AKS Build Release Management

Last updated by | Fabian Flanhardt | Apr 13, 2021 at 10:47 AM GMT+2

Build- and Release-Management in Azure Kubernetes Services

Contents

Goal: Documentation of processes for build, continuous deployment and upgrades, including test framework for new releases, cross-version acceptance criteria, etc.

Build process and continuous integration

This chapter describes concepts and best practices for a build process, i.e. an automated series of tasks that take source code and other parameters as input, run automated tests, publish test results to provide developer feedback, and create ready-to-deploy artifacts as output. Continuous integration (CI) is a term that describes the practice of running the build pipeline whenever changes to the source code are pushed into version control. CI makes it possible to identify any issues quickly, and to automatically provide useful information about such issues to the development team.

Version control and branching

In order to establish an automated build process, the following requirements have to be met.

A version control system such as Git should be used in order to be able to work together in teams with multiple developers, to keep track of all changes to source code (including infrastructure as code), and to be able to revert to any previous version of the code in case of any problems.

A branching convention such as Gitflow should be applied in order to standardize workflows and to allow for simultaneous development on different parts of the software project by multiple team members. A proper branching convention should be accompanied by rules for code reviews as part of pull requests (PRs). A set of rules for PR reviews should be defined for the project, a common rule is to have at least two developers approve any changes before merging them into the main codebase. It should also be forbidden to commit directly into the main branch as this would circumvent the review process. Rules like these should be enforced by the CI/CD platform.

A versioning convention such as SemVer should be used in conjunction with Git tags to be able to link artifacts to their source code at any given point of time and to be able to roll back to any previous version in case any problems should arise after the deployment of a new version.

Build pipeline

A build pipeline should be used to implement an automated, deterministic, reproducible and self-documenting process that is used to create artifacts that can be deployed into a given environment. It can be implemented using platforms such as Azure DevOps, Jenkins or GitLab. Generally speaking, a build pipeline takes source code, runs tests, and creates deployable artifacts. In the context of AKS, the output of a build pipeline is commonly a container image pushed into a container registry.

The following automated tasks are common for build pipelines.

- Check out source code from version control
- Compile the code using build tools that fit the project's programming language, e.g. Maven for Java projects or MSBuild for .NET projects
- Run static code analysis and automated tests such as unit tests and integration tests (see the chapter on test frameworks and global acceptance criteria below for more information)
- Provide feedback about results of the aforementioned analysis and tests to the development team
- If all tests completed successfully, publish artifacts to an artifact management system

In the context of AKS, build pipelines often use a base image upon which the final container image will be built. Base images can be publicly available images created, for instance, by the Docker open source community, or can be custom built images suited to a specific software project. In case of custom base images, the build process for the base image itself should be implemented in a build pipeline.

The build pipeline should automatically run whenever changes to the code are pushed into version control (continuous integration). This means that the whole suite of analytics and tests will run for any new code, providing quick feedback to the development team in case of any issues. Developers should automatically be notified by the system if the build pipeline did not run successfully.

The pipeline definition should also be kept under version control if the CI platform supports this, e.g. pipelines can be described as YAML in Azure DevOps. Putting the pipeline itself under version control provides the possibility to restore any version of the build process, thus making it possible to re-build any historic version of the project even if the original artifacts are lost.

Artifact management

An artifact management system, also called artifact repository, should be used to store outputs from build pipelines. Artifacts in the management system should be versioned in the same way that releases are tagged in version control. This makes it possible to link artifacts back to their exact version of the source code.

Retention rules can be applied to clean up older artifacts that are no longer needed. It's a common practice to regularly clean up artifacts from CI builds and to store only release artifacts indefinitely.

Continuous deployment

This chapter describes best practices for continuous deployment, i.e. the process of automatically deploying new releases of an application into a given environment, with little to no manual intervention. Like continuous integration, continuous deployment is achieved by creating a deployment pipeline that optionally takes the artifacts from a previous build process and performs all the necessary steps to get the new version up and running in the selected environment. It is a common practice to automatically and immediately deploy new versions into a staging environment where testing can be performed, and requiring manual confirmation to deploy into production after testing has completed successfully. Rules like this should be enforced by the CI/CD platform.

