A (somehow) reasoned journey to GitOps

Consistency and Reproducibility
in a Distributed System and Company

TABLE OF CONTENTS

1

Intro

The Operator as the basic Kubernetes pattern

2

Helm

Kubernetes package management for reusability, consistency and reproducibility

3

GitOps

Enforcing workflow in operations & standarizing on system state evolution

4

Flux & beyond

Automating and standarizing CD pipelines for Kubernetes with the right mix of centralization & self-service

GROUP BEHAVIORS

We are good people and create happiness

Positive mindset Humility Open-mindedness

We are the change

Courage Agility How yes

We win as a team

Diversity Commitment Collaborate

We are passionate and result-oriented

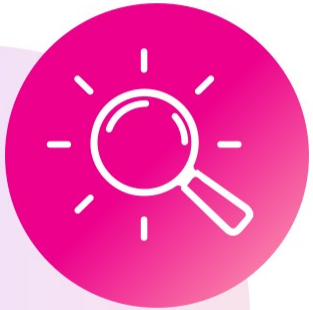
Ownership Pragmatism Hard work

We are transparent and respectful

Exemplarity Trust Integrity

We are customer and brand-centric

Impact Loyalty Engagement



We are passionate and result-oriented

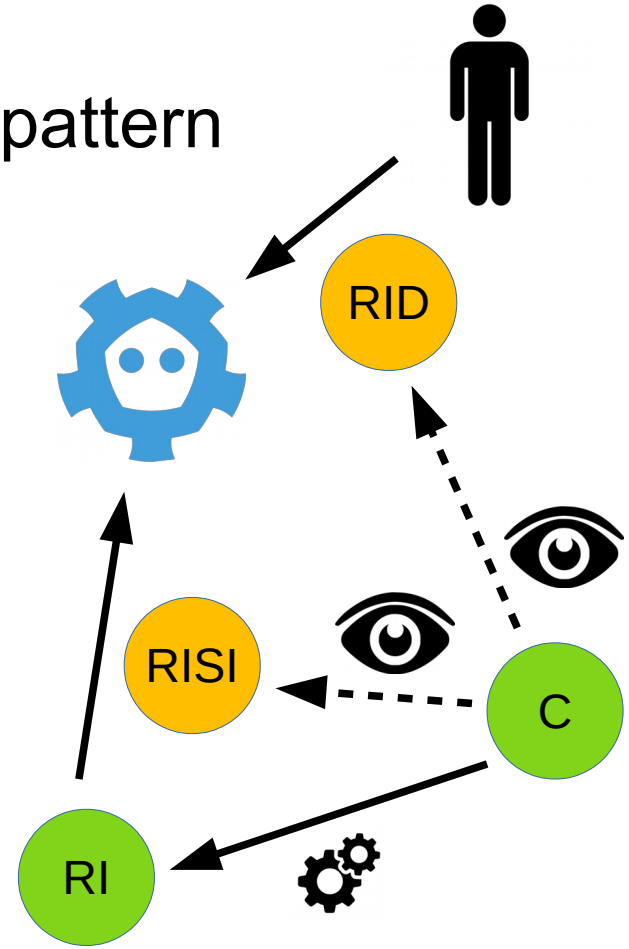
Ownership Pragmatism Hard work

Intro

The Operator as the basic Kubernetes pattern

Operator: The basic Kubernetes pattern

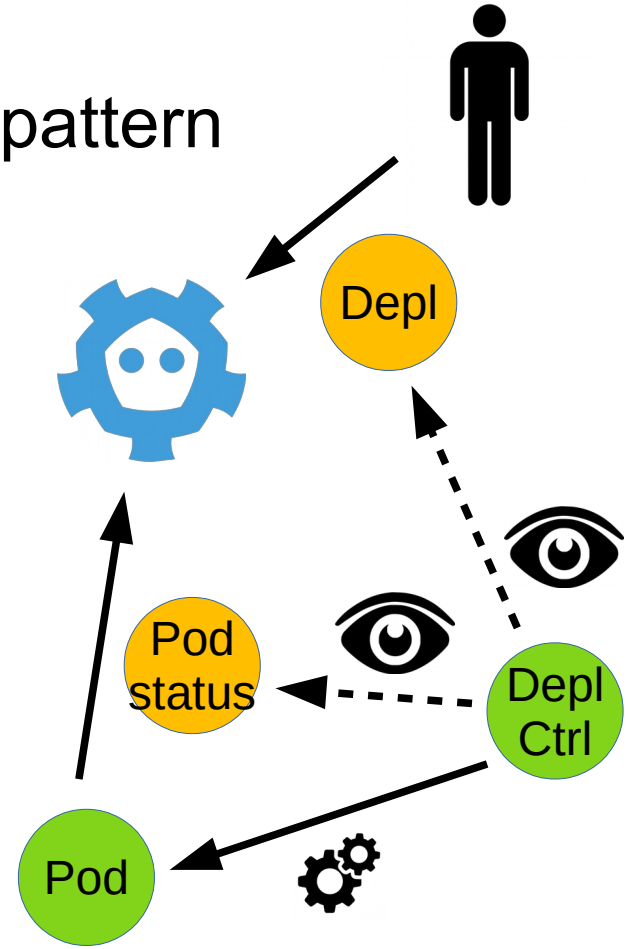
- **Kube uses Etcd as a datastore to hold:**
  - Resource Instance Definitions (= desired state)
  - Resource Instance Status Info (= actual state)
- **Controllers**
  - Watch Resource Instance Definitions (created, updated, deleted)
  - Create the Resource Instances as defined
  - Watch their current related Resource Instance Status Info
  - Strive to continuously match desired and actual state
    - **In doing this, Controllers automate the operational know-how of handling each kind of Resource**



Operator: The basic Kubernetes pattern

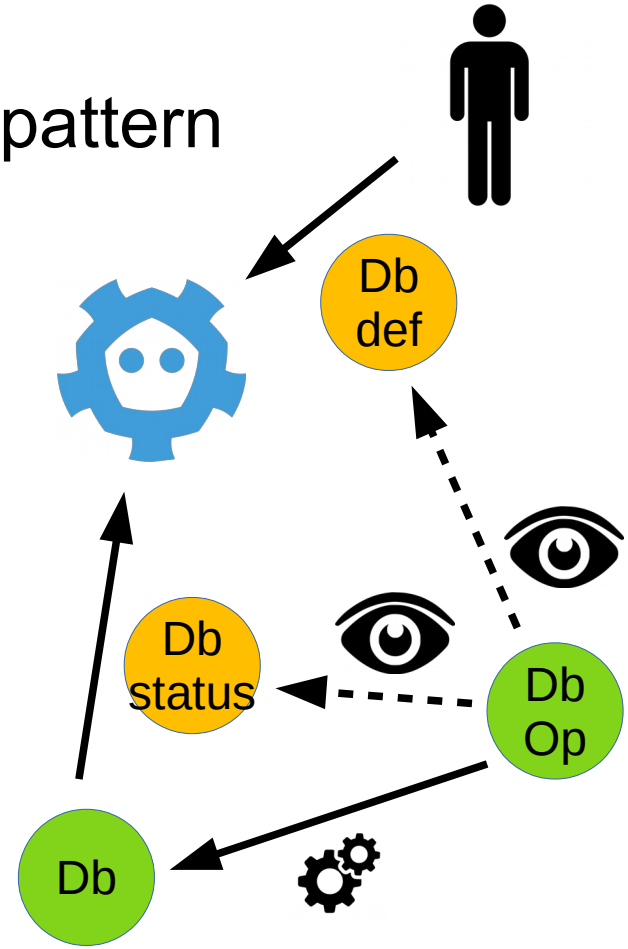
Example: Deployment Controller

- Watches for Deployment definitions
- When a new one is detected, it creates a series of Pods (as required by def)
- Continuously watches Pod status to ensure all required Pods are running and healthy
- When Pods are killed unexpectedly, it creates them again
- When Deployments are changed to update the desired replica count, adds or removes Pods as needed



Operator: The basic Kubernetes pattern

- **Controller** is a term used for code living in the K8s core
- But the same pattern can be applied for specialized applications that run in K8s like any other, but whose purpose is to manage and/or instantiate other applications: the **Operators**
- The pattern is similar: Users create **Custom Resource Definitions (CRD)** defined by the Operator itself, which expresses what they wish, and the Operator strives to make dreams (aka desired state) come true.
- The Operator handles the lifecycle of the **managed resources** by interacting with the **Kube API**, therefore **encapsulating the operational knowledge** of handling them, specially important when dealing with **stateful applications**.



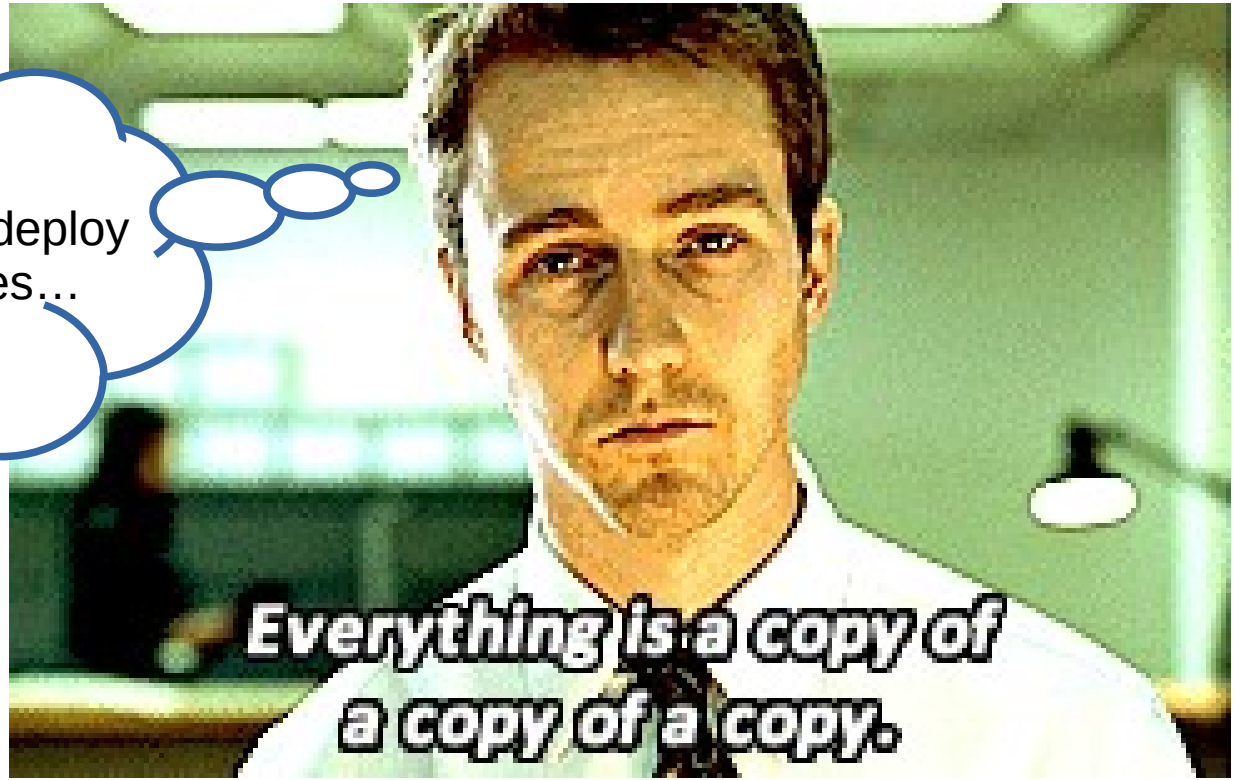
DEMO #1

Basic Kubernetes & the Controller effect

Kubernetes package management for reusability, consistency and reproducibility

Helm

MyBoss asked me to deploy MySQL on Kubernetes…
Again.



The Fight Club (1999) © Twentieth Century Fox Film Corp.

Helm :: Purpose

==> Reusable Kubernetes deployments

- **Externalized configuration + Manifest templating**

  → Reusability of a deployment in many & different environments

- **Composability**

  → Creating a supersystem by reusing smaller systems

- **Package management**

  → Reusing public or privately published & versioned modules

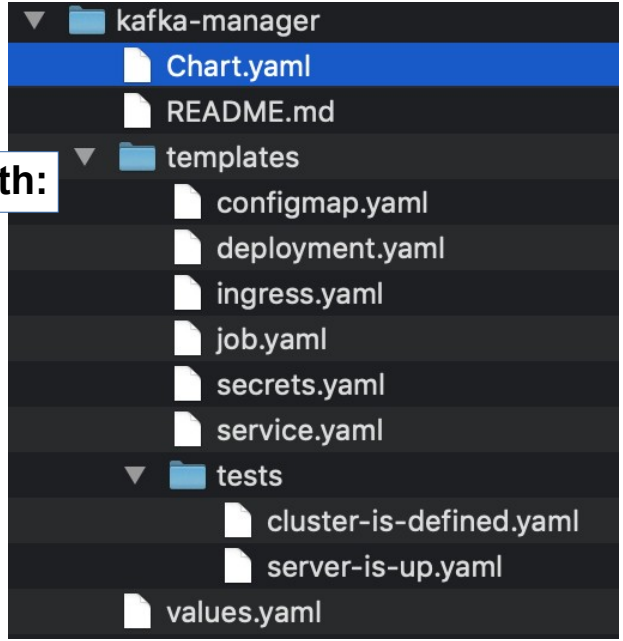
- **Upgrade & rollback support**

  → Keeping track of published upgrades of a deployment

Helm :: Concepts

- **Chart == Packaged (.tgz) set of Kube manifests, along with:**
  - Chart metadata
  - Default configuration values
  - Other support files (scripts, icons...)
- **Chart Configuration**
  - Defaults (inside the Chart)
  - External configuration (using Helm CLI)
    - 0+ config value files (YAML)
    - 0+ overriden values
- **Helm Release = Chart + Configuration**

  == Chart Instance (Identified by a **Release Name**)




```
helm install
    --values global.yaml
    --values environment.yaml
    --set service.port=5432
    stable/postgres
```

Helm :: Concepts

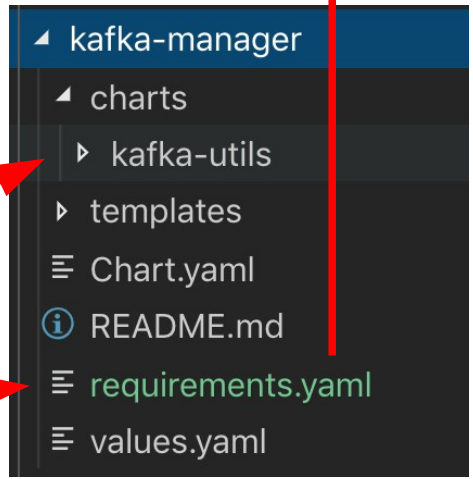
```
dependencies:
- name: postgresql
  version: 0.13.1
  repository: https://kubernetes-charts.storage.googleapis.com/
  condition: postgresql.enabled
```

- **Chart Repository** == Glorified .tgz file server + /index.yaml with Chart metadata

- **Chart Dependencies**

  == Other Charts, required by the main one, that provide additional content for the final Helm Release created

  - Dependent charts can be:
    - Inline: Nested in the main chart
    - Referenced: In requirements.yaml
      - Fetched at build time from Chart Repositories



Helm :: Concepts

- **Helm Release Revision**

  == An update on a Helm Release deployed to K8s

  - Required after a change in the Chart or the Config

```
Chart v1 + Config v1   →   Release Revision #1
Chart v2 + Config v1   →   Release Revision #2
Chart v3 + Config v2   →   Release Revision #3
...
```

Helm :: Concepts

```
kubectl get configmaps -n kube-system
```

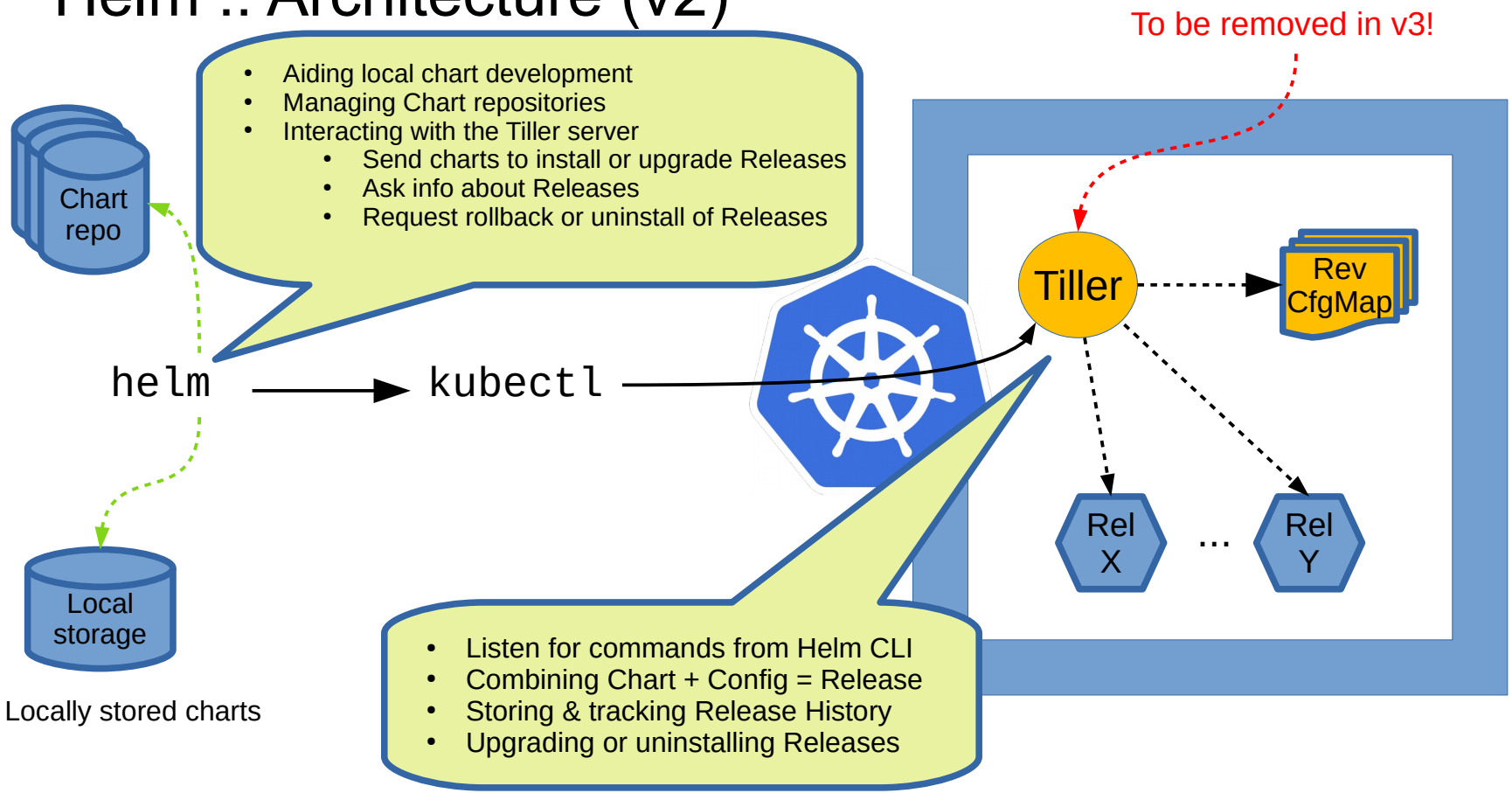
```
NAME                                 DATA   AGE
angry-bronco.v1                      1      67s
angry-bronco.v2                      1      27s
angry-bronco.v3                      1      20s
```

- **HR Revision History**
  - History kept inside K8s using ConfigMaps (do not touch!)
  - Can go back and forth in History with Helm CLI
    - `helm install <releaseName> <chart+config>` → Revision #1
    - `helm upgrade <releaseName> <chart+config>` → Revision #2
    - `helm upgrade <releaseName> <chart+config>` → Revision #3
    - `helm rollback <releaseName> 2` → Revision #4 (!!!)

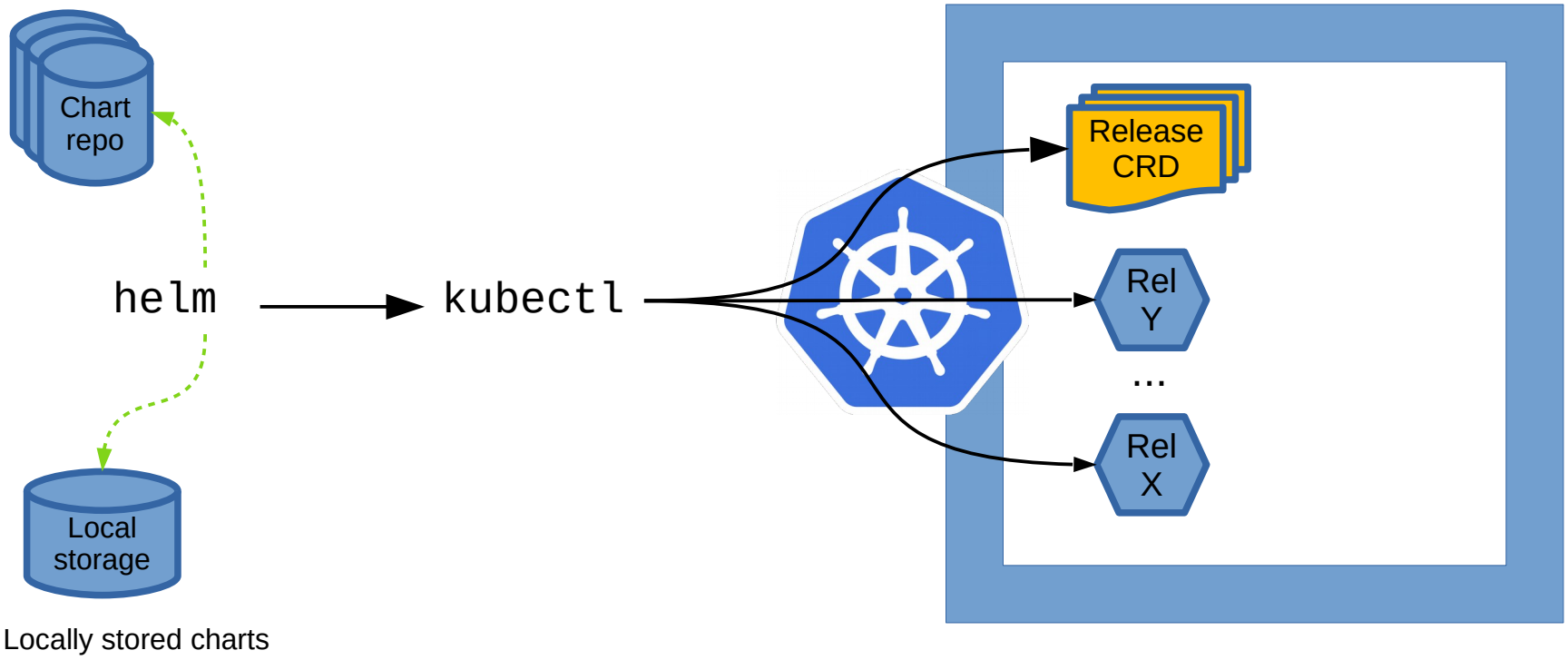
Back to v2 release, but creates a new revision!

```
NAME                                 DATA   AGE
angry-bronco.v1                      1      4m46s
angry-bronco.v2                      1      4m6s
angry-bronco.v3                      1      3m59s
angry-bronco.v4                      1      3s
```

Helm :: Architecture (v2)



Helm :: Architecture (v3)



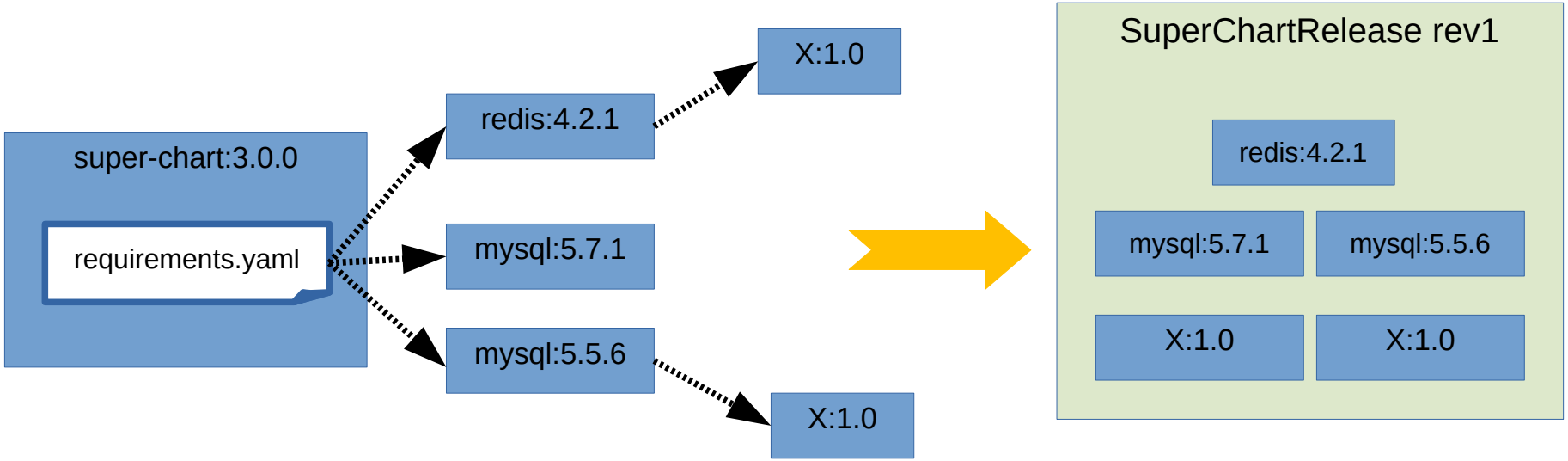
Locally stored charts

DEMO #2

Helm Chart development workflow

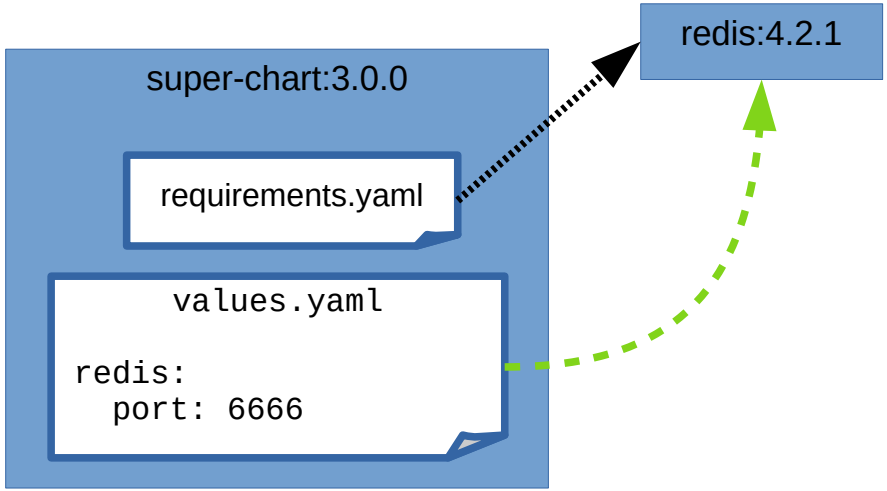
Helm :: Feature Highlights

- **Package Dependency Management**
  - requirements.yaml == Non-shared build-time dependencies



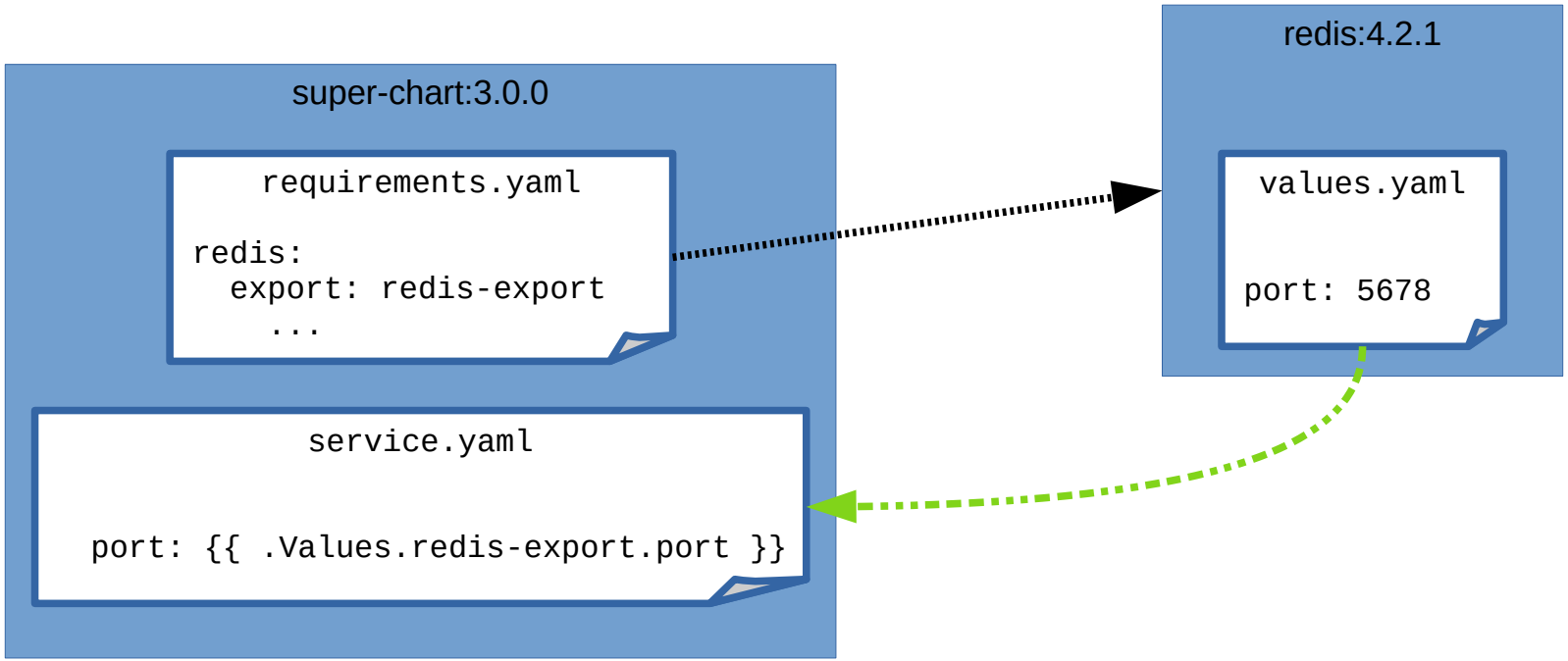
Helm :: Feature Highlights

- **Subchart config override**

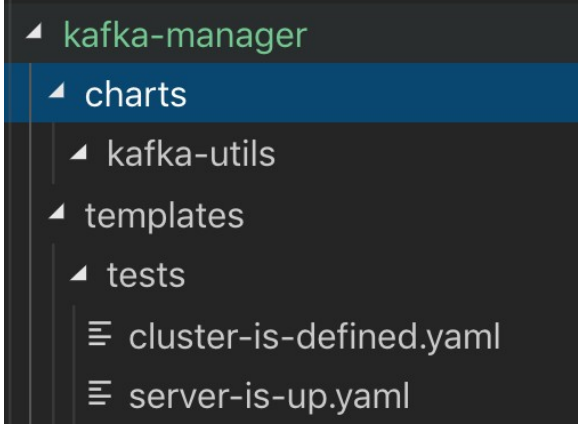


Helm :: Feature Highlights

- **Subchart config exports**



Helm :: Feature Highlights



- **Chart Tests (I)**
  - Kube Pod manifests, defined under: `templates/tests/**`
  - Not deployed on Chart install
  - Stored with Release Revision data (Tiller ConfigMaps)
  - Deployed on command: `helm test <ReleaseName>`
  - **Test success == All Pod containers exit with (success | failure)**
    - Depends on per-test config

Helm :: Feature Highlights

- **Chart Tests (II)**
  - **Recommended use: Smoke Tests**
    - Helps encapsulate knowledge about how to check if an application is working as expected after deployment
      - **Allows Dev staff to provide Ops staff with app-specific sanity checks they can run when in trouble**
    - Can serve as after-deployment tests in CD pipelines
      - E.g. Rolling back to the previous Release Revision on tests failure

Helm :: Feature Highlights

- **Chart Tests (III)**
  - **Drawbacks:**
    - **Cannot filter the tests to be executed with the “helm test” command**
      - Can be done in a home-made fashion, but it’s hard
      - It’s a problem for big Releases, composed of many Chart deps
        → **Big problem for UberChart pattern! (see later)**
    - **Lack of support for sidecar containers during tests**
      - All containers in the test Pod are considered tests
        - i.e. All of them must exit with success/failure according to test cfg
      - Can’t have sidecar containers with ignored end status and auto killed
        - i.e. Must ensure sidecars fail if tests fail, and vice versa
          - Can be done in a home-made fashion, but painful!

