## Module 1: Introduction to DevOps

Azure Repos – provides two types of version control: Git and Team Foundation Version Control

IaC (Infrastructure as Code) - a practice that enables the automation and validation of the creation and teardown of environments to help with delivering secure and stable application hosting platforms.

## Module 2: Plan application lifecycle management (ALM) for Power Platform

Environment in Microsoft Power Platform:

**Sandbox** - A sandbox environment is any nonproduction environment of Dataverse. Isolated from production, a sandbox environment is the place to safely develop and test application changes with low risk.

**Production** - The environment where apps and other software are put into operation for their intended use.

**Community (developer)** - The Power Apps Community Plan gives a user access to Power Apps premium functionality, Dataverse, and Microsoft Power Automate for individual use only. This environment is primarily meant for learning purposes. A developer environment is a single-user environment and can't be used to run or share apps. A Community Plan environment can participate in the Azure DevOps pipeline.

**Default** - A single default environment is automatically created for each tenant and shared by all users in that tenant. The default environment is used by Microsoft 365 services.

**Trial** - Trial environments are to try new features or perform proof of concepts. Trial environments are automatically deleted after 30 days.

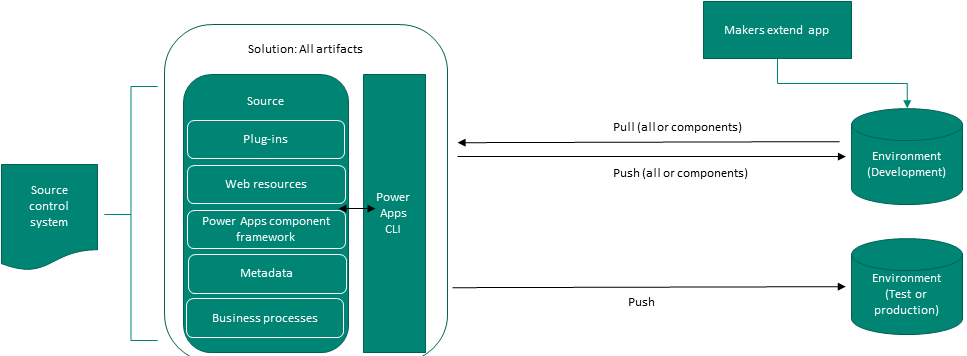
Managing ALM – can be two apporach:

**Environment-centric approach**:

* The dev environment is the master copy of all changes.
* Changes are promoted directly from dev > test > production.

**Source control-centric** approach:

* Source control is the master.
* The dev environment is re-created from source control (process can be automated and repeatable).
* Changes from the dev environment are checked into source control.



Tips: Using a source control-centric approach enables an Azure DevOps approach with build and release pipelines. Using an environment-centric approach means that you need to define the workflow for app makers and developers.

Issues with customizing and developing components within Microsoft Power Platform include:

