OpenShift – Builds

Raghavendra Deshpande
IBM Developer Advocate

Recap/Quiz time

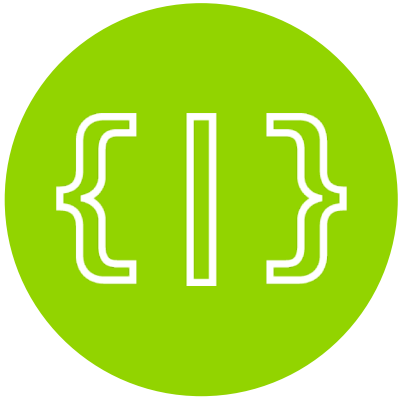
- Q1 – What is the smallest compute unit in Openshift
- Q2 – What is Service in the context of Openshift
- Q3 – What is Route in the context of Openshift
- Q4 – What is a project? Why does it matter?
- Q5 – What are the different components of Master Node
- Q6 – What is the command to create a project through command line
- Q7 – What is the command to create Route through CLI
- Q8 – What is the use of labels/selector for different resource objects
- Q9 – What is the command to create a new application from image
- Q10 – Different operations on Openshift cluster created on IBM Cloud

Agenda

- Building Container Images with OpenShift
- Build Strategies and Triggers

Building Container Images with OpenShift

BUILD AND DEPLOY CONTAINER IMAGES

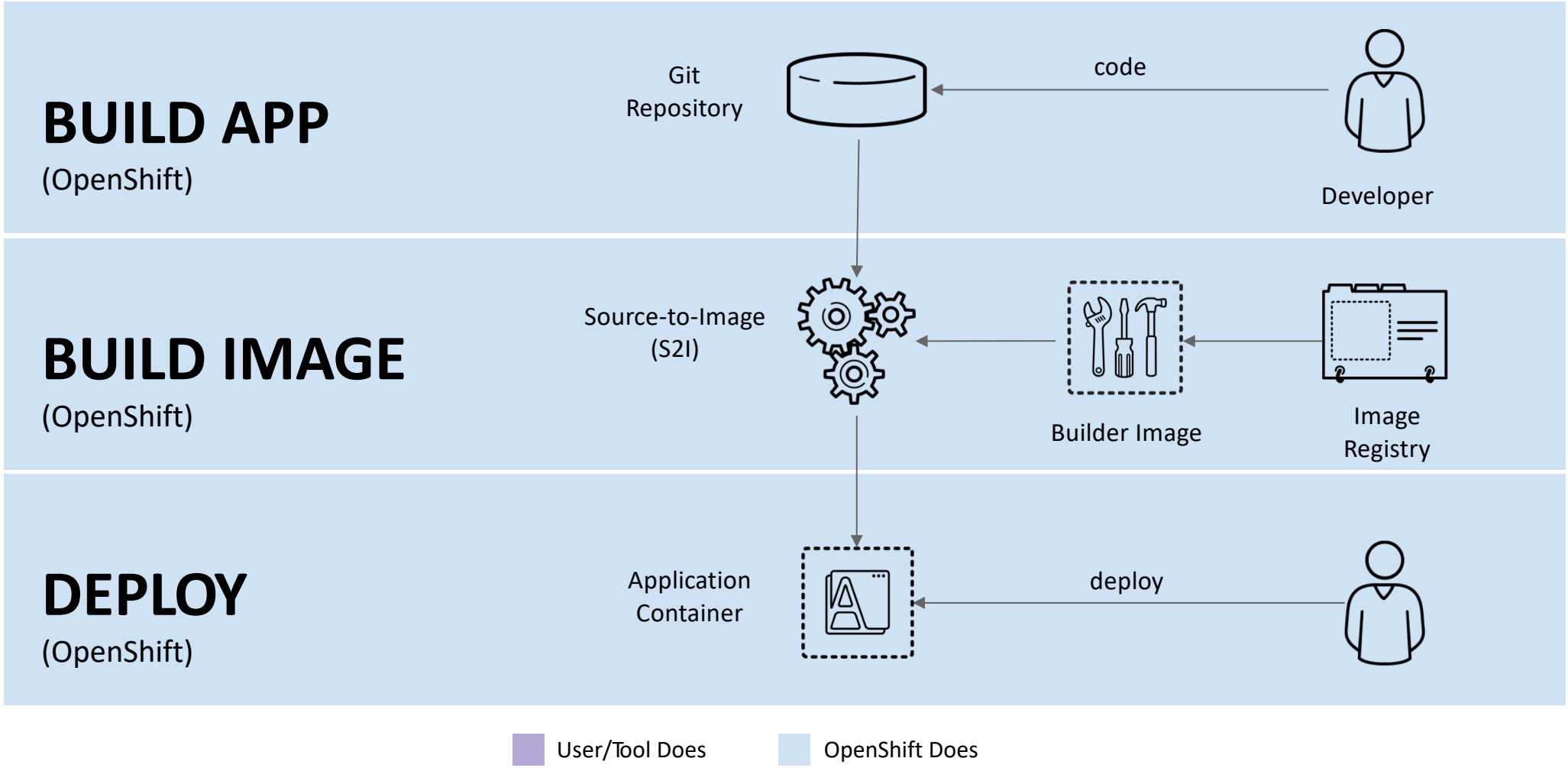


DEPLOY YOUR SOURCE CODE

DEPLOY YOUR APP BINARY

DEPLOY YOUR CONTAINER IMAGE

DEPLOY SOURCE CODE WITH SOURCE-TO-IMAGE (S2I)

