

# Azure Devops 2

Friday, January 2, 2026 1:08 PM

## 1. Where is the codebase stored?

- Your **source code** is stored in **Azure Repos** (Git repositories) or any connected Git provider (like GitHub, Bitbucket).
- When you trigger a pipeline, Azure DevOps **clones** the repository into the build agent (either Microsoft-hosted or self-hosted).

## 2. How does Azure Portal run it?

- **Microsoft-hosted agent:**

Azure spins up a temporary **virtual machine** in the cloud for your pipeline job.

Steps:

1. VM starts → pipeline agent installed.
2. Your repo is cloned into that VM.
3. Build/test/deploy tasks run.
4. VM is destroyed after the job finishes.

- **Self-hosted agent:**

You install an agent on your own machine/server.

Steps:

1. Pipeline triggers → your machine picks up the job.
2. Code is either already present or cloned locally.
3. Tasks run using your hardware.
4. No time limit (unlimited minutes).

These are VMs provided by Azure DevOps that spin up automatically when you run a pipeline.

so suppose it triggered a pipeline, and VMs got triggered and build the job - now my server is live, so now would the VM be destroyed? if destroyed, then won't my server be down again?

Excellent question! Here's the key point:

When you use **Microsoft-hosted agents**, the VM that runs your pipeline is **ephemeral**—it exists only for the duration of the job. After the pipeline finishes, the VM is **destroyed**.

But this does **NOT** affect your actual server or application because:

- The VM is **only for building and testing your code**, not for hosting your live application.
- After the build, the pipeline typically **deploys your code to your real server or cloud service** (e.g., Azure App Service, Kubernetes, VM, etc.).
- Your production server remains live because the deployment step pushes the built artifacts to it.

So:

- VM spins up → builds your code → deploys to your server → VM is destroyed.
- Your server stays up because it's independent of the build VM.

Do you want me to **illustrate this with a simple diagram** showing the flow:  
**Pipeline Trigger → Temporary VM → Build → Deploy → VM Destroyed → Server Live?**

Great question! Let me explain **how the VM pushes build artifacts to your production server** in a clear, step-by-step way:

## What happens after the VM builds your code?

The VM is temporary, but before it is destroyed, it **deploys the build artifacts** (your compiled code, packages, etc.) to your production environment. This is done using **deployment tasks** in your pipeline.

## How does the deployment work?

1. **Build Phase (on VM)**
  - The VM compiles your code and creates build artifacts (e.g., .zip, jar, Docker image).
  - These artifacts are stored temporarily on the VM.
2. **Publish Artifacts**
  - The pipeline uses tasks like PublishBuildArtifacts or PublishPipelineArtifact to upload artifacts to **Azure DevOps Artifacts** or a storage location (Azure Blob, etc.).
3. **Deploy Phase**
  - The pipeline then runs deployment tasks to push these artifacts to your **production server or cloud service**.
  - Common deployment methods:
    - **Azure Web App Deploy** (for web apps)
    - **Azure VM Copy Files + SSH** (for VMs)
    - **Kubernetes Deploy**
    - **FTP/SFTP** (for on-prem servers)
    - **Docker Push** (for container registries)
4. **VM Destroyed**
  - After deployment is complete, the VM is deleted.
  - Your production server now has the updated code and stays live.

## Key Idea

The VM is **never your production server**. It's just a temporary build machine. Your actual server (or cloud service) is separate and remains live because the pipeline **deploys the code to it before the VM is destroyed**.

Do you want me to **draw a simple diagram showing this flow**:  
**Pipeline Trigger → Temporary VM → Build → Push Artifacts → Deploy to Server → VM Destroyed → Server Live?**

Or should I **write a sample YAML pipeline that includes build + deploy steps**?

The pipeline then runs deployment tasks to push these artifacts to your production server or cloud service.

Ok, Now i have 2 questions:

1. Is the pipeline running in the temporary VM?
2. So, production server is already live - but now the pipeline will push the newer changed artifacts to the production server - right? to update the server....How is it doing so - pls lucidly

Great questions! Let's break them down clearly:

## 1. Is the pipeline running in the temporary VM?

Yes.

