Red Hat Reference Architecture Series

Application Release Strategies With OpenShift

Andrew Block

Version 1.0, 2017-01-22

Table of Contents

100 East Davie Street
Raleigh NC 27601 USA
Phone: +1 919 754 3700
Phone: 888 733 4281
PO Box 13588
Research Triangle Park NC 27709 USA

The GPG fingerprint of the security@redhat.com key is: CA 20 86 86 2B D6 9D FC 65 F6 EC C4 21 91 80 CD DB 42 A6 0E

Send feedback to refarch-feedback@redhat.com

Comments and Feedback

In the spirit of open source, we invite anyone to provide feedback and comments on any reference architecture. Although we review our papers internally, sometimes issues or typographical errors are encountered. Feedback allows us to not only improve the quality of the papers we produce, but allows the reader to provide their thoughts on potential improvements and topic expansion to the papers. Feedback on the papers can be provided by emailing refarch-feedback@redhat.com. Please refer to the title within the email.

1. Executive Summary

Red Hat OpenShift Container Platform 3 is built around a core of application containers powered by Docker, with orchestration and management provided by Kubernetes, on a foundation of Atomic Host and Red Hat Enterprise Linux. OpenShift Origin is the upstream community project that brings it all together along with extensions, to accelerate application development and deployment.

This reference provides an overview of the features and approaches that can be taken to deploy containerized applications to the OpenShift Container Platform. Through the use of the methodologies described, applications can be released in a manner that reduces or eliminates downtime and enables additional testing strategies to validate success.

2. Introduction

Releasing software can be a tumultuous process. System libraries, application artifacts, configurations, people and processes must be carefully choreographed in order for a release to be deemed successful. In some instances, even a simple production software deployment can be a multi-hour ordeal that consumes the effort of countless individuals and teams. As a result, software releases became an infrequent occurrence. Over time, new and improved methodologies, approaches and technologies have been introduced to reduce complexities to help streamline and automate the deployment process with the primary goal of reducing time and most importantly risk.

Two of the most influential concepts that have helped revolutionize how software is built and deployed are the principles of Continuous Integration and Continuous Delivery (CI/CD). Continuous Integration introduced a process where application source code was stored in a common repository and regularly tested and validated each time a change occurred. Once established, the principles of Continuous Integration (CI) were further extended by Continuous Delivery (CD), which describes a consistent and repeatable process for deploying software up to and including a production environment. CI/CD and its inherent principles eventually led to the coining of the term "DevOps" as a way of depicting how development and operations have converged from their previously siloed operations to participate in the software development and delivery lifecycle together. As a result of Continuous Integration and Continuous Delivery, the time that it takes to deliver software has been greatly reduced.

One of the greatest advances that further aided the principles introduced by Continuous Integration and Continuous Delivery was the popularization of containerization technology, such as Docker. Now, both the underlying system components, such as the operating system and base libraries, along with the application and required runtime could be packaged into a single atomic unit that is portable and could be deployed to any target environment supported by Docker: bare metal, virtual or cloud. The need to manage containers at scale led to the development of Kubernetes, an orchestration framework to manage the lifecycle of containers. OpenShift, Red Hat's container platform, combines the power and flexibility of both Docker and Kubernetes to provide features for supporting rapid development and deployment to meet today's ever increasing demands. By providing support for implementing the most popular CI/CD methodologies and patterns, users have the flexibility and confidence that they can evolve rapidly, but also safely.

OpenShift supports the deployment of a wide range of applications, frameworks and systems. One of the most common types of deployments to OpenShift are web facing applications as they can take advantage of the autoscaling and lifecycle management features provided by the platform. To illustrate the release strategies and approaches provided by OpenShift, a fictitious application for managing weekly grocery lists called Shopernetes will be introduced in order to provide common use cases for managing application releases. Shopernetes is a Python based application that hosts the interface that allows users to manage their weekly grocery lists. Persistence is provided by an existing persistent data store located outside of the OpenShift cluster.

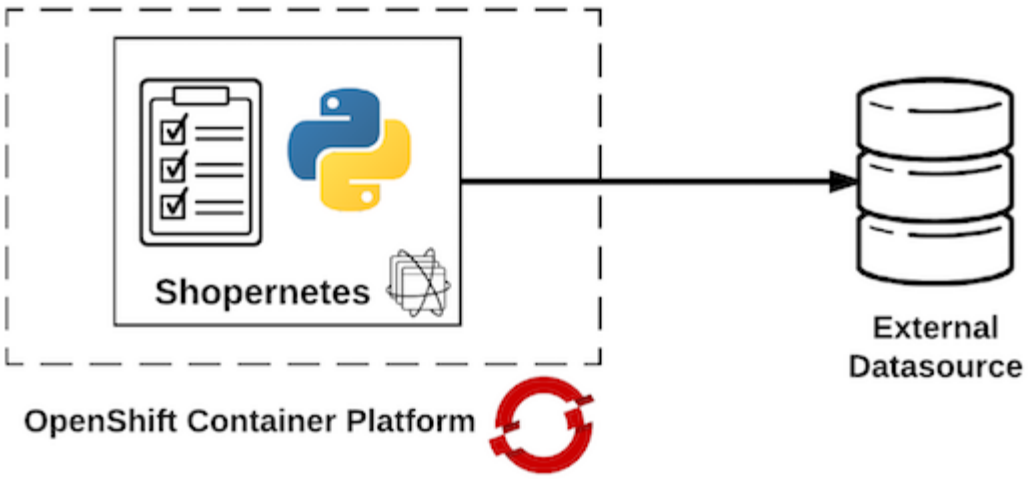

Figure 1. Shopernetes application architecture

In this fictitious scenario, management has emphasized the importance of releasing features on a regular basis in order to provide an enhanced user experience, but most importantly to keep up with market demand and competition.

3. Prerequisites

This document assumes the reader has an understanding of the core concepts of OpenShift including the objects found within the platform.

A properly configured environment must also be available to execute the content described. Official documentation and prior reference architectures can be consulted to describe and implement the available options.

4. Deploy With Confidence

One of the most common reasons why releasing software is perceived as complex can be partially associated with the number of steps that may be involved in the release process. This is further exacerbated by the fact that many of these steps may also involve manual processes, such as managing binaries, configuration values, and overall application availability, which only adds additional complexities. By the time the application is fully deployed, there is also no guarantee that it is in the expected state. OpenShift and its supporting ecosystem provide several solutions for reducing the uncertainties associated with application deployment. First, by using a containerized architecture, the application and its underlying components are packaged and deployed together ensuring each has the correct versions, updates and patches. Second, OpenShift provides a declarative approach for describing the way an application is configured and deployed along with its ancillary components. The result is the assurance that if an application is configured in a particular way, the OpenShift cluster along with its scheduling policies will do everything in its power to conform to the expected configuration. This configuration can be expressed in either YAML or JSON, two popular serialization formats, and can also be stored and versioned in a source code management (SCM) system, such as Git, to provide even greater transparency and stability for cluster management.

