



>>>network.toCode()

# Nautobot Deployment Training

Get ready to deploy Nautobot in Production

Architecture Team

Original: March 2023



## Goals

Understand Nautobot Architecture

Identify the trade-offs and hints per component

Spin up a local development and production environments



## Agenda

Nautobot Stack

Nautobot Operations

Deployment Environments

Deployment Environments (Hands-on)



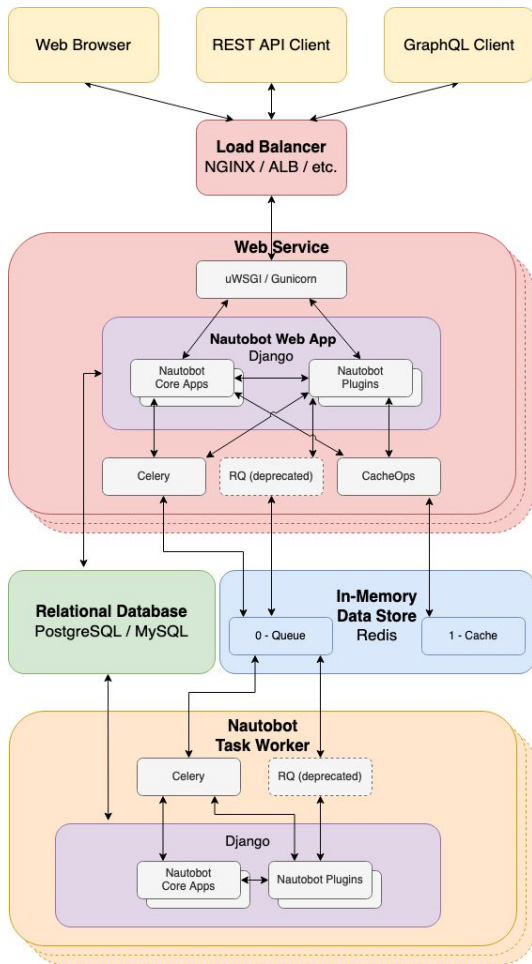


# >>> Nautobot Stack

<https://docs.nautobot.com/projects/core/en/stable/>

# >>> Nautobot Application Stack

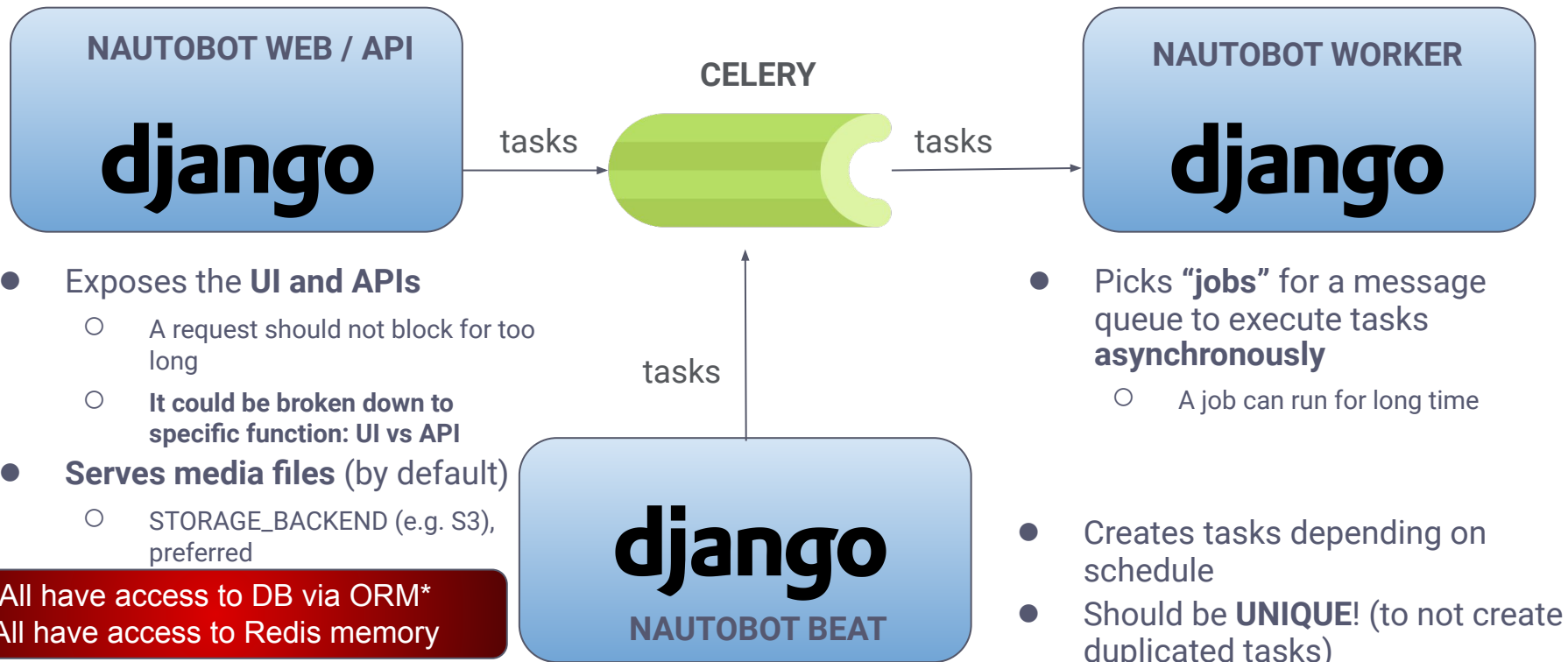
- **Load Balancer** implements the SSL termination (with certificates), serves static content, and implements load balancing.
- **WSGI** connects the web application with a web server.
- **Nautobot Web App** is a **Django** based project that offers an endpoint to a SQL database and some extra features.
- **Relational Database**: Persists data.
- **Task Queuing** offers a multi tenant system to register tasks to be picked by workers when available. It helps to decouple request and task delivery.
- **Nautobot Worker App** is the same Nautobot Django application that is reading from the Task Queuing to pick jobs and executed them asynchronously.



## >>> Understand the Requirements

- The Nautobot Application stack can be deployed in many flavors, from a simple development environment to complex high-available distributed application (with a exponential cost and complexity)
- It's key to understand, from the beginning, two requirements that will guide our design:
  - Which **business workflows** will be supported by Nautobot, and what **level of uptime is expected?**
  - What are the current **standards within the business?**
- Finding the right spot, to meet expectations without **overcomplicating** is key to deliver the right solution.

## >>> Nautobot services: WebUI / API / Worker / Beat

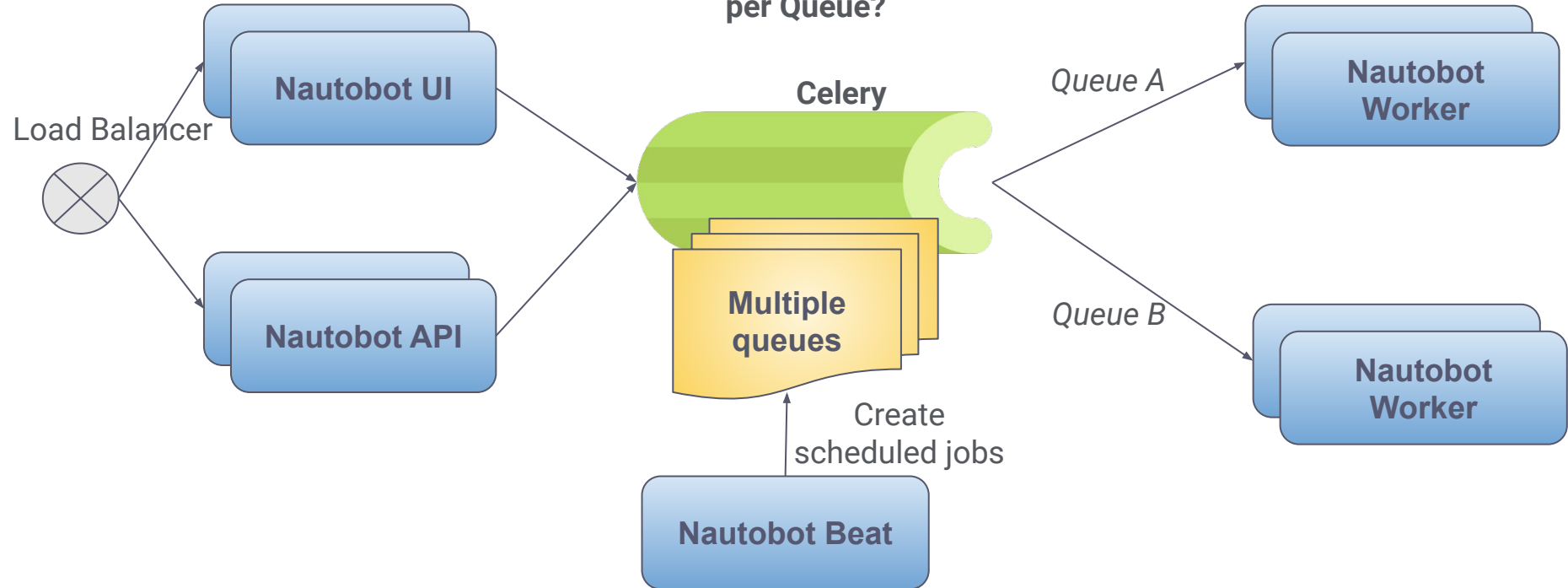


*Django ORM is a powerful feature of the Django web framework that allows developers to interact with application data from various relational databases*

## >>> Recommended Architecture

Always keep a balance between complexity and features

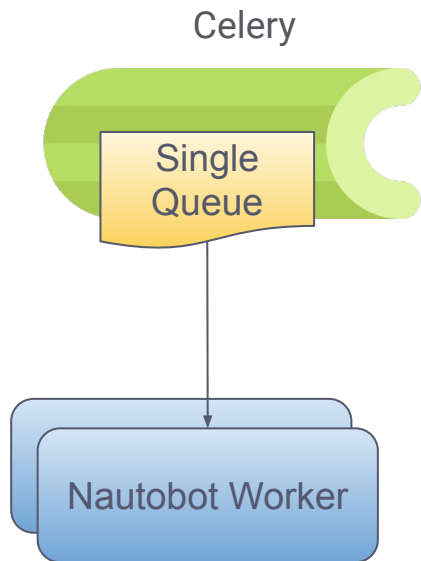
How to distribute Jobs per Queue?



