Case Study: Node JS On Kubernetes Transition From Spot.io

Naor Tedgi (Abu Emma)

https://github.com/ntedgi

https://github.com/ntedgi/infra-meetings

Let’s get started

- Container Refactor
- Graceful Shutdown
- Why You Don't need Cluster Mode in K8s
- Scheduled Tasks
- Questions

Container Refactor

In Kubernetes, we aim to utilize the scale-out deployment mechanism to enhance our system's scalability and performance

The deployment must be executed rapidly and with minimal startup time.

Container Refactor

In our current EC2 setup, we pull images from Amazon ECR that contain essential development tools, such as Python and g++.

Node.js relies on C++, Node-gyp, and Python to compile C++ add-ons.

Examples of such add-ons include:

- node-rdkafka: A high-performance Kafka client.
- bunyan-syslog: Streams logs to a syslog server.

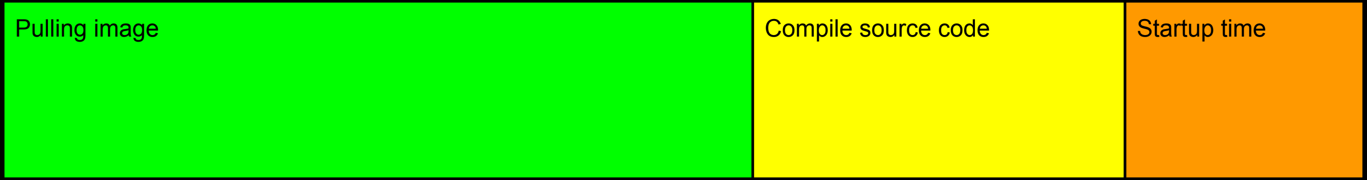
```dockerfile
FROM node:18.20.2-alpine3.18

ENV PYTHONUNBUFFERED=1

WORKDIR /usr/local/platform-js

ENV NODE_PATH /usr/local/platform-js
ENV NODE_CONFIG_DIR /usr/local/platform-js/config
ENV NODE_CONFIG_DEFAULTS_DIR /usr/local/platform-js/config-defaults

RUN apk add --update --no-cache python3 && ln -sf python3 /usr/bin/python \
    && apk add --no-cache build-base libsasl libssl1.1 openssl-dev cyrus-sasl-dev make g++ bash \
    && python3 -m ensurepip \
    && pip3 install --no-cache --upgrade pip setuptools\
    && npm install -g bunyan forever \
    && echo 'alias ll="ls -lah"' >> ~/.bashrc \
    && echo 'alias logs="tail -f /usr/local/platform-js/logs/*_general.log | bunyan -o short"' >> ~/.bashrc \
    && echo 'alias accessLogs="tail -f /usr/local/platform-js/logs/*_access.log | bunyan -o short"' >> ~/.bashrc \
    && echo "export PS1='\\w$ '" >> ~/.bashrc \

COPY . /usr/local/platform-js/

RUN  rm -rf node_modules \
    && rm -f config/local.json \
    && npm ci --quiet --no-progress \
    && npm run build

EXPOSE 3000

CMD ["npm", "start"]
```

Install Development Dependencies

Compile The source Code

Upon EC2 startup, we first compile the code and then proceed to run the compiled application

```bash
#!/usr/bin/env bash
npm run build;
forever dist/app.js dist/apps/$1
```



Container Refactor

Issues with this Approach:

1. **Size of image (Disk size) for single pod / server in disk is contains a lot of OS utils and development dependencies that we don't need at runtime (do you need mocha or g++ on prod?)**
2. **The application startup time is slow because we need to first compile the code before running**

Container Refactor

Container Refactor

To Run node app

1. Node engine installd
2. dist folder with js and add ons compiled
3. node-modules (only prod!)
4. Package.json (?)

```dockerfile
FROM 032106861074.dkr.ecr.eu-west-1.amazonaws.com/platformjs:base-18.20.2-alpine3.18-v1 as builder-dev-dependencies

WORKDIR /usr/local/platform-js
COPY . /usr/local/platform-js/

RUN rm -rf node_modules \
    && npm ci --quiet --no-progress \
    && npm run build \
```

Build and compile using development deps

```dockerfile
FROM 032106861074.dkr.ecr.eu-west-1.amazonaws.com/platformjs:base-18.20.2-alpine3.18-v1 as builder-prod-dependencies

WORKDIR /usr/local/platform-js
COPY . /usr/local/platform-js/

RUN rm -rf node_modules \
    && npm install --omit=dev
```

install only production deps

```dockerfile
FROM node:18-alpine
WORKDIR /usr/local/platform-js
ENV NODE_PATH /usr/local/platform-js
ENV NODE_CONFIG_DIR /usr/local/platform-js/config
ENV NODE_CONFIG_DEFAULTS_DIR /usr/local/platform-js/config-defaults
RUN apk add bash && \
    npm install -g bunyan pm2 \
    && echo 'alias ll="ls -lah"' >> ~/.bashrc \
    && echo 'alias logs="tail -f /usr/local/platform-js/logs/*_general.log | bunyan -o short"' >> ~/.bashrc \
    && echo 'alias accessLogs="tail -f /usr/local/platform-js/logs/*_access.log | bunyan -o short"' >> ~/.bashrc \
    && echo "export PS1='\\w$ '" >> ~/.bashrc

COPY --from=builder-prod-dependencies /usr/local/platform-js/node_modules ./node_modules
COPY --from=builder-prod-dependencies /usr/local/platform-js/config ./config
COPY --from=builder-prod-dependencies /usr/local/platform-js/package.json  /usr/local/platform-js/
COPY --from=builder-dev-dependencies /usr/local/platform-js/dist /usr/local/platform-js/dist
EXPOSE 3000

CMD ["npm", "start"]
```

Bonus we embed a patch management for nodeJS Debian on each push

copy production dependicies from step 2

copy compiled code from step 1

Container Refactor

We improve it by changing to different approach :