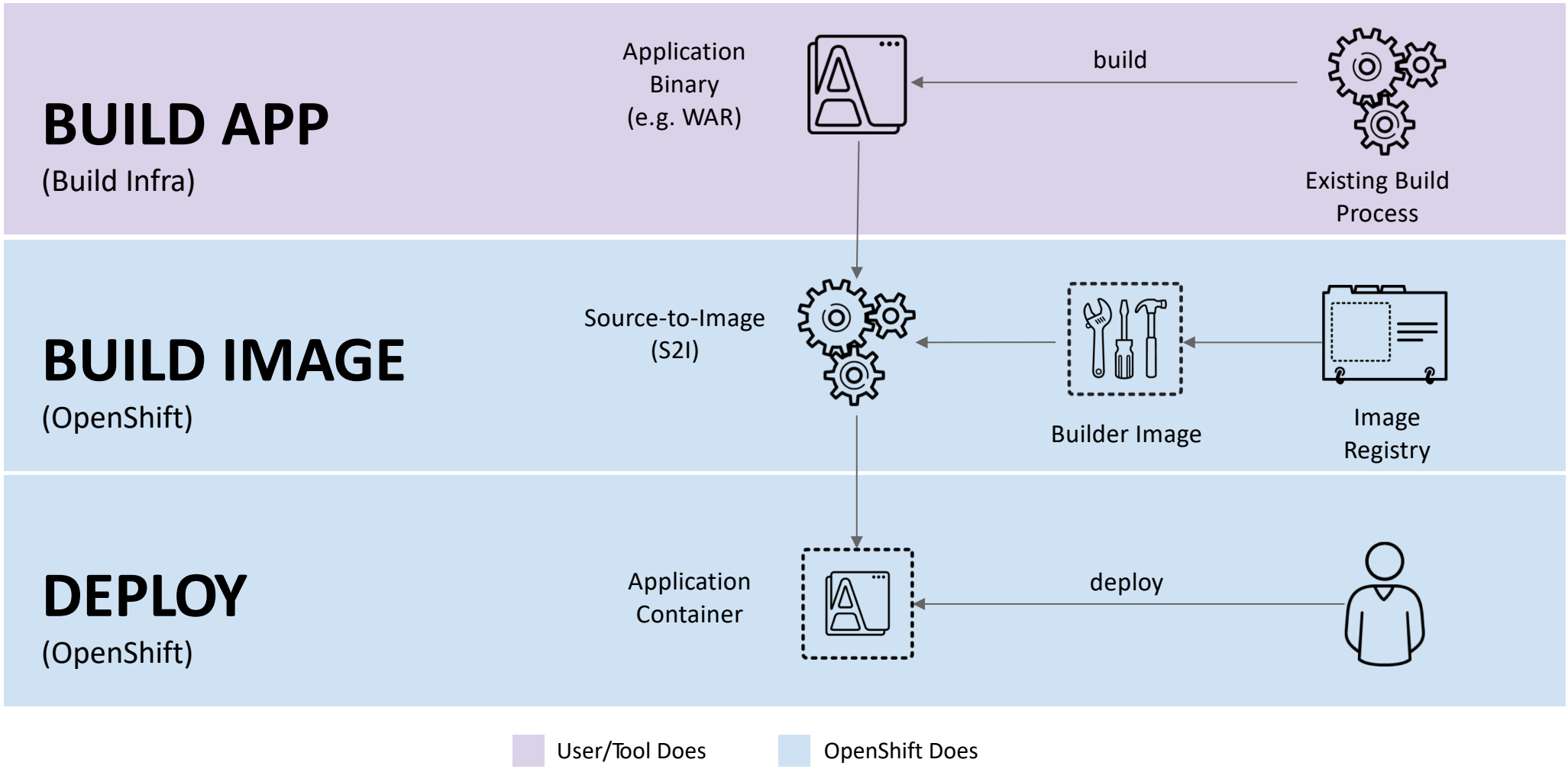


S2I (Source Code) - When the Source build strategy is invoked by oc new-app, it sets up two steps.
The first step is to run the build using S2I, combining the source files with the builder image to create the runnable image.
The second step is to deploy the runnable image and start up the web application.

```
$ oc new-app --name blog \
   python:3.5~https://github.com/openshift-katacoda/blog-django-py

--> Found image 956e2bd (5 days old) in image stream "openshift/python"
    under tag "3.5" for "python"

    Python 3.5
    ----------
    Platform for building and running Python 3.5 applications

    Tags: builder, python, python35, rh-python35

    * A source build using source code from
      https://github.com/openshift-katacoda/blog-django-py will be created
      * The resulting image will be pushed to image stream "blog:latest"
      * Use 'start-build' to trigger a new build
    * This image will be deployed in deployment config "blog"
    * Port 8080/tcp will be load balanced by service "blog"
      * Other containers can access this service through the hostname "blog"

--> Creating resources ...
    imagestream "blog" created
    buildconfig "blog" created
    deploymentconfig "blog" created
    service "blog" created
--> Success
    Build scheduled, use 'oc logs -f bc/blog' to track its progress.
    Run 'oc status' to view your app.
```

```
$ oc new-build --name blog \
  python:3.5~https://github.com/openshift-katacoda/blog-django-py