When you use **Microsoft-hosted agents**, the entire pipeline (build, test, deploy steps) runs inside that temporary VM.

- The VM is created when the pipeline starts.
- The pipeline agent is installed on that VM.
- All tasks in your pipeline execute there.
- After the job finishes, the VM is destroyed.

## 2. Production server is already live, but pipeline pushes newer artifacts to update it — how does that happen?

Here's the **lucid flow**:

- Your production server (or cloud service) is **separate** from the build VM and is already running the old version of your app.
- After the VM builds the new version, the pipeline uses **deployment tasks** to **push the new artifacts to the production server**.

### How does the VM push artifacts?

- The pipeline includes a **deployment stage** (after build).
- Common methods:
  - **Azure Web App Deploy** → Deploys directly to Azure App Service.
  - **Copy Files Over SSH** → Copies files to your on-prem or cloud VM via SSH.
  - **FTP/SFTP** → Uploads files to a server.
  - **Azure CLI / PowerShell** → Executes commands to update resources.
  - **Docker Push + Kubernetes Apply** → Updates containers in a cluster.
- Once deployment is complete, your production server now runs the **new version** of the app.
- The temporary VM is then destroyed, but your production server stays live because it has the updated code.

So the VM is **just a builder and deployer**, not your live server.

Your server stays up because the pipeline **copies the new build to it before the VM disappears**.

Would you like me to:

- **Draw a simple diagram showing this entire flow visually** (Trigger → VM → Build → Deploy → VM Destroyed → Server Live)?
- Or **write a sample Azure DevOps YAML pipeline with build + deploy steps** for clarity?

## **so what is our production server here? kubernetes service?**

Great follow-up! Yes, your **production server** can absolutely be a **Kubernetes cluster** (or any other hosting environment). Let me explain:

### **What is the production server in this context?**

- It's the environment where your application runs for end-users.
- This could be:
  - **Azure Kubernetes Service (AKS)** → for containerized apps.
  - **Azure App Service** → for web apps.
  - **Azure Virtual Machines** → for traditional deployments.
  - **On-prem servers** → via SSH/FTP.
  - **Other cloud services** → AWS, GCP, etc.

### **How does the pipeline deploy to Kubernetes?**

If your production environment is **Kubernetes**, the pipeline will:

1. Build your application (e.g., create Docker images).
2. Push the image to a container registry (Azure Container Registry or Docker Hub).
3. Use kubectl or Helm tasks in the pipeline to **update the Kubernetes deployment** with the new image.

So the flow is:

- **Temporary VM builds** → **pushes image** → **updates Kubernetes cluster** → **VM destroyed** → **Kubernetes stays live with new version**.

Do you want me to:

- Draw a diagram showing this Kubernetes deployment flow?
- Or write a sample Azure DevOps YAML pipeline for Kubernetes deployment?

How does the pipeline deploy to Kubernetes?

If your production environment is Kubernetes, the pipeline will:

1. Build your application (e.g., create Docker images).

Push the image to a container registry (Azure Container Registry or Docker Hub).

Now, my question is - suppose Azure Container Registry got the new image- but how would the kubernetes cluster which serves the live server would know about it?

You just said that the temporary VM updates the kubernetes cluster, but i really ain't got the idea how!

## How does Kubernetes know about the new image in Azure Container Registry (ACR)?

Kubernetes doesn't automatically detect new images in ACR. Instead, the pipeline explicitly tells Kubernetes to update its deployment using commands.

Here's the flow:

### Step 1: Build & Push Image

- The pipeline builds a Docker image of your app.
- Tags it (e.g., `myapp:v2`).
- Pushes it to ACR (or Docker Hub).

## Step 2: Update Kubernetes Deployment

- The pipeline uses **kubectl** or **Helm** commands to update the Kubernetes cluster.
- Example:

```
1 kubectl set image deployment/myapp myapp=acrname.azurecr.io/myapp:v2
```

This tells Kubernetes:

- For the deployment named **myapp**
- Replace the container image with the new one from ACR (v2).
- Kubernetes then:
  - Pulls the new image from ACR.
  - Performs a **rolling update** (gradually replacing old pods with new ones).
  - Keeps the service live during the update.