1. We move all our OS development Utils to different dockerfiles to reduce the time of installation when building the image and save it to ECR. Call it **base-18.20-alpine**
2. We use multi stage docker builder and from each steps only took the necessary output for runtime
3. We move this steps to new Docker file and save the container to private ECR platformjs:base-18.20.2-alpine3.18
4. Then with this base image we divide the process to 3 steps :
   a. **Intermediate docker 1** - Install all dependencies and compile the code into **dist** folder
   b. **Intermediate docker 2 -** Install only production needed dependencies
   c. **Intermediate docker 3 -** Copy the compiled code form step 1 , copy production dependencies from steps 2 and run

Container Refactor

Security Patch Management Automatically :

- **Run Time Layer (Docker Container)**
    1. Linux Distribution (OS)
    2. Runtime Node Engine Version
    3. Building Tools (GCC , Python , OpenSSL)

- **Application Layer (node dependencies)**
    1. Packages Update with **semver** Compatible

- **Provision Layer - (Spot EC2 instances OS,   K8s Machines)**
    1. Host Linux Distribution (OS)

Container Refactor

Security Patch Management Automatically :

```
FROM node:18-alpine
WORKDIR /usr/local/platform-js
ENV NODE_PATH /usr/local/platform-js
ENV NODE_CONFIG_DIR /usr/local/platform-js/config
```

Bonus we embed a patch management for nodeJS Debian on each push

Container Refactor

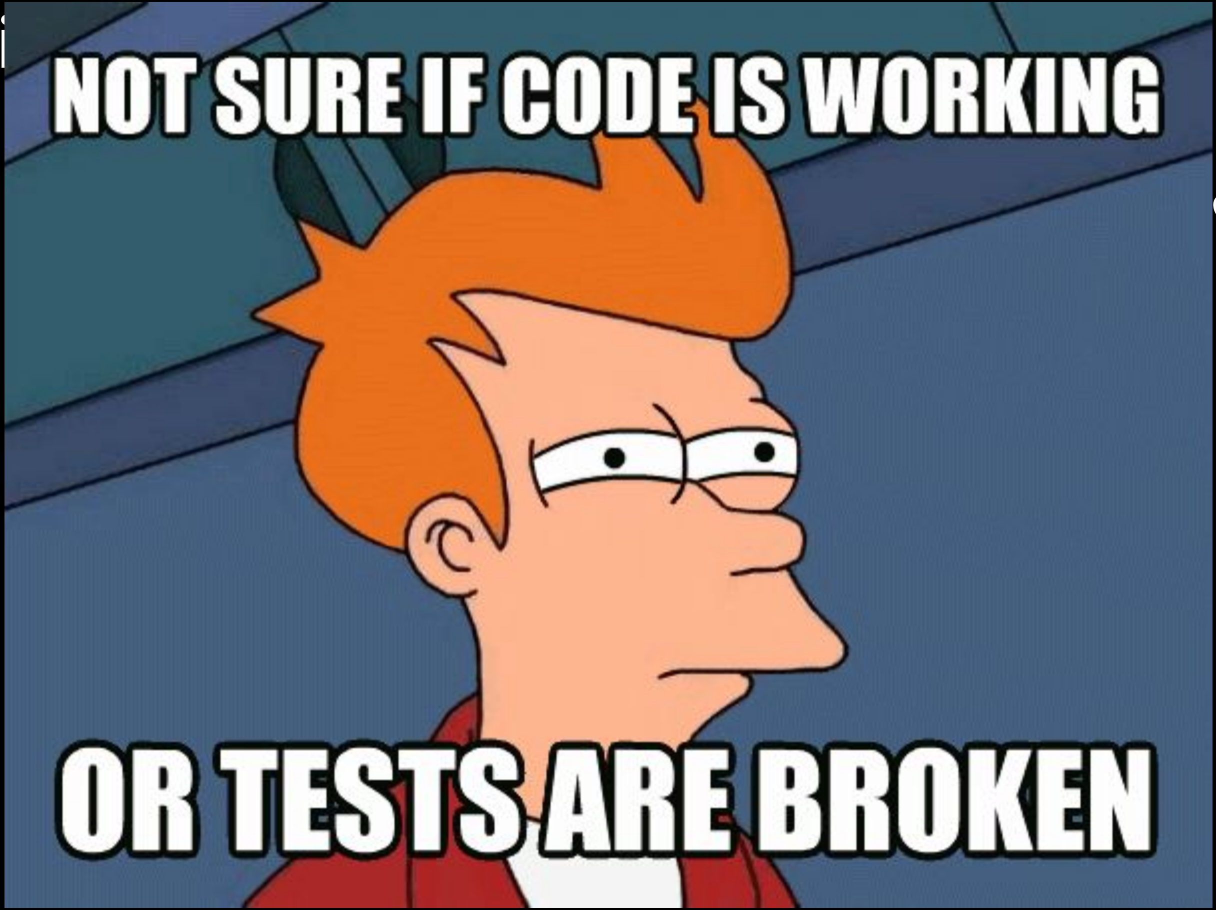
Security Patch Management Automatically

You have a problem!