  --> Found image 956e2bd (5 days old) in image stream "openshift/python"
      under tag "3.5" for "python"

      Python 3.5
      ----------
      Platform for building and running Python 3.5 applications

      Tags: builder, python, python35, rh-python35

      * A source build using source code from
        https://github.com/openshift-katacoda/blog-django-py will be created
        * The resulting image will be pushed to image stream "blog:latest"
        * Use 'start-build' to trigger a new build

  --> Creating resources with label build=blog ...
      imagestream "blog" created
      buildconfig "blog" created
  --> Success
      Build configuration "blog" created and build triggered.
      Run 'oc logs -f bc/blog' to stream the build progress.
$ oc new-app blog
--> Found image 6792c9e (32 seconds old) in image stream "myproject/blog"
    under tag "latest" for "blog"

    myproject/blog-1:0b39e4f7
    -------------------------
    Platform for building and running Python 3.5 applications

    Tags: builder, python, python35, rh-python35

    * This image will be deployed in deployment config "blog"
    * Port 8080/tcp will be load balanced by service "blog"
      * Other containers can access this service through the hostname "blog"

--> Creating resources ...
    deploymentconfig "blog" created
    service "blog" created
$ oc start-build bc/blog --from-dir=.
Uploading directory "." as binary input for the build ...
build "blog-3" started
```

