



CD/CD foundation for JavaScript engineers

CONTINUOUS INTEGRATION TOOLS FOR JS ENGINEERS

EPAM Systems Inc.

CI/CD foundation for JavaScript engineers

Learn & Development

www.epam.com

Table of Contents

CONTINUOUS INTEGRATION TOOLS FOR JS ENGINEERS	1
1. GIT HOOKS AND HUSKY	3
1.1. INTRODUCTION	3
1.2. INTERACTION WITH GIT HOOKS USING HUSKY	3
1.3. COMMITLINT AND COMMITIZEN	4
2. ESLINT AND PRITIER	9
2.1. INTRODUCTION	9
2.2. INSTALLATION AND USAGE	9
2.3. CONFIGURATION	10
2.4. MOST POPULAR CONFIGURATIONS AND PLUGINS READY TO USE	11
2.5. USAGE ESLINT WITH PRETTIER	12
3. STATIC CODE ANALIZERS	14
3.1. INTRODUCTION	14
3.2. SONARQUBE	16
3.3. SNYK CODE	17

1. GIT HOOKS AND HUSKY

1.1. INTRODUCTION

When doing JavaScript development it is common to have linting and test tasks in your package.json file. It is easy to forget to run these common tasks before pushing code and this can result in a broken build (if using continuous integration) or the next developer will see the issues when pulling down the latest code.

A way to work around this problem is to use git hooks that will allow you to hook into the git workflow to run tasks. Git hooks are not easy to share across a development team as git hooks are located in the .git/hooks folder.

Git hooks are scripts that Git executes before or after events such as: **commit**, **push**, and **receive**. Git hooks are a built-in feature - no need to download anything. Git hooks are run locally. Every Git repository has a .git/hooks folder with a script for each hook you can bind to. You're free to change or update these scripts as necessary, and Git will execute them when those events occur.

1.2. INTERACTION WITH GIT HOOKS USING HUSKY

Installing [Husky](#) is as simple as an npm install from the terminal in your project. You can use it to **lint your commit messages**, **run tests**, **lint code**, etc... when you commit or push. Husky supports [all](#) Git hooks.

```
npm install husky --save-dev
```

After installation [Husky](#) need to initialize git repository and enable git hooks.

```
npx husky install
```

To automatically have Git hooks enabled after install, edit package.json

```
npm set-script prepare "husky install"
```

You should have the following script in package.json:

```
// package.json
{
  "scripts": {
    "prepare": "husky install"
  }
}
```

Let's move on and create first hook. To add a command to a hook or create a new one, use **husky add <file> [cmd]** (don't forget to run husky install before).

```
npx husky add .husky/pre-commit "npm test"
```

```
git add .husky/pre-commit
```

If there is a need to run multiple tasks you can use the package [npm-run-all](#) to combine all the tasks into one task and reference that task in your Husky git hook.

1.3. COMMITLINT AND COMMITIZEN

Commitlint checks if your commit messages meet the [conventional commit format](#). In general the pattern mostly looks like this:

type(scope?): subject #scope is optional; multiple scopes are supported (current delimiter options: "/", "\" and ",")

Real world examples can look like this:

- chore: run tests on travis ci
- fix(server): send cors headers
- feat(blog): add comment section

Types other than feat and fix MAY be used in your commit messages.

Why Use Conventional Commits

- Automatically generating CHANGELOGs.
- Automatically determining a semantic version bump (based on the types of commits landed).

- Communicating the nature of changes to teammates, the public, and other stakeholders.
- Triggering build and publish processes.
- Making it easier for people to contribute to your projects, by allowing them to explore a more structured commit history.

Getting started

```
# Install commitlint cli and conventional config:
npm install --save-dev @commitlint/{config-conventional,cli}

# For Windows:
npm install --save-dev @commitlint/config-conventional @commitlint/cli

# Configure commitlint to use conventional config:
echo "module.exports = {extends: ['@commitlint/config-conventional']}" >
  commitlint.config.js
```

To lint commits before they are created you can use Husky's commit-msg hook:

```
# Install Husky v6
npm install husky --save-dev

# or
yarn add husky --dev

# Activate hooks
npx husky install

# or
yarn husky install

# Add hook:
npx husky add .husky/commit-msg 'npx --no-install commitlint --edit "$1"'
```

Check the [husky documentation](#) on how you can automatically have Git hooks enabled after install for different yarn versions.

Detailed Setup instructions

- [Local setup](#) - Lint messages on commit with husky
- [CI setup](#) - Lint messages during CI builds

When you commit with Commitizen, you'll be prompted to fill out any required commit fields at commit time. No more waiting until later for a git commit hook to run and reject your commit

(though [that](#) can still be helpful). No more digging through [CONTRIBUTING.md](#) to find what the preferred format is. Get instant feedback on your commit message formatting and be prompted for required fields.

