Next-generation cluster management architecture

Towards Continuous Uptime

Paul Peltz Jr. and Lowell Wofford

11/11/2018

Operated by Los Alamos National Security, LLC for the U.S. Department of Energy's NNSA

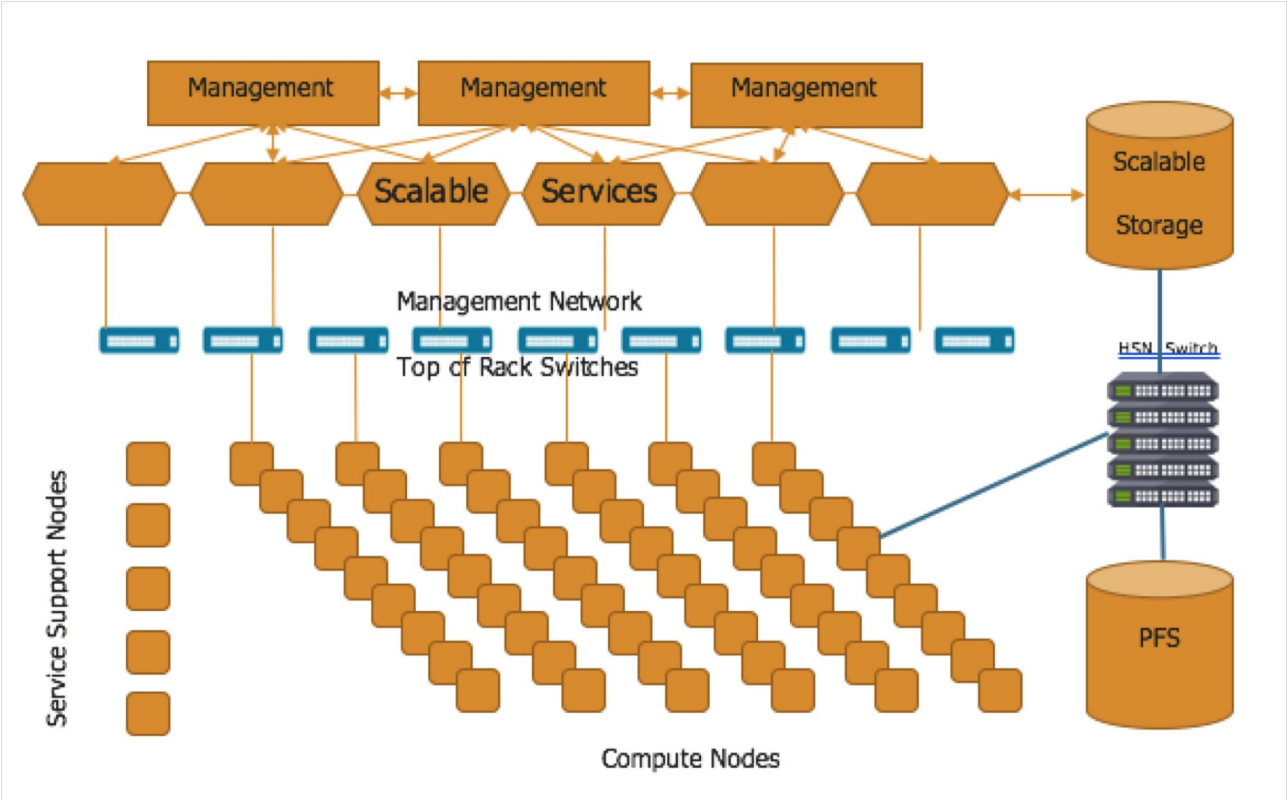
Current Issues

- **Current Cluster Management Tools**
  - Centralized
  - Monolithic
  - Proprietary/Outdated
  - Not Configuration Management and Orchestration capable
- **Downtime**
  - Binary Network Topologies (Up/Down)
  - Updates/Patches
- **Schedulers**
  - Lack integration into the cluster management
  - Inference model only