DEPLOY APP BINARY WITH SOURCE-TO-IMAGE (S2I)



Binary Input Builds

```
$ oc new-build --name blog --binary --strategy=source --image-stream python:3.5
--> Found image 440f01a (6 days old) in image stream "openshift/python"
    under tag "3.5" for "python"

    Python 3.5
    ----------
    Platform for building and running Python 3.5 applications

   Tags: builder, python, python35, rh-python35

   * A source build using binary input will be created
     * The resulting image will be pushed to image stream "blog:latest"
     * A binary build was created, use 'start-build --from-dir' to trigger
       a new build

--> Creating resources with label build=blog ...
    imagestream "blog" created
    buildconfig "blog" created
--> Success
```

```
$ oc start-build blog --from-dir=.
Uploading directory "." as binary input for the build ...
build "blog-1" started
```

```
$ oc new-app --name blog --build-env UPGRADE_PIP_TO_LATEST=1 \
   python:3.5~https://github.com/openshift-katacoda/blog-django-py
```

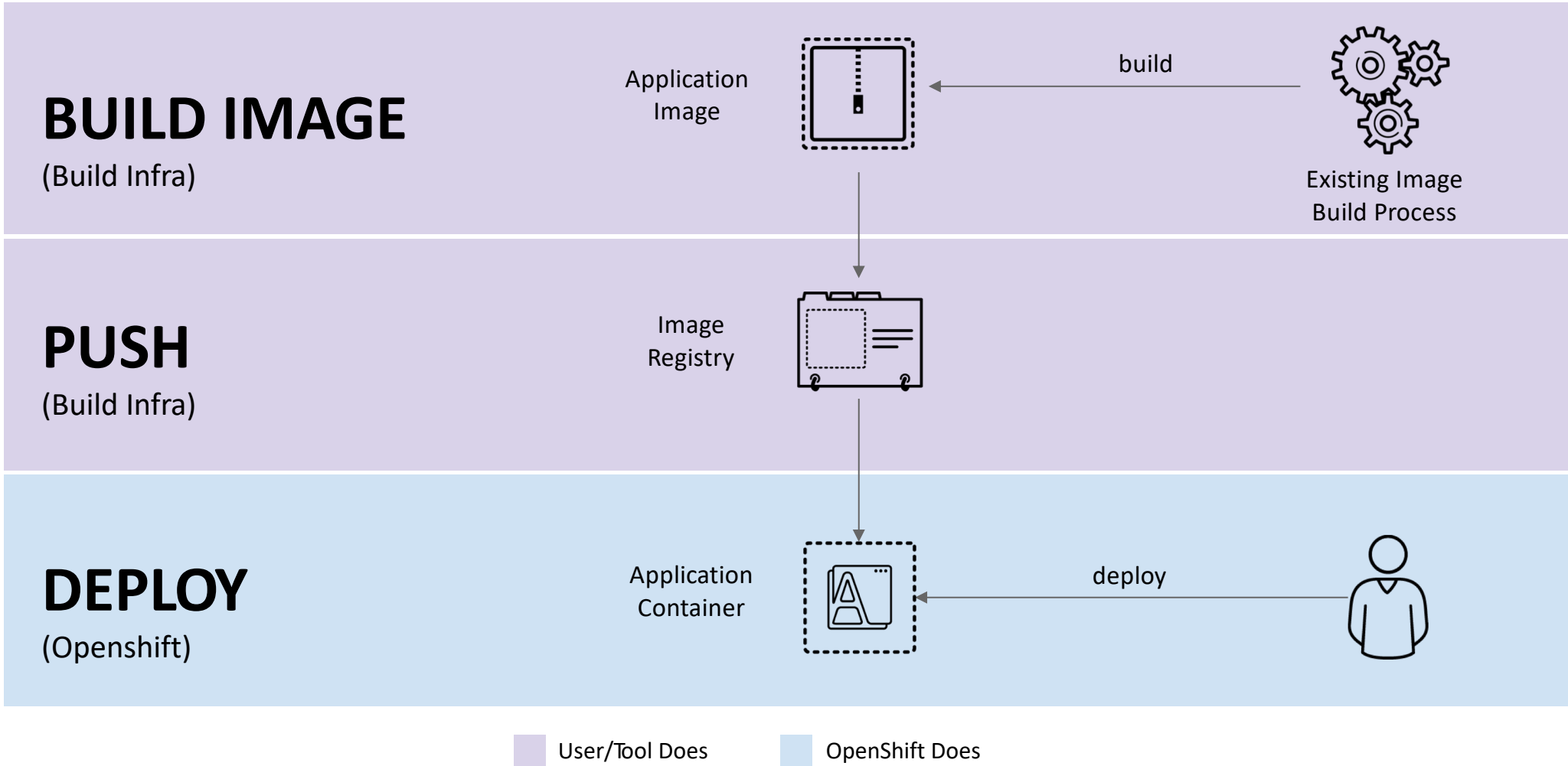
If using the two-step approach of invoking `oc new-build` and `oc new-app`, the environment variables specific to the build step should be passed to `oc new-build` using the `--env` option:

```
$ oc new-build --name blog --env UPGRADE_PIP_TO_LATEST=1 \
   python:3.5~https://github.com/openshift-katacoda/blog-django-py
```

