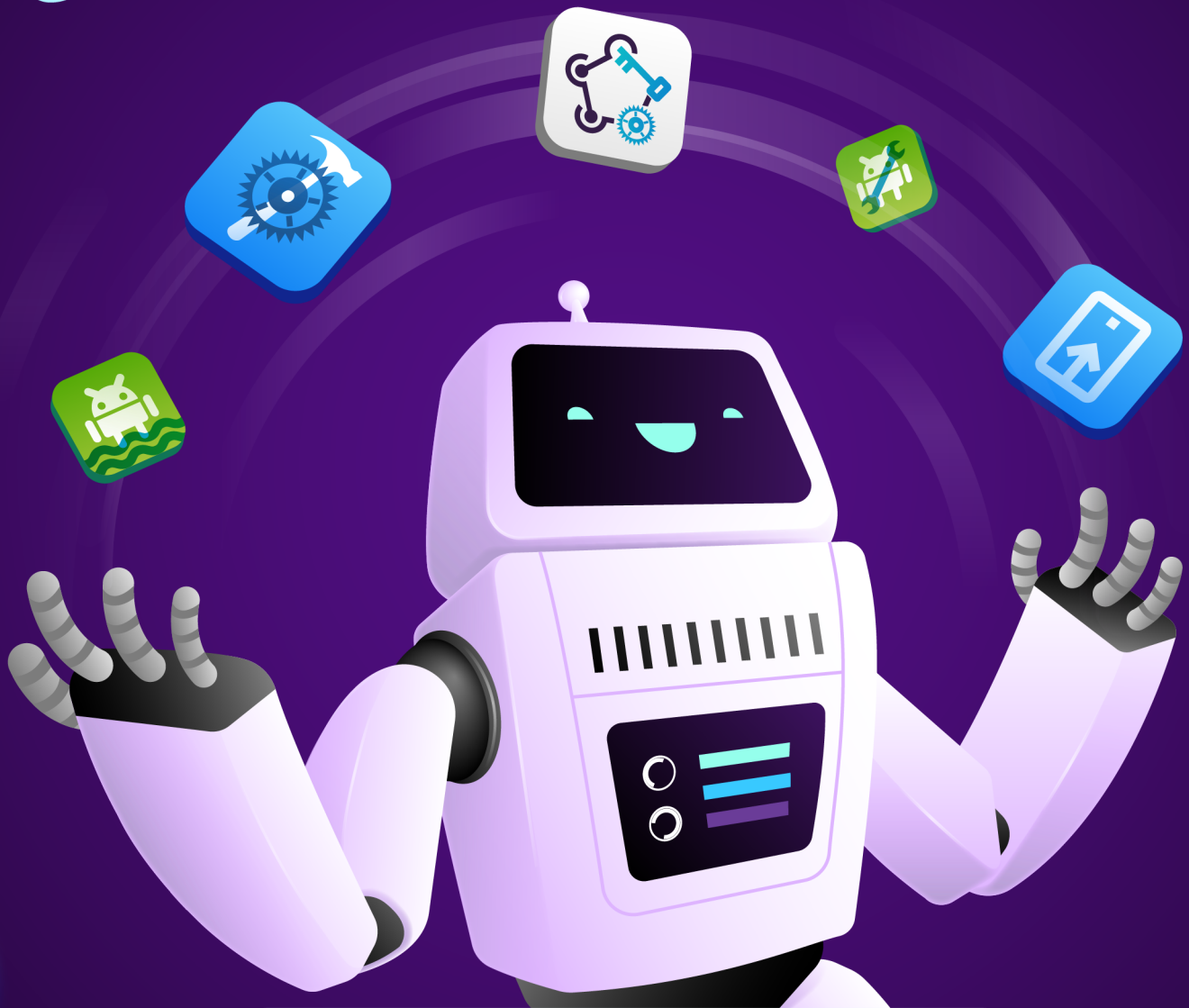


MASTERS OF EFFICIENCY



50+ WORKFLOW RECIPES
FOR PEAK PERFORMANCE


Your all-in-one guide powered by  bitrise

Table of contents

<u>Introduction</u>	4
<u>Chapter 1: Getting Started</u>	6
<u>Chapter 2: Setting Up Your Mobile DevOps Platform</u>	9
<u>Chapter 3: How to Use The Recipes</u>	14
<u>Chapter 4: The Recipes</u>	16
• <u>Cloning & Setup</u>	17
• <u>Dependencies</u>	18
• <u>Testing</u>	19
• <u>Building</u>	24
• <u>Linting</u>	25
• <u>Deploying</u>	26
• <u>Notifications</u>	33
• <u>Caching</u>	35
• <u>Optimisation</u>	42
• <u>Running Steps & Workflows</u>	44
<u>Chapter 5: Technology Specific Recipes</u>	46
• <u>iOS</u>	47
• <u>Android</u>	57
• <u>Other</u>	65
<u>Additional Resources</u>	66

Acknowledgments

This edition has been written with the significant input, contributions and reviews of more than 20 mobile engineers and managers, many of them deep experts in their respective fields.

Thank you very much to all of them.

Special thanks to:

Krisztián Gödrei

Lex Zavala

Olivér Falvai

Szabolcs Tóth

Zsolt Vicze

Introduction

In a fast-paced digital world, where mobile applications are at the forefront of technology, developers and DevOps professionals find themselves constantly navigating the intricacies of the mobile app development landscape. The demand for delivering high-quality mobile applications at an unprecedented pace has given rise to Mobile DevOps, a set of practices that combine mobile app development and operations to streamline the entire process. Just like a master chef perfects their recipes over time, mobile developers and DevOps engineers are constantly refining their workflows to create delightful mobile experiences.

Welcome to “Masters of Efficiency: 50+ Workflow Recipes for Peak Performance.” This pdf is your all-in-one guide to mastering the art of Mobile DevOps, providing a delectable assortment of recipes designed to enhance your mobile development and deployment journey with Bitrise. Much like a chef uses a combination of ingredients, techniques, and creativity to craft a delicious meal, this pdf will empower you to blend the right practices, and methodologies to cook up exceptional mobile applications.

Mobile DevOps is more than just a process; it’s a culinary journey filled with diverse ingredients like continuous integration, continuous delivery, automated testing, and infrastructure as code.

Inside these pages, you will discover a comprehensive collection of **50+ recipes** that cater to the unique tastes and preferences of mobile development and DevOps practitioners. Each recipe is meticulously designed to address specific challenges, encourage best practices, and inspire innovative approaches. Whether you’re a seasoned DevOps expert or a fledgling developer, there’s something for everyone to savor.

Our recipes are organized into chapters, each corresponding to a key stage in the Mobile DevOps process, from setting up, to preparing your ingredients (code and configurations) and onto the process of cooking (building, testing, and deploying your mobile apps). As you advance through the pages, you’ll master each stage of the Mobile DevOps process, learning how to orchestrate the perfect mobile application from start to finish.

The recipes are designed to be practical and adaptable, ensuring that you can incorporate it into your unique workflow, making your mobile development process smoother, faster, and more efficient. Our recipe collection includes both modular recipes and fully backed, plug-and-play workflows tailored for a variety of scenarios.

Modular recipes are designed to enhance specific aspects of your workflow. They consist of a few steps that, when inserted into your existing `bitrise.yml`, complement your workflow but may not function independently. These recipes are ideal for developers seeking to customize and extend their workflows with specific functionalities.

On the other hand, our **fully backed recipes (plug-and-play)** are comprehensive workflows that can be directly integrated into your `bitrise.yml` file, offering a plug-and-play experience. These are complete workflows that require no additional steps to be functional, designed for seamless implementation.

It's important to note that while some recipes appear modular due to their specific focus or limited steps, they are, in fact, whole workflows that can operate "out of the box" when added to an existing `bitrise.yml`. This dual nature means certain recipes may qualify as **both modular and plug-and-play**, offering flexibility and efficiency in workflow customization.

Disclaimer: Recipes that have a full workflow `yml` listed like our (iOS) Nightly for example, do reference Steps that might have older versions compared to what we have available. Therefore our recommendation is to always ensure that you are updating your Steps to the latest stable versions available.

So, whether you're a solo developer, part of a mobile development team, or a seasoned DevOps engineer looking to dive into the world of mobile apps, "Masters of Efficiency: 50+ Workflow Recipes for Peak Performance." will be your go-to guide. Prepare to embark on a journey through these recipes where innovation, efficiency, and excellence converge to create incredible mobile experiences.

Happy building!

The Bitrise Team



Chapter

1

Getting started

Thank you for downloading “Masters of Efficiency: 50+ Workflow Recipes for Peak Performance.”

This pdf is your ultimate guide to mastering the art of Mobile DevOps. In this chapter, we’ll guide you on how to use it effectively.

Who Is This For?

This e-book is designed to cater to a diverse audience, including:

- Mobile App Developers: Looking to streamline their development process.
- DevOps Engineers: Focused on optimizing deployment and operations.
- Team Leaders: Seeking efficient practices for their mobile development teams.
- Anyone Interested in Mobile DevOps: Exploring the world of mobile app delivery.

What to Expect

The recipes are organized into several chapters, each dedicated to a different aspect of the Mobile DevOps process. You’ll find a rich selection of recipes designed to address specific challenges, encourage best practices, and inspire innovation as well as being practical and adaptable, some are fully worked out to give you a plug-and-play experience, whereas others might still need further steps to be added to ensure that you can incorporate them.

How to Get Started:

Before you begin using the recipes, you’ll want to ensure you’ve signed up. In the following chapter, you’ll learn how to set up your workspace, gather the essential tools, and organize your team for a successful Mobile DevOps journey.

Getting Started: Bitrise is a CI/CD Platform as a Service (PaaS), mostly focused on mobile app development. It is a collection of tools and services to help you with the development and automation of your software projects.

To use it, you can sign up via email or via a Git hosting provider, connect a repository, and start building!

Signing up for Bitrise

Signing up for a Bitrise account is easy. You can get started with your:

- [Email](#)
- [GitHub](#)
- [GitLab](#)
- [Bitbucket](#)

Signing up with either of the Git service providers means you connect your Bitrise account to your account on that service provider. With a connected account, you can grant Bitrise access to any of your repositories on that account.

After signing up, you can connect your Bitrise account to all of the three supported Git service providers. For example, after you signed up with GitHub, you can connect your Bitrise account to both your GitLab and Bitbucket accounts, too, and access any repositories you have on those accounts.

- ✓ **Select Your Recipe:** Start by choosing a recipe that aligns with your current needs or interests. Recipes are organized into chapters, making it easy to find what you're looking for.
- ✓ **Gather Your Ingredients:** Each recipe comes with a list of ingredients, which are the tools, techniques, and practices you'll need.
- ✓ **Follow the Instructions:** Read the step-by-step instructions carefully. You'll find detailed guidance on how to implement the recipe in your Mobile DevOps workflow.
- ✓ **Customize to Taste:** Feel free to adapt the recipe to your unique circumstances. We encourage creativity and innovation.
- ✓ **Enjoy the Results:** As you follow the recipe, you'll create an improved mobile development process. Savor the results of your efforts.

Additional Resources

Throughout the pdf, you'll find additional links to tips, techniques, case studies, and recommendations to complement the recipes. Make sure to explore these additional resources to gain a deeper understanding of the Mobile DevOps landscape.

Chapter

2

Setting Up Your Mobile DevOps Platform

Now, it's time to roll up your sleeves and start setting up your Mobile DevOps Platform. Just as a chef needs the right tools, ingredients, and workspace to create culinary masterpieces, you need the right environment to excel in the world of Mobile DevOps.

Creating your first Workspace

After signing up, Bitrise will automatically create your first Workspace. A Workspace is an environment that allows you to manage your Bitrise apps and the team members working on the apps. You need a Workspace to be able to add an app and start running builds. You can:

- Create multiple Workspaces.
- You can be invited to Workspaces by other Bitrise users.