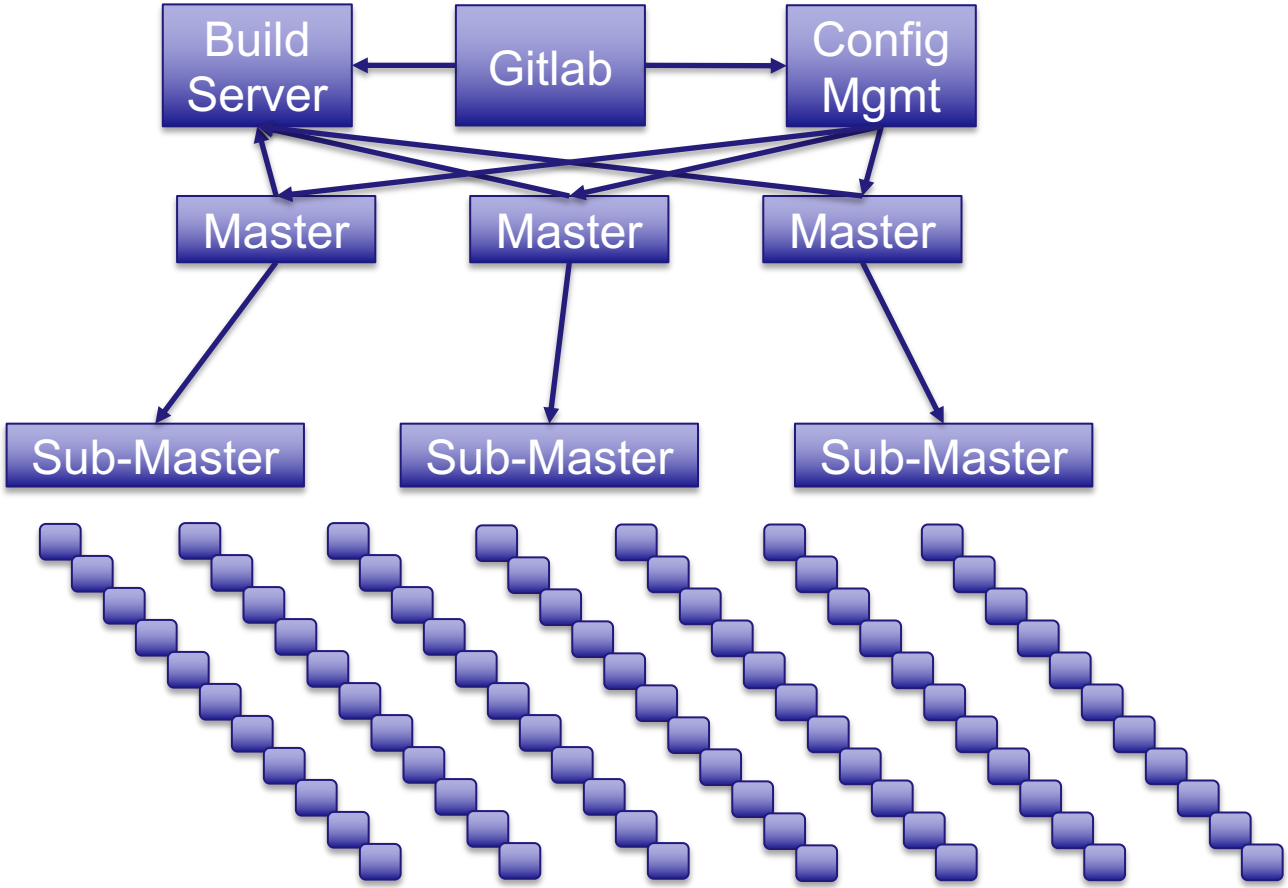
How do we solve these problems?

- **Current Cluster Management Tools**
  - Complete redesign necessary
  - Distributed State Engine, Modular
- **Downtime**
  - System should never go down
    - Degraded modes
    - Distributed Services
    - Rolling Upgrades
  - CI/CD of Patches and Changes
- **Schedulers**
  - Ties directly into the distributed state engine

