

Лекция 6

Node.js

Node.js как платформа

Что такое Node.js?

Node.js - среда выполнения javascript
основанная на браузерном движке
Google V8.



На что Node.js способен

- веб-приложения (бэкенд);
- десктопные приложения ([Electron.js](#));
- микроконтроллеры (Esprimo), [ссылка](#);
- различные консольные утилиты;
- машинное обучение (TensorFlow.js), [ссылка](#);
- бла-бла-бла...

На что Node.js способен

- **веб-приложения (бэкенд);**
- десктопные приложения ([Electron.js](#));
- микроконтроллеры (Esprimo), [ссылка](#);
- различные консольные утилиты;
- машинное обучение (TensorFlow.js), [ссылка](#);
- бла-бла-бла...

ЛІЧНО Я

Backend Node.js Developer



Преимущества Node.js

Скорость

Node.js - это платформа для разработки очень быстрого сервера. По производительности спокойно может сравниться с Java, Kotlin и Golang.

При этом все вышеперечисленные языки имеют вложенные механизмы параллелизма (корутины, потоки и тд), а Node.js однопоточный.

Как так получается?

Асинхронное API

Движок V8 из коробки поддерживает исключительно асинхронное API, т.к. изначально был предназначен для работы в браузере на клиенте.

Это значит, что ни одна из read/write операций не блокирует основной поток исполнения (спасибо Event Loop).

Простота

Поднять простой веб-сервер на Node.js - это 15 строк кода (с натяжкой).

При этом, т.к. мы пишем на javascript, мы можем использовать различные парадигмы для написания кода.

Библиотеки

У Node.js огромное комьюнити, и с 2009 года существуют библиотеки для всего, что может теоретически понадобиться для серверной (и не только) разработки.

Количество javascript библиотек достигает такого огромного количества, что это уже даже достигло какого-то абсурдного уровня.

is-thirteen

2.0.0 • Public • Published 7 years ago

[Readme](#)[Code](#) Beta[1 Dependency](#)[6 Dependents](#)[2 Versions](#)

is-thirteen

[GITTER](#) [join chat](#)

Check if a number is equal to 13.

Installation

```
npm --save i is-thirteen
```

Usage

```
var isThirteen = require('is-thirteen');
```

Install

```
> npm i is-thirteen
```

Repository

github.com/jezen/is-thirteen

Homepage

github.com/jezen/is-thirteen#readme

Weekly Downloads

36



Version

2.0.0

License

WTFPL

JavaScript

Тут не нужно объяснений

Недостатки Node.js



16:54

У тебя было яблоко, тебе дали ещё одно

Сколько у тебя яблок?



16:55

Одно яблоко

Потому что яблоко + одно = одно яблоко



16:55

Твой родной язык это джаваскрипт что ли ~~о~~а ??

JavaScript

RunTime-ошибки

Это когда код запустился, но упал во время исполнения.

Такая возможность есть из-за того, что javascript интерпретируемый язык программирования.

```
1 const a = undefined;  
2 console.log('шалом ');  
3  
4 a.func(); // RunTime Exception  
5  
6 console.log('братья');  
7
```

Отсутствие типизации

Javascript - нестрого типизированный язык.

```
1 true + false
2 12 / "6"
3 "number" + 15 + 3
4 15 + 3 + "number"
5 [1] > null
6 "foo" + + "bar"
7 'true' == true
8 false == 'false'
9 null == ''
10 !!"false" == !!"true"
11 ['x'] == 'x'
12 [] + null + 1
13 0 || "0" && {}
14 [1,2,3] == [1,2,3]
15 {}+[]+{}+[1]
16 !+[]+[]+![]
17 new Date(0) - 0
18 new Date(0) + 0
19
```

Риск наговнокодить

Риск написать плохой код на JavaScript настолько высок, что сравнить его можно, наверно, только с Python. Обилие поддерживаемых парадигм, нестрогая типизация, сложная асинхронность, лексическое окружение, замыкания.

```
async registration(Form) {  
  console.log('ajax post');  
  return await ajax.post(backend.upload, new FormData(Form), true)  
    .then(({status, responseObject}) => {  
    let photo_name;  
    if (status === 200 ) {  
      photo_name = new Promise((resolve, reject) => {  
        resolve(JSON.stringify(responseObject));  
        this.linkImage.push(responseObject.replaceAll('\"', ''));  
      });  
      console.log(photo_name);  
      return ajax.post(backend.signup, this.Json());  
    }  
  
    if (status === 400) {  
      throw new Error('Слишком большой размер фото, пожалуйста, загрузите фото меньшего размера');  
    }  
  
    if (status === 403) {  
      throw new Error('Пожалуйста, загрузите фото с вашим лицом');  
    }  
  
    if (status === 500) {  
      throw new Error('Неизвестная ошибка, пожалуйста, попробуйте позже');  
    }  
  });  
}
```

Большая математика

До недавнего времени в javascript не поддерживались числа больше 9007199254740992. Недавно появился bigint и эта проблема частично решена.

```
1 const bigNumber = 9007199254740992;  
2  
3 console.log(bigNumber); // Infinity  
4
```

А также в главных ролях

- отсутствие многопоточки;
- медленные вычисления;
- плохое ООП, несмотря на его наличие;
- скудные коллекции (по сравнению с java, но великолепные по-сравнению с Go)
- <подставьте сюда свою причину>

Особенности и использование

Как использовать Node.js

1. Создаем файл <название>.js
2. Пишем туда код на javascript
3. Запускаем командой `node <название>.js`
4. Profit!

Стандартная библиотека

- `events` - встроенный `EventEmitter`
- `fs` - работа с файловой системой
- `os` - работа с операционной системой
- `http` - основной модуль для создания `http`-серверов
- `net` - низкоуровневое сетевое соединение
- `path` - для работы с путями в директориях
- и еще много-много, почитать можно [здесь](#)

CommonJS модули

```
1 // file.js
2
3 let count = 0
4 const increase = () => ++count
5 const reset = () => {
6     count = 0
7     console.log('Счетчик сброшен.')
8 }
9
10 exports.increase = increase
11 exports.reset = reset
12
13 // или (эквивалентно)
14 module.exports = {
15     increase,
16     reset
17 }
18
```

```
1 // используем CommonJS модуль
2 const {
3     increase,
4     reset
5 } = require('./file')
6
7 increase()
8 reset()
9
10 // или
11 const file = require('./file')
12 file.increase()
13 file.reset()
14
```

global и globalThis

`global` - аналог `window` в среде Node.js.

Если нет уверенности, где будет исполняться ваш код, или использоваться написанная вами библиотека - используйте `globalThis`.

