Azure Pipelines

Version Azure DevOps Services · verified 2026-11-02

Azure DevOps CI/CD in YAML — triggers, variables and secrets, stages and conditions, artifacts, environments, and templates.

Contents

Mental model 2

Pipeline anatomy 2

Agents and limits 4

Triggers 5

Variables and secrets 7

Stages, jobs, and conditions 9

Passing values between jobs and stages 11

Artifacts 12

Deployments and environments 14

Templates and parameters 16

Further reading 18

Mental model

A YAML pipeline is compiled into a plan **before anything runs**: templates expand and `${{ }}` expressions resolve first, then stages execute in order, the jobs inside them in parallel, and the steps inside a job one after another. Every job lands on its own agent — on Microsoft-hosted pools a freshly imaged VM — so nothing on disk survives from one job to the next unless you publish it as an artifact. Most surprises come from forgetting one of those two facts: when an expression is evaluated, and where a job runs.

Pipeline anatomy

Level	Runs	Contains
Stage	In sequence by default	Jobs
Job	In parallel by default	Steps, on one agent
Step	In sequence	`script`, `bash`, `pwsh`, `task`

The pipeline lives in `azure-pipelines.yml` in the repo. A single-job pipeline can drop `stages:` and `jobs:` and list `steps:` directly. `script` runs Bash on Linux and macOS and `cmd.exe` on Windows; use `bash` or `pwsh` when the shell matters. A `task` is a versioned, prebuilt step such as `Npm@1`.

```yaml
trigger:
- main

pool:
  vmImage: ubuntu-latest

steps:
- checkout: self
  fetchDepth: 0      # full history
- script: npm ci
  displayName: Install
- script: npm test
  displayName: Test
```

> **Gotcha:** In some organizations, pipelines created since September 2022 shallow-fetch with depth 1 by default, so `git describe` and changelog tools fail. Set `fetchDepth: 0` on the `checkout` step.

Agents and limits

Microsoft-hosted	Limit
Free tier, private project	1 parallel job, 1,800 min/ month
Free tier, per job	60 minutes
Paid parallel job, per job	360 minutes
Disk for sources and output	10 GB

Microsoft-hosted agents give you a clean VM per job with common SDKs preinstalled. A self-hosted agent is a machine you register in an agent pool: it keeps state between runs and has no job-length cap.

`timeoutInMinutes` on a job defaults to 60; setting it to `0` means “the maximum the agent allows.”

```yaml
jobs:
- job: Test
  timeoutInMinutes: 30
  pool:
    vmImage: windows-latest
  steps:
  - pwsh: dotnet test
```

> **Note:** A `timeoutInMinutes` above the hosted maximum has no effect; buy a parallel job or use a self-hosted agent to run longer.

Triggers

Key	Fires on
`trigger`	Push to a matching branch or tag

Key	Fires on
`pr`	PR opened or updated (GitHub, Bitbucket)
`schedules`	A cron time, always in UTC
`trigger: none`	Never on push; manual only

With no `trigger` key, every push to every branch starts a run. Use branch and path filters to cut noise; an `exclude` with no `include` means “include everything else”. `batch: true` folds pushes that arrive during a run into one follow-up run.

```
trigger:
  batch: true
  branches:
    include: [main, releases/*]
  paths:
    exclude: [docs/*]

schedules:
```

```
- cron: "0 3 * * 1-5"   # 03:00 UTC weekdays
  displayName: Nightly
  branches:
    include: [main]
```

> **Gotcha:** The YAML `pr` key is ignored for Azure Repos Git. PR validation there comes from a **Build validation** branch policy on the target branch, and it builds the merge commit.

Variables and secrets

Syntax	Resolved	Use for
`${{ variables.x }}`	Compile time	Structure, template logic
`$(x)`	Runtime, before each task	Task inputs, scripts
`$[ variables.x ]`	Runtime	Conditions, output mapping

Define variables in YAML, in a variable group (shared, optionally backed by Key Vault), or in the pipeline UI. Non-secret variables also reach scripts as environment variables: names are uppercased and dots become underscores, so `build.tag` is `$BUILD_TAG`. Use index syntax for names with dots: `variables['Build.SourceBranch']`.

```yaml
variables:
- group: prod-settings   # holds secret apiKey
- name: region
  value: westeurope

steps:
- bash: ./deploy.sh "$REGION"
  env:
    API_KEY: $(apiKey)   # secrets need this
```

> **Gotcha:** Secret variables are **not** exported to the environment. `$API_KEY` is empty unless you map it with `env:` on the step that needs it.