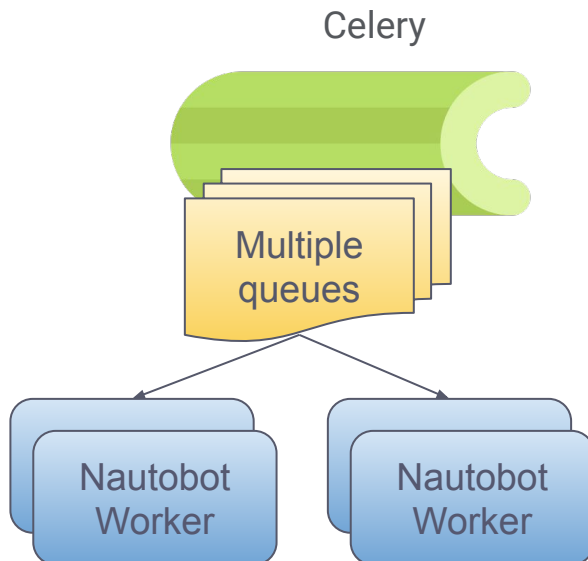


# >>> Recommended Architecture

## Jobs - Queue distribution

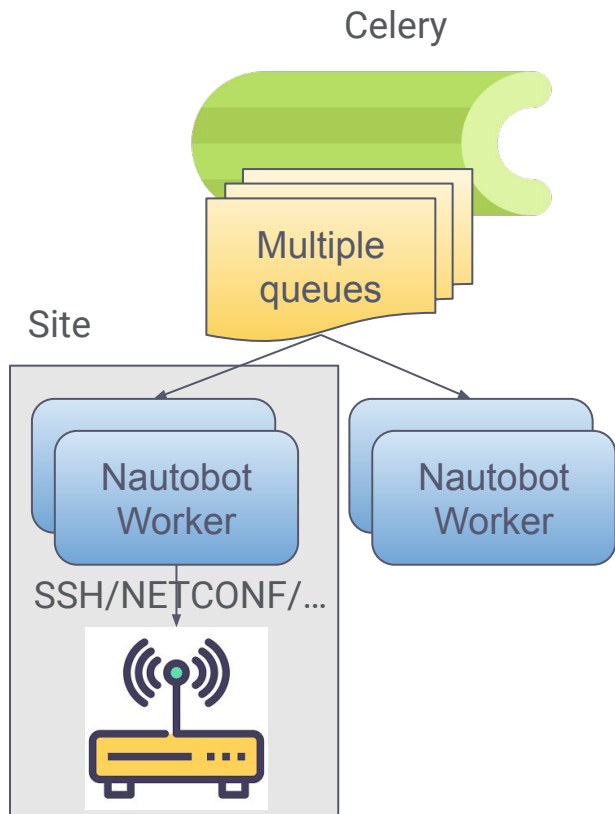


*Simple deployment, one queue for all the Jobs*



***Distribute Jobs per function*** to different workers. For instance, some for SSoT jobs, and the rest for the other tasks

***Distribute Jobs per location***, so the Jobs that need to access network devices are located in the same site.



## MySQL

Stable

Database commonly found in enterprise environments  
Might be first customer's first choice if database team exist

- Prefer Managed service (cloud or company provided)

## Postgres

Stable

Commonly found in "all-in-one" deployment types: single VM, or if small Nautobot instance is administered by network engineering teams

- Prefer Managed service (cloud or company provided)
- Only option in NetBox

## Dolt

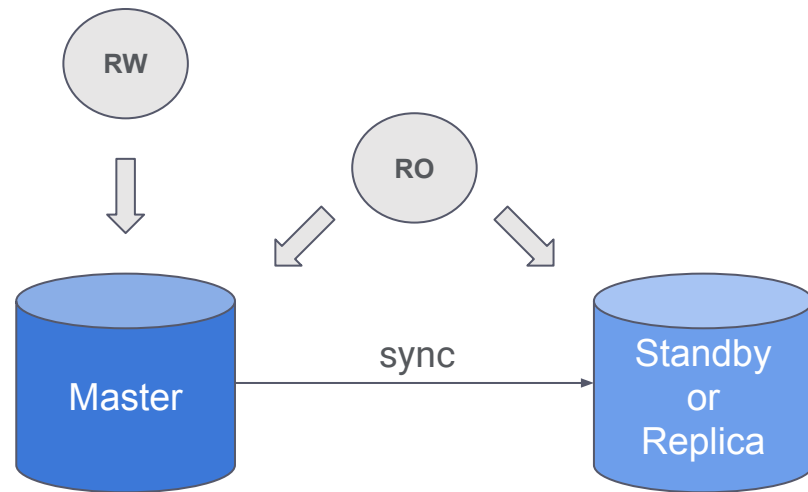
Future

Experimental database offering GIT like experience (Merge Requests).  
Released version 1.0 in May

- Pending integration with latest release
- Good to demonstrate concept
- Provides GIT like merge requests

# >>> Database High Availability

- **High Availability** (keeping nodes in sync) **is always challenging**, especially in distributed environments. There are commercial solutions offering different trade-offs



If possible, **delegate database management** (e.g. AWS RDS)

# >>> Database Replication

A Disaster Recovery strategy is a must for any production environment

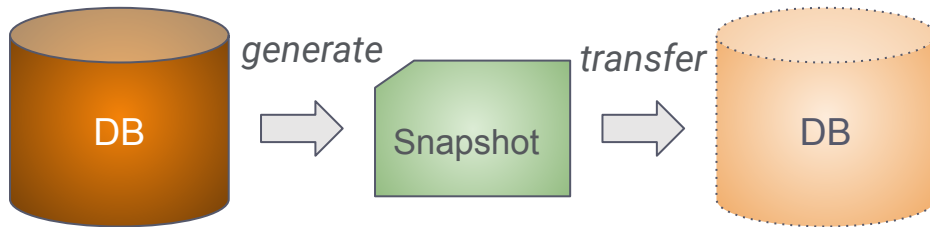
Database replication is the process of creating copies of a database and storing them across various on-premises or cloud destinations. It improves data availability and accessibility.

Use it for:

- **Disaster Recovery:** have standby copies of the database to recover Nautobot state (usually, in a different environment).
- **Scalability:** distribute load can lower data latency (regional), and reduces the server load

Challenges:

- Ensuring data consistency
- Managing multiple servers and technical requirements



# >>> Redis

## In-memory store

01

### Cacheops

- Caching aimed to improve performance, although many issues existed
- Feature disabled by default since 1.5
- **Feature will be removed in 2.0**

02

### Job Queueing (Celery)

- **Celery queue is supported in Redis**, even it could also be implemented in other Brokers such as RabbitMQ

03

### Caching - In-memory

- Nautobot SSoT / Diffsync (can) leverage Redis to share data among processes
- Nautobot Chatops use Redis as session storage

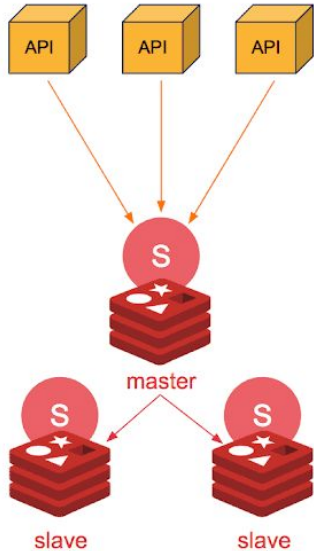


# >>> Redis Considerations

## High-Availability and Scalability

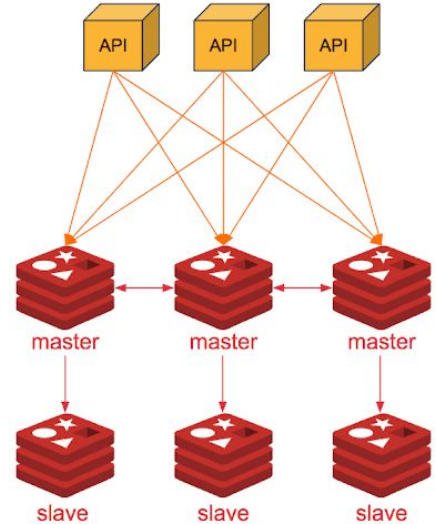
[RabbitMQ](#) is a more solid solution for delegating **tasks** in production