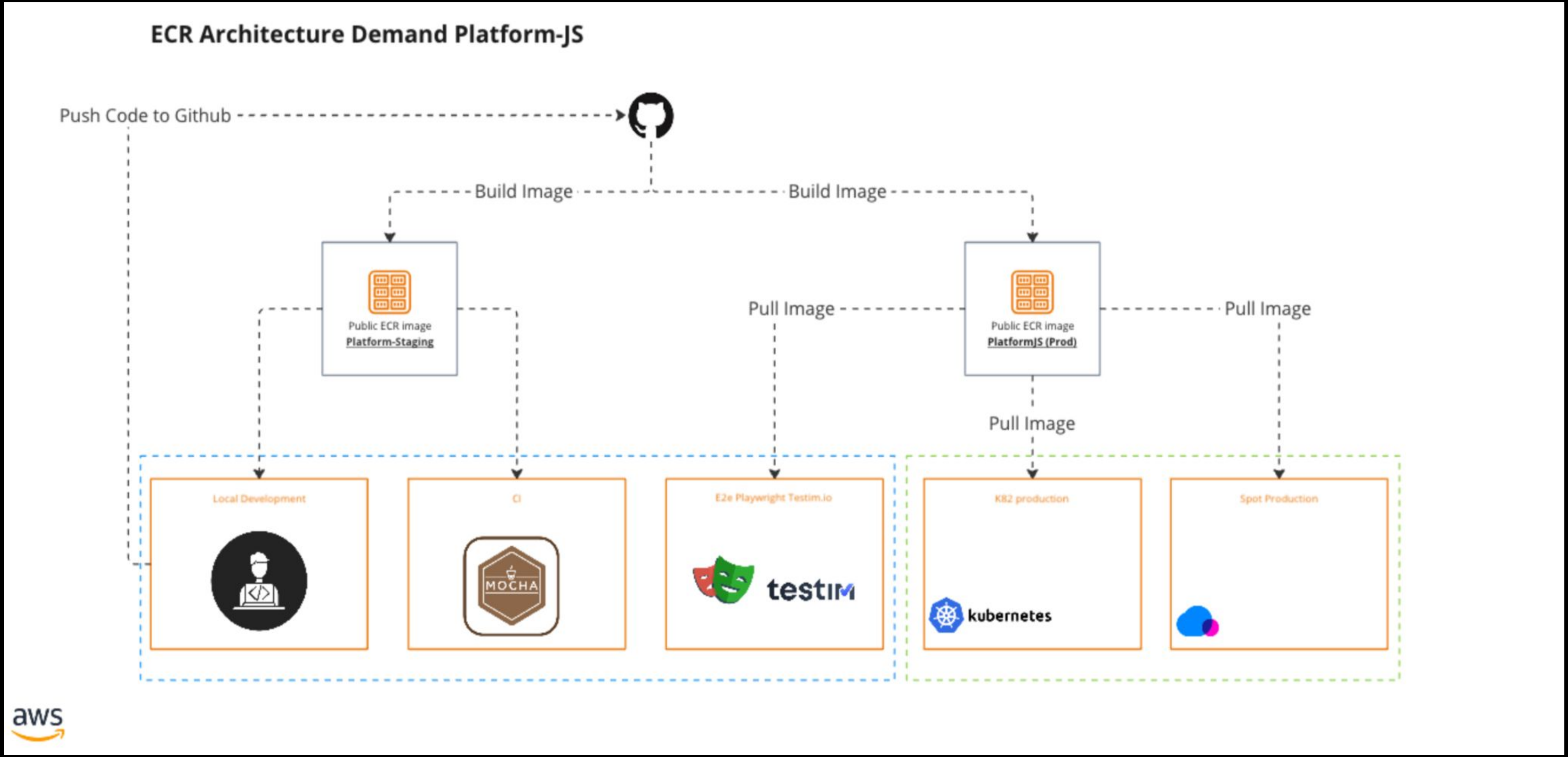
Contai

eploy !!!!!

You have



Container Refactor



Container Refactor

Security Patch Management Automatically :

- **Run Time Layer (Docker Container)**
  1. Linux Distribution (OS)
  2. Runtime Node Engine Version
  3. Building Tools (GCC , Python , OpenSSL)

```
FROM node:18-alpine
WORKDIR /usr/local/platform-js
ENV NODE_PATH /usr/local/platform-js
ENV NODE_CONFIG_DIR /usr/local/platform-js/config
ENV NODE_CONFIG_DEFAULTS_DIR /usr/local/platform-js/config-defaults
```

Bonus we embed a patch management for nodeJS Debian on each push

Container Refactor - Conclusions

Benefits:

1. **Reduced Image Size**:
   - The container images have been reduced from 705 MB to 284.22 MB, achieving an approximate reduction of 59.85%.

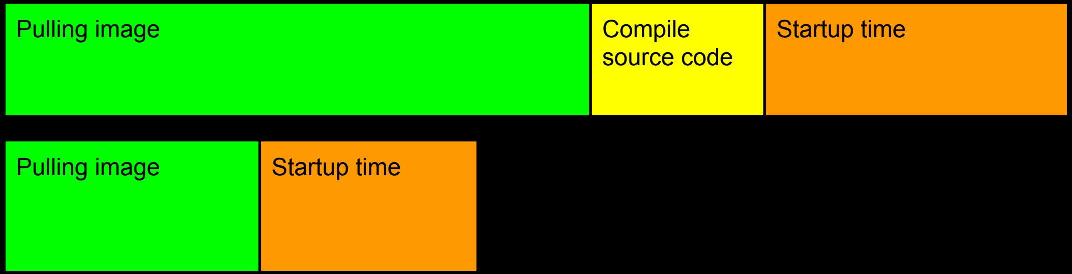
Image tag	Artifact type	Pushed at	Size (MB)	Image URI	Digest
master	Image	July 23, 2024, 16:39:31 (UTC+03)	284.22	Copy URI	sha256:51d578108ea8f2...

Image tag	Artifact type	Pushed at	Size (MB)	Image URI	Digest
9970530584, latest	Image	July 17, 2024, 11:22:00 (UTC+03)	705.94	Copy URI	sha256:efd139c67c3ba97...

Container Refactor - Conclusions

Benefits:

1. **Decreased Pull Time**:
   - The time to pull images from Amazon Elastic Container Registry (ECR) has been reduced from 25 seconds to 11 seconds.
2. **Eliminated Compilation Time**:
   - There is no longer a need for compilation during the container build process.
3. **Enhanced Security**:
   - Automatic patch management is now in place, ensuring that the containers remain up-to-date with the latest security patches.



Container Refactor - Conclusions

Eliminated Compilation Time:

There is no longer a need for compilation during the container build process.

