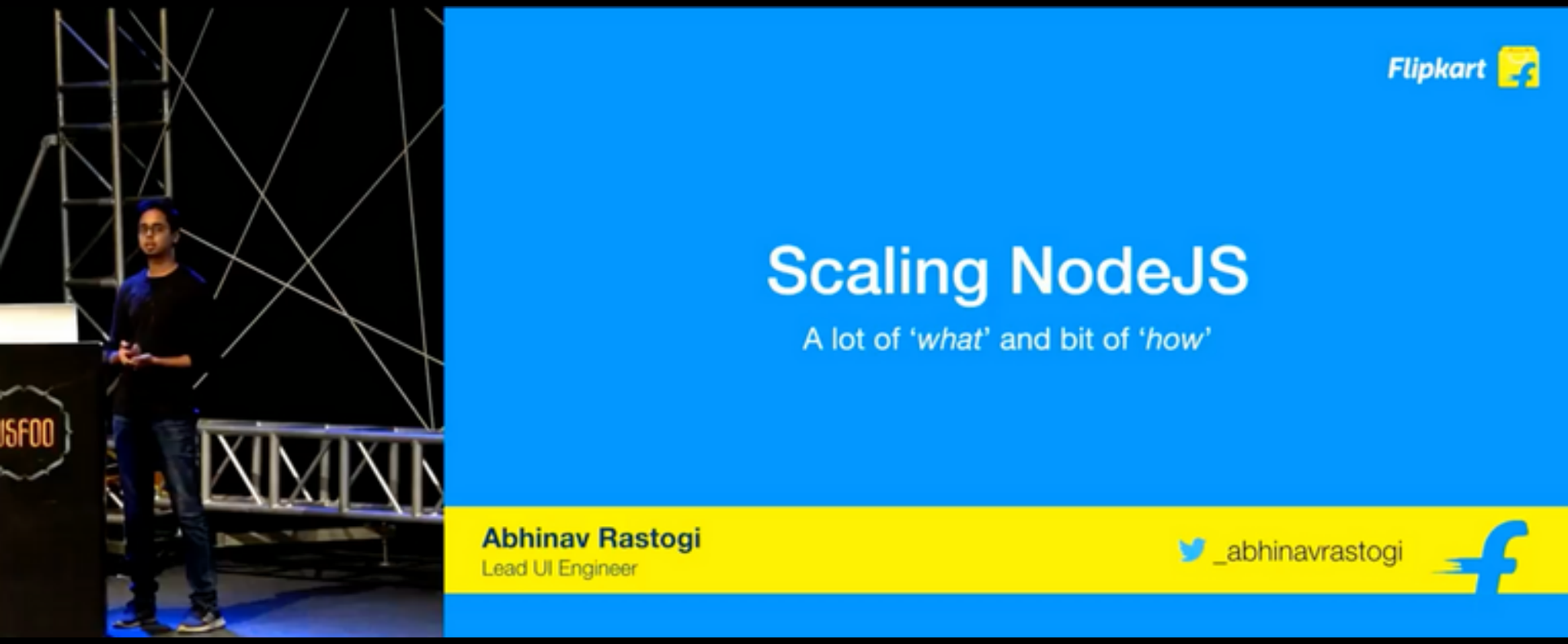


What issues flipkart faced while scaling node.js, what are different types of scaling, how can we make use of Google chrome debugger tools for memory profiling, what is the optimization cycle and how they configured their network, memory, disk, cpu resources, what is connection pooling, ulimit, network profiling, keep alive headers, ephemeral ports, how co-hosted nginx can be used for compression, what is pm2, network profiling and 0x.js ?

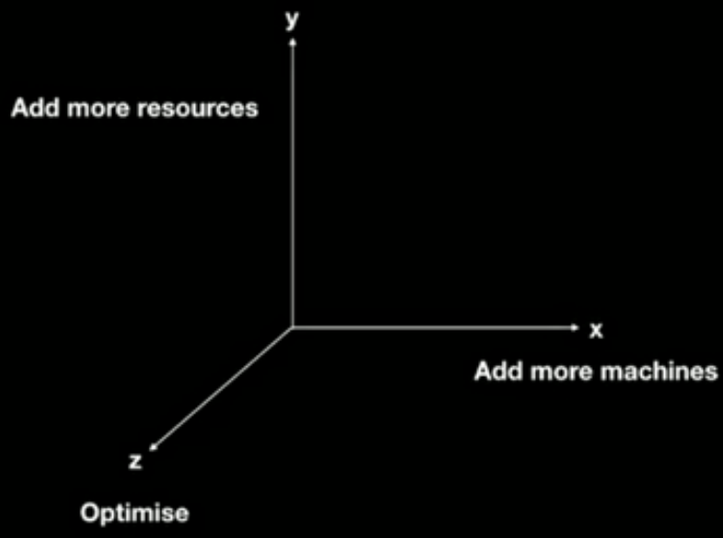
Abhinav manages mobile and web teams at flipkart. They started using Node at 2014.
The talk was more about what went wrong and the fixed they used.

"**Scalability** is the capability of a system to handle a growing amount of work, or its potential to be enlarged to accommodate that growth."

– *Andre Bondi*

Scalability definition is sort of incomplete above and it also includes that one needs to do that efficiently with loosing performance or degrading in any manner.

Types of Scalability



Horizontal scaling is on the x-axis which essentially means to add more number of machines to the cluster to handle more traffic. It definetely improves reliability.