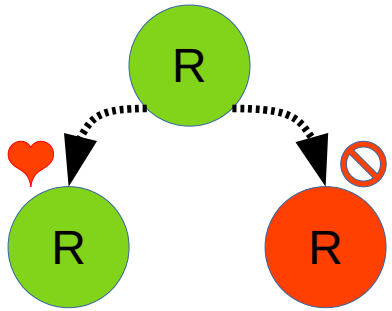
Helm :: Non Features

- **Tiller is not a Helm Release Operator**
  - Releases are not auto redeployed if Release resources are deleted manually
- **Cannot specify the deployment order of sub-Charts in a Release**
  - Applications deployed all at once

    → Must be able to wait for dependencies to be up → **“Limbo” time!**
- **Can’t reference dependency Charts by Git repo URL**
  - Must be published in a Chart repo

    → Chart repos sometimes unnecessary for corporate Charts
    - But always forced to package Charts and maintain a Chart repo

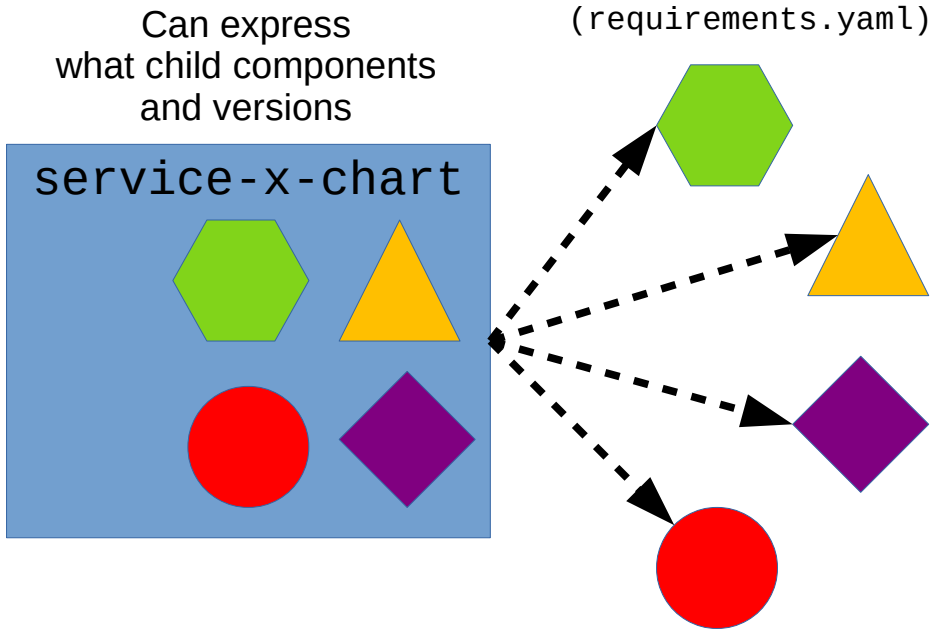
Helm :: Non Features



- **No Release Dependency Management support**
  - Helm =~ Docker && Helm != DockerCompose
  - Chart dependencies in requirements.yaml are build-time deps → All part of a single Release
  - Release Dependencies = Runtime dependencies
    - Dependencies not included in the Release
    - But required to be there at run time for the Release to work properly
  - No support for deploying multiple interdependent Releases
    - Can be done using IaC tools (e.g. Ansible with “helm” module)
  - No support on Release deployment to check for depended-on Releases to be in place
    - Can be done in a custom-made fashion using Helm pre-install hooks
    - But must do it for every chart

Helm :: Non Features

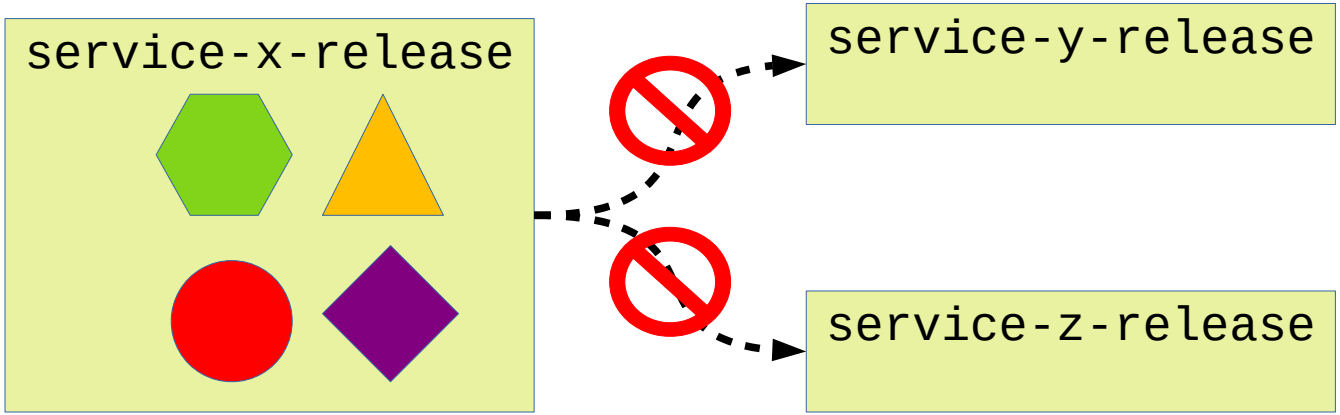
- **No Release Dependency Management support**



Helm :: Non Features

- **No Release Dependency Management support**

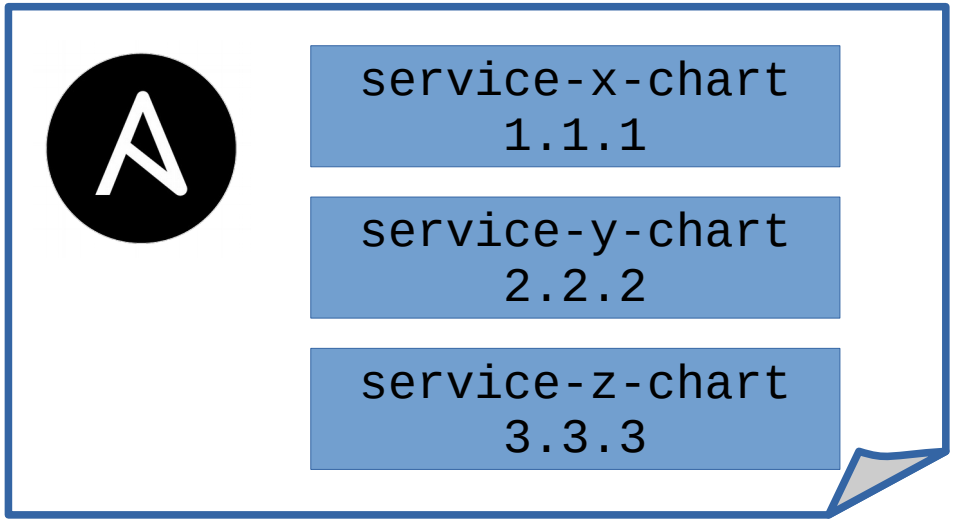
Charts cannot express what sibling releases and versions
→ **Releases can be installed unlinked**



Helm :: Non Features

- **No Release Dependency Management support**

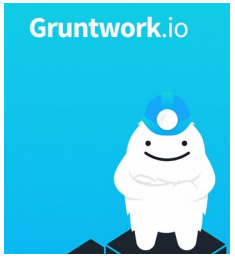
→ Need a higher level tool to specify the right Chart versions to keep system consistency



Helm :: Non Features

- **No Unit Test support**
  - Unit Testing in Helm == Testing conditional Chart template generation
    - Input == Config values + Chart templates
    - Output == K8s manifests (YAML) ← Assert expectations!
  - Specially useful for “library” Charts (i.e. designed for reuse)
    - Not so much for “system definition” Charts (i.e. specific system impl.)
  - External tools available:
    - **HelmTest plugin**
      - Runs “helm template” command and asserts on YAML outputs
    - **Terratest**
      - Multipurpose Go library for infra testing
      - Includes test helpers to assert on Helm-generated templates

Helm :: Non Features



- **No Integration Test support**
  - Helm tests
    - Kinda similar to Int Tests… (runs checks on the deployed application)
    - But can only run tests where the Release has been deployed
      - **Test not in control of the deployment scenario → Need external tool for that**
  - External tools available
    - **Terratest**
      - Multipurpose Go library for infra tests
      - Supports Helm Chart deployment & asserting on the deployed app
    - **KitchenCI**
      - Multipurpose test env generator & test executor
      - Can provision local VM with Minikube & deploy with Helm CLI
      - Supports several infra testing libraries to assert on the deployed app

Helm :: Alternatives

- **Kustomize**
  - Integrated into kubectl tool since K8s 1.14
  - Only for manifest templating
    - Other Helm features not supported
  - YAML-friendly syntax, cleaner than Helm's Go Template syntax
- **Helm + Kustomize**
  - Can use all other Helm features, but do templating with Kustomize
    1) Run "`helm template`" → Outputs the generated K8s manifests
    2) Run Kustomize over the generated manifests → Generates final manifests
    3) Send manifests to Kube with "`kubectl apply`"
  - Helps avoiding the more convoluted uses of Go Template, and still have Package Mgmt

Distributed System Consistency

- **Distributed System**

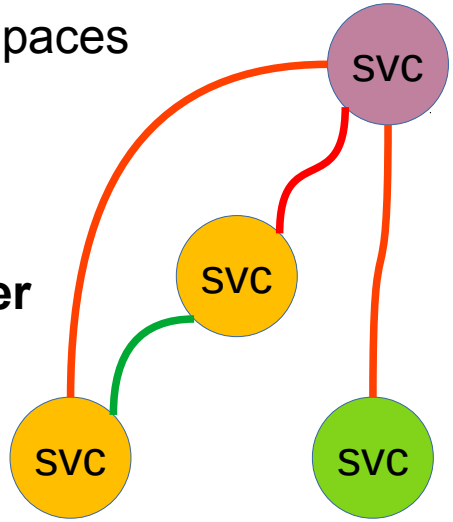
  == A set of interdependent, network connected Services

- **Services**:
  - Are usually developed by different teams, at different paces
  - Need to be able to work correctly with each other

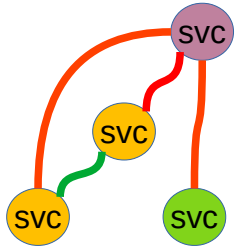
  → Need to keep deployed always the right versions

  → **Every system change must result in another set of compatible versions**

  → **We need to specify somewhere that versions set**



Distributed System Consistency

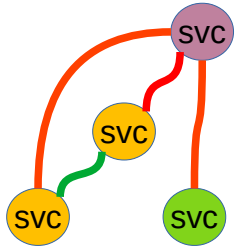


- Besides that, **we should always be able to reconstruct our system from scratch** to recover from disasters.
- For that you need:
  - **A declarative specification of your current system state**
    - What applications, and which versions
    - What configuration
  - **A tool to build the system from the declarative spec**

→ Can we do that with Helm?

- **Problem: Helm only deploys individual releases**

Distributed System Consistency



- **Option #1: Single Super Chart (to rule them all)**

Deep tree (= monorepo)

```
capsule-inc-global/
    charts/
        capsule-inc-hhrr/
            charts/
                ...
        capsule-inc-research/
            ...
        capsule-inc-accounting/
```

Shallow wrapper

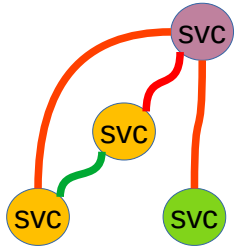
```
capsule-inc-global/
    requirements.yaml
        --> capsule-inc-hhrr
        --> capsule-inc-research
        --> capsule-inc-accounting
```

Single Super Helm Release !

DEMO #3-1

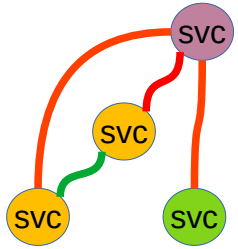
One Super Chart to rule them all

Distributed System Consistency



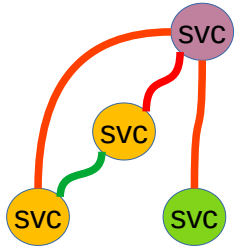
- **Option #1: Single Super Chart (to rule them all)**
  - **Pro**:
    - No extra tool needed, just Helm to build the full system
  - **Con**:
    - **Service charts not independently manageable**
      - Some chart metadata becomes corrupted...
        → Release needs to be deleted...
        → **Full system takedown!!**
      - Independent teams wanting to do their own thing…
      - Unable to smoke-test a single malfunctioning sub Chart
        - **“helm test” command cannot filter tests!**

Distributed System Consistency



- **Option #1: Single Super Chart (to rule them all)**
  - **Con**:
    - **No built-in way to test if Service charts have all their runtime dependencies (i.e. other Service charts) satisfied**
      - Chart metadata doesn’t support expressing runtime dependencies
      - Helm doesn’t implement checking runtime deps
      - Can be done, but in a custom way

Distributed System Consistency

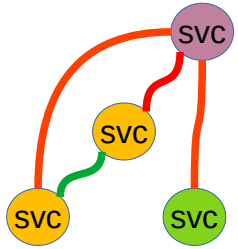


- **Option #2:**

  Orchestrated deployment of multiple Helm Releases

  - Full System = 2+ Helm Releases
    - Usually 1 Release = 1 Logical Service (funct cohesion)
      - Logical Service = 1+ closely integrated applications
  - **Requires: Release orchestration tool**
    - Simple shell script
    - Any IaC tool
      - e.g.: Ansible + “helm” Ansible module
      - **Helmfile !**

Distributed System Consistency

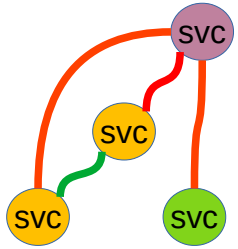


- **Option #2:**

  Orchestrated deployment of multiple Helm Releases

  - Con:
    - Need an extra tool
    - Still no support for runtime dependency management
  - Pro:
    - Independently manageable Helm Releases !

Distributed System Consistency



- **Option #2:**

  Orchestrated deployment of multiple Helm Releases

  - Example tool: **Helmfile**
    - **Features**:
      - Declarative Helm CLI automation
      - Go Template support (like in charts! → familiarity)
        - Allows conditional Release installation & config
      - Environment definition support → Grouping of config values per env
      - Infinitely nestable (Helmfiles can include other Helmfiles)
      - Configurable Release deployment order
        - Releases are deployed by their order in the Helmfile

```
releases:
  - name: kafka-manager
    chart: ./kafka-manager
    namespace: transverse
    labels:
      release-name: kafka-manager
    values:
      - ../values-global.yaml
```

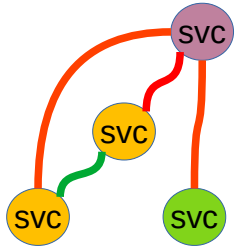
DEMO #3-2

Simple Helmfile

DEMO #3-3

Complex Helmfile

Distributed System Consistency



- **Option #2:**

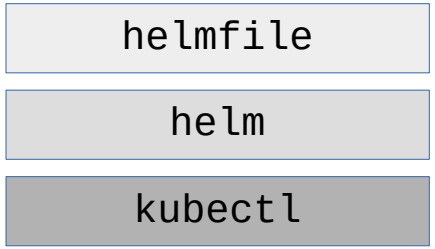
  Orchestrated deployment of multiple Helm Releases

  - Example tool: **Helmfile**
    - **Drawbacks**:
      - An extra tool to learn/install/use
      - No support for Helm v3 yet
      - Beta! (but used in Prod by many companies)
      - Can’t reference Charts directly by Git repo URL
        - Just delegates to Helm behind the scenes, and Helm doesn’t support it.
      - Configurable Release deployment order
        - Releases are deployed by their order in the Helmfile

```
releases:
  - name: kafka-manager
    chart: ./kafka-manager
    namespace: transverse
    labels:
      release-name: kafka-manager
    values:
      - ../values-global.yaml
```

Current Status

- Managing 3 tools for app deployment



Release orchestration

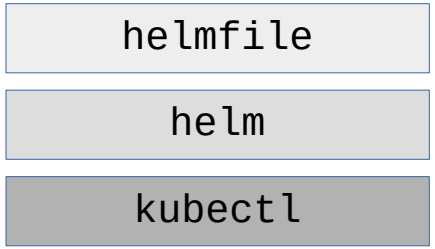
Release installation

K8s manifest execution

… can we do better?

Current Status

- Managing 3 tools for app deployment



Release orchestration

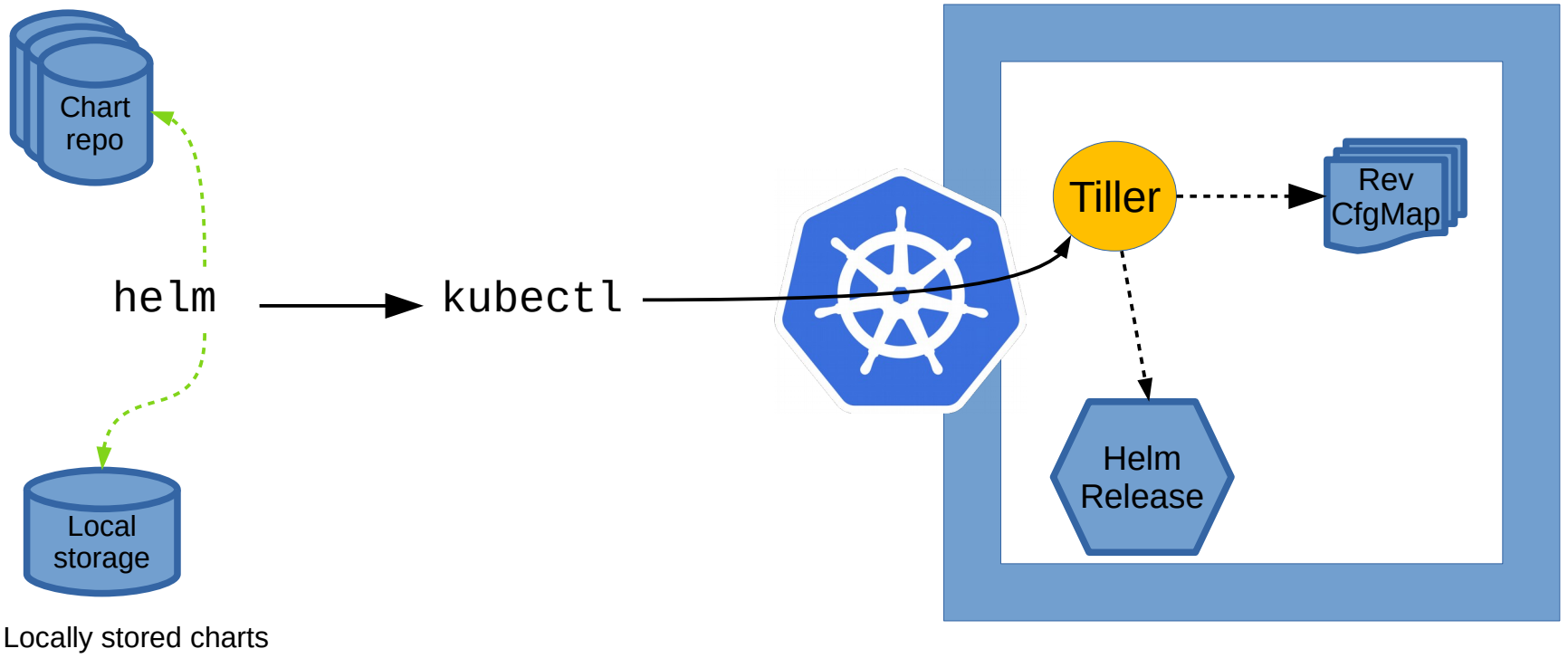
Release installation

K8s manifest execution

… can we do better?

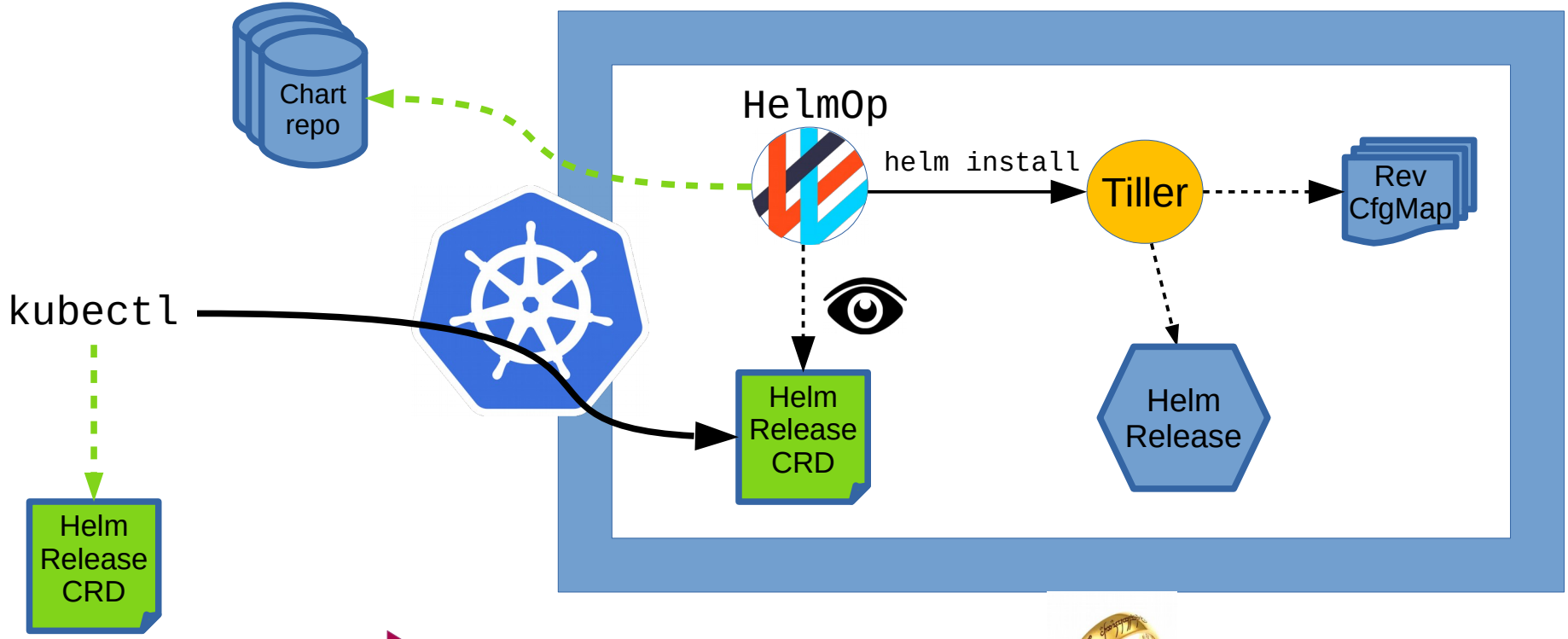
Yes, by needing less tools

Before: Standard Helm



Locally stored charts

Enter Weaveworks' Helm Operator



One tool to rule them all...