If the environment variables need to be added after the build configuration has been created, the `oc set env` command can be used:

```
$ oc set env bc/blog UPGRADE_PIP_TO_LATEST=1
buildconfig "blog" updated
```

DEPLOY DOCKER IMAGE



Example -

```
$ oc new-app openshiftkatacoda/blog-django-py --name blog
--> Found Docker image 0f405dd (5 days old) from Docker Hub
    for "openshiftkatacoda/blog-django-py"

    ...

    * An image stream will be created as "blog:latest" that will track
      this image
    * This image will be deployed in deployment config "blog"
    * Port 8080/tcp will be load balanced by service "blog"
      * Other containers can access this service through the hostname "blog"

--> Creating resources ...
    imagestream "blog" created
    deploymentconfig "blog" created
    service "blog" created
--> Success
    Run 'oc status' to view your app.
```

```
$ oc expose service/blog
route "blog" exposed
```

```
$ oc status
In project My Project (myproject) on server https://127.0.0.1:8443

http://blog-myproject.127.0.0.1.nip.io to pod port 8080-tcp (svc/blog)
  dc/blog deploys istag/blog:latest
    deployment #1 deployed 1 minute ago - 1 pod

View details with 'oc describe <resource>/<name>' or list everything
with 'oc get all'.
```

```
$ oc get all -o name --selector app=blog
imagestreams/blog
deploymentconfigs/blog
replicationcontrollers/blog-1
routes/blog
services/blog
pods/blog-1-36f46
```

```
$ oc scale --replicas=3 dc/blog
deploymentconfig "blog" scaled
```

```
$ oc delete all --selector app=blog
imagestream "blog" deleted
deploymentconfig "blog" deleted
route "blog" deleted
service "blog" deleted
```

