

Azure Devops 3.0

Monday, January 19, 2026 1:07 PM

You don't tell the *agent* what to do—you tell the **pipeline** what to do (via YAML or Classic UI). The agent (from the selected **agent pool**) then **picks up the job** from the queue and executes the **steps/tasks** you defined. You "target" the right agent by specifying the **pool** and (optionally) **demands/tags** to match its capabilities.

How it works (conceptually)

1. **Pipeline defines jobs & tasks** → You describe steps (e.g., restore, build, test, deploy).
2. **Pool selection** → The pipeline points to an **Agent Pool** (e.g., your self-hosted "Default" pool).
3. **Queueing** → The job is queued in that pool.
4. **Agent pulls a job** → An online agent from that pool pulls the job and runs each step.
5. **Capabilities & demands (optional)** → You can constrain which agent is eligible (Windows/Linux, specific tools, tags).

If your pipeline isn't targeting the right pool or the project doesn't have permission to use that pool, your job won't run.

Targeting your self-hosted agent (YAML)

Point the pipeline at your agent pool and define the tasks:

```
# azure-pipelines.yml
trigger:
- main
pool:
  name: Default      # <-- Your self-hosted agent pool name
  # demands:          # (Optional) Require specific capabilities/tags
  #   - Agent.OS -equals Windows_NT
  #   - dotnet
  #   - visualstudio
variables:
  buildConfiguration: 'Release'
stages:
- stage: Build
  displayName: 'Build stage'
  jobs:
    - job: BuildJob
      displayName: 'Build on self-hosted agent'
      steps:
        - checkout: self
        - task: DotNetCoreCLI@2
          displayName: 'Restore'
```

```

inputs:
  command: 'restore'
  projects: './.sln'
- task: DotNetCoreCLI@2
  displayName: 'Build'
  inputs:
    command: 'build'
    projects: './.sln'
    arguments: '--configuration $(buildConfiguration)'
- task: VSTest@2
  displayName: 'Test'
  inputs:
    testSelector: 'testAssemblies'
    testAssemblyVer2: |
      *test.dll
      !*TestAdapter.dll
      !\obj\
    searchFolder: '$(System.DefaultWorkingDirectory)'

- stage: Publish
  dependsOn: Build
  jobs:
    - job: PublishArtifacts
      steps:
        - task: PublishBuildArtifacts@1
          inputs:
            pathToPublish: '$(Build.SourcesDirectory)'
            artifactName: 'drop'

```

Directing to a specific machine (optional)

Add a **tag** to your agent (Agent → Capabilities → *User capabilities*, e.g., role=build or dotnet=true) and demand it:

```

pool:
  name: Default
  demands:
    - role -equals build
Or target a specific OS/tool:

```

```

pool:
  name: Default
  demands:
    - Agent.OS -equals Windows_NT
    - msbuild
    - npm

```

The agent advertises **system capabilities** (like OS, processor) and your **user capabilities** (tags you add). Pipelines' demands must be satisfied by the agent's capabilities.

Using the Classic (GUI) pipeline

1. Pipelines → New pipeline → Classic editor (or edit an existing one).
2. **Agent pool:** Select your self-hosted pool ("Default").
3. Add tasks (e.g., NuGet, VS Build, VS Test, Publish Artifacts).
4. (Optional) Under **Options** → **Demands**, add capability filters (same idea as YAML).

Must-do permissions (often missed)

Even if your agent is online, the project pipeline must be allowed to **use the agent pool**:

- Go to **Project Settings** → **Pipelines** → **Agent Pools**.
- Select your pool (e.g., "Default") → **Security**.
- Ensure your project's **Build Service** (ProjectName Build Service (Org)) has **Use** permission = **Allow**.

Without this, you'll see queue/permission errors when you run.

Typical task libraries you'll use

- **.NET**: DotNetCoreCLI@2, UseDotNet@2
- **Node**: NodeTool@0, Npm@1
- **Java**: Maven@4, Gradle@2
- **Android**: AndroidSigning@3
- **iOS/macOS**: InstallAppleCertificate@2, Xcode@5
- **Containers**: Docker@2
- **Infra**: AzureCLI@2, AzurePowerShell@5, ARM/Bicep
- **Artifacts**: PublishBuildArtifacts@1
- **Tests**: VSTest@2, PublishTestResults@2

On a **self-hosted** agent, make sure the required tools are installed (e.g., .NET SDK, JDK, Node, MSBuild/VS Build Tools), so the agent advertises the right capabilities.

Quick validation checklist

- [] Pipeline pool.name points to your self-hosted pool (e.g., Default).
- [] Agent shows **Online** under that pool.
- [] Project Build Service has **Use** permission on the pool.
- [] Required tools installed on the agent machine (so demands can pass).
- [] Optional: Add **User capabilities** (tags) to route jobs precisely.

Example: route “Windows build” vs “Linux build” to different agents

```
stages:  
- stage: WindowsBuild  
  jobs:  
    - job: Win  
      pool:  
        name: Default  
        demands:  
          - Agent.OS -equals Windows_NT  
          - msbuild  
      steps:  
        - task: VSBuild@1  
          inputs:  
            solution: '/.sln'  
            msbuildArgs: '/p:Configuration=Release'  
- stage: LinuxBuild  
  jobs:  
    - job: Linux  
      pool:  
        name: Default
```

demands:

- **Agent.OS -equals Linux**
- **dotnet**

steps:

- **task: DotNetCoreCLI@2**

inputs:

command: 'build'

projects: './.sln'

If you'd like, tell me your stack (e.g., .NET/Java/Node + OS) and I'll tailor a ready-to-run YAML for your self-hosted agent in the **Default** pool, including the right tasks and demands.