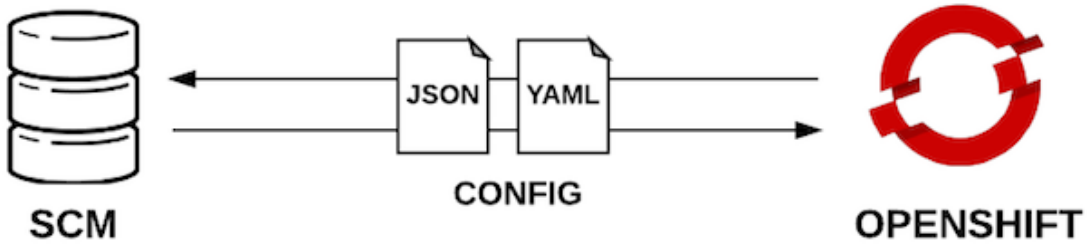

Figure 2. Configuration managed in source control and applied to the OpenShift Container Platform

The values represented within these configurations are comprised of OpenShift API objects including but not limited to Pods, Deployment Configurations (DeploymentConfig) and Services.

4.1. Deployment Strategies

Deploying an application can involve a number of processes and tasks. OpenShift provides several strategies for managing the deployment of an application. The type of strategy chosen is largely dependent on the application's composition and availability requirements. Three types of deployment strategies are supported and are defined in the strategy section of the DeploymentConfig: recreate, rolling and custom.

Listing 1. An example of a DeploymentConfig using the Recreate strategy

```
$ oc export deploymentconfig shopernetes
apiVersion: v1
kind: DeploymentConfig
metadata:
  name: shopernetes
spec:
  strategy:
    type: Recreate
...omitted...
```

The recreate strategy provides the most basic form of deployment within OpenShift. When an application is first deployed, pods are concurrently deployed from 0 up to the number of desired pods, or replicas. If a version of the application already existed and a new deployment was initiated, the existing pods would be scaled down completely to 0 first, followed by a roll out pattern similar to the initial deployment.

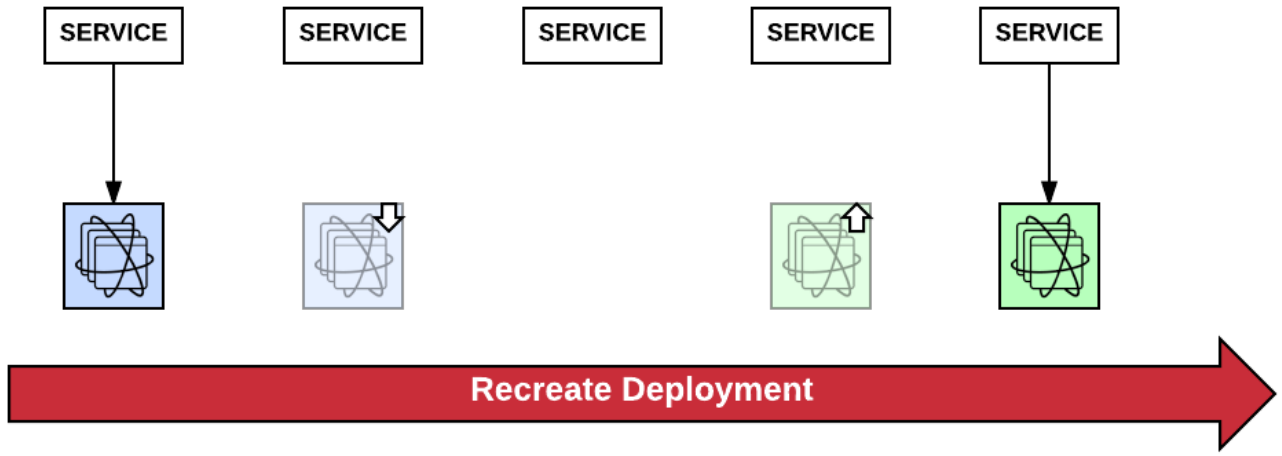

Figure 3. Deployment using the recreate strategy replaces an existing application by scaling down resources fully prior to scaling up new application

While the recreate strategy provides a limited set of options, there are several use cases for this deployment strategy. Since pods containing the previous version are quickly replaced by the new version, it is ideal during the application development phase. Frequent changes occur as new features

and functionality are added. The ability to quickly test and validate the latest changes reduces the overall cycle time for a development team. For a production use case, applications that can handle maintenance windows or outages, such as an application that is utilized only during working hours, can take advantage of the recreate deployment strategy. During an overnight period, a new deployment can be initiated which stops the existing application until the new version is deployed and functionality is restored.

However, for mission critical applications that have a high service level agreement (SLA) and cannot afford any availability outage, an alternate deployment strategy should be utilized. OpenShift provides the rolling strategy to satisfy this use case and guarantee application uptime by providing a finer level of detail of the deployment process. Instead of scaling down existing instances of the application completely prior to deploying the new version, instances of the new version of the application are instantiated over time. Only once a replica of the new version becomes active is an instance from the prior version spun down.

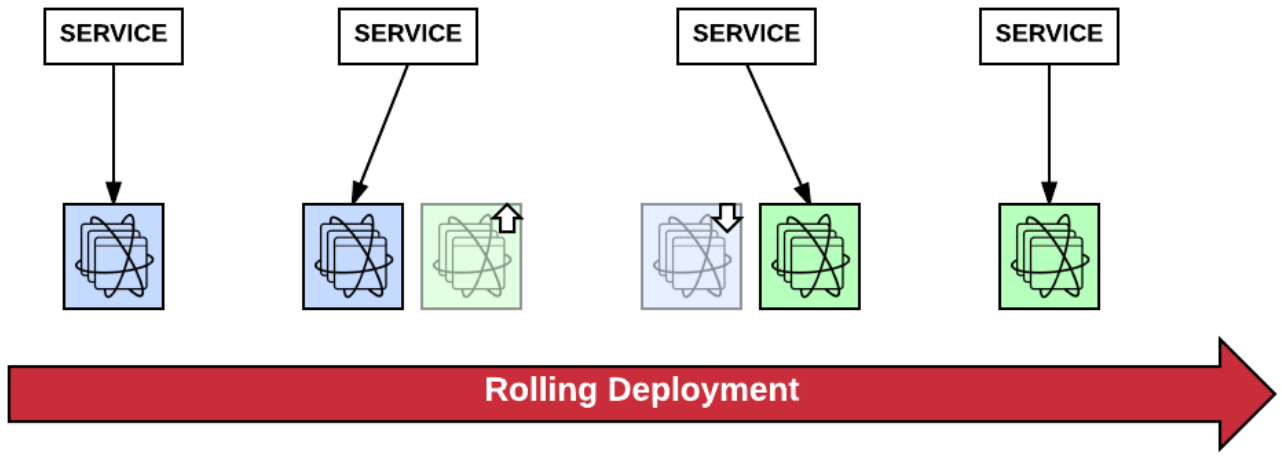

Figure 4. Rolling deployments scale in new application instances based on a configuration and waits for them to be running before scaling out old application instances

