Case Study: Node JS On Kubernetes Production

Sharing some insights from our group transition from EC2(Spot) to K8s

- 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.

Containers Refactor

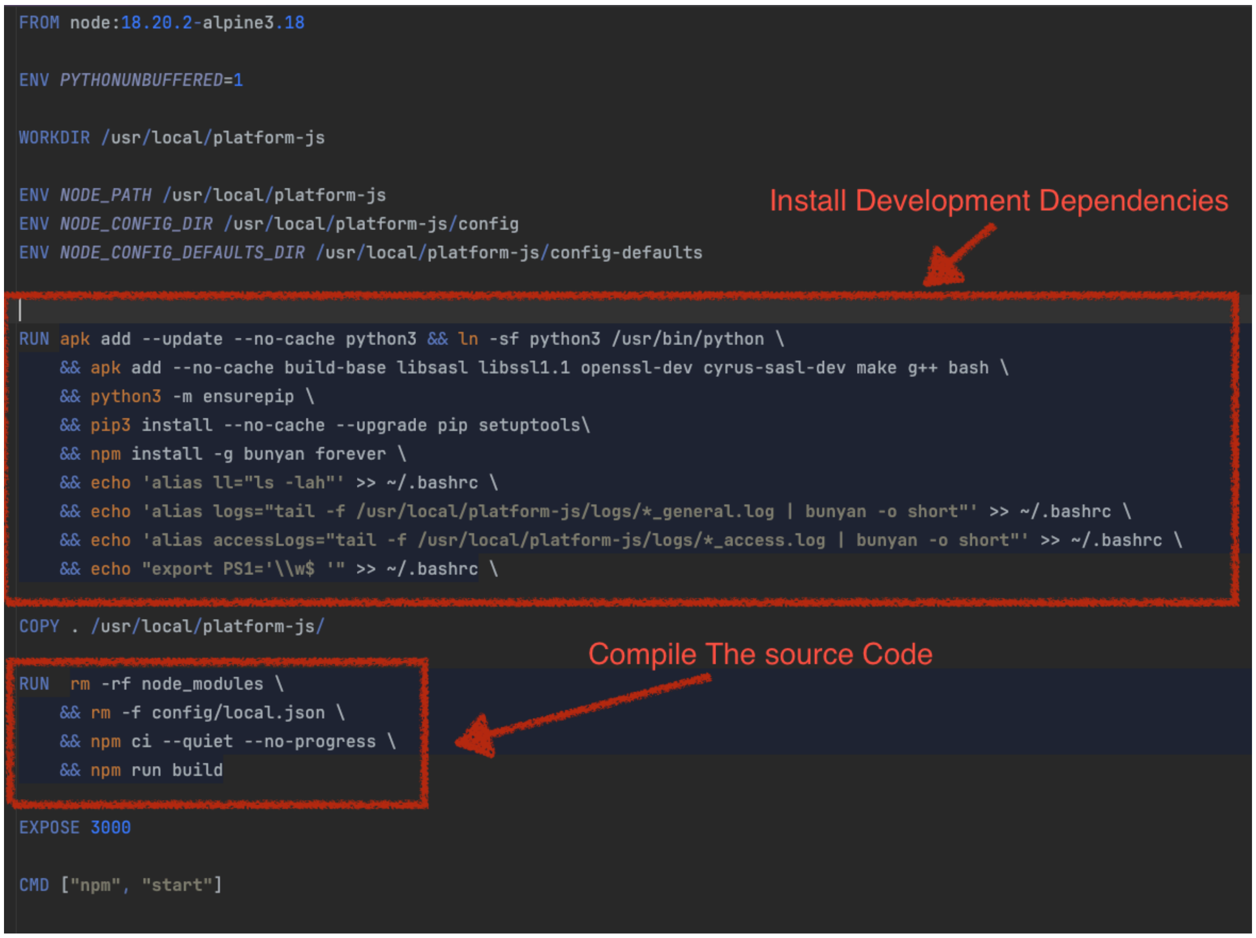
In our current EC2 setup, we pull images from Amazon ECR that contain essential development tools, such as Python and g++. Node.js relies on C++, Node-gyp, and Python to compile C++ add-ons.