Workspace name

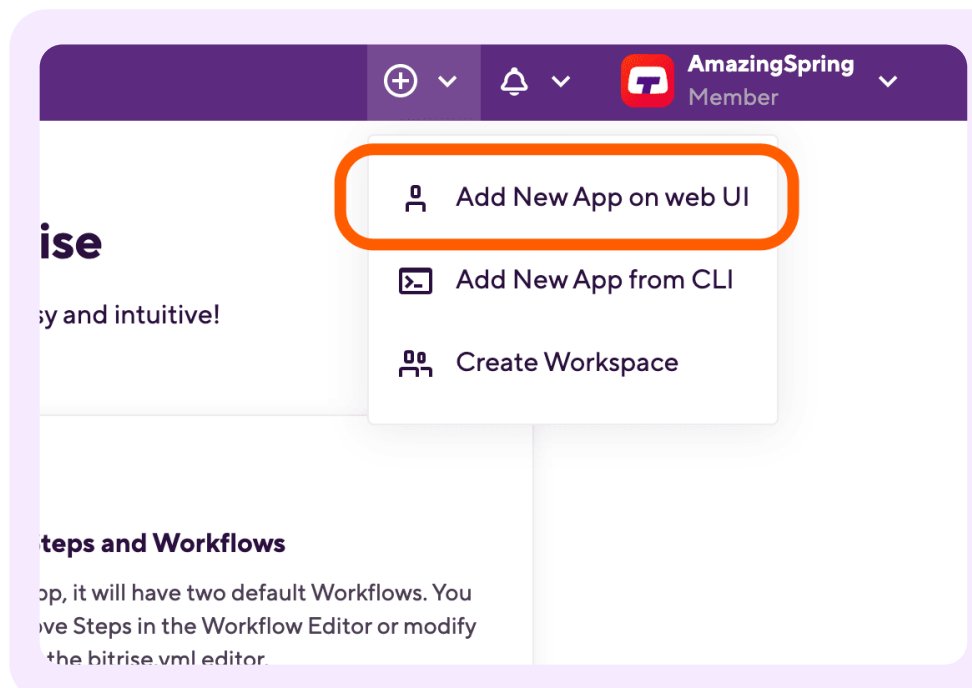
Don't worry: you can rename any of your Workspaces at any time!

To sign up for a paid subscription of your own, you need to have at least one Workspace. Check our [Pricing page](#) for more information or Sign up for a [free trial](#).

Adding a new App

Adding a new app to Bitrise means that you connect a Git repository to Bitrise, allowing us to clone the repository and then build it.

Add a new app any time by clicking the + symbol on the top menu bar and then selecting **Add new app on web UI** from the dropdown menu.



As part of the initial configuration process, you:

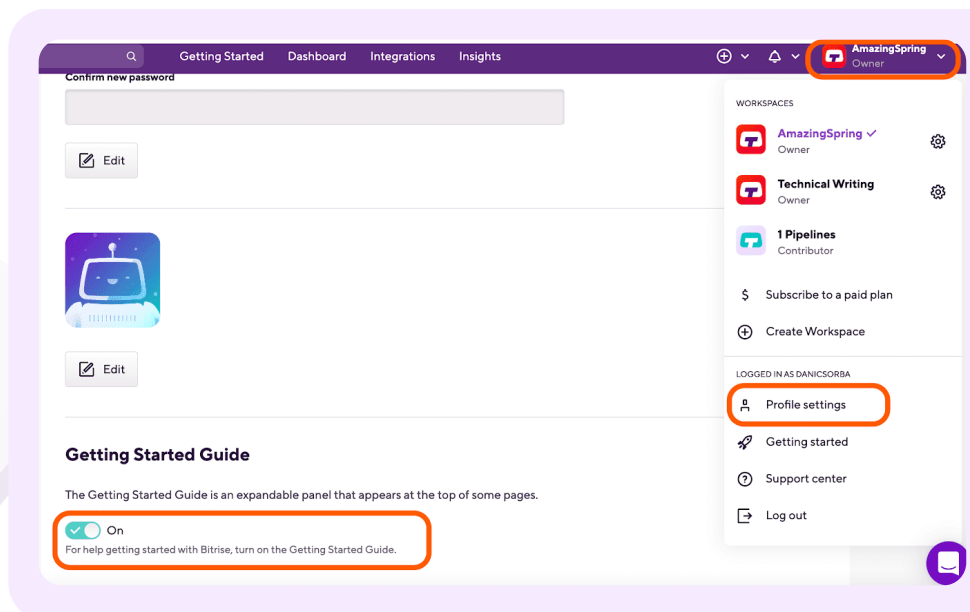
- Decide if an app is private or [public](#). Private app data is only available to those who are invited to work on the app.
- [Specify the repository](#): it can be either a GitHub, GitLab or Bitbucket repository, a manual repository URL, or a self-hosted GitLab repository.
- [Register an SSH key](#): this gives Bitrise access to the repository so we can clone it during the build process.
- [Specify the branch](#) that you want to build.

You can change all this later - and anyway, adding a new app takes a couple of minutes so you can always just do the process from scratch.

As part of the process, Bitrise will [scan](#) and validate your repository and set up an app configuration based on the results of the scan: we can detect the platform type of your app based on the configuration files. If the validation fails, you can set up the app manually.

Read the details of the process in our [Adding your first app guide](#).

You can also enable the Getting Started Guide to receive hints while adding your app: Open your **Profile settings**, scroll down to the **Getting Started Guide** section, and set the toggle to **On**.



Webhooks and Triggers

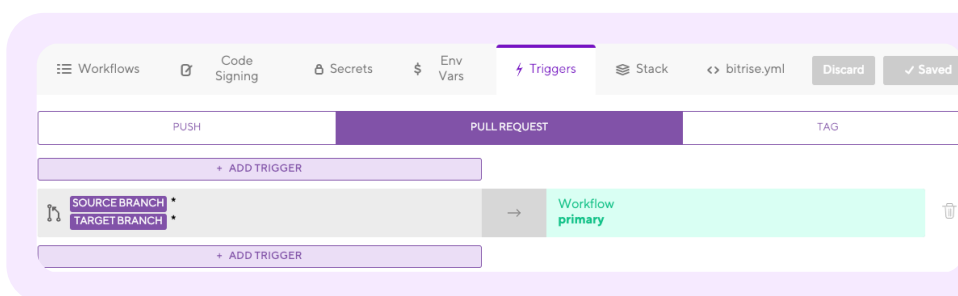
You can set up webhooks as part of the process of adding a new app, or at any time later. Webhooks allow Bitrise to communicate with third party services: for example, a Bitrise webhook set up on a GitHub repository allows Bitrise to start a build automatically when code is modified in the repository.

Once webhooks are set up, configure when to start builds automatically by defining triggers. You can set:

- The event which should trigger the build: for example,

code push or a pull request.

- The branch of your repository that can trigger builds: for example, **main** or **dev**.



This means that you can, for example, set up a trigger that starts a build when a pull request is opened to the main branch.

Webhooks are required for triggers to work! Read more:

- [Adding incoming webhooks](#)
- [Adding outgoing webhooks](#)
- [Triggering builds automatically](#)

Testing and Deploying

[Testing your app](#) and [deploying your app](#) are both done with the help of our [Steps](#): we have Steps dedicated to both these functions, based on the platform type. Unit testing, UI testing, and real device testing are all possible on Bitrise:

- [Device testing for Android](#)
- [Device testing for iOS](#)
- [Running Android unit tests](#)
- [Running unit and UI tests for iOS apps](#)

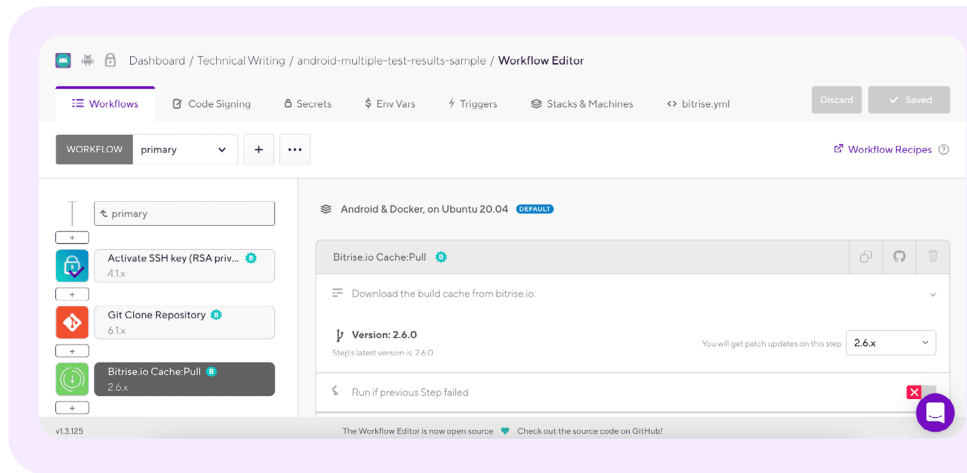
Once your app is tested, built and ready to go, you can quickly deploy it to the store of your choice, for example, Google Play or the App Store.

You can also fast track release cycles by automating boring release tasks, gaining transparency across your release processes, and having one centralized hub to manage it all with the [Bitrise Release Management add-on](#).

Build and Workflows

Once you add an app, your first build will be kicked off automatically. To view your builds, go to your Dashboard - which is the first page once you log in to Bitrise - select the app and click the **Builds** tab to access your builds.

A build is a series of jobs, executed in the order defined in the app's [Workflows](#). The jobs are called [Steps](#), which represent blocks of script execution. The Steps can be arranged on the graphical UI of the [Workflow Editor](#) and they can do a huge number of things: clone your repository, build your app, [run tests](#), [pass values to each other](#), [send notification messages](#) to developers, and many more.



Read more in our relevant guides:

- [Workflows](#)
- [Steps](#)
- [Builds and Pipelines](#)

A build's logs can be viewed on the build's page: go to the **Builds** tab and select the build you want.

All builds run in clean virtual machines that are discarded after the build is complete. Read more about them: [Build machines](#).

Workflows

A Bitrise Workflow is a collection of Steps. When a build of an [app](#) is running, each [Step](#) will be executed in the order that is defined in the Workflow. Workflows can be created, defined and modified in two ways:

- Using the graphical Workflow Editor on [bitrise.io](#), or the [offline version on your own device](#).
- Directly editing the [bitrise.yml](#) file of your project.

Ultimately, both methods modify the **bitrise.yml** file - the **Workflow Editor** is simply a friendlier way of doing so!

By default, a [single build](#) is a single Workflow. But you can also chain Workflows together so they run in succession, as well as to trigger multiple Workflows to run simultaneously.

Workflows can also be arranged into [Pipelines](#). A Pipeline consists of multiple Stages and each Stage consists of one or more Workflows which run in parallel.

Chapter

3

How to Use The Recipes

Workflow Recipes provide ready-made solutions for common Workflow tasks. Here you will find a range of different Recipes along with examples of entire Workflows. We have compiled a vast variety of workflow recipes from the most common use cases on Bitrise to more advanced cases. Special thanks to everyone who contributed with one or more workflow recipes.

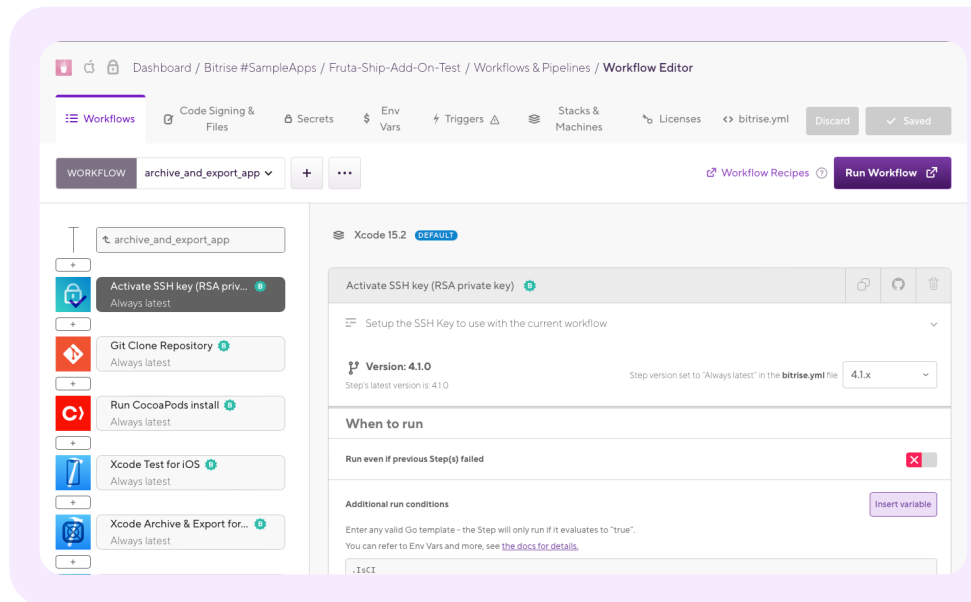
Using Recipes

You can use Workflow Recipes in two ways:

- By adding the Steps into your Workflow via the Workflow Editor.
- By copy-pasting the `bitrise.yml` snippet into your app's [`bitrise.yml` file](#).