The Docker Build Strategy

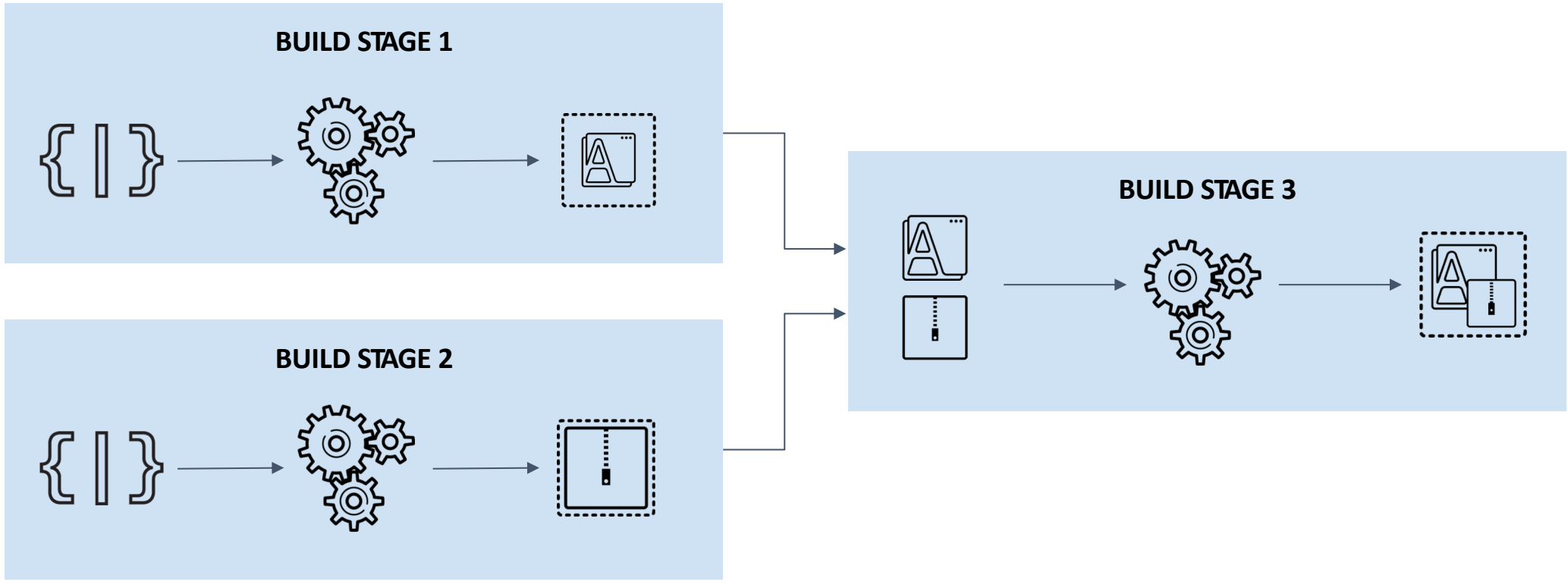
```
$ oc new-build --name blog --strategy=docker \
  https://github.com/openshift-katacoda/blog-django-py
  --> Found Docker image 956e2bd (5 days old) from Docker Hub
      for "centos/python-35-centos7:latest"

      ...

      * An image stream will be created as "python-35-centos7:latest"
        that will track the source image
      * A Docker build using source code from
        https://github.com/openshift-katacoda/blog-django-py will be
        created
        * The resulting image will be pushed to image stream "blog:latest"
        * Every time "python-35-centos7:latest" changes a new build will
          be triggered

  --> Creating resources with label build=blog ...
      imagestream "python-35-centos7" created
      imagestream "blog" created
      buildconfig "blog" created
  --> Success
      Build configuration "blog" created and build triggered.
      Run 'oc logs -f bc/blog' to stream the build progress.
```

```
$ oc new-app blog
--> Found image 1f6debb (5 minutes old) in image stream "myproject/blog"
    under tag "latest" for "blog"

    ...

    * This image will be deployed in deployment config "blog"
    * Port 8080/tcp will be load balanced by service "blog"
      * Other containers can access this service through the
        hostname "blog"

--> Creating resources ...
    deploymentconfig "blog" created
    service "blog" created
--> Success
    Run 'oc status' to view your app.
```

```
$ oc new-app --name blog --strategy=docker \
  https://github.com/openshift-katacoda/blog-django-py
```

```
$ oc new-build --name blog --strategy=docker \
  --build-arg HTTP_PROXY=https://proxy.example.com \
  https://github.com/openshift-katacoda/blog-django-py
```