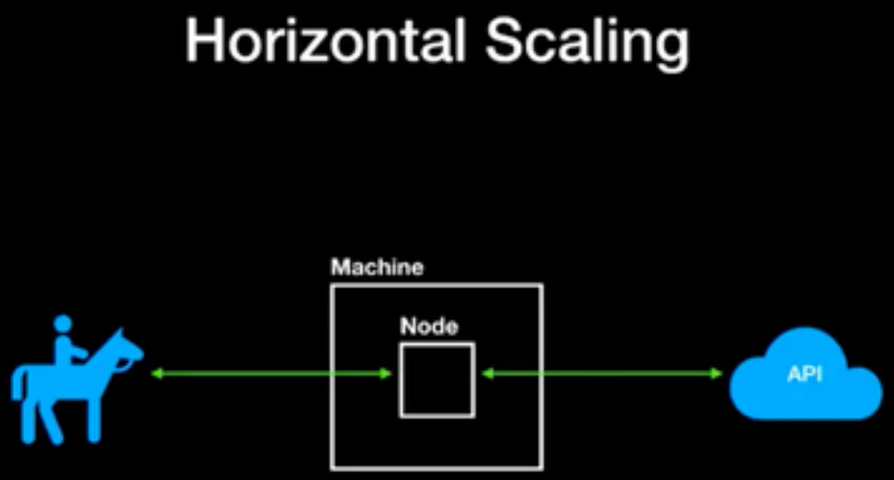
Then, there is vertical scaling on the y axis which is used to add more resources like RAM, CPU, Network, etc in the same machine.

The third one is debatable it's like data partitioning and things like map reduce to distribute the load better to break the data into pieces and run it on parallel. It can also be used to include code optimization which can be done at application or system layer to improve the performance.

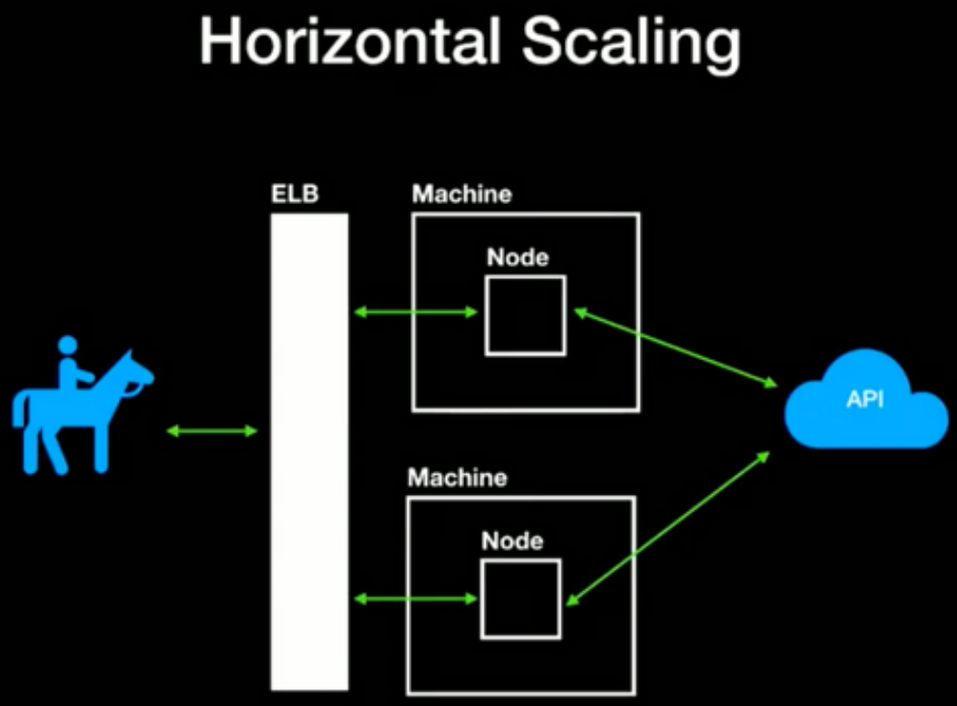
And figure out the bottlenecks and fix them so there will be not much need to add more machines or resources.

Horizontal scaling improves fault tolerance that if one machine goes down others can handle that load. It also provides high availability to

the systems about 99.9%



We've a user , it'll talk to the machine which will be running a node process which will talk to the API layer. We send the request from user to node through a machine layer , it goes to api, it comes back to node and goes to the user. We can scale horizontally by adding more machines.



And a load balancer in front of them and the load balancer based on some algorithm may be the least connection algorithm enable the machine with least number of connections get the desired request or round robin algorithm which essential works in the same way and we'll get reliability and high availability.

The drawback here is it's easy to add more machines , more the code on them, load balancer will take care of the rest but costly at the same time.

With things like heroku, GCP or AWS , it becomes more complex but efficient if we're able to do that dynamically. And users pay as they use.

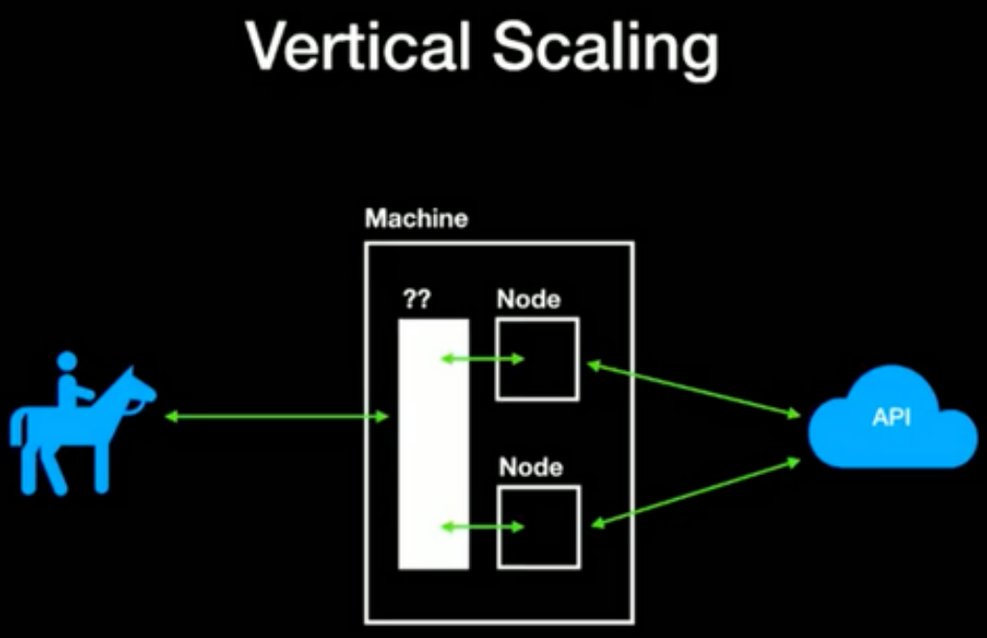
Essentially vertical scaling means adding more resources (CPU, RAM) to the machine.