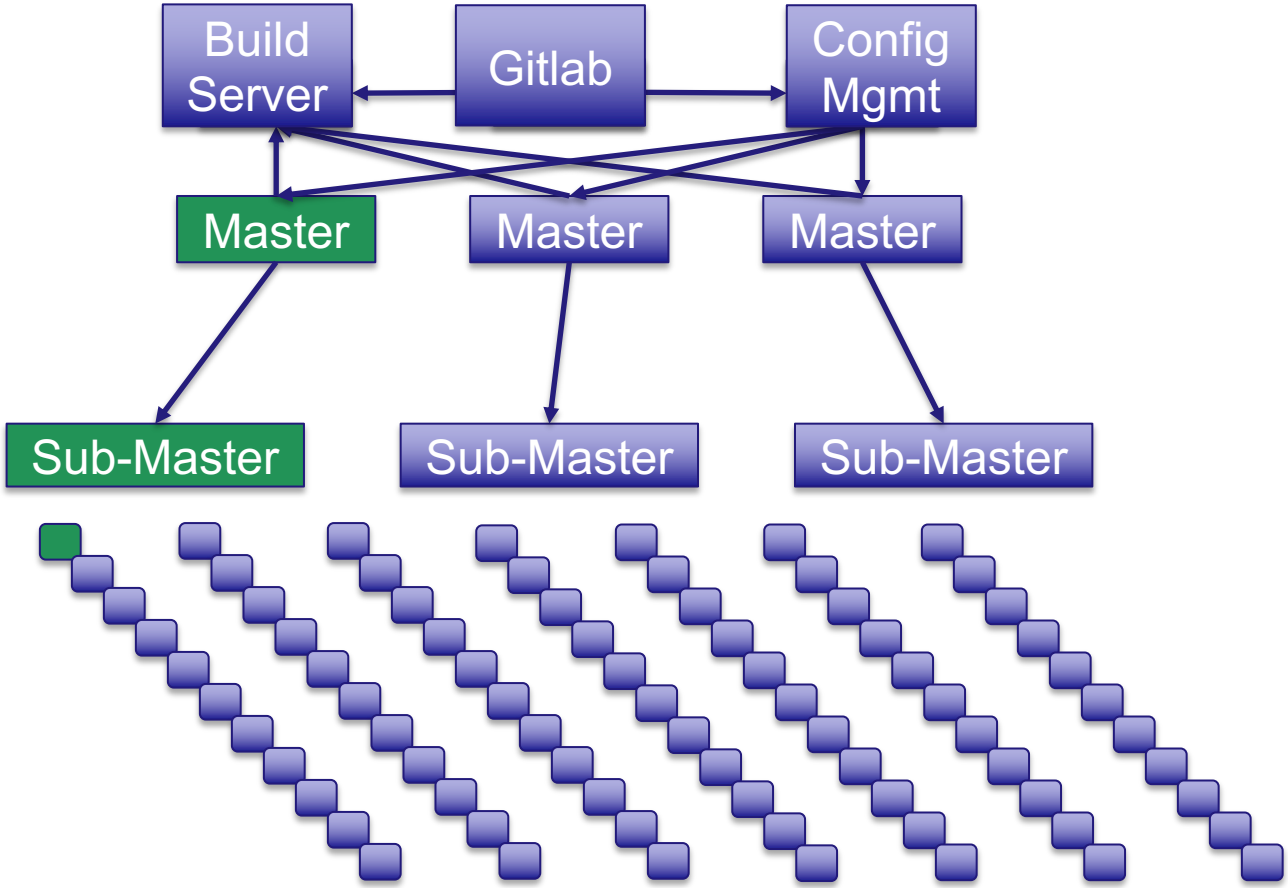
Redesign of HPC Architecture



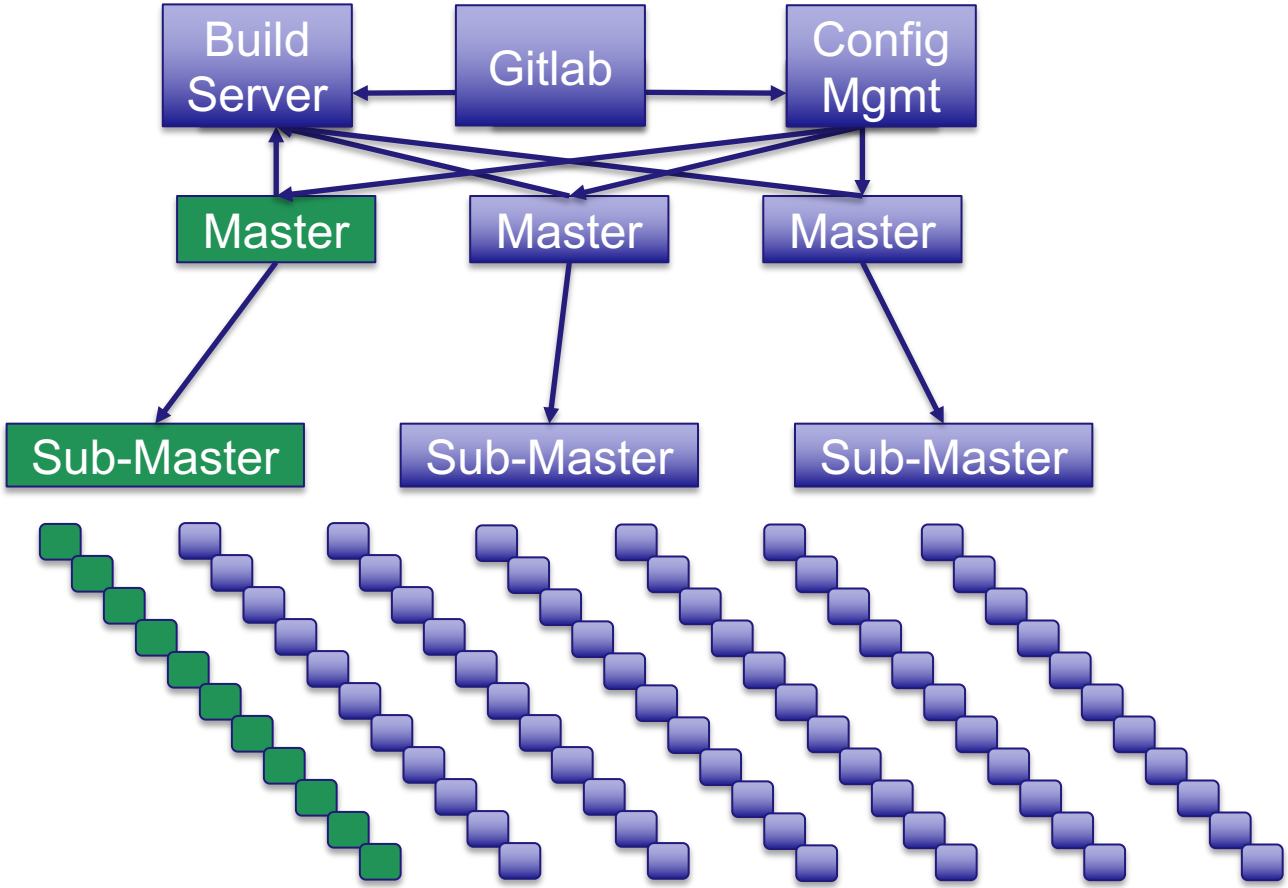
How we will test and deploy changes in the future