Redis Sentinel



	Redis Sentinel	Redis Cluster
High Availability	YES	NO
Scalability	NO	YES
Data Sharding	NO	YES
Replication	Async replication, compromising consistency	
Limitations	Issues with remote connections or DNS support	Only 1 DB per cluster

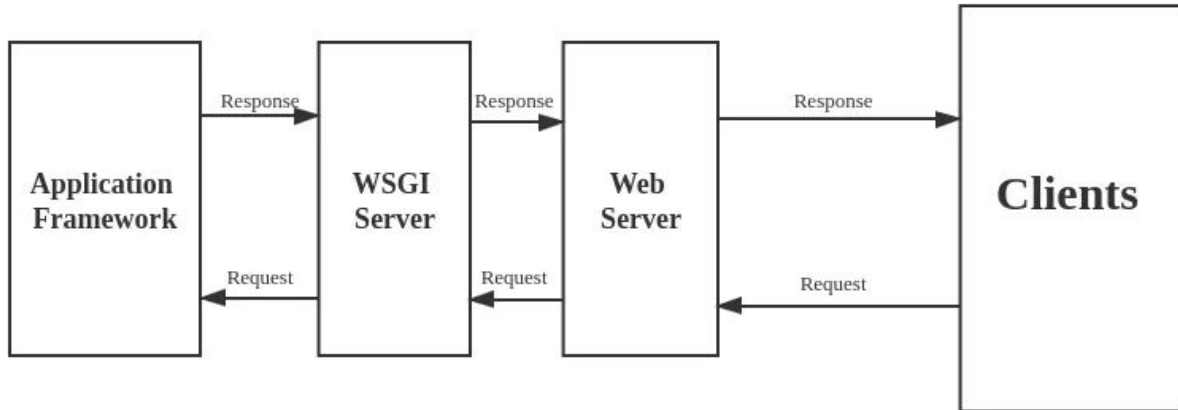
Redis Cluster



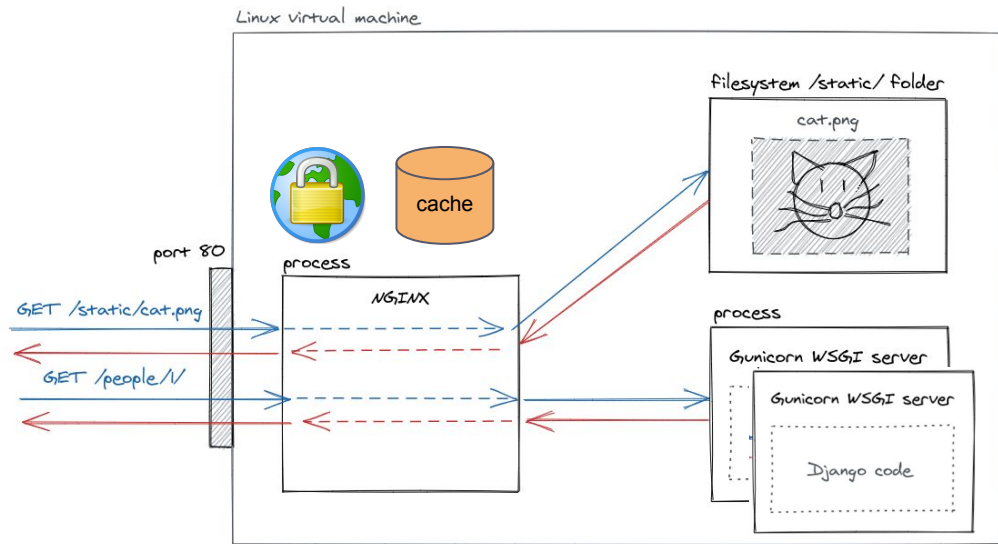
WSGI (Web Server Gateway Interface) is a simple calling convention ([PEP 3333](#)) for **web servers to forward requests to web applications** or frameworks written in the Python programming language.

WSGI apps take care of all the HTTP details, allowing simpler apps

It connects the Web Server (e.g. NGINX) to the Python code implementing both interfaces. It allows having multiple requests in the same application process.



# >>> Load balancer / Reverse Proxy / API Gateways



	Reverse Proxy	API Gateway
URL Rewrite	✓	✓
Load Balance	✓	✓
Prevention from Attack	✓	✓
Caching	✓	✓
SSL Encryption	✓	✓
Orchestration / Aggregation		✓
Protocol Translation		✓
AuthN / AuthZ		✓
IP Whitelisting		✓
Rate Limiting, Throttling, Quota		✓
Retry Policy, Circuit Breaker		✓
Logging, Tracing, Correlation		✓

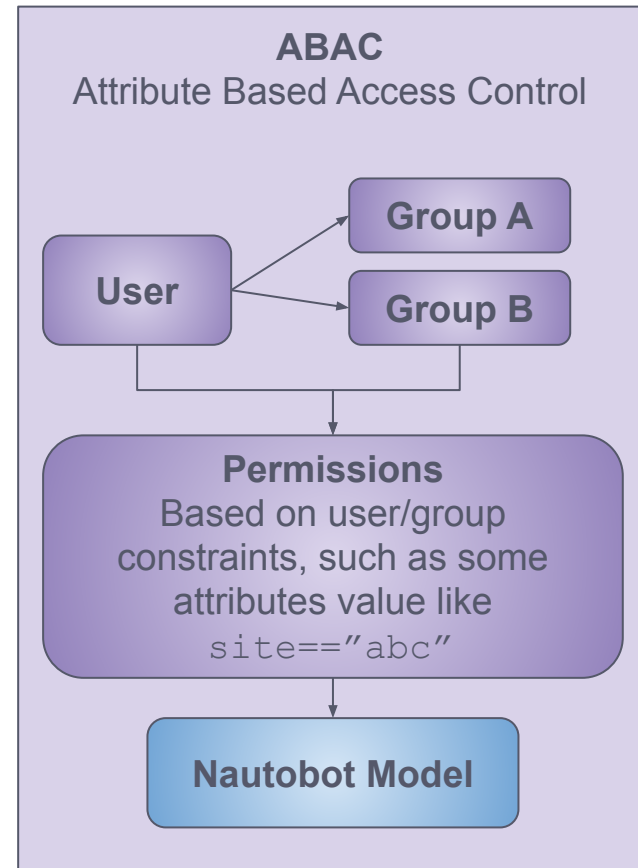
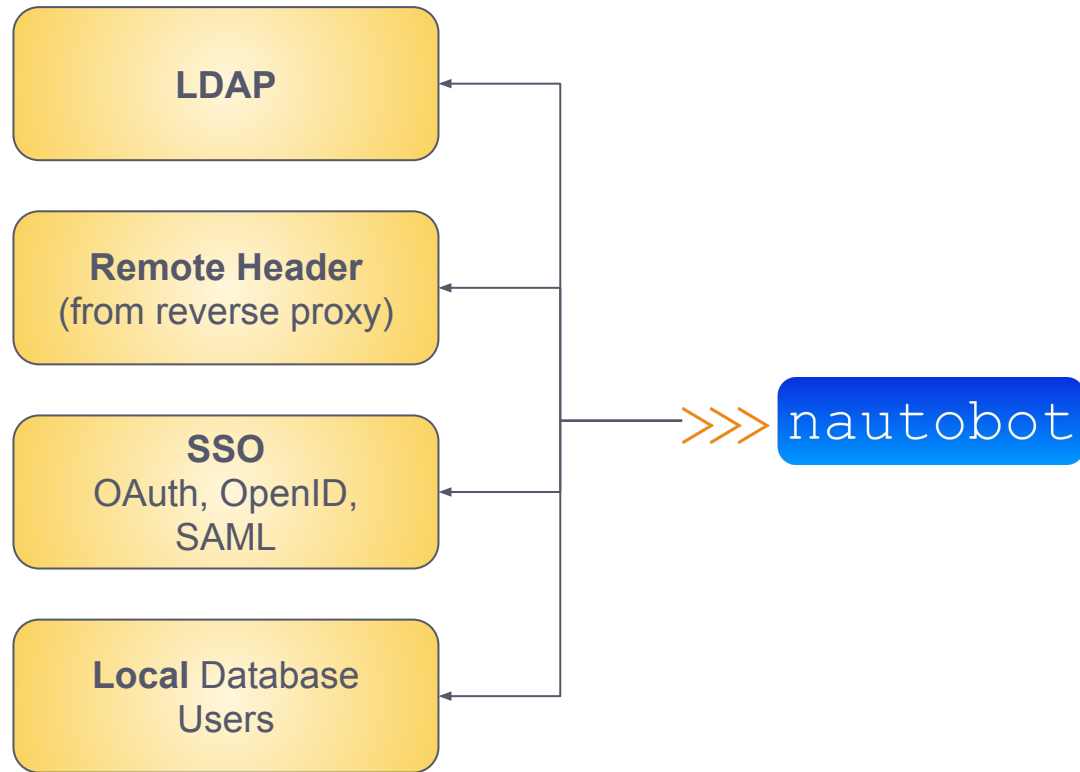
Examples: F5, Avi, Citrix LB, Nginx, HAproxy, Apache, Traefik, AWS ALB, Kong, Apigee, Tyk, Krakend

