# CI/CDs pipeline for App and Infra

## Executive Summary

The aim of this CI/CD deployment methodology is:

* Deploy an Node.js application
* All deployments should use artifacts, same for SQL with DacPac
* Deploy infrastructure services (configuration, servers, pools and services like DHCP or NTP)
* Deployment should be done in the Cloud
* Be compliant with AWS ecosystem
* Software is Stateless

## Background

Since this is a new project, feel free to use any services/tools you think are relevant - there is no legacy.

## Goals

Explain the goals of the project:

* the “NodeApi” needs to run 24/7, it should be well monitored.
* API response format is JSON
* all deployments should use artifacts, same for SQL with DacPac
* modularity
* evolvability
* provide a CI/CDs pipeline for application and infrastructure
* offer high quality service
* deploy easily what we need without effort
* add more values in daily work by creating an infrastructure as code
* reducing the time of release cycles
* reducing outages
* server usage efficiency
* no vendor locking

## Non Goals

List all non-goals of the project:

* We will not manage an external cluster
* We will not test and simulate a full platform (like <https://kitchen.ci/>)
* We will not benchmark platform for architecture and components

## Design

The idea is to streamline a release (both applicative and infrastructure) through the different stages from the build to production with automation, which will also require the completion of those steps:

* Documented
  + Should be easy to share
* Scalable
  + Should be able to scale horizontally (more servers)
* Resiliency
  + Should be able to respawn easily
  + Orchestration
  + Rolling upgrade
* Available
  + Should be available everywhere
  + Usage of percentile like 99.9th
  + High Availability
  + Availability Zone
  + Resource quota and throttling
* Maintainability
  + Should be easy to maintain and support
* Simple
  + Should hide complex parts with high level data-structure
  + Give control back to users
* Instrumentation and visibility
  + Should be accessible with different tools
* Alerting and reporting features (flexible and powerful alerting)
  + Should be able to alert one or many teams easily
  + Should be able to correlate failures or alerts
* Reliable and trustworthy
  + Should be trusted by testing (unit/functional...)
  + Should be simulated
* Measurable
  + maintain historical data
  + easy to create, delete or update new metrics

We can see this CI/CDs platform into 3 XaaS:

1. Software as a Service (SaaS) called to the application Node.js API
2. Platform as a Service (PaaS) called to the rollout of new API version
3. Infrastructure as a Service (IaaS) called to provide and scale instances (SQL, NOSQL, EC2, S3)

AWS vendor provides Beanstalk (its supports Node.js) with Lambda. Using a PaaS concept instead of FaaS (Lambda) permit to not be enclosed on the proprietary language. Beanstalk allows to upload your code, manage load balancing, logs, metrics management, alerting by mail, application version management, and DNS resolution. All in one but we are the same lock by the Cloud vendor.

The main trade-offs considered:

* **Infrastructure Definition**. This is closest to defining actual server, network and storage. It will be done by Terraform
* **Application+ Configuration Definition**. This is often a composite representation of infrastructure resources and relationships that together deliver a functional system. It will be done by Ansible
* **Runtime or system configuration**. This operates on compute resources to overlay system libraries, policies, access control, etc. It will be done by Kubernetes

## Infrastructure Roll-out

Step by step approach:

1. AWS credentials for Terraform and Ansible authentication
2. We define our environment with Terraform (security like ssh key or certificates, LB, VPC, network layer, Elasticsearch, EC2 instances, profile for Kubernetes, etc). We define all AWS cloud services and our K8S clusters
3. Then, Ansible install all components required by Kubernetes (controllers, workers, etcd) and Prometheus.
4. Once all playbooks are applied. The infrastructure rollout is done.
5. All code can be stored on github repository

