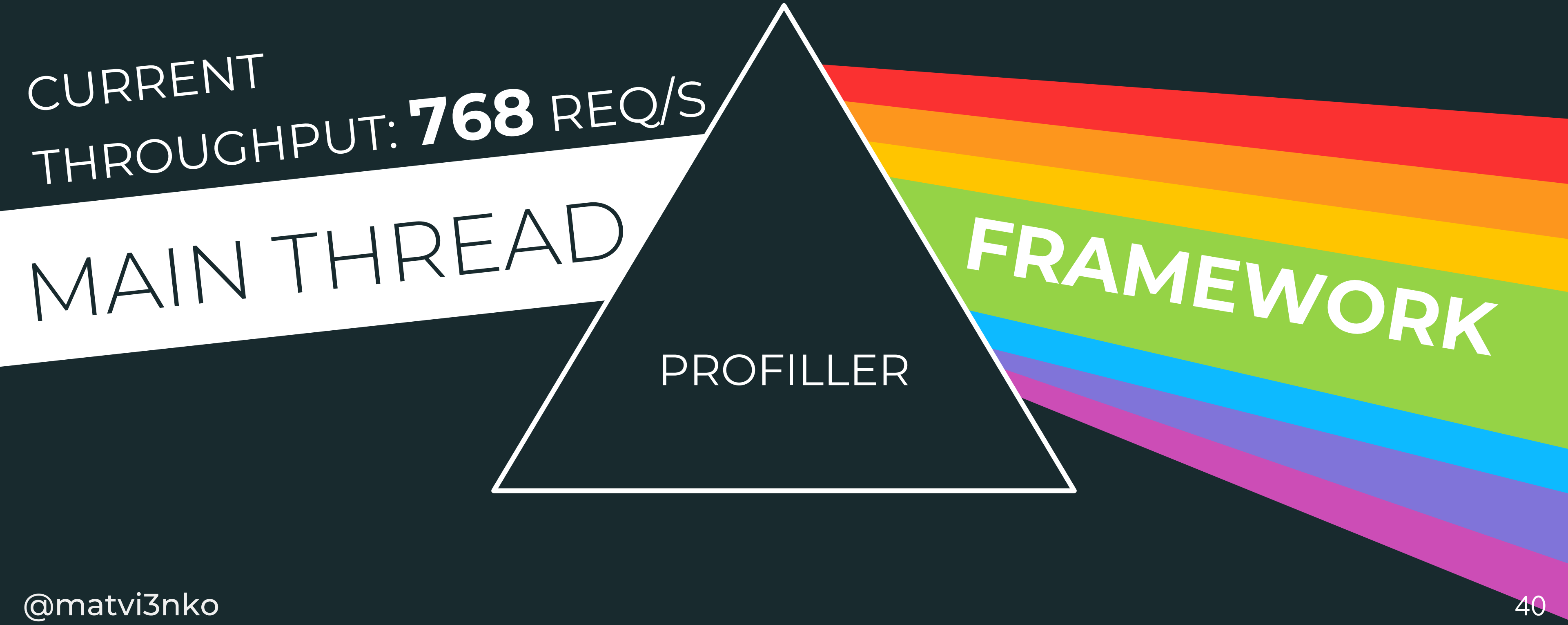
+27%

MORE REQ/S

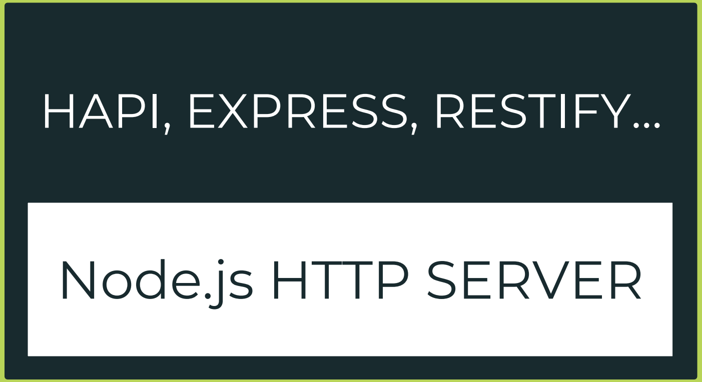
WITH
FAST-JSON-STRINGIFY
&
JITSON/
TURBO-JSON-PARSE



FRAMEWORK


CURRENT
THROUGHPUT: **768** REQ/S
MAIN THREAD
PROFILLER
FRAMEWORK

FRAMEWORK



OVERHEAD

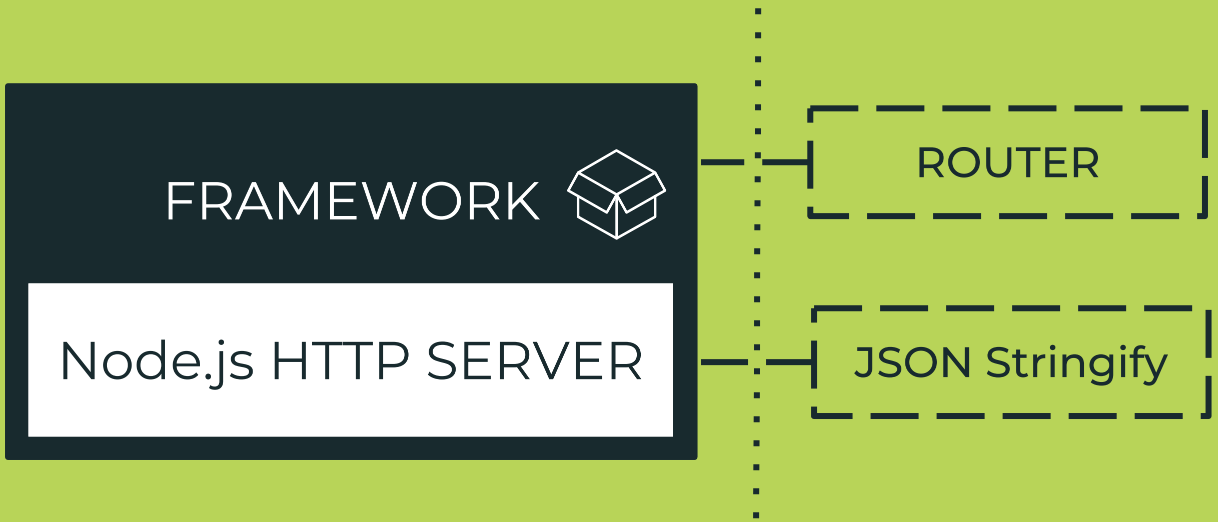
~ 1.5 – 2X SLOWLY than http.createServer

https://github.com/fastify/fast-json-stringify

OPTIMIZATIONS

1. Express
https://github.com/expressjs/express

2. Fastify
https://github.com/fastify/fastify



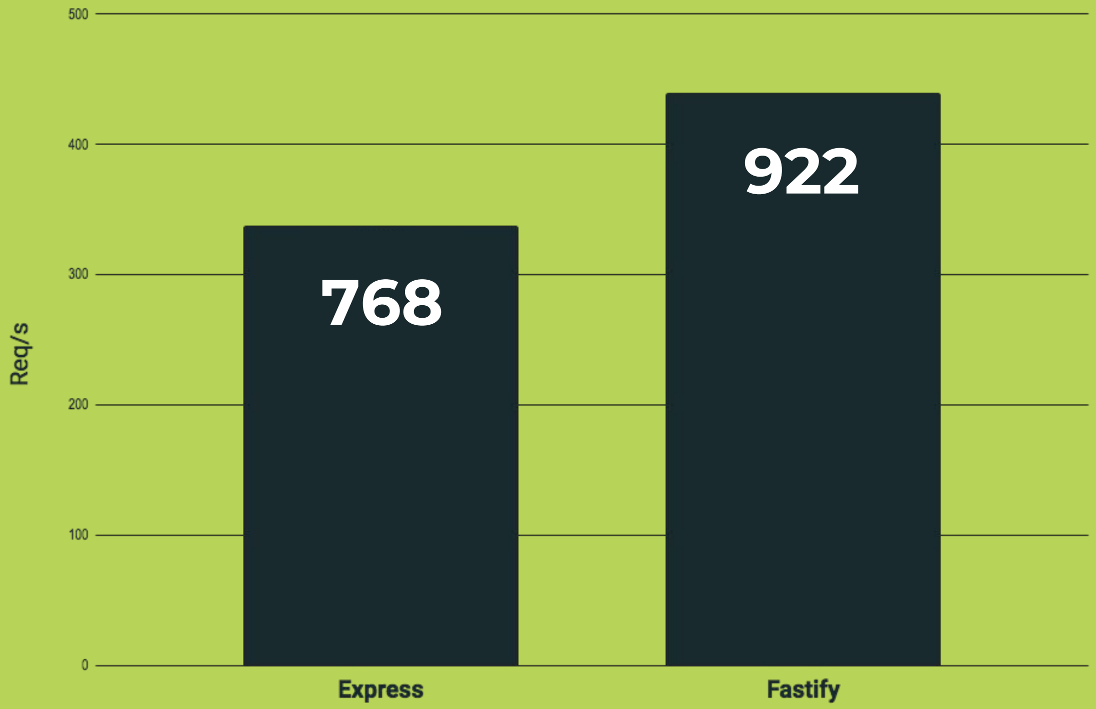
X5 faster
https://github.com/delvedor/router-benchmark