Rolling deployments take advantage of OpenShift services and their corresponding endpoints to manage the availability of applications as new versions are being released. Only once a pod has reported itself as ready will it be added as an endpoint for a service and made eligible to take on incoming traffic. At that time, based on the definition of a rolling deployment, older versions of the application are subsequently unregistered as an endpoint, thus they will no longer receive traffic and can be safely terminated. The definition of a rolling deployment includes the ability to specify how many instances of an application should be scaled up and be ready prior to scaling down older versions, how many instances of the application can remain unavailable during a scaling event, and the maximum amount of time for each scaling event. In the listing below, the rolling deployment will scale up new pods up to 20% above the original number of deployed pods, ensure only 10% of the original replica set become unavailable during the deployment, and each scaling event can amount to 120 seconds total.

Listing 2. Defining a `Rolling Deployment` within a `DeploymentConfig`

```
$ oc export deploymentconfig shopernetes
apiVersion: v1
kind: DeploymentConfig
metadata:
  name: shopernetes
spec:
  strategy:
    type: Rolling
    rollingParams:
      timeoutSeconds: 120
      maxUnavailable: “10%”
      maxSurge: “20%”
...omitted...
```

The SLA for the application typically dictates the range these values could be configured. For example, when performing an upgrade of an existing application and there is a requirement to retain full availability of the existing version of the application as the new version is being rolled out, the *maxUnavailable* parameter must be set at 0. During the development phase, most teams determine an optimal value that both meets the defined SLA and overall application performance.

4.2. Validating Application Health

As new applications are being started, one of the key factors that is important to the rolling deployment strategy is the ability to determine when the application is in a ready state. Only when an application has reported as ready can it act on the next step of the deployment process. OpenShift makes use of various probes to determine the health of an application during its lifespan. A `readiness probe` is one of these mechanisms for validating application health and determines when an application has reached a point where it can begin to accept incoming traffic. At that point, the IP address for the `pod` is added to the list of endpoints backing the service and it can begin to receive requests. Otherwise traffic destined for the application could reach the application before it was fully operational resulting in error from the client perspective. Application health checks are configured on the `DeploymentConfig` object and implemented using one of three primary mechanisms: Executing a HTTP request, performing TCP socket validation, or by executing a script within the running container. Health checks are executed from the host the container is running on depending on the method specified. HTTP or TCP socket methods are used to validate the application can be accessed on a particular port or from a URL.

Listing 3. An example of a TCP application health check

```
$ oc export deploymentconfig shopernetes
apiVersion: v1
kind: DeploymentConfig
metadata:
  name: shopernetes
spec:
  template:
    spec:
      containers:
        - name: shopernetes
          readinessProbe:
            tcpSocket:
              port: 8080
            initialDelaySeconds: 15
            timeoutSeconds: 1
...omitted...
```

More in depth validation can be created to not only confirm the application can be accessed, but also validate dependencies, such as databases or caches, are available. In the example Shopernetes application, an external database is one of the dependencies that should be accessible prior to an application being made available for end users. Otherwise, users would receive errors when performing the most common functions of the application, such as adding and removing items from their grocery list since the backend persistence store is unavailable. Alternate methods can also be implemented to manage application availability and usability when the database is not available. Extended validation of the application and its dependencies can be implemented using several different strategies. First, is to expose health checking logic within the application, such as on an HTTP

endpoint on the */health* context which is common feature found in a number of web frameworks. Alternatively, the logic for determining application health can be delivered as a script and executed as a command against the container (See the listing below). A non zero exit code from the command indicates the application is no longer healthy.

Listing 4. An example of a script based application health check

```
$ oc export deploymentconfig shopernetes
apiVersion: v1
kind: DeploymentConfig
metadata:
  name: shopernetes
spec:
  template:
    spec:
      containers:
        - name: shopernetes
          readinessProbe:
            exec:
              command:
              - /bin/bash
              - c
              - /opt/app/health.sh
            initialDelaySeconds: 15
            timeoutSeconds: 1
...omitted...
```

Once an application is running, there are no guarantees that it will continue to operate with full functionality. Numerous factors including out of memory errors or a hanging process can cause the application to enter an invalid state. While a readiness probe is only responsible for determining whether an application is in a state where it should begin to receive incoming traffic, a liveness probe is used to determine whether an application is still in an acceptable state. If the liveness probe fails, the kubelet will destroy the pod. Additional criteria can be specified to allow the probe to fail a certain number of times before destroying the pod in the event there is a brief interruption that is quickly resolved. After a pod is destroyed, the application may be rescheduled once again. The listing below uses an HTTP based probe to validate an endpoint within the deployed application. If the probe fails 3 consecutive times, the pod is destroyed as defined by the `failureThreshold` property.

Listing 5. A `Liveness Probe` defined within a `DeploymentConfig`

```
$ oc export deploymentconfig shopernetes
apiVersion: v1
kind: DeploymentConfig
metadata:
  name: shopernetes
spec:
  template:
    spec:
      containers:
        - name: shopernetes
          livenessProbe:
            httpGet:
              path: /healthz
              port: 8080
            initialDelaySeconds: 15
            timeoutSeconds: 1
            failureThreshold: 3
...omitted...
```

Fortunately, both the readiness and liveness probes make use of the same API definition which provides potential reuse of configuration or scripting logic.

5. Deployment Methodologies

Rolling deployments as described in detail previously are more than a deployment strategy within OpenShift, but one of many different methodologies that have been created for deploying systems. Each methodology is created to account for a particular goal or goals that should be satisfied prior to a deployment being deemed successful, such as ensuring zero application downtime. A common theme which cuts across all aspects of the software development process is the ability to perform testing. The ability to validate a given scenario provides the necessary assurance that a task was completed successfully. The execution of testing and validation concepts at deployment time can be facilitated either in manual or automated fashion and can also involve different actors and participants including developers, project managers, a quality assurance team, or even end users. The involvement of each party is partially dependent on the methodology being utilized.

