Azure Devops 2

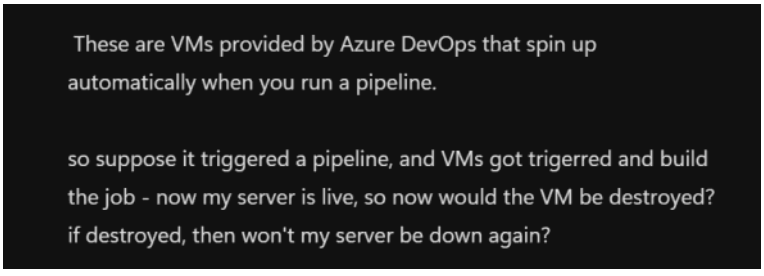
Friday, January 2, 2026 1:08 PM

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).



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 Contain 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:

```
Shell

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.

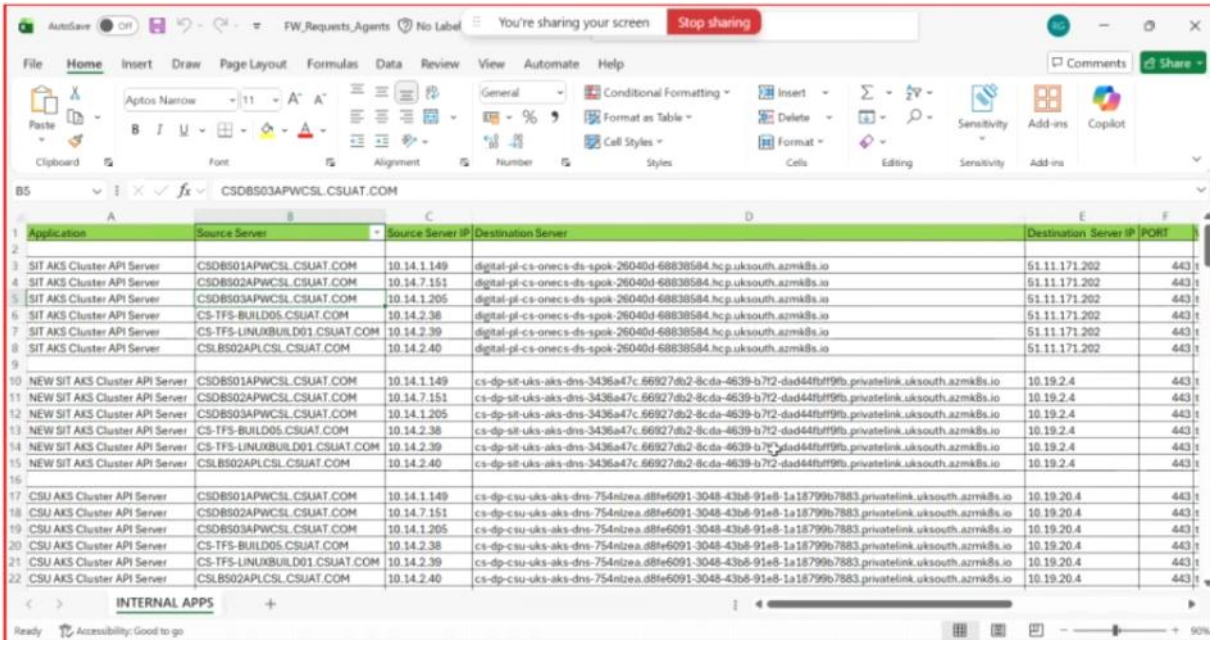
☑ Step 3: Cluster Stays Live

- 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 environmrnt



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.azmk8s.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.azmk8s.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.azmk8s.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.azmk8s.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.azmk8s.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.azmk8s.io	51.11.171.202	443
NEW SIT AKS Cluster API Server	CSDBS01APWCSL.CSUAT.COM	10.14.1.149	cs-dp-sit-uks-aks-dns-3436a47c.66927db2-8cda-4639-b7f2-dad44fbff9fb.privatelink.uksouth.azmk8s.io	10.19.2.4	443
NEW SIT AKS Cluster API Server	CSDBS02APWCSL.CSUAT.COM	10.14.7.151	cs-dp-sit-uks-aks-dns-3436a47c.66927db2-8cda-4639-b7f2-dad44fbff9fb.privatelink.uksouth.azmk8s.io	10.19.2.4	443
NEW SIT AKS Cluster API Server	CSDBS03APWCSL.CSUAT.COM	10.14.1.205	cs-dp-sit-uks-aks-dns-3436a47c.66927db2-8cda-4639-b7f2-dad44fbff9fb.privatelink.uksouth.azmk8s.io	10.19.2.4	443
NEW SIT AKS Cluster API Server	CS-TFS-BUILD05.CSUAT.COM	10.14.2.38	cs-dp-sit-uks-aks-dns-3436a47c.66927db2-8cda-4639-b7f2-dad44fbff9fb.privatelink.uksouth.azmk8s.io	10.19.2.4	443
NEW SIT AKS Cluster API Server	CS-TFS-LINUXBUILD01.CSUAT.COM	10.14.2.39	cs-dp-sit-uks-aks-dns-3436a47c.66927db2-8cda-4639-b7f2-dad44fbff9fb.privatelink.uksouth.azmk8s.io	10.19.2.4	443
NEW SIT AKS Cluster API Server	CSLBS02APLCSL.CSUAT.COM	10.14.2.40	cs-dp-sit-uks-aks-dns-3436a47c.66927db2-8cda-4639-b7f2-dad44fbff9fb.privatelink.uksouth.azmk8s.io	10.19.2.4	443
CSU AKS Cluster API Server	CSDBS01APWCSL.CSUAT.COM	10.14.1.149	cs-dp-csu-uks-aks-dns-754nlzea.d8fe6091-3048-43b8-91e8-1a18799b7883.privatelink.uksouth.azmk8s.io	10.19.20.4	443
CSU AKS Cluster API Server	CSDBS02APWCSL.CSUAT.COM	10.14.7.151	cs-dp-csu-uks-aks-dns-754nlzea.d8fe6091-3048-43b8-91e8-1a18799b7883.privatelink.uksouth.azmk8s.io	10.19.20.4	443
CSU AKS Cluster API Server	CSDBS03APWCSL.CSUAT.COM	10.14.1.205	cs-dp-csu-uks-aks-dns-754nlzea.d8fe6091-3048-43b8-91e8-1a18799b7883.privatelink.uksouth.azmk8s.io	10.19.20.4	443
CSU AKS Cluster API Server	CS-TFS-BUILD05.CSUAT.COM	10.14.2.38	cs-dp-csu-uks-aks-dns-754nlzea.d8fe6091-3048-43b8-91e8-1a18799b7883.privatelink.uksouth.azmk8s.io	10.19.20.4	443
CSU AKS Cluster API Server	CS-TFS-LINUXBUILD01.CSUAT.COM	10.14.2.39	cs-dp-csu-uks-aks-dns-754nlzea.d8fe6091-3048-43b8-91e8-1a18799b7883.privatelink.uksouth.azmk8s.io	10.19.20.4	443
CSU AKS Cluster API Server	CSLBS02APLCSL.CSUAT.COM	10.14.2.40	cs-dp-csu-uks-aks-dns-754nlzea.d8fe6091-3048-43b8-91e8-1a18799b7883.privatelink.uksouth.azmk8s.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-ci-aks-cs-oncs-ds-spoke-...hcp.eusouth.azmk8s.io, privatelink.uksouth.azmk8s.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?

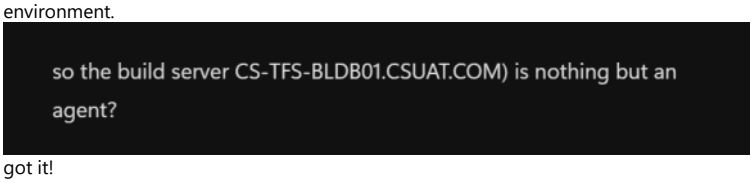
- A **build server** is a machine (physical or virtual) that runs **build jobs** for your application.
- It compiles source code, runs tests, packages artifacts, and prepares them for deployment.
- In Azure DevOps, this role is typically performed by **Build Agents** (either Microsoft-hosted or self-hosted).

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.



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?

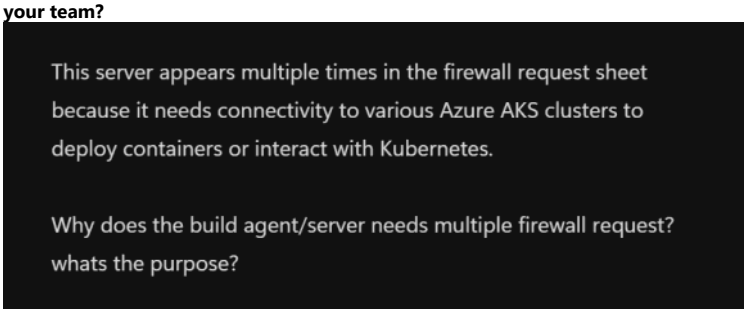
- In Azure DevOps/TFS terminology:
  - **Pipelines** define the steps (build, test, deploy).
  - **Agents** execute those steps on a machine.
- A **self-hosted agent** is installed on a server you manage (on-prem or private cloud).
- The name BLDB strongly suggests **Build**, and TFS indicates Team Foundation 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?**



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**.

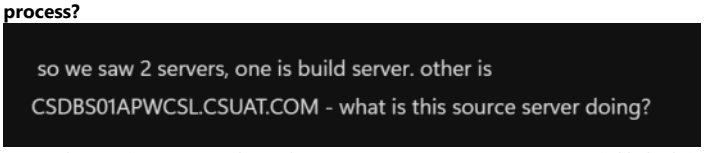
    - The build agent needs to connect to **each cluster** to deploy workloads or run Kubernetes commands.
2. **Different Environments**
    - The applications listed include **SIT**, **NEW SIT**, **CSU**, etc.
    - Each environment likely has its own AKS cluster or namespace.
    - The build agent must 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?**



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).

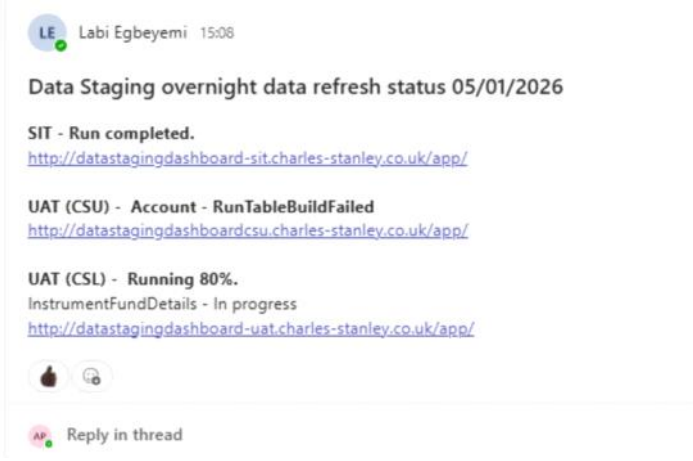
Why Firewall Rules Are Needed

- 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?**



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?

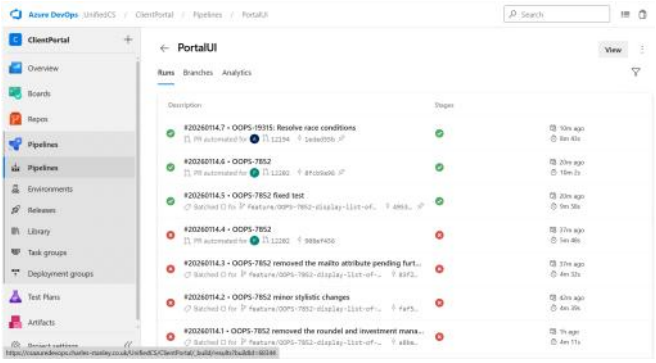
- **SIT (System Integration Testing)** and **CSU (a UAT environment)** are **different environments for the same application or system**, not completely different apps.
- Why?
  - In enterprise setups, the same application goes through multiple stages:
    - **Development → SIT → UAT → Production**
  - 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?**



Here PortalUI is a pipeline and each of the rows is a run.
A pipeline run consists of **Stages** → **Jobs** → **Steps**:

What Happens Inside Each Run

When you click on any of these runs (for example, #20260114.7), you will see:

☑ Stages

- High-level phases of the pipeline (e.g., Build, Test, Deploy).
- In your screenshot, the green check under "Stages" means the stage completed successfully.

☑ Jobs

- Each stage contains one or more **jobs**.
- Jobs run on agents (machines) and execute tasks like building code, running tests, packaging artifacts.

☑ Steps

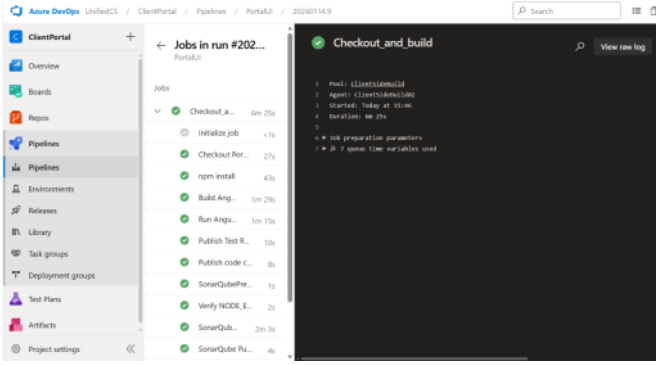
- Jobs are broken down into **steps** (small tasks).
- Examples:
  - Checkout source code
  - Install dependencies
  - Run build commands
  - Execute tests
  - Publish artifacts

☑ Logs

- Every step generates logs.
- Logs show:
  - Commands executed
  - Output and errors
  - Duration
- You can view logs by clicking on the job or step inside the run details.

How to Navigate

1. Click on any run (e.g., #20260114.7).
2. You'll see a **visual pipeline view** with stages.
3. Click on a stage → job → step to see detailed logs.



☑ What It Says

1. **Job Name**: Checkout_and_build
   - This job is responsible for checking out the code and building it.
2. **Pool**: ClientSideBuild
   - This means the job ran on an **agent pool** named ClientSideBuild.
3. **Agent**: ClientSideBuild02
   - The specific agent (machine) that executed this job.
4. **Started**: Today at 15:46
   Duration: 6m 25s
   - The job took about 6 minutes and 25 seconds to complete.
5. **Status**: ☑ Green check → **Success**.
6. **Log Details**:
   - **Job preparation parameters**: The system prepared environment variables and settings.
   - **7 queue time variables used**: These are variables passed when the pipeline was queued (e.g., branch name, commit ID).

Yes, **ClientSideBuild02 is a server (or a machine/VM)** that acts as an **Azure DevOps build agent**. Here's why:

- In your screenshot, the job ran on:
  - **Pool**: ClientSideBuild → This is an **Agent Pool**.
  - **Agent**: ClientSideBuild02 → This is a specific **agent machine** inside that pool.
- An **agent** is a piece of software installed on a machine (physical server, virtual machine, or container) that executes pipeline tasks.
  So, **ClientSideBuild02 is the name of that machine**.

☑ What does this mean practically?

- This machine is dedicated to **running build jobs** for your pipelines.
- It could be:
  - A self-hosted server in your infrastructure.
  - A VM in your cloud environment.
- It runs tasks like npm install, build Angular app, run tests, etc.

☑ Key Difference from Deployment Targets:

- **ClientSideBuild02** = Build agent (executes pipeline steps).