Ok, so the agent pool is mentioned in the pipeline, but the tasks each agent in an agent pool - would that be allocated to any agent randomly by agentpool?

Great follow-up!

TL;DR:

- A **single job** in a pipeline runs **entirely on one agent** (not split across multiple agents).
- Within that job, **tasks/steps run sequentially on the same agent** (unless you explicitly parallelize with multiple jobs or a strategy).
- When a job is queued to a pool, Azure DevOps assigns it to the **first available agent** in that pool that **meets its demands**. That's **not random**—it's essentially **first-eligible, first-served**.

How scheduling works in Azure Pipelines

1) Scope: Job vs. Task

- **Tasks** are the steps within a **job**.
- A **job** is the unit of work that's scheduled to an **agent**.
- Therefore, **all tasks in a job execute on that same agent** in order. They are **not distributed** across multiple agents.

If you want parallelism or distribution, you create **multiple jobs** (e.g., using strategy: matrix or separate jobs/stages). Each job will then be scheduled to any suitable agent in the pool.

2) Agent selection (inside a pool)

When your job is queued to a pool:

1. Azure DevOps looks at your **pool** and any **demands** (e.g., Agent.OS -equals Windows_NT, node, msbuild, or even Agent.Name -equals MyAgent01).
2. It finds **eligible agents** (online + capabilities satisfy demands).
3. It assigns the job to the **first agent that becomes available** among those eligible.
 - o This is generally **FIFO** (first-in, first-out) at the queue level with **availability-based dispatch**.
 - o From your perspective, it's **not random**, but you **don't control** the exact machine unless you constrain it (see next section).

3) How to control which agent gets the job

If you need more control than "any available agent":

- **Demands by capability**

```
pool:  
  name: Default  
  demands:  
    - Agent.OS -equals Windows_NT  
    - dotnet
```

- **Target a specific machine** (not recommended long-term, but possible):

```
pool:  
  name: Default  
  demands:  
    - Agent.Name -equals MyAgent01
```

- **Use user capabilities (tags)**—add custom capabilities on the agent (e.g., role=build, sdk=8.0) and demand them:

```
pool:  
  name: Default  
  demands:  
    - role -equals build  
    - sdk -equals 8.0
```

- **Split work by OS/jobs**—create separate jobs that each target different demands, so they're routed to the appropriate subset of agents.

4) Parallelism & multiple agents

- **Multiple jobs** in the same stage or with strategy: matrix/parallel can run **in parallel**, each on a **different agent** from the pool (subject to your org's **parallel job** limits and pool capacity).

- Example:

```
jobs:  
  - job: Linux  
    pool:  
      name: Default  
      demands:  
        - Agent.OS -equals Linux  
    steps: # ...  
  - job: Windows  
    pool:  
      name: Default  
      demands:  
        - Agent.OS -equals Windows_NT  
    steps: # ...  
  ``
```

5) Common gotchas

- **Tasks fail / “capability not found”** → The agent machine is missing the tool; install it so the agent advertises that capability (or remove the demand).
- **Job sits queued** → Either no agent is online, or none meets the demands, or the **project’s Build Service** doesn’t have **Use** permission on the pool.
- **Want to reuse the same machine across jobs** → Azure Pipelines doesn’t guarantee the **same** agent for different jobs. If you need persistent state or controlled machine, use **deployment jobs with environments**, or package artifacts and pass them between jobs instead.

Mental model

- **Pool** = group of agents
- **Job** = scheduled unit → runs on **one** agent
- **Tasks** = sequential steps within a job
- **Dispatch** = first available **eligible** agent (not random)
- **Control** = use **demands/tags** or split into **multiple jobs**

If you tell me your scenario (e.g., “I have 3 Windows agents; I want UI tests to run only on Agent-UI and builds to run on the others”), I can sketch the exact YAML with the right demands/tags and a clean routing strategy.

Topic:

How to **install and configure a self-hosted Azure DevOps agent as a service.**

1. Prerequisites

Before installing the agent, you need:

✓ **Azure DevOps Organization**

You must belong to an Azure DevOps organization to get the organization URL (e.g., <https://dev.azure.com/yourorg>).

✓ **A machine to host the agent**

This can be:

- A virtual machine
- Your personal laptop
- Any Windows/Mac/Linux machine

✓ A Personal Access Token (PAT)

Used for authentication during agent setup.

✓ The agent installation package

Downloaded from Azure DevOps → Agent pools → New Agent.

2. Downloading the Agent

- Go to **Azure DevOps** → **Organization Settings** → **Agent Pools**.
- Select a pool (e.g., **Default**).
- Open the **Agents** tab → Click **New Agent**.
- Choose OS (Windows/Mac/Linux) → Download the ZIP package.

3. Preparing the Agent Directory

- Create a folder on your machine where the agent will live.
- Extract the downloaded ZIP into that folder.
- You should now see files like config.cmd, run.cmd etc.

4. Configuring the Agent

Steps:

1. Open **Command Prompt as Administrator**.
2. Navigate to the agent folder.
3. Run: config.cmd
4. Provide the following inputs when prompted:

◊ **Azure DevOps URL**

Paste your organization URL.

◊ **Authentication type**

Choose **PAT**.

◊ **Paste the PAT token**

Generate it via:

User Settings → **Personal Access Tokens** → **New Token**

Give it **Full Access** (for demo purposes).

◊ **Agent Pool**

Press Enter to use the default pool.

◊ **Agent Name**

Enter any name (e.g., myagent).

◊ **Work Folder**

Press Enter to use default _work.