5.1. Out of the Box Functionality

One such methodology for deploying applications is called a canary release. During a canary release, validation against the state of the target system occurs during the deployment process. Only when the state can be successfully validated can the deployment process proceed. The theory behind canary releases may sound familiar as it parallels the functionality of application level health checking and rolling deployments of OpenShift as described in earlier sections. By simply configuring `DeploymentConfig` with a readiness probe, then an application can participate in the concepts emphasized by a canary release. The term canary release originates from the former practice of taking caged canaries into coal mines. Canaries were used as an early warning system by miners to detect dangerous carbon monoxide or methane gas. If the canary should perish, it was an indication toxic fumes were present and allow the miner to take action or exit the dangerous area safely. In a rolling deployment within OpenShift, if a readiness probe failed as a new version of the application was being deployed, the deployment itself would stop and fail, like the canary, without affecting the existing operational application.

5.2. Traffic Shaping Patterns

So far, each of the strategies and patterns that have been previously discussed have focused on an in place deployment of a single application environment. Alternative deployment methodologies have been developed which instead utilizes separate environments representing a different release of the software. The rationale behind this architectural approach is that it provides the flexibility necessary to test, validate and potentially roll back the application in isolation without disturbing the original operational application. The usage of elastic computing resources became one of the primary drivers to the adoption of advanced deployment methodologies and cloud based technologies. Previously, creating a secondary environment with each application release was a challenge due to the amount of time and resources necessary to provision an environment. In some cases, the time it took from the moment resources were requested to the time they were made available could be as high as 30 days. Add in any additional amount of time to finalize the environment in preparation of the release and it became clear that executing releases on a regular basis would become a logistical challenge. Once running in an OpenShift environment, allocating a second set of resources for the new version of the application could be accomplished with as little as a simple `oc new app` command. Both versions could be deployed and run within the same OpenShift project as it would provide a comparable environment and access to any shared resources, such as user privileges and services.

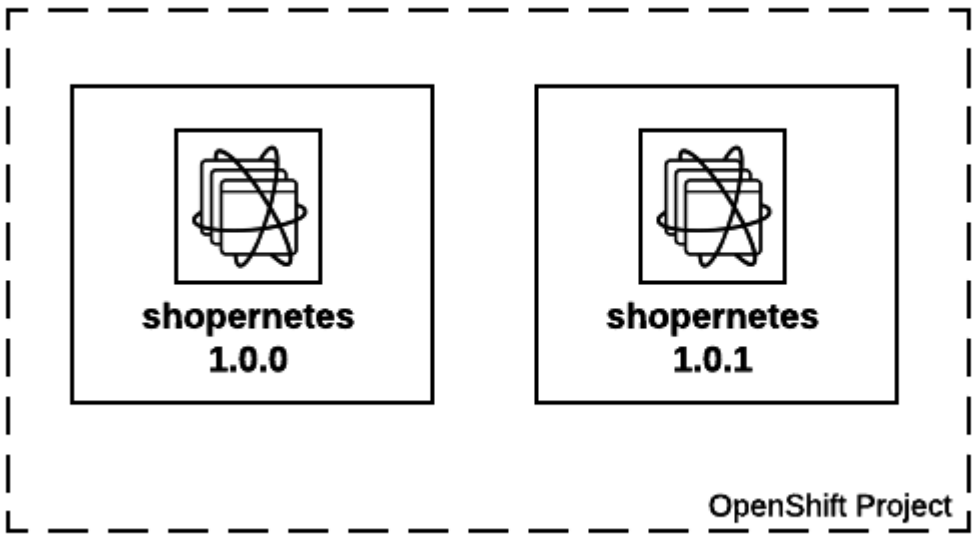

Figure 5. Multiple versions of an application within a project

Controlling and managing traffic flow gives operators the flexibility to determine which version of the application will ultimately receive the incoming request. When attempting to access a typical web application deployed to OpenShift, traffic originating outside the cluster must traverse through multiple network layers prior to reaching the underlying application running in a pod. The ingress point for the cluster is managed by the routing layer consisting of one of two plugins: a customized HAProxy router or an integration with an externally configured F5 router. The router is responsible for providing access to OpenShift's `Software Defined Network (SDN)` and specifically the service

network. Access to specific services are defined using routes, another OpenShift API object, and provide an association between an HTTP hostname of a request and the targeted OpenShift service. Since routes and services are the backbone for exposing applications, the ability to customize their definition and interaction with each other ultimately determines which application a request will receive, and allows users to implement the most popular deployment and traffic shaping methodologies in the industry.

5.2.1. Blue-Green Deployments

Developers of the Shopernetes application have been hard at work and have prepared a new version of the application containing numerous improvements to both stability and user interaction. One of the challenges faced when releasing new software is validating not only the stability of the application, but the interaction with internal and external integration points, among others. Rolling deployments provide mechanisms to automate the roll out of an application and zero downtime by ensuring new versions conform to a set of preconfigured conditions. However, in many cases, teams may wish to perform additional steps to validate the new version of the application. Only once those steps are completed will the release process move forward and traffic will reach the new version of the application. The concept of having a separate environment to fully test and validate an application prior to shifting traffic completely from the old environment to the new environment is called a Blue-Green deployment. The term Blue-Green refers to the status of each application during the deployment process. As traffic is shifted in a hard cutover from the old version of the application to the new version, the new version takes on all requests and is now live, or green. With all traffic routed to the new version of the application, the older version of the application is put into an idle state, or blue. If necessary, traffic could then be shifted back to the original version of the application with minimal effort or downtime for the end user.

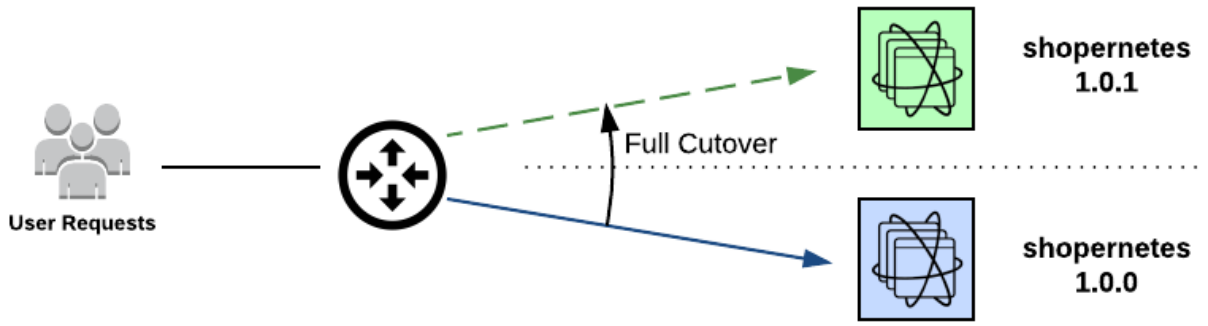

Figure 6. Blue-Green deployment

In OpenShift, a Blue-Green deployment involves creating the two separate versions of the application (1.0.0 and 1.0.1), each having their own respective DeploymentConfigs and Services. A route sits in front of both services and plays a key role in determining which service will retrieve the incoming traffic.