Helm Operator explained

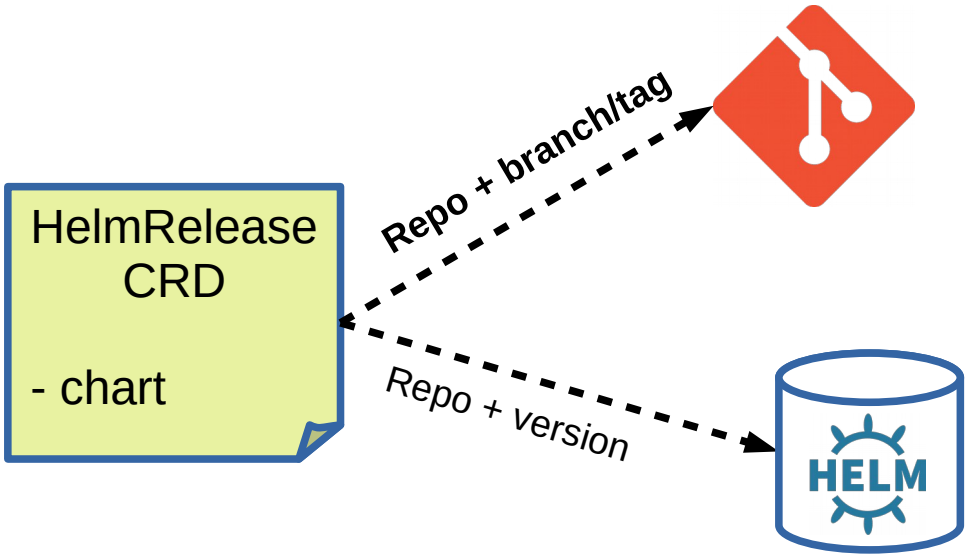
- **Operator application**
  - User creates K8s CRD's of type HelmRelease (defined by HO)
  - HO watches for HR objects and interacts with Tiller to install or upgrade Releases
- **Can run inside (usual) or outside Kube (reuse and/or security hardening)**
- **Centralizes:**
  - Installation of the Helm CLI tool
  - Access to Helm chart repos and their credentials
  - Auditing of deployments (who = Helm Release CRD creator)

Simplified user setup and authorization

Helm Operator explained

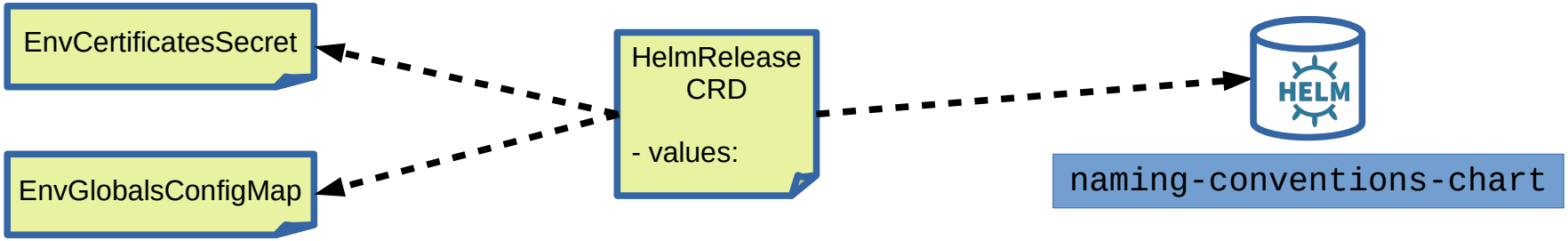
- **HelmRelease CRD allows referencing Helm Charts by:**
  - Helm chart repo reference
  - Git repo URL → **Missing feature in Helm!**

    → **No need to package and upload to chart repo!**



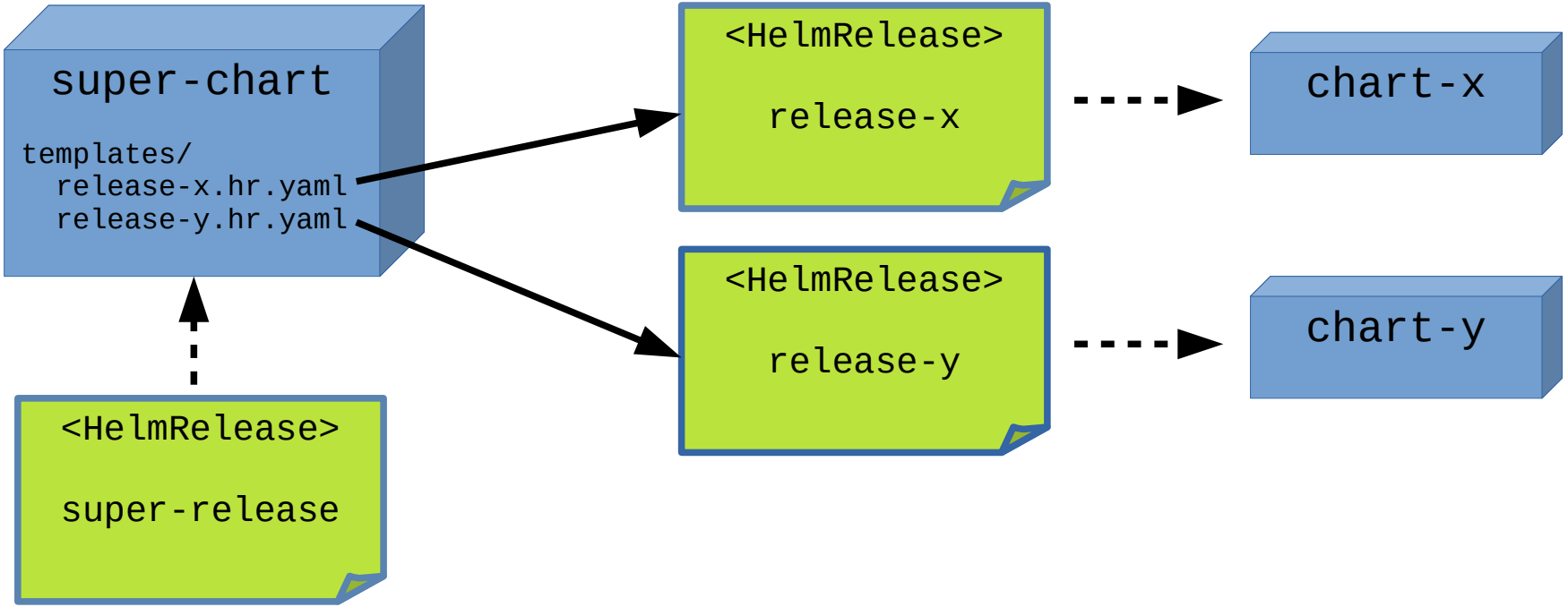
Helm Operator explained

- **Much higher composability for a chart's config values**
  - Config value sources (can use many)
    - ConfigMaps, Secrets
    - External file (URL)
    - Another chart
  - Example uses:
    - Deploy a single ConfigMap for all environment-specific configs, and have all charts use it as config value source
    - Sharing common config values among all the charts



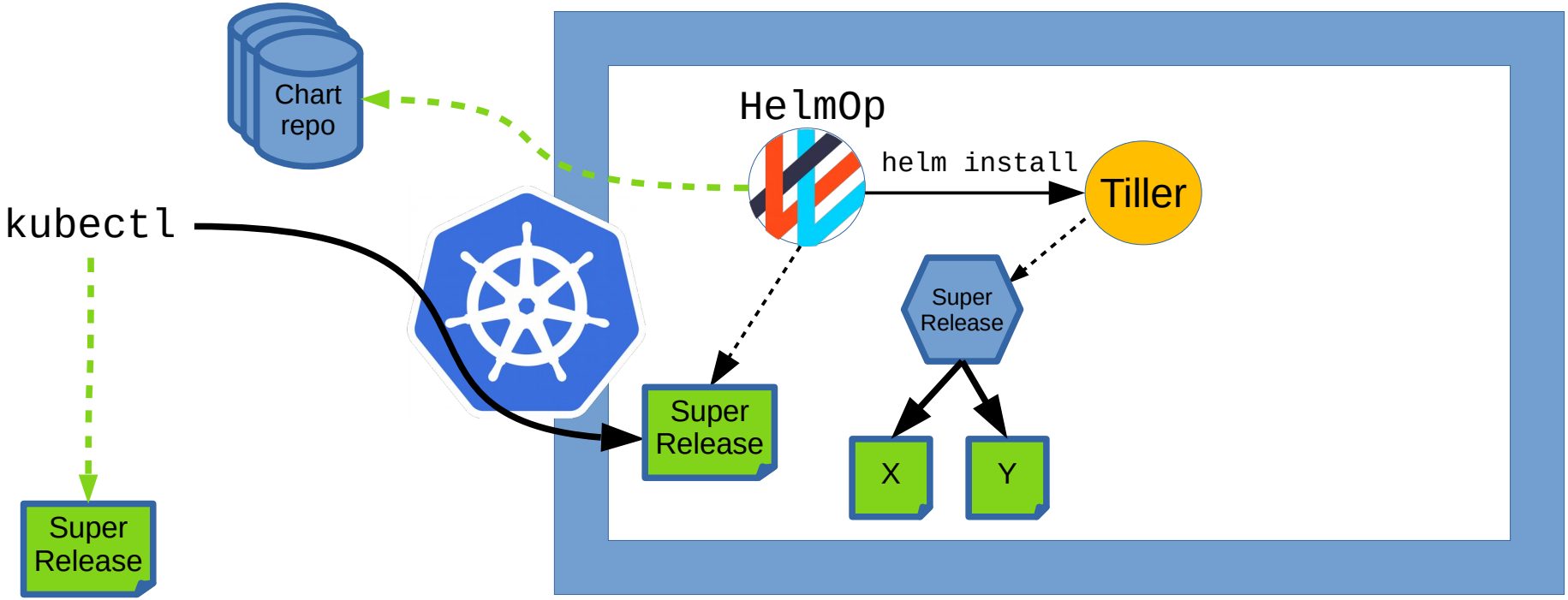
Helm Operator explained

Helm Release Orchestration revisited



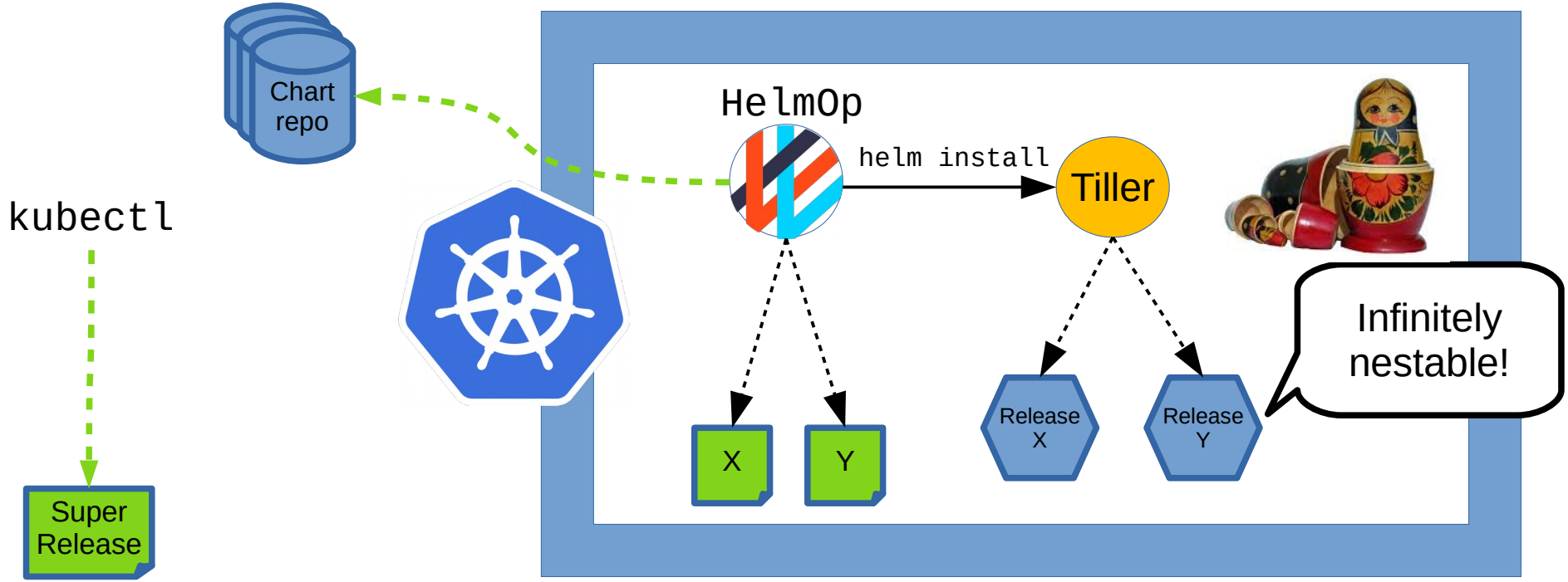
Helm Operator explained

Helm Release Orchestration revisited



Helm Operator explained

Helm Release Orchestration revisited



DEMO #4

Helm Operator

Helm Operator explained

Drawbacks

- **Still delegating finally to Helm/Tiller to perform the deployment**

  → Still carrying Helm's limitations

  - Can't specify HelmRelease deployment order
  - Can't filter executed tests in a single HelmRelease

- **The chained sequence of HelmRelease deployments makes "limbo time" longer**

  - "Limbo time"

    = The time that a deployed app might have to wait to its runtime dependencies (other apps)

  - During "limbo time" apps might get restarted because of missing runtime deps

    → Introduces "noise" in logs during deployment

- **We no longer have a way to trigger ALL the tests (no link between HelmReleases)**

  - Need to loop all the HelmReleases and test

Quick status recap...

- **Deploying Helm charts just using `kubectl` <== Sweet!**
  - No need to install Helm / Helmfile, just kubectl
  - No need to have locally configured access to chart repos, chart git repositories, etc. All centralized in the Helm Operator deployment.
- **Can do deployments at many levels of granularity**
  - Extremely coarse grained: Single superchart, single Helm release
  - Extremely fine grained = 1 Helm Release per application
  - Recommended:
    - Superchart as the global system definition
      - Contains 1 HelmRelease per Logical Service
      - Contains global config values shared by all LS's

Can we do better?

Yes, we can!

© by Firas Alkhateeb, 2009

- **One single tool (`kubectl`) is still one too much**
  - Ops still need to configure `kubectl` locally
- **Auditability**
  - We have no centralized log about who deployed what and when
- **Need careful access control for Ops**
  - Because Ops must use `kubectl`, permissions must be carefully assigned to avoid security issues.

Yes, we can!

- **Lack of a consistent workflow**
  - **System consistency**
    - We have no process to ensure that teams don’t step over each other in the deployments (e.g. breaking runtime dependencies between deployed HelmReleases)
  - **System change promotion between environments**
    - We have no process to ensure an orderly progression of system changes from one environment to the other

Yes, we can!

- **Lack of a consistent workflow (II)**
  - **System divergence control**
    - We have no way to know when the current system state is diverging from the desired state
      - e.g. Ops making system changes in troubleshooting scenarios, but forgetting to translate those changes into Git
        - Relying on Ops' discipline & good practices → Danger!

Yes, we can!

- **Lack of a consistent workflow (II)**
  - **System divergence contr**
    - We have no way t he current system state is the desired state
      - e.g system changes in g scenarios, but translate those changes

Relying on Ops' discipline & good practices → Danger!

CD Pipeline !

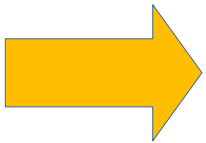
Yes, we can!

- **Lack of a consistent workflow (II)**
  - **System divergence contr**
    - We have no way t… …he current system state is… …n the desired state
      - e.g… …ystem changes in t… …g scenarios, but …translate those changes
      - …Relying on Ops'… …ood practices → Da…

CD Pipeline !

Is a CD pipeline enough?

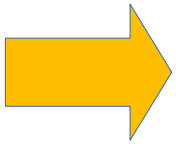
- **One single tool (`kubectl`) is still one too much** ✅
  - ~~Ops still need to configure `kubectl` locally~~
- **Need careful access control for Ops** ✅
  - ~~Because Ops must use `kubectl`, permissions must be carefully assigned to avoid security issues.~~



- **Yes, deployments centralized by the CD pipeline**
  → Less need for kubectl & less privileges needed
- **However, if Ops still need `kubectl` for fine tuning or troubleshooting apps or Kube resources, we are stuck**

Is a CD pipeline enough?

- **Auditability**
  - ~~We have no centralized log about who deployed what and when~~ 

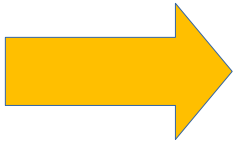


- **Yes, now the CD pipeline tooling provides a tracing mechanism for deployments**

- However:
  - **If Ops still use `kubectl` for tweaking or troubleshooting**
    → We have no log about those actions
  - **Sometimes CD pipelines are just about individual applications, not global system definition**
    - No single place to track global sytem evolution
      - *“When did we have service X working with service Y?”*

Is a CD pipeline enough?

- **Lack of a consistent workflow**
  - **System consistency** ✅
    - ~~We have no process to ensure that teams don't step over each other in the deployments (e.g. breaking runtime dependencies between deployed HelmReleases)~~

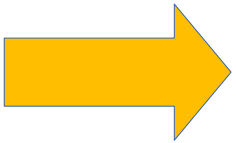


Yes, we can achieve that

- As long as CD pipelines perform integration or smoke tests after deployment, and rollback in case of failure
- If they don't rollback, temptation to leave it broken is too strong

Is a CD pipeline enough?

- **Lack of a consistent workflow**
  - **System change promotion between environments** 
    - ~~We have no process to ensure an orderly progression of system changes from one environment to the other~~



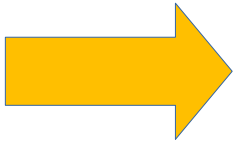
Yes, we can achieve that

- **As long as we check for overall system consistency after deployment at each environment, and refuse promotion on failure**
- Example:
  - CI pipeline for X performs integration tests of X against Y with version v2, and CD deploys to INT & STAGING
  - Works fine in INT env, but STAGING env still has v1 of Y

  → **Integration tests pass, INT env works, but STAGING breaks**

Is a CD pipeline enough?

- **Lack of a consistent workflow**
  - **System divergence control** ✅
    - ~~We have no way to know when the current system state is diverging from the desired state~~



Yes, we can achieve that, but not by default

- **It's something very dependent on the deployment environment**

  → Not provided by CD tool, unless it's specialized in env X

→ It's very usual to not have this, and after CD deployments the environments becoming the source of truth

- "Would you be able to redeploy in Mars tomorrow?"

Other CD pipeline considerations

- **Usually very ad-hoc**
  - Well-known mature CD tooling (Concourse, Jenkins, Spinnaker…)
  - But every org needing their own deployment pipeline
    - Different target runtime environments (K8s, AWS, Nomad...)
    - Different approval processes

→ **Little chance for reuse across organizations**

→ **Deployment process tightly coupled to the CD tool**

- Hooking into the process usually means hooking into the tool

Other CD pipeline considerations

- **Sometimes CD pipelines are just about individual applications**
  - 1 application → 1 CD pipeline

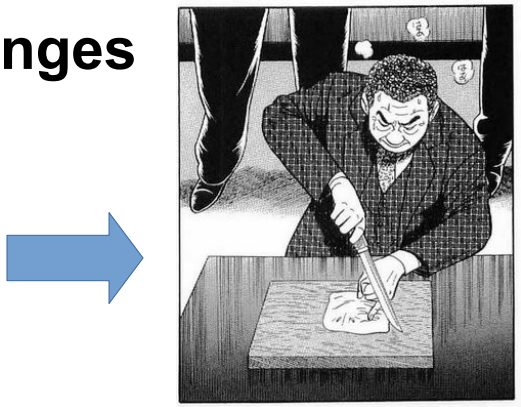
    → **No coordinated effort to keep system consistency**
    - **Nothing prevents us from deploying incompatible services!** ← Specially if integration tests are lacking...
- **Usually CD tooling allows us diverging the system away from the System Definition**
  - E.g. Your typical button to rerun an old deployment job

    → **Environments become the real source of truth!**

    → **We stop being able to reproduce the system from scratch**

The GitOps proposal

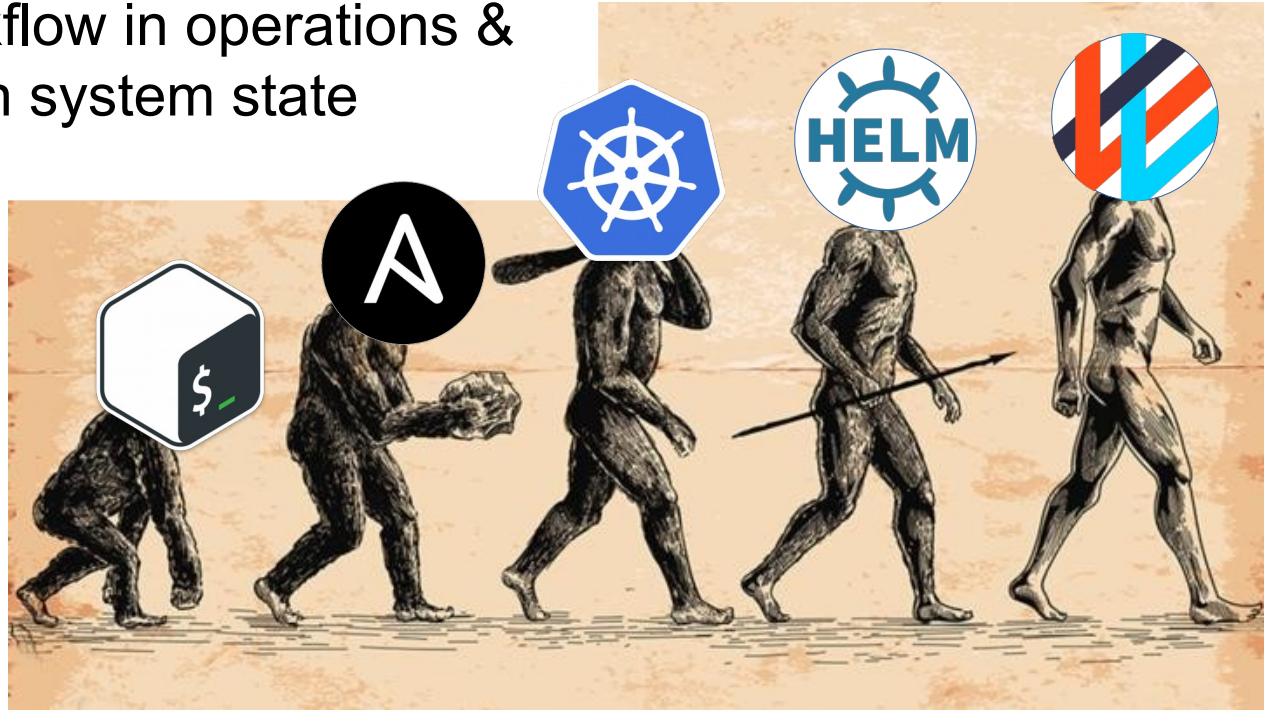
- **Have a declarative Global System Definition in Git**
- **Allow system changes only through GSD changes**
  - No direct operations on the System by Ops
    - That includes manual deployments, too
  - No rerunning old deployment jobs either!



- **Leverage Git's transactional capabilities to enforce the proper workflow between teams for System changes**
- **Continuously monitor System divergence and automatically roll it back to latest accepted GSD**

GitOps

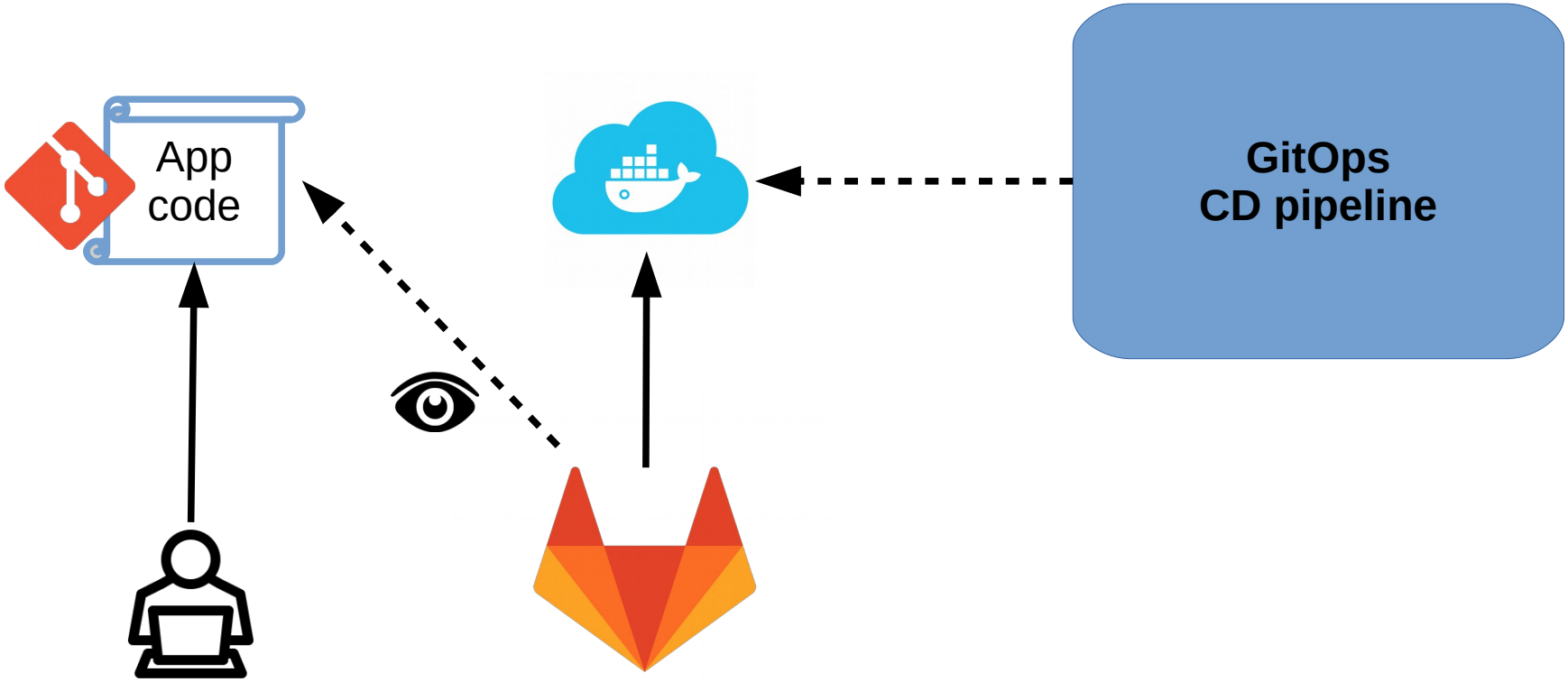
Enforcing workflow in operations & standarizing on system state evolution



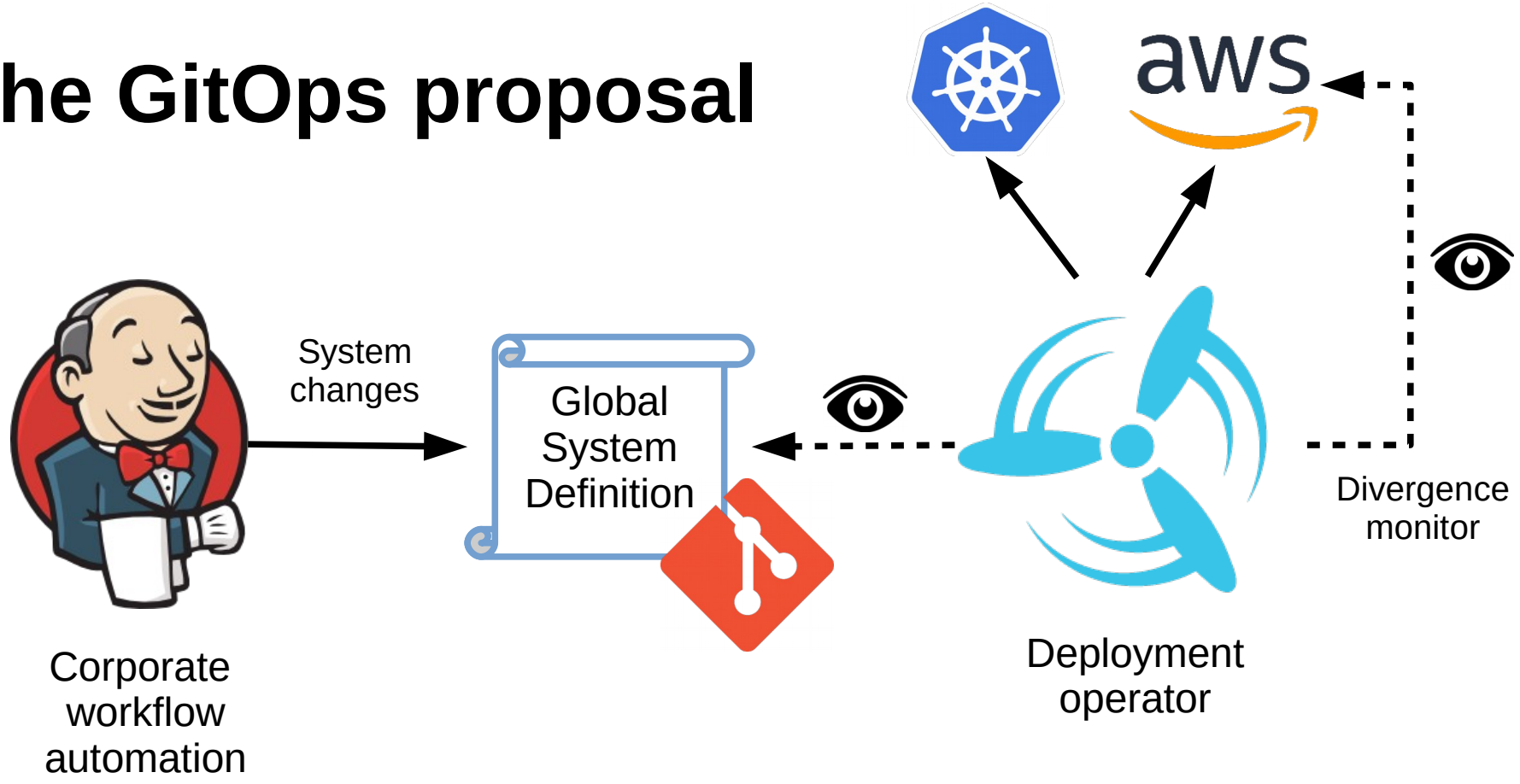
GitOps History

- 2014 – Weaveworks
  - SaaS CD solutions for K8s
- 2016 – Flux OSS project
- 2017 – GitOps article →
- Currently
  - GitOps = Pattern
  - Codefresh: GitOps 4 Terraform