## Step 3: Cluster Stays Live

- Your Kubernetes cluster is your **production server**.
- The temporary Azure VM only runs the pipeline and sends the update command.
- After the VM is destroyed, Kubernetes continues running the new version.

## Key Idea

The pipeline doesn't just push the image to ACR; it also **executes deployment commands** to tell Kubernetes to use that new image.

Link mobility ,CSU, connection from test environment

The screenshot shows an Excel spreadsheet titled "CSDBS03APWCSL.CSUAT.COM". The table has columns: Application, Source Server, Source Server IP, Destination Server, Destination Server IP, and Port. The data includes various AKS Cluster API Servers and their corresponding internal and external endpoints.

Application	Source Server	Source Server IP	Destination Server	Destination Server IP	Port
SIT AKS Cluster API Server	CSDBS01APWCSL.CSUAT.COM	10.14.1.149	digital-pl-cs-onecs-ds-spok-26040d-68838584.hcp.uksouth.azuremk8s.io	51.11.171.202	443
SIT AKS Cluster API Server	CSDBS02APWCSL.CSUAT.COM	10.14.7.151	digital-pl-cs-onecs-ds-spok-26040d-68838584.hcp.uksouth.azuremk8s.io	51.11.171.202	443
SIT AKS Cluster API Server	CSDBS03APWCSL.CSUAT.COM	10.14.1.205	digital-pl-cs-onecs-ds-spok-26040d-68838584.hcp.uksouth.azuremk8s.io	51.11.171.202	443
SIT AKS Cluster API Server	CS-TFS-BUILD05.CSUAT.COM	10.14.2.38	digital-pl-cs-onecs-ds-spok-26040d-68838584.hcp.uksouth.azuremk8s.io	51.11.171.202	443
SIT AKS Cluster API Server	CS-TFS-LINUXBUILD01.CSUAT.COM	10.14.2.39	digital-pl-cs-onecs-ds-spok-26040d-68838584.hcp.uksouth.azuremk8s.io	51.11.171.202	443
SIT AKS Cluster API Server	CSLBS02APLCSL.CSUAT.COM	10.14.2.40	digital-pl-cs-onecs-ds-spok-26040d-68838584.hcp.uksouth.azuremk8s.io	51.11.171.202	443
NEW SIT AKS Cluster API Server	CSDBS01APWCSL.CSUAT.COM	10.14.1.149	cs-dp-sit-aks-aks-dns-3436a47c.66927db2-8cda-4639-b7f2-dad44fbff9fb.privatelink.uksouth.azuremk8s.io	10.19.2.4	443
NEW SIT AKS Cluster API Server	CSDBS02APWCSL.CSUAT.COM	10.14.7.151	cs-dp-sit-aks-aks-dns-3436a47c.66927db2-8cda-4639-b7f2-dad44fbff9fb.privatelink.uksouth.azuremk8s.io	10.19.2.4	443
NEW SIT AKS Cluster API Server	CSDBS03APWCSL.CSUAT.COM	10.14.1.205	cs-dp-sit-aks-aks-dns-3436a47c.66927db2-8cda-4639-b7f2-dad44fbff9fb.privatelink.uksouth.azuremk8s.io	10.19.2.4	443
NEW SIT AKS Cluster API Server	CS-TFS-BUILD05.CSUAT.COM	10.14.2.38	cs-dp-sit-aks-aks-dns-3436a47c.66927db2-8cda-4639-b7f2-dad44fbff9fb.privatelink.uksouth.azuremk8s.io	10.19.2.4	443
NEW SIT AKS Cluster API Server	CS-TFS-LINUXBUILD01.CSUAT.COM	10.14.2.39	cs-dp-sit-aks-aks-dns-3436a47c.66927db2-8cda-4639-b7f2-dad44fbff9fb.privatelink.uksouth.azuremk8s.io	10.19.2.4	443
NEW SIT AKS Cluster API Server	CSLBS02APLCSL.CSUAT.COM	10.14.2.40	cs-dp-sit-aks-aks-dns-3436a47c.66927db2-8cda-4639-b7f2-dad44fbff9fb.privatelink.uksouth.azuremk8s.io	10.19.2.4	443
CSU AKS Cluster API Server	CSDBS01APWCSL.CSUAT.COM	10.14.1.149	cs-dp-csu-aks-aks-dns-754nizea.d8fe6091-3048-43b8-91e8-1a18799b7883.privatelink.uksouth.azuremk8s.io	10.19.20.4	443
CSU AKS Cluster API Server	CSDBS02APWCSL.CSUAT.COM	10.14.7.151	cs-dp-csu-aks-aks-dns-754nizea.d8fe6091-3048-43b8-91e8-1a18799b7883.privatelink.uksouth.azuremk8s.io	10.19.20.4	443
CSU AKS Cluster API Server	CSDBS03APWCSL.CSUAT.COM	10.14.1.205	cs-dp-csu-aks-aks-dns-754nizea.d8fe6091-3048-43b8-91e8-1a18799b7883.privatelink.uksouth.azuremk8s.io	10.19.20.4	443
CSU AKS Cluster API Server	CS-TFS-BUILD05.CSUAT.COM	10.14.2.38	cs-dp-csu-aks-aks-dns-754nizea.d8fe6091-3048-43b8-91e8-1a18799b7883.privatelink.uksouth.azuremk8s.io	10.19.20.4	443
CSU AKS Cluster API Server	CS-TFS-LINUXBUILD01.CSUAT.COM	10.14.2.39	cs-dp-csu-aks-aks-dns-754nizea.d8fe6091-3048-43b8-91e8-1a18799b7883.privatelink.uksouth.azuremk8s.io	10.19.20.4	443
CSU AKS Cluster API Server	CSLBS02APLCSL.CSUAT.COM	10.14.2.40	cs-dp-csu-aks-aks-dns-754nizea.d8fe6091-3048-43b8-91e8-1a18799b7883.privatelink.uksouth.azuremk8s.io	10.19.20.4	443