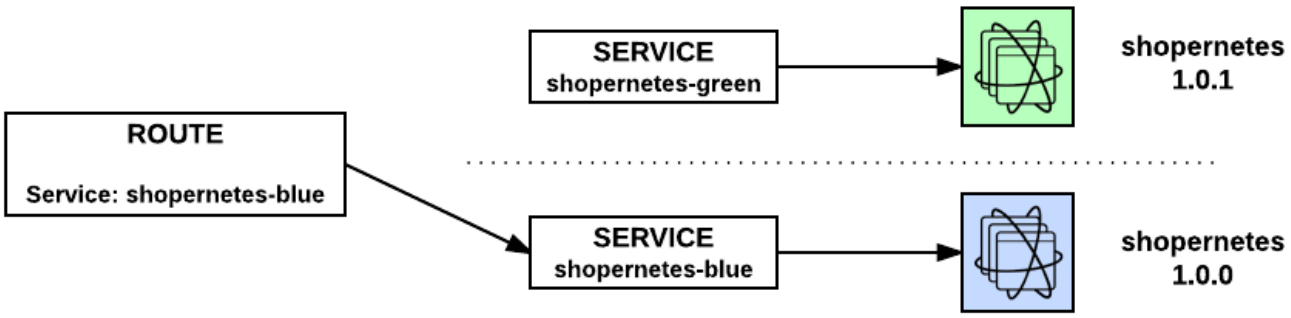

Figure 7. Routing in a Blue-Green deployment

Initially, this route is configured to target the blue service containing the original version of the application as shown below:

Listing 6. The Shopernetes route

```
$ oc export route shopernetes
apiVersion: v1
kind: Route
metadata:
  name: shopernetes
spec:
  host: shopernetes.ocp.example.com
  port:
    targetPort: 8080
  to:
    kind: Service
    name: shopernetes-blue
```

This same route can also be created by exposing a service called *shopernetes-blue* with the following command using the OpenShift Command Line Interface (CLI):

Listing 7. Creating a route by exposing a service

```
$ oc expose svc shopernetes-blue --name=shopernetes
--hostname=shopernetes.ocp.example.com --port=8080
```

Even though both versions of the application are active, the new version of the application is not accessible by end users. It is a common practice for application teams to create additional ingress points to facilitate testing and validation of the deployment by internal teams. In many cases this will include the creation of an additional route targeting the service of the new version of the application which is only known or accessible by internal resources.

Once it has been determined that the new version of the application is ready to take on production

workload, the route can be updated to reference the service representing the new version of the application:

Listing 8. Updating the route the target the `shopernetes-green` service

```
$ oc patch route/app -p '{"spec":{"to":{"name":"shopernetes-green"}}}'
```

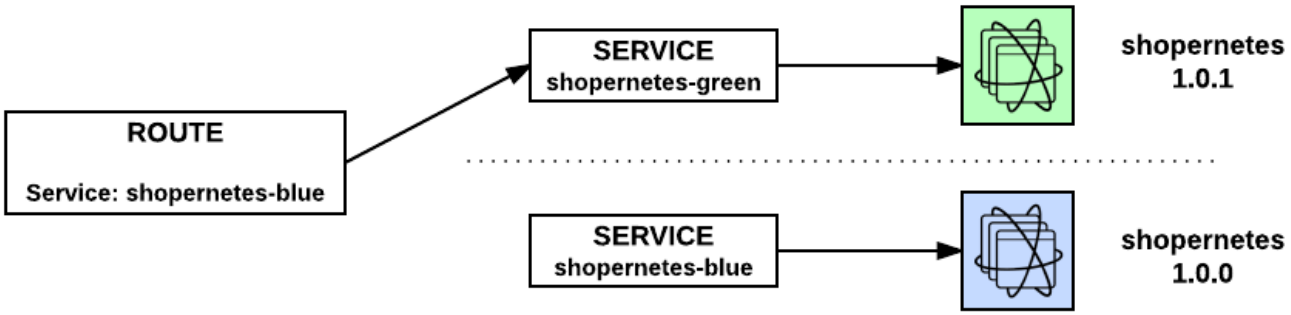

Figure 8. Blue-Green deployment after traffic has shifted to the new version of the application

With the latest version of application beginning to accept production traffic, it can be monitored to confirm any expected results. Once it has been determined that the release was successful, the older version of the application can be removed to free up additional resources in the environment.

While Blue-Green deployments provide a method for reducing risk by validating the state of an application prior to fully shifting production traffic, one must be cognizant of any data points that may be shared by the two application versions. In the Shopernetes application, a single data store is used to store records maintained by the application. Additional effort must be undertaken to verify both the schema of the datastore and the format of the records can be utilized by both versions of the applications. The ability to swap between the two versions of the application at a moments notice is one of the hallmarks of this deployment methodology. Taking additional measures at both development and during application validation will certify the datastore is compatible with both application versions and provide for a smooth and successful release process.

5.2.2. A/B Deployments

As the time between release cycles continues to decrease, there may be a desire to forgo traditional testing approaches facilitated by internal teams and instead solicit in feedback from the most important player in any application: the end user. Blue-Green deployments provided several mechanisms for reducing several of the risks that plague traditional application release methodologies. However, since the release involves a full cutover to the new version of the application, there was still a possibility of failure; either from a technical fault or an impact on the end users. To minimize the risk in this situation, another release strategy termed A/B deployment can be utilized. Also known as Online Controlled Experiments, instead of performing a full cutover from the old version to the new version of the application as demonstrated in a Blue-Green deployment, multiple versions of the application are run simultaneously. The new version of the application is introduced to a subset of the overall

users while the remaining users continue to access the old version to allow for testing and experimentation of the new version.