Deployment types

Deployments can be categorized into infrastructure deployments and application deployments.

Infrastructure deployments

In the context of cloud infrastructure such as AKS, infrastructure deployments perform tasks that set up the cloud environment to support later application deployments, e.g. the cluster itself, policies, shared resources and so on. It is a good practice to use intrastructure as code (IaC), meaning that the infrastructure is declared in a way that can be used to automatically set up resources in the cloud, using tools like Terraform, Pulumi or Bicep, without requiring manual intervention. Subsequent changes to infrastructure should also be described as IaC and deployed using the pipeline. Developers should refrain from performing manual changes that circumvent the deployment pipeline, as this would invalidate the advantages that an automated process provides, like predictability and reproducibility.

Application deployments

Application deployments, also called workload deployments, perform tasks necessary to deploy an application into a given environment. In the context of AKS, this is commonly a container to be deployed into a Kubernetes Pod, where the container is the output of a previously run build pipeline. As with build processes and infrastructure deployments, application deployments should also be implemented using automated pipelines.

When using AKS, developers should make use of Kubernetes features and additional tooling to support deployments, like liveness/readiness probes and Helm.

Liveness probes are used to check if a container is "live", meaning that it is running as expected and able to reply to incoming requests. If the liveness probe fails, a container can be restarted as an effort to get it "live" again. Cases of failed liveness probes should be investigated nonetheless, as they could indicate a bug in the hosted service. Readiness probes should be used to check if a container is "ready", meaning that it is in a state where it's able to receive traffic. For example, a container might be "not ready" to receive traffic during a long-running, blocking operation, but will report to be "ready" again after the process completes. The following example defines a simple readiness probe that reponds to HTTP calls, waiting 10 seconds for the initial check and then running the check every 5 seconds:

```
  readinessProbe:
    httpGet:
      path: /ready
      port: 3000
    initialDelaySeconds: 10
    periodSeconds: 5
```

Helm is a package manager for Kubernetes that provides so-called "Charts", i.e. ready-to-use configurations of a given application that can be deployed into a Kubernetes cluster. For example, a database chart could be used to get a specific type of database up and running quickly, without having to create all of the required boilerplate configuration by hand.

Environments

Environments are commonly grouped into testing, staging and production, all separated from each other and each one serving a specific purpose as described below.

- Testing environments are used by developers to test changes to code, be it IaC code, application code, or changes to the build or deployment pipelines themselves. Deployments can be triggered on demand in order to support developer testing.
- Staging is where a new release is commonly deployed to before promoting it to production. The staging environment should mirror the production environment as closely as possible in order to be able to run meaningful tests. Manual user acceptance tests are commonly performed in the staging environment (see chapter below for more information).
- Production is the environment where service instances are put into actual operation, accessible to the service's users. Whenever a new release has been promoted through all of the previous stages, and all tests have completed successfully, the release will then be pushed into production.

Error handling

Even with mandated code reviews, a broad suite of automated and manual tests and a strict process for promoting releases, there is always a possibility that bugs will find their way into production. Whenever a bug is discovered (either by automated monitoring or by users), the development team has two options that need to be discussed and decided upon. The first option is to roll back to a previous version, the second option is a new release where the bug is fixed.

With deployment pipelines that follow best practices as described above, rolling back should be as simple as pushing a button that performs the deployment process for the last known working version of an application. Rolling back is preferred when the problem has an impact that needs immediate mitigation.

Fixing the bug and creating a new release (sometimes referred to as rolling forward) takes longer, because code changes are usually involed, as well as running all the tests and pipelines and promoting the new release through all of the stages as described above. This option can be preferred if the problem does not require immediate mitigation, or if the benefit of other new features in the current release outweigh the disadvantages of the problem that was found.

Development teams need to decide on a case-by-case basis which way forward is the better option.

Documentation on Test Framework and global acceptance criteria

This chapter describes several concepts of testing as well as best practices within agile development, which should be applied especially in the environment of micro-services and Kubernetes.

