# Becoming cloud native without starting from scratch

**Marin Jankovski**
Engineering Manager, Distribution and Delivery,
GitLab

# About me

- 2012: Joined GitLab as Ruby on Rails developer
- 2013: GitLab.com deployments and feature development
- 2014: Responsible for software packaging
- 2015: Technical lead of the Build team
- 2017: Engineering Manager of the Distribution team
- 2018: Additionally leading Delivery team

Twitter: **@maxlazio**

GitLab.com: **@marin**

LinkedIn: www.linkedin.com/in/marin-jankovski

# Outline

1. GitLab intro
2. Application architecture
    a. Requirements for Cloud Native architecture
3. Application rewrite
    a. Dealing with stateful application
    b. Choosing the right solution
4. Zero downtime upgrades
    a. Existing process
    b. Creating the operator
5. What we learned

# GitLab is a single application for the entire DevOps lifecycle

## Manage

**Since 2016**
**GitLab added:**

- Cycle Analytics
- DevOps Score
- Audit Management
- Authentication and Authorization

## Plan

**Since 2011**
**GitLab added:**

- Kanban Boards
- Project Management
- Agile Portfolio Management
- Service Desk

## Create

**Since 2011**
**GitLab added:**

- Source Code Management
- Code Review
- Wiki
- Snippets
- Web IDE

## Verify

**Since 2012**
**GitLab added:**

- Continuous Integration (CI)
- Code Quality
- Performance Testing

## Package

**Since 2016**
**GitLab added:**

- Container Registry
- Maven Repository

## Release

**Since 2016**
**GitLab added:**

- Continuous Delivery (CD)
- Release Orchestration
- Pages
- Review Apps
- Incremental Rollout
- Feature Flags

## Configure

**Since 2018**
**GitLab added:**

- Auto DevOps
- Kubernetes Configuration
- ChatOps

## Monitor

**Since 2016**
**GitLab added:**

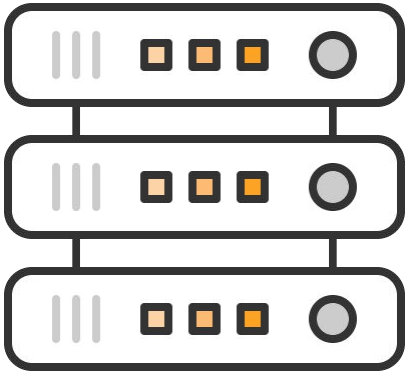
- Metrics
- Logging
- Cluster Monitoring