In summary, running npm install in a project that utilizes C++ add-ons requires g++ compiler make and python3. Examples of such add-ons include:

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

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

Pulling image	Compile source code	Startup time



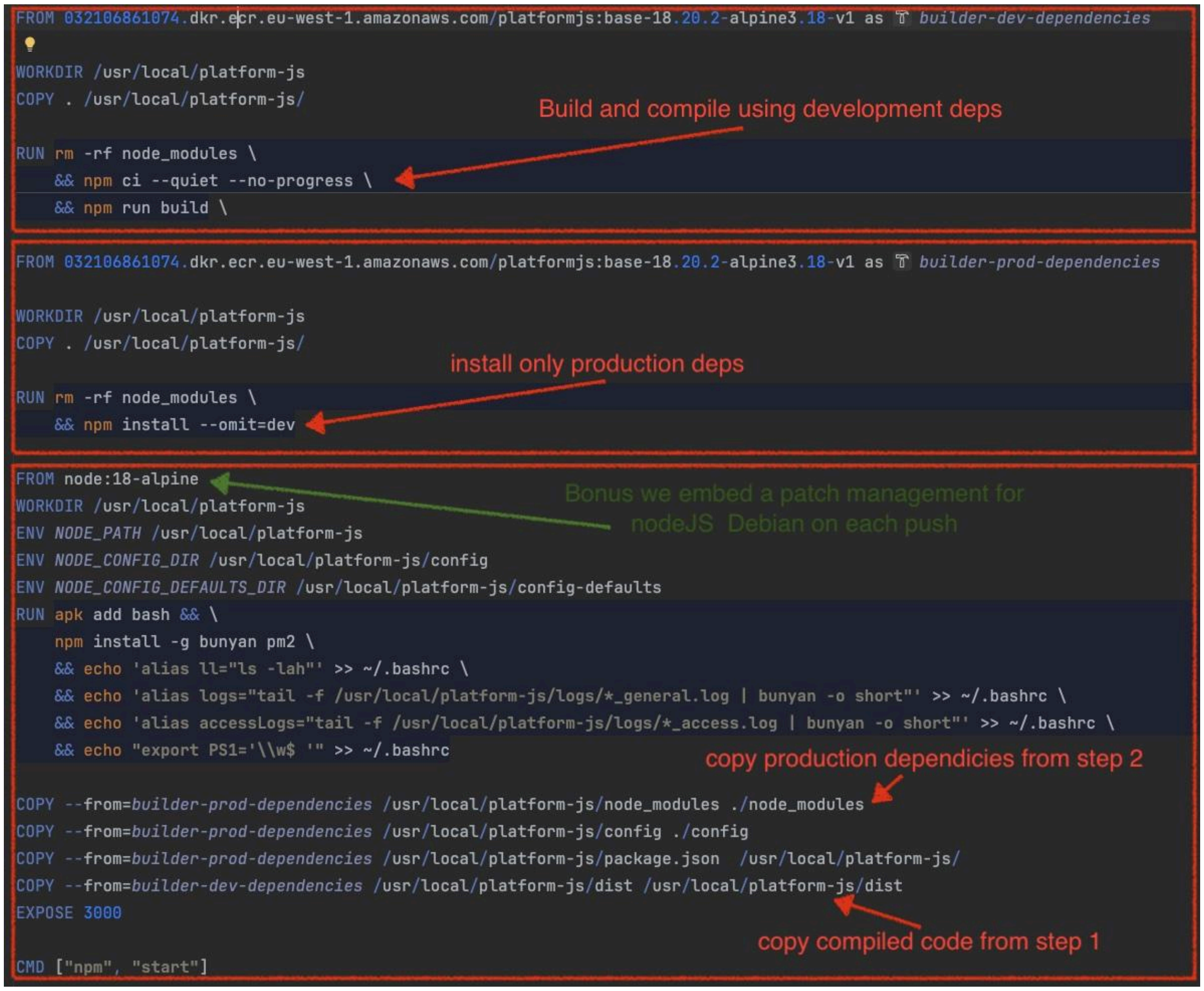
```
FROM node:18.20.2-alpine3.18

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"]
```

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
2. The application startup time is slow because we need to first compile the code before running

We improve it by changing to different approach :

1. We move all our OS development Utils to different docker files 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 032106861074.dkr.ecr.eu-west-1.amazonaws.com/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


Build and compile using development deps
install only production deps
Bonus we embed a patch management for nodeJS Debian on each push
copy production dependicies from step 2
copy compiled code from step 1

```
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 \
```

```
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
```

```
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"]
```

Security Patch Management Automatically :

In node production we have 3 layers of security vulnerabilities

- **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. Linux Distribution (OS)
    2. Building Tools (GCC , Python , OpenSSL ..)

