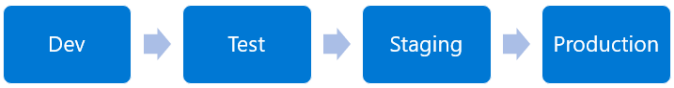
Deployment Patterns

A deployment pattern is a way of how you choose to move your application to production.

Traditional Deployment Pattern:



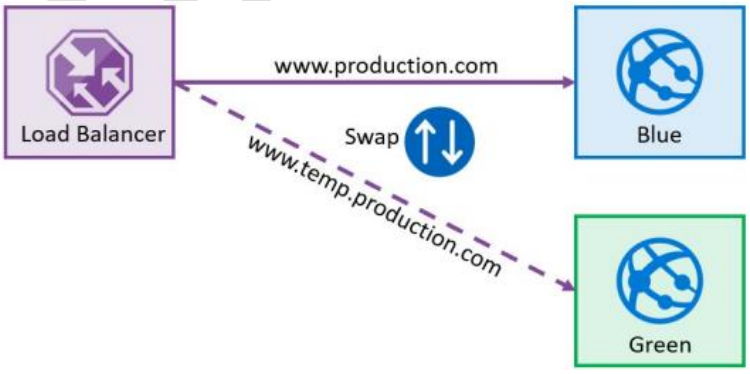
Modern Deployment Pattern

Blue Green Deployment

The Blue-Green deployment is all about ensuring you have two production environments, as identical as possible. After deployment and tests, the environments are swapped.

Why do you need it?

- When you want to deploy with zero downtime
- When you need to test in production
- When you want easy rollback



For this example, Blue is currently live, and Green is idle.

- As you prepare a new version of your software, the deployment and the final stage of testing takes place in the environment that is not live: in this example, Green. Once you have deployed and thoroughly tested

the software in Green, you switch the router or load balancer, so all incoming requests now go to Green instead of Blue. Green is now live, and Blue is idle.

- This technique can eliminate downtime due to app deployment. Besides, Blue-Green deployment reduces risk: if something unexpected happens with your new version on Green, you can immediately roll back to the last version by switching back to Blue.
- When using web apps, you can use an out-of-the box feature called Deployment Slots. Deployment Slots are a feature of Azure App Service. They are live apps with their own hostnames. You can create different slots for your application (e.g., Dev, Test or Stage). The Production slot is the slot where your live app resides. With deployment slots, you can validate app changes in staging before swapping it with your production slot.

App Service:

1. Blue-Slot is current production slot
2. Green-Slot
3. Publish to Green-Slot
4. Now we test - UI/Load on Green-Slot
5. Wait for Approval
6. Swap Green-Slot to Blue-Slot

Virtual Machine:

1. Blue VM is the current production machine
2. Create a New VM – Green VM
3. Publish the application to Green VM
4. Now we test - UI/Load on Green VM
5. Wait for Approval
6. In Load balancer first Add Green VM and remove Blue VM
7. Now Green VM is the current production machine.
8. Deprovision the BlueVM

Feature Toggles Feature

Feature toggles are also known as feature flippers, feature flags, feature switches, conditional features, etc. In today's fast-paced, feature-driven markets, it's important to continuously deliver value and receive feedback on features quickly and continuously. Partnering with end users to get early versions of features vetted out is valuable.

Why do you need it?

- It enables you to give control back to the business on when to release the feature
- Enables A/B testing, canary releases and dark launching

- It provides an alternative to keeping multiple branches in version control.
- Enables change without redeployment

The question is what strategy you want to use in releasing a feature to an end user.

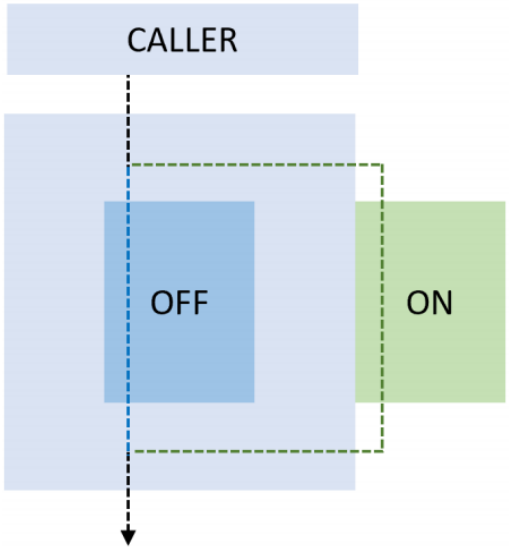
- Reveal the feature to a segment of users, so you can see how the new feature is received and used.
- Reveal the feature to a randomly selected percentage of users.
- Reveal the feature to all users at the same time.