## Secure

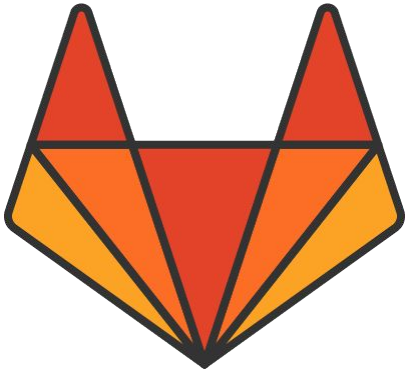
**Since 2017**
**GitLab added:**

- SAST
- DAST
- Dependency Scanning
- Container Scanning
- License Management

# Self-Managed and GitLab.com

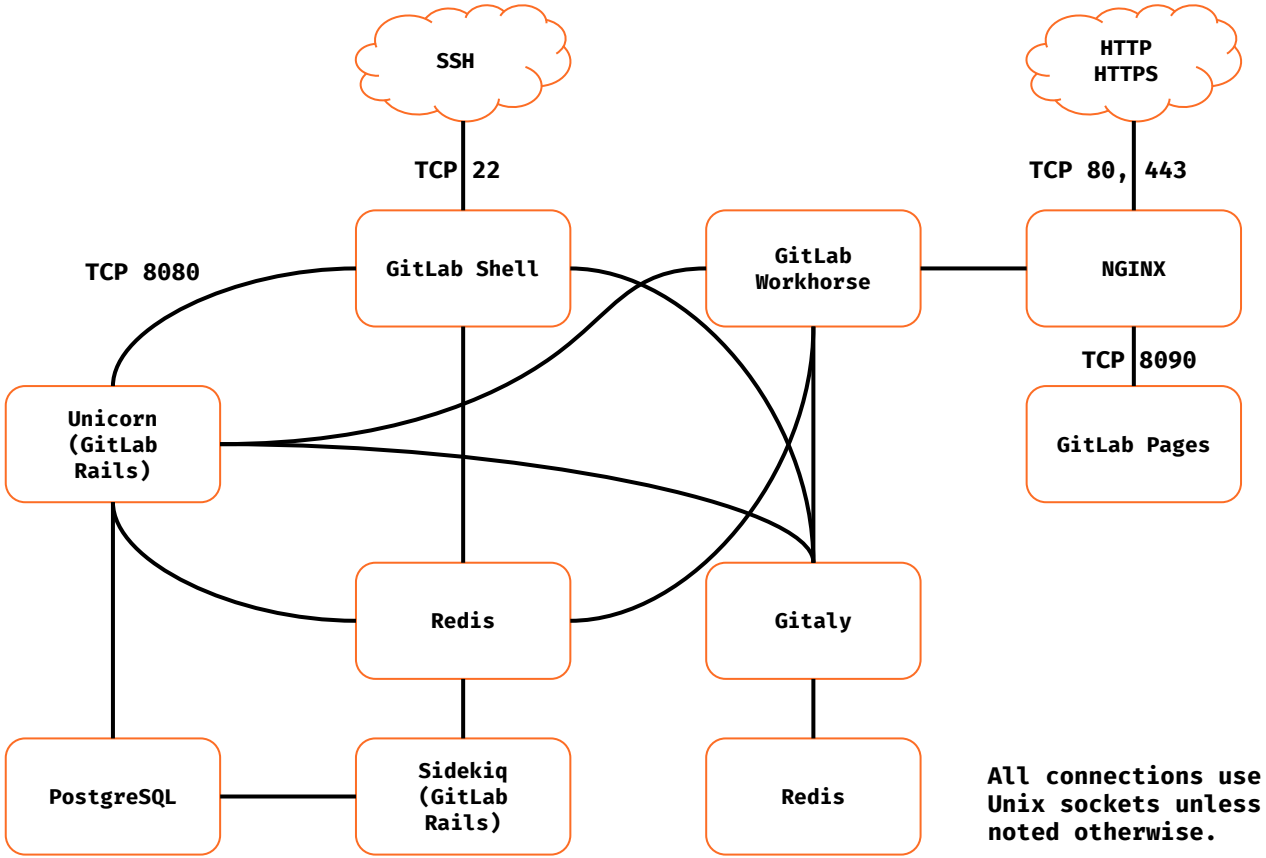


GitLab Self-Managed



GitLab.com

# GitLab Application Architecture

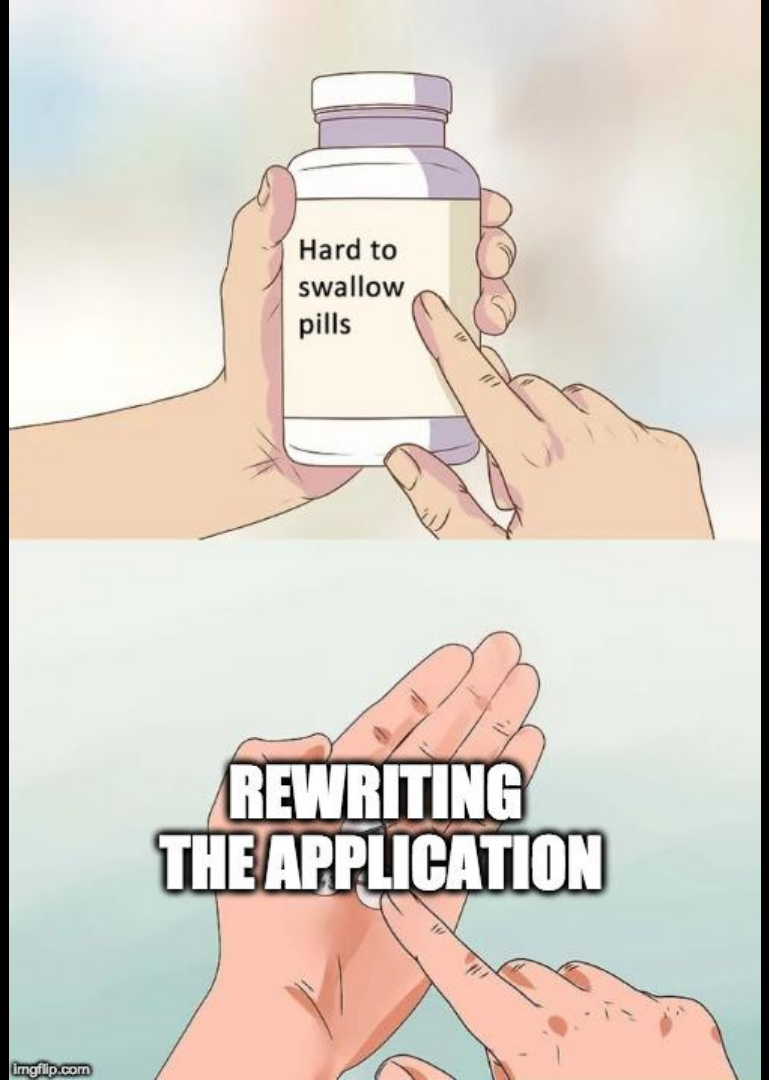


# Requirements

- Cloud Native term
  - Definition in https://github.com/cncf/toc/blob/master/DEFINITION.md