X2-3 faster
https://github.com/fastify/fast-json-stringify

FRAMEWORK CHANGE RESULT

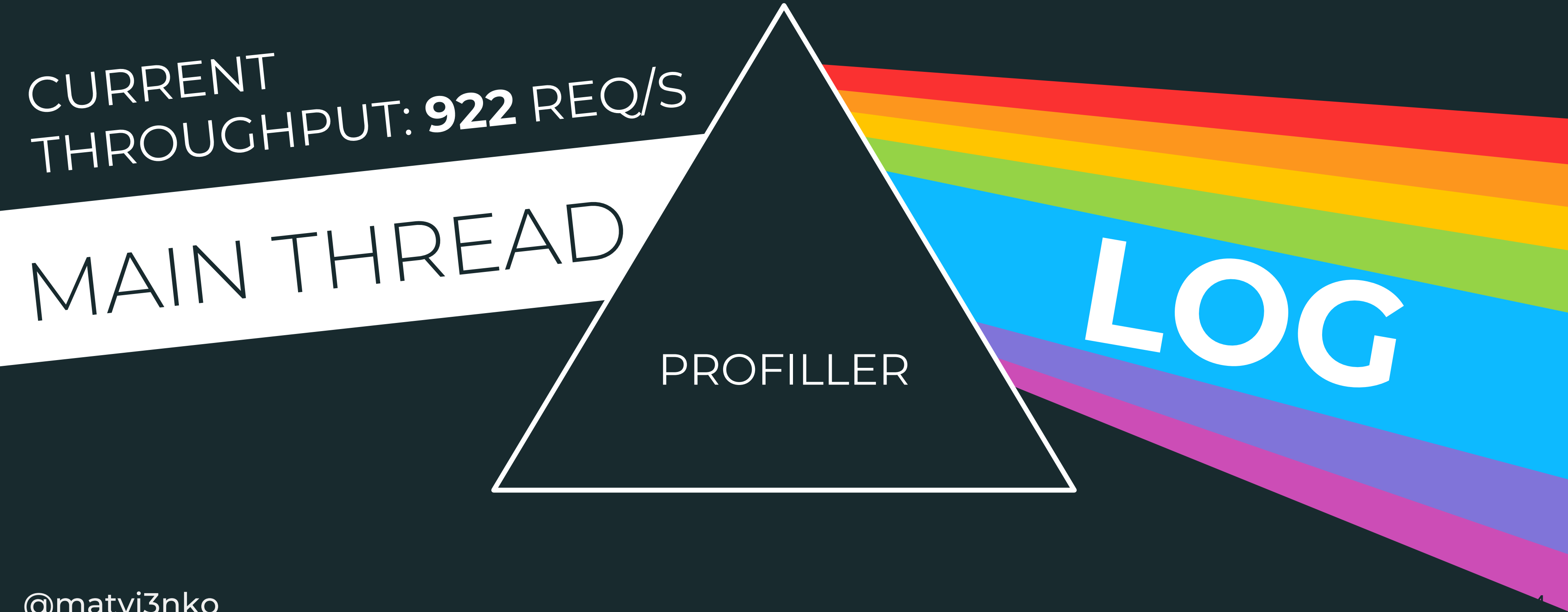
+20%

MORE REQ/S
WITH FASTIFY



LOGGING

CURRENT THROUGHPUT: **922** REQ/S



LOGGING

MAIN THREAD
APP CODE EXECUTION

WRITE LOG

WRITE LOG

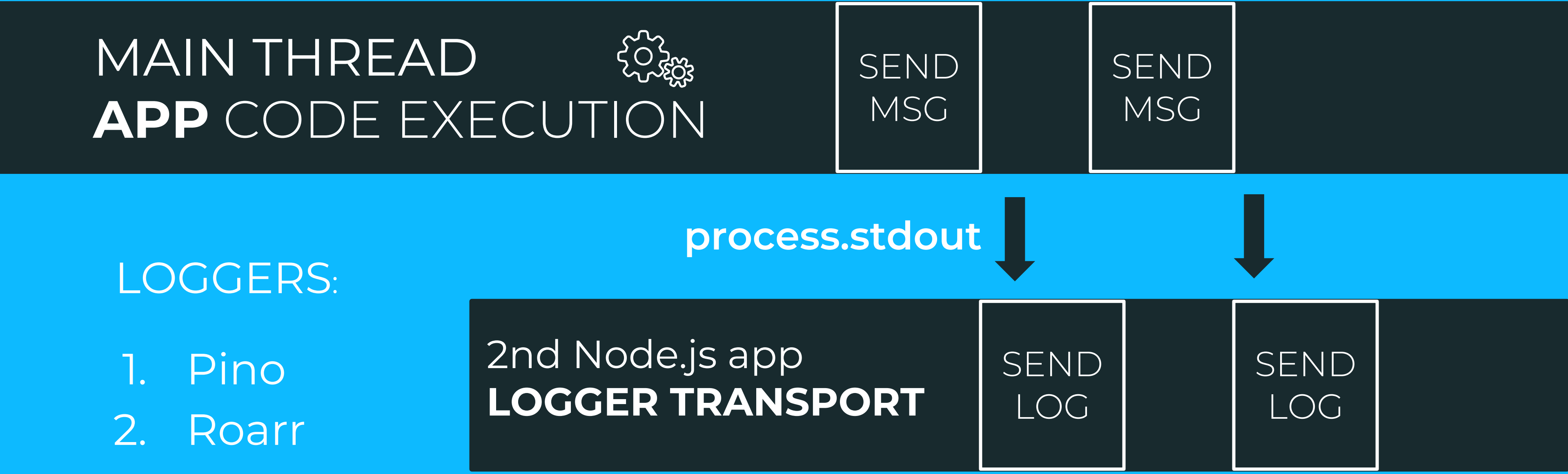
- FORMAT MESSAGE
- SERIALIZE MESSAGE
- HANDLE TRANSPORT LOGIC

process.stdout and .stderr

FILES: sync on Windows and POSIX
TERMINALS::
async on Windows, sync on POSIX
PIPES, SOCKETS:
sync on Windows, async on POSIX

@matvi3nko

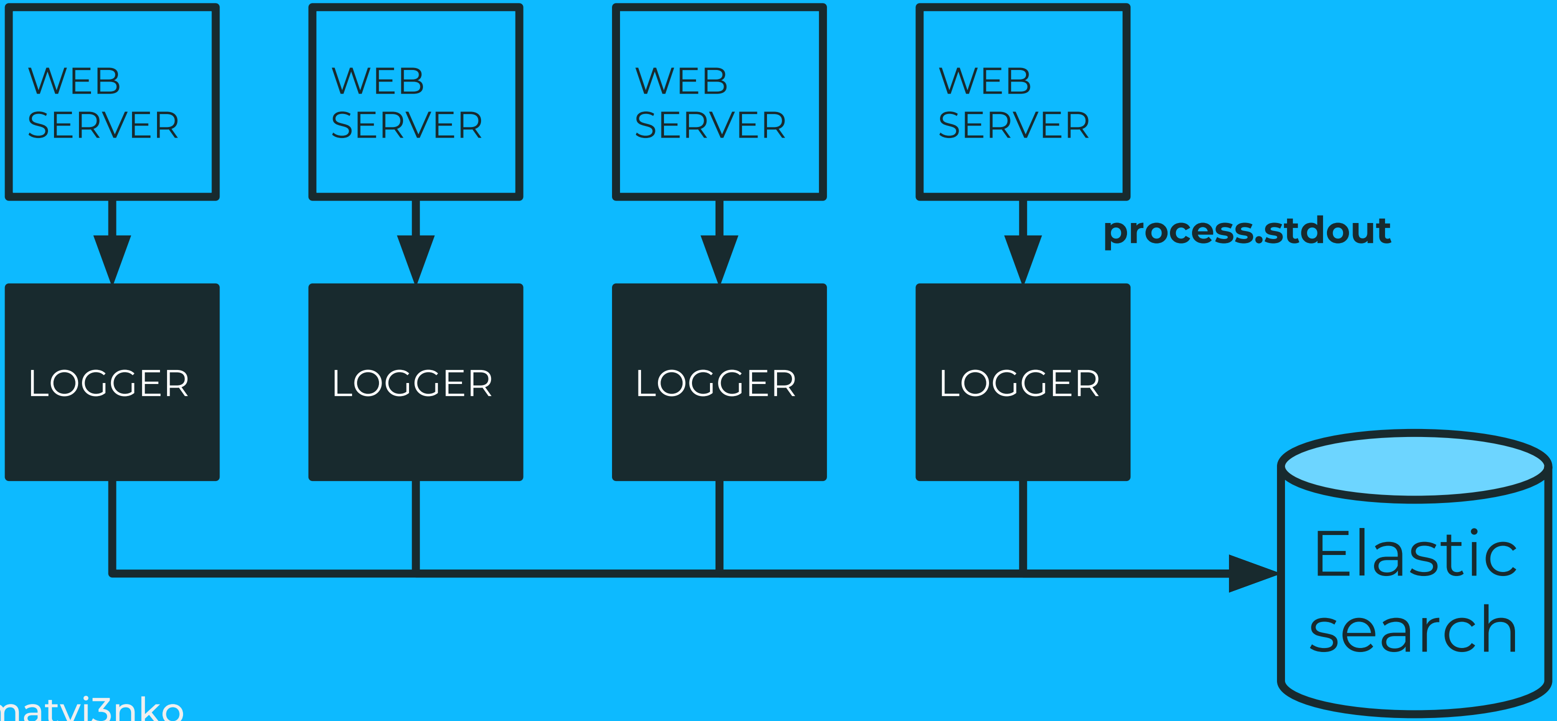
OFF-PROCESS LOGGER TRANSPORT



LOGGERS:

1. Pino
2. Roarr

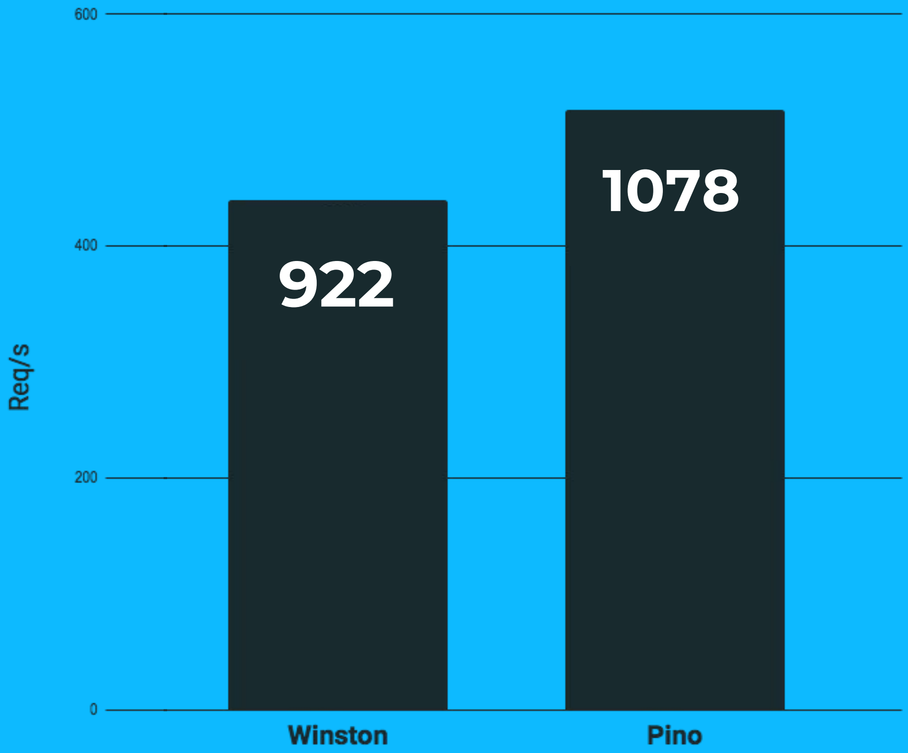
CLUSTER


WEB SERVER
WEB SERVER
WEB SERVER
WEB SERVER
process.stdout
LOGGER
LOGGER
LOGGER
LOGGER
Elastic search