- Deployment targets = Servers where the **application is deployed** (UAT, Prod).

You are absolutely correct — thanks for catching that! Let me clarify this properly:

☑ ClientSideBuild02 is an Agent, Not a Server

- **ClientSideBuild02** is the **name of an Azure DevOps agent instance**.
- An **agent** is a piece of software that runs pipeline jobs.
- This agent **must be installed on a machine** (which could be a physical server, VM, or container).
- So, the **agent itself is not the server** — it's the software running on that server.

☑ How It Works

- The **server/VM** provides the hardware and OS.
- The **agent software** runs on that machine and communicates with Azure DevOps to execute tasks.
- You can have multiple agents on different machines in an **Agent Pool**.

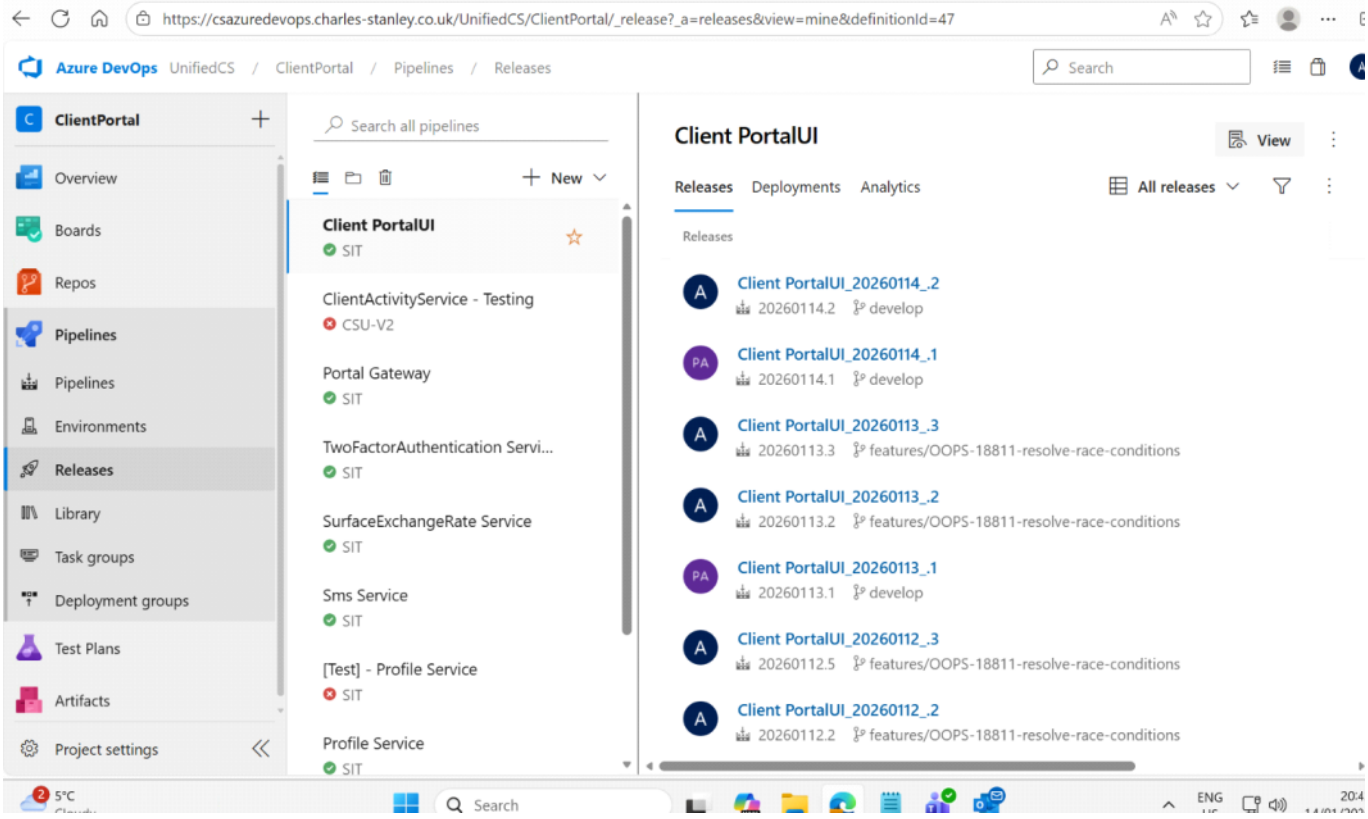
☑ Why the Confusion Happens

- We often refer to "agents" by their machine name (like ClientSideBuild02), so it feels like the agent *is* the server.
- In reality:
  - **Server/VM** = Host machine
  - **Agent** = Software installed on that machine

☑ In Your Screenshot

- **Pool**: ClientSideBuild → A collection of agents.
- **Agent**: ClientSideBuild02 → The agent software running on a specific machine.

👉 Do you want me to **draw a simple diagram showing the relationship between Azure DevOps, Agent Pools, Agents, and the underlying servers**? It will make this crystal clear visually.



From this screenshot, here's what can be inferred:

☑ Context

You are in **Azure DevOps → Releases** section for the project **ClientPortal**. This page shows **release pipelines and their recent releases**.

☑ Left Panel

- Lists multiple **release pipelines**:
  - **Client PortalUI** (highlighted)
  - ClientActivityService – Testing
  - Portal Gateway
  - TwoFactorAuthentication Service
  - SurfaceExchangeRate Service
  - Sms Service
  - Profile Service
- Each pipeline has an environment status:
  - **SIT** (System Integration Testing) → Green check ☑ means successful deployment.
  - Red ✗ means deployment failed (e.g., ClientActivityService – Testing, [Test] – Profile Service).

☑ Right Panel

- Shows **recent releases for Client PortalUI pipeline**:
  - Releases like Client PortalUI_20260114_.2, .1, .3, etc.
  - Each release has:
    - **Release ID** (e.g., 20260114.2)
    - **Branch name** (e.g., develop, features/OOPS-18811-resolve-race-conditions)
    - Status icons (A or PA) indicating approval or automated deployment.
- These represent **different deployments triggered from different branches**.

☑ What This Means

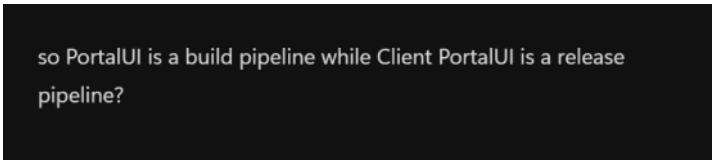
- The **Client PortalUI release pipeline** has had multiple deployments recently:
  - Some from develop branch.
  - Some from a feature branch (OOPS-18811-resolve-race-conditions).
- All listed releases appear to have completed (no red ✖ here).
- Other pipelines (like ClientActivityService – Testing) have failures in SIT environment.

☑ In Simple Terms

This page shows:

- Which **release pipelines exist** for your project.
- Their **environment status** (SIT mostly successful except a few failures).
- Recent **release deployments** for Client PortalUI, including branch and release number.

👉 Do you want me to **explain what the icons (A, PA) mean and how approvals work in release pipelines**? Or should I **give you a clear diagram of how build → release → environment deployment flows in Azure DevOps**?



Yes, you've got it right:

☑ PortalUI = Build Pipeline

- This is the pipeline that **builds the application**.
- It runs tasks like:
  - Checkout code
  - Install dependencies
  - Compile/build the Angular app
  - Run tests
  - Perform code analysis (SonarQube)
- Output: **Build artifacts** (e.g., packaged app files).

☑ Client PortalUI = Release Pipeline

- This pipeline **deploys the built artifacts** to environments (like SIT, UAT, PROD).
- It uses:
  - **Deployment groups** or service connections to push code to servers.
  - Approval gates (manual or automated).
- Output: **Application deployed to target environment**.

☑ Relationship

1. **PortalUI (Build)** → Produces artifacts.
2. **Client PortalUI (Release)** → Takes those artifacts and deploys them to SIT/UAT/PROD.

👉 Do you want me to **draw a simple diagram showing the flow from PortalUI (build) → artifacts → Client PortalUI (release) → environments**? It will make this crystal clear visually.

Artifacts:

ClientPortalAppPackage (version 20260114.2) from the develop branch.
This means the release is deploying the build artifact created earlier by the build pipeline.

Where does the build pipeline save the artifact which is later picked up by release pipeline?

Excellent question! Here's how it works in Azure DevOps:

☑ Where Does the Build Pipeline Save Artifacts?

- When a **build pipeline** (like your **PortalUI pipeline**) completes, it produces **artifacts** (e.g., compiled code, packaged app).
- These artifacts are **published to Azure DevOps artifact storage** using the PublishBuildArtifacts task (or similar).
- They are stored in the **Azure DevOps server/cloud**, associated with that specific build run.

☑ How Release Pipeline Picks It Up

- The **release pipeline** (Client PortalUI) is linked to the build pipeline as an **artifact source**.
- When you create a release, Azure DevOps fetches the artifact from:
  - The **build run's artifact storage** in Azure DevOps.
  - It uses the artifact name (e.g., ClientPortalAppPackage) and version (e.g., 20260114.2).

☑ Where to See It

1. Go to **Pipelines** → **Builds** → **Select the build run**.
2. Click **Artifacts** tab.
3. You'll see the artifact files (e.g., zip, DLLs, Angular build output).
4. These files are stored in Azure DevOps and can be downloaded.

☑ Physical Location

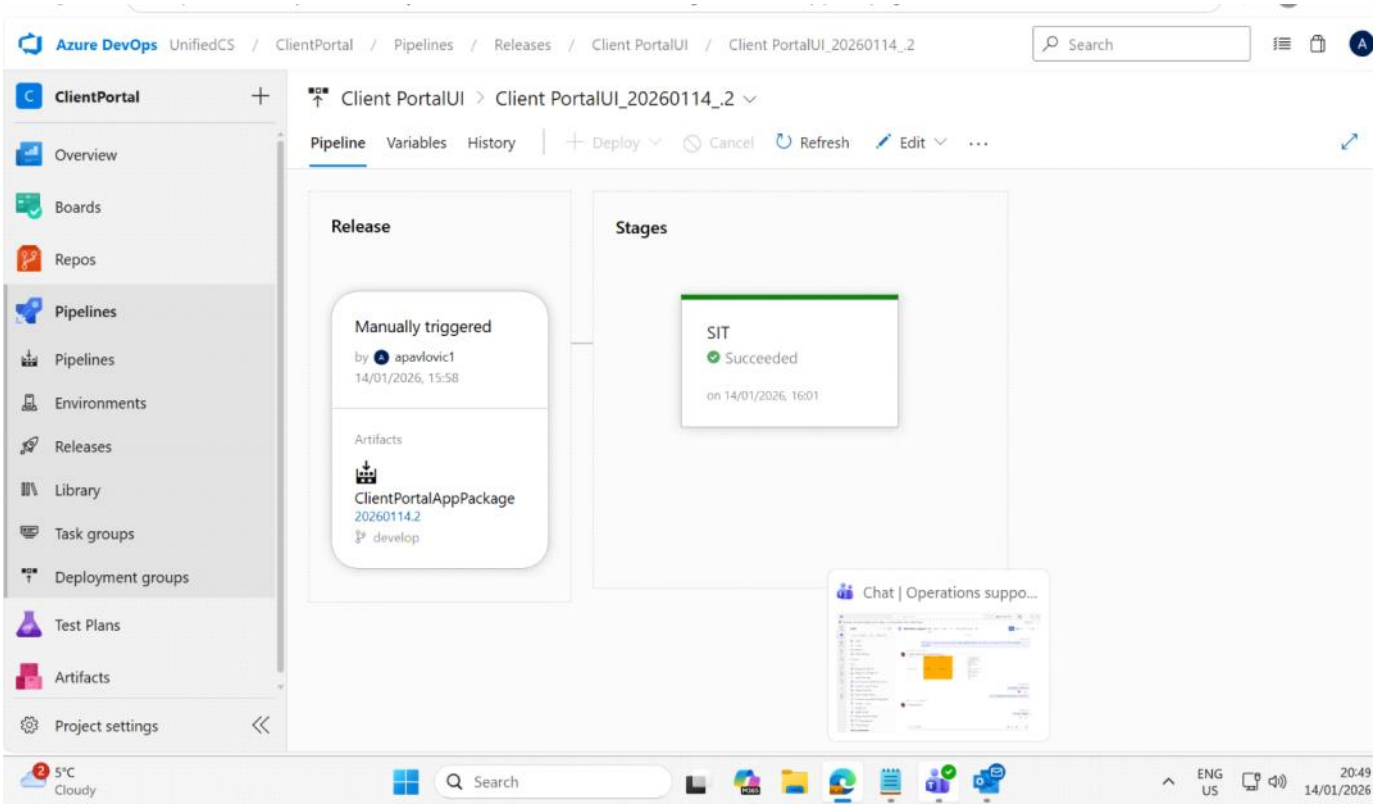
- If you use **Microsoft-hosted agents**, artifacts are stored in Azure DevOps cloud.
- If you use **self-hosted agents**, artifacts are uploaded from your server to Azure DevOps storage after the build completes.

☑ Flow

Build Pipeline (PortalUI)
→ Runs on agent → Produces files → **PublishBuildArtifacts** → Stored in Azure DevOps artifact storage
→ **Release Pipeline (Client PortalUI)** picks artifact → Deploys to SIT/UAT/PROD.

👉 Do you want me to **show you a diagram of this flow (Build → Artifact → Release → Environment)** or **give you the exact steps to configure artifact publishing and linking to release pipeline**?

Now we take one of the releases of Client PortalUI release pipeline



From this screenshot, here's what can be inferred:

☑ Context

You are viewing the details of a **specific release** in the **Client PortalUI release pipeline** in Azure DevOps.

☑ Key Observations

1. **Release Name**: Client PortalUI_20260114_.2
   - This is the release identifier.
2. **Trigger**:
   - **Manually triggered** by a user (apavlovic1) on **14/01/2026 at 15:58**.
   - So this release was not automated; someone manually started it.
3. **Artifacts**:
   - ClientPortalAppPackage (version 20260114.2) from the **develop branch**.
   - This means the release is deploying the build artifact created earlier by the build pipeline.
4. **Stage**:
   - **SIT** (System Integration Testing environment).
   - Status: ☑ **Succeeded** at **16:01** on the same day.
   - So the deployment to SIT completed successfully.