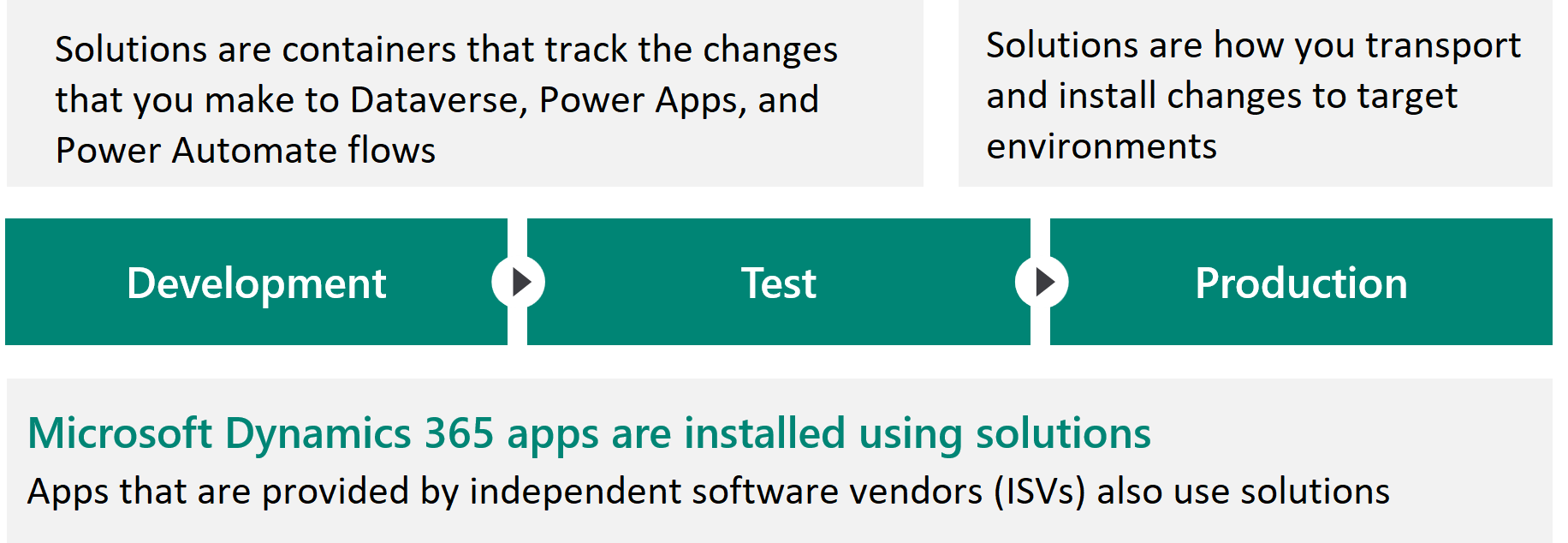
* Microsoft Power Platform doesn't support versioning of components (except for canvas apps).
* Users can't work on the same Microsoft Power Platform component simultaneously.
* Model-driven apps have multiple components, each with their own editors, allowing work to be divided between makers. Conversely, canvas apps have only one editor and only one person can work on an app at any one time. By using canvas components, you can allow multiple makers to work on the same app simultaneously.

Solutions - are **written**, **packaged**, and **maintained** by a **publisher** allowing customizations to components that extend your Microsoft Power Platform organization.

Solutions allow distribution of business functionality across Dyanmics 365 Customer Engagement environments.

A solution is either managed or unmanaged. When you first create a solution in an environment, it is considered unmanaged. Unmanaged solution has a collection of references to components (no components are contained in the unmanaged solution, only reference). Only when solution is exported, the application behavior of each of its components is created and saved to XML document. Solution should only be unamnaged under active development.

A managed solution is a “sealed binary”. Components can be added or removed. It cannot be exported. Mnaaged solution is recommended when solution is not actively being customized.



The characteristics of solutions are that they:

* Include metadata and certain entities with configuration data. Solutions don't contain business data.
* Contain many different Microsoft Power Platform components, such as model-driven apps, canvas apps, site maps, flows, tables, table metadata, columns, forms, views, business rules, process definitions, custom connectors, web resources, choices, charts, and components that are created by developers such as scripts or compiled code.
* Are packaged as a unit to be exported and imported to other environments, or they're deconstructed and checked into source control as source code for assets.
* Are used to apply changes to existing solutions.

Solution Structure

Strategies for creating solutions, listed in order from simplest to most complex, are:

* Single solution
* Multiple solutions
* Multiple solutions with shared components

By creating a single solution, you'll establish a working set of customizations. Recommended when you only want to create a single managed solution. If you think that you might have to split up the solution in the future, consider using multiple solutions.

You might have multiple solutions that share components.

Some components can be included in more than one solution, if any changes that were made to them are compatible with all other solutions that use them. It's important that all solutions share the same solution publisher. If the solution publisher isn't identical, you won't be able to install more than one of your solutions.