Great question! Based on the image, here's what I can infer:

- The **Source Servers** listed (e.g., CSDBS01APWCSL.CSUAT.COM, CS-TFS-BLDB01.CSUAT.COM) look like **internal servers in a CSUAT environment**, which could be either **physical machines or virtual machines** hosted on-premises or in a private cloud.
- The **Destination Servers** (e.g., digital-pl-cs-onecs-ds-spoke-...hcp.eusouth.azuremk8s.io, privatelink.uksouth.azuremk8s.io) are clearly **Azure Kubernetes Service (AKS)** endpoints. These are not VMs; they are part of Azure's managed Kubernetes infrastructure.
- The IP addresses in the **Source Server IP** column (e.g., 10.x.x.x) are private IPs, which strongly suggests these source servers are in a **private network**, likely virtual machines in an on-prem or private cloud environment.
- The **Destination Server IPs** include both public IPs (51.x.x.x) and private IPs (10.x.x.x), indicating a mix of public-facing AKS endpoints and private link connections.

## Answer

- The **source servers** are most likely **virtual machines** (or possibly physical servers) in your internal environment.
- The **destination servers** are **Azure AKS cluster endpoints**, which are not VMs but managed Kubernetes services.

What agents talk to what server?

What is source server doing with destination server?

What are we actually doing.

What is csu?

What is link mobility>

How many servers? How many build servers?

Each row represents a firewall rule for a specific **Source Server → Destination Server** connection -> what is firewall rule doing

## What is a Build Server?

- A **build server** is a machine (physical or virtual) that runs **build jobs** for your application.
- It compiles source code, runs tests, packages artifacts, and prepares them for deployment.
- In Azure DevOps, this role is typically performed by **Build Agents** (either Microsoft-hosted or self-hosted).