**Load Balancers / Reverse Proxies are getting closer to API Gateway features**



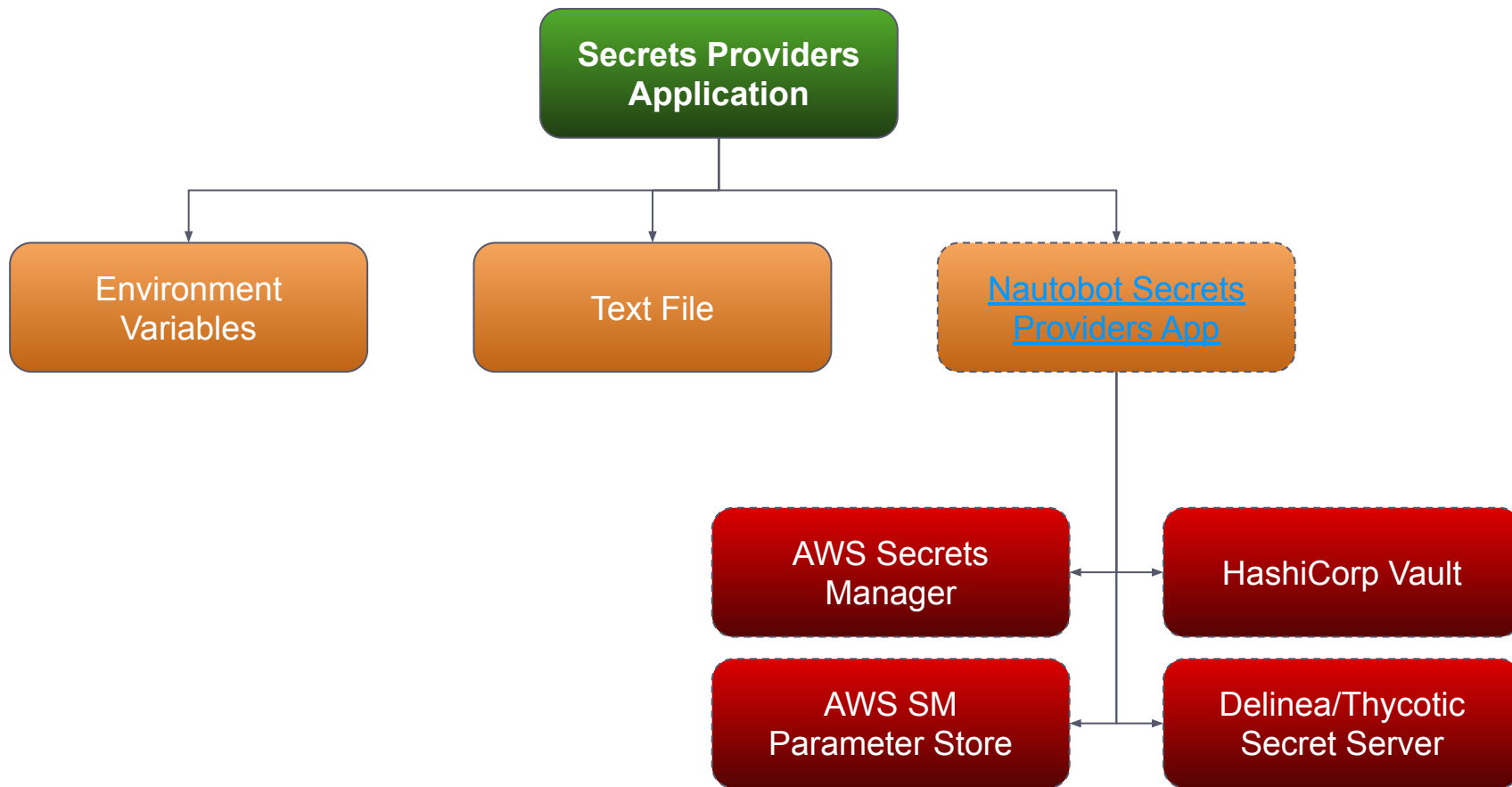
# >>> Nautobot Operations

# >>> Authentication and Authorization





## >>> Secrets Integration



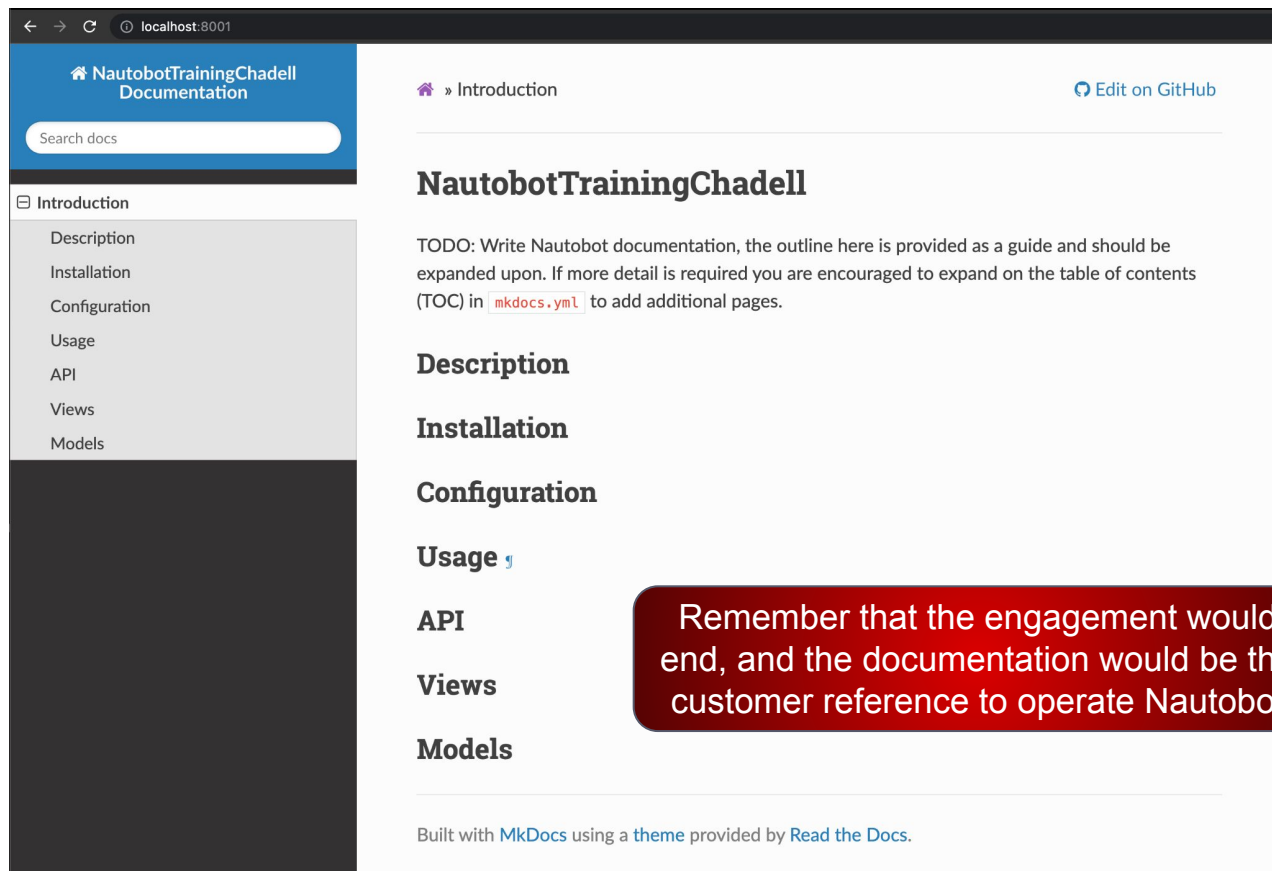
# >>> Documentation

[Nautobot](#) cookie cutter comes with a ready to use **DOCS server** based on MkDocs

You could also use simple README.md as last resort

**Documentation should help Nautobot users to:**

- Install & Configure
- Usage via UI/API
- Development of new features (e.g. Jobs)



# >>> Tuning Nautobot

01	Global	<ul style="list-style-type: none"><li>• Never use DEBUG in Production!</li></ul>
02	NAUTOBOT web/api	<ul style="list-style-type: none"><li>• MAX_PAGE_SIZE (GUI)</li><li>• PAGINATE_COUNT (API)</li><li>• CHANGELOG_RETENTION</li></ul>
03	NAUTOBOT worker	<ul style="list-style-type: none"><li>• <b>Multiple Job queues to isolate Job responsibilities</b></li><li>• Concurrency for long standing Jobs with low CPU usage</li><li>• <b>Celery task limits:</b> hard and soft (by default &lt;10 &amp; 5 min)</li></ul>
04	WSGI	<ul style="list-style-type: none"><li>• By default uWSGI is used (others: Gunicorn)</li><li>• <b>Processes:</b> <math>2n+1</math> (<math>n</math> = cores)</li><li>• <b>Harakiri:</b> bigger than reverse-proxy timeout</li></ul>
05	Reverse Proxy	<ul style="list-style-type: none"><li>• <b>Caching:</b> important for high-stress environments (i.e. telemetry)</li><li>• External <b>static content</b></li></ul>

# >>> Monitoring and Troubleshooting

**Metrics**  
METRICS\_ENABLED  
(*Prometheus format*)



**Nautobot Capacity  
Metrics App**

**Logs**  
LOGGING

<https://github.com/nautobot/nautobot/tree/develop/examples/logging>

**Django Shell**  
nautobot-server shell-plus

An application without observability and troubleshooting tools, it's not production ready