Feature toggles are a great alternative to branching as well. Branching is what we do in our version control system. To keep features isolated, we maintain a separate branch. The moment we want the software to be in production, we merge it with the release branch and deploy.

With feature toggles, you build new features behind a toggle. When a release occurs, your feature is "off" and should not be exposed to or impacting the production software.

How to implement a feature toggle:

In the purest form, a feature toggle is an **IF statement**.



When the switch is off, it executes the code in the IF, otherwise the ELSE. Of course, you can make it much more intelligent, controlling the feature toggles from a dashboard, or building capabilities for roles, users, query string, UI in application etc.

The most important thing is to remember that you need to remove the toggles from the software as soon as possible. If you do not do that, they will become a form of technical debt if you keep them around for too long.

Canary Release

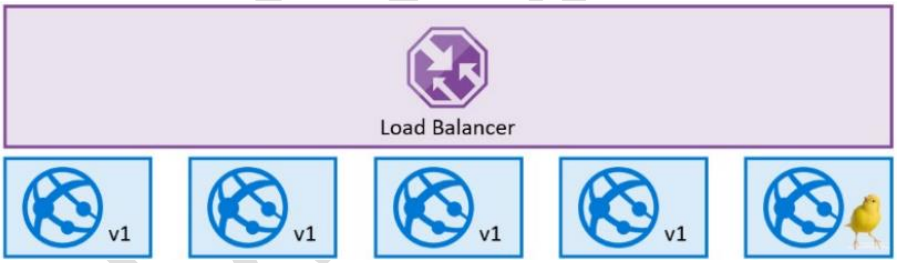
The term canary release comes from the days that miners took a canary with them into the coal mines. The purpose of the canary was to identify the existence of toxic gasses. The canary would die much sooner than the miner, giving them enough time to escape the potentially lethal environment.

- **A canary release is only pushed to a small number of users**.
- The idea is to first deploy the change to a small subset of servers, test it, and then roll the change out to the rest of the servers.
- Therefore, its impact is relatively small should the new code prove to be buggy. Changes can be reversed quickly. The rest of the servers aren't impacted.

How do you do it?

- Use a combination of feature toggles, traffic routing and deployment slots
- Random selection of users:
    - Setup deployment slots with feature enable
    - Route % of traffic to the instance with the new feature enabled
- Target specific user segment. Use feature toggle to enable feature for specific user segment

By closely monitoring what happens the moment you enable the feature, you can get relevant information from this set of users and decide to either continue or rollback (disable the feature). If the canary release shows potential performance or scalability problems, you can build a fix for that and apply that in the canary environment. After the canary release has proven to be stable, you can move the canary release to the actual production environment.



About Traffic Manager

What is it?

- Traffic manager provides the ability to route traffic between Azure app services
- Within an app service, routes traffic between deployment slots

Why do you need it?

- It enables failover and load distribution capabilities
- It enables you to deploy to a slot and then slowly move traffic over to the other slot

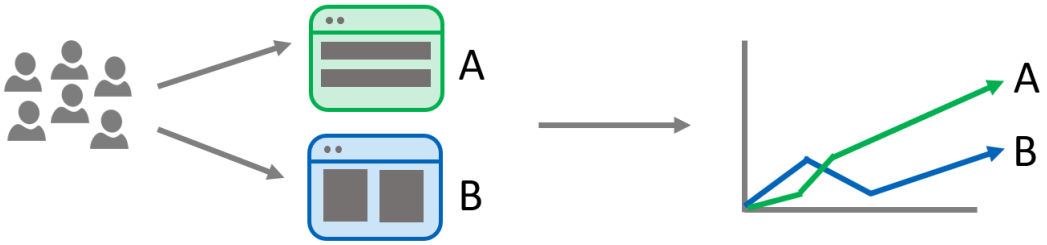
Traffic manager provides following options:

1. **Route traffic based on availability.**
2. Route traffic round robin
3. **Route traffic based on weight.**
4. Route traffic based on latency.