## Does it build pipelines?

- **No, the build server does not create pipelines.**
- Pipelines are defined in **Azure DevOps** (YAML or Classic UI) and stored in your project.
- The build server (or agent) **executes the pipeline steps** when triggered.
- So, the pipeline logic lives in Azure DevOps, but the actual work (compiling, testing, deploying) happens on the build server.

## In Your Spreadsheet

- The server named CS-TFS-BLDB01.CSUAT.COM suggests:
  - **TFS** = Team Foundation Server (older version of Azure DevOps).
  - **BLDB** = Build.
  - This is likely a **self-hosted build agent/server** used to run pipelines for your CSUAT environment.

so the build server CS-TFS-BLDB01.CSUAT.COM is nothing but an agent?

Yes, you've got it!

The server **CS-TFS-BLDB01.CSUAT.COM** is most likely functioning as a **self-hosted build agent** in your Azure DevOps (or TFS) environment. Here's why:

## Why is it an Agent?

- In Azure DevOps/TFS terminology:
  - **Pipelines** define the steps (build, test, deploy).
  - **Agents** execute those steps on a machine.
- A **self-hosted agent** is installed on a server you manage (on-prem or private cloud).
- The name BLDB strongly suggests **Build**, and TFS indicates Team Foundation Server (older Azure DevOps).
- This server appears multiple times in the firewall request sheet because it needs connectivity to various Azure AKS clusters to deploy containers or interact with Kubernetes.

## Key Role

- It does **not create pipelines**.
- It **runs pipeline jobs** (builds, tests, deployments) when triggered from Azure DevOps/TFS.

**Summary in one line:**

**CS-TFS-BLDB01.CSUAT.COM is a self-hosted build agent server used to execute Azure DevOps/TFS pipelines.**

Do you want me to: ✓ **Draw a diagram showing how this build agent fits into your Azure DevOps pipeline architecture?**

✓ **Or write a short documentation snippet explaining the role of this server as an agent for your team?**

This server appears multiple times in the firewall request sheet because it needs connectivity to various Azure AKS clusters to deploy containers or interact with Kubernetes.

Why does the build agent/server needs multiple firewall request? what's the purpose?

Great question! Here's the detailed reasoning behind **why the build agent/server needs multiple firewall requests**:

## Purpose of Multiple Firewall Requests

### 1. Multiple AKS Clusters

- Your spreadsheet shows different destination servers like:
  - digital-ci-aks-cs-oncs-ds-spoke-...azmk8s.io
  - privatelink.uksouth.azmk8s.io
- Each AKS cluster has its own **endpoint** and sometimes its own **private IP**.
- The build agent needs to connect to **each cluster** to deploy workloads or run Kubernetes commands.

### 2. Different Environments

- The applications listed include **SIT, NEW SIT, CSU**, etc.
- Each environment likely has its own AKS cluster or namespace.
- The build agent must reach all these environments during pipeline execution.

### 3. Private Link Connectivity

- Some endpoints use **Azure Private Link** (e.g., privatelink.uksouth.azmk8s.io).
- Private Link requires explicit firewall rules for each private endpoint IP.

### 4. Security Principle

- Firewalls are configured with **specific rules per destination** for security.
- Instead of opening "any" traffic, you define **granular rules** for each AKS cluster and port (443 for HTTPS).

## Why is This Needed for Azure DevOps?

- When a pipeline runs on the build agent, it may:
  - **Push container images** to AKS.
  - **Run kubectl commands** to deploy manifests.
  - **Interact with cluster APIs** for health checks or scaling.
- All these actions require **network connectivity** from the agent to the AKS API server.
- If the firewall blocks this, deployments fail.

### In short:

The build agent needs multiple firewall rules because it interacts with **multiple AKS clusters and endpoints across different environments**, and each endpoint requires a separate rule for secure connectivity.

Do you want me to: ✓ **Create a diagram showing the build agent and its connections to multiple AKS clusters with firewall rules?**

✓ **Or write a short documentation snippet explaining this for your team's firewall approval process?**

so we saw 2 servers, one is build server. other is  
CSDBS01APWCSL.CSUAT.COM - what is this source server doing?