At the latest step, we are utilizing the node:18-alpine Docker image instead of a specific version. This allows us to leverage semantic versioning of container tags. Consequently, for each deployment, any new patch versions of Node.js (e.g., from 18.1.2 to 18.2.3) or updates to the Linux kernel are automatically incorporated. This approach effectively achieves an automated patch management system, ensuring we always deploy with the latest security updates for the runtime layer.

Containers 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%.

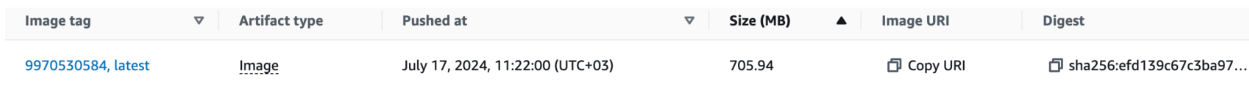


Image tag	Artifact type	Pushed at	Size (MB)	Image URI	Digest
9970530584, latest	Image	July 17, 2024, 11:22:00 (UTC+03)	705.94	Copy URI	sha256:efd139c67c3ba97…

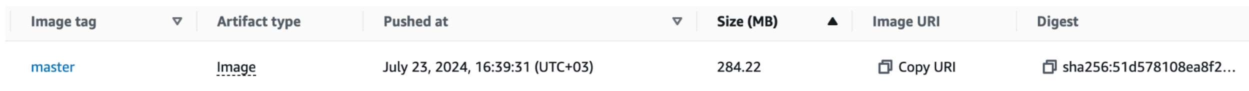
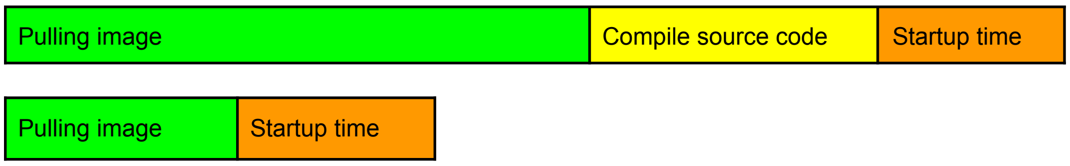
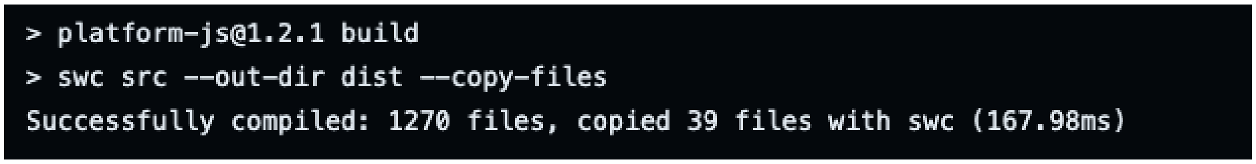


Image tag	Artifact type	Pushed at	Size (MB)	Image URI	Digest
master	Image	July 23, 2024, 16:39:31 (UTC+03)	284.22	Copy URI	sha256:51d578108ea8f2…