# >>> Nautobot Deployment Checklist



# >>> Nautobot Deployment Checklist

<b>Nautobot Services</b>	<ul style="list-style-type: none"><li>• Access to/from services (LB, DB, REDIS, Network, etc.)</li><li>• Scalability and resilience: decouple functions, multiple queues</li></ul>
<b>Nautobot Configuration</b>	<ul style="list-style-type: none"><li>• <a href="#">Are Nautobot logs being collected/stored?</a></li><li>• Are logs being automatically rotated?</li></ul>
<b>Database</b>	<ul style="list-style-type: none"><li>• Do all the Apps support MySQL?</li><li>• Determine the level of HA required</li><li>• Data Replication, Disaster recovery plan</li></ul>
<b>Redis (+ other brokers)</b>	<ul style="list-style-type: none"><li>• Which Apps will use it? (aside of Celery)</li><li>• Determine the level of HA required: Sentinel vs Cluster</li></ul>
<b>WSGI</b>	<ul style="list-style-type: none"><li>• uWSGI by default</li><li>• Follow general recommendation, don't overcomplicate it</li></ul>

<b>Reverse Proxy</b>	<ul style="list-style-type: none"><li>• Certificates generation and install</li><li>• Static files and other features needed (ACL)</li><li>• Determine the level of HA required</li></ul>
<b>Authentication</b>	<ul style="list-style-type: none"><li>• Determine source: local, LDAP vs SSO (ideal)</li><li>• Custom RBAC is required? criteria?</li></ul>
<b>Secrets</b>	<ul style="list-style-type: none"><li>• Determine the required integration</li><li>• Never store a secret in the DB</li></ul>
<b>Documentation</b>	<ul style="list-style-type: none"><li>• Content should help to maintain, operate, and update Nautobot</li><li>• Accessibility (web server or readme)</li></ul>
<b>Tuning</b>	<ul style="list-style-type: none"><li>• DEBUG disabled!</li><li>• Celery task limits and Nautobot settings (PAGINATE, RETENTION, etc.)</li><li>• Caching in Reverse Proxy*</li></ul>
<b>Observability</b>	<ul style="list-style-type: none"><li>• Determine tool to collect metrics</li><li>• Determine tool to export logs</li><li>• How direct app debugging happens</li></ul>

>>>network.toCode()

# Deployment Environments

# >>> Deployment methods

## VM : Python / pip install

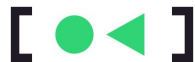
- Systemd
- Python “runserver”



HashiCorp  
**Nomad**

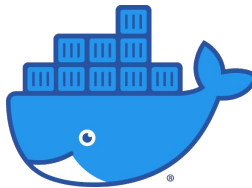


**RED HAT**  
OPENSIFT  
Container Platform



## VM : Container

- Docker

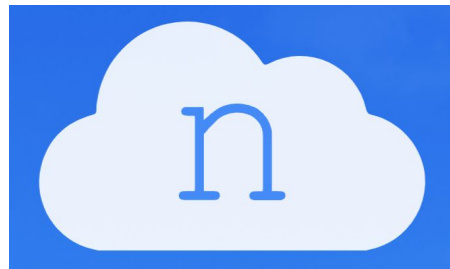


## Cloud container platforms

- Amazon ECS/Fargate
- Kubernetes (Amazon EKS, Google GKS, etc.)
- Openshift
- Nomad

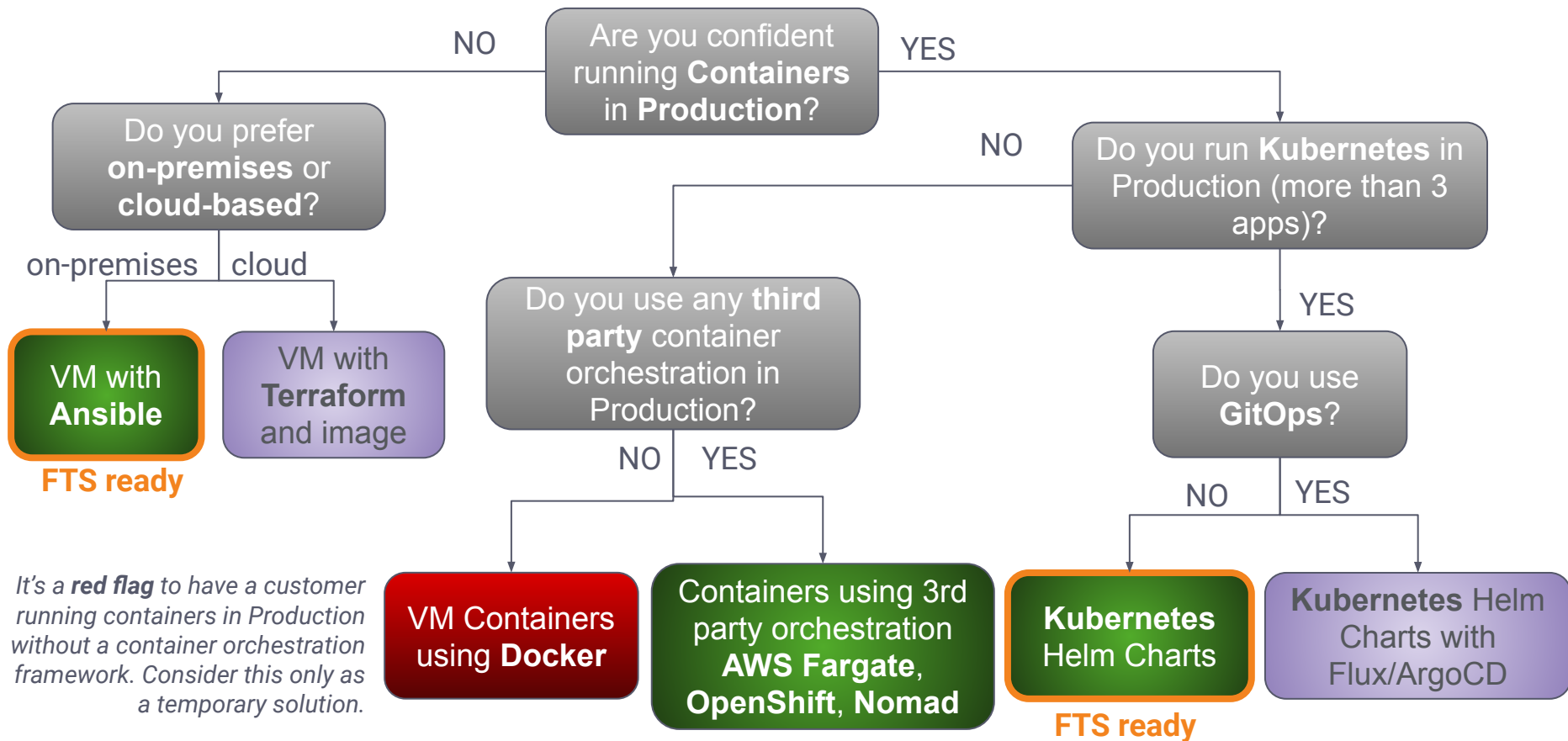
## Nautobot Cloud

Not a problem for us!



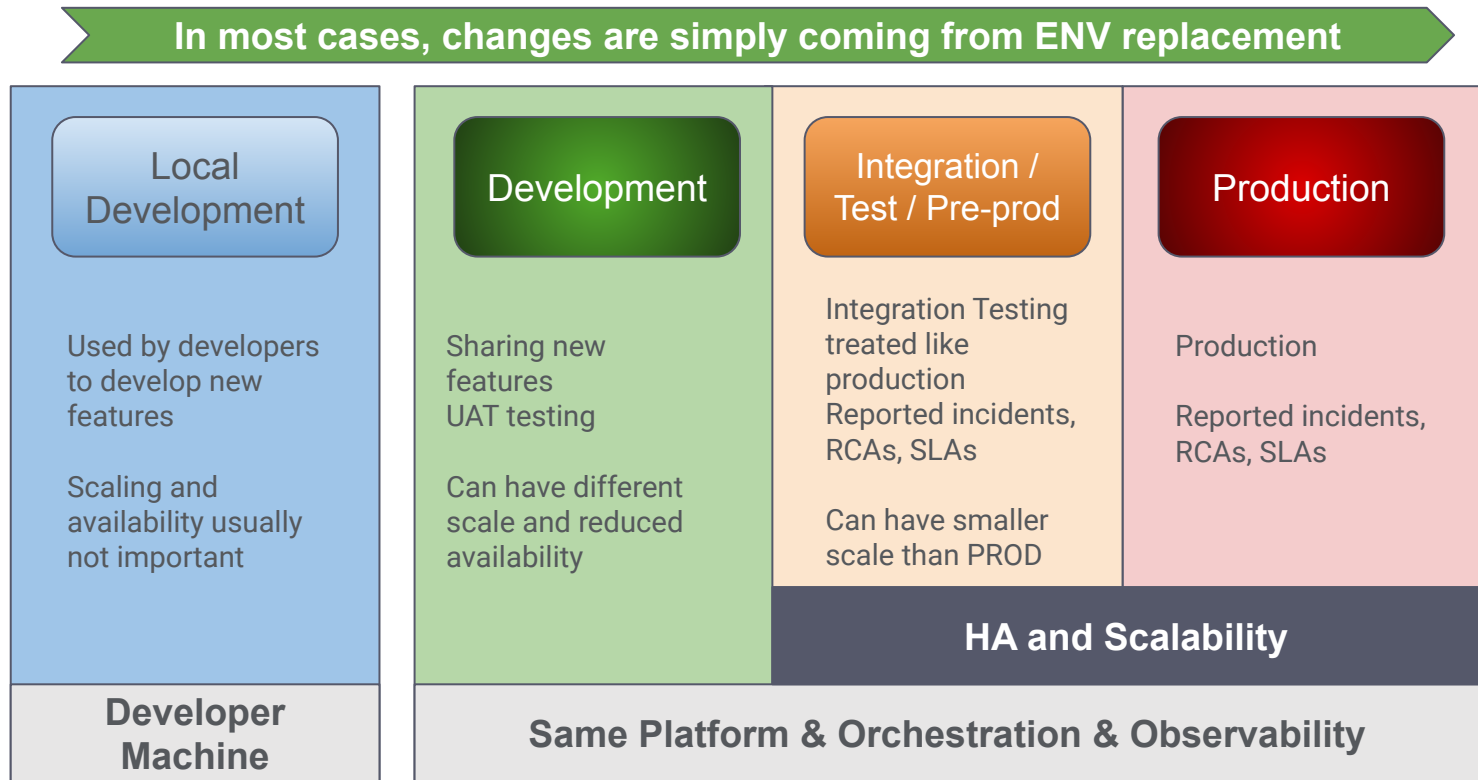
If supported by the customer, go with  
orchestrated container solutions

