AZ 400

part 1

Outline

1. Analyzing Metrics
2. Designing and Implementing a Source Control Strategy
3. Planning and Implementing Branching Strategies for the Source Code
4. Configuring Repositories

Analyzing Metrics

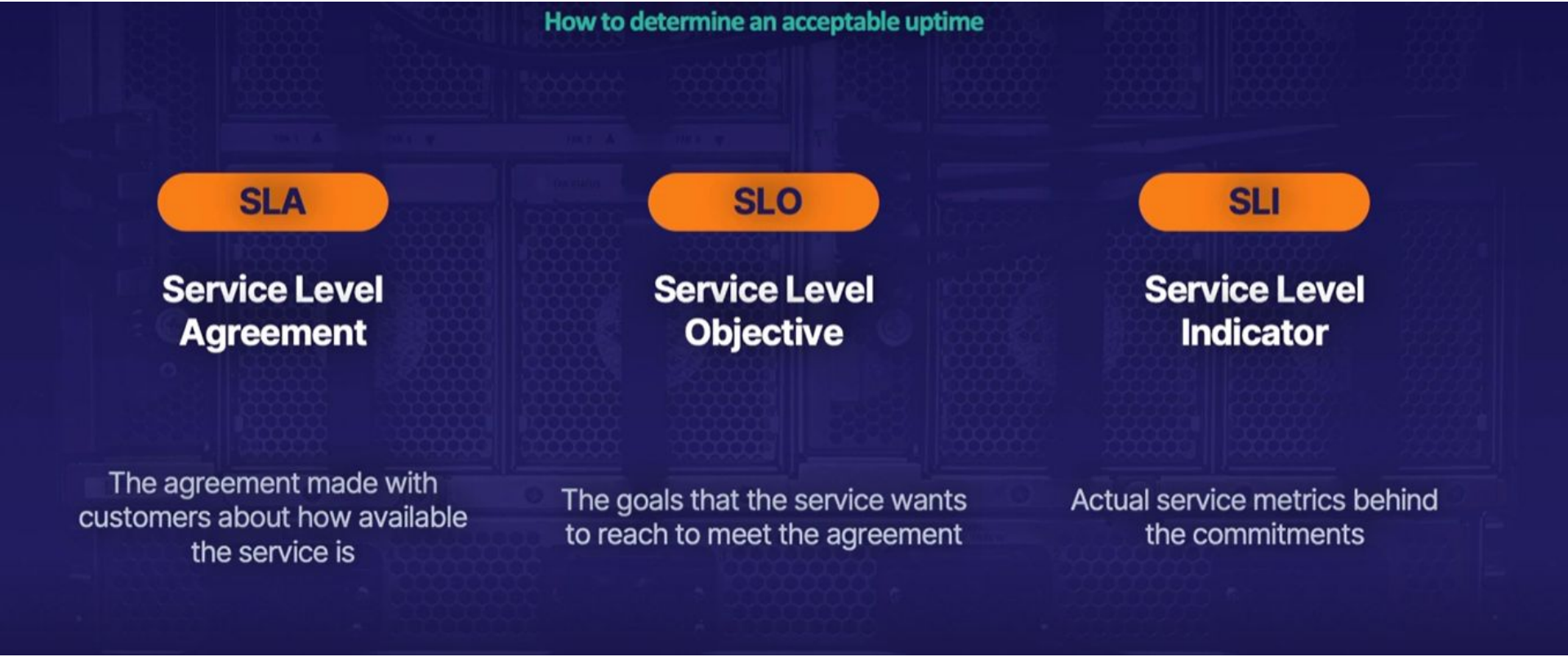
Big Picture: What Is Site Reliability Engineering (SRE)?

What Is Reliability?

The application can be used when and how it is expected to

- Availability
- Performance
- Latency
- Security

Measuring Reliability



Why Do We Need Site Reliability Engineering?

Ownership

Generally, not focused on by the teams best-suited to handle the issues

Work Silos

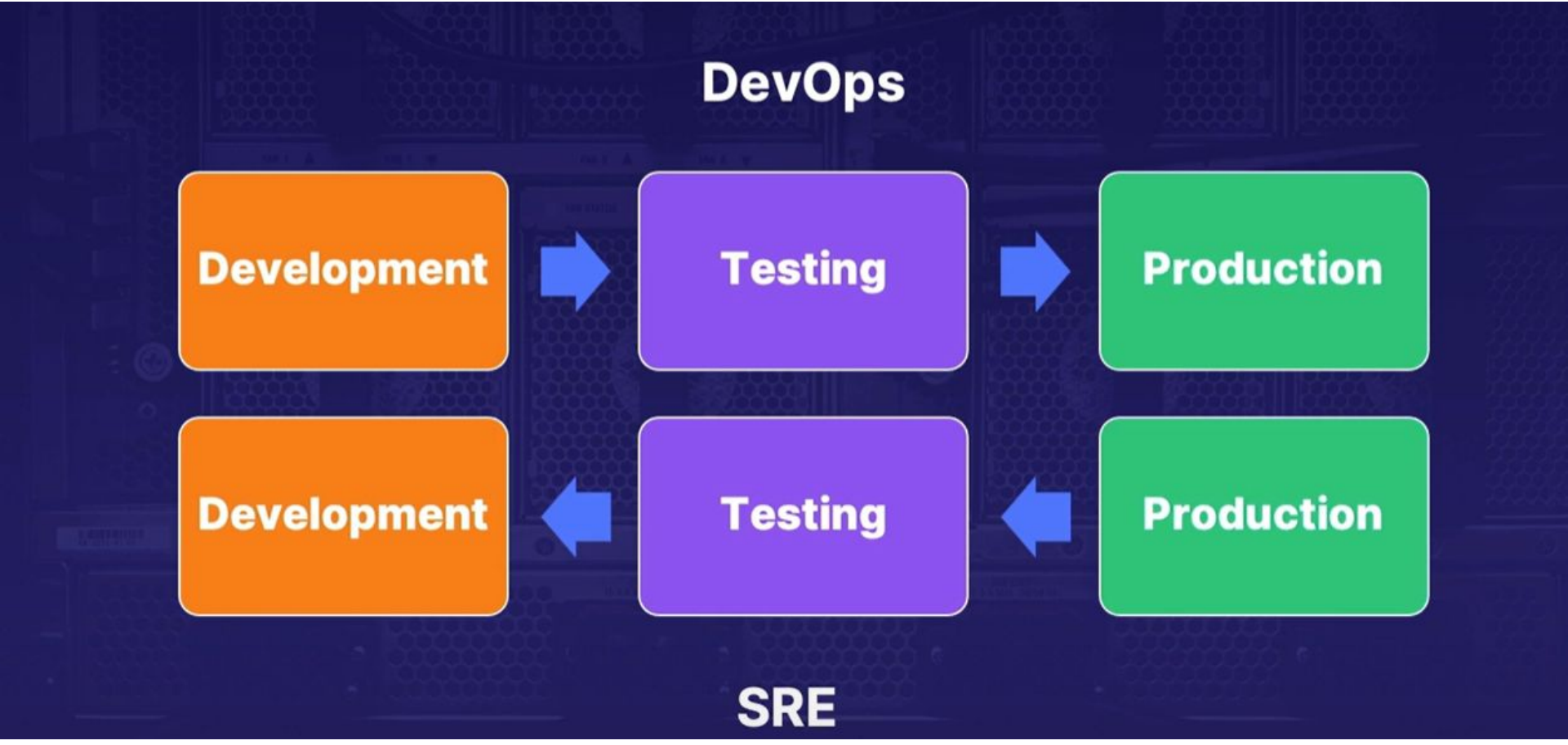
Software engineers would develop the code and then hand it off to deploy and maintain

Full Stack

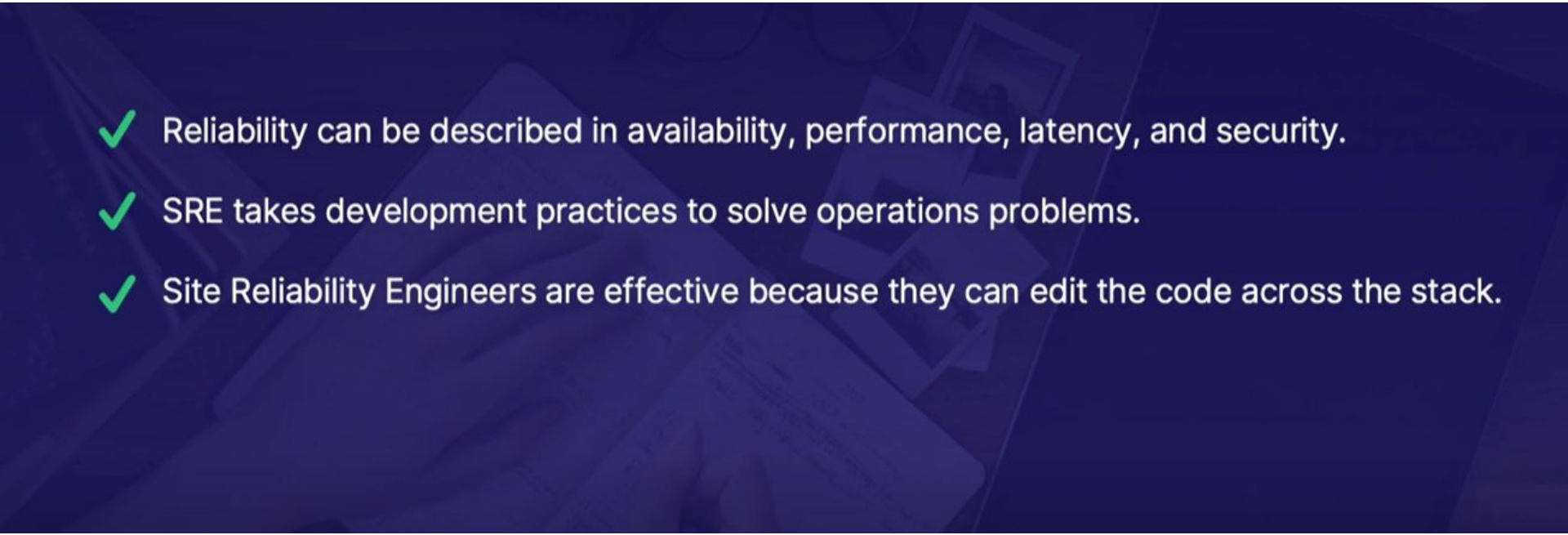
Code can be edited to increase performance and develop automation to reduce errors

Key Concepts

SRE vs DevOps

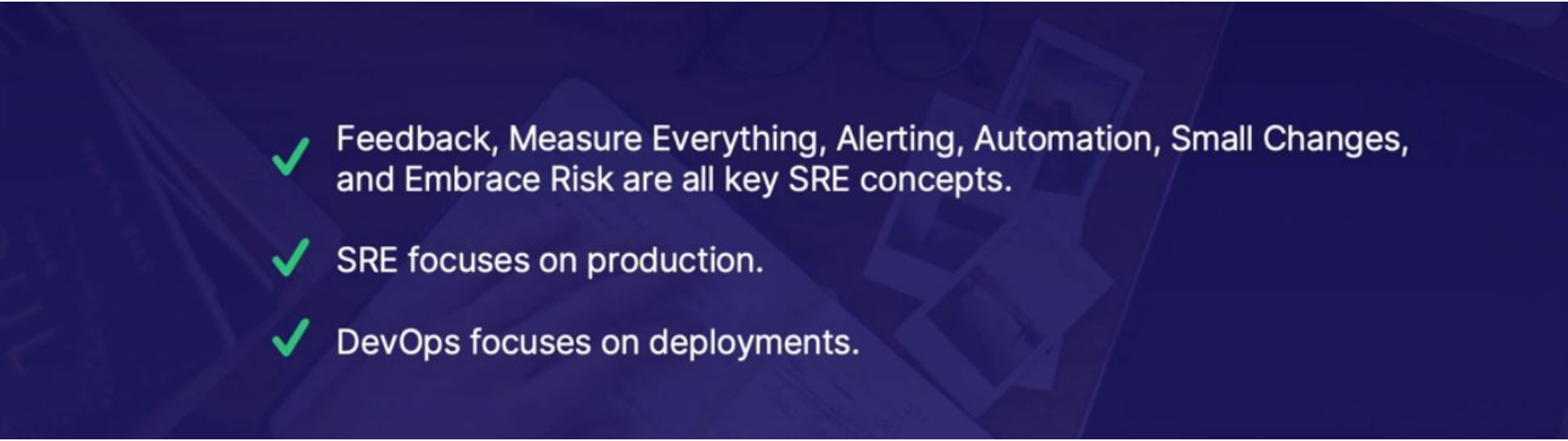


Key Concepts

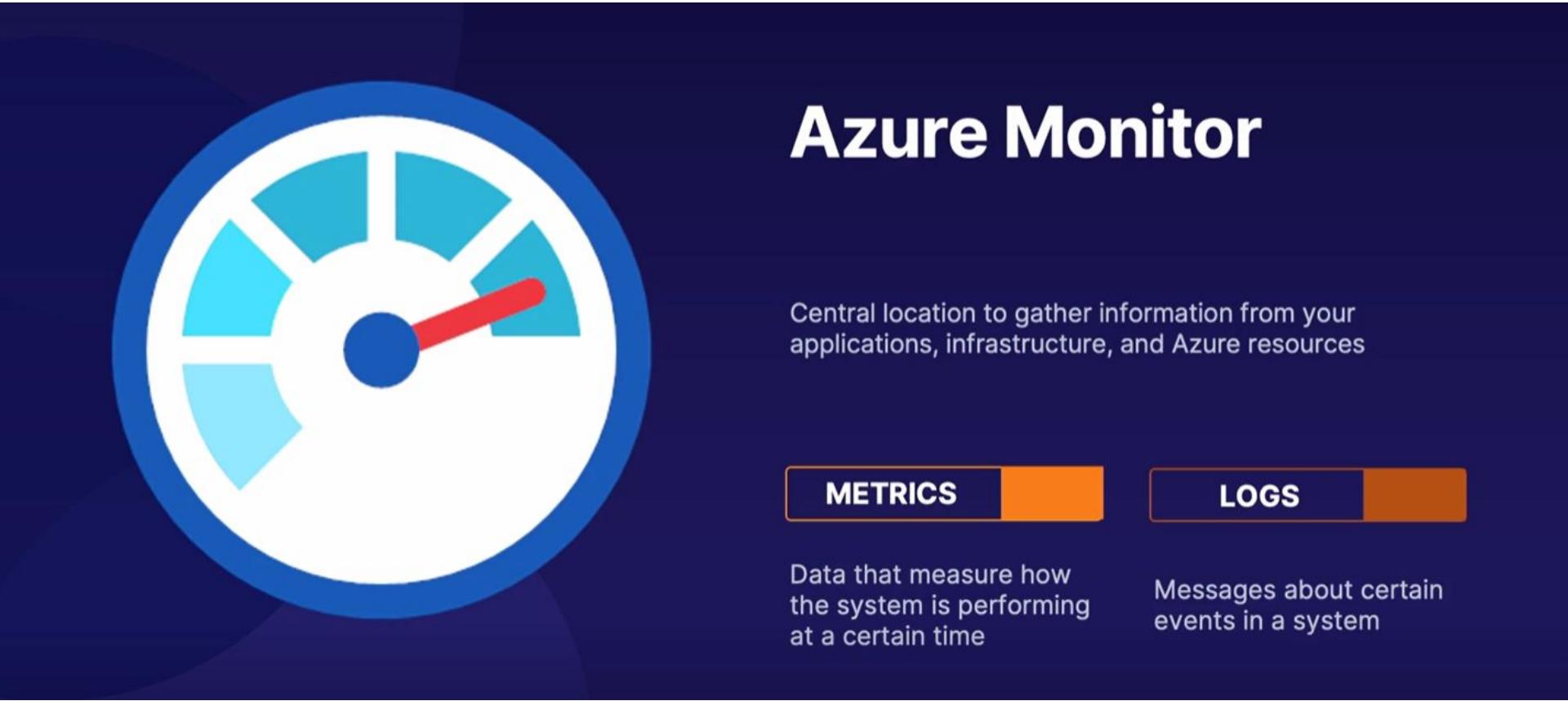


Key Concepts

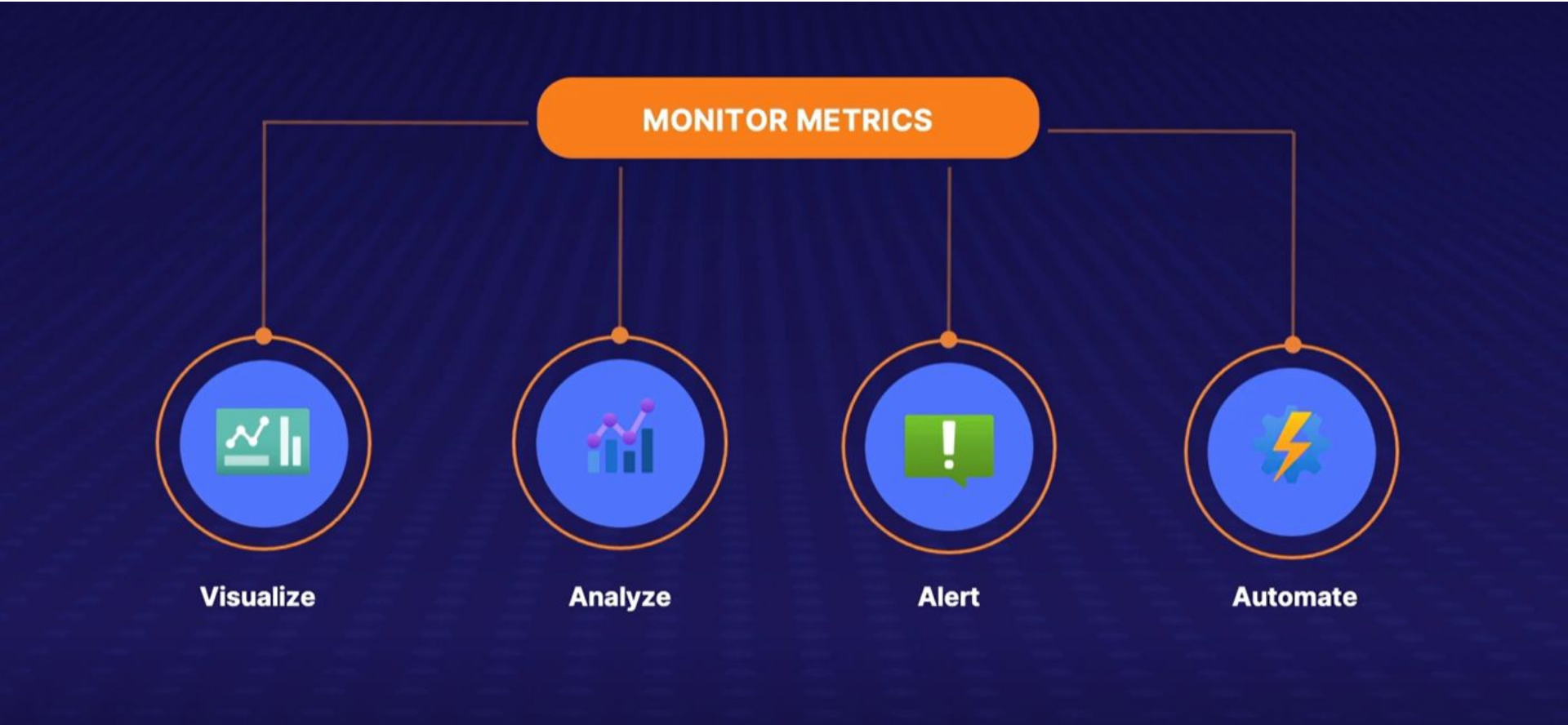
Key Concepts



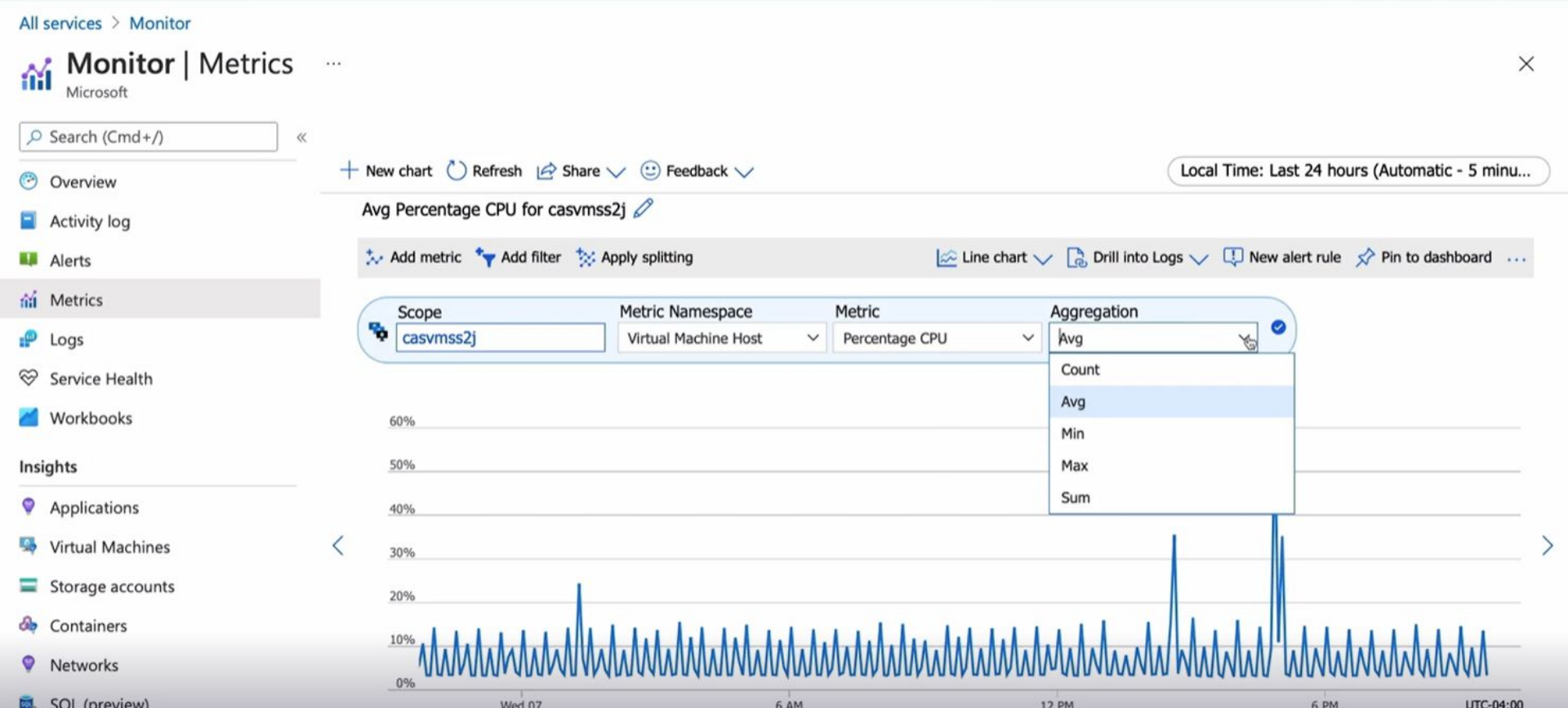
Azure Monitor



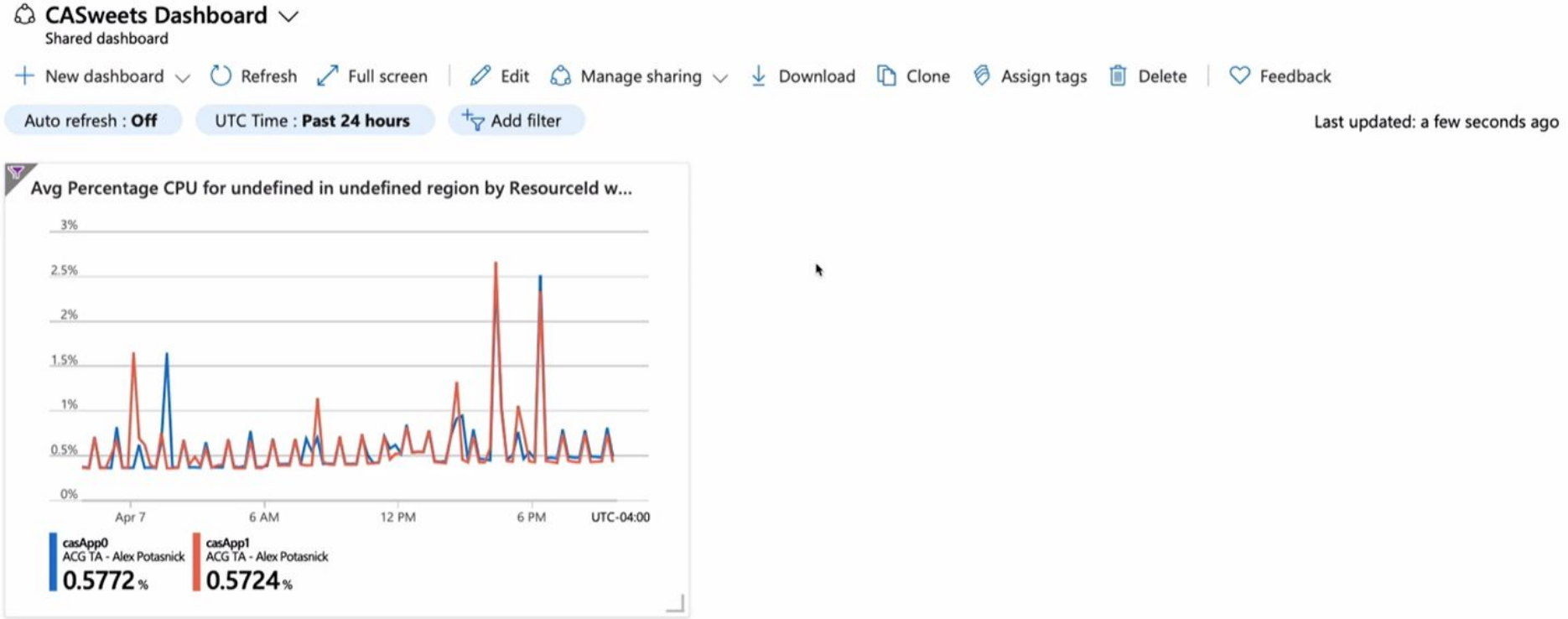
Azure Monitor



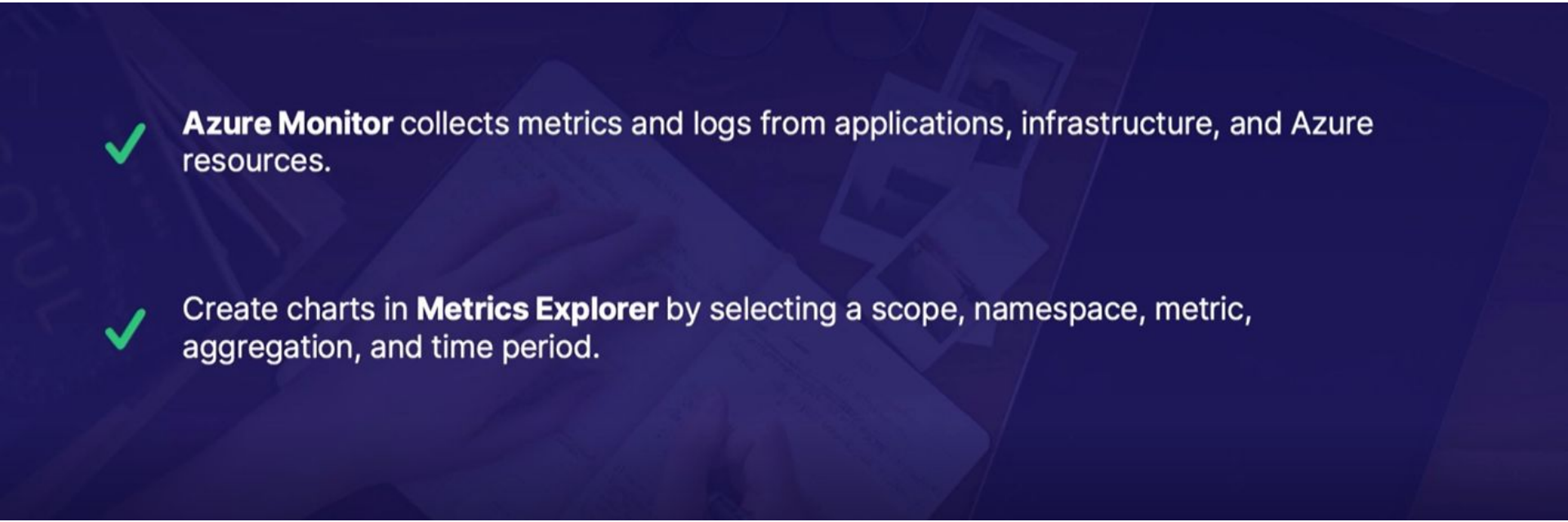
Azure Monitor



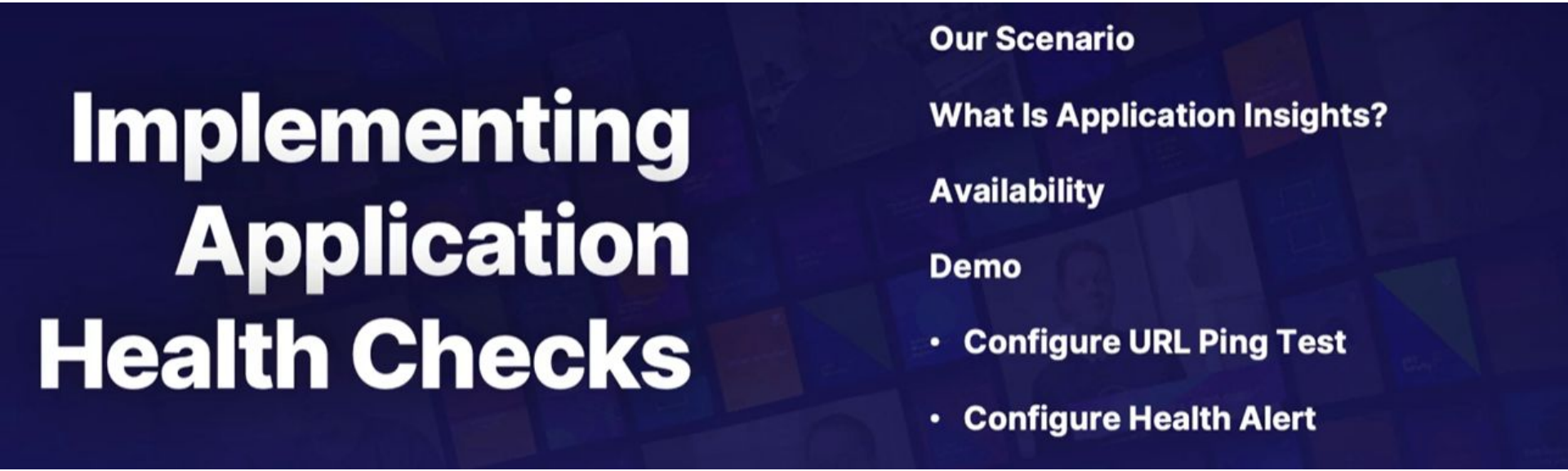
Azure Monitor



Azure Monitor



Implementing Application Health Checks



Our Scenario

Ctrl Alt Sweets

How is the team handling this?

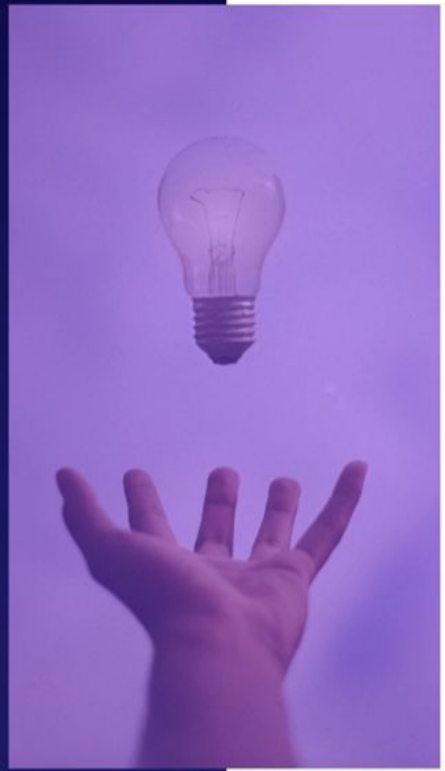
- Using Azure Monitor Metrics for infrastructure health
- Manual checks for application health
- Looking for ways to automate

Availability

What Is Application Insights?

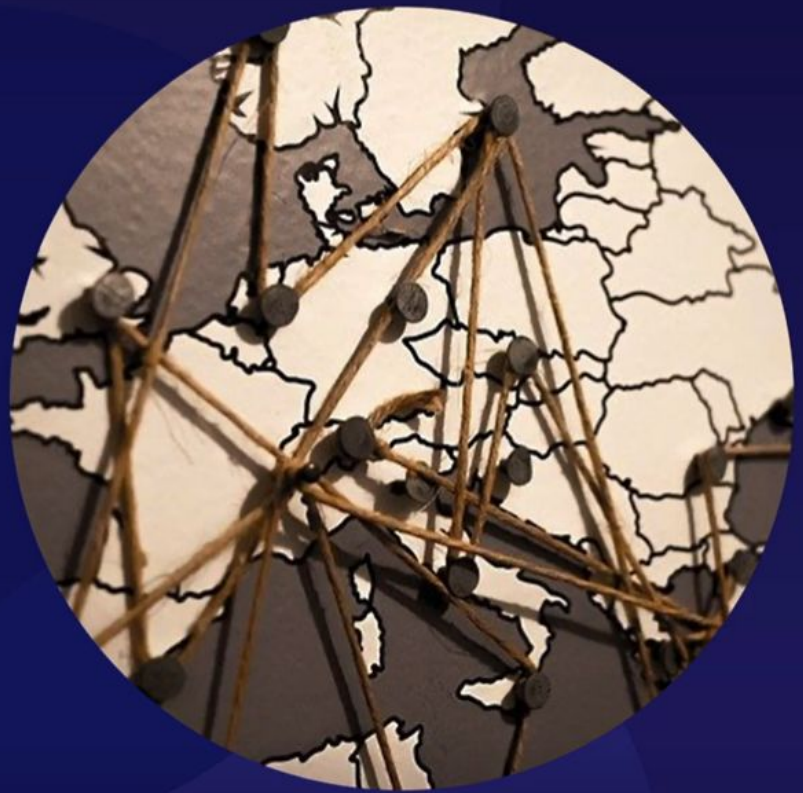
Specifically designed to help you understand the health and usage of your application

How is the application doing and how is it being used?



- Application exceptions
- Application dependencies
- Performance information

Availability

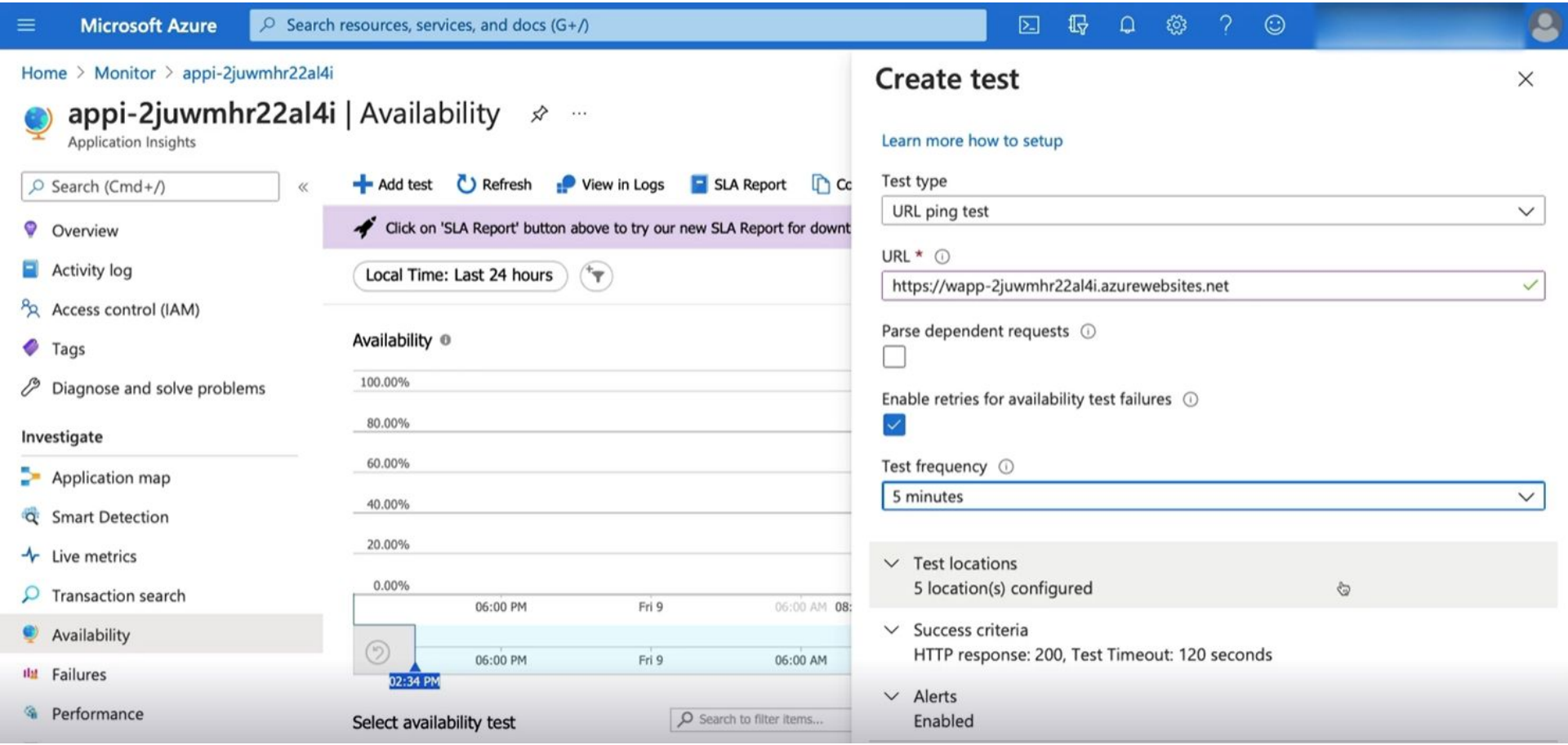


URL Ping Test

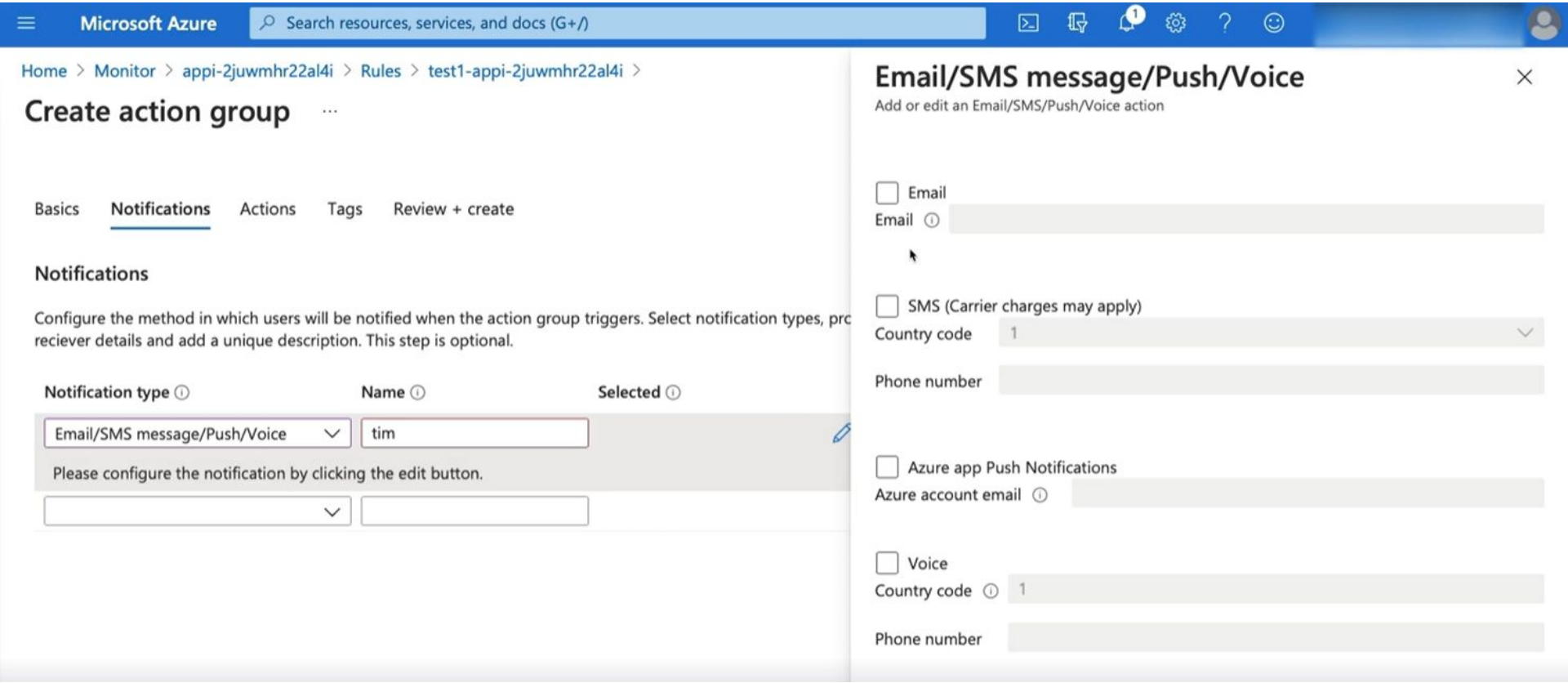
Send an HTTP request to a specific URL to test availability

- Lets you know the request duration and success rate
- Dependent requests test availability of files on a webpage
- Enable retries is recommended
- Test frequency
- Minimum of 5 and maximum of 16 test locations
- Can create alerts

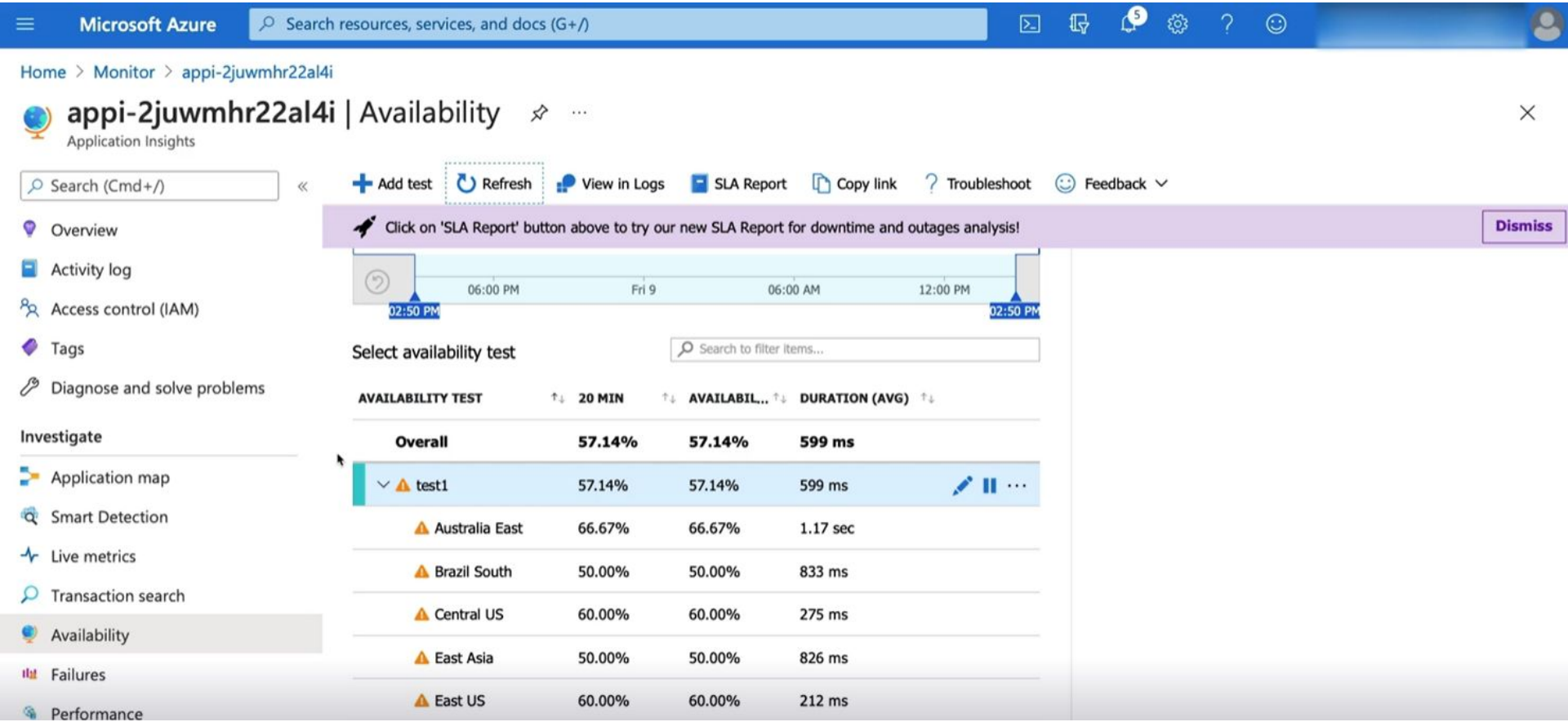
Create a test



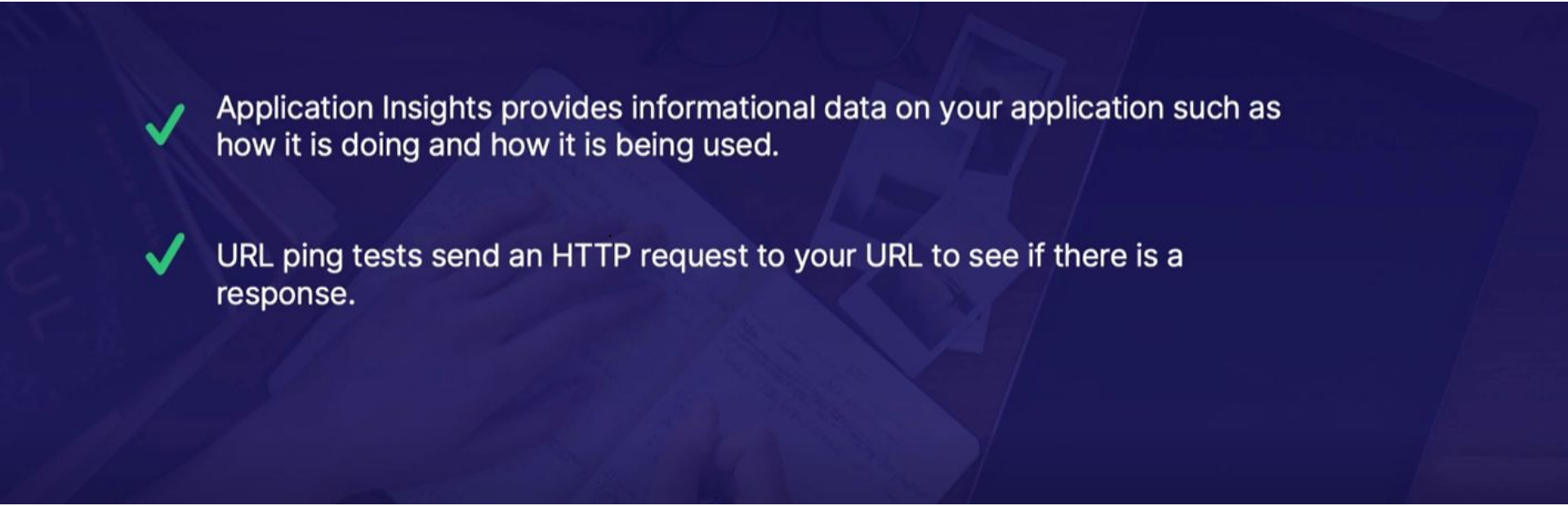
Create an action group



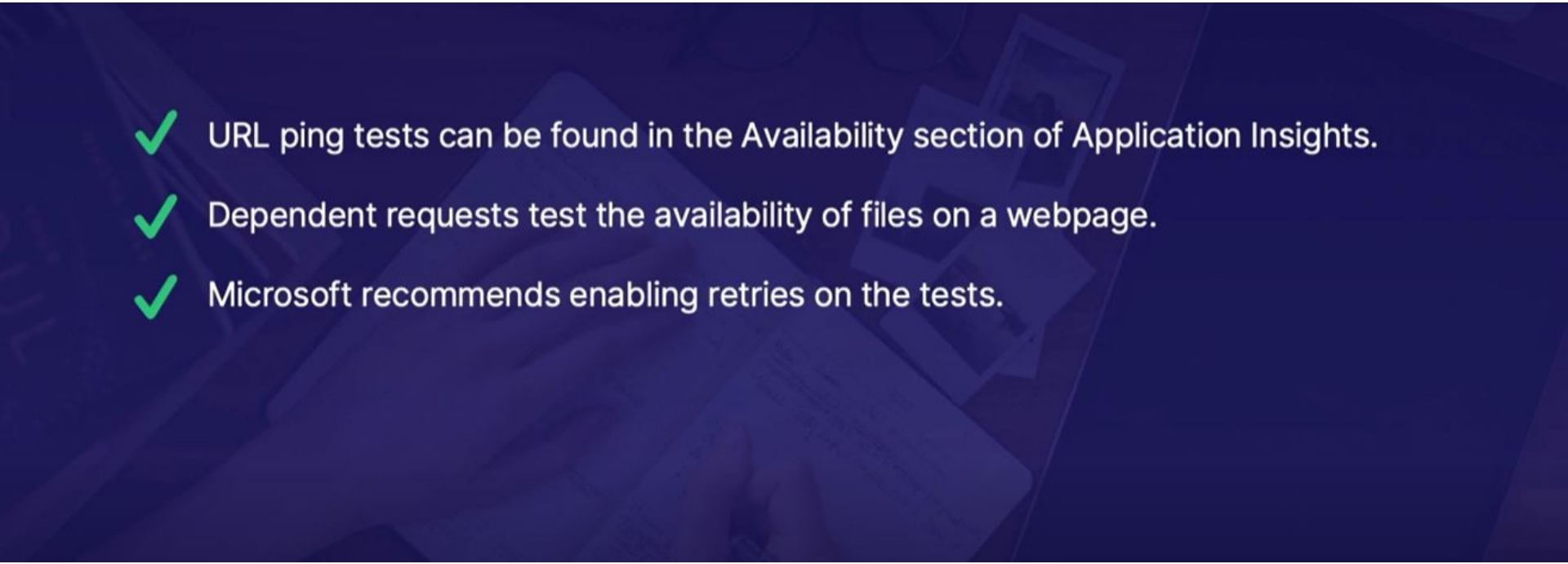
Failing availability test



Implementing Application Health Checks



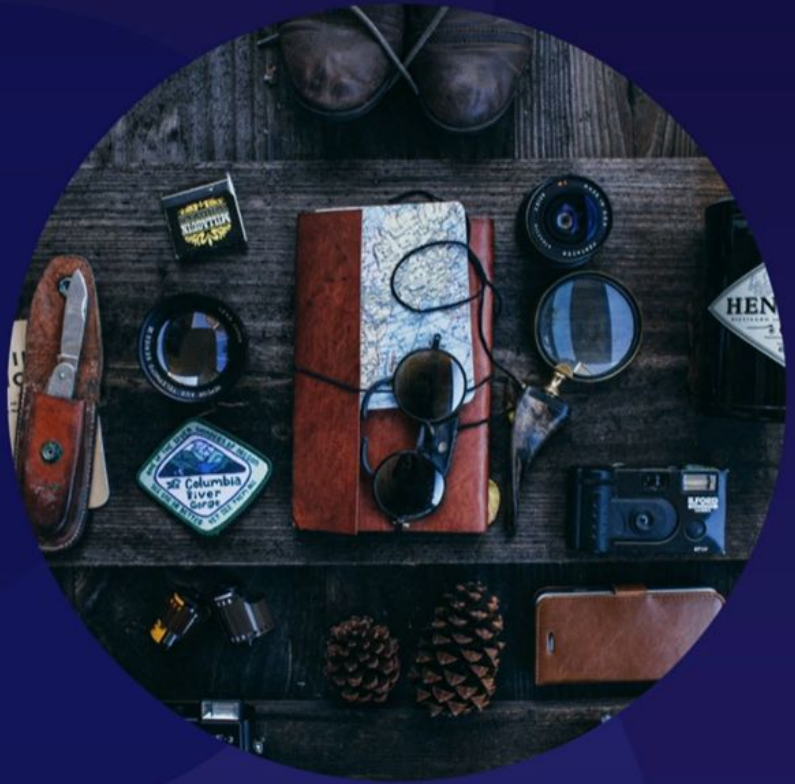
Implementing Application Health Checks



Implementing Application Health Checks

- Once a test has been created, you can configure alerts to be sent on failures.
- Create Action Groups to tell the alert who to notify and how.