◊ **Run agent as a service?**

Choose **Yes**.

◊ **Choose service account**

They selected:

Network Service

◊ **Start service after installation?**

Choose **Yes**.

5. Verify the Agent as a Service

- Open **Services.msc** on the machine.
- You'll see a service named like: Azure Pipelines Agent (OrgName.AgentPool.AgentName)
- Start it if it's not already running.

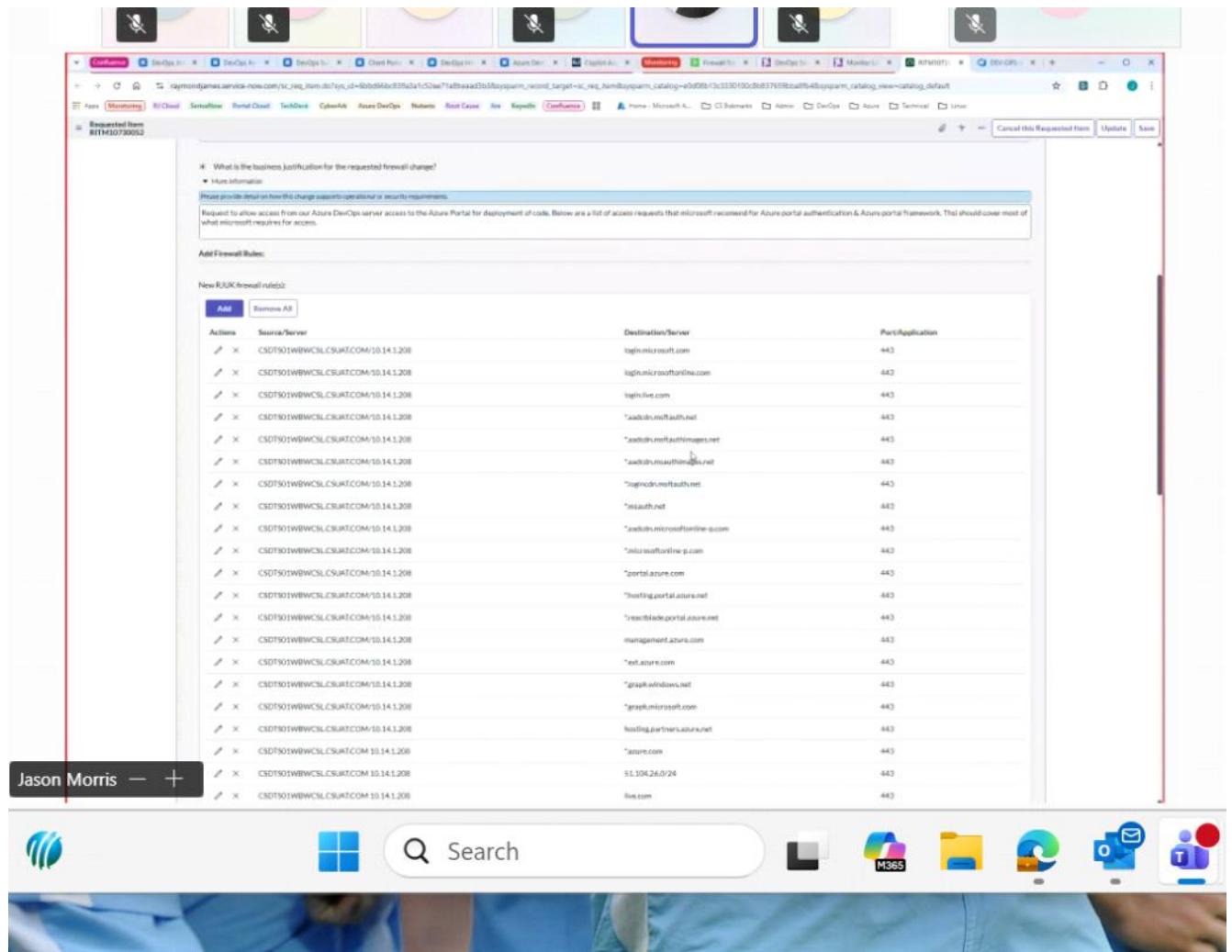
6. Verify in Azure DevOps

- Go to Azure DevOps → Agent Pools → Default → Agents
- You should now see the newly installed agent **Online**.

End result:

Your **self-hosted agent** is installed, configured, running as a service, and ready to execute Azure Pipelines.

If you want, I can also give you a **short 5-line summary**, a **diagram flow**, or a **YAML example** showing how a pipeline uses this agent.



Above is a

A corporate **firewall rule request** page.

✓ The request lists outbound firewall rules:

All the destination URLs you see —

- login.microsoftonline.com
 - aadcdn.msftauth.net
 - portal.azure.com
 - graph.windows.net
 - management.azure.com
 - *.azure.com
 - live.com

— are Azure AD / Microsoft login / Azure Portal endpoints.

This means you (or your team) were trying to ensure that the Azure DevOps build agent or Azure portal access is not blocked by the corporate firewall.

2. What your manager told you & what it means

Your manager said:

Firewall team checked everything — no issues. It might have been DNS.

Then he opened Azure DevOps → Project → Settings → Service connections → saw a **JWT error**.

Here's what that means.

3. What is actually happening? (in simple terms)

Azure DevOps uses **Service Connections** to authenticate with:

- Azure subscriptions
- Service Principals
- Repos
- External services

These service connections internally rely on:

- ✓ **Azure AD authentication**
- ✓ **JWT tokens**
- ✓ **DNS resolution**
- ✓ **TLS/HTTPS connections to Microsoft endpoints**

If any one of these breaks, Azure DevOps may show:

"Failed to obtain JWT token"
"Invalid JWT signature"
"Unable to validate token issuer"

4. How Firewall or DNS issues cause JWT errors

This is the key part you want explained.