Adding Steps via the Workflow Editor

All you need to do here is follow the step-by-step instructions in the Recipe.



Copy-pasting the bitrise.yml snippet

You can also simply copy-paste the snippet to your `bitrise.yml` file directly. Don't forget to:

- ✓ Check the formatting of the copy-pasted YAML
- ✓ Read the instructions as they may contain some important information on configuration
- ✓ Check and customize the input variables

Chapter

4

The Recipes

4.1 Cloning & Setup



MODULAR

Cloning the repository

Description

Clone a git repo.

Instructions

1. Make sure that the Workflow has the [Activate SSH key \(RSA private key\)](#) Step. This allows the Git client on the build VM to access private repositories.
2. Add the [Git Clone Repository](#) Step.
 - Check out the optional inputs in the Workflow Editor or in the Step documentation.main or dev.

bitrise.yml

```
- activate-ssh-key@4:  
  run_if: '{{getenv "SSH_RSA_PRIVATE_KEY" | ne ""}}'  
- git-clone@8: {}
```

Install Flutter SDK

Description

Install the latest stable/beta or a specific version of Flutter.

Instructions

1. Add the [Flutter Install](#) Step. Use this Step before the Cache Pull Step to make sure caching works correctly.
2. You can install either the latest stable/beta versions or a specific version:
 - By default, the Step installs the latest stable version.
 - To install the latest beta, set the Flutter SDK git repository version input to **beta**.
 - **Recommended:** To install a specific version, set the Flutter SDK installation bundle URL input. You can find the list of Flutter installation bundles here: <https://flutter.dev/docs/development/tools/sdk/releases>. Make sure you set the bundle based on the stack (MacOS or Linux).

bitrise.yml

Specific version (recommended):

```
- flutter-installer@0:  
  inputs:  
    - installation_bundle_url: https://storage.  
googleapis.com/flutter_infra_release/releases/stable/  
macos/flutter_macos_2.5.3-stable.zip
```



Latest stable:

```
- flutter-installer@0: {}
```

Latest beta:

```
- flutter-installer@0:  
  inputs:  
    - version: beta
```

4.2 Dependencies



MODULAR

(iOS) Install CocoaPods Dependencies

Description

Installing CocoaPods dependencies. Make sure that you are using the workspace and not the project file in your steps. Check the value of **\$BITRISE_PROJECT_PATH** env var.

Instructions

1. Add the [Run CocoaPods install](#) Step.
2. (Optional) If your Podfile is not in the root, you can set the Podfile path input.

bitrise.yml

```
- cocoapods-install@2: {}
```

(iOS) Install Carthage Dependencies

Description

Installing Carthage dependencies.

Instructions

Add the [Carthage](#) Step. Set the input variables:

- **Github Personal Access Token:** We recommend adding a GitHub access token to your Secrets (**\$GITHUB_ACCESS_TOKEN**). We need this token to avoid GitHub rate limit issue. See the GitHub guide: [Creating an access token for command-line use](#) on how to create Personal Access Token. Uncheck every scope box when creating this token. There is no reason this token needs access to private information.
- (Optional) **Additional options for carthage command:** See the [Carthage docs](#) for the available options, for example, **--use-xcframeworks --platform iOS**.

bitrise.yml

```
- carthage@3:  
  inputs:  
    - carthage_options: "--use-xcframeworks --platform iOS"
```



MODULAR



4.3 Testing



MODULAR

(React Native) Install dependencies

Description

Install dependencies using either yarn or npm.

Instructions

1. Add either the [Run yarn command](#) or the [Run npm command](#) Step based on your project setup.
2. Set the yarn command to run or the npm command with arguments to run input to **install**.

bitrise.yml

Using yarn:

```
- yarn@0:
  inputs:
    - command: install
```

Using npm:

```
- npm@1:
  inputs:
    - command: install
```



MODULAR

(iOS) Run tests on a simulator

Description

Run unit or UI tests of an iOS app on a simulator.

Instructions

1. Add an [Xcode Test for iOS](#) Step. Override any of the following inputs if needed:
 - **Project path:** The default value is **\$BITRISE_PROJECT_PATH** and in most cases you don't have to change it.
 - **Scheme:** The default value is **\$BITRISE_SCHEME**, this variable stores the scheme that you set when adding the app on Bitrise. You can specify a different scheme if you want but it must be a shared scheme.
 - **Device destination specifier** (default: **platform=iOS Simulator,name=iPhone 8 Plus,OS=latest**).
2. Add a [Deploy to Bitrise.io - Apps, Logs, Artifacts](#) Step that makes the test results available in the [Test Reports add-on](#). The failed tests will be also available under the **Test Results** tab on the build details page.

bitrise.yml

```
- xcode-test@5: {}
- deploy-to-bitrise-io@2: {}
```



MODULAR

(iOS) Run Tests on a Physical Device

Description

Run unit or UI tests on a physical device. Our device testing solution is based on Firebase Test Lab. You can find the resulting logs, videos and screenshots on Bitrise.

Prerequisites

1. The source code is cloned and the dependencies (for example, CocoaPods, Carthage) are installed.
2. You have code signing set up. See [iOS Code Signing](#) for more details.

Instructions

1. Add an [Xcode Build for testing for iOS](#) Step.
2. Add a [\[BETA\] iOS Device Testing](#) Step.
 - Setup code signing for the Step.
3. Add a [Deploy to Bitrise.io - Apps, Logs, Artifacts](#) Step that makes the test results available in the [Test Reports add-on](#).

bitrise.yml

```
- xcode-build-for-test@3:
  inputs:
    - automatic_code_signing: api_key
- virtual-device-testing-for-ios@1: {}
- deploy-to-bitrise-io@2: {}
```

Relevant links

- <https://devcenter.bitrise.io/en/testing/device-testing-for-ios.html>

(Android) Run unit tests

Description

Run unit tests (for example, `testDebugUnitTest`).

Instructions

1. Add an [Android Unit Test](#) Step. Input variables you might set:
 - **Project Location:** Use the default `$BITRISE_SOURCE_DIR` or `$PROJECT_LOCATION`. You can set a specific path but the automatically exposed Environment Variables are usually the best option.
 - **Variant:** Use the `$VARIANT` Environment Variable, or specify a variant manually.
 - **Module:** Specify one or leave it blank to run tests in all of the modules.



MODULAR



2. Add a [Deploy to Bitrise.io - Apps, Logs, Artifacts](#) Step that makes the test results available in the [Test Reports add-on](#). The failed tests will be also available under the **Test Results** tab on the build details page.

bitrise.yml

```
- android-unit-test@1:
  inputs:
    - project_location: $PROJECT_LOCATION
    - variant: $VARIANT
- deploy-to-bitrise-io@2: {}
```

Relevant links

- [Android unit tests](#)

(Android) Run UI / instrumentation tests on virtual devices

Description

Run UI / instrumentation (for example, Espresso) or robo/gameloop tests on virtual devices. [Our device testing solution](#) is based on Firebase Test Lab. You can find the resulting logs, videos and screenshots on Bitrise.

Instructions

1. Add an [Android Build for UI Testing](#) Step. Set the input variables:
 - **Project Location:** Use the default **\$BITRISE_SOURCE_DIR** or **\$PROJECT_LOCATION**. You can set a specific path but the automatically exposed Environment Variables are usually the best option.
 - **Variant:** Use the **\$VARIANT** Environment Variable, or specify a variant manually.
 - **Module:** Specify one or leave it blank to run tests in all of the modules.
2. Add a [\[BETA\] Virtual Device Testing for Android Step](#). Set the input variables:
 - **Test type: instrumentation** (or **robo** or **gameloop**).
 - (Optional) Test devices (default: **NexusLowRes,24,en,portrait**).
3. Add a [Deploy to Bitrise.io - Apps, Logs, Artifacts](#) Step that makes the test results available in the [Test Reports add-on](#).

bitrise.yml



MODULAR





```
- android-build-for-ui-testing@0:
  inputs:
    - variant: $VARIANT
    - module: $MODULE
- virtual-device-testing-for-android@1:
  inputs:
    - test_type: instrumentation
- deploy-to-bitrise-io@2: {}
```

Relevant links

- [Device testing for Android](#)

(Android) Run UI / instrumentation tests on local emulator

MODULAR

Description

Run UI / instrumentation tests on a local emulator instance.

Instructions

1. Add an [AVD Manager](#) Step. To customize the emulator, see the [Step configuration](#).
2. Add a [Wait for Android emulator](#) Step.
3. Add a [Gradle Runner](#) Step. Set the input variables:
 - **gradlew file path:** for example, `./gradlew`.
 - **Gradle task to run:** for example, `connectedDebugAndroidTest`.
4. Add an [Export test results to Test Reports add-on](#) Step with the following inputs:
 - **The name of the test:** `Emulator tests`.
 - **Test result base path:** `$BITRISE_SOURCE_DIR/app/build/outputs/androidTest-results`. You might want to adjust the path based on the module name(s) in your project.
 - **Test result search pattern:** `*.xml`.
5. Add a [Deploy to Bitrise.io - Apps, Logs, Artifacts](#) Step that makes the test results available in the [Test Reports add-on](#). The failed tests will be also available under the **Test Results** tab on the build details page.

bitrise.yml

```
- avd-manager@1: {}
- wait-for-android-emulator@1:
- gradle-runner@2:
  inputs:
    - gradlew_path: ./gradlew
    - gradle_task: connectedDebugAndroidTest
```





```
- custom-test-results-export@0:
  inputs:
    - search_pattern: "*.xml"
    - base_path: $BITRISE_SOURCE_DIR/app/build/outputs/
      androidTest-results
    - test_name: Emulator tests
- deploy-to-bitrise-io@2:
```

(React Native) Run tests

Description

Run tests (for example, Jest).

Instructions

1. Add either the [Run yarn command](#) or the [Run npm command](#) Step based on your project setup.
2. Set the The yarn command to run or The npm command with arguments to run input to **test**.

bitrise.yml

Using yarn:

```
- yarn@0:
  inputs:
    - command: test
```

Using npm:

```
- npm@1:
  inputs:
    - command: test
```

(Flutter) Run tests

Description

Performs any test in a Flutter project.

Instructions

1. Add the [Flutter Test](#) Step to your Workflow. Set the input variables:
 - Project Location: For example, **\$BITRISE_FLUTTER_PROJECT_LOCATION**.
 - Check out optional inputs in the Workflow Editor or in the Step description.
2. Add a [Deploy to Bitrise.io - Apps, Logs, Artifacts](#) Step that makes the test results available in the [Test Reports add-on](#). The failed tests will be also available under the **Test Results** tab on the build details page.

bitrise.yml

```
- flutter-test@1:
  inputs:
    - project_location: $BITRISE_FLUTTER_PROJECT_LOCATION
- deploy-to-bitrise-io@2: {}
```

MODULAR

MODULAR



4.4 Building

PLUG & PLAY

MODULAR

(React Native) Expo: Build using Turtle CLI

Description

Publish an app to Expo's servers and build an iOS App Store **.ipa** and Android **.aab** files from your Expo project using [Turtle CLI](#).

Prerequisites

1. Generate an iOS Distribution Certificate and an App Store Provisioning Profile based on the [Generating iOS code signing files](#) guide.
2. Generate an Android Keystore by following the [Android code signing with Android Studio](#) guide.
3. Make sure you can [Publish your Expo project](#) locally.