Let's say we're running a node process by its implementation details using a single thread.

If there are 20 cores on the machines and we've processes which takes up lot of CPUs. 4 out of those 20 will go to 100 percent and other will remain to 0 that's how the OS and node handles it.

From the outside we've servers which can do all which can be done and it's bottleneck we can't scale up further either the response goes up or request input doesn't go.

The obvious solution is to add more node processes on the same machine.



In the above picture they've added a mystery layer which sort of looks like a load balancer that's what it sort of does. That layer distributes the requests. User and nodes talks to that layer.

There are two ways to do this:

- we can either run the server on multiple ports, let's say we've a server.js which takes a port from the environment and we put up 20 node processes on 20 machines where every process gets a different port.
- The other option is to fork child processes such that all listening on the same port and parent process is distributing the requests.
- cluster node module is essentially a multi core server manager.

npm install cluster

At the following pic, we can see we can identify whether the cluster is a master or a slave.

```
// cluster.js
const cluster = require('cluster');

if (cluster.isMaster) {
  for (let i=0; i < 20; i++) {
    cluster.fork();
  }
} else {
  require('./server.js');
}
```

If it is master we can fork any number of slave processes, essentially it creates a bunch of them. And those slave processes get the flag as false so we require server.js

We'll need to do routing and log management ourself over this.
To make it easier there's a tool built called pm2 on top of this.

npm install pm2

pm2 claims to be a advanced production process manager.

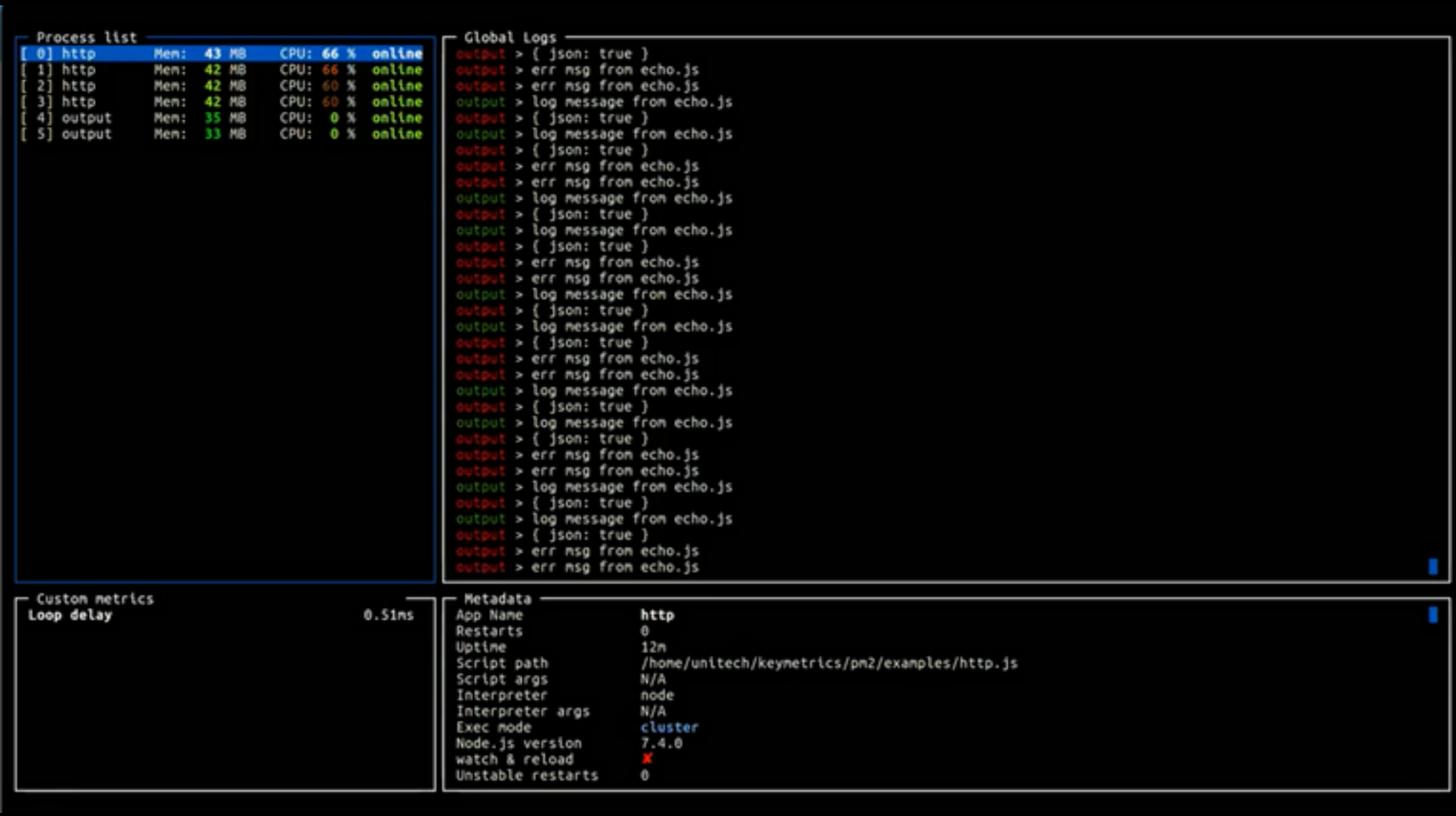
pm2 provides features like graceful reloading, crashes, auto restart and a bunch of monitoring tools.

```
{
  "name": “my-app",
  "script": "./app.js",
  "env": {
    "NODE_ENV": "production"
  },
  "instances": 0,
  "exec_mode": "cluster",
  "merge_logs": true,
  "error_file": “/var/log/my-app/error.log",
  "out_file": “/var/log/my-app/out.log",
  "log_date_format": "DD-MM-YYYY HH:mm:ss Z"
}
```