BUILD IMAGES IN MULTIPLE STAGES



BUILD STRATEGIES

The following are available build strategies in OpenShift:

- Source
- Docker
- Custom

* Pipeline

BUILDCONFIG RESOURCE

BuildConfig object definition

```
kind: BuildConfig
apiVersion: build.openshift.io/v1
metadata:
  name: "ruby-sample-build" (1)
spec:
  runPolicy: "Serial" (2)
  triggers: (3)
    -
      type: "GitHub"
      github:
        secret: "secret101"
    - type: "Generic"
      generic:
        secret: "secret101"
    -
      type: "ImageChange"
  source: (4)
    git:
      uri: "https://github.com/openshift/ruby-hello-world"
  strategy: (5)
    sourceStrategy:
      from:
        kind: "ImageStreamTag"
        name: "ruby-20-centos7:latest"
  output: (6)
    to:
      kind: "ImageStreamTag"
      name: "origin-ruby-sample:latest"
  postCommit: (7)
      script: "bundle exec rake test"
```

1. This specification creates a new `BuildConfig` named **ruby-sample-build**.
2. The `runPolicy` field controls whether builds created from this build configuration can be run simultaneously. The default value is `Serial`, which means new builds run sequentially, not simultaneously.
3. You can specify a list of triggers, which cause a new build to be created.
4. The `source` section defines the source of the build. The source type determines the primary source of input, and can be either `Git`, to point to a code repository location, `Dockerfile`, to build from an inline Dockerfile, or `Binary`, to accept binary payloads. It is possible to have multiple sources at once. Refer to the documentation for each source type for details.
5. The `strategy` section describes the build strategy used to execute the build. You can specify a `Source` , `Docker`, or `Custom` strategy here. This example uses the `ruby-20-centos7` container image that Source-To-Image uses for the application build.
6. After the container image is successfully built, it is pushed into the repository described in the `output` section.
7. The `postCommit` section defines an optional build hook.

BUILD INPUT SOURCES

- **Git**: The input source is cloned from a Git repository. It is possible to configure the default location inside the repository where the build looks for application source code.

- **Dockerfile**: Specifies the Dockerfile inline to build an image. This Dockerfile overrides a Dockerfile from a Git repository.

- **Binary**: Allows streaming binary content from a local file system to the builder.

- **Image**: Additional files can be provided to the build process from images.

- **Input secrets**: Allow creating credentials for the build that will not be available in the final application image.

- **External artifacts**: Allow copying binary files to the build process.
  Combining multiple inputs into a single build is possible. Binary and Git are mutually exclusive inputs.

Source definition: An example

```
source:
  git:
    uri: https://github.com/openshift/ruby-hello-world.git (1)
  images:
  - from:
      kind: ImageStreamTag
      name: myinputimage:latest
      namespace: mynamespace
    paths:
    - destinationDir: app/dir/injected/dir (2)
      sourcePath: /usr/lib/somefile.jar
  contextDir: "app/dir" (3)
  dockerfile: "FROM centos:7\nRUN yum install -y httpd" (4)
```

(1) The repository to be cloned into the working directory for the build.

(2) */usr/lib/somefile.jar* from `myinputimage` will be stored in *<workingdir>/app/dir/injected/dir*.

(3) The working directory for the build will become *<original_workingdir>/app/dir*.

(4) A Dockerfile with this content will be created in *<original_workingdir>/app/dir*, overwriting any existing file with that name.