Exploring System Load and Failure Conditions



Everything fails sometimes

Whether it's hardware or software, it will eventually fail at some point.

The goal is to be prepared for any eventuality before being notified by the end user.

Exploring System Load and Failure Conditions

Find Single Points of Failure

- Fault points: any place in the architecture that can fail
- Fault modes: all the ways a fault point can fail

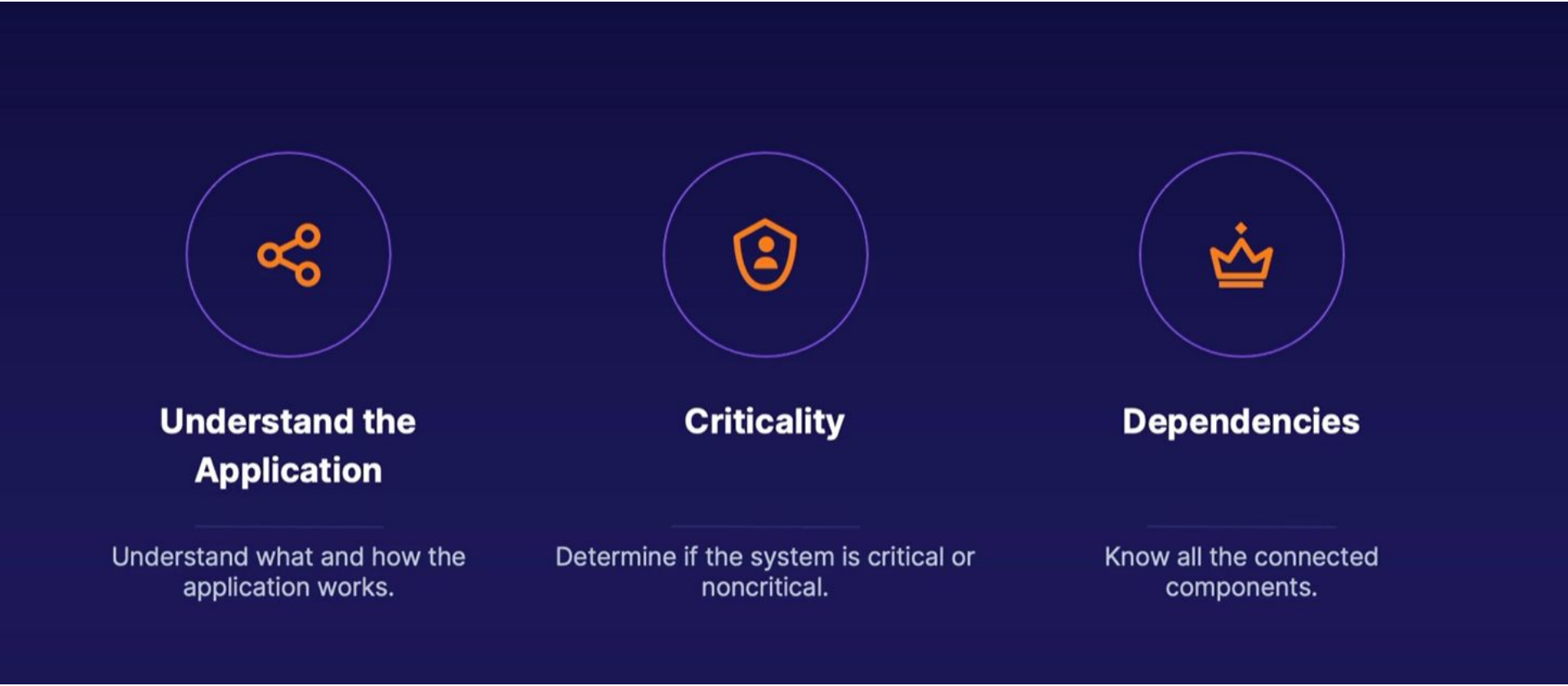
Rate Risk and Severity

How likely it will fail and what is the impact

Determine Response

How the application will respond and recover from a failure

Exploring System Load and Failure Conditions



Exploring System Load and Failure Conditions

Exploring System Load and Failure Conditions

Performance Testing

1. **Load Testing**
   Test the application can handle normal to heavy load.

2. **Stress Testing**
   Attempt to overload the system.

3. **Spikes**
   A best practice is to include a buffer to account for random spikes.

Understanding Baseline Metrics

Our Scenario

Ctrl Alt Sweets

How is the team handling this?

- Configuration drift in Testing environment from Production
- “Normal” system behavior is different for each environment
- Difficult to do performance testing without a proper baseline

Tom

TEST/QA

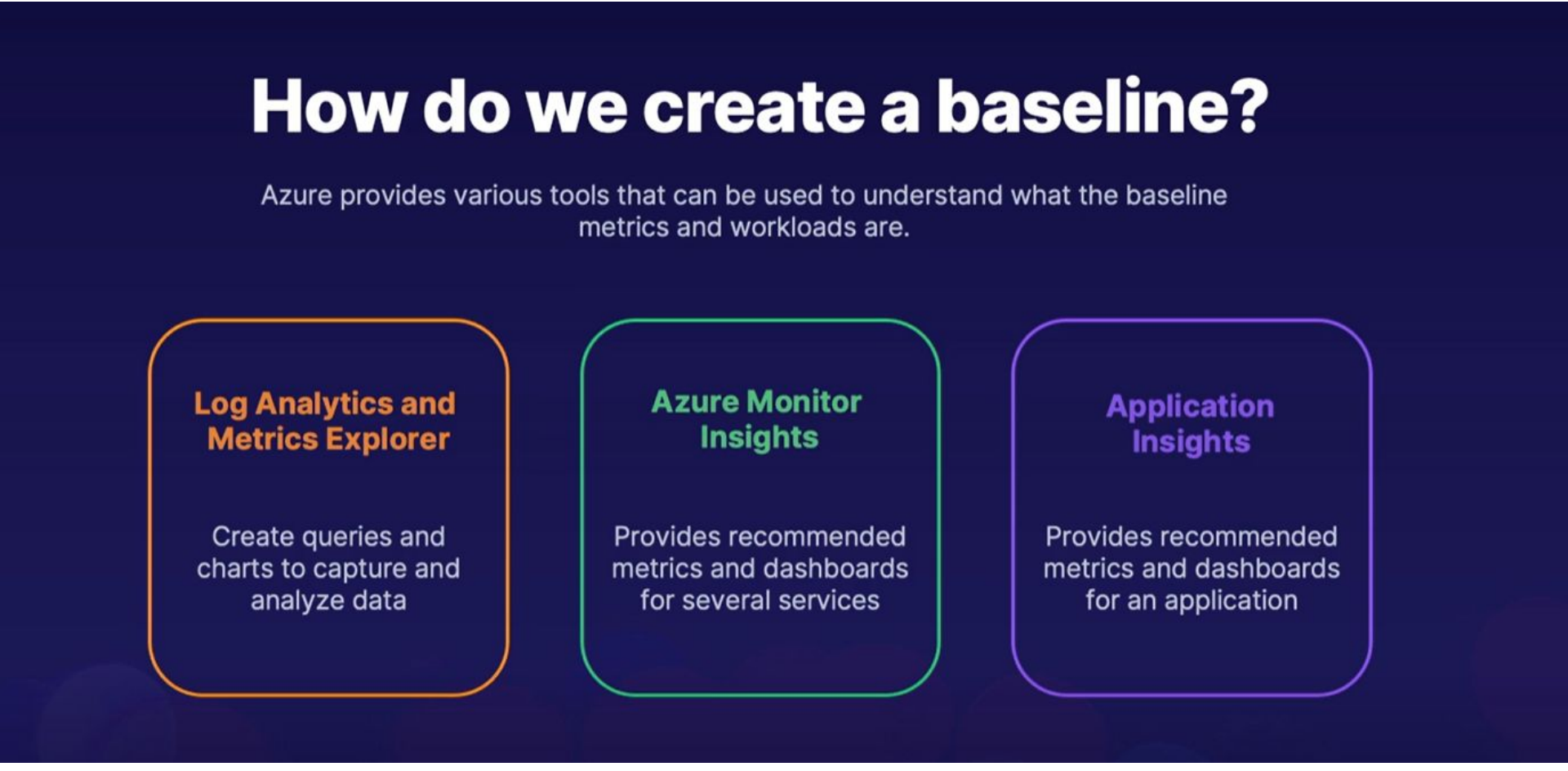
Understanding Baseline Metrics

Why create a baseline?

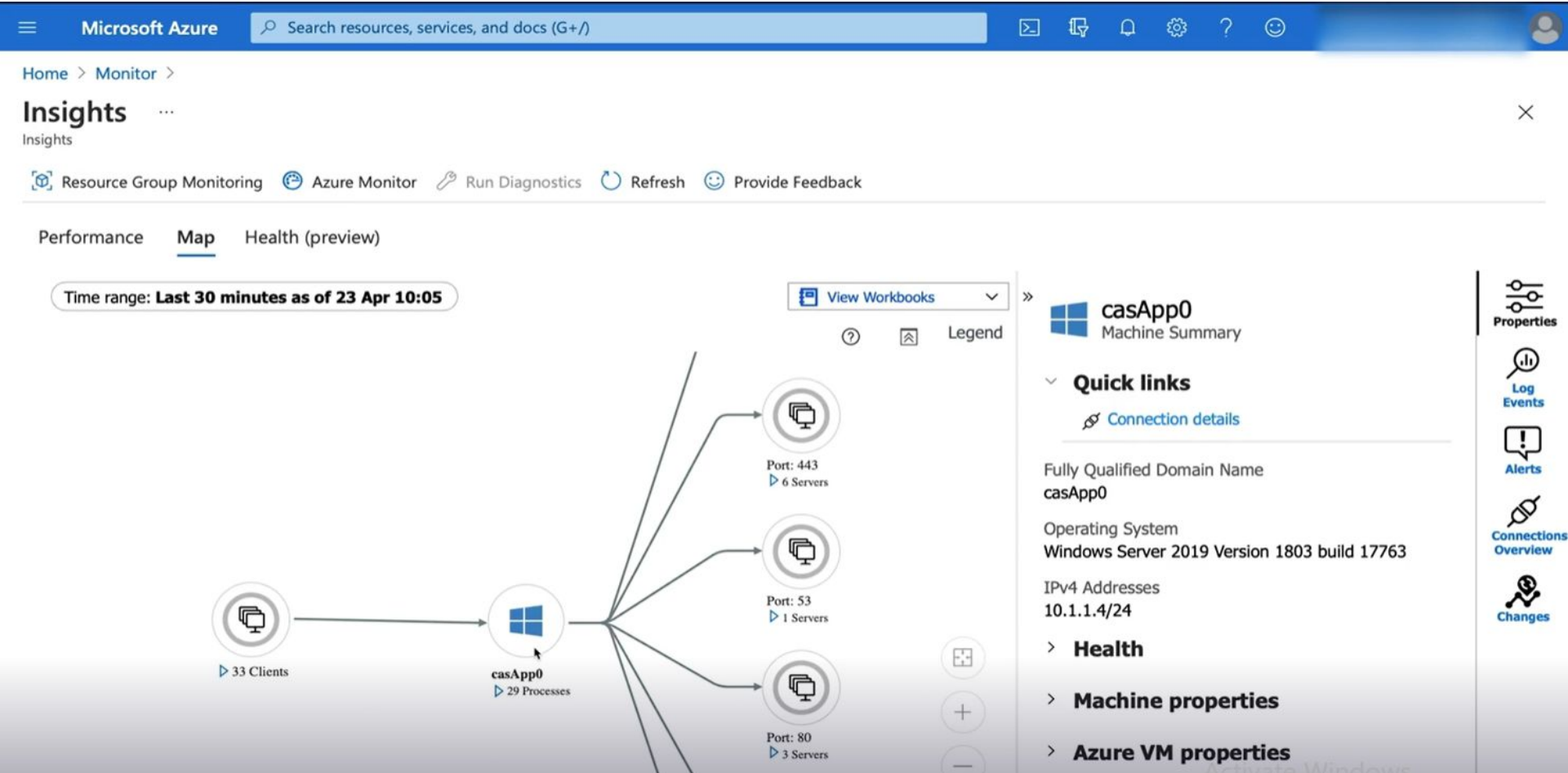
Once we have a baseline, we can understand what a healthy state is.

Any deviations from the baseline can be alerted on and can also be used as starting points for improvements.

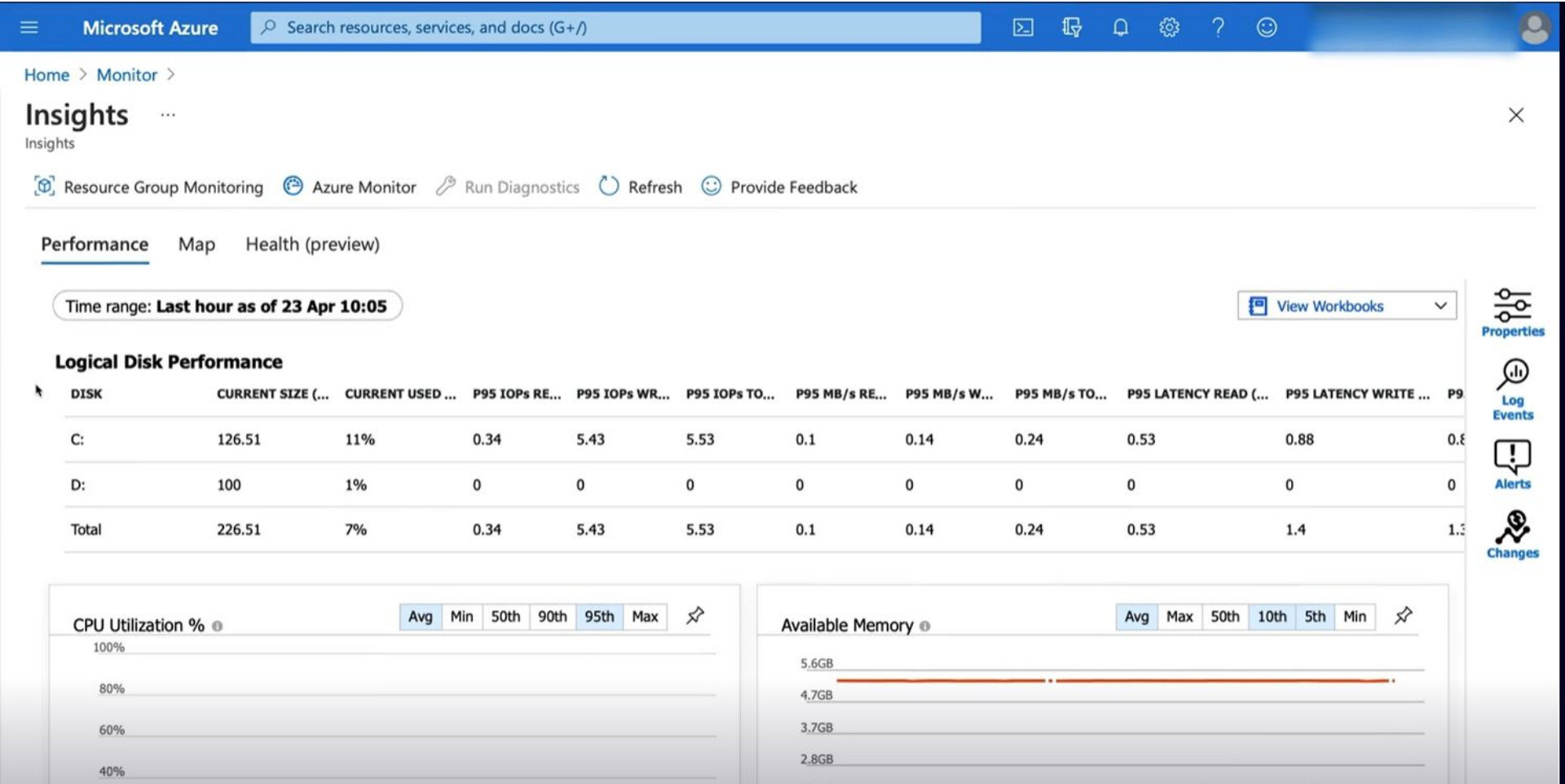
Understanding Baseline Metrics



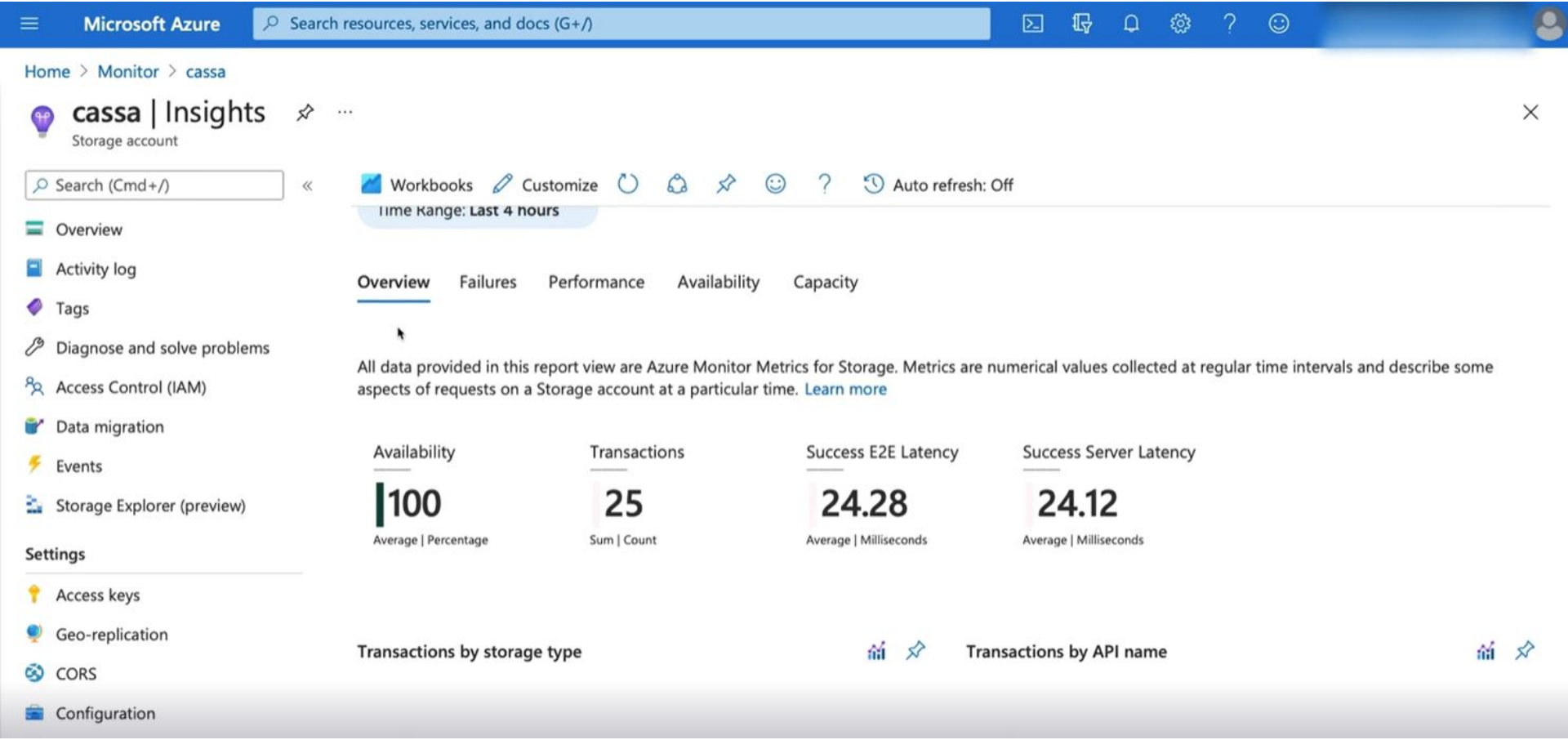
Understanding Baseline Metrics



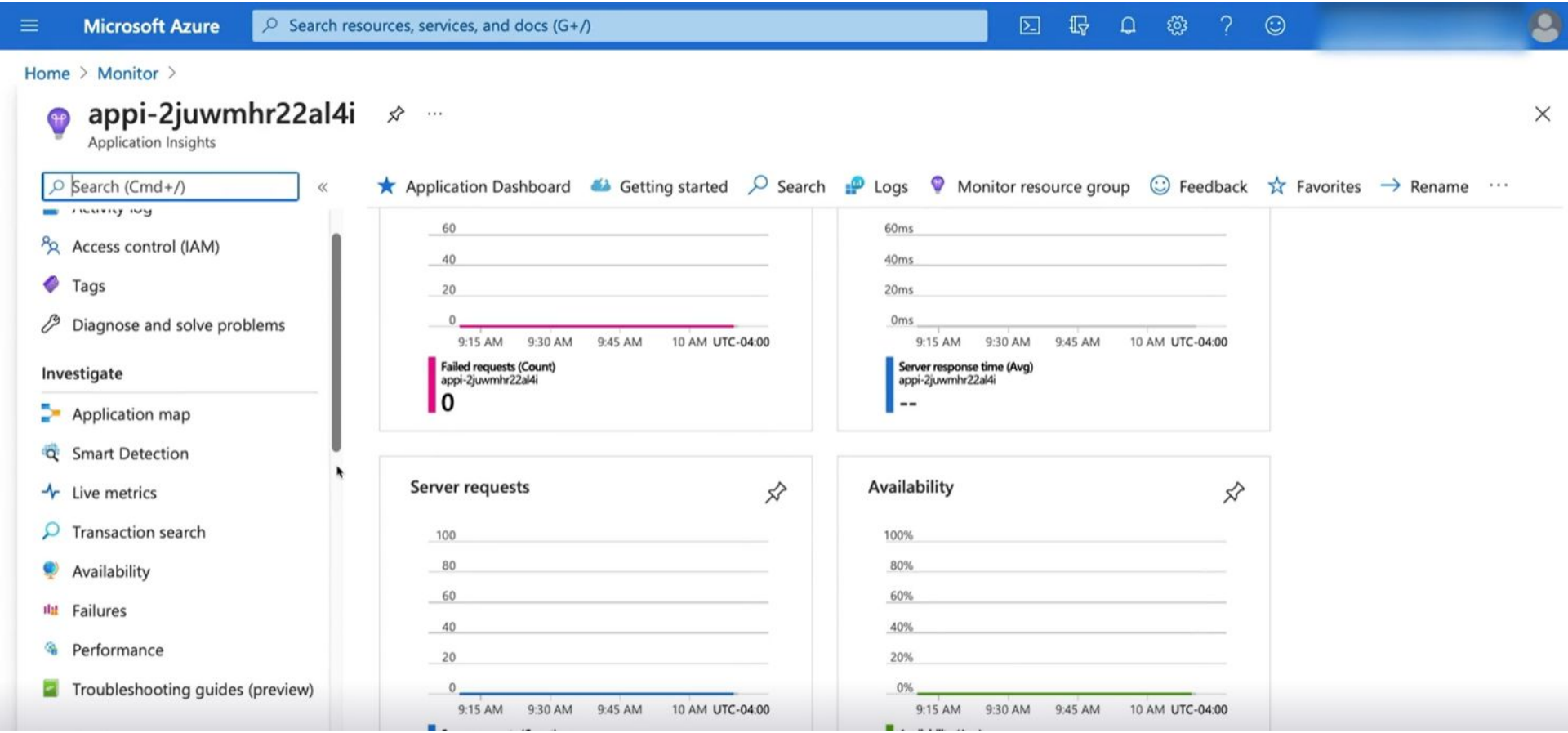
Understanding Baseline Metrics



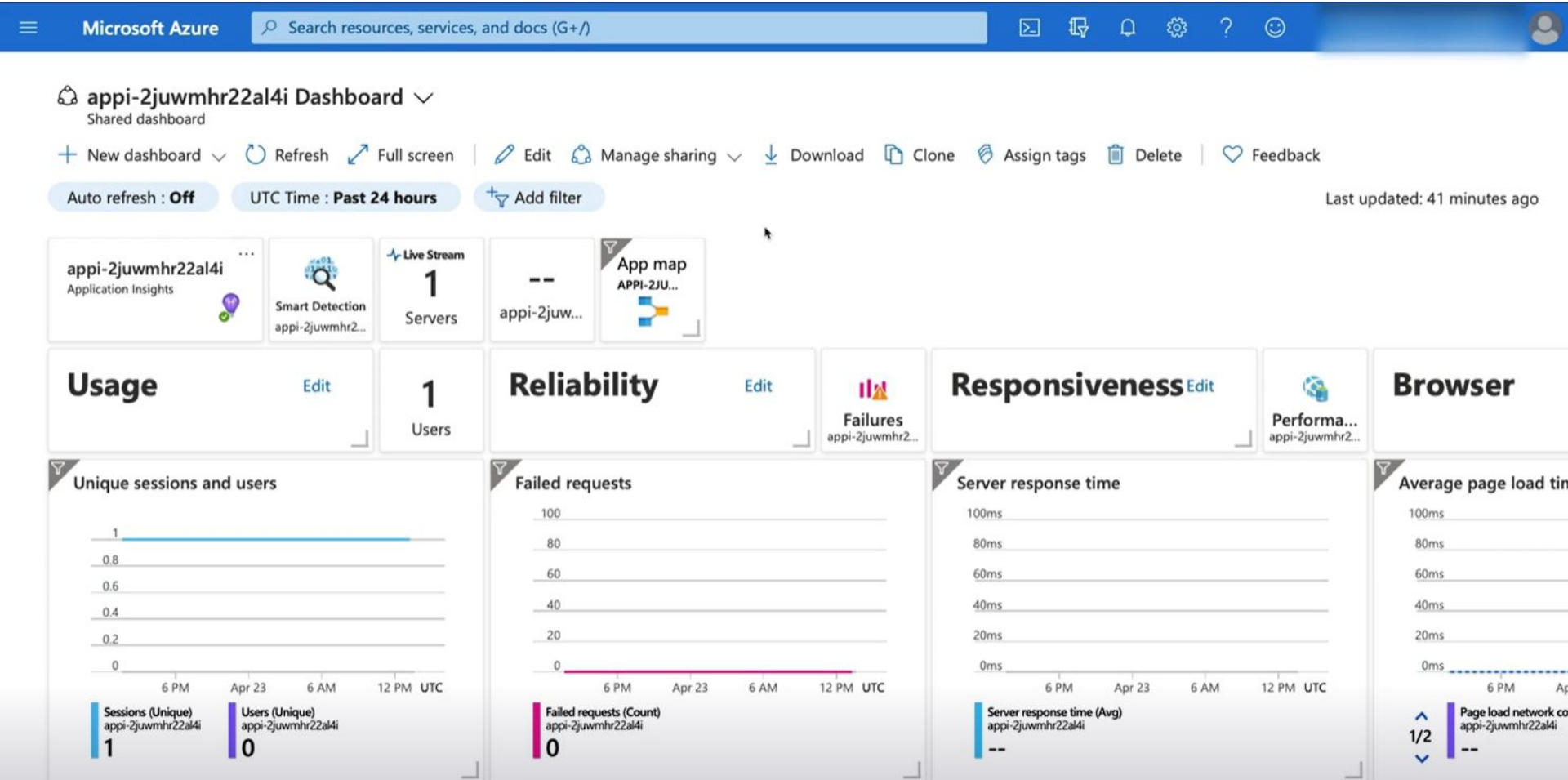
Understanding Baseline Metrics



Understanding Baseline Metrics



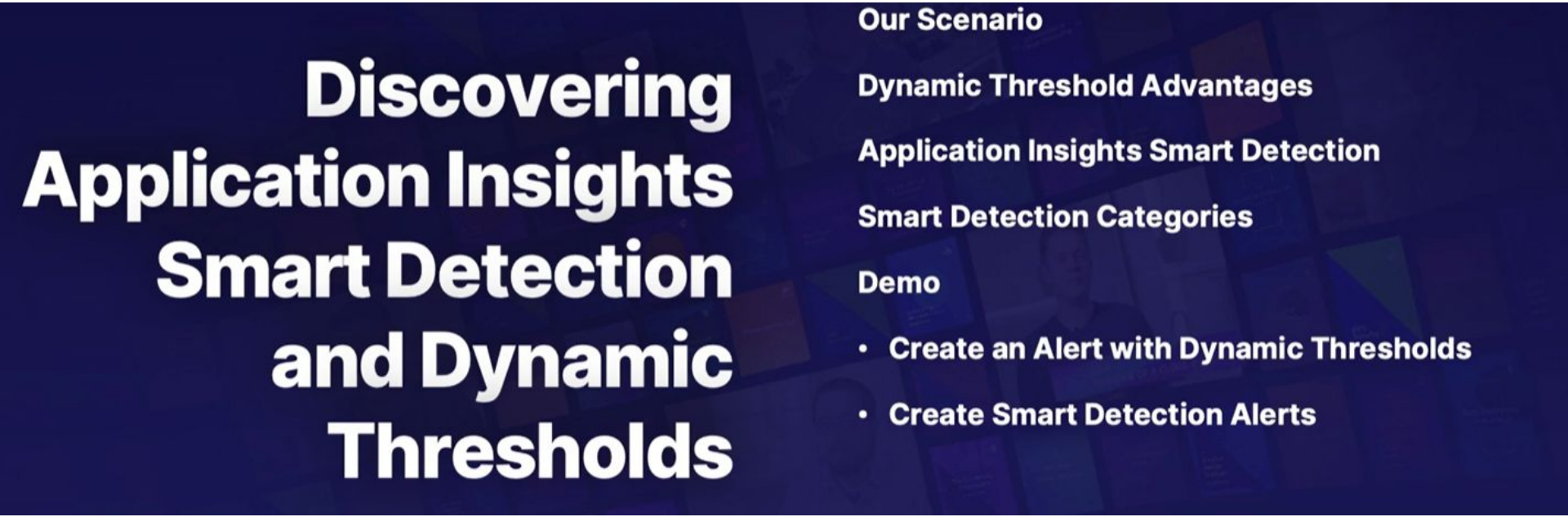
Understanding Baseline Metrics



Understanding Baseline Metrics

- A baseline can help you identify when a system is not in a healthy state so that alerts and improvements can be implemented.
- Azure Monitor Insights provides recommended charts and metrics for Azure resources.
- Application Insights provides recommended charts and metrics for applications.

Discovering Application Insights Smart Detection and Dynamic Thresholds



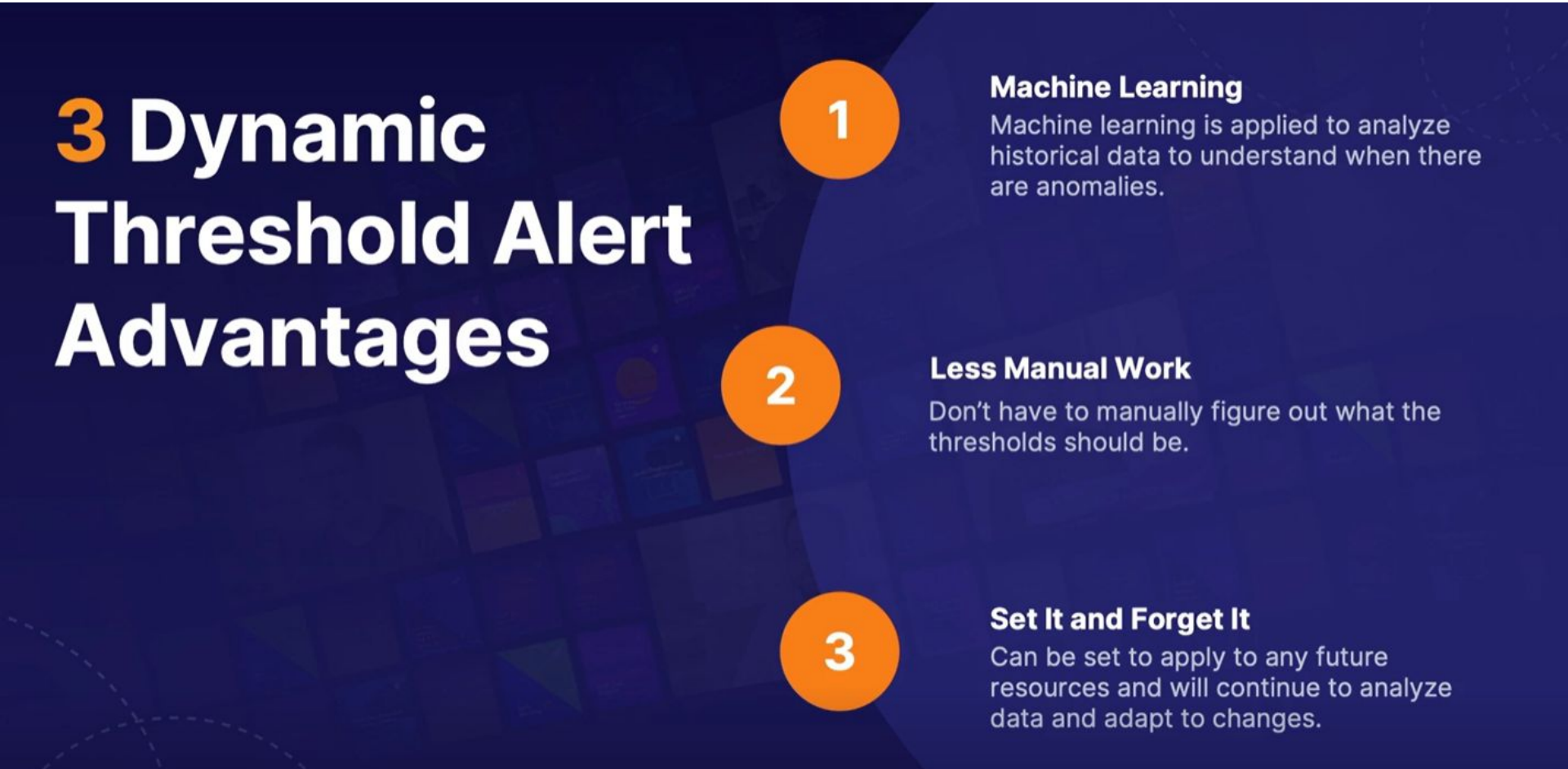
Our Scenario

Ctrl Alt Sweets

How is the team handling this?

- Now using baselines for their performance testing
- Looking into using those health baselines to create alerts
- Want alerts to be adaptable to future changes that might alter the baseline

Dynamic Thresholds



Application Insights Smart Detection

Application Insights Smart Detection

1. **Machine Learning**
   Analyze telemetry data to detect anomalies.

2. **Built-In**
   Once there is enough time and data to analyze, it will be configured automatically.

3. **Alerting**
   Provides information based on findings as well as information as to why there might be an issue.