Above json file can be used to start the pm2 server.

Giving instances as 0 means take the count of CPUs and run equal number of processes. exec_mode has two modes : cluster and fork. It's able to merge logs and format it in custom way.

pm2 is very useful so apart from having easy node start up and handling scripts, it provides really great monitoring tools.



pm2 provides useful built in monitoring tools used for parameters like CPU usage, logs, interesting metadata and one can track the custom metrics aggregated across all the machines.

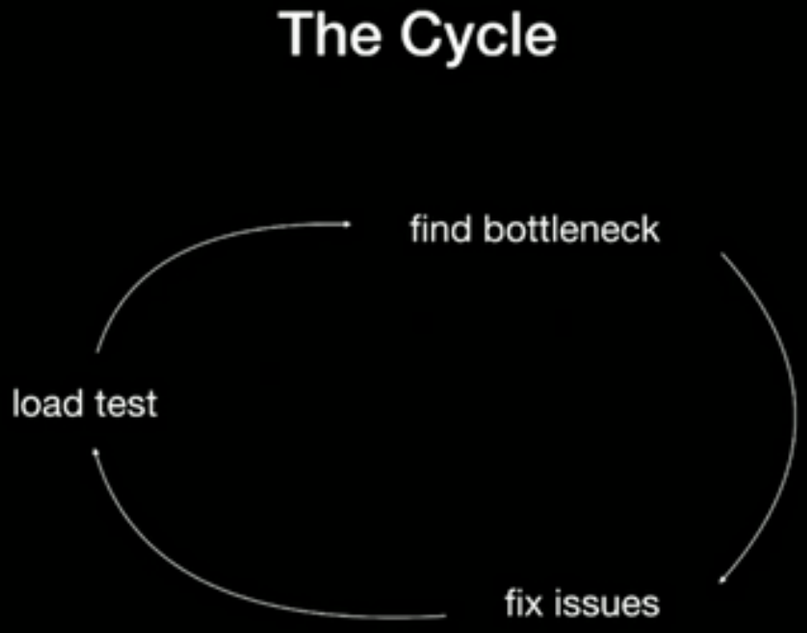
Flipkart uses it quite heavily.

Since they are using multiple node processes and code is running on multiple copies where they consume 100% of the resources (Network, CPU, Memory and Disk)

The Optimisation Cycle

When we code node, there's something else we need apart from writing the code.

Following picture depicts the optimization cycle of the node especially for production systems:



They run load test to generate artificial load on the system by simulating real users and making artificial api calls using any system they can stress starting from the backend and find the breaking points or the bottlenecks and fix the associated issues which are not always easy, some may be present at the OS level.

We should keep doing it to find the bottlenecks and fixing them.

If one machine is doing 10 QPS and we can make some fix to achieve 20 QPS and repeat this for better results.

network, disk, memory, cpu

They categorised the bottlenecks as application layer or system layer.

Application layer is essentially the code that we write, the bugs in the code whereas system layer works on the system we're using like centOS.

Network:

Bandwidth:

We might think that on a server why the bandwidth be a concern.

Servers have a very good data centre connections and a good bandwidth but if look at the entire system, request and response goes through multiple channels using very different network drops even before coming to the cloud provider.

And each of those links might've a different bandwidth, what happens if we're rendering a html page on server using any technology may be node but it can choke certain networks because HTML rendering as a text moves very fast like we see hot reloading in react. Let's take the follow image as reference:

1000 kb per page
100 qps per machine
100 machines

= 10gbps

As we can see we've 10gbps as output and this what the node cluster is outputting . This is one of the issue they ran into at flipkart, when they're serving HTML , at some point of time the load balancer itself got chopping , numbers were quite different from what shown in the above pic.

It might be some other server doing it whoever is handling the traffic.

The solution is pretty we should start compressing that HTML.

compression

It means that we should gzip the html and plain text gets great compression ratio's and two simple ways to do it is to used built in compression module with express.


```
var app = express()
app.use(compression)
```

And output is automatically be json but the problem here is it has a very high CPU cost.

Since node again is a single threaded, we end up with the main thread being blocked for a long amount of time.