Instructions

1. Upload the project's iOS Distribution Certificate and App Store Provisioning Profile on the Bitrise project's Workflow Editor / Code signing tab.
2. Upload the project's Android Keystore on the Bitrise project's Workflow Editor / Code signing tab.
3. Create a Secret (**IOS_DEVELOPMENT_TEAM**) with the ID of the iOS Development Team, issued the project's Certificate and Provisioning Profile.
4. Store the Expo account, used for publishing the Expo app and fetching the app manifest, in **EXPO_USERNAME** and **EXPO_PASSWORD** secrets.
5. Copy-paste **envs** from **bitrise.yml** below to your Workflow.
6. Copy-paste **steps** from **bitrise.yml** below to your Workflow.
 - The built **.ipa** and **.aab** files are exposed via **BITRISE_IPA_PATH** and **BITRISE_AAB_PATH** env vars.

bitrise.yml

```
turtle_build:
  envs:
    - KEYSTORE_PATH: /tmp/keystore.jks
    - KEYSTORE_ALIAS: $BITRISEIO_ANDROID_KEYSTORE_ALIAS
    - EXPO_ANDROID_KEYSTORE_PASSWORD: $BITRISEIO_
      ANDROID_KEYSTORE_PASSWORD
    - EXPO_ANDROID_KEY_PASSWORD: $BITRISEIO_ANDROID_
      KEYSTORE_PRIVATE_KEY_PASSWORD
    - PROFILE_PATH: /tmp/profile.mobileprovision
    - CERTIFICATE_PATH: /tmp/certificate.p12
    ...
  ...
  PLEASE LOOK AT GITHUB PAGE FOR THE REST OF BITRISE.YML
```

4.5 Linting



MODULAR

(Android) Run Lint

Description

Runs Lint on your Android project and generates a report with the results.

Instructions

1. Add the [Android Lint](#) Step. Set the input variables:
 - **Project Location:** Use the default `$BITRISE_SOURCE_DIR` or `$PROJECT_LOCATION`. You can set a specific path but the automatically exposed Environment Variables are usually the best option.
 - **Variant:** Use the `$VARIANT` Environment Variable, or specify a variant manually.
 - **Module:** Specify one or leave it blank to run lint in all of the modules.
2. Add a [Deploy to Bitrise.io](#) Step. This Step uploads the lint report as a [build artifact](#).

bitrise.yml

```
- android-lint@0:  
  inputs:  
    - variant: $VARIANT  
- deploy-to-bitrise-io@2: {}
```

(Flutter) Run Dart Analyzer

Description

Runs the Dart Analyzer.

Instructions

1. Add the [Flutter Analyze](#) Step to your Workflow.

bitrise.yml

```
- flutter-analyze@0:  
  inputs:  
    - project_location: $BITRISE_FLUTTER_PROJECT_LOCATION
```



MODULAR

4.6 Deploying

MODULAR



(iOS) Deploy to App Store Connect / TestFlight

Description

Archiving the app and uploading to App Store Connect to either release it to App Store or to TestFlight.

Prerequisites

1. The source code is cloned and the dependencies (for example, CocoaPods, Carthage) are installed.
2. You have code signing set up. See [iOS Code Signing](#) for more details.
3. You have Apple Developer connection set up. See [Apple services connection](#) for more details.

Instructions

1. (Optional) Add the Set [Xcode Project Build Number](#) Step. Set the input variables:
 - **Info.plist file path:** for example, **MyApp/Info.plist**.
 - **Build Number:** for example, **42**.
 - **Version Number:** for example, **1.1**.
2. Add the [Xcode Archive & Export for iOS](#) Step. Set the input variables:
 - **Project path:** by default **\$BITRISE_PROJECT_PATH**. Normally, you don't have to change this.
 - **Scheme:** by default **\$BITRISE_SCHEME**. This Environment Variable stores the scheme that you set when adding the app. The scheme always must be a shared scheme.
 - **Distribution method:** it must be set to **app-store**.
3. [Add the Deploy to App Store Connect - Application Loader \(formerly iTunes Connect\)](#) Step. Set the input variables:
 - **Bitrise Apple Developer Connection:** for example, **api_key**.

Alternatively you can use the [Deploy to App Store Connect with Deliver \(formerly iTunes Connect\)](#) Step as well, which gives you more options.

bitrise.yml

```
- set-xcode-build-number@1:
  inputs:
    - build_short_version_string: '1.0'
    - plist_path: BitriseTest/Info.plist
- xcode-archive@5:
  inputs:
    - project_path: $BITRISE_PROJECT_PATH
    - scheme: $BITRISE_SCHEME
    - automatic_code_signing: api_key
    - distribution_method: app-store
- deploy-to-itunesconnect-application-loader@1:
  inputs:
    - connection: api_key
```

(Android) Deploy to Google Play (Internal, Alpha, Beta, Production)

MODULAR

Description

Building the app and uploading to Google Play to internal, alpha, beta or production track.

Prerequisites

1. An Android keystore file is uploaded to Bitrise. For details, see [Android code signing using the Android Sign Step](#).
2. Google Play API Access is set up. For details, see [Deploying Android apps to Bitrise and Google Play](#).

Instructions

1. (Optional) Add the [Change Android versionCode and versionName](#) Step. Set the input variables:
 - **Path to the build.gradle file:** The default value is `$PROJECT_LOCATION/$MODULE/build.gradle` and in most cases you don't have to change it.
 - **New versionName:** for example, `1.0.1`.
 - **New versionCode:** for example, `42`.
2. Add the [Android Build](#) Step and set the following inputs:
 - **Build type:** Set this to `aab`.
 - **Variant:** Use `release`, `debug`, or one of your custom variants if you have any.
 - **Module:** for example `$MODULE`.
3. Add the [Android Sign](#) Step.
4. Add the [Google Play Deploy](#) Step and set the following inputs:
 - **Service Account JSON key file path:** `$BITRISEIO_SERVICE_ACCOUNT_JSON_KEY_URL`.
 - **Package name:** for example, `com.your.package.name`.
 - **Track:** Choose one of `internal`, `alpha`, `beta`, or `production`.
 - **Status:** The status of a release. For more information, see the [API reference](#). Recommended draft for `production` and `completed` for internal test builds.
 - Check the other options in the Workflow Editor or in the Step documentation.

bitrise.yml

```
- change-android-versioncode-and-versionname@1:
  inputs:
    - new_version_name: 1.0.1
    - new_version_code: '42'
    - build_gradle_path: "$PROJECT_LOCATION/$MODULE/build.gradle"
```



```

- android-build@1:
  inputs:
    - project_location: $PROJECT_LOCATION
    - module: $MODULE
    - build_type: aab
    - variant: release
- sign-apk@1: {}
- google-play-deploy@3:
  inputs:
    - service_account_json_key_path: $BITRISEIO_SERVICE_ACCOUNT_JSON_KEY_URL
    - package_name: io.bitrise.sample.android
    - status: completed
    - track: internal

```

(iOS) Deploy to bitrise.io

MODULAR

Description

Build and distribute your app to testers via [Bitrise.io Ship](#).

Prerequisites

1. You have code signing set up. See [iOS Code Signing](#) for more details.

Instructions

1. Add the [Xcode Archive & Export for iOS](#) Step.
Set the input variables:
 - **Project path:** by default, **\$BITRISE_PROJECT_PATH**.
 - **Scheme:** by default, **\$BITRISE_SCHEME**.
 - **Distribution method:** **development**, **ad-hoc** or **enterprise**.
2. Add the [Deploy to Bitrise.io - Apps, Logs, Artifacts](#) Step.

bitrise.yml

```

- xcode-archive@5:
  inputs:
    - project_path: $BITRISE_PROJECT_PATH
    - scheme: $BITRISE_SCHEME
    - automatic_code_signing: apple-id
    - distribution_method: development
- deploy-to-bitrise-io@2: {}

```





(Android) Deploy to Bitrise.io

Description

Build and distribute your app to testers via the [Bitrise.io Ship add-on](#).

Prerequisites

1. If you want to deploy a release build, don't forget to set up [code signing on Bitrise](#) to build and sign the APK with your release key.

Instructions

1. Add the [Android Build](#) Step and set the following inputs:
 - **Build type:** Set this to **apk**.
 - **Variant:** Use **release**, **debug**, or one of your custom variants if you have any.
2. If you build a release variant, add the [Android Sign](#) Step. You can skip this if you plan to deploy an unsigned debug variant.
3. Add a [Deploy to Bitrise.io - Apps, Logs, Artifacts](#) Step.

bitrise.yml

```
- android-build@1:
  inputs:
    - variant: release
    - build_type: apk
- sign-apk@1: {}
- deploy-to-bitrise-io@2: {}
```

(iOS) Deploy to Firebase App Distribution

Description

Build and distribute your app to testers via Firebase App Distribution.

Prerequisites

1. An existing Firebase project where your exact bundle ID is registered. Follow the [Firebase documentation](#) for details.
2. Obtain a token from Firebase by running **firebase login:ci** locally. See the [Firebase CLI](#) docs for more details.
3. Add this token as a [secret](#) your Bitrise project with the name **FIREBASE_TOKEN**.
4. Get your Firebase App ID from your project's General Settings page and pass this value as an input to the **firebase-app-distribution** Step.
5. Setting up code signing on Bitrise is not part of this guide, please follow our [code signing docs](#) for instructions.



Instructions

1. Add the [Xcode Archive](#) Step and set the required inputs, such as scheme, distribution method and the desired code signing method.
2. Add the [Firebase App Distribution](#) Step and set the following inputs:
 - **Firestore token:** use the secret env var previously defined: **\$FIREBASE_TOKEN**.
 - **Firestore App ID:** see the Prerequisites section above for details.
 - Optionally, you can define test groups or individual testers in the Step inputs.

bitrise.yml

```
- xcode-archive@5:
  inputs:
    - distribution_method: development
    - scheme: $BITRISE_SCHEME
    - automatic_code_signing: api-key
- firebase-app-distribution@0:
  inputs:
    - firebase_token: $FIREBASE_TOKEN
    - app: 1:1234567890:ios:321abc456def7890 # your app
    ID from Firebase
    - testers: email@company.com # optional
    - groups: qa-team #optional
```



(Android) Deploy to Firebase App Distribution

Description

Build and distribute your app to testers via Firebase App Distribution. This example builds and deploys an APK, but the workflow can be tweaked to distribute AAB instead.

Prerequisites

1. An existing Firebase project where your exact bundle ID is registered. Follow the [Firestore documentation](#) for details.
2. Obtain a token from Firebase by running **firebase login:ci** locally. See the [Firestore CLI](#) docs for more details.
3. Add this token as a [secret](#) your Bitrise project with the name **FIREBASE_TOKEN**.
4. Get your Firestore App ID from your project's General Settings page and pass this value as an input to the **firebase-app-distribution** Step.
5. Settings up code signing on Bitrise is not part of this guide, please follow our [code signing docs](#) for instructions.

MODULAR





Instructions

1. Add the [Android Build](#) Step and set the following inputs:
 - **Build type:** Set this to **apk**.
 - **Variant:** Use **release**, **debug**, or one of your custom variants if you have any.
2. If you build a release variant, add the [Android Sign](#) Step. You can skip this if you plan to deploy an unsigned debug variant.
3. Add the [Firebase App Distribution](#) Step and set the following inputs:
 - **Firebase token:** use the secret env var previously defined: **\$FIREBASE_TOKEN**.
 - **App path:** this should point to the APK that the previous steps have built and signed. By default, it's located at **\$BITRISE_DEPLOY_DIR/app-release-bitrise-signed.apk**, but the exact file name might be different based on your project config.
 - **Firebase App ID:** see the Prerequisites section above for details.
 - **Optional:** you can define test groups or individual testers in the Step inputs.

bitrise.yml

```
- android-build@1:
  inputs:
    - variant: release
    - build_type: apk
- sign-apk@1: {}
- firebase-app-distribution@0:
  inputs:
    - firebase_token: $FIREBASE_TOKEN
    - app_path: $BITRISE_DEPLOY_DIR/app-release-bitrise-signed.apk
    - app: your_app_id_from_firebase
    - testers: email@company.com # optional
    - groups: qa-team #optional
```

(iOS) Deploy to Visual Studio App Center

MODULAR

Description

Build and distribute your app to testers via AppCenter.

Prerequisites

1. An existing [Visual Studio App Center](#) project where your app is registered.
2. Adding the API token as a [Secret](#) your Bitrise project with the name **APPCENTER_API_TOKEN**.
3. You have code signing set up. See [iOS Code Signing](#) for more details.





