AZ-400: Designing and Implementing Microsoft DevOps Solutions Certification

Cheat Sheet

Quick Bytes for you before the exam!

AZ-400 Whizcard Index

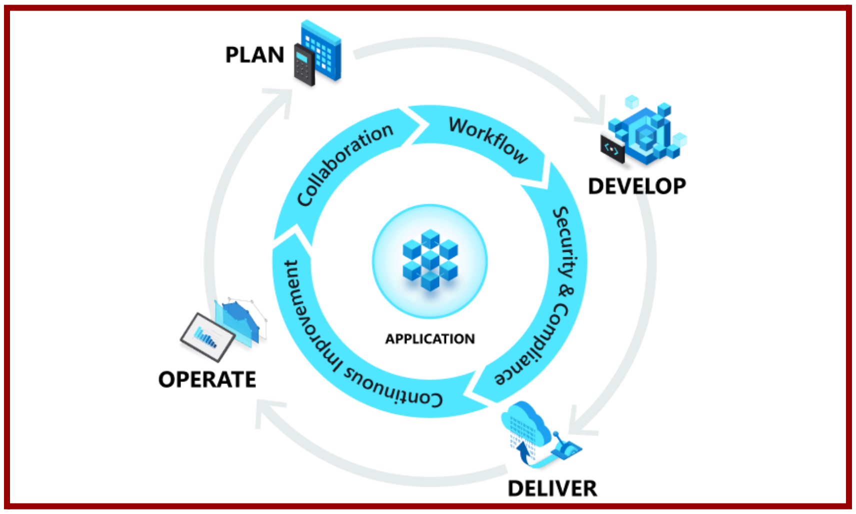
—-Back to Index—

Introduction to DevOps

What is DevOps?

Understanding DevOps

DevOps: A blend of development (Dev) and operations (Ops), integrating people, processes, and technology to continuously deliver value to customers.



[Source: Microsoft Documentation]

Impact on Teams

- Improved Collaboration
  - **Unified Roles**: Breaks down barriers between development, IT operations, quality engineering, and security for better teamwork.
- Higher Product Quality
  - **Reliable Products**: Collaborative efforts lead to more dependable and higher-quality outcomes.
- Adopting DevOps Culture
  - **Cultural Shift**: Embracing DevOps practices and tools enhances team efficiency.
- Better Customer Responsiveness
  - **Customer Focus**: Enables prompt responses to customer needs and feedback.
- Increased Application Confidence
  - **Trust in Development**: Streamlined processes and continuous integration build confidence in application performance.
- Faster Achievement of Business Goals
  - **Efficient Progress**: DevOps' collaborative nature accelerates the achievement of business objectives.

—-Back to Index—

What Azure DevOps?

Azure DevOps Overview

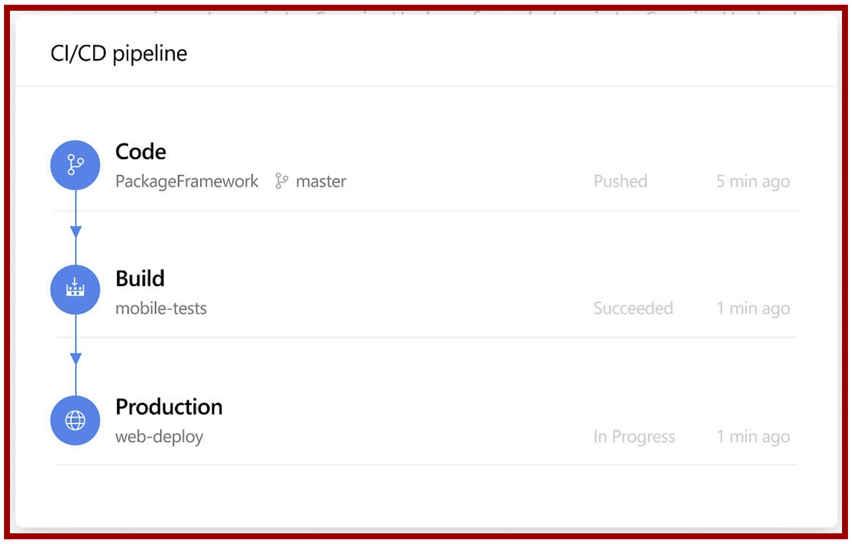
Enhance your planning, collaboration, and delivery processes with a suite of modern development services.

Azure Boards

- **Agile Tools:** Accelerate value delivery with agile tools to plan, track, and discuss work across teams.

Azure Pipelines

- **CI/CD Integration:** Build, test, and deploy using continuous integration/continuous deployment (CI/CD) that supports any language, platform, and cloud. Seamlessly connect to GitHub or other Git providers for continuous deployment.



CI/CD pipeline

Code PackageFramework master	Pushed	5 min ago
Build mobile-tests	Succeeded	1 min ago
Production web-deploy	In Progress	1 min ago

[Source: Microsoft Documentation]

Azure Repos

- **Unlimited Git Repositories:** Access unlimited, cloud-hosted private Git repositories and enhance code collaboration with pull requests and advanced file management.

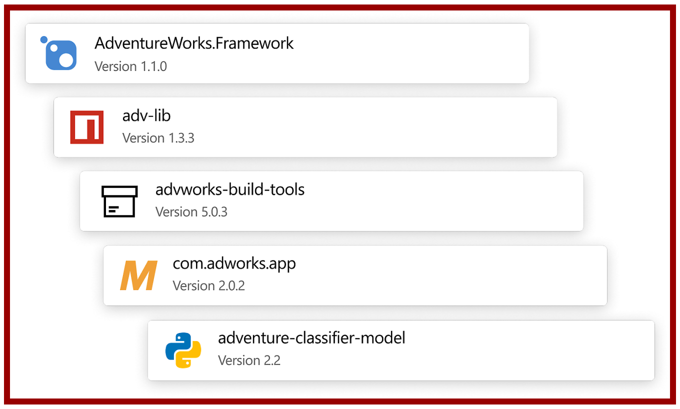
GitHub Advanced Security for Azure DevOps

- **Secure Development:** Ensure security throughout the development lifecycle from inception to deployment.

Azure Test Plans

- **Confident Testing:** Utilize manual and exploratory testing tools to test and ship applications with confidence.

—-Back to Index—



[Source: Microsoft Documentation]

Azure Artifacts

- **Package Management:** Create, host, and share packages with your team, and effortlessly add artifacts to your CI/CD pipelines with a single click.
- **Maven, npm, NuGet, and Python Packages:** Distribute packages from both public and private sources to your entire team.
- **Team Integration:** Seamlessly share packages within your team.
- **CI/CD Pipeline Integration:** Easily incorporate package sharing into your continuous integration/continuous deployment pipelines.
- **Simple and Scalable:** Ensure the package sharing process is straightforward and can scale with your needs.

How Microsoft plans with DevOps?

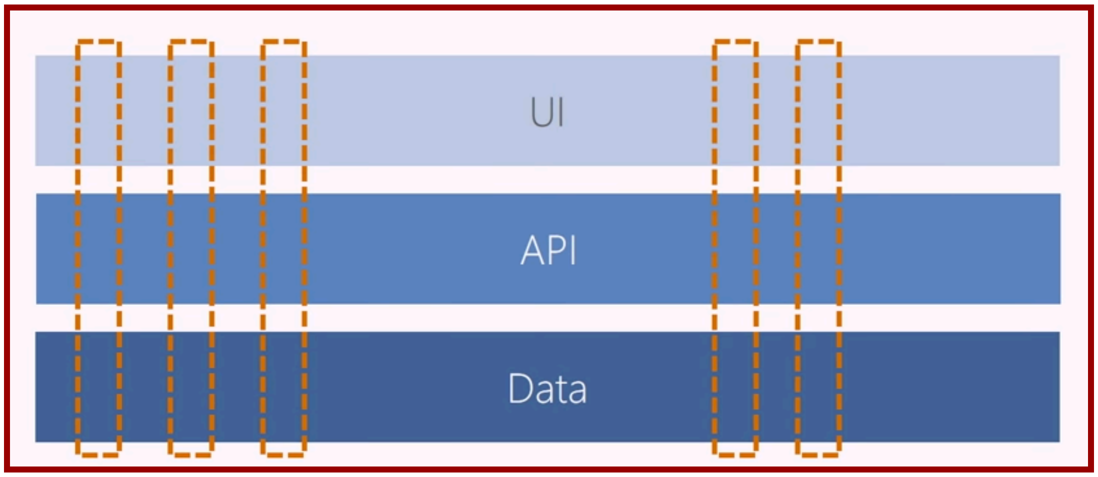
Essential Changes for Enhanced Development and Shipping Cycles

To boost the efficiency and health of development and shipping cycles, key changes should focus on fostering teamwork, improving planning and learning, and enhancing transparency and accountability.

Key Improvements:

➔ **Enhance Cultural Alignment and Autonomy**
  - ◆ **Unified Culture**: Align organizational goals and values across all teams to ensure everyone works towards a common objective.
  - ◆ **Team Autonomy**: Empower teams to make their own decisions, boosting morale and productivity by allowing quick adaptations without top-down directives.

➔ **Shift Focus from Individuals to Teams**
  - ◆ **Team Achievements**: Emphasize team achievements over individual performance to foster collaboration and shared responsibility.
  - ◆ **Collective Responsibility**: Encourage a supportive environment where team members help each other towards common goals.

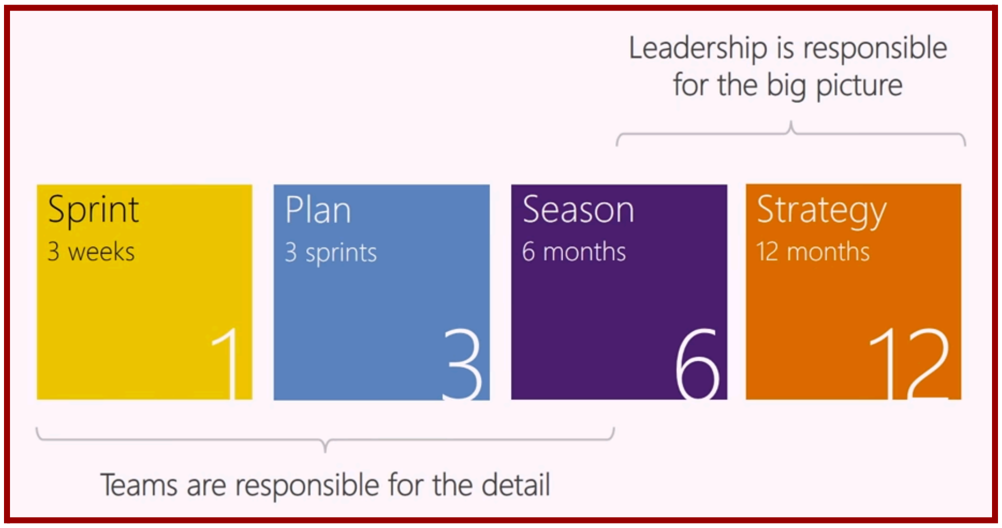


[Source: Microsoft Documentation]

➔ **Develop New Planning and Learning Strategies**

◆ **Flexible Planning**: Implement agile planning methods that allow for iterative development and frequent reassessment of goals.

◆ **Continuous Learning**: Promote a culture of continuous improvement through regular training, workshops, and certifications.


Leadership is responsible for the big picture
Sprint
3 weeks
1
Plan
3 sprints
3
Season
6 months
6
Strategy
12 months
12
Teams are responsible for the detail

[Source: Microsoft Documentation]

➔ **Adopt a Multi-Crew Model**

◆ **Collaborative Teams**: Use a multi-crew approach where different teams work together, bringing diverse perspectives and expertise.

◆ **Scalability**: Scale efficiently by adding teams as needed without disrupting existing workflows.

➔ **Improve Code Quality Practices**

- **Code Health**: Maintain and enhance code quality through regular reviews, automated testing, and refactoring.
- **Best Practices**: Standardize coding practices across teams for consistency and reliability.

➔ **Encourage Transparency and Accountability**

- **Clear Processes**: Ensure transparency by documenting workflows, providing regular updates, and maintaining open communication channels.
- **Accountability**: Establish clear goals and regularly review performance to hold teams accountable for their outcomes.

Azure Devops Services Overview

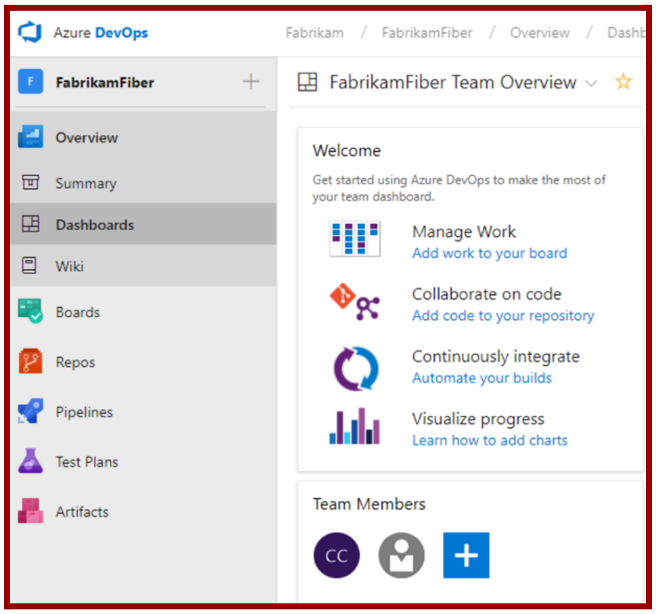
Azure DevOps Overview: Azure DevOps offers a comprehensive suite of services and tools to manage software projects, covering all stages from planning to deployment.

Service Delivery Model

- **Client/Server Model:** Azure DevOps delivers its services through a client/server architecture.
- **Web Interface:** Most services can be accessed via a web interface compatible with all major browsers.
- **Client Management:** Services such as source control, build pipelines, and work tracking can also be managed through a client.

Access and Pricing

- **Free and Subscription Options:** Many services are free for small teams, with additional options available through subscription or per-use models.
- **Hybrid Approach:**
  - **On-Premises and Cloud Integration**: Use an on-premises deployment to manage code and work.
  - **Cloud Services:** Purchase cloud build or testing services on an as-needed basied on greater flexibility.



[Source: Microsoft Documentation]

Azure DevOps Services Overview

Dashboards

- Customizable Dashboards: Access dashboards tailored to manage and monitor project progress.
- Task Management: Perform diverse tasks like adding, configuring, and managing dashboards, widgets, and navigating project areas efficiently.

Repositories (Repos)

- Version Control: Utilize Git or Team Foundation Version Control (TFVC) to manage code versions.
- Git Repos: Developers maintain local copies of source repositories, with changes shared between repositories after committing locally.
- TFVC: Developers store files locally with historical data on the server, and branches are server-based.

Boards

- Agile Planning Tools: Utilize Agile tools for efficient planning and work progress tracking.
- Software Development Support: Effectively manage tasks, issues, and code defects, aligning with Agile methodologies.

Pipelines

- Automated Build, Test, and Release: Automate build, test, and release processes for swift and reliable software deployment.
- Build Pipelines: Set up builds to run automatically upon code changes, including post-build testing.
- Release Pipelines: Control deployment of software builds to staging or production environments.

Test Plans

- Manual and Exploratory Testing: Create and oversee manual, exploratory, and continuous tests.
- Customized Workflows: Tailor workflows with test plan, suite, and case work items, ensuring comprehensive traceability from requirements to bugs.
- Execution and Monitoring: Execute tests on various platforms and monitor test activity using real-time charts

Configure activity traceability and flow of work

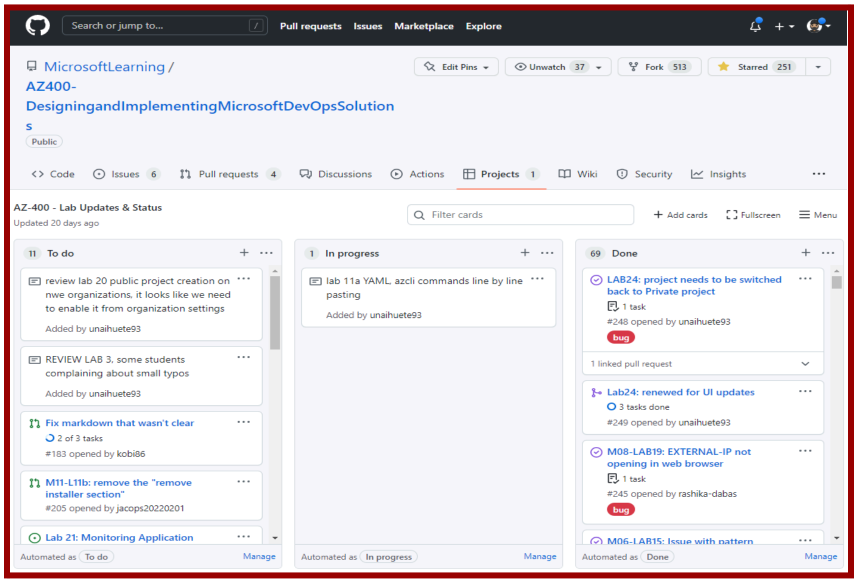
GitHub Projects and Project boards

Project:

- Projects are a newly introduced, adaptable tool on GitHub designed for planning and tracking work, offering enhanced customization and flexibility.
- You can create and personalize various views by filtering, sorting, and grouping your issues and pull requests. Visualize your work with customizable charts, and add custom fields to track team-specific metadata.
- Instead of adhering to a specific methodology, a project offers adaptable features that you can tailor to suit your team's unique needs and workflows.

Project Boards

During the application or project lifecycle, it's crucial to plan and priorities work. With Project boards, you can control specific feature work, roadmaps, release plans, etc.
Project boards are made up of issues, pull requests, and notes categorized as cards you can drag and drop into your chosen columns. The cards include pertinent metadata for issues and pull requests, such as labels, assignees, status, and the creator.



AZ-400 - Lab Updates & Status
Updated 20 days ago

11 To do

- review lab 20 public project creation on nwe organizations, it looks like we need to enable it from organization settings
  Added by unaihuete93
- REVIEW LAB 3, some students complaining about small typos
  Added by unaihuete93
- Fix markdown that wasn't clear
  2 of 3 tasks
  #183 opened by kobi86
- M11-L11b: remove the "remove installer section"
  #205 opened by jacops20220201
- Lab 21: Monitoring Application

Automated as To do — Manage

1 In progress

- lab 11a YAML, azcli commands line by line pasting
  Added by unaihuete93

Automated as In progress — Manage

69 Done

- LAB24: project needs to be switched back to Private project
  1 task
  #248 opened by unaihuete93
  bug
  1 linked pull request
- Lab24: renewed for UI updates
  3 tasks done
  #249 opened by unaihuete93
- M08-LAB19: EXTERNAL-IP not opening in web browser
  1 task
  #245 opened by rashika-dabas
  bug
- M06-LAB15: Issue with pattern

Automated as Done — Manage

[Source: Microsoft Documentation]

Types of project board:

- **User-owned project boards:** Can contain issues and pull requests from any personal repository.

- **Organization-wide project boards:** Can contain issues and pull requests from any repository that belongs to an organization.
- **Repository project boards:** Focus exclusively on issues and pull requests within a specific repository.

To create a project board for your organization, you must be an organization member. It's possible to use templates to set up a new project board that will include columns and cards with tips. The templates can be automated and already configured.

GitHub integration overview - Azure DevOps

Azure Boards and GitHub Integration

Integrating Azure Boards with GitHub repositories allows for seamless linking of commits, pull requests, and issues to work items. This setup facilitates the use of GitHub for development and Azure Boards for work planning and tracking. Here's a breakdown of the features and their descriptions:

- **Connect Azure Boards to GitHub Repositories**:
  - Enables the connection of one or more GitHub repositories to an Azure Boards project.
- **Connect Azure Boards to GitHub Enterprise Server**:
  - Supports establishing a connection with GitHub repositories hosted on a GitHub Enterprise Server.
- **Link Work Items to GitHub Commits, Pull Requests, and Issues**:
  - Allows linking GitHub commits, pull requests, and issues to Azure Boards work items. Work items mentioned in GitHub comments become hyperlinks for easy navigation to Azure Boards.
- **Add Azure Boards Status Badges to GitHub README**:
  - Provides support for adding Markdown syntax to a GitHub repository's README.md file to display the status of a board.
- **View Linked Work Items in Release Summary**:
  - Displays a list of all work items linked to GitHub commits in the Release summary page, helping teams track and gather more information about deployed commits.
- **Sync GitHub Issues to Azure Boards**:
  - Utilizes the GitHub Action, "GitHub Issues to Azure DevOps," to synchronize GitHub Issues with Azure Boards work items.

—-Back to Index—

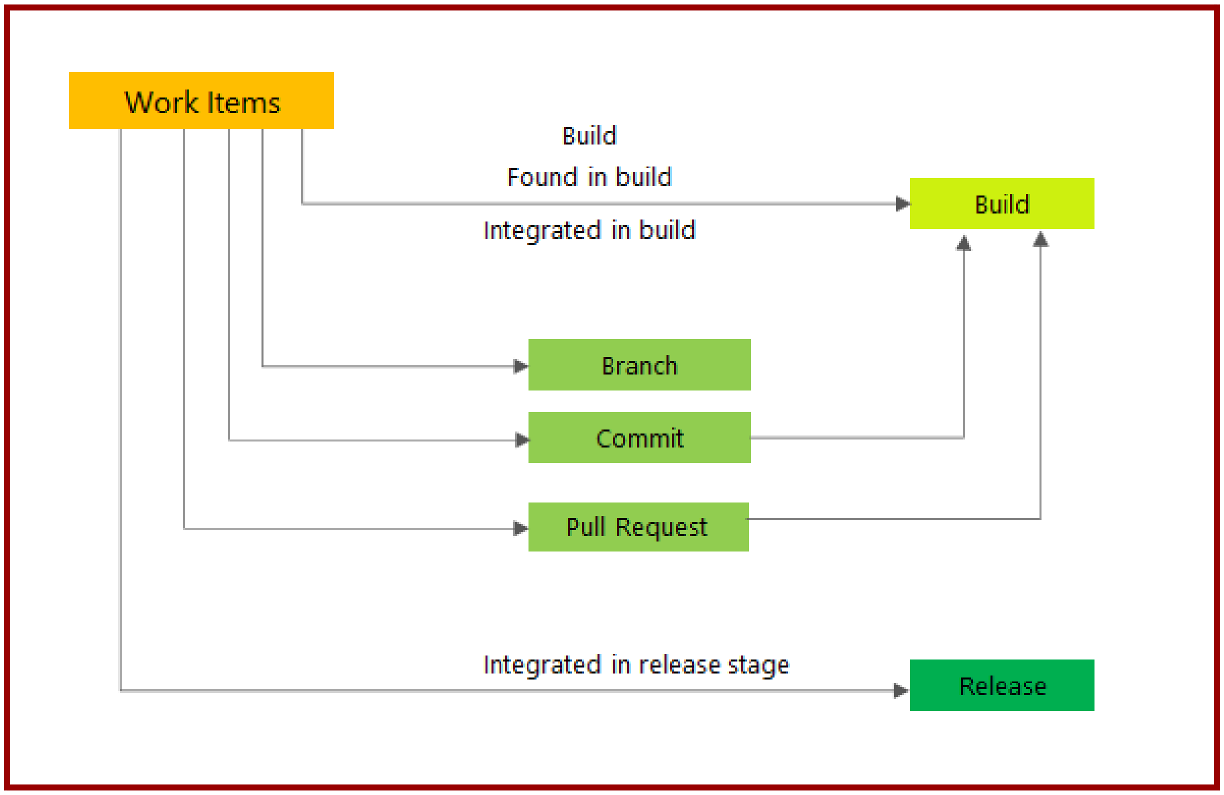
Azure Pipelines and GitHub Integration

Azure Pipelines can be used to automatically build, test, package, release, and deploy code from GitHub repositories. Here's how the integration works:

- **Build, Test, Package, Release, and Deploy GitHub Repository Code**:
  - Automates the entire CI/CD process for code stored in GitHub repositories using Azure Pipelines.
- **Map GitHub Repositories to Azure DevOps Projects**:
  - Allows mapping of GitHub repositories to one or more projects within Azure DevOps for streamlined project management and development workflows.