☑ What This Means

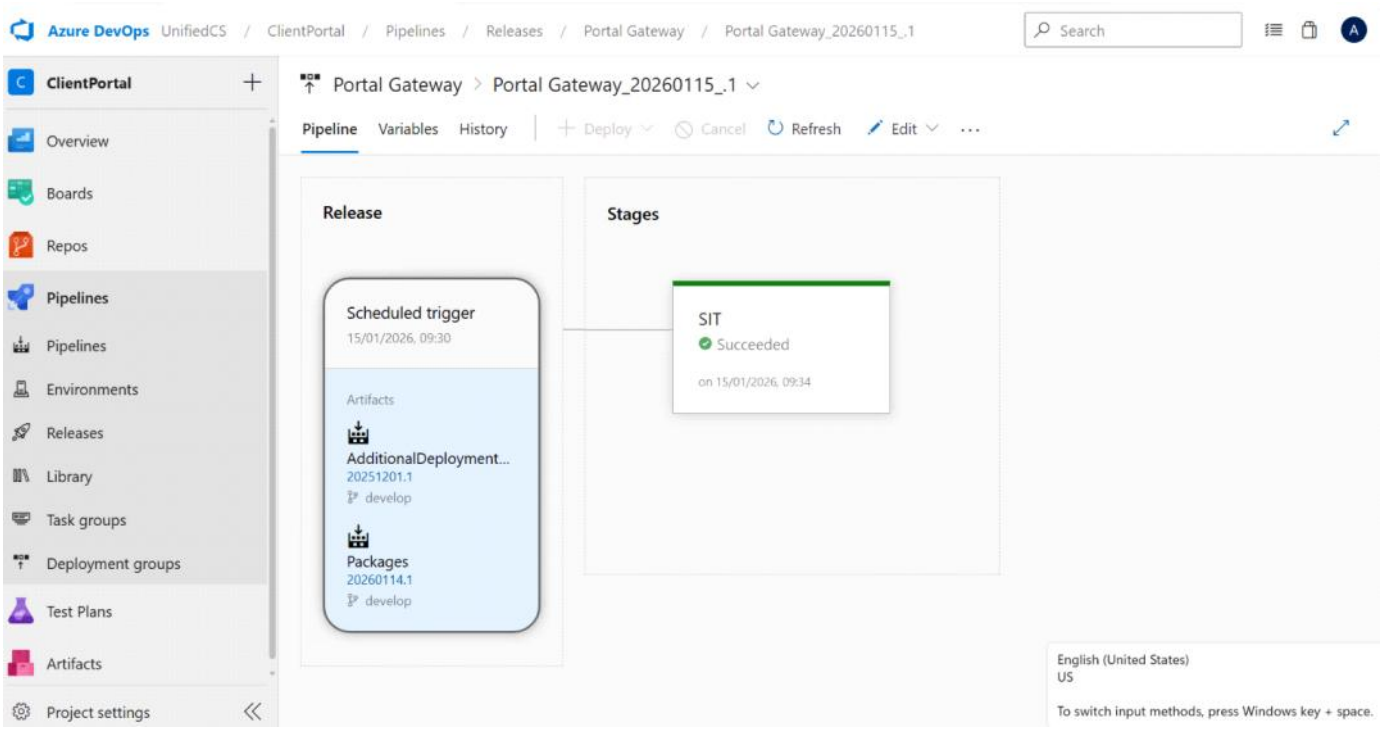
- A manual deployment was done for the **Client PortalUI application** using the artifact from the **develop branch**.
- The deployment targeted the **SIT environment** and finished successfully.
- No other stages (like UAT or PROD) are shown here, so this release only involved SIT.

☑ In Simple Terms

This release:

- Took the latest build from **develop**.
- Was manually triggered.
- Successfully deployed to the SIT environment in about **3 minutes** after starting.

👉 Do you want me to **explain what happens inside this SIT stage (tasks, deployment steps)**? Or should I **show you how SIT fits into the overall release pipeline flow (SIT → UAT → PROD)**?



From this screenshot, here's what can be inferred:

☑ Context

You are viewing a **specific release** in the **Portal Gateway release pipeline** in Azure DevOps.

☑ Key Observations

1. **Release Name**: Portal Gateway_20260115_.1
   - This is the release identifier.
2. **Trigger**:
   - **Scheduled trigger** at **15/01/2026, 09:30**.
   - This means the release was automatically triggered by a schedule, not manually.
3. **Artifacts**:
   - **AdditionalDeployment...** (version 20251201.1) from develop branch.
   - **Packages** (version 20260114.1) from develop branch.
   - So this release uses **two artifacts** from previous build pipelines.
4. **Stage**:
   - **SIT** (System Integration Testing environment).
   - Status: ☑ **Succeeded** at **09:34** on the same day.
   - Deployment took about **4 minutes** after the scheduled trigger.

☑ What This Means

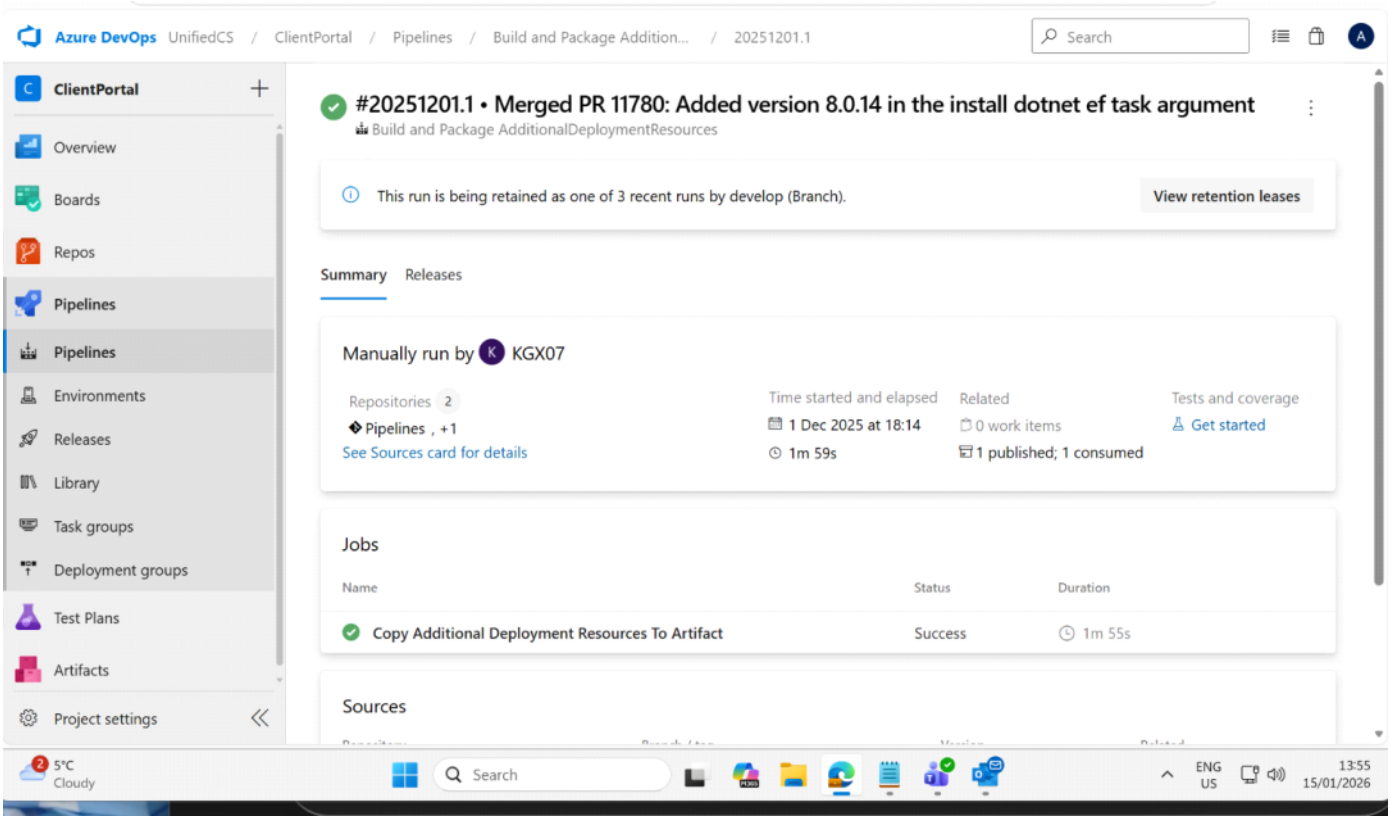
- The **Portal Gateway release pipeline** deployed two artifacts (likely different components or modules) to the **SIT environment**.
- The deployment was **scheduled**, meaning it runs automatically at a predefined time.
- The deployment completed successfully without errors.

☑ In Simple Terms

This release:

- Was triggered automatically at 09:30.
- Used two artifacts from the **develop branch**.
- Successfully deployed to SIT in about 4 minutes.

👉 Do you want me to **explain why there are two artifacts here and how multiple artifacts work in a release pipeline**? Or should I **show you a diagram of how scheduled triggers and multi-artifact deployments fit into the CI/CD flow**?



i went to one of the artifact and seeing this

Here is exactly what can be inferred from this screen — **and this finally answers your question about which build pipeline produced the artifact.**

☑ What You Are Looking At

You are viewing a **build pipeline run**.
The pipeline name is at the top:

👉 Build and Package AdditionalDeploymentResources

This is the **name of the build pipeline** that produced the artifact you selected earlier.
This confirms:

🎯 **This artifact came from the build pipeline:

👉 "Build and Package AdditionalDeploymentResources"**

☑ Breakdown of What the Screen Shows

1. Build run ID

#20251201.1
This identifies the specific run of the build pipeline.

2. Commit message

"Merged PR 11780: Added version 8.0.14 in the install dotnet ef task argument"

3. Trigger

Manually run by user **KGX07**.

4. Job executed

- **Copy Additional Deployment Resources To Artifact**
- Status: **Success**
- Duration: **1m 55s**

This job copies files into the artifact staging directory and then publishes them.

5. Artifacts published

The screen shows:

- **1 published; 1 consumed**

Meaning:

- One artifact was created by this pipeline.
- One artifact was consumed (likely a dependency or sub-pipeline artifact).

☑ So What Does This Establish?

The artifact used in your **Portal Gateway** release pipeline (seen earlier as "AdditionalDeployment...") **was created by this build pipeline**:

★ Build and Package AdditionalDeploymentResources

This is the exact pipeline responsible for producing the artifact.

🧠 Connecting It Back to Your Release Pipeline

Earlier, the release pipeline showed:

- Artifact: **AdditionalDeployment...**
- Version: **20251201.1**

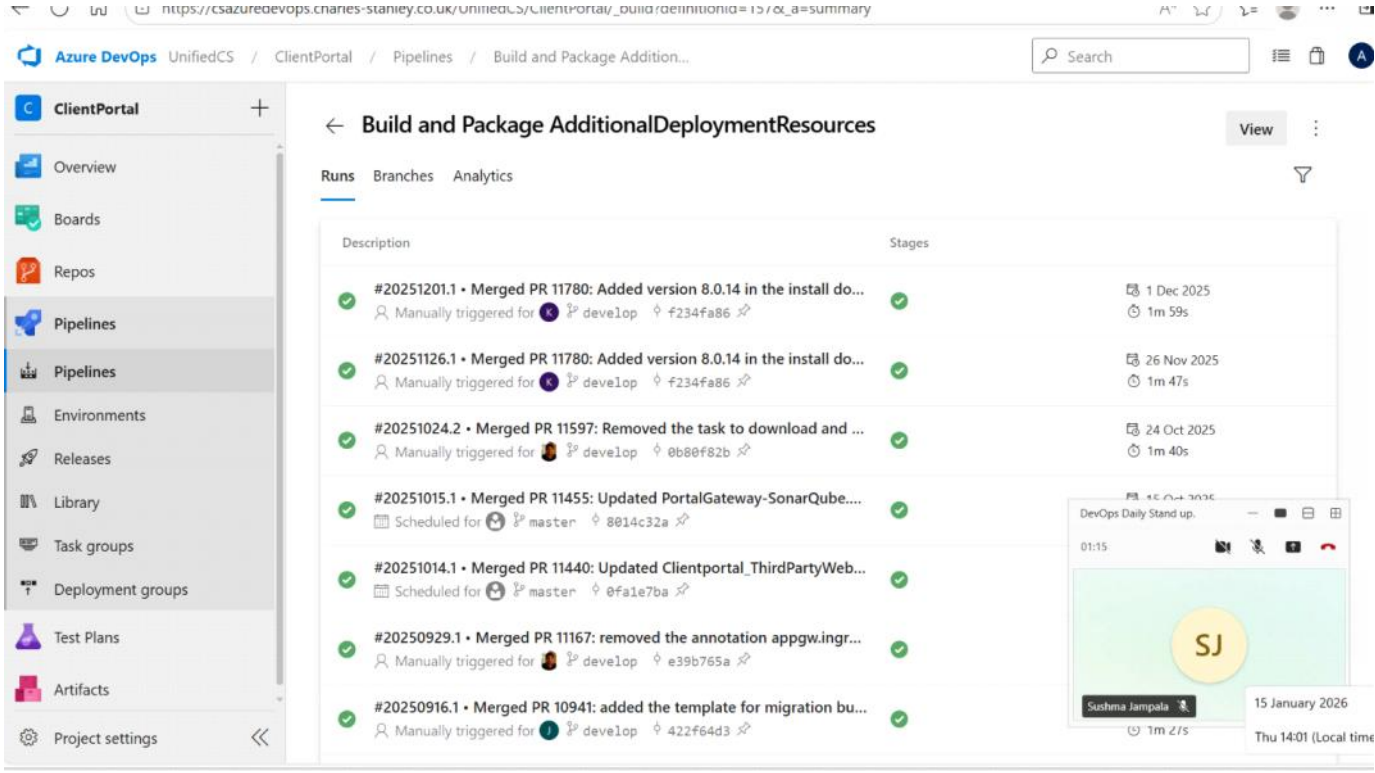
Now you are seeing:

- Build run: **20251201.1**
- Build pipeline: **Build and Package AdditionalDeploymentResources**

Perfect matching → that is your artifact source.

🔍 Want me to help with the second artifact too?

If you click the **Packages** artifact (the other one in your release), I can tell you **which build pipeline created that one too**.
Send that screenshot, and I'll map it out for you.



☑ What "SIT" Actually Means

SIT = System Integration Testing environment.
It is simply **a deployment stage** in your release pipeline.
Azure DevOps does **not** decide what infrastructure sits behind "SIT".
Your organization decides that.

☑ SIT Could Be Any of These:

✓ A group of Windows or Linux servers

(deployed using *Deployment Groups* — which YOU have in your project)

✓ A VM scale set
✓ A set of on-prem servers
✓ An AKS cluster
(only if Kubernetes deployment tasks are used)
✓ App Services

❗ But based on everything you've shown so far…
Your entire pipeline setup strongly suggests:
SIT is *NOT* an AKS cluster.
It is most likely:
A Deployment Group of servers
where the release pipeline deploys your application.
Here's why:

🔍 Evidence from your screenshots
1. You looked at Deployment Groups earlier
You saw **18 online servers** and **2 offline** in a Deployment Group.
This means your release pipeline probably uses *these servers* as targets.
2. Your pipeline uses typical steps:
- "Copy Additional Deployment Resources to Artifact"
- "Packages"
- .NET, Angular, or scripting-based deployment
  These are *not* Kubernetes deployments.

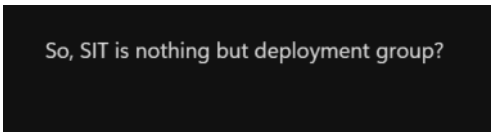
3. No Helm charts, no kubectl commands, no AKS service connection
In an AKS-based SIT environment, you would see tasks like:
- kubectl apply
- helm upgrade
- Kubernetes YAML manifests
- A Kubernetes service connection in Azure DevOps

None of your screenshots show that.
4. The "SIT" stage in the release shows a single green box
AKS deployments usually show more detailed gating or multiple tasks.
Deployment Groups → simple with one green check.

🎯 Conclusion
Based on everything visible:
👉 SIT is almost certainly a Deployment Group of servers, not AKS.
Your release pipeline is:
1. Picking artifacts from build pipelines
2. Deploying them to SIT
3. Through a **Deployment Group** (servers/VMs)
4. Using deployment tasks, not Kubernetes tasks

If you want to be 100% certain:
I can tell for sure if SIT is AKS if you show:
✓ The tasks inside the SIT stage
(click the green SIT box → show the tasks)
or
✓ The Release definition → Stage → Phase → Target type
It will say:
- Deployment Group
- Agent Job
- Kubernetes
- App Service
- VMSS
- etc.