Rules that you should follow with solutions:

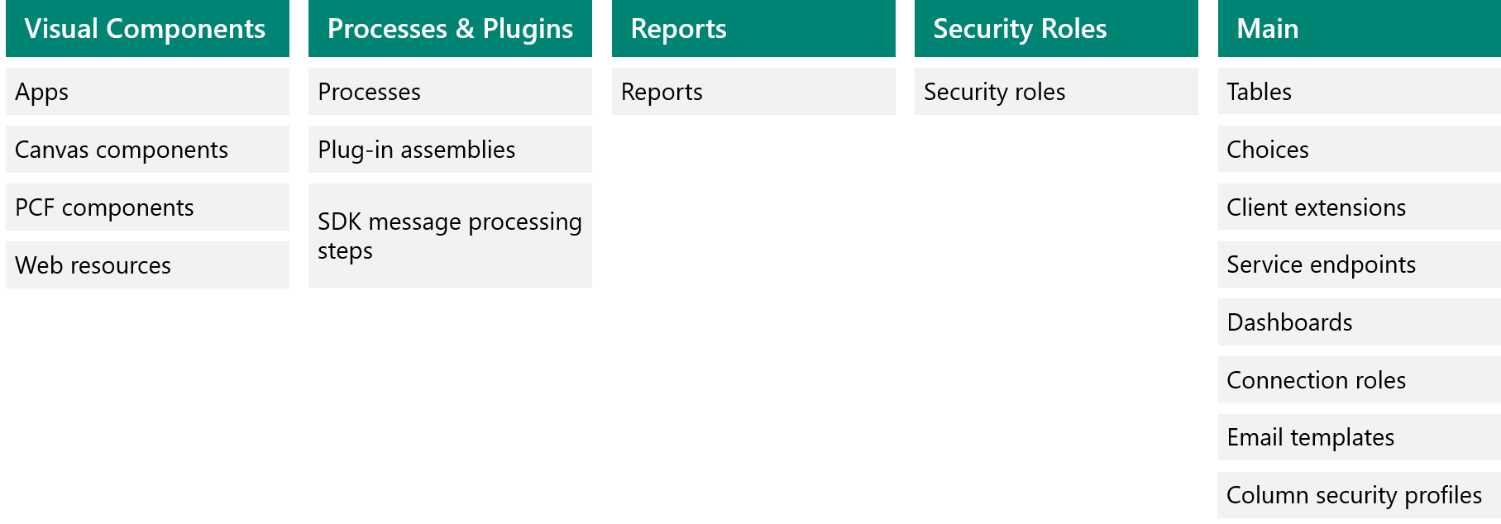
* Create a solution publisher and use it for all solutions.
* Don’t use the default publisher, the default solution, or the Dataverse default solution.
* Keep the solution structure as simple as possible.
* Avoid selecting the Include all components check box unless you're adding an unmanaged table.
* Include table metadata only when you're changing table properties.
* Add the subcomponents of a table (columns, forms, views, and so on) only when you're changing them.

Tips: Adding only what is required to a solution is known as ***segmenting*** a solution.

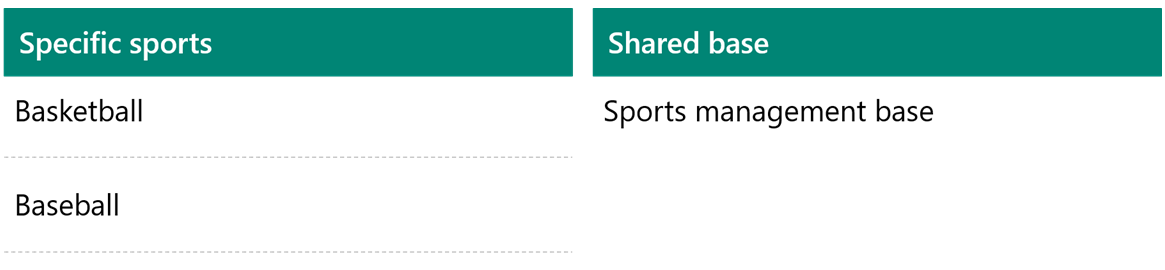
With segmentation, you can export solutions with selected table assets, such as columns, forms, and views. Segmentation also enables granular development. Avoid using **“Add All Assets”** option.

Solution Splitting - common patterns for multiple solution splittings are horizontal and vertical partitioning.

Horizontal Splitting - refers to creating solutions that only contain components of the same type.