End-to-end traceability - Azure DevOps

End-to-end traceability in Azure DevOps refers to the ability to track the lifecycle of work items, code changes, builds, and deployments from start to finish. This traceability ensures that all aspects of the development process are linked and visible, enabling teams to understand the relationships between work items, code, and releases, and ensuring transparency and accountability throughout the project lifecycle.



[Source: Microsoft Documentation]

Key Components of End-to-End Traceability

Work Items:

- **Creating Work Items:** Utilize Azure Boards for managing and documenting work items such as user stories, tasks, bugs, and features, ensuring each item includes detailed descriptions, acceptance criteria, and relevant attachments.
- **Linking Work Items:** Establish links between related work items (e.g., linking bugs to user stories) to clarify relationships and dependencies across tasks.

Source Control:

- **Commit Messages:** Provide descriptive commit messages and include work item IDs to directly associate code changes with corresponding work items.
- **Pull Requests:** Initiate pull requests (PRs) for code reviews and merging changes, linking PRs to associated work items to provide contextual traceability.
- **Branch Policies:** Implement policies in source control to enforce practices such as work item linking, code reviews, and build validations.

Builds and Releases:

- **Build Pipelines:** Set up automated build pipelines in Azure Pipelines to build and test code, linking builds to the work items they address or resolve.
- **Release Pipelines:** Create release pipelines to automate deployment across different environments, linking release stages to associated work items for comprehensive traceability.

Test Management:

- **Test Plans and Suites:** Utilize Azure Test Plans to organize test plans, suites, and cases, linking tests to work items to track validation efforts against requirements.
- **Automated Tests:** Integrate automated testing into build and release pipelines, linking test results back to work items to ensure thorough test coverage and traceability.

Dashboards and Reporting:

- **Custom Dashboards:** Develop customized dashboards in Azure DevOps to visualize key metrics and status updates of work items, builds, and releases. Utilize widgets to highlight traceability links between various artifacts.
- **Reports and Queries:** Leverage built-in reporting tools and queries to generate detailed reports on work item progress, code changes, and deployment status. These reports aid in identifying gaps and ensuring comprehensive coverage of all work items.

Implementation Steps

- **Setup Azure Boards:** Utilize Azure Boards to create and manage work items, establishing clear relationships between tasks. Track task progress and ensure linkage to associated code changes, builds, and releases.

—-Back to Index—

- **Configure Git Repositories:** Manage source code using Azure Repos. Ensure each commit and pull request references relevant work items by incorporating work item IDs in commit messages and PR descriptions.
- **Build and Release Pipelines:** Develop CI/CD pipelines within Azure Pipelines. Connect build and release processes directly to work items, facilitating tracking of which builds and deployments address specific tasks and issues.
- **Integrate Testing:** Implement Azure Test Plans for both manual and automated testing. Associate test cases and results with work items to validate adherence to requirements and provide transparency on test coverage.
- **Dashboards and Monitoring:** Establish customized dashboards to visualize the status and progression of work items, builds, and releases. Utilize queries and reports to monitor traceability across the development lifecycle, ensuring comprehensive connectivity.

Best Practices

- **Consistent Naming and Tagging:** Use consistent naming conventions and tagging for work items, branches, commits, and releases to make it easier to track and link them.
- **Automation:** Automate as much as possible, including the creation of links between work items, code changes, builds, and releases, to reduce manual errors and improve traceability.
- **Regular Audits:** Periodically review the traceability links and ensure that all work items are properly linked to their respective code changes, builds, and deployments.
- **Training and Documentation:** Provide training to team members on the importance of traceability and how to use Azure DevOps tools effectively. Maintain documentation on best practices and procedures.

Azure DevOps dashboards, charts, reports, and widgets

Azure DevOps Dashboards:

- An Azure DevOps dashboard is a versatile, user-friendly interface within Azure DevOps that consolidates project data, metrics, and status updates.
- It assists teams in visualizing and monitoring various aspects of their projects, improving project management and collaboration.
- Dashboards are linked to a specific team or project and feature customizable charts and widgets.

Components of Azure DevOps dashboards:

- **Charts:** These are query-based status or trend charts generated from a work item query or test results.
- **Widgets:** Configurable items that display various information and charts on dashboards. The widget catalog provides brief descriptions of available widgets, and you can also add widgets from the Azure DevOps Marketplace.

—-Back to Index—

- **Reports:** System-generated charts supporting specific services, such as team velocity, sprint burndown, the Cumulative Flow Diagram (CFD), and the Test Failures report. These are displayed on the Analytics tab for a specific service and are derived from Analytics data.

Access and permissions:

- **Viewing Dashboards:** Only valid project members can view dashboards.
- **Creating and Editing Dashboards:** All Azure DevOps project members can view dashboards, but only those with Basic access or higher can create and edit them.
- **Granular Permissions:** Using permission management command-line tools, you can set more specific permissions, such as Read, Create, and Materialize Dashboard permissions.

Examples of Dashboard Widgets

- **Build and Release Widgets:** Show the status of recent builds and releases.
- **Work Item Widgets:** Display counts and lists of work items based on queries.
- **Pull Request Widgets:** Provide details and statuses of pull requests.
- **Code Coverage Widgets:** Show the percentage of code covered by automated tests.
- **Test Results Widgets:** Display results from automated test runs.

Benefits

- **Visibility:** Offers a clear view of project status and progress.
- **Collaboration:** Improves team collaboration by sharing critical project information.
- **Efficiency:** Saves time by centralizing essential project data in one location.
- **Proactive Management:** Enables teams to proactively manage issues and risks by monitoring key metrics.

How to Access and Create Dashboards

1. **Log in to Azure DevOps:** Access your Azure DevOps organization.
2. **Select a Project:** Choose the project for which you want to create a dashboard.
3. **Access Dashboards:** Click on "Dashboards" in the left-hand navigation menu.
4. **Create a New Dashboard:** Click "New Dashboard" and follow the prompts to name and configure your dashboard.
5. **Add Widgets:** Use the "+ Add Widget" button to include the desired widgets.
6. **Customize Layout:** Arrange the widgets to suit your needs.

—-Back to Index—

Azure DevOps Services REST API Reference

The Azure DevOps Services REST API serves as a versatile tool for interacting with Azure DevOps services programmatically. This API empowers users to automate tasks, integrate with external systems, and access detailed data concerning work items, repositories, builds, releases, and more. Below is an exploration of utilizing the Azure DevOps Services REST API, including key components and practical implementation steps:

Understanding Azure DevOps REST API

The Azure DevOps REST API facilitates interactions with Azure DevOps services, covering various areas such as:

- **Work Items:** Allows manipulation of work items, including creation, retrieval, updating, and deletion.
- **Git Repositories:** Provides functionalities for managing repositories, branches, pull requests, and commits.
- **Build and Release Pipelines:** Enables triggering builds, managing build definitions, and handling release pipelines.
- **Test Management:** Offers capabilities for creating and managing test plans, test suites, and test cases.
- **Identity and Access Management:** Supports user, group, and permission management.

Authentication Mechanisms

Authentication is essential for accessing the Azure DevOps REST API, and common authentication methods include:

- **Personal Access Tokens (PATs):** Simple tokens used for authentication, suitable for scripts and integrations.
- **OAuth:** Facilitates applications to access resources on behalf of users, ideal for third-party integrations.
- **Microsoft Entra ID:** Ideal for enterprise-level applications requiring robust authentication and authorization mechanisms.

Crafting API Requests

Azure DevOps REST API requests follow a structured format, typically incorporating the following components within the URL:

https://dev.azure.com/{organization}/{project}/_apis/{area}/{resource}?api-version={version}

Sample API Requests

Retrieving a List of Work Items

GET
https://dev.azure.com/{organization}/{project}/_apis/wit/workitems?ids={ids}&api-version=6.0

Creating a New Work Item

```
POST
https://dev.azure.com/{organization}/{project}/_apis/wit/workitems/${type}?api-version=6.0

Content-Type: application/json-patch+json

[

 {

   "op": "add",

   "path": "/fields/System.Title",

   "from": null,

   "value": "Sample work item"

 }

]
```

Triggering a Build

```
POST https://dev.azure.com/{organization}/{project}/_apis/build/builds?api-version=6.0

Content-Type: application/json

{

 "definition": {

   "id": {buildDefinitionId}

 },

 "sourceBranch": "refs/heads/main"

}
```

Versioning Considerations

It's crucial to specify the API version in requests using the api-version query parameter, ensuring compatibility with future updates. For example, api-version=6.0.

Using cURL for API Requests

cURL, a command-line tool for HTTP requests, can be utilized to interact with the Azure DevOps REST API. Here's an example:

```
curl -u :{personal-access-token} \
  -X GET \
'https://dev.azure.com/{organization}/{project}/_apis/wit/workitems?ids={ids}&api-version=6.0'
```

Best Practices

- **Security**: Ensure secure handling and storage of personal access tokens, adhering to least privilege principles.
- **Rate Limiting**: Be mindful of rate limits and implement appropriate handling mechanisms within applications.
- **Error Management**: Implement robust error handling strategies to address API errors and retries effectively.

Git history, Repository Settings and Policies in Azure Repos

Git history

Git history refers to the record of all changes made to a repository over time. This history is an essential aspect of Git's version control system, enabling users to track, review, and manage the evolution of a project's files and code.

Key Components of Git History

- Commits: A commit represents a state of the repository at a particular moment.
- Branches: Branches are pointers to specific commits, enabling users to work on features, fix bugs, or experiment separately from the main codebase.
- Merge: The merging process combines changes from different branches. A merge commit documents the integration of these changes into a specific branch.
- Rebase: Rebasing involves relocating or integrating a series of commits onto a new base commit.
- Tags: Tags are used to label significant points in the Git history, such as releases (e.g., v1.0, v2.0). Unlike branches, tags are static and do not change.
- Commit Graph: The commit graph visually displays the history of commits within a repository, illustrating the connections between commits, branches, and merges.

Benefits of Git History

- Traceability: Every change is recorded with details about who made the change, when it was made, and why.
- Collaboration: Multiple developers can work on different parts of the project simultaneously. The history helps in merging changes and resolving conflicts.
- Revert Changes: If a bug or issue is introduced, the history allows you to revert to previous states of the project.

- **Review and Auditing**: Commits can be reviewed and audited to ensure code quality and compliance with standards.

Best Practices for Managing Git History

- **Write Meaningful Commit Messages:** Clear and descriptive commit messages help in understanding the purpose of each change.
- **Commit Often**: Small, frequent commits make it easier to track changes and identify issues.
- **Use Branches**: Create branches for new features, bug fixes, or experiments to keep the main branch clean and stable.
- **Rebase Carefully**: Use rebase to maintain a clean history, but be cautious when rebasing shared branches.

Common Git History Commands

- **git log:** Outputs a list of commits in the repository's history.
- **git show <commit>:** Displays the details of a specific commit.
- **git blame <file>:** Identifies who made changes to each line of a file and when those changes were made.
- **git reflog:** Presents a log of all reference updates in the repository.
- **git bisect:** Uses a binary search method to pinpoint the commit that introduced a bug in the commit history.

Repository Settings and Policies in Azure Repos

Azure Repos provides various settings and policies that allow you to manage and control your repositories effectively. These settings and policies help ensure code quality, security, and adherence to workflow standards.

Repository Settings

Repository settings in Azure DevOps allow you to configure different aspects of your repositories to fit your project's requirements.

Key Settings

1. **General Settings**
    - **Repository Name and Description:** Set or update the repository name and provide a brief description of its purpose.
    - **Default Branch:** Define the default branch (commonly main or master) for the repository.
2. **Permissions**
    - **User Access Levels:** Manage who can access the repository and their permission levels (read, contribute, manage).
    - **Group Permissions:** Assign permissions to groups for streamlined access management.

3. **Policies**
    - Implement policies to enforce coding standards and maintain code quality.
    - Customize policies for different branches as needed.
4. **Service Hooks**
    - Integrate the repository with external services using webhooks or built-in service hooks.
    - Automate actions like triggering builds, sending notifications, or updating work items.
5. **Security**
    - Configure security settings, including access controls and SSH keys.
    - Make sure that access and modifications to the repository are restricted to authorized users only.

Branch Policies

Branch policies are rules applied to branches to ensure code quality, enforce standards, and manage the workflow effectively.

Key Policies

1. **Code Review**
    - Require a certain number of reviewers to approve pull requests before they can be merged.
    - Ensure that code changes are peer-reviewed.
2. **Build Validation**
    - Automatically trigger builds for new changes in a branch.
    - Require that the code builds successfully before it can be merged.
3. **Status Checks**
    - Mandate that specific checks (like passing tests or code analysis) must be successful before a pull request can be completed.
    - Integrate with continuous integration tools to automate these checks.
4. **Branch Protection**
    - Prevent force pushes and deletions of branches.
    - Restrict who can push to protected branches.
5. **Work Item Linking**
    - Require that pull requests are linked to work items.
    - Ensure traceability between code changes and related work items.
6. **Commit Signatures**
    - Require commits to be signed with a verified signature.
    - Ensure commits are made by verified contributors, enhancing security.

Webhooks and Serviehooks with Azure DevOps

Webhooks

Webhooks are HTTP callbacks that get triggered by specific events. When such an event occurs, like a code commit, the webhook sends an HTTP POST request to a specified URL with details about the event.

Key Characteristics:

- **Instant notifications:** Webhooks provide immediate updates and can initiate workflows in response to events.
- **Customizable payload:** The information sent can be tailored to include pertinent details.
- **External service integration:** Webhooks enable Azure DevOps to connect with external systems such as Slack, Jenkins, or custom-built applications.

Use Cases:

- **Custom Integrations:** When you need to integrate Azure DevOps with a custom-built application.
- **Advanced Workflows:** When the default integrations don't meet your specific requirements and you need to control the data flow precisely.

Service Hooks

Service Hooks in Azure DevOps offer a user-friendly approach to integrating with other services. They provide ready-made integrations with popular tools, allowing actions to be automated based on events in Azure DevOps.

Key Characteristics:

- **Prebuilt integrations:** Service Hooks support numerous services such as GitHub, Jenkins, Slack, and Trello.
- **Easy setup:** The configuration process is straightforward, with guided instructions.
- **Event-driven actions:** Trigger actions in external services based on Azure DevOps events

Examples and Use Cases

- **CI/CD Integration:** Initiate Jenkins jobs or GitHub actions when code is pushed to an Azure DevOps repository.
- **Notifications:** Send updates to a Slack channel for events like new pull requests or work item updates.
- **Issue Tracking:** Automatically create Trello cards or GitHub issues for new bugs reported in Azure DevOps.

Comparison summary

Feature	Webhooks	Service Hooks
Flexibility	High	Moderate
Ease of Use	Requires technical knowledge	User-friendly, guided setup
Customization	Fully customizable payloads	Limited to predefined configurations
Integration	Custom and any HTTP endpoint	Popular services with built-in support
Setup Complexity	More complex, and needs custom handling	Simple and quick setup

When to Use Each:

- **Webhooks** are best when you need detailed control over the integration process or when integrating with custom services that aren't supported by Service Hooks.
- **Service Hooks** are ideal for quick setups and integrations with popular services where the predefined configurations meet your needs.

GitHub to Azure Boards

What are Azure Boards?

Azure Boards is an online tool designed to help teams organize, monitor, and communicate about their projects throughout the entire development lifecycle, supporting agile practices. It offers a flexible platform for managing tasks, enabling effective team collaboration and optimizing work processes.

Integration of Azure boards and GitHub Repositories

- Connect your Azure Boards with GitHub repositories to synchronize work across both platforms.
- Enable linking of GitHub commits, pull requests, and issues to specific work items in Azure Boards.
- This integration ensures that all development activities in GitHub are directly associated with your planning and tracking efforts in Azure Boards, fostering better coordination and visibility.

How do Azure board apps help integrate with Git Hub?

- **Streamline Project Planning and Tracking with Azure Boards:**
    - Use Azure Boards to organize, plan, and track your work throughout the development cycle.

    - Benefit from its support for agile methodologies to enhance your team's workflow and productivity.
    - Customize work item tracking to fit your team’s unique requirements and processes.
- **Manage Source Code with GitHub:**
    - Use GitHub as your primary tool for source control in software development.
    - Utilize GitHub's features for efficient code management, including version control, pull requests, and issue tracking.
- **Integrate Azure Boards with GitHub for Enhanced Collaboration:**
    - Connect Azure Boards with your GitHub repositories to synchronize development activities.
    - Link GitHub commits, pull requests, and issues to specific work items in Azure Boards.
    - This integration ensures seamless coordination, allowing you to track the progress of your development work comprehensively across both platforms.

How to connect GitHub with Azure Boards?

- **Connect Azure Boards to GitHub:**
    - Authenticate to GitHub when setting up the connection to Azure Boards.
    - For GitHub in the cloud, you can choose from the following authentication options:
        - Username/Password
        - Personal Access Token (PAT)
- **Steps to Make the Connection:**
    - Follow the detailed walkthrough available at: Connect Azure Boards to GitHub.
- **Configure Azure Boards and GitHub Integration:**
    - From the Azure Boards app page, you can configure various Azure Boards/Azure DevOps Projects and GitHub.com repositories.
    - Adjust or update the existing configuration as needed.
- **Manage Repositories Post-Integration:**
    - Once Azure Boards is integrated with GitHub using the Azure Boards app, you can manage repositories directly from the Azure Boards web portal.
    - Add or remove repositories as necessary to keep your integration up-to-date.

Configure GitHub Projects

Creating a Project on GitHub

1. Set Up an Organization Project:

- Visit the main page of your organization on GitHub.
- Click "Projects."
- From the "New project" drop-down list, select "New project."

2. Set Up a User Project:

- On any GitHub page, click your avatar and choose "Your projects."
- From the "New project" drop-down list, select "New project."

3. Add a Project Description or README File:

- Navigate to your project.
- Click the menu icon in the top-right corner.
- Select "Settings" from the dropdown menu.
- Under "Add a description," input your description in the text box and click "Save."
- To update the README file, type your content in the provided text box under "README" and click "Save."

4. Add Issues:

- Upon initializing your new project, you will be prompted to add items.
- Click the plus (+) icon to add additional issues.

5. Adjust Project Settings and Permissions:

- Go to your project and click in the upper-right corner to quickly view or modify your project description and README.
- Create custom fields to include in your project.
- Manage access and permissions by adding collaborators.

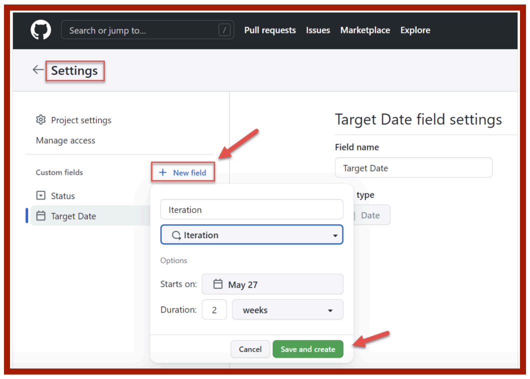
Manage work with GitHub Project Boards

1. **Create Iterations:**
    - Utilize GitHub Projects to manage project deliverables, release dates, and iterations for planning upcoming work.
    - Create iterations to associate items with specific recurring time blocks, accommodating any length of time and including breaks.
2. **Set Up Iteration Field:**
    - Access the project interface.
    - Click on the plus (+) sign in the rightmost field header.

—-Back to Index—

- Choose "New field" from the drop-down menu.
- Provide a name for the new iteration field.
- Select "Iteration" from the dropdown menu.
- Optionally, adjust the starting date and duration for each iteration.
- Click "Save and create."

3. **Adding New Iterations:**
   - Navigate to your project.
   - Open the menu by clicking on the top-right corner.
   - Access project settings by clicking "Settings" from the menu.
   - Choose the iteration field you want to adjust.
   - Click "Add iteration" to include a new iteration of the same duration.
   - Optionally, customize the duration and starting date of the new iteration.
   - Click "Add" to confirm the new iteration.
   - Click "Save changes" to finalize the adjustments.
4. **Insert Breaks into Iterations:**
   - Communicate scheduled breaks by inserting them into your iterations.
   - This allows you to indicate when you're taking time away from the planned work.
   - Follow similar steps as above to adjust the duration and starting date to accommodate breaks.

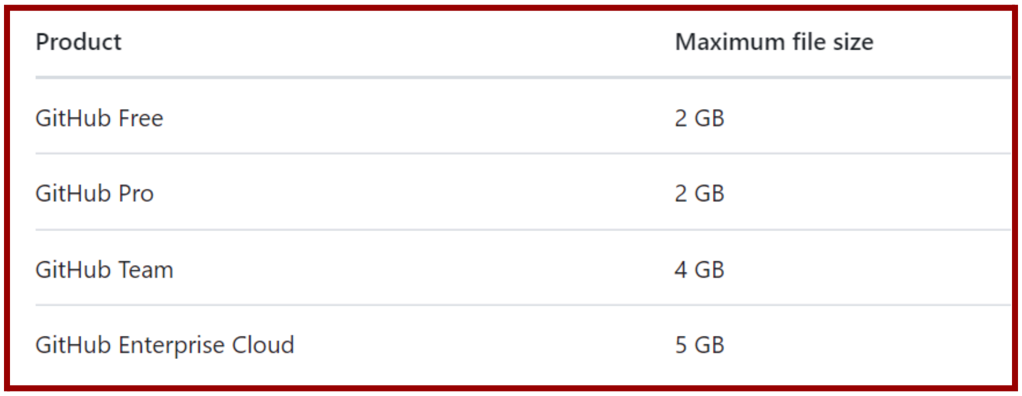


[Source: Microsoft Documentation]

Configure collaboration and communication

What is Git LFS?