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

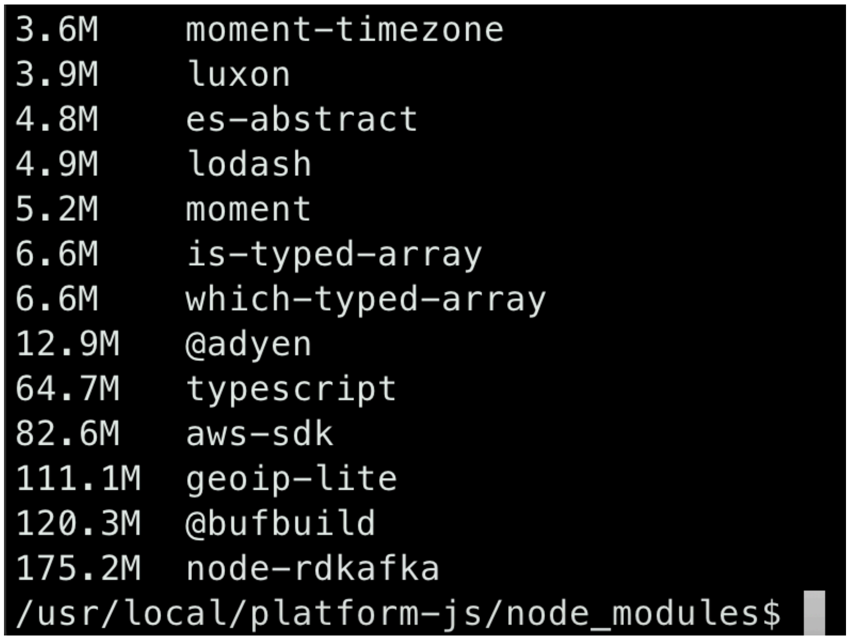


Now you can say compile source code? Its very fast



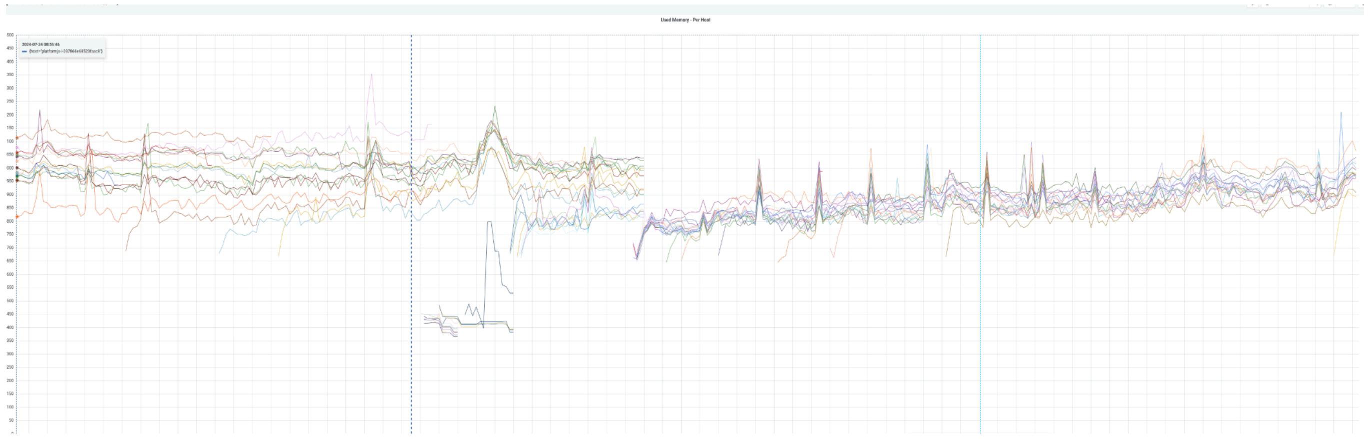
```
> platform-js@1.2.1 build
> swc src --out-dir dist --copy-files
Successfully compiled: 1270 files, copied 39 files with swc (167.98ms)
```