◊ **JWT relies on talking to Azure AD endpoints**

To validate a JWT, Azure DevOps must reach:

- login.microsoftonline.com
- login.live.com
- aadcdn.msftauth.net
- management.azure.com
- graph.microsoft.com

If these URLs are blocked, unreachable, or mis-routed → JWT validation fails.

◊ How DNS issues lead to JWT failures

If DNS is misconfigured:

- ✗ **The endpoint might resolve to the wrong IP**
- ✗ **The endpoint might not resolve at all**
- ✗ **It may resolve slowly → token request times out**

Example:

<https://login.microsoftonline.com> → FAILED to resolve

Then Azure DevOps cannot fetch:

- Azure AD metadata
- Token signing keys (public keys)
- Token itself

And the UI shows a **JWT error** because:

It cannot validate or obtain the token.

◊ How firewall issues lead to JWT failures

Even if DNS resolves properly:

- ✗ **Firewall may block outbound port 443**
- ✗ **Firewall may block specific Microsoft domains**
- ✗ **Firewall SSL inspection may break certificate trust**
- ✗ **Firewall may allow DNS but block HTTPS**

Azure AD uses HTTPS-based token exchange.

If HTTPS is blocked or intercepted → JWT validation breaks.

🔒 5. Why JWT error appears on “Service Connections” page

Azure DevOps loads service connections by fetching:

- Azure AD token
- Subscription details
- Service principal metadata

If any request to Microsoft endpoints fails, the UI can't retrieve connection info.

So you see:

! “JWT error while loading service connection”

This doesn't necessarily mean JWT itself is broken —

it means **Azure DevOps could not reach the Azure AD endpoints needed to validate it.**

🌐 6. Putting it all together (simple explanation)

Here's the chain of events simplified:

1. Azure DevOps needs to get a JWT from Azure AD.
2. Azure AD endpoints must be reachable for this.
3. If DNS fails → Azure DevOps cannot resolve those endpoints.
4. If firewall blocks them → token cannot be fetched or validated.
5. Azure DevOps UI → shows **JWT error** because the token request failed.

So the JWT error is actually a **symptom**, not the root cause.

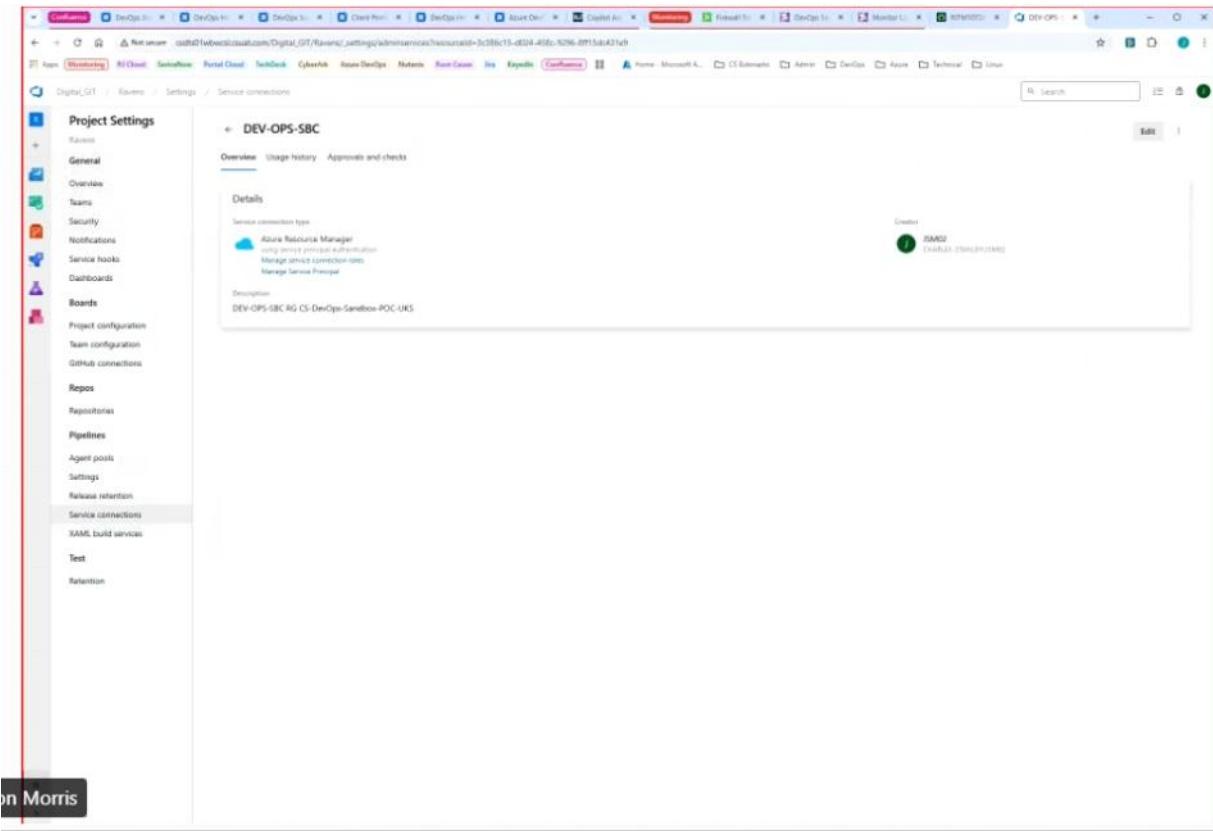
⌚ 7. What is most likely happening in your case?