GitHub imposes restrictions on the size of files that can be stored in repositories. To manage files exceeding these limits, you can utilize Git Large File Storage (Git LFS). Git LFS is an extension of Git that tracks large files by storing their binary contents in a separate remote storage. It provides metadata describing the large files within commits to your repository.
When you clone your repository or switch branches, Git LFS retrieves the correct version of these files from the remote storage. Your local development tools handle the files as if they were committed directly to your repository, ensuring a seamless workflow.
The maximum file size limits for Git LFS vary based on your GitHub plan.



Product	Maximum file size
GitHub Free	2 GB
GitHub Pro	2 GB
GitHub Team	4 GB
GitHub Enterprise Cloud	5 GB

[Source: Microsoft Documentation]

Benefits:

- **Seamless Workflow:** Git LFS allows your team to use the familiar Git workflow regardless of the types of files created.
- **Efficient Management:** It handles large files, preventing them from negatively impacting the overall repository performance.
- **File Locking:** Starting with version 2.0, Git LFS supports file locking, which is useful for managing large, non-diffable assets like videos, audio files, and game maps.

Limitations:

- **Client Requirements:** Every Git client used by your team must have the Git LFS client installed and properly configured to track large files.
- **Binary File Visibility:** Without the Git LFS client installed, you won't see the actual binary files when you clone the repository. Instead, you'll only see metadata describing the large files. Committing large binaries without the Git LFS client will push the binaries directly to the repository.
- **Merging Limitations:** Git cannot merge changes from two different versions of a binary file, even if they share a common parent.

—-Back to Index—

- **SSH Support:** Azure Repos currently do not support Secure Shell (SSH) for repositories with Git LFS-tracked files.
- **Upload Time Limit:** There is a one-hour time limit for uploading a single file.

Introducing Scalar: Git at scale for everyone

Scalar is a tool created by Microsoft to enhance Git's performance and usability, particularly for large repositories. It's designed to help teams manage extensive codebases more efficiently. Scalar is a .NET Core application designed to enhance Git command performance on Windows and macOS.

Key Features of Scalar:

1. **Integration with Virtual File System (VFS) for Git:** Scalar integrates seamlessly with VFS for Git, virtualizing the file system to streamline access to repository data. This approach reduces access times for large repositories by focusing on necessary parts.
2. **Automated Background Maintenance:** Scalar automates Git maintenance tasks like fetching and pruning in the background. This automation minimizes disruptions to developers' workflows while ensuring the repository remains optimized.
3. **Enhanced Performance:** Scalar boosts the speed of critical Git operations such as cloning, fetching, and branch checkouts, particularly beneficial for handling large repositories efficiently.
4. **Multi-Platform Support:** With support for Windows and macOS, Scalar caters to diverse development environments, offering flexibility in deployment and usage across different platforms.
5. **Simple Setup and Configuration:** Scalar simplifies the initial setup and configuration of Git in large repositories, enabling developers to start working promptly without extensive Git expertise.
6. **Support for Large Repositories:** Designed to address performance bottlenecks and scalability challenges inherent in large repositories, Scalar ensures smooth operations even with extensive codebases.

Benefits of Using Scalar:

- **Enhanced Productivity:** Scalar accelerates Git operations, minimizing developers' wait times and enhancing productivity during work sessions.
- **Lower Cognitive Burden:** Automated maintenance and optimizations streamline repository management, enabling developers to concentrate more on coding tasks.
- **Enhanced Collaboration:** Efficient management of large repositories fosters smoother and more productive teamwork among developers.

Getting Started with Scalar:

1. **Installation:** Scalar is installable through package managers or from source, depending on your platform preferences.

2. **Configuration:** Post-installation, Scalar offers straightforward commands to configure and optimize repositories for large-scale operations.
3. **Integration:** Scalar seamlessly integrates into existing Git workflows with minimal developer intervention required.

Git branching strategy, Policies and Settings in Azure Repos

Establishing a robust Git branching strategy, along with effective policies and settings in Azure Repos, is essential for maintaining code quality and facilitating seamless collaboration within development teams. Here's an overview of these elements:

Git Branching Strategy

1. **Main Branching Models:**
    - **Feature Branching:** Developers create separate branches to work on new features or tasks, branching from the main branch. Once completed and reviewed, these branches are merged back into the main branch.
    - **Release Branching:** Derived from the main branch, release branches are used to finalize a release by incorporating final bug fixes and adjustments.
    - **Hotfix Branching:** Branches created urgently to address critical issues in the production code, typically branching from the main branch or a release branch.
2. **Common Branch Types:**
    - **Main (or Master) Branch:** Represents a stable, production-ready state at all times.
    - **Development Branch:** Serves as an integration branch where feature branches are combined and tested before merging into the main branch.
    - **Feature Branches:** Temporary branches created for developing new features.
    - **Release Branches:** Used to prepare and finalize releases.
    - **Hotfix Branches:** Quickly created to resolve critical issues in the production environment.

Branch Policies in Azure Repos

- **Review Requirements:** Establish policies that mandate a minimum number of reviewers to approve pull requests before merging, fostering code quality and knowledge sharing.
- **Build Validation:** Enforce policies that necessitate successful completion of Continuous Integration (CI) builds prior to merging pull requests, preventing build failures caused by new code.
- **Status Checks:** Integrate external services to perform checks such as code quality analysis and security scans, ensuring that all prerequisites are met before merging pull requests.
- **Comment Resolution:** Require resolution of all comments within a pull request before it can be merged, ensuring that feedback is adequately addressed.

- **Work Item Linking:** Require pull requests to be linked to associated work items (e.g., tasks, bugs, user stories) to maintain traceability between code changes and project requirements.

Settings in Azure Repos

Secure vital branches (e.g., main, release) by prohibiting direct pushes and mandating pull requests for changes.

1. **Merge Strategies:** Customize merge techniques (e.g., merge commit, squash merge, rebase) to align with your team's workflow preferences.
2. **Repository Permissions:** Define precise permissions for different users and groups to regulate repository access for reading, contributing, and administering.
3. **Branch Naming Conventions:** Establish and enforce uniform branch naming conventions (e.g., feature/*, bugfix/*, release/*) to enhance clarity and organization.
4. **Commit Policies:** Enforce policies for consistent commit message formats, enhancing transparency and maintaining a clear commit history.

Best Practices

- **Regularly Merge Changes**: Frequently merge changes from the main branch into feature branches to keep them updated and resolve conflicts early.
- **Clean-Up Branches:** Regularly delete merged and stale branches to keep the repository organized and manageable.
- **Document Policies**: Clearly document branching strategies, policies, and guidelines to ensure all team members understand and follow them consistently.

By establishing a well-defined Git branching strategy and implementing effective policies and settings in Azure Repos, development teams can enhance code quality, streamline workflows, and improve collaboration.

Package Management Strategy with Azure Artifacts

Developing an effective package management strategy using Azure Artifacts is essential for optimizing the development process, ensuring consistency, and enhancing security. Azure Artifacts, part of Azure DevOps, enables teams to create, host, and share packages seamlessly. Here's how to create a successful package management strategy with Azure Artifacts:

Core Elements of a Package Management Strategy

Repository Structure:

- **Feed Organization:** Organize feeds by teams, projects, or package types (e.g., NuGet, npm, Maven, Python) to streamline access control and permissions management.
- **Public vs. Private Feeds:** Utilize public feeds for open-source projects and private feeds for internal use to regulate package distribution and access.

Versioning Approach:

- **Semantic Versioning:** Implement semantic versioning (MAJOR.MINOR.PATCH) to clearly communicate changes, aiding in dependency management and impact assessment.
- **Pre-release Versions:** Employ pre-release versions (e.g., 1.0.0-alpha, 1.0.0-beta) for testing purposes before stable releases, ensuring controlled testing phases.

Package Promotion:

- **Lifecycle Stages:** Define stages (e.g., Development, Testing, Production) for promoting packages through their lifecycle, ensuring only validated packages move to production.
- **Automated Promotion:** Automate package promotion using CI/CD pipelines based on predefined criteria such as successful builds and tests, enhancing efficiency and reliability.

Managing Dependencies:

- **Centralized Dependency Management:** Centralize dependency management to maintain consistency across projects, controlling versions and updates centrally.
- **Dependency Resolution:** Implement tools and practices to ensure consistent resolution of dependencies across diverse environments, reducing compatibility issues.

Security and Compliance:

- **Access Controls:** Establish access controls to govern package publishing and consumption, leveraging Azure DevOps security groups for streamlined permissions management.
- **Package Scanning:** Integrate security scanning tools to detect vulnerabilities in packages, ensuring regular scans prior to promotion to maintain system integrity.
- **License Management:** Track and manage package licenses to ensure compliance with open-source and third-party software licensing requirements, mitigating legal risks.

Implementation Steps

Set Up Feeds:

- Establish Azure Artifacts feeds tailored for various teams, projects, or package types.
- Configure permissions to regulate package publishing and consumption.

Configure CI/CD Pipelines:

- Create CI/CD pipelines to automate build, test, and deployment processes.
- Include steps in the pipeline to publish packages to Azure Artifacts feeds.

Automate Versioning:

- Implement GitVersion or similar tools to automate versioning based on commit history and branch strategy.
- Ensure adherence to semantic versioning principles for consistent version management.

Integrate Dependency Management:

- Utilize package management tools such as NuGet, npm, or Maven to manage dependencies.
- Configure projects to retrieve packages from designated Azure Artifacts feeds.

Monitor and Maintain:

- Regularly audit feeds to remove obsolete or unused packages, maintaining cleanliness and efficiency.
- Monitor feed usage and performance to optimize resource allocation and meet team requirements effectively.

Best Practices

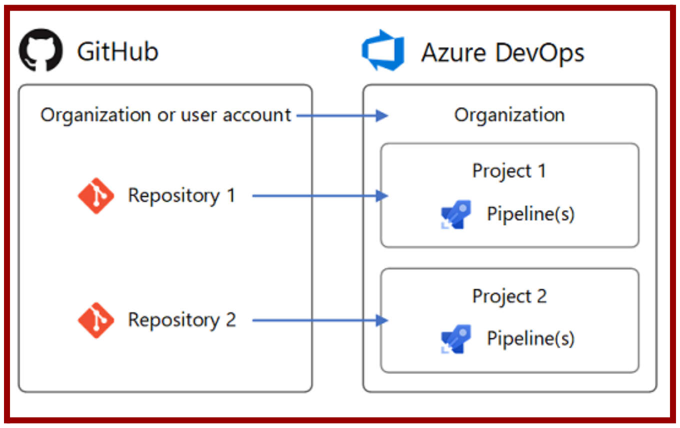
- **Documentation**: Thoroughly document the package management process, including instructions on creating, publishing, consuming, and promoting packages. Ensure all team members are well-versed in these practices.
- **Consistent Naming Conventions**: Apply consistent naming conventions for packages to simplify identification and management.
- **Regular Audits:** Conduct regular audits of packages and dependencies to identify and resolve security vulnerabilities and licensing issues.
- **Feedback Loop:** Establish a feedback loop with developers to continuously improve the package management process and address issues promptly.

Integrate GitHub repositories with Azure Pipelines

Azure DevOps GitHub integration

What is GitHub?

GitHub is a cloud-based platform where developers can collaborate on code. It offers features like version control, access control, bug tracking, and task management for software projects. GitHub can be integrated with Azure DevOps Boards and Azure DevOps Pipelines.



[Source: Microsoft Documentation]

Azure DevOps Board & GitHub Integrations

When you integrate Azure Boards with GitHub repositories, you establish a connection that allows you to link GitHub commits, pull requests, and issues to your work items. This means you can use GitHub for software development while leveraging Azure Boards to manage and track your work.

Integration: Prerequisites:

- You need to have an Azure Boards or Azure DevOps project
- You must belong to the Project Administrators group.
- You must be an administrator or owner of the GitHub repository to connect to.

Steps:

1. From Azure Boards to GitHub
2. Sign in to your Azure DevOps project
3. Go to Project settings > GitHub connections
4. If this is your first time establishing a connection from the project, select "Connect your GitHub account" to use your GitHub credentials. If not, choose "New connection" and select your authentication method from the New Connection dialog.
5. From GitHub to Azure Boards
   Set up and configure the Azure Boards app for GitHub.
6. Authentication options:
   User credentials
   PAT Token

Benefits:

- Smooth Collaboration: Work items in Azure Boards can be linked to GitHub commits, pull requests, and issues.
- Enhanced Traceability: By linking code changes to specific work item in Azure Boards, its progress can be tracked.
- Effective Planning: Utilize Azure Boards for planning, backlog management, and work tracking.
- Increased Visibility: Work items can directly be seen within GitHub, making it simpler to comprehend code changes and prioritize tasks.
- Automated Updates: When GitHub commits or pull requests is associated with work item, Azure Board automatically updates the status of the respective work item.

Use Cases:

- Link Work Items to GitHub Activity: Establish connections between Azure Boards work items (such as user stories, bugs, and tasks) and GitHub commits, pull requests, and issues. This facilitates collaboration between development and project management teams.
- Comment from Work Items to GitHub: Post comments directly from Azure Boards work items to linked GitHub commits, pull requests, or issues. Keep all stakeholders informed.
- Visualize GitHub Objects on Kanban Boards: Display GitHub commits, pull requests, and issues directly on your Azure Boards Kanban board. Gain a comprehensive view of development progress.
- Manage Repository Access: Control access to GitHub repositories from Azure Boards. Adjust connections, suspend integration, or uninstall the app as needed.

Azure DevOps Pipeline & GitHub Integration

Azure Pipelines enables automatic building, testing, packaging, releasing, and deployment of your code from GitHub repositories.You can accomplish this integration through GitHub Actions.

GitHub Actions:

GitHub Actions is a CI/CD (Continuous Integration/Continuous Deployment) platform that automates the build, test, and deployment processes directly within your GitHub repository. GitHub Actions enables the creation of personalized workflows for various tasks such as code building, testing, application deployment, and beyond.

These workflows are outlined through a YAML file, allowing you to define the necessary steps for execution.

Whether your aim is to validate pull requests, conduct tests, or deploy applications, GitHub Actions offers a versatile and robust approach to enhancing your development processes.

GitHub Actions can be used for variety of tasks:

- Automated testing.
- Automatically responding to new issues, mentions.
- Triggering code reviews.
- Handling pull requests.
- Branch management.

Integration Steps:

- Log into your Azure Dev Ops organization and go to your project.
- Go to Pipelines, and then select Newpipeline or Create pipeline if it’s your first pipeline).
- In the wizard,select GitHub as the location for your sourcecode.If prompted, sign in to GitHub by entering your credentials.

Benefits:

AutomatedBuilds: Azure Pipelines can automatically build and validate every pull request and commit to your GitHub repository.

ContinuousDeployment(CD): You can createa continuous deployment pipeline using Azure Pipelines to deploy your application to actual running infrastructure after successful builds.

Free for Open Source Projects: Azure Pipelines provides unlimited CI/CD minutes and 10 parallel jobs for every GitHub open source project, running on the same infrastructure as paying customers.

Best Practices:

- CLI Usage Alongside YAML Tasks: Utilize the command-line interface (CLI) for tasks not directly supported by YAML tasks.
- Terraform Partial Configuration: When working with Terraform, consider using partial configuration to manage specific resources or modules. This helps prevent unnecessary resource re-creation during pipeline runs.
- Secure Authentication via Azure Key Vault: Store sensitive credentials (such as service principal secrets) securely in Azure Key Vault. Reference these secrets in your pipelines to authenticate with Azure services.
- YAML Pipelines Over UI: Opt for defining your build and release pipelines using YAML files. This approach ensures version control, consistency, and better visibility into your pipeline configurations.

Use Cases:

- Continuous Integration (CI): Set up CI pipelines that trigger automatically whenever there's a code commit or a pull request in your GitHub repository. CI helps catch issues early and ensures code quality.
- Automated Builds and Testing: Utilize Azure Pipelines to automatically build, test, and package your GitHub repository code. This ensures consistent and reliable builds for every commit or pull request.
- Continuous Deployment (CD): Implement CD pipelines in Azure Pipelines to deploy your GitHub code to various environments (e.g., staging, production) based on predefined conditions. This streamlines the release process.
- Multi-Platform Builds: Azure Pipelines supports building code for different platforms (Windows, Linux, macOS). You can build cross-platform applications hosted on GitHub using Azure Pipelines.
- Integration with GitHub Actions: Combine the capabilities of Azure Pipelines with GitHub Actions. Use Azure Pipelines for more complex workflows or scenarios that require customizations beyond what GitHub Actions provides.

—-Back to Index—

Agile Plan and Portfolio Management with Azure Boards

Features:

Azure Boards provides agile planning and portfolio management tools and processes that enable you to effectively plan, manage, and track work across your team. Learn how to use the product backlog, sprint backlog, and task boards to oversee work progress throughout an iteration. Furthermore, explore the new features in this release that enhance scalability for larger teams and organizations.

Azure Boards' agile planning and portfolio management tools:

1. **Product Backlog:**
   - The product backlog acts as a prioritized list of work items, including user stories and bugs.
   - It facilitates tracking and managing tasks within your project.
   - You can assign these work items to teams and monitor their progress.
2. **Sprint Backlog:**
   - The sprint backlog focuses on work specific to a particular iteration (sprint).
   - It includes tasks, user stories, and other planned work items for that sprint.
   - Teams collaborate and track their workflow during the development cycle.
3. **Task Board:**
   - Task boards visually represent work items within a sprint.
   - You can move items across columns (e.g., "To Do," "In Progress," "Done") to track their status.
   - This provides a clear overview of work distribution and progress.
4. **Dashboard and Analytics report:**
   - Azure Boards offers dashboards and analytics for monitoring project health.
   - Customizable dashboards with widgets display relevant metrics.
   - Analytics reports provide insights into team performance, velocity, and trends.
5. **Scalability for Larger Teams:**
   - Recent enhancements make Azure Boards suitable for larger teams and organizations.
   - It supports multiple teams, areas, and iterations. Also manage membership, notifications and other setting for each team

—-Back to Index—

Design and implement pipeline automation

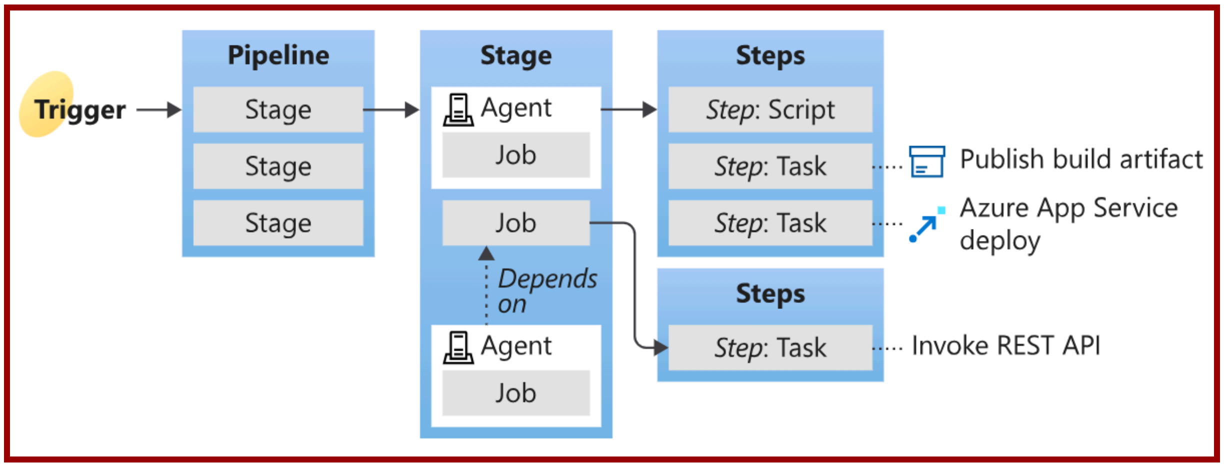
Azure Pipelines Overview and Working

- Azure Pipelines is a cloud-based service from Microsoft Azure that facilitates continuous integration (CI) and continuous delivery (CD).
- It automates the building, testing, and deployment of code to various platforms, ensuring seamless development and release cycles.
- It is compatible with all major programming languages and types of projects.

Overview

1. **Continuous Integration (CI):**
   - **Automated Builds:** Ensures code compiles successfully by automating the build process.
   - **Automated Testing:** Validates code functionality and quality through automated tests.
   - **Developer Feedback:** Provides immediate feedback on code quality and status, aiding in swift issue resolution.
2. **Continuous Delivery (CD):**
   - **Automated Deployment:** Simplifies the deployment process across different environments.
   - **Release Management:** Ensures reliable and predictable code deployments.

Azure Pipeline Structure



[Source: Microsoft Documentation]

Key Features

- **Multi-Language and Platform Support:** Compatible with languages like Java, JavaScript, Python, .NET, PHP, etc., and platforms such as Windows, macOS, and Linux.

—-Back to Index—

- Repository Integration: Works with GitHub, Azure Repos, Bitbucket, and other Git repositories.
- Scalability: Accommodates projects of all sizes, offering scalable build and deployment infrastructure.
- Customization: Supports custom scripts and tasks for extensive flexibility.

Azure Pipelines Working

1. **Pipeline Setup:**
   - **YAML Configuration:** Pipelines are defined in YAML files (azure-pipelines.yml), detailing stages, jobs, and steps.
   - **Classic Editor:** A visual tool for configuring and managing pipelines without writing YAML.
2. **Pipeline Structure:**
   - **Stages:** Major phases such as Build, Test, and Deploy.
   - **Jobs:** Units of work within stages, which can run on different agents or concurrently.
   - **Steps:** Tasks within jobs, like script execution or deployment tasks.
3. **Building and Testing:**
   - **Triggers:** Pipelines can be triggered by code commits, pull requests, or scheduled times.
   - **Agents:** Virtual machines (either Azure-hosted or self-hosted) that execute the pipeline tasks.
   - **Artifacts:** Outputs from the build process (e.g., compiled code, packages) stored for later use.
4. **Deployment:**
   - **Environments:** Deployment targets like development, staging, or production.
   - **Approvals and Gates:** Optional checks and approvals before deployments to ensure quality.
   - **Release Pipelines:** Manage the deployment of build artifacts across environments.

Azure Pipelines Pricing

Free Tier: Azure Pipelines extends a complimentary tier with the following features:

Public Projects:

- Unrestricted parallel jobs, accommodating any necessary concurrency.
- Ten complimentary parallel jobs supported by Microsoft-hosted agents.
- Self-hosted agents boast unlimited usage.

Private Projects:

- One complimentary parallel job, deployable via Microsoft-hosted agents.

- A monthly allowance of 1,800 minutes (equivalent to 30 hours) for private projects.
- Unlimited usage for self-hosted agents, inclusive of one complimentary parallel job.

Paid Plans: Azure Pipelines furnishes paid options tailored to users and organizations with heightened resource demands:

Microsoft-hosted CI/CD:

- Extra parallel jobs beyond the free tier incur charges per parallel job.
- Priced at $40 per parallel job each month.

Self-hosted CI/CD:

- Additional parallel jobs beyond the free tier are subject to charges per parallel job.
- Costing $15 per parallel job monthly.

Azure Test Plans Overview and Working

Azure Test Plans constitute a test management component within Azure DevOps, enabling users to oversee test plans, test suites, and test cases across all members involved in software development. Through test plans, users can facilitate organized test management. Additionally, Azure Test Plans offers a browser extension tailored for exploratory testing and soliciting input from stakeholders.