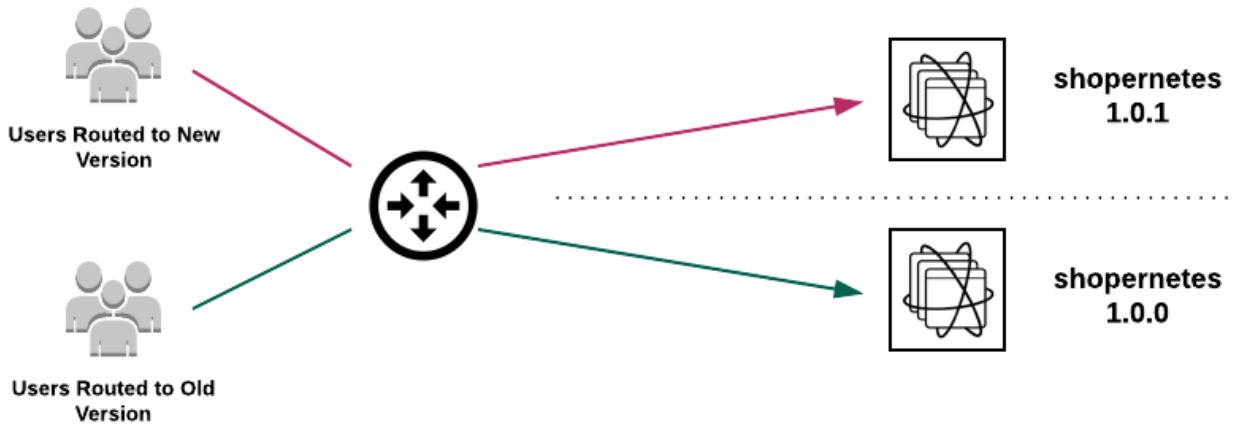

Figure 9. Routing within an A/B deployment

The ability to gather user feedback and execute additional testing of the new version of an application is one of the primary reasons why this deployment methodology is popular and in use by some of the largest companies in the industry. The determination of whether a new feature will or will not be a success is largely driven by user assessment using this method of testing. A/B deployments do involve a higher amount of risk as there is the potential to negatively affect end users. However, it is the metrics gathered from those users selected to receive the new version of the application which has a higher overall value than any adverse effects.

Shared Services

OpenShift provides several implementations for running multiple versions of an application in parallel to facilitate A/B testing. The first method is to use a single service to reference both versions of the application. Recall that services in OpenShift can be thought of as an internal load balancer over a set of related applications. The relation between the service and the underlying application is through the use of labels. A label is a key/value pair attached to various OpenShift objects, such as pods, to aid in the organization and grouping of related items. Services use label selectors to identify the pods they will proxy connections for. To allow multiple versions of an application to be referenced by the same service, they each must have a set of matching labels. To apply this release method to the Shopernetes application, the DeploymentConfig for each version of the application must be configured with a matching label, such as *“application=shopernetes”*. The common service called *shopernetes-shared* would use a selector to match this label to spread the requests across both versions of the application.

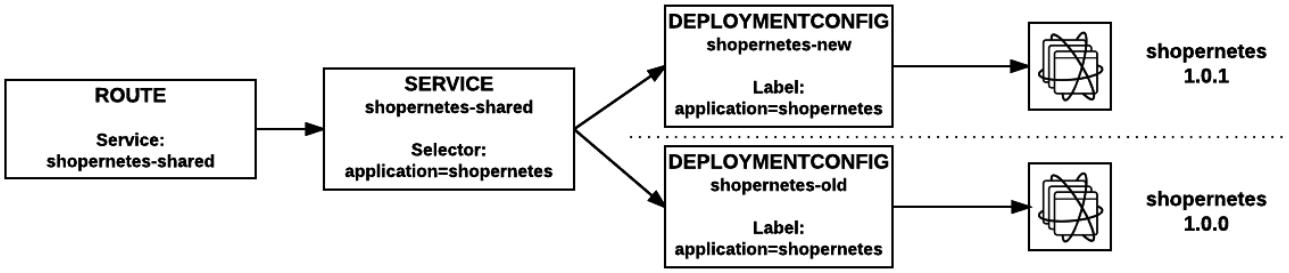

Figure 10. Using a service to balance traffic across multiple versions of the application

As shown in the diagram above, the DeploymentConfig called *shopernetes-old* represents the original version of the application and *shopernetes-new* represents the latest version.

The common label should be configured under both the *.spec.selector* and *.spec.template.metadata.labels* sections within the DeploymentConfig for the objects associated with each version of the application as shown below:

Listing 9. Labels and Selectors within a DeploymentConfig

```
$ oc export deploymentconfig shopernetes-old
apiVersion: v1
kind: DeploymentConfig
metadata:
  name: shopernetes-old
spec:
  selector:
    app: shopernetes-old
    deploymentconfig: shopernetes-old
    application: shopernetes
  template:
    metadata:
      labels:
        app: shopernetes-old
        deploymentconfig: shopernetes-old
        application: shopernetes
...omitted...
```

Next, a service should be created to target the label defined on each DeploymentConfig. The following contains the definition of a shared service called *shopernetes-shared*.

Listing 10. Example service used to balance requests between multiple versions of the Shopernetes application

```
$ oc export service shopernetes-shared
apiVersion: v1
kind: Service
metadata:
  name: shopernetes-shared
spec:
  ports:
  - port: 8080
    protocol: TCP
    targetPort: 8080
  selector:
    application: shopernetes
  sessionAffinity: None
  type: ClusterIP
```

Alternatively, the *shopernetes-shared* service can be created using the OpenShift Command Line Interface by exposing one of the DeploymentConfigs:

Listing 11. Creating a Service by exposing a DeploymentConfig

```
$ oc expose dc shopernetes-new --name=shopernetes-shared
--selector=application=shopernetes --port=8080
```

Finally, to allow this service to be accessed from outside the cluster, the *shopernetes-shared* service should be exposed as a route:

Listing 12. Exposing the service externally

```
oc expose svc shopernetes-shared
```

Creating a route by exposing a service automatically creates a hostname to access the application. To determine the generated hostname, execute the following command:

Listing 13. Using a Go template to obtain the host within a route

```
$ oc get route shopernetes-shared --template='{{ .spec.host }}'
```

A custom hostname could have been used when exposing the service by providing the *--hostname* parameter.