Vertical Layering - group components into functional areas. Often, you'll have a shared base/common solution with separate solutions for each key business area.



Tips: You can combine vertical and horizontal partitioning, for instance, the base that contains all tables and processes with separate solutions for each app.

Configuration and Refernce Data

As you transport solutions through environments, those components might have configuration settings or will refer to data that also needs to be transported through the release process.

Environment Variables - app configuration data that is environment-specific. Applications often require different configuration settings or input parameters when they're deployed to different environments.

Because environment variables *are* solution components, you can transport the references (keys) and change the values when you migrate the solutions to other environments.

Each environment variable can have a default value and a current environment value.

Power Apps, Power Automate, and developer code can retrieve and modify the values of environment variables.

Connection References - use connection references for connections that are environment-specific. A connection reference contains information about a ***connector***. Canvas apps and operations within a Power Automate flow bind to a connection reference.

Because connection references are solution components, you can transport the references and change the connection when solutions are migrated to other environments. Connection references enable you to change a connection that is associated with a canvas app or flow without editing the app or flow.

Configuration Migration Tool - Solutions don't contain data. Often, your application relies on reference or configuration data. This data also needs transportation from one environment to another.

The Configuration Migration tool can help move data between environments. Importantly, the Configuration Migration tool can maintain the same primary record identifier (GUID) for the rows in this data. This feature prevents issues with components that refer to specific data from having to be updated each time that it's imported.

Release Process

Solutions applied to test and production environments will be ***managed*** solutions. Solutions have version numbers. The version numbers are automatically incremented when you export a solution. You can't import a solution with a lower version number if the solution has already been imported with a higher version number.

When importing a new version of a managed solution, you should consider the following import options:

* Update - Applies the changes to managed solution.
* Upgrade (Default Option) - Imports changes and applies them immediately, including removing any components that are not part of the new solution. The old solution is overwritten.
* Stage for Upgrade - Similar to upgrade, but it pauses after a new solution is imported before you have removed components so that you can complete data migration. Then, you can manually trigger the final application of the solution.

Updates to a managed solution are deployed to the previous version of the managed solution. This action doesn't create an additional solution layer. You can't delete components by using an update. Must not use “overwrite customizations option" to leverage optimizations. Update is only applied to changes made, therefore, it allow faster import time.

Upgrading a solution installs a new solution layer immediately above the base layer. Solution upgrades will delete components that existed but are no longer included in the upgraded version. If you need to make changes before upgrade, use “stage solution for upgrade”.

Tips: Upgrade is the default option.

Patch Solution - A patch solution contains only the changes for a parent-managed solution, such as adding or editing components and assets.

More items that you should keep in mind concerning patches include:

* Use patches when making small updates (similar to a hotfix).
* When patches are imported, they're layered on top of the parent solution. You can't delete components with a patch.
* Using patches isn't recommended

Azure DevOps - contains many features to help the development of applications:

* Azure Boards - Plan, track, and discuss work across your teams.
* Azure Pipelines - Use to automate continuous integration and continuous deployment (CI/CD) builds and releases.
* Azure Repos - Source control to store and track changes.
* Azure Test Plans - Plan, implement, and track scripted tests.
* Azure Artifacts - Publish solutions that are built by build pipelines.

Pipelines - Power Apps builds tools to automate the common build and deployment tasks