OFF-PROCESS LOGGING RESULT

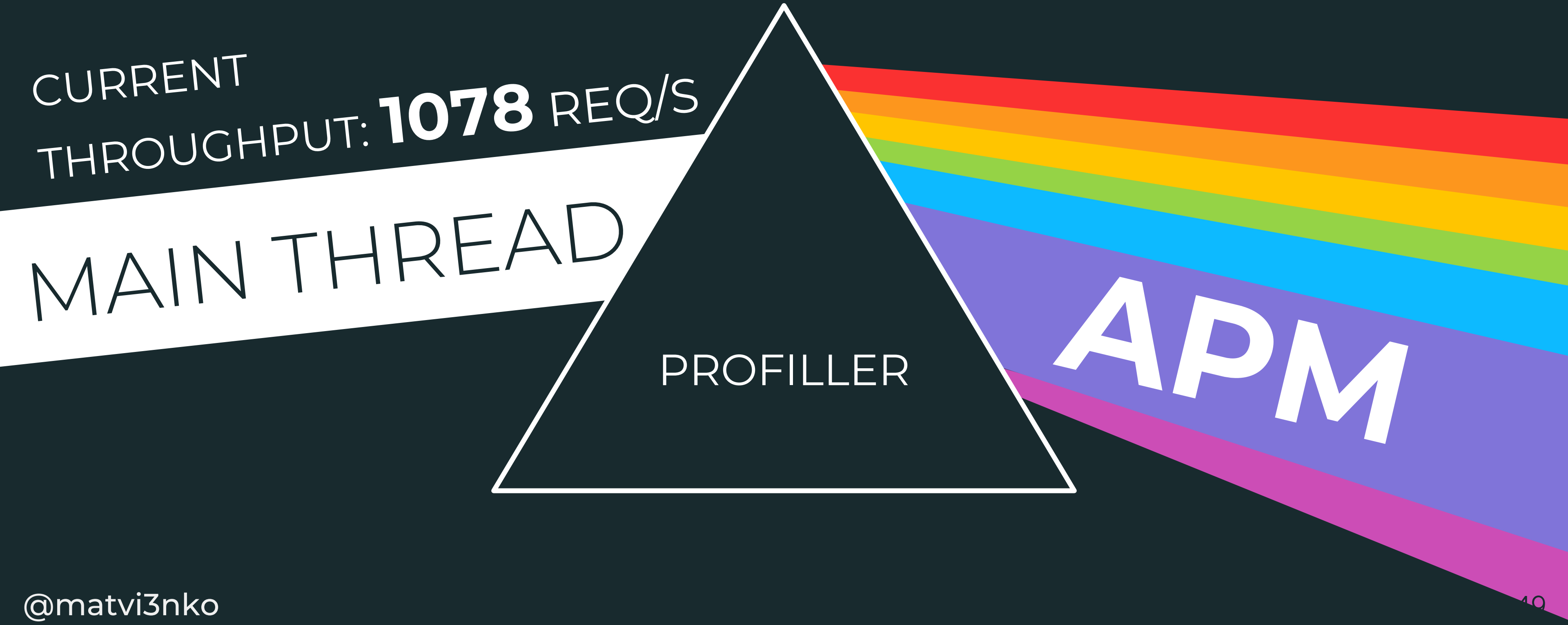
+17%

MORE REQ/S
WITH
OFF-PROCESS
LOGGER
TRANSPORT

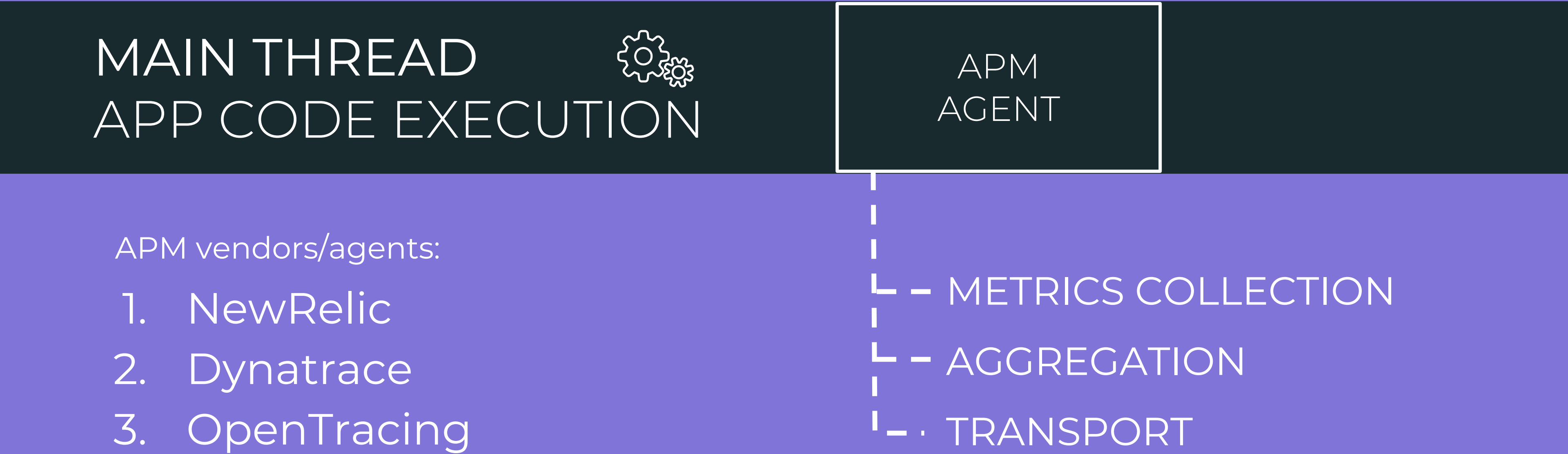


@matvi3nko

APPLICATION PERFORMANCE MONITORING



APPLICATION PERFORMANCE MONITORING

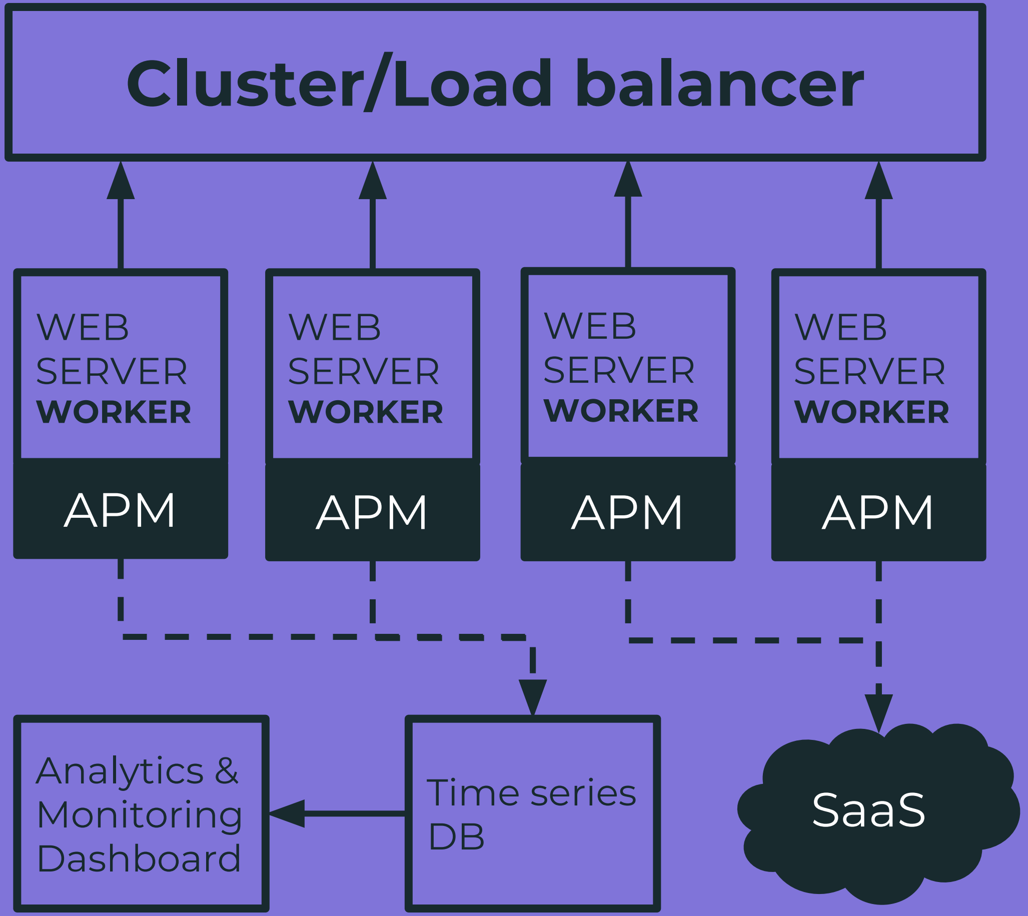


APM vendors/agents:

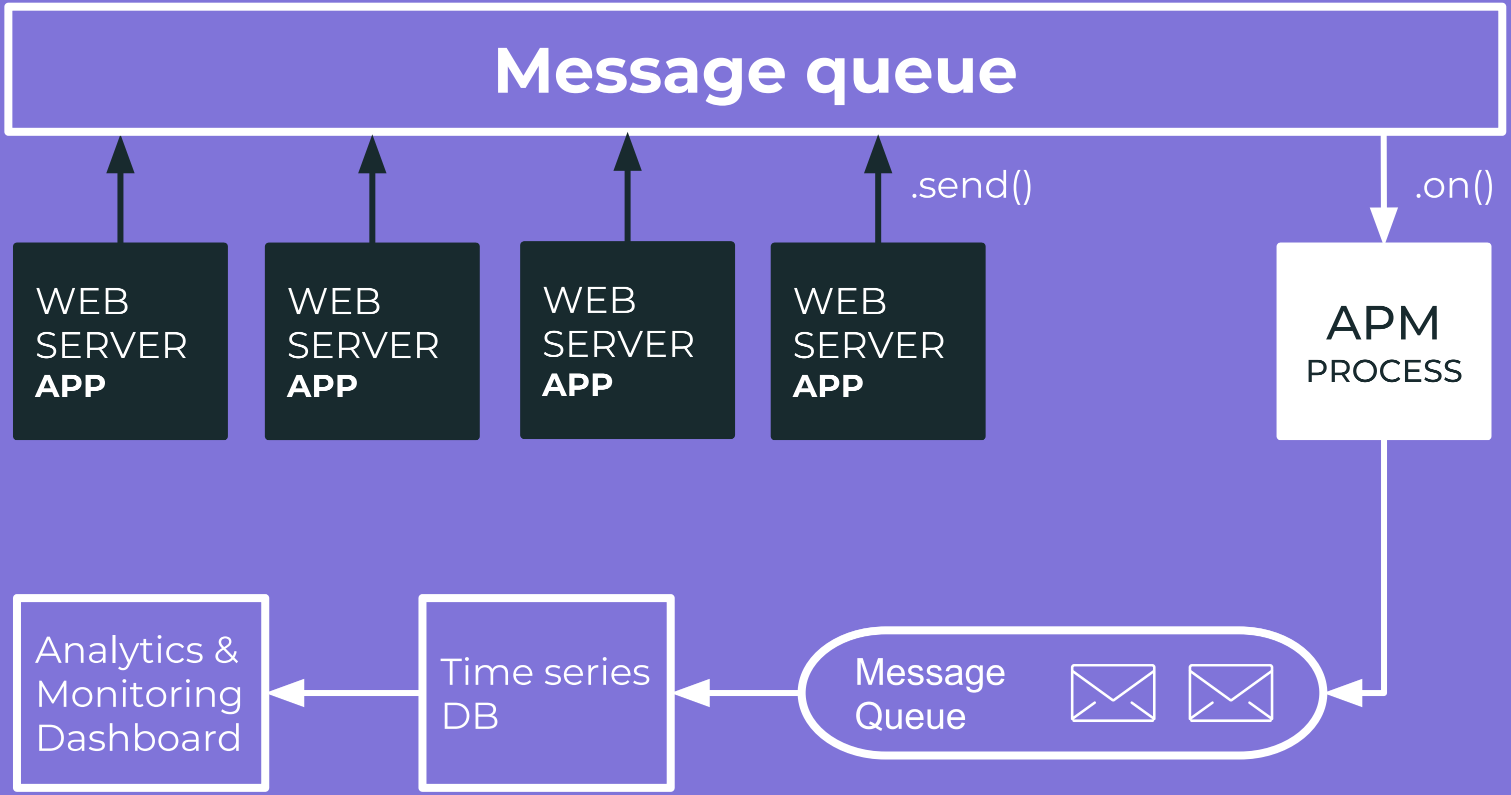
1. NewRelic
2. Dynatrace
3. OpenTracing
4. node-measured