# >>> Choosing Deployment Platform & Method



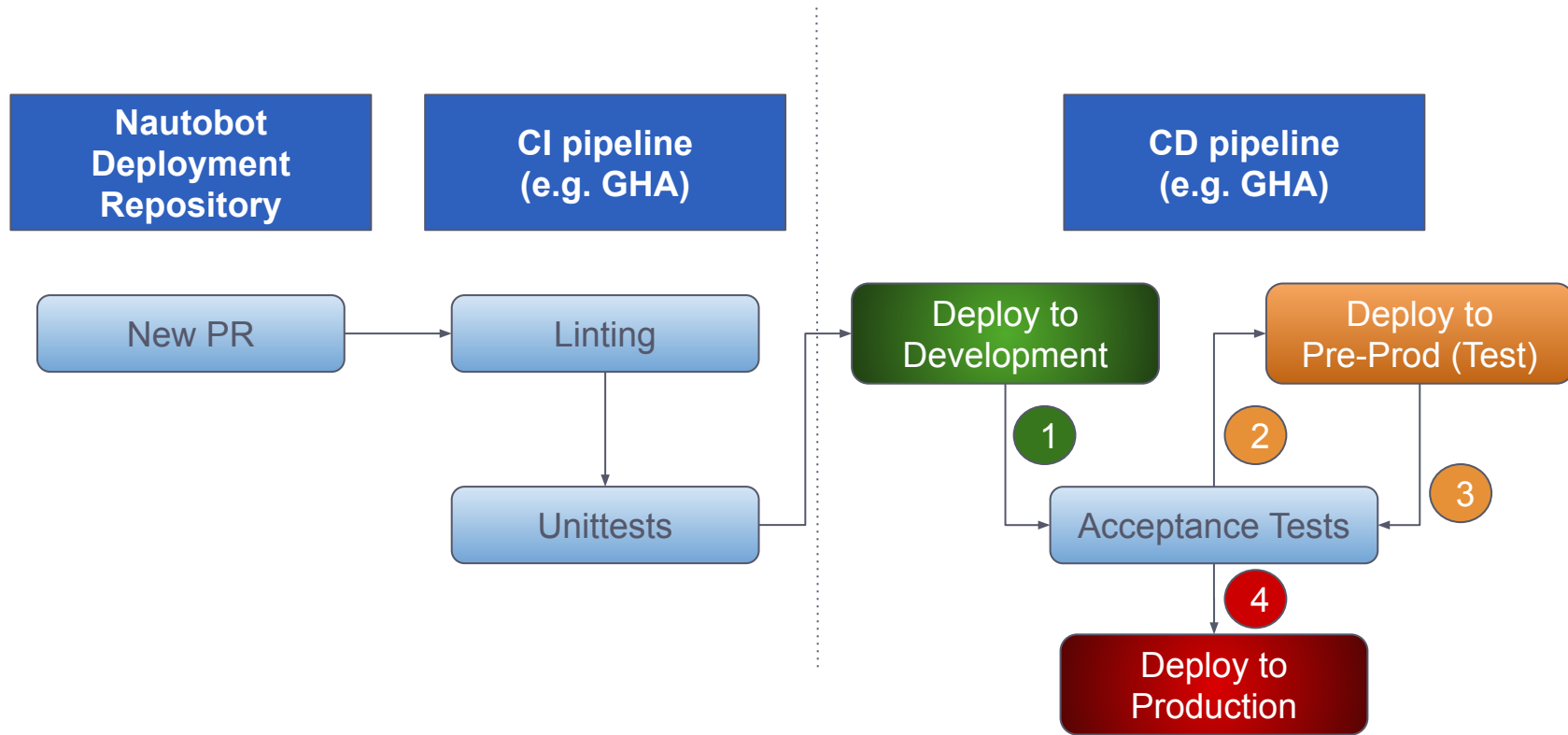
*It's a **red flag** to have a customer running containers in Production without a container orchestration framework. Consider this only as a temporary solution.*

## >>> Deployment environments





# >>> Continuous Integration & Deployment






>>>network.toCode()




# Hands-on Deployment Environments

## >>> Hands-on plan

### **Local Development Environment**

-  Create a local Docker development environment
-  Install Nautobot Apps and custom Jobs
-  Run Continuous Integration in GitHub Actions for every PR

### **Production Nautobot Environment - Ansible**






-  3 VMs running in a Cloud Provider
-  Use internal Ansible playbooks to deploy Nautobot
-  Integrate Ansible deployment tasks in the Nautobot project

# >>> Local Development Environment

Using Docker-compose

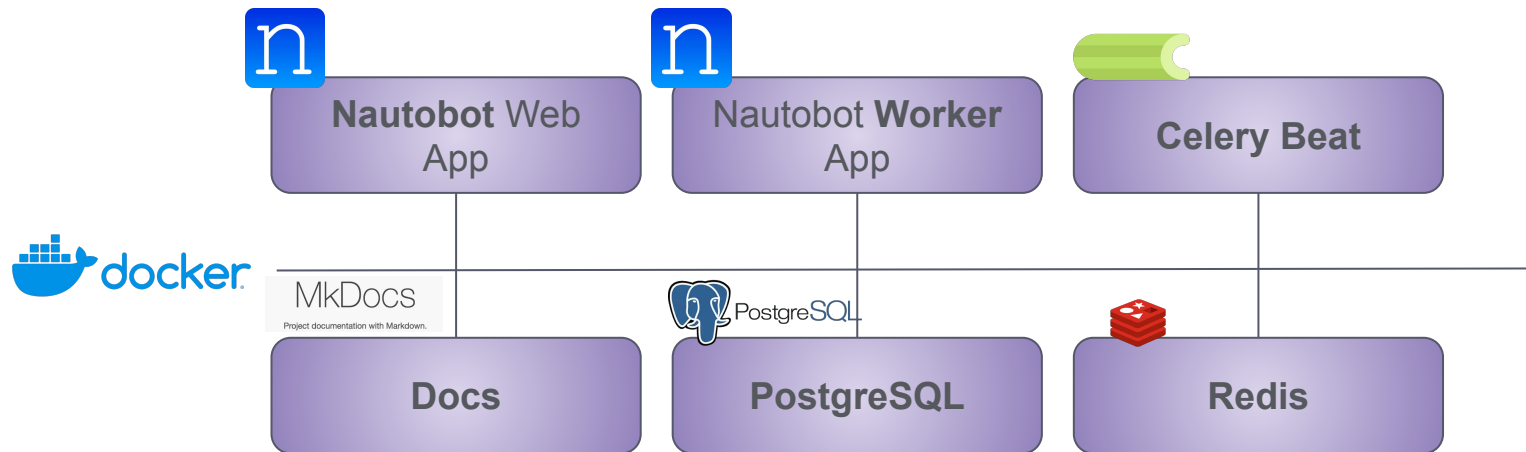
<https://github.com/networktoencode-llc/nautobot-deployment-training>

## Local Development Environment

-  Create a local Docker development environment
  -  You should be able to run and debug Nautobot locally
-  Install Nautobot Apps and create custom Jobs
-  Run Continuous Integration in GitHub Actions for every PR
  -  With passing tests!



## >>> Local Development Environment



**WHY?** It's an environment to help development Nautobot components (e.g. jobs, configuration) integrated with some common components used in production. It's not necessary to be 100% the same, but as close as it could be.