Overview of Azure Test Plans

1. **Test Management:**
    - Facilitates the creation, organization, and tracking of test cases throughout the project lifecycle.
    - Enables teams to define test steps, expected outcomes, and associated attachments for each test case.
2. **Test Execution:**
    - Supports manual and automated test execution, capturing results and progress accurately.
    - Allows for collaboration during manual test execution and feedback collection.
3. **Test Analysis and Reporting:**
    - Provides comprehensive insights into test results, allowing teams to make informed decisions based on quality metrics and trends.

Key Features

1. **Test Case Organization:**
    - Test cases can be organized into test plans and suites, ensuring efficient management and execution.

- Test configurations and variables can be defined to support diverse testing environments.

2. **Manual Testing:**
   - Manual tests can be executed directly within Azure DevOps, facilitating result capture and stakeholder collaboration.
   - Stakeholder feedback can be efficiently incorporated during manual test execution.
3. **Automated Testing Integration:**
   - Seamlessly integrates with popular automated testing frameworks like Selenium, NUnit, and MSTest.
   - Automated tests can be executed alongside manual tests within test plans.
4. **Exploratory Testing:**
   - Supports exploratory testing sessions for defect identification.
   - Allows testers to capture observations, screenshots, and recordings during exploratory testing.
5. **Test Analytics and Reporting:**
   - Interactive dashboards and reports provide insights into test progress, pass rates, and failure patterns.
   - Enables data-driven decisions and continuous improvement of testing processes.

Azure Test Plans Workflow

1. **Test Plan Creation:**
   - Create test plans and organize test cases into suites according to project needs.
   - Define test configurations to accommodate various testing environments.
2. **Test Execution:**
   - Assign test cases to testers for manual execution and feedback collection.
   - Integrate automated tests into test plans using build and release pipelines.
3. **Analysis of Test Results:**
   - Utilize interactive dashboards and reports to analyze test trends and outcomes.
   - Prioritize critical defects for resolution based on test analytics.

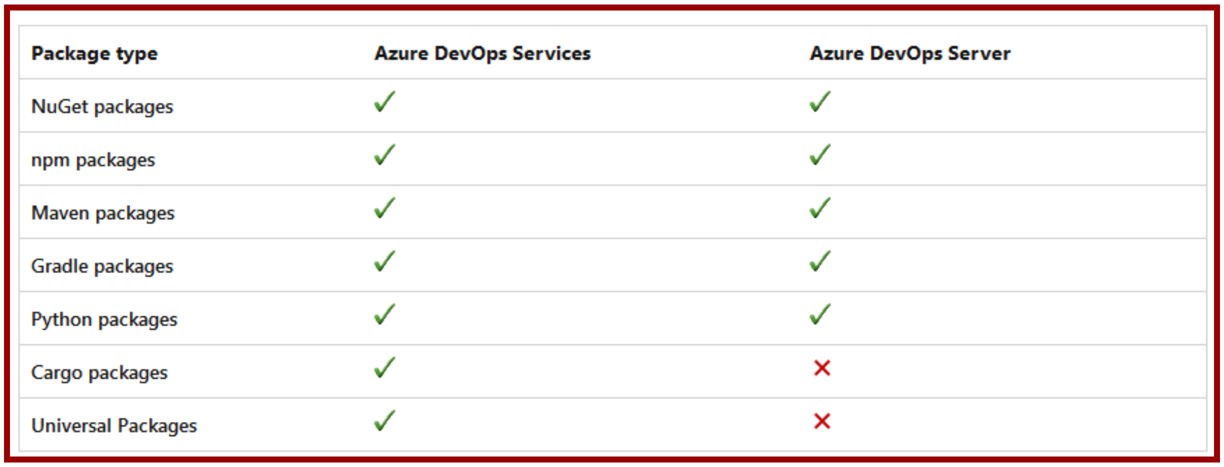
Azure Artifacts Upstream Sources

- Azure Artifacts allows developers to effectively manage all their dependencies in a single location.
- With this service, developers can publish packages to their feeds and distribute them within their team, across various organizations, or publicly on the internet.
- A notable feature of Azure Artifacts is "Upstream Sources," which lets you link your feed to both public and private repositories for enhanced dependency management.

—-Back to Index—

- Azure Artifacts offers compatibility with various package formats, encompassing NuGet, npm, Python, Maven, Cargo, and Universal Packages.

Features Availability



Package type	Azure DevOps Services	Azure DevOps Server
NuGet packages	✓	✓
npm packages	✓	✓
Maven packages	✓	✓
Gradle packages	✓	✓
Python packages	✓	✓
Cargo packages	✓	✗
Universal Packages	✓	✗

[Source: Microsoft Documentation]

Artifacts free tier and upgrade

- Azure Artifacts is available for free to every organization, offering up to 2 GiB of storage.
- Upon reaching the maximum storage limit, you'll need to either delete some of your existing artifacts or set up billing to increase your storage limit.

Upstream Sources

- Upstream sources are external repositories connected to your Azure Artifacts feed.
- These connections let you fetch packages not stored in your feed, simplifying dependency management and ensuring access to the latest packages.
- Using upstream sources in your feed enables you to manage your application dependencies from a single feed.

Advantages of Using Upstream Sources

1. **Centralized Management:** Manage all dependencies from a single feed, even if they come from multiple sources.
2. **Improved Performance and Reliability:** Packages from upstream sources are cached in your feed, enhancing reliability and performance.
3. **Enhanced Control:** Gain better control and visibility over external dependencies by bringing them into your feed.
4. **Build Isolation:** Isolate your builds from external repository changes by using cached package versions.

Types of Upstream Sources

Azure Artifacts supports various upstream sources, such as:

1. **Public Feeds:** Connect to public repositories like NuGet.org, npmjs.com, and Maven Central.
2. **Azure Artifacts Feeds:** Link to other feeds within your organization or different organizations, given proper permissions.
3. **GitHub Packages:** Connect to GitHub repositories hosting packages.

Setting Up Upstream Sources

To configure upstream sources in Azure Artifacts, follow these steps:

1. Access Your Feed:

- Open the Azure DevOps portal.
- Select your project and navigate to Artifacts.
- Choose the feed you want to configure.

2. Configure Upstream Sources:

- In the feed settings, find the "Upstream sources" section.
- Add a new upstream source by selecting the repository type.
- Provide the URL for public repositories, or credentials/tokens for Azure Artifacts or GitHub Packages.

3. Integrate the Feed into Your Projects:

- Set up your project to use the feed. For example, in a .NET project, add the feed URL to your NuGet.config file.
- Your package manager (NuGet, npm, Maven, etc.) will now access packages from upstream sources through your Azure Artifacts feed.

Best Practices - Package Consumers:

- **Use a single feed in your configuration file:** To ensure your feed provides a deterministic restore, ensure that your configuration file (such as nuget. config or .npmrc) references only one feed with upstream sources enabled.
- **Order your upstream sources intentionally:** When managing multiple sources, each upstream source is searched in the order it's listed in the feed's configuration settings. So, it’s recommended to place the public registries first in the list of upstream sources.
- **Use the suggested default view:** When you add a remote feed as an upstream source, you must select its feed's view. This allows the upstream sources to compile a set of available packages.
- **Stay Updated:** Regularly monitor the packages fetched from upstream sources and update them to benefit from the latest features, performance improvements, and security patches.
- **Automate Updates:** Consider using automation tools to check for and apply updates to packages regularly, ensuring your projects are always using the latest versions.

- **Manage Permissions:** Properly configure access permissions to your feeds and upstream sources. Ensure only authorized users and teams have access to modify or fetch packages.
- **Audit Logs:** Enable and regularly review audit logs to track changes and access to your feeds. This assists in detecting any unauthorized access or suspicious activities.
- **License Compliance:** Ensure that the packages from upstream sources comply with your organization's licensing policies. Use tools to automatically check for license compliance.
- **Security Scans:** Regularly scan packages for vulnerabilities. Use security tools to detect and mitigate risks associated with dependencies fetched from upstream sources.

Best Practices - Feed Owners/Package Publishers

- **Use the default view -** The default view for all newly created feeds is the @Local view, which contains all the packages published to your feed or saved from upstream sources.
- **Construct a package graph -** To construct a package graph, simply connect to the feed's default view and install the package you wish to share. When the package is saved to the default view, users looking to use it will be able to resolve the package graph and install the desired package.

Implement a versioning strategy in Azure Artifacts

- Software requirements and functionality evolve over time, adapting based on feedback. Original implementations often contain flaws, necessitating continuous updates.
- As software relies on various components and packages, managing their versions becomes crucial.
- A robust versioning strategy in Azure Artifacts is crucial for effective dependency management, helps maintain the codebase, track software usag and ensuring project stability.

Versioning Strategy

1. **Select a Versioning Scheme**
   - **Semantic Versioning:** Utilize the MAJOR.MINOR.PATCH format (e.g., 1.2.3).
     - **MAJOR:** Increment for incompatible API changes.
     - **MINOR:** Increment for backward-compatible new features.
     - **PATCH:** Increment for backward-compatible bug fixes.
2. **Set Up Versioning in Azure Artifacts**
   - **Automate Versioning:** Use build pipelines to automate version number increments based on changes.
   - **Version Prefixes:** Define prefixes or suffixes as needed (e.g., 1.0.0-alpha, 1.0.0-beta).
3. **Configure Your Build Pipeline**
   - **Version Increment Script:** Create scripts that update versions based on commit messages or branch names (e.g., increment patch for bug fixes, minor for new features).

- **CI/CD Integration:** Ensure your CI/CD pipeline updates package versions and publishes them to Azure Artifacts.

4. **Maintain Consistent Package Naming**

- **Standard Naming Conventions:** Ensure packages have consistent naming to prevent confusion.
- **Include Metadata:** Add metadata if necessary (e.g., build number, commit hash).

Best Practices

1. Automate Versioning

- **Use Build Pipelines:** Automate versioning in your build pipeline using scripts and versioning tools.
- **Branch-Based Versioning:** Differentiate versioning schemes for different branches (e.g., main for stable releases, develop for ongoing development).

2. Handle Pre-Release Versions

- **Pre-Release Tags:** Use tags like alpha, beta, rc (release candidate) for packages not yet production-ready.
- **Manage Release Channels:** Separate stable and pre-release versions within Azure Artifacts.

3. Manage Dependencies Carefully

- **Pin Dependencies:** Lock dependencies to specific versions to prevent unexpected changes.
- **Use Version Ranges:** Specify compatible versions with version ranges (e.g., ^1.0.0 for any compatible 1.x.x version).

4. Document and Communicate

- **Documentation:** Clearly document your versioning policies and ensure team members are familiar with them.
- **Regular Updates:** Communicate any changes in the versioning strategy to your team promptly.

5. Monitor and Audit

- **Track Package Versions:** Utilize Azure DevOps tools to monitor versions of your packages.
- **Maintain Audit Logs:** Keep detailed logs to track who published each version and when.

Use Cases for Versioning Strategy

Software Development

- **Consistent Releases:** Ensure stable and consistent software releases by incrementing version numbers appropriately.
- **Beta Testing:** Use pre-release versions (alpha, beta) to test new features without affecting the stable release.

Dependency Management

- **Internal Libraries:** Manage internal libraries effectively by tracking and updating their versions, ensuring compatibility across different projects.
- **Open Source Contributions:** For open-source projects, semantic versioning helps maintain clarity on the impact of changes for the community.

What is Source control?

A source control system, or version control system, lets developers collaborate and track code changes, essential for multi-developer projects. Our systems support Git (distributed) and Team Foundation Version Control (TFVC) (centralized, client-server). Both Git and TFVC enable checking in files and organizing them into folders, branches, and repositories. Manage repositories, branches, and development tasks with Azure Repos.

Features:

- With Git, every developer has a complete copy of the source repository on their local machine, including all branches and history.
- Developers work directly with their local repositories and share changes with others as a separate step.
- Version control operations, such as committing changes, viewing history, and comparing versions, can be performed offline without needing a network connection.
- Branches in Git are lightweight, allowing developers to create private local branches for different tasks or contexts.
- Developers can quickly switch between branches to work on various code base variations.
- Branches can be easily merged, published, or deleted when no longer needed.
- In TFVC, developers have a single version of each file on their local machines.
- Historical data is stored exclusively on the server.
- Branches are path-based and must be created on the server.

Authentication Strategy in Azure DevOps

Authentication in Azure DevOps

- **Purpose:** Verifies account identity based on provided credentials.
- **Integration:** Works with security features from:
    - Microsoft Entra ID
    - Microsoft account (MSA)
    - Active Directory (AD)
- **Cloud Authentication:**
    - **Microsoft Entra ID:** Recommended for managing large user groups.
    - **Microsoft Accounts (MSA):** Suitable for small user bases.

- **On-Premises Deployments:**
  - **Active Directory (AD):** Recommended for managing large user groups.

Authentication Methods and Integration in Azure DevOps

- **Integration with Apps and Services:**
  - Other apps and services can access Azure DevOps without requiring multiple logins.
- **Authentication Methods:**

  Personal Access Tokens (PATs):
  - Generate tokens for accessing specific resources or activities.
  - Used by clients like Xcode and NuGet that need basic credentials.
  - Access Azure DevOps REST APIs.

  Azure DevOps OAuth:
  - Generate tokens for users to access REST APIs.
  - Supported by Accounts and Profiles APIs.

  SSH Authentication:
  - Generate encryption keys for Linux, macOS, or Windows running Git for Windows.
  - Used when Git credential managers or PATs for HTTPS aren't available.

  Service Principals or Managed Identities:
  - Generate Microsoft Entra tokens for applications or services automating workflows.
  - Can replace service accounts and PATs for many actions.
- **Default Access:**
  - All authentication methods are allowed by default.
  - Restrict access by denying specific methods if needed.
  - Denying a method results in authentication errors for any app using that method.

Create a project wiki to share information - Azure DevOps

1. **Pre-requisites:**
   - Set up a team project in Azure DevOps.
   - Ensure you have Basic access to modify a wiki.
   - Obtain permission to create a repository (usually granted to Project Administrators).
2. **Open the Wiki:**
   - Connect to your project using a supported web browser.
   - Navigate to the Repos hub and select “Wiki.”
   - Switch to your desired team project if necessary.

3. Create a Git Repository

- On the wiki landing page, choose “Create Project wiki.”
- Even if you use TFVC for source control, you can create a wiki with a Git repository.
- If you lack permission to create a wiki Git repository, request an administrator to grant access.
- The wiki Git repo follows the naming convention: TeamProjectName.wiki (e.g., “foobar.wiki” for project “foobar”).
- Multiple wiki repos can be set up within a single project.

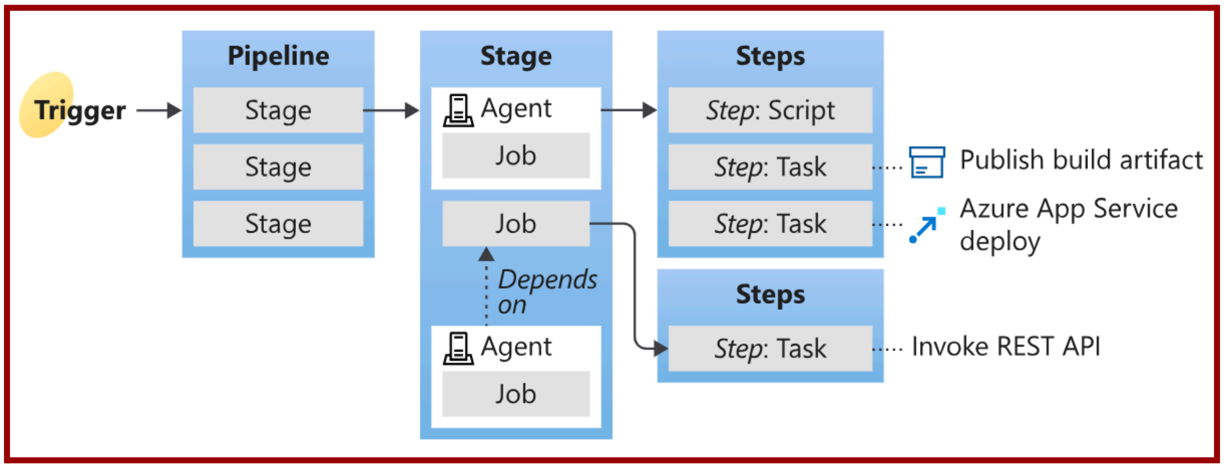
—-Back to Index—

Design and implement deployments

Azure Pipelines Overview, Stages, and Deployment Jobs

- Azure Pipelines stands as a cloud-based service offered within the Microsoft Azure ecosystem, designed to streamline continuous integration (CI) and continuous delivery (CD) processes.
- Its primary function revolves around automating various stages of application development, including building, testing, and deployment.
- Azure Pipelines caters to a wide range of programming languages, platforms, and CI/CD scenarios, ensuring adaptability and scalability to suit diverse development requirements.

Azure Pipeline Structure

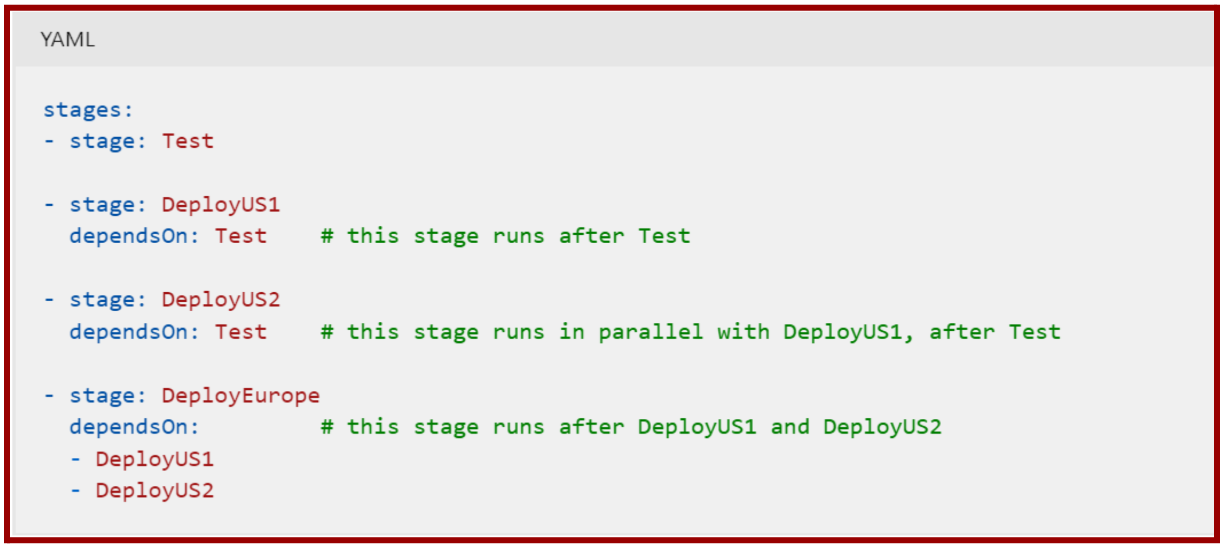


[Source: Microsoft Documentation]

- A **trigger** commences the execution of a pipeline.
- A **pipeline** is composed of one or more stages and has the capability to deploy to multiple environments.
- **Stages** serve as organizational units within the pipeline, accommodating one or more jobs.
- Each **job** runs on a designated agent, although it can also be agentless.
- Agents execute jobs, which consist of one or more steps.
- **Steps** are the fundamental elements of the pipeline and can be tasks or scripts.
- **Tasks** are prepackaged scripts that perform specific actions, such as invoking a REST API or publishing a build artifact.
- **Artifacts** are collections of files or packages generated and published by a pipeline run.

Pipeline Stages

- Stages in Azure Pipelines are logical divisions within a pipeline that facilitate the organization and sequential execution of tasks.
- Each stage represents a distinct phase in the software development lifecycle (for example, build the app, run tests, deploy to preproduction) and allows for the modularization of pipeline workflows.
- Stages enable developers to break down complex processes into smaller, manageable units, providing better control and visibility throughout the pipeline execution process.
- In a pipeline, if you specify multiple stages, they typically execute sequentially by default. However, stages can also be configured to have dependencies on each other using the "dependsOn" keyword.
- Additionally, stages can be set to run based on the outcome of a preceding stage, utilizing conditions for execution.



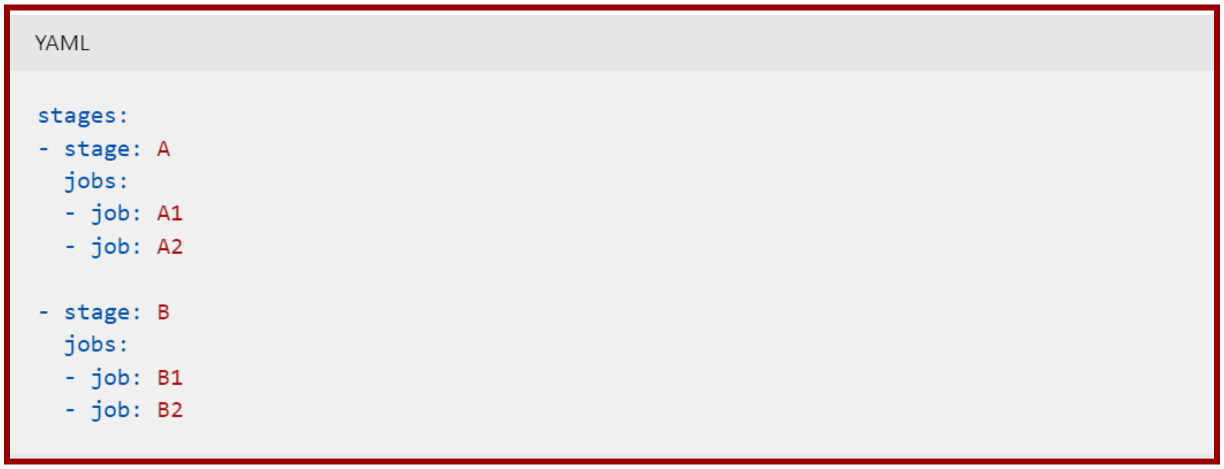
YAML

```yaml
stages:
- stage: Test

- stage: DeployUS1
  dependsOn: Test    # this stage runs after Test

- stage: DeployUS2
  dependsOn: Test    # this stage runs in parallel with DeployUS1, after Test

- stage: DeployEurope
  dependsOn:         # this stage runs after DeployUS1 and DeployUS2
  - DeployUS1
  - DeployUS2
```

[Source: Microsoft Documentation]

Deployment jobs

- Deployment tasks within Azure Pipelines are specialized operations tailored to deploying application artifacts to designated target environments.
- These tasks automate deployment processes, ensuring consistent and reliable application releases across various environments.
- Deployment tasks encompass a range of activities, including file copying, script execution, infrastructure configuration, and deployment validation.
- It's possible for both a deployment job and a traditional job to coexist within the same stage.
- In a deployment job, the source repository is not automatically cloned. You can incorporate the checkout of the source repository within your job using ***checkout: self.***
- Azure DevOps supports the *runOnce*, *rolling*, and the *canary* strategies.