Smart Detection categories

Categories

- **Failures category:** Needs a minimum amount of data and 24 hours to start
- **Performance category:** Needs a minimum amount of data and 8 days to start

Failures

Failure Rates

Figures out what the expected **number** of failures should be

Continuous Monitoring

Alerts in near real time

Alert Context

Provides information as to why it might have failed

Performance

Page Responses

If page takes too long to load or if operations or responses from dependencies are too slow

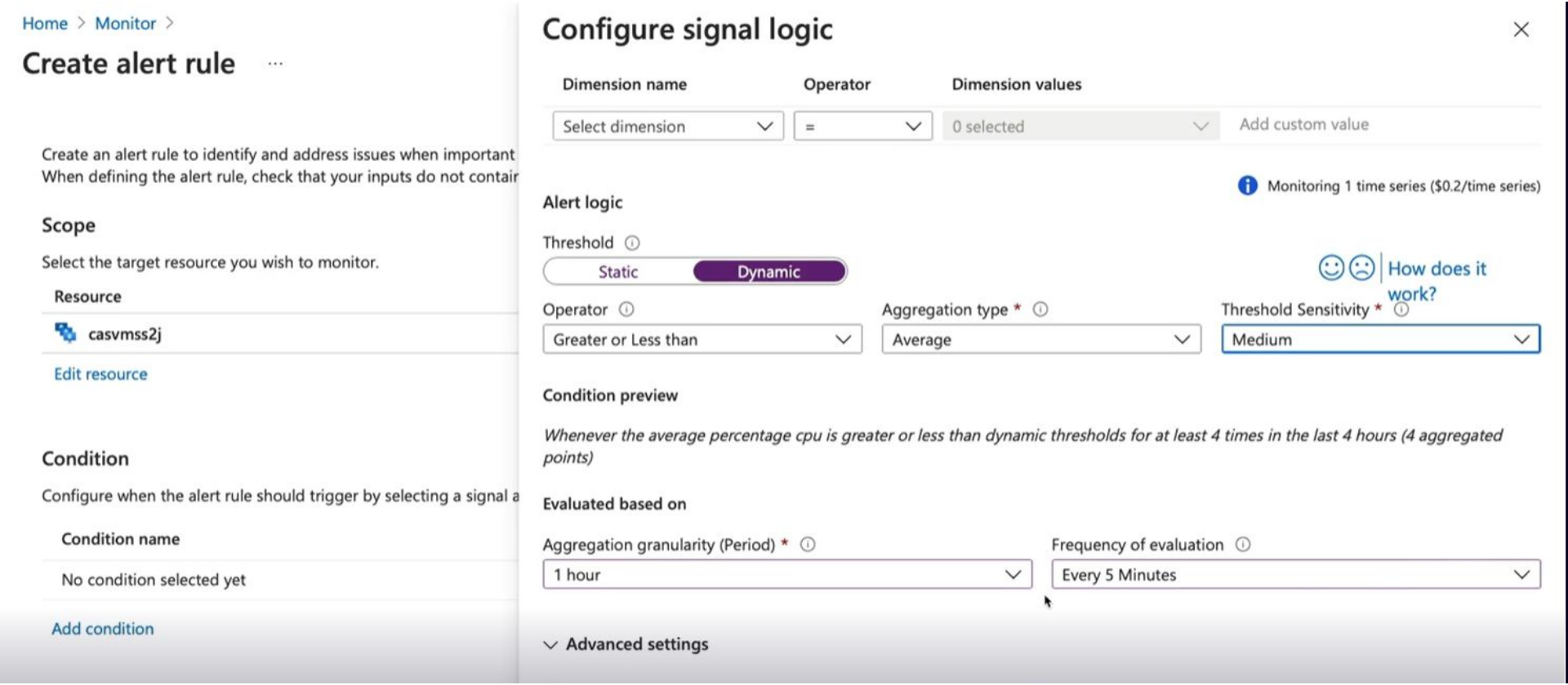
Daily Analysis

Sends a notification once a day

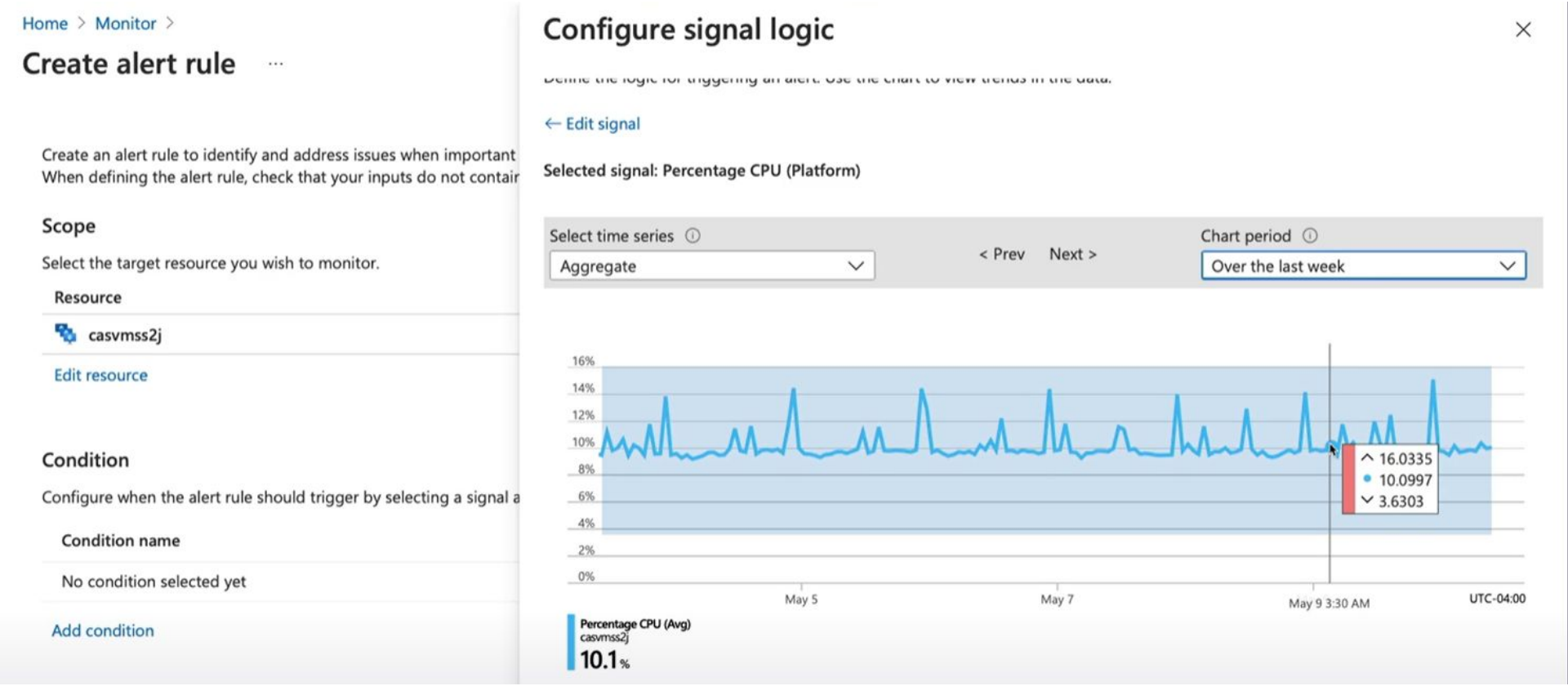
Alert Context

Provides information as to why it is running slow

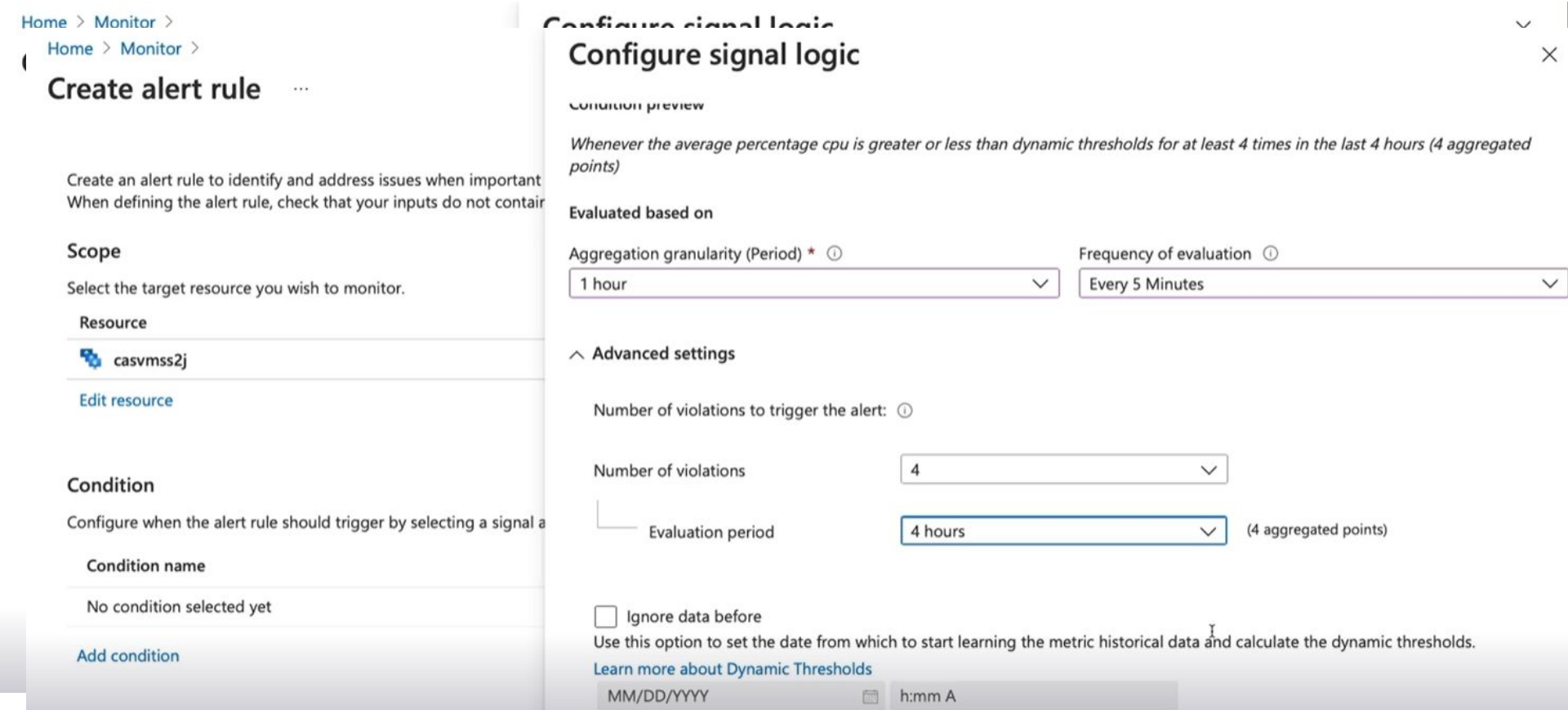
Discovering Application Insights Smart Detection and Dynamic Thresholds



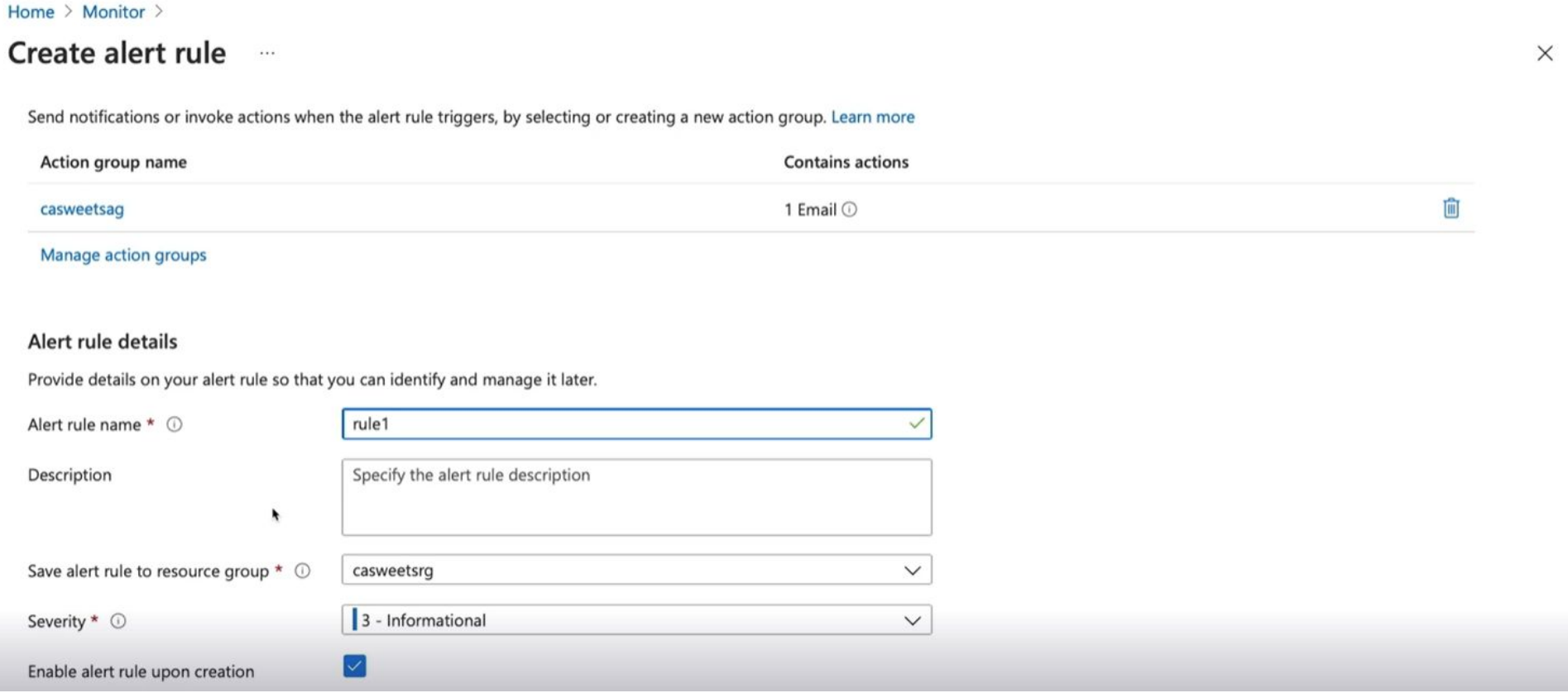
Discovering Application Insights Smart Detection and Dynamic Thresholds



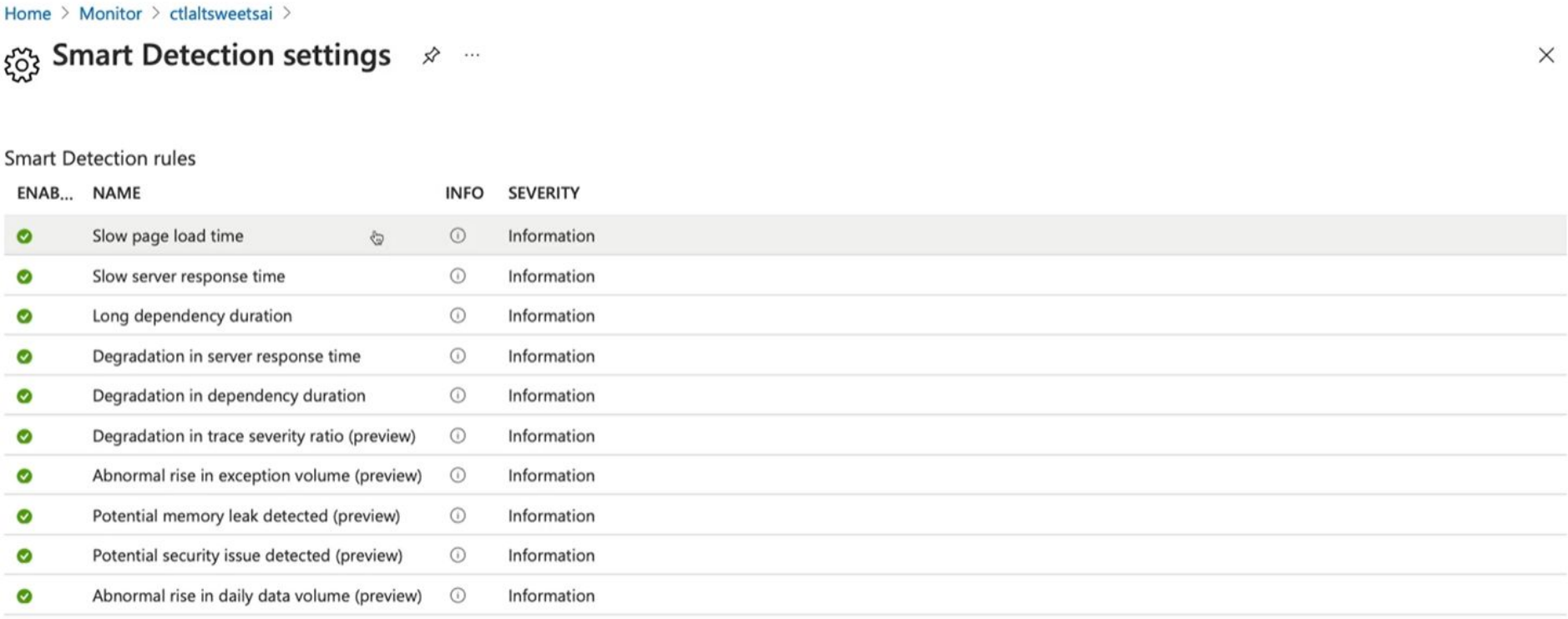
Discovering Application Insights Smart Detection and Dynamic Thresholds



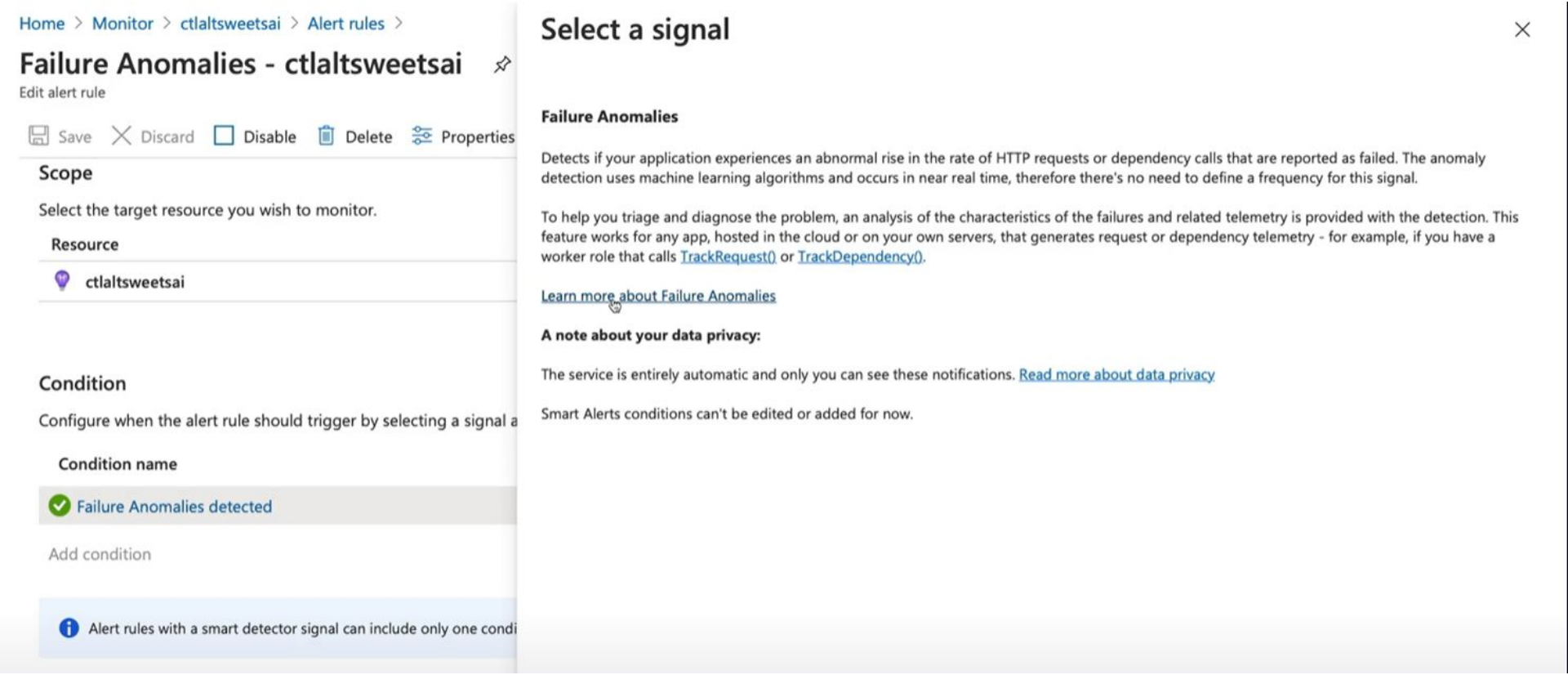
Discovering Application Insights Smart Detection and Dynamic Thresholds



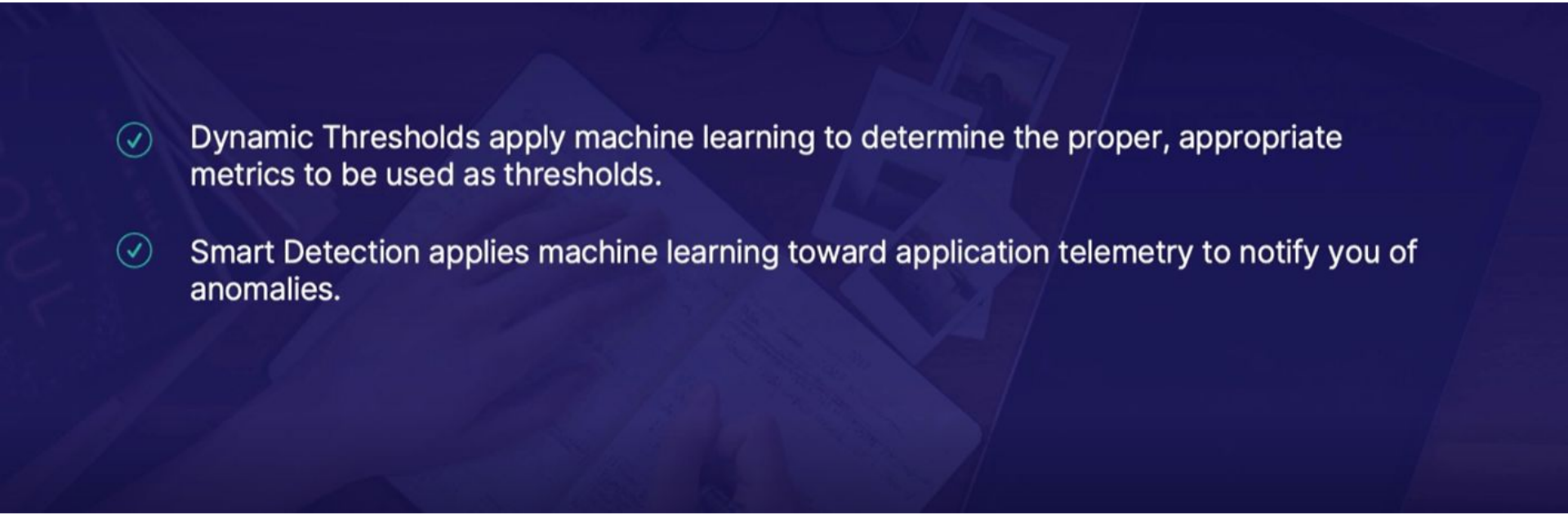
Smart Detection rules for apps



Smart Detection rules for apps



Discovering Application Insights Smart Detection and Dynamic Thresholds



Deciding Which Dependencies to Set Alerts On

Deciding Which Dependencies to Set Alerts On

What Is a Dependency?

One component relies on another component to perform some function.

Dependencies exist because each component brings something unique.

- HTTP, database, and file system calls
- Internal vs. External
- Strong vs. Weak

Deciding Which Dependencies to Set Alerts On

Application Insights Dependency Tracking

1. **Track and Monitor**

   Helps identify strong dependencies by tracking and monitoring calls

2. **Automatic Tracking with .NET/.Net Core**

   Tracking is configured by default if using the .NET and .NET Core SDK for Application Insights

3. **Manual Dependency Tracking**

   Configured using the TrackDependency API

4. **AJAX from Web Pages**

   Application Insights JavaScript SDK will collect AJAX calls automatically

Which Dependencies are tracked in Application Insights

Automatic

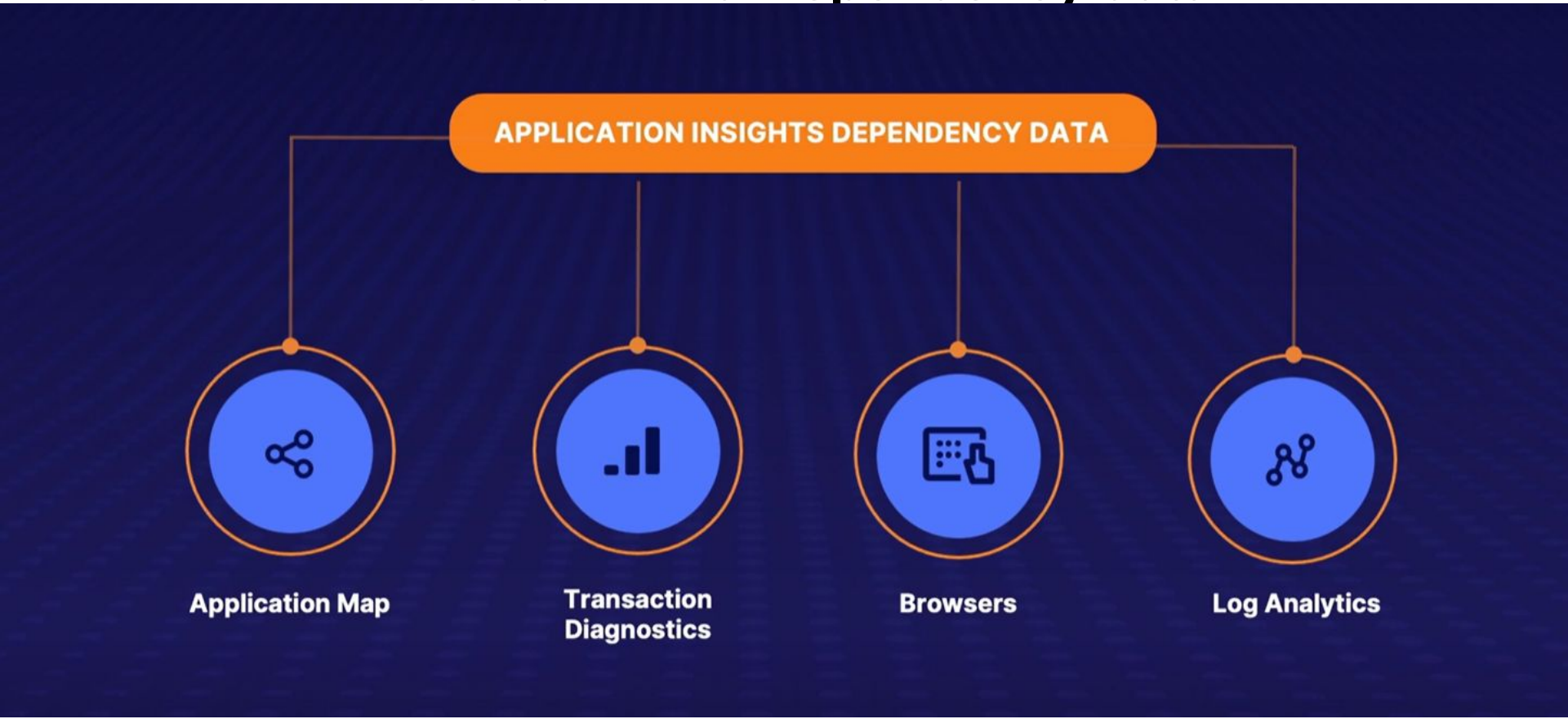
- HTTP and HTTPS calls
- WCF if using the HTTP bindings
- SQL calls using SqlClient
- Azure Storage with Azure Storage Client
- EventHub Client SDK
- ServiceBus Client SDK
- Azure Cosmos DB with HTTP/HTTPS

VS

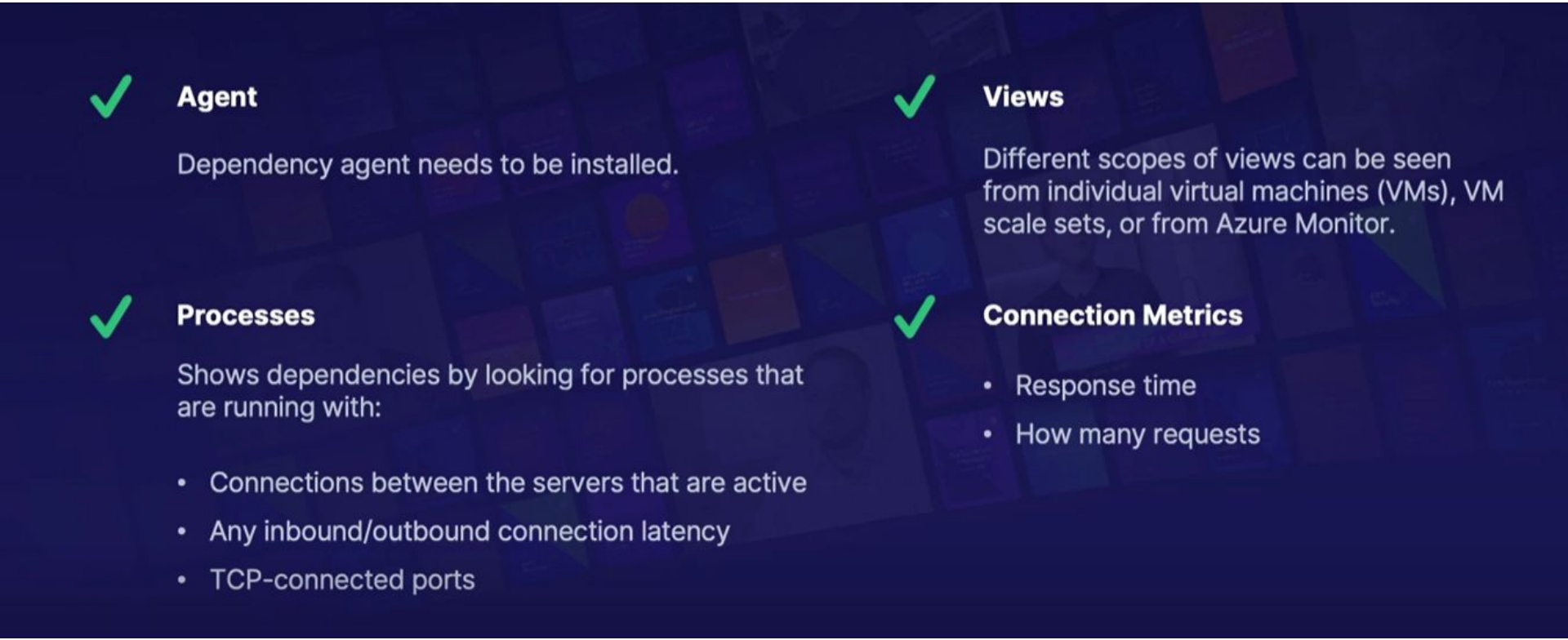
Manual

- Cosmos DB with TCP
- Redis

Where can i find Dependency data?



Deciding Which Dependencies to Set Alerts On



Agent

Dependency agent needs to be installed.

Processes

Shows dependencies by looking for processes that are running with:

- Connections between the servers that are active
- Any inbound/outbound connection latency
- TCP-connected ports

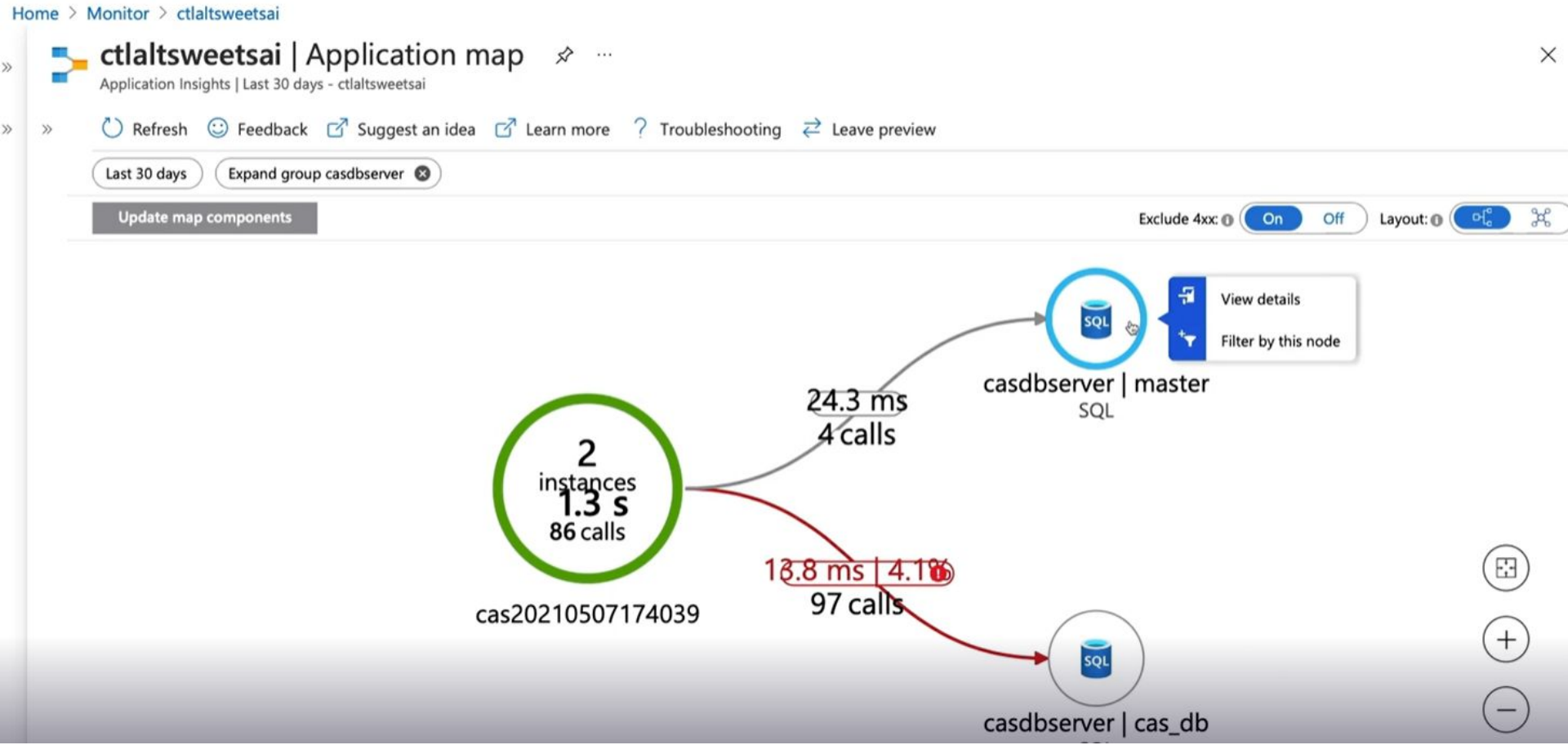
Views

Different scopes of views can be seen from individual virtual machines (VMs), VM scale sets, or from Azure Monitor.

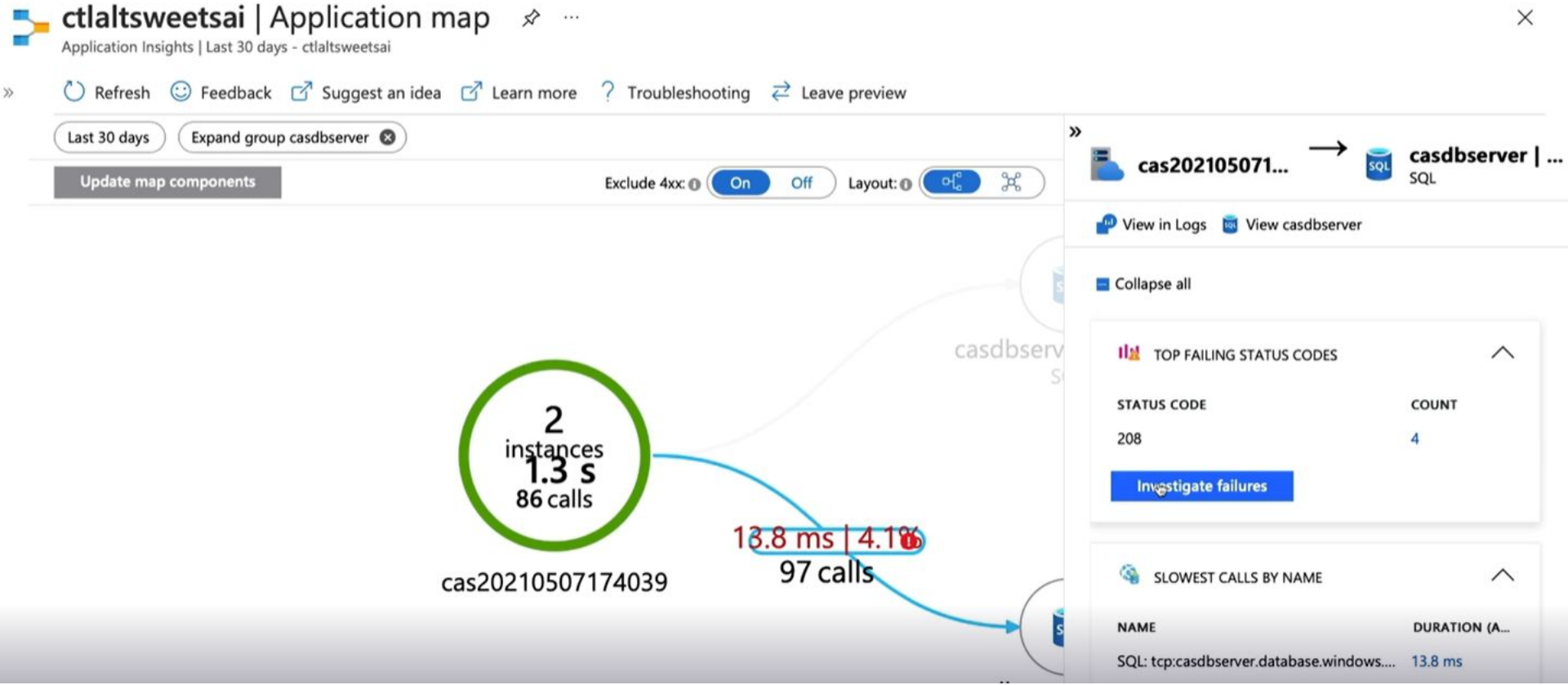
Connection Metrics

- Response time
- How many requests

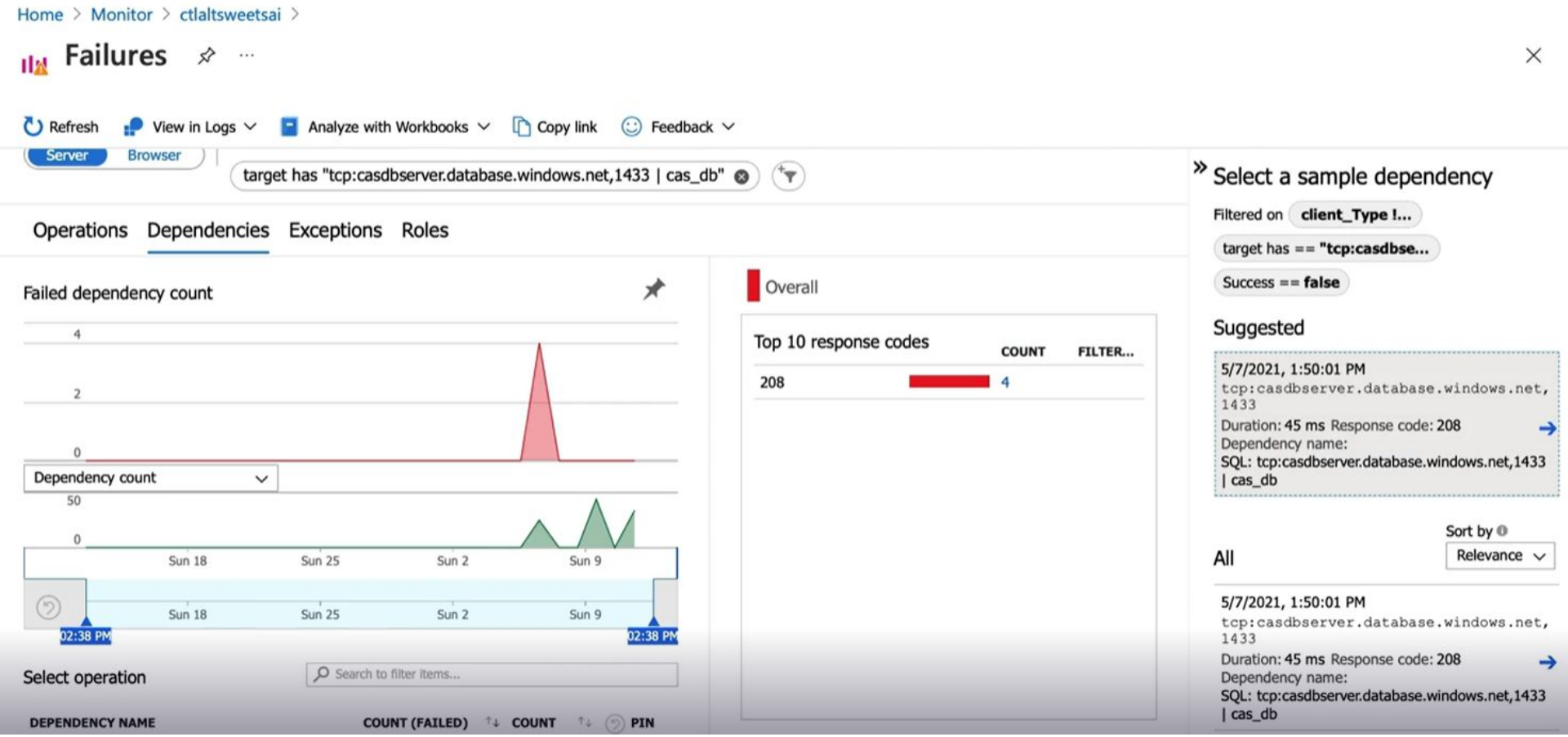
Deciding Which Dependencies to Set Alerts On



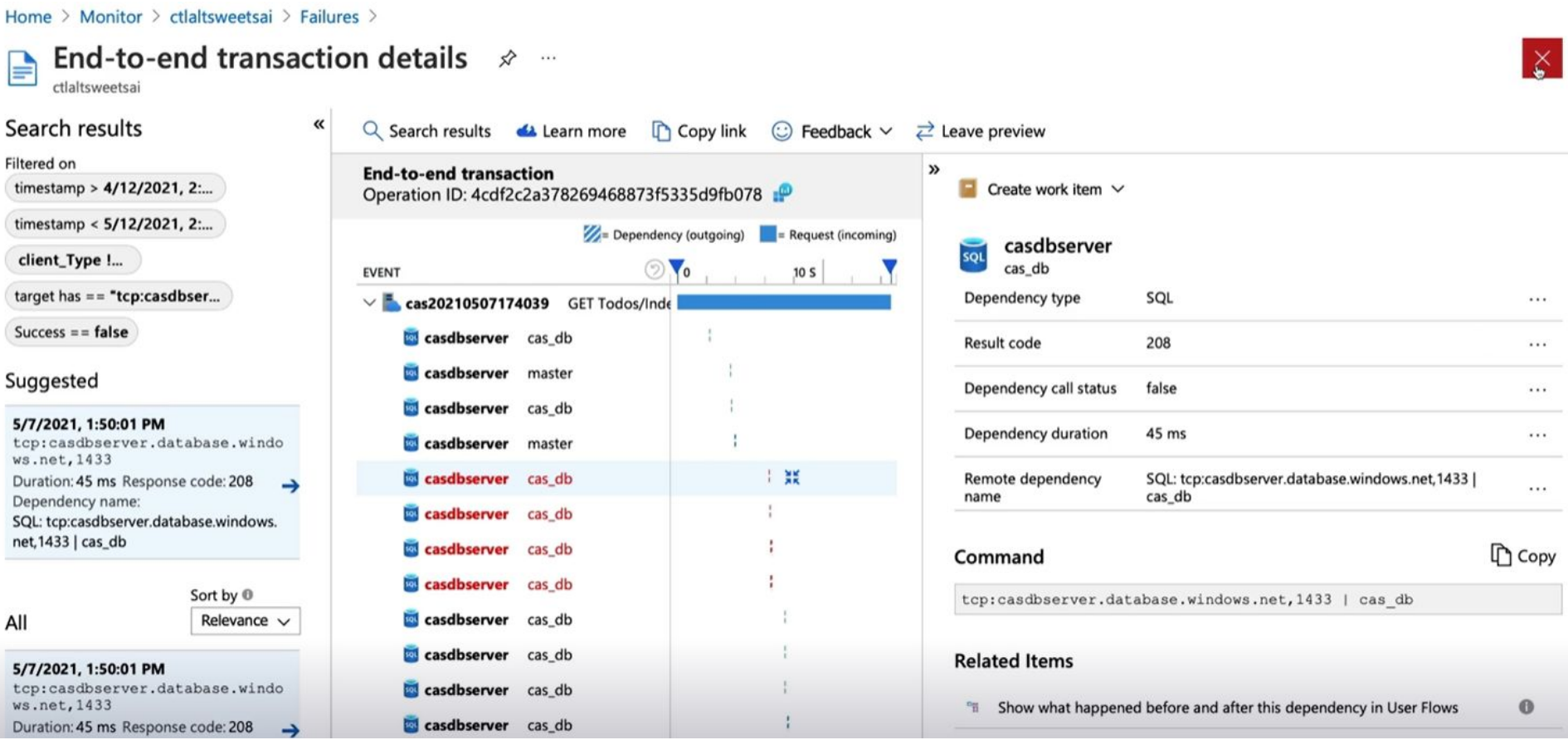
Deciding Which Dependencies to Set Alerts On



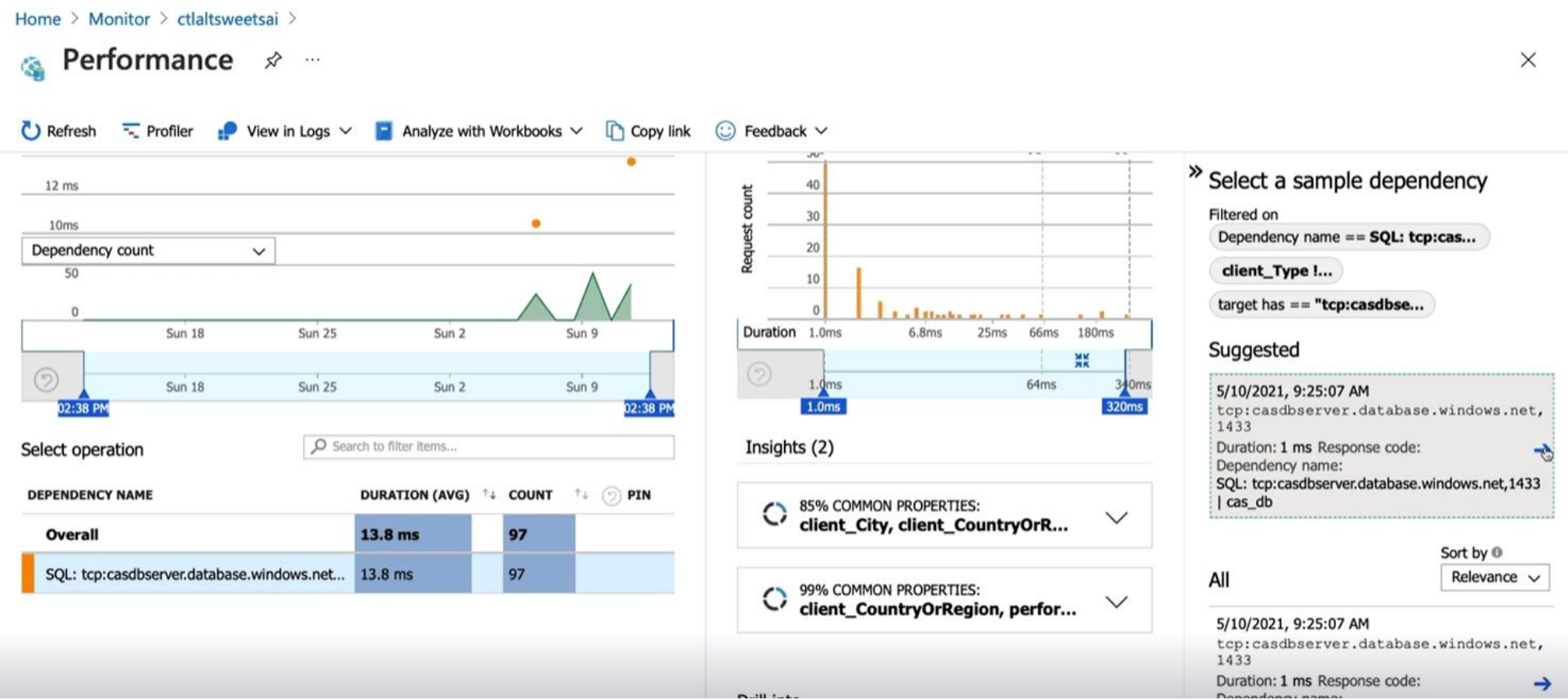
Deciding Which Dependencies to Set Alerts On