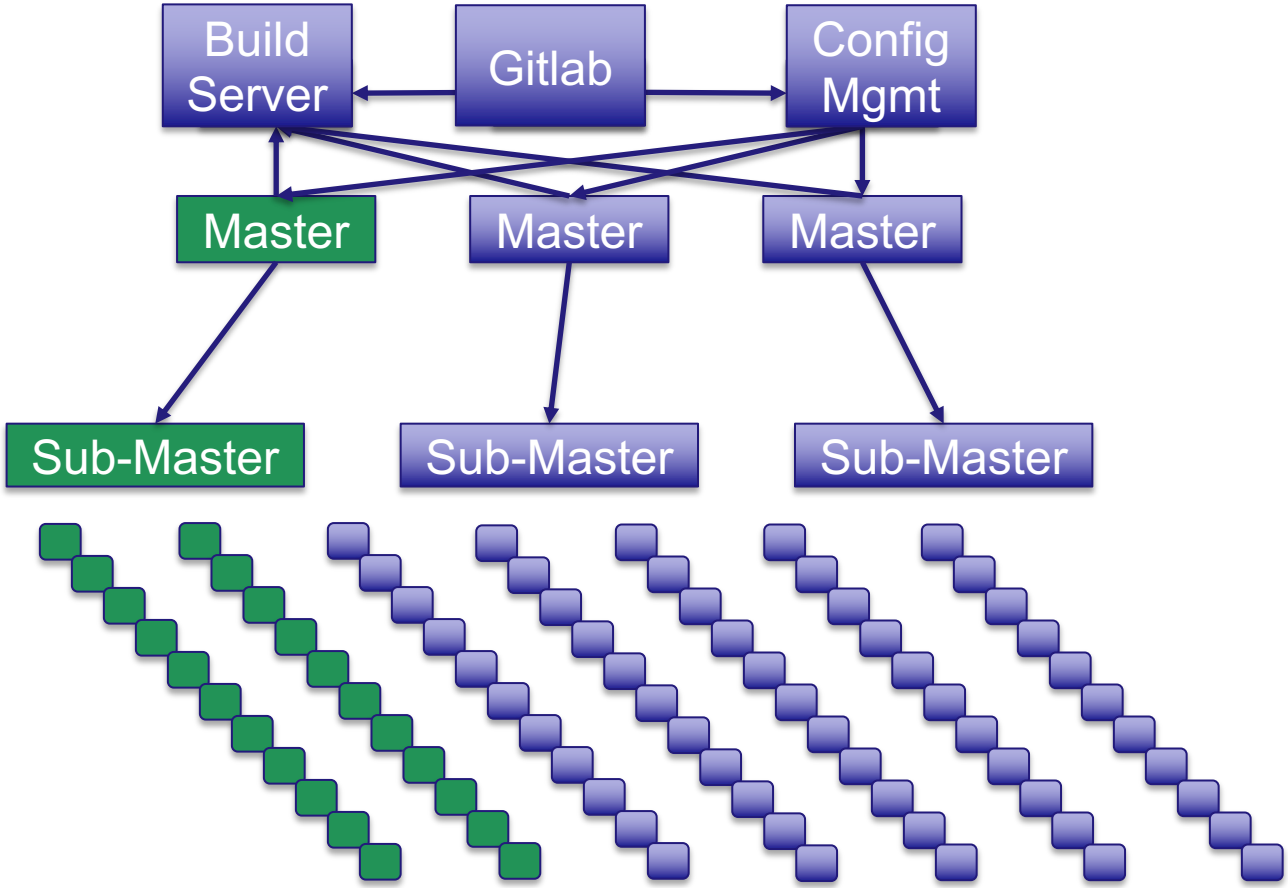
Stage 1 – Functional Testing



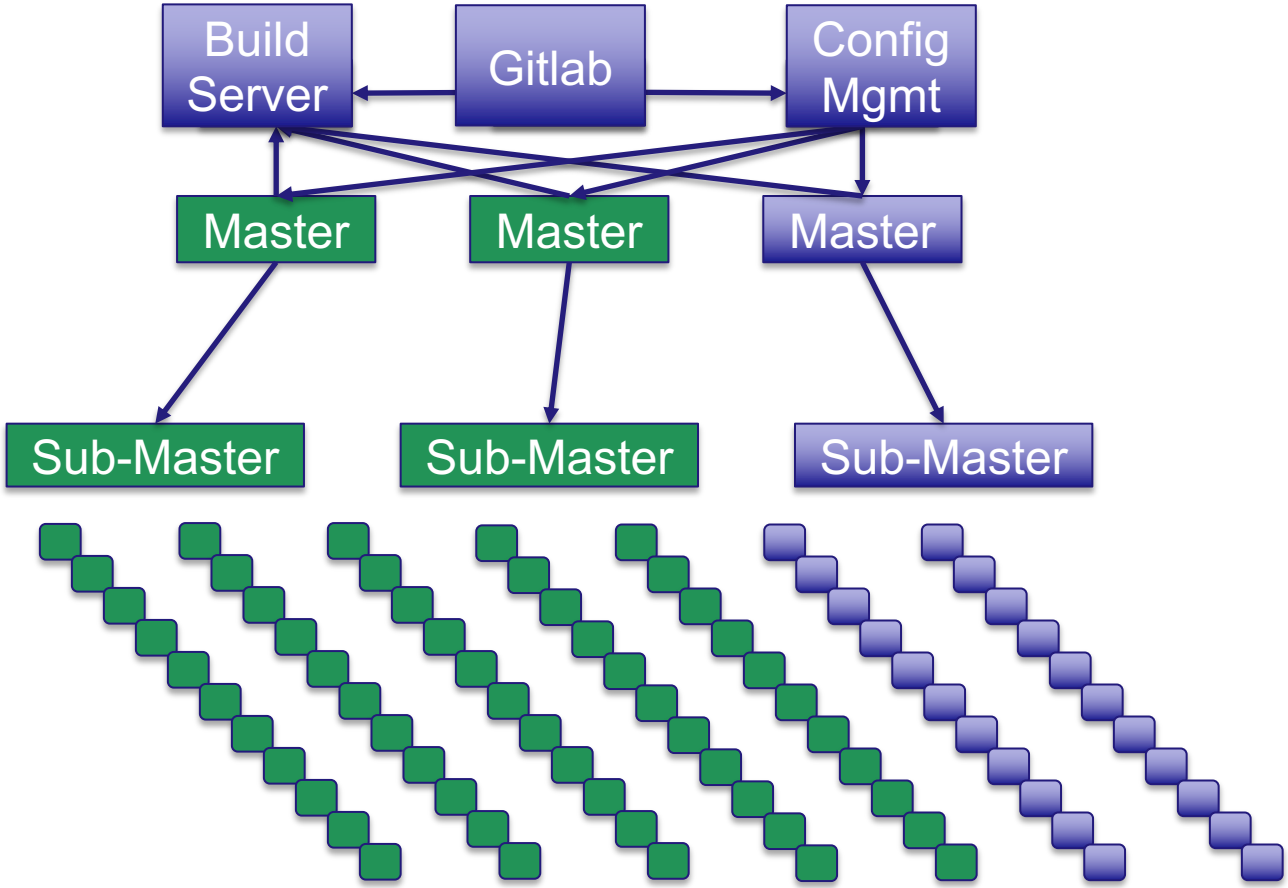
Stage 2 – Small Scale Test



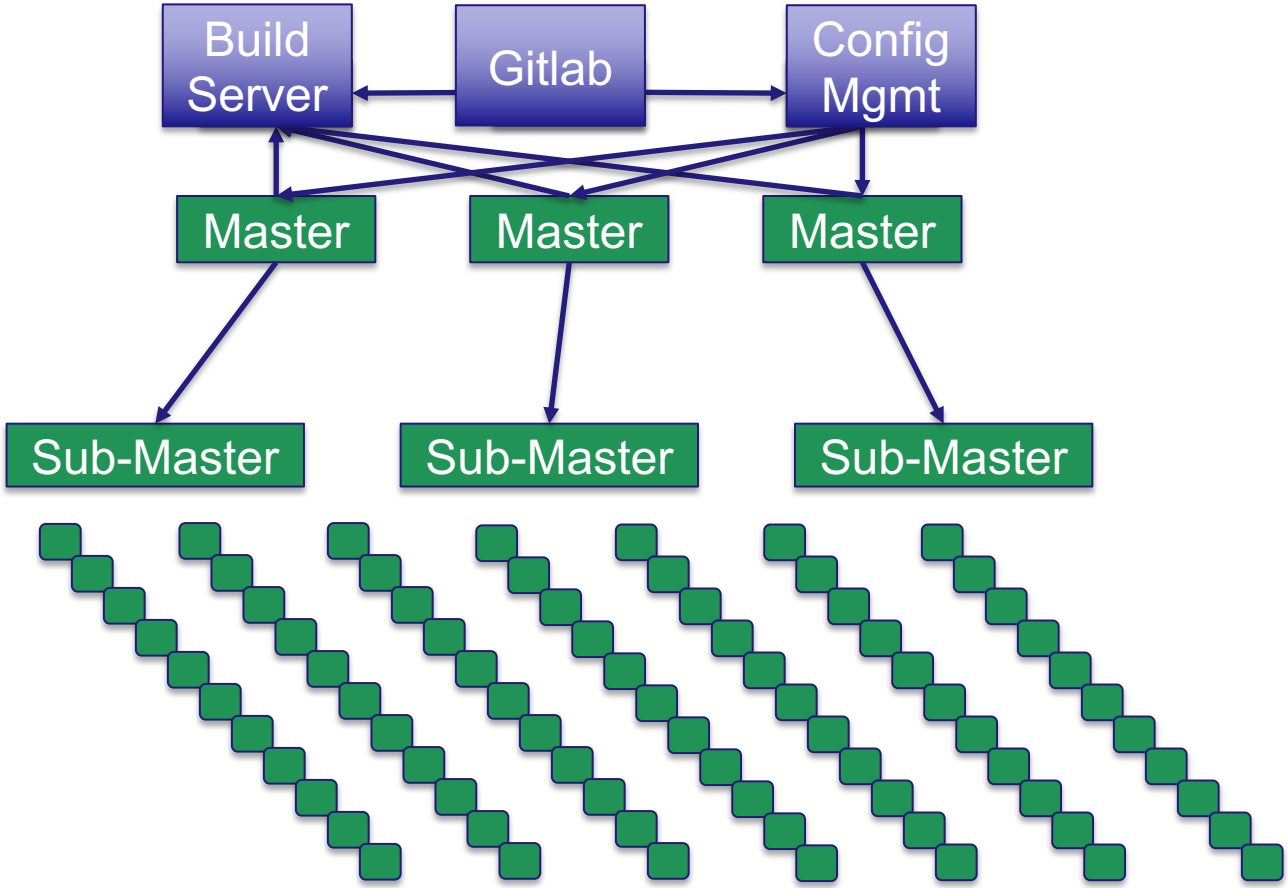
Stage 3 Scale Testing



Stage 4 – Update Additional Master/Sub-Master



Final Stage – Continue to reboot nodes into new images once jobs complete