Так как его используют? Этот ваш Node.js

Как уже повторяли для серверной разработки. Например:

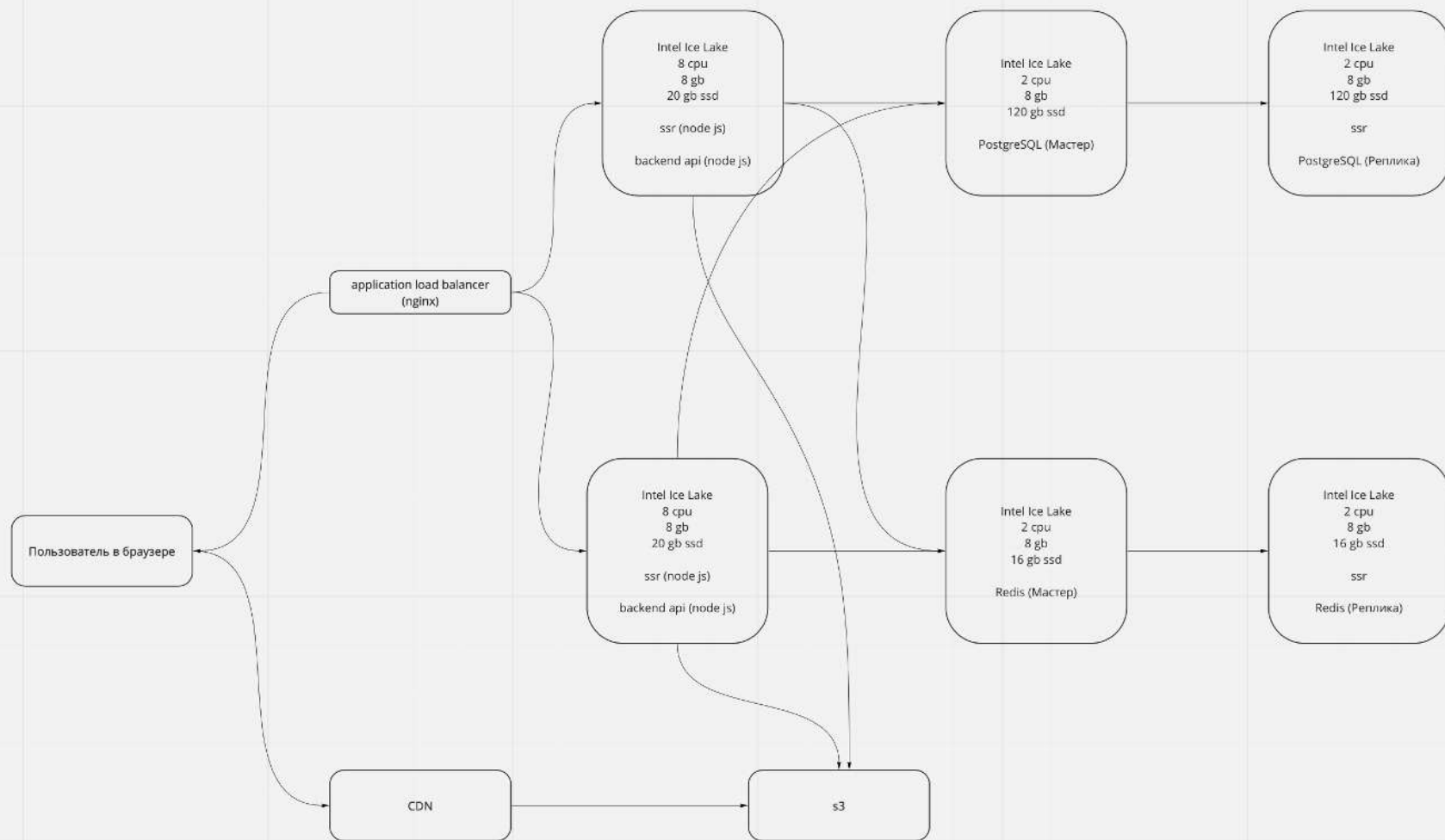
- SSR - серверный рендеринг для SPA
- BFF - бэкенд-прокси для агрегации данных
- Full-Backend API - полноценный бэкенд
- различные микросервисы со специфической логикой

Поглядим примерчики

А что там все таки по скорости?
Давайте посмотрим пример

RPS

RPS - это метрика, которая определяет сколько запросов в секунду выдерживает сервис. Необходим для оценки нагрузки от пользователей и зачастую используется в расчетах.



Высоконагруженность 200 rps

Data delay: 3s, RPS: 202

Percentiles (all/last 1m/last), ms:

100.0%	<	5,060.0	1,085.0	149.2
99.5%	<	489.0	99.0	142.0
99.0%	<	262.0	87.0	122.3
95.0%	<	71.0	66.0	68.7
90.0%	<	61.0	60.0	61.2
85.0%	<	56.0	56.0	59.2
80.0%	<	54.0	54.0	56.2
75.0%	<	52.0	52.0	54.0
70.0%	<	50.0	51.0	53.0
60.0%	<	48.0	48.0	50.0
50.0%	<	46.0	46.0	47.3
40.0%	<	44.0	44.0	45.0
30.0%	<	42.0	42.0	44.0
20.0%	<	41.0	41.0	42.0
10.0%	<	39.0	38.0	39.0

HTTP codes:

89,498 +202 100.00% : 200 OK

Net codes:

89,498 +202 100.00% : 0 Success

Average Sizes (all/last), bytes:

Request: 56.0 / 56.0

Response: 23,462.9 / 23,462.7

Average Times (all/last), ms:

Overall: 53.65 / 50.06

Connect: 3.73 / 1.20

Send: 0.01 / 0.01

Latency: 43.91 / 42.82

Receive: 6.00 / 6.02

Cumulative Cases Info:

	name	count	% last	net_e	http_e	avg ms	last ms
OVERALL:	89,498	100.00%	+202	0	0	53.7	50.1

Duration: 0:10:00

ETA: 0:00:00

Hosts: ef31i12ouueo70d8epl6 => 51.250

Ammo:

Count: 90000

Load: line(100, 200, 10m)