Instructions

1. Add the [Xcode Archive & Export for iOS](#) Step. Set the input variables:
 - Project path: by default **\$BITRISE_PROJECT_PATH**.
 - Scheme: by default **\$BITRISE_SCHEME**.
 - Distribution method: **development**, **ad-hoc** or **enterprise**.
2. Add the [AppCenter iOS Deploy](#) Step and set the following inputs:
 - API Token: **\$APPCENTER_API_TOKEN**.
 - Owner name: for example, **my-company**.
 - App name: for example, **my-app** Use the [App Center CLI](#) to get the app name since it might not be the same as the one you can see on the Visual Studio App Center website.
 - Check out other options in the Step documentation or in the Workflow Editor.

bitrise.yml

```
- xcode-archive@5:
  inputs:
    - project_path: $BITRISE_PROJECT_PATH
    - scheme: $BITRISE_SCHEME
    - automatic_code_signing: apple-id
    - distribution_method: development
- appcenter-deploy-ios@2:
  inputs:
    - owner_name: my-company
    - app_name: my-app
    - api_token: $APPCENTER_API_TOKEN
```

(Android) Deploy to Visual Studio App Center

Description

Build and distribute your app to testers via AppCenter.

Prerequisites

1. An existing [Visual Studio App Center](#) project where your app is registered.
2. Adding the API token as a [Secret](#) your Bitrise project with the name **APPCENTER_API_TOKEN**.
3. If you want to deploy a release build, don't forget to [set up code signing on Bitrise](#) to build and sign the APK with your release key.

Instructions

1. Add the [Android Build](#) Step and set the following inputs:
 - **Build type:** Set this to **apk**.
 - **Variant:** Use **release**, **debug**, or one of your custom variants if you have any.



MODULAR





2. If you build a release variant, add the [Android Sign](#) Step. You can skip this if you plan to deploy an unsigned debug variant.
3. Add the [AppCenter Android Deploy](#) Step and set the following inputs:
 - **API Token:** `$APPCENTER_API_TOKEN`.
 - **Owner name:** For example, `my-company`.
 - **App name:** For example, `my-app`. Use the [App Center CLI](#) to get the app name since it might not be the same as the one you can see on the Visual Studio App Center website.
 - Check out other options in the Step documentation or in the Workflow Editor.

bitrise.yml

```
- android-build@1:
  inputs:
    - variant: release
    - build_type: apk
- sign-apk@1: {}
- appcenter-deploy-android@2:
  inputs:
    - owner_name: my-company
    - app_name: my-app
    - app_path: $BITRISE_APK_PATH
    - api_token: $APPCENTER_API_TOKEN
```

4.7 Notifications

(iOS/Android) Slack - send build status

Description

Sending a slack message to Slack with the build status after a build has finished.

Prerequisites

1. You have a Slack webhook set up and added to Env Vars (for example, `$SLACK_WEBHOOK`). For details, see [Configuring Slack integration](#).

Instructions

1. Add the [Send a Slack message](#) Step. Set the input variables:
 - **Slack Webhook URL:** for example, `$SLACK_WEBHOOK`.
 - **Target Slack channel, group or username:** for example, `#build-notifications`.
 - Check out the other optional input variables in the Workflow Editor or in the Step description.

bitrise.yml

```
- slack@4:
  inputs:
    - channel: "#build-notifications"
    - webhook_url: $SLACK_WEBHOOK
```

MODULAR





MODULAR

(iOS/Android) Send QR code to Slack

Description

Sending a QR code of the iOS or Android build uploaded to bitrise.io to Slack.

Prerequisites

1. You have your iOS or Android app archived.
2. You have a Slack webhook set up and added to Env Vars (for example, `$SLACK_WEBHOOK`). For details, see [Configuring Slack integration](#).

Instructions

1. Add the [Deploy to Bitrise.io - Apps, Logs, Artifacts](#) Step.
2. Add the [Create install page QR code](#) Step.
3. Add the [Send a Slack message](#) Step. Set the input variables:
 - **Slack Webhook URL:** for example, `$SLACK_WEBHOOK`.
 - **Target Slack channel, group or username:** for example, `#build-notifications`.
 - **A URL to an image file that will be displayed as a thumbnail:** `$BITRISE_PUBLIC_INSTALL_PAGE_QR_CODE_IMAGE_URL`.

bitrise.yml

```
- deploy-to-bitrise-io@2: {}
- create-install-page-qr-code@1: {}
- slack@4:
  inputs:
    - channel: "#build-notifications"
    - thumb_url: $BITRISE_PUBLIC_INSTALL_PAGE_QR_CODE_
      IMAGE_URL
    - webhook_url: $SLACK_WEBHOOK
```

Relevant links

- [Deploying an iOS app to Bitrise.io](#)
- [Deploying Android apps to Bitrise and Google Play](#)

MODULAR

GitHub pull request - send the build QR code

Description

Sending a QR code of the iOS or Android build uploaded to bitrise.io to Slack.

Prerequisites

1. You have your iOS or Android app archived.
2. Generate a [GitHub personal access token](#) and add it as a Secret (`$GITHUB_ACCESS_TOKEN`). Make sure to select the **repo** scope.





Instructions

1. Add the [Deploy to Bitrise.io - Apps, Logs, Artifacts](#) Step.
2. Add the [Create install page QR code](#) Step.
3. Add the [Comment on GitHub Pull Request](#) Step. Set the following input variables:
 - **GitHub personal access token:** Set it to the previously created Secret, `$GITHUB_ACCESS_TOKEN`.

bitrise.yml

```
- deploy-to-bitrise-io@2: {}
- create-install-page-qr-code@1: {}
- comment-on-github-pull-request@0:
  inputs:
    - body: |-
        ![QR code]($BITRISE_PUBLIC_INSTALL_PAGE_QR_CODE_
        IMAGE_URL)

        $BITRISE_PUBLIC_INSTALL_PAGE_URL
    - personal_access_token: $GITHUB_ACCESS_TOKEN
```

4.8 Caching



MODULAR

(iOS) Cache Swift Package Manager dependencies (Beta)

Description

Cache the resolved Swift package dependencies with the new key-based caching Steps, **Save Cache** and **Restore Cache**.

Instructions

1. Add the [Restore SPM Cache](#) Step to the Workflow.
2. Add one of the usual iOS build Steps, such as [Xcode Test for iOS](#).
3. Add the [Save SPM Cache](#) Step.

Fine tune cache behaviour

The SPM specific cache Steps use optimal cache key and path configurations maintained by Bitrise. If you want full control over what should be cached then please check out the generic [Restore Cache](#) and [Save Cache](#) Steps.

You can always check out what key and path settings the SPM cache Step uses: [Github code snippet](#).

bitrise.yml

```
- restore-spm-cache@1: {}
- xcode-test@5: {}
- save-spm-cache@1: {}
```



(iOS) Cache CocoaPods dependencies (Beta)

MODULAR

Description

Cache the contents of the **Pods** folder with the new key-based caching Steps, **Save Cache** and **Restore Cache**.

Instructions

1. Add the [Restore Cocoapods Cache](#) Step to the Workflow.
2. Add the [Run CocoaPods install](#) Step.
3. Add the [Save Cocoapods Cache](#) Step.

Fine tune cache behaviour

The Cocoapods specific cache Steps use optimal cache key and path configurations maintained by Bitrise. If you want full control over what should be cached then please check out the generic [Restore Cache](#) and [Save Cache](#) Steps. You can always check out what key and path settings the Cocoapods cache Step uses: [Github code snippet](#).

bitrise.yml

```
- restorecocoapods-cache@1: {}  
- cocoapods-install@2: {}  
- save-cocoapods-cache@1: {}
```

(iOS) Cache Carthage dependencies (Beta)

MODULAR

Description

Cache the contents of the **Carthage** folder with the new key-based caching Steps, **Save Cache** and **Restore Cache**.

Instructions

1. Add the [Restore Carthage Cache](#) Step to the Workflow.
2. Add the [Carthage](#) Step.
3. Add the [Save Carthage Cache](#) Step.

Fine tune cache behaviour

The Carthage specific cache Steps use optimal cache key and path configurations maintained by Bitrise. If you want full control over what should be cached then please check out the generic [Restore Cache](#) and [Save Cache](#) Steps. You can always check out what key and path settings the Carthage cache Step uses: [Github code snippet](#).

bitrise.yml

```
- restore-carthage-cache@1: {}  
- carthage@1: {}  
- save-carthage-cache@1: {}
```


(Android) Cache Gradle dependencies (Beta)

MODULAR

Description

Cache project dependencies that Gradle downloads with the new key-based caching Steps, **Save Cache** and **Restore Cache**.

Instructions

1. Add the [Restore Gradle Cache](#) Step to the Workflow.
2. Add the usual Android Steps, such as [Android Build](#).
3. Add the [Save Gradle Cache](#) Step.

Fine tune cache behaviour

The Gradle specific cache Steps use optimal cache key and path configurations maintained by Bitrise. If you want full control over what should be cached then please check out the generic [Restore Cache](#) and [Save Cache](#) Steps. You can always check out what key and path settings the Gradle cache Step uses: [Github code snippet](#).

bitrise.yml

```
- restore-gradle-cache@1: {}  
- android-build@1:  
  inputs:  
    - variant: debug  
    - build_type: apk  
- save-gradle-cache@1: {}
```

(Android) Cache Gradle build tasks (Beta)

MODULAR

Description

Cache Gradle tasks with the new key-based caching Steps, **Save Gradle Cache** and **Restore Gradle Cache**.

Prerequisites

Make sure to read how to [cache Gradle dependencies](#) and set up the Workflow according to the guide. Caching build tasks is an opt-in feature that builds on caching Gradle dependencies.

Instructions

[Gradle build cache](#) is a feature that enables the storage of the task outputs in the shared Gradle cache folder. Caching this folder in CI builds means that Gradle can reuse the task outputs from previous builds and can skip running the tasks when the inputs are unchanged. This is an opt-in feature. There are two ways to enable the build cache in a Gradle project:

- add **org.gradle.caching = true** to the **gradle.properties** file in the project.
- pass the **--build-cache** CLI flag to each Gradle execution.





If you choose the second option and use Bitrise Android Steps, there is a Step input for additional Gradle arguments where you can define **--build-cache**.

bitrise.yml

```
- restore-gradle-cache@1: {}
- android-build@1:
  inputs:
    - variant: debug
    - build_type: apk
    - arguments: --build-cache
- save-gradle-cache@1: {}
```

(Flutter) Cache Dart dependencies (Beta)

MODULAR

Description

Cache the contents of the [Dart pub system cache](#) folder with the new key-based caching Steps, **Save Dart Cache** and **Restore Dart Cache**.

Instructions

1. Add the [Restore Dart Cache](#) Step to the Workflow.
2. Add one of Flutter Steps to the workflow, such as [Flutter Build](#).
3. Add the [Save Dart Cache](#) Step.

Fine tune cache behaviour

The Dart specific cache Steps use optimal cache key and path configurations maintained by Bitrise. If you want full control over what should be cached then please check out the generic [Restore Cache](#) and [Save Cache](#) Steps. You can always check out what key and path settings the Dart cache Step uses: [Github code snippet](#).

bitrise.yml

```
- restore-dart-cache@1: {}
- flutter-build@0: {}
- save-dart-cache@1: {}
```

(React Native) Cache NPM dependencies (Beta)

MODULAR

Description

Cache the contents of the **node_modules** with the new key-based caching Steps, **Save Cache** and **Restore Cache**.

Instructions

1. Add the [Restore NPM Cache](#) Step to the Workflow.





2. Add either the [Run yarn command](#) Step or the [Run npm command](#) Step based on your project setup. Set the input variables:
 - Set the **The yarn command to run** or **The npm command with arguments to run** input to **install**.
3. Add the [Save NPM Cache](#) Step.

Fine tune cache behaviour

The NPM specific cache Steps use optimal cache key and path configurations maintained by Bitrise. If you want full control over what should be cached then please check out the generic [Restore Cache](#) and [Save Cache](#) Steps. You can always check out what key and path settings the NPM cache Step uses: [Github code snippet](#).

bitrise.yml

```
- restore-npm-cache@1: {}
- npm@1:
  inputs:
    - command: install
- save-npm-cache@1: {}
```

Advanced key-based cache recipes

MODULAR

These workflow recipes are based on the **Save cache** and **Restore cache** Steps. For recipes about the most popular platforms and dependency managers, check out the **Key-based caching** section in the [README](#).

Key templates

The **Save cache** and **Restore cache** Steps use a string key when uploading and downloading a cache archive. To always download the most relevant cache archive for each build, the **Cache key** input can contain template elements. The Steps evaluate the key template at runtime and the final key value can change based on the build environment or files in the repo.