Sounds great, how do we do this?

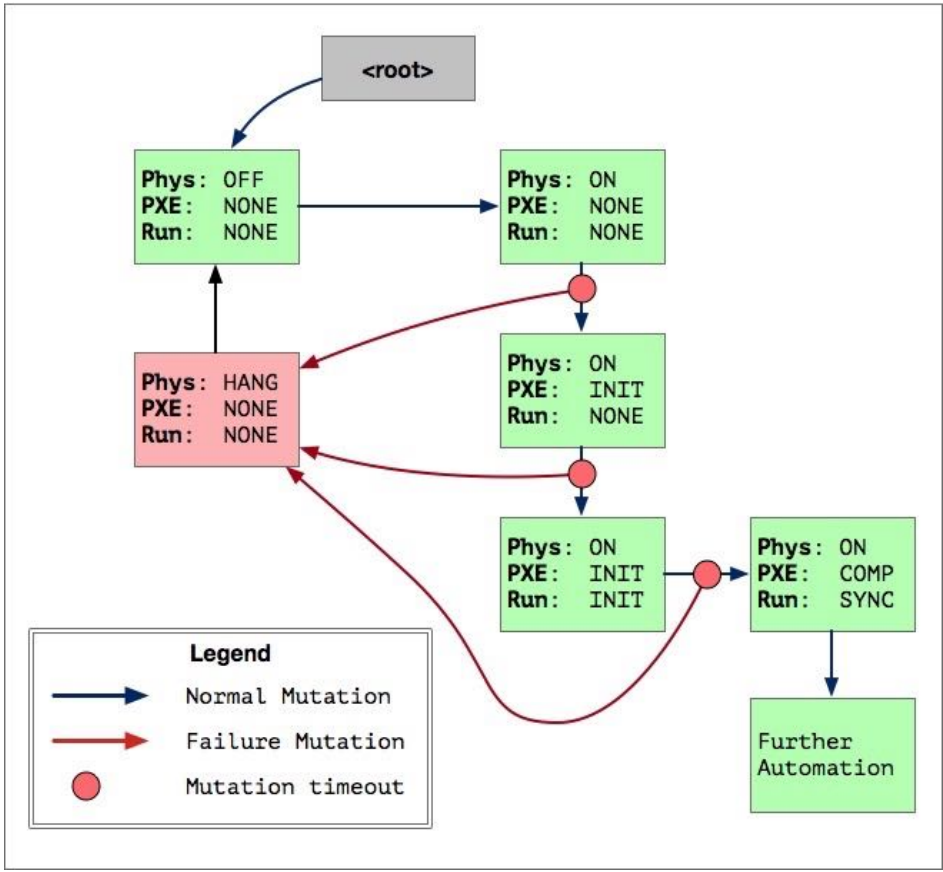
- **Service decentralization**
  - HA is not acceptable
  - VMs/Micro Services
  - Migrate services
- **Distributed State Engine**
  - The cluster manager must be able to track state, transitions, and final states of the system including the service nodes.
- **Backwards compatible APIs**
  - Need at least one version compatibility between service APIs.
- **Scheduler Integration**
  - Must be able to schedule the transition of release states on the system
  - Handle A/B configuration if there are compatibility issues