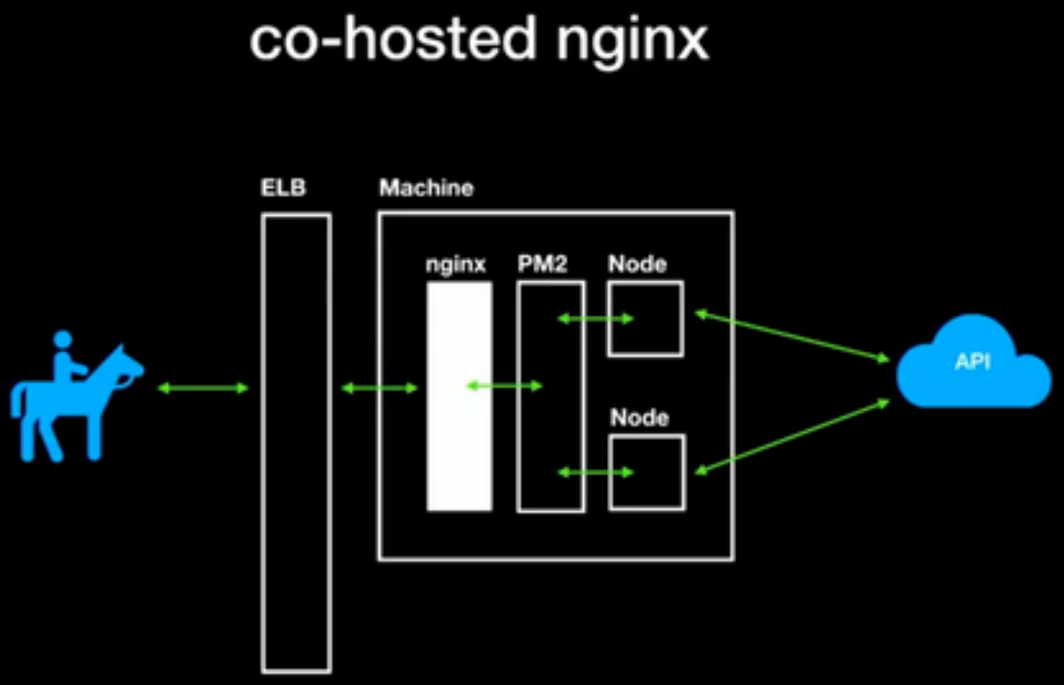
The main thread is blocked in compression and making sure it's not taking any more requests even though it's not actually rendering it.

The way they solved is with the help of co-hosted nginx or we can use any kind of reverse proxy for the compression.

What it does is if allows node to off load the HTML to another process living on the same box doesn't hit the network layer. It returns the HTML stream and asks nginx to gzip it.

Nginx is using a low level api, super efficient in compressing and hardly consumes any CPU at these scales.

We can also use nginx for multiple other uses like caching static files like we've service worker in the js file something like that.



They added the PM2 layer that was the question mark in the previous pic.

Node renders the HTML and goes to PM2 and the nginx is compressing it so the machine it will hit the network layer by already getting compressed.

Usually text gets around 10:1 compression that means 10gbps essentially becomes 1gbps and scales linearly with time. That's at system layer.

network profiling

The next thing is network profiling, we need to profile all the resources and we'll talk about each of them one by one.

We need to monitor all the n/w resources like ports , sockets and all the files we've and that's not just bandwidth.

There's a limit to how many upstream and downstream calls a machine is allowed to make. Following are the commands we can use to lookup that data.

netstat
lsof
watch

netstat gives the network statistics for the TCP connections we've.

lsof gives all the open files on the system. When er put load on the machines, we see that we run into some sort of bottlenecks that the machines just stops accepting connections like it can say connection timeout or some machine above the current layer can say service unavailable.

watch is a simple command which runs any command we give it and runs every 2s to provide the output.

Why it gets blocked, the reason is everything is a file in unix:

everything is a file in unix!

It's partially true, everything that we do in linux is represented by the streams of data.

And a stream of data is identified by a file descriptor in unix. It means that we've a unique id generated for every stream. It could be stdout, stdin, network, streams, ephemeral ports and sockets.

OS comes with a limit of how many file descriptors its allowed to make to prevent Dos kind of attacks. They use something called ulimit.

ulimit

The ulimit is the user limit command on all the machines which essentially tells the that users limit should be this. Output is shown below:

ulimit

```
core file size          (blocks, -c) 0
file size               (blocks, -f) unlimited
pending signals                 (-i) 32767
max locked memory       (kbytes, -l) 32
max memory size         (kbytes, -m) unlimited
open files                      (-n) 1024
max user processes              (-u) 50
```

The max no. of open files is 1024. This gets applied across all the machines. It means this will also cause issues with number of cores and sockets that can be opened.

max memory size that any process is allowed to consume and node processed by default allow 1GB of memory, if one process consume more than 1GB memory we'll get a kill signal in node.

That's limit we can set . FIle sizes also has limits.

They're using some c bindings , low level code from js using a library and those libraries run in a OS like in c. If somethings crashes, we debug the code using **core dump.**

Core dump is like entire stack trace kind of a thing that's the OS creates when the low level c is crash.

What they saw is **core dump** was not getting generated for them.

It was getting very hard to debug. And the reason was core file size is 0 which is the default for most production OS.

And when it is 0, OS doesn't generate a core dump. So while we're debugging we need to increases this number and we should disable when its not required otherwise it'll keep creating core dump files.

We can only increases the limits upto a certain extents. These are the system level issues.

We're still hitting the limits after increasing them. The next step is to optimize to get the application running.

One obvious thing to optimize the number of connections we're creating is to make use of keep-alive header.

keep-alive header

```
Object.assign(headers, {
    'compress': true,
    'Connection': 'keep-alive',
    'Keep-Alive': 'timeout=200'
});
```

This'll say that for 200s keep the connection alive that re uses connection for any new TCP requests and we won't head in the overhead of creating new ones and create far viewer connections.

connection pooling

We can use it to say that this is a pool of 30 or 100 connections. And the node process when it wants to make a request will pick one connection from this pool and use that. Once it's done, it'll close it and put that back.

If nothing is free, actual client request has to wait and that's actually better than letting a system to go 100% usage and then fail.

This artificial connection limit allows to use sockets on user's end and to this in node we need to pass custom http agent in fetch.

connection pooling

```
const http = require('http');

fetch(url, {
    agent: new http.Agent({
        keepAlive: true,
        maxSockets: 24
    })
});
```

It essentially allows to send a keep alive connection and the difference between this keep alive and the previous one is this is for the fetch calls and previous one is for the response header that node send to the client.