Installation is as simple as running the following command (if you see EACCES error, reading [fixing npm permissions](#) may help):

```
npm install -g commitizen
```

Using the command line tool

If your repo is [Commitizen-friendly](#) simply use `git cz` or just `cz` instead of `git commit` when committing. You can also use `git-cz`, which is an alias for `cz`.

Alternatively, if you are using **NPM 5.2+** you can [use npx](#) instead of installing globally:

```
npx cz
```

or as an npm script:

```
...  
"scripts": {  
  "commit": "cz"  
}
```

When you're working in a Commitizen friendly repository, you'll be prompted to fill in any required fields and your commit messages will be formatted according to the standards defined by project maintainers.

```
→ ng-poopy master X git add .  
→ ng-poopy master X git cz
```

All commit message lines will be cropped at 100 characters.

```
? Select the type of change that you're committing: (Use arrow keys)  
> feat:      A new feature  
  fix:       A bug fix  
  docs:      Documentation only changes  
  style:     Changes that do not affect the meaning of the code  
            (white-space, formatting, missing semi-colons, etc)  
  refactor:  A code change that neither fixes a bug or adds a feature  
  perf:     A code change that improves performance  
  test:     Adding missing tests  
  chore:    Changes to the build process or auxiliary tools  
            and libraries such as documentation generation
```

If your repo is NOT Commitizen friendly then git cz will work just the same as git commit but npx cz will use the [streamich/git-cz](https://github.com/streamich/git-cz) adapter. To fix this, you need to first [make your repo Commitizen-friendly](#)

Making your repo Commitizen-friendly

For this example, we'll be setting up our repo to use [AngularJS's commit message convention](#) also known as [conventional-changelog](#).

First, install the Commitizen cli tools:

```
npm install commitizen -g
```

Next, initialize your project to use the cz-conventional-changelog adapter by typing:

```
commitizen init cz-conventional-changelog --save-dev --save-exact
```

Or if you are using Yarn:

```
commitizen init cz-conventional-changelog --yarn --dev --exact
```

Note that if you want to force install over the top of an old adapter, you can apply the --force argument. For more information on this, just run commitizen help.

The above command does three things for you.

1. Installs the cz-conventional-changelog adapter npm module
2. Saves it to package.json's dependencies or devDependencies
3. Adds the config.commitizen key to the root of your **package.json** as shown here:

```
...  
"config": {  
  "commitizen": {  
    "path": "cz-conventional-changelog"  
  }  
}
```

Alternatively, commitizen configs may be added to a .czrc file:

```
{  
  "path": "cz-conventional-changelog"  
}
```

This just tells Commitizen which adapter we actually want our contributors to use when they try to commit to this repo.

2. ESLINT AND PRITIER

2.1. INTRODUCTION

ESLint is an open source JavaScript linting utility. Code [linting](#) is a type of static analysis that is frequently used to find problematic patterns or code that doesn't adhere to certain style guidelines. There are code linters for most programming languages, and compilers sometimes incorporate linting into the compilation process.

JavaScript, being a dynamic and loosely-typed language, is especially prone to developer error. Without the benefit of a compilation process, JavaScript code is typically executed in order to find syntax or other errors. Linting tools like ESLint allow developers to discover problems with their JavaScript code without executing it.

The primary reason ESLint was created was to allow developers to create their own linting rules. ESLint is designed to have all rules completely pluggable. The default rules are written just like any plugin rules would be. They can all follow the same pattern, both for the rules themselves as well as tests. While ESLint will ship with some built-in rules to make it useful from the start, you'll be able to dynamically load rules at any point in time. ESLint is written using Node.js to provide a fast runtime environment and easy installation via [npm](#).

2.2. INSTALLATION AND USAGE

Prerequisites: [Node.js](#) (^12.22.0, ^14.17.0, or >=16.0.0) built with SSL support. (If you are using an official Node.js distribution, SSL is always built in.)

You can install ESLint using npm:

```
npm install eslint --save-dev
```

You should then set up a configuration file:

```
./node_modules/.bin/eslint --init
```

After that, you can run ESLint on any file or directory like this:

```
./node_modules/.bin/eslint yourfile.js
```

2.3. CONFIGURATION

After running `eslint --init`, you'll have a `.eslintrc` file in your directory. In it, you'll see some rules configured like this:

```
{
  "rules": {
    "semi": ["error", "always"],
    "quotes": ["error", "double"]
  }
}
```

The names "semi" and "quotes" are the names of [rules](#) in ESLint. The first value is the error level of the rule and can be one of these values:

- "off" or 0 - turn the rule off
- "warn" or 1 - turn the rule on as a warning (doesn't affect exit code)
- "error" or 2 - turn the rule on as an error (exit code will be 1)

The three error levels allow you fine-grained control over how ESLint applies rules (for more configuration options and details, see the [configuration docs](#)).