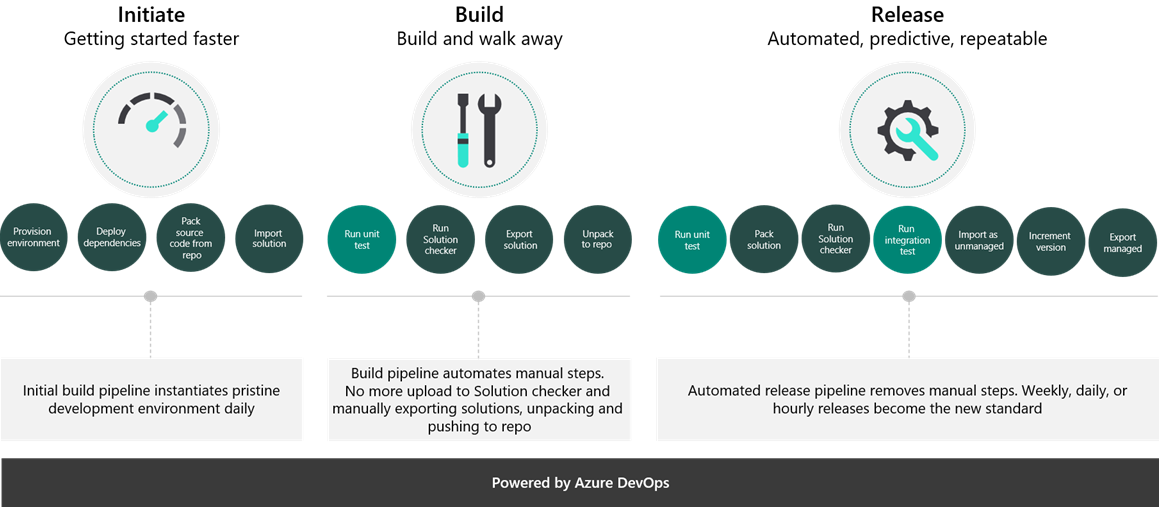
Build pipelines can be used to:

* Create dev environments.
* Commit changes from dev to source control.
* Enable the Solution checker tool.
* Perform automated testing.
* Build output solutions from source control (for example, managed or unmanaged).

Release pipelines can be used to:

* Take solutions from build pipelines and deploy them to one or more test or production environments.
* Perform automated testing as part of the release process.
* Pause for approvals before progressing to the next environment.

Pipelines that teams will commonly establish include Initiate, Export from Dev, Build, and Release.



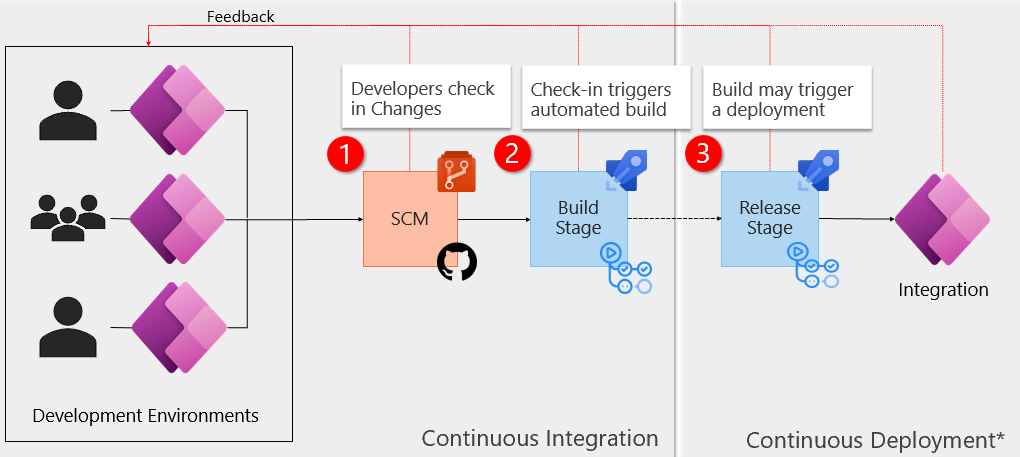
Alternatives for automating deployments **without using Azure DevOps** are:

* Dataverse and admin APIs can be used to automate from any supported language.
* PowerShell can be used instead of build tasks for more control.
* Power Automate can be used with the platform admin connectors to automate deployments.
* GitHub actions are currently in preview.

Introduction to team development for Microsoft Power Platform

Continous Integration - the process of automating the build and testing of code whenever a team member commits changes to version control. It encourages developers to share their code and unit tests by merging their changes into a shared version control repository after every small task completion.