Just send me a screenshot of the SIT tasks, and I'll confirm it precisely.



Short answer: *Not exactly. SIT is not "the deployment group" itself — but in *your* project, SIT is an environment that *uses* a deployment group to deploy to servers.*
Let me explain very clearly and simply.

☑ What SIT Actually Is

SIT = System Integration Testing environment.
It is just an **environment name** in your Azure DevOps Release pipeline.
By itself, *SIT is not a deployment group or AKS or VM.*
It's just a **stage**.

🧱 Inside the SIT stage, YOU choose what to deploy to:

A SIT stage can deploy to:

- Deployment Group (servers)
- AKS cluster
- Azure App Services
- VM Scale Sets
- Linux/Windows VM via SSH/WinRM
- Kubernetes cluster
- Docker containers
- Anything else

The environment name "SIT" does NOT tell you the infrastructure.

🎯 But based on YOUR pipelines, here is the truth:

✓ In YOUR project, SIT = environment that deploys to a Deployment Group of servers

Why?

1️⃣ You already viewed Deployment Groups with 20 servers

Your screenshot showed:

- 18 online servers
- 2 offline
- Named things like
  CSU-DSGETLD01-dataService,
  CSU-DSGA01-graphQLApi

These look like **on-prem / VM servers**, not AKS nodes.

2️⃣ Your SIT tasks are simple release tasks

You are using tasks like:

- Copy files
- Deploy packages
- Run scripts on servers
  These are typical **Deployment Group tasks**, not Kubernetes tasks.

3️⃣ No AKS indicators

You do **not** see:

- kubectl apply
- helm upgrade
- Kubernetes YAML files
- Kubernetes service connection
- Namespace selection
- Container registry image deployment

So **your SIT environment is NOT AKS.**

4️⃣ Release Pipeline UI is the classic Deployment Group style

AKS deployments look different (multi-step Kubernetes actions).

Your SIT box is a **single green block**, typical of Deployment Group deployments.

🧠 Therefore:

SIT is NOT the deployment group.

BUT:

Your SIT stage *uses* a Deployment Group as its deployment target.

So the structure is:
SIT (environment / stage)
↓
Deployment Group
↓
Servers/VMs (your 20 machines)

🧩 Final Answer:

✓ SIT is an environment in your release pipeline.
✓ In your project, this SIT environment deploys to a Deployment Group.
✓ That Deployment Group contains the SIT servers where the app is installed.
❌ SIT is not AKS in your case.
❌ SIT is not itself a server group, it *uses* the group.

If you want, send me **one screenshot of the SIT stage tasks**, and I will tell you EXACTLY which servers, agents, or clusters SIT is deploying to.

So, SIT releases to deployment groups which are different from the deployment groups that will be used to be released by UAT environment? or same?

Great question — and this is a **very important concept** in Azure DevOps environments.
Let me give you the **clearest possible explanation**.

☑ Do SIT and UAT use the SAME Deployment Group or DIFFERENT ones?

It depends on how your organization configured environments, but in **almost all real enterprise setups**, the answer is:

⭐ SIT and UAT almost always use DIFFERENT Deployment Groups.

Here's why 👇

🎯 Why They Are Usually Different

1️⃣ SIT and UAT run on different servers

- SIT = testing by developers/QA teams
- UAT = testing by business users/end users before going to PROD

So:

- SIT servers are usually more experimental and frequently updated.
- UAT servers are more stable and mimic production.

➡ Therefore **each has its own deployment group**.

**2️⃣ Deployment Groups represent actual machines

A deployment group is literally a **collection of servers** with the agent installed.
Example:

```
DeploymentGroup_SIT
  - SIT-APP01
  - SIT-APP02
  - SIT-WEB01
  - SIT-SERVICE01
DeploymentGroup_UAT
  - UAT-APP01
  - UAT-APP02
  - UAT-WEB01
```

SIT and UAT servers are **different machines**, so deployment groups must be different.

3️⃣ Release pipeline stages map to different target groups

Your release pipeline likely looks like this:

```
SIT  →  UAT  →  PROD
|       |       |
DG-SIT   DG-UAT   DG-PROD
```

Each stage has:

- Different approvals
- Different target servers
- Different deployment logic

🧠 In Your Case (Based on All Your Screenshots)

✓ You already saw a deployment group with ~20 targets (servers)

Names like:

- CSU-DSGETLD01-dataService
- CSU-DSGA01-graphQLApi

This is likely the **SIT deployment group**.

✓ If you navigate to UAT stage in the release pipeline

You will probably see **another deployment group** like:

- UAT-APP01
- UAT-API01
- etc.

✓ The release pipeline decides

which deployment group corresponds to which stage.

🎯 Final Answer (Crystal Clear)

⭐ SIT stage uses one Deployment Group.

⭐ UAT stage will use a DIFFERENT Deployment Group.

Unless your organization explicitly chose to reuse the same servers for both (very rare and not recommended).

📌 If you want 100% confirmation

Just click:

Releases → Your Release Pipeline → Edit → Stages → SIT → Tasks → Phase

You will see:

- **Target: Deployment Group**
- Deployment group name (e.g., DL-SIT)

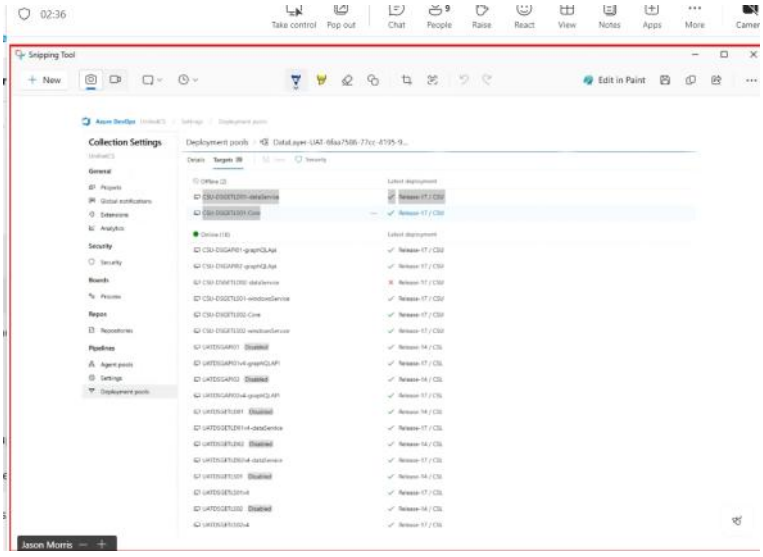
Then do the same for:

UAT → Tasks → Phase

You'll see whether it points to:

- DL-UAT (most likely),
  or
- same group (rare).

If you send me a screenshot of the **UAT stage**, I can immediately tell you whether it uses the same or different Deployment Group.



In your screenshot, you are looking at **Azure DevOps → Deployment pools → Targets tab** for the pool DataLayer-UAT.
Here's what these represent:

- **These are not agents**.
  They are **deployment targets (machines)** that belong to a **deployment group** inside a deployment pool.
  Deployment targets are typically servers or VMs where your application gets deployed during a **release pipeline**.

Details in Your Screenshot

- **Offline (2)**: Two targets are currently offline.
- **Online (18)**: Eighteen targets are online and available.
- Each row shows:
  - **Target name** (e.g., CSU-DSGETLD01-dataService, CSU-DSGA01-graphQLApi)
  - **Status** (Online, Offline, Disabled)
  - **Latest deployment** (e.g., Release-17 / CSU)
- Some targets are marked **Disabled**, meaning they are not active for deployments.

Difference Between Agents and Deployment Targets

- **Agents**: Execute pipeline jobs (build/test tasks) in **build and release pipelines**.
- **Deployment Targets**: Are machines where your app is deployed during **release pipelines**.

☑ So, these are **deployment targets**, not build agents.

Do you want me to **show you where to find actual build agents in Azure DevOps** and explain the difference between **agent pools** and **deployment pools**?

Ok so we have seen agent pools and deployment pools - both are servers - right?

Great question — and your understanding is *very* close now.
Let me clarify it **cleanly and accurately**:

☑ Are Agent Pools and Deployment Pools both servers?

Not exactly.
Both *use* servers — but they are not the same thing.
Let's break it down simply:

▧ 1. Agent Pools → Contain Build/Release Agents

An **Agent Pool** is a *collection of agents*.
An **agent** is *software installed on a server* (or VM) that runs pipeline jobs.

✓ The agent itself is NOT the server

It's just a piece of software running on a server.

✓ But it requires a server to run on

So an agent pool indirectly uses servers.

Purpose of agents:

- Run CI builds
- Run tests
- Execute scripts
- Build and pack artifacts

You saw examples like:

- ClientSideBuild02 (an agent)
  This agent runs on a server (but the agent name is not the server name).

▨ 2. Deployment Pools → Contain Deployment Targets

A **Deployment Pool** contains **deployment targets**.
A **deployment target** is:

- A server/VM where your application gets deployed
- With the Deployment Agent installed

✓ Here, the "deployment target" *is* the server

Because the deployment agent is installed *directly* on the server that hosts the app.
You saw deployment targets like:

- CSU-DSGETLD01-dataService
- CSU-DSGA01-graphQLApi
  These **are actual servers** (VMs).

🆚 Summary: Are they servers?

Concept	Is it a server?	Runs on a server?	Purpose
Agent	❌ No	✓ Yes	Build/test pipelines
Agent Pool	❌ No	✓ Contains agents that run on servers	Group of build agents
Deployment Target	✓ Yes	✓ Itself *is* a server	Where app is deployed
Deployment Pool	❌ No	✓ Contains servers	Group of deployment targets

🎯 Crystal Clear Conclusion

❌ Agent Pool ≠ Servers

But:

✓ Agents inside agent pools run *on* servers.

✓ Deployment Pool = A list of servers that host the app

Every deployment target *is a real server*.

🧠 In simple language:

- **Agent Pool → Workers that build the app (agents), running on servers**
- **Deployment Pool → Actual environment servers where app is deployed**

If you want, I can also give you a **diagram** showing the exact relationship between:

- Build agents
- Deployment targets
- SIT/UAT/PROD servers
- Release pipeline

Just let me know!

Ok agents run on servers, right - can that server also act as a deployment group? or for deployment group, we would require different servers?

Great question — and this is where **most people get confused**, so let's clear it once and for all in the simplest and most accurate way possible.

☑ Can the same server act both as a Build Agent AND a Deployment Target?

Yes, technically it *can*.

But…

In real-world enterprise setups, it almost NEVER does.

Let's break this down clearly.

▧ Build Agent (Agent Pool) server

A server that has the **Azure Pipelines Agent** installed to run:

- Builds
- Tests
- Compilation
- Packaging
- Static code analysis

This server is meant to be:

- Fast
- Clean
- Disposable
- Not running your real application
- Not hosting customer traffic

▨ Deployment Target (Deployment Group) server

A server that has the **Deployment Agent** installed to:

- Receive deployed application files
- Host the actual running application
- Serve SIT, UAT, PROD traffic or backend services

This server is meant to be:

- Stable
- Hosting business services
- Environment-specific (SIT/UAT/PROD)
- Not used for building code

So… can a server perform both roles?

Technically = YES

You *can* install both:

- Build Agent
- Deployment Agent

on the same physical VM/server.
Azure DevOps does not block this.

But Practically = Not Recommended

Here is why companies **never** do this:

1. Build servers need high CPU usage & toolchains

During builds:

- npm install
- Angular build
- .NET compile
- SonarScan
  These can max out the CPU.

This would **interrupt your SIT/UAT app** if it's on the same server.

2. Deployment Targets should be stable servers

These servers run:

- APIs
- Web apps
- Background services
- Business logic

You don't want your UAT/SIT/PROD server to suddenly slow down because a **huge build job is running**.

3. Security risk

Build servers have:

- build secrets
- pipelines
- developer tools
  Deployment servers have:
- app secrets
- DB connections
- production-like environment

Mixing them is bad practice.

4. Environment separation

SIT/UAT/PROD must be isolated.
Build servers should NEVER be part of any environment.

Final Clear Answer

CAN the same server act as both Agent and Deployment Target?

Yes, technically.

SHOULD it?

No — strongly discouraged.

In your project, are they the same?

No.
Your screenshots clearly show:

- **Agent servers** (like ClientSideBuild02) run builds.
- **Deployment group servers** (like CSU-DSGETLD01-dataService) host SIT services.

They are **different sets of servers**.

In Simple Terms

> **Build agents build your app. Deployment groups run your app. They should be on different servers.**

If you want, I can draw a simple architecture diagram showing the separation between:

- Build Agent Servers
- Deployment Group Servers
- SIT/UAT/PROD environments

Just say **"show diagram"**.

so, build agents, release agents and deployment agents each require different server in organization - right?

Great question — and this is where everything finally clicks into place.
Let's make it *very clear and very correct* 👇

⭐ Do build agents, release agents, and deployment agents each require different servers?

☑ YES — in a real enterprise environment, they are almost always on different servers.

But let me explain **why**, and what each one does.

▧ 1. Build Agents (Agent Pools)

✓ Purpose

- Build the application
- Run tests
- Do SonarQube analysis
- Package artifacts

✓ Characteristics

- High CPU usage
- High disk I/O
- Need toolchains (Node, .NET, Java, Angular, etc.)
- Used heavily throughout the day

❌ SHOULD NOT run on SIT/UAT/PROD servers

Because it would slow down or break the actual applications.

✓ Typically runs on:

- Dedicated **build servers**
- Hosted build agents
- Azure VM build agents

▨ 2. Release Agents

✓ Purpose

- Execute release pipeline tasks
- Coordinate deployments
- Interact with artifact storage
- Call deployment groups or Kubernetes clusters

✓ Characteristics

- Can run on Microsoft-hosted or self-hosted agents
- Do *not* host the application
- Trigger tasks like copying files, running PowerShell, etc.

❌ SHOULD NOT run on SIT/UAT/PROD servers

Because they're only for orchestration, not hosting.

✓ Usually runs on:

- Dedicated **release agent servers**

- Shared agent pools
- (Often the same servers as build agents, but still **not** SIT/UAT servers)

🔲 3. Deployment Agents (Deployment Groups / Targets)

✓ Purpose

- Host the actual running application
- Receive deployments
- Serve SIT / UAT / PROD traffic

✓ Characteristics

- Stable
- Protected
- Represent actual environments

❌ SHOULD NOT run builds or release pipelines

These are your **SIT / UAT / PROD machines**, so they must stay stable.

✓ Typically:

- SIT servers
- UAT servers
- Production servers

🎯 So: Do they require different servers?

☑ YES — here's the correct separation:

BUILD SERVERS (Build Agents)
  └── Do builds + tests
RELEASE SERVERS (Release Agents)
  └── Orchestrate deployments
SIT SERVERS / UAT SERVERS / PROD SERVERS (Deployment Targets)
  └── Host the actual deployed application
Each has a different function, so **they must not overlap**.

❗ CAN they be on the same server?