TRIGGERING BUILDS

- **Image change triggers**
- **Configuration change triggers**
- **Web hooks**

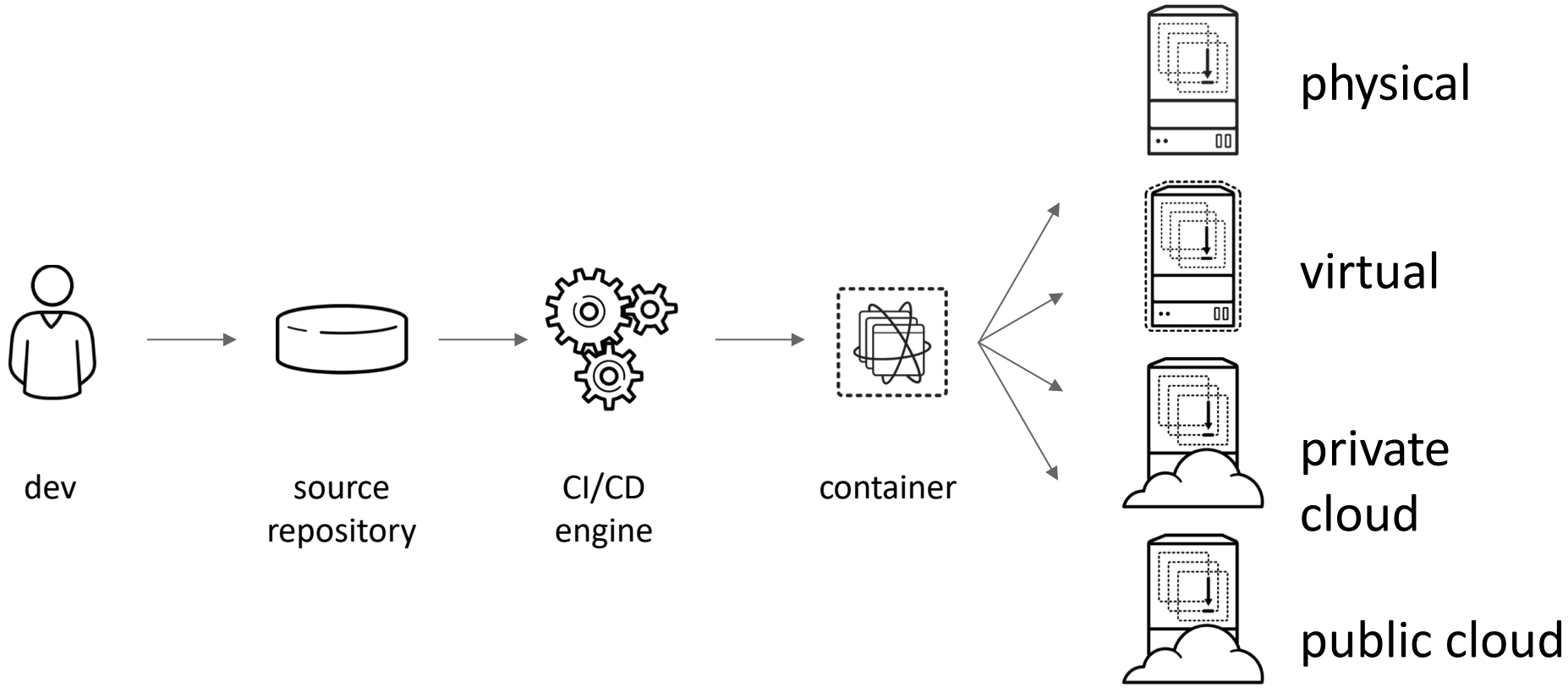
Post-commit Build hooks

There are two types of Post-commit build hooks:

1) Command: A command is executed using the exec system call

2) Script: Runs a build hook with the **/bin/sh -ic** command.

CONTINUOUS DELIVERY WITH CONTAINERS



Quick Recap

- ✓ How builds work?
- ✓ How triggering builds and applying post-commit build hooks work on OpenShift?

Thank you!

Raghavendra Deshpande
(Developer Advocate- IBM)
 @ragdeshp