The following variables are supported in the **Cache key** input:

- **cache-key-{{ .Branch }}**: Current git branch the build runs on.
- **cache-key-{{ .CommitHash }}**: SHA-256 hash of the git commit the build runs on.
- **cache-key-{{ .Workflow }}**: Current Bitrise workflow name (eg. **primary**).
- **{{ .Arch }}-cache-key**: Current CPU architecture (**amd64** or **arm64**).
- **{{ .OS }}-cache-key**: Current operating system (**linux** or **darwin**).

Functions available in a template:

checksum: This function takes one or more file paths and computes the SHA256 [checksum](#) of the file contents. This is useful for creating unique cache keys based on files that describe content to cache.

Examples of using **checksum**:

- **cache-key-{{ checksum "package-lock.json" }}**
- **cache-key-{{ checksum "**/Package.resolved" }}**
- **cache-key-{{ checksum "**/*.gradle*" "gradle.properties" }}**

getenv: This function returns the value of an environment variable or an empty string if the variable is not defined.

Examples of **getenv**:

- **cache-key-{{ getenv "PR" }}**
- **cache-key-{{ getenv "BITRISEIO_PIPELINE_ID" }}**

Skip saving the cache in PR builds (restore only)

If you want builds triggered by pull requests to only restore the cache and skip saving it, you can run the **Save cache** Step conditionally:

```
steps:
- restore-cache@2:
  inputs:
  - key: node-modules-{{ checksum "package-lock.json" }}

# Build steps

- save-cache@1:
  run_if: ".IsCI | and (not .IsPR)" # Condition that is
false in PR builds
  inputs:
  - key: node-modules-{{ checksum "package-lock.json" }}
  - paths: node_modules
```

Separate caches for each OS and architecture

Cache is not guaranteed to work across different Bitrise Stacks (different OS or same OS but different CPU architecture). If a Workflow runs on different stacks, it's a good idea to include the OS and architecture in the **Cache key** input:

```
steps:
- save-cache@1:
  inputs:
  - key: '{{ .OS }}-{{ .Arch }}-npm-cache-{{ checksum
"package-lock.json" }}
```

Multiple independent caches

You can add multiple instances of the cache Steps to a Workflow:



steps:

- save-cache@1:
 - title: Save NPM cache
 - inputs:
 - paths: node_modules
 - key: npm-cache-{{ checksum "package-lock.json" }}
- save-cache@1:
 - title: Save Python cache
 - inputs:
 - paths: venv/
 - key: pip-cache-{{ checksum "requirements.txt" }}

Cache warm-up for pull requests

Caching works best when the cached content is up to date and contains useful data for dependency managers and build systems. It's a good idea to run a Workflow periodically that builds the project from the latest code on the main branch and saves the result in the cache. This way, other builds triggered by pull requests can restore an up-to-date cache.

By including a checksum in the Cache key input, the Save cache Step will save multiple unique cache archives when the project files change (instead of overriding the previous cache). This way PRs not targeting the latest state of the main branch can still download a relevant cache archive.

workflows:

pr-validation:

steps:

- restore-cache@2:

inputs:

- key: |-

```
node-modules-{{ checksum "package-lock.
json" }}
```

```
node-modules-
```

```
# Rest of the PR validation workflow
```

cache-warm-up:

description: This Workflow should either be run on a scheduled basis or triggered by a push event on the main branch.

steps:

Build steps

- save-cache@1:

inputs:

```
- key: node-modules-{{ checksum "package-
lock.json" }}
```

- paths: node_modules/

Restore cache from the PR target branch

If you have a setup where the cache key is based on the current workflow (such as **cache-{{ .Workflow }}**), then you can configure the Restore Step with the following keys:

```
cache-{{ .Workflow }}
cache-{{ getenv "BITRISEIO_GIT_BRANCH_DEST" }}
cache-
```

The keys listed in the Step input are processed in priority order. If there is a cache entry for the exact same branch, the first rule will match that. You can also compute the cache key of the pull requests's target branch (such as **main** or **trunk**) via the **BITRISEIO_GIT_BRANCH_DEST** env var, which is automatically set for PR builds. Restoring the cache from the target branch can be useful when there are multiple long-lived branches and PRs are targeting different branches.

4.9 Optimisation

(Android) Turn on Gradle build profiling

Description

Generate and store a performance report of every Gradle build to spot build speed issues or compare different builds.

Instructions

No matter what Android or Gradle Step you use in your Bitrise Workflow, there is an option to define additional command line arguments for Gradle. Add **--profile** to this input to generate a performance report of the Gradle tasks. In the example below, we are adding the argument to the Android Unit Test Step.

To sum up the procedure:

1. Add the [Android Unit Test](#) Step to your Workflow.
2. Add a [Script](#) Step to compress the reports and copy the ZIP file to the deploy directory.
3. Trigger a manual build and download and open the HTML report.
4. Check the various aspects of the build in the report.

Adding the Android Unit Test Step

Add an [Android Unit Test](#) Step to your Workflow. Set the necessary input values:

- Project location: "\$PROJECT_LOCATION"
- Module: "\$MODULE"
- Variant: "\$VARIANT"
- Arguments: "--profile"

MODULAR



Compressing the report files and copying the ZIP file

Add a Script Step to the end of the Workflow in order to compress the report files and copy the ZIP file to the deploy directory:

```
#!/usr/bin/env bash
# fail if any commands fails
set -e
# debug log
set -x

zip -r $BITRISE_DEPLOY_DIR/gradle-profile.zip $PROJECT_
LOCATION/build/reports/profile
```

Gradle creates the HTML report in `build/reports/profile/`, so we need to take all files in that folder (HTML, CSS and JS files), compress them, and move the ZIP archive to `$BITRISE_DEPLOY_DIR`. Files in this folder can be accessed on the build page's Apps & Artifacts tab.

Downloading the report file

Trigger a manual build of the Workflow you edited previously. Download and unarchive **gradle-profile.zip**, then open the HTML report in your browser.

LOGS

APPS & ARTIFACTS

You can register your devices on your [Account Settings](#) page.

gradle-profile.zip (N/A)

Download

app-tests.zip (N/A)

Download

app-test-results.zip (N/A)

Download

Profile report

Profiled build: :app:testDebugUnitTest
Started on: 2021/02/16 - 11:09:06

Summary

Configuration

Dependency Resolution

Artifact Transforms

Task Execution

Description	Duration
Total Build Time	34.600s
Startup	1.902s
Settings and buildSrc	0.031s
Loading Projects	0.006s
Configuring Projects	0.655s
Artifact Transforms	3.217s
Task Execution	31.727s

Checking the build report

You can check various aspects of a build in the report:

- The **Summary** tab shows time spent on things other than task execution.
- The **Task execution** tab lists all tasks sorted by execution time.
- Cached tasks are marked as **UP-TO-DATE**. This helps to fine-tune the [Bitrise Cache Steps](#) by comparing the reports of multiple builds.





For Gradle optimization ideas, check out [this article by Google](#).

If you only want to display task execution times only in the build log, you can use the [build-time-tracker](#) project.

bitrise.yml

```
- android-unit-test@1:
  inputs:
    - project_location: $PROJECT_LOCATION
    - module: $MODULE
    - arguments: "--profile"
    - variant: $VARIANT
- script@1:
  title: Collect Gradle profile report
  inputs:
    - content: |-
        #!/usr/bin/env bash
        # fail if any commands fails
        set -e
        # debug log
        set -x

        zip -r $BITRISE_DEPLOY_DIR/gradle-profile.zip
        $PROJECT_LOCATION/build/reports/profile
  - deploy-to-bitrise-io@1: {}
```

4.10 Running Steps & Workflows

MODULAR

Start (parallel) builds from the Workflow

Description

Start one or more builds with specified Workflows from the parent Workflow and optionally wait for their completion.

Prerequisites

1. Make sure you have a valid Bitrise API key in your Secrets (**\$BITRISE_API_KEY**). See [Personal access](#) tokens for more details.
2. Have Workflow(s) you would like to run in parallel (**workflow-1** and **workflow-2** in the example).

Instructions

1. Add a [Bitrise Start Build](#) Step. Set the input variables:
 - **workflows:** The Workflow(s) to start. One Workflow per line.
 - **Bitrise Access Token:** **\$BITRISE_API_KEY**.
2. (Optional) Add any Step you would like to run in parallel in the parent Workflow while the triggered Workflows are running.
3. (Optional) Add a [Bitrise Wait for Build](#) Step. Set the input variables:
 - **Bitrise Access Token:** **\$BITRISE_API_KEY**.





With the Steps above you can only start a build for the same app. If you would like to start a build for an other app, you can use the Trigger Bitrise workflow Step.

bitrise.yml

```
parent-workflow:
  steps:
    - build-router-start@0:
      inputs:
        - workflows: |-
            workflow-1
            workflow-2
        - access_token: $BITRISE_API_KEY
    - script@1:
      inputs:
        - content: echo "Doing something else..."
    - build-router-wait@0:
      inputs:
        - access_token: $BITRISE_API_KEY
```



Chapter

5

Technology Specific Recipes

5.1 iOS

PLUG & PLAY

(iOS) Pull request

Description

Example Workflow for iOS Pull Request validation.
The Workflow contains:

1. Installing [Cocoapods](#) and [Carthage](#) dependencies.
2. [Running all unit and UI tests on simulator](#).
3. [Building a test app and uploading to bitrise.io](#).
4. [Sending the QR code of the test build to the Pull Request](#).
5. Triggering the workflow for Pull Requests.

bitrise.yml

```
---
format_version: '13'
default_step_lib_source: https://github.com/bitrise-io/
bitrise-steplib.git
project_type: ios

meta:
  bitrise.io:
    stack: osx-xcode-15.0.x
    machine_type_id: g2-m1.4core

workflows:
  pull-request:
    steps:
      - activate-ssh-key@4:
          run_if: '{{getenv "SSH_RSA_PRIVATE_KEY" | ne ""}}'
      - git-clone@8: {}
      - restore-cocoapods-cache@1: {}
      - cocoapods-install@2: {}
      - restore-carthage-cache@1: {}
      - carthage@3:
          inputs:
            - carthage_options: "--use-xcframeworks"
      --platform ios"
      - restore-spm-cache@1: {}
      - xcode-test@5:
          inputs:
            - log_formatter: xcodebuild
            - xcodebuild_options: "--enableCodeCoverage YES"
      - xcode-archive@5:
          inputs:
            ...

PLEASE LOOK AT GITHUB PAGE FOR THE REST OF BITRISE.YML
```



(iOS) CI

Description

Example Workflow for commits on the main branch of an iOS app.
The Workflow contains:

1. Installing [Cocoapods](#) and [Carthage](#) dependencies.
2. [Running all unit and UI tests on simulator](#).
3. [Building a test app and uploading to bitrise.io](#).
4. [Sending a Slack notification with the build status](#).

bitrise.yml

```

---
format_version: '13'
default_step_lib_source: https://github.com/bitrise-io/
bitrise-steplib.git
project_type: ios

meta:
  bitrise.io:
    stack: osx-xcode-15.0.x
    machine_type_id: g2-m1.4core

workflows:
  ci:
    steps:
      - activate-ssh-key@4:
          run_if: '{{getenv "SSH_RSA_PRIVATE_KEY" | ne ""}}'
      - git-clone@8: {}
      - restore-cocoapods-cache@1: {}
      - cocoapods-install@2: {}
      - restore-carthage-cache@1: {}
      - carthage@3:
          inputs:
            - carthage_options: "--use-xcframeworks"
      --platform iOS"
      - restore-spm-cache@1: {}
      - xcode-test@5:
          inputs:
            - log_formatter: xcodebuild
            - xcodebuild_options: "-enableCodeCoverage YES"
      - xcode-archive@5:
          inputs:
            - project_path: $BITRISE_PROJECT_PATH
            - scheme: $BITRISE_SCHEME
            - automatic_code_signing: apple-id
      ...
PLEASE LOOK AT GITHUB PAGE FOR THE REST OF BITRISE.YML

```





(iOS) Nightly

Description

Example Workflow for nightly builds for iOS apps.

The Workflow contains:

1. Installing [Cocoapods](#) and [Carthage](#) dependencies.
2. [Setting the version and build number](#). By default, the app will get the build number (`$BITRISE_BUILD_NUMBER`) as the version code.
3. [Building a release build and uploading to TestFlight](#).
4. [Building a test app and uploading to bitrise.io](#).
5. [Sending the QR code of the test build to the Pull Request](#).