✓ Technically yes

Azure DevOps allows installing:

- Build Agent + Deployment Agent on the same machine

❌ Practically no

Because:

- Build load disrupts your running application
- Security concerns
- Server isolation is mandatory for environments
- Best practices forbid mixing these roles

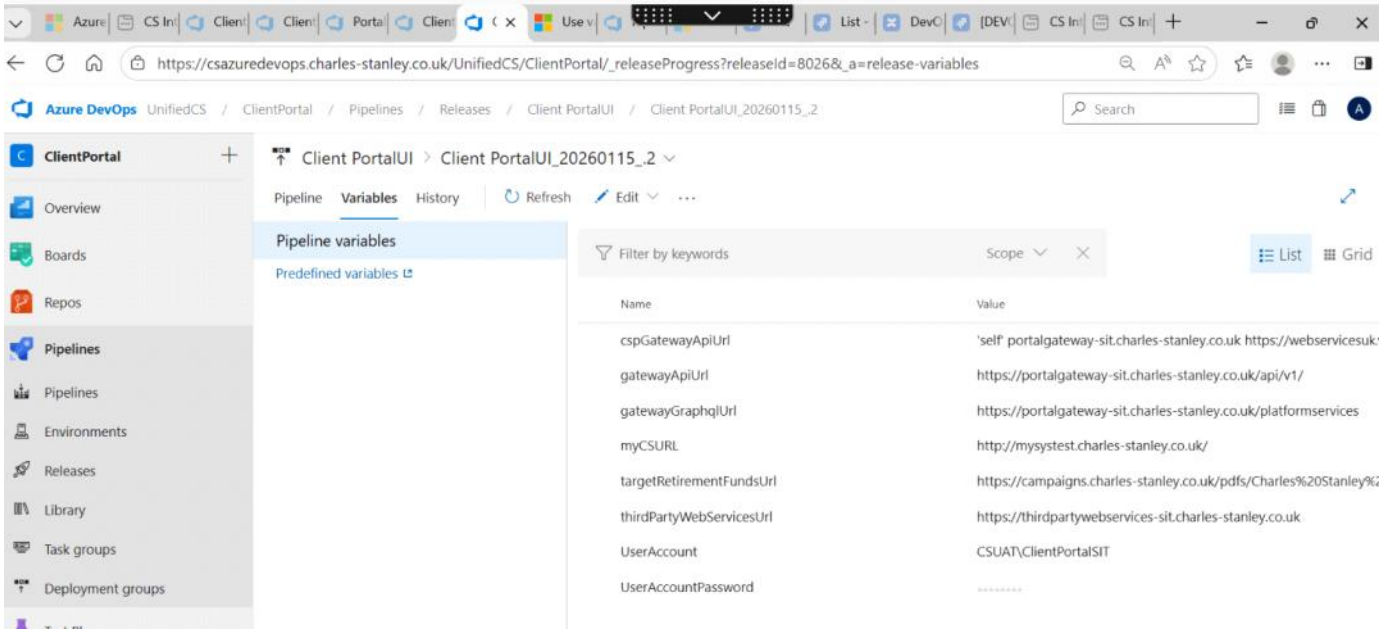
👉 In your organization (based on screenshots):

They **are already separated**, which is the correct practice.

⭐ Final Clear Answer

Build agents, **release agents**, and **deployment agents** each require **different servers** in a real enterprise environment —
and your organization is correctly following this separation.

If you want, I can draw a full architecture diagram showing **CI → Artifact → CD → SIT → UAT → PROD** with agent and server separation.



Here's a **clear and simple explanation** of exactly what you are seeing in this screenshot, what these variables mean, and where they are used.

☑ What Is Depicted Here?

This screen shows the **Pipeline Variables** tab for a **Release Pipeline** run:
Client PortalUI → Release: Client PortalUI_20260115_.2 → Variables tab
These variables belong to the **release pipeline**, not the build pipeline.
You are looking at **release-time variables** that get injected into your deployment tasks when deploying to the SIT environment.

🎯 What Are These Variables?

These are **release pipeline variables**, specifically configured to:

- Store connection URLs
- Store environment-specific API endpoints
- Store login credentials for deployment scripts
- Store service URLs used by the UI (ClientPortalUI) to call other backend services

Examples from your screenshot:

Variable Name	Meaning
cspGatewayApiUrl	URL of the CSP API gateway for SIT environment
gatewayApiUrl	Another API endpoint used by the UI
gatewayGraphqlUrl	GraphQL endpoint for the backend
myCSURL	URL of the MyCS application in SIT
targetRetirementFundsUrl	URL pointing to retirement fund service
thirdPartyWebServicesUrl	URL for third-party integrations
UserAccount	Account name used during deployment tasks
UserAccountPassword	Password used by scripts (hidden)

These URLs all end with -sit.charles-stanley.co.uk → meaning they are **SIT-specific configuration values**.

🤔 Where Are These Variables Used?

These variables are used in:

✓ Release pipeline tasks

Inside the SIT stage → tasks like:

- PowerShell scripts
- JSON configuration replacements
- AppSettings/Web.config transformation
- Angular environment file replacement
- Deployment scripts
- API endpoint injection

Example usage inside a PowerShell task:
Write-Host "Using API URL: $(gatewayApiUrl)"
Example in a config transform step:
"API_URL": "$(gatewayApiUrl)"
Example in an Angular environment replacement step:
ng build --configuration=sit --baseApiUrl=$(cspGatewayApiUrl)

✓ Inside Deployment Groups

When the release pipeline deploys to SIT servers, the deployment tasks **read these variables** and inject them into:

- Web.config
- appsettings.json
- Angular environment.ts
- Config files under /assets/
- .env files
- or script parameters

🧩 Why Do These Variables Exist?

Because:

- SIT
- UAT
- PROD

each require **different URLs**, **different endpoints**, **different credentials**, and **different environment configs**.
Keeping them in **release variables** allows you to:

- Change URLs per environment
- Avoid hardcoding
- Keep secrets out of the code
- Update values without modifying pipelines

🔥 How Azure DevOps Uses Them in Deployment

During the SIT deployment:

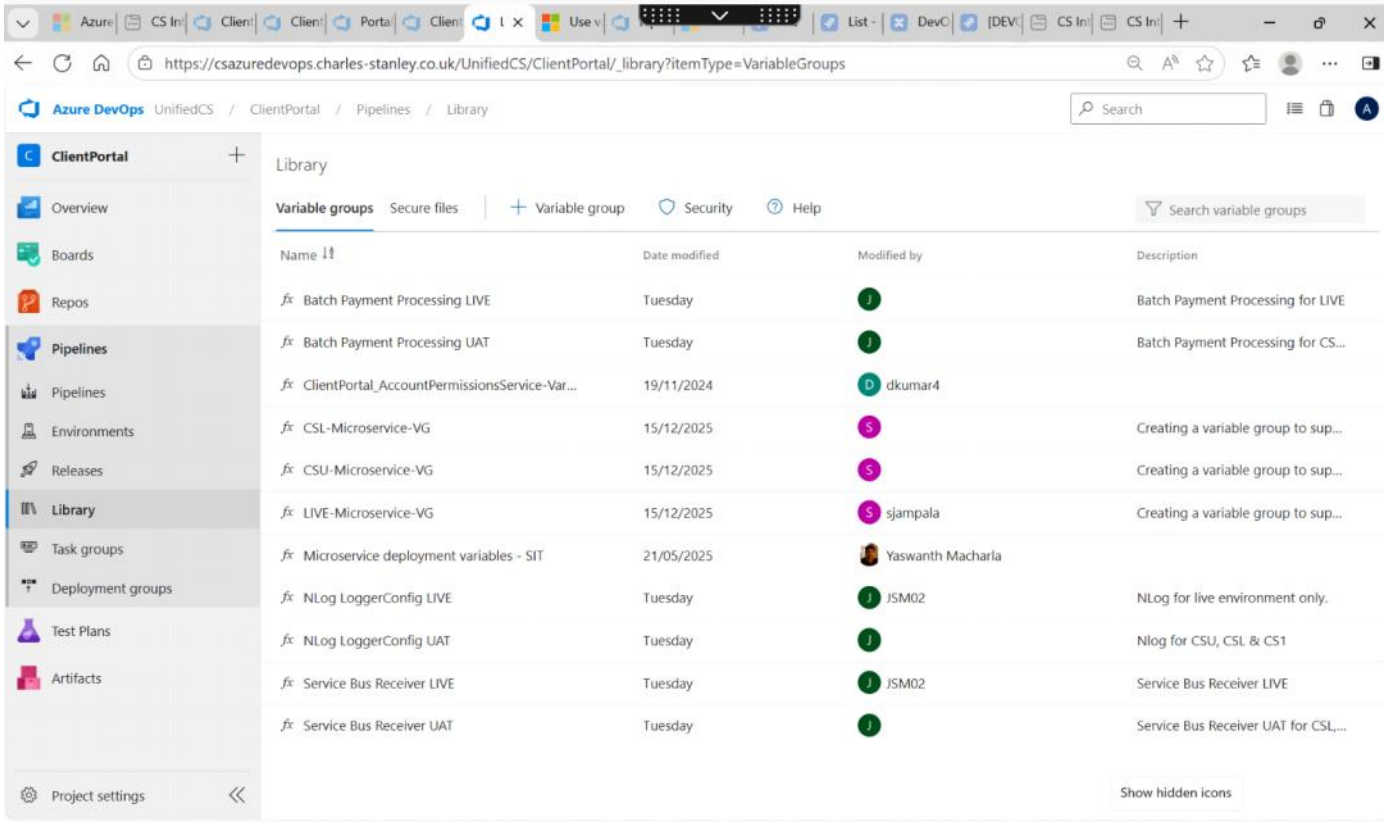
1. Release pipeline starts
2. Variables are loaded
3. Deployment tasks run
4. The tasks substitute these variables into:
    - Config files
    - Scripts
    - App settings
    - Environment files

These values become part of the deployed SIT application.

🏁 In Simple Terms

> **These are environment-specific configuration values used by the release pipeline when deploying to SIT. They determine what URLs, APIs, credentials, and endpoints your deployed SIT application will use.**

If you want, send me a screenshot of the **SIT tasks** and I will show you **exactly which task uses which variable, line by line**.



Got it — let's make this **super clear, simple, and crisp** so you understand exactly what you are looking at and how these variables differ from the release-pipeline variables you saw earlier.

☑ What is depicted here?

This screen shows the **Library → Variable Groups** in Azure DevOps.
These are **Variable Groups** — collections of variables stored centrally and reused across multiple pipelines.
Think of them like:

Global/shared environment configuration stored in one place.

🎯 **How these differ from the previous variables?

(Lucid, crisp & simple)**

▧ 1. Previous Variables (in the screenshot before this)

Those were:

✓ Release Pipeline Variables

They belong to **ONE SPECIFIC release** — for example, *Client PortalUI release*.

Where they apply:

- Only inside that **release pipeline**
- Only for that **stage/run**
- Typically used for environment-specific deployment config (like SIT URLs)

Example from before:

- gatewayApiUrl
- myCSURL
- UserAccount

These variables exist only in your **Client PortalUI release**.

▨ 2. Variables in THIS new screenshot

These are:

✓ Variable Groups (Library Variables)

These belong to the entire **project** and can be shared across:

- Multiple build pipelines
- Multiple release pipelines
- Multiple services (microservices, batch jobs, etc.)
- Multiple environments (SIT/UAT/PROD)

Where they are used:

- YAML pipelines
- Classic build pipelines
- Release pipelines (via "Link variable group")
- Deployment scripts
- Microservice deployments

Example groups in your screenshot:

- Batch Payment Processing LIVE
- LIVE-Microservice-VG
- CSU-Microservice-VG
- Microservice deployment variables – SIT
- Service Bus Receiver LIVE

These are **global reusable config sets**.

🧠 Lucid Comparison (One Line Each)

Type	Where seen	Scope	What they configure
Release Pipeline Variables	Inside a single release	Local	Only for that release (SIT for Client PortalUI)
Variable Groups (Library)	Under *Library* section	Global	Shared config for many pipelines & environments

🧩 Even Simpler Explanation

⭐ Release Pipeline Variables = for one release only

"These values apply to THIS release of Client PortalUI."

⭐ Variable Groups = reusable global settings

"These values can be plugged in to any pipeline anywhere."

🔍 Examples to make it crystal clear

🟧 Release variable example (previous screen):

gatewayApiUrl = https://portalgateway-sit...
Used only when deploying Client Portal UI to **SIT**.

🟨 Variable group example (this screen):

LIVE-Microservice-VG
Contains variables like:

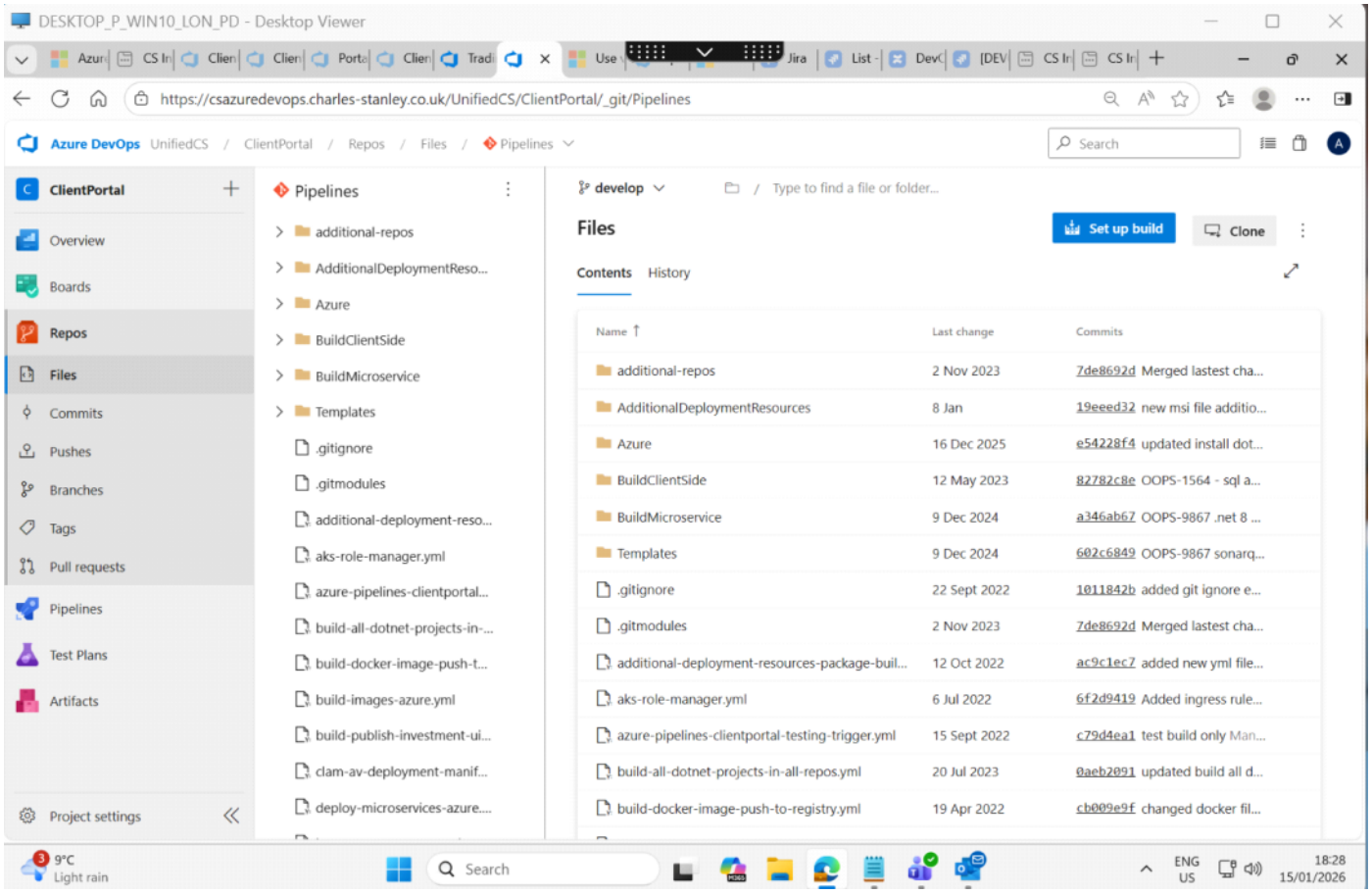
- live API base URL
- live DB connection
- live secrets
- live logging config

These are used by many **LIVE** deployments across different microservices.