One side we're solving for incoming connections from the client and the other one we're solving for outgoing connections from node to api.

ephemeral ports

ephemeral ports are actually the short lived end points that are created by the OS when a program requests a available user port.

And usually this ephemeral (temporary) port is typically assigned a number between 1024 and 65535 that's us about 60k numbers to interact and since the OS is itself using and ports for using various things. It also includes sockets within the system. So we end up with 40-50k ports which user process can actually access.

tcp connection states

When we've a tcp connection it goes through multiple states.

A connection will go to the whole single phase where we send a request in return we'll get a data packet.

TCP goes through multiple states, the main one we look at is **established,** when either the client or server it goes to **closed** or **time wait** state.

These are very natural and that's the life cycle of a TCP connection. The default time wait is the time wait for tcp connection and default time out is 2mins i.e. 120s.

If there are 30 or 40 ports available and if there's a timeout of 120s , users will run out of ports on the machine if we get about 400 queries per sec.

It depends on the load machine can handle. 400 qps is a very small number for node if we're not doing heavy CPU task. We need to modify ulimt or do connection pooling to prevent getting the max limit.

disk

For disk, we need to implement auto update and don't let the logs fill up.

Keep deleting old logs and compress them, write disk I/O on the hot path. By hot path we mean not use app.use and not access disk in that.

we should it once and not for every request, it can be mostly and logging must not be sync .

They earlier used various synchronous libraries for logging, it actually blocks the main thread and write the logs on different port during compilation and comes back. Removing synchronous Ihelped them a lot

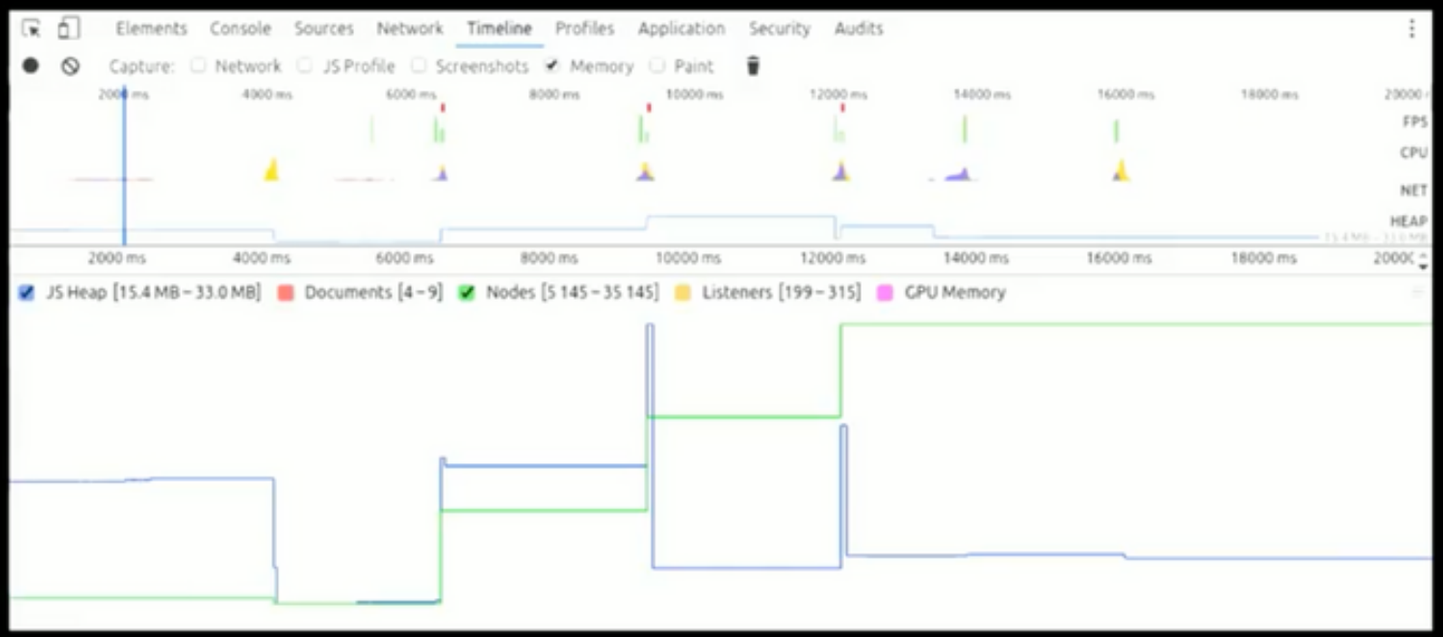
memory

Memory is essentially the RAM we've on the machine. For memory we need memory profiling:

memory profiling

Memory usage should become a straight line for a given qps. It shouldn't be going up or down ideally. Memory consumption should not go up at a same request rate.

memory profiling



There are many different tools, one of the best tools is chrome dev tools.