Common concepts for testing include the following:

- Continuous testing / Fast feedback
- Unit testing
- Integration testing
- User acceptance testing
- Kubernetes testing

Continuous testing / Fast feedback

All types of tests are used throughout the software development cycle, and are often subject to the same requirements as the software itself. Tests must therefore also be fully automated in order to be able to provide reliable results throughout the entire CI/CD process. They must be executed at every run of a build pipeline and supported by the build tools used. The automatic execution of build and test after the developer has uploaded his code state to the corresponding repository results in feedback loops. The pipelines used should therefore inform the developer about the result if the build was aborted at any point in the CI process.

Unit testing

Unit tests are typically automated tests written and run by software developers to ensure that a section of an application meets its design and behaves as intended. They are thus subject to the same iterative development process as the actual application. Developers should ensure that a corresponding unit test exists for each sub-functionality of an application. Often, separate tasks for writing the tests are scheduled for each development task in the development sprints. For unit tests, some frameworks have been established for implementation (e.g. TestNG, JUnit, XUnit) and in some programming languages these are already built-in (e.g. Go, Python, Rust). As development progresses, the tools used can calculate test coverage, which can be used as a metric in a software acceptance process, as well as an indicator for the quality and automation degree of the software project. Simply speaking, a test coverage indicates how much of the implemented application logic is actually used by the written tests. Stable tests and high test coverage give confidence in the stability of the application and quickly reveal any gray areas.

Integration testing

Integration tests are used to test several software components in combination, and are often used in micro-service architectures or other distributed systems. It is particularly important that the test environments used have a similar structure to the later production environment in order to deliver maximum comparable results. Containerized applications and Infrastructure as Code help to create reproducible environments on the development system, via CI/CD and acceptance systems through to production. A definition of the enclosing Infrastructure as Code (e.g., as a Docker image + Helm Chart) provides the developer the framework, to test his application on his local system against another one for functionality, thus simplifying debugging, accelerating and increasing the quality of development.

User acceptance testing

User acceptance testing (UAT) is often one of the last steps in the chain of tests to be performed. While unit and integration tests should be automated as much as possible, there are good reasons for the UAT to be performed by human users in some project iterations in order to identify error patterns that would not occur with the expected user behavior. However, this step can also be partially automated with suitable tools in order to achieve comparable results.

Kubernetes testing

During the deployment of an application, there are various mechanisms, especially in Kubernetes, to ensure that the deployment was completed successfully. These mechanisms can be used to implement automatic rollbacks or other self-healing processes. While mechanisms anchored in Kubernetes provide simple health checks during operation, such as startup, readiness or liveness probes, the package manager Helm has a possibility to execute tests during or after the deployment.

Upgrade Process Documentation

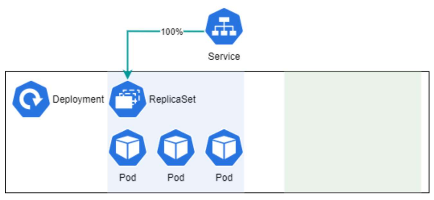
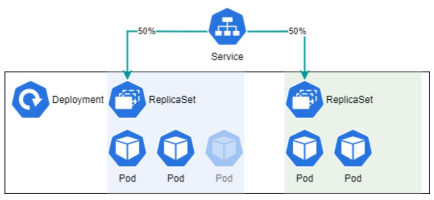
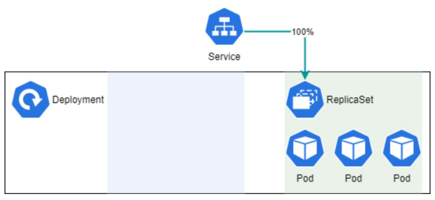
Zero-Downtime deployments

In order to deploy an application without downtime, various approaches exist and often the implementation of the deployment mechanism is the easier part while designing a corresponding system architecture. Before considering the rollout processes, the most important thing to ensure is that requests to the application are stateless, or that the state is not stored in an application server, which is part of the rollout. Instead, an external, central system, such as Redis or a database should be used. Sticky sessions should not be used.