- We required a bit more:
  - Less complex vertical and horizontal scaling
  - Engineering velocity unchanged
  - Cloud platform agnostic
  - GitLab.com and self-managed

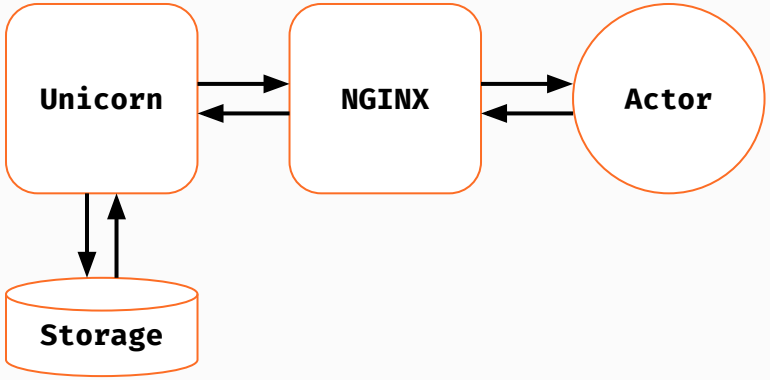
# Examples

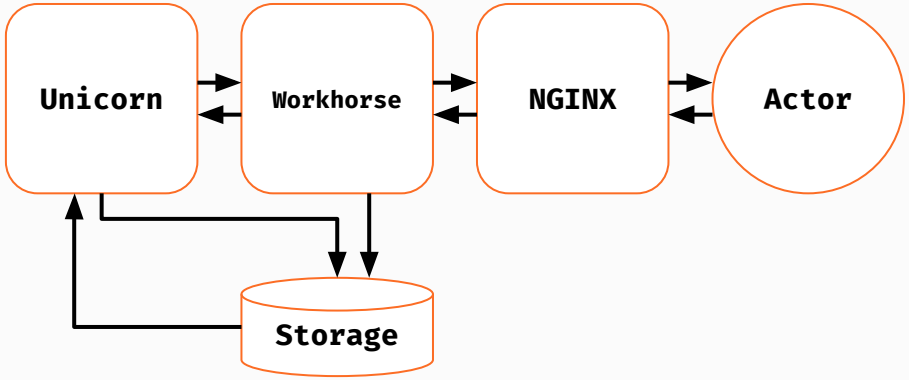
- Less complex scaling + engineering velocity => **Rewriting the application**
- Cloud platform agnostic + suitable for self-managed and SaaS => **Zero downtime upgrades**