Active instances: 14

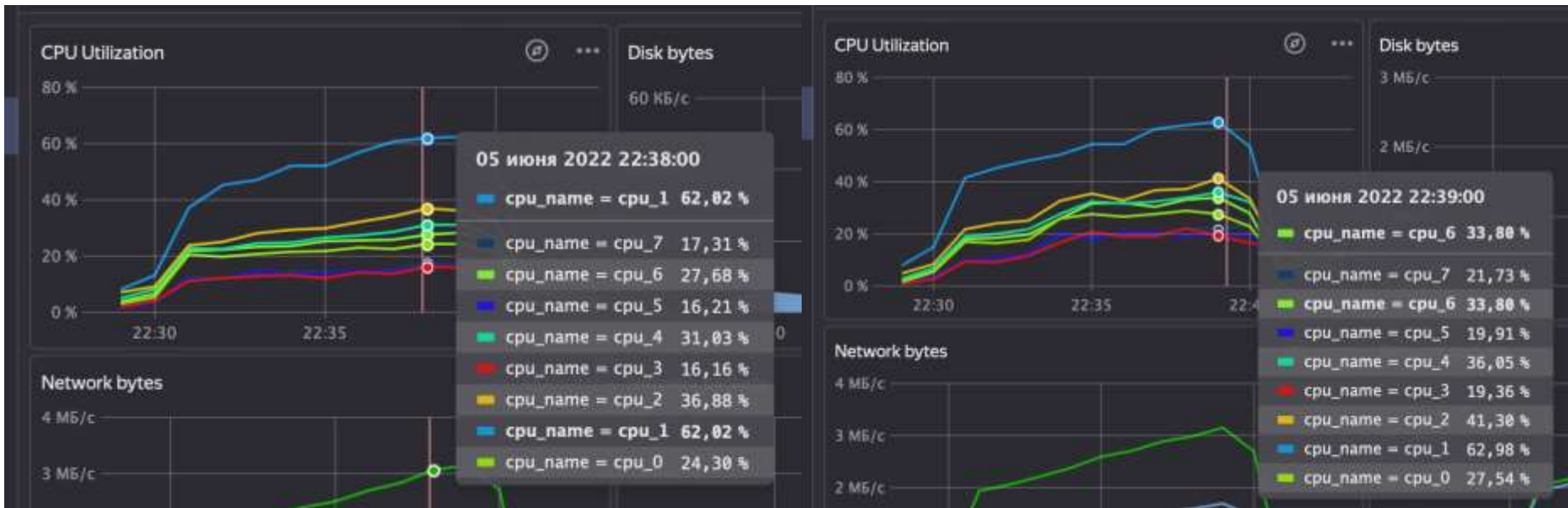
Planned requests: 200.0 for 0:00:00

Actual responses: 202

Accuracy: 0.00%

Time lag: 0:00:00

Высоконагруженность 200 rps



Высоконагруженность 200 rps

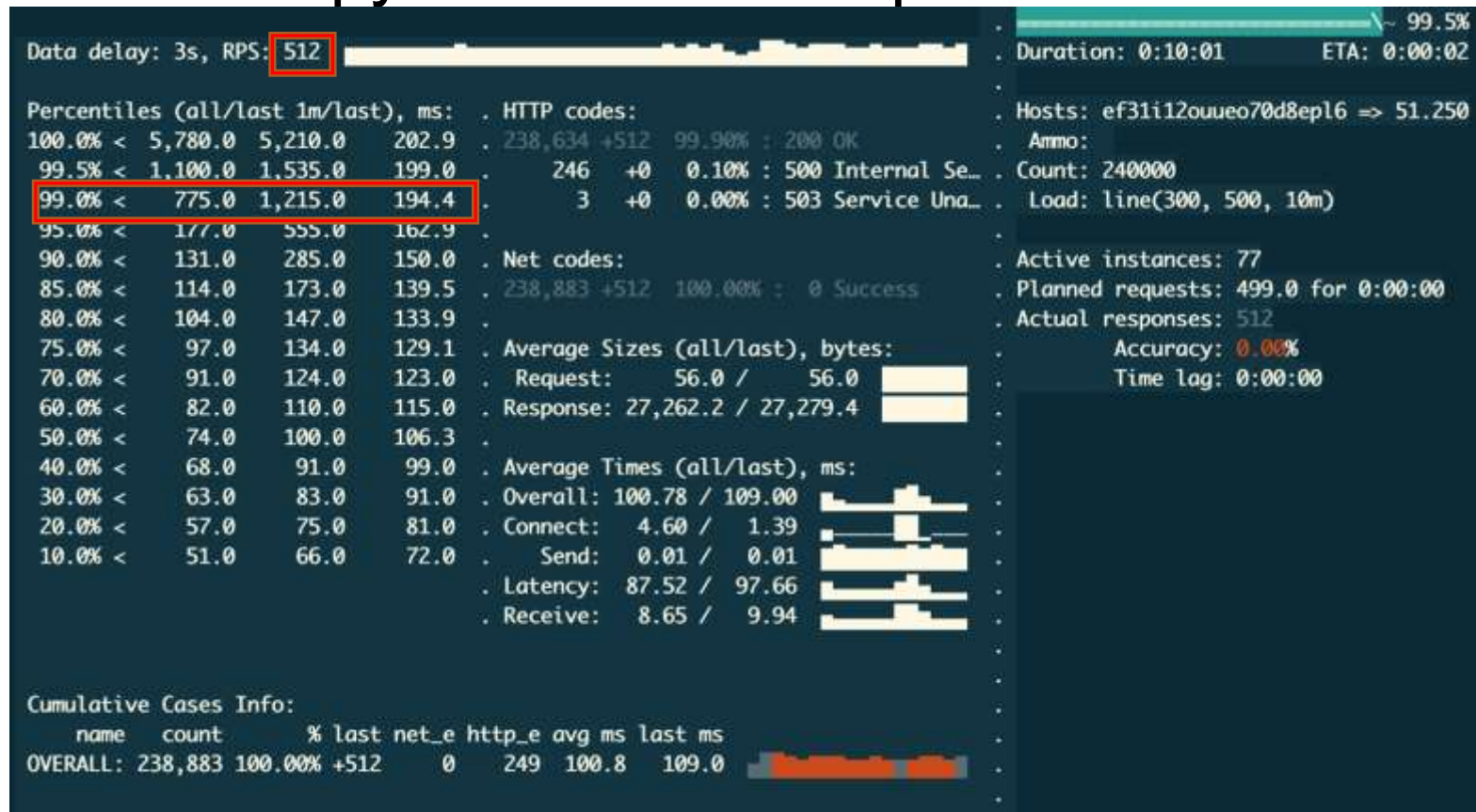
Process List				
[8]	svarog-api	Mem: 123 MB	CPU: 0 %	onlin
[9]	svarog-api	Mem: 123 MB	CPU: 0 %	onlin
[10]	svarog-api	Mem: 124 MB	CPU: 6 %	onlin
[11]	svarog-api	Mem: 128 MB	CPU: 4 %	onlin
[12]	svarog-api	Mem: 123 MB	CPU: 4 %	onlin
[13]	svarog-api	Mem: 130 MB	CPU: 2 %	onlin
[14]	svarog-api	Mem: 125 MB	CPU: 2 %	onlin
[15]	svarog-api	Mem: 128 MB	CPU: 4 %	onlin
[26]	svarog-ui	Mem: 193 MB	CPU: 15 %	online
[27]	svarog-ui	Mem: 198 MB	CPU: 2 %	online
[28]	svarog-ui	Mem: 212 MB	CPU: 37 %	online
[29]	svarog-ui	Mem: 195 MB	CPU: 15 %	online
[30]	svarog-ui	Mem: 195 MB	CPU: 13 %	online
[31]	svarog-ui	Mem: 187 MB	CPU: 19 %	online
[32]	svarog-ui	Mem: 203 MB	CPU: 4 %	online
[33]	svarog-ui	Mem: 202 MB	CPU: 8 %	online

[8]	svarog-api	Mem: 123 MB	CPU: 2 %	onlin
[9]	svarog-api	Mem: 123 MB	CPU: 0 %	onlin
[10]	svarog-api	Mem: 125 MB	CPU: 9 %	onlin
[11]	svarog-api	Mem: 128 MB	CPU: 9 %	onlin
[12]	svarog-api	Mem: 123 MB	CPU: 7 %	onlin
[13]	svarog-api	Mem: 131 MB	CPU: 4 %	onlin
[14]	svarog-api	Mem: 125 MB	CPU: 0 %	onlin
[15]	svarog-api	Mem: 128 MB	CPU: 0 %	onlin
[26]	svarog-ui	Mem: 196 MB	CPU: 14 %	online
[27]	svarog-ui	Mem: 198 MB	CPU: 2 %	online
[28]	svarog-ui	Mem: 224 MB	CPU: 41 %	online
[29]	svarog-ui	Mem: 199 MB	CPU: 7 %	online
[30]	svarog-ui	Mem: 195 MB	CPU: 12 %	online
[31]	svarog-ui	Mem: 187 MB	CPU: 2 %	online
[32]	svarog-ui	Mem: 153 MB	CPU: 9 %	online
[33]	svarog-ui	Mem: 201 MB	CPU: 22 %	online