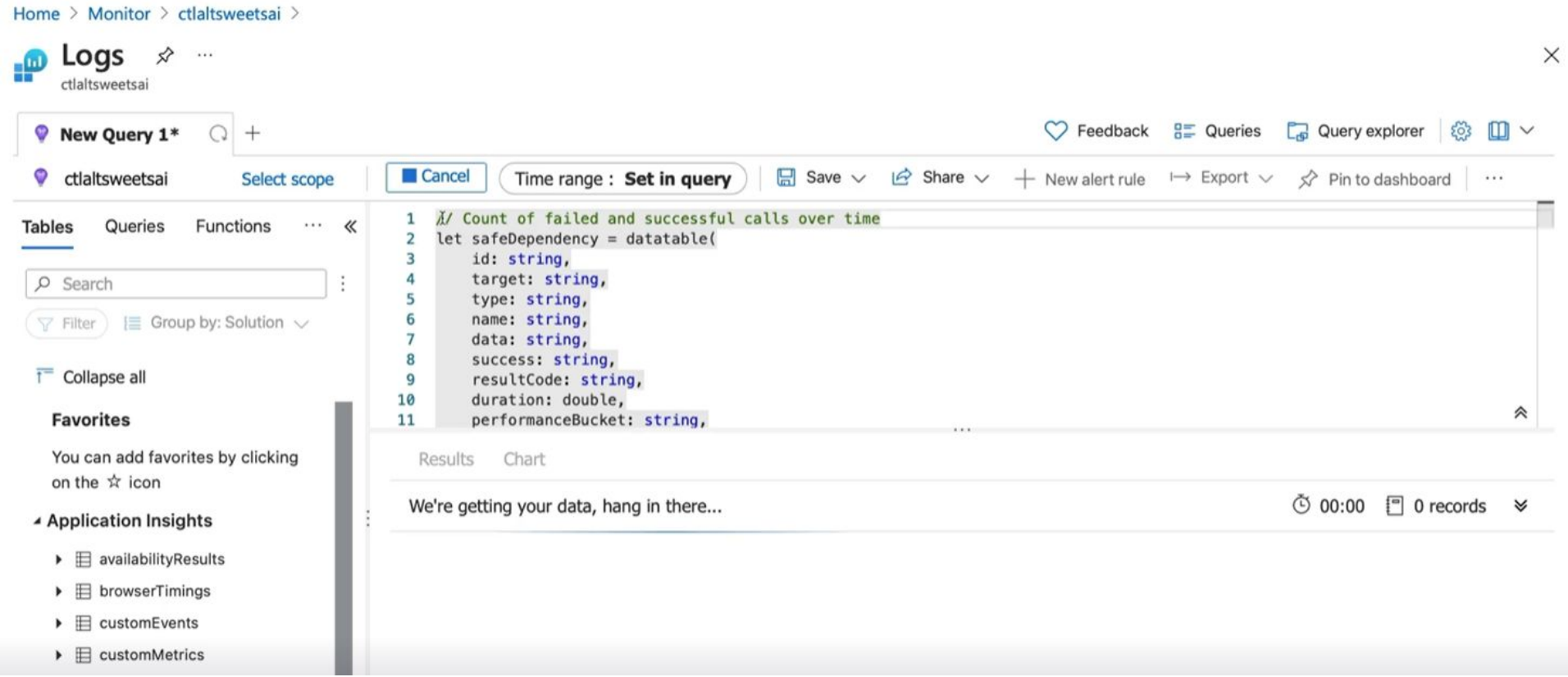
Deciding Which Dependencies to Set Alerts On



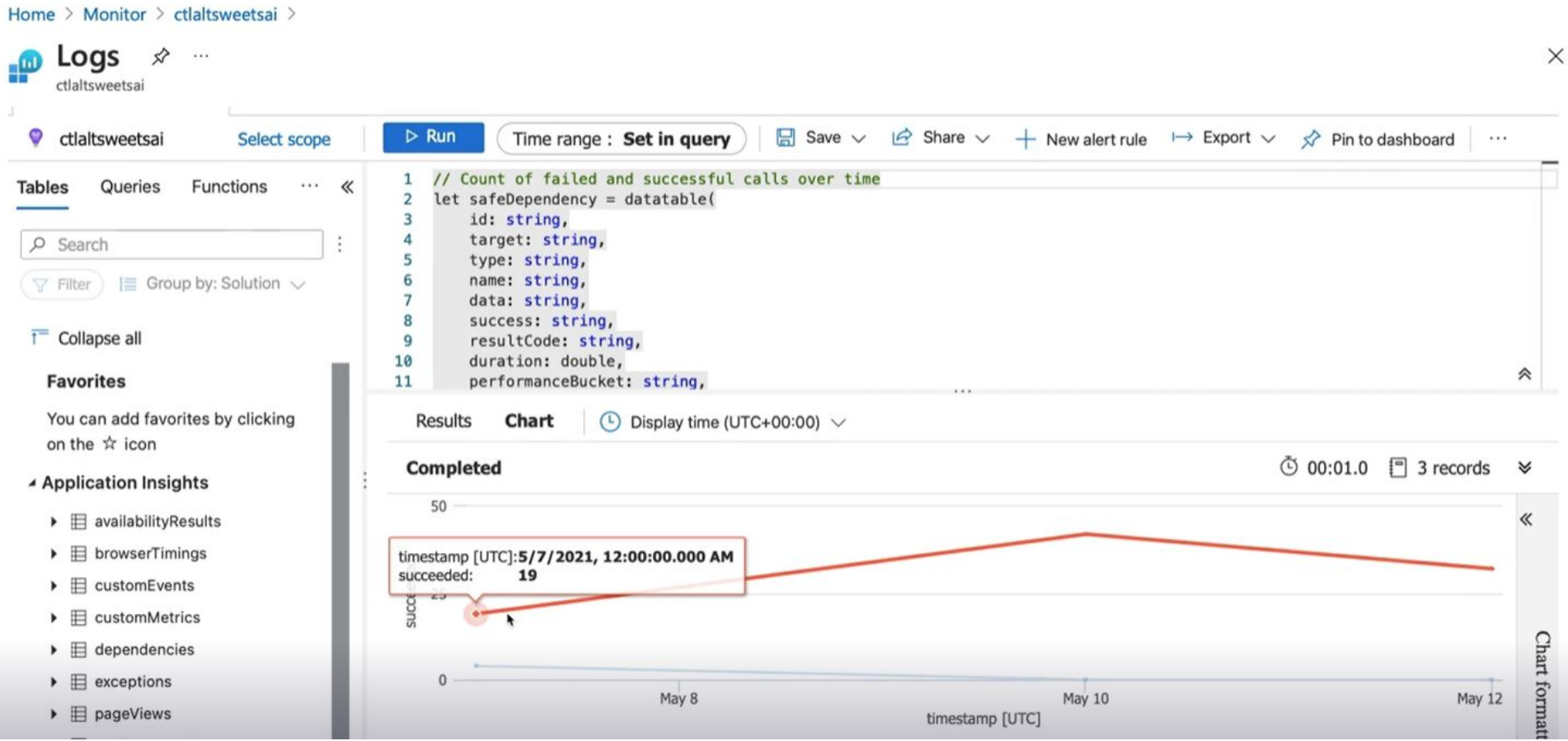
Deciding Which Dependencies to Set Alerts On



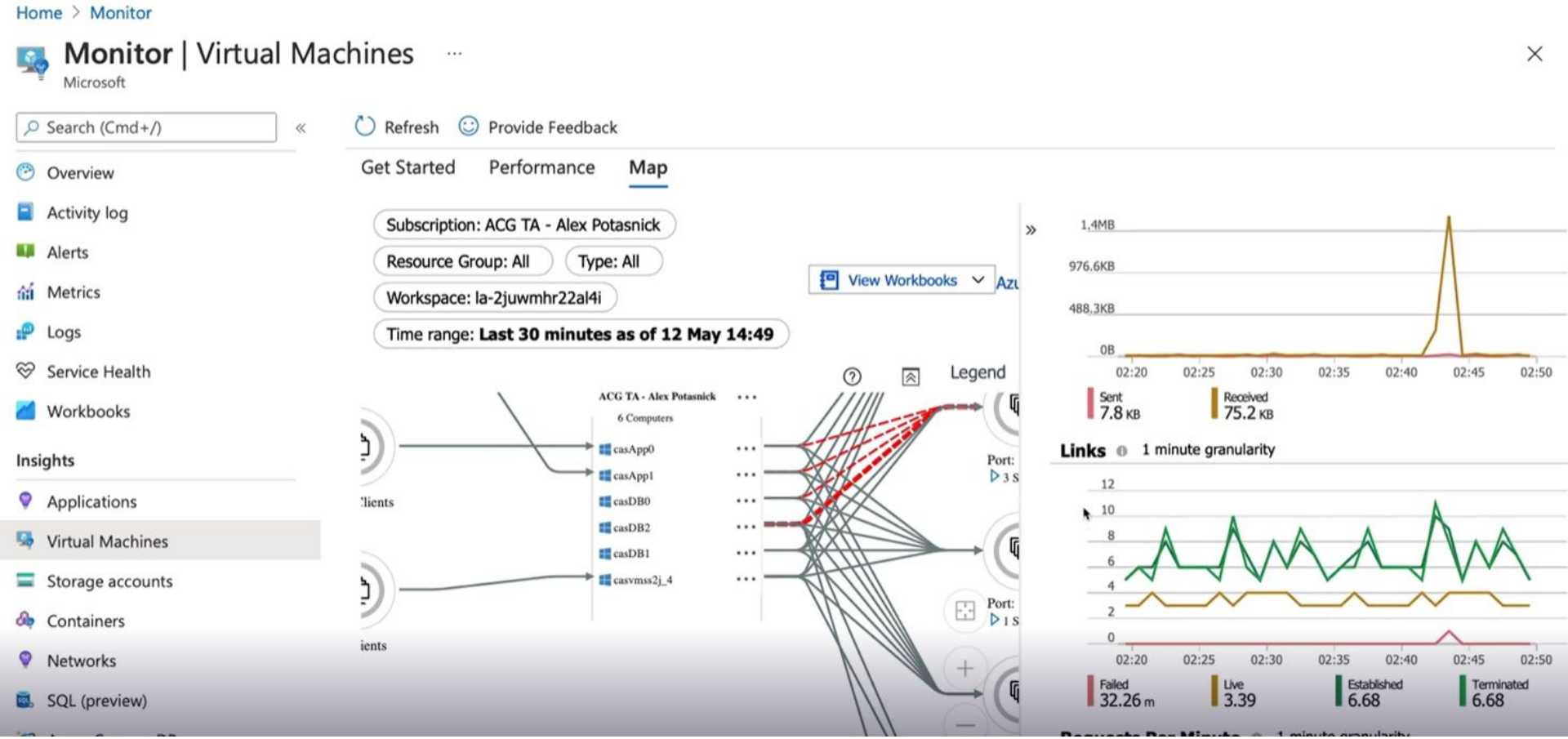
Deciding Which Dependencies to Set Alerts On



Deciding Which Dependencies to Set Alerts On



Deciding Which Dependencies to Set Alerts On



Deciding Which Dependencies to Set Alerts On

- Dependencies are components in an application that rely on each other for specific functions.
- Dependency Tracking is automatically configured with the .NET and .NET Core SDK for Application Insights.
- Manual dependency tracking can be configured using the TrackDependency API.

Deciding Which Dependencies to Set Alerts On

- The Application Map provides visualizations of application dependencies and correlated data.
- A virtual machine application dependency map can be found in Azure Monitor with system information and connection metrics.

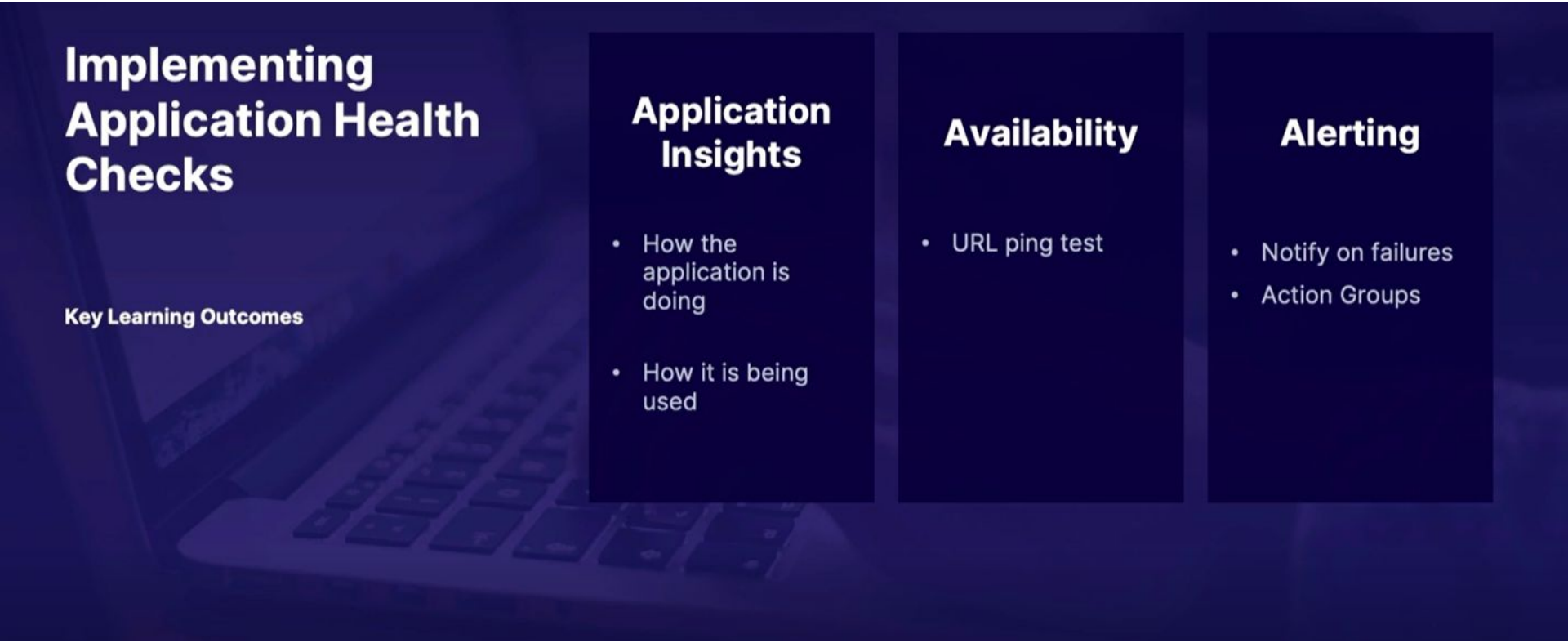
Summary


Big Picture: What Is Site Reliability Engineering (SRE)?
Key Learning Outcomes
Reliability
• Application is up when you expect it to be
• SLA
• SLO
• SLI
Key Concepts
• Feedback
• Measure everything
• Alerting
• Automation
• Small changes
• Embrace risk
SRE vs. DevOps
• SRE focuses on production
• DevOps focuses on deployments

Summary


Exploring Metric Charts and Dashboards
Key Learning Outcomes
Azure Monitor
Metrics
Logs
Monitor Metrics
Visualize
Analyze
Alert
Automate
Charts
Scope
Namespace
Metric
Aggregation
Time period

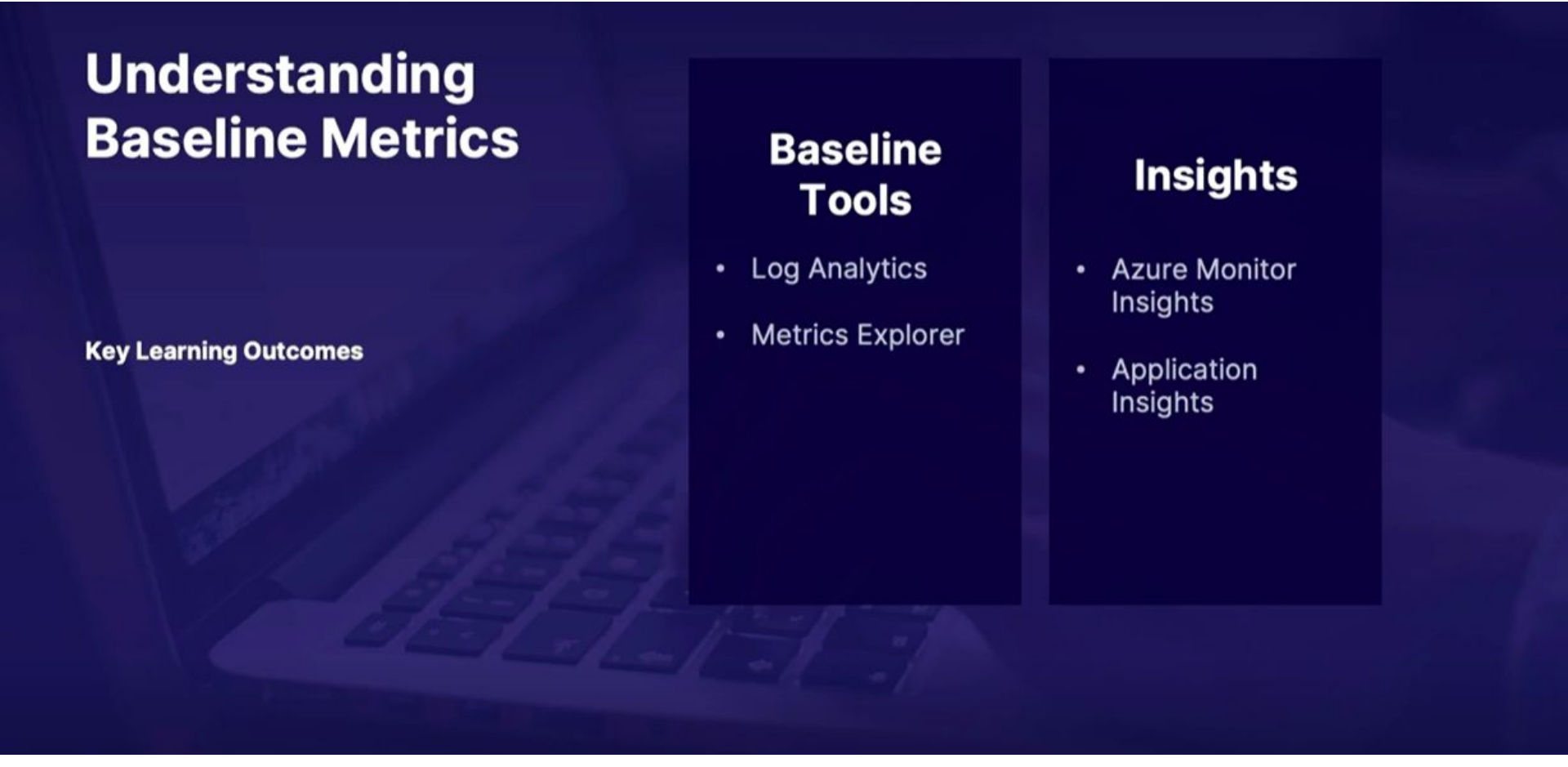
Summary


Implementing Application Health Checks
Key Learning Outcomes
Application Insights
How the application is doing
How it is being used
Availability
URL ping test
Alerting
Notify on failures
Action Groups

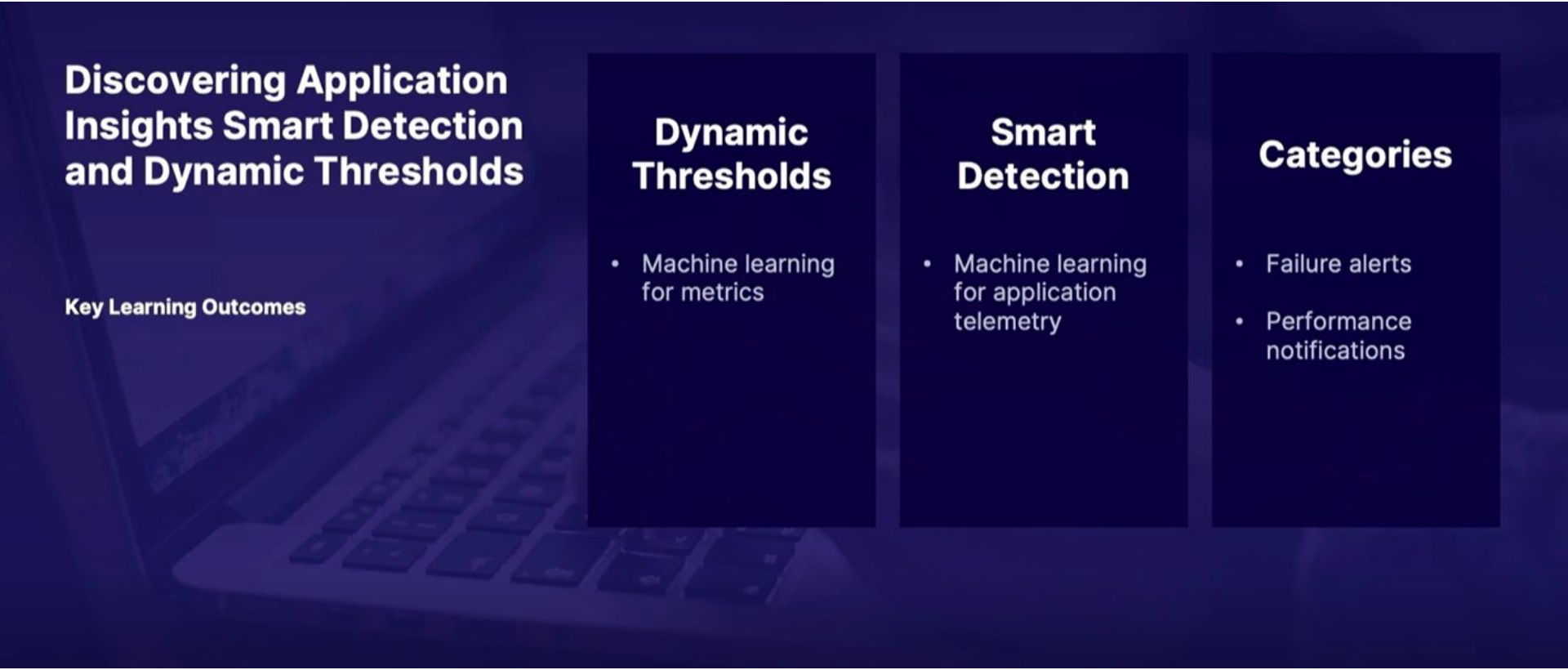
Summary


Exploring System Load and Failure Conditions
Key Learning Outcomes
Failure Mode Analysis
• Failure points and modes
• Risk and severity
• Response
Failure Reduction
• Understand the application and the environment
• Implement redundancy
Performance Testing
• Load Testing
• Stress Testing

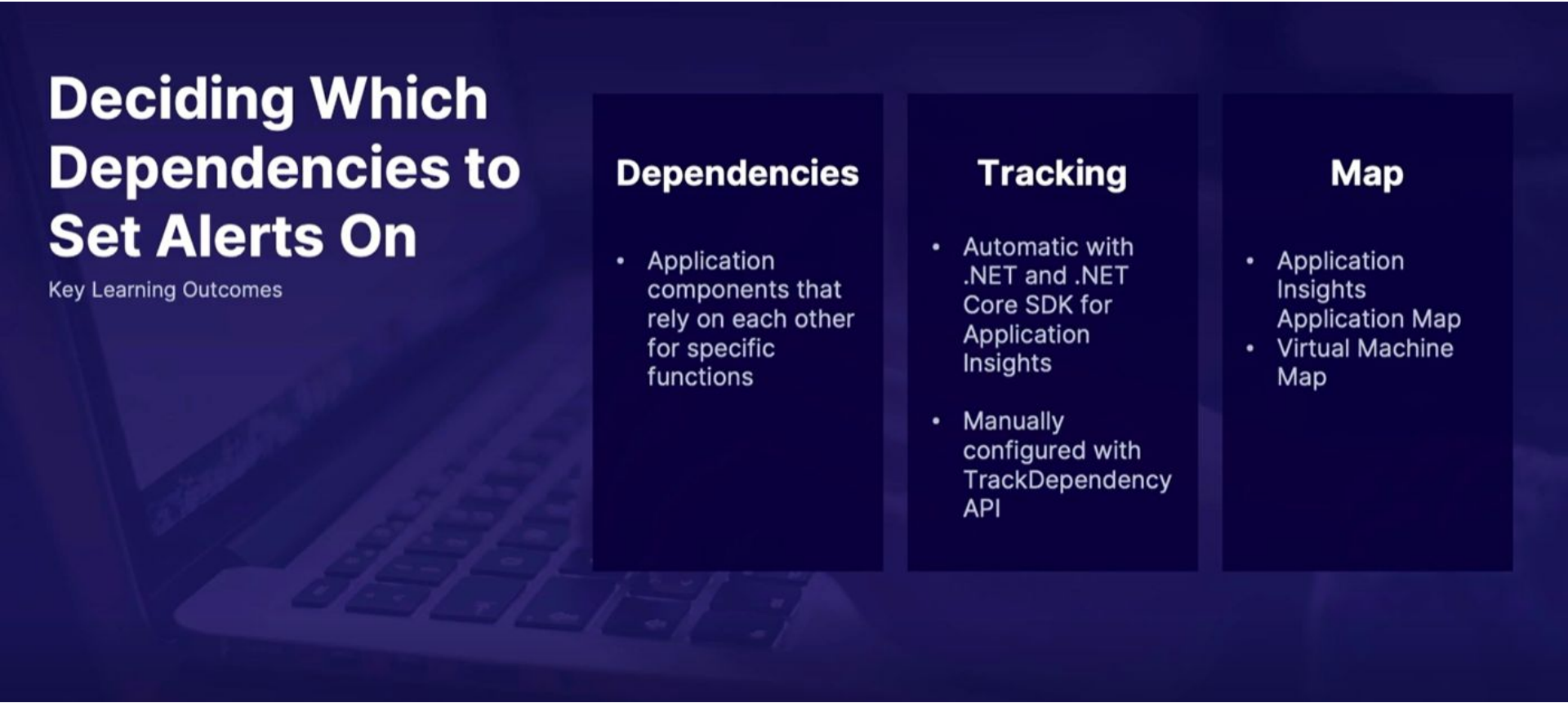
Summary


Understanding Baseline Metrics
Key Learning Outcomes
Baseline Tools
• Log Analytics
• Metrics Explorer
Insights
• Azure Monitor Insights
• Application Insights

Summary

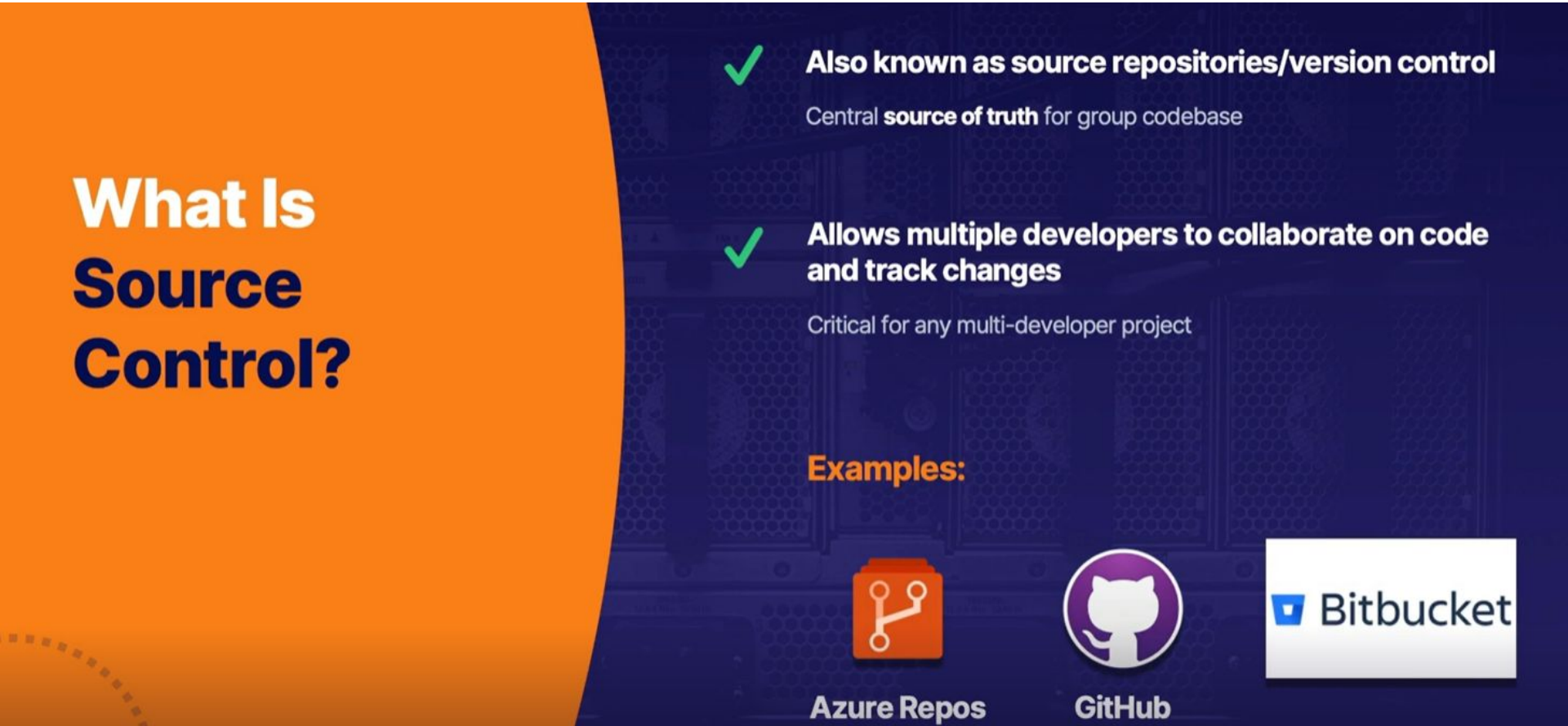

Discovering Application Insights Smart Detection and Dynamic Thresholds
Key Learning Outcomes
Dynamic Thresholds
• Machine learning for metrics
Smart Detection
• Machine learning for application telemetry
Categories
• Failure alerts
• Performance notifications

Summary

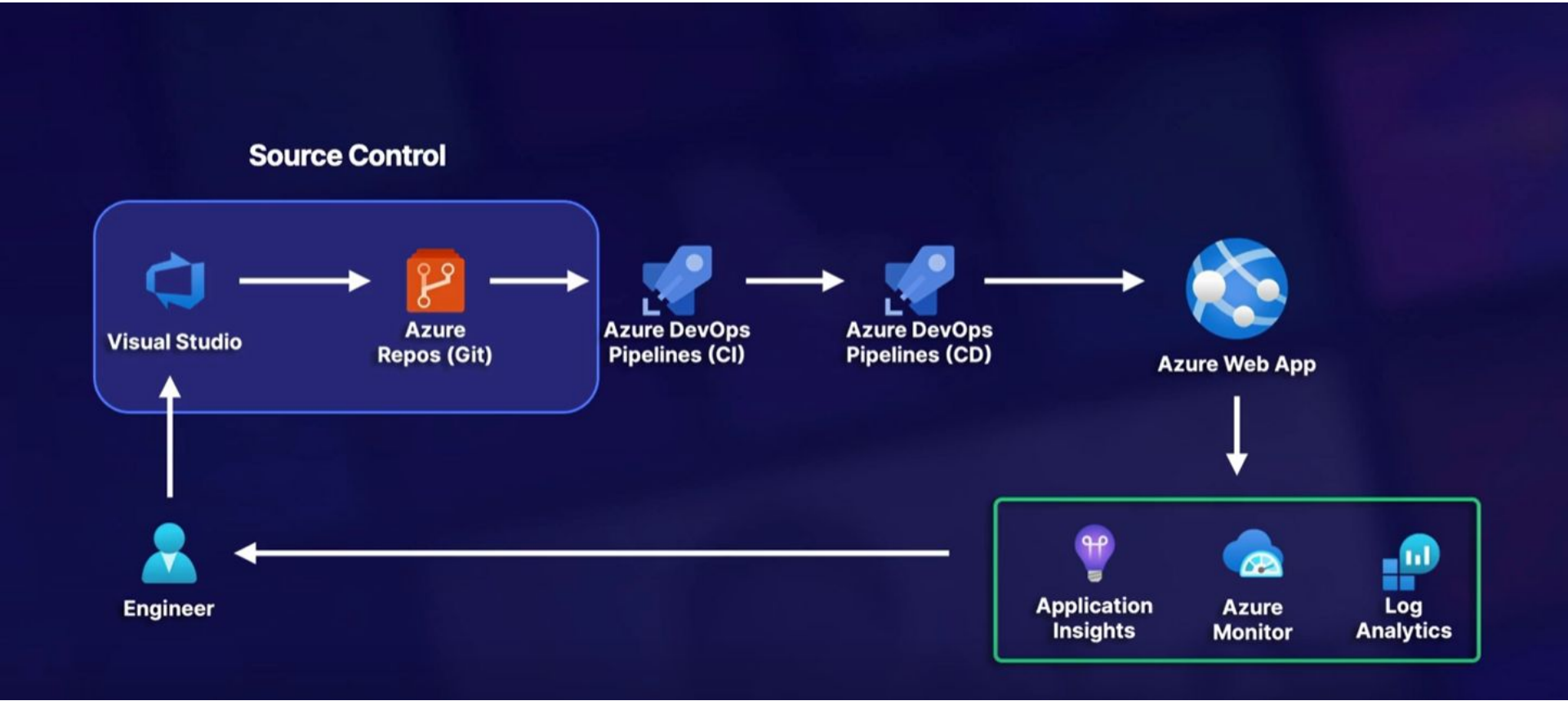

Deciding Which Dependencies to Set Alerts On
Key Learning Outcomes
Dependencies
• Application components that rely on each other for specific functions
Tracking
• Automatic with .NET and .NET Core SDK for Application Insights
• Manually configured with TrackDependency API
Map
• Application Insights Application Map
• Virtual Machine Map

Designing and Implementing a Source Control Strategy

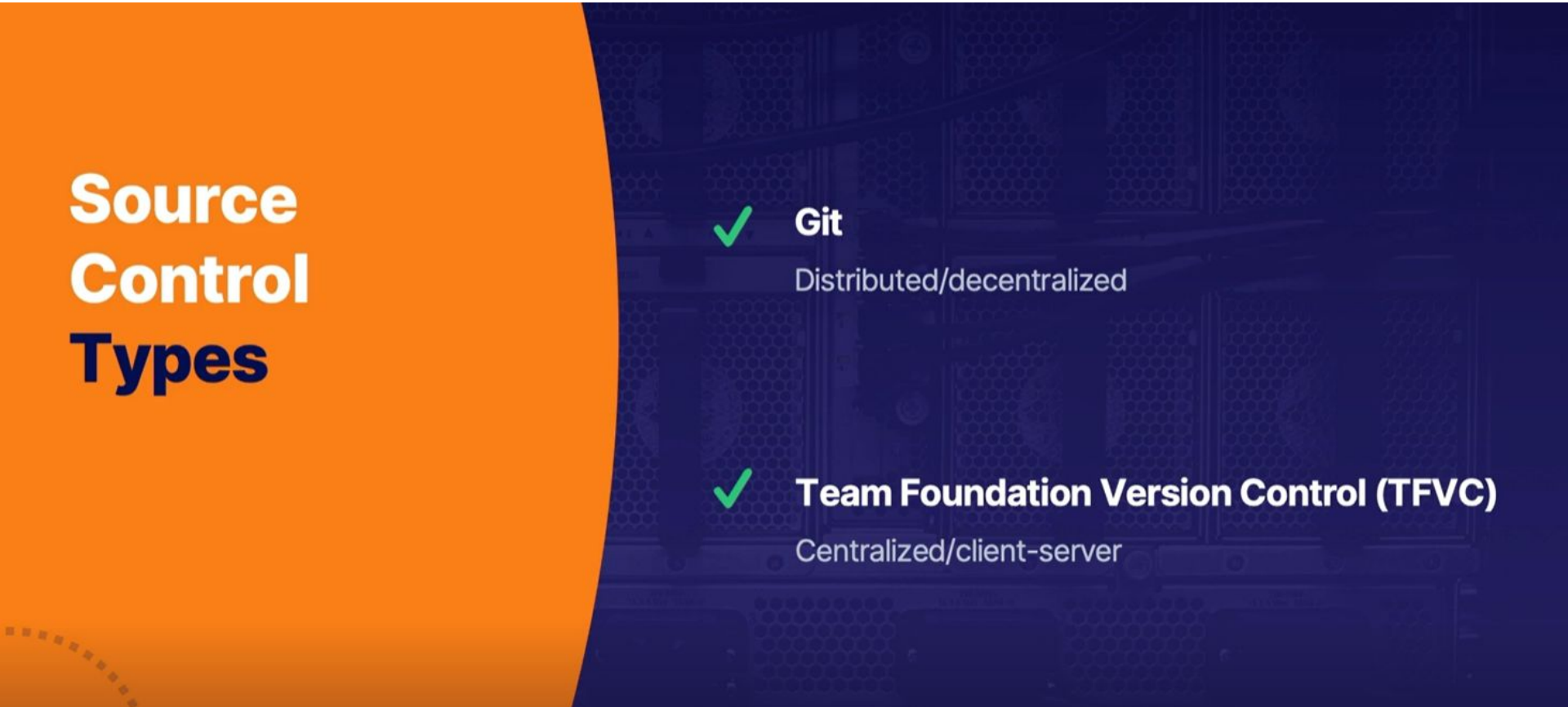
Introduction to Source Control



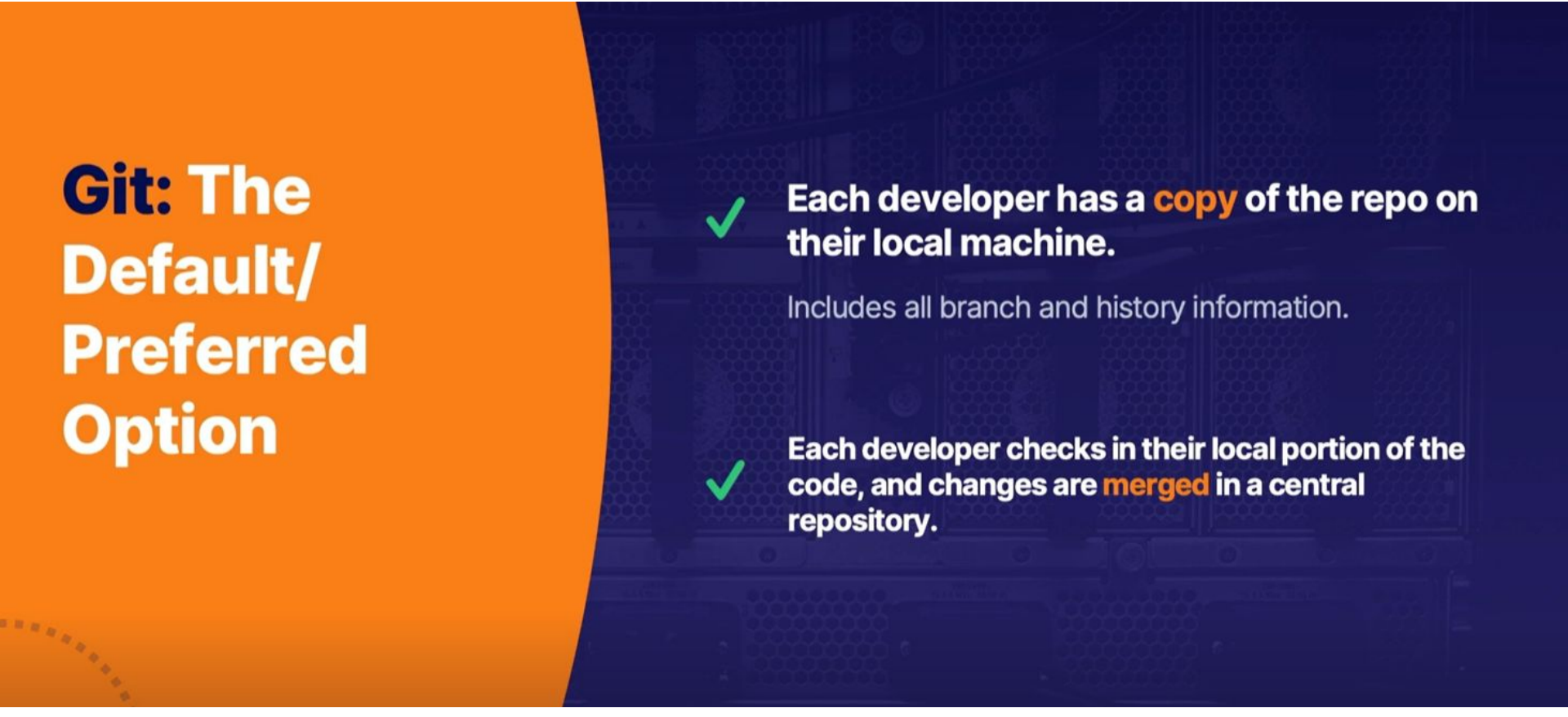
Introduction to Source Control



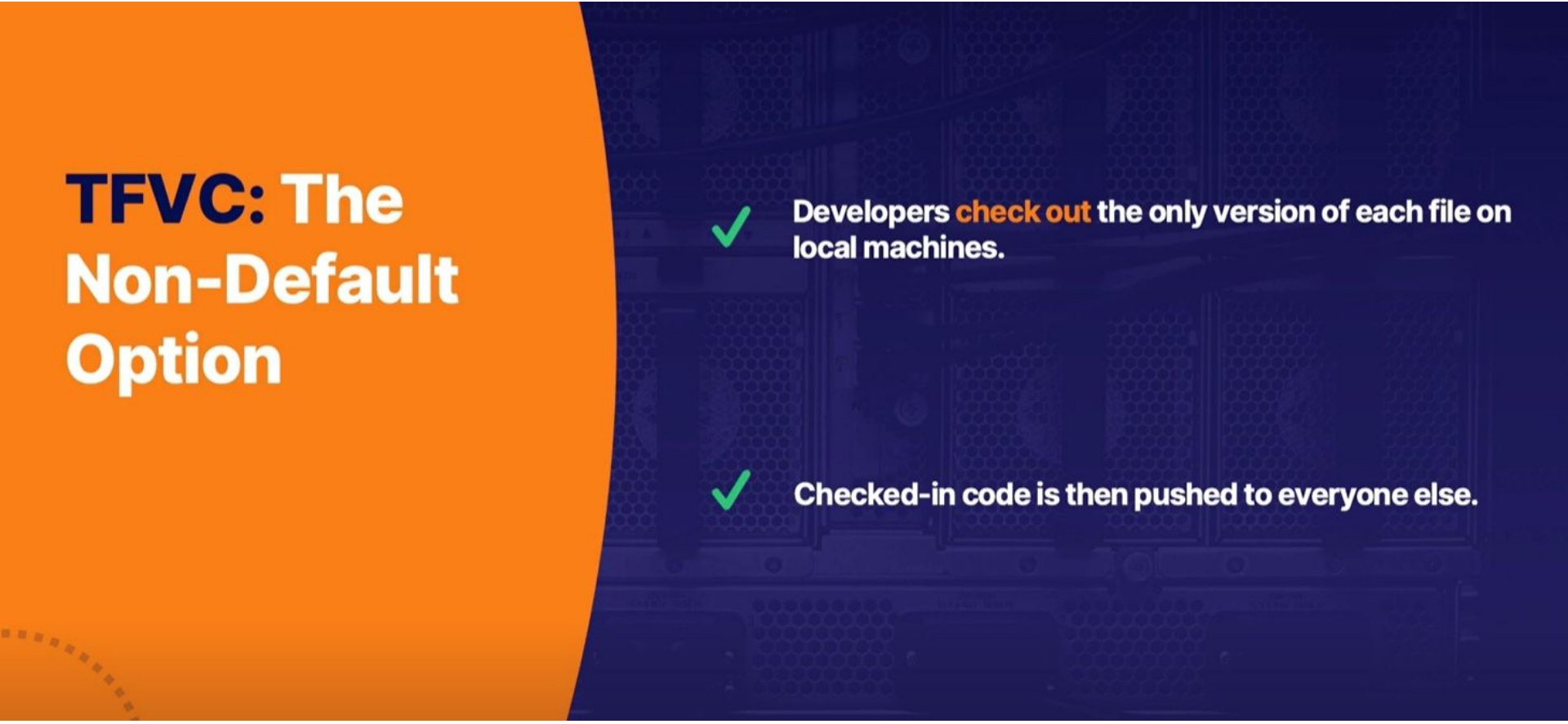
Introduction to Source Control



Introduction to Source Control



Introduction to Source Control



Introduction to Source Control

Which One Should I Use?

✓ **Per Microsoft: *Git* is preferred unless there is a specialized need for centralized version control in TFVC.**

As we move along, we will cover:

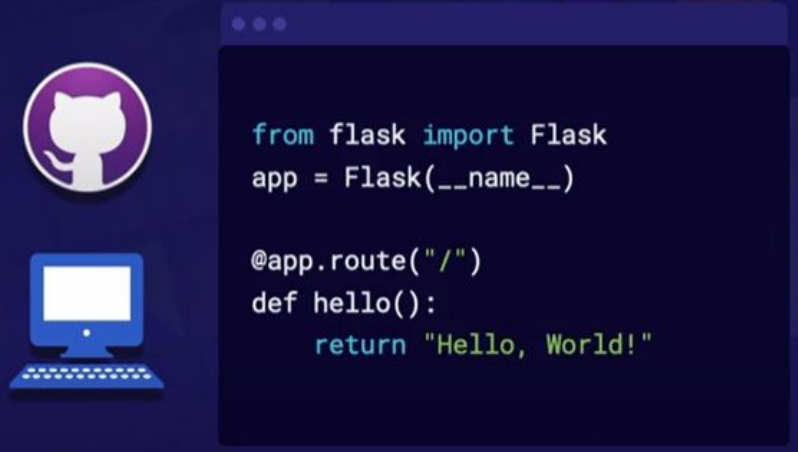
- Working with repos inside of Azure Repos
- General Git concepts
- Connecting repos to Pipelines (later)

Exploring Azure Repos

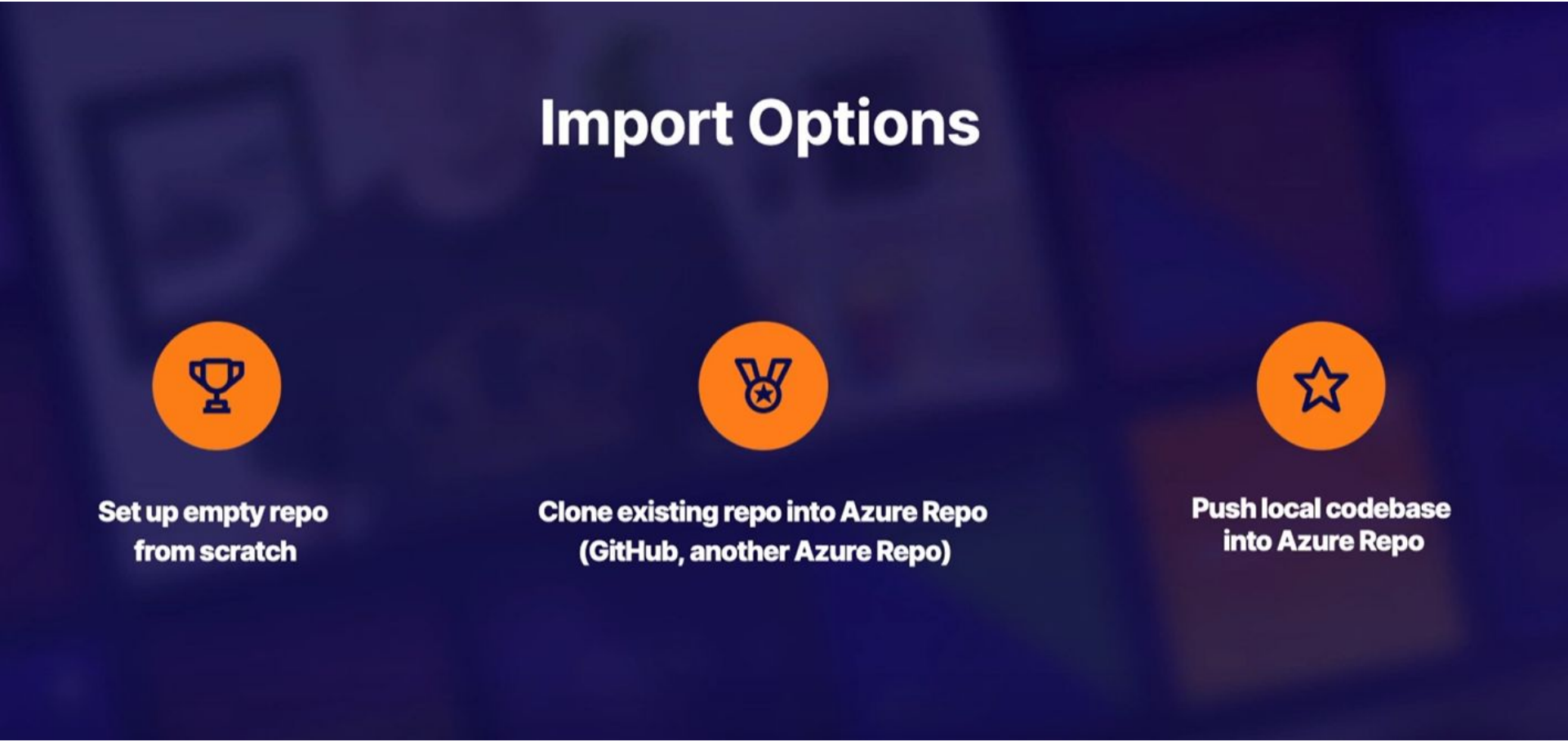
Setting Up Azure Repos
Multiple Options

All options involve getting code from *somewhere else* into Azure Repos

Azure Repos then becomes the **source of truth**



Exploring Azure Repos



Exploring Azure Repos

Repository Sharing with Submodules

Why do we care about this?