In the context of continuous delivery, we are most interested in option 1 and 3

A/B Testing

A/B testing is mostly an experiment where two or more variants of a page are shown to users at random, and statistical analysis is used to determine which variation performs better for a given conversion goal.



A/B testing is mostly an experiment where two or more variants of a page are shown to users at random, and statistical analysis is used to determine which variation performs better for a given conversion goal.

Why A/B Testing:

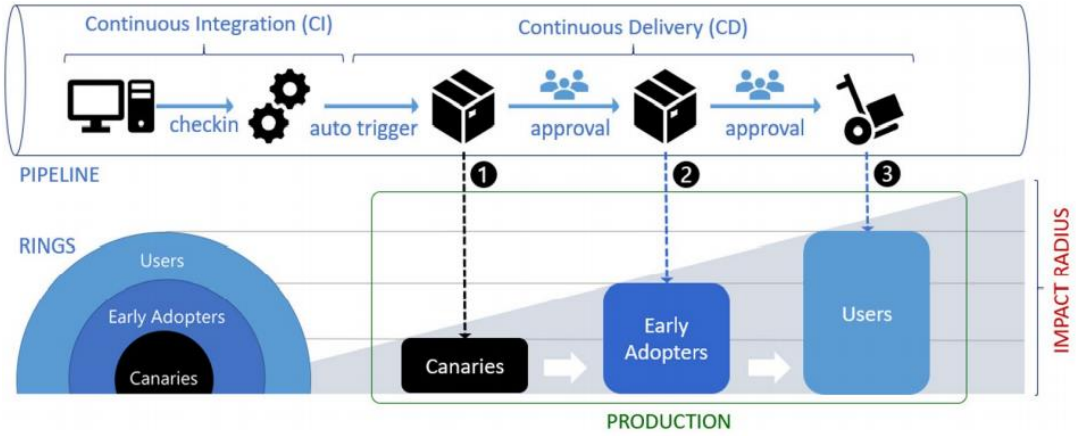
- Experiment on features and usage
- Improve conversion rate
- Continuous experiments
- Measure impact of change

Progressive Exposure Deployment

In a progressive exposure deployment, you expose the new software to a subset of users, that you extend gradually over time. Impact (also called blast radius), is evaluated through observation, testing, analysis of telemetry, and user feedback.

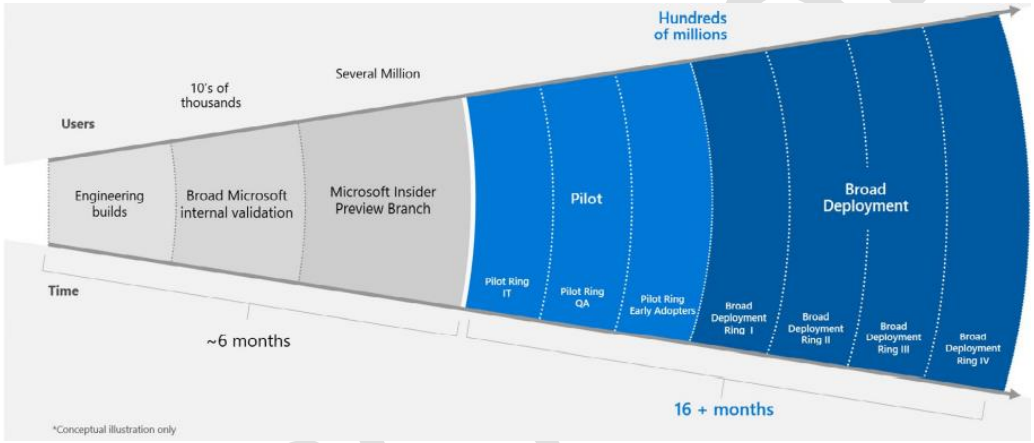
Progressive Exposure Deployment, or also called Ring based deployment. They support the production-first DevOps mindset and limit the impact on end users, while gradually deploying and validating changes in production. Impact (also called blast radius), is evaluated through observation, testing, analysis of telemetry, and user feedback.