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



```
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



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.

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

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.

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.

To application that not runs as http server for example

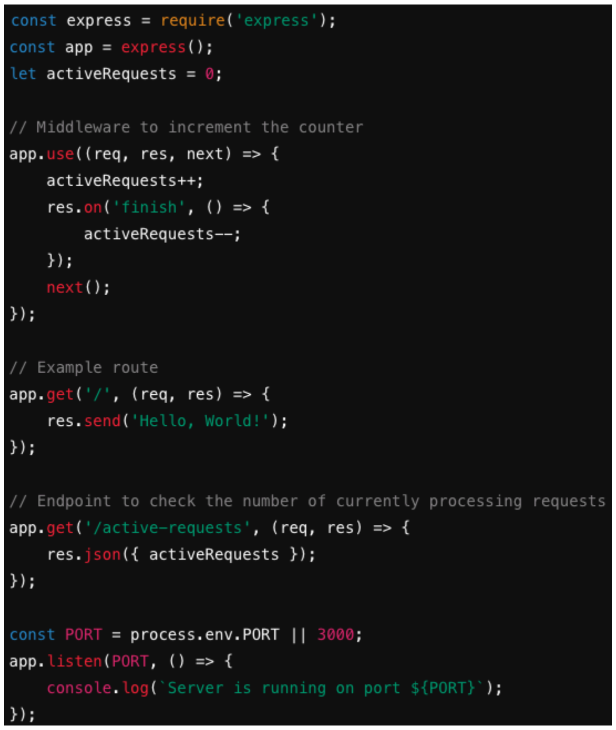
Kafka consumer **(aka offline js)**
CronJobs **(aka schedule tasks)**

BTW we didn't have a health check until now to deploy!!!!

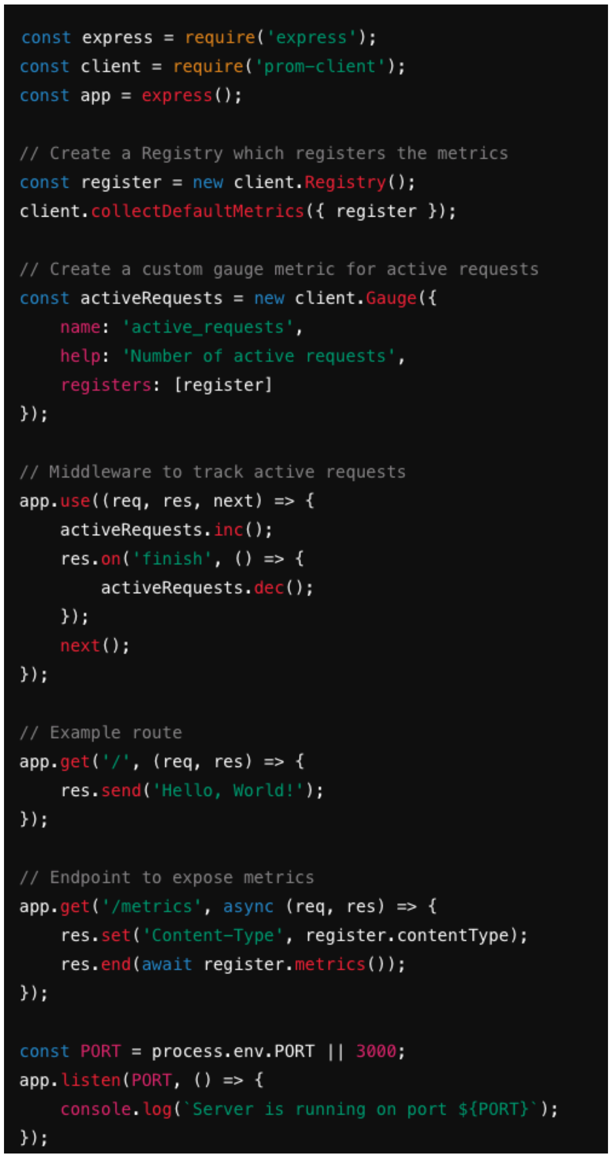
We add default http server that handles the signal and shot down the process for them

For now we only give our running request a 30sec timeout to finish before killing thar app

TBD : We want to check inner counter for running request on server natively or using metric time series db