1. **Challenge: Need to incorporate resources from a different Git project in your current repo**
   - ✓ Examples: Third-party library used in your projects
   - ✓ Need to treat external resources as separate entities, yet seamlessly included within your project

2. **Solution: Submodules**
   - ✓ Not limited to Azure Repos, core Git feature — with an Azure twist!

Repository Sharing with Submodules

How it Works

1. **Add the external repo as a submodule to your repo.**

```
git submodule add
https://github.com/account/added-package
```

2. **When cloning a project with a submodule into your local directory, extra steps are required:**

✓ Initialize and update submodule.

```
git submodule init

git submodule update
```

Repository Sharing with Submodules

Submodules in Azure Pipelines

1. **Requirements to include in build pipelines:**

- ✓ **Unauthenticated** - i.e., publicly available
- ✓ **Authenticated within your main repository's account**
  - Same GitHub organization, Azure organization, etc.
  - Same account to access the primary repo must also be the same one to access the submodule repository
  - Submodules must be registered via HTTP — not SSH

Scalar

What Is Scalar?

Git client tool used to optimize new and existing repositories

Git was not designed to grow to certain sizes, so you might notice Git becoming sluggish when projects get too large, which makes it difficult to scale.

Scalar


SCALAR
Data Transfer
Command Runtime
Relevant Files
Organization

Scalar

OPTION 1 **NEW REPOSITORY**

- Create repository
- Use partial clone feature
- Use sparse checkout feature
- Set up Git maintenance
- Configure optimization settings

OPTION 2 **EXISTING REPOSITORY**

Apply features and settings to optimize existing repository

Scalar

- Scalar is a Git tool that helps improve performance on large repositories.
- `scalar clone` is used for new repositories.
- `scalar register` is used for existing repositories.

Connecting to GitHub Using Azure Active Directory

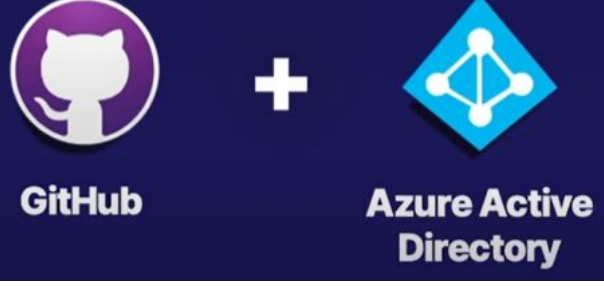
GitHub and Azure Active Directory (AAD) Integration

1 Why This Matters

- ✓ By default, GitHub and AAD identities are separately maintained.
- ✓ However, we can integrate GitHub identities with AAD using **Single Sign On (SSO)**.

2 Advantage of GitHub/AAD Integration

- ✓ Manage GitHub accounts from a central location: Azure Active Directory.



Scalar

Requirements for GitHub/AAD SSO

1. **GitHub Enterprise Cloud Organization**
   - ✓ GitHub Team plan unable to use SSO
2. **Permissions**
   - ✓ GitHub: Administrator
   - ✓ Azure: Create SSO – Global Admin, Cloud Application Administrator, Application Administrator

How it works: Azure AD SSO Configuration

1. **Add GitHub in Enterprise Applications.**

2. **Configure SAML SSO configuration with GitHub Enterprise account.**

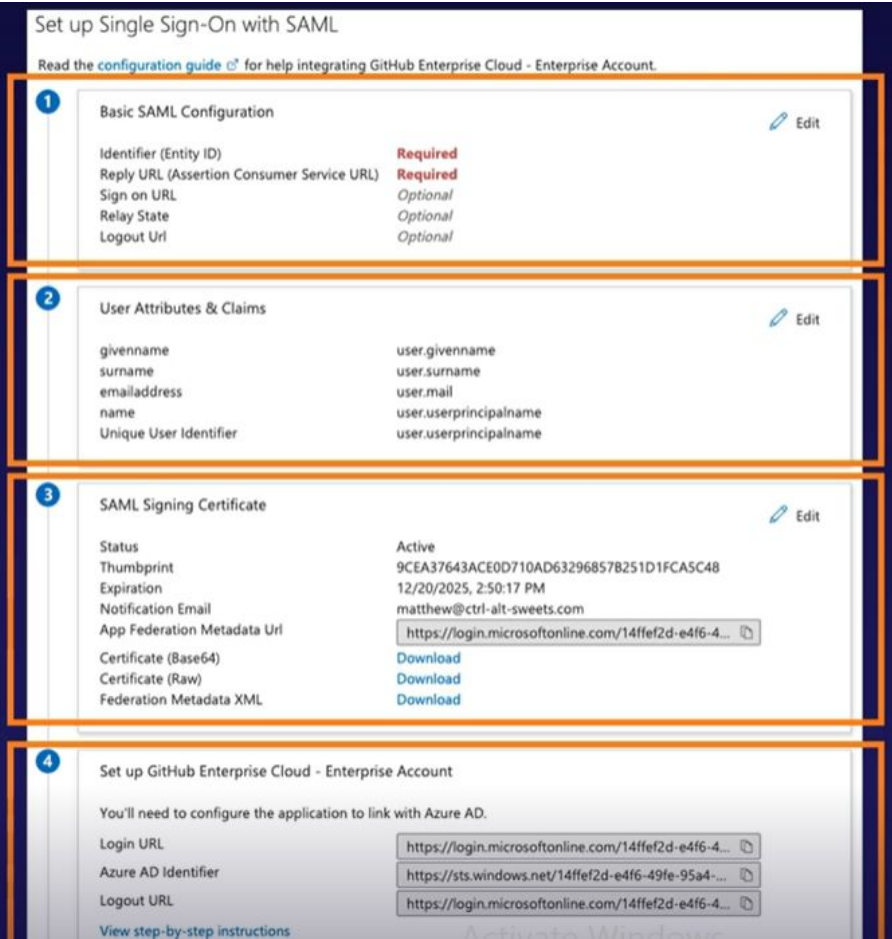
   Link to GitHub Enterprise Organization.

   Set User Attributes.

   Download Base64 Signing Certificate (for GitHub side).

   Generate GitHub links to AAD.

3. **Add AAD Users to GitHub SSO.**



How it works: Github Enterprise Configuration

1. **Enable SAML Authentication.**

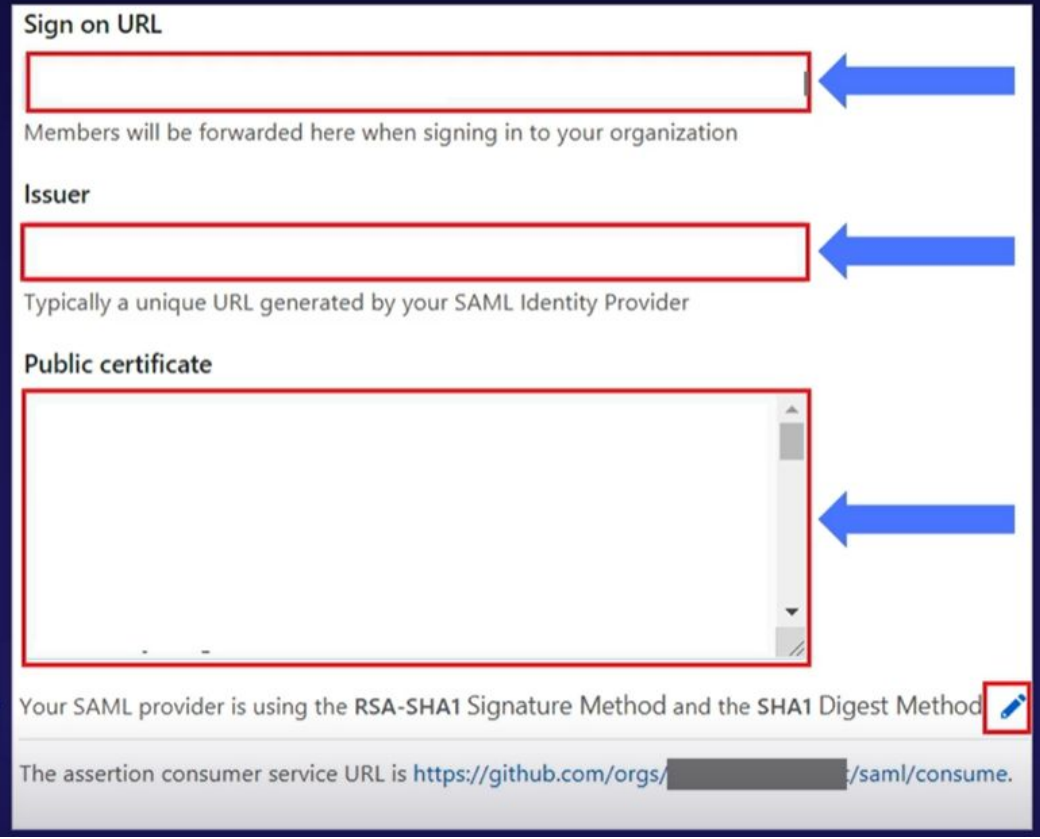
2. **Configure link to AAD tenant.**

   Login URL → Sign on URL

   AAD Identifier → Issuer

   Open and copy/paste Signing Certificate from AAD

   Set Signature/Digest Method to RSA-SHA256/SHA256



Incorporating Changelogs

What Are Git Changelogs?

1. **Record of changes (commits) in a project's lifetime**

2. **Why do we care about changelogs?**

- ✓ Keep running list of changes/updates
- ✓ Useful for teams working on a single project

Incorporating Changelogs

Manually Creating/Viewing Changelogs

1. **Git log command**

```
git log
```

2. **Options to clean log output**

- ✓ One-line summaries
- ✓ Remove commit ID, custom elements

```
git log --oneline

git log --pretty="- %s"
```

Incorporating Changelogs

Automation Options

Third-Party Applications

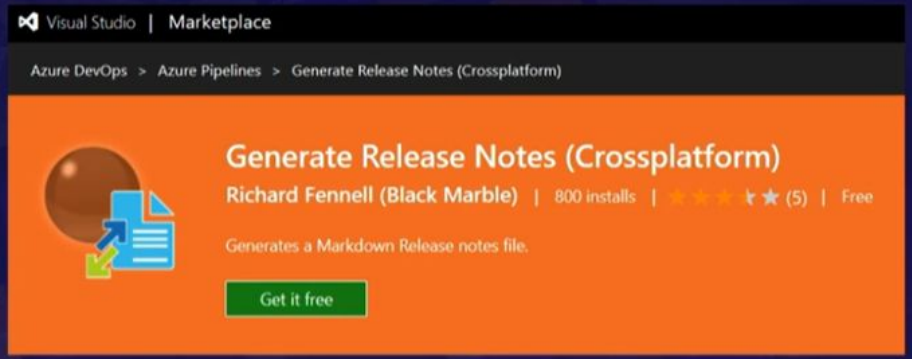
GitHub Changelog Generator
Auto Changelog

IDE Plugins

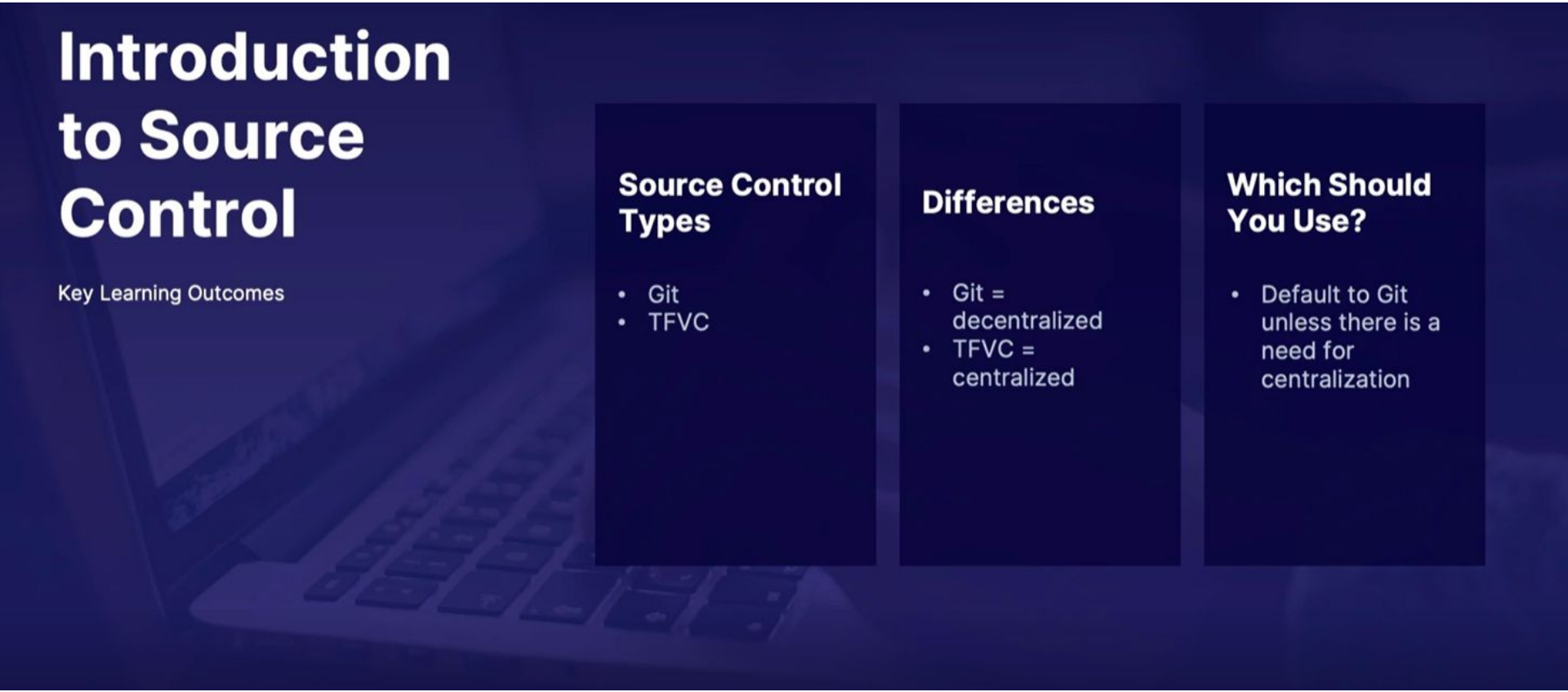
Example: Visual Studio changelog plugin

Pipeline Plugins

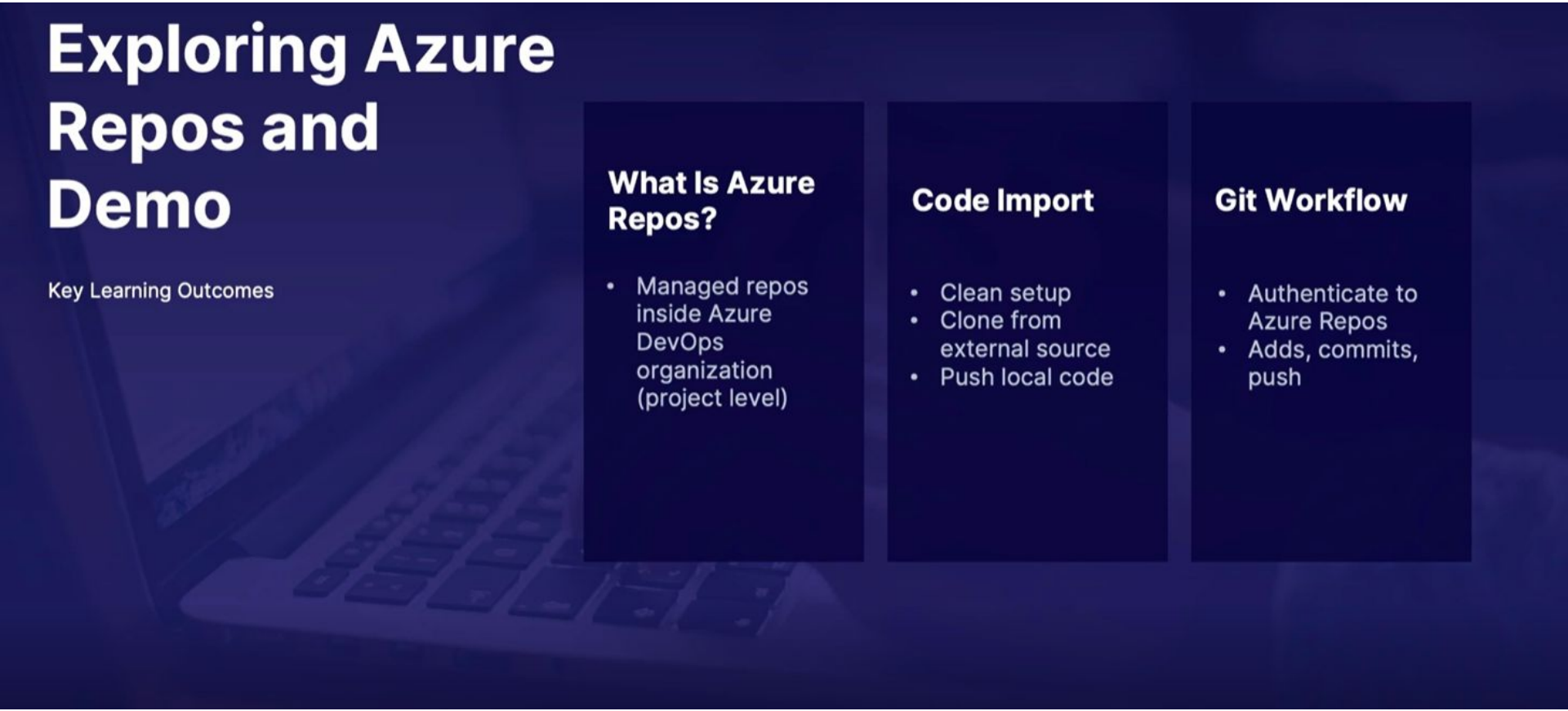
Example: Jenkins has a changelog plugin



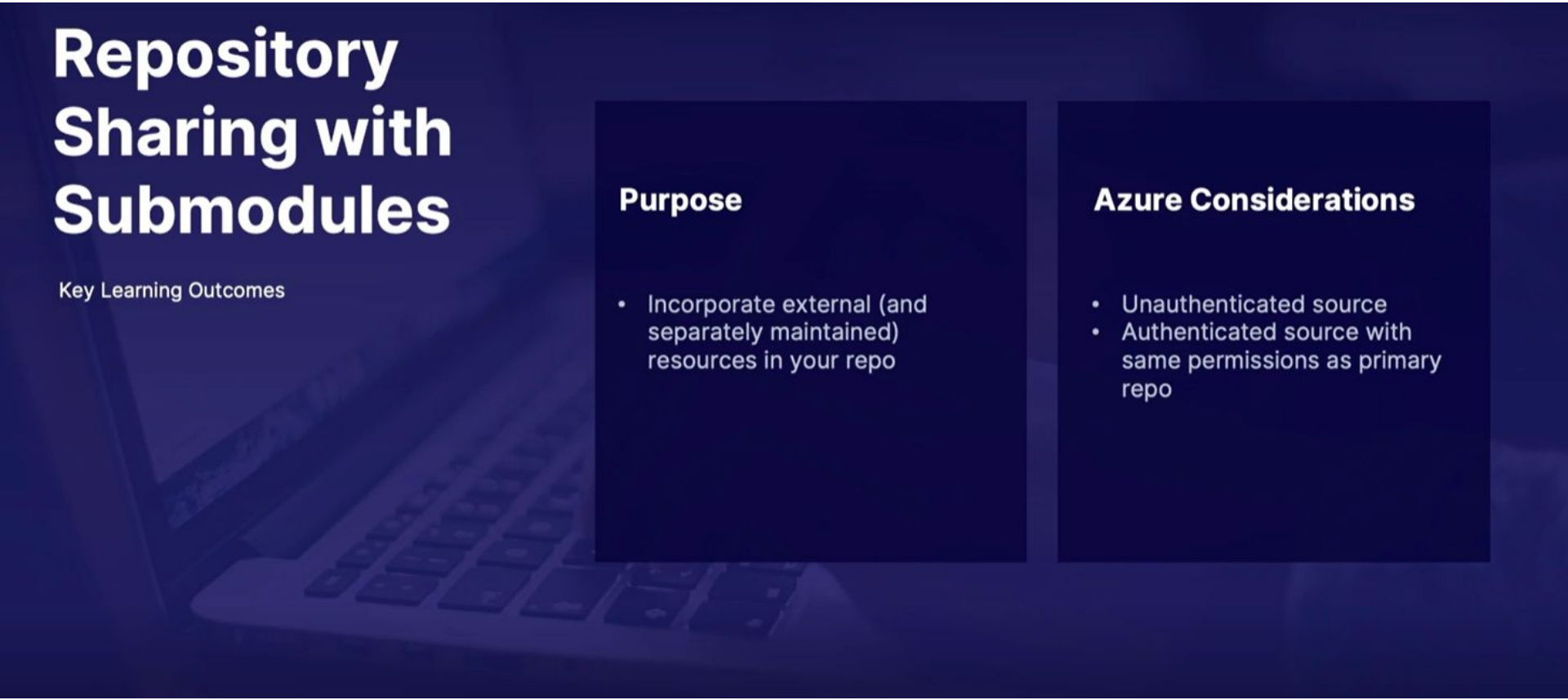
Summary


Introduction to Source Control
Key Learning Outcomes
Source Control Types
Git
TFVC
Differences
Git = decentralized
TFVC = centralized
Which Should You Use?
Default to Git unless there is a need for centralization

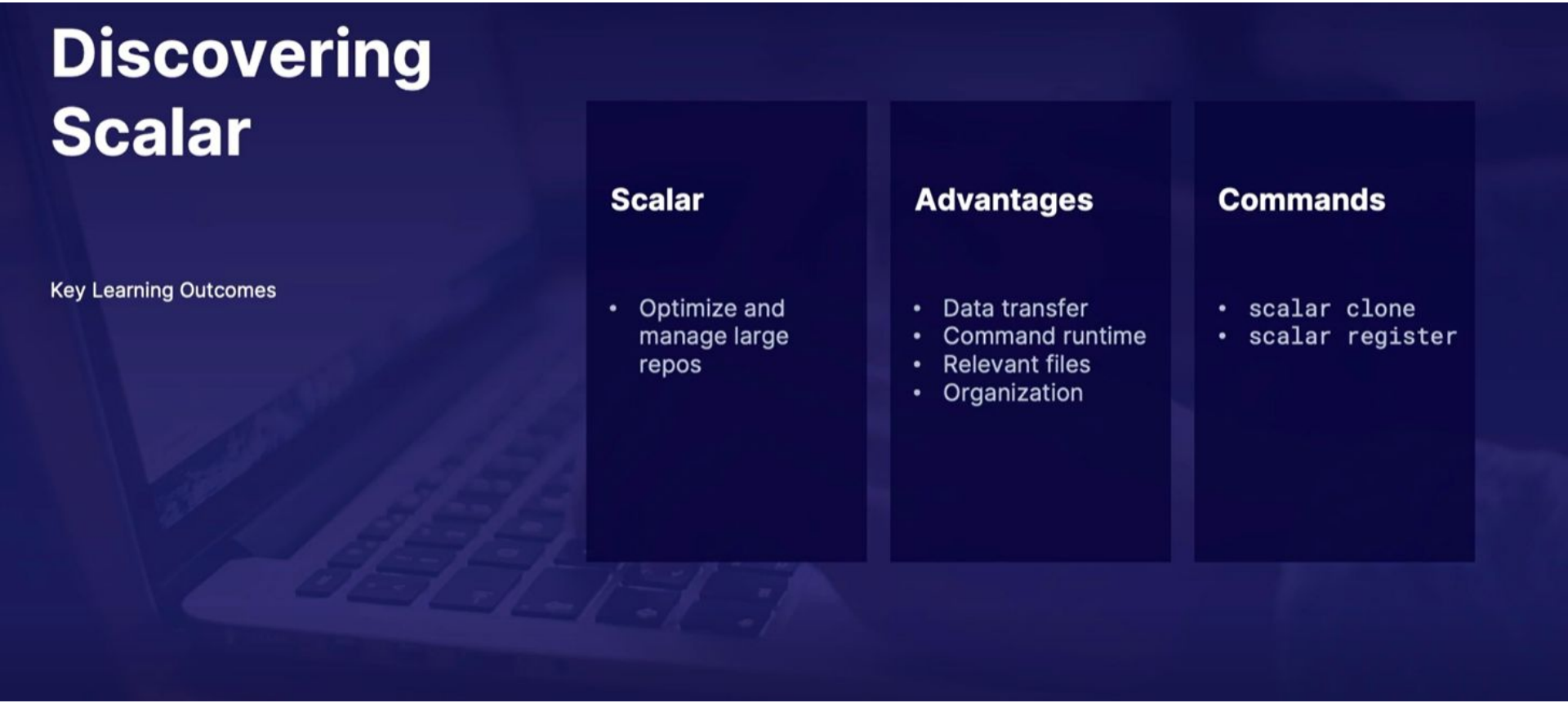
Summary


Exploring Azure Repos and Demo
Key Learning Outcomes
What Is Azure Repos?
Managed repos inside Azure DevOps organization (project level)
Code Import
Clean setup
Clone from external source
Push local code
Git Workflow
Authenticate to Azure Repos
Adds, commits, push

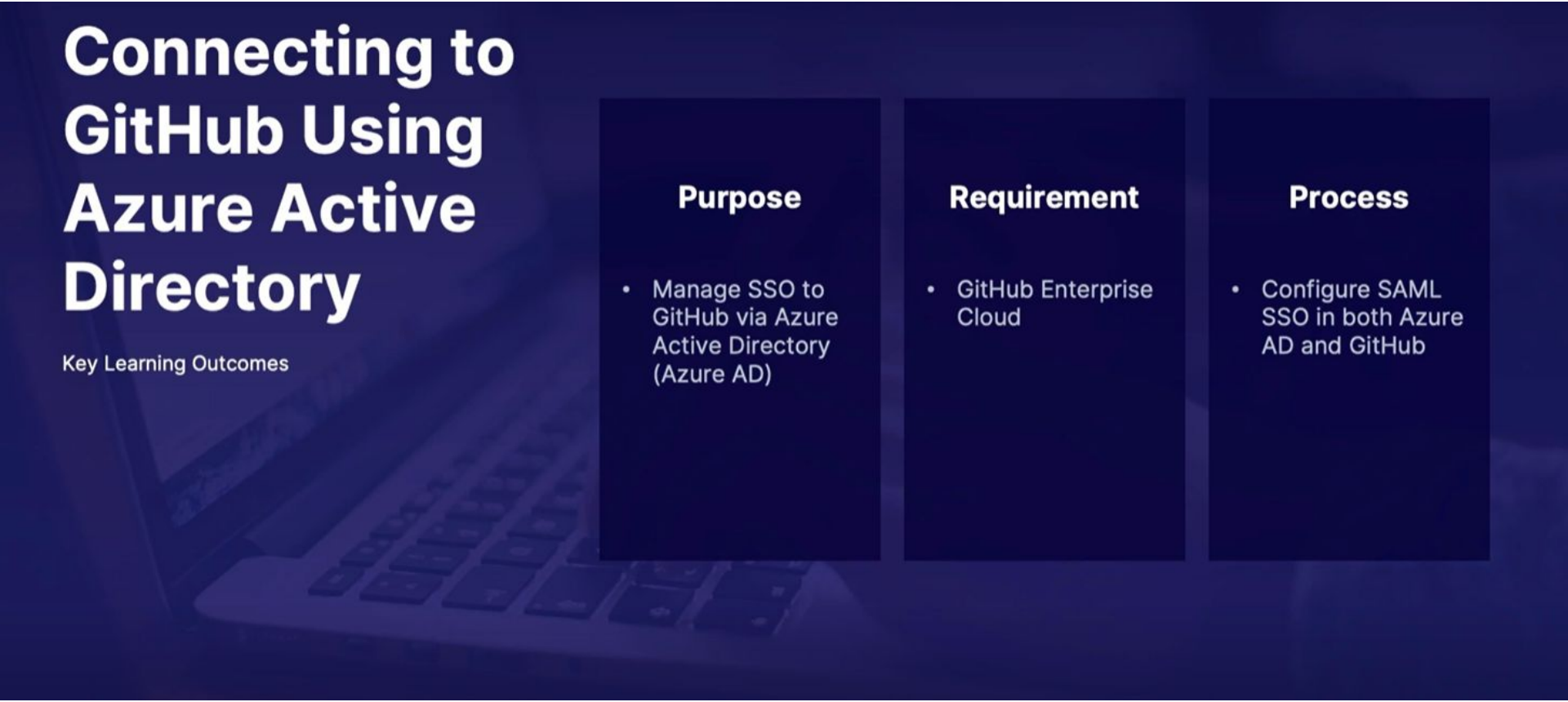
Summary


Repository Sharing with Submodules
Key Learning Outcomes
Purpose
• Incorporate external (and separately maintained) resources in your repo
Azure Considerations
• Unauthenticated source
• Authenticated source with same permissions as primary repo

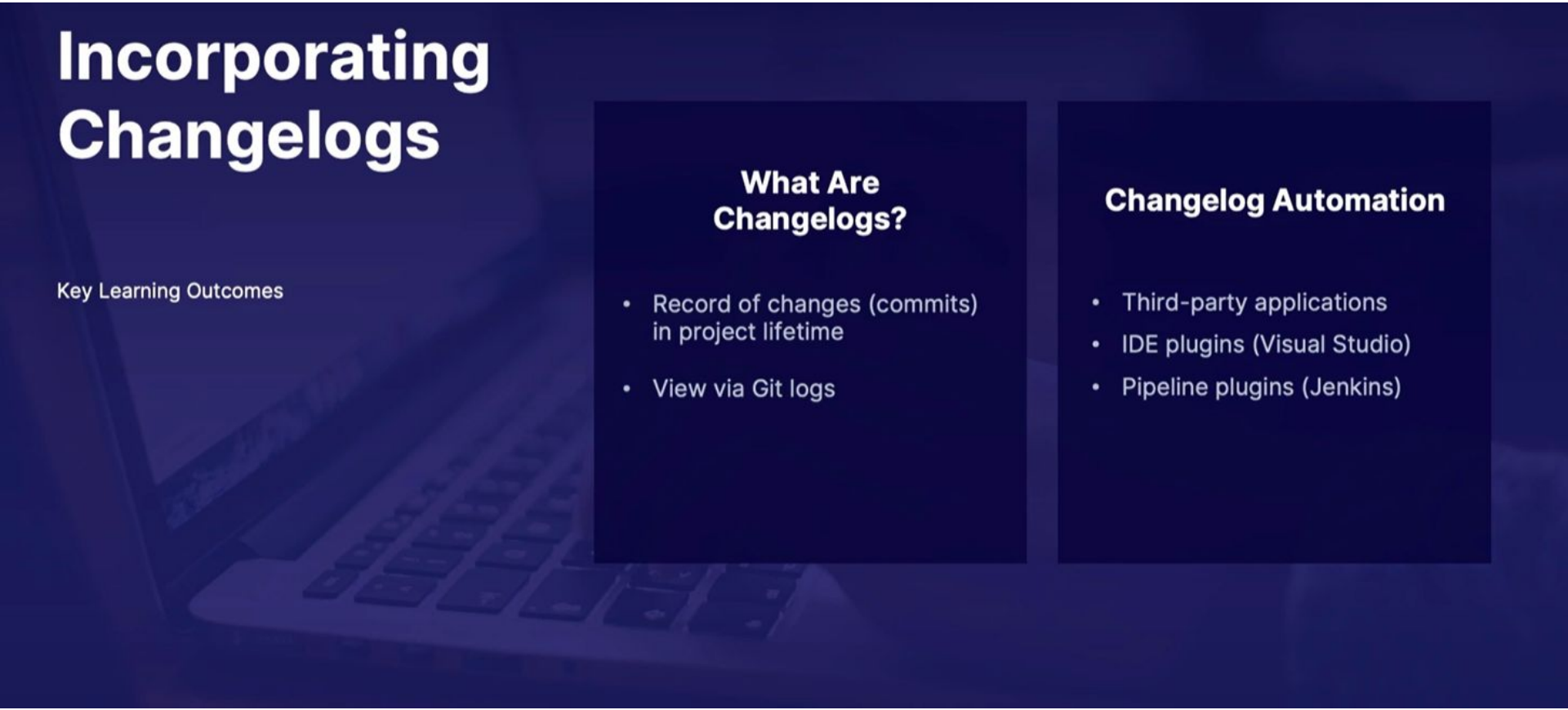
Summary


Discovering Scalar
Key Learning Outcomes
Scalar
• Optimize and manage large repos
Advantages
• Data transfer
• Command runtime
• Relevant files
• Organization
Commands
• scalar clone
• scalar register

Summary

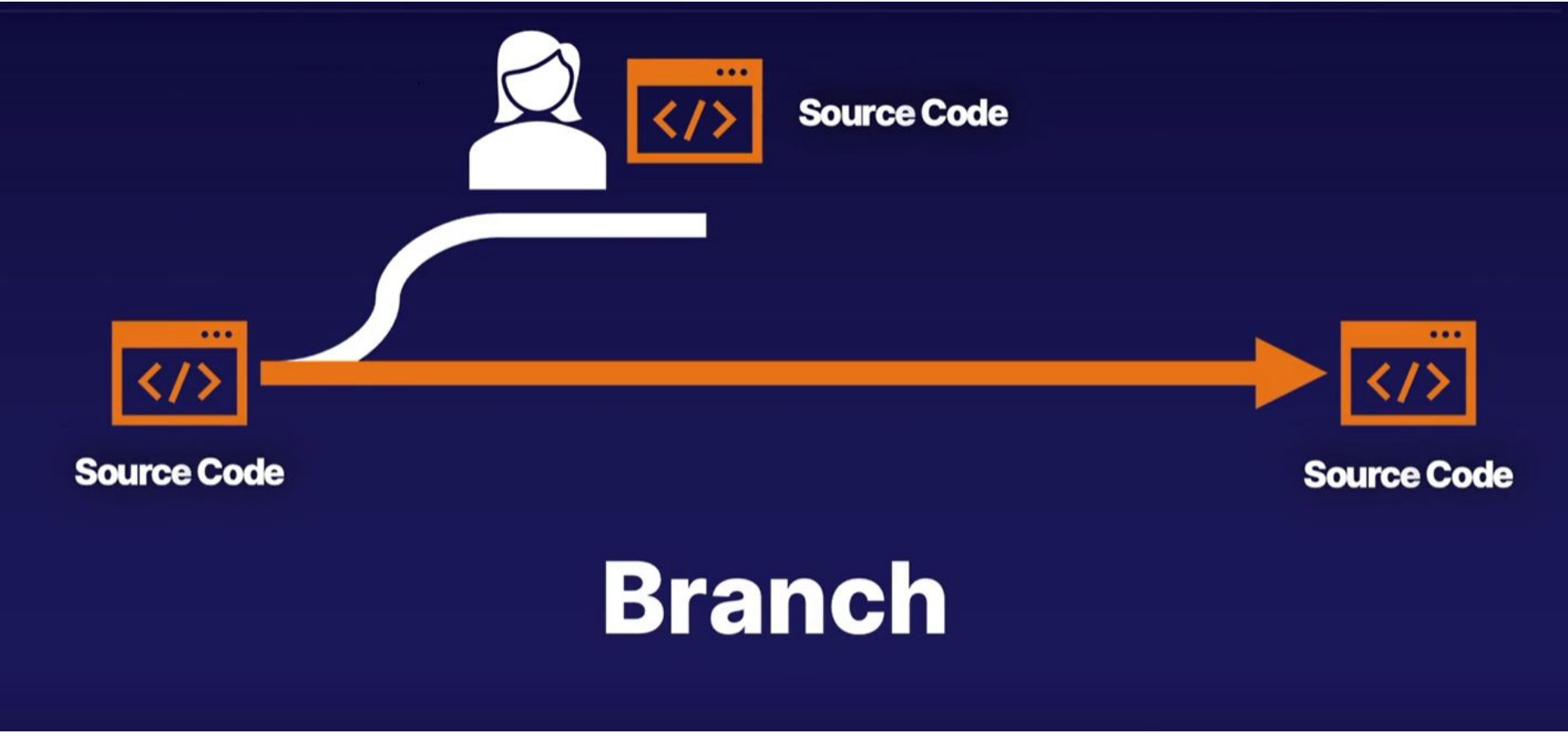

Connecting to GitHub Using Azure Active Directory
Key Learning Outcomes
Purpose
• Manage SSO to GitHub via Azure Active Directory (Azure AD)
Requirement
• GitHub Enterprise Cloud
Process
• Configure SAML SSO in both Azure AD and GitHub

Summary


Incorporating Changelogs
Key Learning Outcomes
What Are Changelogs?
• Record of changes (commits) in project lifetime
• View via Git logs
Changelog Automation
• Third-party applications
• IDE plugins (Visual Studio)
• Pipeline plugins (Jenkins)

Planning and Implementing Branching Strategies for the Source Code

Configuring Branches

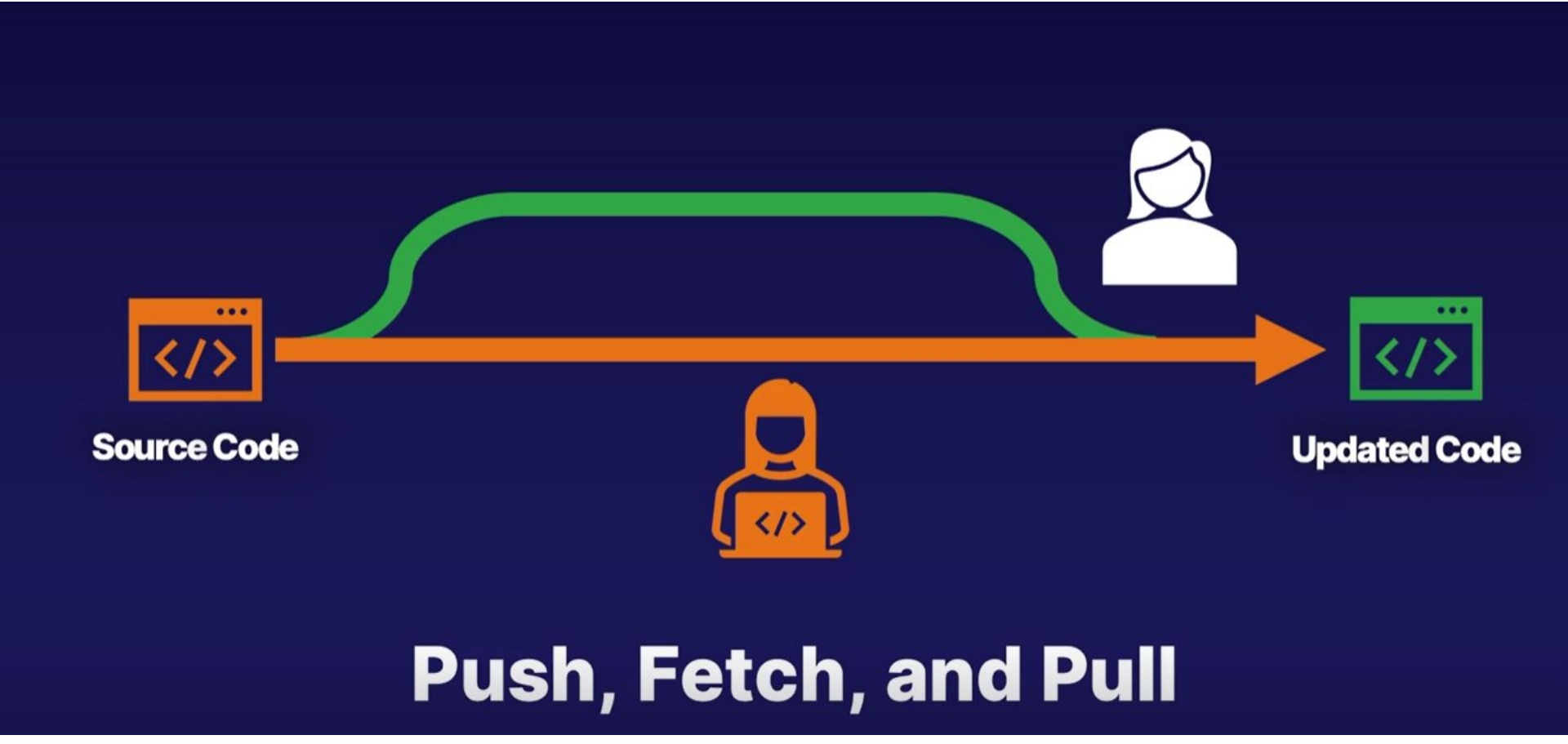


Configuring Branches

Configuring Branches

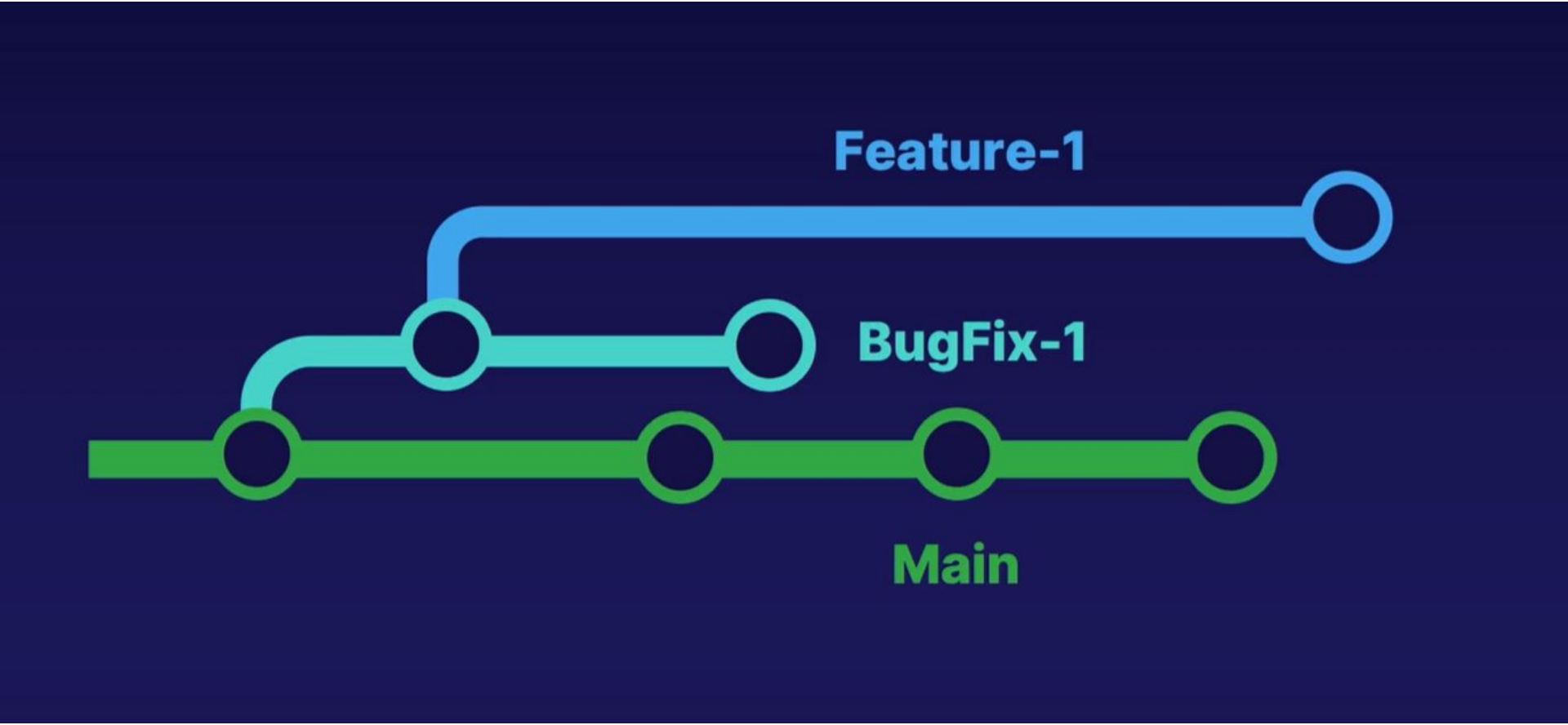
Configuring Branches

Configuring Branches



Configuring Branches

Configuring Branches



Configuring Branches

Configuring Branches

1. **Require a Minimum Number of Reviewers**
   Require approval from a specified number of reviewers on pull requests.