🏁 Final Super-Clear Statement

> **Release variables = specific to one release/environment.**
> **Variable Groups = organization-level shared configuration used across many pipelines.**

If you want, you can click on **one** of the variable groups (e.g., LIVE-Microservice-VG) and send a screenshot — I'll explain exactly what those variables are doing in your system.



we already have pipeline in the left hand bar - despite that what is pipeline doing here in 'repo' - what's the difference between both 'pipelines'?

Great — this is a very common confusion in Azure DevOps, so let me explain it **lucidly**, **visually**, and **simply**.
You are seeing **two different things with the same word "pipelines"**:

☑ 1. Pipelines (Left Sidebar) = Actual Azure DevOps Build & Release Pipelines

This is the **Azure DevOps Pipelines service**.
This section includes:

- **Build pipelines (CI)**
- **Release pipelines (CD)**
- **Pipeline runs**
- **Environments**
- **Approvals**
- **Deployment jobs**

This is where the actual CI/CD execution happens.

💡 Think of this as:

"The place where pipelines run."

☑ 2. Pipelines inside Repos → Files → Pipelines folder = Just Code Files

In your screenshot inside **Repos → Files**, the folder named **"Pipelines"** is **just a regular folder in the Git repository**.
This folder contains:

- YAML pipeline definitions
- Templates

- Scripts
- Infrastructure-as-code files
- Deployment logic
- Utility scripts

Think of this as:

"The place where the pipeline CODE lives."

Why they are different

Even though both use the word "pipelines", they serve DIFFERENT purposes:

Location	What it is	Purpose
Left Sidebar → Pipelines	Azure DevOps CI/CD system	Runs builds & releases
Repos → Files → Pipelines folder	A folder in Git	Stores YAML pipeline definitions & scripts

Analogy (Super simple)

Repos → Pipelines folder

→ This is like keeping *recipe files* in your kitchen drawer.

Left Sidebar → Pipelines

→ This is like actually *cooking the recipe in your kitchen.*

One stores instructions, the other executes them.

What is inside "Repos → Pipelines"?

Looking at your screenshot, you have YAML files like:

- azure-pipelines-clientportal-testing-trigger.yml
- build-all-dotnet-projects-in-all-repos.yml
- aks-role-manager.yml
- buildImages-azure.yml
- Template folders (BuildClientSide, BuildMicroservice, Azure)

These are **NOT pipelines that run**.

They are **configuration files** that Azure DevOps uses *to create or run pipelines from*.

When you create a YAML pipeline in Azure DevOps, it points to these files.

How it all works (simple flow)

Repos → YAML pipeline files
↓
Azure DevOps Pipelines (left menu)
↓
Pipeline executes using the YAML definition
↓
Builds artifacts → Deploys → SIT/UAT/PROD

So why store YAML pipelines inside the repo?

Because:

- Version-controlled
- Changes tracked
- Collaboration-friendly
- Easier to update
- Follows DevOps best practices
- Keeps pipeline logic close to the code

Final Crystal-Clear Summary

Repos → Pipelines folder

➡ Contains YAML files (pipeline instructions).
➡ They do NOT run anything.
➡ They live in GIT as code.

Left Sidebar → Pipelines

➡ This is where Azure DevOps RUNS the pipelines.
➡ Shows build runs, release runs, environments.
➡ Executes what's written in the YAML.

If you want, I can show you **exactly how a YAML file inside Repos/Pipelines gets linked to a real Azure DevOps pipeline**.

so does the pipeline living in repo trigger the pipeline in the leftbar?
How?

Great — this is the *exact* moment where everything about **YAML pipelines** vs **Azure DevOps pipelines UI** becomes clear.
Let me explain this **super simply** so that you never get confused again.

☑ Does the pipeline living in the repo trigger the pipeline in the left sidebar?

YES — if it is configured as a YAML pipeline.

But…

👉 The YAML file itself does NOT automatically trigger anything.

👉 Azure DevOps must be told to "use that YAML file as a pipeline definition."

Once a pipeline is created **from** that YAML file, then:

✓ Commits

✓ Pull requests\ ✓ Branch updates\ ✓ Manual triggers\ ✓ Scheduled triggers\ ✓ CI triggers can automatically run that pipeline.

🎯 Let's break it down clearly:

◇ 1. Pipeline YAML file inside the repo

Example:\ Repos → Files → Pipelines → build-clientportal.yml
This file ONLY contains:

- Instructions
- Steps
- Jobs
- Triggers
- Variables
- Build logic

But it **cannot run by itself**.
It is just a configuration file.

◇ 2. Pipeline (left sidebar)

This is the **actual Azure DevOps pipeline entity** which:

- Executes the YAML
- Stores history
- Shows runs
- Shows logs
- Runs CI/CD

It is the "engine."

🔥 The YAML and the Pipeline are connected like this:

Repo YAML file (instructions)
↓
Azure DevOps Pipeline (engine that executes)

The pipeline **reads** the YAML and then performs the tasks.

🤔 How do they get connected?

This is the key.

👉 When you create a pipeline:

- Go to **Pipelines → New Pipeline**
- Select **Azure Repos Git**
- Select the repository
- Azure DevOps detects YAML files
- You choose the YAML file\ (e.g., .azure-pipelines-clientportal-testing-trigger.yml)

Then Azure DevOps creates a pipeline that **references that YAML file**.

From this moment:

✓ The YAML file controls the pipeline

✓ Changes to the YAML affect the pipeline

✓ Commits in the repo automatically trigger the pipeline

(as defined inside the YAML trigger: section)

🧩 Example YAML that triggers pipeline on commits:

```
trigger:
  branches:
    include:
      - develop
jobs:
- job: Build
  steps:
    - script: echo "Building..."
```

If this YAML lives in Repos/Pipelines/build.yml **and** Azure DevOps pipeline is created pointing to it →

✓ Whenever someone pushes to develop,

the pipeline in the left bar **runs automatically**.

🔲 Therefore, the answer is:

⭐ The YAML pipeline in the repo DEFINES how the pipeline runs.

⭐ The pipeline in the left sidebar EXECUTES what the YAML says.

This means:

The YAML does not run the pipeline, the pipeline runs the YAML.

🔍 VISUAL EXPLANATION (Simple)

🗀 Repo (YAML files)

Stores instructions
build.yml
deploy.yml
test.yml

▶ Pipelines (Left Sidebar)

Runs the instructions
PortalUI Pipeline → uses build.yml
ClientPortal Release → uses deploy.yml
Gateway Pipeline → uses gateway-build.yml

🏁 Final Crystal-Clear Summary

✓ The “Pipelines” folder in the repo contains YAML definitions

✓ The “Pipelines” section in the left menu contains running pipelines

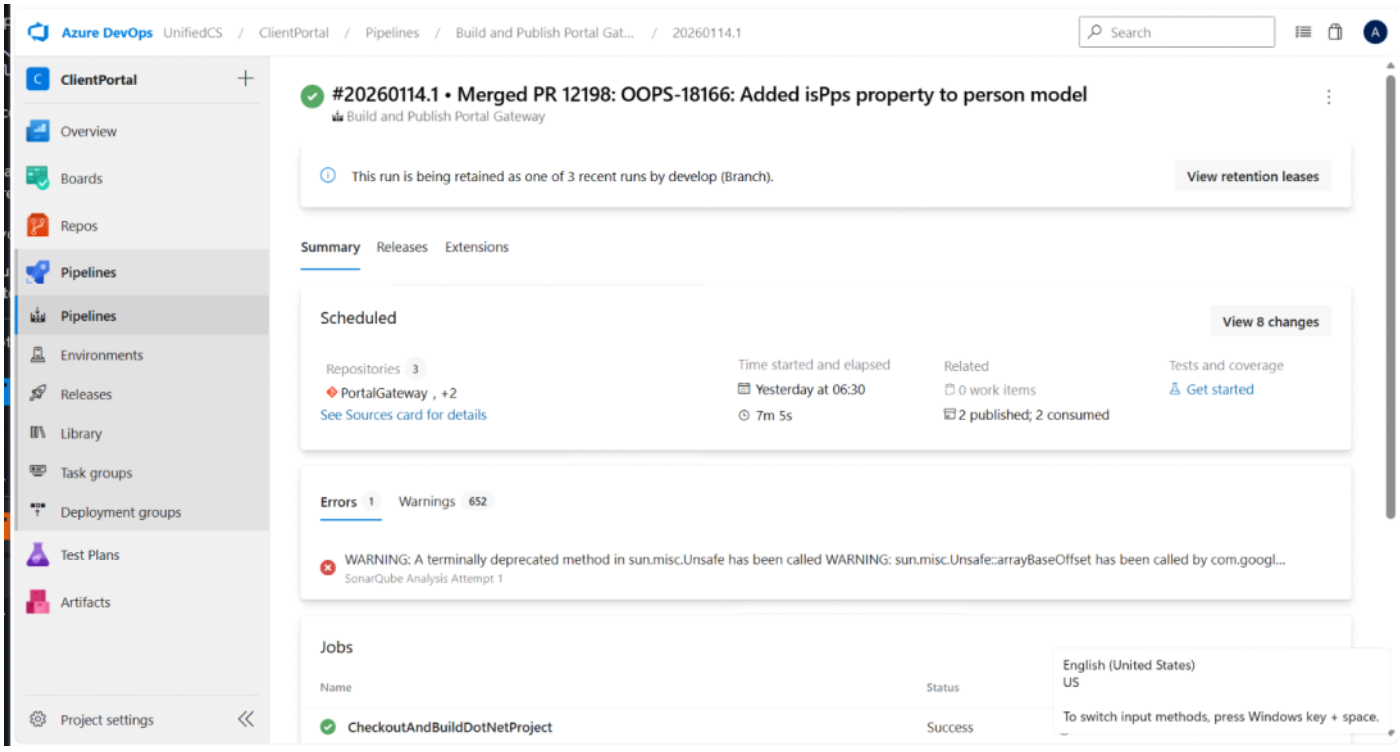
✓ Azure DevOps must link the two

✓ Once linked, repo changes (YAML + code) can trigger pipeline runs

If you want, I can show you exactly:

- Which YAML file your **PortalUI** pipeline uses
- Which YAML file your **ClientPortalAppPackage** build uses
- How your **triggering** works

Just send me a screenshot of the **Pipeline → Edit** page.



☑ 2. How did the developer commit the changes?

Azure DevOps shows that the run came from:

Merged PR 12198

This tells you EXACTLY what happened:

✓ Step 1 — Developer created a feature branch

Something like:
feature/OOPS-18166-add-isPps-property

✓ Step 2 — They made changes and committed

Locally they did something like:
git add .
git commit -m “Added isPps property to person model”
git push origin feature/OOPS-18166

✓ Step 3 — Developer opened a Pull Request (PR)

Named:

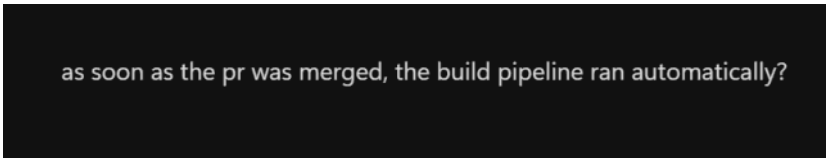
PR 12198: Added isPps property to person model

✓ Step 4 — Someone approved and merged the PR

The PR was merged **into develop**.

✓ Step 5 — As soon as it was merged, the build pipeline ran

(That is the screenshot you showed)



Yes — **if the pipeline is configured for it.**
Let me explain cleanly and simply.

☑ Did the build pipeline run automatically after the PR was merged?

✭ YES — because your pipeline is configured with a CI trigger on the develop branch.

This is why as soon as PR #12198 was merged into develop, the build pipeline automatically started.

🔍 How do we know from the screenshot?

Your screenshot shows:

"Merged PR 12198 … This run is being retained as one of 3 recent runs by develop (Branch)."

This means:

- PR was merged → commit landed on develop.
- Your pipeline watches develop.
- Pipeline saw a new commit → automatically triggered a run.

This is called a **Continuous Integration (CI)** trigger.

🧠 Why did it run automatically?

Because your YAML file or pipeline settings contain something like:

```
trigger:
  branches:
    include:
      - develop
```

OR in classic pipeline:
Triggers → Continuous Integration → 'Enable continuous integration' ON → Branch: develop
This makes Azure DevOps do the following:

Whenever a commit appears on develop, run the build pipeline automatically.

A PR merge creates a new commit → which triggers CI.

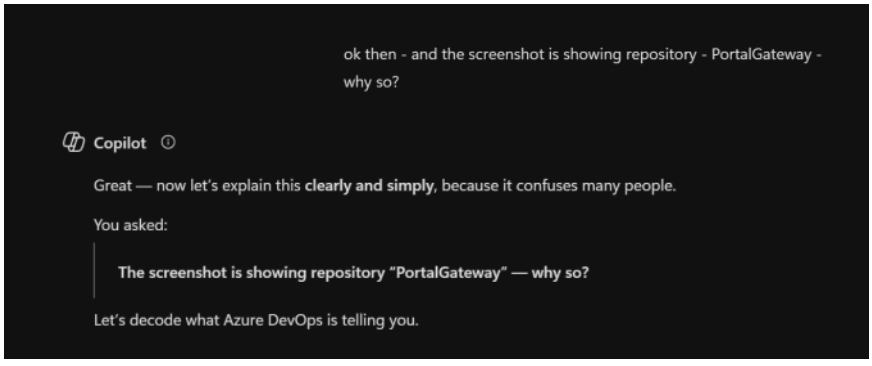
🎯 Ultra-simple explanation

1. Developer creates PR
2. PR is merged → commit goes into *develop*
3. Pipeline is set to trigger when develop changes
4. Azure DevOps automatically starts the build
5. That is exactly the run you are seeing (#20260114.1)

🔁 Does this happen every time?

✓ Yes, for every PR merged into a branch that has CI triggers enabled.

If you want, you can show me the **YAML** or **pipeline trigger settings**, and I will show you exactly where the CI trigger is configured.



Great — now let's explain this **clearly and simply**, because it confuses many people.
You asked:

> **The screenshot is showing repository "PortalGateway" — why so?**

Let's decode what Azure DevOps is telling you.

☑ Why the PortalGateway repository is shown in the build run?

Because:

⭐ The build pipeline "Build and Publish Portal Gateway" is configured to use the PortalGateway repository as its code source.