Great observation! Let's analyze what **CSDBS01APWCSL.CSUAT.COM** is likely doing based on the context:

## Clues from the Name

- CSDBS01APWCSL.CSUAT.COM:
  - **CSDBS** → Could indicate **Central Services Database Server** or **Customer Service Database Server**.
  - **AP** → Often stands for **Application**.
  - **WCSL** → Might be an internal code for a specific system or environment.
  - **CSUAT.COM** → Clearly part of the **UAT (User Acceptance Testing)** environment.

So, this looks like an **application/database server** in the CSUAT environment.

## Clues from the Spreadsheet

- This server appears under **Application: SIT AKS Cluster API Server**.
- It has a **Source IP** in the private range (10.x.x.x).
- It needs connectivity to **Azure AKS clusters** (destination servers are AKS endpoints).
- Port is **443 (HTTPS)**, which suggests API communication rather than database replication.

## Likely Role

- This server is probably:
  - **Hosting APIs or services** that need to interact with AKS clusters.
  - Could be part of **backend services** that deploy or manage workloads in AKS.
  - Might also be a **jump host or integration point** for UAT applications to communicate with containers running in AKS.

### In short:

CSDBS01APWCSL.CSUAT.COM is likely an **application or API server** in the CSUAT environment that needs firewall access to AKS clusters for:

- **API calls** to services running in AKS.
- Possibly **deployment triggers** or **integration tasks**.

Do you want me to: ✓ **Create a role-based summary table for both servers (Build Agent vs Application Server)?**

✓ **Or draw an architecture diagram showing how these two servers interact with AKS clusters and Azure DevOps pipelines?**

What is source server doing with destination server?

Great question! Based on the spreadsheet and context, here's what the **Source Server** is doing with the **Destination Server**:

## Source Server → Destination Server Interaction

- **Source Server:** Internal servers like CSDBS01APWCSL.CSUAT.COM or CS-TFS-BLDB01.CSUAT.COM.
- **Destination Server:** Azure AKS cluster endpoints (\*.azmk8s.io), often via **Private Link**.

### Purpose of Communication

1. **For Build Server (CS-TFS-BLDB01)**
  - Executes Azure DevOps/TFS pipelines.
  - Needs to:
    - **Deploy containers** to AKS clusters.
    - **Run kubectl commands** to apply manifests.
    - **Interact with Kubernetes API** for scaling, health checks, etc.
  - This requires HTTPS (port 443) access to AKS API endpoints.
2. **For Application Server (CSDBS01APWCSL)**
  - Likely hosts **APIs or backend services** in the CSUAT environment.
  - Needs to:
    - **Call services running inside AKS clusters** (e.g., microservices).
    - Possibly **trigger deployments** or **fetch data** from AKS-hosted apps.
  - Again, this happens over HTTPS (port 443).

## Why Firewall Rules Are Needed

- Firewalls block traffic by default.
- Each AKS cluster has a unique endpoint (public or private).
- To allow secure communication, **specific rules per destination** are created:
  - Source IP → Destination AKS endpoint → Port 443.

**In short:**

The source servers are making **API calls or deployment actions** to AKS clusters.

- **Build server:** Deploys workloads and interacts with Kubernetes API.
- **Application server:** Communicates with services hosted in AKS (integration or runtime calls).

Do you want me to: ✓ **Draw a diagram showing these interactions (Source Servers → AKS clusters → Azure DevOps)?**

✓ Or **write a short documentation snippet explaining the purpose of these firewall rules for your team?**



Labi Egbelemy 15:08

Data Staging overnight data refresh status 05/01/2026

SIT - Run completed.

<http://datastagingdashboard-sit.charles-stanley.co.uk/app/>

UAT (CSU) - Account - RunTableBuildFailed

<http://datastagingdashboardcsu.charles-stanley.co.uk/app/>

UAT (CSL) - Running 80%.

InstrumentFundDetails - In progress

<http://datastagingdashboard-uat.charles-stanley.co.uk/app/>



Reply in thread

## The Relationship Between Build Servers, Destination Servers, and SIT/UAT Environments

### 1. Build Server (CS-TFS-BLDB01)

- This is a **self-hosted Azure DevOps/TFS agent**.
- Its job: **Run pipelines** that build and deploy applications.
- When a pipeline runs, the build server:
  - Builds the code.
  - Creates container images.
  - Deploys those images into **AKS clusters**.