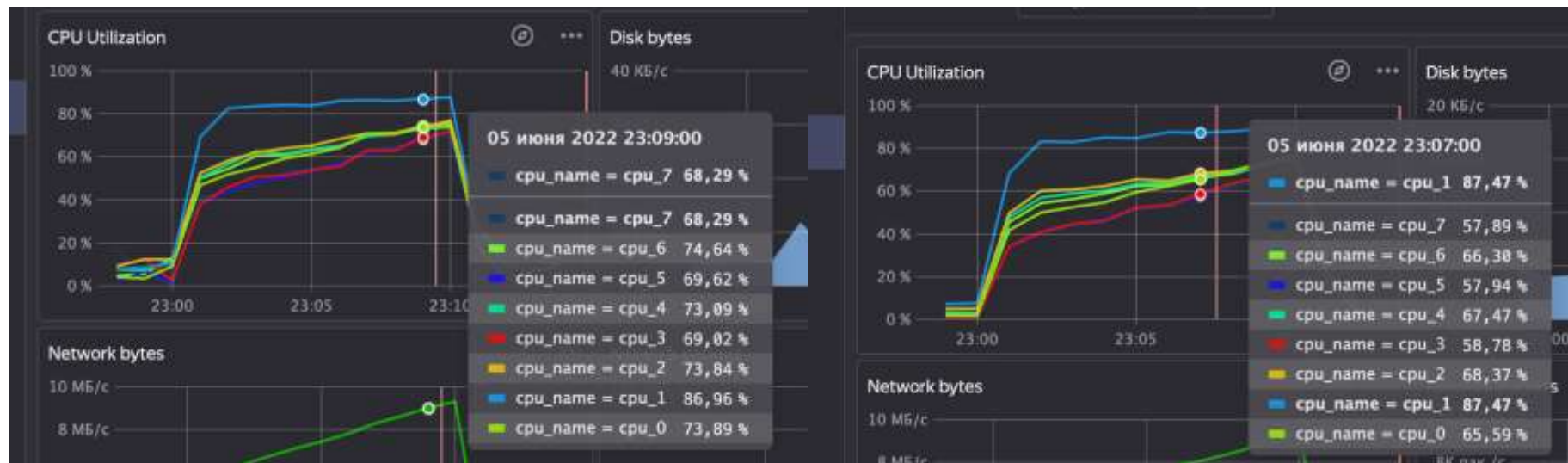
Высоконагруженность 200 rps



Высоконагруженность 500 rps



Высоконагруженность 500 rps

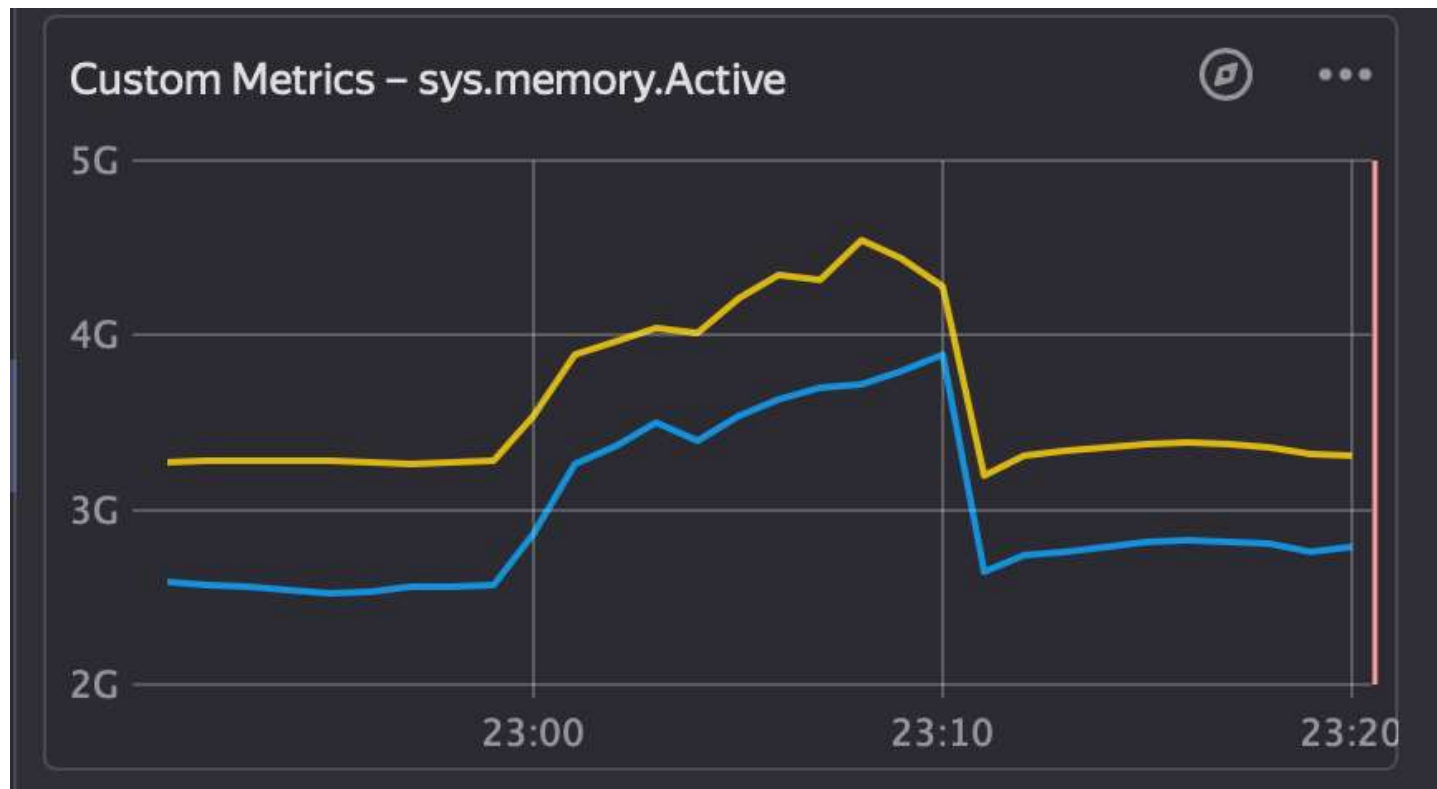


Высоконагруженность 500 rps

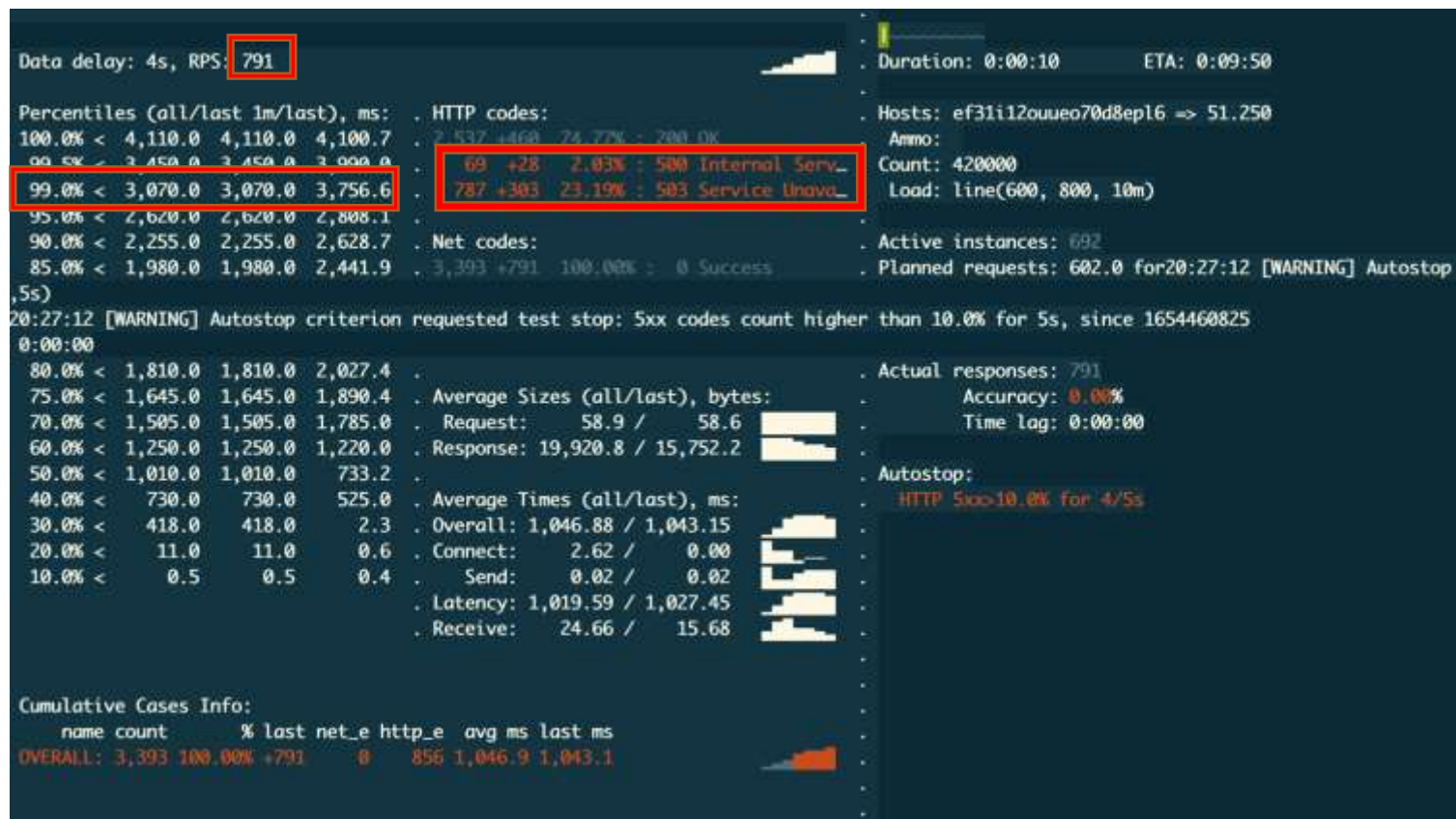
Process List				
[8]	svarog-api	Mem: 126 MB	CPU: 28 %	onlin
[9]	svarog-api	Mem: 127 MB	CPU: 19 %	onlin
[10]	svarog-api	Mem: 128 MB	CPU: 8 %	onlin
[11]	svarog-api	Mem: 130 MB	CPU: 14 %	onlin
[12]	svarog-api	Mem: 101 MB	CPU: 26 %	onlin
[13]	svarog-api	Mem: 130 MB	CPU: 14 %	onlin
[14]	svarog-api	Mem: 126 MB	CPU: 21 %	onlin
[15]	svarog-api	Mem: 129 MB	CPU: 11 %	onlin
[26]	svarog-ui	Mem: 373 MB	CPU: 60 %	online
[27]	svarog-ui	Mem: 268 MB	CPU: 85 %	online
[28]	svarog-ui	Mem: 269 MB	CPU: 55 %	online
[29]	svarog-ui	Mem: 291 MB	CPU: 68 %	online
[30]	svarog-ui	Mem: 265 MB	CPU: 58 %	online