It comes with auto-reload function, and volume sharing to easy getting access to code changes within the containers without rebuilding them (worker process requires restart)



## >>> NTC Cookiecutter

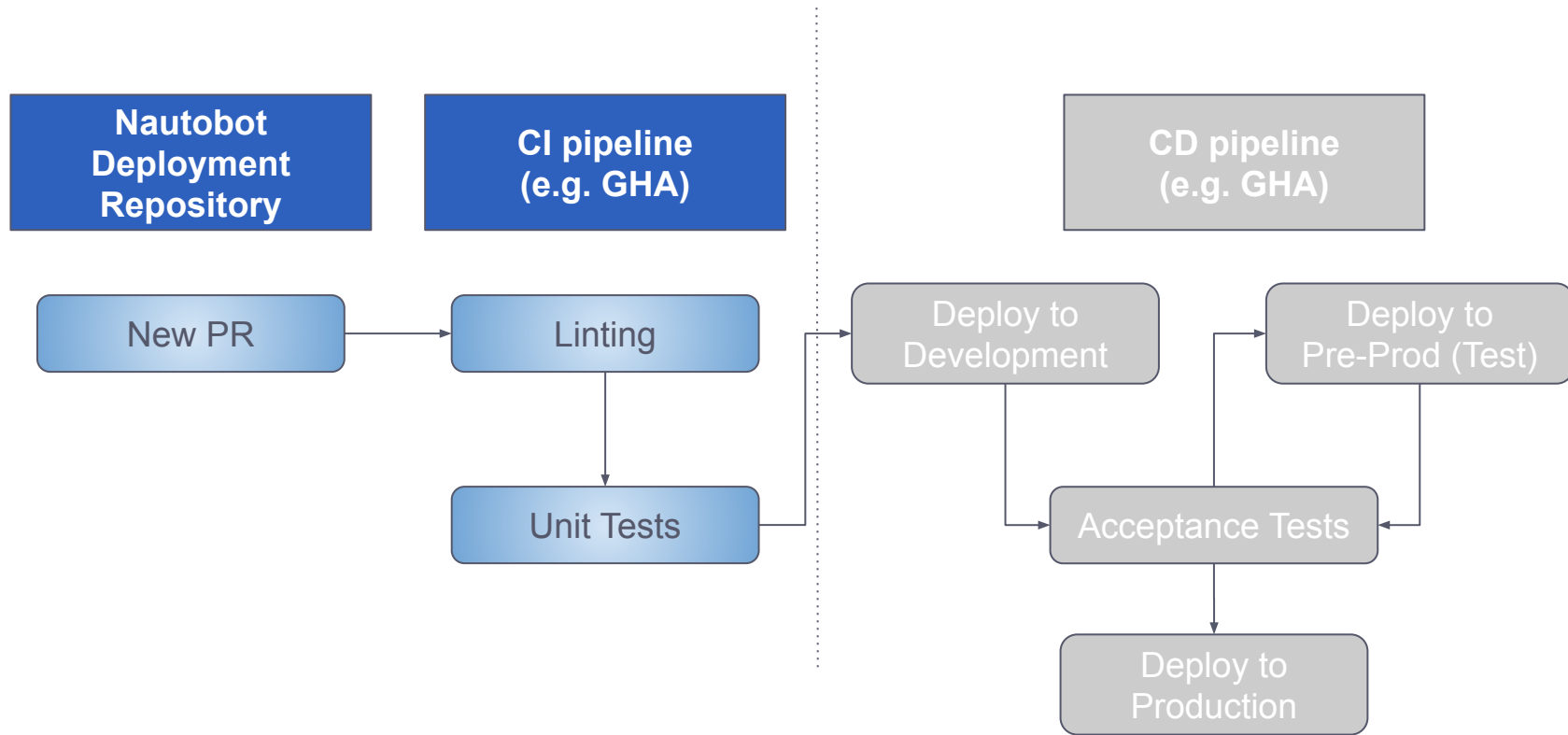
- It contains all the **best practices** built up for several years of internal NTC contributions
  - Led by **#dev-standards** group
  - Supports different project types, such as Ansible, Nautobot Plugins, Python, etc. One of them, **Nautobot Deployment**
- Strong dependencies:
  - Python project
    - Use **Poetry** (Python dependencies management)
    - Use **Invoke** (alternative to Makefile)

Cookiecutter repository: <https://github.com/networktocode-llc/cookiecutter-ntc>

# >>> Installing Applications

- Nautobot is a Django application that can “include” other Django apps and modules
  - There are public Nautobot plugins (most of them under <https://github.com/nautobot> organization, and maintained by NTC)
  - In some engagements, you will use private applications. Proper credentials, and dependency management have to be updated.
- There are TWO steps to use a Nautobot Application extension:
  - Install in the Python environment (using Pip or Poetry)
  - Register the Nautobot Applications in the PLUGINS list in nautobot-config.py (analogous to Django settings.py)
    - Update the Nautobot Application custom config in the PLUGINS\_CONFIG dictionary, in the same file
- Common Nautobot Applications
  - [Nautobot Secrets Providers](#): offers integration of secrets with external secrets managers (e.g. HashiCorp Vault, AWS Secrets Manager, etc.)
  - [Nautobot Capacity Metrics](#): offers extra application metrics to better understand Nautobot operations (observability it's a critical support for operations)

# >>> Continuous Integration



# >>> Production Nautobot Environment




*Running as Linux services*

<https://github.com/networktocode-llc/nautobot-deployment-training>

<https://github.com/networktocode-llc/nautobot-deployment-ansible>

## >>> Hands-on plan

### **Production Nautobot Environment - Ansible**

-  You get 3 VMs running in a Cloud Provider (with your credentials)
-  Use internal Ansible playbooks to deploy Nautobot following a specific design
-  ~~Integrate Ansible deployment tasks in the Nautobot project~~

# >>> Ansible Deployment

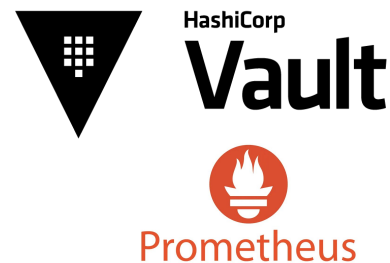
## Supported Distributions



## Managed Services



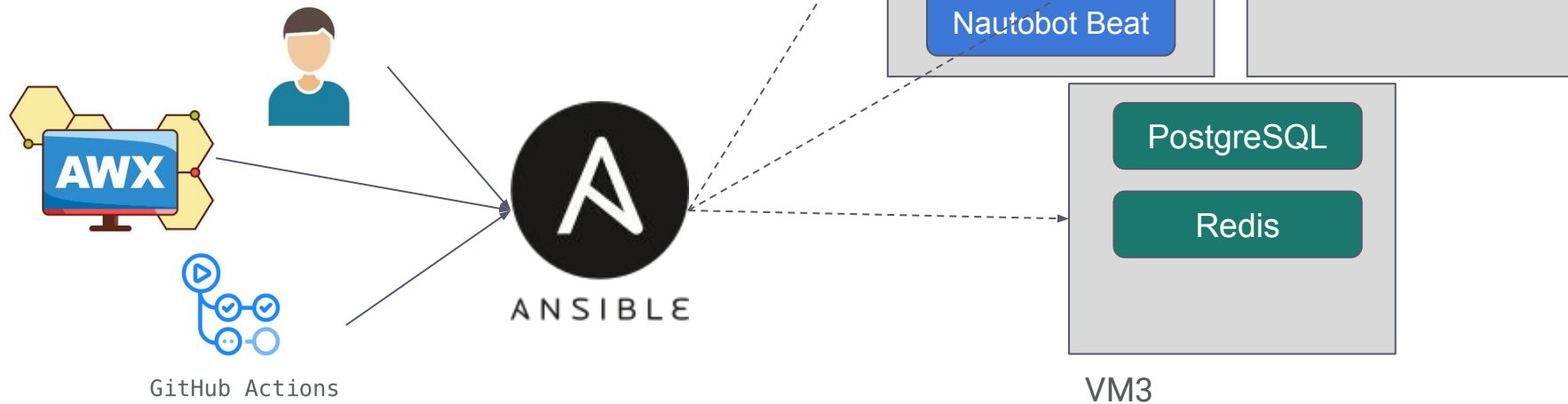
## Coming services...