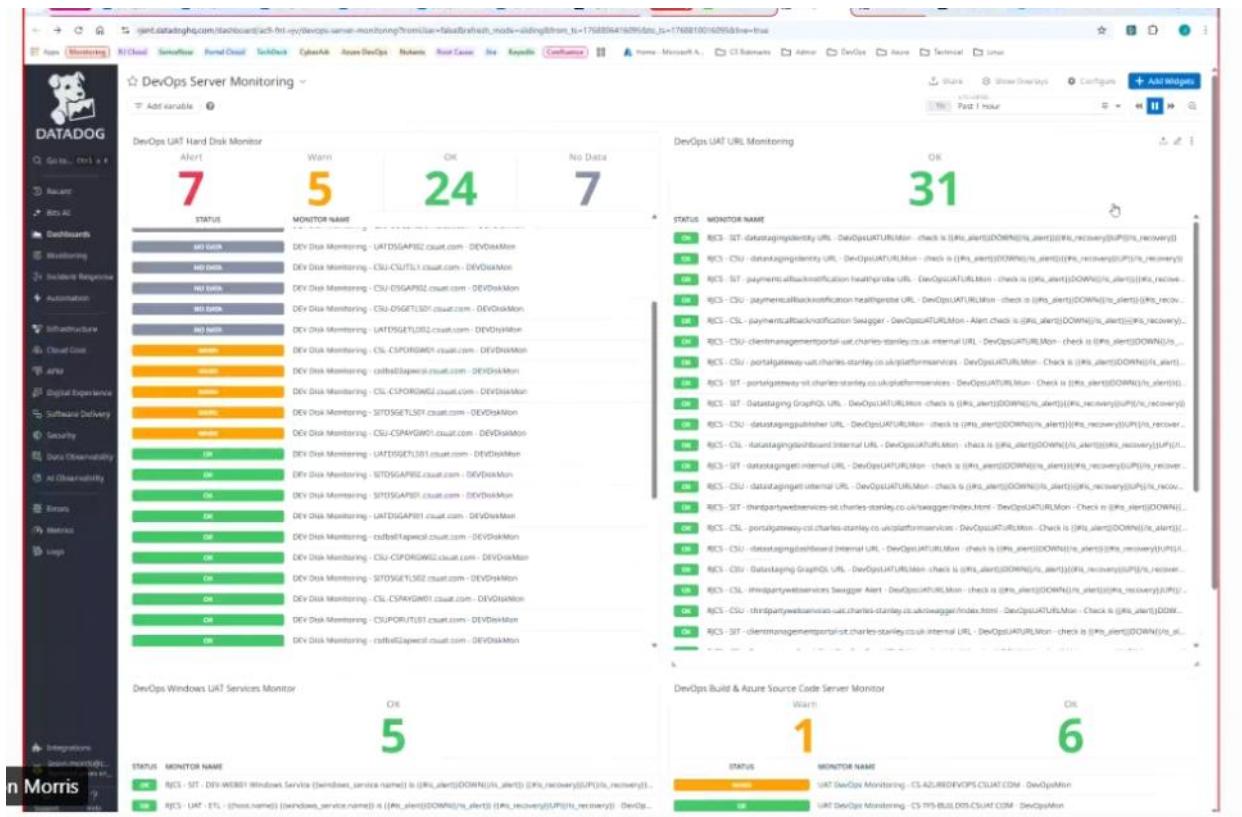
From what you described:

- ✓ **Firewall team found no issues → rules are in place**
- ✓ **Manager suspects DNS → common root cause**
- ✓ **Service Connections page shows JWT error → matches network failure**

Therefore the most logical explanation is:

★ DNS resolution or intermittent network connectivity prevented Azure DevOps from talking to Azure AD → leading to JWT validation failure.





Agents will be here eventually

Here's a **simple, clear explanation** of what "**a corporate firewall rule request page**" means.

What "corporate firewall rule request page" means

A **corporate firewall rule request page** is an **internal web page or portal** used inside a company where employees can **request changes to the company's firewall**.

Think of it like a formal form or ticket system where you say:

"Please allow this server to talk to that server on this port."

Because in large companies, you **cannot directly open ports, cannot allow new URLs, or cannot update firewall rules** yourself — everything goes through security approval.

Why companies use such a page

Large organizations have strict security.

To prevent attacks, they block:

- unknown websites

- unknown IP addresses
- outbound internet traffic
- inbound traffic

So when a DevOps engineer needs Azure DevOps to connect to Azure AD, or when a team needs a server to access a cloud URL, they raise a request like:

Source: build-server.corp.com

Destination: login.microsoftonline.com

Port: 443

Reason: Azure DevOps authentication

A security team or firewall team then reviews and approves it.

How it relates to your screenshot

Your screenshot showed:

- Source server names like CSDTSDVWEXCLSLCARTCOM...
- Destination like login.microsoftonline.com, aadcdn.msftauth.net, portal.azure.com
- Port 443
- Buttons for "Add Firewall Rule", "Save", "Update", etc.

That is exactly what a **corporate firewall rule request tool/page** looks like.

It is not Azure DevOps —

It is **your company's internal web portal to request firewall access that Azure DevOps needs.**

In one line

A corporate firewall rule request page = a form where employees request the security team to open network access needed for company systems (like Azure DevOps) to work.

If you want, I can also break down:

- Why Azure DevOps often requires firewall rule changes
- What the entries in your screenshot actually mean
- How DNS or firewall can cause service connection/JWT issues

Just tell me!