Now you can say compile source code? It's very fast

```
> platform-js@1.2.1 build
> swc src --out-dir dist --copy-files
Successfully compiled: 1270 files, copied 39 files with swc (167.98ms)
```

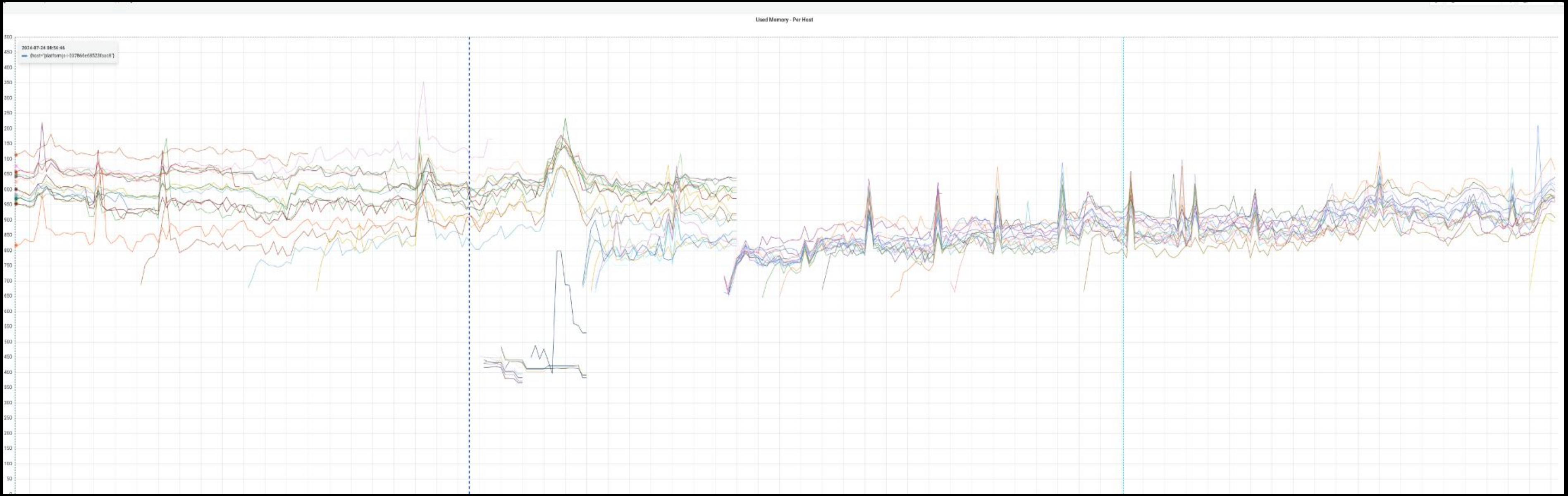
You're right but in order to do it in production you need to ship your container with all development dependencies!

Container Refactor - Conclusions

```
3.6M    moment-timezone
3.9M    luxon
4.8M    es-abstract
4.9M    lodash
5.2M    moment
6.6M    is-typed-array
6.6M    which-typed-array
12.9M   @adyen
64.7M   typescript
82.6M   aws-sdk
111.1M  geoip-lite
120.3M  @bufbuild
175.2M  node-rdkafka
/usr/local/platform-js/node_modules$
```

We found out there is some dependencies that we use there deps are huge for example
Geoip-lite - we use only to get the ip address of request is **111 mb !!**
We can reduce it to **10KB** !!! by using https://github.com/pbojinov/request-ip

Container Refactor - Conclusions



Let's get started

- Container Refactor
- Graceful Shutdown
- Why You Don't need Cluster Mode in K8s
- Scheduled Tasks
- Questions

Graceful Shot Down

As mentioned on the first page, we aim to leverage the horizontal scaling capabilities of Kubernetes. Instead of scaling up, we will focus on scaling out by utilizing machines with lower resources. This approach, however, will result in a higher number of instances being started and shut down frequently.

Graceful Shot Down

The consequences of an abrupt shutdown can range from minor inconveniences to significant data loss, and degraded user experience.

Graceful Shot Down

Prevent data loss: If a service is shut down abruptly, any in-progress transactions or requests may be lost, leading to data corruption or data loss. A graceful shutdown ensures that all data is saved and any pending requests are processed before shutting down.

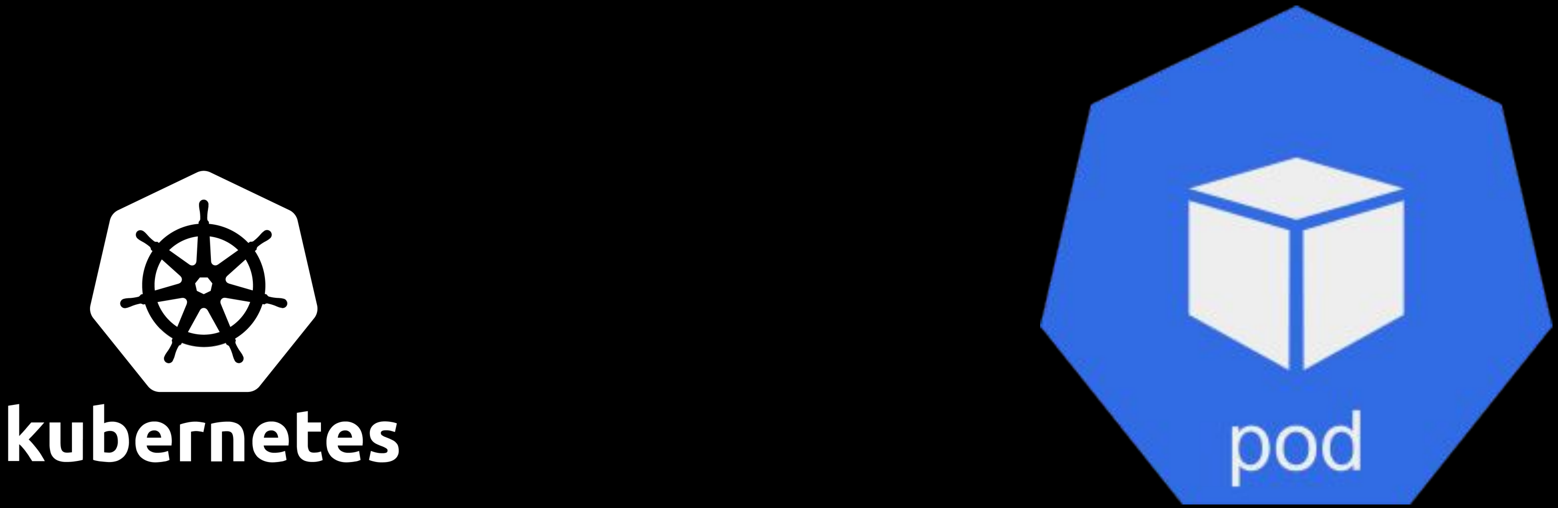
Avoid cascading failures: When a service goes offline, it can trigger a cascade of failures in other services that depend on it. A graceful shutdown allows dependent services to prepare for the outage and gracefully handle the failure.