Stages, jobs, and conditions

Condition	Runs when
(none)	All dependencies succeeded
`succeeded()`	Same, stated explicitly
`failed()`	A dependency failed
`always()`	Even if failed or canceled

`dependsOn` turns parallel jobs into a sequence; `dependsOn: []` lets a stage start without waiting for the one above it. A condition replaces the default “previous succeeded” check rather than adding to it.

```
stages:
- stage: Build
  jobs:
```

```
  - job: Compile
    steps:
    - script: make
- stage: Deploy
  dependsOn: Build
  condition: >-
    and(succeeded(),
    eq(variables['Build.SourceBranch'],
    'refs/heads/main'))
```

> **Gotcha:**
> `condition: eq(variables['Build.SourceBranch'], 'refs/heads/main')` alone runs even after the previous stage failed or was canceled. Always wrap it in `and(succeeded(), …)`.

Passing values between jobs and stages

Consumer	Reference
Later step, same job	`$(stepName.var)`
Job in same stage	`dependencies.J.outputs['s.v']`
Job in later stage	`stageDependencies.S.J.outputs['s.v']`
Stage condition	`dependencies.S.outputs['J.s.v']`

Set a value with the `task.setvariable` logging command and `isOutput=true`, and give the step a `name`. The value exists only for **later** steps; the step that sets it still sees the old one. Map it into the consumer with a `$[ ]` runtime expression.

```
- stage: A
  jobs:
  - job: J
    steps:
    - bash: |
```

```
        echo "##vso[task.setvariable \
        variable=ver;isOutput=true]1.4.2"
      name: s
- stage: B
  dependsOn: A
  variables:
    ver:
$[ stageDependencies.A.J.outputs['s.ver'] ]
```

> **Gotcha:** A stage-level `condition` uses `dependencies.<stage>.outputs['<job>.<step>.<var>']` while variable mapping uses `stageDependencies`. The two are not interchangeable.

Artifacts

Step	Does
`publish: <path>`	Upload a folder as a pipeline artifact
`download: current`	Fetch this run's artifacts

Step	Does
`download: none`	Stop the automatic download

Because each job gets a fresh agent, build output must be published to reach a later job. Downloads land in `$(Pipeline.Workspace)/<artifact>`. A consuming job must depend on the publishing job — artifacts exist only once that job has finished.

```
# Build job
- publish: $(Build.ArtifactStagingDirectory)
  artifact: web

# Later job (dependsOn the build job)
- download: current
  artifact: web
- script: ls $(Pipeline.Workspace)/web
```

> **Gotcha:** Deployment jobs download every artifact automatically; regular jobs download nothing. A plain `job:` that "can't find the build" is missing its `download:` step.

Deployments and environments

Piece	Role
`deployment:` job	Records history against an environment
`environment:`	Target; owns approvals and checks
`strategy: runOnce`	Simplest strategy; also `rolling`, `canary`
Service connection	Credential for Azure or another system

Approvals and checks are not written in YAML: an environment owner adds them to the environment (or to a service connection) under **Approvals and checks**,

and the stage waits until they pass. For Azure, create an Azure Resource Manager service connection using **workload identity federation** — no stored secret to rotate — and pass its name as `azureSubscription`.

```yaml
- deployment: DeployWeb
  environment: production
  strategy:
    runOnce:
      deploy:
        steps:
        - task: AzureCLI@2
          inputs:
            azureSubscription: prod-arm
            scriptType: bash
            scriptLocation: inlineScript
            inlineScript: az webapp list -o table
```

> **Gotcha:** A deployment job does not check out the repo. Add `- checkout: self` if the deploy step needs scripts or manifests from source.

Templates and parameters

Form	Reuses
`- template: x.yml` under `steps:`	A list of steps
`- template: x.yml` under `jobs:`	Jobs or stages
`extends: template:`	The whole pipeline (enforceable)
`x.yml@repo`	A template from another repo

Parameters are typed, resolved at compile time, and read as `${{ parameters.name }}`. Top-level parameters in the main file become a form when someone runs the

pipeline manually. `${{ if }}` inserts or removes YAML before the run starts, which `condition:` cannot do.

```yaml
# templates/test.yml
parameters:
- name: extended
  type: boolean
  default: false

steps:
- script: npm test
- ${{ if eq(parameters.extended, true) }}:
  - script: npm run test:e2e
```

> **Tip:** Pin cross-repo templates with `ref: refs/tags/v1` on the repository resource, so a change to the shared template cannot break every pipeline at once.

Further reading

- Azure Pipelines documentation — concepts, how-tos, and ecosystem guides.
- YAML schema reference — every key, with types and defaults.
- Expressions — functions and status checks for `condition:`.
- Approvals and checks — gating environments and service connections.
- Task reference — inputs for every built-in task.