Check out the [guide](#) to run scheduled builds.

bitrise.yml

```
---
format_version: '13'
default_step_lib_source: https://github.com/bitrise-io/
bitrise-steplib.git
project_type: ios

meta:
  bitrise.io:
    stack: osx-xcode-15.0.x
    machine_type_id: g2-m1.4core

workflows:
  nightly:
    steps:
      - activate-ssh-key@4:
          run_if: '{{getenv "SSH_RSA_PRIVATE_KEY" | ne ""}}'
      - git-clone@8: {}
      - cocoapods-install@2: {}
      - carthage@3:
          inputs:
            - carthage_options: "--use-xcframeworks
--platform iOS"
      - set-xcode-build-number@1:
          inputs:
            - build_short_version_string: '1.0'
            - plist_path: BitriseTest/Info.plist
      - xcode-archive@5:
          inputs:
```

...

PLEASE LOOK AT GITHUB PAGE FOR THE REST OF BITRISE.YML

(iOS) Release

Description

Example Workflow for uploading a release draft of an iOS app to the App Store. The Workflow contains:

1. Installing [Cocoapods](#) and [Carthage](#) dependencies.
2. [Setting the version number](#) based on [env vars passed to build \(\\$VERSION_NUMBER\)](#).
3. [Building a release build and uploading to App Store](#).

bitrise.yml

```
---
format_version: '13'
default_step_lib_source: https://github.com/bitrise-io/
bitrise-steplib.git
project_type: ios

meta:
  bitrise.io:
    stack: osx-xcode-15.0.x
    machine_type_id: g2-m1.4core

workflows:
  release:
    steps:
      - activate-ssh-key@4:
          run_if: '{{getenv "SSH_RSA_PRIVATE_KEY" | ne ""}}'
      - git-clone@8: {}
      - restore-cocoapods-cache@1: {}
      - carthage@3:
          inputs:
            - carthage_options: "--use-xcframeworks"
      --platform iOS"
      - set-xcode-build-number@1:
          inputs:
            - build_short_version_string: $VERSION_NUMBER
            - build_version: $BITRISE_BUILD_NUMBER
            - plist_path: BitriseTest/Info.plist
      - recreate-user-schemes@1:
          inputs:
            - project_path: $BITRISE_PROJECT_PATH
      - xcode-archive@5:
          inputs:
            - project_path: $BITRISE_PROJECT_PATH

...

PLEASE LOOK AT GITHUB PAGE FOR THE REST OF BITRISE.YML
```



(iOS) Run tests in parallel on multiple simulators

Description

This example uses the [sample-swift-project-with-parallel-ui-test](#) iOS Open Source sample app, which has some example Unit and UI tests and uses Test Plans to group the tests.

The example Pipeline config showcases how to run all the test cases of the project on different iOS simulators.

build_and_run_tests Pipeline runs two Stages sequentially:

1. **build_tests** Stage that runs the **xcode_build_for_test** Workflow. This Workflow git clones the sample project and runs the **xcode-build-for-test** Step to build the target and associated tests. The built test bundle is transferred to the next Stage (**run_tests_on_simulators**) via the **deploy-to-bitrise-io** Step.
Note: **xcode-build-for-test** Step compresses the built test bundle and moves the generated zip to the **\$BITRISE_DEPLOY_DIR**, that directory's content is deployed to the Workflow artifacts by default via the **deploy-to-bitrise-io** Step.
2. **run_tests** Stage runs three Workflows in parallel: **run_tests_on_iphone**, **run_tests_on_ipad**, and **run_tests_on_ipod**. Both of these Workflows use the new **xcode-test-without-building** Step, which executes the tests based on the previous Stage built test bundle. The pre-built test bundle is pulled by the **_pull_test_bundle** utility Workflow.

Instructions

1. Visit the [Create New App page](#) to create a new App.
2. When prompted to select a git repository, choose **Other/Manual** and paste the sample project repository URL (<https://github.com/bitrise-io/sample-swift-project-with-parallel-ui-test>) in the **Git repository (clone) URL** field.
3. Confirm that this is a public repository in the resulting pop-up.
4. Select the **master** branch to scan.
5. Wait for the project scanner to complete.
6. Select any of the offered Distribution methods (for example development, it does not really matter as now we are focusing on testing).
7. Confirm the offered stack, skip choosing the app icon and the webhook registration and kick off the first build.
8. Open the new Bitrise project's Workflow Editor.
9. Go to the **bitrise.yml** tab and replace the existing **bitrise.yml** with the contents of the example **bitrise.yml** below.
10. Click the **Start/Schedule a Build** button, and select the **build_and_run_tests** option in the **Workflow, Pipeline** dropdown menu at the bottom of the popup.



bitrise.yml

```

---
format_version: '13'
default_step_lib_source: https://github.com/bitrise-io/
bitrise-steplib.git
project_type: ios

app:
  envs:
    - BITRISE_PROJECT_PATH: BullsEye.xcworkspace
    - BITRISE_SCHEME: BullsEye

meta:
  bitrise.io:
    stack: osx-xcode-13.2.x

pipelines:
  build_and_run_tests:
    stages:
      - build_tests: {}
      - run_tests: {}

stages:
  build_tests:
    workflows:
      - xcode_build_for_test: {}

  run_tests:
    workflows:
      - run_tests_on_iPhone: {}
      - run_tests_on_iPad: {}
      - run_tests_on_iPod: {}

workflows:
  xcode_build_for_test:
    steps:
      - git-clone@8: {}
      - xcode-build-for-test@3:
          inputs:
            - destination: generic/platform=iOS Simulator
      - deploy-to-bitrise-io@2:
          inputs:
            - pipeline_intermediate_files: "$BITRISE_TEST_
BUNDLE_PATH:BITRISE_TEST_BUNDLE_PATH"

  run_tests_on_iPhone:
    ...

```

PLEASE LOOK AT GITHUB PAGE FOR THE REST OF BITRISE.YML



(iOS) Run test groups in parallel

Description

This example uses the [sample-swift-project-with-parallel-ui-test](#) iOS Open Source sample app, which has some example Unit and UI tests and uses Test Plans to group the tests.

Note: Xcode Test Plans provide a way to run a collection of tests with different test configurations. [raywenderlich.com](#) has a great [tutorial on how to get started with Xcode Test Plans](#).

The example Pipeline config showcases how to run all the test cases of the project on different iOS simulators.

build_and_run_tests Pipeline runs two Stages sequentially:

1. **build_tests** Stage that runs the **xcode_build_for_test** Workflow. This Workflow git clones the sample project and runs the **xcode-build-for-test** Step to build the target and associated tests. The built test bundle is transferred to the next Stage (**run_tests**) via the **deploy-to-bitrise-io** Step.
2. **run_tests** Stage runs two Workflows in parallel: **run_ui_tests** and **run_unit_tests**. Both of these Workflows use the new **xcode-test-without-building** Step, which executes the tests based on the previous Stage built test bundle. The pre-built test bundle is pulled by the **_pull_test_bundle** utility Workflow.

Instructions

1. Visit the [Create New App page](#) to create a new App.
2. When prompted to select a git repository, choose **Other/Manual** and paste the sample project repository URL (<https://github.com/bitrise-io/sample-swift-project-with-parallel-ui-test>) in the **Git repository (clone) URL** field.
3. Confirm that this is a public repository in the resulting pop-up.
4. Select the **master** branch to scan.
5. Wait for the project scanner to complete.
6. Select any of the offered Distribution methods (for example development, it does not really matter as now we are focusing on testing).
7. Confirm the offered stack, skip choosing the app icon and the webhook registration and kick off the first build.
8. Open the new Bitrise project's Workflow Editor.
9. Go to the **bitrise.yml** tab and replace the existing **bitrise.yml** with the contents of the example **bitrise.yml** below.
10. Click the **Start/Schedule a Build** button, and select the **build_and_run_tests** option in the **Workflow, Pipeline** dropdown menu at the bottom of the popup.



bitrise.yml

```

---
format_version: '13'
default_step_lib_source: https://github.com/bitrise-io/
bitrise-steplib.git
project_type: ios

app:
  envs:
    - BITRISE_PROJECT_PATH: BullsEye.xcworkspace
    - BITRISE_SCHEME: BullsEye

meta:
  bitrise.io:
    stack: osx-xcode-13.2.x

pipelines:
  build_and_run_tests:
    stages:
      - build_tests: {}
      - run_tests: {}

stages:
  build_tests:
    workflows:
      - xcode_build_for_test: {}

  run_tests:
    workflows:
      - run_ui_tests: {}
      - run_unit_tests: {}

workflows:
  xcode_build_for_test:
    steps:
      - git-clone@8: {}
      - xcode-build-for-test@3:
        inputs:
          - destination: generic/platform=iOS Simulator
      - deploy-to-bitrise-io@2:
        inputs:
          - pipeline_intermediate_files: "$BITRISE_TEST_
BUNDLE_PATH:BITRISE_TEST_BUNDLE_PATH"

  run_ui_tests:
    before_run:

...
PLEASE LOOK AT GITHUB PAGE FOR THE REST OF BITRISE.YML

```



PLUG & PLAY

(iOS) Merging test results and deploying to the Test Reports add-on

Description

Test Reports add-on is tied to Bitrise builds. To make all the test reports generated in different builds appear on a single page in the add-on, the reports need to be merged and deployed in an additional build.

This example uses the [sample-swift-project-with-parallel-ui-test](#) iOS Open Source sample app, which has some example Unit and UI tests and uses Test Plans to group the tests.

run_ui_tests and **run_unit_tests** Workflows are extended with a **deploy-to-bitrise-io** Step to make the generated test results available for the next Stage.

build_and_run_tests Pipeline is extended with a new Stage: **deploy_test_results**.

This Stage runs the **deploy_test_results** Workflow:

1. **pull-intermediate-files** Step downloads the previous stage (**run_tests**) generated test results.
2. **script** Step moves each test result into a new test run directory within the Test Report add-on deploy dir and creates the related **test-info.json** file.
3. **deploy-to-bitrise-io** Step deploys the merged test results.

Instructions

1. Visit the [Create New App page](#) to create a new App.
2. When prompted to select a git repository, choose **Other/Manual** and paste the sample project repository URL (<https://github.com/bitrise-io/sample-swift-project-with-parallel-ui-test>) in the **Git repository (clone) URL** field.
3. Confirm that this is a public repository in the resulting pop-up.
4. Select the **master** branch to scan.
5. Wait for the project scanner to complete.
6. Select any of the offered Distribution methods (for example development, it does not really matter as now we are focusing on testing).
7. Confirm the offered stack, skip choosing the app icon and the webhook registration and kick off the first build.
8. Open the new Bitrise project's Workflow Editor.
9. Go to the **bitrise.yml** tab and replace the existing **bitrise.yml** with the contents of the example **bitrise.yml** below.
10. Click the **Start/Schedule a Build** button, and select the **build_and_run_tests** option in the **Workflow, Pipeline** dropdown menu at the bottom of the popup.
11. Open the Pipeline's build page.



12. Select the **deploy_test_results** build.

13. Click on **Details & Add-ons** on the build details page and select the Test Reports add-on to view the merged test reports.

bitrise.yml

```
---
format_version: '13'
default_step_lib_source: https://github.com/bitrise-io/
bitrise-steplib.git
project_type: ios

app:
  envs:
    - BITRISE_PROJECT_PATH: BullsEye.xcworkspace
    - BITRISE_SCHEME: BullsEye

meta:
  bitrise.io:
    stack: osx-xcode-13.2.x

pipelines:
  build_and_run_tests:
    stages:
      - build_tests: {}
      - run_tests: {}
      - deploy_test_results: {}

  stages:
    build_tests:
      workflows:
        - xcode_build_for_test: {}

    run_tests:
      workflows:
        - run_ui_tests: {}
        - run_unit_tests: {}

    deploy_test_results:
      workflows:
        - merge_and_deploy_test_results: {}

  workflows:
    xcode_build_for_test:
      steps:
        - git-clone@8: {}
        - xcode-build-for-test@3: {}

...
PLEASE LOOK AT GITHUB PAGE FOR THE REST OF BITRISE.YML
```

5.2 Android



PLUG & PLAY

(Android) Pull request

Description

Example Workflow for Android Pull Request validation.
The Workflow contains:

1. [Running unit tests.](#)
2. [Running UI tests on a virtual device.](#)
3. [Running lint.](#)
4. [Building a test app and uploading to bitrise.io.](#)
5. [Sending the QR code of the test build to the Pull Request.](#)
6. Triggering the Workflow for pull requests.