How to use eslint with TypeScript

When it comes to linting TypeScript code, there are two major linting options to choose from: [TSLint](#) and [ESLint](#). TSLint is a linter that can only be used for TypeScript, while ESLint supports both JavaScript and TypeScript.

In the [TypeScript 2019 Roadmap](#), the TypeScript core team explains that **ESLint has a more performant architecture than TSLint** and that they will **only be focusing on ESLint** when providing editor linting integration for TypeScript. For that reason, I would recommend using ESLint for linting TypeScript projects.

Installation

Make sure you have [@typescript-eslint/parser](#) installed:

```
npm i --save-dev typescript @typescript-eslint/parser
```

Then install the plugin:

```
npm i --save-dev @typescript-eslint/eslint-plugin
```

It is important that you use the same version number for `@typescript-eslint/parser` and `@typescript-eslint/eslint-plugin`.

Note: If you installed ESLint globally (using the `-g` flag) then you must also install `@typescript-eslint/eslint-plugin` globally.

Usage

Add `@typescript-eslint/parser` to the `parser` field and `@typescript-eslint` to the `plugins` section of your `.eslintrc` configuration file, then configure the rules you want to use under the `rules` section.

```
{
  "parser": "@typescript-eslint/parser",
  "plugins": ["@typescript-eslint"],
  "rules": {
    "@typescript-eslint/rule-name": "error"
  }
}
```

You can also enable all the recommended rules for our plugin. Add `plugin:@typescript-eslint/recommended` in `extends`:

```
{
  "extends": ["plugin:@typescript-eslint/recommended"]
}
```

Note: Make sure to use `eslint --ext .js,.ts` since by [default](#) eslint will only search for `.js` files.

2.4. MOST POPULAR CONFIGURATIONS AND PLUGINS READY TO USE

There are many configuration packages in the ecosystem - these packages that exist solely to provide a comprehensive base config for you, with the intention that you add the config and it gives you an opinionated setup. A few popular all-in-one-configs are:

- Airbnb's ESLint config - [eslint-config-airbnb-typescript](#).
- Standard - [eslint-config-standard-with-typescript](#).

To use one of these complete config packages, you would replace the `extends` with one of these, for example:

```
module.exports = {
  root: true,
  parser: '@typescript-eslint/parser',
  plugins: [
    '@typescript-eslint',
  ],
  extends: [
    - 'eslint:recommended',
    - 'plugin:@typescript-eslint/recommended',
    + 'airbnb-typescript',
  ],
};
```

There are many plugins, each covering a different slice of the JS development world. Below are just a few examples, but there are [many, many more](#).

- Jest testing: [eslint-plugin-jest](#)
- ESLint comment restrictions: [eslint-plugin-eslint-comments](#)
- Import/export conventions : [eslint-plugin-import](#)
- React best practices: [eslint-plugin-react](#) and [eslint-plugin-react-hooks](#)
- NodeJS best practices: [eslint-plugin-node](#)

Every plugin that is out there includes documentation on the various rules they offer, as well as recommended their own recommended configurations (just like we provide). You can add a plugin and a recommended config as follows:

```
module.exports = {
  root: true,
  parser: '@typescript-eslint/parser',
  plugins: [
    '@typescript-eslint',
    + 'jest',
  ],
  extends: [
    'eslint:recommended',
    'plugin:@typescript-eslint/recommended',
    + 'plugin:jest/recommended',
  ],
};
```

2.5. USAGE ESLINT WITH PRETTIER

Prettier is an opinionated code formatter. It enforces a consistent style by parsing your code and re-printing it with its own rules that take the maximum line length into account, wrapping code when necessary.

If you use [prettier](#), there is also a helpful config to help ensure ESLint doesn't report on formatting issues that prettier will fix: [eslint-config-prettier](#).

Using this config is as simple as adding it to the end of your extends:

Note: [Since version 8.0.0 of eslint-config-prettier](#), all you need to extend is 'prettier'. That includes all plugins. Otherwise for <8.0.0, you need include 'prettier/@typescript-eslint'.

```
module.exports = {
  root: true,
  parser: '@typescript-eslint/parser',
  plugins: [
    '@typescript-eslint',
  ],
  extends: [
    'eslint:recommended',
    'plugin:@typescript-eslint/recommended',
+   'prettier',
  ],
};
```