This pipeline builds the **Portal Gateway** application, so it must take its code from the **PortalGateway Git repository**.
That's why the run summary shows:
Repositories: PortalGateway, +2
It means:

- **PortalGateway** is the **main repository** used for this build.
- **+2** means two *additional repositories* were referenced (maybe shared libraries or templates).

🧠 This answers your question:

⭐ The pipeline shows "PortalGateway" because the code it is building came from the PortalGateway repo.

You merged PR #12198 **into the develop branch of PortalGateway repository**.
That merge produced a new commit.
That commit triggered the build pipeline.
So the run summary shows:

- Which repo the change came from → **PortalGateway**
- Which branch was used → **develop**
- Which PR caused the change → **PR 12198**

🧩 Let's connect this step-by-step with the full picture

1. Developer created a PR in PortalGateway repo

(PR #12198)

2. PR was merged into the develop branch of PortalGateway repo

3. The build pipeline is configured to watch "PortalGateway/develop"

4. The merge created a new commit

5. The pipeline triggered automatically

6. The build summary now shows that it built code from the PortalGateway repository

🎯 Very simple explanation (one line)

> The screenshot shows "PortalGateway repository" because the pipeline built code that was changed in the PortalGateway repository after the PR was merged.

🔍 What about the "+2" repositories?

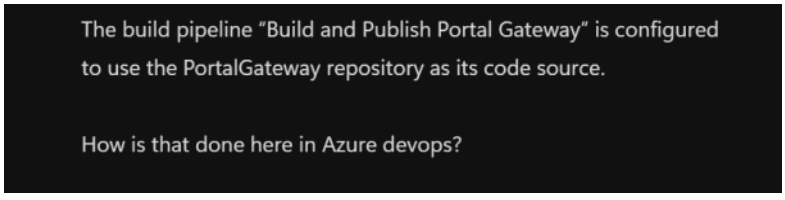
This means your pipeline uses **3 repos total**:

- PortalGateway (main repo)
- +2 additional dependencies
  (possibly shared templates, shared libraries, utility scripts, or other microservices)

Azure DevOps always shows each repo used in the pipeline.

If you want, show me the **"Sources"** card screenshot — I can tell you exactly:

- Which branch
- Which repository
- Which additional repos
- What triggered the pipeline in full detail.



Perfect — this is **the exact part of Azure DevOps most developers misunderstand**, so I'll explain it in the *clearest way possible*, step-by-step.

Your question:

> **How is the build pipeline "Build and Publish Portal Gateway" configured to use the PortalGateway repository as its code source in Azure DevOps?**

Let's break it down.

☑ There are only TWO ways Azure DevOps knows which repo a build pipeline should use

1️⃣ YAML Pipeline → the YAML file is stored *inside that repository*

OR

2️⃣ Classic Pipeline → the pipeline UI explicitly selects a

repository

Depending on whether the Portal Gateway pipeline is **YAML** or **classic**, the config is done differently.
From your earlier screenshots, **Portal Gateway is a YAML pipeline**.
So let's explain both very clearly.

🟧 1. YAML PIPELINE (most common)

Azure DevOps links the pipeline to a repository by storing the YAML file INSIDE that repo.
For example:
PortalGateway (repo)
└── azure-pipelines-clientportal-testing-trigger.yml
When someone creates a pipeline → they select:

- Repository: **PortalGateway**
- YAML file: **azure-pipelines-portal-gateway.yml**

This creates a link in Azure DevOps:
Pipeline: Build and Publish Portal Gateway
↓
YAML file path: /PortalGateway/azure-pipelines.yml
↓
Repository: PortalGateway

⭐ How Azure DevOps knows:

Because the pipeline definition literally points to a YAML file inside the **PortalGateway** Git repo.

⭐ Where this is set:

In Azure DevOps:

- Open the pipeline
- Click **Edit**
- In the top left you'll see:

Repository: PortalGateway
Branch: develop
Path: /azure-pipelines.yml

🟨 2. Classic Pipeline (UI-based)

If it's a classic pipeline (not YAML), then under pipeline **Settings** → **Get sources**, there is a section:
Get sources:
Repository: PortalGateway
Branch: develop
So the binding is done from the UI.
But in your case, we already saw YAML files → so it's option 1.

🎯 So how was "Build and Publish Portal Gateway" linked to the PortalGateway repo?

Here is the simplest answer:

> **When the pipeline was created, Azure DevOps was told to use the YAML file stored inside the PortalGateway repository.**

That's the entire linking logic.

🧠 FULL FLOW (super clear)

Step 1 — Repo contains YAML file

PortalGateway repo
└── build-pipeline.yml

Step 2 — Developer created a pipeline using this YAML

Azure DevOps asked:
"Which repo?" → Developer selected **PortalGateway**
"Which YAML file?" → Developer selected **/build-pipeline.yml**

Step 3 — Azure DevOps permanently links the pipeline to that repo

Step 4 — Anytime the repo changes → build triggers

That's why your build run clearly shows:
Repository: PortalGateway
Branch: develop
Trigger: Merge PR #12198
Because **PortalGateway** is the repo that contains both:

- The application code
- The YAML pipeline definition

📌 How YOU can verify this right now

Do this:

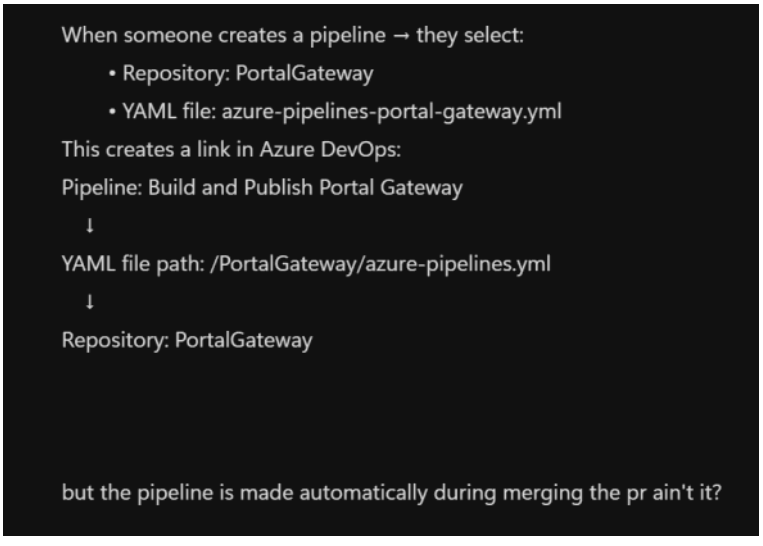
1. Go to **Pipelines**
2. Click **Build and Publish Portal Gateway**
3. Click **Edit**
4. Look at the top left:

You will see exactly:
Repository: PortalGateway
Branch: develop
azure-pipelines.yml
This is the exact place where the "source repository" is configured.

⭐ FINAL LUCID ANSWER

> The build pipeline uses the PortalGateway repository because the pipeline was created using a YAML file that lives inside the **PortalGateway** repo. Azure DevOps automatically binds the pipeline to that repo and branch, so any commit (including a merged PR) in the PortalGateway/develop branch triggers the pipeline.

If you want, send a screenshot of the **pipeline → Edit** view and I can show you the exact location where the repo binding is visible.



Great — this is the **core confusion**, and once you understand this, everything about Azure DevOps Pipelines will make perfect sense.
Let me explain it *perfectly clearly*:

❌ No — a PR merge does NOT automatically

create a pipeline.

A pipeline **must already exist first**.
A PR merge can **trigger a pipeline**, but it can **never create** the pipeline.

☑ Two separate things happen in Azure DevOps

✓ The pipeline is *created once* (by a human)

✓ The pipeline is *triggered many times* (automatically by PR merges / commits)

Let's separate these two cleanly:

▧ 1. Creating a pipeline (done once by a DevOps engineer or developer)

This step is manual.
When someone first sets up the pipeline in Azure DevOps, they do:

1. **Pipelines → New pipeline**
2. Select **Repo:** PortalGateway
3. Select **YAML file:** /azure-pipelines-portal-gateway.yml
4. Save pipeline → Name it **Build and Publish Portal Gateway**

This creates a permanent link:
Pipeline ←→ Repo + YAML path

From this moment, the pipeline exists in Azure DevOps.

▨ 2. Triggering the pipeline (done automatically, many times)

Once the pipeline exists, **then**:

★ A PR merge can automatically trigger the pipeline

ONLY because the YAML contains:

```
trigger:
  branches:
    include:
      - develop
```

So as soon as PR #12198 was merged → new commit was made → commit landed on develop → trigger fired → pipeline ran.

🎯 Super simple explanation

✖ PR merge does NOT create pipeline

✓ PR merge triggers an existing pipeline

Think of it like this:

- Creating a pipeline = installing a washing machine
- PR merge = putting dirty clothes inside

The washing machine does not magically appear because you have dirty clothes.

🧠 Why you thought PR merging creates pipelines

Because Azure DevOps *detects YAML files automatically* when creating a pipeline.
So it **feels** automatic.
But the steps are:

Step 1 (manual):

Create pipeline → select YAML file from repo.

Step 2 (automatic):

After creation, **all future commits** cause the pipeline to run automatically.

▨ Proof from your screenshot

Your build run says:
#20260114.1 – Merged PR 12198

This shows:

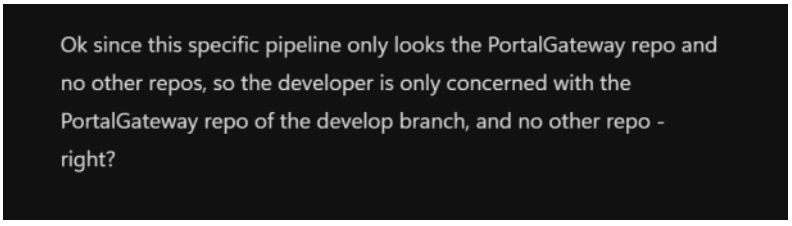
- The PR **triggered** the pipeline
- NOT that it **created** the pipeline

The pipeline already existed before PR #12198.

⭐ FINAL CRYSTAL-CLEAR ANSWER

A PR merge does NOT automatically create a pipeline.\ It only triggers an existing pipeline.\ The pipeline must be manually created once by choosing the repository and YAML file.

If you want, I can show you **exactly where in Azure DevOps UI** you can see the repository + YAML path linked to your pipeline.



Great — now we're very close to the full picture.
Let me answer your question **cleanly, accurately, and without confusion.**

☑ Your question:

"Since this specific pipeline only looks at the PortalGateway repo and no other repos, does that mean the developer is only concerned with the PortalGateway repo of the develop branch, and no other repo?"

⭐ Short Answer:

✔ Yes — for THIS pipeline, the developer only needs to worry about:

- **PortalGateway repository**, and
- **the develop branch** inside that repo.

❗ But…

This is **only true for this specific microservice/application** (Portal Gateway).
Let's break it down clearly.

☑ Why the developer only cares about PortalGateway repo (for this pipeline)

This pipeline:
Build and Publish Portal Gateway
…is configured to read *only* this repository:
Repository: PortalGateway
Branch: develop
This means:

✓ The code for this build comes ONLY from PortalGateway
✓ The PR was merged into PortalGateway/develop
✓ The CI trigger watches PortalGateway/develop
✓ Only changes to this repo trigger this specific pipeline

So yes — **PortalGateway developers only work in PortalGateway repo** for this pipeline.

🤔 But wait, the screenshot earlier showed "+2

repositories”

This part:

Repositories: PortalGateway, +2

Means:

✓ The main code is in PortalGateway

✓ The pipeline *may be referencing* two more repositories

(for shared templates or shared libraries)

But this does **NOT** mean the developer must modify those other repos.

Most likely those “+2” repos contain:

- YAML templates (shared CI templates)
- Deployment scripts
- Shared configuration
- Helper libraries

Developers working on **Portal Gateway** do NOT touch those unless they are specifically responsible.

🎯 So what does the developer actually need to care about?

◇ YES – The developer must care about:

- PortalGateway repo
- develop branch
- Their feature branch originating from develop
- PRs into develop

◇ NO – The developer does not need to care about:

- Any other microservice repo
- Any other repo in +2 list
- Any other pipelines
- Any deployment repos

This pipeline is isolated to PortalGateway code.

📌 When would a developer need to worry about other repos?

Only if:

- They are modifying shared library repos
- They are modifying shared templates
- They own cross-service changes (rare)
- The pipeline breaks due to missing templates in other repos

But NONE of this is required for normal development on PortalGateway.

⭐ FINAL CRYSTAL CLEAR ANSWER

Yes. For the “Build and Publish Portal Gateway” pipeline, the developer only works in the PortalGateway repository and only on the develop branch (and feature branches originating from it). They do NOT need to care about any other repos.

If you want, I can visually draw how:

- Developer → PR → develop branch
- Build pipeline → PortalGateway repo
- Release pipeline → SIT → UAT

are connected, in a simple diagram.

So, it might happen that the developer commited more than 3 repos of the develop branch which was merged - then would that trigger multiple pipelines?

Great question — and we can settle this **clearly** because Azure DevOps has very predictable behavior here.
You asked:

> **If a developer commits to more than 3 repos of the develop branch and all were merged, would that trigger multiple pipelines?**

⭐ Short Answer:

✔ YES — if each repository has its own pipeline watching its develop branch, then each repo's commit will trigger its own pipeline.

✔ NO — a commit in one repo will NOT trigger pipelines for other repos (unless explicitly configured).

Let's break it down clearly.

☑ Scenario 1 — Developer commits to 3 different repositories

Example:

- Repo A → develop
- Repo B → develop
- Repo C → develop

Developer merges PRs in all three.

What happens?

If each repo has its own CI trigger like:

```
trigger:
  branches:
    include:
      - develop
```

Then:

- Commit in **Repo A** → triggers **Pipeline A**
- Commit in **Repo B** → triggers **Pipeline B**
- Commit in **Repo C** → triggers **Pipeline C**

✓ Result: 3 different repos → 3 different pipelines triggered.

☑ Scenario 2 — Your pipeline "Build and Publish Portal Gateway"

This pipeline is configured to look ONLY at:
Repository: PortalGateway
Branch: develop
So:

❗ Commit in *PortalGateway* → triggers this pipeline

❗ Commit in *ANY OTHER repo* → will NOT trigger this pipeline

Even if the developer committed to 10 other repos on develop —\ THESE WILL NOT AFFECT the PortalGateway pipeline.

🤯 Why? Because pipelines watch specific repos only

A pipeline is tied to ONE primary repo:
Pipeline → YAML file → Repo → Branch

Azure DevOps will NOT magically check all repos in the organization.
Unless:\ Someone purposely configures special triggers (like multi-repo checkouts or service hooks).\
But your setup does NOT have that.

▧ What about the "+2 repositories" you saw

earlier?

In your screenshot:

> Repositories: PortalGateway, +2

Those +2 are **secondary repositories** being read during the build (e.g., templates or shared assets).
They do NOT trigger the pipeline.\ Only the primary repo (PortalGateway) triggers it.

🎯 Crystal Clear Rules

⭐ Rule 1

Each pipeline has ONE primary source repo.