### 2. Destination Servers (AKS Clusters)

- These are **Azure Kubernetes Service endpoints** (e.g., \*.azmk8s.io).
- Each AKS cluster hosts applications for a specific **environment**:
  - **SIT cluster** → For System Integration Testing.
  - **UAT-CSU cluster** → For UAT testing for CSU business unit.
  - **UAT-CSL cluster** → For UAT testing for CSL business unit.
- The build server needs **firewall access** to these AKS API endpoints to:
  - Apply Kubernetes manifests.
  - Push container images.

- Trigger deployments.

### 3. SIT / UAT-CSU / UAT-CSL Environments

- These environments **live inside AKS clusters**.
- After deployment, testers use these environments to validate functionality.
- The **data staging process** (shown in your Teams message) refreshes **test data** inside these environments so that applications have up-to-date data for testing.

## How They Work Together

- Pipeline runs on Build Server → Connects to AKS cluster → Deploys app into SIT/UAT environment → Data staging refresh updates test data → Testers validate the app.

**In short:**

- **Build server** = Executes pipelines.
- **Destination servers (AKS)** = Host SIT/UAT environments.
- **SIT/UAT environments** = Where apps run after deployment.
- **Data staging** = Prepares data for those environments.

Great observation! Let's confirm this based on the **Excel image you shared earlier**:

## From the Excel Image

- The **Application** column listed:
  - **SIT AKS Cluster API Server**
  - **NEW SIT AKS Cluster API Server**
  - **CSU AKS Cluster API Server**
- These clearly indicate **AKS clusters for SIT and CSU**.
- There was no explicit mention of CSL in the Excel image, but CSL appeared in the **Teams message** about data staging.

## So, What Does This Mean?

- From the Excel image:
  - SIT AKS cluster**
  - CSU AKS cluster**
- CSL was mentioned in the **data refresh status message**, which suggests CSL is another environment, but **it may or may not have its own AKS cluster** (it could share one or have a separate namespace).

## Most Likely Scenario

- **SIT** → Has its own AKS cluster (confirmed in Excel).
- **CSU** → Has its own AKS cluster (confirmed in Excel).
- **CSL** → Probably has its own AKS cluster or a namespace in an existing cluster (inferred from Teams message).

**Answer:**

Yes, there are at least **two AKS clusters (SIT and CSU)** confirmed from the Excel image, and **CSL is very likely a third AKS cluster or namespace** based on the data refresh message.

Do you want me to: ✓ **Draw a diagram showing these three environments and their AKS clusters, and how the build server connects to them?**

✓ **Or write a short explanation for documentation about why multiple AKS clusters exist for**

## SIT/UAT environments?

Great question! Here's the clear explanation:

### Do SIT and CSU contain different apps or the same app?

- SIT (System Integration Testing) and CSU (a UAT environment) are **different environments for the same application or system**, not completely different apps.
- Why?
  - In enterprise setups, the same application goes through multiple stages:
    - **Development → SIT → UAT → Production**
  - SIT is used for **integration testing** (technical validation).
  - UAT (CSU) is used for **business validation** (user acceptance testing).
- The application deployed in SIT and CSU is **functionally the same**, but:
  - It may have **different configurations** (test data, endpoints).
  - It may run on **different AKS clusters** for isolation.

### Why separate environments?

- To avoid conflicts:
  - SIT testers focus on technical integration.
  - UAT testers focus on business workflows.
- Each environment needs its own AKS cluster or namespace to keep deployments and data separate.

**In short:**

SIT and CSU host **the same application**, but in **different environments for different testing purposes**. CSL is another UAT environment for a different business unit, but still likely the same core app.

Do you want me to: ✓ **Draw a diagram showing the same app deployed across SIT, CSU, and CSL AKS clusters?**

✓ **Or write a short documentation snippet explaining why these environments exist and how they relate?**