Reduce downtime: By shutting down in a controlled manner, you can minimize the amount of downtime required for maintenance or updates. This helps keep the system up and running and reduces the impact on users.

Clean up resources: A service may be using resources such as file handles, database connections, or network sockets. A graceful shutdown allows the service to release these resources in a controlled manner, reducing the risk of resource leaks or conflicts with other services.

Graceful Shot Down

Naive Solution



Graceful Shot Down

Naive Solution

Send Kill Signal

kubernetes

Graceful Shot Down

Naive Solution

<u>Stop Incoming Traffic</u>

kubernetes

Graceful Shot Down

Naive Solution

set ***<u>terminationGracePeriodSeconds</u>***
to magic number (1m)

kubernetes

Graceful Shot Down

Naive Solution

Kill Pod

kubernetes

Graceful Shot Down

Probably going to work

Graceful Shot Down

Probably going to work for 90% of use cases

Graceful Shot Down

Probably going to work for 90% of the cases

Issues with this approach.

1. Maybe 1m is not enough? To finish all running request
2. What about Kafka/sqs Consumers ?

Graceful Shot Down

After the application gets kill signal K8s stop sending request to that POD

In our application, we have implemented signal handling for SIGKILL, SIGINT, and SIGHUP signals. Upon receiving any of these signals, we monitor all active connections and ensure they are completed. After all active requests are finished, we proceed to close all connections to databases and message queues. Finally, we gracefully shut down our application, exiting with code 0.

Graceful Shot Down

Count actively running requests

```
const express = require('express');
const app = express();
let activeRequests = 0;

// Middleware to increment the counter
app.use((req, res, next) => {
    activeRequests++;
    res.on('finish', () => {
        activeRequests--;
    });
    next();
});

// Example route
app.get('/', (req, res) => {
    res.send('Hello, World!');
});

// Endpoint to check the number of currently processing requests
app.get('/active-requests', (req, res) => {
    res.json({ activeRequests });
});

const PORT = process.env.PORT || 3000;
app.listen(PORT, () => {
    console.log(`Server is running on port ${PORT}`);
});
```

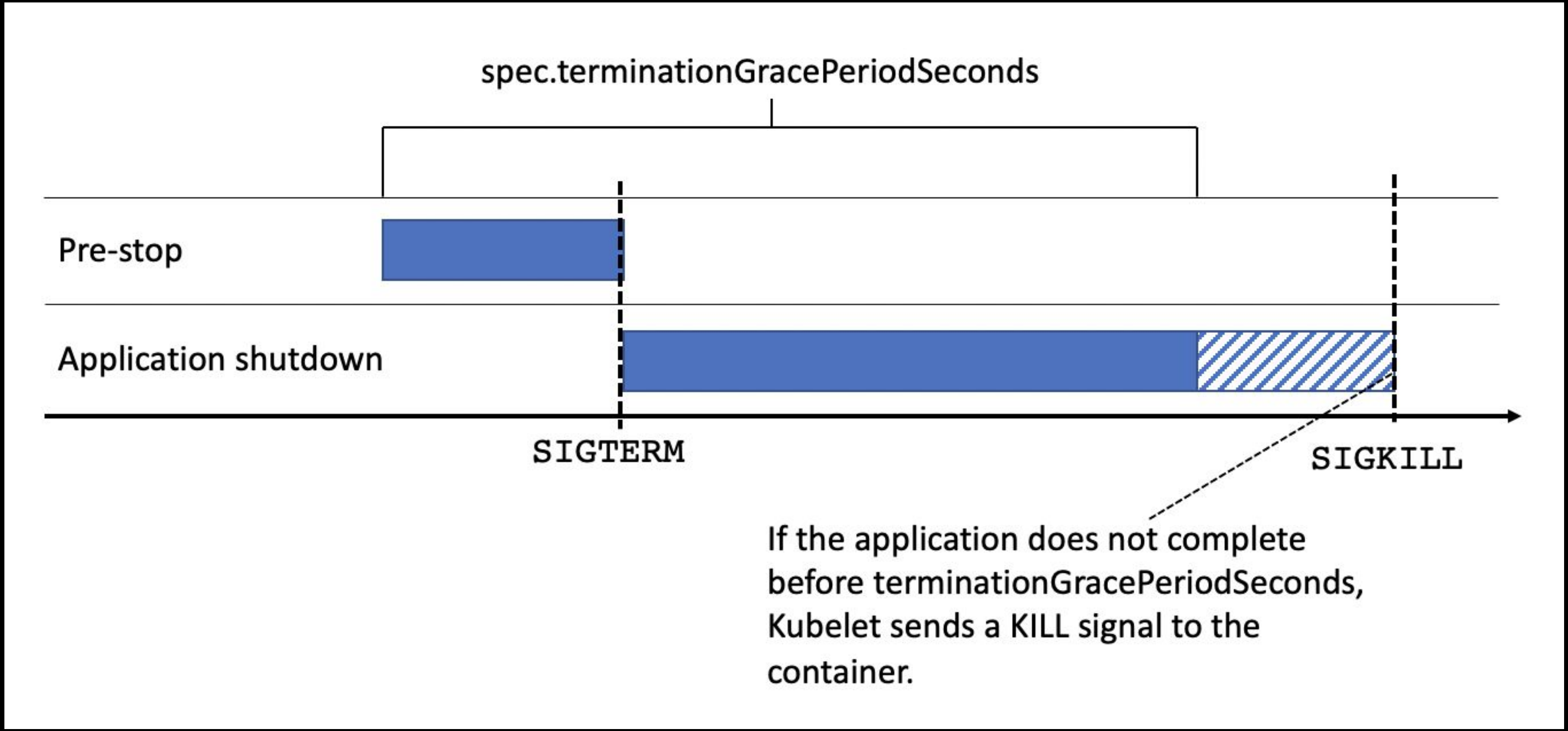
Graceful Shot Down

Set ***PreStop*** Hook

hooks are not executed asynchronously from the signal to stop the Container; the hook must complete its execution before the TERM signal can be sent.

