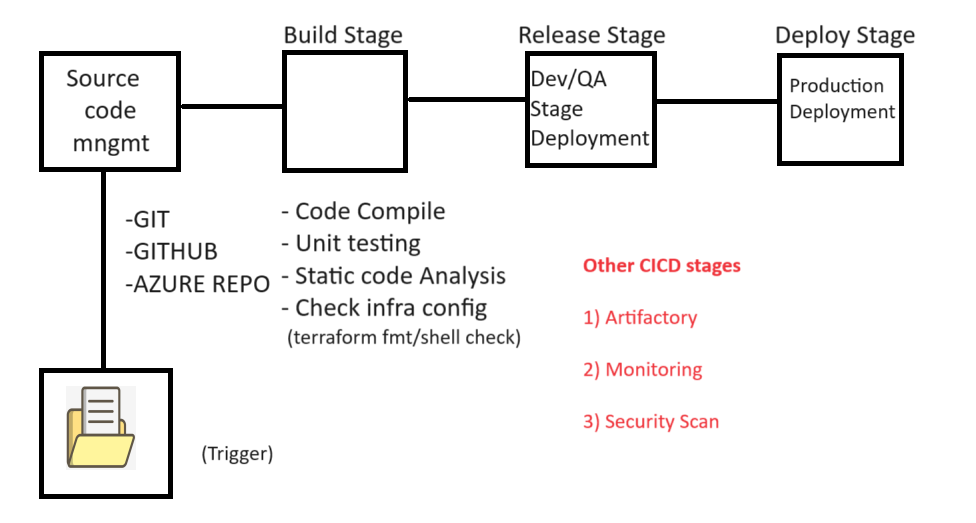
**Introduction to CICD pipeline:**

A multi-stage CI/CD pipeline automates software development processes: code commit triggers build, followed by automated testing, deployment to staging, and finally production, ensuring continuous integration and delivery with rapid feedback loops.



**Azure DevOps Architecture:**

Azure DevOps is a set of development tools and services provided by Microsoft to facilitate DevOps practices. It covers the entire development lifecycle, including planning, development, testing, deployment, and monitoring. The architecture of Azure DevOps involves various components that work together to support the DevOps workflow. As of my last knowledge update in January 2022, here is an overview of the key components:

**Azure DevOps Services:**

* **Azure Boards:** Handles work item tracking, including user stories, tasks, bugs, and more.
* **Azure Repos:** A version control system that supports Git repositories for source control.
* **Azure Pipelines:** Provides Continuous Integration (CI) and Continuous Deployment (CD) services. It allows you to build, test, and deploy applications.
* **Azure Test Plans:** Helps in planning, tracking, and managing testing efforts.

**Azure DevOps Server (formerly Team Foundation Server - TFS):**

For organizations that prefer to host their own DevOps infrastructure, Azure DevOps Server is available. It includes the same components as Azure DevOps Services but is installed and managed on the organization's servers.

**Azure DevOps Extensions:**

These are add-ons and integrations that extend the functionality of Azure DevOps. Extensions can be developed using various technologies, including Azure DevOps Services REST APIs.

**Azure DevOps Agent:**

Agents are responsible for running jobs in Azure Pipelines. They can run on Windows, Linux, or macOS and can be hosted on Microsoft-hosted infrastructure or your own infrastructure.

**Azure DevOps API:**

Azure DevOps provides REST APIs that allow you to programmatically interact with and extend the functionality of Azure DevOps Services.

**Azure DevOps Data Tier:**

The data tier is where the data is stored. This includes information related to work items, source code, builds, releases, and more. This data is stored in a SQL Server-based data store.

**Identity and Access Management:**

Azure DevOps leverages Azure Active Directory (Azure AD) for identity and access management. Users and groups are managed through Azure AD, providing a single sign-on experience.