Correct Meaning of Inbound vs Outbound Traffic

Outbound = traffic leaving your machine/server/network

It is **NOT specifically "downloading"**.

Outbound can include:

- Your server calling an API
- Your agent connecting to Azure DevOps
- Your computer sending a request to a website
- Uploading files
- Authentication requests
- Ping, DNS queries, etc.

Outbound = **your machine** → **external destination**

Inbound = traffic entering your machine/server/network

It is **NOT specifically “uploading”**.

Inbound can include:

- Azure DevOps agents receiving tasks
- Clients accessing your web server
- Remote desktop traffic
- API calls hitting your system
- File downloads coming *to you*

Inbound = **external source** → **your machine**

Why “download = inbound” is misleading

When you download a file:

- Your machine **sends an outbound request**
- Server **sends back an inbound response**

So downloading includes **both** outbound and inbound traffic.

Same for uploading:

- You send outbound traffic (file data)
- Server returns inbound acknowledgement

Simple analogy

Think of your PC as a house:

- **Outbound** = you go **out of your house** to buy something
- **Inbound** = someone **comes to your house** (delivery, friend visiting)

Buying something online (download) requires:

- You go out (outbound request)
- Delivery comes in (inbound response)

In the context of your earlier Azure DevOps/firewall questions

When your manager and firewall team talk about:

Outbound rules

They mean:

“Allow this server to call Azure AD, Azure DevOps, login.microsoftonline.com, etc.”

Azure DevOps build agents need **outbound access** to Microsoft URLs.

Inbound rules

These would mean:

“Allow external systems to connect *into* your server.”

Most DevOps pipelines don't need inbound rules unless you're hosting:

- Webhooks
- Git servers

- Artifact feeds
- Self-hosted services

Final simple summary

Term	Real Meaning	NOT Accurate Meaning
Outbound	Traffic leaving your machine/network	downloading
Inbound	Traffic entering your machine/network	uploading

Downloads and uploads involve **both inbound and outbound traffic**.

If you'd like, I can also explain how inbound/outbound rules affect:

- Azure DevOps self-hosted agents
- Service connections
- Firewall approvals
- DNS-related failures

Just tell me!

Azure AD / Microsoft login / Azure Portal endpoints.

What are these 3 things?

Here's a **clear, simple breakdown** of the three things you mentioned:

Azure AD, Microsoft login, and Azure Portal endpoints — what they are and why they matter.

1. Azure AD (Azure Active Directory)

What it is:

Azure AD is Microsoft's **cloud identity and authentication service**.

Think of it as:

"The system that manages user accounts, passwords, sign-in, permissions, and access for Microsoft cloud services."

Azure AD is used for signing into:

- Azure Portal
- Microsoft 365
- Azure DevOps
- Teams
- SharePoint
- Any app using Microsoft authentication

Examples of Azure AD endpoints (URLs):

- <https://login.microsoftonline.com>
- <https://graph.microsoft.com>
- <https://aadcdn.msftauth.net>

These URLs allow Azure AD to:

- Authenticate users
- Issue tokens (like JWT)
- Validate tokens
- Provide user info
- Maintain sessions

2. Microsoft Login Endpoints

These are the **actual URLs** used when you sign in using a Microsoft account (corporate or personal).

Examples:

- <https://login.microsoftonline.com> (corporate Azure AD account)
- <https://login.live.com> (personal Microsoft account)
- <https://aadcdn.msftauth.net> (loads login pages + authentication scripts)

What they do:

When you type your username/password and click **Sign in**, the browser contacts these endpoints to:

- Verify your credentials
- Generate authentication tokens
- Redirect you to the application (Azure DevOps, Azure portal, etc.)

So these login URLs are part of the Azure AD sign-in system.

3. Azure Portal Endpoints

The **Azure Portal** is the web interface where you manage Azure resources.

URL:

<https://portal.azure.com>

There are supporting endpoints like:

- <https://management.azure.com> (API to manage resources)
- <https://graph.windows.net> (old Azure AD graph)
- <https://resources.azure.com>

These endpoints allow you to:

- View/manage VMs
- Manage networking
- Access storage accounts
- Access Azure subscriptions
- Deploy cloud resources

Azure DevOps also calls some of these URLs when connecting to Azure subscriptions via service connections.

How they connect with Azure DevOps

Azure DevOps needs all three:

Azure AD

To authenticate users and service principals.

Microsoft Login URLs

To sign in and validate tokens.

Azure Portal/Management Endpoints

To deploy pipelines, ARM templates, Terraform, resources, etc.

So if any of these endpoints are blocked by firewall or DNS is broken:

- Login fails
- Service connections break
- JWT token errors appear

- Build agents can't authenticate
- Pipelines fail

Ultra-simple summary

Term	Meaning	Why Azure DevOps needs it
Azure AD	Identity system	Authentication & tokens
Microsoft Login URLs	The actual login pages and token issuers	Logging into Azure DevOps
Azure Portal Endpoints	The backend APIs for Azure resources	Pipelines deploy resources

If you want, I can also show:

- How JWT tokens flow between these services
- Why missing DNS causes token validation failures
- Visual diagram of the authentication flow

Just tell me!

Here's a **clear, simple, no-jargon explanation of what a Service Connection is in Azure DevOps**, why it exists, and why it causes JWT / DNS / firewall errors when something breaks.

What is a Service Connection in Azure DevOps? (Lucid Explanation)

A **Service Connection** is basically:

A secure saved login that Azure DevOps Pipelines use to connect to other systems.

Think of it as a **password locker** or **authentication bridge** for your pipelines.