https://kubernetes.io/docs/concepts/containers/container-lifecycle-hooks/

Graceful Shot Down



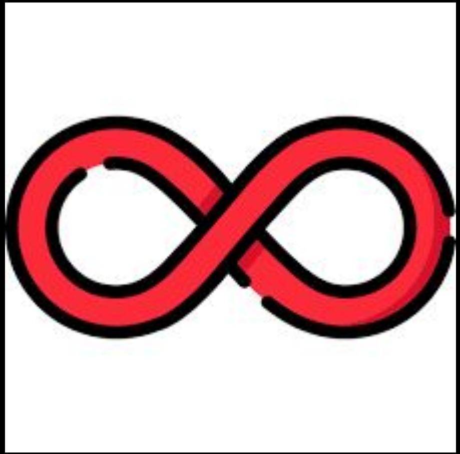
Graceful Shot Down

In ***PreStop*** *Script*

1. No incoming traffic arriving now
2. You can now wait for all running active request to finish (sleep)
3. Close Kafka Consumer
4. Disconnect from Mysql , Redis …
5. EXIT (0)
6. Then Let K8s killing the pod

Graceful Shot Down

Because we have this orchestration , our deploy is much faster
and a server can day safely (in peace)
We decide to stop using process manager tool like **forever** , PM2



Let’s get started

- Container Refactor
- Graceful Shutdown
- Why You Don't need Cluster Mode in K8s
- Scheduled Tasks
- Questions

Why You Don't need Cluster Mode in K8s

There is all kind of server models for example

Thread-Per-Request

- In this model, a new thread is created for each request received by the server.

Languages/Frameworks:

- **Java (Servlets)**: Traditional Java web applications using Servlets often used a thread-per-request model.
- **Python (WSGI frameworks like Flask, Django)**: When deployed with traditional WSGI servers, this model is sometimes used.
- **C++**: Can implement thread-per-request with frameworks like Apache HTTP Server (with worker MPM).

Why You Don't need Cluster Mode in K8s

There is all kind of server models for example

1. Thread-Per-Connection

- In this model, a new thread is created for each connection to the server, which then handles multiple requests over the same connection.

Languages/Frameworks:

- **Java (Blocking I/O servers)**: Older Java web servers, like Tomcat in its older configurations.
- **C/C++**: Can be implemented in servers like Apache HTTP Server or Nginx (with specific modules).
- **Go**: While Go tends to favor goroutines for concurrent connections, the pattern can be built with explicit threads.
- **Python**: Servers like Twisted or Tornado could use this model for handling long-running connections.

Why You Don't need Cluster Mode in K8s

Why You Don't need Cluster Mode in K8s

Why to use it in general?

How Many Threads Node uses By Default?