```javascript
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}`);
});
```



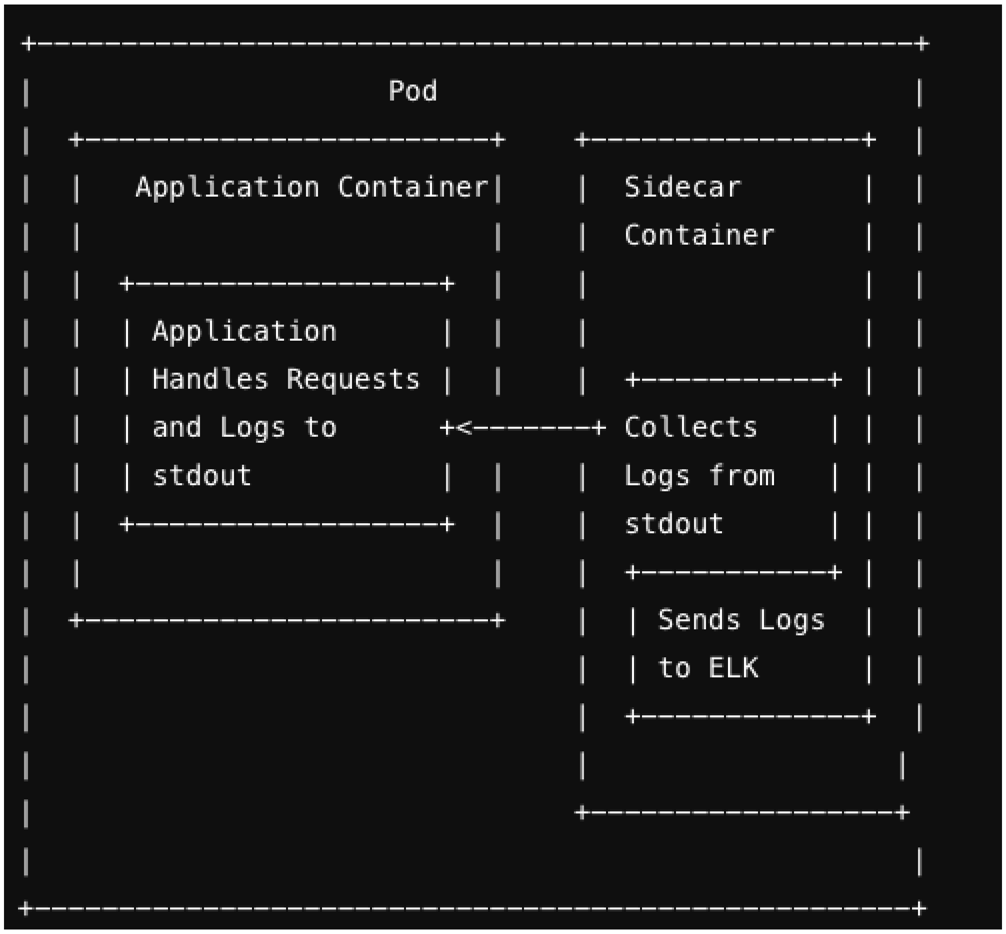
```javascript
const express = require('express');
const client = require('prom-client');
const app = express();

// Create a Registry which registers the metrics
const register = new client.Registry();
client.collectDefaultMetrics({ register });

// Create a custom gauge metric for active requests
const activeRequests = new client.Gauge({
    name: 'active_requests',
    help: 'Number of active requests',
    registers: [register]
});

// Middleware to track active requests
app.use((req, res, next) => {
    activeRequests.inc();
    res.on('finish', () => {
        activeRequests.dec();
    });
    next();
});

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

// Endpoint to expose metrics
app.get('/metrics', async (req, res) => {
    res.set('Content-Type', register.contentType);
    res.end(await register.metrics());
});

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

Logging

In spot logs written to files on disk then ingest using ELK stack to ELK
Our pods will runs with sidecar that listen to stdout and redirect the output to elk

1. **Main Application Pod**:
   - Contains the primary application container.
   - The application handles incoming requests and produces logs to stdout.
2. **Sidecar Container**:
   - Runs alongside the main application container within the same pod.
   - Collects the stdout output from the main application.
   - Forwards these logs to the ELK stack (Elasticsearch, Logstash, Kibana).

Here is a diagram to represent this setup:



```
+----------------------------------------------------+
|                     Pod                             |
|  +------------------------+    +----------------+  |
|  |   Application Container|    |  Sidecar       |  |
|  |                        |    |  Container     |  |
|  |  +------------------+  |    |                |  |
|  |  | Application      |  |    |                |  |
|  |  | Handles Requests |  |    |  +-----------+ |  |
|  |  | and Logs to      +<-------+ Collects    | |  |
|  |  | stdout           |  |    |  Logs from   | |  |
|  |  +------------------+  |    |  stdout      | |  |
|  |                        |    |  +-----------+ |  |
|  +------------------------+    |  | Sends Logs  |  |
|                                |  | to ELK      |  |
|                                |  +-------------+  |
|                                |                  |
|                                +------------------+
|                                                    |
+----------------------------------------------------+
```

Monitoring

Metrics i want to getter for each node process
https://github.com/RafalWilinski/express-status-monitor/blob/master/src/helpers/gather-os-metrics.js
https://github.com/sematext/spm-agent-nodejs/blob/master/lib/httpServerAgent.js
https://github.com/bripkens/event-loop-stats#readme
https://github.com/siimon/prom-client

- Event loop statistics
- Heap Usage
- GC Usage-

Fine Tuning How Many Resources Should You use

Using PM2 we can monitor our services during runtime and understand how many resources they need