At this point, a web browser could be used to access the application from the hostname obtained previously which will return one of the versions of the application. The actual version returned

depends primarily on the configuration of OpenShift's routing layer. Even though the majority of the configuration for this implementation of A/B deployments originated at the service layer, it is only used to filter the available pods based on their labels and selectors. The IP addresses of the pods matching the service criteria are added as endpoints for the service. OpenShift's routing plugin monitors changes to the endpoints for the service and modifies the backends corresponding to the route as pods are created and destroyed. For OpenShift's default HAProxy routing plugin, requests communicating insecurely (non HTTPS) are weighted equally across both versions of the application and load balanced based on the application having the least number of connections. An example of HAProxy's configuration for the shopernetes-shared backend is shown below:

Listing 14. HAProxy configuration for a shared service

```
$ cat /var/lib/haproxy/conf/haproxy.config
...omitted...
backend be_http_shopernetes_shopernetes-shared

  mode http
  option redispatch
  option forwardfor

  balance leastconn

  timeout check 5000ms
  http-request set-header X-Forwarded-Host %[req.hdr(host)]
  http-request set-header X-Forwarded-Port %[dst_port]
  http-request set-header X-Forwarded-Proto http if !{ ssl_fc }
  http-request set-header X-Forwarded-Proto https if { ssl_fc }

  cookie 4750d13e6015cced828fd9cfafb0c621 insert indirect nocache httponly

  http-request set-header Forwarded for=%[src];host=%[req.hdr(host)];proto=%[req.hdr(X-
Forwarded-Proto)]


  server 77efd45f7b173c759015763107fe7640 172.17.0.5:8080 check inter 5000ms cookie
77efd45f7b173c759015763107fe7640 weight 100

  server 89426151b8d2f5b687416eabd3705b71 172.17.0.7:8080 check inter 5000ms cookie
89426151b8d2f5b687416eabd3705b71 weight 100
...omitted...
```

A drawback with the shared service approach is that it becomes difficult to tailor the exact amount of traffic received by each version of the application due to the default weighting and load balancing logic. One of the approaches to this situation is to modify the number of replicas deployed for each version of the application. For example, if there was a desire for a 75/25 split between the old versions of the application and the new version, 6 replicas of the old versions and 2 replicas of the new version

could be deployed in order to maintain a minimal amount of high availability within each version. While the existing routing logic would continue to dictate how requests are handled, there is a good likelihood the number of requests reaching each application would be closer to the desired ratio between each deployed version. Alternatively, support is also available to modify the default configuration template within HAProxy to introduce additional logic at the routing layer, but this is beyond the scope of this document.

Weighted Traffic Flow

If the limitations associated with the shared service approach for A/B deployments proves to be too much of a hindrance, alternate A/B deployment mechanisms are available within OpenShift. Incoming requests can be balanced across multiple services, one for each application deployment, rather than leveraging a shared service. Logic is instead maintained at the routing layer within a route and specifies the amount of traffic that should be sent to each service.

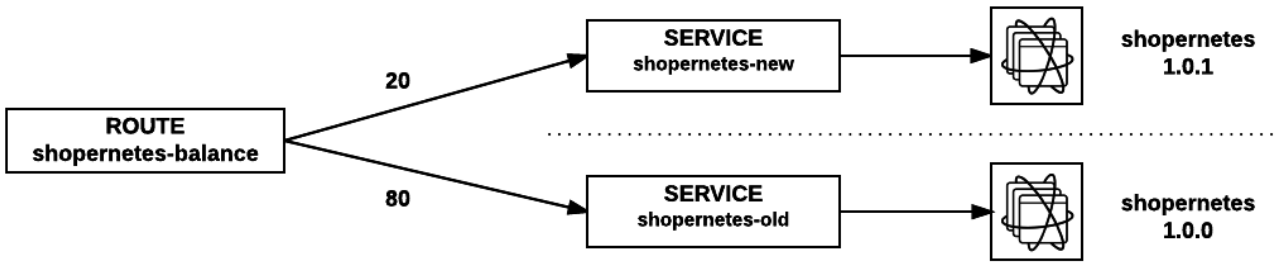

Figure 11. A/B deployments with traffic flow control within a route

The route defines both a primary service as seen previously, along with a new set of alternate backends (or services) that may also receive requests. An example of a route using this type of configuration is shown below.

Listing 15. Example of a weighted route

```
$ oc export route shopernetes-weighted
apiVersion: v1
kind: Route
metadata:
  name: shopernetes-weighted
spec:
  alternateBackends:
  - kind: Service
    name: shopernetes-new
    weight: 30
  host: shopernetes.ocp.example.com
  port:
    targetPort: 8080
  to:
    kind: Service
    name: shopernetes-old
    weight: 70
```

Both the primary service and alternate backends specify a weight value corresponding to the frequency requests should be sent to the applications in each service. Weighting is based on HAProxy's internal load balancing mechanisms where values range from 1 to 256 with higher values taking precedence. The combined weights across all services in a backend sets the overall relative proportion of traffic. The only difference in the resulting router configuration between typical routes and routes configured with an alternate backend is the weight property on each server:

Listing 16. HAProxy configuration for a weighted route

```
$ cat /var/lib/haproxy/conf/haproxy.config
...omitted...
backend be_http_shopernetes_shopernetes-weighted

  mode http
  option redispatch
  option forwardfor

  balance leastconn

  timeout check 5000ms
  http-request set-header X-Forwarded-Host %[req.hdr(host)]
  http-request set-header X-Forwarded-Port %[dst_port]
  http-request set-header X-Forwarded-Proto http if !{ ssl_fc }
  http-request set-header X-Forwarded-Proto https if { ssl_fc }

  cookie 0e53298f05e0ffccf54fb276e2dc5054 insert indirect nocache httponly

  http-request set-header Forwarded for=%[src];host=%[req.hdr(host)];proto=%[req.hdr(X-
Forwarded-Proto)]

  server 6e71445c89045923fb8f0c37cdf44a4f 172.17.0.5:8080 check inter 5000ms cookie
6e71445c89045923fb8f0c37cdf44a4f weight 70
  server 986ee5605b4936896954ca8ec1cb25d3 172.17.0.7:8080 check inter 5000ms cookie
986ee5605b4936896954ca8ec1cb25d3 weight 30
...omitted...
```

To introduce additional users to the new version of the application, adjust the weight value higher for the service representing the new version of the application and adjust the weight value lower for the service representing the old version. Continue to do so until a point in time where all requests will be received by the new version of the application. At that point, the alternate backend can be removed from the route and the service reference can instead point to the service representing the newly released version of the software to complete the cutover.

For those seeking a robust method for running multiple versions of an application concurrently, proper traffic flow control using the weighted method of A/B deployments in OpenShift provides a way to gauge real world user responses for testing and validation purposes. This level of feedback is crucial to safely and reliably release new software.