Kubernetes has introduced the object `Deployment` to execute rolling updates of the rolled out ReplicaSets or Pods. Provided that the prerequisites mentioned at the beginning are implemented, this built-in mechanism can be used for zero-downtime deployments.

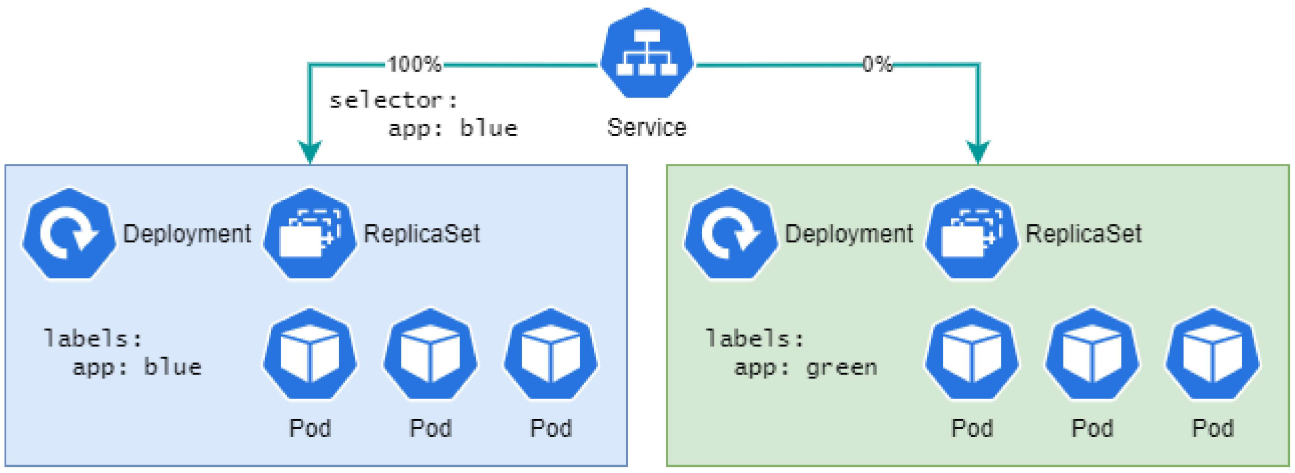
Example for the configuration of a `RollingUpdate` strategy. It is possible to configure the maximum number of unavailable replicas as well as the maximum number of additional replicas.

```
replicas: 5
strategy:
  type: RollingUpdate
  rollingUpdate:
    maxSurge: 25%
    maxUnavailable: 25%
```

Original deployment state	Parallel deployment of the new version. Incoming traffic is distributed in equal parts	Old pods have been removed. Only the new release is now active and receives the entire traffic
 100% Service Deployment ReplicaSet Pod Pod Pod	 50% Service 50% Deployment ReplicaSet ReplicaSet Pod Pod Pod Pod Pod	 Service 100% Deployment ReplicaSet Pod Pod Pod