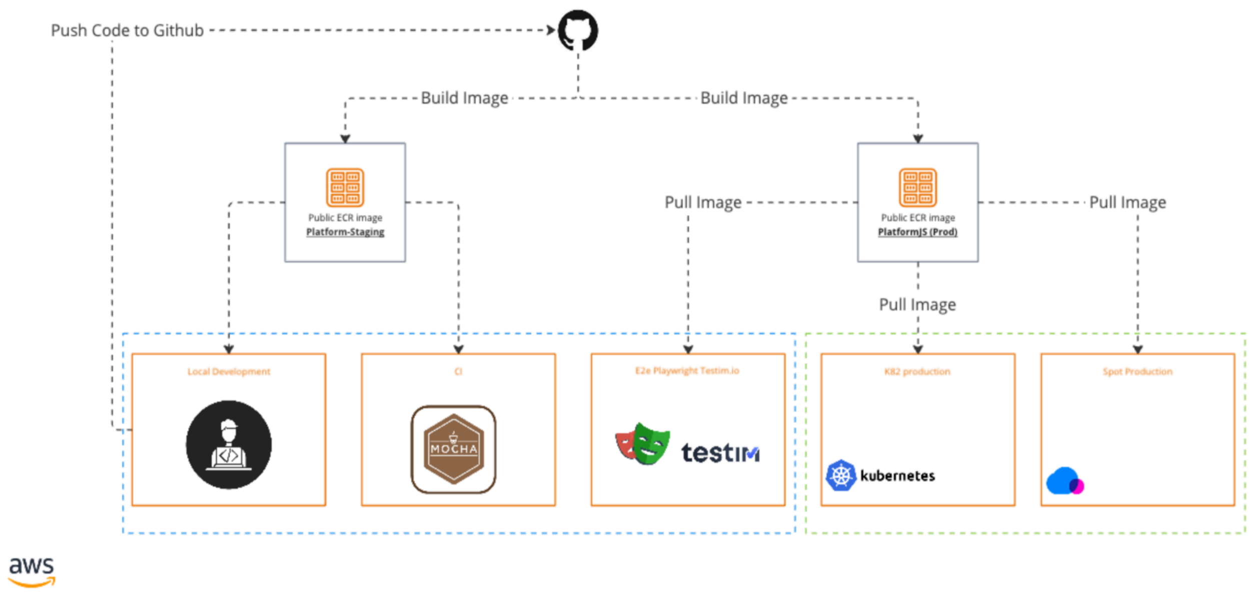
Support All Environment with new Docker containers

<u>We have 5 runtime Environments:</u>

1.) **Local Development**
2.) **CI - github-actions**
3.) **E2E Tests**
4.) **Production - K8s**
5.) **Production - Spot (EC2)**

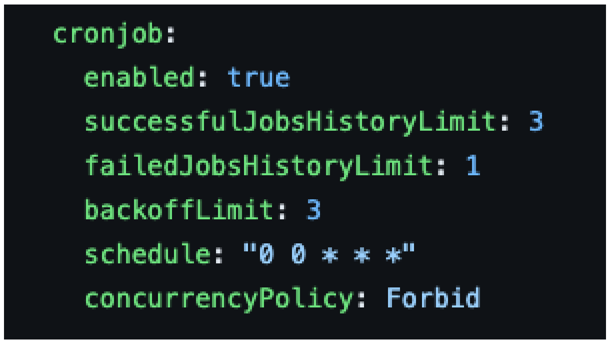
We now using the staging docker container for local development and CI
And using the production image for E2E black box test thats help us closing the gap between running integration test on docker container that not going to deployed to production

ECR Architecture Demand Platform-JS



Schedule tasks

Today we run on EC machine and running multiple cron process on single machine
We have ~15 cron jobs running in different timing some of them once a day other every 15 minutes
Instead we move to Kube cron job for each task



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

Don’t Use Cluster Mode In K8s!!

Micro Services Separation

Today platform js is maintaining all services and workers we need to separate all this Infrastructure to multiple github repositories

IService:
- **Build**
- **Publish**
- **Run**

IOrchestrator : IService:
- **Build**
- **Publish**
- **Run**

Setups.yaml

Services:

```
        [platform-js] : @commit
        [Audit]:@commit
        [Network-operations]:
        [Campaign]
        [Reporting]
```

executionType: [parallel | order]

Network-ops-setup.yaml

Secret Manager Separation (Consul)