Rings are in essence an extension of the canary stage. The canary release releases to a stage to measure impact. Adding another ring is essentially the same thing.



With a ring based deployment, you deploy your changes to risk-tolerant customers first, and the progressively roll out to a larger set of customers.

The Microsoft Windows team, for example, uses these rings.



Demo: Ring Based Deployment Pipeline:

Stage: Ring 0 (Canary)

Azure CLI Task: az webapp traffic-routing set --resource-group $(RGName) --name $(WebsiteName) --distribution staging=10

Stage: Ring 1 (Early Adaption)

Azure CLI Task: az webapp traffic-routing set --resource-group $(RGName) --name $(WebsiteName) --distribution staging=30

Stage: Public

Azure CLI Task: az webapp deployment slot **swap** -g $(RGName) -n $(WebsiteName) --slot staging --target-slot **production**

Azure CLI Task: az webapp traffic-routing set --resource-group $(RGName) --name $(WebsiteName) --distribution staging=0

Note:

1. **You should add Pre-Deployment Approval for Ring1 and Public stages.**
2. **Disable continuous deployment trigger.**

Dark Launching

The idea is that rather than launch a new feature for all users, you instead release it to a small set of users. Usually, these users are not aware they are being used as test users for the new feature and often you do not even highlight the new feature to them, hence the term "**Dark**" launching.

Dark Launching is in many ways similar to Canary Releases. However, the difference here is that you are looking to assess the response of users to **new features in your frontend**, rather than testing the performance of the backend.

Whereas dark launches are used to test features before making them widely available, canary releases focus on minimizing the risk and impact of production bugs.

Lab: Feature Flag Management with LaunchDarkly and AzureDevOps

https://azuredevopslabs.com/labs/vstsextend/launchdarkly/

Create Feature Flat in LaunchDarly

1. Create your account in https://app.launchdarkly.com/
2. Create a feature flag → Name = My Demo Flag → Save Flag
3. Account Settings → Projects Tab → Copy SDK Key under Production Row.

Integrate Launch Darkly in your Web Application

4. Add reference to Nuget Package: **LaunchDarkly.Client**
5. Edit _Layout.cshtml

```
@{
  LaunchDarkly.Client.LdClient client = new LaunchDarkly.Client.LdClient("sdk-fa127c1b-cfdf-4e3d-99a1-acda8fcdf65a");
  LaunchDarkly.Client.User user = LaunchDarkly.Client.User.WithKey("sandeepsoni@deccansoft.com");
  bool value = client.BoolVariation("my-demo-app", user, false);
  if (value)
  {
    <li class="nav-item">
      <a class="nav-link text-dark" asp-area="" asp-controller="Home" asp-action="Privacy">Privacy</a>
    </li>
  }
}
```

6. Run and Test the application locally.

Rollout LaunchDarly Feature flag during release

7. LaunchDarkly website → Account Settings → Authentication Tab → +Token → Name=AzureDevOps & **Role=Writer** → Save Token
8. Azure DevOps → Project → Service Connection → LaunchDarkly → Access Token = <Previous Step> → Save
9. Azure Boards → Work Items → Add User Story = Implement FeatureFlag Management using LaunchDarkly
    a. Go to LaunchDarkly Tab
    b. Select Default Project = Production
    c. Select feature flag = My Demo Flag

    Note that the Flag is set to "**0% rollout**"
10. Release Pipeline: Add the following Variables
    a. launchdarkly-account-name = <DevOps Organization Name>
    b. launchdarkly-project-name = <DevOps Project Name>
    c. launchdarkly-pat= txqjfi74a4atszjmfplsg5ijhgbgu5lbtshnhxru2exuhhx3dpoq

11. Release Pipeline: Add Task → LaunchDarkly Rollout and link to LaunchDarkly Service Connection.
12. Enable Continuous Delivery for the pipeline.
13. Run the pipeline and note that in WorkItem the Flag is now set to 100%

Deccansoft Software Services H.No: 153, A/4, Balamrai, Rasoolpura, Secunderabad-500003 TELANGANA, INDIA. http://www.deccansoft.com | http://www.bestdotnettraining.com
Phone: +91 40 2784 1517 OR +91 8008327000 (INDIA)