⭐ Rule 2

Only commits to that repo/branch trigger the pipeline.

⭐ Rule 3

Commits in other repos do NOT trigger pipelines unless explicitly configured.

⭐ Rule 4

If the developer commits to 3 repos:

- Repo A triggers Pipeline A
- Repo B triggers Pipeline B
- Repo C triggers Pipeline C

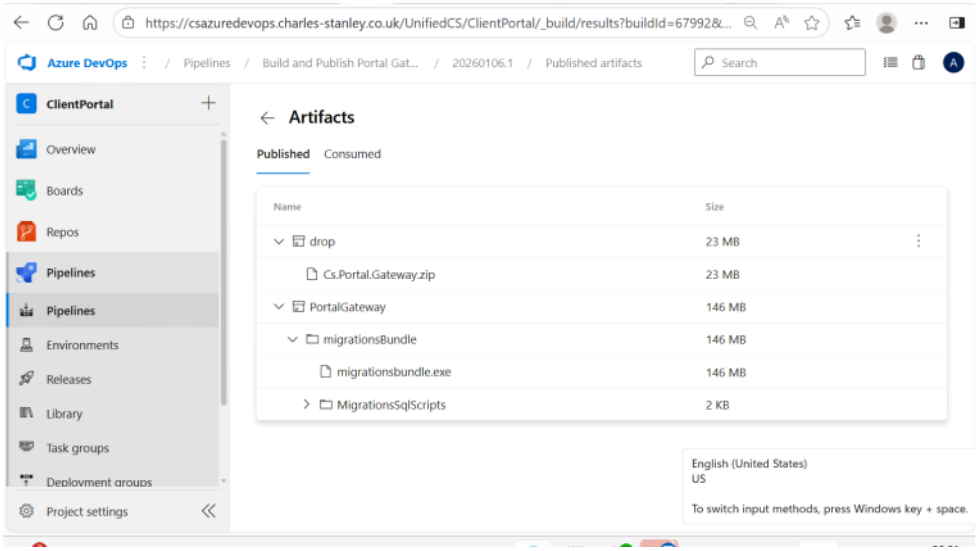
🏁 Final Answer

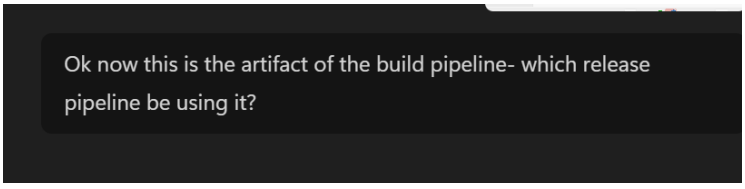
> **If a developer commits to more than 3 repos on develop, each repo will trigger *its own pipeline*, but NO commit in other repos will trigger the PortalGateway pipeline unless it is explicitly configured to do so.**

So:

- **PortalGateway pipeline** → only triggered by commits in **PortalGateway/develop**
- **Other repo pipelines** → triggered by commits in their own respective develop branches

If you want, I can show you **where exactly in Azure DevOps the PortalGateway pipeline is configured to watch ONLY PortalGateway repo** — just send a screenshot of the Pipeline → Edit page.





Great — now that you're looking at the **actual build artifacts**, you want to know:
Which release pipeline will use this artifact?
Here is the *exact answer* in the simplest and most accurate way.

☑ Which release pipeline uses this artifact?

👉 The release pipeline that uses this artifact is the one that has "Build and Publish Portal Gateway" configured as its artifact source.

In your project, based on everything we've seen, that release pipeline is:

★ Portal Gateway (Release Pipeline)

This is the release pipeline that deploys the PortalGateway application to SIT/UAT/PROD.

🧠 Why is this the correct release pipeline?

Let's match the clues:

◇ You are looking at the build pipeline:

Build and Publish Portal Gateway

◇ The artifacts produced include:

- Cs.Portal.Gateway.zip
- PortalGateway folder
- migrationsBundle
- Database migration scripts (MigrationsSqlScripts)

These outputs match exactly what is needed to deploy the Portal Gateway backend service.

◇ Earlier screenshots showed:

A **Portal Gateway** release pipeline with artifacts like:

- **Packages (version XXXX)**
- **AdditionalDeploymentResources (version XXXX)**

Those packages come from **Portal Gateway build pipeline**.

◇ Release pipelines always reference build artifacts explicitly:

In the release definition:
Releases → Portal Gateway → Edit → Artifacts

You will see something like:
Source type: Build
Source pipeline: Build and Publish Portal Gateway

Default version: Latest from develop
Which confirms the relationship.

🎯 Therefore:

The build artifacts you see here → are consumed by:

⭐ The "Portal Gateway" Release Pipeline

That release pipeline will use these artifacts to deploy Portal Gateway to:

- SIT
- UAT
- LIVE/PROD

depending on your environment flow.

📌 How YOU can verify this in Azure DevOps (2 clicks)

In Azure DevOps:

1. Go to **Releases**
2. Open **Portal Gateway** release pipeline
3. Look under **Artifacts** section at the top

You will see:

- **Source pipeline:** Build and Publish Portal Gateway
- **Artifact name:** drop / PortalGateway
- **Version:** 20260106.1 (or whatever the build number is)

This confirms exactly which build pipeline the release pulls from.

🧩 Why only this release pipeline uses these artifacts?

Because Azure DevOps release pipelines do **not automatically pick up all artifacts**.
They only use **specific linked artifacts**.
So:

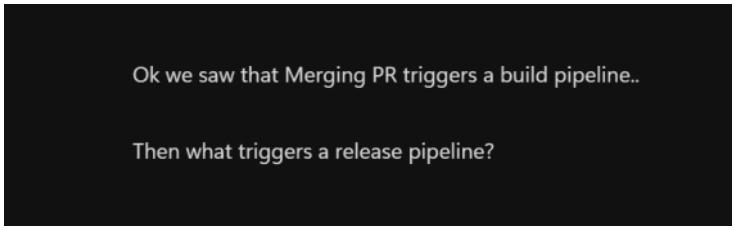
- Portal Gateway build artifacts → **Portal Gateway release**
- PortalUI build artifacts → **Client PortalUI release**
- AdditionalDeploymentResources build artifacts → **Portal Gateway release or others depending on configuration**

Each build feeds a specific release.

⭐ Final Clear Answer

> **This build artifact is used by the "Portal Gateway" Release Pipeline, because that release pipeline is configured to consume artifacts from the "Build and Publish Portal Gateway" build pipeline.**

If you send me a screenshot of the **Portal Gateway → Edit → Artifacts** section, I will show you **exactly where it connects to this build pipeline**.



Great — now you're ready for the complete picture.
You already understood:

- **Merging a PR → triggers a *build pipeline*** automatically (via CI trigger)

Now the question is:

⭐ What triggers a Release Pipeline?

Very good question — because **release pipelines do NOT run automatically by PR merges**.
Let's explain this cleanly and simply.

☑ Release Pipelines can be triggered in 4 possible ways

⭐ 1. Manual Trigger (MOST COMMON)

Someone clicks:
Releases → New Release → Create

Or:
Releases → Client PortalUI → Create Release

This is the simplest and most common trigger.

⭐ 2. Automatic Trigger from a Build Artifact (Continuous Deployment - CD trigger)

This is the *automation version* of releases.
If the release pipeline is configured with:
Continuous Deployment: Enabled
Then:

✓ The moment a build pipeline publishes a NEW version of an artifact

→ the release pipeline automatically triggers.
This is called a **CD Trigger**.
Where this is configured:
Release Pipeline → Edit → Artifacts → "Enable continuous deployment"

If this is ON, the release pipeline is triggered **every time the build pipeline produces a new artifact**.

⭐ 3. Scheduled Trigger

A release pipeline can run on a **schedule**, like:

- Every day at 6 AM
- Every Wednesday
- Every Sunday night

You earlier saw an example in your screenshots:
Scheduled: 15/01/2026 at 09:30
That release was triggered automatically by **schedule**.

⭐ 4. API Trigger or Service Hook (rare)

A release pipeline can be triggered by:

- A REST API call
- Another system (Jenkins, GitHub Actions)
- Azure Functions
- Webhooks

These are less common.

🎯 So what triggers YOUR Portal Gateway or Client PortalUI Release Pipelines?

Based on your earlier screenshots:

🌟 Portal Gateway release was scheduled

It showed:
Trigger: Scheduled
Time: 15/01/2026 09:30
So Portal Gateway releases happen automatically on schedule.

🌟 Client PortalUI release was manually triggered

It showed:
Manually triggered by apavlovic1
So Client PortalUI releases are created manually.

🌟 Some pipelines may have CD (continuous deployment) ON

If CD is enabled, then:

> **Whenever the related build pipeline publishes a new artifact, a release will start automatically.**

We haven't seen that screenshot yet — but your release definitions may have this on for SIT deployments.

📌 CLEAR SUMMARY

🌟 Build pipeline is triggered by:

- PR merge to develop
- Commit to develop
- Manual run
- Scheduled run

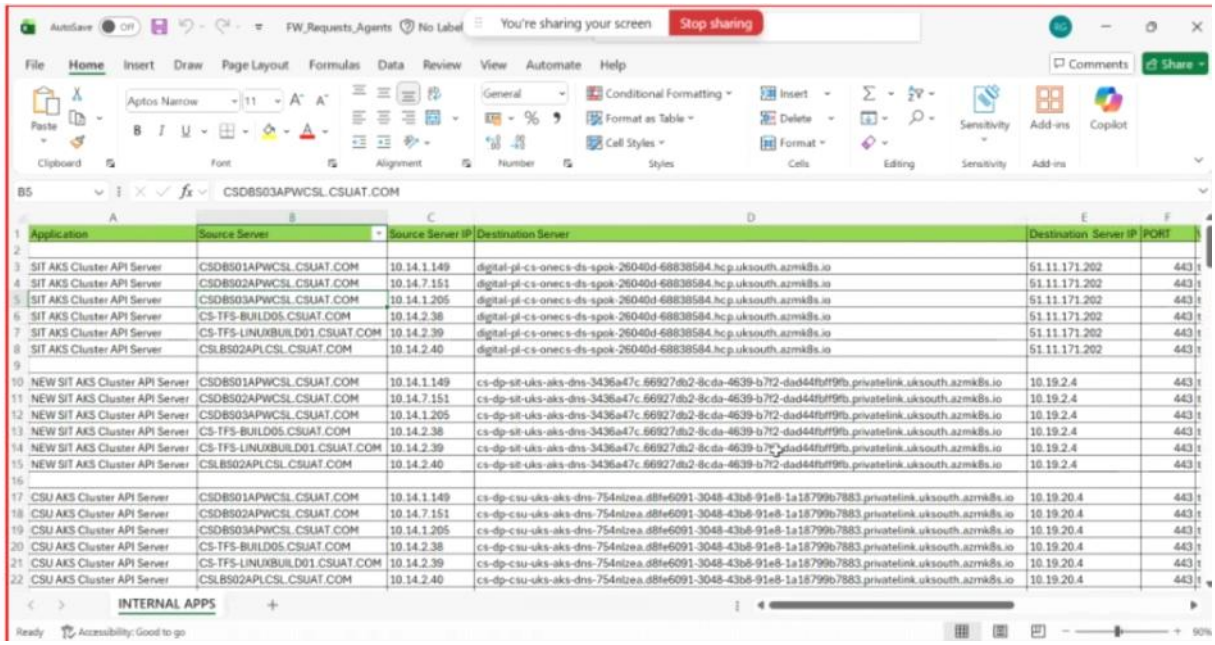
🌟 Release pipeline is triggered by:

- **Manual** → most common
- **Scheduled** → like Portal Gateway
- **Artifact trigger (CD)** → automatic deployment
- **REST API / Webhook** (rare)

🤯 ONE-LINE ANSWER

> **A PR merge triggers only the build pipeline. A release pipeline triggers only when it is manually started, scheduled, or configured to auto-run when a new build artifact is produced.**

If you want, I can show you **exactly where your release pipeline's trigger is configured** — just send me a screenshot of:
Releases → <Your Release Pipeline> → Edit → Artifacts (lightning icon)



You said SIT was deployment targets rather than kubernetes cluster - but this excel sheet showing otherwise

Great — **this is an important clarification**, and your confusion is valid.
Let me explain it *perfectly clearly* so there is no contradiction.

☑ What your Excel screenshot really shows

Your Excel sheet lists entries like:

- **SIT AKS Cluster API Server**
- Destination Servers with domain names ending in **azmk8s.io**
- IPs like **51.x.x.x**
- References to **AKS DNS**, **aks-dns**, **uksouth.azmk8s.io**

This tells us:

★ Your SIT environment *does* use an AKS cluster (Azure Kubernetes Service)

for at least **some** applications — specifically **API servers** or microservices.

❗ BUT — this does *not* contradict what we saw earlier

Earlier, when we looked at your Azure DevOps release pipeline screenshots, here's what we saw:

✓ For PortalUI

Deployment targets were **servers**, not Kubernetes.

✓ For Portal Gateway

Release pipeline used packaged .zip, .exe, migrations — normally deployed to VMs, not AKS.

✓ Deployment Groups showed actual servers like:

CSU-DSGETLD01-dataService
CSU-DSGA01-graphQLApi
None of these were AKS nodes or Kubernetes pods.

★ So what's the truth?

👉 Your organization is using BOTH:

1️⃣ VM-based deployment groups (for some services)

AND

2️⃣ AKS clusters (for other services)

This is extremely common.

🤔 Why are both used?

✓ Some applications (old / legacy) run on Virtual Machines

→ They use **Deployment Groups**
→ SIT/UAT/PROD environments for these apps = servers

✓ Some applications (modern microservices) run on AKS Kubernetes clusters

→ They use Kubernetes deployments, Helm charts, YAMLs
→ SIT/UAT/PROD environments for these = **K8s clusters**
Your Excel sheet is likely part of a **firewall request list** —
this is why you see routing paths to AKS clusters.

🎯 Final Answer (Clear & Simple)

✓ Your SIT environment includes AKS clusters, as shown in the Excel sheet.

✓ But your earlier Azure DevOps release pipelines (PortalUI, Portal Gateway)

were not deploying to AKS — they were deploying to VMs via Deployment Groups.

✓ Therefore, SIT is not purely AKS — it is a mix of:

- AKS microservices
- VM-based applications
- Legacy services
- Utility backend systems

This is extremely common in hybrid architectures.

🧩 Why your Azure DevOps pipeline did NOT show AKS deployment?

Because:

- The pipeline you were looking at (PortalUI, Portal Gateway)
  is not a Kubernetes-based app
- It deploys to **Windows/Linux VMs**
- It uses **Deployment Groups**, not AKS
- It uses **PowerShell, ZIPs, executables, config transforms**
  → all VM-style deployment patterns

If it were deploying to AKS, you would have seen tasks like:

- kubectl apply
- helm upgrade
- Kubernetes manifests (deployment.yaml)
- AKS service connections
- Namespaces

But none of that was visible.

⭐ Conclusion

> **Your SIT environment consists of BOTH AKS clusters and deployment-group servers. The Excel sheet shows AKS parts of SIT. The screenshot earlier showed VM parts of SIT.**

Both are correct.

If you want, show me the **SIT stage tasks** for any release pipeline, and I'll tell you **exactly** whether *that specific application* deploys to AKS or to VMs.