**Summary**

In this module, you defined a *deployment pattern* as an automated way to smoothly roll out new application features to your users. A good deployment pattern can help you minimize downtime. It can also enable you to roll out new features progressively to your users.

You can choose from several deployment patterns. The deployment pattern that you choose depends on your reasons for the deployment as well as your resources. Do you have canary testers in place? Will you employ a dark launch and choose testers who don't know that they are testers? If you have a trusted set of testers that progressively increases from a small set to a larger set, then could choose a progressive-exposure deployment. Or if you want to know if one version performs better than another version, you could choose A/B testing.

The Tailspin team chose to implement the blue-green deployment pattern. They took advantage of deployment slots in Azure App Service. *Deployment slots* are live apps that have their own host names. The team can swap two deployment slots. By swapping, they can promote changes to production instantly. Although the team isn't yet ready to release their website to the public, they've proven that they can get new features to their users without incurring downtime.

As a bonus, in this module you also learned how to roll forward an unintended change by reverting a Git commit and then pushing the reverted change through the pipeline.

**How is the team measuring up?**

In the *Assess your existing software development process* module, Mara did a [value-stream mapping exercise](https://docs.microsoft.com/en-us/learn/modules/assess-your-development-process/4-assess-process-efficiency). The exercise helped the team analyze their current release-cycle process.

Recall that the *activity ratio*, or efficiency, is *process time* divided by *total lead time*:

Activity ratio = Process timeTotal lead timeActivity ratio = Process timeTotal lead time

The Tailspin web team initially used this metric to determine that they were 23 percent efficient.

The team first reduced some inefficiencies when they implemented continuous integration (CI). By applying continuous delivery (CD), they have now reduced inefficiencies even further.

In previous learning paths, the team reduced:

* The time it takes to set up source control for new features. The required time went from *three days* to *zero days*.

The team achieved this improvement by moving from centralized source control to Git, a form of distributed source control. By using distributed source control, they don't need to wait for files to be unlocked.

* The time it takes to deliver code to Amita, the tester. The required time went from *two days* to *zero days*.

The team achieved this improvement by moving their build process to Azure Pipelines. Azure Pipelines automatically notifies Amita when a build is available. Developers no longer need to update Amita's spreadsheet to notify her.

* The time it takes Amita to test new features. The required time went from *three days* to *one day*.

The team achieved this improvement by unit-testing their code. They run unit tests each time a change moves through the build pipeline, so fewer bugs and regressions reach Amita. The reduced workload means that Amita can complete each manual test faster.

The release pipeline that you and the team built in this learning path reduced:

* The time it takes to get the build into the *Test* stage. The required time went from *three days* to *one day*.

The team achieved this improvement by using a scheduled trigger to deploy to *Test* every day at 3:00 AM.

* The time it takes to get the tested build into *Staging*. The required time went from *two days* to *zero days*.

The team achieved this improvement by adding Selenium UI tests, a form of functional testing, to the *Test* stage. These automated tests are much faster than the manual versions.

* The time it takes to get the approved build from *Staging* to live. The required time went from *one day* to *less than one day*.

The team achieved this improvement by adding manual approval checks to the pipeline. When management signs off, Tim can release the changes from *Staging* to live.

These changes reduce the total lead time from 22 days to 10 days. We substitute these numbers into the equation:

Activity ratio = 5 days10 days=0.50Activity ratio = 5 days10 days=0.50

Multiply the result by 100 percent, and we get a *50 percent* reduction.

Although there's always room for improvement, this change is a win for the team. Not only do customers get value more quickly, the Tailspin team now spends less time waiting and more time doing what they enjoy most: delivering features that they know their customers will love.

**Learn more**

Use these additional resources to learn more about App Service, deployment slots, and rolling back changes:

* [App Service documentation](https://docs.microsoft.com/en-us/azure/app-service)
* [Deploy a website to Azure by using App Service](https://docs.microsoft.com/en-us/learn/paths/deploy-a-website-with-azure-app-service)
* [Stage a web app deployment for testing and rollback by using App Service deployment slots](https://docs.microsoft.com/en-us/learn/modules/stage-deploy-app-service-deployment-slots)
* [Set up staging environments in App Service](https://docs.microsoft.com/en-us/azure/app-service/deploy-staging-slots)

**Appendix: Recommendations for using feature flags**

Feature flags were one of the release-cadence methods that the team considered. The team decided not to use feature flags, but many people have found them useful. This section provides more information about feature flags.

*Feature flags*, sometime called *feature toggles*, allow you to change how a system works without changing the code. These flags allow you to push new code into your central development branch and have the code deployed but not necessarily functional. The flags are commonly implemented as the value of variables that control conditional logic.

Imagine that your team is working in the central development branch of a bank application. You decided to do all the work in the main branch to avoid messy merge operations later. But you face a problem. You're substantially changing the interest calculations, and people depend on that code every day. Worse, the changes will take you weeks to complete. You can't leave the main code broken for so long.

In this scenario, a feature flag might be a good solution. You can change the code so that users who don't have the feature flag set can keep using the original interest calculation code. Meanwhile, your team does have the feature flag set, so they can see the code that they're changing.

Another type of feature flag is a release flag. Imagine that after you complete the work on the interest calculation code, you want to try it out before you release it publicly. You have a group of users who are well positioned to deal with new code and any possible issues. You'll let them try the feature first. You change the configuration so that they also have the feature flag set and can test the new code. If problems happen, then you can quickly disable the flag.