Instructions

Copy the yaml contents from below and make sure that the following env vars have the correct settings:

- **\$PROJECT_LOCATION**
- **\$MODULE**
- **\$VARIANT**

Also generate a new Github access token and add a new secret called **GITHUB_ACCESS_TOKEN** with the newly generated token value.

bitrise.yml

```
---
format_version: '13'
default_step_lib_source: https://github.com/bitrise-io/
bitrise-steplib.git
project_type: android

meta:
  bitrise.io:
    stack: linux-docker-android-20.04
    machine_type_id: standard

workflows:
  pull-request:
    steps:
      - activate-ssh-key@4:
          run_if: '{{getenv "SSH_RSA_PRIVATE_KEY" | ne ""}}'
      - git-clone@8: {}
      - restore-gradle-cache@1: {}
      - android-unit-test@1: {}
    ...
PLEASE LOOK AT GITHUB PAGE FOR THE REST OF BITRISE.YML
```

(Android) CI

Description

Example Workflow for Android Pull Request validation.
The Workflow contains:

1. [Running unit tests.](#)
2. [Running UI tests on a virtual device.](#)
3. [Running lint.](#)
4. Building a test app.
5. [Sending a Slack notification with the build status.](#)

Instructions

Use the yaml below and change the following env var values to match your project settings:

- **\$PROJECT_LOCATION**
- **\$MODULE**
- **\$VARIANT**
- **\$SLACK_WEBHOOK**

bitrise.yml

```
---
format_version: '13'
default_step_lib_source: https://github.com/bitrise-io/
bitrise-steplib.git
project_type: android

meta:
  bitrise.io:
    stack: linux-docker-android-20.04
    machine_type_id: standard

workflows:
  ci:
    steps:
      - activate-ssh-key@4:
          run_if: '{{getenv "SSH_RSA_PRIVATE_KEY" | ne ""}}'
      - git-clone@8: {}
      - restore-gradle-cache@1: {}
      - android-unit-test@1:
          inputs:
            - project_location: $PROJECT_LOCATION

...
PLEASE LOOK AT GITHUB PAGE FOR THE REST OF BITRISE.YML
```





(Android) Nightly

Description

Example workflow for Android nightly builds. The workflow contains:

1. [Setting the version code and version name](#). By default the app will get the build number (`$BITRISE_BUILD_NUMBER`) as the version code.
2. [Building a release Android App Bundle and uploading to Google Play internal testing](#).
3. [Building a test app and uploading to bitrise.io](#).
4. [Sending the QR code of the test build to Slack](#).

Check out the [guide](#) to run scheduled builds.

Prerequisites

1. An Android keystore file is uploaded to Bitrise. For details, see [Android code signing using the Android Sign Step](#).
2. Google Play API Access is set up. For details, see [Deploying Android apps to Bitrise and Google Play](#).

bitrise.yml

```
---
format_version: '13'
default_step_lib_source: https://github.com/bitrise-io/
bitrise-steplib.git
project_type: android

meta:
  bitrise.io:
    stack: linux-docker-android-20.04
    machine_type_id: standard

workflows:
  nightly:
    steps:
      - activate-ssh-key@4:
          run_if: '{{getenv "SSH_RSA_PRIVATE_KEY" | ne ""}}'
      - git-clone@8: {}
      - restore-gradle-cache@1: {}
      - change-android-versioncode-and-versionname@1:
          inputs:
            - new_version_name: 1.0.0
            - build_gradle_path: "$PROJECT_LOCATION/$MODULE/
build.gradle"

...
PLEASE LOOK AT GITHUB PAGE FOR THE REST OF BITRISE.YML
```

(Android) Release

Description

Example workflow for uploading a release draft of an app to Google Play. The workflow contains:

1. [Setting the version name](#) based on [Env Vars passed to the build \(\\$VERSION_NAME\)](#).
2. [Creating a release Android App Bundle and uploading it to Google Play](#).

Prerequisites

1. An Android keystore file is uploaded to Bitrise. For details, see [Android code signing using the Android Sign Step](#).
2. Google Play API Access is set up. For details, see [Deploying Android apps to Bitrise and Google Play](#).

Instructions

Copy the yaml contents from below and make sure that the following env vars have the correct settings:

- **\$PROJECT_LOCATION**
- **\$MODULE**
- **\$VARIANT**

This workflow will require setting the **\$VERSION_NAME** env var for the build. Follow this [guide](#) on how to do it.

bitrise.yml

```
---
format_version: '13'
default_step_lib_source: https://github.com/bitrise-io/
bitrise-steplib.git
project_type: android

meta:
  bitrise.io:
    stack: linux-docker-android-20.04
    machine_type_id: standard

workflows:
  release:
    steps:
      - activate-ssh-key@4:
          run_if: '{{getenv "SSH_RSA_PRIVATE_KEY" | ne ""}}'
      - git-clone@8: {}
    ...
PLEASE LOOK AT GITHUB PAGE FOR THE REST OF BITRISE.YML
```




(Android) Parallel testing of unit test shards by module

Description

Running the unit tests of a modularized app in parallel Workflows utilizing Pipelines.

This Pipeline contains one Stage — `stage_unit_test` — that executes two Workflows in parallel:

1. **unit_test_app**: This Workflow runs the unit tests of the **app** module using the **android-unit-test** Step.
2. **unit_test_library**: This Workflow runs the unit tests of the **lib-example** module using the **android-unit-test** Step.

Instructions

To test this configuration in a new Bitrise example project, do the following:

1. Visit the [Create New App page](#) to create a new App.
2. When prompted to select a git repository, choose Other/Manual and paste the sample project repository URL (<https://github.com/bitrise-io/Bitrise-Android-Modules-Sample.git>) in the **Git repository (clone) URL** field.
3. Confirm that this is a public repository in the resulting pop-up.
4. Select the **main** branch to scan.
5. Wait for the project scanner to complete.
6. Enter **app** as the specified module.
7. Enter **debug** as the specified variant.
8. Continue through the prompts as normal — no changes are needed.
9. Open the new Bitrise project's Workflow Editor.
10. Go to the **bitrise.yml** tab, and replace the existing yaml contents with the contents of the example **bitrise.yml** below.
11. Click the **Start/Schedule a Build** button, and select the **pipeline_unit_test** option in the Workflow, Pipeline dropdown menu at the bottom of the popup.

bitrise.yml

```
format_version: "13"
default_step_lib_source: https://github.com/bitrise-io/
bitrise-steplib.git
project_type: android
```

...

PLEASE LOOK AT GITHUB PAGE FOR THE REST OF BITRISE.YML





PLUG & PLAY

(Android) Parallel UI tests on multiple devices

Description

Running the UI or instrumented tests of a single module in parallel Workflows utilizing Pipelines. You can run the tests in parallel by shards or by devices.

The Pipeline contains two Stages that are run serially:

1. **build_tests:** This Stage executes a Workflow — named **build_for_ui_testing** — that runs the **android-build-for-ui-testing** Step to build APKs for use in testing, and runs the **deploy-to-bitrise-io** Step to save those APKs for use in the later Stages. Performing this Stage separately from the actual testing allows for each test Stage to use these pre-built APKs rather than having to rebuild them for each test Stage.
2. **run_tests:** This Stage executes three UI test Workflows in parallel — **ui_test_on_phone**, **ui_test_on_tablet**, **ui_test_on_foldable** — which use the **android-instrumented-test** Step to run the UI tests on the APKs built in the previous Workflow on each specific device type.

Instructions

To test this configuration in a new Bitrise example project, do the following:

1. Visit the [Create New App page](#) to create a new App.
2. When prompted to select a git repository, choose Other/Manual and paste the sample project repository URL (<https://github.com/bitrise-io/Bitrise-Android-Modules-Sample.git>) in the **Git repository (clone) URL** field.
3. Confirm that this is a public repository in the resulting pop-up.
4. Select the **main** branch to scan.
5. Wait for the project scanner to complete.
6. Enter **app** as the specified module.
7. Enter **debug** as the specified variant.
8. Continue through the prompts as normal — no changes are needed.
9. Open the new Bitrise project's Workflow Editor.
10. Go to the **bitrise.yml** tab, and replace the existing yaml contents with the contents of the example **bitrise.yml** below.
11. Click the **Start/Schedule** a Build button, and select the **ui_test_on_multiple_devices** option in the Workflow, Pipeline dropdown menu at the bottom of the popup.

bitrise.yml

```

format_version: "13"
default_step_lib_source: https://github.com/bitrise-io/
bitrise-steplib.git
project_type: android

meta:
  bitrise.io:
    stack: linux-docker-android-20.04
    machine_type_id: standard

pipelines:
  ui_test_on_multiple_devices:
    stages:
      - build_tests: {}
      - run_rests: {}

stages:
  build_tests:
    workflows:
      - build_for_ui_testing: {}

  run_rests:
    workflows:
      - ui_test_on_phone: {}
      - ui_test_on_tablet: {}
      - ui_test_on_foldable: {}

workflows:
  build_for_ui_testing:
    steps:
      - git-clone@8: {}
      - android-build-for-ui-testing@0:
          inputs:
            - module: app
            - variant: debug
      - deploy-to-bitrise-io@2:
          inputs:
            - pipeline_intermediate_files: |-
                $BITRISE_APK_PATH:BITRISE_APK_PATH
                $BITRISE_TEST_APK_PATH:BITRISE_TEST_APK_PATH

  ui_test_on_phone:
    envs:
      - EMULATOR_PROFILE: pixel_5
    before_run:
      - _pull_apks
    after_run:

```

...

PLEASE LOOK AT GITHUB PAGE FOR THE REST OF BITRISE.YML

(Android) Parallel unit and UI tests

PLUG & PLAY

Description

Running unit tests and UI tests in parallel utilizing Pipelines.

The Pipeline contains one Stage — **stage_unit_and_ui_test** — that executes two Workflows in parallel:

1. **unit_tests**: This Workflow simply runs the unit tests of the given module and variant using the **android-unit-test** Step.
2. **ui_tests**: This Workflow builds the given module and variant using the **android-build-for-ui-testing** Step, spins up an emulator using the **avd-manager** Step, waits for the emulator to boot using the **wait-for-android-emulator** Step, and runs the UI tests using the **android-instrumented-test** Step.

Instructions

To test this configuration in a new Bitrise example project, do the following:

1. Visit the [Create New App page](#) to create a new App.
2. When prompted to select a git repository, choose Other/Manual and paste the sample project repository URL (<https://github.com/bitrise-io/Bitrise-Android-Modules-Sample.git>) in the **Git repository (clone) URL** field.
3. Confirm that this is a public repository in the resulting pop-up.
4. Select the **main** branch to scan.
5. Wait for the project scanner to complete.
6. Enter **app** as the specified module.
7. Enter **debug** as the specified variant.
8. Continue through the prompts as normal — no changes are needed.
9. Open the new Bitrise project's Workflow Editor.
10. Go to the **bitrise.yml** tab, and replace the existing yaml contents with the contents of the example **bitrise.yml** below.
11. Click the **Start/Schedule** a Build button, and select the **pipeline_unit_and_ui_test** option in the Workflow, Pipeline dropdown menu at the bottom of the popup.

bitrise.yml

```
format_version: "13"
default_step_lib_source: https://github.com/bitrise-io/
bitrise-steplib.git
project_type: android
```

```
meta:
```

```
...
```

```
PLEASE LOOK AT GITHUB PAGE FOR THE REST OF BITRISE.YML
```



5.3 Other

Create Gitflow release branch

Description

An example Workflow that creates a Gitflow release branch for a specific version. The version can be passed as an Environment Variable for the Workflow.

Prerequisites

Make sure that Bitrise has write access to your repository. You need to [manually add an SSH key](#) with write permission on GitHub.

bitrise.yml

```
# Run the workflow with $VERSION env set up to, for
example, '2.4.3'
create-release-branch:
  steps:
    - activate-ssh-key@4:
        run_if: '{{getenv "SSH_RSA_PRIVATE_KEY" | ne ""}}'
    - git-clone@6: {}
    - script@1:
        inputs:
          - content: |-
              #!/usr/bin/env bash
              # fail if any commands fails
              set -e
              # debug log
              set -x

              git checkout -b release-$VERSION
              git push origin release-$VERSION
```

Additional Resources

[Case studies](#)

[Support Center](#)

[DevCenter](#)

[Book a demo](#)



bitrise