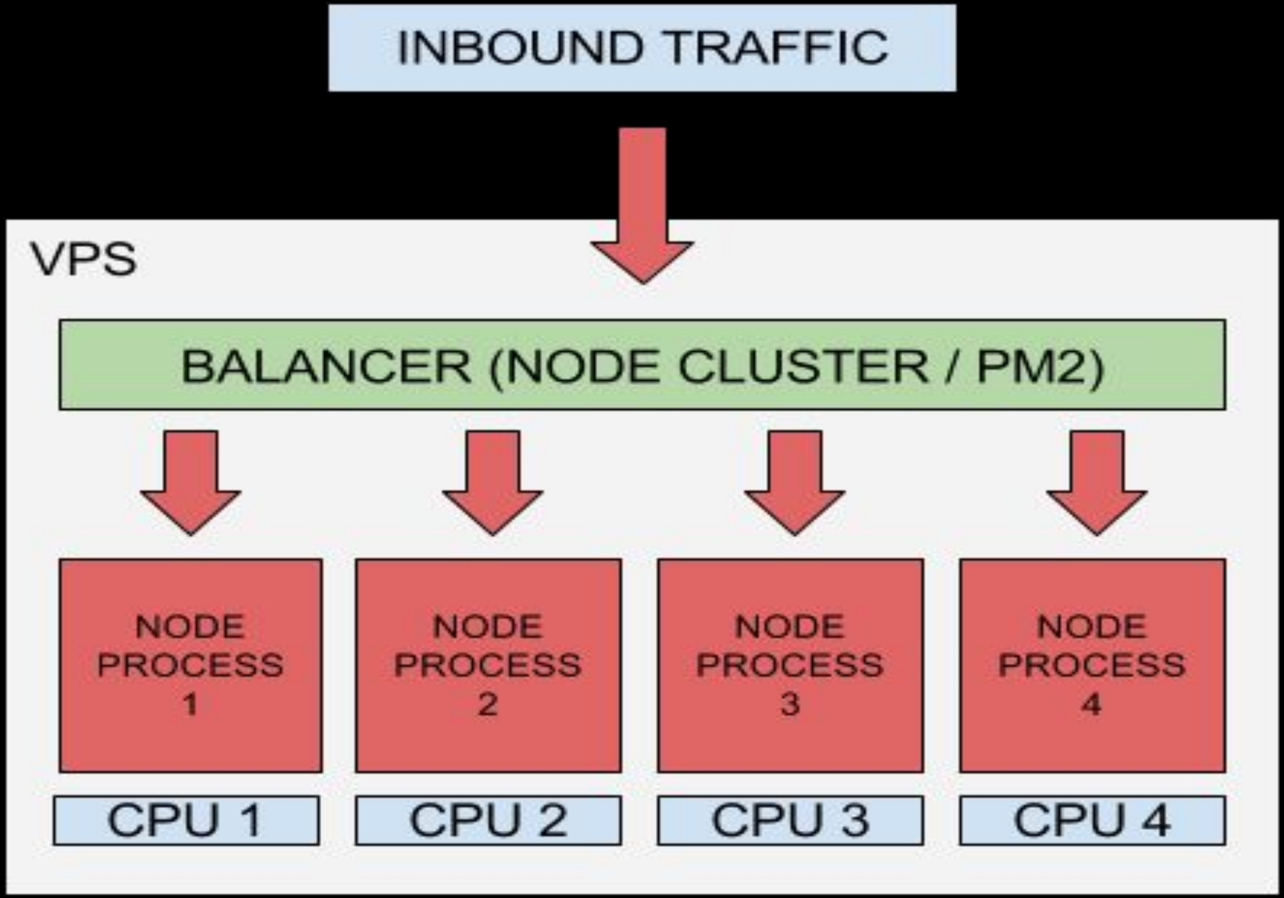
Why You Don't need Cluster Mode in K8s

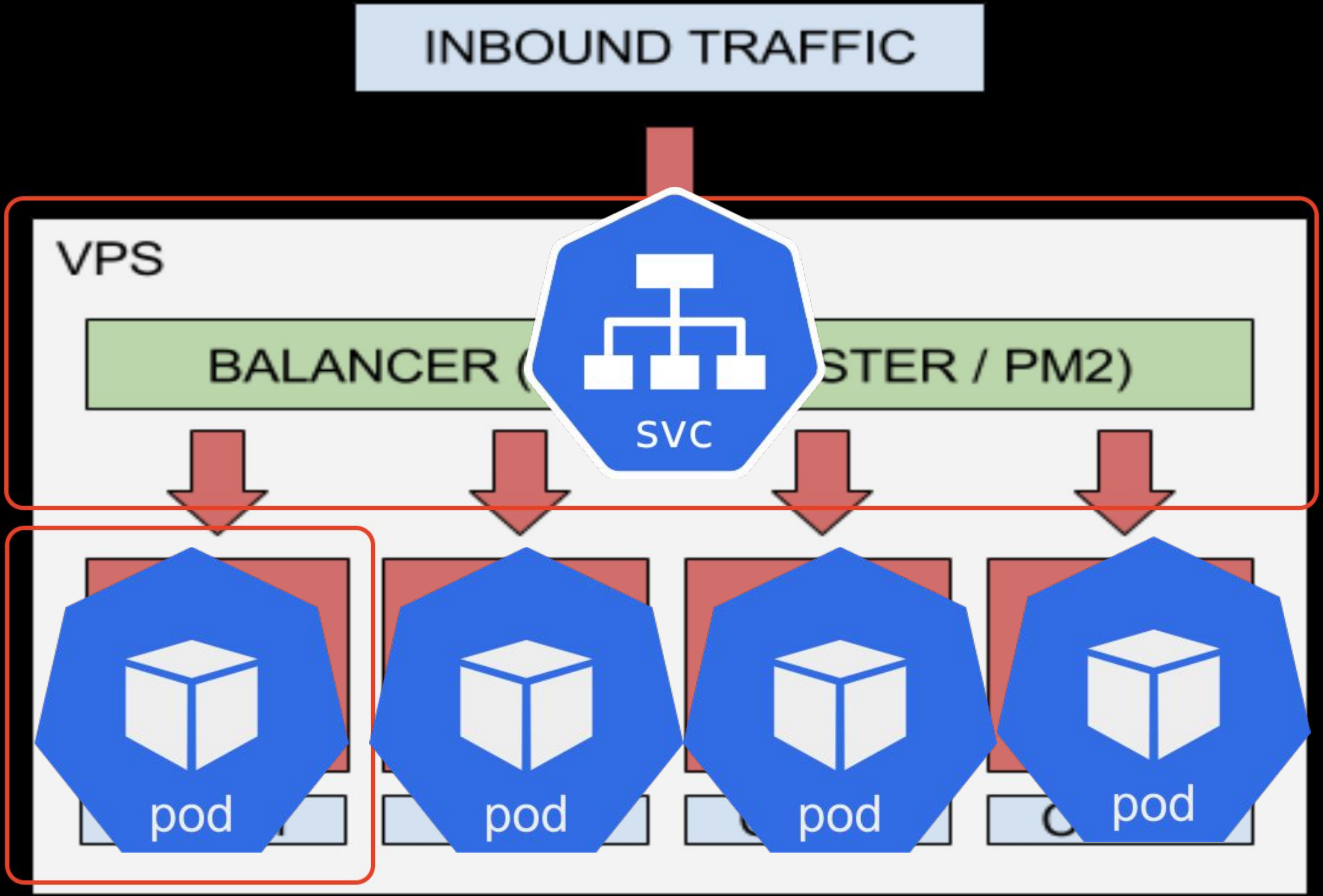
Why to use it in general?

- Code Interpreter
- GC
- Event Loop
- 4 Libuv threads for async tasks (net, fs ,dnslookup ..)
  Note: UV_THREADPOOL_SIZE

Why You Don't need Cluster Mode in K8s

Single-Threaded Limitation: Node.js can’t utilize multiple CPU cores in a single process. Therefore, vertical scaling by adding more CPU cores may not always improve performance unless you use a clustering strategy (e.g., Node.js cluster module) within a pod.


INBOUND TRAFFIC
VPS
BALANCER (NODE CLUSTER / PM2)
NODE PROCESS 1
NODE PROCESS 2
NODE PROCESS 3
NODE PROCESS 4
CPU 1
CPU 2
CPU 3
CPU 4