YAML

```yaml
stages:
- stage: A
  jobs:
  - job: A1
  - job: A2

- stage: B
  jobs:
  - job: B1
  - job: B2
```

[Source: Microsoft Documentation]

Deployment jobs offer the following advantages:

1. **Deployment Tracking:** They provide a comprehensive deployment history across pipelines, offering insights into specific resources and deployment statuses for auditing purposes.
2. **Custom Deployment Strategies:** You can specify the deployment strategy to govern how your application is progressively rolled out.

Using containerized services in Azure Pipelines

- Each pipeline job may require one or more supporting services, examples include databases or memory caches needed for integration tests.
- These services should be created, connected, and cleaned up specifically for each job to ensure a fresh environment.
- Containers offer a straightforward and portable solution to run the required services for your pipeline, using containers simplifies the process of managing dependencies and service lifecycle.
- Service containers are a specialized type of container used to run services that a job in your pipeline depends on. They allow for the automatic creation, networking, and management of the service's lifecycle.
- These containers ensure that each service is accessible only to the job that needs it, enhancing security and isolation.

Benefits of Containerized Services in Azure Pipelines

- **Uniform Environments**: Containers create consistent environments across different stages—development, testing, and production—minimizing discrepancies.
- **Isolated Jobs**: Each pipeline job can run in its own isolated container, avoiding conflicts between dependencies and configurations.
- **Easy Portability**: Containers bundle application code with its dependencies, making it easy to deploy across various environments without changes.

Best Practices

- **Version Tags:** Avoid using the latest tag for Docker images in production. Instead, use specific tags to ensure predictable builds.
- **Secure Configurations:** Use Azure DevOps service connections or Azure Key Vault to securely manage credentials and connections.
- **Clean-Up Resources:** Ensure that containers and other resources are properly cleaned up after builds to avoid unnecessary costs and resource usage.

GitHub runner or Azure DevOps agent Overview

In CI/CD, GitHub Runners and Azure DevOps Agents are pivotal components for automating workflows. These agents execute tasks such as building, testing, and deploying software. Understanding their functionalities is essential for effectively managing your development pipelines.

GitHub Runner

GitHub Runner is the engine behind GitHub Actions, enabling the execution of workflows defined in GitHub repositories. Here's a closure looks at its features and capabilities:

- **Functionality:**
    - Executes the steps defined in GitHub Actions workflows.
    - Suitable for a range of tasks including code building, testing, and deployment automation.
- **Runner Types:**
    - **GitHub-Hosted Runners:** Managed by GitHub with pre-configured environments for Linux, Windows, and macOS.
    - **Self-Hosted Runners:** Deployed and managed by the user on personal hardware or cloud instances.
- **Configuration:**
    - GitHub-hosted runners come ready with popular tools and dependencies.
    - Self-hosted runners can be set up on any system, allowing for the installation of specific software needed for your workflows.
- **Workflow Usage:**
    - Define workflows in YAML files in the directory of your repository.
    - Specify which runner to use for each job within these YAML files.
- **Security and Control:**
    - GitHub-hosted runners provide secure, isolated environments but limited control.
    - Self-hosted runners offer full control, which is ideal for sensitive data and custom configurations.

Azure DevOps Agent

Azure DevOps Agent is part of Azure Pipelines and is used to run jobs for CI/CD workflows. Here's a closer look at its features:

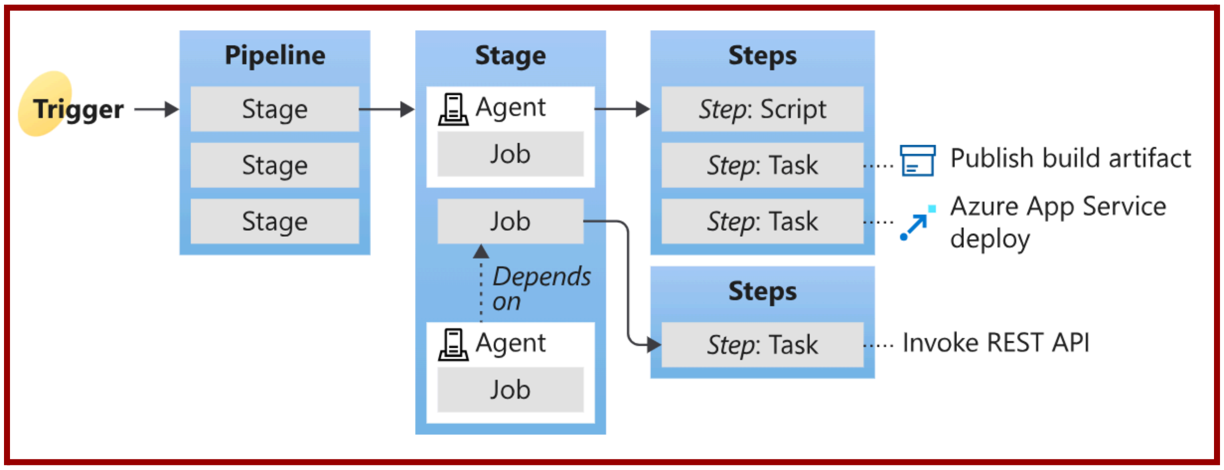
1. **Functionality:**
    - Executes tasks defined in Azure Pipelines, supporting builds, tests, and deployments.
    - Supports a broad array of languages and platforms.
2. **Agent Types:**
    - **Microsoft-Hosted Agents:** Provided by Azure with predefined environments.
    - **Self-Hosted Agents:** Managed by the user, deployed on custom hardware or cloud VMs.
3. **Configuration:**
    - Microsoft-hosted agents come pre-configured with essential tools and libraries.
    - Self-hosted agents offer the flexibility to install any required software on the agent machine.
4. **Pipeline Usage:**
    - Define pipelines using YAML files or the visual designer in Azure DevOps.
    - Specify the agent pool and type in the pipeline configuration.
5. **Security and Control:**
    - Microsoft-hosted agents offer secure and managed environments with limited customization.
    - Self-hosted agents provide extensive control, suitable for specific security requirements and custom performance tuning.

Azure Pipelines by using YAML

- Azure Pipelines facilitates continuous integration (CI) and continuous delivery (CD), enabling the continuous testing, building, and deployment of your code.To achieve this, you create a pipeline.
- Azure Pipelines offers a YAML-based editor for building and modifying your pipelines.
- This editor, built on the Monaco Editor framework, features helpful tools such as Intellisense and a task assistant to guide you through the process of editing a pipeline.
- YAML pipelines can also be edit by modifying the azure-pipelines.yml file directly in your pipeline's repository using a text editor of your choice, or by using a tool like Visual Studio Code and the Azure Pipelines for VS Code extension.

—-Back to Index—

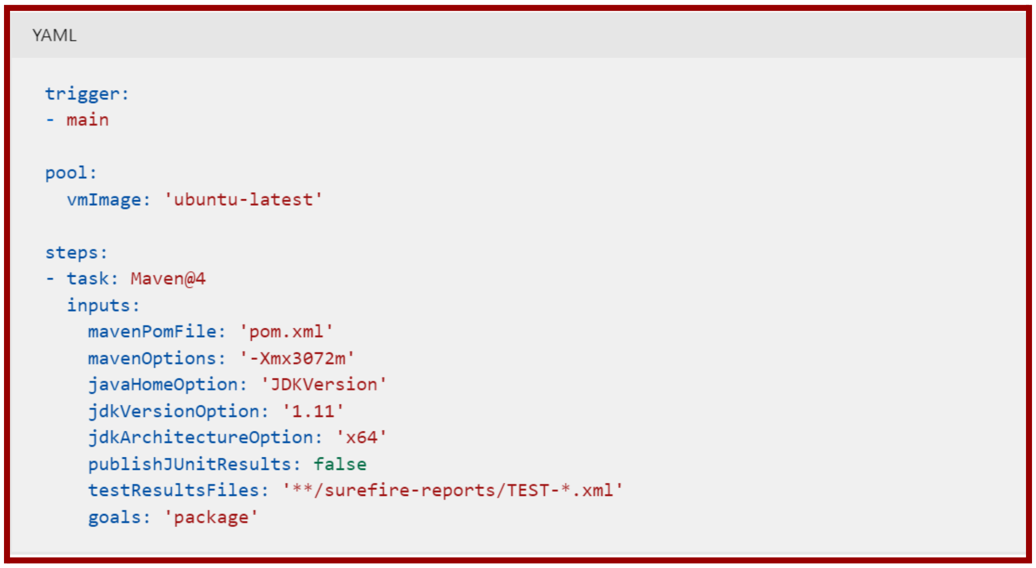
Key Components of Azure Pipelines YAML



[Source: Microsoft Documentation]

- A trigger initiates a pipeline to execute.
- A pipeline consists of one or more stages and can deploy to multiple environments.
- Stages organize jobs within a pipeline, with each stage containing one or more jobs.
- Each job is executed on an agent, although jobs can also be agentless.
- An agent runs a job comprised of one or more steps.
- Steps, which can be either tasks or scripts, are the smallest units in a pipeline.
- A task is a predefined script designed to carry out specific actions, such as calling a REST API or publishing a build artifact.
- An artifact is a collection of files or packages generated by a pipeline run.

Yaml Pipeline Sample



YAML

```yaml
trigger:
- main

pool:
  vmImage: 'ubuntu-latest'

steps:
- task: Maven@4
  inputs:
    mavenPomFile: 'pom.xml'
    mavenOptions: '-Xmx3072m'
    javaHomeOption: 'JDKVersion'
    jdkVersionOption: '1.11'
    jdkArchitectureOption: 'x64'
    publishJUnitResults: false
    testResultsFiles: '**/surefire-reports/TEST-*.xml'
    goals: 'package'
```

[Source: Microsoft Documentation]

Advantages of Using YAML

Version Control: Storing YAML files in source control allows for versioning and tracking changes.
Consistency: Using templates and reusable components ensures uniformity across various pipelines.
Clarity: Clearly defined pipeline structures improve team collaboration and understanding.
Flexibility: YAML's flexibility helps in defining complex workflows and conditions.

Azure Pipelines Agents, Triggers and Variables

Azure Pipelines Agents

Agents are crucial for executing the tasks defined in your CI/CD pipelines. They provide the computing infrastructure with specific capabilities, such as operating system types, software, tools, and other dependencies necessary to run pipeline jobs.

Types of Agents

1. **Microsoft-hosted Agents:**
    - **Managed by Microsoft:** These agents are maintained and updated by Microsoft, ensuring they have the latest tools and software.
    - **Quick Setup:** No need to manage or update the agents yourself.
    - **Image Options:** Available in various pre-configured images, including different versions of Windows, Ubuntu, and macOS.
    - **Usage Costs:** Typically included with your Azure DevOps subscription, but with usage limits.
2. **Self-hosted Agents:**
    - **Managed by You:** Full control over the agent machines, including installed software and configurations.
    - **Custom Configuration:** Ideal for specific requirements or software not available on Microsoft-hosted agents.
    - **Cost-Effective:** Potentially more economical for extensive usage since they run on your infrastructure.
    - **Flexibility:** Can run on virtual machines, physical servers, or containers.

Managing Agent Capabilities

Agents report their capabilities, which include the operating system, installed software, and tools. These capabilities can be used to match jobs to appropriate agents.

1. **Automatic Capabilities**: Automatically detected by the agent (e.g., OS type, version).
2. **User-defined Capabilities**: Custom capabilities defined by the user to indicate specific tools or software available on the agent.

Agent Pools and Parallelism

Each Azure DevOps organization has a certain number of parallel jobs (pipelines that can run simultaneously) based on your license and purchased parallelism. Both Microsoft-hosted and self-hosted agents contribute to this parallelism.

- **Microsoft-hosted Agents**: Limited by your Azure DevOps plan.
- **Self-hosted Agents**: Unlimited, but only constrained by your own hardware resources.

Best Practices for Using Azure Pipelines Agents

1. **Choose the Right Agent Type**: Use Microsoft-hosted agents for quick setup and standard builds, and self-hosted agents for custom environments or high-control scenarios.
2. **Optimize Agent Pools**: Create separate pools for different purposes (e.g., production vs. development builds) to manage resources effectively.
3. **Use Demands and Capabilities**: Ensure jobs are matched with appropriate agents by specifying demands and managing agent capabilities.
4. **Maintain Self-hosted Agents**: Regularly update and maintain self-hosted agents to ensure they have the necessary software and security patches.
5. **Monitor Agent Utilization**: Use Azure DevOps monitoring tools to track agent utilization and optimize the number of agents and pools as needed.
6. **Scale Self-hosted Agents**: Use scaling strategies (e.g., autoscaling with VMs or containers) to handle varying workloads efficiently.

Azure Pipeline Triggers

Azure Pipeline triggers are configurations that determine when a pipeline should be executed. These triggers automate the process of running builds and deployments in response to various events, such as code changes, pull requests, or scheduled times. Understanding and configuring these triggers appropriately can greatly enhance the efficiency and reliability of your CI/CD processes.

Types of Triggers

1. Continuous Integration (CI) Triggers
2. Pull Request (PR) Triggers
3. Scheduled Triggers
4. Pipeline Completion Triggers

Continuous Integration (CI) Triggers

CI triggers automatically run the pipeline when changes are pushed to the specified branches in your repository. This ensures that your code is constantly integrated and tested.

Below pipeline triggers on changes to the main branch or branches inside the releases folder:

```yaml
trigger:
- main
- releases/*
```

[Source: Microsoft Documentation]

Advanced Configuration with Include and Exclude:

```yaml
# specific branch build
trigger:
  branches:
    include:
    - main
    - releases/*
    exclude:
    - releases/old*
```

[Source: Microsoft Documentation]

- include: Specifies which branches should trigger the pipeline.
- exclude Specifies which branches should be ignored.

You can also configure path filters to run the pipeline only when specific files or directories are changed.

```yaml
# specific path build
trigger:
  branches:
    include:
    - main
    - releases/*
  paths:
    include:
    - docs
    exclude:
    - docs/README.md
```

[Source: Microsoft Documentation]

Use Cases:

- Automated Builds on Code Updates
- Branch-specific Workflows

- Selective Builds with Path Filters

Pull Request (PR) Triggers

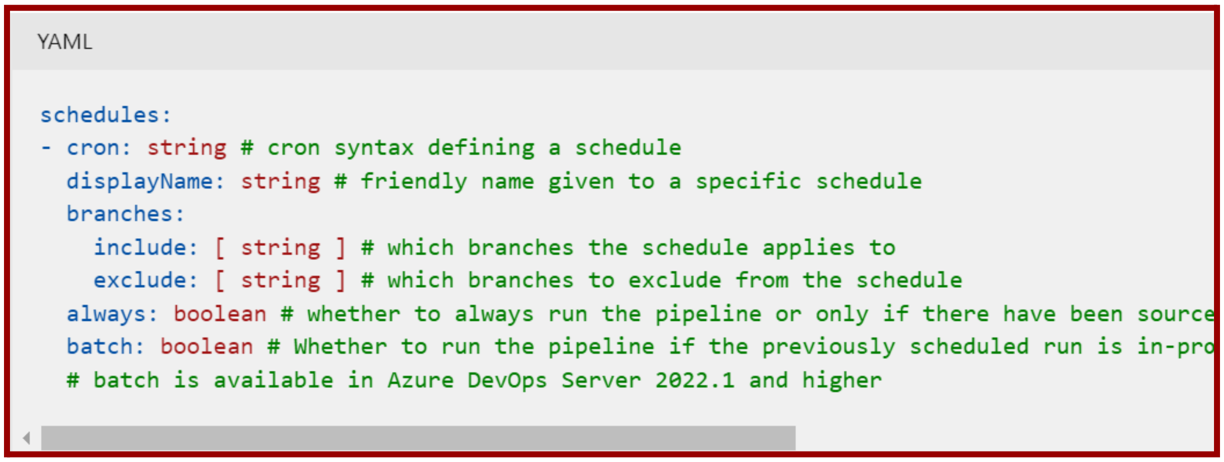
- PR triggers automatically run the pipeline when a pull request is created or updated, ensuring that the changes proposed in the pull request are tested before merging.
- To enable PR validation, access the branch policies for the selected branch, and set up the Build validation policy specifically for that branch.
- When you have an active PR and you make updates to its source branch, it can trigger multiple pipelines to run.

Use Cases

- **Automated Testing for Pull Requests:** Ensure every pull request undergoes testing before merging.
- **Maintaining Code Quality Standards:** Enforce linting and style checks on all pull requests.
- **Selective PR Builds with Path Filters:** Execute specific tests based on the files modified in the pull request.

Scheduled Triggers

Scheduled triggers run the pipeline at specified times, regardless of code changes. This is useful for regular tasks such as nightly builds or periodic deployments.



YAML

```yaml
schedules:
- cron: string # cron syntax defining a schedule
  displayName: string # friendly name given to a specific schedule
  branches:
    include: [ string ] # which branches the schedule applies to
    exclude: [ string ] # which branches to exclude from the schedule
  always: boolean # whether to always run the pipeline or only if there have been source
  batch: boolean # Whether to run the pipeline if the previously scheduled run is in-pro
  # batch is available in Azure DevOps Server 2022.1 and higher
```

[Source: Microsoft Documentation]

Scheduled YAML pipelines have certain limitations:

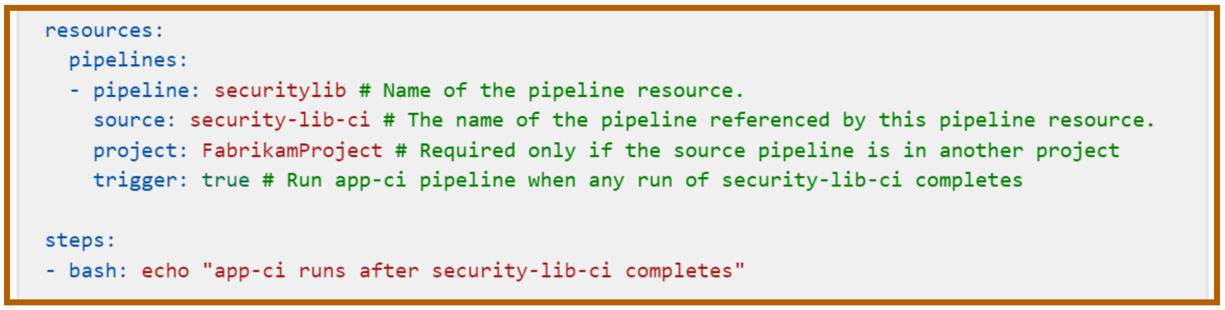
- The cron schedules operate in the UTC time zone.
- When excluding branches without including any, it's akin to including all branches with "*".
- Pipeline variables cannot be utilized when defining schedules.
- When using templates in a YAML file, it's important to define schedules in the primary YAML file rather than in the template files.

Use Cases:

- **Nightly Builds**: Conduct nightly builds to verify the stability of the latest codebase.
- **Weekly Integration Tests**: Perform comprehensive integration tests weekly.
- **Routine Security Scans**: Carry out regular security scans to detect vulnerabilities.

Pipeline Completion Triggers

- Pipeline completion triggers start a pipeline after the successful completion of another pipeline.
- This is useful for orchestrating complex workflows that span multiple pipelines.



```yaml
resources:
  pipelines:
  - pipeline: securitylib # Name of the pipeline resource.
    source: security-lib-ci # The name of the pipeline referenced by this pipeline resource.
    project: FabrikamProject # Required only if the source pipeline is in another project
    trigger: true # Run app-ci pipeline when any run of security-lib-ci completes

steps:
- bash: echo "app-ci runs after security-lib-ci completes"
```

[Source: Microsoft Documentation]

- resources.pipelines: Defines the upstream pipeline.
- trigger: Enables the pipeline completion trigger.

Use Cases:

- Chained Pipelines: Trigger a pipeline automatically after another pipeline completes.
- Environment-specific Deployments: Deploy to various environments following the completion of previous deployment stages.
- Complex Workflow Orchestration: Manage multiple pipelines for different microservices, with each one triggering upon the completion of another.

Azure Pipeline Variables

Azure Pipeline variables are essential for customizing and configuring pipelines, allowing you to store and reuse values throughout your pipeline execution. These variables can be defined at multiple levels and have different scopes, providing flexibility and control over your CI/CD workflows.

Types of Variables

1. Pipeline Variables
2. Variable Groups
3. Predefined (System) Variables
4. Secrets and Secure Files

Pipeline Variables

- Pipeline variables are user-defined variables that can be set directly within the YAML file or through the Azure DevOps portal.

- These variables can be used to store values that change frequently or need to be shared across multiple jobs and steps.

Variable Groups

- Variable groups are collections of variables that can be managed and reused across multiple pipelines.
- They are particularly useful for sharing common variables such as database connection strings or API keys.

Predefined (System) Variables

- Predefined variables are provided by Azure Pipelines and contain information about the system, the pipeline, the agent running the job, and other relevant data.
- These variables are readily accessible within your pipeline.

  Examples of System Variables:
  - Build.BuildId: represents the distinct identifier assigned to the ongoing build process.
  - Build.SourceBranch: The branch being built.
  - Agent.OS: The operating system of the agent.

Secrets and Secure Files

- Secrets and secure files allow you to store sensitive information securely in Azure Pipelines. These values are encrypted and not visible in logs or output.
- Secrets can be defined in the Azure DevOps portal and used in pipelines as regular variables.

Common Use Cases

- **Storing Environment-specific Configurations:**
  - Define variables for different environments (e.g., development, staging, production) to configure your application accordingly.
- **Managing Sensitive Information:**
  - Use secret variables to store sensitive data like passwords, tokens, and connection strings securely.
- **Dynamic Pipeline Behavior:**
  - Use variables to control pipeline flow, such as specifying different build configurations or deployment targets based on the branch or environment.

Pipeline deployment approvals - Azure DevOps

Deployment approvals are a critical feature in Azure DevOps, designed to control and manage the release process. This feature ensures that deployments to production or other sensitive environments are reviewed and authorized by designated individuals or groups, maintaining high levels of security and compliance within the CI/CD pipeline.

Key Components of Deployment Approvals

1. Pre-deployment Approvals
2. Post-deployment Approvals
3. Approval Policies
4. Manual Intervention

Pre-deployment Approvals

Pre-deployment approvals must be granted before a deployment can begin, ensuring that it is reviewed by authorized personnel.

- **Use Case:**
    - Require approval from a QA team lead before deploying to the staging environment.
- **Configuration:** Add an approval check in the release pipeline for environments requiring pre-deployment approval.

Post-deployment Approvals

Post-deployment approvals are necessary after a deployment has completed but before it is marked as successful, allowing for validation and testing.

- **Use Case:**
    - Require sign-off from a product manager after the application is deployed to production.
- **Configuration:** Add an approval check in the release pipeline for environments requiring post-deployment approval.

Approval Policies

Approval policies define the rules and conditions under which approvals are needed. They specify who can approve the deployment and any additional criteria.

- **Use Case:**
    - Require that at least two team members from the operations team approve the deployment to production.
- **Configuration:** Configure the approval policy in the release pipeline settings, specifying the approvers and any additional conditions.