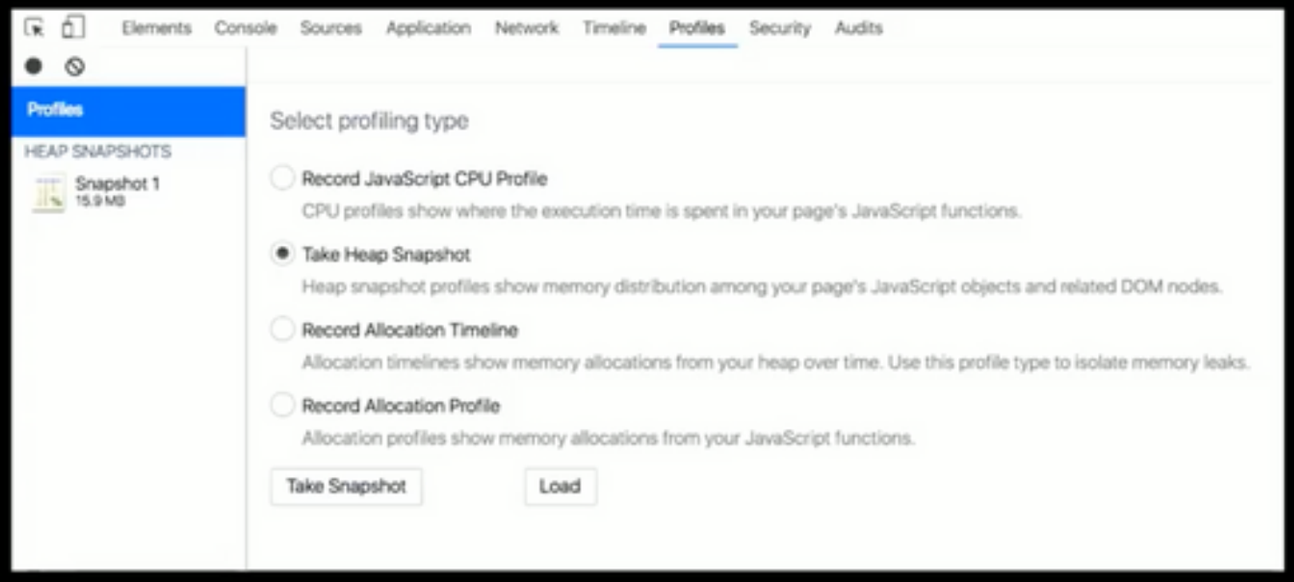
We can connect chrome devTools to node very easily.

As we can see we used timeline and enabled memory profiling below.

we get the timeline when the application loads, JS heap size is the humber of html nodes that we're creating and specially for memory, the blue line shown above should go straight or down.

Going up is when we create new objects and going down is essentially the garbage collection. If any variable goes out of scope, garbage collector will clear out the memory. It's costly and CPU usage can be high. We can see skipped frames whenever GC happens.

memory profiling



We can use snapshots for memory profiling. We can do heap snapshots or allocate timelines from devTools.

A heap is the memory distribution for heap memory essentially javascript memory object.

It tooks a snapshot and gives us the entire tree of how js heaps looks like. We can use that via snapshots.

We can take the snapshot before and after scrolling the page to compare via devTools . We can then identify that this object has been created this many times.

Usually that number is very large as it's getting created again and again or the GC is destroying them again and again.

```
const timeoutPromise = new Promise((resolve, reject) => {
    setTimeout(reject, 1000);
});

const actualPromise = fetch(...);

return Promise.race([actualPromise, timeoutPromise]);
```

Let's say if api call doesn't return in one second , we want to fail that request.

We would rather server user an error page rather than block the system.

Promise.race shown above resolves if actualPromise gets completed in 1s.

The problem here is when we're doing setTimeout inside promise, it doesn't get garbage collected.

The easy solution is to store the timeout outside and clear it once the race method triggers. This was hitting them very hard.

With CPU, we can do a very little system level operation.

cpu

They used it to schedule tasks at CPU and manage load.

Obvious thing is to be running in NODE_ENV production especially if we're using react as a server or lot of other libraries.

logging and error checking can be disabled at production to speed the things up. CPU profiling is pretty straight forward with node. We literally need to make use of --prof flag.

cpu profiling

This generates cross CPU profiling logs at high level. It will take snapshots on the basis of code memory addresses. If a function call is running in CPU, it will do it in a 100x seconds and if we reject a profile run for 10s. The output will give something as shown below:

```
[Summary]:
   ticks  total  nonlib   name
      79   0.2%    0.2%  JavaScript
   36703  97.2%   99.2%  C++
       7   0.0%    0.0%  GC
     767   2.0%          Shared libraries
     215   0.6%          Unaccounted
```

The above pic is the output for following code

```
import crypto from 'crypto';

app.get('/auth', (req, res) => {
    const hash = crypto.pbkdf2Sync(password, users[username].salt, 100, 512);

    if (users[username].hash.toString() === hash.toString()) {
        res.sendStatus(200);
    } else {
        res.sendStatus(401);
    }
});
```

We can check what c++ is doing using following picture:

```
[C++]:
   ticks   total   nonlib    name
   19557   51.8%    52.9%   node::crypto::PBKDF2(v8)
    4510   11.9%    12.2%   _sha1_block_data_order
    3165    8.4%     8.6%   _malloc_zone_malloc
```

crypto method is taking up 97% of the CPU if we replace this with a better library this code becomes entirely free. Rest of the script is not doing much and this is how we profile the CPU.

```
[Summary]:
   ticks   total   nonlib    name
      79    0.2%     0.2%   JavaScript
   36703   97.2%    99.2%   C++
       7    0.0%     0.0%   GC
     767    2.0%            Shared libraries
     215    0.6%            Unaccounted
```