6. Automate Everything

Finally, as the options surrounding releasing applications in the OpenShift ecosystem continues to expand, there is a desire to further automate and integrate existing tools and methodologies into the application release process. Regardless of the tool or process being investigated, the key concern typically centers on how to communicate with the platform. Aside from the OpenShift web console which serves as an entrypoint for users to interact with the platform in a graphical manner, two primary mechanisms are available as options: The Command Line Interface (CLI) and Rest API. The CLI, as demonstrated throughout the course of the discussion, provides the greatest amount of the flexibility; from a usability point of view, the CLI can run across most user platforms and support is available for managing the majority of components within OpenShift. Since the CLI can be executed in either an interactive or non-interactive manner, it is an ideal execution tool to be invoked within automated scripts. These scripts can be executed in a standalone manner but in most cases are incorporated into an automated release process or as part of a Continuous Integration and Continuous Delivery solution. Further adoption of CI/CD principles allows for the creation of a build and deployment pipeline which models the lifecycle of an application into a repeatable process. Typical use cases for invoking the CLI in a pipeline include creating and modifying routes, services, deployment configuration and policy.

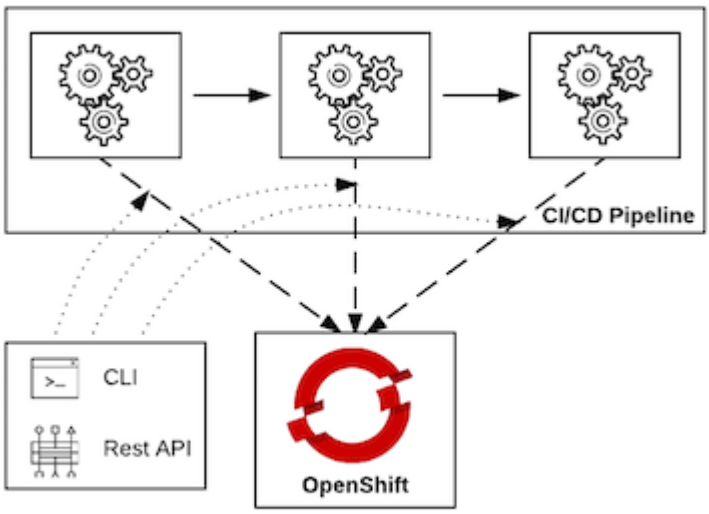

Figure 12. Mechanisms to manage a CI/CD pipeline

Each remote invocation of the CLI ultimately communicates with the Rest API to perform the actual execution. The API is the definitive mechanism for communicating with OpenShift and exposes the full set of capabilities and options supported by the platform. The API is not only used by the CLI, but also used by the web console and even the platform itself as the distributed components communicate with one another. Instead of incorporating the CLI in an automated solution, an alternate approach is to invoke the Rest API directly. Eliminating the CLI would also remove the requirement for a binary executable from being needed and maintained on the source system. When developing integrations with OpenShift using the API, aside from browsing the provided documentation, one of the options

available is to investigate the communication already being leveraged by the CLI and model solutions to mimic the execution. A detailed level of API requests and responses can be viewed by adding *--loglevel=8* to any CLI command as shown below:

Listing 17. Debugging a CLI command invocation

```
$ oc get routes --loglevel=8
...omitted...
I1123 09:34:57.151843   15231 round_trippers.go:296] GET
https://ocp.example.com:8443/oapi/v1/namespaces/shopernetes/routes
I1123 09:34:57.151856   15231 round_trippers.go:303] Request Headers:
I1123 09:34:57.151861   15231 round_trippers.go:306]     Accept: application/json, /
I1123 09:34:57.151865   15231 round_trippers.go:306]     User-Agent: oc/v3.3.0.34
(darwin/amd64) openshift/83f306f
I1123 09:34:57.151869   15231 round_trippers.go:306]     Authorization: Bearer
_ZcwFxPHI6uNZ5Au5bj64ug7oAVkq7b5IT-nv80Ogzo
I1123 09:34:57.156020   15231 round_trippers.go:321] Response Status: 200 OK in 4
milliseconds
I1123 09:34:57.156046   15231 round_trippers.go:324] Response Headers:
I1123 09:34:57.156055   15231 round_trippers.go:327]     Cache-Control: no-store
I1123 09:34:57.156062   15231 round_trippers.go:327]     Content-Type: application/json
I1123 09:34:57.156070   15231 round_trippers.go:327]     Date: Tue, 22 Nov 2016 07:08:26
GMT
I1123 09:34:57.156074   15270 request.go:901] Response Body:
{"kind":"RouteList","apiVersion":"v1","metadata":{"selfLink":"/oapi/v1/namespaces/shopern
etes/routes","resourceVersion":"106915"},"items":[{"metadata":{"name":"shopernetes","name
space":"shopernetes","selfLink":"/oapi/v1/namespaces/shopernetes/routes/shopernetes-
weighted","uid":"a870bde4-aa2a-11e6-b8cd-
5254007f3655","resourceVersion":"35056","creationTimestamp":"2016-11-
14T05:24:47Z","labels":{"createdBy":"shopernetes-
template","name":"shopernetes"},"annotations":{"openshift.io/host.generated":"true"}},"sp
ec":{"host":"shopernetes.ocp.example.com","to":{"kind":"Service","name":"shopernetes","we
ight":100},"port":{"targetPort":8080}},"status":{"ingress":[{"host":"shopernetes.ocp.exam
ple.com","routerName":"router","conditions":[{"type":"Admitted","status":"True","lastTran
sitionTime":"2016-11-14T05:24:47Z"}]}]}}]}
...omitted...
```

Alternatively, instead of developing custom solutions, there are a number of tools and plugins readily available to help automate the release process. One such plugin is the OpenShift Pipeline Plugin for the Jenkins CI server. By communicating with the Rest API, the Jenkins plugin abstracts the majority of the interaction surrounding the build and deployment of an application to configure and orchestrate a repeatable process targeting OpenShift.

The flexibility of the OpenShift is one of the hallmarks enjoyed by all users consuming the platform. Through an API centric approach for communication and the implementation of common release strategies and methodologies, robust solutions can be developed and implemented in order for accelerating application delivery. While though there are numerous options available within

OpenShift, the best advice that can be given is to experiment with each of the available mechanisms and to determine the path that proves to be the most successful for the particular use case. Continue to refine and iterate in order to adapt and be successful in the ever changing technological landscape.

Appendix A: Revision History

Revision	Release Date	Author(s)
1.0	Sunday January 22, 2017	Andrew Block
PDF generated by Asciidoctor PDF		
Reference Architecture Theme version 1.2		

Appendix B: Contributors

Shachar Borenstein, content reviewer

Scott Collier, content reviewer

Erik Jacobs, content reviewer

Kevin Newman, content reviewer