A/B or Blue/Green Deployments

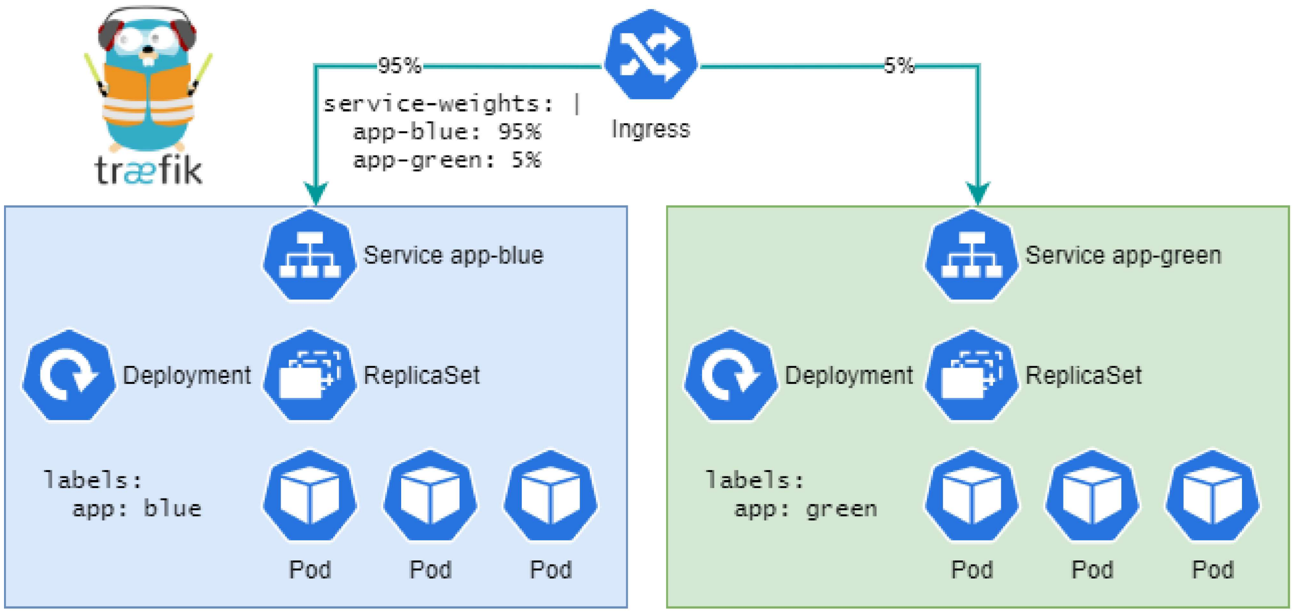
This type of deployment describes an approach in which a complete application part is deployed in parallel and traffic is distributed either to one branch or to the other. The described approach of `RollingUpdate` is extended by the possibility of simple switching between two versions. Typically, this involves using two parallel `Deployment` or `ReplicaSet` objects that have been labeled accordingly. A central Kubernetes `Service` object can be used to route the traffic. The type of the `Service` object determines whether the routing is DNS-based (via ClusterIP) or is done using an Azure loadbalancer.



Canary Deployments

Canary deployments extend the Blue/Green approach by the possibility to first distribute a smaller portion of the traffic (e.g. 5% of the total number of users) to another incarnation of the application and obtain an approximate statement about its performance. If unexpected error patterns suddenly appear with this part of the users, the routing can be adjusted again. If successful, all traffic is routed to the new instance. Comprehensive monitoring using Prometheus or similar is essential to detect these types of errors and to make a reliable rollback or roll-forward decision possible. There are various options for implementing canary deployments in Azure. Either DNS-based load balancing via Azure Traffic Manager, or a canary deployment-capable IngressControllers can be used (e.g., Traefik or Nginx Ingress). Weighting can be set granular as percentage for both solutions. Traffic Manager can be used if traffic is to be distributed across multiple Azure regions. The solution using IngressController is recommended when the application load is distributed across nodes within an single AKS cluster. The Azure Application IngressController as well as Azure Loadbalancer do not support canary deployments at this time.

```
annotations:
  traefik.ingress.kubernetes.io/service-weights: |
    my-app: 99%
    my-app-canary: 1%
```



Rollback/Roll-forward / Patch deployments

Each of the named deployment methods already brings options for rolling back an application. For example, in the case of a rolling update of the deployment, a simple `kubectl rollout undo` `deployment.v1.apps/<deployment>` or `helm rollback <RELEASE> [REVISION]` can be used to roll back to the last state. The approaches for A/B and Canary deployments also allow a rollback by configuring the routing. The prerequisite for this, however, is that the application itself also offers backward compatibility for internally used APIs or database schemas. This is often time-consuming to implement, which is why it is often not done in projects. However, before such deployment methods are used, precisely these questions must be clarified.

Alternatively, if there is sufficient test coverage, it can be decided to dispense with rollbacks altogether and instead rely on a roll-forward by means of patch deployments. This is only possible if the development processes allow this kind of flexibility and CI/CD is actively practiced. Automatically backing up all relevant components is a must-have.