[31]	svarog-ui	Mem: 248 MB	CPU: 71 %	online
[32]	svarog-ui	Mem: 231 MB	CPU: 46 %	online
[33]	svarog-ui	Mem: 364 MB	CPU: 40 %	online

Process List				
[8]	svarog-api	Mem: 127 MB	CPU: 18 %	onlin
[9]	svarog-api	Mem: 127 MB	CPU: 12 %	onlin
[10]	svarog-api	Mem: 123 MB	CPU: 4 %	onlin
[11]	svarog-api	Mem: 137 MB	CPU: 16 %	onlin
[12]	svarog-api	Mem: 90 MB	CPU: 24 %	onlin
[13]	svarog-api	Mem: 131 MB	CPU: 10 %	onlin
[14]	svarog-api	Mem: 130 MB	CPU: 25 %	onlin
[15]	svarog-api	Mem: 130 MB	CPU: 1 %	onlin
[26]	svarog-ui	Mem: 308 MB	CPU: 21 %	online
[27]	svarog-ui	Mem: 279 MB	CPU: 39 %	online
[28]	svarog-ui	Mem: 275 MB	CPU: 40 %	online
[29]	svarog-ui	Mem: 327 MB	CPU: 62 %	online
[30]	svarog-ui	Mem: 237 MB	CPU: 25 %	online
[31]	svarog-ui	Mem: 267 MB	CPU: 22 %	online
[32]	svarog-ui	Mem: 246 MB	CPU: 47 %	online
[33]	svarog-ui	Mem: 368 MB	CPU: 54 %	online