INBOUND TRAFFIC
VPS
BALANCER (
STER / PM2)
SVC
pod
pod
pod
pod

Why You Don't need Cluster Mode in K8s

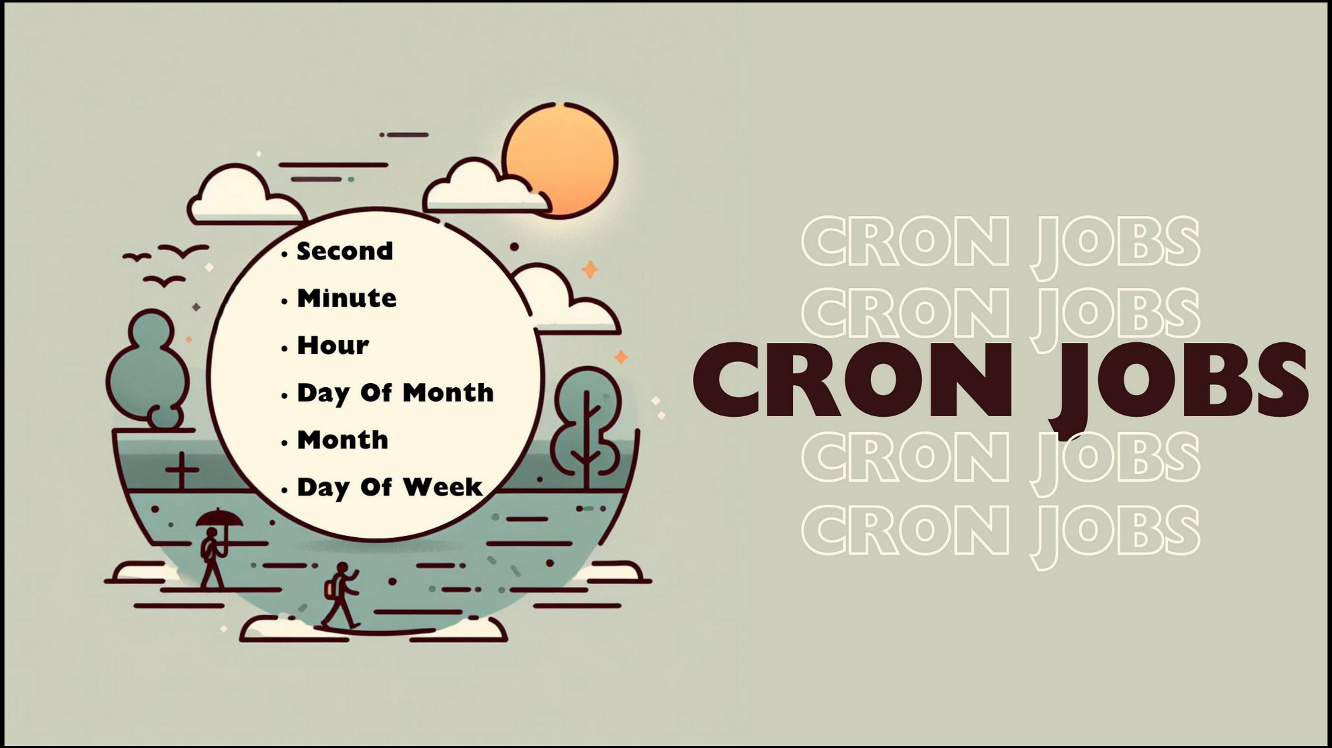
If your app using mainly io/ops
If you don’t use communication between threads
If you don’t use heavy in memory data processing

So It better to scale horizontal and not vertical

Let's get started

- Container Refactor
- Graceful Shutdown
- Why You Don't need Cluster Mode in K8s
- Scheduled Tasks
- Questions

Scheduled Tasks

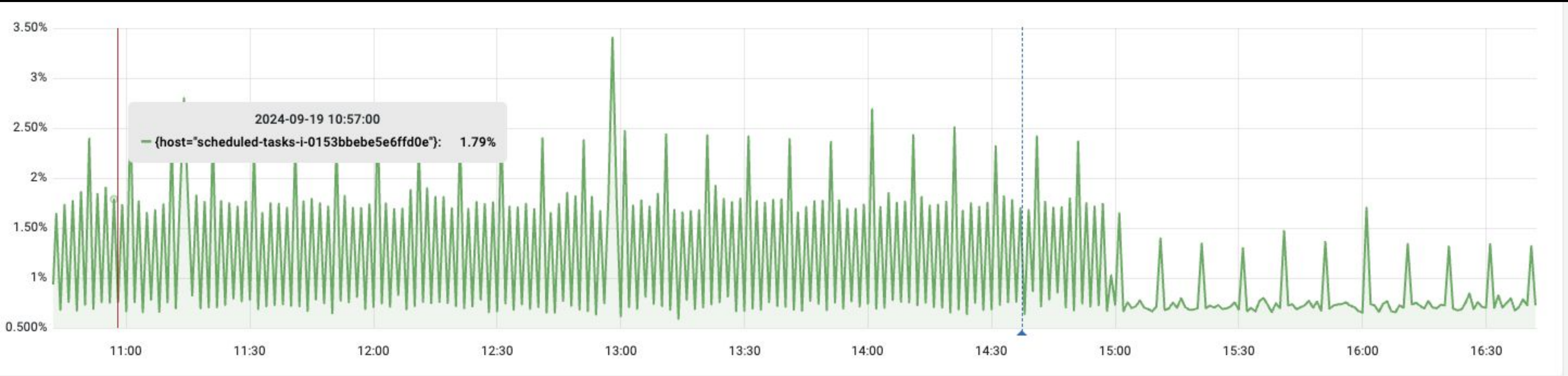


Scheduled Tasks

Today in EC2 we have :
Single Node Server That Run Multiple Cron Jobs on different times

Scheduled Tasks

```
avg(100 - cpu_usage_idle
{service="scheduled-tasks"}) by (host)
```



Scheduled Tasks

CPU Usage Between 2-5 percent

Scheduled Tasks

We are paying hourly no matter if the service is running or not so why not use it only when it runs?

Instance name	On-Demand hourly rate	vCPU	Memory	Storage	Network performance
r5dn.16xlarge	$5.344	64	512 GiB	4 x 600 NVMe SSD	75 Gigabit
m6idn.16xlarge	$5.09184	64	256 GiB	2 x 1900 NVMe SSD	100000 Megabit
i3.metal	$4.992	72	512 GiB	8 x 1900 NVMe SSD	25 Gigabit
i3.16xlarge	$4.992	64	488 GiB	8 x 1900 NVMe SSD	20 Gigabit
m5ad.24xlarge	$4.944	96	384 GiB	4 x 900 NVMe SSD	20 Gigabit
i4g.16xlarge	$4.94208	64	512 GiB	4 x 3750 SSD	37500 Megabit

Scheduled Tasks

```yaml
cronjob:
  enabled: true
  successfulJobsHistoryLimit: 3
  failedJobsHistoryLimit: 1
  backoffLimit: 3
  schedule: "0 0 * * *"
  concurrencyPolicy: Forbid
```

Deploy Pod -> Run Task -> Kill Pod

Let's get started

- Container Refactor
- Graceful Shutdown
- Why You Don't need Cluster Mode in K8s
- Scheduled Tasks
- Questions

Questions ?

Naor Tedgi (Abu Emma)

Thanks!

https://github.com/ntedgi

https://github.com/ntedgi/infra-meetings