2. **Check for Linked Work Items**
   Encourage traceability by checking for linked work items on pull requests.

3. **Check for Comment Resolution**
   Check to see that all comments have been resolved on pull requests.

4. **Limit Merge Types**
   Control branch history by limiting the available types of merge when pull requests are completed.

Configuring Branches

5 **Build Validation**

Validate code by pre-merging and building pull request changes.

6 **Status Checks**

Require other services to post successful statuses to complete pull requests.

7 **Automatically Included Reviewers**

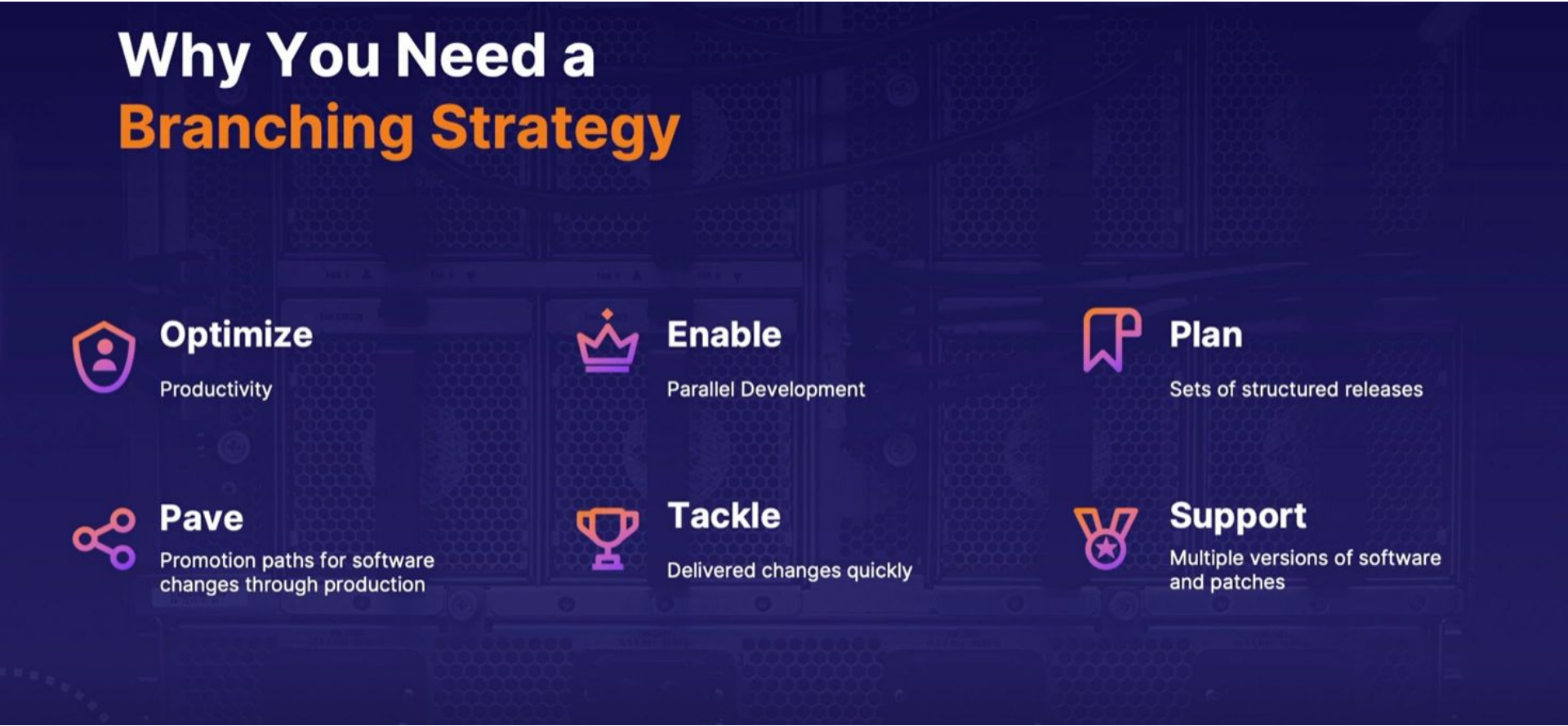
Designate code reviewers to automatically include when pull requests change certain areas of code (manual approvals).

8 **Restrict Who Can Push to the Branch**

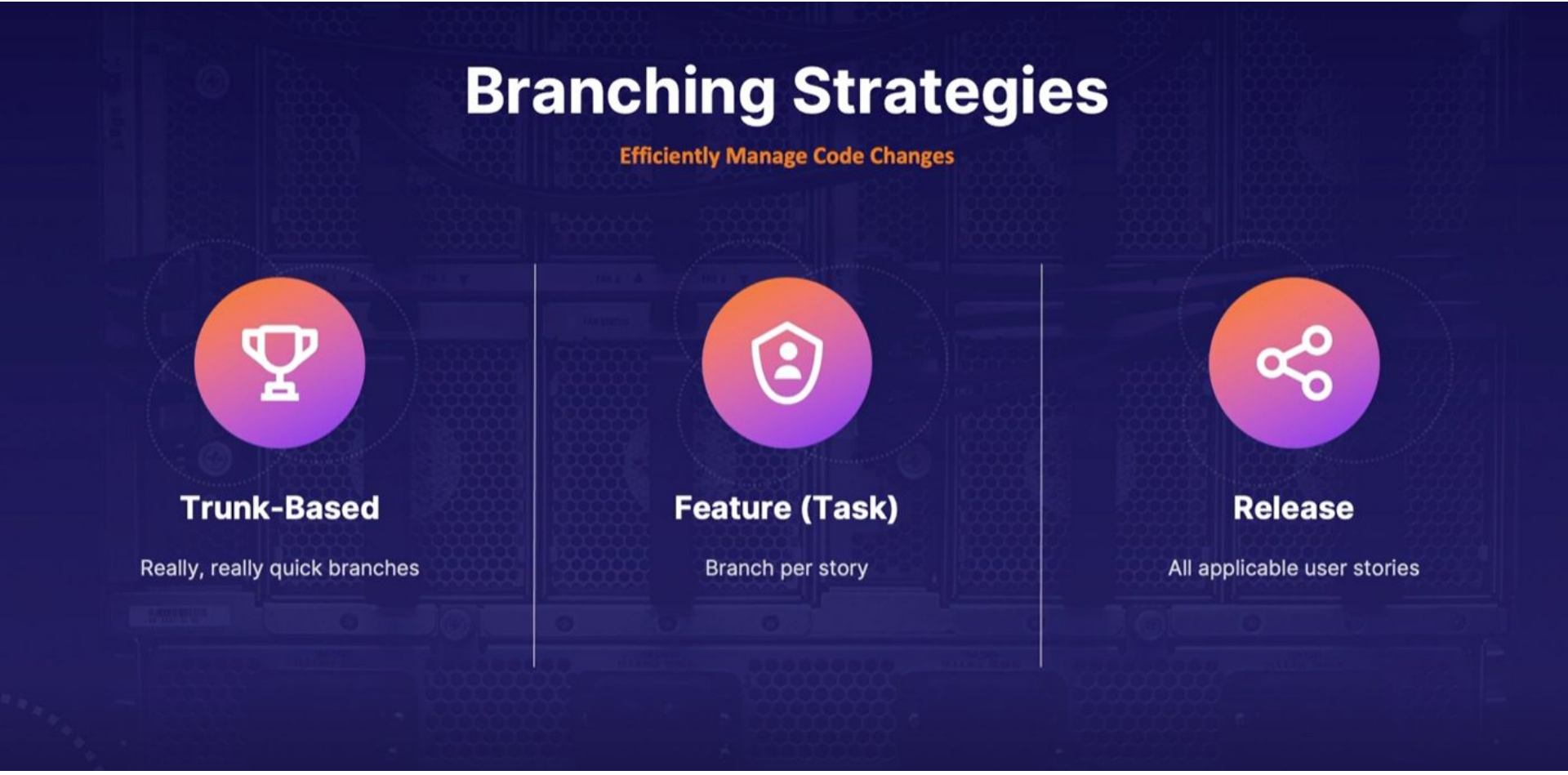
Use security permissions to allow only certain collaborators the ability to push to the branch.

Configuring Branches

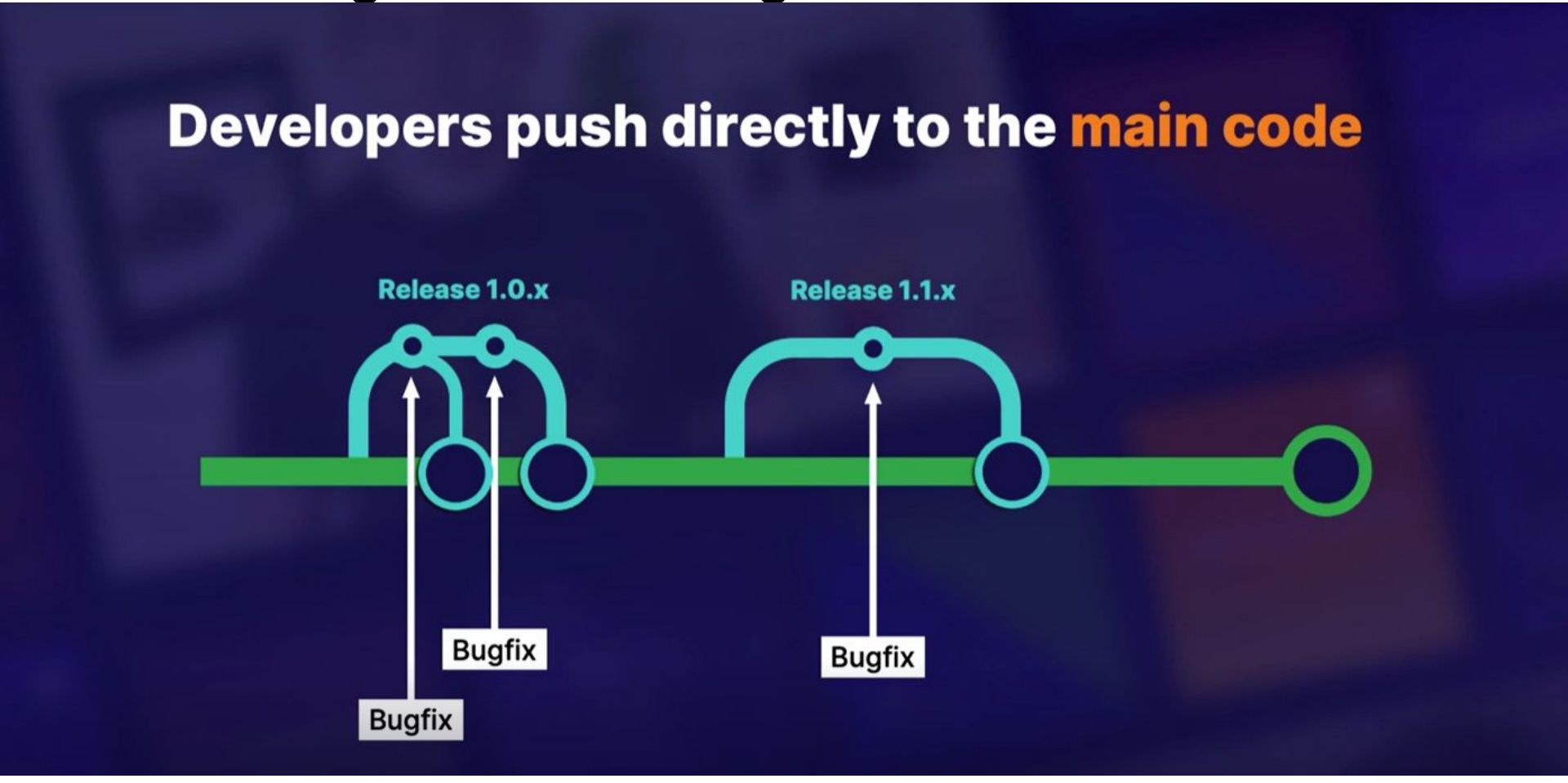
Discovering Branch Strategies



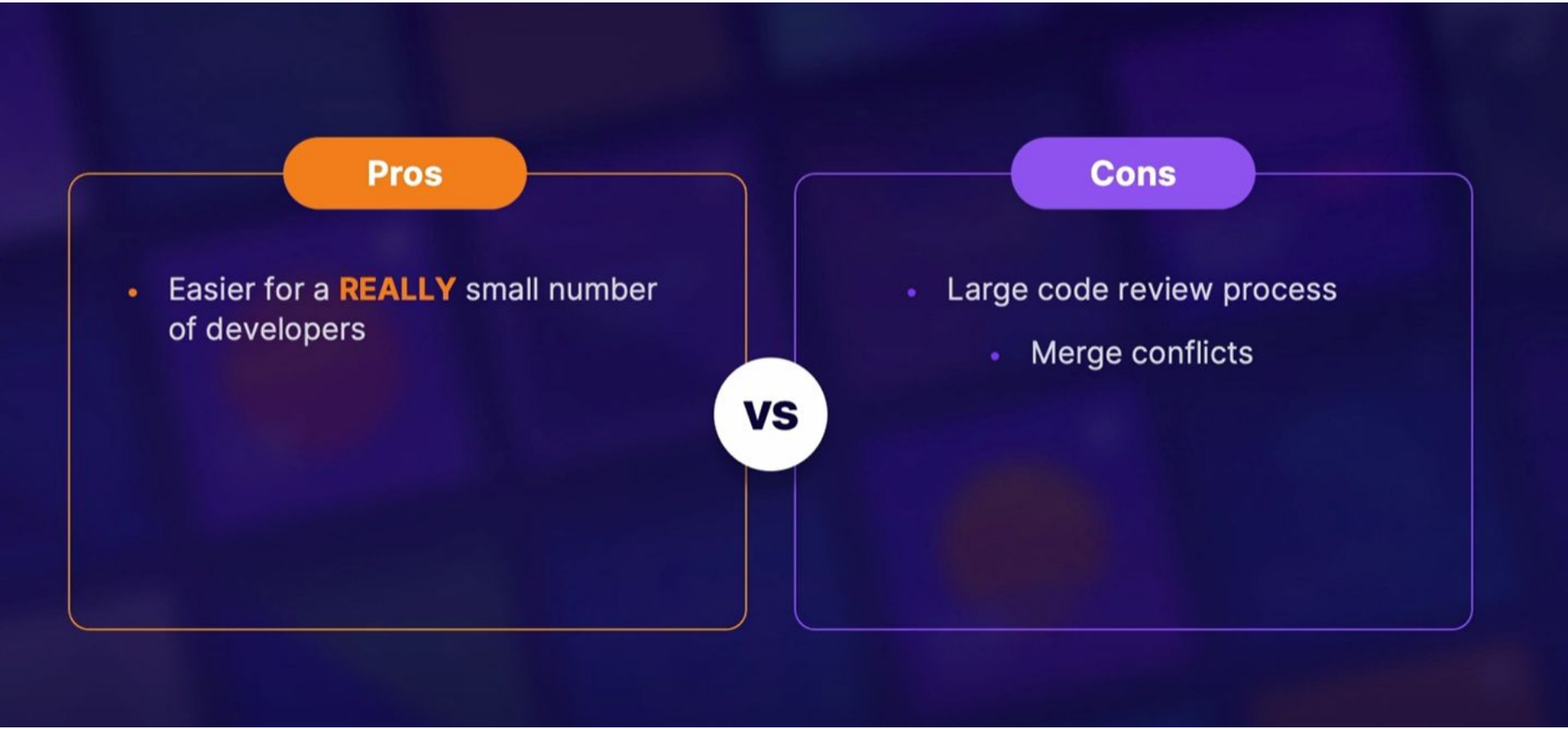
Discovering Branch Strategies



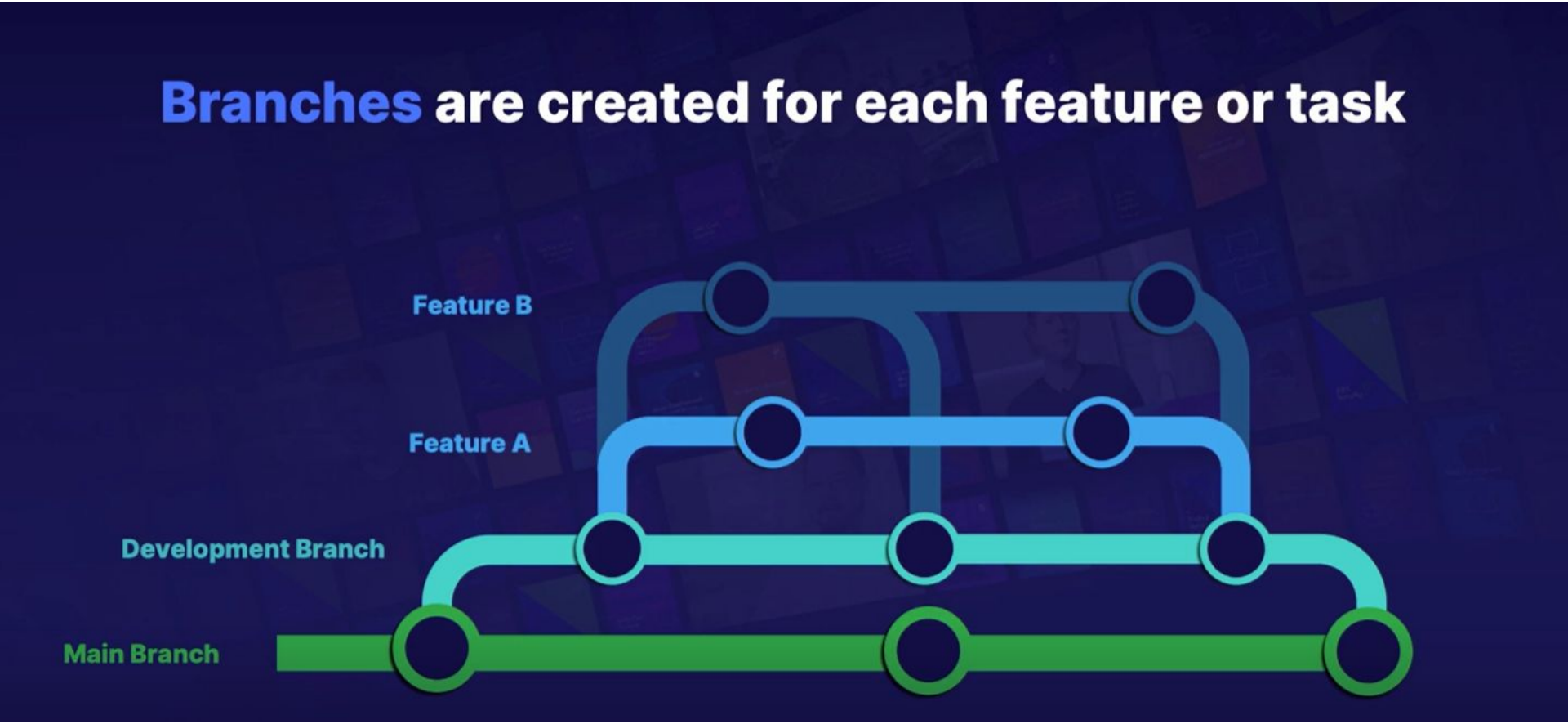
Discovering Branch Strategies



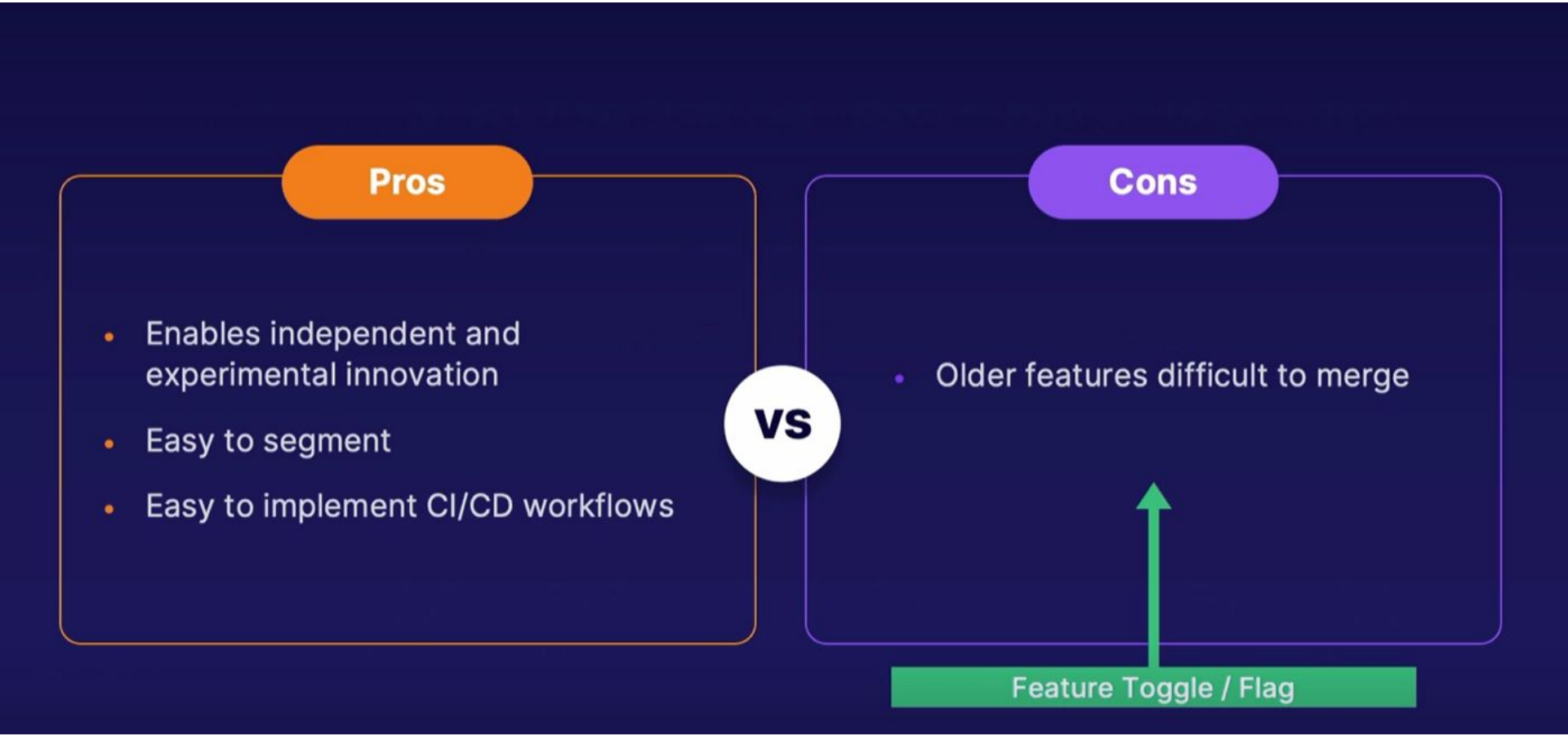
Discovering Branch Strategies



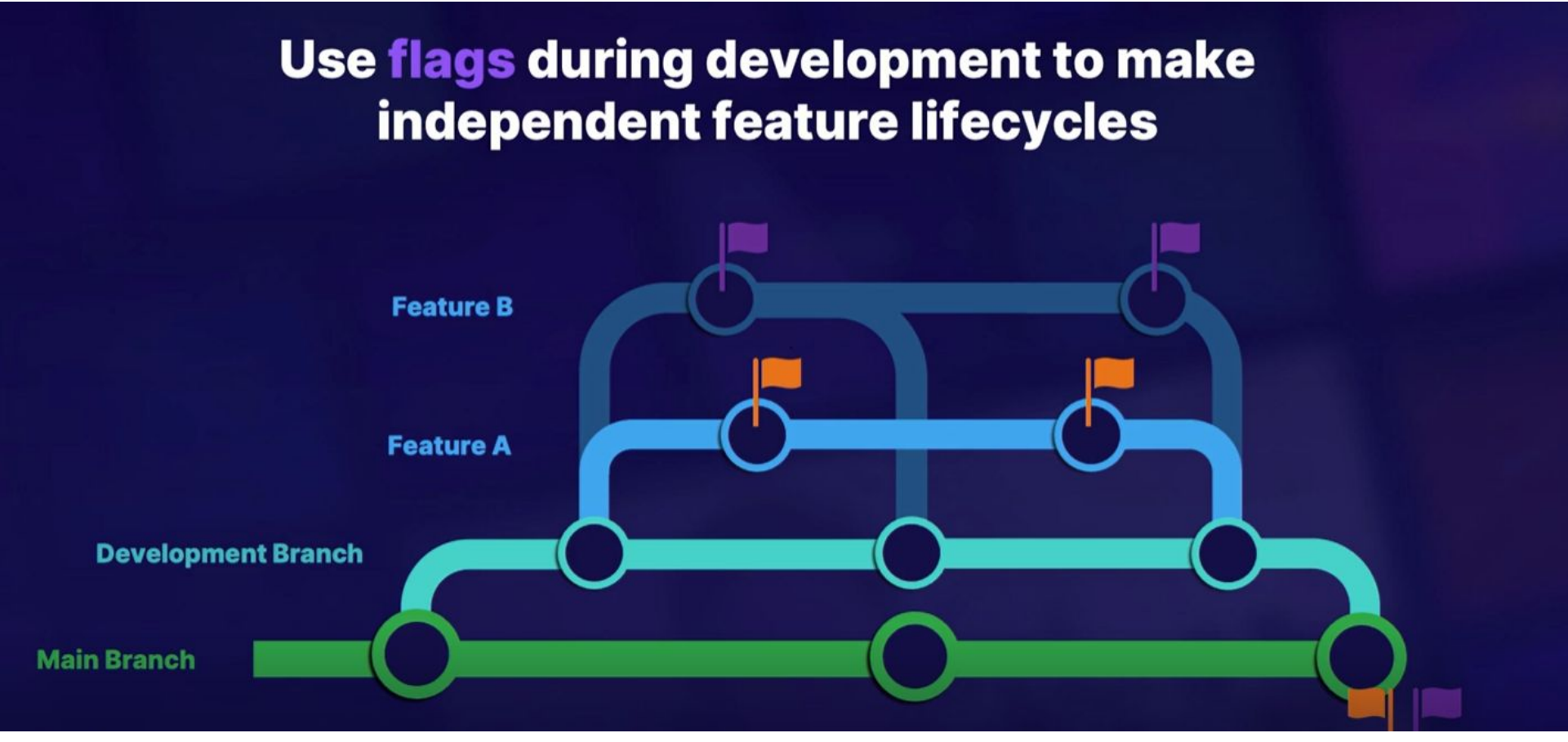
Discovering Branch Strategies



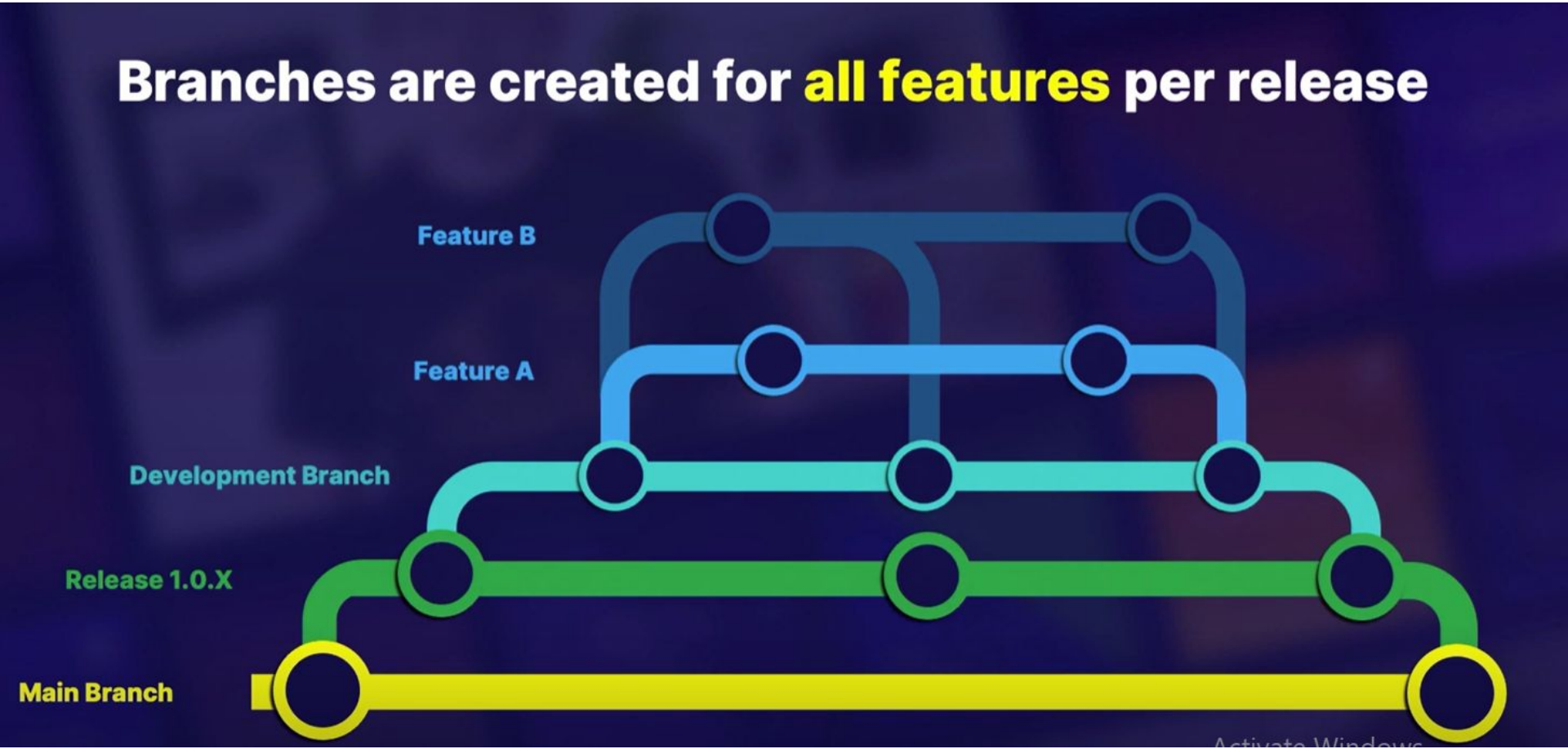
Discovering Branch Strategies



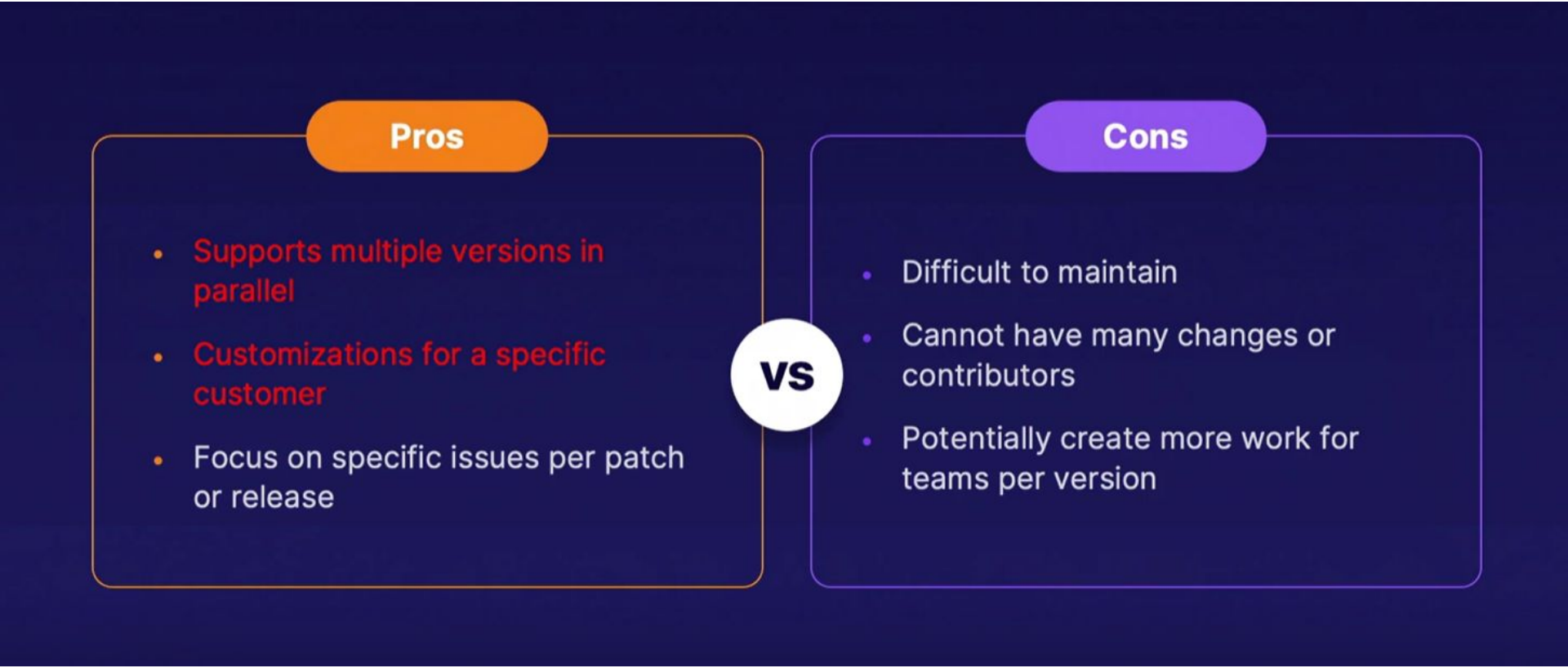
Discovering Branch Strategies



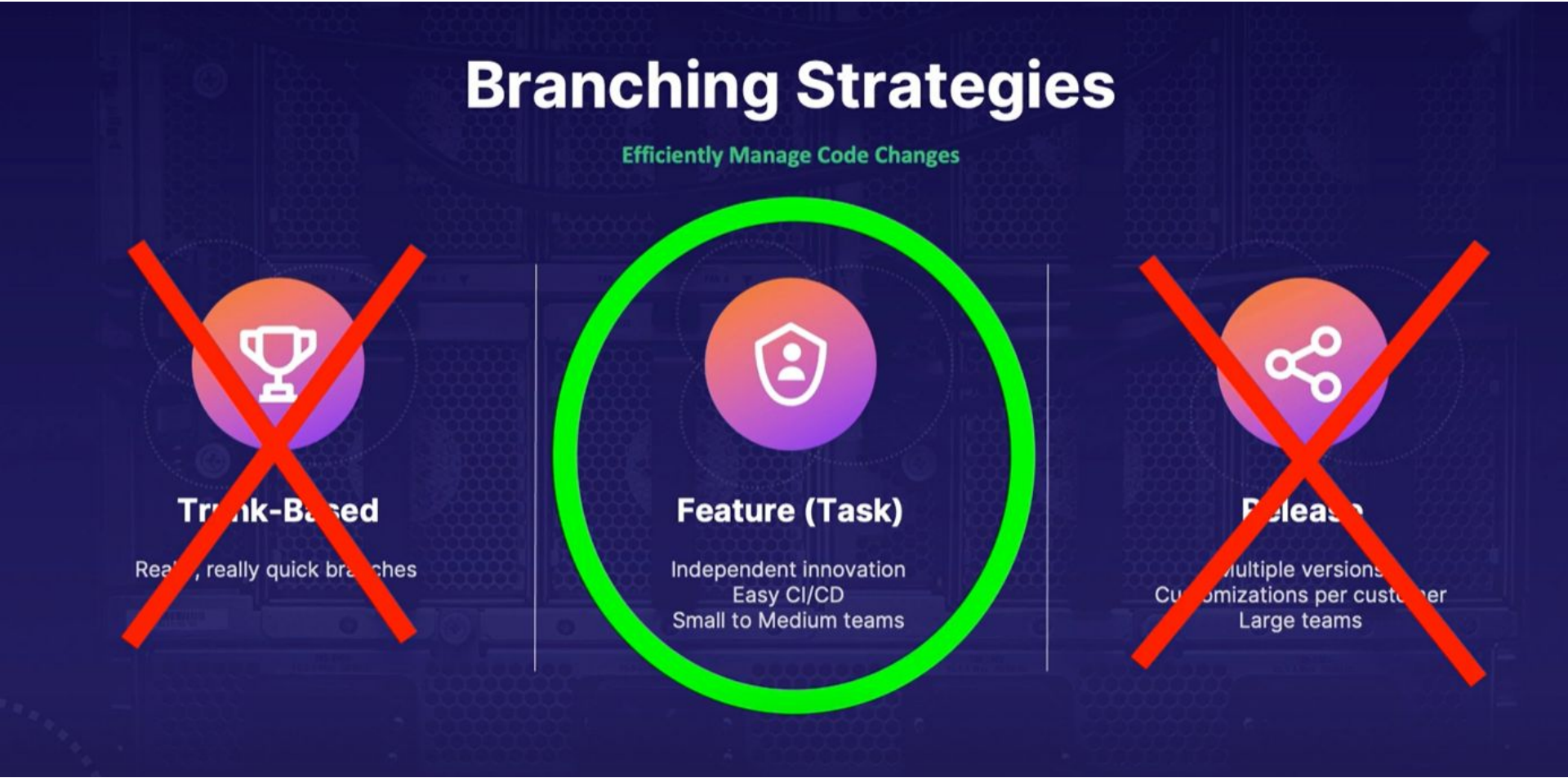
Discovering Branch Strategies



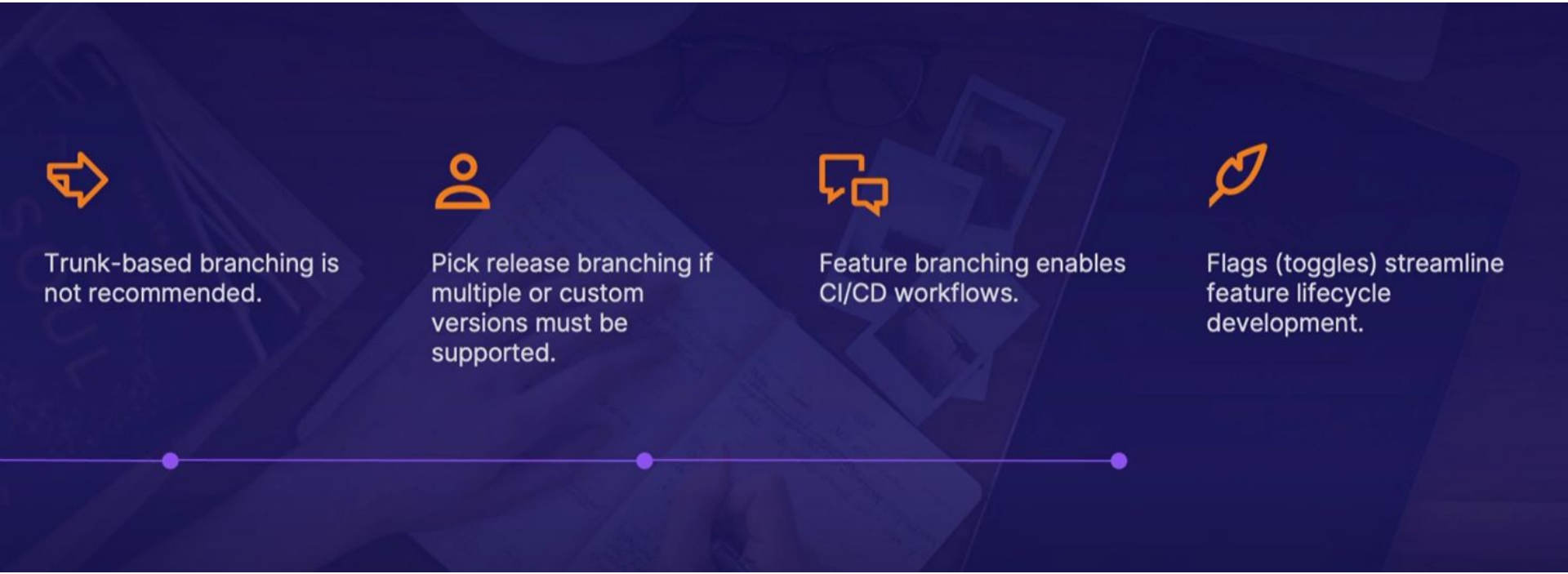
Discovering Branch Strategies



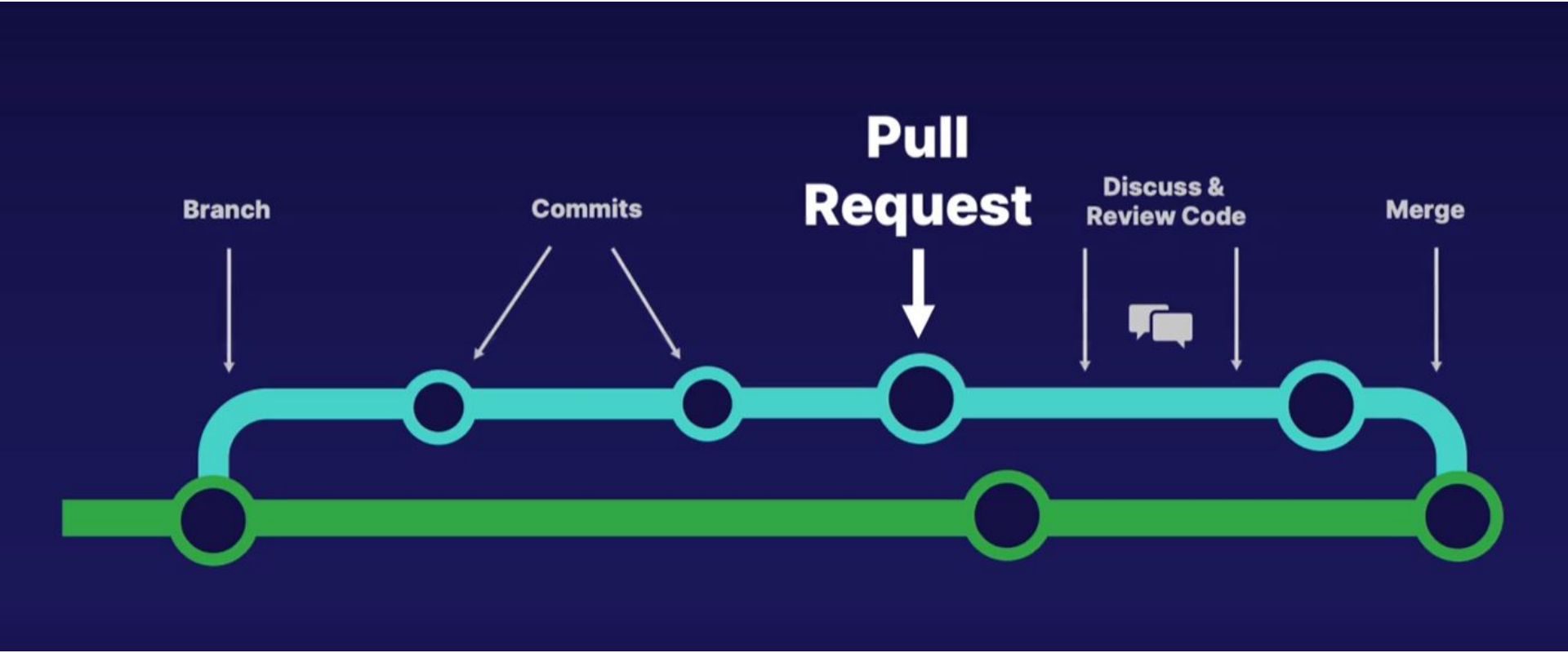
Discovering Branch Strategies



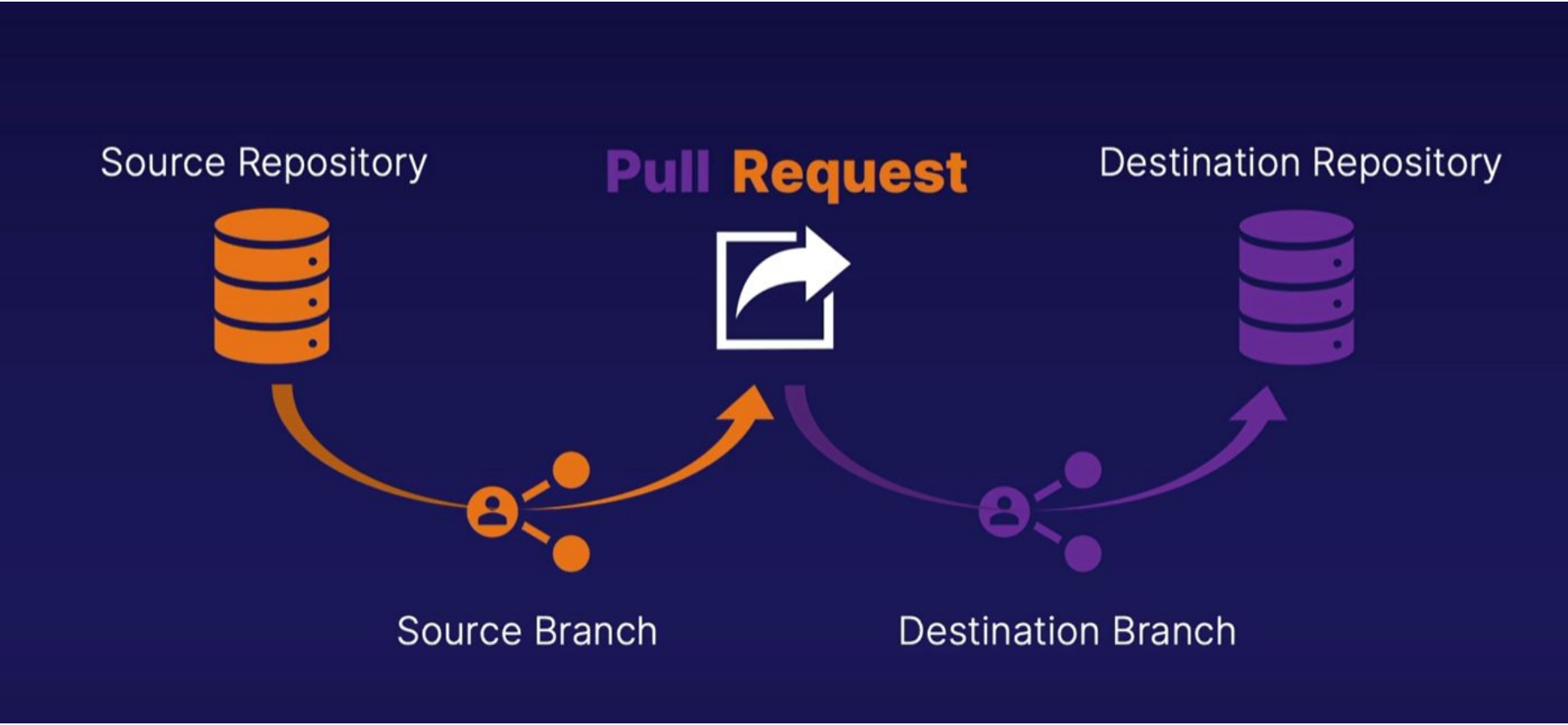
Branch Strategies Summary



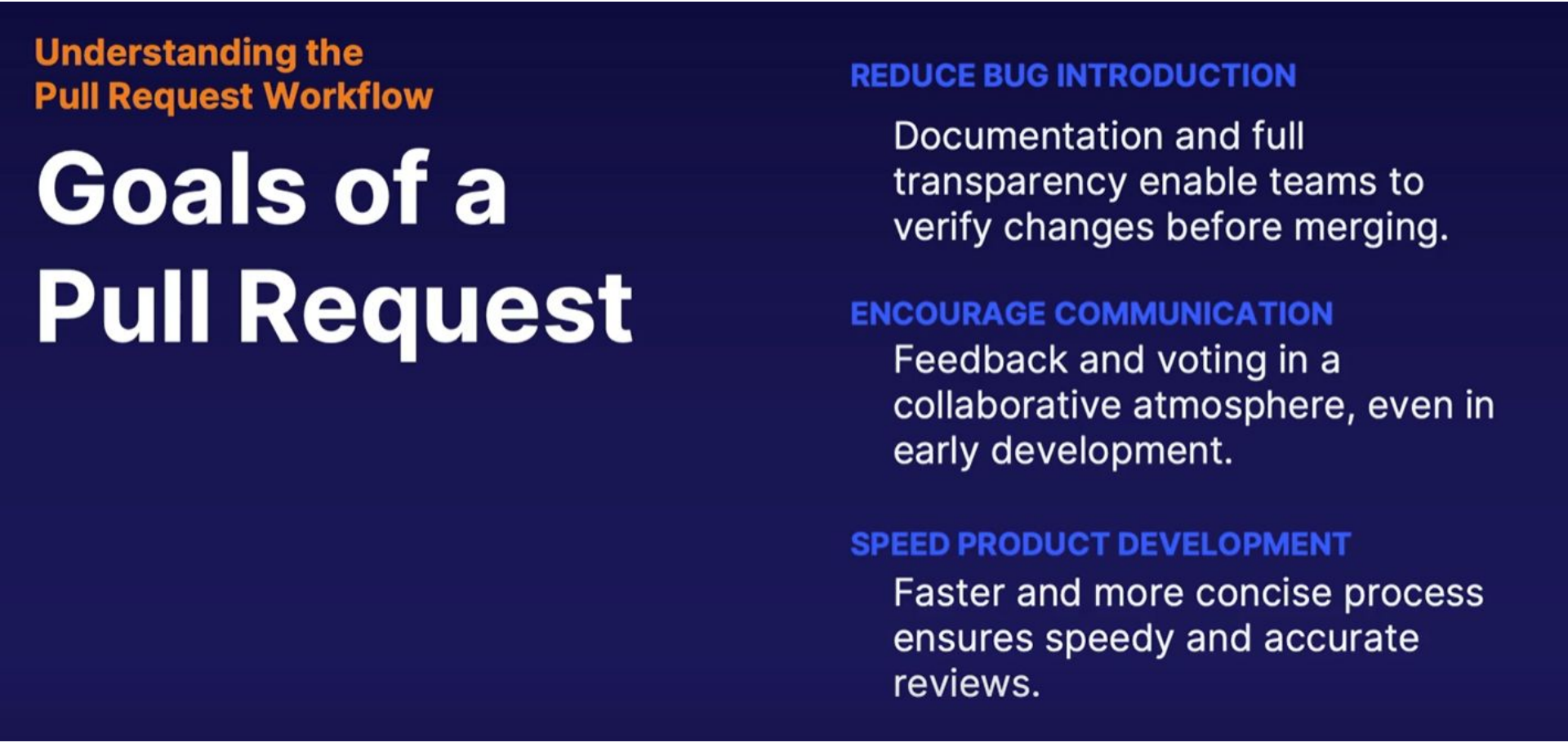
Understanding the Pull Request Workflow



Understanding the Pull Request Workflow



Understanding the Pull Request Workflow



Understanding the Pull Request Workflow

1. **What**
   An explanation of the changes that you made
   (context)