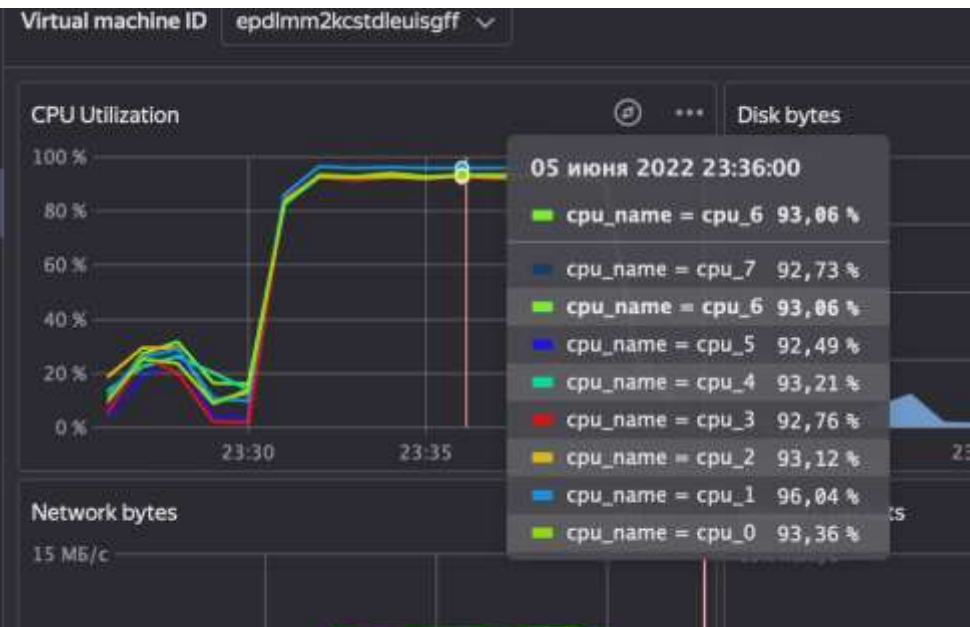
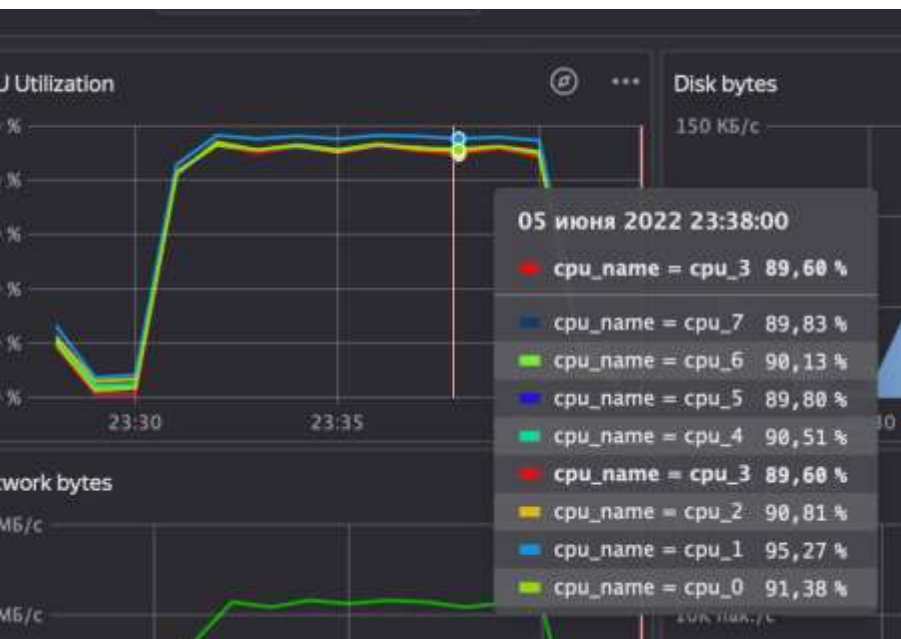
Высоконагруженность 500 rps