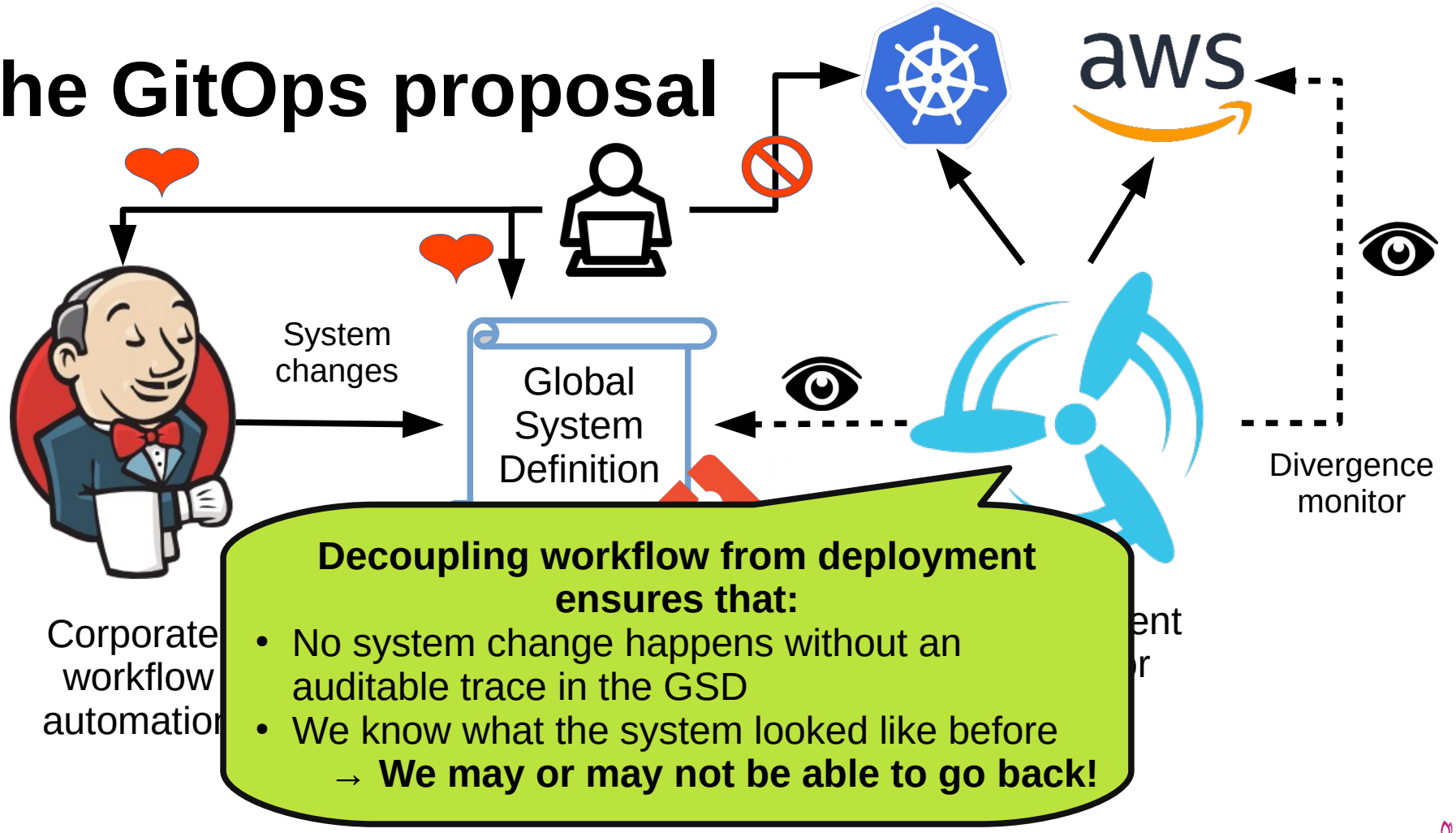
The GitOps proposal



The GitOps proposal



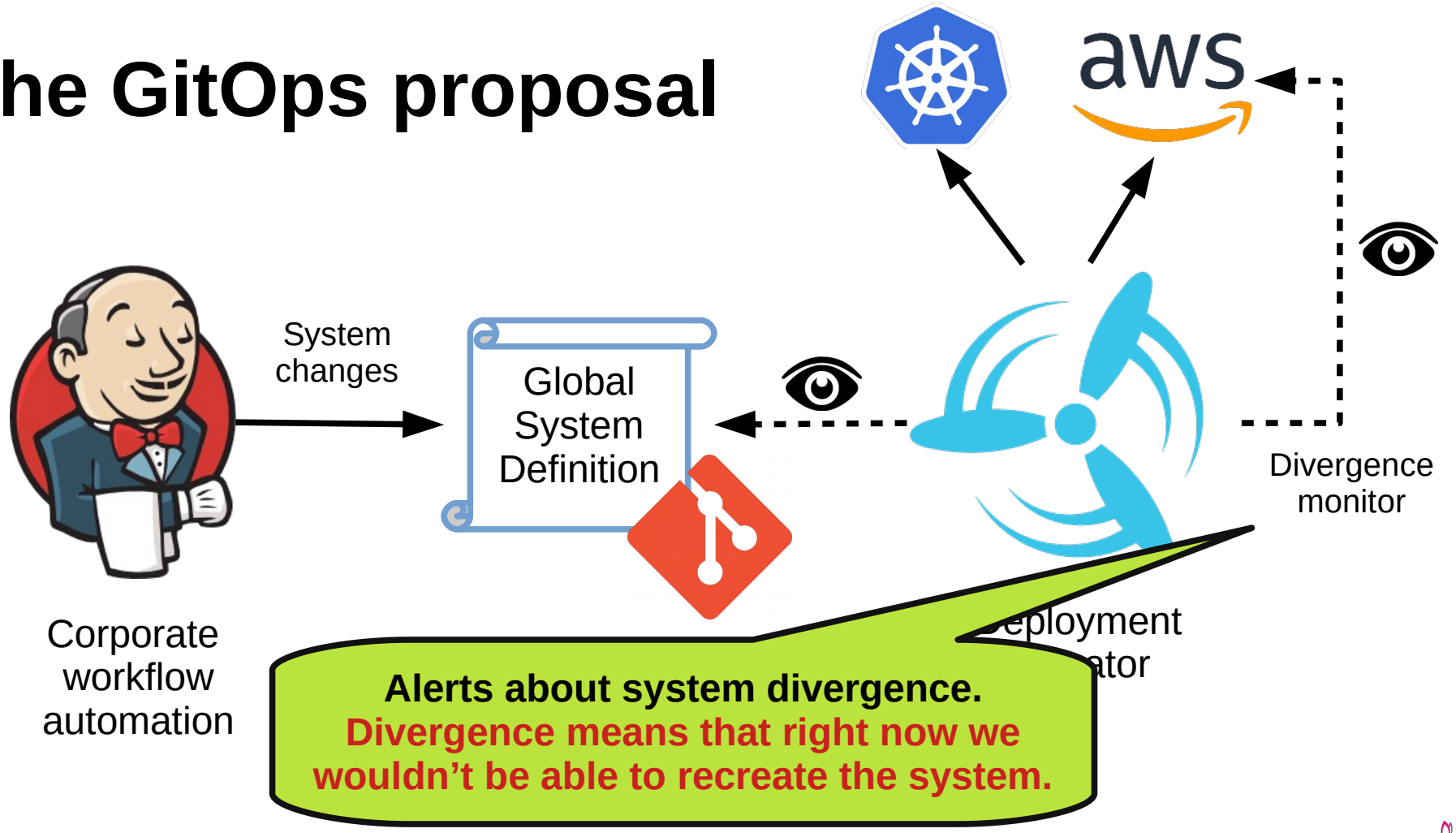
The GitOps proposal



Decoupling workflow from deployment ensures that:

- No system change happens without an auditable trace in the GSD
- We know what the system looked like before
  → **We may or may not be able to go back!**

The GitOps proposal



roposal

Workflow Engine & Sys log

- Keeps the **GSD** change log
- Keeping the **current state** of all the running workflows
- Provides **transactional support** for the workflows
- Multi purpose **state machine**
- Can easily **plug-in** new tools into the workflows

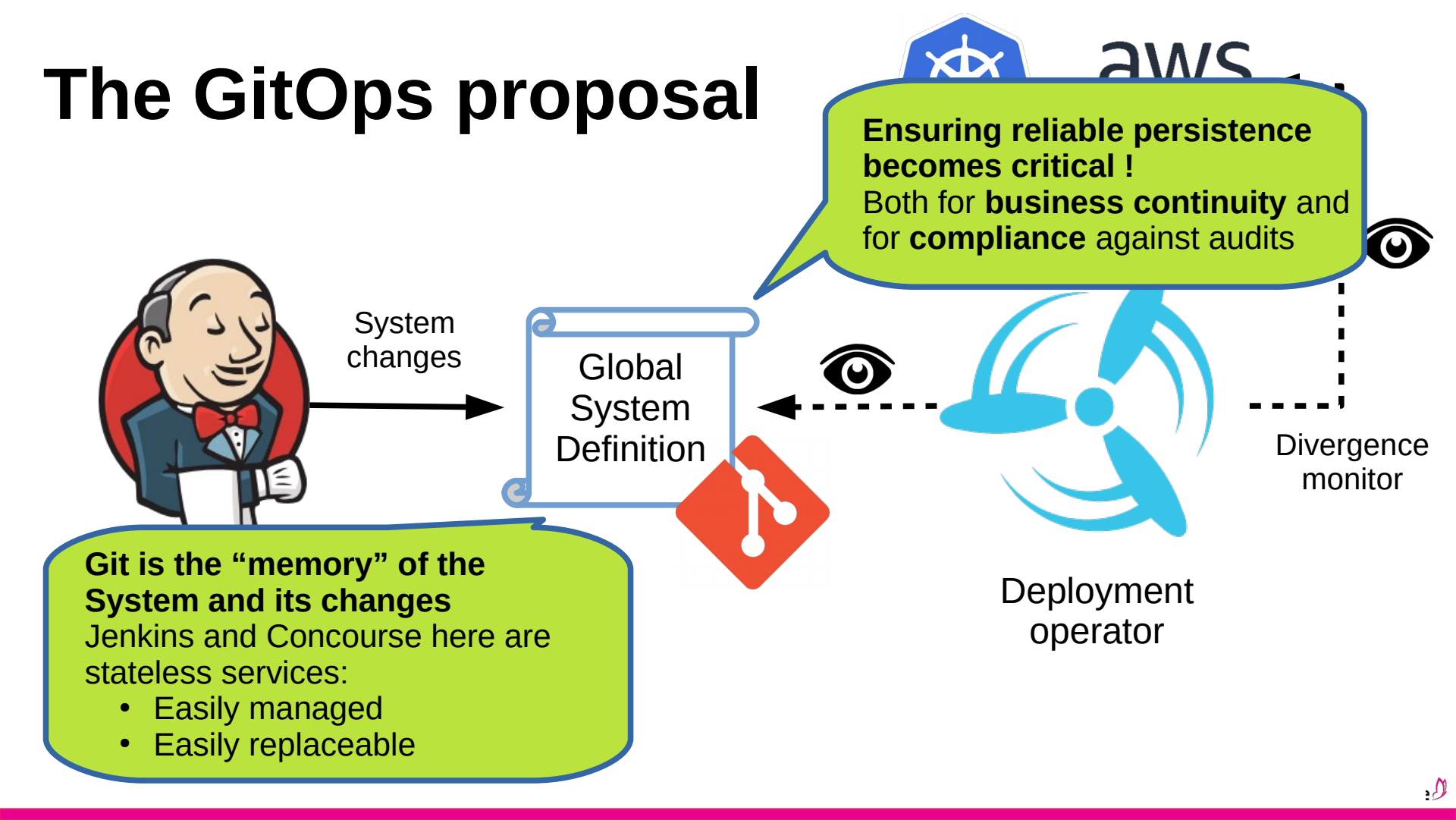
Global System Definition

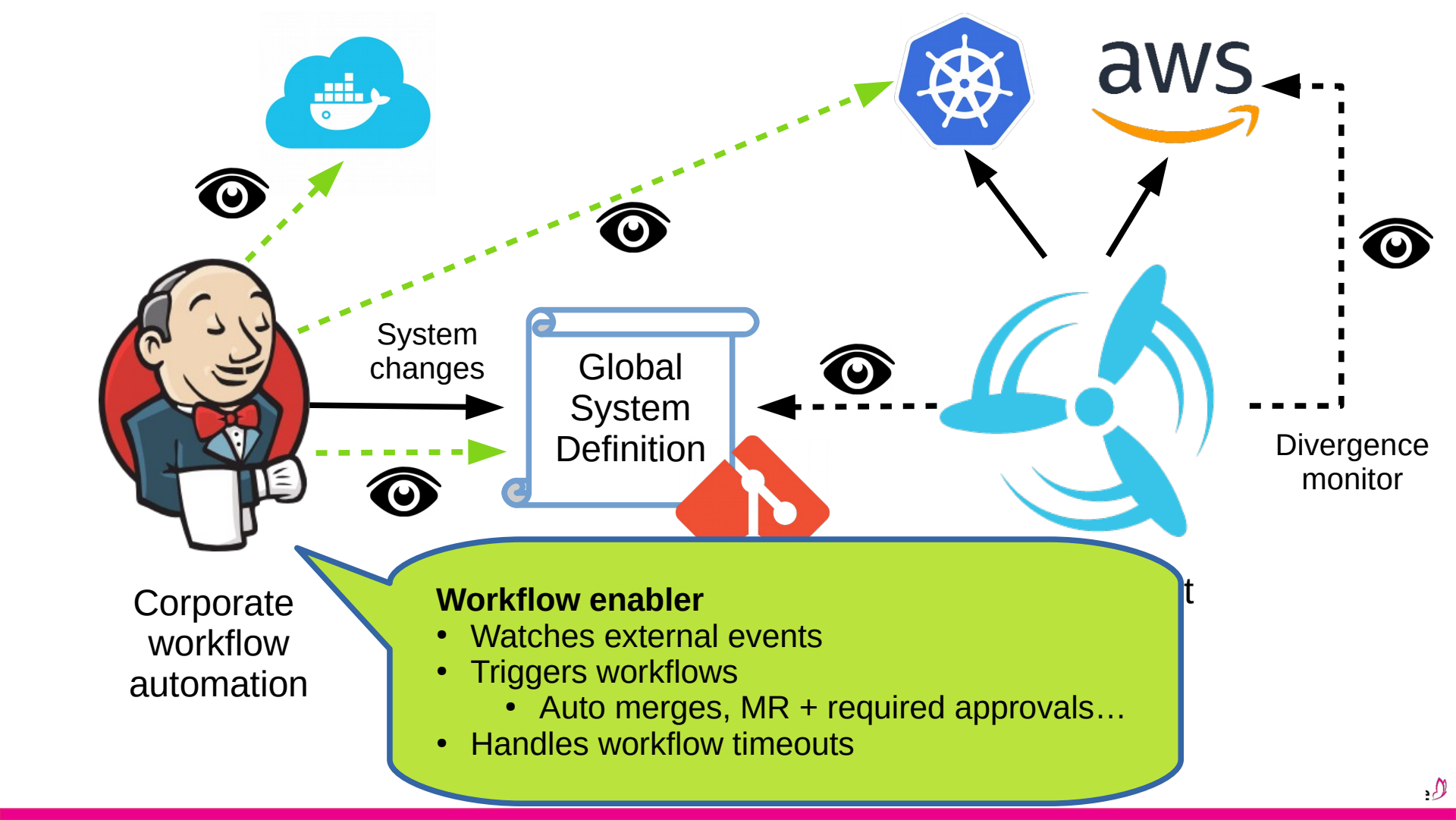
Corporate workflow automation

Not just Git!

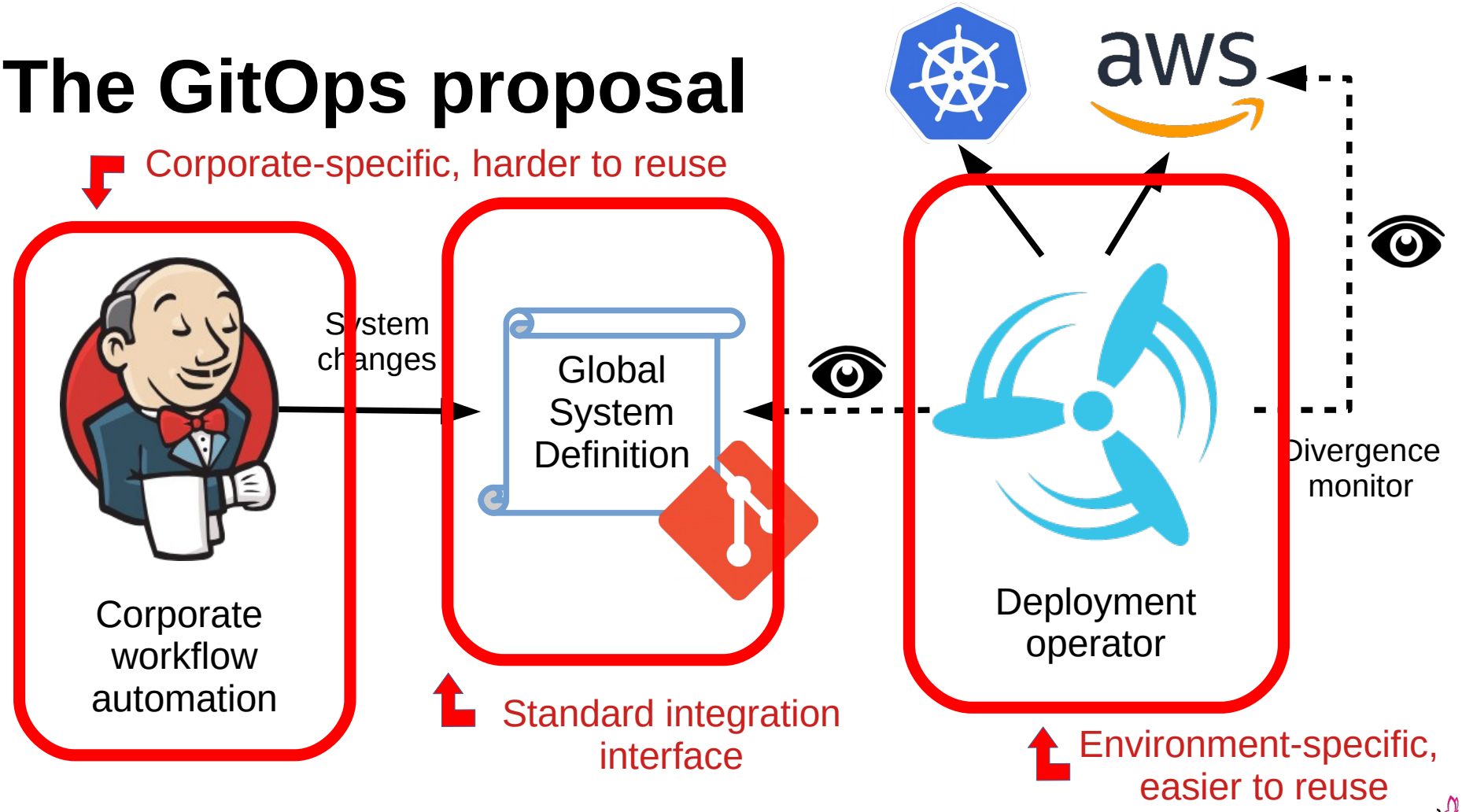
- Can be just Git...
- But for complex workflows it’s much more convenient to have MR/PR, approvals...
- So use GitHub, GitLab or equivalent!

The GitOps proposal

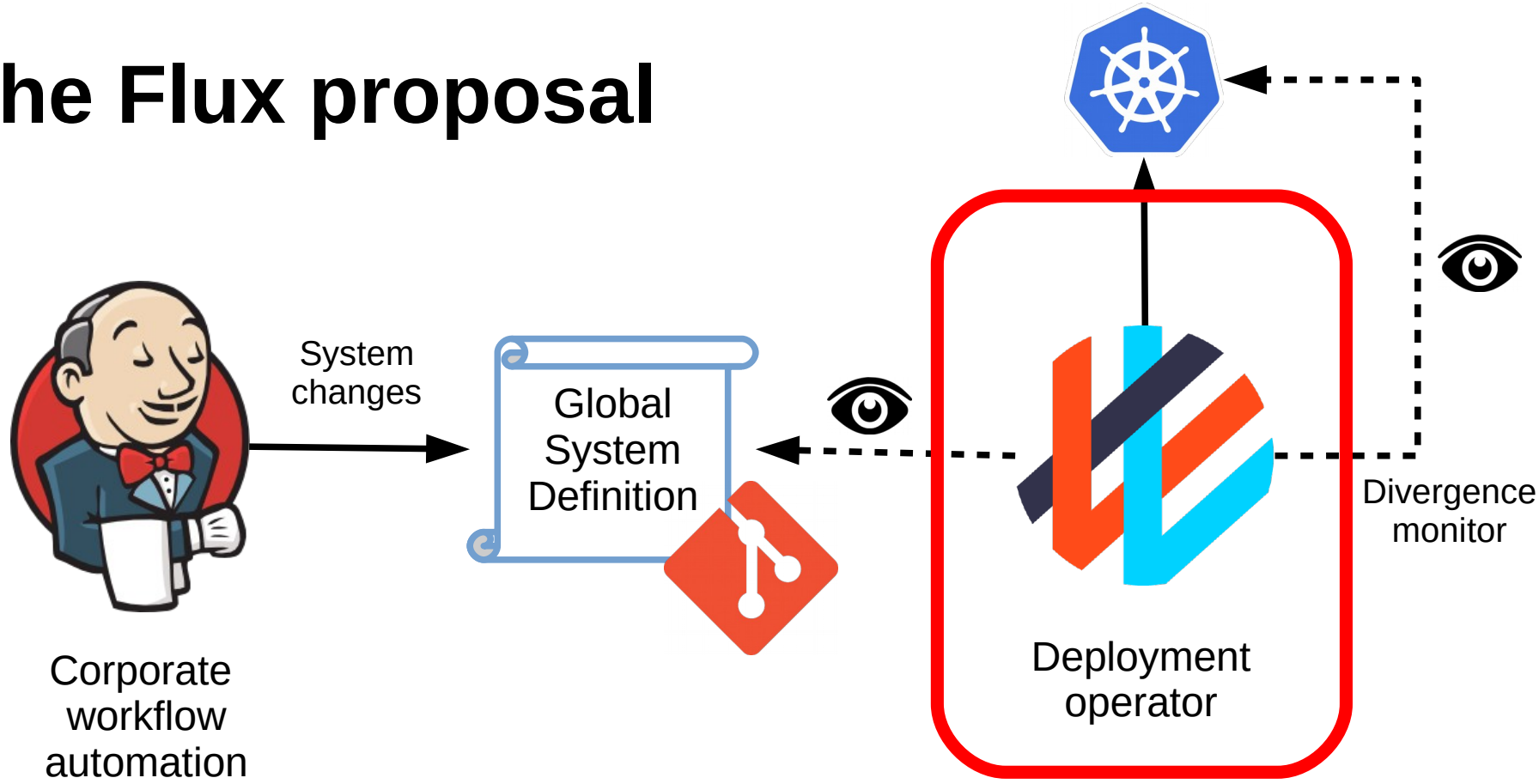



aws
System changes
Global System Definition
Divergence monitor
Corporate workflow automation
Workflow enabler
• Watches external events
• Triggers workflows
• Auto merges, MR + required approvals…
• Handles workflow timeouts

The GitOps proposal

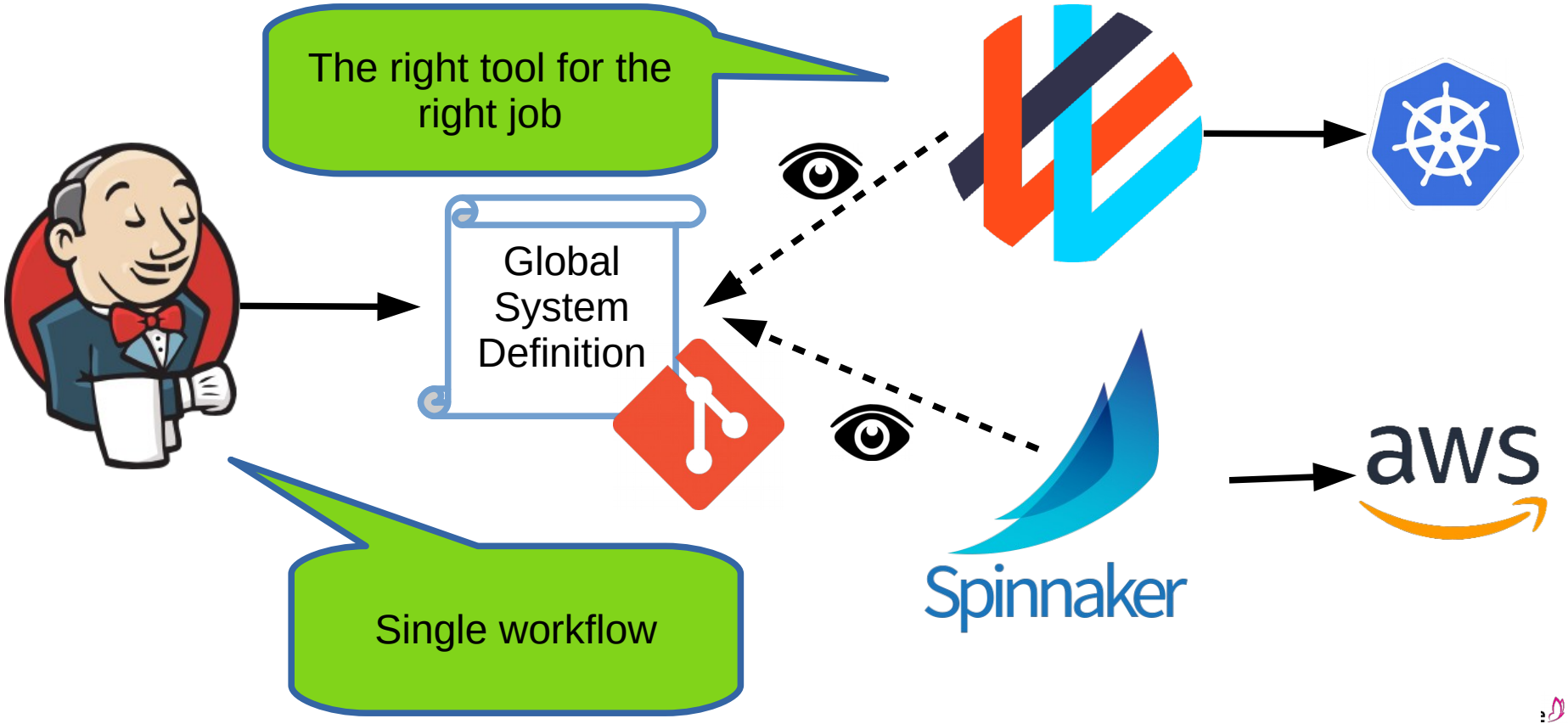


The Flux proposal

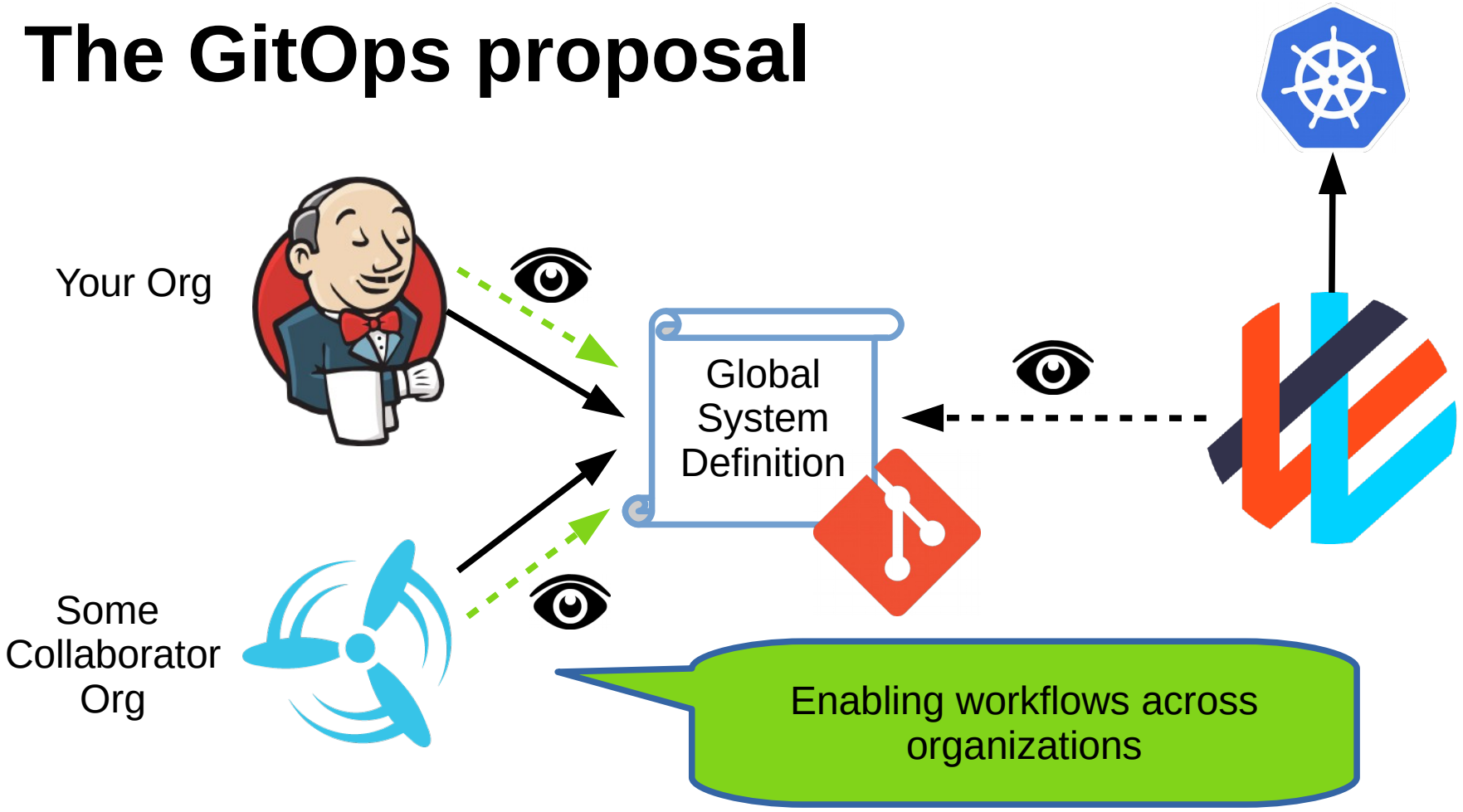


Kubernetes-specific, easier to reuse.
Kubernetes-specific concerns.