IN-PROCESS APM AGENT

THE APM AGENT PROBLEMS ARE APPLICATION PROBLEMS



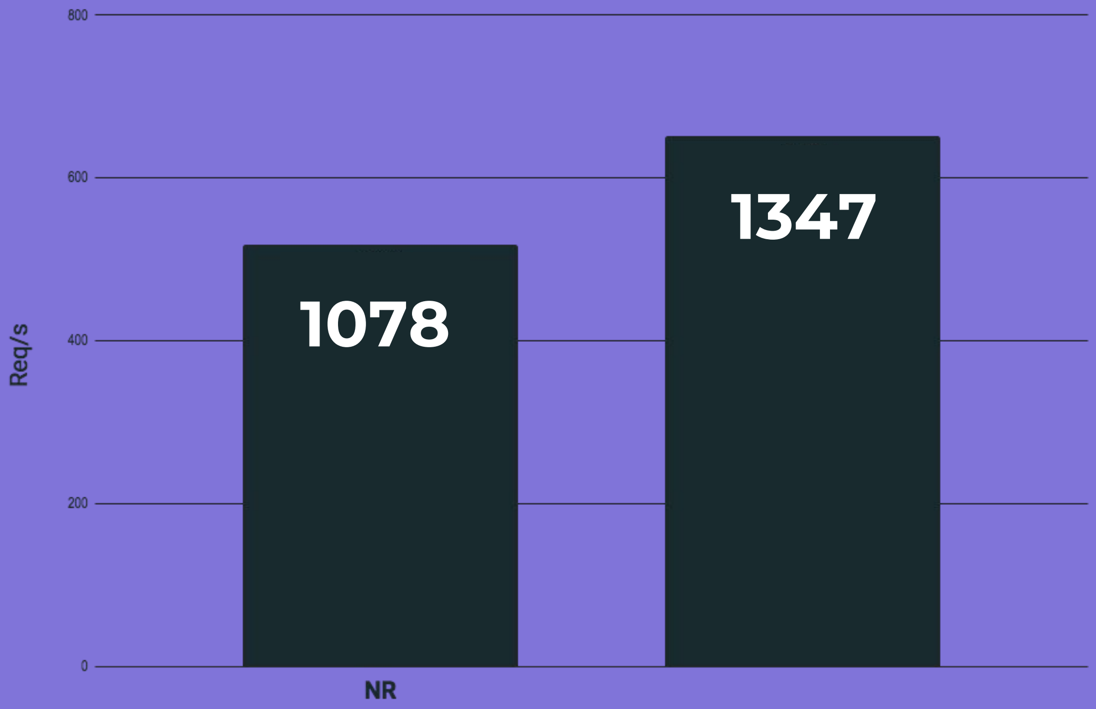
OFF-PROCESS APM AGENT



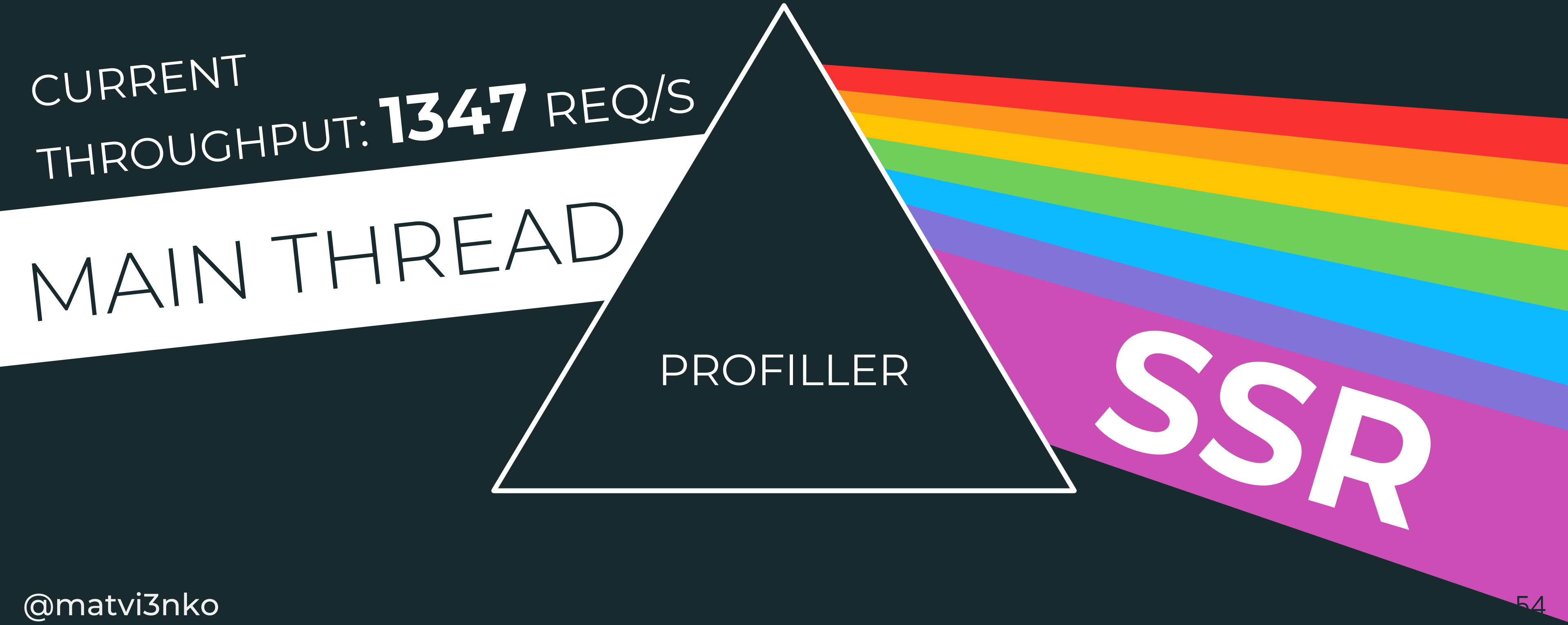
OFF-PROCESS MONITORING RESULT

+25%

MORE REQ/S
WITH
OFF-PROCESS
METRIC AGENT



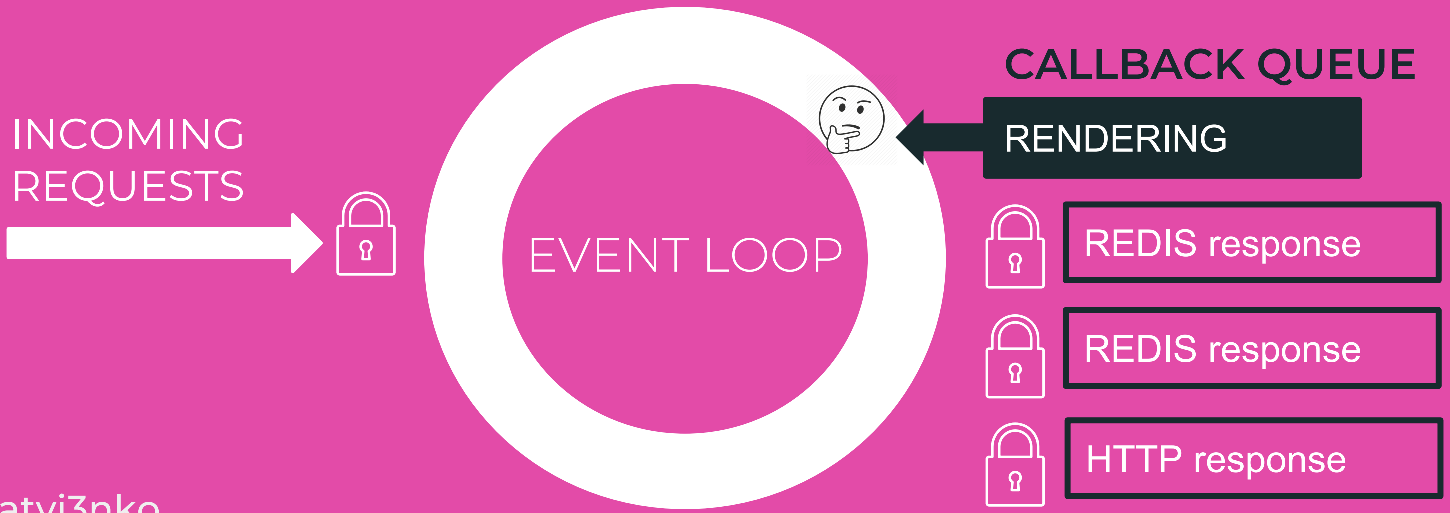
SERVER SIDE RENDERING



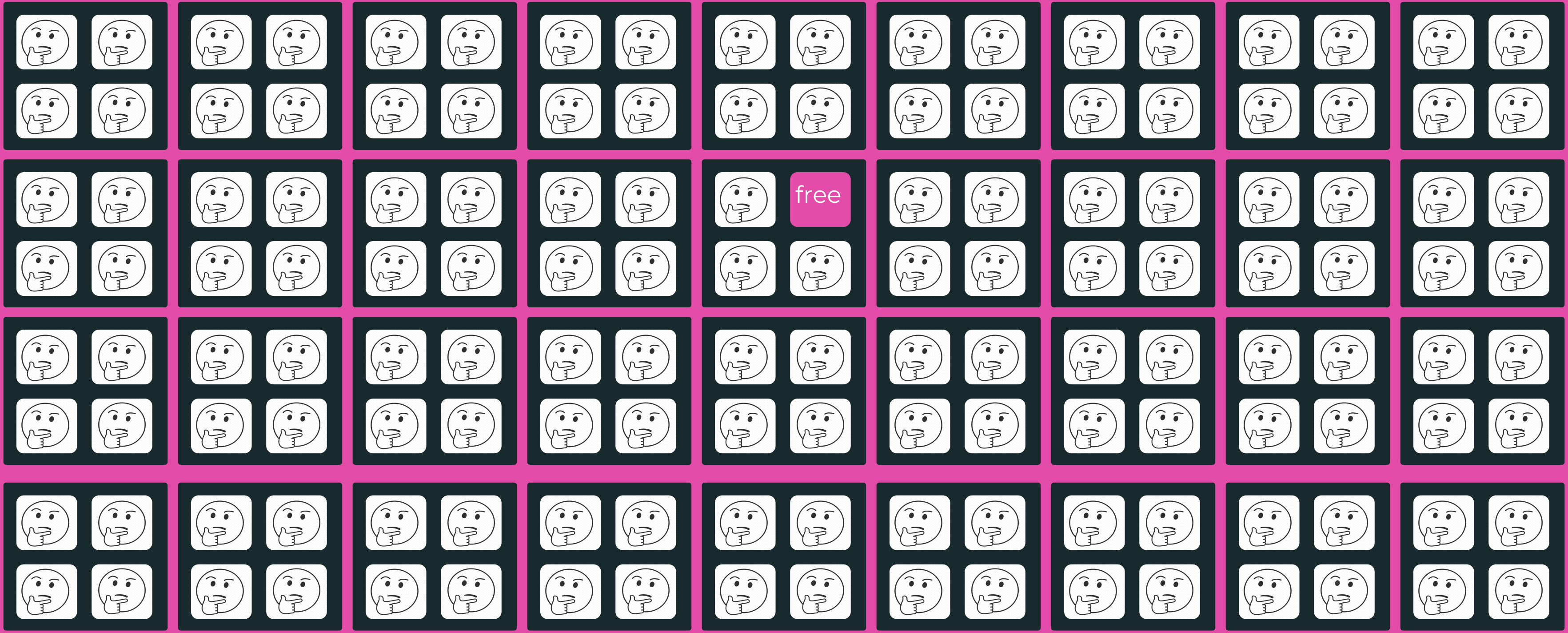