**Integration with Azure Services:**

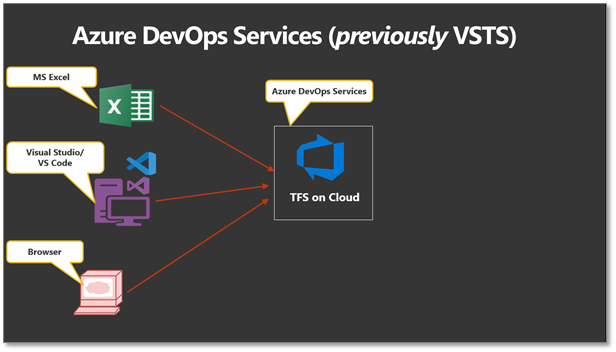
Azure DevOps integrates seamlessly with other Azure services, such as Azure Repos integrating with Azure Boards, and Azure Pipelines supporting deployment to Azure services like Azure App Service, Azure Kubernetes Service (AKS), etc.

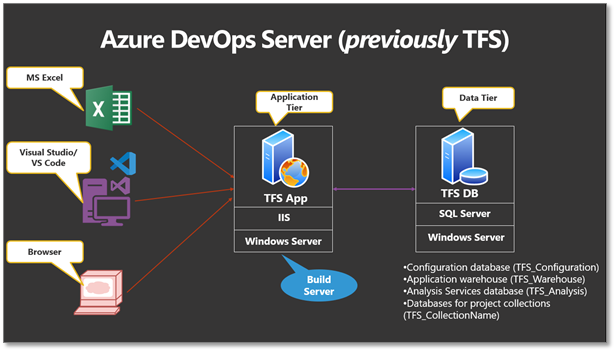
**Build and Release Agents:**

These agents execute the build and release tasks defined in Azure Pipelines. They can run on various platforms and can be configured based on the requirements of your build and release processes.

Azure DevOps services is a pipeline as code-based solution provided by Microsoft that enable Devops Capabilities.

**Azure DevOps services vs Azure DevOps server:** Azure DevOps service is a cloud based solution implemented by Microsoft, which has Microsoft agent and self-hosted agent whereas Azure DevOps server is a PAAS based implementation which support only self-hosted agent.





Azure DevOps Pipelines are a part of Microsoft's Azure DevOps Services that enable you to continuously build, test, and deploy to any platform and cloud. Pipelines in Azure DevOps are defined using YAML syntax. A pipeline can be divided into stages, jobs, and tasks to organize and manage the workflow effectively.

**Stages:** [stages> stage > job > steps]

Stages represent a logical boundary in your pipeline and allow you to group jobs together. Each stage can be considered as a phase in your pipeline. For example, you might have stages like "Build," "Test," and "Deploy." Stages are defined at the root level of your pipeline YAML file. Stages can be executed sequentially or in parallel, and they can have dependencies on each other.

Example of defining stages in a pipeline:

yaml

stages:

- stage: Build

jobs:

- job: BuildJob

pool:

vmImage: 'windows-latest'

steps:

- script: echo 'Building the application'

- stage: Test

jobs:

- job: TestJob

pool:

vmImage: 'windows-latest'

steps:

- script: echo 'Running tests'

- stage: Deploy

jobs:

- job: DeployJob

pool:

vmImage: 'windows-latest'

steps:

- script: echo 'Deploying the application'

**Jobs:**

Jobs are the building blocks of a stage. A stage can have one or more jobs that run in parallel on different agents or sequentially on the same agent. Jobs contain a set of steps, and each step represents a task or action. Jobs can be defined to run on specific agent pools, specifying the type of agent (e.g., Windows, Linux, macOS) and other configurations.

Example of defining jobs in a pipeline:

yaml

jobs:

- job: BuildJob

pool:

vmImage: 'windows-latest'

steps:

- script: echo 'Building the application'

- job: TestJob

pool:

vmImage: 'windows-latest'

steps:

- script: echo 'Running tests'

- job: DeployJob

pool:

vmImage: 'windows-latest'

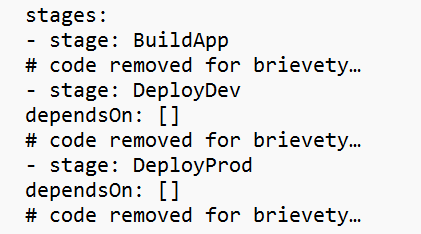
steps:

- script: echo 'Deploying the application'

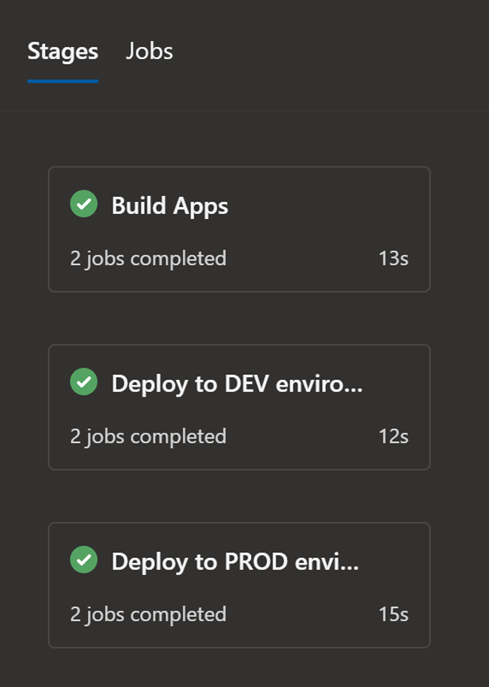
In the above examples, each stage contains one or more jobs, and each job has a set of steps to be executed. These stages and jobs help structure and organize your pipeline, making it easier to understand and manage the various tasks involved in the CI/CD process.

**Adding dependencies between stages and jobs**

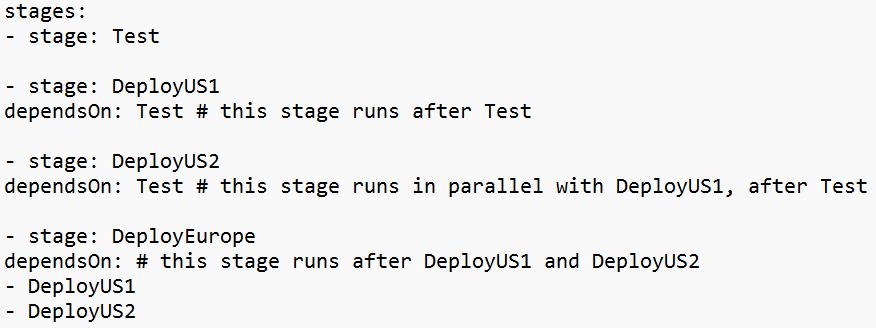
We said earlier that Stages runs in sequence unless specified using the syntax dependsOn:[]. Let’s try it in action in the previous pipeline:

****

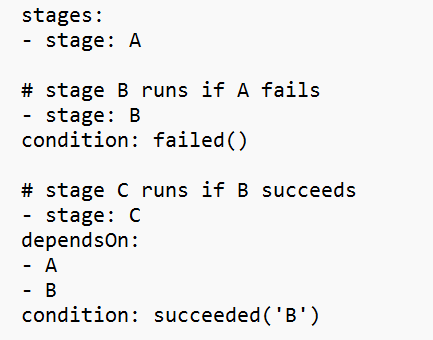
**Execution will look like-**



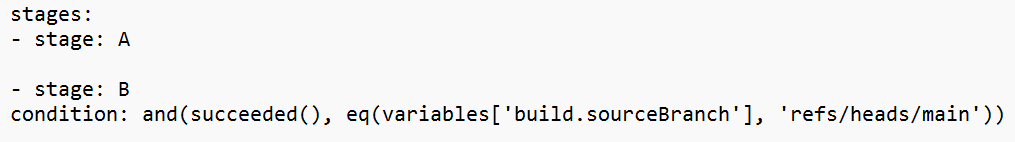
Let’s take another example where we have dependencies between Stages.

**** If we customize the default condition of the preceding steps for a stage, we remove the conditions for completion and success. So, if we use a custom condition, it's common to use and(succeeded(), custom\_condition) to check whether the preceding stage ran successfully. Otherwise, the stage runs regardless of the outcome of the preceding stage.

Example to run a stage based upon the status of running a previous stage:

****

Example of using a custom condition:

****