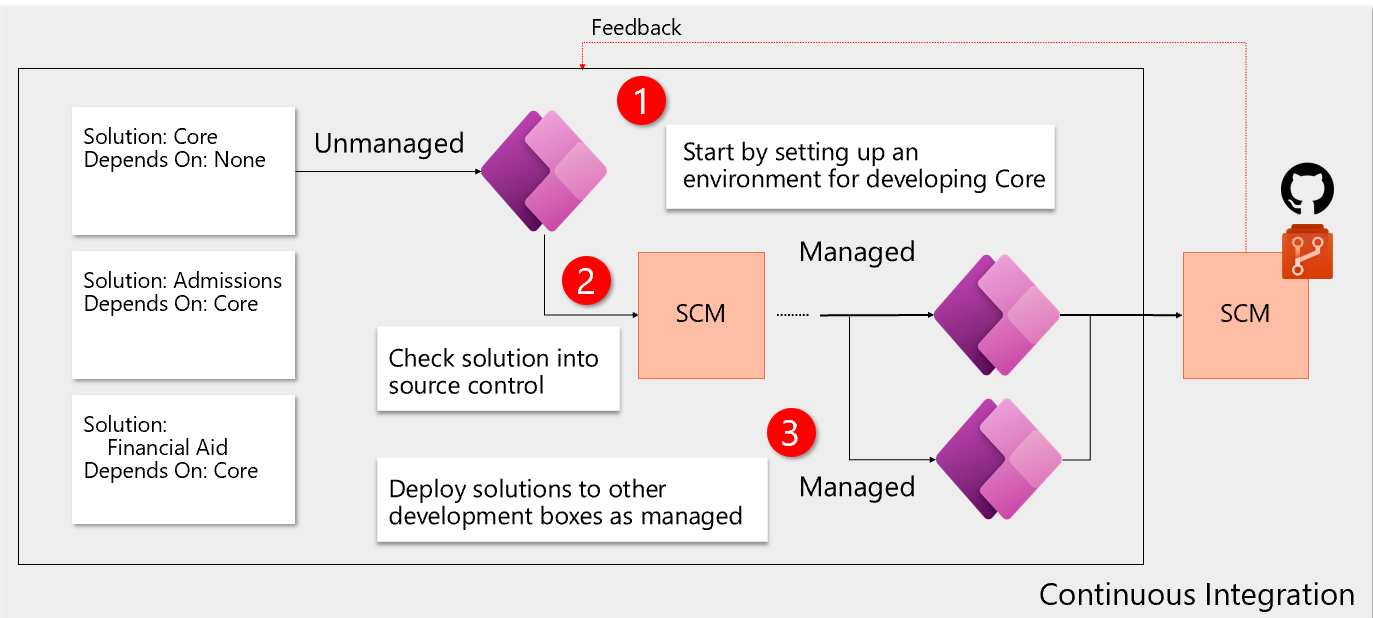
A better pattern for team development will involve the use of one or more development environments enabling multiple developers to efficiently collaborate on the creation of new content.



In this more optimized development loop:

1. Developers make changes in their respective environments.
2. Changes in those environments are exported, then checked into source control.
3. The action of checking a solution into source control may trigger one of several events, such as a build process to move the solution to downstream environments.

Development process setup - Assume that the top environment will be dedicated to developing the Core solution, which contains shared components.