When your pipeline needs to connect to Azure, GitHub, Docker, AWS, an API, etc., Azure DevOps must authenticate somewhere.

You don't want to hardcode passwords or tokens in YAML.

So Azure DevOps provides **Service Connections**.

Why Service Connections exist

Azure DevOps pipelines often need to:

- Deploy resources to **Azure**
- Access **Azure Key Vault**
- Push/pull code from **GitHub**
- Publish to **Docker registries**

- Call **REST APIs**
- Use **Service Principals**
- Connect to external systems (SonarCloud, Jenkins, Artifactory, etc.)

Each of these needs:

- Credentials
- Permissions
- Tokens
- Identity verification

Instead of storing passwords or keys manually, Azure DevOps stores them securely as **Service Connections**.

How a Service Connection works (simple idea)

A Service Connection stores:

- What system you are connecting to
- Credentials (SPN, PAT, certificates, tokens)
- The scope/permissions
- Authentication type (OAuth, Service Principal, PAT, etc.)

When a pipeline runs:

1. Azure DevOps retrieves the stored credentials.
2. It authenticates with Azure AD or external service.
3. It gets a **JWT token**.
4. It uses this token to deploy or interact.

Types of common Service Connections

Azure Resource Manager (ARM)

Uses an **Azure AD Service Principal**.

GitHub / GitHub Enterprise

Uses OAuth or a personal access token.

Docker Registry

Uses username/password or token.

Generic Service Connection

For custom APIs.

Azure Classic

Old ASM model.

AWS service connection

Uses Access Key + Secret.

Why Service Connections rely on Azure AD + JWT + DNS + TLS

Because Azure DevOps needs to:

1) Authenticate using Azure AD

(with URLs like login.microsoftonline.com)

2) Obtain JWT tokens

(needed to authorize deployments)

3) Resolve Microsoft endpoints using DNS

(if DNS fails → endpoint unreachable)

4) Connect over HTTPS/TLS

(if firewall blocks HTTPS traffic → authentication fails)

! When something breaks → you get errors like:

- "Failed to obtain JWT token"
- "Invalid JWT signature"
- "Unable to validate token issuer"
- "Could not reach Azure management endpoint"

These errors appear *inside Service Connections UI* because Azure DevOps cannot authenticate with Azure AD or Microsoft APIs.

⌚ Simplest Explanation Ever

A Service Connection is the saved “login credential” that Azure DevOps Pipelines use to connect to Azure or other services.

If DNS or firewall blocks those authentication URLs, Azure DevOps cannot get the token, so Service Connections show JWT errors.

If you want, I can also show:

- A diagram of how Service Connections authenticate
- What each authentication endpoint does
- A real example of an ARM service connection in JSON
- How build agents use service connections during deployment

Just ask!

7. What is most likely happening in your case?

From what you described:

- ✓ Firewall team found no issues → rules are in place
- ✓ Manager suspects DNS → common root cause
- ✓ Service Connections page shows JWT error → matches network failure

Therefore the most logical explanation is:

➡ **★ DNS resolution or intermittent network connectivity prevented Azure DevOps from talking to Azure AD → leading to JWT validation failure.**

What is meant by DNS resolution here? The domain name uses wrong endpoint? pls say lucidly.

Here is the **clearest, simplest explanation** of what “**DNS resolution issue**” means in your situation — without jargon, and directly tied to your Azure DevOps problem.

✓ What is DNS Resolution? (Lucid Explanation)

DNS resolution means:

Converting a domain name (like login.microsoftonline.com) into the correct IP address.

Just like a phonebook converts someone's name → phone number.

Example:

login.microsoftonline.com → 20.50.2.12 (example IP)

If DNS cannot do this correctly, your computer doesn't know *where* to connect.

✗ So what is a “DNS issue” in your case?

A DNS issue can be any of the following:

1 The domain name cannot be resolved

Meaning:

Your system asks: "What is the IP of login.microsoftonline.com?"

DNS replies: "I don't know."

Azure DevOps → cannot reach Azure AD → JWT fails.

2 DNS returns the wrong IP address

This is what you asked:

"Does it mean the domain name uses the wrong endpoint?"

YES — this is one possibility.

Example (incorrect):

login.microsoftonline.com → 10.0.0.15 (wrong internal IP)

Your system tries to connect to the wrong server → token validation fails.

3 DNS is slow or times out

If the DNS server is overloaded, delayed, or unreachable:

- The request times out
- Azure DevOps cannot get the Azure AD metadata
- JWT cannot be validated

This also produces errors like:

- Failed to obtain JWT token
- Token issuer validation failed

4 DNS is blocking Microsoft cloud endpoints

Some companies use **DNS filtering** for security.

If Microsoft authentication URLs are blocked, you get these failures.

⌚ So what EXACTLY happened in your case?

Based on your manager's hint:

- ✓ Firewall rules are correct
- ✓ No port blocks
- ✓ Service Connections shows JWT errors
- ✓ Problems only with Azure authentication endpoints

Therefore:

- ★ The DNS server in your network likely failed to correctly resolve Microsoft authentication domains (Azure AD endpoints) — causing Azure DevOps to fail token validation.

💧 Why does a DNS issue cause JWT errors?

Azure DevOps needs to contact Azure AD to:

- Download signing keys
- Validate token issuer URL
- Fetch metadata
- Obtain a fresh JWT token

All of these require contacting domains like:

- login.microsoftonline.com
- management.azure.com
- aadcdn.msftauth.net
- graph.microsoft.com

If DNS cannot resolve these → Azure DevOps cannot talk to Azure AD → **JWT validation breaks**.

❖ Ultra-simple analogy

Imagine you need to call your bank.