3. STATIC CODE ANALIZERS

3.1. INTRODUCTION

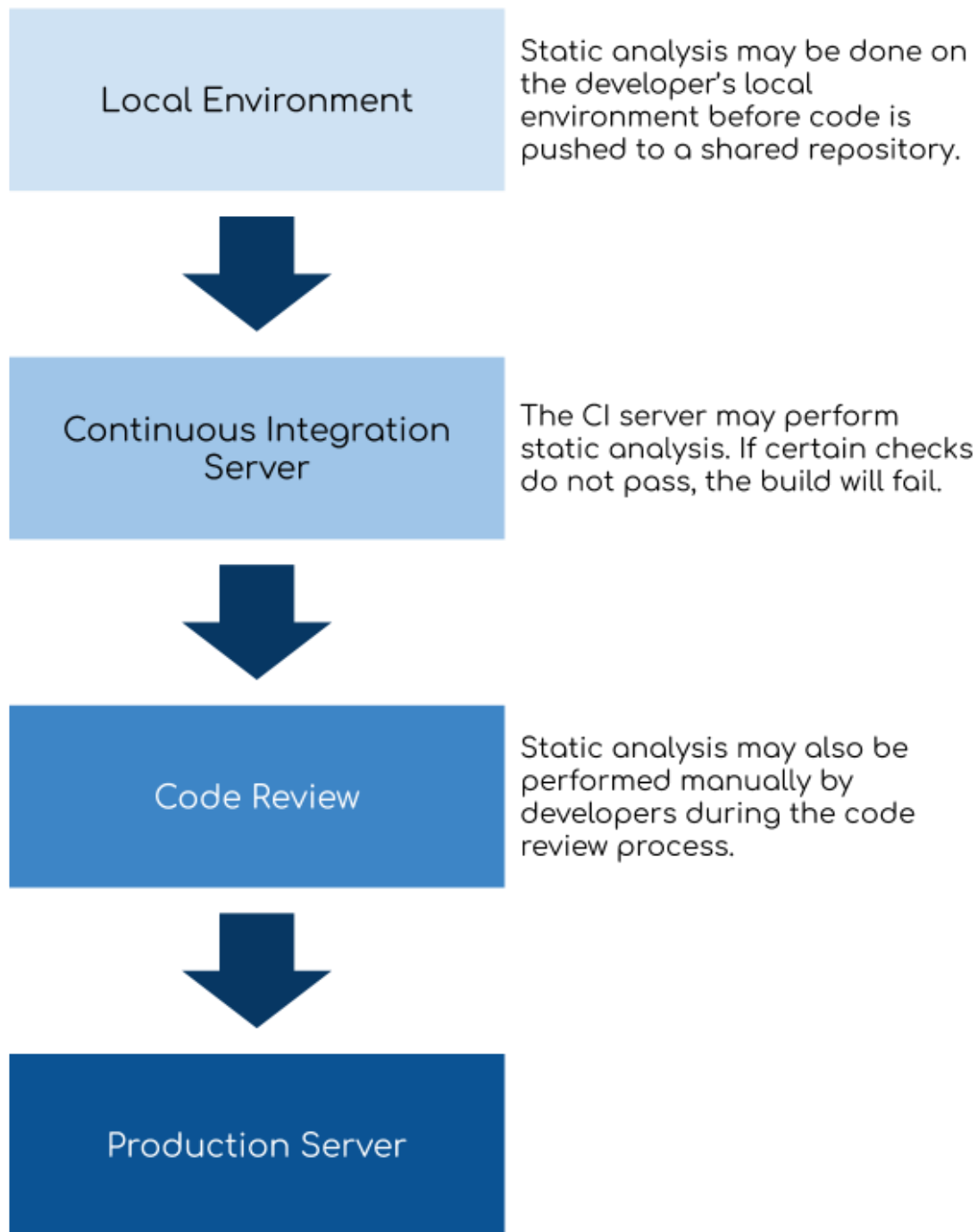
Static analysis is the process of verifying that your code meets certain expectations without actually running it. Unlike [unit and integration testing](#), static analysis can be performed on raw source code without the need for a web server or build process.

Static analyzers typically parse your code and turn it into what is known as an [abstract syntax tree](#). This tree is then traversed and the pieces are checked based on the rules dictated by the static analyzer. Most static analyzers also include a way for developers to write their own custom rules, but that varies from tool to tool.

Static analysis is most commonly used to:

- Ensure consistent style and formatting
- Check for common mistakes and possible bugs
- Limit the complexity of code
- Verify type consistency
- Minimize security risks
- Keep third-party dependencies up to date

In a dynamically interpreted language such as JavaScript, developers must decide when and how they want to run static analysis on their code. I've most commonly seen static analysis run on each developer's machine before they push changes (as a [Git pre-commit hook](#)) as part of a continuous integration server's workflow or as part of every code review.



No matter when or how static analysis happens, the goal remains the same: to help make code more consistent, maintainable, and correct. It won't replace automated or manual testing, but it may catch errors that other quality assurance tools miss.