The GitOps proposal

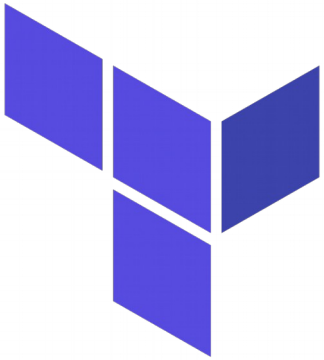


The GitOps proposal



GitOps practices

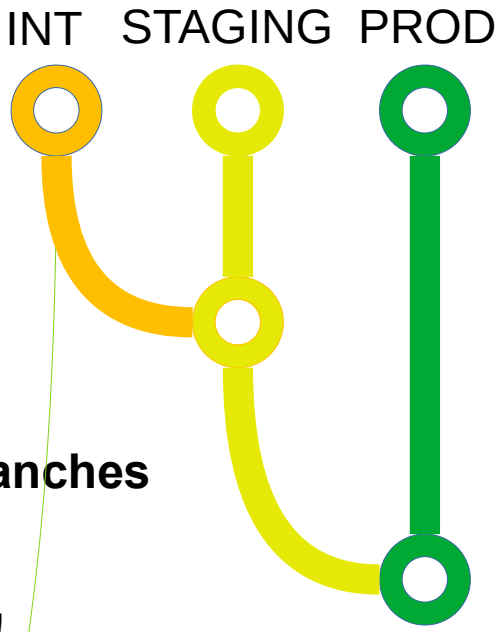
- **Global System Definition in Git**
  - Expressed in declarative form using any appropriate tool



GitOps practices

- **Git as the system's Single Source of Truth**
  - All the config in Git:
    - **Application config**
    - **Infrastructure config**
      - K8s templates...
    - **Observability config**
      - Grafana dashboards definitions...
    - **Controllability config**
      - Automated operation playbooks...
        - Extra value if integrated with Obs tools (actionable dashboards)

GitOps practices



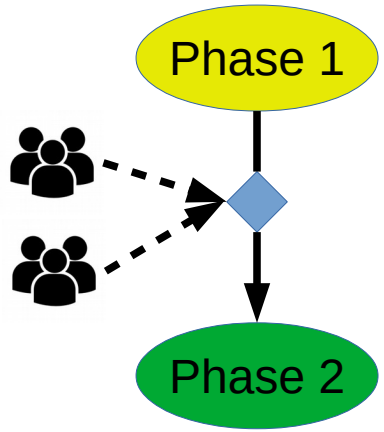
- **Branch by Environment**
  - Production env config in **master** branch
  - Other branches for previous envs
  - **Promotion between envs = PR + Merging env branches**
    - e.g. Merging INTEGRATION → STAGING
    - **No direct commits allowed on env branches!**
    - **Brings Git workflow capabilities to system evolution**
      - Prevents “race conditions” during deployments
      - Forces teams to integrate system upgrades
        - Prevents accidental regressions

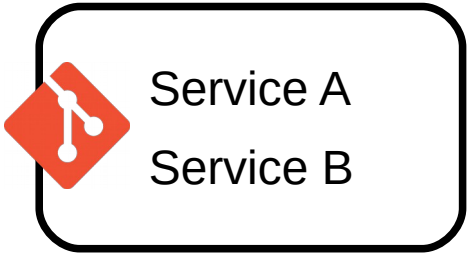
GitOps practices

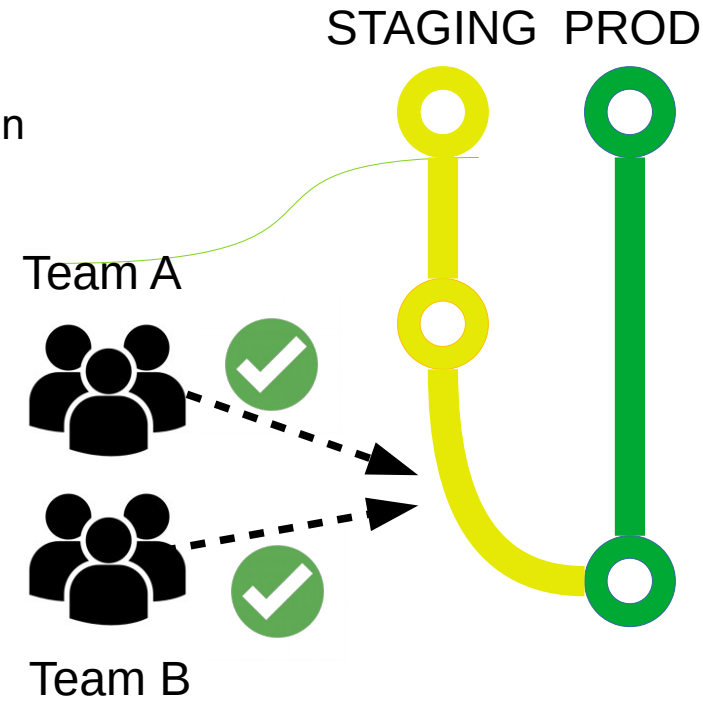
Git Flow applied to system evolution

GitOps practices

- **Branch by Environment**
  - **Workflow modeling using Git**
    - **Branches**
      - Can be used to model **stages** in the workflow
    - **Merge Requests**
      - Can be used to model **transitions** between stages
    - **MR approvals**
      - Can be used to model **conditions or inputs** for transitions







GitOps practices

- **Divergence control**
  - Git == Single Source of Truth

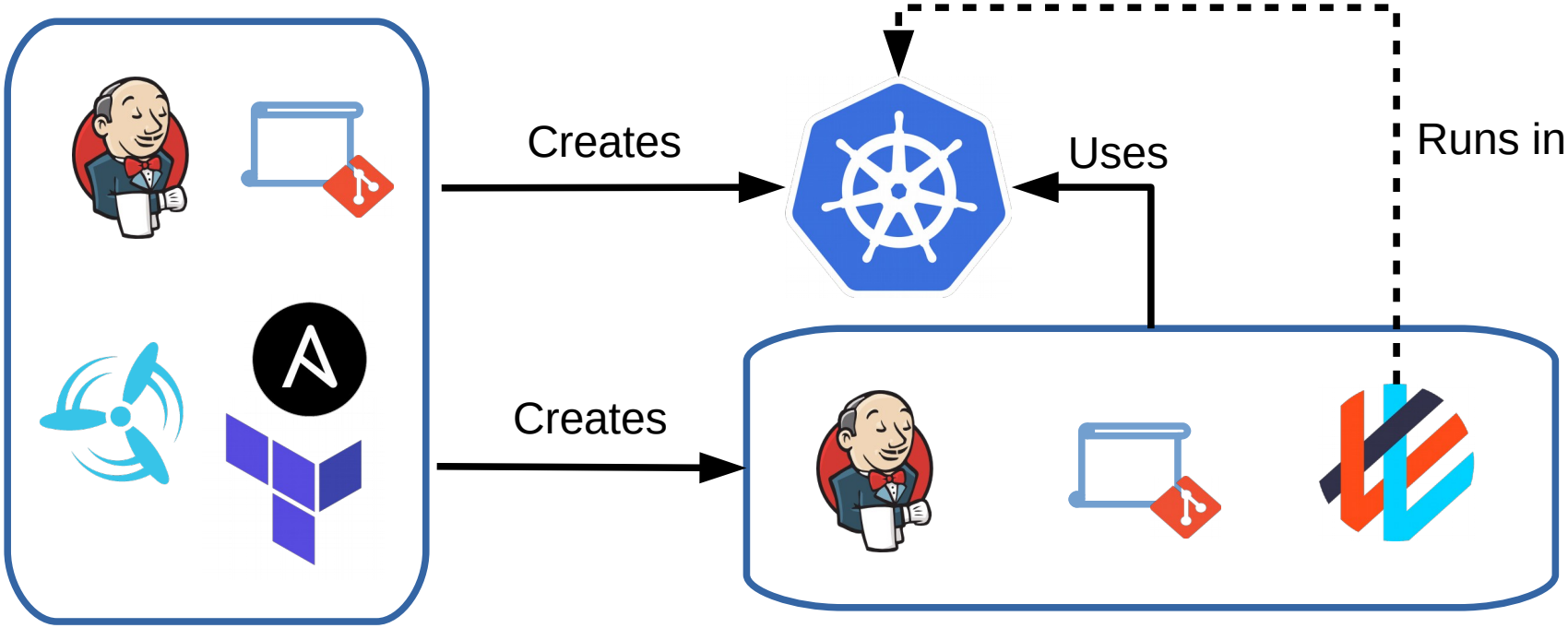
    → **Divergence == Danger**
    - We can’t reproduce the system anymore
  - Use **agents** in the target environments to watch for divergence

    → **Alerting**

    → **Automated convergence (if applicable/possible)**
    - i.e. Rolling back to last accepted GSD state

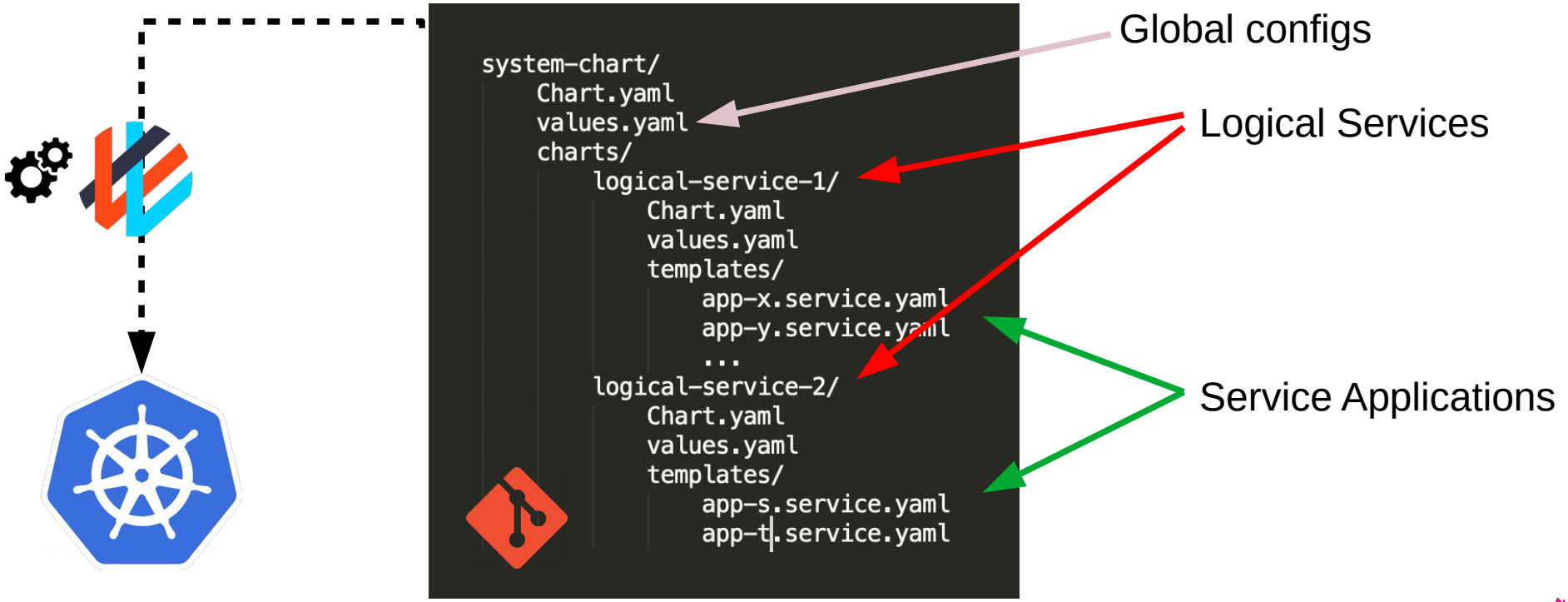
GitOps practices

- **GitOps 4 GitOps**
  - Your environments & GitOps platform also needs consistency!



GitOps 4 Kube

GitOps example code structure: Deep Tree System Chart



GitOps 4 Kube

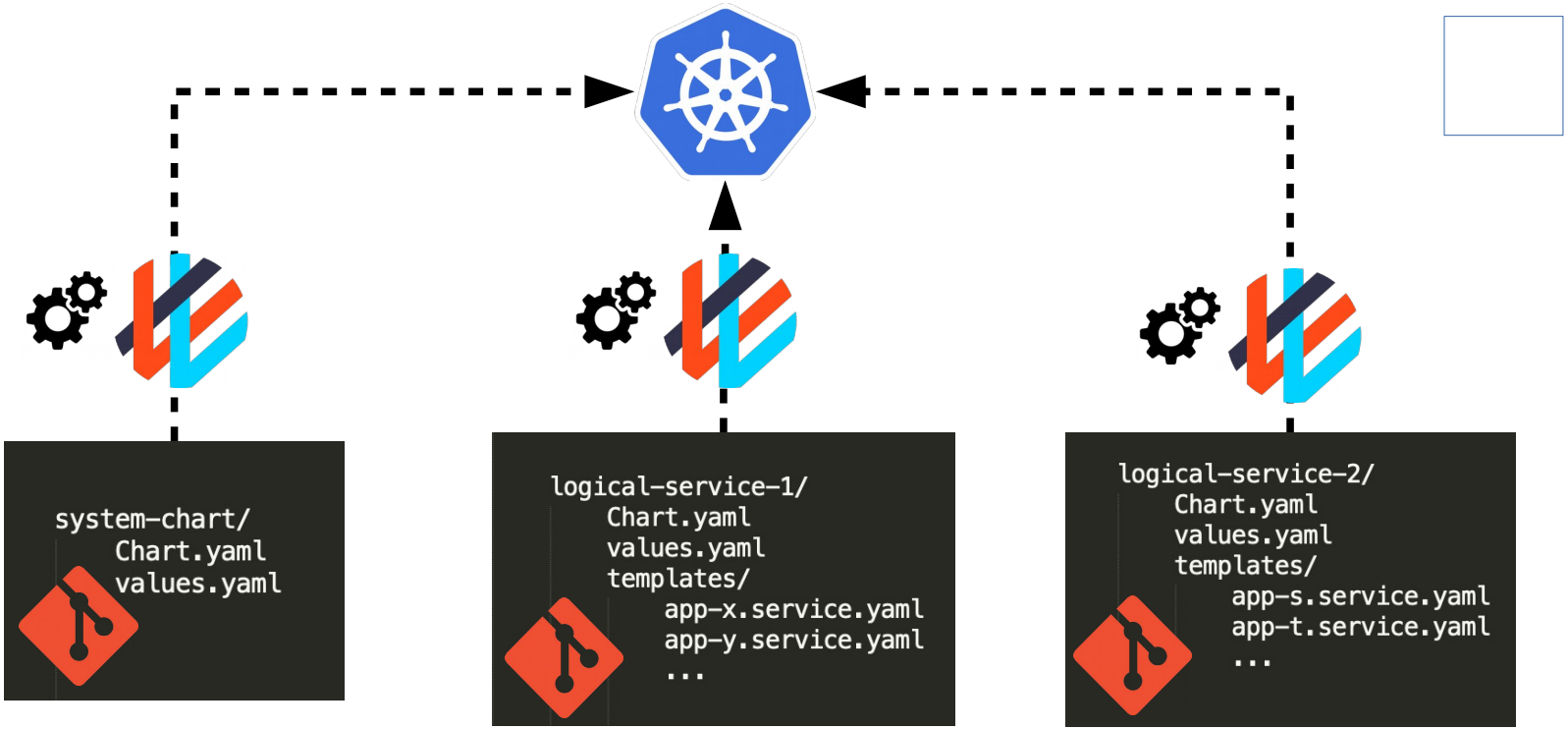
GitOps example code structure: Deep Tree System Chart

```
system-chart/
    Chart.yaml
    values.yaml
    charts/
        logical-service-1/
            Chart.yaml
            values.yaml
            templates/
                app-x.service.yaml
                app-y.service.yaml
                ...
        logical-service-2/
            Chart.yaml
            values.yaml
            templates/
                app-s.service.yaml
                app-t.service.yaml
```

- **Pro:**
  - Single Git repo
  - Single GitOps pipeline
- **Con:**
  - Can get cumbersome if Merge Requests require several manual approvals to merge into Production branch
    - e.g. LS-1 deploying changes all the time, and every MR requiring also the approval of LS-2
    - e.g. Doc changes in LS-1 require MR

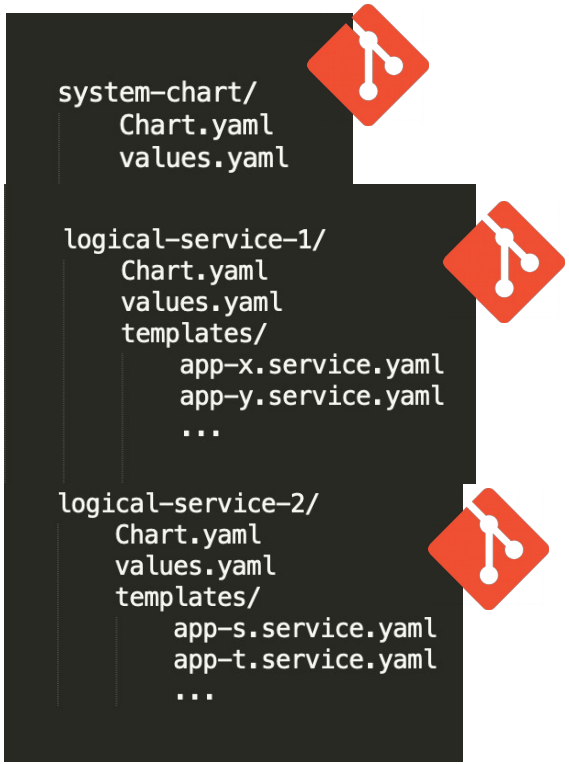
GitOps 4 Kube

GitOps example code structure: Pipeline per Logical Service



GitOps 4 Kube

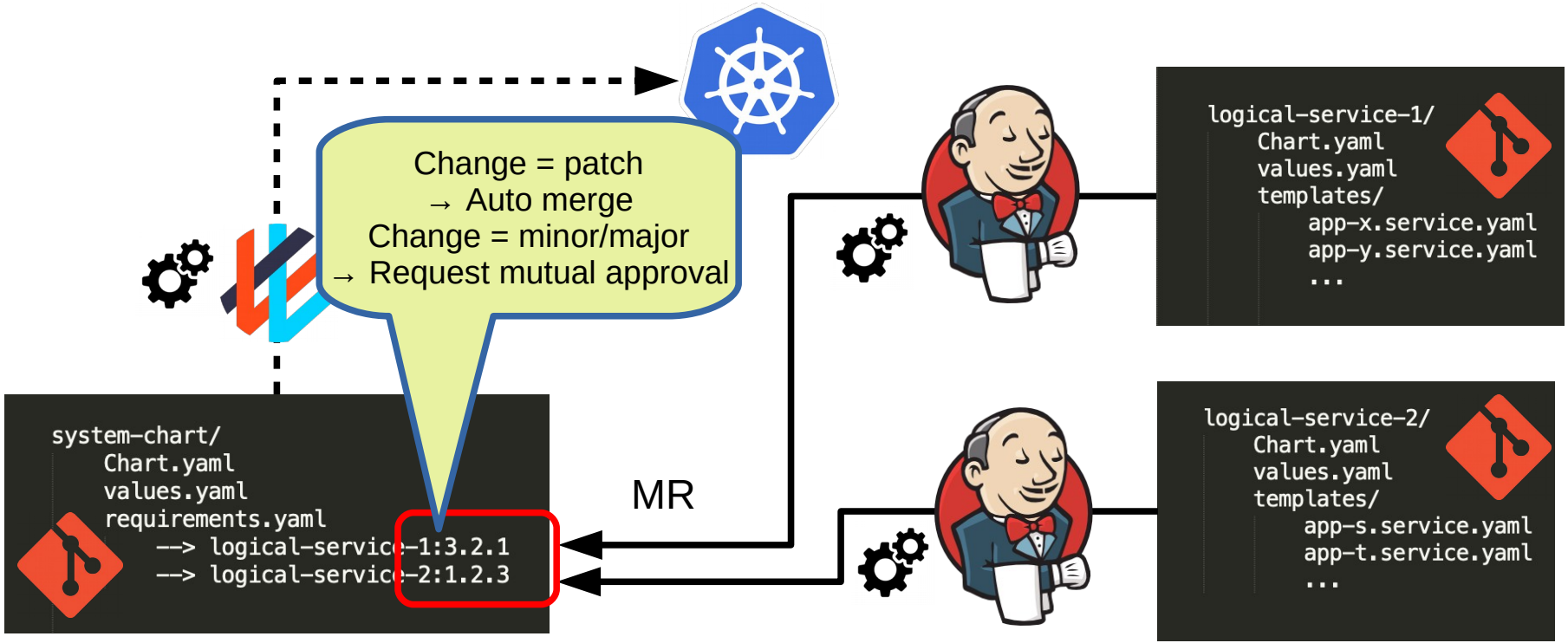
GitOps example code structure: Pipeline per Logical Service



- **Pro:**
  - Separate Git repos for every chart
    - e.g. doc changes in LS chart don't require MR in SystemChart
  - Simpler MR approval in each Git repo
    - As the Git repo belongs to the team owning each Logical Service, approving the MR only requires their own approval
- **Con:**
  - If the LS depend on each other, incompatible versions of both LS could end up deployed in Production

GitOps 4 Kube

GitOps example code structure: Chained pipelines



GitOps 4 Kube

GitOps example code structure: Chained pipelines

```
logical-service-1/
    Chart.yaml
    values.yaml
    templates/
        app-x.service.yaml
        app-y.service.yaml
        ...
```

```
logical-service-2/
    Chart.yaml
    values.yaml
    templates/
        app-s.service.yaml
        app-t.service.yaml
        ...
```

```
system-chart/
    Chart.yaml
    values.yaml
    requirements.yaml
        --> logical-service-1:3.2.1
        --> logical-service-2:1.2.3
```

- **Pro:**
  - Separate Git repos for every chart

    → Doc changes don’t require a MR in SystemChart
- **Con:**
  - More complex setup, CI pipelines must create a MR to SystemChart to bump up the versions of the LS charts
    - Alternatively, when bumping just patch versions the MR could be automatically merged

GitOps benefits

https://www.weave.works/technologies/gitops/

1) ~~**Increased Productivity**~~

   – Faster deployments  **CD benefit**

2) ~~**Enhanced Developer Experience**~~

3) **Improved Stability**

4) **Higher Reliability**

5) **Consistency and Standardization**

6) **Stronger Security Guarantees**

GitOps benefits

https://www.weave.works/technologies/gitops/

1) ~~**Increased Productivity**~~

2) ~~**Enhanced Developer Experience**~~

   – Devs & Ops only use Git  **CD benefit**

3) **Improved Stability**

4) **Higher Reliability**

5) **Consistency and Standardization**

6) **Stronger Security Guarantees**

GitOps benefits

https://www.weave.works/technologies/gitops/

1) ~~**Increased Productivity**~~

2) ~~**Enhanced Developer Experience**~~

3) **Improved Stability** 

   – Full system change log in Git → Easier troubleshooting

   (both for deployments & operations)

4) **Higher Reliability**

5) **Consistency and Standardization**

6) **Stronger Security Guarantees**

GitOps benefits

https://www.weave.works/technologies/git

1) ~~**Increased Productivity**~~

2) ~~**Enhanced Developer Experience**~~

3) **Improved Stability** 

4) **Higher Reliability** 

   – Full system change history in Git

   → Can recreate system. **Can rollback accurately.**

5) **Consistency and Standardization**

6) **Stronger Security Guarantees**

Really, George???
Be careful with data...