Hard to
swallow
pills
REWRITING
THE APPLICATION
imgflip.com

# Serving content to users

- Unicorn
  - Git operations via HTTP(S)
- User uploads + LFS files + CI Artifacts => GitLab Workhorse
- Shared storage between the two components



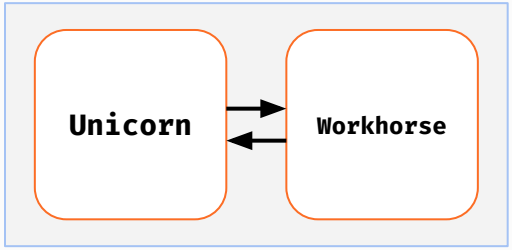



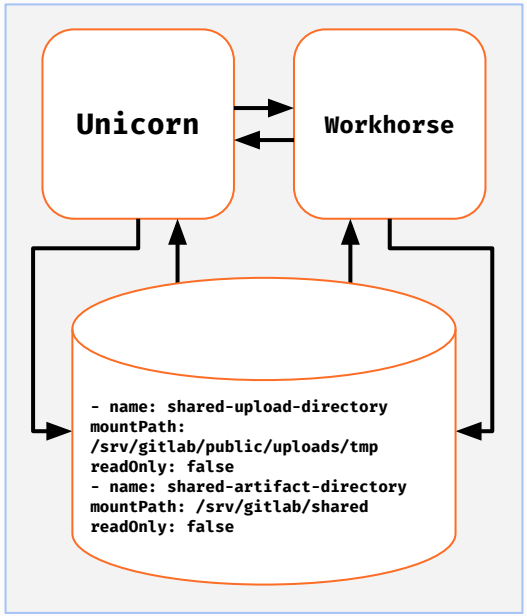
# One process per container

- Horizontal scaling is much simpler
- Image reuse
- Upgrading/patching/rollback application is much simpler as you need to roll one component
- Concern is limited to one process thus positively affecting security

# Choose your destiny

1. Completely separate workhorse and unicorn and put them in their own pods
2. Separate them into their own images but let them share the pod
3. Have the two services in the same image



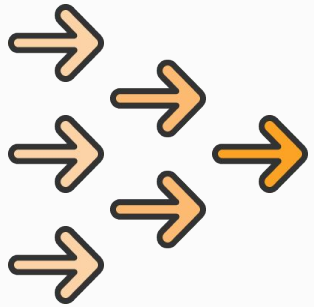

# Choose your destiny

- Complete separation
  - Pro: Right way to do it
  - Con: Possible major application functionality disruption
  - Con: Not clear how much time it would take
- Sharing pod resources
  - Pro: Requires minimal immediate application rewrite
  - Con: Still requires shared storage
- All in one image
  - Pro: Confirmed to be working correctly
  - Pro: Requires only placing the components in one image
  - Con: Still requires shared storage


HAVE YOU TRIED UPGRADING
WITHOUT DOWNTIME?
imgflip.com

# Current process

- Roll out new versions of components without touching existing ones
- Run online database migrations
- Roll down Gitaly service and use new version (if the version changed)
- Restart other services
- Run migrations that can be completed in the background
- Restart 2 services that could be affected by the latest set of migrations



# Kubernetes rolling updates

- Stop routing connections to terminating pods
- Send TERM signal to each container in the pod
- Wait for **`terminationGracePeriodSeconds`**

# Kubernetes rolling updates

- Challenge 1: Ensuring graceful termination
  - What should `terminationGracePeriodSeconds` value be?
- Challenge 2: The rollout order?
  - Gitaly first
  - Database migrations

# Helm

- “Helm Charts helps you define, install, and upgrade even the most complex Kubernetes application.” `(source: helm.sh)`
- **`$ helm install`**
- Challenge 1: Ensuring graceful termination
  - No special functionality to ensure this