SERVER-SIDE RENDERING

MAIN THREAD
APP CODE EXECUTION

RENDERING



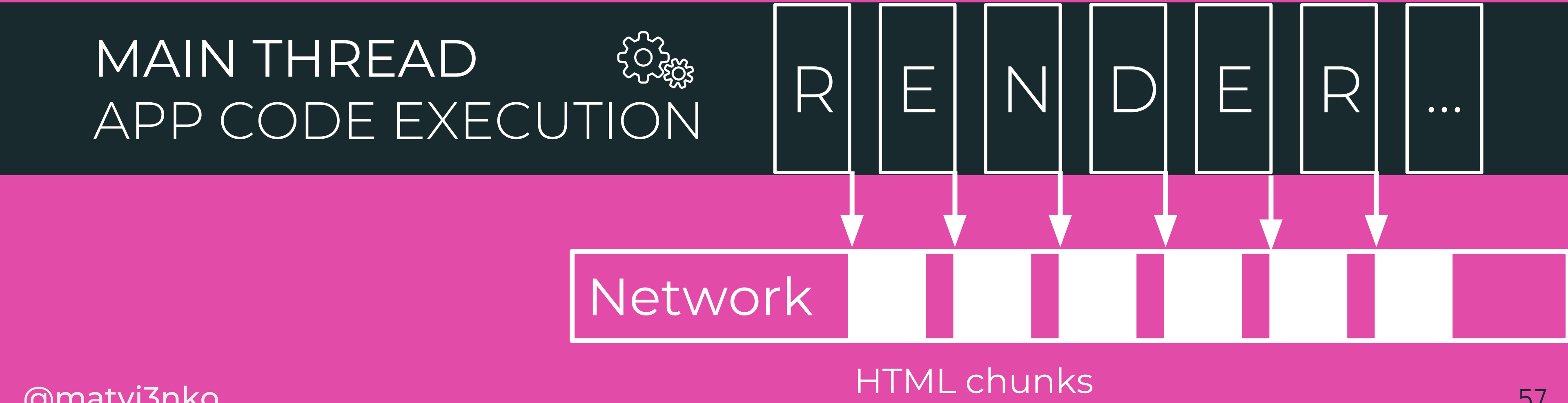
RENDERING ON NODE.JS



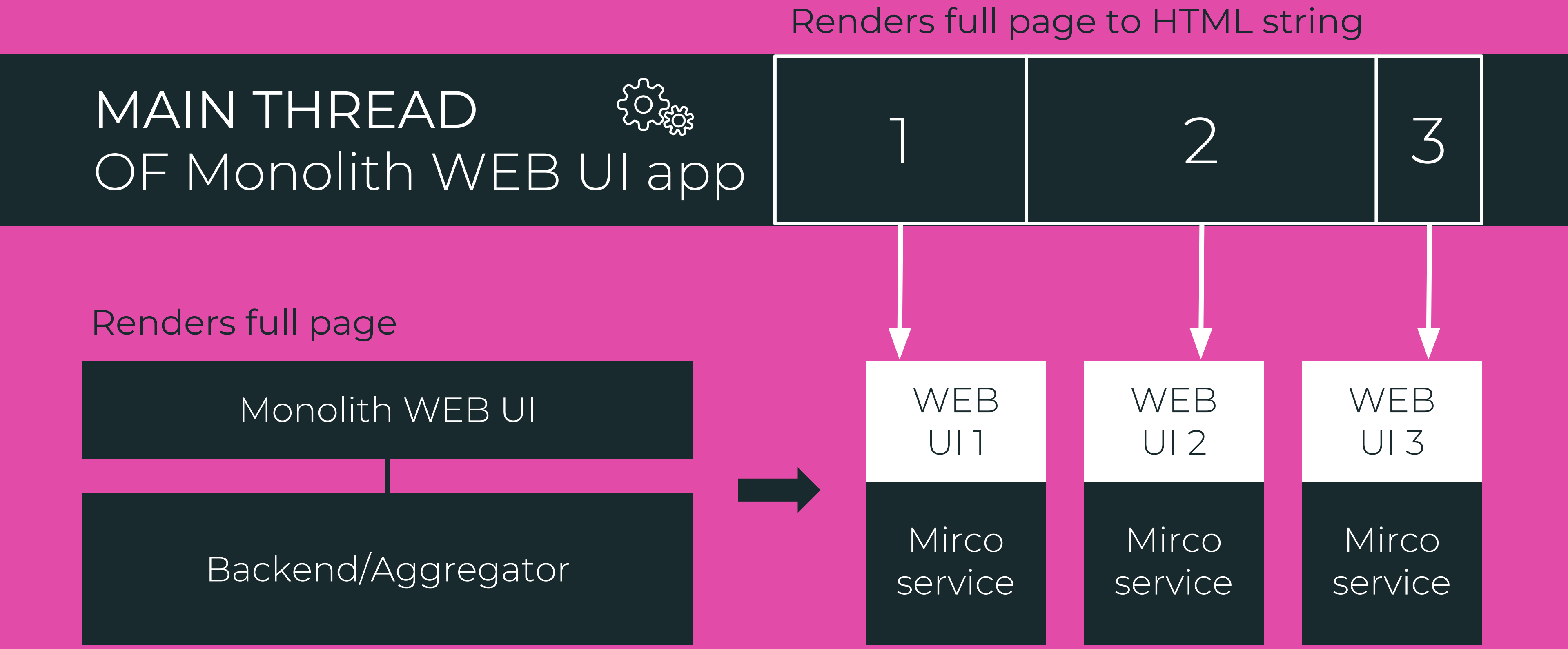
STREAMING SERVER-SIDE RENDERING

```
renderStream.pipe(res, { end: 'false' });
renderStream.on('end', () =>
{response.end('</div></body></html>'); });
```