GitOps benefits

https://www.weave.works/technologies/gitops/

1) ~~**Increased Productivity**~~

2) ~~**Enhanced Developer F**~~

3) **Improved Stability** 

4) **Higher Reliability** 

5) **Consistency and Standardization** 

– Consistent workflow for system changes by using Git

– Git as the standard workflow tool, easy to hook-in external tools

6) **Stronger Security Guarantees**

GitOps benefits

https://www.weave.works/technologies/gitops/

1) ~~**Increased Productivity**~~

2) ~~**Enhanced Developer Experience**~~

3) **Improved Stability** 

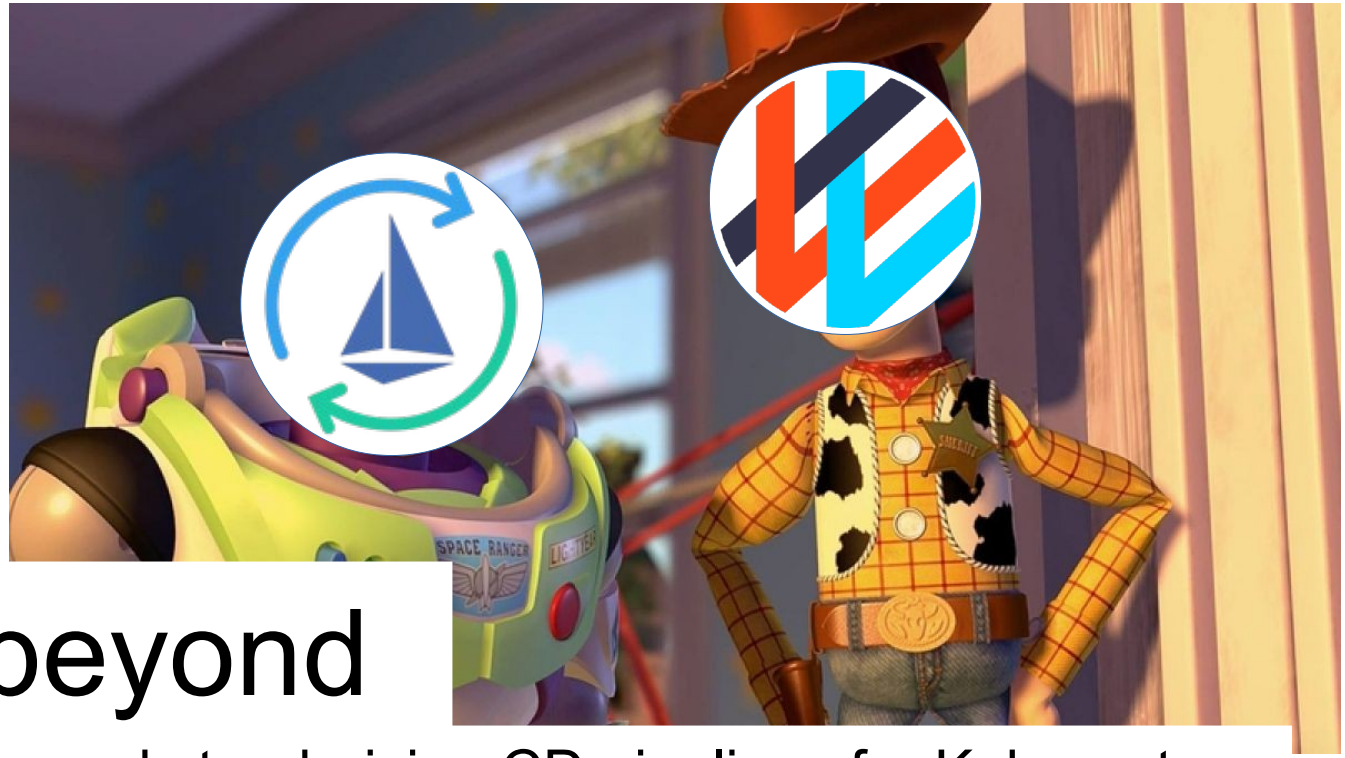
4) **Higher Reliability** 

5) **Consistency and Standardization** 

6) **Stronger Security Guarantees** 

– Sys changes only through Git (requires auth & authoriz)

→ **Auditability of system changes**



Flux & beyond

Automating and standarizing CD pipelines for Kubernetes with the right mix of centralization & self-service

The Flux proposal

Corporate-specific, harder to reuse

System changes

Global System Definition

Corporate workflow automation

Flux

Divergence monitor

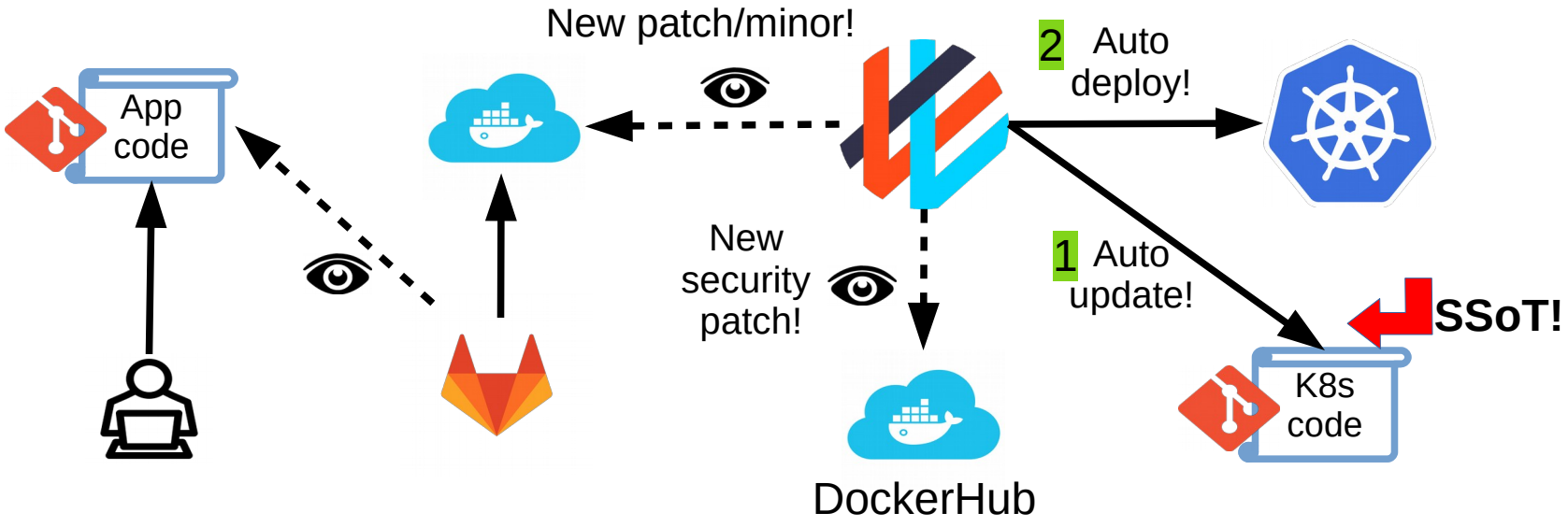
Kubernetes-specific, easier to reuse.
Kubernetes-specific concerns.

Weaveworks Flux

- **Implements a GitOps CD pipeline for Kube & Helm**
- **Flux = Helm Operator + Deployment Operator**
- **Deployment Operator**
  - Watches 1 Git repo+branch for changes
    - Note: Maybe in the future Flux will support 1 instance → * repo+env
    - **Need 1 DO instance per repo & environment**
  - Can deploy:
    - Kube workloads: CronJob, DaemonSet, Deployment, StatefulSet
    - Helm workloads: HelmRelease (CRD)

Weaveworks Flux

- **Deployment Operator**
  - **Can auto scan container registries for updated images of installed workloads & redeploy them**



Weaveworks Flux

- **Deployment Operator**
  - **Can auto scan container registries for updated images of installed workloads & redeploy them**
    - **USEFUL !!!**
      - Reduces Kube routine operation efforts (less MRs)
      - **Improves system security** → Auto installs latest patches
      - **Simplifies workflows by skipping MR approvals in some cases**
        - e.g. auto install app patch & minor upgrades, MR for majors
      - Can be fine tuned
        - e.g. auto install only images tagged as `*-release`
    - **Git SSoT: After auto deploy, the workload is modified & committed**
    - Drawback: We loose some control about system changes (but tolerable)

DEMO #5-1

Automated Kube releases with Flux

DEMO #5-2

GitOps workflows with Flux

DEMO #5-3

Controlling workloads with Flux

A Secret problem

What about Secrets?

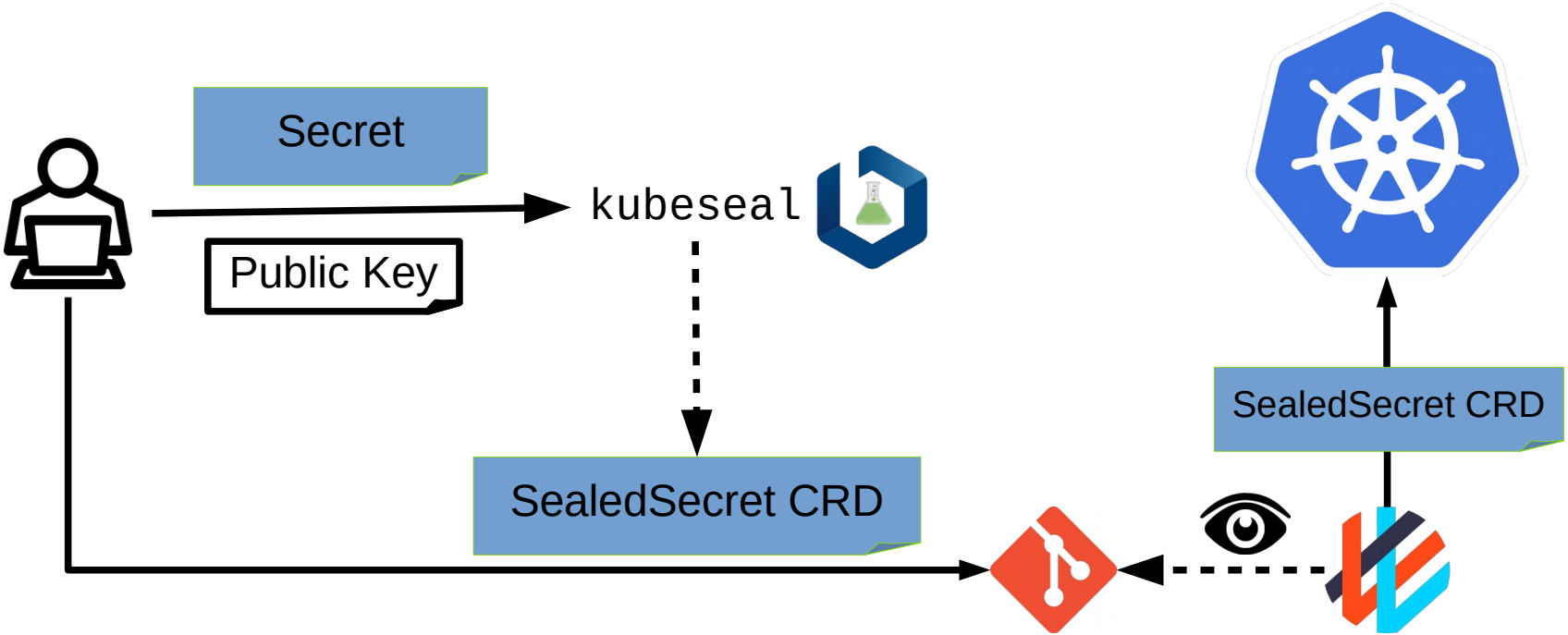
- **To keep the GitOps benefits, Secrets need also to be stored in Git**
  - They can be stored externally (e.g. using Vault)
    - But then we loose the “reliable rollback” capability
      - e.g. Upgrade workload, change secret in Vault, rollback
        → Cannot recover old secret
        - New secret may lack some key/values of the old
    - Or we need to complicate the workflow to synch with Vault
- **If we want to keep them in Git, however, they can’t be in plain text**

What about Secrets?

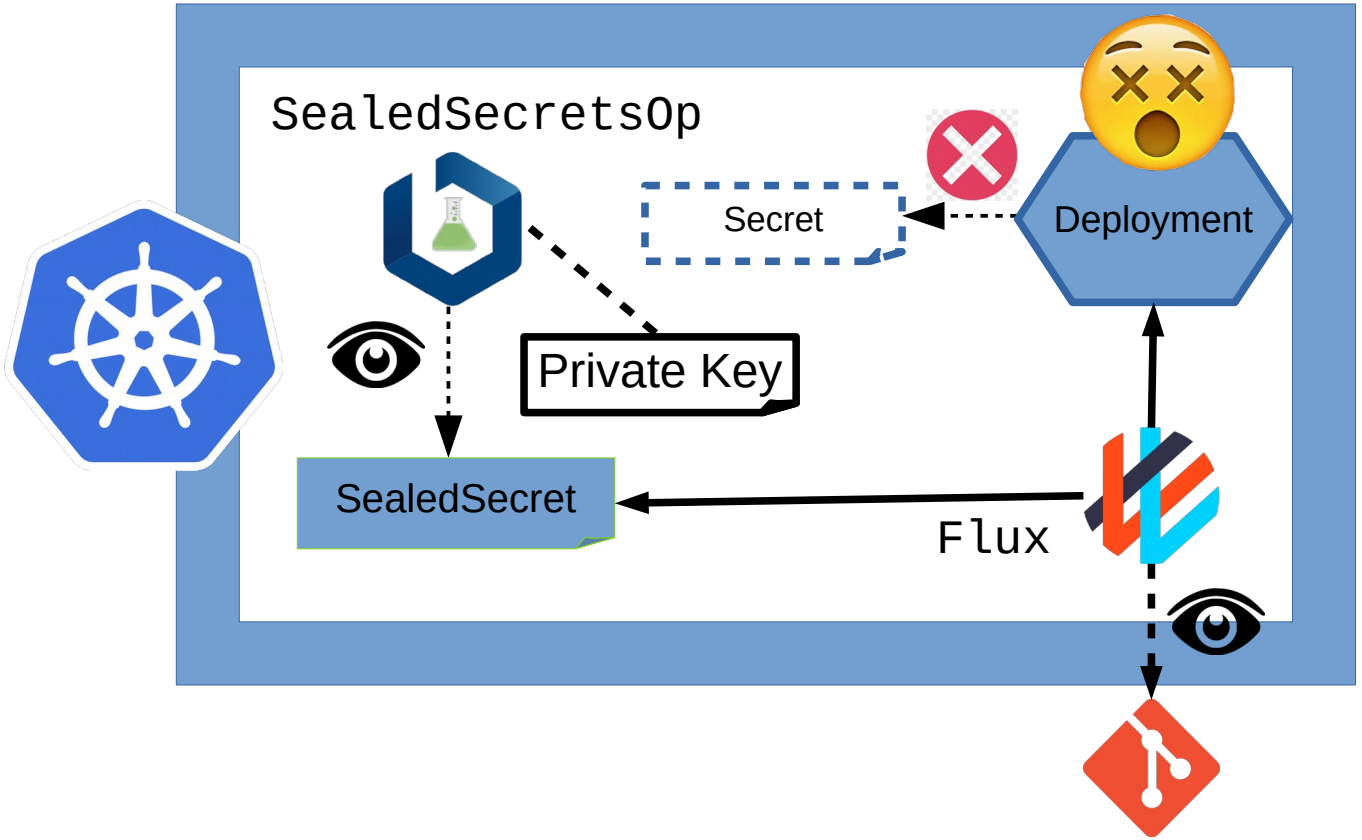
- **Option #1 – Bitnami's SealedSecrets Operator**
  - See: https://github.com/bitnami-labs/sealed-secrets/
  - An early solution by Bitnami to solve the Secret problem

What about Secrets?

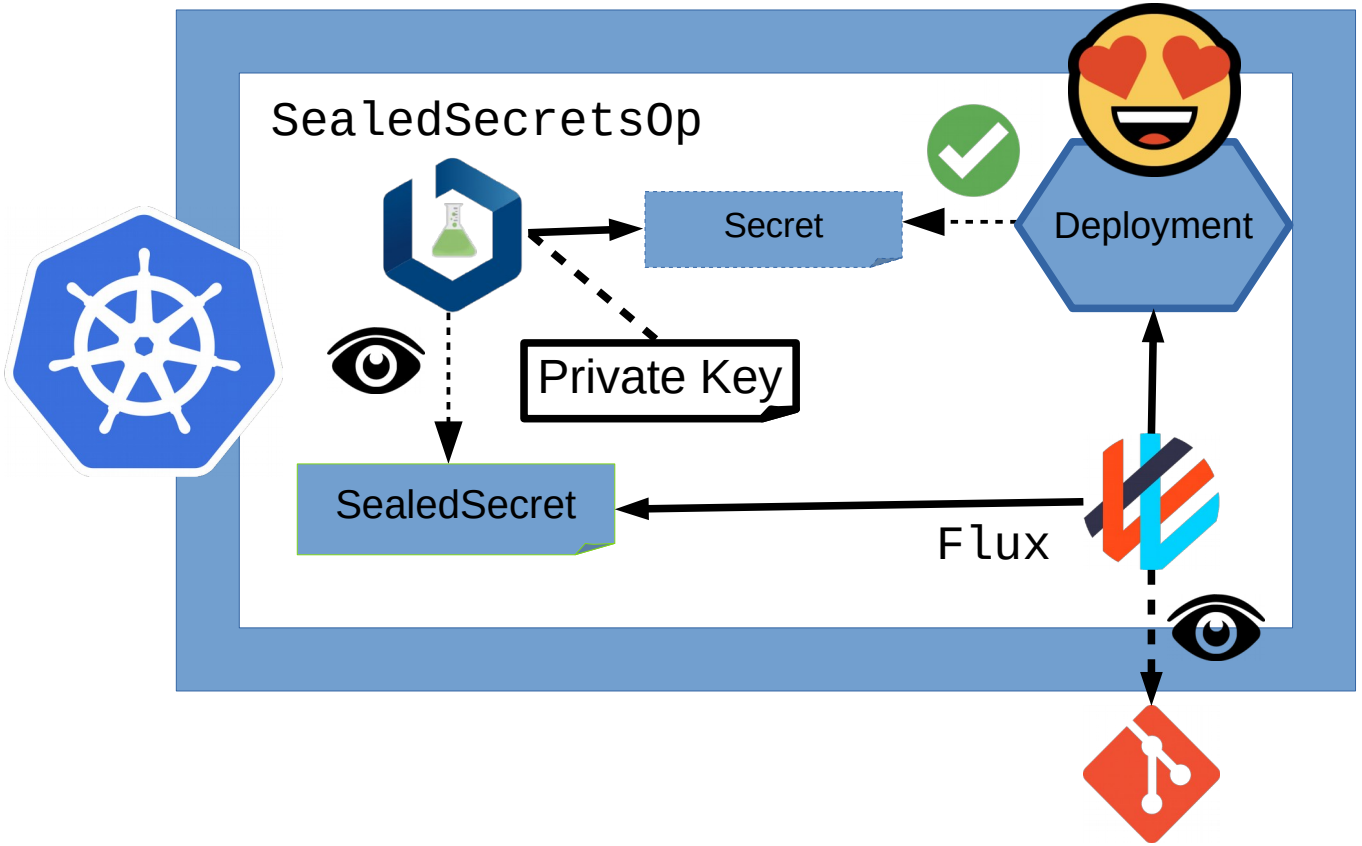
- **Option #1 – Bitnami's SealedSecrets Operator**



Option #1 – Bitnami's SealedSecrets Operator



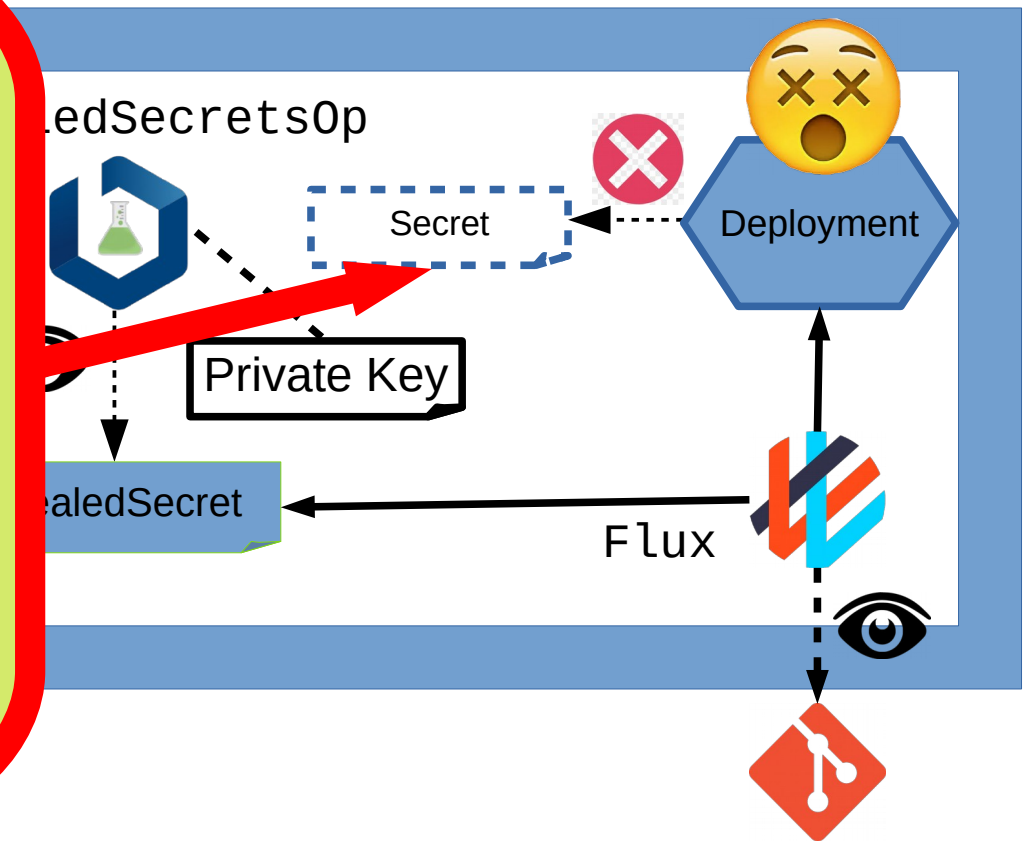
Option #1 – Bitnami's SealedSecrets Operator



Option #1 – Bitnami's SealedSecrets Operator

Drawbacks

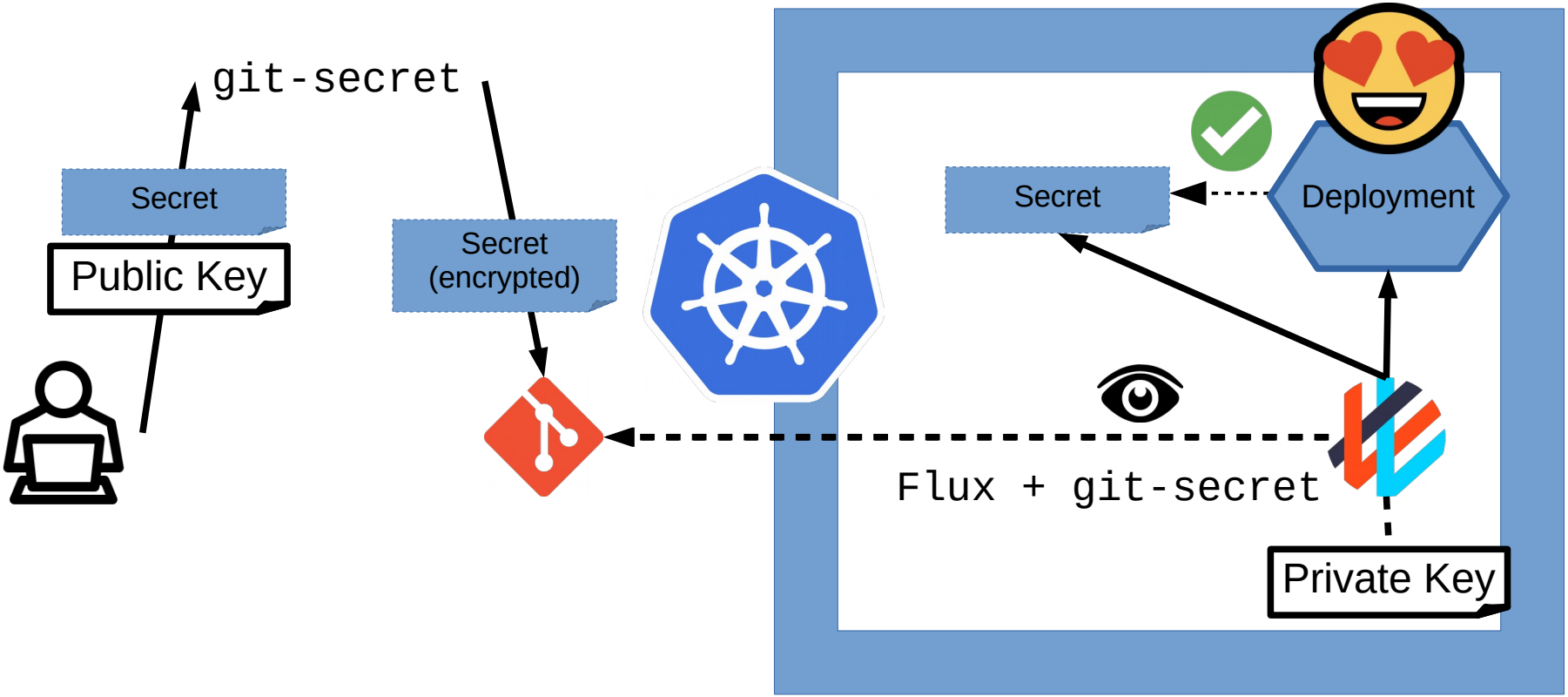
- **SSOp: Another moving part of the system...**
- **SSOp & Flux disconnected from each other**
  → Depending on config, Flux may rollback the deployment because of the missing Secret!
- **SSOp: Another Operator…**
  → **Increases "limbo time"**



What about Secrets?

- **Option #2 – Flux + git-secret**
  - See:
    - https://github.com/fluxcd/flux/pull/2159 ← **Very recent feature!**
    - https://git-secret.io/ ← Transparent enc/dec for Git
  - Manifest files with Secrets stored in encrypted form in Git repo
  - Flux decrypts Secrets with git-secret before installing them in K8s

Option #2 – Flux + git-secret



DEMO #5-4

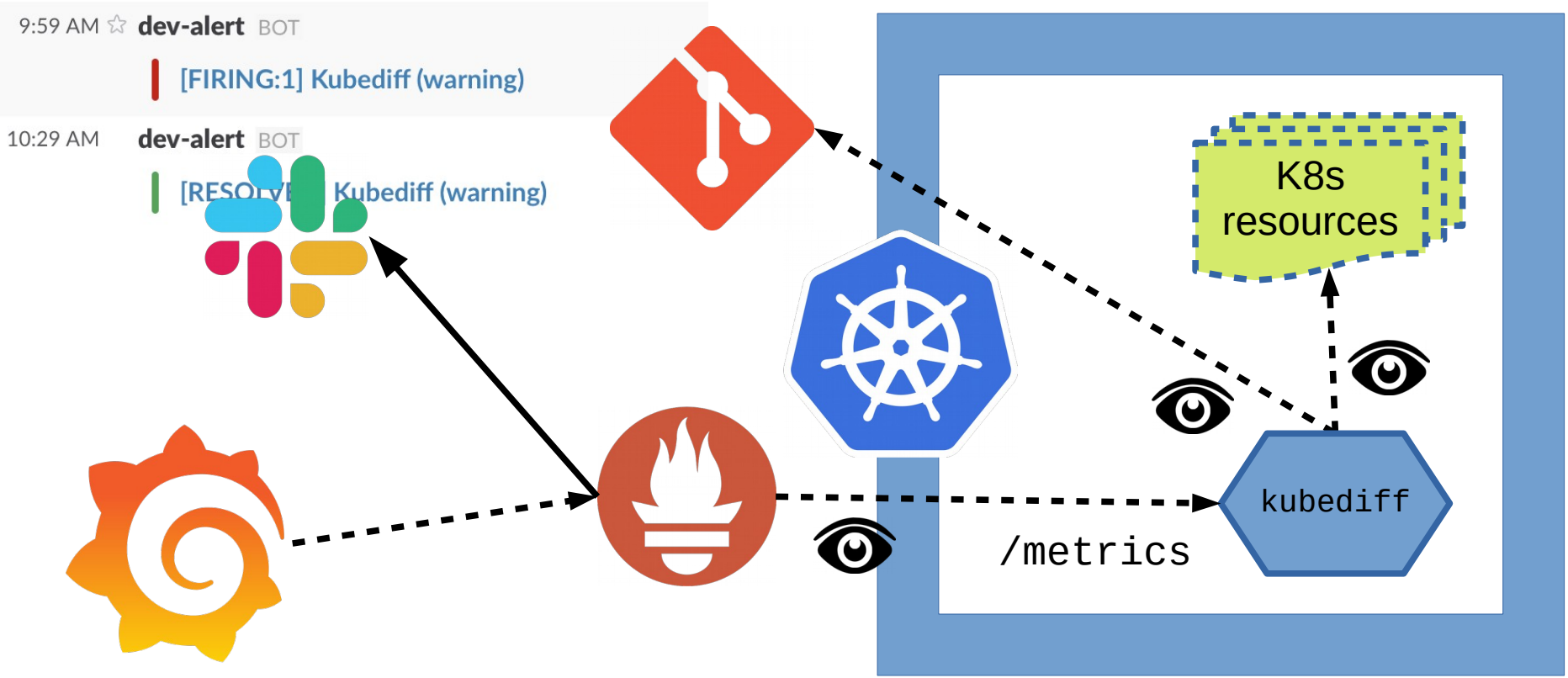
Flux + git-secret

A Secret “no problemo”

What about System divergence?

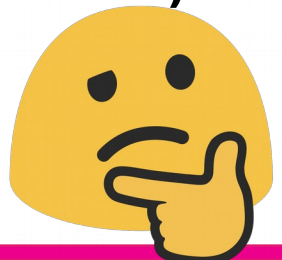
- Weaveworks works currently with three main techs
- For each one they use a matching tool for system divergence control:
  - **K8s → kubediff:** https://github.com/weaveworks/kubediff
  - **Terraform → terradiff**
    - Internal tool, wrapper for standard TF CLI “plan” command
  - **Ansible → ansiblediff**
    - Internal tool, wrapper for standard Ansible “check mode”

Kubediff


9:59 AM dev-alert BOT
[FIRING:1] Kubediff (warning)
10:29 AM dev-alert BOT
Kubediff (warning)
K8s resources
kubediff
/metrics

Current Status

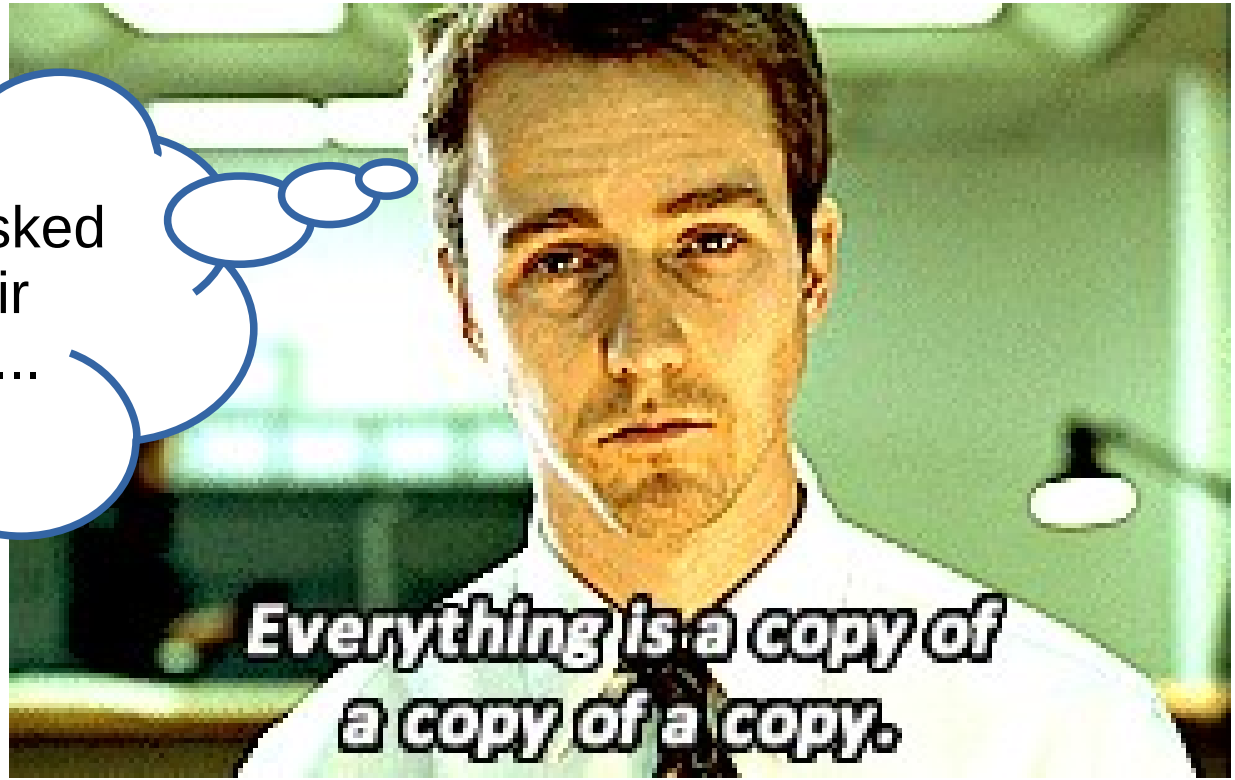
- kubectl-less operations
  - Offloading of Helm Release installation to CD (Flux)
- Guaranteed reproducibility by using Git as SSoT
  - Solved the problem of Secrets in Git
- Separated corp workflow from env deployment (CD reuse)



Can we do better?