2. **Why**
   The business or technical goal of the changes
   (the bigger picture)

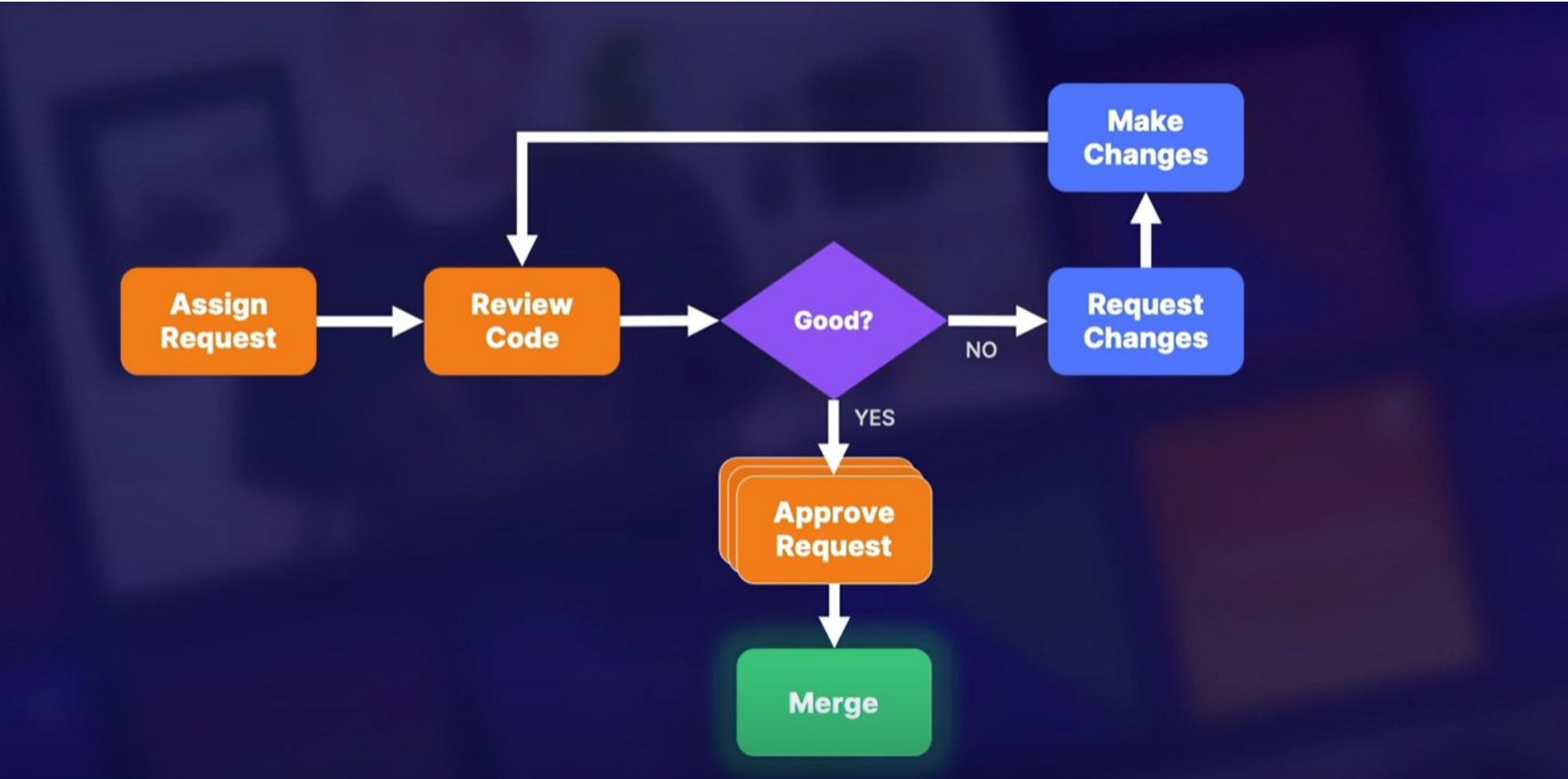
3. **How**
   Design decisions and rationale on code change approaches
   (reasoning)

4. **Tests**
   Verification of tests performed and all results
   (verification)

5. **References**
   Work Items, screenshots, links, additional downloads, or documentation
   (validation)

6. **The Rest**
   Challenges, improvements, optimizations, budget requirements
   (other)

Understanding the Pull Request Workflow



Understanding the Pull Request Workflow

Understanding the Pull Request Workflow

Riya

DEVELOPER

Author

RESPONSIBILITIES

- Ensuring the merge can succeed
- What/Why/How is documented
- Applies suggested code changes

Understanding the Pull Request Workflow

Tom

TEST/QA

Reviewer

RESPONSIBILITIES

- Review the request details
- Review the code*
- Make comments and suggestions
- Approve the code for merging

Understanding the Pull Request Workflow

Avery

SECURITY

Assignee

RESPONSIBILITIES

- Determine merge timing
- Confirm stakeholder approval
- Delete the old branch

Exploring Code Reviews

1

Code Review Assignments

Automatically choose reviewers.

2

Scheduled Reminders

Send messages to focus attention.

3

Pull Analytics

Metrics help find opportunities!

Exploring Static Code Analysis

Exploring Static Code Analysis

1

Size Limits

Less than 400 lines of code.

Less than 60 minutes of review at a time.

2

Annotations

Authors should guide reviewers.

Provides more in-depth context.

3

Checklists

People make mistakes.

The same ones.

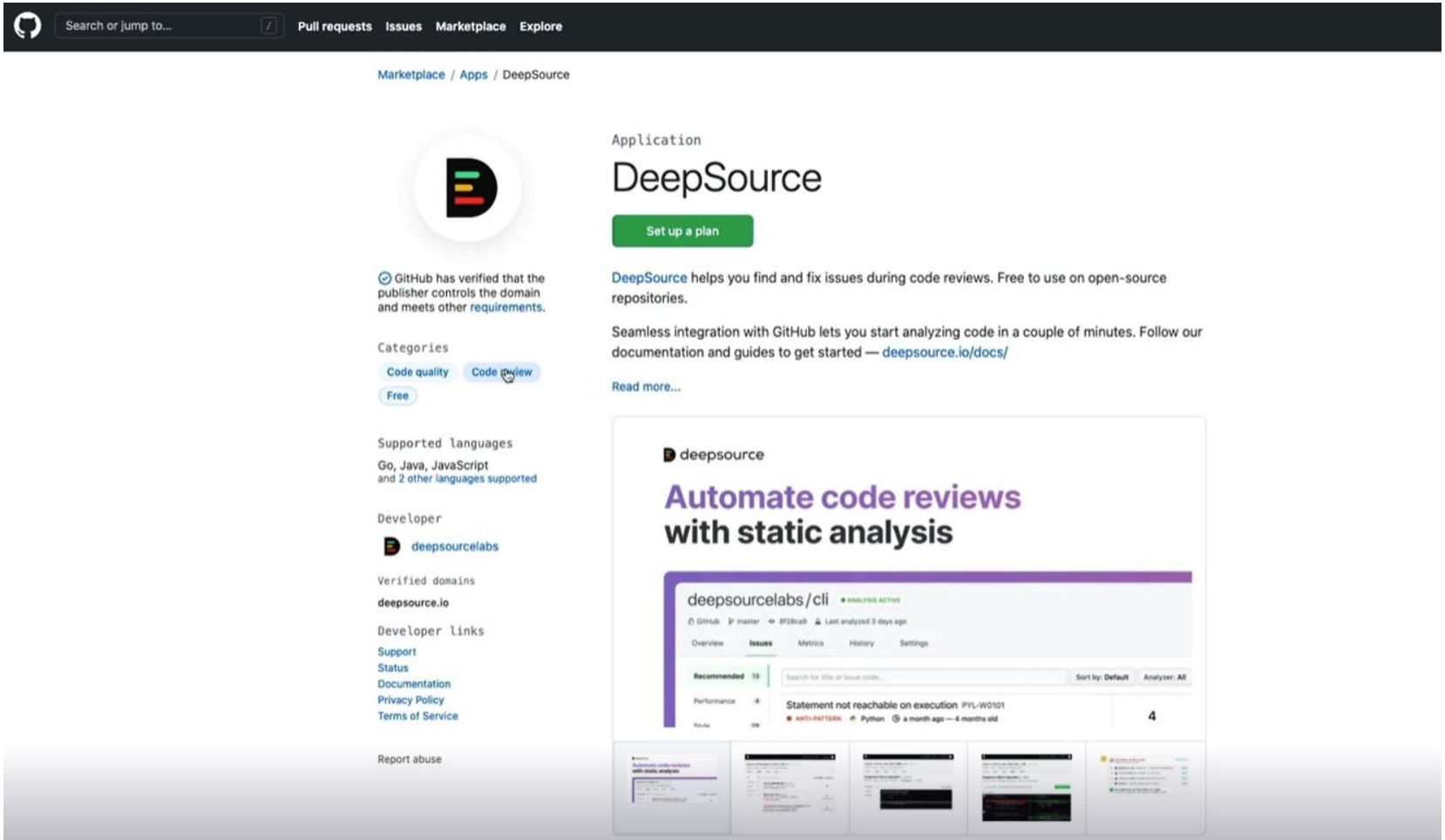
A lot.

Exploring Static Code Analysis

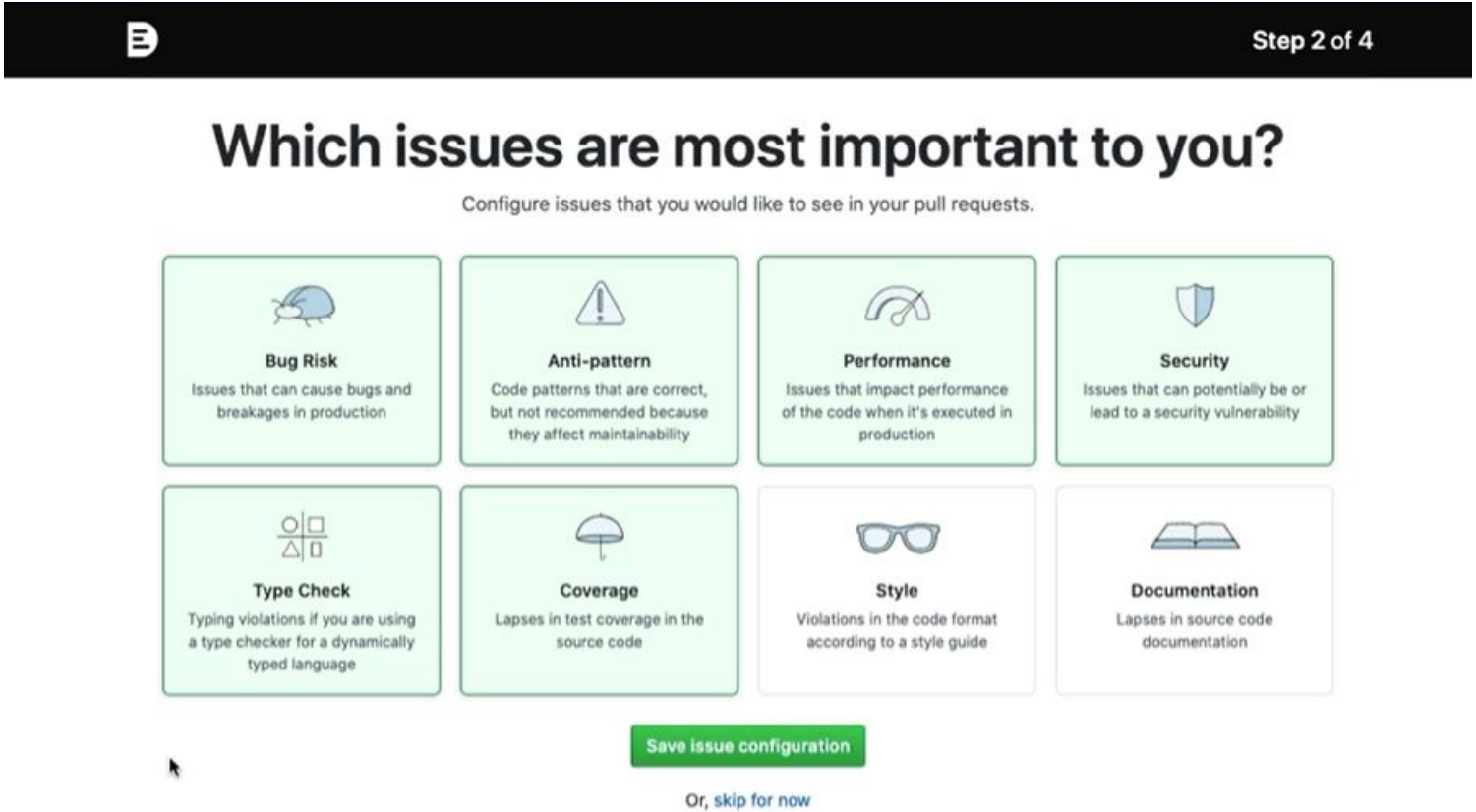
Code Scanning

- Coding Errors
- Security Vulnerabilities
- Find, Triage, Prioritize

Exploring Static Code Analysis



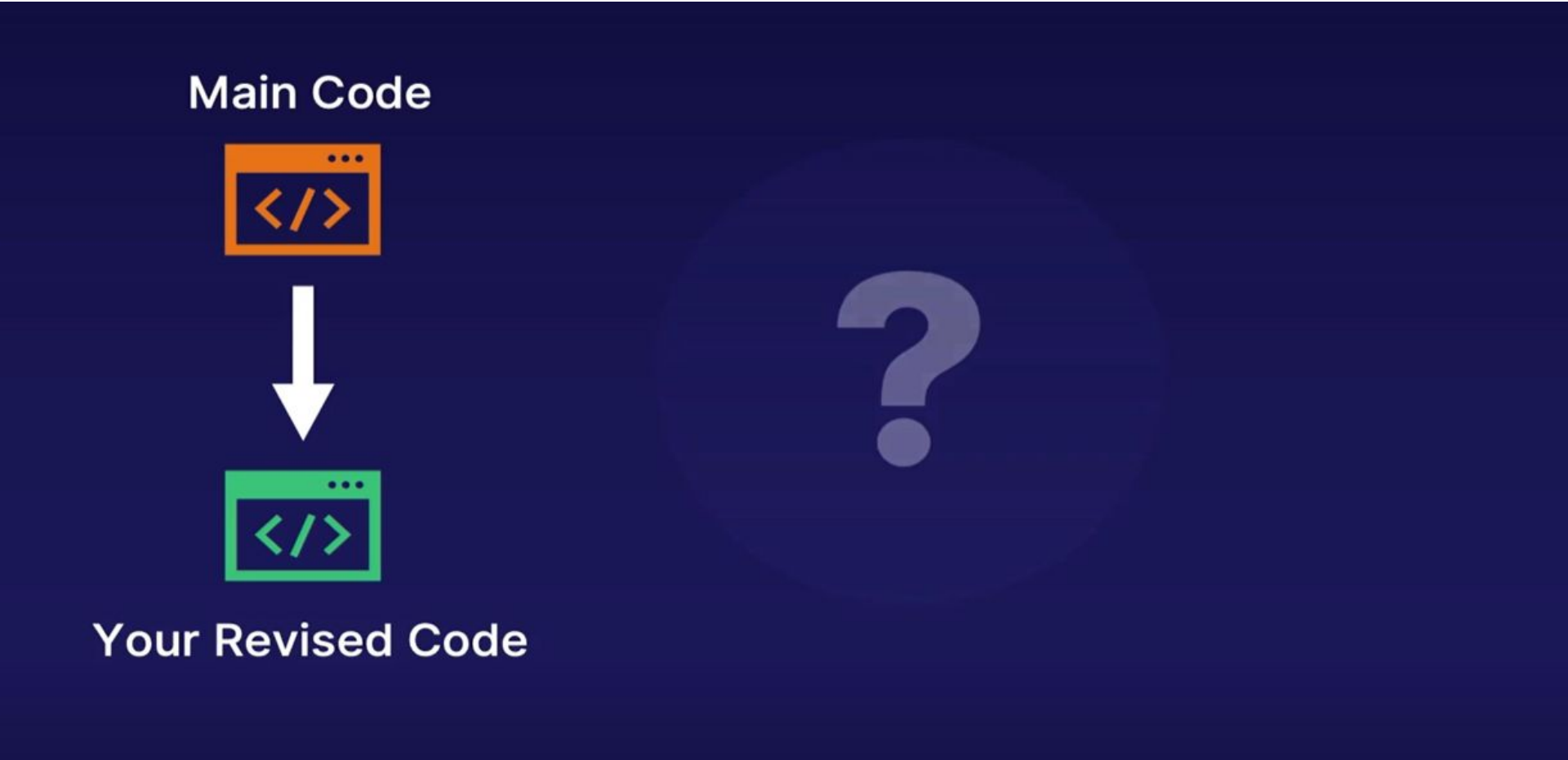
Exploring Static Code Analysis



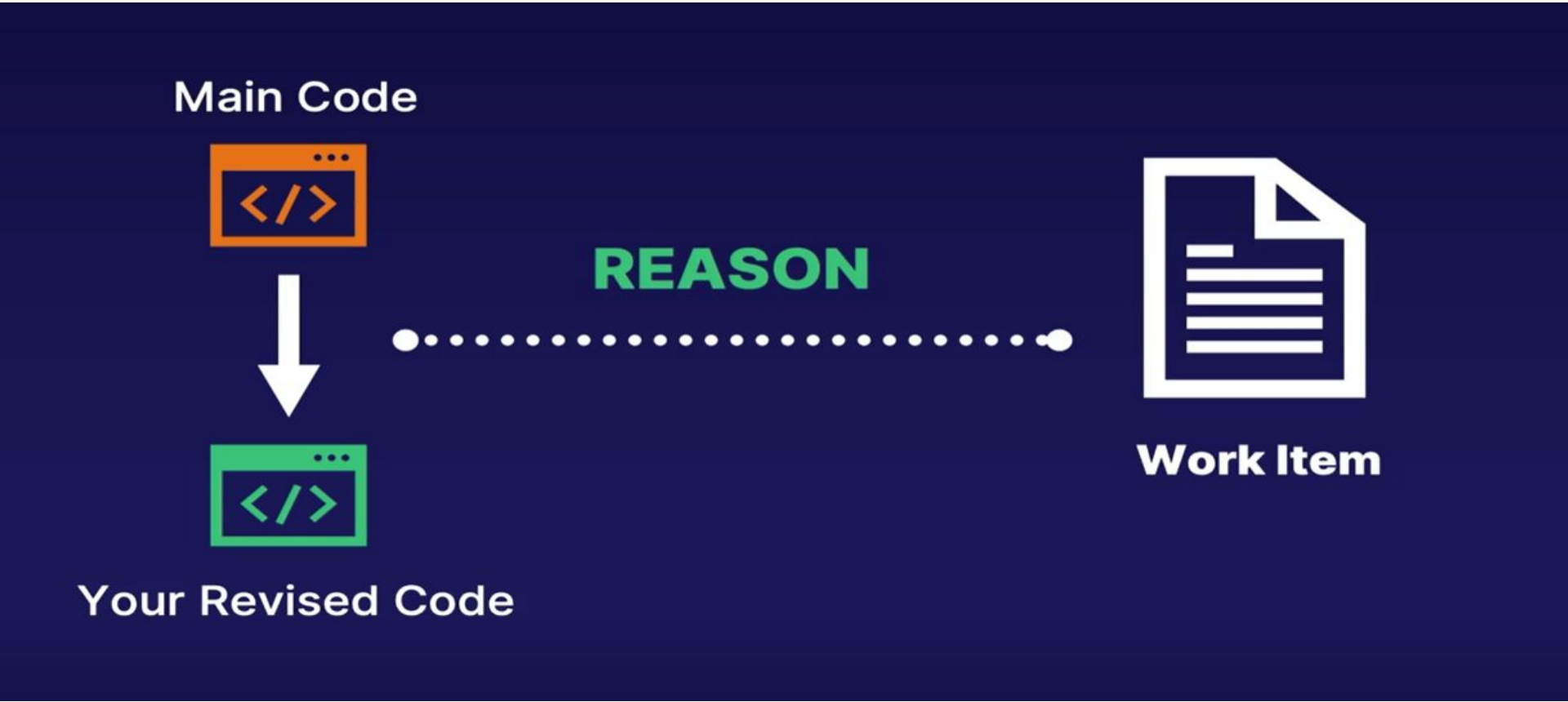
Exploring Static Code Analysis

- **Code Analysis Approach:** combination of both Static and Dynamic.
- **Integrate Code Scanning Tools** to automatically test quality and security.
- **Use annotations and checklists** to speed up code reviews and analysis.

Using Pull Requests with Work Items

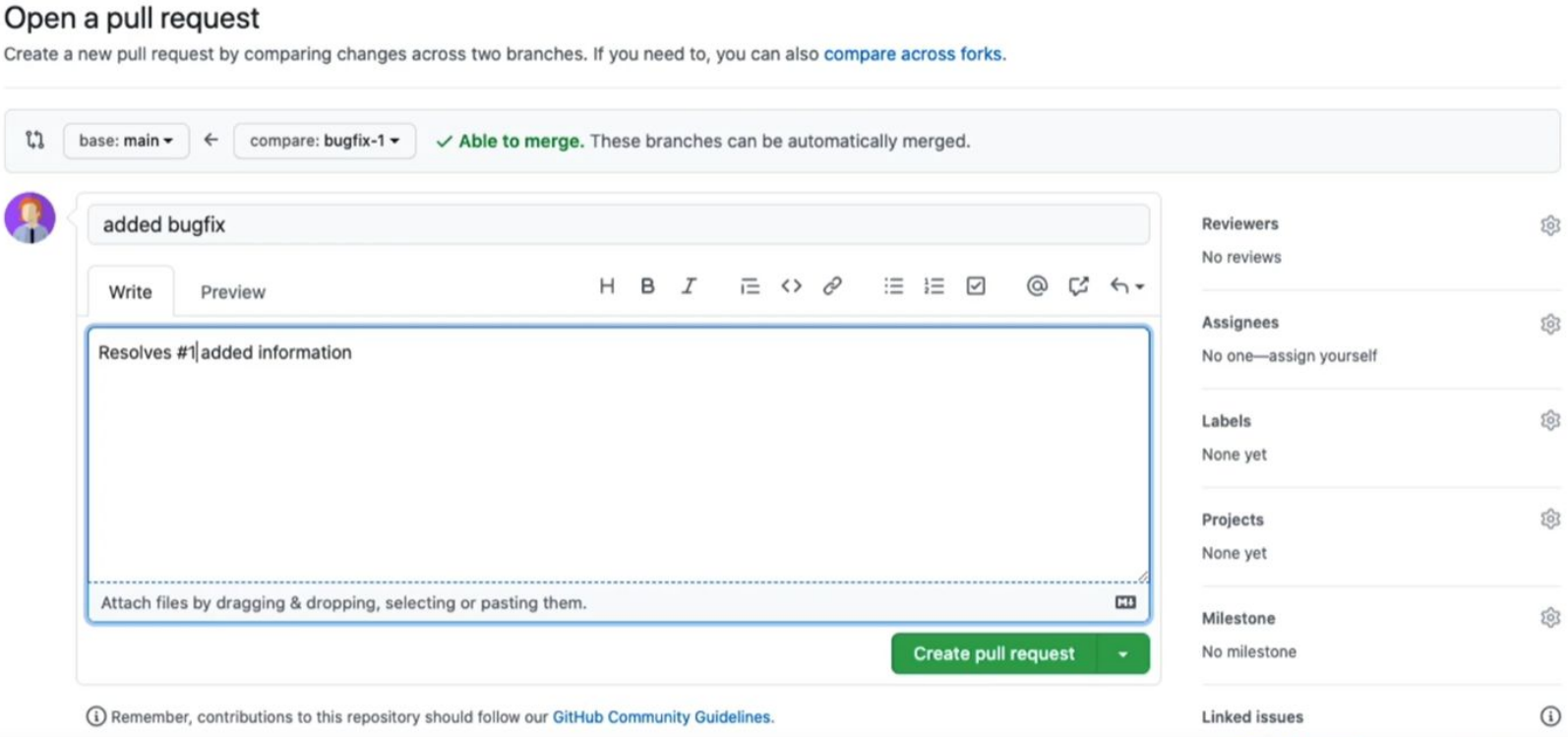


Using Pull Requests with Work Items

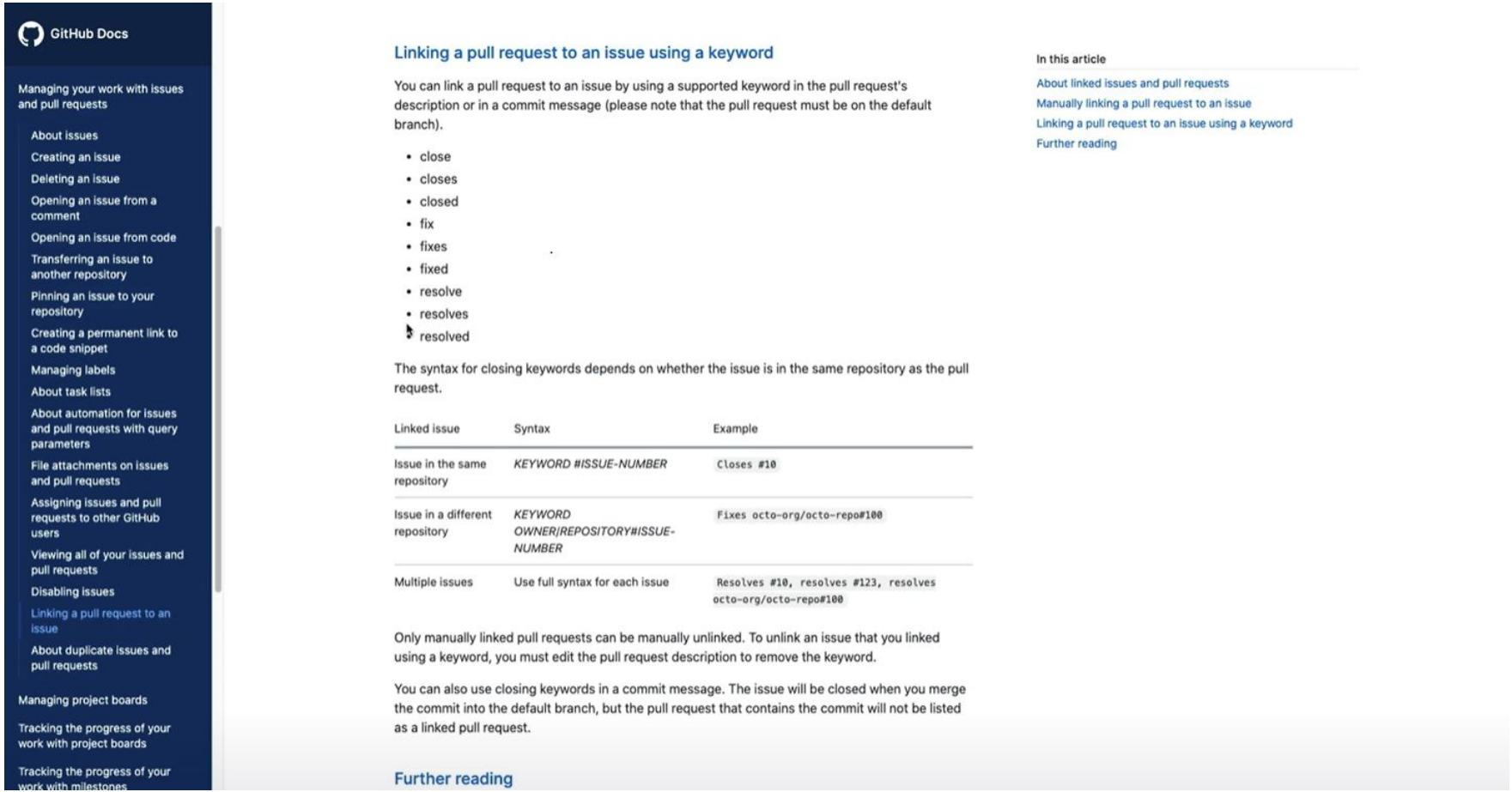


Using Pull Requests with Work Items

Using Pull Requests with Work Items



Using Pull Requests with Work Items



GitHub Docs

Managing your work with issues and pull requests

- About issues
- Creating an issue
- Deleting an issue
- Opening an issue from a comment
- Opening an issue from code
- Transferring an issue to another repository
- Pinning an issue to your repository
- Creating a permanent link to a code snippet
- Managing labels
- About task lists
- About automation for issues and pull requests with query parameters
- File attachments on issues and pull requests
- Assigning issues and pull requests to other GitHub users
- Viewing all of your issues and pull requests
- Disabling issues
- Linking a pull request to an issue
- About duplicate issues and pull requests

Managing project boards

Tracking the progress of your work with project boards

Tracking the progress of your work with milestones

Linking a pull request to an issue using a keyword

In this article

You can link a pull request to an issue by using a supported keyword in the pull request's description or in a commit message (please note that the pull request must be on the default branch).

- close
- closes
- closed
- fix
- fixes
- fixed
- resolve
- resolves
- resolved

The syntax for closing keywords depends on whether the issue is in the same repository as the pull request.

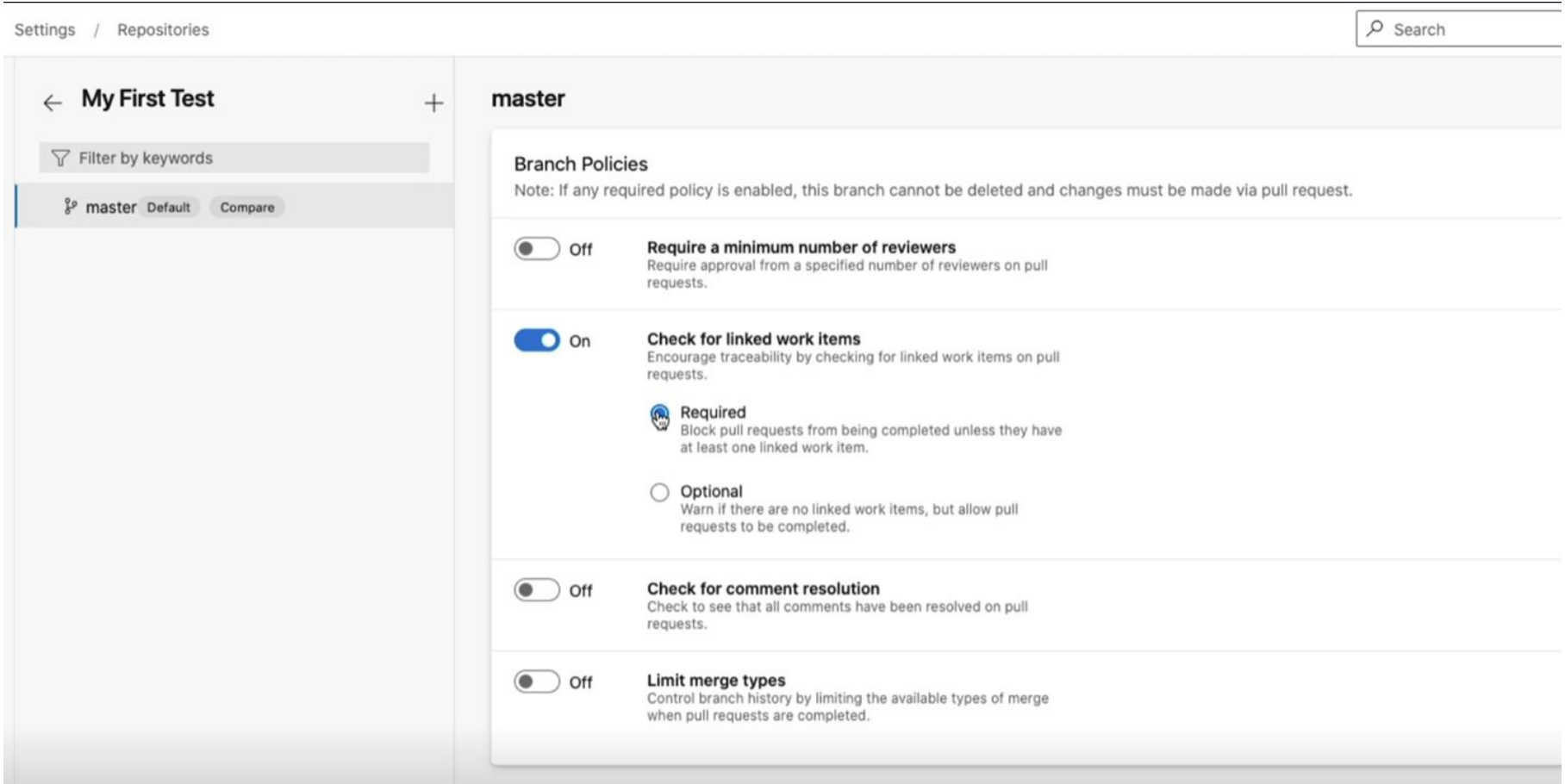
Linked issue	Syntax	Example
Issue in the same repository	*KEYWORD #ISSUE-NUMBER*	`Closes #10`
Issue in a different repository	*KEYWORD OWNER/REPOSITORY#ISSUE-NUMBER*	`Fixes octo-org/octo-repo#100`
Multiple issues	Use full syntax for each issue	`Resolves #10, resolves #123, resolves octo-org/octo-repo#100`

Only manually linked pull requests can be manually unlinked. To unlink an issue that you linked using a keyword, you must edit the pull request description to remove the keyword.

You can also use closing keywords in a commit message. The issue will be closed when you merge the commit into the default branch, but the pull request that contains the commit will not be listed as a linked pull request.

Further reading

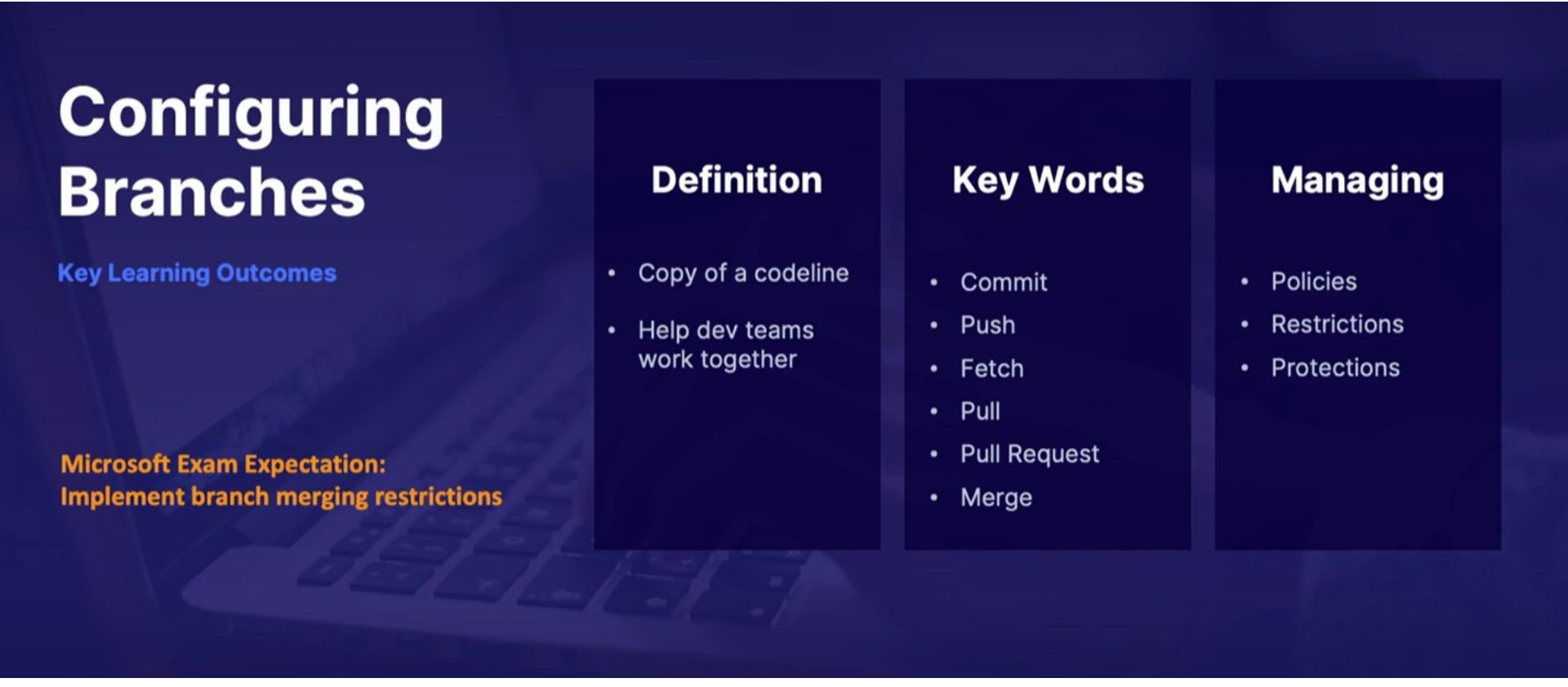
Using Pull Requests with Work Items



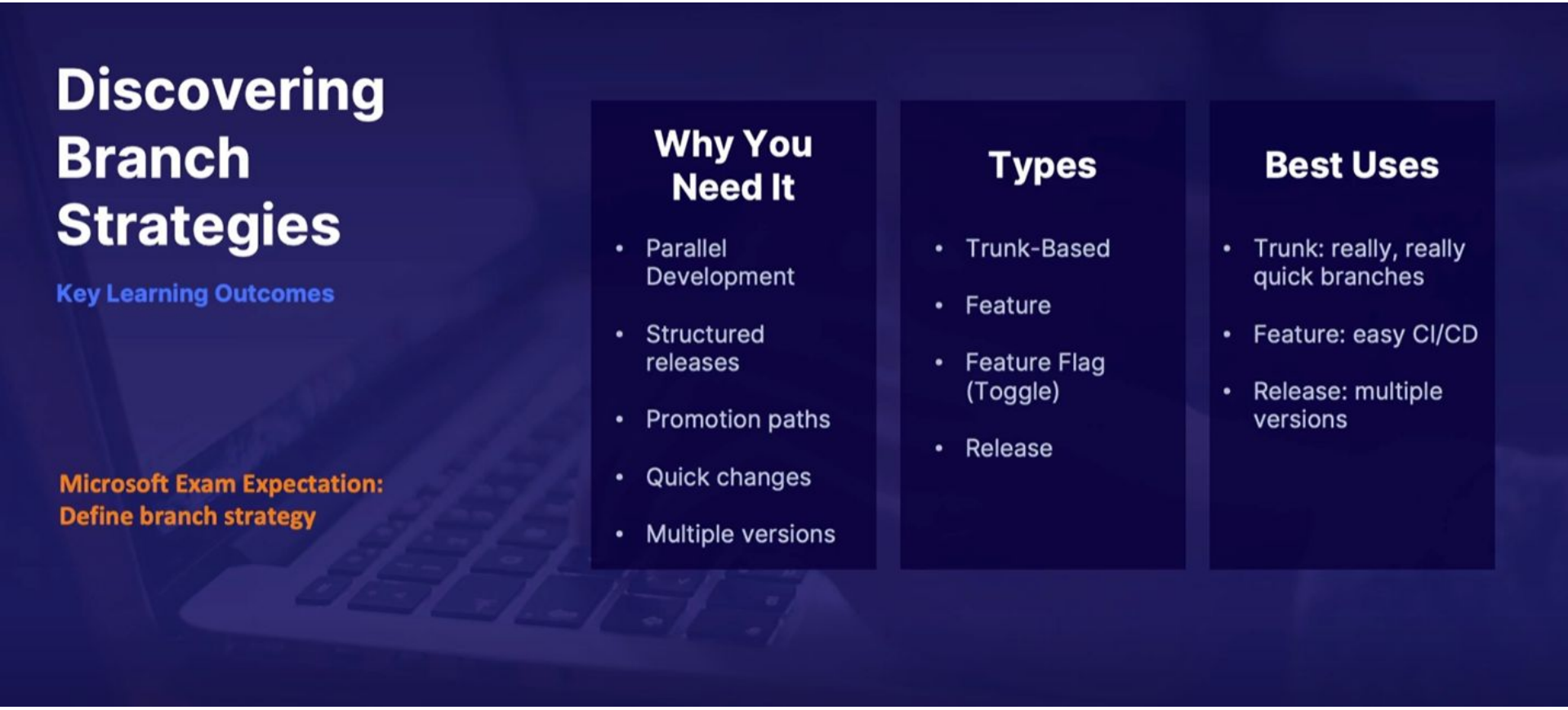
Using Pull Requests with Work Items

- Use "#" to add work item references in a pull request.
- It's recommended to always correlate work items with pull requests.
- It's possible to close work items with completed pull requests.