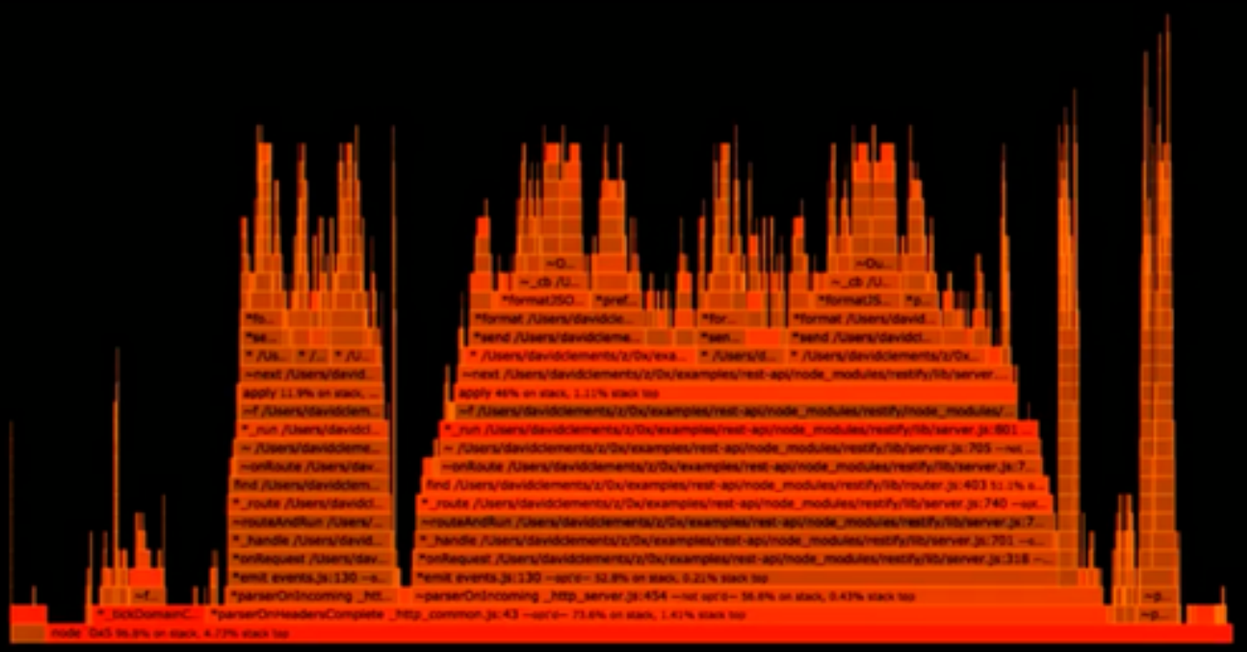
There's a better way of doing this with 0x. It's a library which essentially gives us a one line profiling for node.

0x

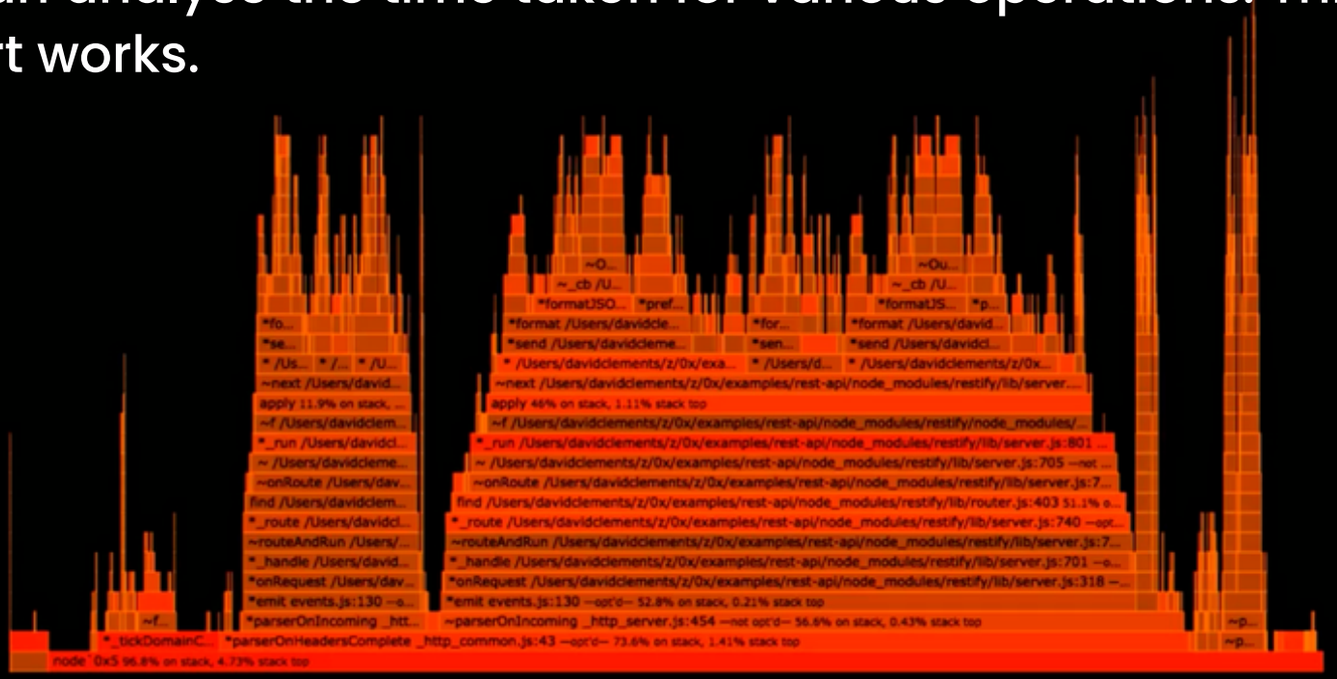
We can install 0x and use it instead of node. It gives output as a chart.

```
$ npm install -g 0x
$ 0x app.js
```

It looks very fancy but it's essentially the same thing it provides a better graphical representation.



On the x-axis we've the amount of time spent on the CPU and on the y-axis we've the call stack. As we see node as a process is running for 100 percent of the time at x-axis . The function which is taking lot of time (flat top is very bad) and not executing much functions is blocking the CPU. We can analyse the time taken for various operations. This is how a flame chart works.



They found that react library is taking 60% of their CPU time. They made use of memoizatio and 2nd highest things was JSON.parse and JSON.stringify. We need to have real time monitoring in the system.

real-time monitoring

Essentially just doing load test is not enough we need to look at real time production systems. We need to run artificial load test on the machines.

load, profile, fix, repeat!