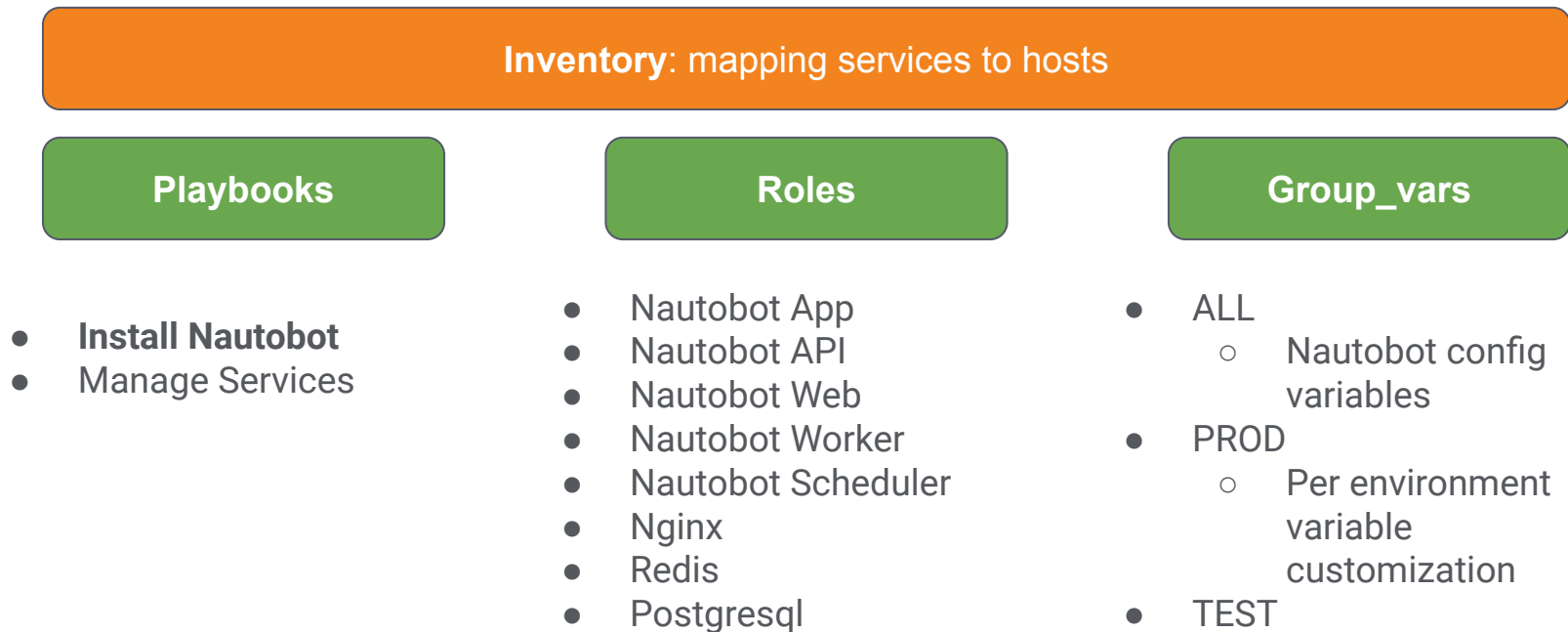


## >>> Running Nautobot as Linux services

- VMs used to host Nautobot stack components: web, worker, beat + NGINX
- PostgreSQL Database and Redis on a VM (simulating external services)
- VMs are provisioned and configured by an **opinionated** ansible playbook

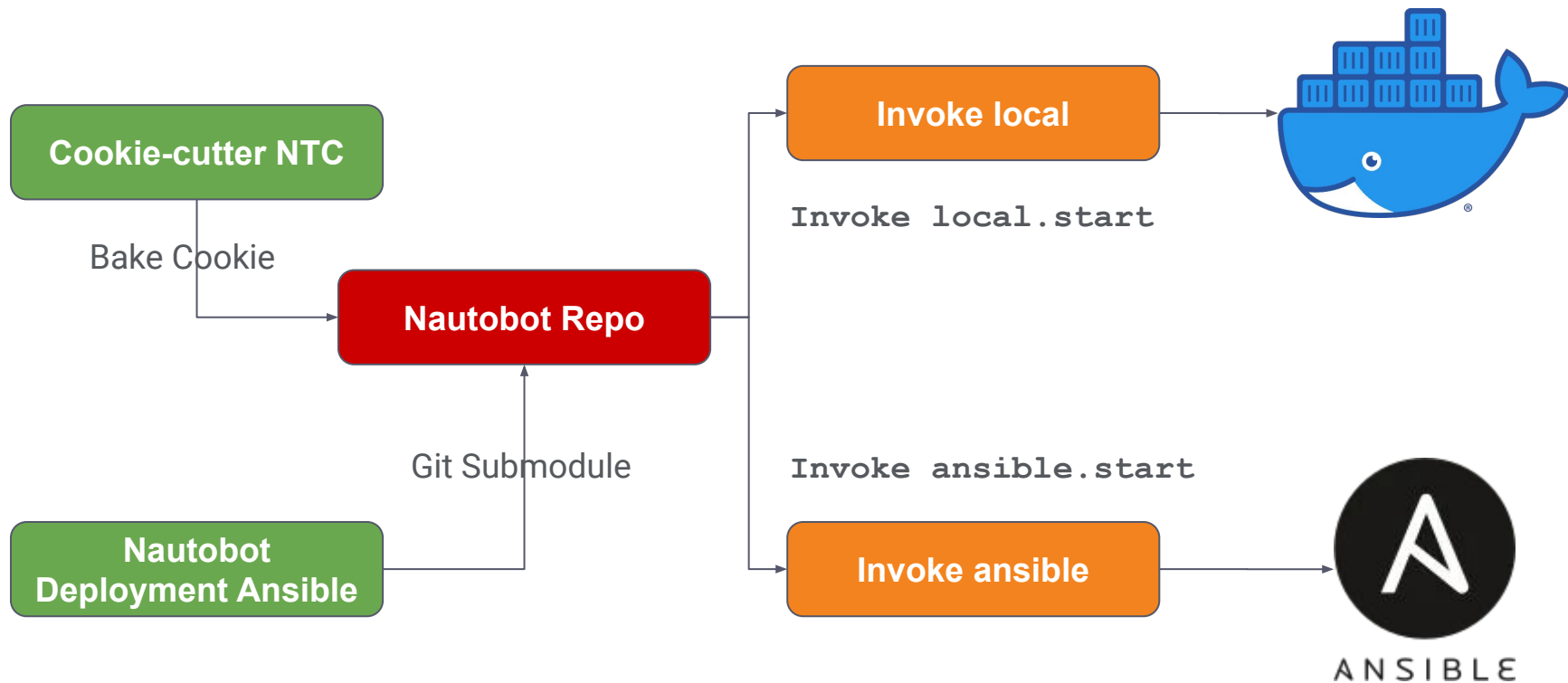


## >>> Ansible Repository Structure



***Configuration file templates are here!!***

## >>> Deploying from a Continuous Deployment pipeline



## >>> Resource Sizing

	Nautobot Web	Nautobot Worker	Nautobot Scheduler	DB	Redis
CPU	4 cores	4 cores	1 core	2 cores	2 cores
Memory	16G	8G	2G	8G	10G
Disk	20G	2G	2G	20G	20G

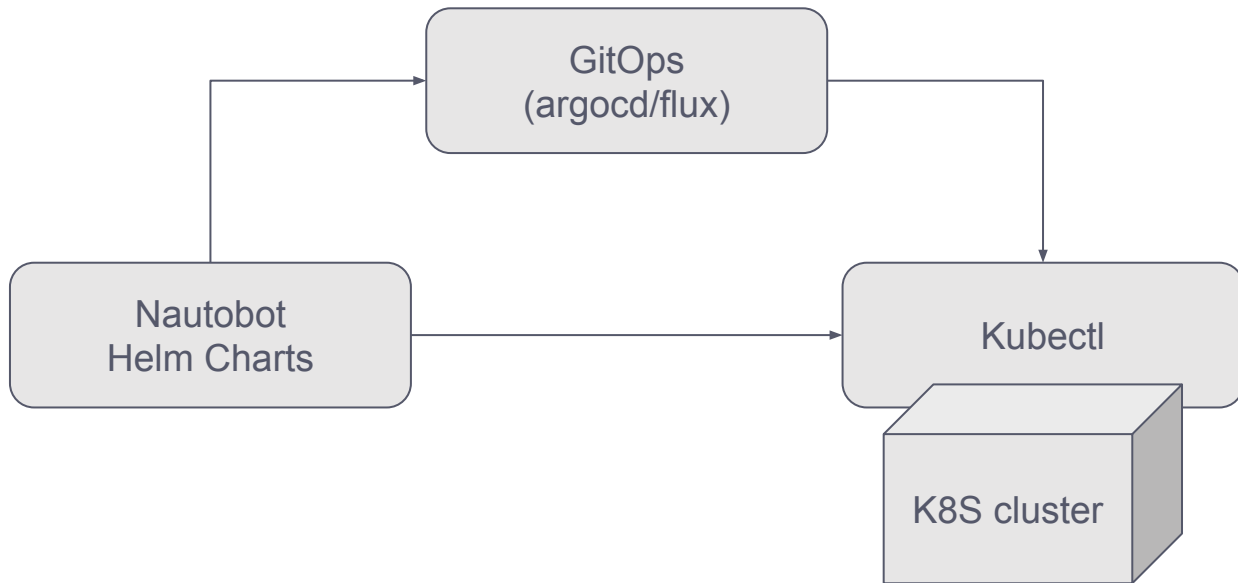
These are just reference numbers!  
Depending on the observation of the  
actual use cases, adjust as required.

**NOT READY YET!**

# >>> Production Nautobot Environment

*Running as containers in Kubernetes*

## >>> K8S workflow

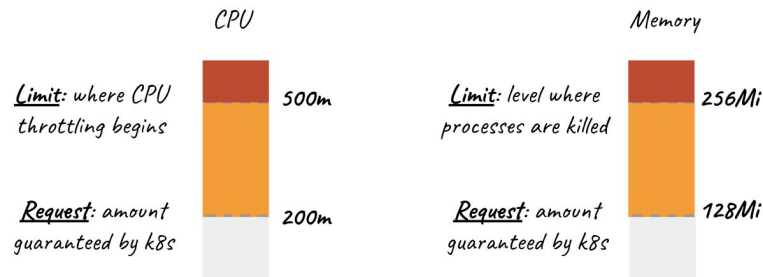


It is recommended that Kubernetes **provisioning is managed as code**  
(too many things to track!)



## >>> Recommended Settings

These are just reference numbers!  
Depending on the observation of the  
actual use cases, adjust as required.



	Nautobot Web	Nautobot Worker	Nautobot Scheduler
<b>Request CPU</b>	100 millicores = 0.1 CPU	400m	5m
<b>Limit CPU</b>	2000m	2000m	2000m
<b>Request Memory</b>	1280M	1G	256M
<b>Limit Memory</b>	8704M	6656M	6656M
<b>Autoscaling</b>	Min:2 Max:5 Average CPU > 150	Min:2 Max:5 Average CPU > 50	NO

Resource Management for Pods and Containers:

<https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/>

>>>network.toCode()

Thanks