Manual Intervention

Manual intervention steps allow for human interaction within the pipeline, useful for scenarios requiring manual checks or actions during the deployment process.

- **Use Case:**
    - Pause the deployment process for manual testing or validation steps.
- **Configuration:** Add a manual intervention step in the pipeline, specifying the required action and who can perform it.

Best Practices

- **Define Clear Approval Policies:** Clearly document and define approval policies, specifying who has the authority to approve deployments and under what conditions.
- **Use Multiple Approvers:** For critical environments, require multiple approvers to reduce the risk of unauthorized deployments.
- **Regularly Review Approval Policies:** Periodically review and update approval policies to reflect changes in team structure or project requirements.
- **Integrate Manual Intervention Steps:** Include manual intervention steps for scenarios that require human oversight or interaction to ensure all necessary checks are performed.

Azure Artifacts overview

- Azure Artifacts allows developers to manage all dependencies in one central location.
- Developers can publish packages to their feeds and share them within their team, across organizations, or publicly on the internet.
- Azure Artifacts enables consuming packages from various feeds and public registries like NuGet.org and npmjs.com.
- Supported package types in Azure Artifacts include NuGet, npm, Python, Maven, Cargo, and Universal Packages.

Azure Artifacts Storage Plans

- **Free Storage:** Azure Artifacts offers 2 GiB of free storage for every organization.
- **Storage Limit Reached:** When the storage limit is reached, publishing new artifacts will be blocked.
- **Continuing Publishing:** To continue publishing, either delete some existing artifacts or set up billing to increase storage capacity.
- **Cost Information:** For detailed information on costs, use the Azure DevOps Pricing Calculator.

Access Organization Billing Settings

- **Sign in:** Log in to your Azure DevOps organization account.
- **Open Settings:** Click the gear icon for Organization settings.
- **Select Billing:** Choose the Billing option from the settings menu.
- **Check Usage:** View your Artifacts tier and current usage limit.

Get Started with Azure Artifacts

Azure Artifacts allows you to publish and consume various types of packages. Choose your package type to begin:

- Publish and consume different types of packages using Azure Artifacts.
- To start, select the package type you want to work with.

—-Back to Index—

Securing Azure Pipelines

Security Considerations in Azure Pipelines

- **Unique Challenges:**
    - Azure Pipelines presents specific security concerns due to its capability to execute scripts and deploy code to production environments.
- **Preventing Malicious Code:**
    - It's crucial to prevent pipelines from being exploited to execute malicious code.
- **Code Deployment Control:**
    - Ensuring only authorized and intended code is deployed is essential for maintaining security and reliability.
- **Balancing Security and Flexibility:**
    - Security measures must strike a balance between providing necessary safeguards and empowering teams with the flexibility to run their pipelines effectively.

Security Considerations in Azure Pipelines with YAML

- **Traditional Security Approaches:**
    - In the past, organizations often enforced strict security measures, limiting access and use of code, pipelines, and production environments. While manageable in small organizations, this approach becomes challenging in larger organizations with numerous users and projects.
- **Assuming Breach:**
    - Larger organizations must adopt an "assume breach" mindset, considering the possibility of adversaries gaining contributor access to repositories. The focus shifts towards preventing adversaries from running malicious code in pipelines, which could lead to secrets theft or production environment corruption.
- **YAML Pipelines for Security:**
    - YAML pipelines offer superior security compared to classic build and release pipelines. They can undergo code reviews like any other code, allowing prevention of malicious steps via Pull Requests enforced by branch policies.
- **Resource Access Management:**
    - Resource owners have control over whether a YAML pipeline can access a resource, reducing the risk of attacks like repository theft. Approvals and checks provide access control for each pipeline run.
- **Runtime Parameters:**

        - Runtime parameters help mitigate security issues related to variables, such as Argument Injection, enhancing overall pipeline security.
- **Guidance for Secure CI/CD Pipelines:**
        - A series of articles outlines recommendations for constructing secure YAML-based CI/CD pipelines, addressing trade-offs between security and flexibility. Familiarity with Azure Pipelines, Azure DevOps security constructs, and Git is assumed.

Configuring for UI testing - Azure Pipelines

UI testing considerations

- Automated tests in the CI/CD pipeline often require specific setup to execute UI tests like Selenium, Appium, or Coded UI tests.
- Selenium tests for web applications offer two browser launch options.
- The first option involves running the browser in headless mode, where the browser operates without visible UI components. This mode is ideal for unattended testing in CI/CD pipelines.
- Alternatively, tests can be executed in visible UI mode, where the browser functions normally with UI components visible. This mode may require special configuration, particularly on Windows environments.
- The choice between headless and visible UI modes depends on the testing requirements and environment configurations.

Running UI Tests in CI/CD Pipelines with Selenium

- **Headless Mode:**
    - Browsers like Chrome and Firefox can be run without UI components visible, useful for unattended automated tests in CI/CD pipelines.
    - Consumes fewer resources and speeds up test execution, enabling more tests to run in parallel on the same machine.
- **Visible UI Mode:**
    - In this mode, browsers run normally with UI components visible, requiring special agent configuration for Windows-based tests.
    - Microsoft-hosted agents come preconfigured for UI testing with popular browsers and web-driver versions.
    - Self-hosted Windows agents need special configuration to run UI tests, including selecting 'No' to run as a service and enabling autologon for interactive process execution.
- **Microsoft-hosted Agents:**
    - Preconfigured for UI testing with updated browsers and web-drivers for Selenium tests of both web and desktop apps.

- **Self-hosted Windows Agents:**
    - Can run Selenium tests with headless browsers by default.
    - For visible UI testing, must be configured as an interactive process with autologon enabled, bypassing the need for service configuration.

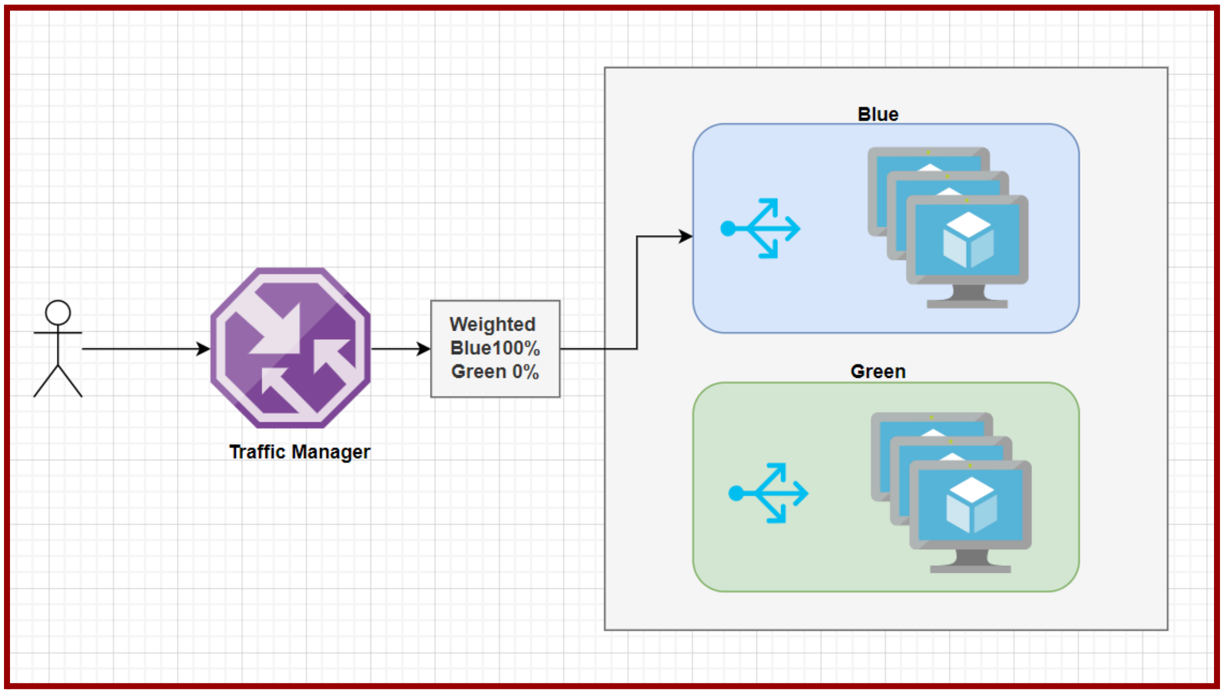
Maintain pipelines

Blue-Green deployments using Azure Traffic Manager

Azure Traffic Manager

Azure Traffic Manager is a powerful DNS-based load balancing service offered by Microsoft Azure. It caters to multiple needs, such as directing a worldwide user base to the Azure endpoints that offer the best performance and lowest latency. Additionally, it ensures seamless auto-failover for essential operations and supports the transition from on-premises infrastructure to the cloud. A notable application of Traffic Manager is in facilitating software deployments with minimal disruption through the Blue-Green deployment strategy, utilizing its weighted round-robin routing capabilities.

Blue-green deployment is a release management strategy that minimizes downtime and risk by running two identical production environments: Blue and Green. Azure Traffic Manager (ATM) supports this approach by facilitating seamless traffic routing between these environments.



[Source: Microsoft Documentation]

Key Components of Blue-Green Deployments

- Two Identical Environments (Blue and Green)
- Azure Traffic Manager for Traffic Routing
- Swap and Rollback Strategy

Steps to Implement Blue-Green Deployments Using Azure Traffic Manager

Step 1: Set Up Two Identical Environments

- Deploy your application to two distinct environments, Blue and Green. Initially, Blue serves all user traffic, while Green is on standby.

Step 2: Configure Azure Traffic Manager

- **Create a Traffic Manager profile** in the Azure portal to manage traffic flow between the Blue and Green environments.
- **To create the profile**: Navigate to "Create a resource," search for "Traffic Manager profile," set a unique DNS name, and choose the "Priority" routing method.

Step 3: Add Environments as Endpoints

- **Add both environments** to the Traffic Manager profile.
- **To add endpoints**: Go to the "Endpoints" section within the Traffic Manager profile, and set Blue as the primary endpoint and Green as the secondary.

Step 4: Deploy to the Green Environment

- Deploy the latest application version to the Green environment to prepare it for testing without affecting live traffic.

Step 5: Test the Green Environment

- Conduct comprehensive tests in the Green environment to validate the new version's performance and functionality.

Step 6: Switch Traffic to the Green Environment

- **Switch user traffic** to the Green environment after successful testing.
- **To switch traffic**: Adjust the Traffic Manager profile to make Green the primary endpoint and Blue the secondary. Monitor the transition and system performance closely.

Step 7: Roll Back if Needed

- **Revert traffic back to Blue** if issues arise with the Green environment.
- **To roll back**: Update the Traffic Manager profile to restore Blue as the primary endpoint and Green as the secondary.

Step 8: Prepare for Future Deployments

- **Ready the Blue environment** for the next deployment cycle once Green is stable and rollback is unnecessary.

Best Practices

- **Automate Testing:** Automate as much testing as possible to ensure the new environment is thoroughly validated before switching traffic.
- **Monitor Performance:** Continuously monitor both environments for performance and errors during and after the traffic switch.
- **Gradual Traffic Shift:** Consider a gradual traffic shift to the new environment using a weighted traffic routing method to minimize risk.
- **Clear Rollback Plan:** Have a clear and tested rollback plan in place in case any issues arise with the new deployment.

Use Cases

- **Zero-Downtime Updates:** Seamlessly switch traffic between Blue and Green environments for website updates without downtime.
- **Continuous Integration/Continuous Deployment (CI/CD):** Deploy new code continuously to Green while Blue remains stable, ensuring thorough testing before promoting changes to production.
- **A/B Testing:** Conduct real-time comparison of website versions by routing traffic to different environments using Traffic Manager.
- **Disaster Recovery:** Maintain identical environments in separate regions for failover capabilities, ensuring business continuity.
- **Seasonal Traffic Peaks:** Scale up Green to handle increased demand during peak periods while serving traffic from Blue.
- **Canary Releases:** Gradually roll out new features to a subset of users for feedback and risk mitigation.

Swap or switch deployments in Azure Cloud Services

- Swapping or switching deployments in Azure Cloud Services is a crucial feature that enables seamless deployment updates without downtime.
- This capability allows you to deploy a new version of your application to a staging environment and then switch it with the production environment.
- Swapping deployments can be executed via an Azure Resource Manager template (ARM template), the Azure portal, or the REST API.
- It is not possible to switch between an Azure Cloud Services (classic) deployment and an Azure Cloud Services (extended support) deployment.

Steps to Swap Deployments in Azure Cloud Services:

Step 1: Deploy to the Staging Environment

Deploy the new version of your application to the staging environment.

Step 2: Test and Validate

Thoroughly test and validate the new deployment in the staging environment to ensure it functions as expected.

Step 3: Swap Deployments

Initiate the swap process to switch the staging deployment with the production deployment.

Step 4: Monitor and Verify

Monitor the swap operation and verify that the production environment now reflects the new deployment.

Step 5: Rollback if Necessary

If any issues arise after the swap, you can quickly rollback to the previous deployment by initiating another swap operation.

Benefits:

- **Zero-Downtime Deployment:** Users experience no interruption during the deployment process.
- **Risk Mitigation:** Easy rollback capability ensures minimal impact in case of issues.
- **Configuration Management:** Maintain separate configurations for staging and production environments.

Using a hotfix production environment - Azure Data Factory

- If you deploy a factory to production and discover an urgent bug that needs fixing immediately, but you are unable to deploy the current collaboration branch, you might need to implement a hotfix.
- This method is often referred to as quick-fix engineering (QFE).
- Managing a hotfix production environment in Azure Data Factory (ADF) is crucial for maintaining stable data workflows while addressing urgent issues promptly.
- This method ensures that critical fixes can be deployed without disrupting the main production environment.

Steps to Implement a Hotfix Production Environment

1. **Set Up Parallel Environments:**
    - **Primary Production Environment:** The main environment for running data workflows under normal operations.
    - **Hotfix Environment:** A secondary environment specifically for deploying urgent fixes.
2. **Clone the Production Environment:**
    - Create an identical copy of your primary production environment in ADF. This clone will serve as your hotfix environment.
3. **Deploy Hotfixes:**
    - Develop and test hotfixes in a separate development environment.
    - Once tested, deploy the hotfixes to the hotfix environment.
4. **Validate Hotfixes:**
    - Thoroughly test the hotfixes in the hotfix environment to ensure they resolve the issues without introducing new problems.
    - Use the same data and workloads as the primary production environment for accurate validation.
5. **Promote Hotfixes to Production:**
    - After successful validation, promote the hotfixes from the hotfix environment to the primary production environment.
    - Automate this step using CI/CD pipelines to ensure a smooth and error-free deployment.

6. **Monitor and Rollback:**
    - Continuously monitor the production environment after deploying hotfixes to ensure stability.
    - Have a rollback plan in place to revert to the previous state if any issues arise post-deployment.

Benefits of Using a Hotfix Production Environment

- **Minimized Downtime:** Quickly deploy critical fixes without affecting the main production environment, reducing downtime.
- **Reduced Risk:** Validate changes in a controlled environment before applying them to production, minimizing the risk of new issues.
- **Efficient Issue Resolution:** Promptly address urgent problems, maintaining the reliability and performance of your data workflows.

Best Practices

- **Regular Synchronization:** Keep the hotfix environment in sync with the primary production environment to ensure consistency.
- **Automation:** Use CI/CD pipelines to automate the deployment process, reducing manual errors and ensuring repeatability.
- **Documentation:** Maintain detailed documentation of all changes and deployments for traceability and compliance.

Use Azure App Configuration to manage feature flags

Managing Feature Flags in Azure App Configuration

Overview:

- Azure App Configuration includes feature flags to enable or disable specific functionalities, along with variant feature flags (preview) that allow for multiple variations of a feature flag.

Feature Manager UI:

- The Feature Manager in the Azure portal provides a user-friendly interface for creating and managing feature flags and variant feature flags used in applications.

Prerequisites:

- An active Azure account subscription is required.
- Create an Azure App Configuration store before proceeding.

Adding a New Feature Flag:

- ➢ **Access Azure App Configuration Store:**
    - ■ Open your Azure App Configuration store in the Azure portal.
- ➢ **Navigate to Feature Manager:**

- From the Operations menu, select "Feature manager" and then choose "Create".

➢ **Select Feature Flag:**
  - Under the "Create" section, opt for creating a "Feature flag".

➢ **Enter Information:**
  - Provide or select the necessary information as prompted, which may include flag name, description, and settings related to the feature flag.

Managing Views in Feature Manager

Feature Manager Display:

- The Feature manager menu shows all feature flags and variant feature flags stored in Azure App Configuration.
- You can customize the display by selecting "Manage view" in the Azure portal.

Settings:

- Adjust the number of feature flags loaded per "Load more" action.
- The "Load more" button appears only if there are more than 200 feature flags.

Edit Columns:

- Add or remove columns and rearrange the order of columns.
- Feature flags created with the Feature Manager are stored as regular key-values with the prefix .appconfig.featureflag/ and content type application/vnd.microsoft.appconfig.ff+json;charset=utf-8.

Viewing Underlying Key-Values:

1. **Access Configuration Explorer:**
   - In the Operations menu, open the "Configuration explorer".
2. **Manage View Settings:**
   - Select "Manage view" > "Settings".
   - Enable "Include feature flags in the configuration explorer" and click "Apply".

Build container images to deploy apps - Azure Pipelines

Quickstart: Building a Container Image for App Deployment using Azure Pipelines

Prerequisites:

- **Azure Account:** An active subscription (create for free if you don't have one).
- **GitHub Account:** Sign up for free if you don't have one.

Building a Linux or Windows Image:

- **Sign in to Azure DevOps:**
  - Log in to your Azure DevOps organization.
  - Navigate to your project.
- **Create a New Pipeline:**

- Go to the Pipelines section.
    - Select "New Pipeline" or "Create Pipeline" if it's the first pipeline in the project.
- **Select Source Code Location:**
    - Choose GitHub as the source code location.
    - Select your repository.
    - Choose the "Starter pipeline".
- **GitHub Authentication:**
    - If redirected, sign in with your GitHub credentials.
    - If prompted, approve and install the Azure Pipelines app on GitHub.
- **Configure Pipeline:**
    - Replace the contents of azure-pipelines.yml with the following code:

```
pool:
vmImage: ubuntu-latest # Change to windows-latest for Windows containers
steps:
- task: Docker@2
  inputs:
    containerRegistry: '$(dockerRegistryServiceConnection)'
    repository: 'myapp'
    command: 'buildAndPush'
    Dockerfile: '**/Dockerfile'
    tags: |
      $(Build.BuildId)
```

    - **Ensure vmImage is set to ubuntu-latest for Linux or windows-latest for Windows.**
    - **Save and run the configuration.**
- **Commit Changes:**
    - When prompted, add a commit message.
    - Select "Save and run".
- **Using Self-Hosted Agents:**
    - Ensure Docker is installed on the agent's host.
    - The Docker engine/daemon must be running with elevated privileges.

Pushing the Image:

- **Storage on Agent:**
    - The container images are built and stored on the agent.
- **Pushing to Registries:**
    - Push your image to Google Container Registry, Docker Hub, or Azure Container Registry.

    - For detailed steps, refer to the documentation on pushing an image to these registries.

Additional Resources:

- Docker Task Documentation: Learn more about the Docker task used in this sample.
- Command Line Tasks: You can directly invoke Docker commands using a command line task.

Design and implement infrastructure as code (IaC)

Infrastructure as code (IaC) Overview and Recommendations

- Infrastructure as Code (IaC) is a DevOps practice that involves managing and provisioning computing infrastructure through machine-readable configuration files instead of through manual hardware setup or interactive configuration tools.
- IaC facilitates infrastructure automation, reduces manual tasks, and ensures consistency and repeatability.
- Similar to how the same source code consistently produces the same binary, an Infrastructure as Code (IaC) model reliably creates the same environment with each deployment..

Key Features

1. **Automation**: Automates infrastructure setup and management, minimizing manual interventions.
2. **Version Control**: Stores infrastructure definitions in version control systems, enabling change tracking and collaborative development.
3. **Consistency**: Ensures consistent provisioning of infrastructure, reducing configuration drift and errors.
4. **Scalability**: Allows quick scaling of environments by automating the setup process.
5. **Reusability**: Enables reuse of infrastructure code across different environments (e.g., development, testing, production).

Infrastructure as Code (IaC) can be implemented using two main approaches: declarative and imperative. Each has distinct advantages and is suited for different scenarios.

Declarative Approach

The declarative approach involves specifying the desired state of your infrastructure without detailing the steps to achieve that state. The IaC tool determines the necessary steps to reach and maintain this state.

Key Characteristics:

1. **State Management:** Defines the desired state explicitly, with the IaC tool ensuring the actual state matches it.
2. **Idempotence:** Running the configuration multiple times results in the same state, avoiding unintended changes.
3. **Simplicity:** Focuses on "what" the end state should be, rather than "how" to achieve it.

Advantages:

- **Ease of Use:** Simplifies infrastructure management by focusing on the desired end state.
- **Consistency:** Ensures consistent infrastructure setups, reducing configuration drift.

- **Automatic Dependency Management:** Determines the order of operations to achieve the desired state automatically.

Imperative Approach

The imperative approach involves defining specific commands or instructions needed to achieve the desired state of your infrastructure. This method focuses more on the sequence of operations.

Key Characteristics:

1. **Step-by-Step Instructions:** Explicitly defines the steps to configure and manage the infrastructure.
2. **Procedural:** Emphasizes "how" to achieve the desired state.
3. **Examples:** Ansible (with playbooks), Shell scripts, and other scripting languages.

Advantages:

- **Fine-Grained Control:** Offers precise control over the infrastructure provisioning process.
- **Transparency:** Clearly shows the steps being executed, making it easier to understand the operations.
- **Flexibility:** Better suited for dynamic or complex infrastructure setups.

Benefits of IaC

- **Speed and Efficiency**: Automates repetitive tasks, speeding up the infrastructure provisioning process.
- **Reduced Errors**: Minimizes human errors associated with manual configuration.
- **Improved Collaboration**: Facilitates team collaboration on infrastructure code using version control systems.
- **Cost Savings**: Lowers operational costs by automating infrastructure management and reducing the need for manual intervention.

Recommendations for Implementing IaC

- **Use a Version Control System (VCS)**: Save your infrastructure code in a version control system such as Git.This allows for change tracking, reverting to previous versions, and collaborative development.
- **Select the Right IaC Tool**: Choose a tool that aligns with your environment and requirements, such as Terraform, AWS CloudFormation, Azure Resource Manager (ARM) templates, or Ansible.
- **Modularize Your Code**: Break your infrastructure code into smaller, reusable components to improve maintainability and manage complex environments more effectively.
- **Integrate with CI/CD**: Automate the deployment of infrastructure changes by integrating IaC into your Continuous Integration and Continuous Deployment (CI/CD) pipelines, ensuring consistent testing and deployment processes.