New first world problems...

Team Kolontái asked me to update their Flux configuration... **Again.**



Flux friction points

- **Flux instance per Git repo + env (branch)**
  - Note: Future = 1 inst → * repo+env
- **If Ops handling the full Flux installation:**
  - Devs requiring new or updated Flux configurations

    → **Ops overload**
- **To avoid Ops overload**

  → Devs given full control of Flux installation

  → Devs with access to sec creds & config options they shouldn't have

  → **Not DevSecOps'ey !!**

Flux separation of concerns by role

Ops (aka “the Platform team”) : Setting up Flux defaults

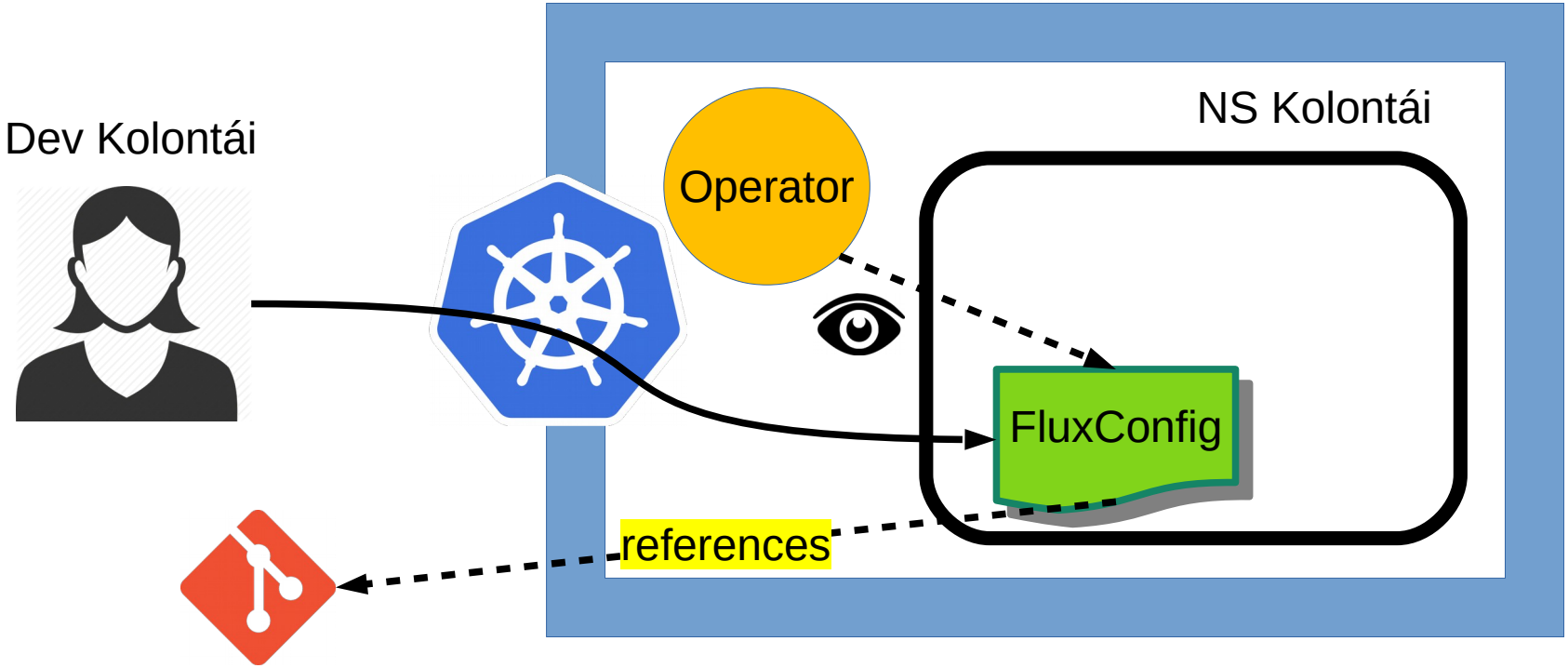
- **Container image repo management**
  - Allowed corporate/external repos
  - Access credentials
  - Image/Chart white/black listing & caching
  - ...
- **System Def Git repo management**
  - Git repo whitelisting
  - Access credentials
  - ...
- **Deployment target setup**
  - Kube clusters + namespaces
  - Credentials
  - …

Flux separation of concerns by role

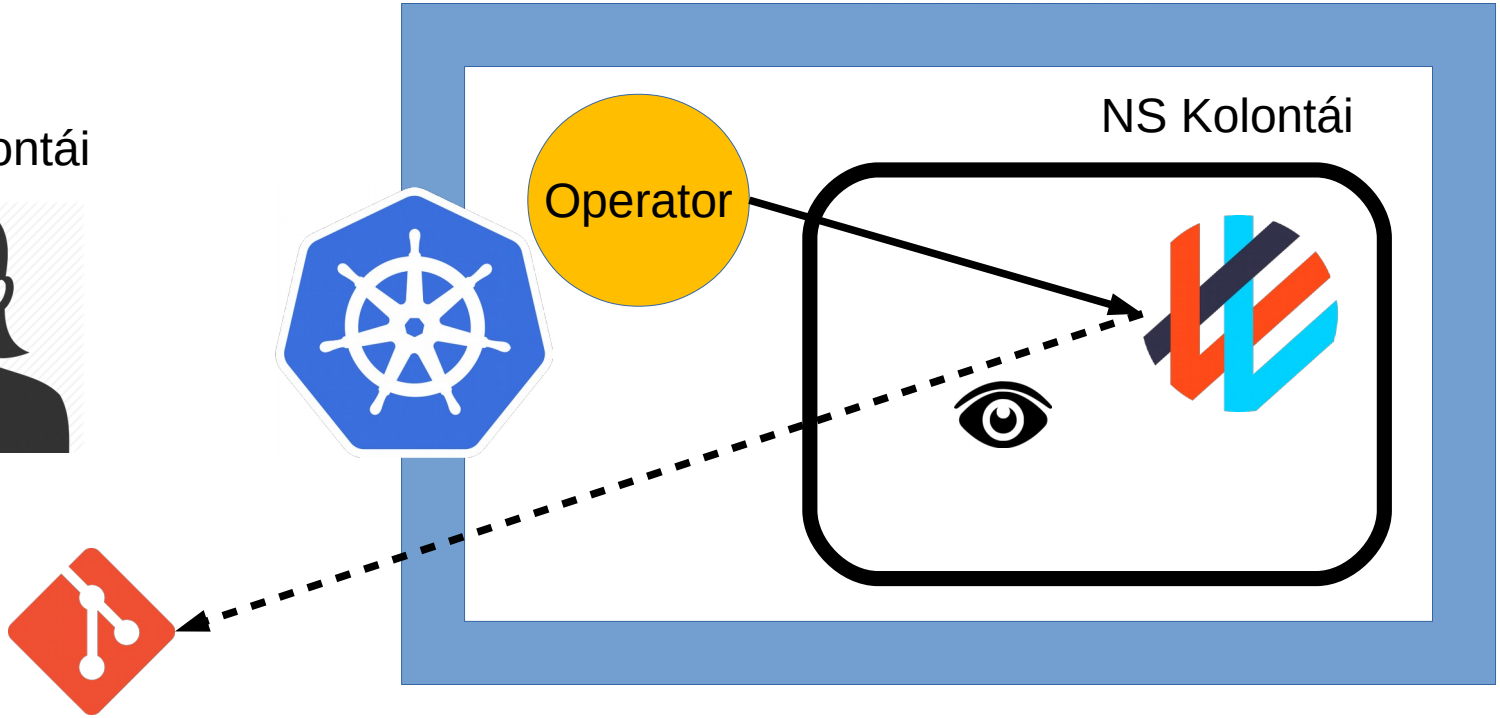
Devs : Configuring specific Flux use cases

- **Specifying what to deploy**
  - i.e. The source Git repo K8s configs
- **Specifying where to deploy**
  - i.e. Identifying the target K8s cluster by a logical ID
    - And the target namespace,
      (if using 1 cluster → * environments)

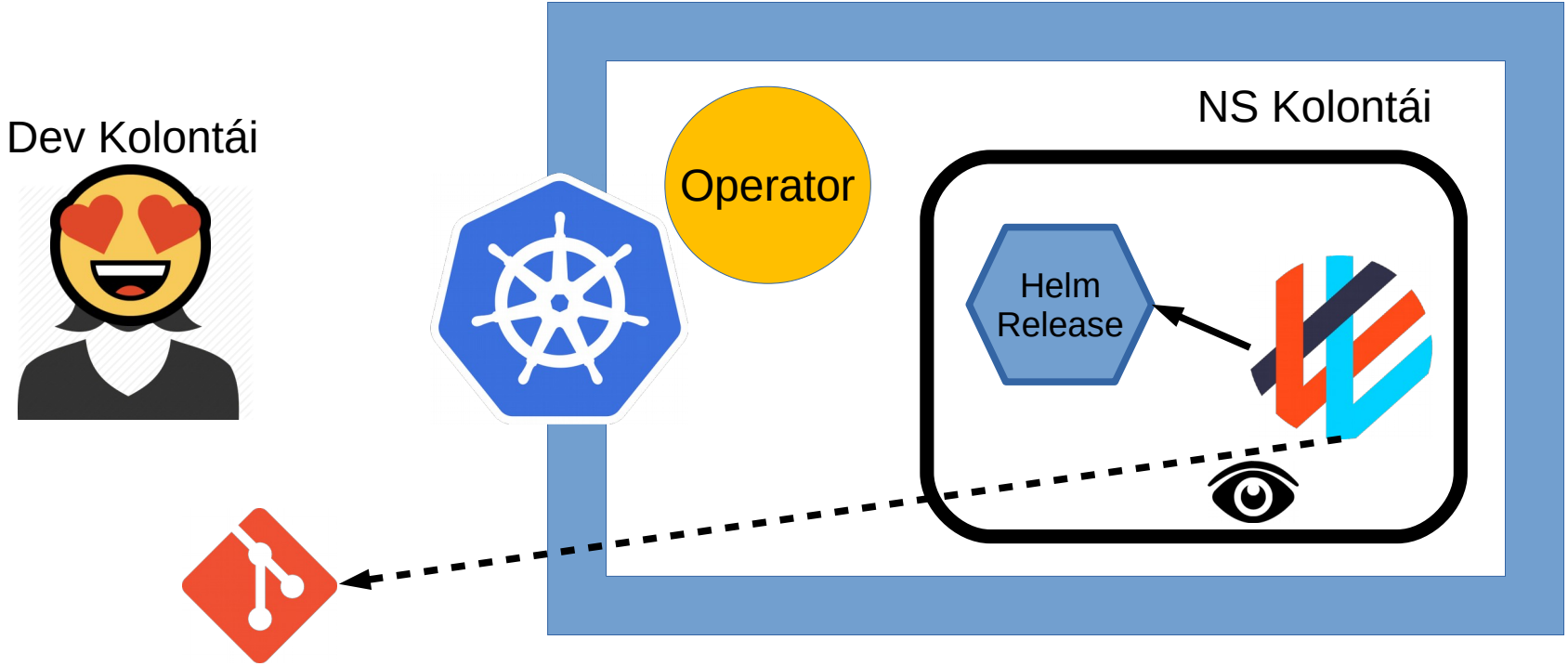
Wouldn’t it be great if...



Wouldn’t it be great if...



Wouldn’t it be great if...

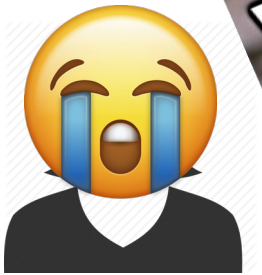


Quick quiz:

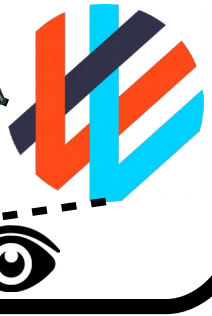
Looks DevOps'ey, doesn't it?

Wouldn’t it be great if.

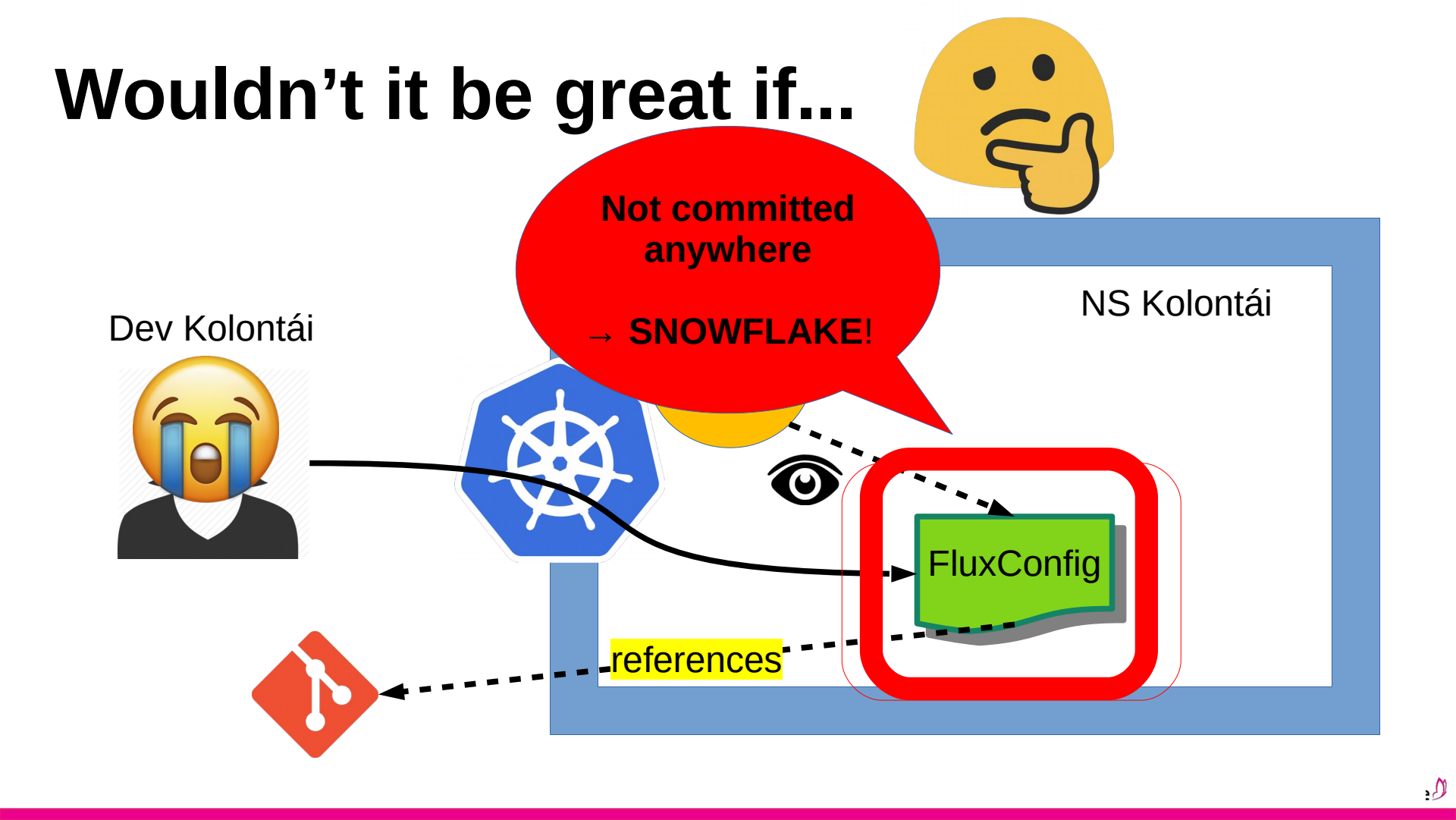
Dev Kolontái



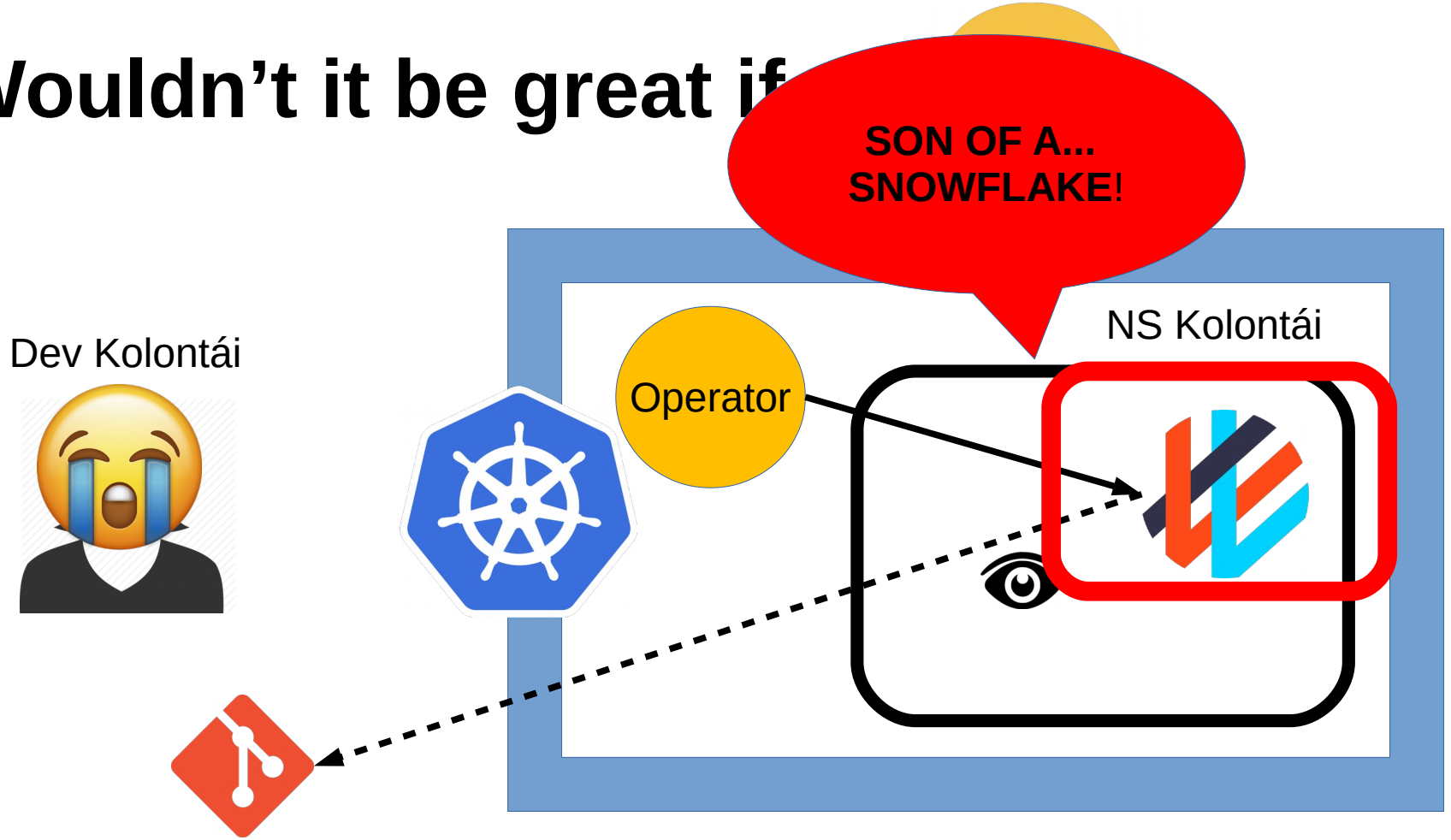
NS Kolontái



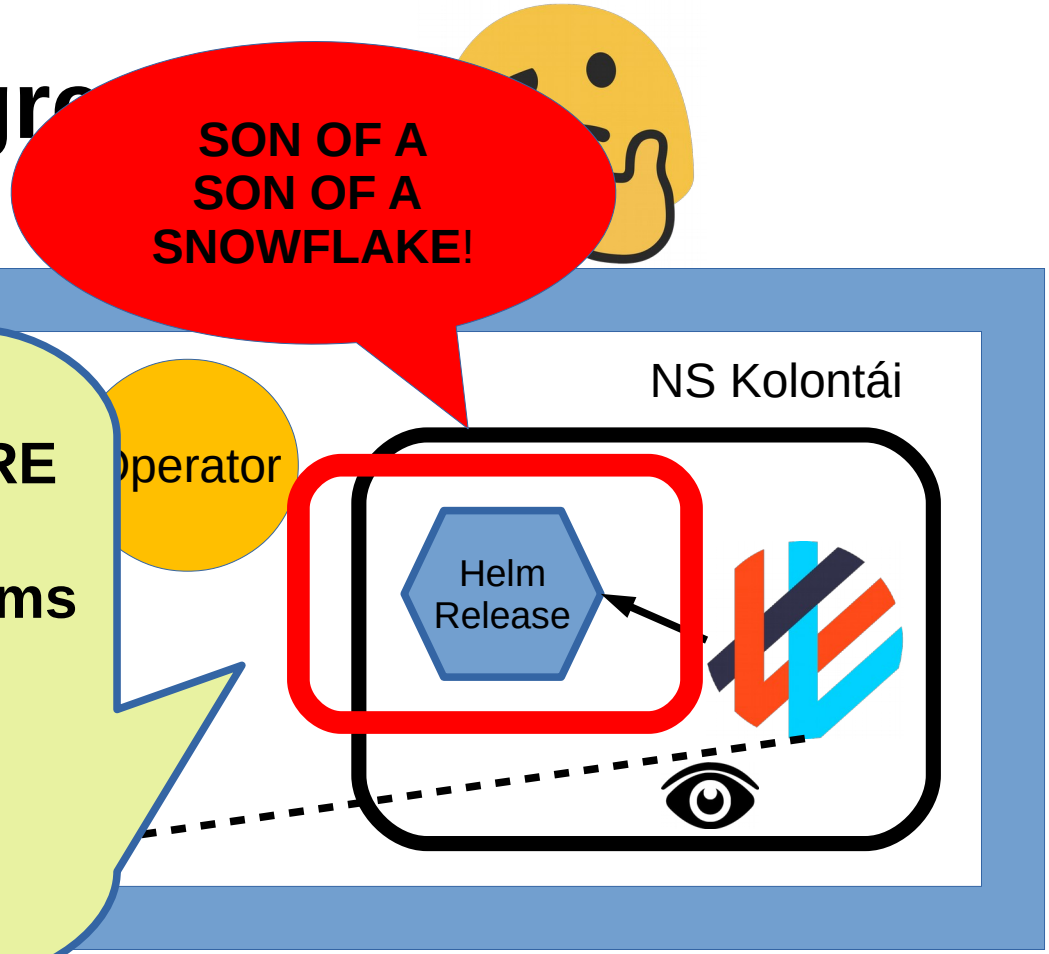
Wouldn’t it be great if...