3.2. SONARQUBE

If you are trying to set up Sonarqube for your project you must follow the steps below. First, you need to download Sonarqube from the [official website](#). The easiest way is run Sonarqube in docker container. Once Sonarqube is ready and running on your local machine or remotely we can move on to the next step and install sonar-scanner:

```
npm install sonar-scanner --save-dev
```

Create a file called “sonar-project.properties” in your project in root directory and add the following configuration

```
sonar.host.url=http://localhost:9000
sonar.login=admin
sonar.password=admin
sonar.projectKey=demo-app
sonar.projectName=demo-app
sonar.projectVersion=1.0
sonar.sourceEncoding=UTF-8
sonar.sources=src
sonar.exclusions=**/node_modules/**
sonar.tests=src
sonar.test.inclusions=**/*.spec.ts
sonar.typescript.lcov.reportPaths=coverage/lcov.info
```

Add a script called sonar to your package.json, you can give any name

```
"scripts": {
  "sonar": "sonar-scanner"
}
```

Finally, run the below command to integrate the coverage with the Sonar server.

You will need Java JDK 11 to run sonar scanner, here is useful link that may help to resolve this issue: <https://medium.com/beingcoders/setup-sonarqube-with-angular-project-in-6-minutes-57a87b3ca8c4>

Link to Github with example of integration: <https://github.com/EPAM-JS-Competency-center/shop-angular-cloudfront/tree/sonar-integration>.

3.3. SNYK CODE

Setting up process of Snyk is relatively simple, first you need to install snyk CLI:
<https://docs.snyk.io/features/snyk-cli/install-the-snyk-cli>.

Once you installed the Snyk CLI, you can verify it's working by running:

```
snyk -version
```

Snyk CLI depends on [Snyk.io](https://snyk.io) APIs. Connect your Snyk CLI with [Snyk.io](https://snyk.io) by running:

```
snyk auth
```

If you are already in a folder with a supported project, start by running:

```
snyk test
```

Or scan a Docker image by its tag with [Snyk Container](#):

```
snyk container test ubuntu:18.04
```

Good practices of Snyk's application within CI/CD pipeline are described [here](#).

Typical stages of adoption

Developer teams typically adopt Snyk in the following stages:

[Expose vulnerabilities](#) (snyk monitor)

[Use Snyk as a gatekeeper](#) (snyk test)

[Continuous monitoring](#) (snyk test + snyk monitor)

Stage 1: Expose vulnerabilities (snyk monitor)

This is a typical initial approach, using Snyk results to expose to your team vulnerabilities during the development process, which increases visibility of these vulnerabilities amongst your team.

When you first implement Snyk in your pipeline, we recommend you use only the **snyk monitor** command, or if you use one of Snyk's CI plugins, to configure the plugin to not fail the build. This is because all projects are vulnerable, and after you set Snyk to fail the build, every build will fail

because of Snyk, which may cause problems with your team being quickly overwhelmed with failure messages.

Using **snyk monitor** to expose results will provide information, without disrupting processes.

Stage 2: Use Snyk as a gatekeeper (snyk test)

This next approach prevents the introduction of new vulnerabilities (sometimes known as "stopping the bleeding").

After your teams understand the vulnerabilities in their applications, and develops a process for fixing them early in the development cycle, you can configure Snyk to fail your builds, to prevent introducing vulnerabilities into your applications..

Add **snyk test** to your build or enable the fail functionality to make Snyk fail your builds, providing the results output to the console. Your Devs or DevOps teams can use the results to decide whether to stop or continue the build.

Stage 3: Continuous monitoring (snyk test + Snyk monitor)

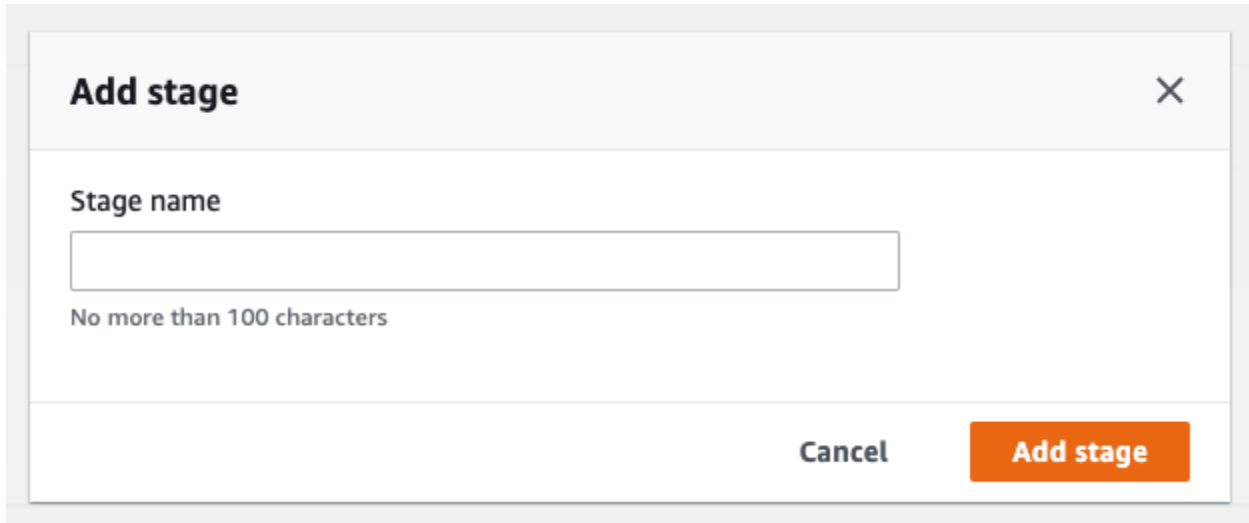
After you configure Snyk to fail the build when vulnerabilities are detected, you can now configure Snyk to send a snapshot of your project's successful builds to Snyk for ongoing monitoring.