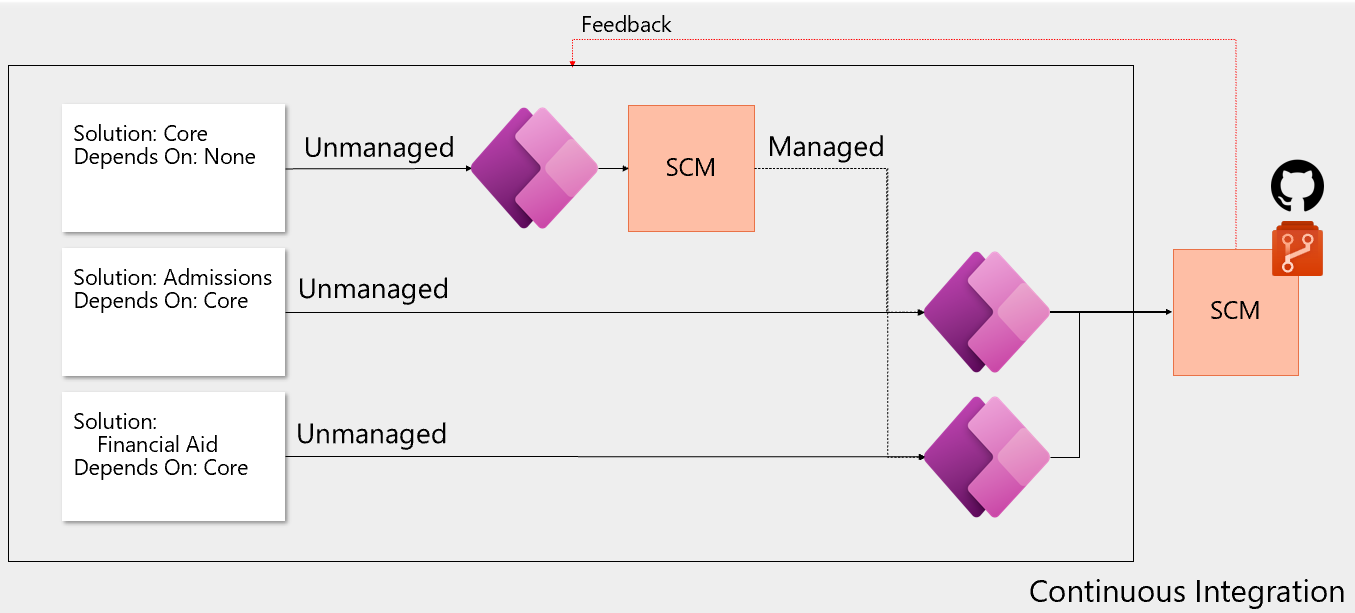
To get started, you'll create the Core solution in the target development environment. When creating solutions, you’ll create them as unmanaged, and you can only change them to managed during the export process.

Assume that you've finished your development sprint on the Core solution and have checked your changes into your repository. Then, you’ll deploy the Core solution as managed in the Admissions and Financial Aid development environments.

By introducing **only one unmanaged solution for each environment**, you’ll eliminate the potential for injecting unwanted dependencies.

Additionally, this approach allows you to effectively use segmentation and solution layering.

With the dependencies in place, we can now introduce our other solutions.



You'll create or import your Admissions and Financial Aid solutions in their respective environments.

Note: An environment might have more than one unmanaged solution installed if it can be guaranteed that the other solutions won't introduce dependencies.

Introduction to GitHub Actions

Github actions - primary mechanism for automation (e.g., Continous Integration) within GitHub.

|  |  |
| --- | --- |
| Actions Structure. Events, trigger, workflows, contain, jobs, use, actions. | * GitHub tracks events that occur. Events can trigger the start of workflows. * Workflows can also start on cron-based schedules and can be triggered by events outside of GitHub. * They can be manually triggered. * Workflows are the unit of automation. Workflows contain Jobs. * Jobs use Actions to get work done. |

Actions are defined in YAML and stay within GitHub repositories. Actions are executed on "runners," either hosted by GitHub or self-hosted.

Workflows are written in YAML and live within a GitHub repository at the place .github/workflows.

You can find a set of starter workflows here: [Starter Workflows](https://github.com/actions/starter-workflows).

You can see the allowable syntax for workflows here: [Workflow syntax for GitHub Actions](https://docs.github.com/actions/learn-github-actions/workflow-syntax-for-github-actions).

Standard Workflow Syntax

* **Name:** is the name of the workflow. It's optional but is highly recommended. It appears in several places within the GitHub UI.
* **On:** is the event or list of events that will trigger the workflow.
* **Jobs:** is the list of jobs to be executed. Workflows can contain one or more jobs.
* **Runs-on:** tells Actions which runner to use.
* **Steps:** It's the list of steps for the job. Steps within a job execute on the same runner.
* **Uses:** tells Actions, which predefined action needs to be retrieved. For example, you might have an action that installs node.js.
* **Run:** tells the job to execute a command on the runner. For example, you might execute an NPM command.