Wouldn’t it be great if



Wouldn’t it be gr

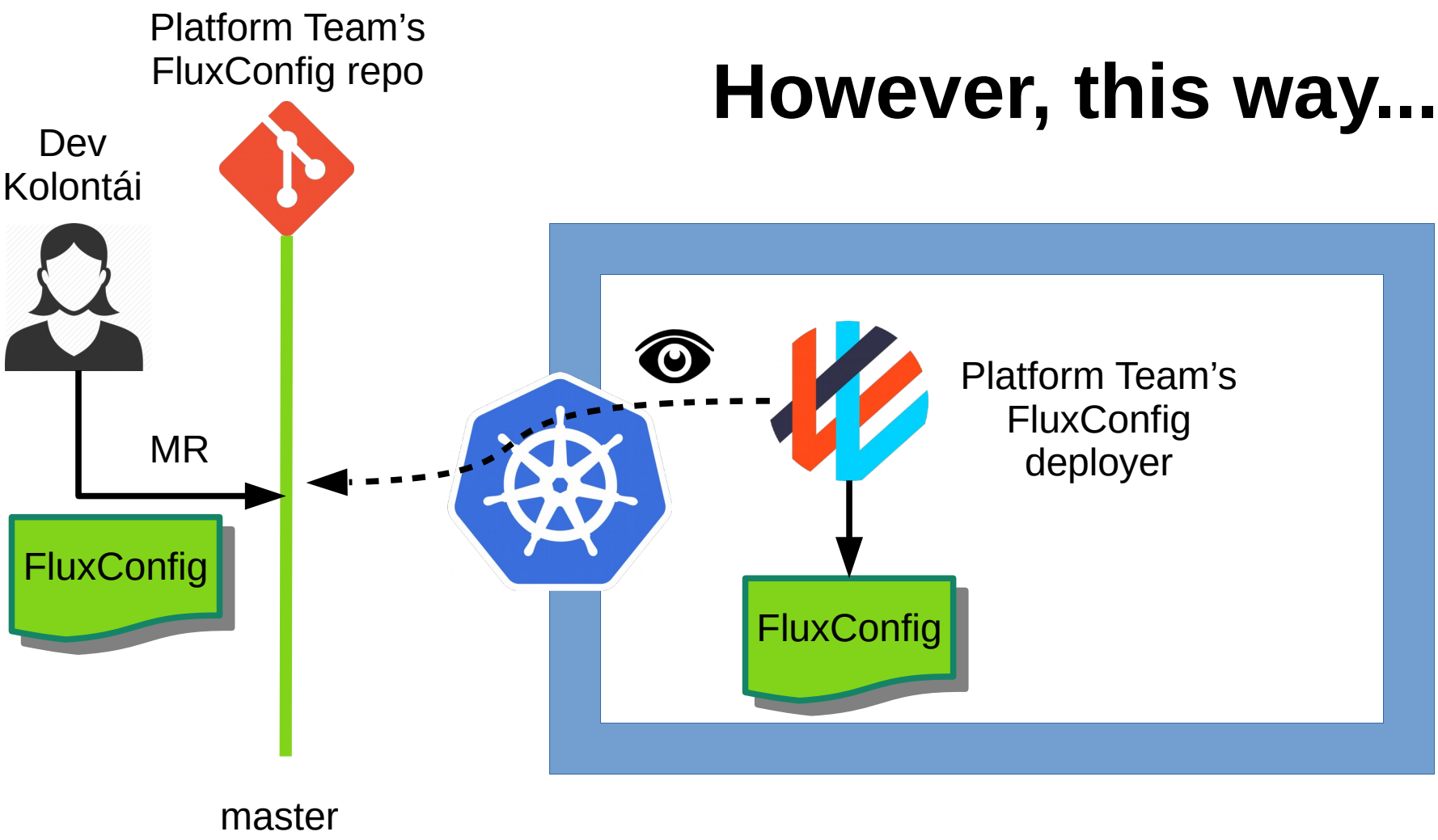


DevOps = IaC + CD + SRE

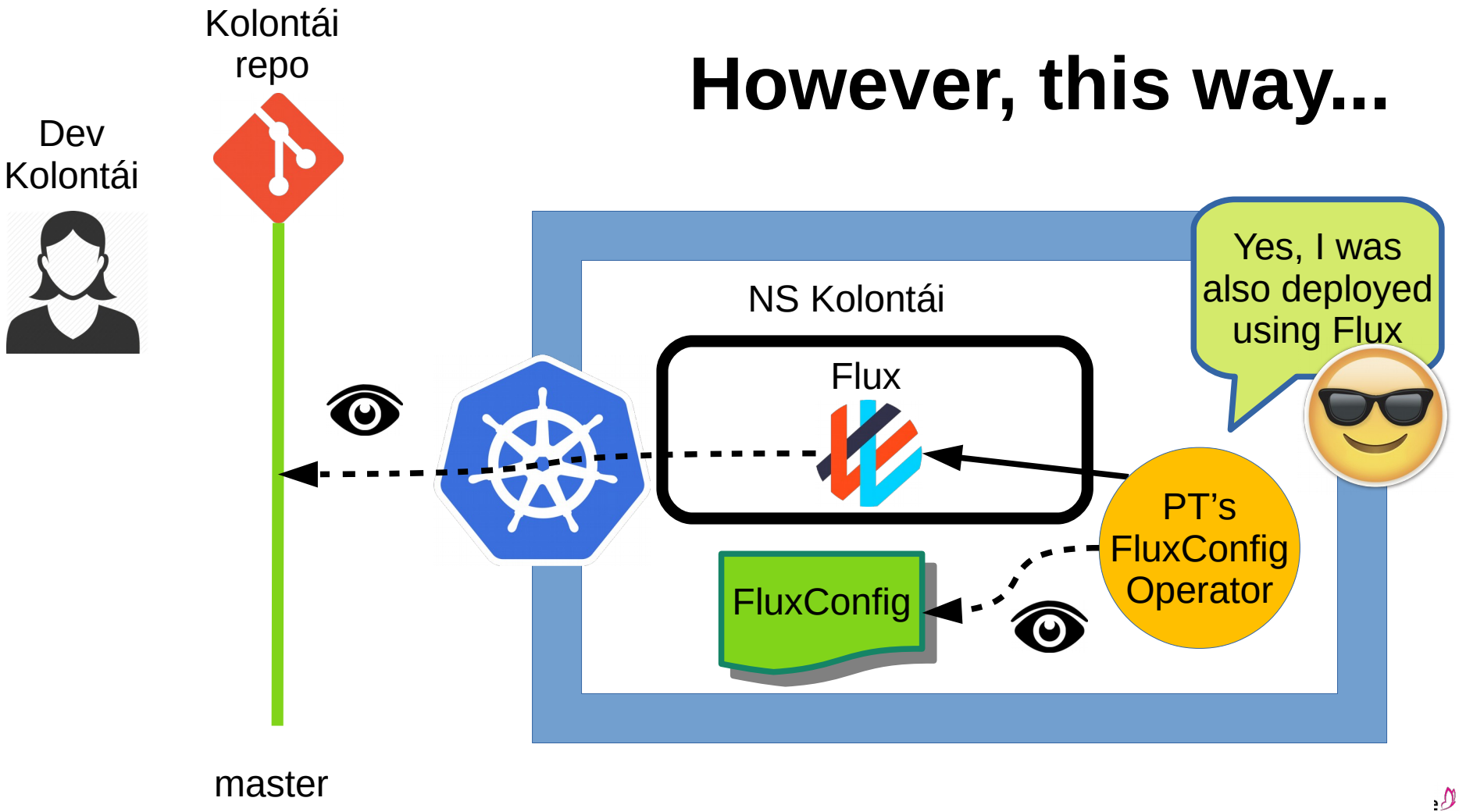
IaC = Reproducible systems

Manually created stuff
==
Irreproducible stuff

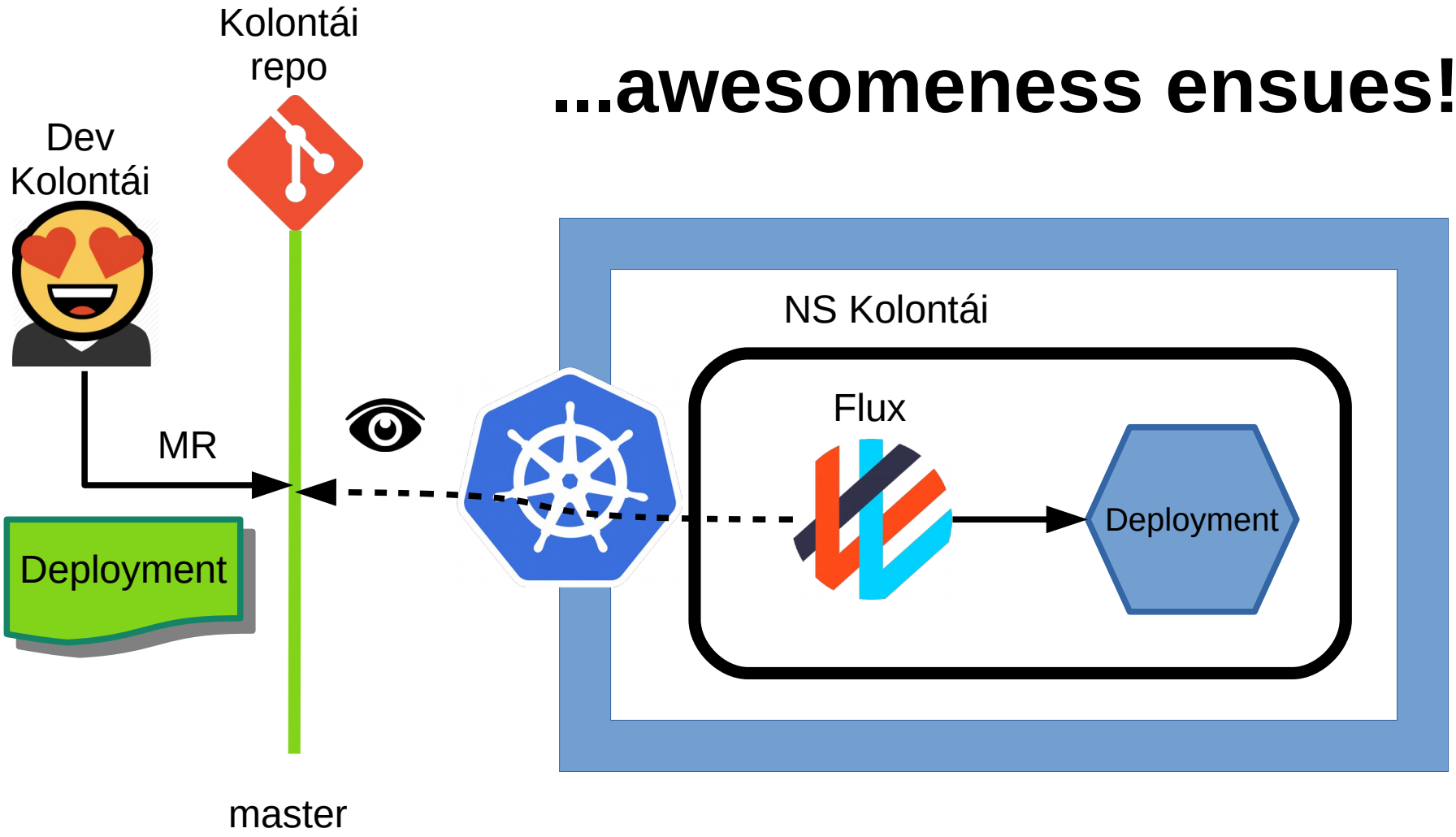
However, this way...



However, this way...



...awesomeness ensues!



Enter the Flux Operator!

https://github.com/justinbarrick/flux-operator

- **Includes a Flux CRD**
  - Teams wishing for a Flux instance deploy a resource of this CRD type
- **Two modes of operation:**
  - **Cluster mode:** Flux resources indicate in which namespace the Flux instance must be deployed

    → **When Flux definitions come from admins → Any NS**
  - **Namespace mode:** Flux instance created in the namespace where we deploy the Flux CRD

    → **When Flux definitions come from dev teams → Single NS**

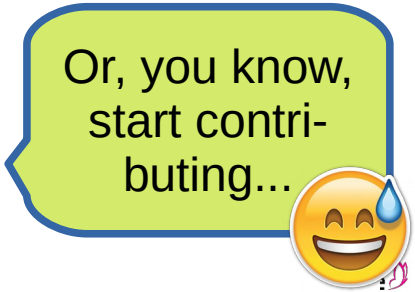
DEMO #6

Flux Operator FTW!

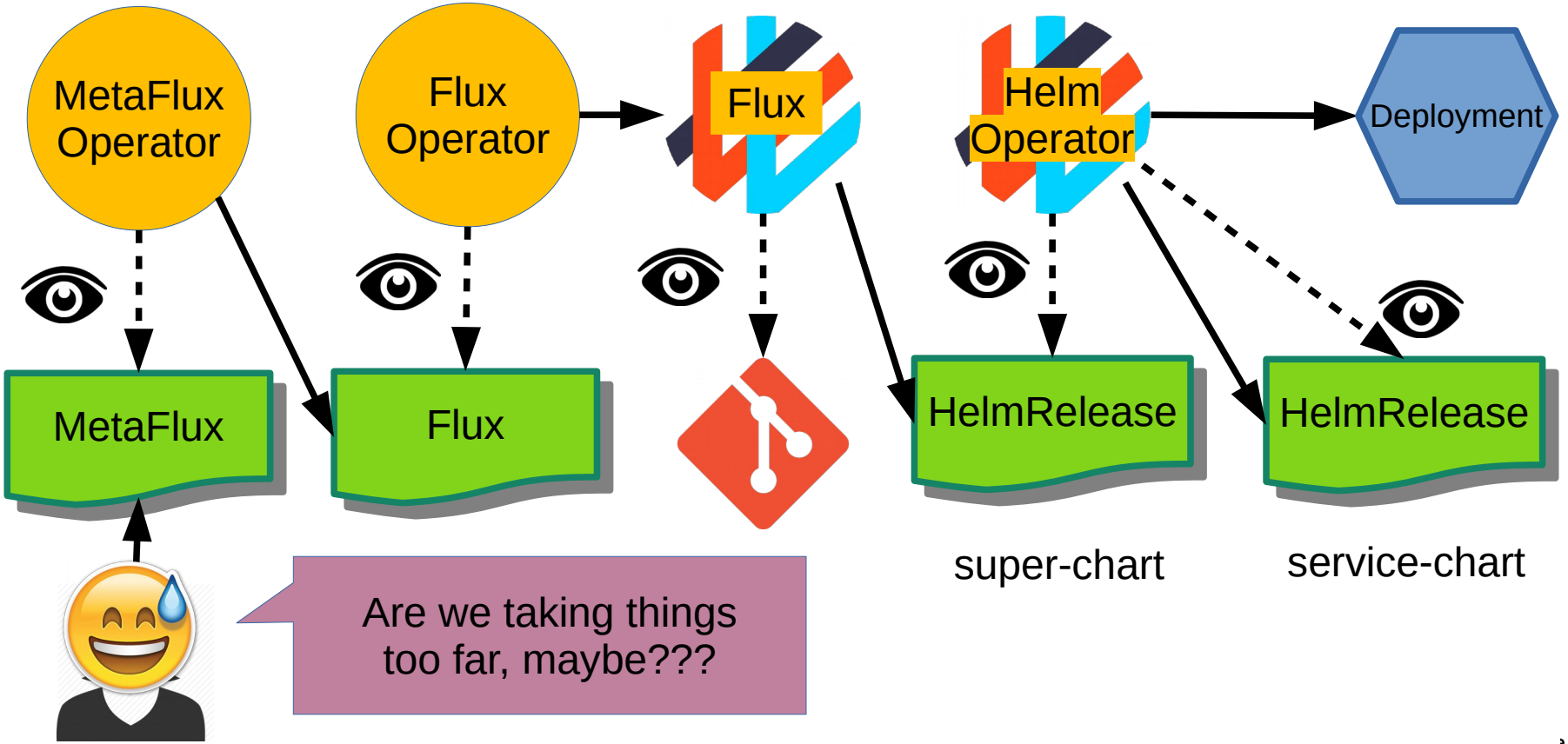
Flux Operator

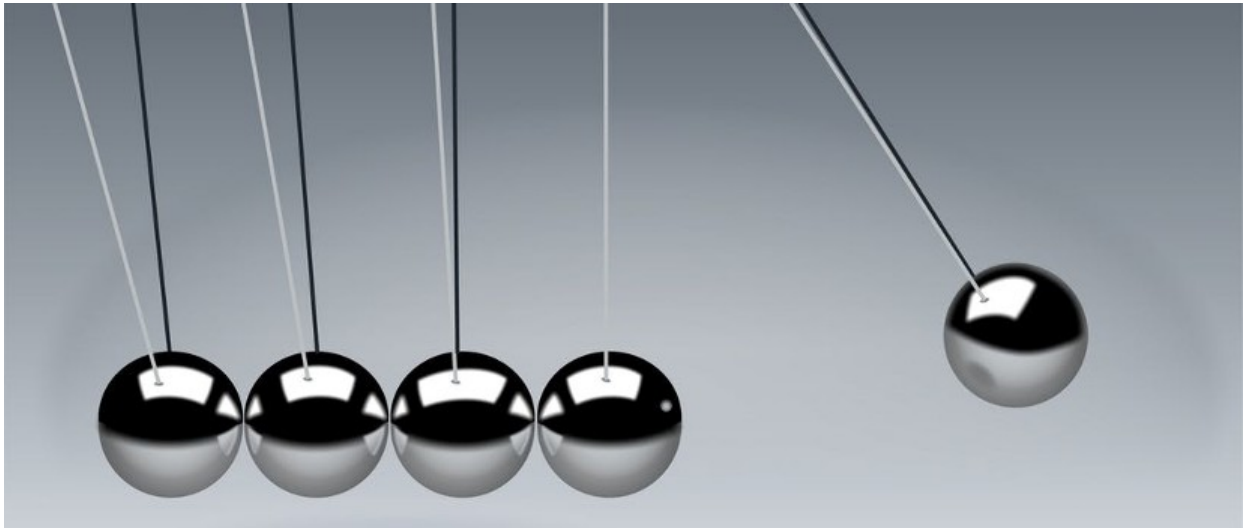
https://github.com/justinbarrick/flux-operator

- **Drawbacks:**
  - **Very young project! (v0.0.5)**
  - **Doesn't really implement the ideal "separation of concerns" mentioned before :-(**
    - The definer of the Flux CRD resource provides **the full configuration** for the Flux instance
      - Can't define defaults for the Flux instance config in the Operator
        - e.g. Can't define default credentials for some repos centrally...
      - We can't validate the requested Flux instance config
        - e.g. Can't do white/black listing of Git repos...
    - **Therefore, we need to keep on searching…**

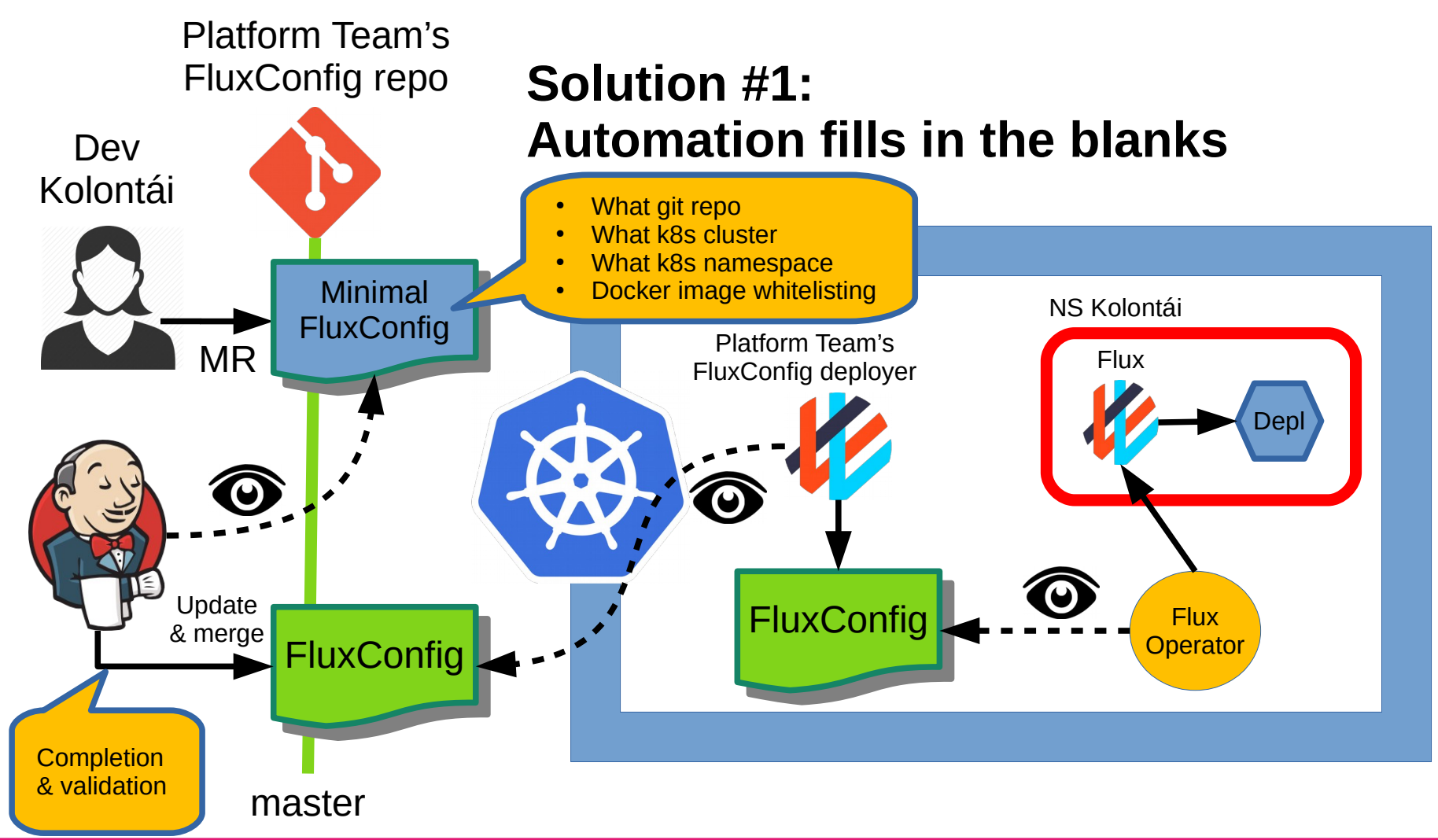


Enter our own Meta-Flux-Operator???

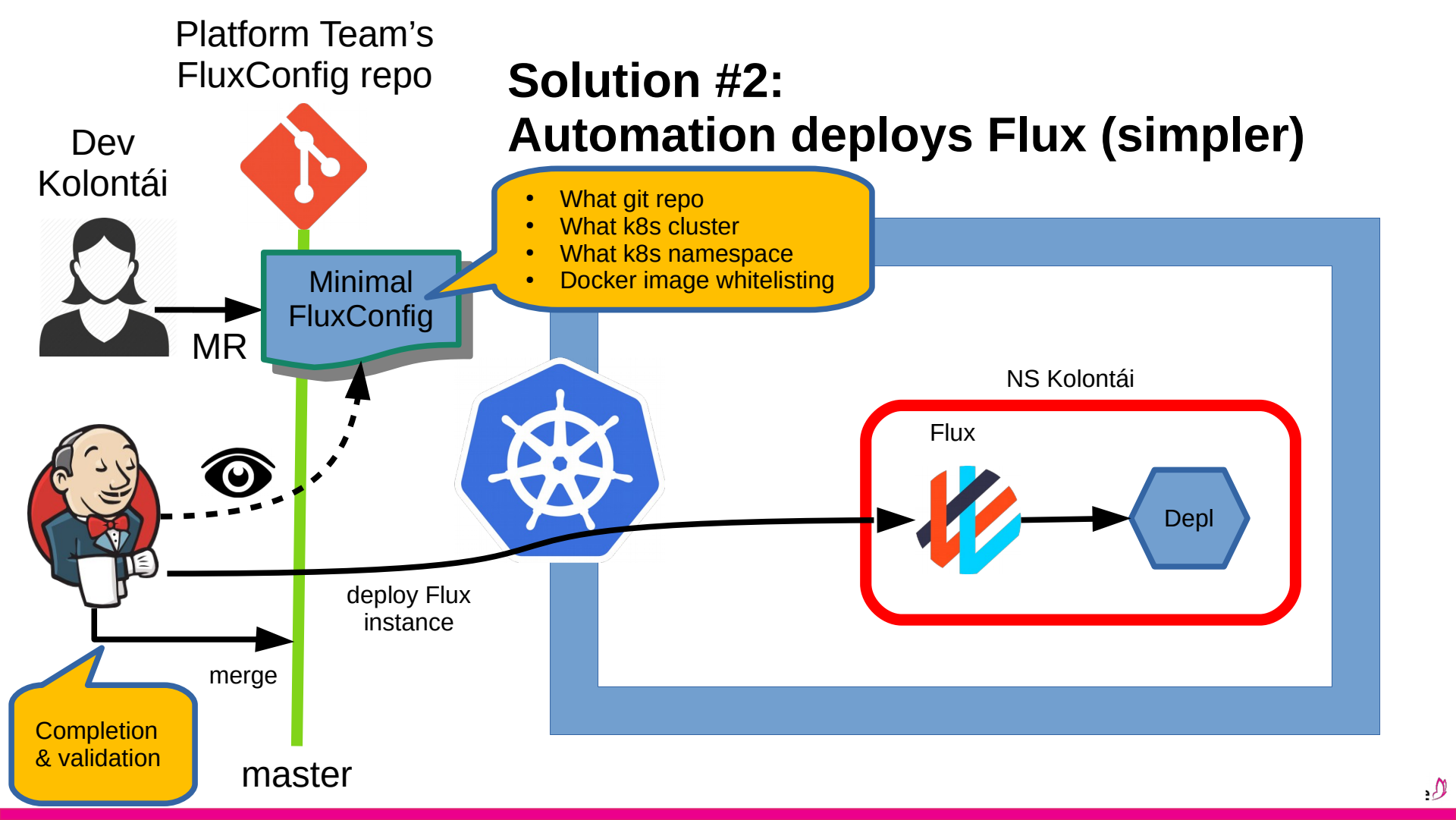




Solution #1: Automation fills in the blanks



Solution #2:
Automation deploys Flux (simpler)

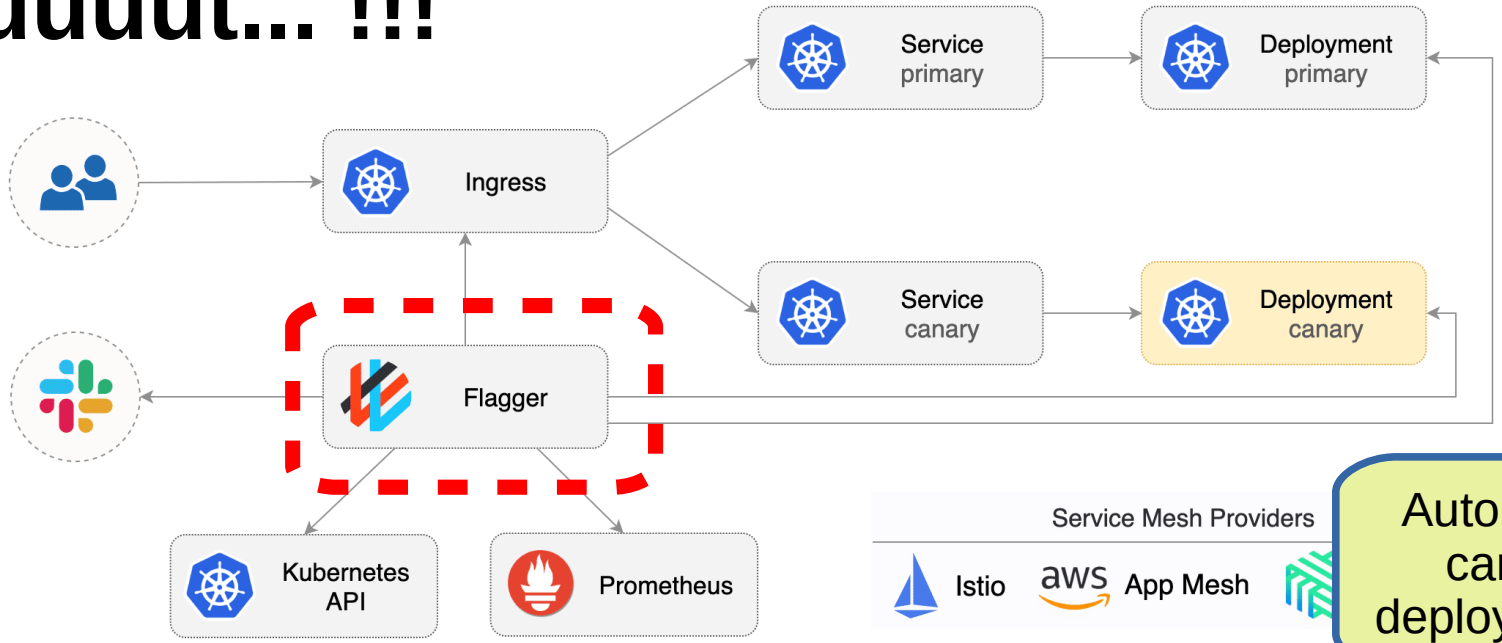


But…!!!

Can we do better?

… not today.

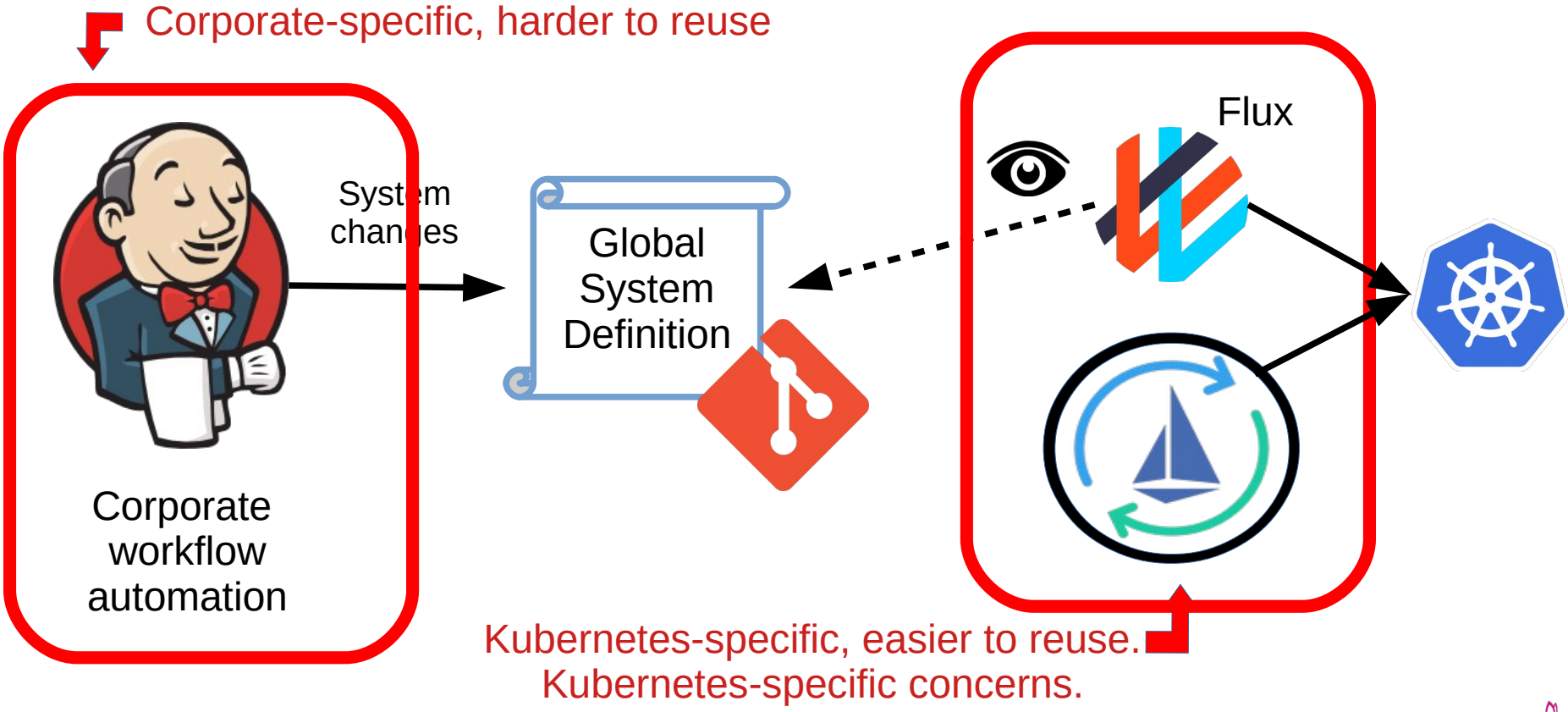
Buuuut... !!!



Automated canary deployments!

Let’s talk some other day about Flagger…

The Flagger proposal




THANK YOU