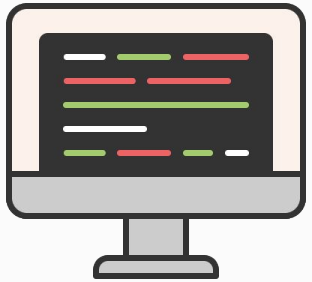
# Helm

- Challenge 2: the rollout order?
  - Gitaly
  - Database migrations
    - Job resource created after deployment

```
var InstallOrder SortOrder = []string{
        "Namespace",
        "ResourceQuota",
        "LimitRange",
        "PodSecurityPolicy",
        "Secret",
        "ConfigMap",
        "StorageClass",
        "PersistentVolume",
        "PersistentVolumeClaim",
        "ServiceAccount",
        "CustomResourceDefinition",
        "ClusterRole",
        "ClusterRoleBinding",
        "Role",
        "RoleBinding",
        "Service",
        "DaemonSet",
        "Pod",
        "ReplicationController",
        "ReplicaSet",
        "Deployment",
        "StatefulSet",
        "Job",
        "CronJob",
        "Ingress",
        "APIService",
}
```

# Write our own operator?

- “An Operator is an application-specific controller that extends the Kubernetes API to create, configure, and manage instances of complex stateful applications on behalf of a Kubernetes user.” (source: https://coreos.com/blog/introducing-operators.html)
- We can use it everywhere!
  - Automated backup/restore would fit great in the operator concept!
  - Database initialisation and database migrations?
  - Predictable upgrade process?!

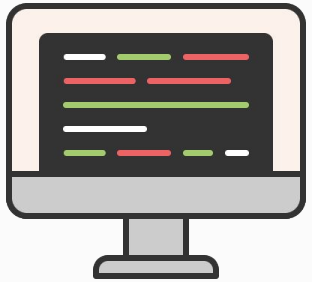


# Attempt 1: CustomController

- Initial example from https://github.com/trstringer/k8s-controller-core-resource
- Watch for changes in deployment
- Roll out in order
- Problems
  - Tracking all events?
  - Rolling pod restart?
  - Handing ConfigMap?
  - Rolling secrets?

# Attempt 2: Use an existing tool

- Operator-sdk: https://github.com/operator-framework/operator-sdk
- Kubebuilder: https://github.com/kubernetes-sigs/kubebuilder
  - Canonical project structure
  - Generating code to register custom types with controller manager
  - Generating CRD definitions
  - Generating RBAC rules for the controller



# Helm + Operator

- **`$ helm install`**
  - Operator is installed using a Helm hook
  - Operator now controls the rollout process
- **`$ helm upgrade`**
  - Helm updates all versions at the same time
  - Operator can’t control the process
- Solution - share the responsibility:
  - Helm installs the new resources
  - Operator pauses the workloads to prevent Kubernetes from rolling resources
  - Operator controls the rest of the rollout

# Helm + Operator

- Helm Chart deployment template:

```
spec:
  {{- if .Values.global.operator.enabled }}
  paused: true
  {{- end }}
```

- Operator StatefulSet rolling strategy:

```
if pause {
   statefulSet.Spec.UpdateStrategy.RollingUpdate.Partition =
statefulSet.Spec.Replicas
} else {
   statefulSet.Spec.UpdateStrategy.RollingUpdate.Partition =
int32Pointer(0)
}
```

# Helm + Operator

- Operator Job rolling strategy

```
if pause {
    parallelism = int32Pointer(0)
}
```

- Operator DaemonSet rolling strategy

```
if pause {
    maxUnavailable = &intstr.IntOrString{IntVal: 0}
}
```

# Final hurdle

```
$ helm upgrade --install <release-name> .
 --set global.operator.enabled=true
 --set global.operator.bootstrap=true

$ helm upgrade <release-name> .
 --set global.operator.enabled=true
 --set global.operator.bootstrap=false
```

# What we learned

# Lesson 1

- Kubernetes moves (too) fast
- Kubernetes is very powerful

# Lesson 2

- Great for organising configuration
- Working with Tiller is challenging
- Error handling  doesn’t appear to be always reliable
  - `Error: Upgrade failed: “gitlab” has no deployed releases”`

# Lesson 3

- Slitting the application
- Hiring experts at the right time
- Starting early

Thank you!