To do this, configure your pipeline to run **snyk monitor** if your **snyk test** returns a successful exit code.

[AWS CodePipeline integration](#)

Step 1: Add stage

At any point after the Source stage, you can add a Snyk scan stage, allowing you to test your application at different stages of the CI/CD pipeline.



Add stage ✕

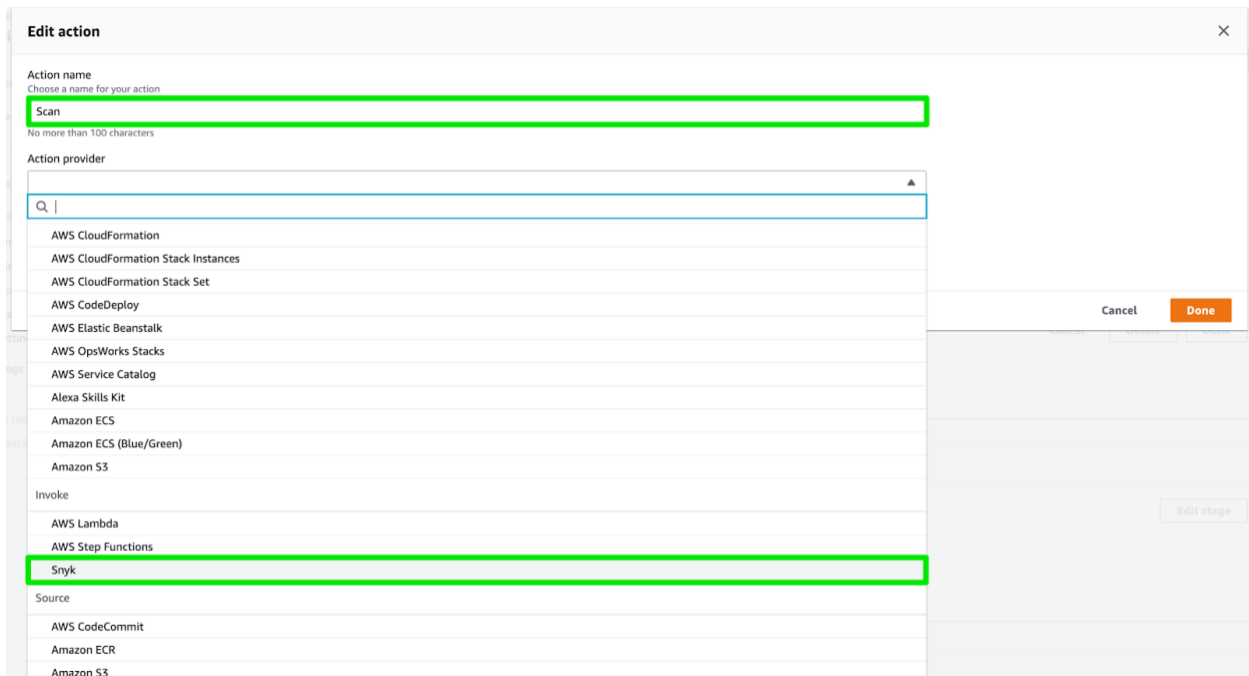
Stage name

No more than 100 characters

Cancel **Add stage**

Step 2: Add action group

Click Add an Action Group to open the Edit Action window:



Edit action ✕

Action name
Choose a name for your action

No more than 100 characters

Action provider

- AWS CloudFormation
- AWS CloudFormation Stack Instances
- AWS CloudFormation Stack Set
- AWS CodeDeploy
- AWS Elastic Beanstalk
- AWS OpsWorks Stacks
- AWS Service Catalog
- Alexa Skills Kit
- Amazon ECS
- Amazon ECS (Blue/Green)
- Amazon S3
- Invoke
- AWS Lambda
- AWS Step Functions
- Snyk**
- Source
- AWS CodeCommit
- Amazon ECR
- Amazon S3