Allowable syntax for workflows here: [Workflow syntax for GitHub Actions](https://docs.github.com/actions/learn-github-actions/workflow-syntax-for-github-actions)

For more information on events, see [Events that trigger workflows](https://docs.github.com/actions/learn-github-actions/events-that-trigger-workflows).

Jobs - Workflows contain one or more jobs. A job is a set of steps that will be run in order on a runner. Steps within a job execute on the same runner and share the same filesystem. By default, if a workflow contains multiple jobs, they run in parallel. Sometimes you might need one job to wait for another job to complete. You can do that by defining dependencies between the jobs.

For more information on job dependencies, see the section **Creating Dependent Jobs** at [Managing complex workflows](https://docs.github.com/actions/learn-github-actions/managing-complex-workflows).

Github runners - are compute resources that execute GitHub Actions workflows. Each runner can run a single job at a time. They allow developers to perform build, test, and deployment tasks directly within from GitHub repositories. There are two main types of GitHub runners:

* GitHub-hosted runners are virtualized or containerized compute resources provided and managed by GitHub.
* Self-hosted runners are physical, virtualized, or containerized compute resources that GitHub users and organizations provision and manage themselves.

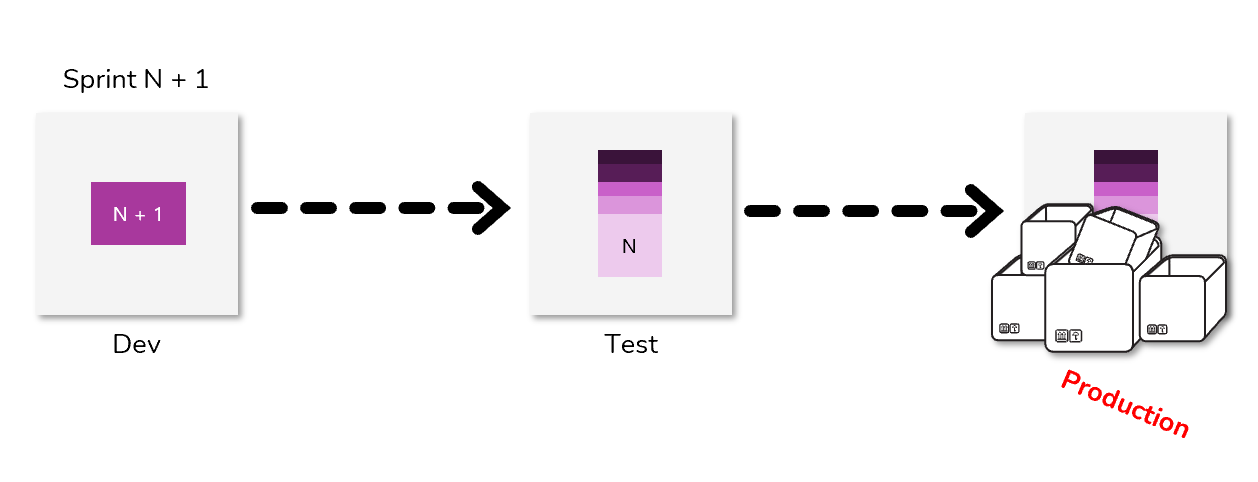
GitHub offers several learning tools for actions. Refer to [GitHub Actions: hello-world](https://github.com/skills/hello-github-actions).

Introduction to continuous deployment for Microsoft Power Platform

The top benefits of continuous deployment are that it:

* Encourages configuration as code.
* Minimizes the time to deploy.
* Enables automated testing throughout the pipeline.
* Makes deploying to production a low-stress activity.
* Provides visibility and feedback cycles.
* Reduces time to mitigate incidents (TTM).
* Reduces time to remediate incidents (TTR).
* Provides a faster release cadence so that hotfixes can become part of the normal release cycle.

You learned about the challenges with common patterns when you’re deploying a new unmanaged solution with each sprint. Eventually, production will begin to fail because of hundreds of solutions in the pipeline.



A modern approach to solution development will include layering and support deployments that are deterministic. The phrase deployments are deterministic means that, after an import order has been established, the impact on runtime behavior should be more clearly understood.