Automated and Continuous Testing!

- **Continuous Testing**
  - Testing first on development systems (TDS)
  - Automated testing for functionality and performance during each stage of deployment
  - Once release is approved that branch is released into production
  - Configuration management notifies state engine to begin rolling through upgrade
  - Testing is performed for each step
  - If a test exception occurs, human intervention will be required to resolve the issue
  - Roll forward or roll back support

Kraken is a State Engine for System Automation

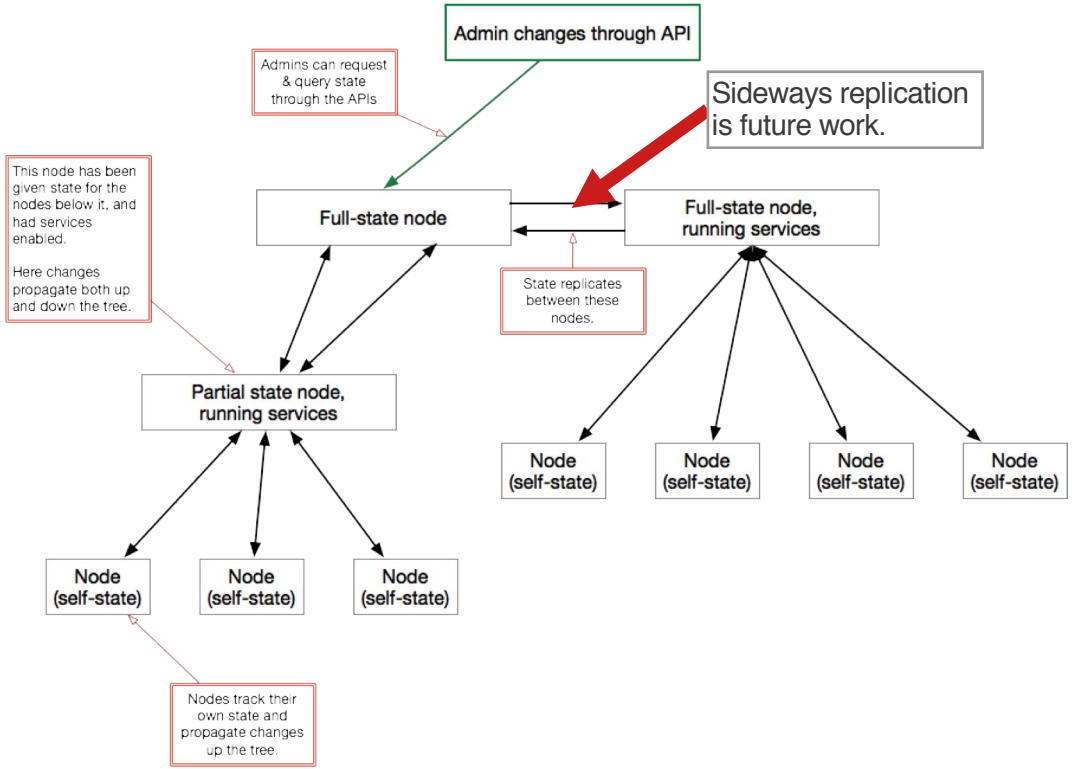
- **It can:** boot systems, load images, recover from failures… and more.
- Kraken tracks two kinds of **state**:
  - *What state are we in? (“Discoverable”)*
  - *What state should we be in? (“Configuration”)*
- Kraken knows how to **mutate** from the state you’re in to the state you want:
  - Maintains a directed graph of known state *mutations* (e.g. *power_off -> power_on*)
  - Searches the graph to create a chain of mutations to get from where we are to where we want to be.



Kraken is Distributed

Kraken uses a *protobuf*-based, light-weight datagram protocol to **synchronize** state across the system.