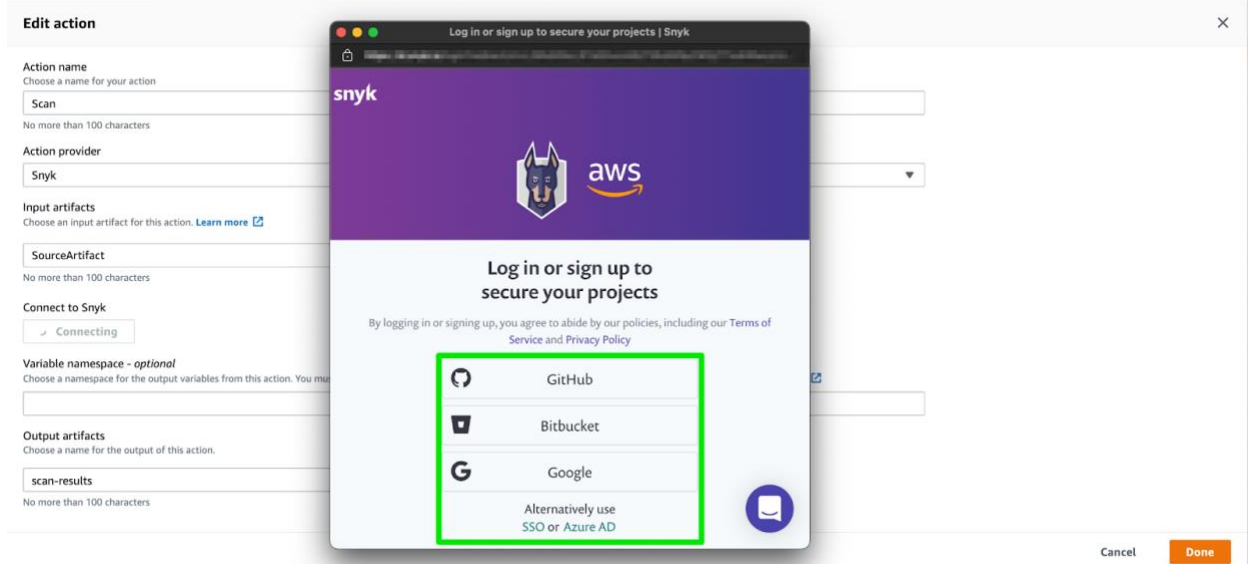
Cancel **Done**

Edit stage

Name the action, then select Snyk as the Action Provider. Click Connect with Snyk to begin the connection process.