You know the bank's name ("Azure AD"),
but DNS tells you the phone number.

If DNS gives:

- **No number**
- **Wrong number**
- **Takes too long**
- **A blocked number**

→ You cannot reach the bank.

Azure DevOps needs Azure AD the same way.
If it can't reach it → the JWT token cannot be verified → **errors appear**.

Final 1-sentence answer

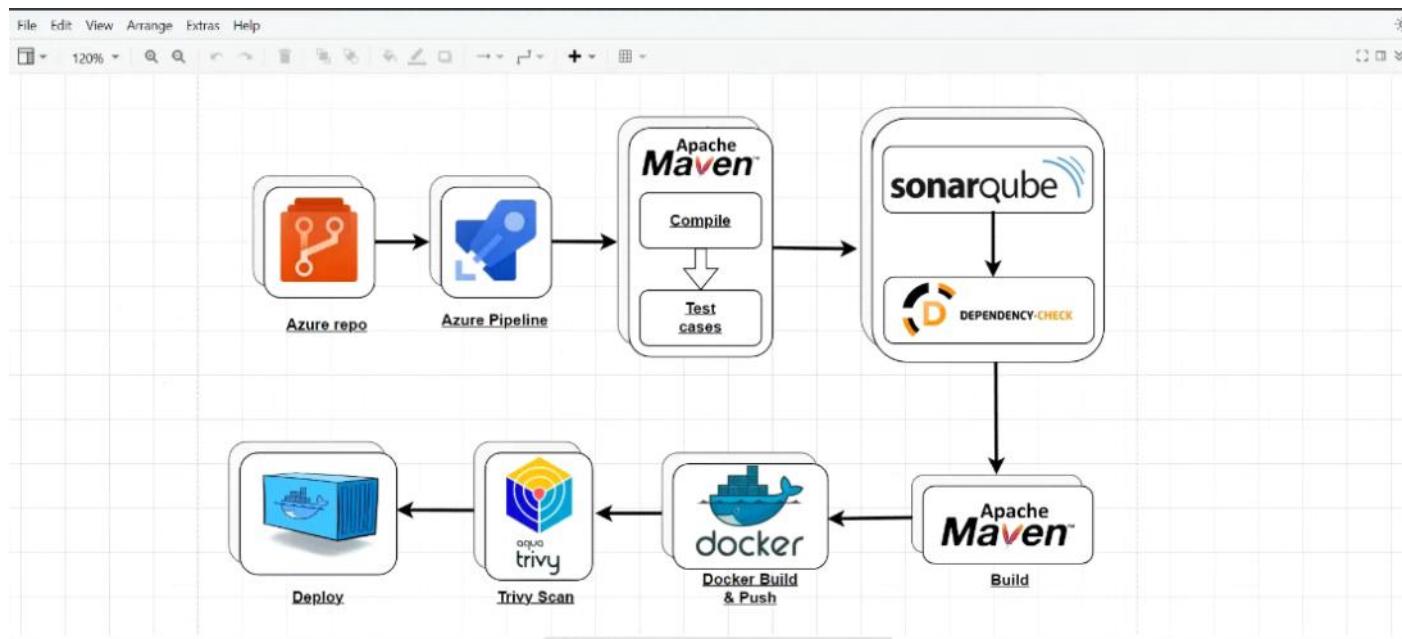
DNS resolution here means “Azure DevOps could not correctly convert Microsoft authentication domain names into the right IP addresses,” causing it to fail when contacting Azure AD and resulting in JWT errors.

If you'd like, I can also:

- Show a small diagram of how DNS failure leads to JWT failure
- Explain how to check DNS resolution on your system
- Explain why firewall issues and DNS issues look similar

Just tell me!

Azure DevOps Real-Time End To End CI/CD Pipeline Project



1. DNS (normal domain → IP resolution)

YES — normally **your DNS server handles everything**.

Example:

You type:

login.microsoftonline.com

Your computer asks your DNS server:

Hey DNS, what is the IP for login.microsoftonline.com?

DNS server replies:

20.190.150.20

This is the **normal** workflow.

It is *automatic* and handled by DNS servers, NOT by you.

2. Hosts File (manual override, special case)

The hosts file is something **your computer checks before DNS**.

It is a **local shortcut rule**.

If your hosts file contains:

10.0.0.5 login.microsoftonline.com

Then your computer will trust this line and **NOT** ask DNS.

So:

- If hosts file has a wrong entry → you go to the wrong IP
- DNS never gets used
- Authentication/login breaks
- Azure DevOps → JWT fails

This is where confusion happens:

Even though "DNS already does the job", the hosts file **overrides** DNS if entries exist.

So the answer to your question:

 No, the hosts file is NOT done by the server.

 It is NOT part of normal DNS.

 The hosts file is a local manual override on your computer/server.

Why your manager mentioned it (in context of Azure DevOps/DNS problem)

Because if the hosts file has even ONE wrong entry like:

login.microsoftonline.com → incorrect IP

portal.azure.com → wrong IP

graph.microsoft.com → wrong IP

Then:

- Azure DevOps cannot talk to Azure AD
- DNS is bypassed
- JWT cannot be validated
- Service Connections fail
- Login fails

Many companies modify hosts files for testing, VPNs, or proxies — so it becomes a common root cause.

Final simplified summary

Component	Who controls it?	Purpose
DNS server	Company/server/internet provider	Normal automatic domain → IP
Hosts file	Your local machine	Manual override; if entries exist, they replace DNS

- ✓ DNS = automatic
- ✓ Hosts file = manual
- ✓ Hosts file overrides DNS