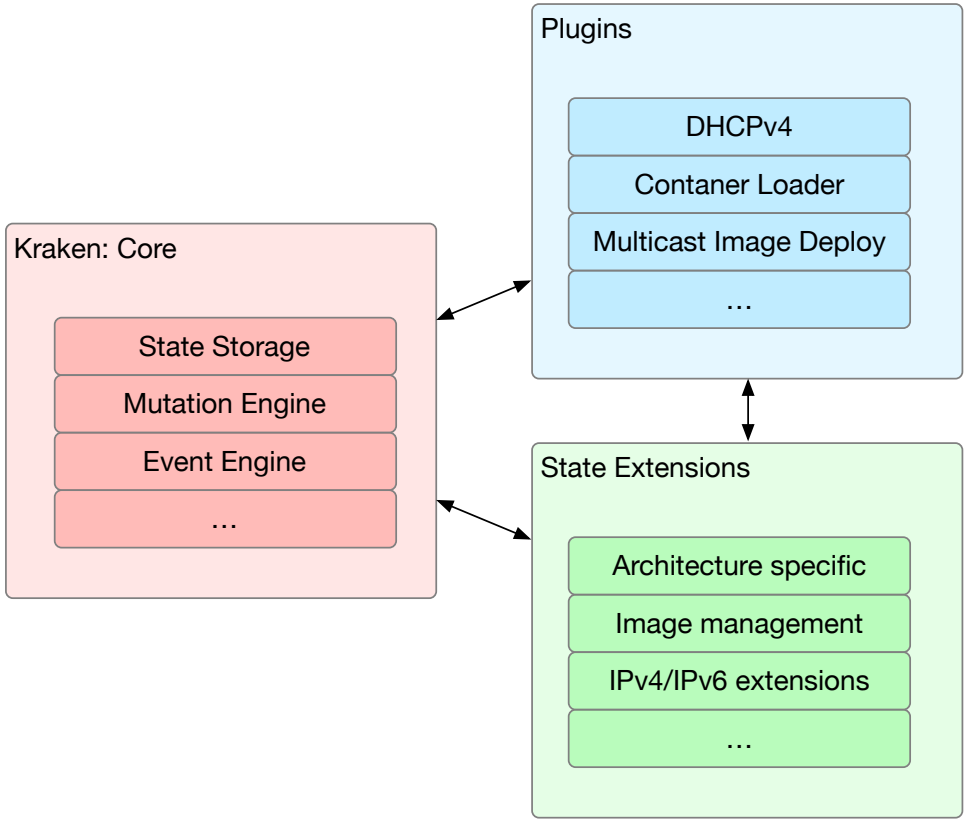
- This uses an **eventual consistency** model to converge on synchronized state.
- Discoverable state migrates **up the tree** from the end-point nodes (computes).
- Configuration state migrates **down the tree** from the master, or “full-state node.”



Kraken is Modular

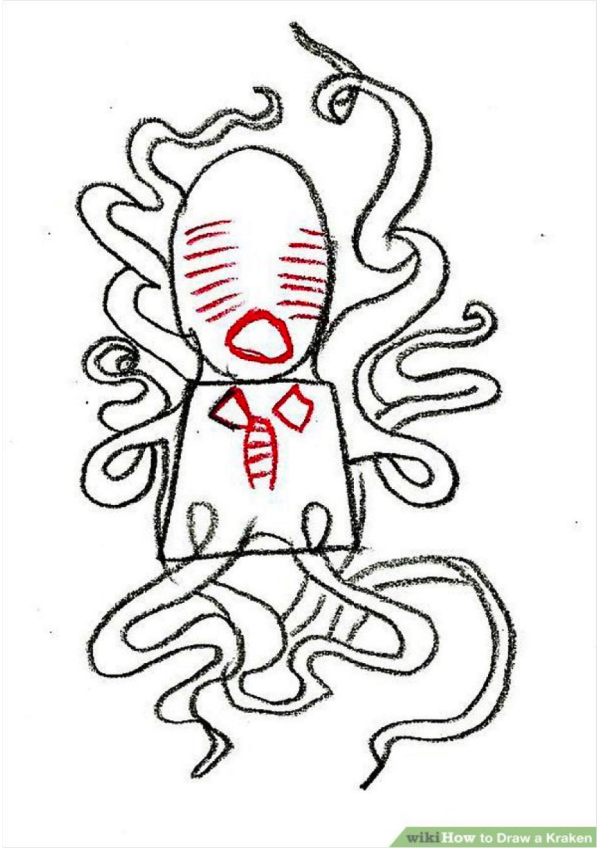
Modules make kraken actually do work.

- The **core** of Kraken:
  - State management
  - Plugin/Extension management
- **Modules** can:
  - Instantiate **microservices**
  - Declare that they can **mutate** state
  - Declare that they can **discover** state
- Extensions add new **variables** to the state



State of the Kraken

- Basic set of functional microservices & plugins
- Can boot multiple architectures
- Uses layered container images in reference implementation
- Open source (BSD-3): https://github.com/hpc/kraken
- Working towards first full release



Thanks!