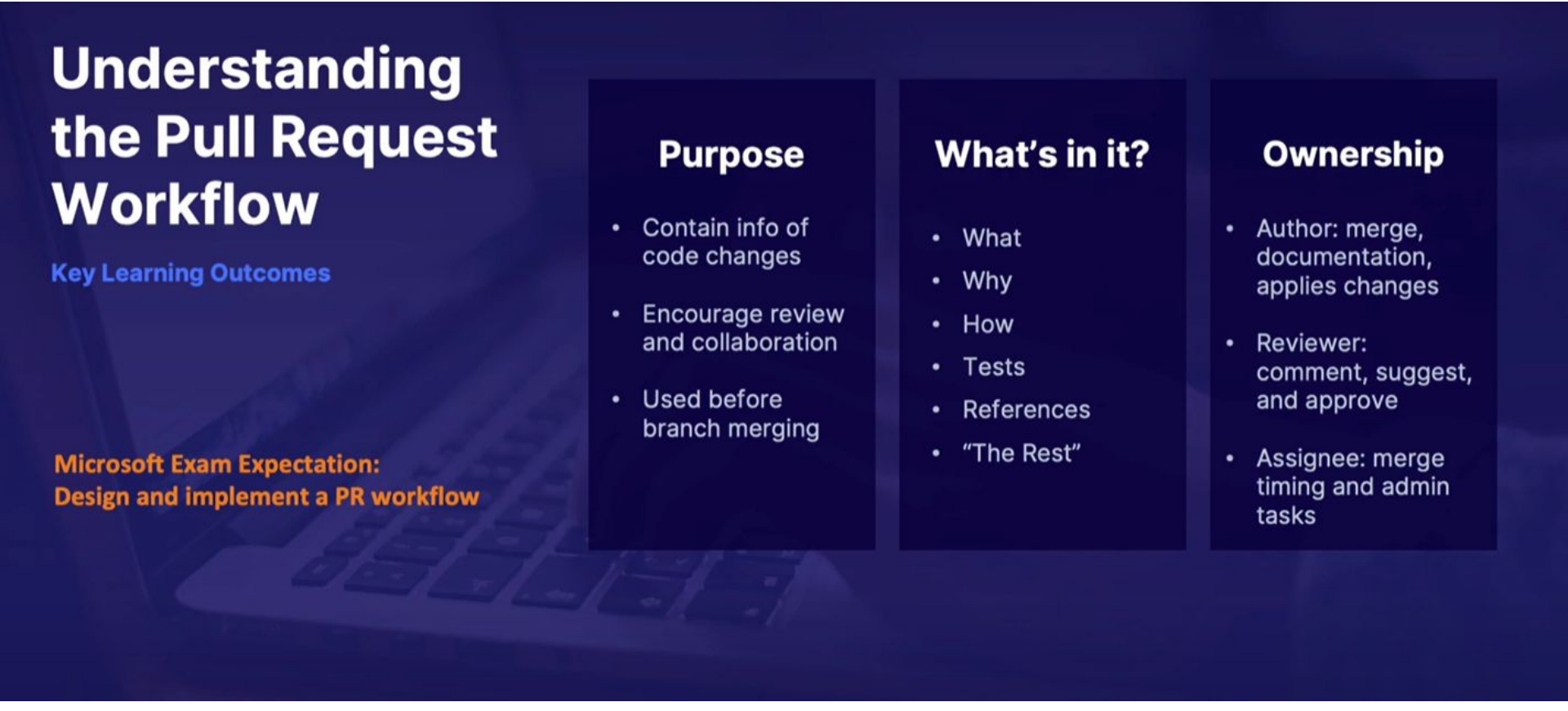
Summary


Configuring Branches
Key Learning Outcomes
Microsoft Exam Expectation:
Implement branch merging restrictions
Definition
Copy of a codeline
Help dev teams work together
Key Words
Commit
Push
Fetch
Pull
Pull Request
Merge
Managing
Policies
Restrictions
Protections

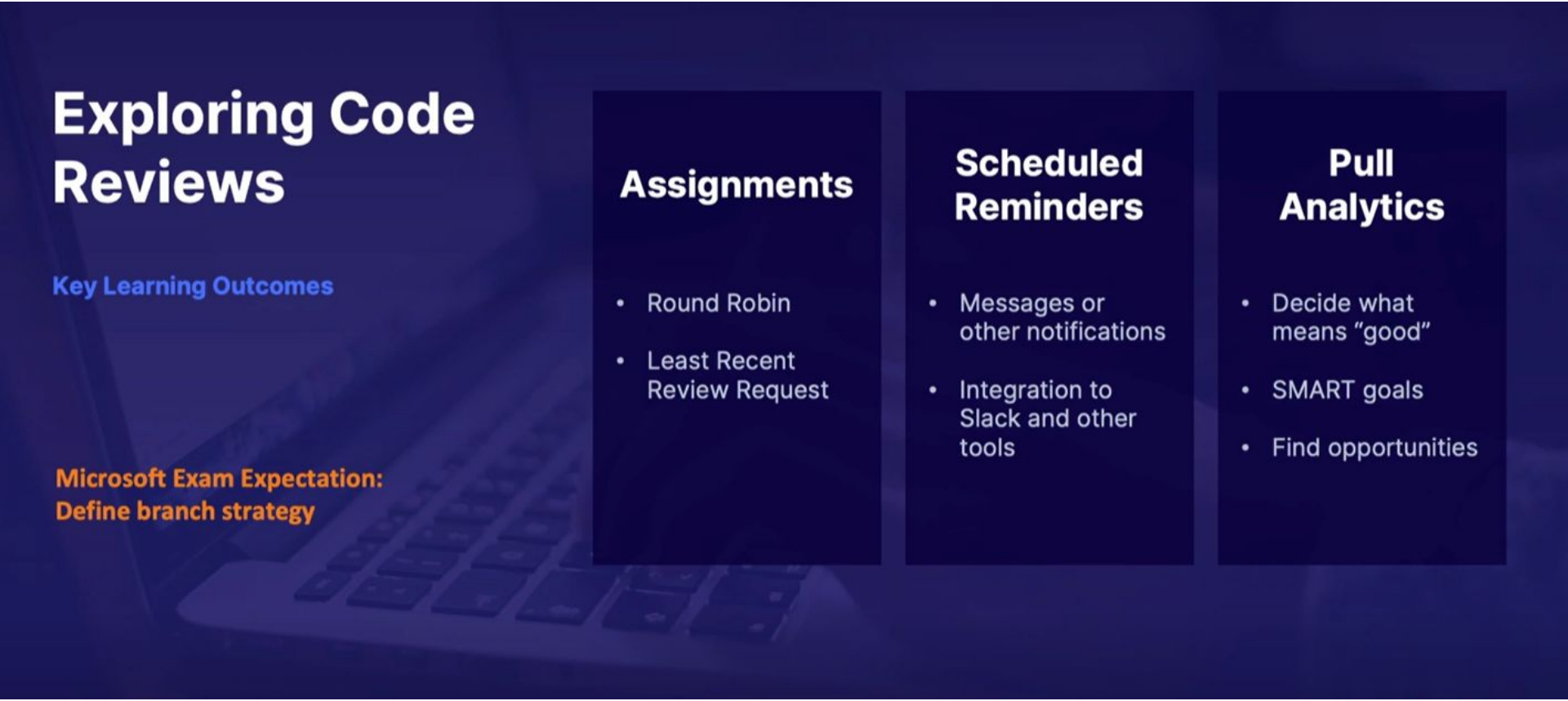
Summary


Discovering Branch Strategies
Key Learning Outcomes
Microsoft Exam Expectation: Define branch strategy
Why You Need It
Parallel Development
Structured releases
Promotion paths
Quick changes
Multiple versions
Types
Trunk-Based
Feature
Feature Flag (Toggle)
Release
Best Uses
Trunk: really, really quick branches
Feature: easy CI/CD
Release: multiple versions

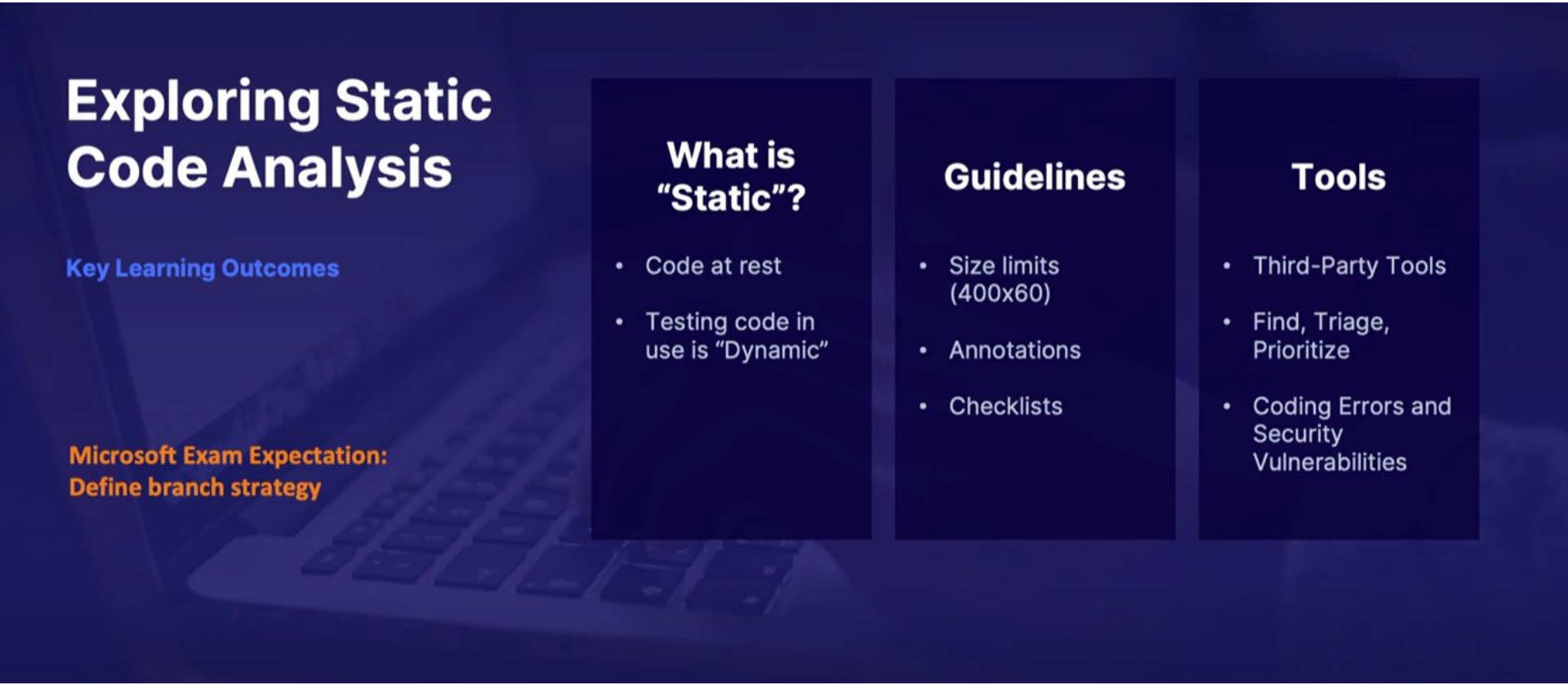
Summary


Understanding the Pull Request Workflow
Key Learning Outcomes
Microsoft Exam Expectation:
Design and implement a PR workflow
Purpose
Contain info of code changes
Encourage review and collaboration
Used before branch merging
What's in it?
What
Why
How
Tests
References
"The Rest"
Ownership
Author: merge, documentation, applies changes
Reviewer: comment, suggest, and approve
Assignee: merge timing and admin tasks

Summary


Exploring Code Reviews
Key Learning Outcomes
Microsoft Exam Expectation:
Define branch strategy
Assignments
Round Robin
Least Recent Review Request
Scheduled Reminders
Messages or other notifications
Integration to Slack and other tools
Pull Analytics
Decide what means "good"
SMART goals
Find opportunities

Summary


Exploring Static Code Analysis
Key Learning Outcomes
Microsoft Exam Expectation: Define branch strategy
What is "Static"?
Code at rest
Testing code in use is "Dynamic"
Guidelines
Size limits (400x60)
Annotations
Checklists
Tools
Third-Party Tools
Find, Triage, Prioritize
Coding Errors and Security Vulnerabilities

Summary


Using Pull Requests with Work Items
Key Learning Outcomes
Microsoft Exam Expectation: Define branch strategy
Relationship
Reasons and more context
Audit trail for production changes
Adding References
"#" and the work item number
Enforcing References
Branch Policies

Summary

EXAM TIP

Branch Strategies

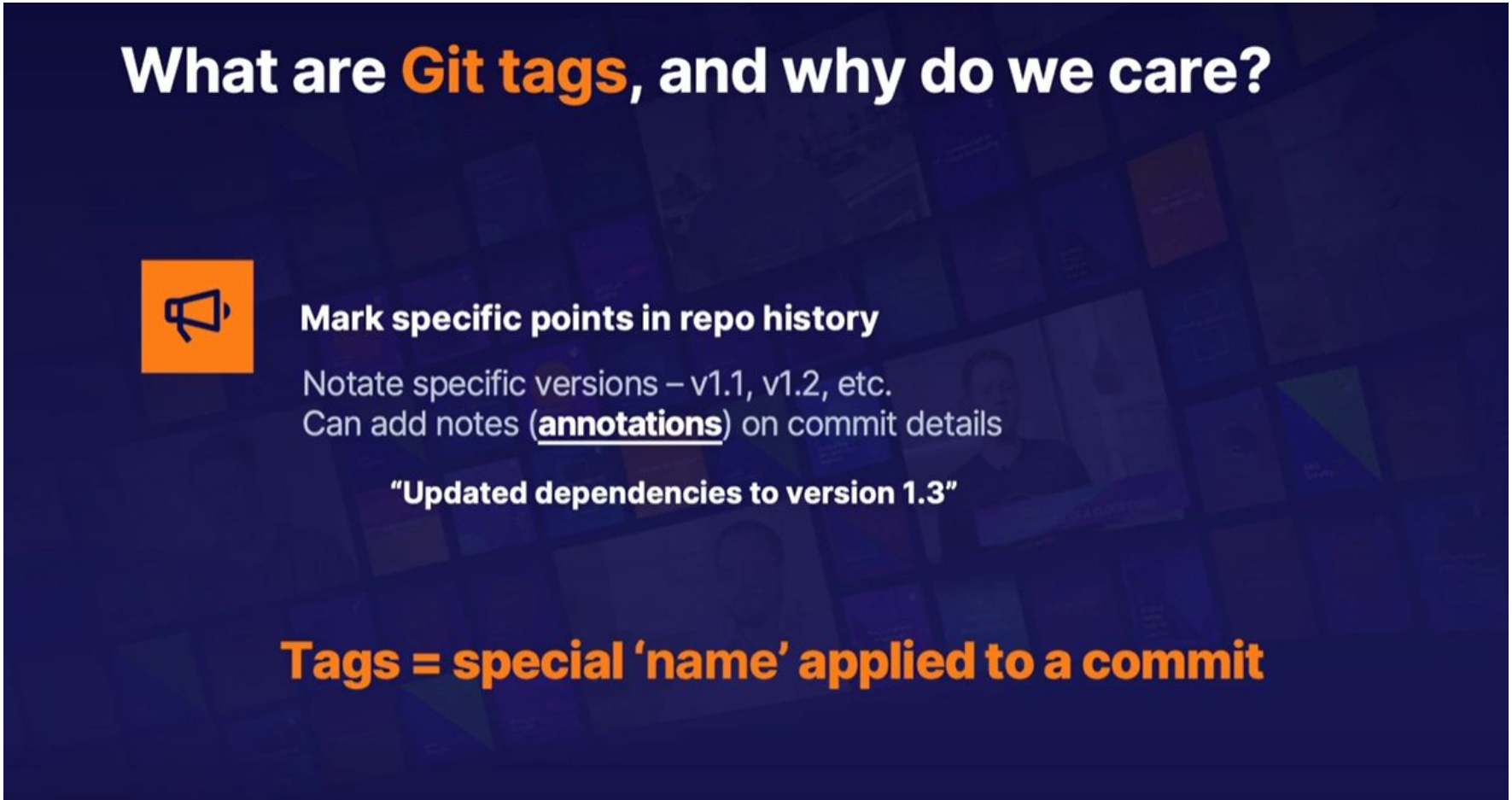
This is dependency-level knowledge for more advanced-level questions.

Focus on definitions and process.

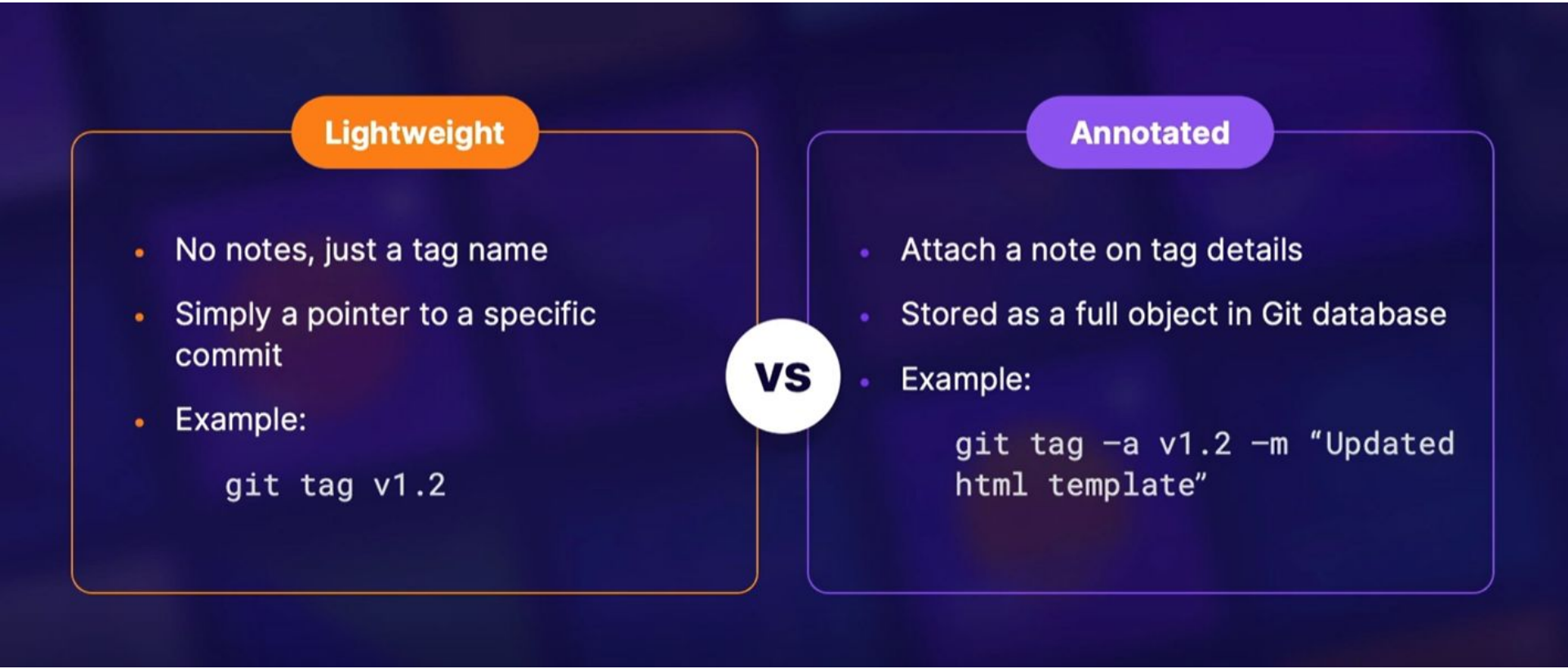
Configuring Repositories

Configuring Repositories

Using Git Tags to Organize Your Repository

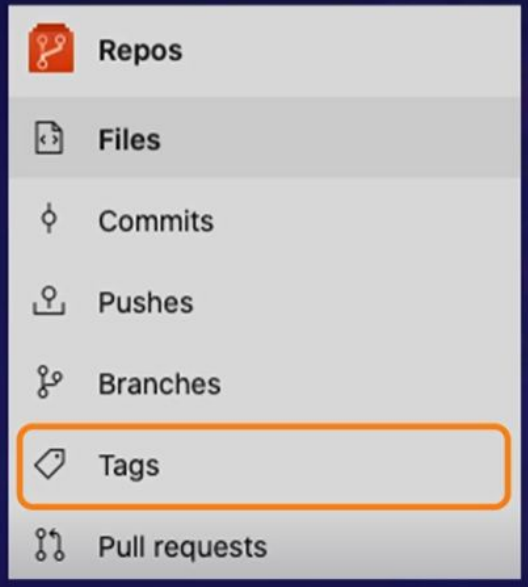


Using Git Tags to Organize Your Repository



Using Git Tags to Organize Your Repository

Azure Repos Perspective



Can view and create tags via web portal

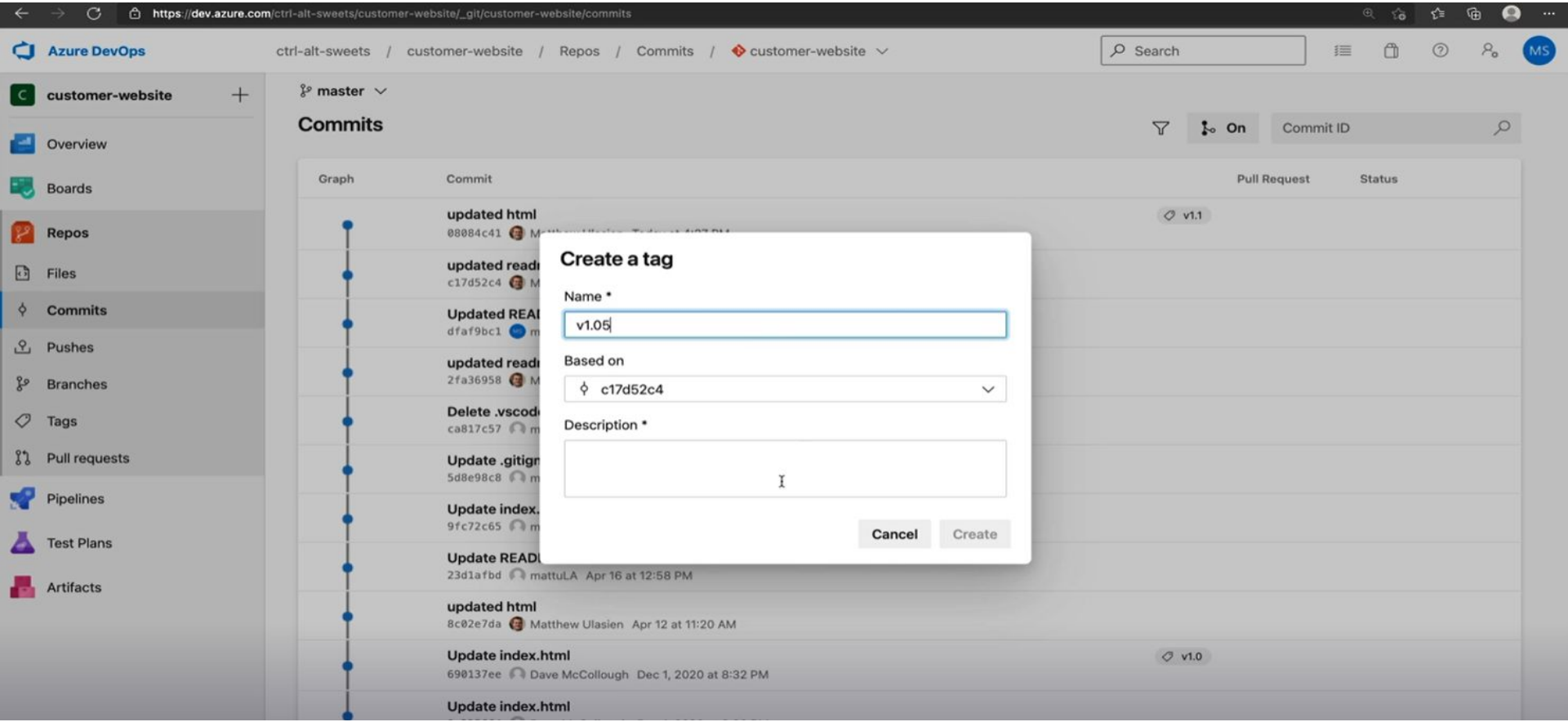
Web portal = create annotated tags only

Tags in remote repo require separate push

```
git push origin <tag>

git push origin v1.2
```

Using Git Tags to Organize Your Repository



Using Git Tags to Organize Your Repository

- **Git tags:** special notes on importance of specific commits
- **Tag types:** lightweight, annotated
- **Azure Repos:** annotated tags only
- **Working with remote repositories:** separate tag push required

Handling Large Repositories

Why do we care about source repository size?

1. **Challenge: Git repos are not bulk file storage**
   - ✓ Small footprint intended
   - ✓ Some file types should not be used in repos
   - ✓ However, some large binaries must be included

2. **Why is Git footprint size important?**
   - ✓ Cloning repo copies **full** history of all file versions
   - ✓ Frequently updated large files = serious performance issues!

Handling Large Repositories

Solution(s)

1. **Use best practices to avoid size 'bloat'.**
   - ✓ Know what **not** to include.
2. **For unavoidable large files, use Git LFS.**
3. **Clean up accumulated 'clutter' with git gc**

Handling Large Repositories

Git LFS

1. **Large file management built into Git**

- ✓ **Open-source extension (separately installed)**
- ✓ **Supported by popular remote repos (GitHub, Azure Repos)**
  - Tagged large files stored by remote repo, but as a placeholder in actual source code

Handling Large Repositories

1. **Install Git LFS for your OS.**

2. **Initiate LFS on your local environment.**

```
git lfs install
```

3. **Tag files to be added to LFS before committing them.**

   ✓ This results in a **.gitattributes** file.

```
git lfs track “*.psd”
```

4. **Commit as usual. The remote repo will store tagged files separately.**

   ✓ Tagged file will be marked in line with source code.

Handling Large Repositories

Do not commit...

Frequently Updated Binaries

With LFS: intended for infrequent updates.

Without LFS: leads to multiple copies in history.

Code Outputs

Outputs do not belong in source code.

Azure Boards can be used to better track work items.

Dependencies

Use package management solutions instead (discussed later).

Handling Large Repositories

1. **git gc = 'garbage collection'**
   - ✓ Loose files in unreferenced commits
   - ✓ Large binaries

```
git gc --(option)
```

2. **git gc + option compresses large files and/or removes loose files**

3. **git gc options**
   - ✓ `--aggressive` – Full optimization/compression (takes longer to run)
   - ✓ `--prune` – prune (delete) all loose objects `--prune=<date>` – prune (delete) loose objects older than <date>
   - ✓ `--no-prune` – do not prune any loose objects

Handling Large Repositories

- **Avoid large file 'bloat':** Large files in history drag down performance
- **Git LFS:** Open-source large file management | Remote repo markers
- **Best practices:** Keep unnecessary large files out of source code – alternative solutions
- **git gc:** Garbage collection – know flags to keep/prune loose files

Exploring Repository Permissions

Branch Permissions in Azure Repos

1 **Branch-level permission access**

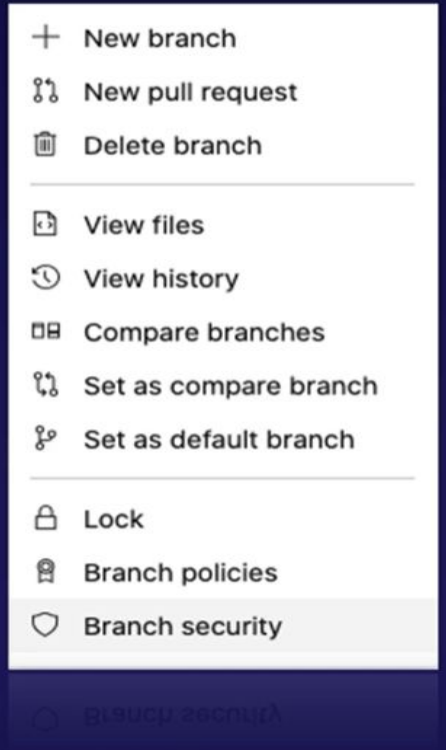
- ✓ Provides users different access to different branches
- ✓ By default: inherited from organization/project-level roles
- ✓ Provides access to main branch but not sub-branch, or vice versa

Exploring Repository Permissions

How Branch Permissions Work

1. Access the **Branches** menu from the web portal.
2. From the context menu, select **Branch security**.
3. Assign **permissions** to users.

- Permissions are inherited from the project/organization level by default.
- You can explicitly deny permissions granted by higher-level roles.



Exploring Repository Permissions

Branch Locks

Lock branch in ‘read-only’ mode

No new commits
Cannot change existing commit history

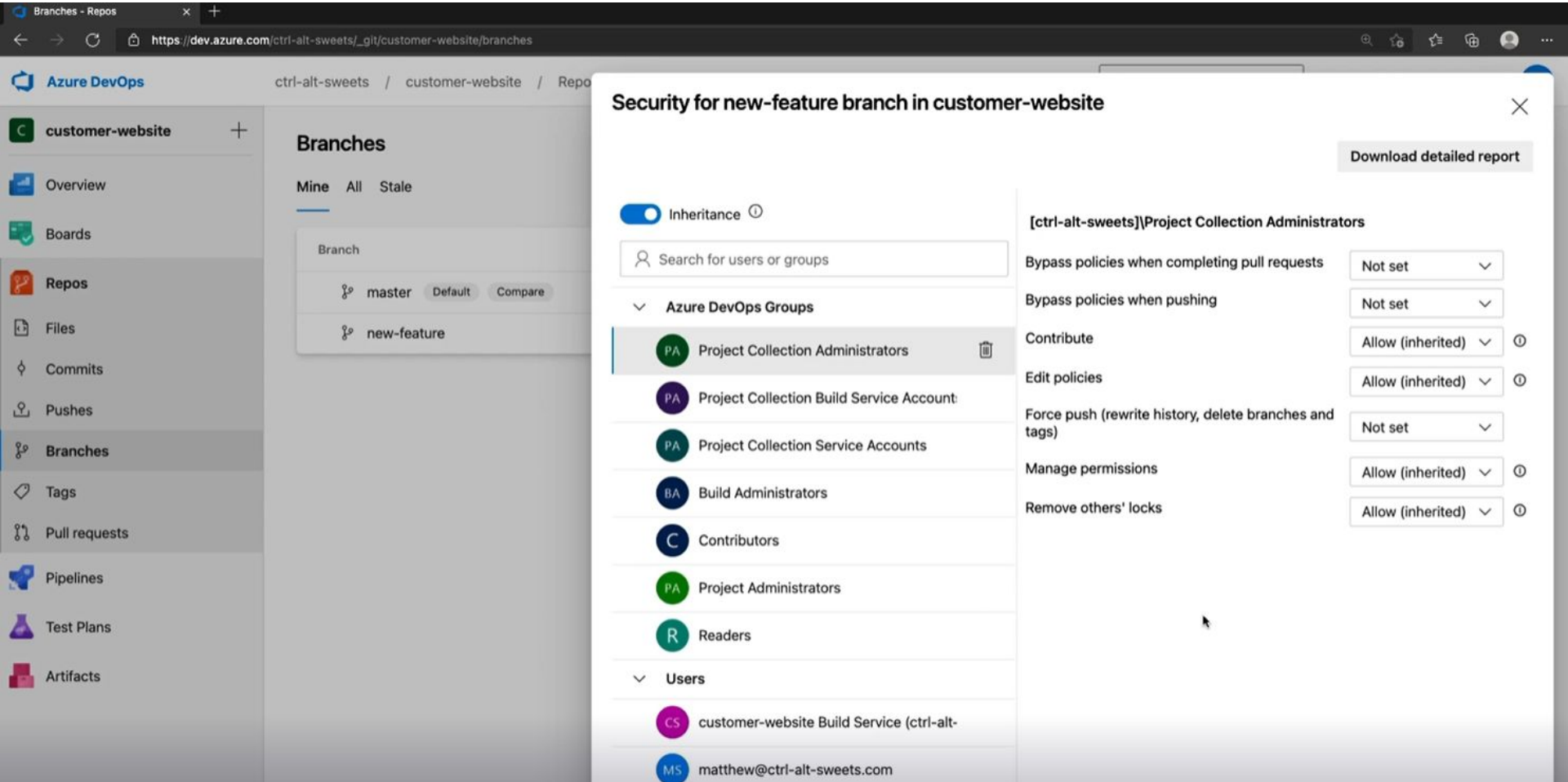
Purpose: Pair with pull requests

Prevent changes while being reviewed

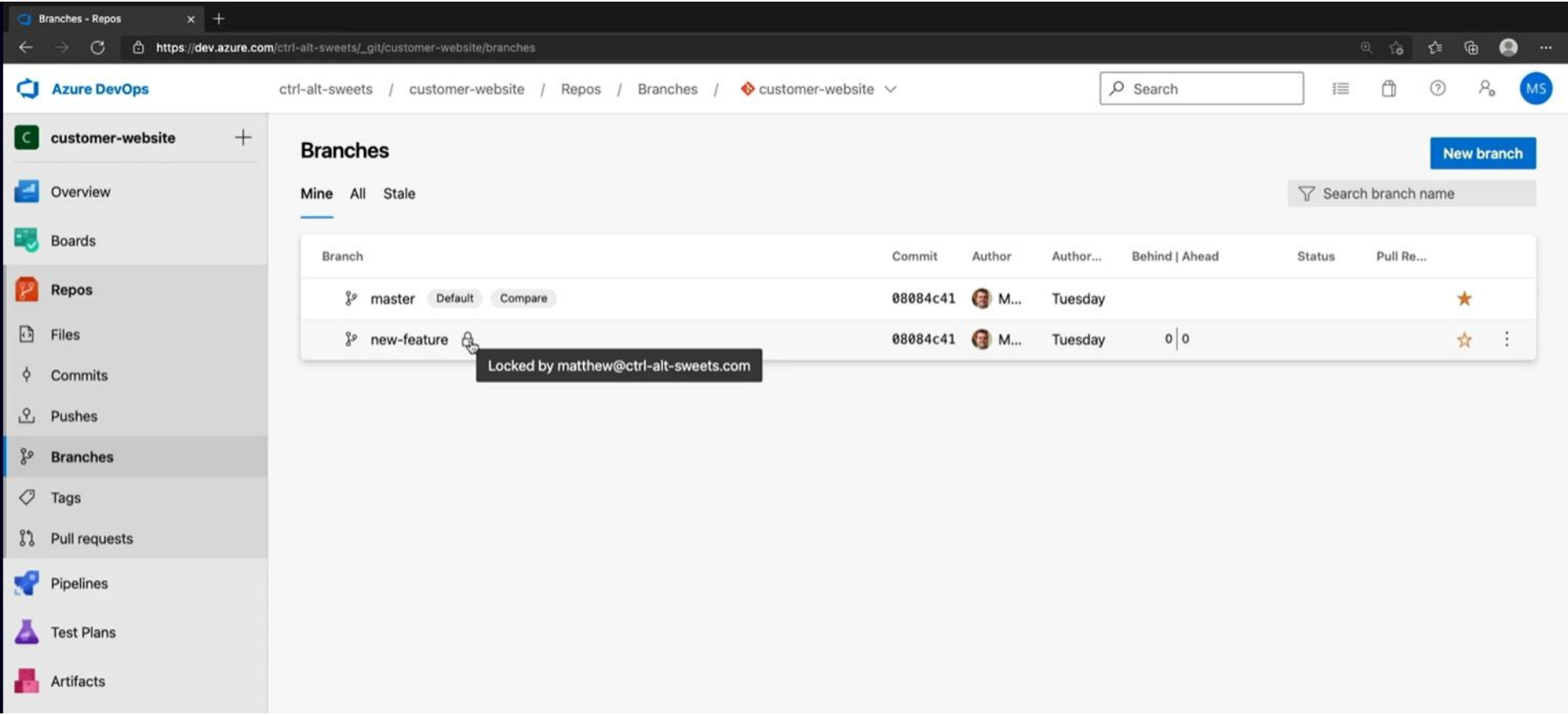
Removing branch locks

User who locked it
Project Administrator
Explicit ‘remove lock’ permission assigned

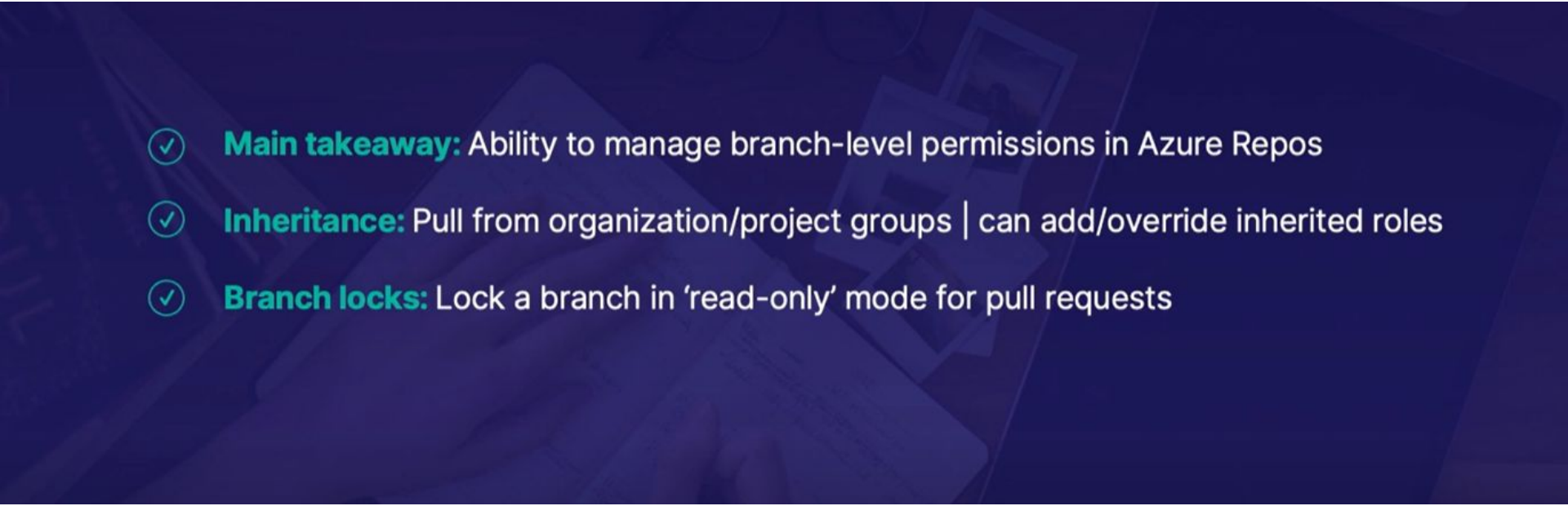
Exploring Repository Permissions



Exploring Repository Permissions



Exploring Repository Permissions



Removing Repository Data

Unwanted Files in Git

Scenario: You mistakenly commit and/or push files that you should not include.

- ✓ Very large files
- ✓ Sensitive data (passwords, SSH keys, secrets)

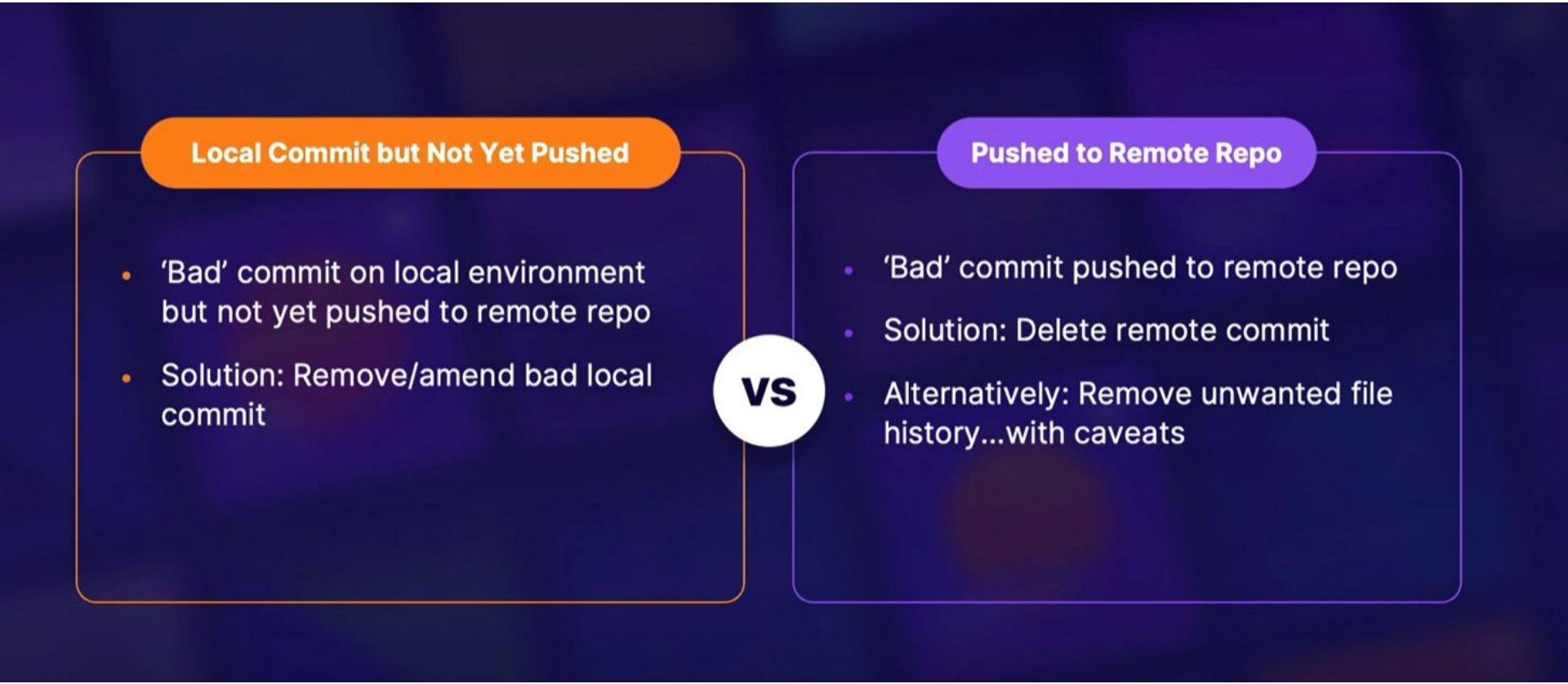
Problem: Deleted files from new commit still exist in the repo's history.

- ✓ Still discoverable by searching through history

Solution: Remove 'bad' commits before or after pushing to a remote repo.

- ✓ Can also remove files from history (with caveats)

Removing Repository Data



Removing Repository Data

Removing/Amending Local Commit Before Push

1. Delete unwanted file.
2. Remove file from Git tree/index.

```
git rm --cached <filename>
```

3. Delete or amend previous commit (depending on what other data changed).

Entirely Delete Commit

```
git reset HEAD^
```

Amend Commit

```
git commit --amend -m “comment”
```

Removing Repository Data

Remove Commit Already Pushed to Remote Repo

Best for recent pushes

1. Reset back to last 'good' commit.

```
git reset --hard #commitSHA
```

2. Force push to remove commits past the last good one.

```
git push --force
```

3. All branch commits past the reset one will be deleted.

Removing Repository Data

Remove Unwanted File from Past Commits' History

Short version: no single best solution

1. There are multiple tools to remove files from past history. Some are 'official', others are community-created.

   - ✓ **git filter-branch** : built-in method
   - ✓ **git filter-repo** : officially recommended community solution
   - ✓ **BFG Repo Cleaner**

2. Even after successful removal, you can still view file history in the Azure Repos web portal.

Recovering Repository Data

Mistakes happen!

Scenario: Changes made/deleted in error

- ✓ Pushed commits containing errors
- ✓ Mistakenly deleted a branch in Azure Repos
- ✓ Mistakenly deleted entire Azure repository

Need to know how to recover or 'rewind time' in the above scenarios

Recovering Repository Data

Scenario: Commits contain errors — need to 'roll back'

Same process as previous 'reset commits' lesson

1. **Reset back to last 'good' commit, and resume development from there.**

```
git reset --hard #commitSHA
```

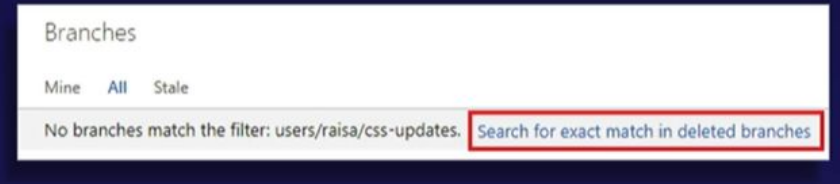
2. **Coordinate with development team members to merge changes to reverted code.**

- ✓ Known as 'rebase'

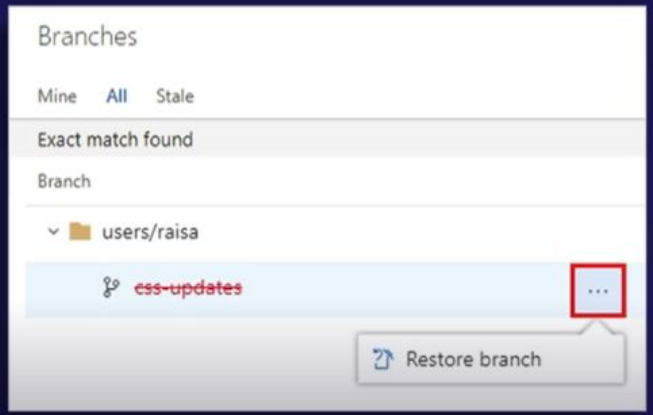
Recovering Repository Data

Scenario: Restore deleted Azure Repos branch

1. In Azure Repos, from the Branches view, search for the deleted branch.



2. From the deleted branches search, click Restore branch.

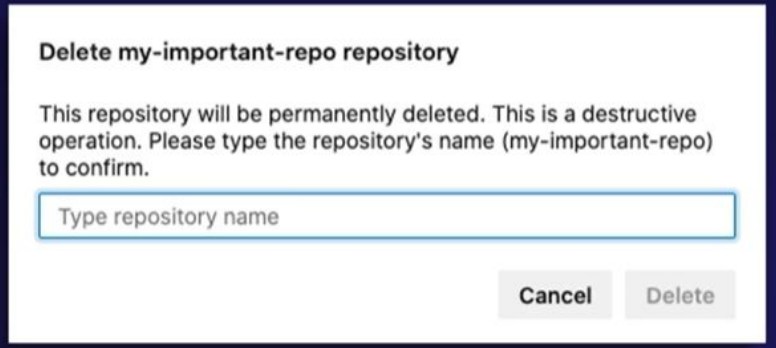


Recovering Repository Data

Scenario: Restore entire deleted Azure repository

1. Despite warning, repo is in a 'soft delete' state and can be restored.

2. Restore via an authenticated API call.




```
PATCH
https://dev.azure.com/{organization}/{project}/_apis/git/recycle
Bin/repositories/{repositoryId}?api-version=6.0-preview.1
```

Thank YOU