## Applicative Roll-out

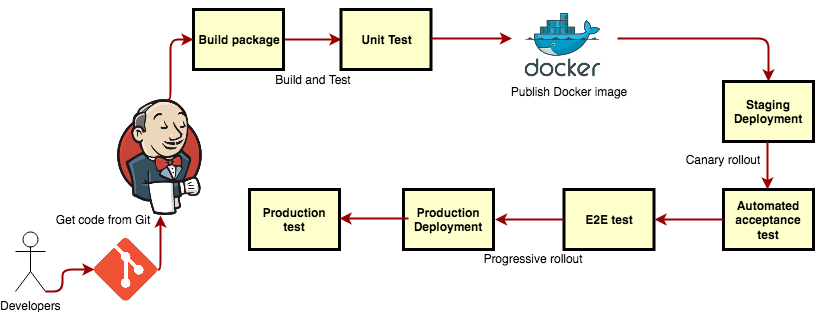
We use a current structure which consists of 3 environments:

1. Development
2. Staging
3. Production

Step by step approach:

1. Get code from Git
   1. Developer pushes code to Git, which triggers a Jenkins build webhook.
   2. Jenkins pulls the latest code changes (small commit method)
2. Run build and unit tests
   1. Jenkins runs the build and create the artifacts and the docker image (docker image with built artifact)
3. Publish Docker image
   1. Docker image is pushed to the registry
4. Deploy to Staging
   1. Application is deployed to Kubernetes staging cluster
   2. Run automated tests against the deployed application in the Kubernetes staging environment.
   3. Run also End to End tests against the deployed application
5. Deploy to Production
   1. The application is deployed to the production cluster if the application meets the defined criteria. Progressive rollout, canary or blue/green deployment can be used at this stage.
   2. Sanity tests also called production tests run against the deployed application.
6. All code can be stored on github

A rollback can be done at any stage and ABTesting could be implemented.



The main benefit of this stack is flexibility since it allows you to implement practically any type of workflow. This workflow can be extended or complexified depending on your development needs. A good option is canary rollout, which you can define in three phases. If you have six pods, upgrade and verify two of these pods, or 33%. Then, do the same with three pods, or 50% of them. Finally, upgrade and verify all six pods, or the complete environment. Kubernetes essentially keeps multiple versions of controllers active within the same environment. It can also resize the percentage or count for any controller.

## Monitoring and Alerting

Monitoring, Alerting, Logging, Tracing are often called event management who refers to any type of software used to promote, plan, execute and evaluate events. Metrics, monitoring, and alerting are all interrelated concepts that together form the basis of a monitoring system. Alerting is the responsive component of a monitoring system that performs actions based on changes in metric values. Alerts definitions are composed of two components: a metrics-based condition or threshold, and an action to perform when the values fall outside of the acceptable conditions. There are three distinct kinds of monitoring offers, from low to high added value:

* **Monitoring operations:** the software monitors operations faster and more efficiently than would otherwise be possible.
* **Diagnosing problems:** the reports produced by the monitoring tool will tell you if a website went down because of a website issue, a web server problem or a network failure.
* **Avoiding failures:** the monitor will identify hardware or software that is experiencing some type of failure on a regular basis. The admin can troubleshoot the issue before it turns into a devastating crash.

In this case, we will use the basic function of CloudWatch from AWS (no escalation management or all part of observability like tracing) added to Prometheus exporter+server on each service with AlertManager.

Log management will be done by Elasticsearch and FluentD and Grafana for dashboarding.

## SLA

We are going to use the current SLA for the platform with 99.99% of availability. Qos implementation with Grafana.

## Resiliency

* HA design for slaves and master. Not for DC
* LB is redundant in every DC
* Master is redundant in every Kubernetes cluster
* Slave is redundant in every Kubernetes cluster

## Security

Kubernetes API is powerful and allow many actions on containers and cluster; it needs to be secure and control. Kubernetes authorization, control how app are configured (privileged mode, capabilities, net, host) can be controlled through kubernetes API and security profile.  
We need to define security profile with IAM from AWS (access keys/secret keys). We can also do pen, stress and monkey tests, S3 buckets or docker image auditing/scanning.

## Backup

Snapshot will be done with S3 buckets.