Высоконагруженность 800 rps



Высоконагруженность 800 rps

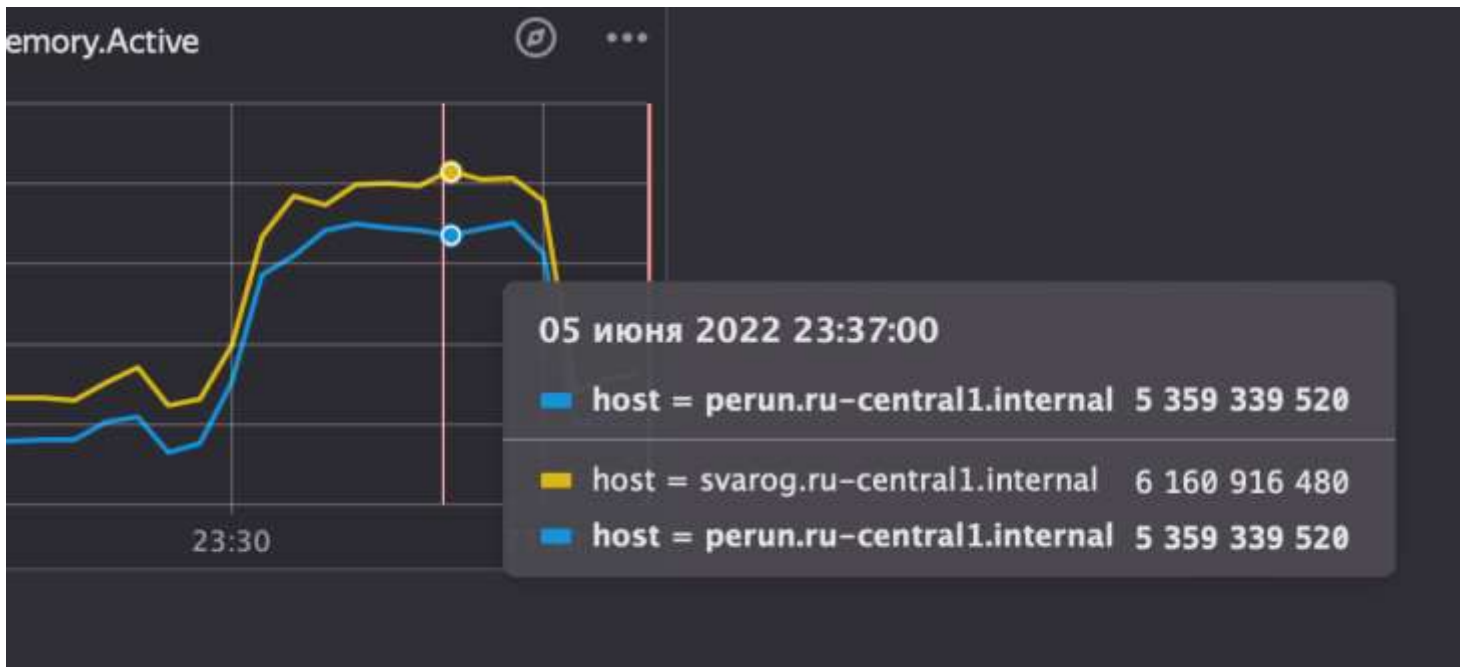


Высоконагруженность 800 rps

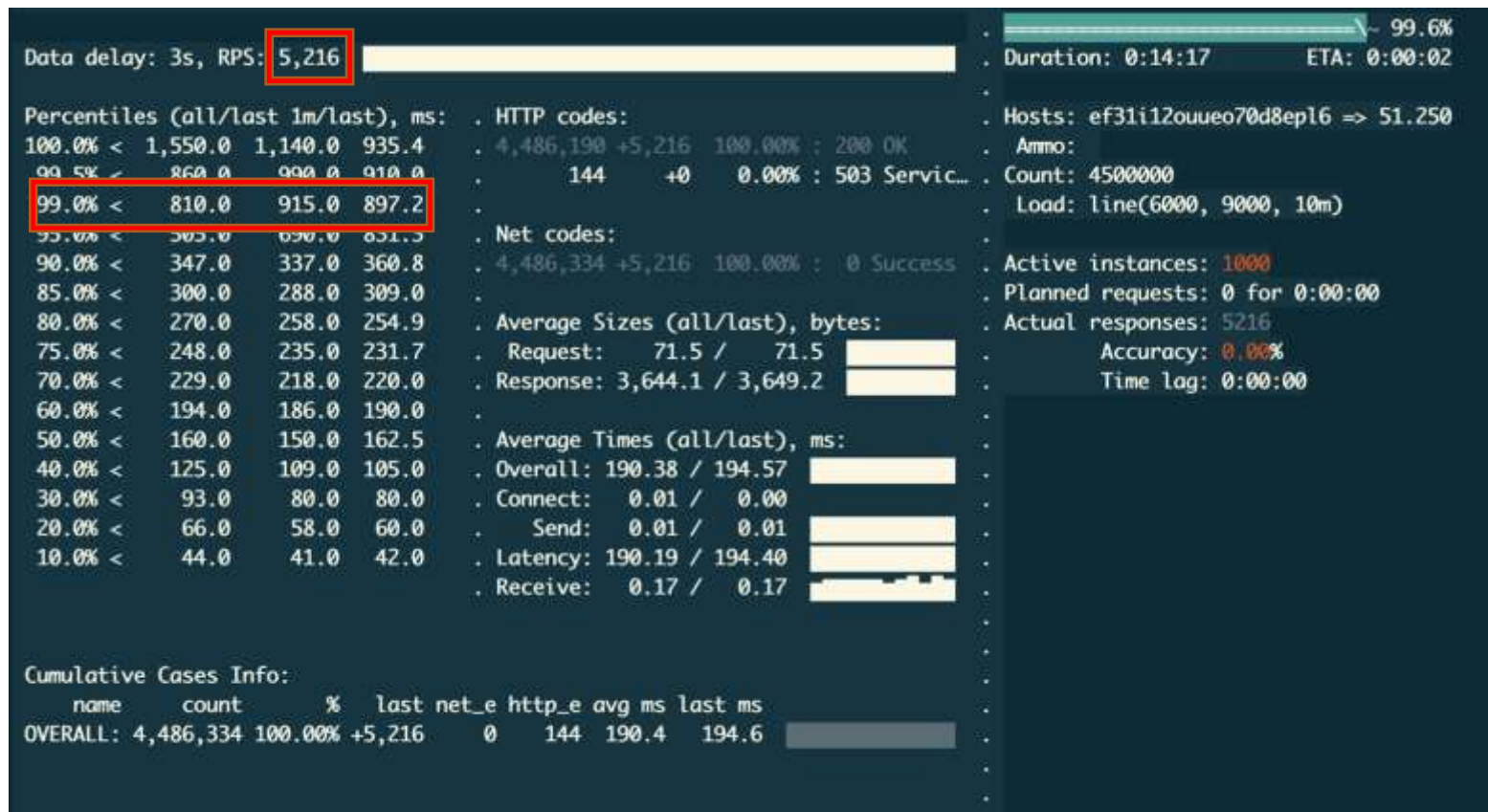
```
[ 8] svarog-api      Mem: 224 MB    CPU: 15 %  onlin
[ 9] svarog-api      Mem: 201 MB    CPU: 10 %  onlin
[10] svarog-api      Mem: 152 MB    CPU: 19 %  onlin
[11] svarog-api      Mem: 231 MB    CPU: 11 %  onlin
[12] svarog-api      Mem: 163 MB    CPU: 24 %  onlin
[13] svarog-api      Mem: 159 MB    CPU: 20 %  onlin
[14] svarog-api      Mem: 153 MB    CPU: 24 %  onlin
[15] svarog-api      Mem: 145 MB    CPU: 14 %  onlin
[26] svarog-ui       Mem: 481 MB    CPU: 64 %  online
[27] svarog-ui       Mem: 314 MB    CPU: 69 %  online
[28] svarog-ui       Mem: 462 MB    CPU: 65 %  online
[29] svarog-ui       Mem: 393 MB    CPU: 68 %  online
[30] svarog-ui       Mem: 467 MB    CPU: 65 %  online
[31] svarog-ui       Mem: 480 MB    CPU: 67 %  online
[32] svarog-ui       Mem: 447 MB    CPU: 69 %  online
[33] svarog-ui       Mem: 468 MB    CPU: 65 %  online
```

```
Process list
[ 8] svarog-api      Mem: 136 MB    CPU: 9 %   onlin
[ 9] svarog-api      Mem: 214 MB    CPU: 11 %  onlin
[10] svarog-api      Mem: 216 MB    CPU: 11 %  onlin
[11] svarog-api      Mem: 205 MB    CPU: 5 %   onlin
[12] svarog-api      Mem: 159 MB    CPU: 27 %  onlin
[13] svarog-api      Mem: 136 MB    CPU: 15 %  onlin
[14] svarog-api      Mem: 159 MB    CPU: 23 %  onlin
[15] svarog-api      Mem: 145 MB    CPU: 23 %  onlin
[26] svarog-ui       Mem: 455 MB    CPU: 64 %  online
[27] svarog-ui       Mem: 411 MB    CPU: 86 %  online
[28] svarog-ui       Mem: 460 MB    CPU: 60 %  online
[29] svarog-ui       Mem: 375 MB    CPU: 70 %  online
[30] svarog-ui       Mem: 483 MB    CPU: 68 %  online
[31] svarog-ui       Mem: 482 MB    CPU: 70 %  online
[32] svarog-ui       Mem: 466 MB    CPU: 39 %  online
[33] svarog-ui       Mem: 468 MB    CPU: 70 %  online
```