Desired State Configuration for Azure Overview

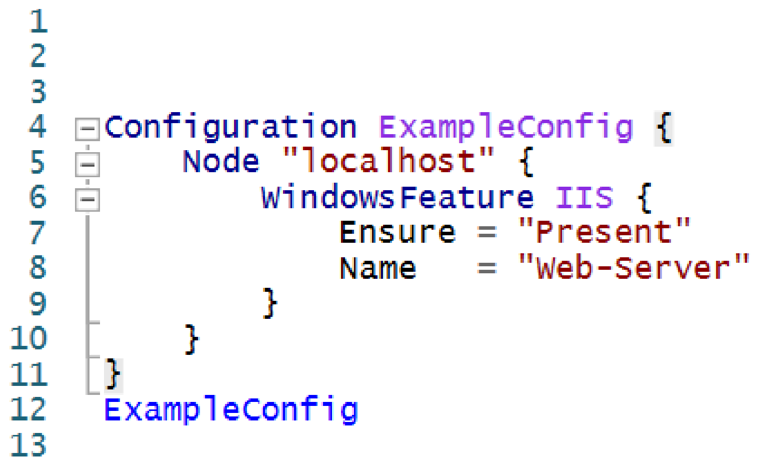
- Desired State Configuration (DSC) is a management platform built on PowerShell that enables administrators to manage IT and development infrastructure through configuration as code.
- DSC includes PowerShell language extensions, cmdlets, and resources that allow for the definition and maintenance of the desired state of your environment. Utilizing
- Azure DSC helps with achieving efficient, repeatable, and scalable configuration management, supporting DevOps practices and operational excellence.

Key Features of DSC

- **Declarative Syntax:** Use simple PowerShell scripts to specify the desired state of your infrastructure.
- **Consistency:** Maintains systems in the defined state by automatically correcting any deviations.
- **Idempotent:** Ensures that applying the same configuration repeatedly results in the same outcome without side effects.
- **Extensibility:** Allows you to develop custom DSC resources if the provided ones are insufficient.

Components of DSC

1. **Configuration**: A declarative PowerShell script that specifies the desired state of a node by configuring instances of resources.



```



Configuration ExampleConfig {
    Node "localhost" {
        WindowsFeature IIS {
            Ensure = "Present"
            Name   = "Web-Server"
        }
    }
}
ExampleConfig

```

2. **Resources**: Include the code that establishes and maintains the target configuration in the defined state.
    - **Built-in Resources**: Provided by DSC (e.g., WindowsFeature, File, Service).
    - **Custom Resources**: Created by users to extend DSC functionality.
3. **Local Configuration Manager (LCM)**: The engine that DSC uses to manage the interaction between resources and configurations. The LCM frequently checks the

system, utilizing the control flow defined by resources, to ensure the configuration-defined state is consistently maintained.

Benefits of DSC in Azure

- **Automation:** Automates infrastructure configuration management, reducing manual effort.
- **Consistency:** Ensures consistent configuration across environments.
- **Compliance:** Helps maintain compliance by ensuring systems adhere to defined configurations.
- **Scalability:** Manages a large number of systems through centralized configuration management.

Use Cases

1. **Automating Server Configuration:** Streamlines server setup and ensures uniform configurations.
2. **Maintaining Configuration Compliance:** Keeps systems aligned with predefined standards and automatically corrects any deviations.
3. **Deploying Web Servers:** Simplifies the deployment and configuration of IIS or Apache web servers.
4. **Database Server Configuration:** Ensures consistent setup and management of database servers.
5. **Configuration of Azure VMs:** Manages Azure VM configurations from creation, including application deployment, network settings, and security policies.
6. **Scaling and Consistency in Multi-Node Environments:** Guarantees identical configurations across nodes in load-balanced or clustered setups.

Manage flaky tests - Azure Pipelines

In software development, testing plays a crucial role in validating that software operates correctly and ensuring that code changes don't introduce new issues. However, a common challenge faced during automated testing is dealing with flaky tests.
A flaky test is one that produces inconsistent outcomes, like passing or failing, even when the source code or execution environment remains unchanged.

Flaky Tests Issue:

- Produce inconsistent results on different runs without code changes.
- Caused by factors like race conditions, non-deterministic behavior, or environment variations.
- Leads to wasted time, frustration, and reduced confidence in tests.

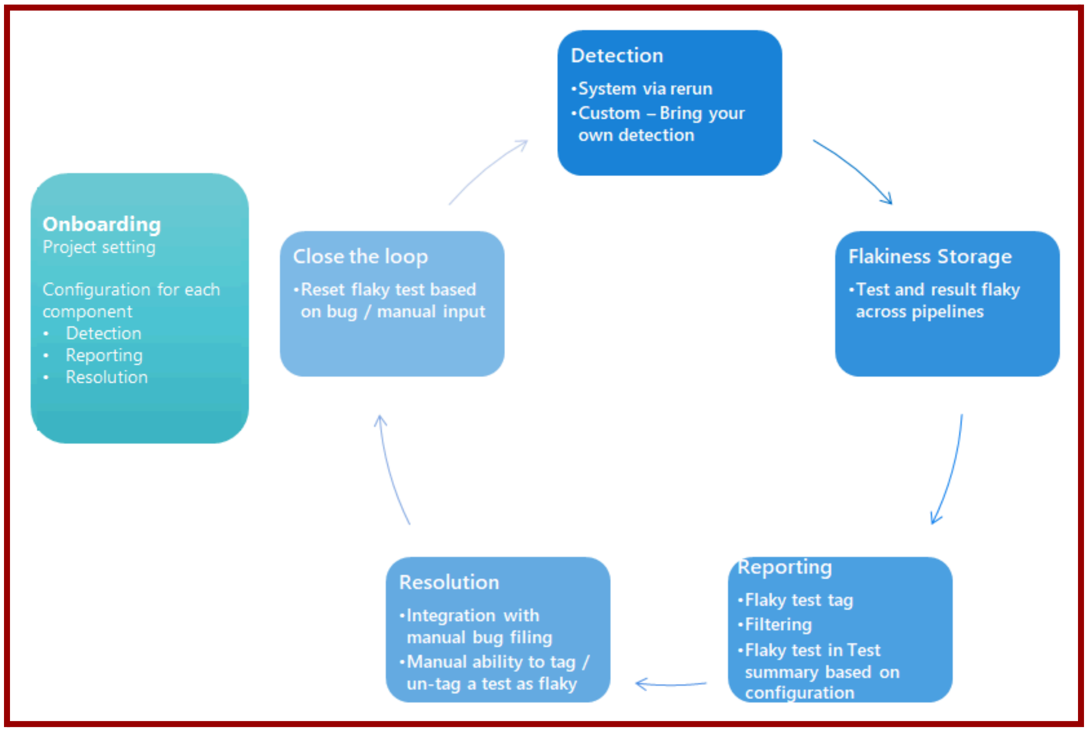
To mitigate the issues caused by flaky tests, Azure DevOps offers built-in features for managing them.

Azure DevOps Solution:

- **System Detection:** Automatically identifies flaky tests.
- **Custom Detection:** Allows tailored identification based on specific criteria.

The purpose of incorporating flaky test management into the product is to alleviate the issues caused by flaky tests and streamline the entire workflow. The benefits of flaky test management include:

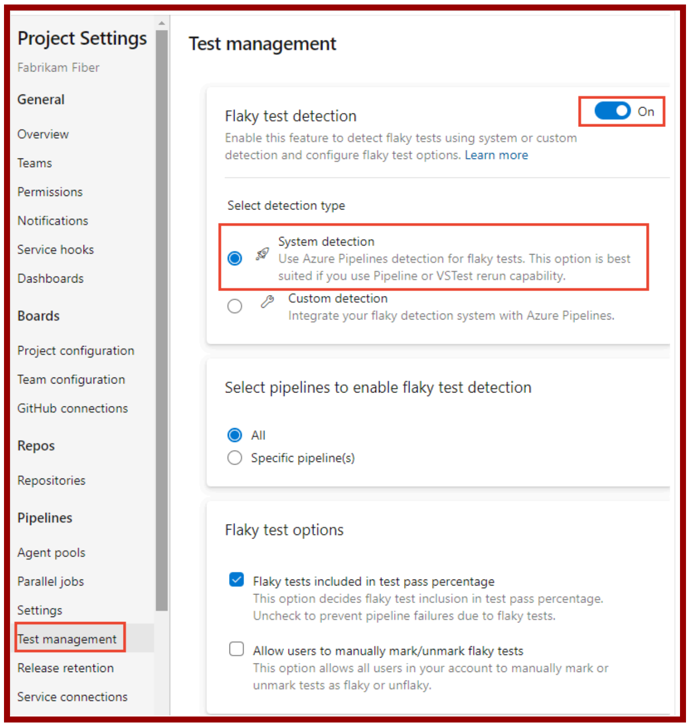
- **Detection:** Automatically identifies flaky tests through reruns or allows integration of custom detection methods.
- **Management:** Once a test is marked as flaky, the information is accessible across all pipelines for that branch.
- **Reporting:** Offers options to either prevent build failures due to flaky tests or use the flaky tag solely for troubleshooting.
- **Resolution:** Supports manual creation of bugs and the ability to manually mark or unmark tests as flaky based on analysis.
- **Feedback Loop:** Resets the status of flaky tests following bug resolution or manual input.


Detection
•System via rerun
•Custom – Bring your own detection
Flakiness Storage
•Test and result flaky across pipelines
Reporting
•Flaky test tag
•Filtering
•Flaky test in Test summary based on configuration
Resolution
•Integration with manual bug filing
•Manual ability to tag / un-tag a test as flaky
Close the loop
•Reset flaky test based on bug / manual input
Onboarding
Project setting
Configuration for each component
• Detection
• Reporting
• Resolution

[Source: Microsoft Documentation]

To enable flaky test management in Azure Pipelines:

1. Navigate to your **Project settings**.
2. Under the **Pipelines** section, choose **Test management**.
3. Toggle the switch to **On**.



Project Settings
Fabrikam Fiber

General
Overview
Teams
Permissions
Notifications
Service hooks
Dashboards

Boards
Project configuration
Team configuration
GitHub connections

Repos
Repositories

Pipelines
Agent pools
Parallel jobs
Settings
Test management
Release retention
Service connections

Test management

Flaky test detection — On
Enable this feature to detect flaky tests using system or custom detection and configure flaky test options. Learn more

Select detection type
- [✓] System detection
  Use Azure Pipelines detection for flaky tests. This option is best suited if you use Pipeline or VSTest rerun capability.
- [ ] Custom detection
  Integrate your flaky detection system with Azure Pipelines.

Select pipelines to enable flaky test detection
- [✓] All
- [ ] Specific pipeline(s)

Flaky test options
- [✓] Flaky tests included in test pass percentage
  This option decides flaky test inclusion in test pass percentage. Uncheck to prevent pipeline failures due to flaky tests.
- [ ] Allow users to manually mark/unmark flaky tests
  This option allows all users in your account to manually mark or unmark tests as flaky or unflaky.

[Source: Microsoft Documentation]

Advancing application reliability with performance testing

- Performance testing plays a vital role in ensuring the dependability and stability of applications.
- It helps pinpoint bottlenecks, understand system behavior under various loads, and confirm that the application can handle both expected and unexpected traffic.
- Performance testing should be continuously done by integrating it into CI/CD pipelines to detect issues promptly.
- Tools like Azure Load Testing can be used for performance testing.

Types of Performance Testing

- **Load Testing:** Determines whether the application can manage expected levels of traffic.
- **Stress Testing:** Assesses the application's response and recovery under extreme conditions.
- **Soak/Endurance Testing:** Evaluate the application's performance over prolonged durations.
- **Breakpoint Testing:** Identifies the maximum load capacity the application can withstand.
- **Scalability Testing:** Measures how effectively the application scales with varying loads.

Best Practices for Performance Testing

- **Establish Clear Goals** - Clearly define your objectives for performance testing, such as assessing capacity limits, pinpointing bottlenecks, or adhering to SLAs.

- **Emulate Realistic Load Scenarios -** Employ data and traffic patterns that replicate real-world conditions to achieve meaningful and applicable test outcomes.
- **Automate Performance Testing -** Integrate performance tests into your CI/CD pipeline to proactively detect performance regressions and ensure consistent testing across all releases.
- **Thoroughly Evaluate Results -** Move beyond basic pass/fail evaluations and scrutinize detailed metrics and logs to uncover the underlying causes of performance challenges.
- **Continuous Monitoring and Testing-** Implement ongoing monitoring of your application in production, coupled with regular performance assessments.

Tools for Performance Testing

- **JMeter -** A tool available for load testing and performance measurement, designed for open-source use.
- **Gatling -** A powerful tool for simulating high loads and analyzing performance.
- **Apache Bench (ab) -** A simple tool for quickly testing HTTP server performance.
- **LoadRunner -** A comprehensive performance testing tool for various applications and protocols.
- **Azure Load Testing -** A cloud-based load testing service that integrates with Azure DevOps for simulating high user traffic on your applications.

DevSecOps for infrastructure as code (IaC)

Implementing DevSecOps for IaC via GitHub

Operational Excellence, Security, and Cost Optimization:

Test-Driven Development:

- **Commit Changes:** Update infrastructure code, including IaC templates, on GitHub for version control.
- **Simultaneous Testing:** Develop unit tests, integration tests, and Policy as Code (PaC) alongside IaC to ensure quality.
- **Automated Testing:** Use GitHub Actions to automate unit tests triggered by PRs, streamlining validation.

GitHub Actions Configuration:

- **Workflow Setup:** Configure GitHub to test IaC thoroughly, identifying issues in infrastructure states and plans.
- **Security Scans:** Leverage GitHub Actions for code quality and security scans using CodeQL, alerting stakeholders of vulnerabilities.

Infrastructure Provisioning and Testing:

- **Dynamic Allocation:** Dynamically provision and modify resources with the IaC tool, tailoring specifications for each environment.

—-Back to Index—

- **Automated Integration Testing:** Automated integration tests are conducted on provisioned resources for performance validation.

Manual Updates and Reconciliation:

- **Temporary Access:** Temporarily elevate administrator access for manual updates.
- **Reconciliation:** Document post-update steps in GitHub for transparency and accountability.

Security Monitoring and Governance:

- **Continuous Monitoring:** Implement SecOps practices for continuous threat detection and defense.
- **Policy Enforcement:** Use Azure Policy to enforce governance standards, alerting GitHub upon anomalies.

Features

➔ **GitHub:** A collaborative platform for version control and code management.
➔ **GitHub Actions:** Workflow automation for CI/CD and testing.
➔ **GitHub Advanced Security:** Enhanced security features for IaC with additional licensing.
➔ **CodeQL:**Static code analysis tool for identifying vulnerabilities.
➔ **Terraform:**Infrastructure automation tool for various environments.
➔ **Microsoft Defender for Cloud:**Comprehensive security management for hybrid cloud workloads.
➔ **Microsoft Sentinel:**Cloud-native SIEM and SOAR platform for threat detection.
➔ **Azure Policy:**Governance enforcement for cloud resources.
➔ **Azure Monitor:**Telemetry collection and analysis for proactive monitoring.

Potential Scenarios

Multicloud Strategy for Contoso:

- Centralized IaC development for seamless infrastructure deployment.
- DevSecOps implementation for security and compliance.
- Audit trails for tracking infrastructure modifications.

What is Azure Deployment Environments?

Azure Deployment Environments Overview

Swift Infrastructure Provisioning: Development teams can rapidly provision app infrastructure using project-based templates, ensuring consistency, best practices, and security.

Components of Deployment Environments: Deployment environments comprise Azure infrastructure resources defined in environment definition templates.

Management by Platform Engineers: Platform engineers oversee environment definitions and associate them with projects, specifying available environments for developers and assigning permissions.

Empowering Platform Engineers:

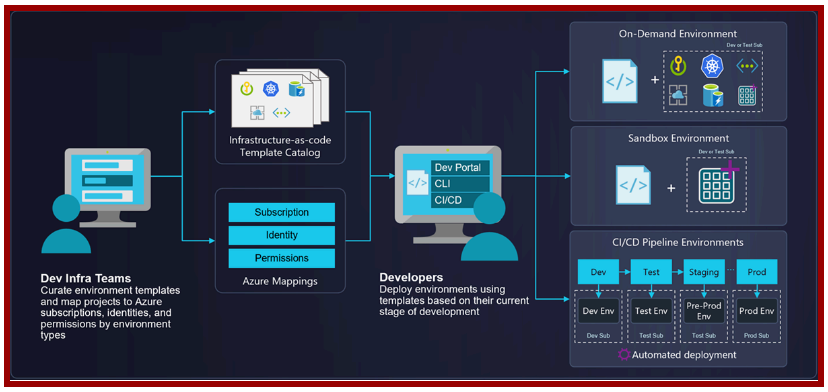
- Azure Deployment Environments empower platform engineers to enforce policies and settings on different environment types, control resource configurations, and track environments across projects.

Roles and Responsibilities:

- Platform engineers configure subscriptions, identity, and permissions, while developers deploy applications on the infrastructure based on templates.

Scalable Environment Support:

- Environments cater to various scenarios such as on-demand provisioning, testing sandboxes, and facilitating CI/CD pipelines for continuous integration and deployment.



Infrastructure-as-code Template Catalog
Subscription
Identity
Permissions
Azure Mappings
Dev Infra Teams
Curate environment templates and map projects to Azure subscriptions, identities, and permissions by environment types
Dev Portal
CLI
CI/CD
Developers
Deploy environments using templates based on their current stage of development
On-Demand Environment
Dev or Test Sub
Sandbox Environment
Dev or Test Sub
CI/CD Pipeline Environments
Dev
Test
Staging
Prod
Dev Env
Test Env
Pre-Prod Env
Prod Env
Dev Sub
Test Sub
Test Sub
Prod Sub
Automated deployment

[Source: Microsoft Documentation]

Benefits of Azure Deployment Environments

Standardization and Collaboration:

- Encourage standardization and collaboration by storing Infrastructure as Code (IaC) templates in source control. Easily create on-demand environments and foster collaboration through sharing templates within your team or organization.

Compliance and Governance:

- Platform engineering teams can ensure compliance with enterprise security policies by curating environment definitions. Map projects to Azure subscriptions, identities, and permissions based on environment types to enforce governance.

Project-Based Configurations:

- Organize environment definitions according to the type of application being developed. Avoid the complexity of managing an unorganized list of templates or traditional IaC setups.

Seamless Self-Service:

- Facilitate worry-free self-service for development teams to swiftly create app infrastructure resources (PaaS, serverless, etc.) using preconfigured templates. Track resource costs to maintain budget constraints.

Integration with Existing Toolchain:

- Seamlessly integrate with your existing toolchain using APIs to provision environments directly from preferred CI tools, integrated development environments (IDEs), or automated release pipelines. Utilize the comprehensive command-line tool for additional flexibility.

Automate security and compliance scanning

Azure DevOps: Authentication & Authorization Methods

Azure DevOps provides a robust framework for authentication and authorization to ensure secure access and proper permissions management for your projects and resources.
Here's an overview of the methods used:

Authentication Methods

1. **Microsoft Account:**
   - Users can sign in with their Microsoft accounts (e.g., Outlook, Hotmail).
2. **Microsoft Entra ID:**
   - Provides single sign-on (SSO) for enterprises using Entra ID.
   - Users can log in using their organizational credentials.
3. **Personal Access Tokens (PATs):**
   - Tokens generated for individual users to authenticate API calls.
   - Can be scoped to specific tasks and have an expiration date.
4. **SSH Keys:**
   - Secure Shell (SSH) keys for secure access to Git repositories.
   - Users generate SSH keys and add them to their Azure DevOps account.
5. **OAuth:**
   - OAuth 2.0 protocol support for third-party applications to authenticate and access Azure DevOps resources on behalf of the user.

Authorization Methods

1. **Role-Based Access Control (RBAC):**
   - Assign roles to users and groups to manage permissions.
   - Roles include project contributors, project administrators, readers, and custom roles.
2. **Security Groups:**
   - Use Azure DevOps or Entra ID security groups to manage permissions.
   - Add users to groups to streamline permission management across multiple projects.
3. **Access Control Lists (ACLs):**
   - Fine-grained control over permissions for specific resources such as repositories, pipelines, and boards.
   - Define who can read, contribute, or administer each resource.
4. **Branch Policies:**
   - Define policies for branch protection, such as requiring pull requests for changes, enforcing code reviews, and requiring successful builds.
   - Ensure that code changes meet quality standards before being merged.

—-Back to Index—

5. **Service Connections:**
    - Securely connect to external services and resources.
    - Define and manage service connections to control access to external resources from Azure DevOps pipelines.

Security best practices - Azure DevOps

Ensuring the security of your Azure DevOps environment is crucial to safeguard your code, data, and resources. Below are essential security measures to enhance protection:

1. **Employ Robust Authentication:**
    - Mandate the use of multi-factor authentication (MFA) for all user accounts.
    - Utilize Entra ID integration to streamline authentication and enable (SSO).
2. **Choose Right Authentication Method:**
    - Consider service principals and managed identities
    - Use personal access tokens (PATs) sparingly
3. **Follow Least Privilege Access:**
    - Assign permissions based on the principle of least privilege, granting only necessary access.
    - Regularly review and adjust user access to align with their roles and responsibilities.
4. **Secure Code Repositories:**
    - Implement branch policies requiring prerequisites like pull requests, code reviews, and status checks.
    - Control access to sensitive repositories to ensure only authorized users have write access.
5. **Protect Build Pipelines:**
    - Secure service connections, avoid storing sensitive data, and restrict pipeline access based on user roles.
    - Audit pipeline configurations and logs regularly for unauthorized changes or access.
6. **Enable Security Features:**
    - Activate audit logging to monitor user activities and detect suspicious behavior.
    - Utilize Microsoft Defender for Cloud to identify and address security vulnerabilities.
7. **Enforce Security Policies:**
    - Establish and enforce security policies to comply with regulations and internal standards.
    - Review and update policies regularly to address evolving threats and compliance requirements.
8. **Encrypt Data:**
    - Use HTTPS for all communication with Azure DevOps services to encrypt data in transit.
    - Utilize secure storage solutions like Azure Key Vault for sensitive information.
9. **Keep Systems Updated:**

- Ensure Azure DevOps services, agents, and dependencies are up to date with security patches.
- Stay informed about security advisories and apply patches promptly.

10. **Educate Users:**
    - Offer security training to raise awareness about common threats and best practices for data protection.
11. **Implement Monitoring:**
    - Deploy continuous security monitoring to detect and respond to incidents promptly.
    - Use Azure Monitor and Microsoft Defender for Cloud for ongoing monitoring and analysis.