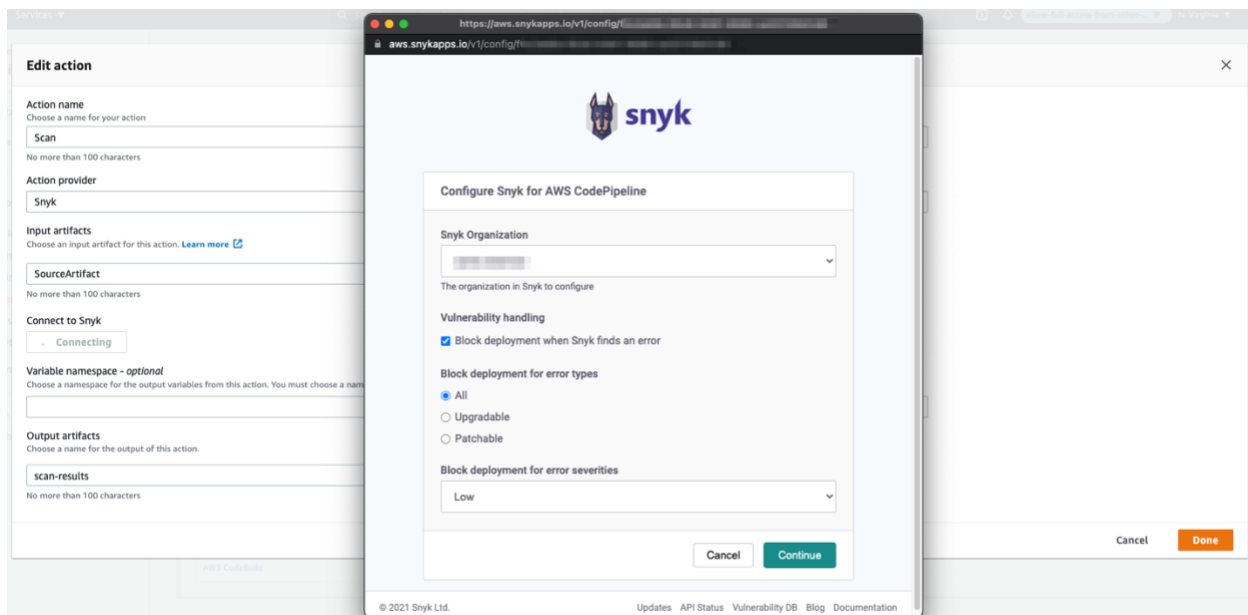
Step 3: Connect to Snyk

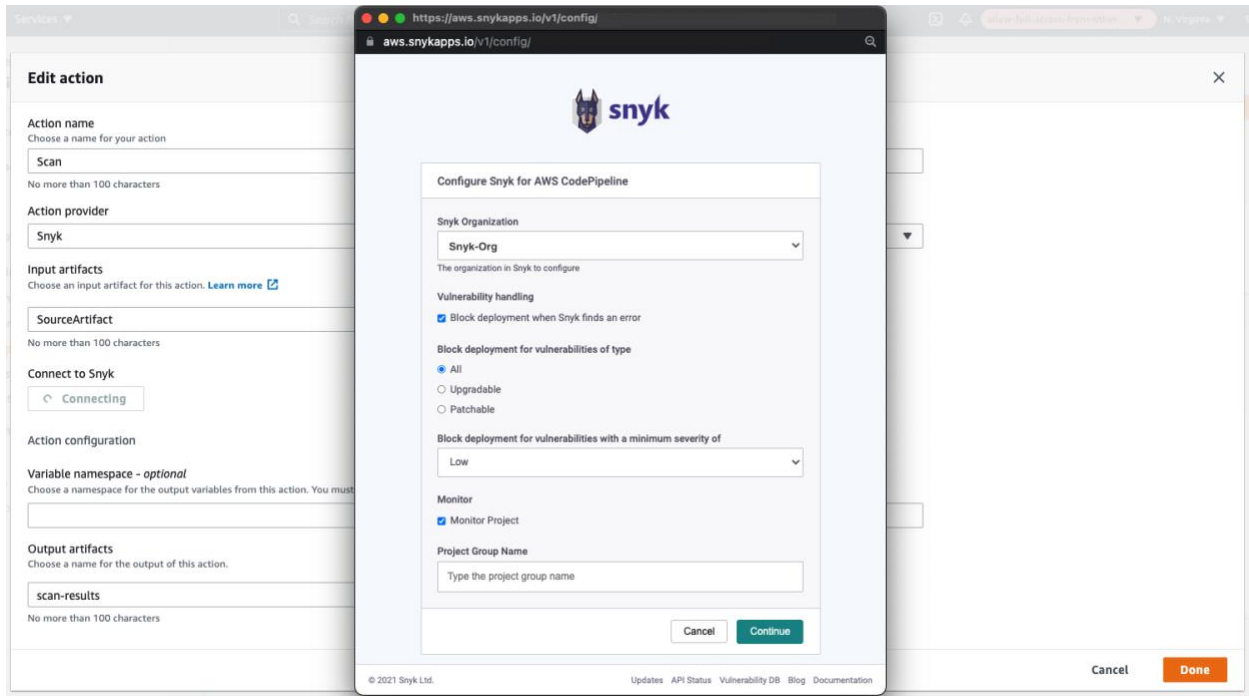
Select how you would like to authenticate with Snyk to give AWS CodePipeline permission to begin scanning your open source code.



Step 4: Configure settings

The following options are available for configuration:





1. **Snyk Organization:** Select the Snyk organization where findings reports are saved.
2. **Vulnerability handling:** Select to fail a pipeline if a vulnerability is found. If Fail on issues is selected, the pipeline will fail depending on the sub-options selected. The sub options available are:
 1. **All:** Selecting all fails when there is at least one vulnerability that can be either upgraded or patched.
 2. **Upgradable:** Selecting upgradable fails when there is at least one vulnerability that can be upgraded.
 3. **Patchable:** Selecting patchable fails when there is at least one vulnerability that can be patched.
3. **Block deployment for vulnerabilities of type**
4. **Block deployment for vulnerabilities with a minimum severity of:**
(low | medium | high | critical) Only report vulnerabilities of provided level or higher.
5. **Snyk Organization:** Select the Snyk organization where findings reports are saved.

6. **Vulnerability handling:** Select to fail a pipeline if a vulnerability is found. If Fail on issues is selected, the pipeline will fail depending on the sub-options selected. The sub options available are:

- **Block deployment for vulnerabilities of type:**
 - **All:** Selecting all fails when there is at least one vulnerability that can be either upgraded or patched.
 - **Upgradable:** Selecting upgradable fails when there is at least one vulnerability that can be upgraded.
 - **Patchable:** Selecting patchable fails when there is at least one vulnerability that can be patched.
- **Block deployment for vulnerabilities with a minimum severity of:**
(low|medium|high|critical) Only report vulnerabilities of provided level or higher.

7. **Monitor project:** Select the monitor project checkbox to monitor projects from the AWS CodePipeline. The project snapshot will be created under the Snyk organization selected. Whenever you select the Monitor Project option please note that the Project Group Name will be required. This is to prevent any unintentional project overrides due to naming conflicts.

8. **Project Group Name:** Enter the project group name for your projects.. This is the same as using [--remote-repo-url](#) when using the CLI. The field does not allow any spaces in the names.

Name the action, then select Snyk as the Action Provider. Click Connect with Snyk to begin the connection process.

You can view scan results in the AWS CodePipeline console, by clicking **Details** in the Scan stage:

Click **Link to execution details** to view your detailed vulnerability report.