Asynchronous execution in STREAM with **REACT 16**

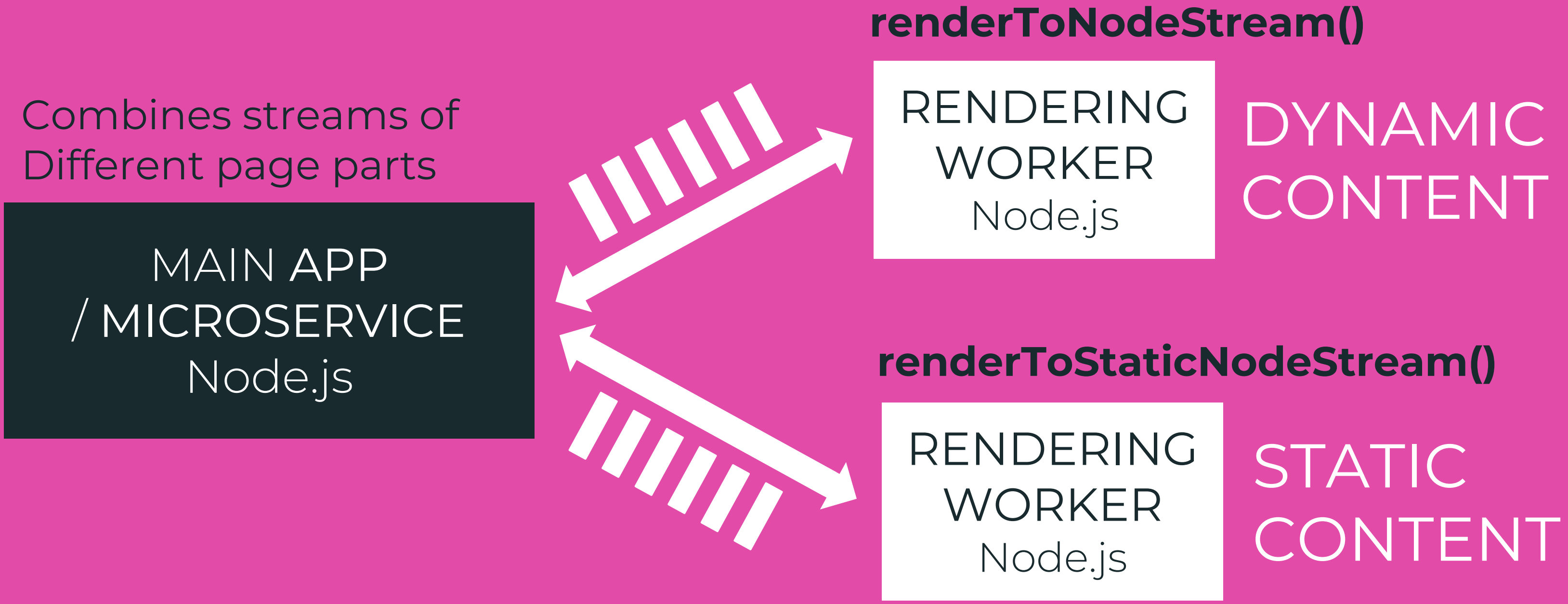


MICRO FRONTENDS



@matvi3nko

PARALLEL RENDERING WITH WORKERS


renderToNodeStream()
Combines streams of Different page parts
MAIN APP / MICROSERVICE Node.js
RENDERING WORKER Node.js
DYNAMIC CONTENT
renderToStaticNodeStream()
RENDERING WORKER Node.js
STATIC CONTENT

REACT 16

2X

* Average value.

THROUGHPUT

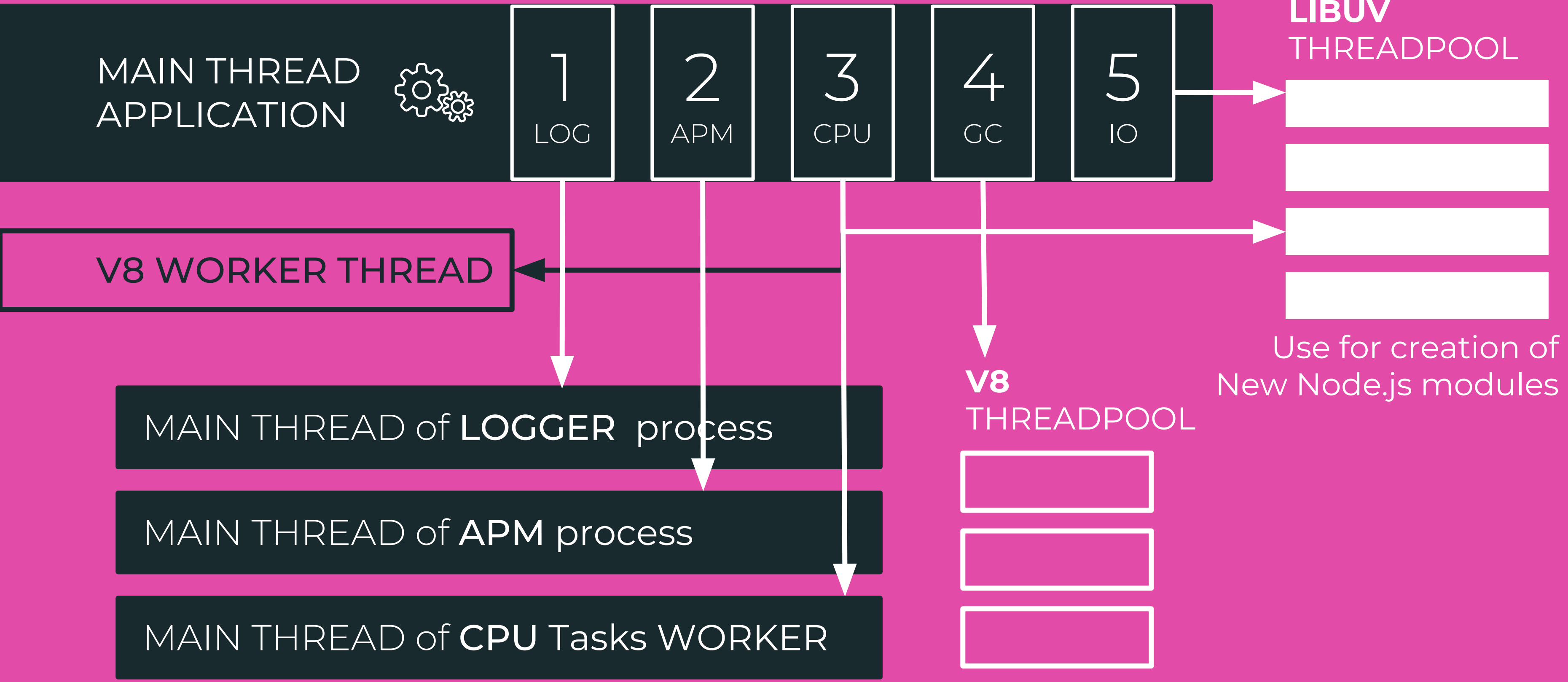
FROM

190 REQ/S

TO 1350

@matvi3nko

DECOMPOSED MAIN THREAD



@matvi3nko

ORDER

FROM LONGEST OPERATION

- CPU BOUND TASKS
- RENDERING
- SERIALIZATION
- ....

FINALLY

10X REQ/S

THROUGHPUT

REFERENCES

https://github.com/**matvi3nko**/Decomposition-of-the-Main-Thread-in-Node.js

THANKS!

Nikolay Matvienko

matvi3nko@gmail.com

Twitter.com/matvi3nko

github.com/matvi3nko

REFERENCES

Long-running Background Process in Node.js

https://vimeo.com/229536743

Background tasks in Node.js

https://www.youtube.com/watch?v=NNTsHzER31I&t=2207s

https://blog.evantahler.com/background-tasks-in-node-js-a-survey-with-redis-971d3575d9d2

Streaming Server-Side Rendering and Caching

https://zeit.co/blog/streaming-server-rendering-at-spectrum

https://github.com/zalando/tailor

Microservices on UI

https://www.youtube.com/watch?v=3l9IP9j5n1o

https://www.youtube.com/watch?v=E6_UyQPmiSg&t=2997s

REFERENCES

New Garbage Collection with threads

https://v8project.blogspot.ru/2017/11/

https://v8project.blogspot.com/2016/04/jank-busters-part-two-orinoco.html

Pino

https://github.com/pinojs/pino

New Worker API in Node.js discussion

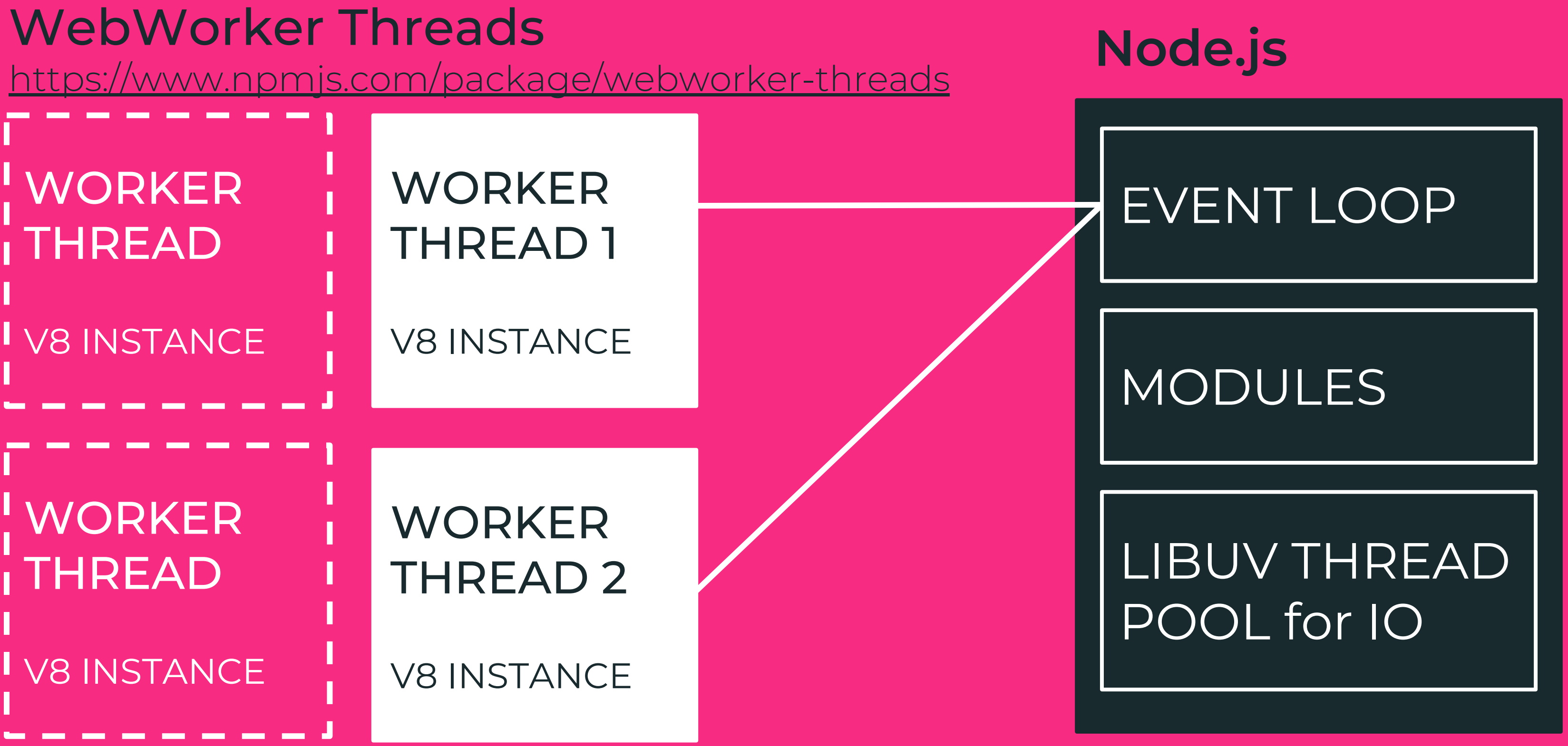
https://github.com/nodejs/worker/issues/4