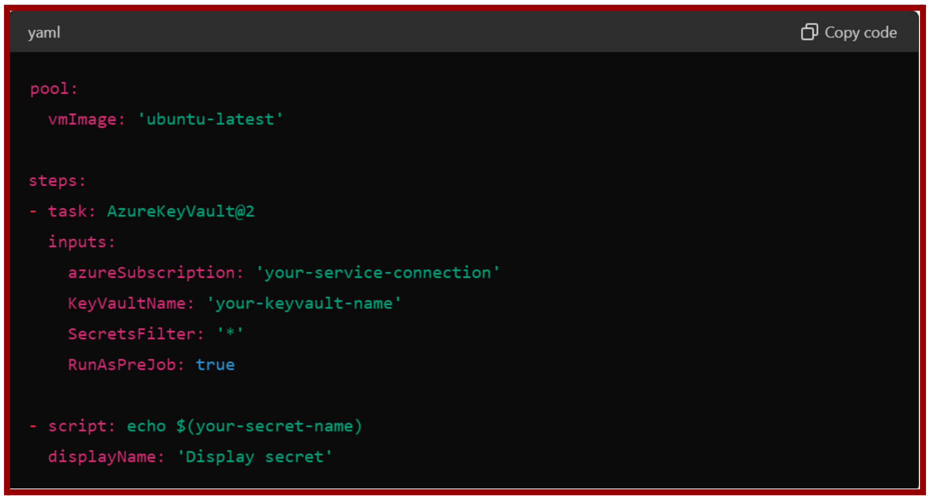
Azure KeyVault Secretes in Azure Pipelines

Integrating Azure Key Vault secrets into Azure Pipelines offers a secure method to manage confidential data such as API keys, connection strings, and passwords. Here's a guide on how to incorporate Azure Key Vault secrets into Azure Pipelines securely:

Implementation Steps

1. **Create an Azure Key Vault:**
    - Use the Azure portal, CLI, or PowerShell to set up a new Key Vault and add secrets to it.
2. **Grant Azure Pipelines Access:**
    - Use an Microsoft Entra ID service principal or a managed identity to grant Azure Pipelines the necessary access to the Key Vault.
    - Configure Key Vault access policies to allow the service principal to read secrets.
3. **Configure Azure Pipelines:**
    - Use the Azure Key Vault task in your pipeline YAML or classic pipeline editor.
    - Specify the Key Vault name and the secrets to retrieve.

Example YAML Configuration:



```yaml
pool:
  vmImage: 'ubuntu-latest'

steps:
- task: AzureKeyVault@2
  inputs:
    azureSubscription: 'your-service-connection'
    KeyVaultName: 'your-keyvault-name'
    SecretsFilter: '*'
    RunAsPreJob: true

- script: echo $(your-secret-name)
  displayName: 'Display secret'
```

[Source: Microsoft Documentation]

Use Cases

1. **Secure Storage of Pipeline Secrets:** Store sensitive data like API keys, database connection strings, or passwords in Azure Key Vault rather than directly in pipeline definitions.
2. **Retrieving Secrets in Build and Release Pipelines:** Configure Azure Pipelines to fetch secrets from Azure Key Vault during build or release stages, ensuring that sensitive data is accessed securely only when necessary.
3. **Rotating Secrets:** Regularly update secrets stored in Azure Key Vault without altering the pipeline configuration. Pipelines will automatically use the updated secrets, enhancing security by reducing the risk of compromised credentials.
4. **Environment-Specific Secrets:** Manage secrets for different environments (e.g., development, production) using separate Key Vaults or secret prefixes.
5. **Conditional Access Policies:** Implement conditional access policies to ensure that only authorized users and applications can access secrets during pipeline execution.
6. **Compliance and Auditing:** Utilize Azure Key Vault's logging and monitoring features to track secret access. This supports compliance and auditing by providing a detailed history of secret access events.

Benefits

- **Enhanced Security:** Secrets are stored and accessed securely, reducing the risk of exposure.
- **Centralized Management:** Manage secrets in a central location, simplifying administration and rotation.
- **Flexibility:** Easily update secrets without modifying pipeline configurations.
- **Compliance:** Track and audit secret access for compliance purposes.

GitHub Advanced Security for Azure DevOps

- GitHub Advanced Security offers robust security features that can seamlessly integrate with Azure DevOps to enhance your DevOps processes.
- These features include code scanning, secret scanning, and dependency reviews, all aimed at ensuring the highest level of security for your applications.

Overview of how GHAS can be used within Azure DevOps:

1. Code Scanning

- **Static Code Analysis:** GHAS performs static analysis on your codebase to identify vulnerabilities at an early stage of development.
- **CI/CD Integration:** Integrate code scanning into Azure DevOps pipelines to assess each code modification for potential security threats before integration.
- **Customizable Rules:** Utilize predefined rules or customize rules to match the specific requirements of your project.

2. Secret Scanning
   - **Sensitive Information Detection:** GHAS scans repositories to identify sensitive data like API keys and passwords that should not be stored in code.
   - **Alert System:** Automatically notifies developers upon detecting sensitive information to prevent inadvertent exposure.
   - **Preventive Policies:** Implement policies to prevent the inclusion of sensitive data in repository commits.
3. Dependency Reviews
   - **Vulnerability Scanning:** Examines your project dependencies for known security vulnerabilities, assisting in mitigating risks associated with third-party libraries.
   - **Dependency Graph:** Provides a detailed overview of your project's dependencies to facilitate effective management and updates.
   - **Automated Security Updates:** Recommends and in some cases applies security patches to vulnerable dependencies automatically.
4. Integration with Azure DevOps
   - **Unified Workflow:** GHAS features can be integrated directly into Azure DevOps pipelines and workflows, providing a comprehensive security solution.
   - **Automated Security Checks:** Set up automated security checks as part of your CI/CD pipeline to ensure code changes meet security standards before deployment.
   - **Detailed Security Reports:** Access detailed security reports and dashboards within Azure DevOps to effectively track and manage security issues.

Benefits of Using GitHub Advanced Security with Azure DevOps

- **Early Vulnerability Detection:** Incorporating security checks early in the development cycle helps identify vulnerabilities before deployment, reducing the likelihood of security incidents.
- **Regulatory Compliance:** Facilitates adherence to regulatory and compliance standards by enforcing security best practices across the development lifecycle.
- **Developer Productivity:** Automated security checks and notifications streamline security audits, freeing up developers to concentrate on writing secure code.
- **Holistic Security Management:** Offers a unified platform for overseeing code, dependencies, and secrets security, streamlining policy enforcement and issue tracking.

Troubleshoot pipeline runs - Azure

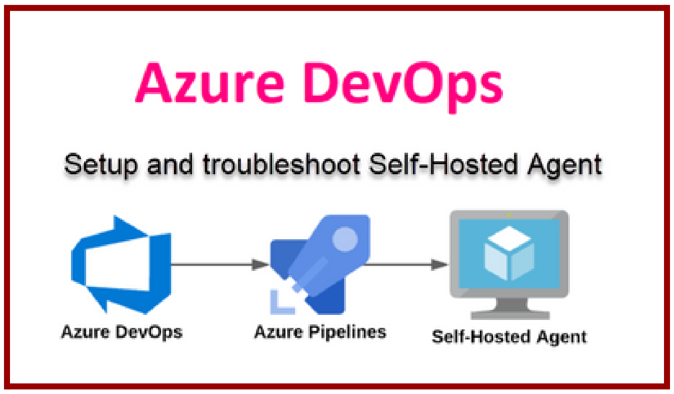
Pipeline Run Failure Troubleshooting

Utilise Diagnostic Information:

- Access the pipeline run summary page to view diagnostic information and logs.
- Choose "View raw log" to examine the raw log for the failing task.
- Use the "Find" feature to search the log for error details and clues.

Configure Verbose Logs:

- If default logs are insufficient, configure verbose logs for more detailed information.



[Source: Microsoft Documentation]

Error Analysis Page

Error Analysis Assistance:

- Navigate to the Error Analysis page for assistance with troubleshooting.
- Hover over the error information line and select the "View Analysis" icon.
- Choose "View agent" (for self-hosted agents) or "About hosted agent image" (for Microsoft-hosted agents) for additional agent details.
- Access pipeline run logs by selecting "View log".
- Click on the task name under "Run-time details" for task-specific information.
- For detailed task documentation, choose "About this task".

Troubleshooting Pipeline Run Failures

Task Insights: Enable Task Insights for Failed Pipeline Runs to quickly access detailed reports on build failures.

Job Time-out: Adjust job time-out settings, considering factors like repository type and agent usage.

File and Folder Management

Detecting Usage Issues: Utilize tools like Process Monitor to identify file and folder usage problems.

Anti-virus Exclusion: Exclude agent directories from anti-virus scans to prevent interference during builds.

MSBuild Best Practices

Process Management:

- Ensure proper MSBuild process handling to avoid issues with multiple processes.

Identifying Hangs:

- Analyze process dumps to identify and resolve hanging processes.

Line Ending Consistency

Ensuring Uniformity:

- Configure Git for consistent line endings to avoid cross-platform issues.

Bash Script Variables

Variable Handling:

- Temporarily disable debug mode in Bash scripts to prevent variable appending.

Python Application Dependencies

Dependency Installation:

- Use post-deployment scripts to ensure dependency installation for Python applications.

Service Connection Troubleshooting

Connection Resolution:

- Refer to the service connection troubleshooting guide for assistance.

Agent Resource Monitoring

Resource Utilization:

- Monitor agent resource usage to ensure adequate performance during pipeline runs.

Deploying Static Content

Content Upload:

- Utilize Azure File Copy task for uploading static content to static website containers.

Additional Resources

Common Issues Check:

- If the issue persists, refer to the "Common issues" section for potential solutions.
- Explore "Review logs to diagnose pipeline issues" for guidance on downloading comprehensive logs with additional diagnostic data.

Set retention policies for builds, releases, and tests

Setting Retention Policies

Pipeline Retention:

- Control storage usage by specifying how long to keep artifacts, symbols, attachments, runs, and pull request runs.

Release Retention:

- Determine whether to save builds and configure default and maximum retention settings for classic releases.

Test Retention:

- Manage the duration to retain automated and manual test runs, results, and attachments.

Configuring Retention

Accessing Project Settings:

- Navigate to the gear icon Settings tab within your project's settings.

Selecting Retention Options:

- Choose between Settings or Release Retention under Pipelines, or Retention under Test.

Setting Run Retention:

- Define the number of days to retain artifacts, symbols, and attachments.
- Specify the duration to keep runs and pull request runs.
- Configure the number of recent runs to retain for each pipeline.

Interpreting Recent Runs

Azure Repos:

- Retains the specified number of latest runs for the pipeline's default branch and each protected branch.
- Protected branches include those with configured branch policies.

Other Git Repositories:

- Retains the configured number of latest runs for the entire pipeline, irrespective of branches.

TFVC:

- Retains the specified number of latest runs for the entire pipeline, regardless of branches.

Automatically Managing Retention Leases

Retention Lease Overview:

- Retention leases regulate the lifespan of pipeline runs beyond configured retention periods.
- Leases can be added or deleted via the Lease API, callable within pipelines using scripts and predefined variables.

Dynamic Retention Leases:

- Leases can vary based on pipeline run characteristics, such as deployment environment.
- For instance, test environment runs may have shorter retention periods compared to production runs.

Manual Retention Lease Management

Manual Lease Setting:

- Set retention leases manually via the "More actions" menu on the Pipeline run details page.

Configuring Release Retention Policies

Release Retention Overview:

- Release retention policies dictate the lifespan of releases and their associated runs.
- Policies include specifying retention duration and minimum retained releases.

Release Behavior:

- The retention timer resets with every modification or deployment to a stage.

- The minimum retained releases setting takes precedence, ensuring a specified number of recent releases are retained indefinitely.
- Authors can customize retention policies within release pipelines on the Retention tab.

Pipeline Retention Settings

Accessing Retention Settings:

- View and adjust pipeline retention settings within Project Settings for Pipelines in the Settings section.

Analyze metrics

Monitor usage of Azure DevOps Services

- Monitoring the usage of Azure DevOps Services is essential for ensuring optimal performance, efficient resource utilization, and the detection of potential issues.
- You can review user activity in Azure DevOps Services over the past 28 days.

Below are effective methods for monitoring and managing Azure DevOps Services usage:

1. **Utilize Azure DevOps Analytics**
   - Employ built-in Analytics Views to track work item progress, build success rates, and other critical metrics.
   - Customize dashboards to display relevant data such as work item states, build and release statuses, and repository statistics.
2. **Implement Service Hooks and Notifications**
   - Set up service hooks to deliver real-time notifications to external services for events like code pushes, work item updates, and build completions.
   - Configure email notifications to alert team members about important events like build failures and work item changes.
3. **Integrate with Azure Monitor**
   - Connect Azure DevOps with Azure Monitor to gather, analyze, and act on telemetry data from your DevOps environment.
   - Utilize Log Analytics to query and analyze data from Azure DevOps logs, enabling the identification of usage patterns and potential issues.
4. **Review Usage Reports**
   - Access usage reports within the Azure DevOps portal to monitor the consumption of pipelines, agent jobs, and other services.
   - Generate custom reports using Azure DevOps Analytics and Azure Monitor data to gain deeper insights into service usage and performance.
5. **Audit and Security Monitoring**
   - Review audit logs to track changes and access events within your Azure DevOps organization, ensuring compliance and security.
   - Monitor user activity to detect any unusual behavior and ensure adherence to organizational policies.
6. **Cost Management**
   - Monitor billing and usage data to track costs associated with Azure DevOps Services and identify opportunities for optimization.
   - Utilize Azure Cost Management to analyze and manage spending across Azure services, including Azure DevOps.
7. **Performance Monitoring**

—-Back to Index—

- Track the performance of CI/CD pipelines to identify bottlenecks and optimize build times.
- Monitor resource utilization to ensure effective usage and identify scaling needs.

Monitoring Tools and Features

1. **Azure DevOps Dashboards:** Customize dashboards to display key metrics and project progress.
2. **Azure Monitor:** Collect and analyze telemetry data from Azure DevOps and other Azure services.
3. **Log Analytics:** Query and analyze logs from Azure DevOps for insights into usage patterns and performance.
4. **Service Hooks:** Configure notifications for real-time monitoring of external systems.
5. **Azure Cost Management:** Monitor and manage spending on Azure DevOps and other Azure services.

Analyze metrics in Azure DevOps

Analyzing metrics within Azure DevOps is essential for understanding the performance and health of your development workflows. Here's how you can effectively examine metrics within Azure DevOps:

1. **Work Item Metrics:**
   - Track the progression of work items, including the distribution across different states (New, Active, Closed).
   - Analyze trends in work item completion rates to identify bottlenecks and improve workflow efficiency.
   - Monitor cycle time to understand how long it takes for work items to move through your process.
2. **Build and Release Metrics:**
   - Measure the success and failure rates of builds to assess the stability of your codebase.
   - Evaluate build duration and queue times to find opportunities for optimization.
   - Track release frequency and deployment lead time to gauge the efficiency of your release process.
3. **Code Metrics:**
   - Assess code coverage to determine the effectiveness of your testing strategies and pinpoint areas needing more tests.
   - Track code review metrics, such as review cycle time and reviewer participation, to ensure thorough and timely code assessments.
4. **Pipeline Metrics:**
   - Measure pipeline execution times and queue lengths to identify performance bottlenecks and optimize resource allocation.

—-Back to Index—

    - Track success and failure rates of pipelines, including reasons for failures, to improve pipeline reliability.
5. **User Metrics**:
    - Track user activity and engagement to understand how users interact with Azure DevOps.
    - Analyze adoption rates of features and services to identify opportunities for training and improvement.
6. **Integration with Azure Monitor**:
    - Integrate Azure DevOps with Azure Monitor to gather and analyze telemetry data from your DevOps environment.
    - Use Log Analytics to query and analyze Azure DevOps logs for advanced analytics and troubleshooting.
7. **Custom Reporting and Dashboards**:
    - Create customized reports and dashboards using Azure DevOps Analytics to visualize team-specific metrics and processes.
    - Customize dashboards to highlight key metrics and trends, enabling stakeholders to make informed decisions based on data.

Kusto Query Language (KQL) in Azure DevOps

- Utilizing the Kusto Query Language (KQL) within Azure DevOps offers a robust method for extracting insights from your data.
- The Kusto Query Language excels in querying telemetry, metrics, and logs, offering robust capabilities such as extensive support for text search and parsing, time-series operators and functions, analytics and aggregation, geospatial analysis, vector similarity searches, and numerous other language features essential for efficient data analysis.
- It employs schema entities structured in a hierarchy akin to SQL, including databases, tables, and columns.

What is a Kusto query?

A Kusto query is a request to process data and retrieve results. These queries are written in plain text and utilize a data-flow model that is straightforward to understand, create, and automate. A Kusto query typically consists of one or more query statements.

Types of query statements:

- **User query statements:** These are primarily used by users to directly interact with Kusto and retrieve data.
- **Application query statements:** Designed to support scenarios where mid-tier applications modify user queries and send them to Kusto for processing.

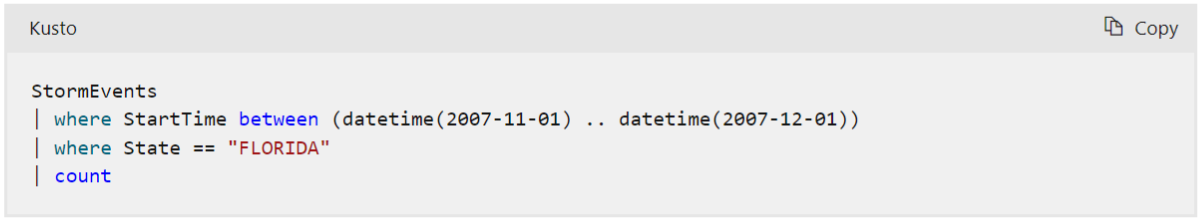
User query statements includes:

- **Let statement:** Establishes a binding between a name and an expression, aiding in breaking down complex queries into more manageable parts.
- **Set statement:** Defines request properties that influence query processing and result presentation.
- **Tabular expression statement:** The cornerstone of querying, retrieves pertinent data as query results.

Application query statements include:

- **Alias statement:** Establishes an alias to another database, whether within the same cluster or on a remote cluster.
- **Pattern statement:** Integrates applications built on Kusto, enabling them to influence query name resolution.
- **Query parameters statement:** Shields applications from injection attacks by managing query parameters, akin to safeguarding against SQL injection in SQL databases.
- **Restrict statement:** Empowers applications to confine queries to specific data subsets within Kusto, regulating access to designated columns and records.

Kusto query example:



```
StormEvents
| where StartTime between (datetime(2007-11-01) .. datetime(2007-12-01))
| where State == "FLORIDA"
| count
```

[Source: Microsoft Documentation]

Best Practices

Reduce the amount of data being processed:

- Apply time range filters early
- Use filters at the beginning of the query
- Select only the needed columns
- Perform data aggregation early
- Limit the number of rows with take
- Use distinct to remove duplicate rows
- Use let to store and reuse intermediate results
- Use sample for quick data subsets
- Optimize joins with lookup

Avoid using redundant qualified references:

- Use let for common filters.
- Simplify column references with project.

- Use short aliases for tables.
- Avoid repeating table names and conditions,
- Combine similar operations into single steps.

Implement Security and Compliance in an Azure DevOps pipeline

Ensuring Security and Compliance in Azure DevOps Pipelines

Maintaining security and compliance within Azure DevOps pipelines is paramount to safeguarding sensitive data and meeting regulatory standards. Here's a breakdown of essential practices and strategies to achieve this:

Understanding the Significance:

Security and Compliance: Protecting data integrity and adhering to regulatory frameworks like GDPR or HIPAA are non-negotiable aspects in today's digital landscape. Practices and Tools: Employing a combination of practices, tools, and processes is necessary to fortify defenses and ensure adherence to standards.

Critical Components:

- **Access Controls:**
  - Role-Based Access Control (RBAC): Implement RBAC to regulate access and safeguard resources from unauthorized use.
- **Data Encryption:**
  - Encryption Protocols: Utilize encryption techniques to secure data both at rest and in transit, enhancing overall data protection.
- **Audit Trails:**
  - Logging Mechanisms: Maintain comprehensive logs to track modifications, access attempts, and potential breaches, facilitating timely detection and response.
- **Compliance Standards:**
  - Regulatory Adherence: Align with industry-specific regulations such as GDPR or HIPAA, ensuring that the pipeline operations comply with relevant standards.

Implementation Strategies:

1. **Securing Pipeline Configuration:**
   - Secret Management: Safeguard sensitive information by securely storing secrets in dedicated repositories like Azure Key Vault or Variable Groups.
   - Repository Security: Enforce stringent access controls and branch policies within code repositories to mitigate unauthorized access and ensure code integrity.
2. **Continuous Security Scanning:**
   - Static and Dynamic Analysis: Integrate tools such as SonarQube to perform static code analysis (SAST) and dynamic scans (DAST) throughout the development lifecycle, identifying vulnerabilities early on.

3. **Compliance Checks:**
    - Policy Enforcement: Utilize Azure Policy to enforce compliance standards across various resources, ensuring alignment with regulatory requirements.
    - Regular Audits: Conduct periodic audits to validate compliance status and address any deviations promptly.
4. **Vulnerability Management:**
    - Patch Management: Stay proactive by regularly updating software and dependencies to mitigate known vulnerabilities and enhance overall system security.
    - Vulnerability Scans: Employ automated tools to scan for potential security flaws within both infrastructure and applications, addressing any identified vulnerabilities promptly.
5. **Incident Response and Recovery:**
    - Automated Detection: Implement automated incident detection mechanisms to promptly identify and respond to security breaches, minimizing potential damages.
    - Disaster Recovery Planning: Develop robust disaster recovery plans and conduct regular testing to ensure the resilience of the pipeline infrastructure and processes.

Monitoring Applications using Application Insights

Key Features

- **Automatic Instrumentation**
    - **ASP.NET Applications**: Provides out-of-the-box telemetry data for ASP.NET applications including usage, exceptions, requests, performance, and logs.
- **Cross-Platform Monitoring**
    - **Multiple Languages**: Supports Java, Ruby, Python, PHP, Node.js, and more through open-source SDKs.
    - **Status Monitor**: Installable on Azure App Services and virtual machines for performance monitoring without the need for application updates or redeployments.

Benefits

- **Exception and Performance Diagnostics**
    - Quickly identify and resolve application exceptions and performance bottlenecks.
- **Interactive Data Analysis**
    - Allows for in-depth, interactive analysis of application data to understand performance and user behavior.
- **Azure Diagnostics**

        - Provides comprehensive diagnostic information within the Azure ecosystem.
- **Proactive Detection**
    - Detects potential issues before they impact users, enabling preemptive troubleshooting.

Implementation

- **Installation**
    - **Status Monitor**: Install on Azure App Services and virtual machines via the Azure portal for seamless monitoring.
- **Integration**
    - **Open-Source SDKs**: Integrate SDKs into applications across different languages and platforms for comprehensive telemetry data.

Usage

- **Real-Time Monitoring**
    - Continuously monitor live web applications to gain real-time insights into performance and usage patterns.
- **Platform Versatility**
    - Supports applications hosted both on-premises and in the cloud, providing flexibility in monitoring various deployment environments.
- **Extensibility**
    - Extend the service by adding custom telemetry data and integrating with other Azure services for enhanced monitoring capabilities.

Enhancements

- **Custom Dashboards**
    - Create and customize dashboards to visualize key metrics and performance indicators.
- **Alerting**
    - Set up alerts to receive notifications for critical performance issues or threshold breaches.
- **Continuous Improvement**
    - Use insights from Application Insights to continuously improve application performance and user experience.

—-Back to Index—