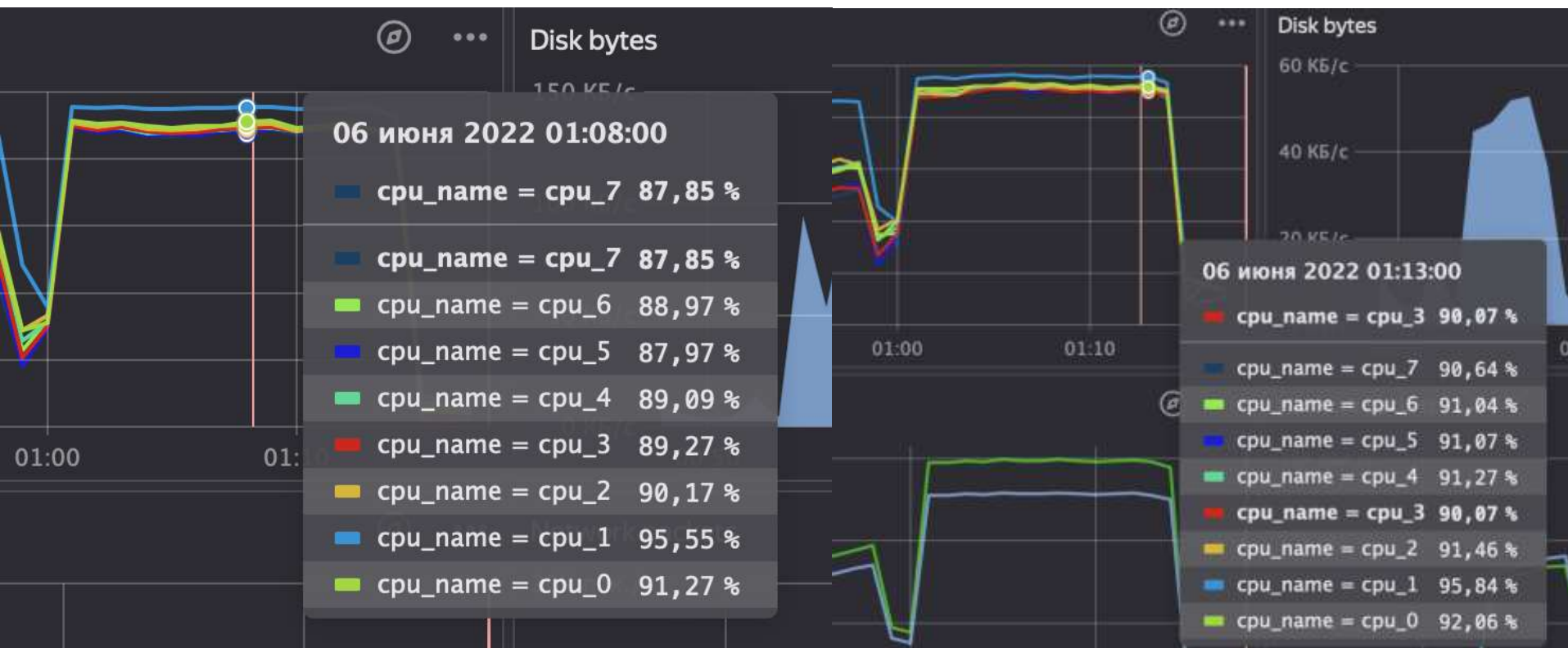
Высоконагруженность 800 rps



Высоконагруженность 5000 rps



Высоконагруженность 5000 rps

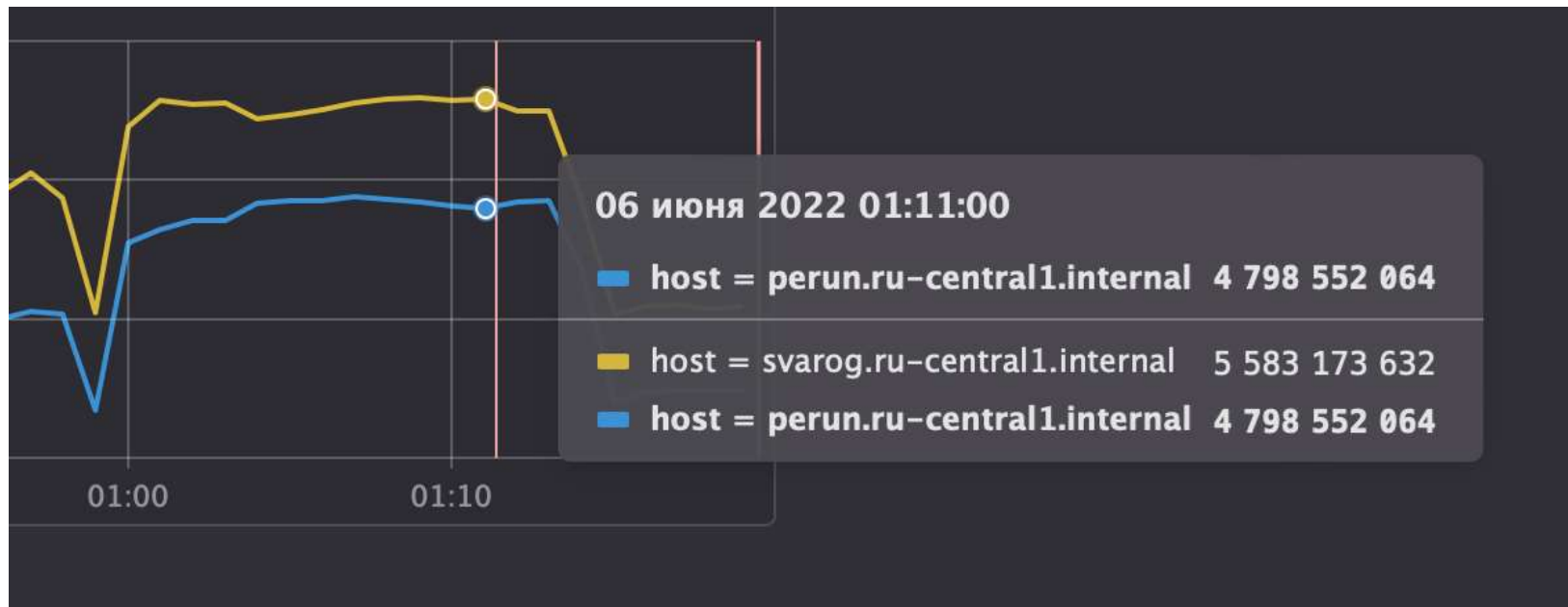


Высоконагруженность 5000 rps

Process List				
[8]	svarog-api	Mem: 257 MB	CPU: 29 %	onlin
[9]	svarog-api	Mem: 332 MB	CPU: 82 %	onlin
[10]	svarog-api	Mem: 268 MB	CPU: 81 %	onlin
[11]	svarog-api	Mem: 280 MB	CPU: 90 %	onlin
[12]	svarog-api	Mem: 280 MB	CPU: 97 %	onlin
[13]	svarog-api	Mem: 281 MB	CPU: 81 %	onlin
[14]	svarog-api	Mem: 315 MB	CPU: 89 %	onlin
[15]	svarog-api	Mem: 312 MB	CPU: 86 %	onlin
[26]	svarog-ui	Mem: 241 MB	CPU: 2 %	online
[27]	svarog-ui	Mem: 245 MB	CPU: 4 %	online
[28]	svarog-ui	Mem: 228 MB	CPU: 2 %	online
[29]	svarog-ui	Mem: 246 MB	CPU: 1 %	online
[30]	svarog-ui	Mem: 246 MB	CPU: 4 %	online
[31]	svarog-ui	Mem: 248 MB	CPU: 2 %	online
[32]	svarog-ui	Mem: 253 MB	CPU: 2 %	online
[33]	svarog-ui	Mem: 251 MB	CPU: 0 %	online

Process List				
[8]	svarog-api	Mem: 273 MB	CPU: 21 %	onlin
[9]	svarog-api	Mem: 334 MB	CPU: 83 %	onlin
[10]	svarog-api	Mem: 257 MB	CPU: 75 %	onlin
[11]	svarog-api	Mem: 277 MB	CPU: 92 %	onlin
[12]	svarog-api	Mem: 299 MB	CPU: 87 %	onlin
[13]	svarog-api	Mem: 275 MB	CPU: 83 %	onlin
[14]	svarog-api	Mem: 314 MB	CPU: 106 %	onli
[15]	svarog-api	Mem: 313 MB	CPU: 90 %	onlin
[26]	svarog-ui	Mem: 242 MB	CPU: 3 %	online
[27]	svarog-ui	Mem: 245 MB	CPU: 2 %	online
[28]	svarog-ui	Mem: 234 MB	CPU: 0 %	online
[29]	svarog-ui	Mem: 246 MB	CPU: 1 %	online
[30]	svarog-ui	Mem: 246 MB	CPU: 5 %	online
[31]	svarog-ui	Mem: 248 MB	CPU: 2 %	online
[32]	svarog-ui	Mem: 253 MB	CPU: 0 %	online
[33]	svarog-ui	Mem: 251 MB	CPU: 1 %	online

Высоконагруженность 5000 rps



Высоконагруженность 5000 rps



А можно ли не городить свои
костыли?

Можно! Давайте посмотрим на Nest.js

Nest.js

[Nest.js](https://nestjs.com/) - это фреймворк для создания эффективных, расширяемых серверных приложений.



Какие преимущества дает Nest.js

- готовую архитектуру;
- готовые решения для middleware;
- удобные механизмы валидации;
- возможность grpc почти из коробки;
- dependency injection
- java-подобное приложение.