IPC Communication Performance
https://60devs.com/performance-of-inter-process-communications-in-nodejs.html

List of Parallel JS Projects

https://github.com/SyntheticSemantics/List-of-Parallel-JS-Projects

WEBWORKER THREADS



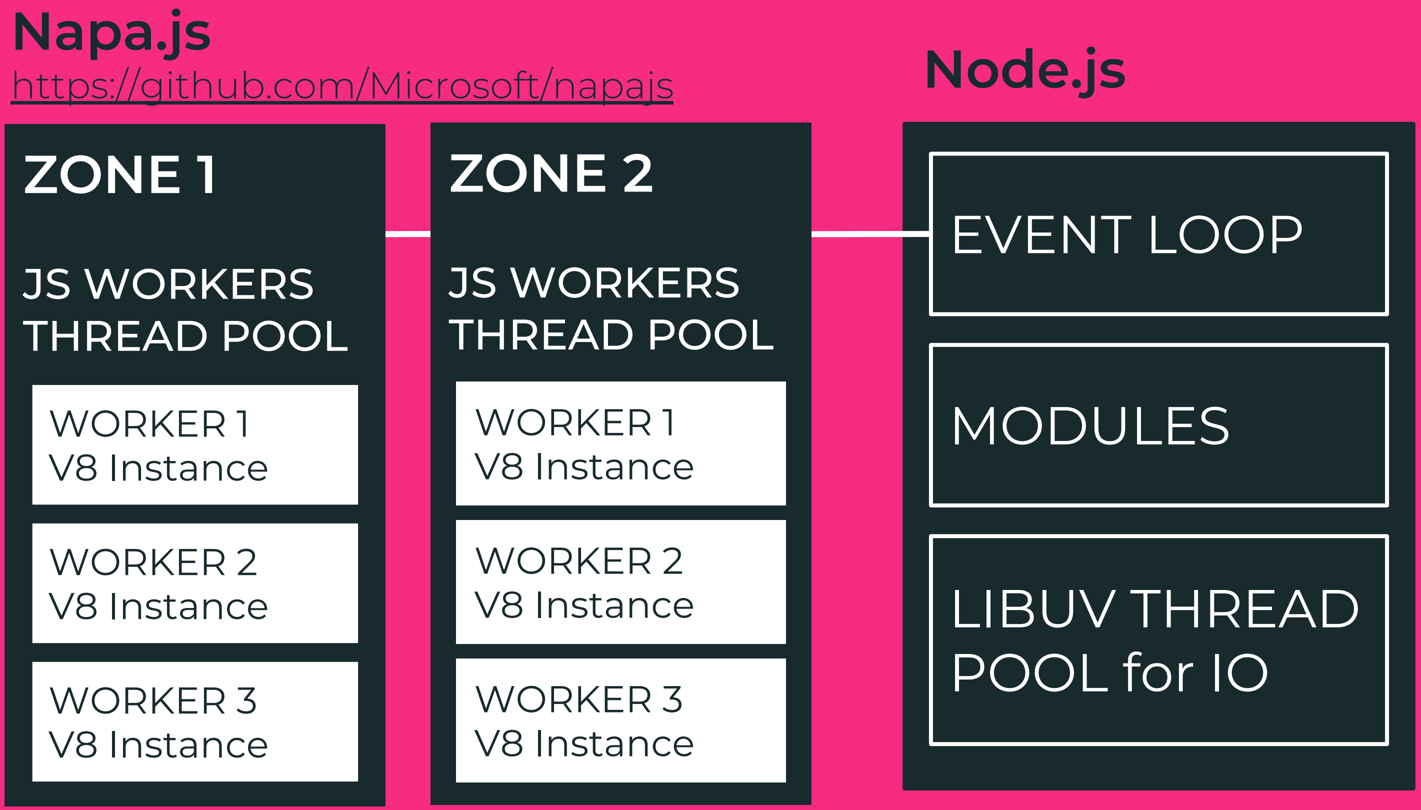
MICROSOFT NAPA.JS

MESSAGE PASSING
2x vs IPC

MEMORY USAGE
6.7 MB
vs 8 MB

STARTUP TIME
50 ms
vs 70 ms

Napa.js
https://github.com/Microsoft/napajs



ALIBABA ALIOS

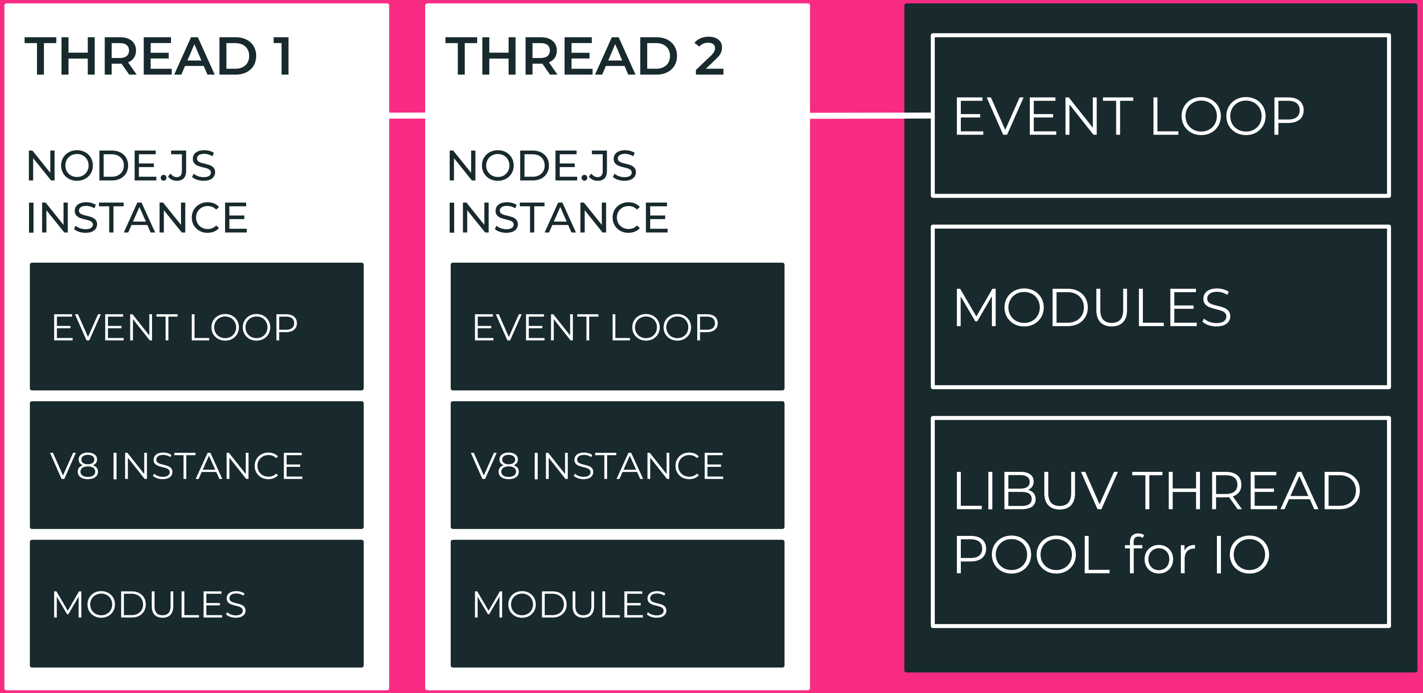
SHARED GLOBAL MEMORY

MEMORY USAGE
2.5 MB
vs 8 MB

STARTUP TIME
13 ms
vs 70 ms

ALiOS-node.js
https://github.com/alibaba/AliOS-nodejs

Node.js



@matvi3nko