

How to set up a CI/CD pipeline for your iOS app using fastlane and GitHub Actions

APP DEVELOPMENT

fastlane iOS



Isabel Barrera

MARCH 28, 2024

Prerequisites

Steps

Install fastlane and set up your Fastfile

Set up your GitHub Actions workflow .yaml file

Running your build

How to generate an iOS CI/CD pipeline automatically

So you've built an iOS app and deployed a bunch of updates to your users manually from your laptop. Maybe your team is growing, and being "*the* release person" – the only one with the required keys and ability to archive and ship builds – is getting a bit old. Maybe your team is also realizing that it's an unnecessary bottleneck and single point of failure and that *it's time* to automate your build process so that anyone can create App Store builds, confidently and without breaking anything. Enter CI/CD.

But spinning up a build pipeline isn't straightforward, and if you're a busy, growing team you probably can't afford to spend a week figuring out which CI provider is the best and poring over documentation to cobble together a functional workflow. We're here to help! We've done this before, and have collected the out-of-the-box building blocks that will save you having to waste time figuring out all the pieces and reinventing the wheel.

In this tutorial, we'll spin up a complete iOS build & upload pipeline in just 30 minutes using [fastlane](#) and [GitHub Actions](#).

Let's get started!

Prerequisites

First off, this tutorial will assume a few things:

1. You already have an app on the App Store
2. Your app is a vanilla native iOS app (you'd need to make a few tweaks to get things working for a React Native app, for example)

Steps

1. Install fastlane and set up your Fastfile
2. Configure your secrets in GitHub's encrypted secrets
3. Set up a basic GitHub Actions workflow `.yaml` file
4. Run your build!

Install fastlane and set up your Fastfile

fastlane is a Ruby library created to automate common mobile development tasks. Using fastlane, you can configure custom “lanes” which bundle a series of “actions” that perform tasks that you’d normally perform using Xcode or xcodebuild. You can do a lot with fastlane, but for the purposes of this tutorial, we’ll be using only a handful of core actions.

Install fastlane using one of the [recommended installation methods](#). Here we’ll use Bundler, running `brew install fastlane` from our main app project directory.

Then, we’ll run `fastlane init` and choose the “manual setup” option when prompted.

You’ll end up with a fastlane directory, a Fastfile, and an Appfile.

We’ll first modify our Appfile, filling in our app’s bundle identifier and removing everything else:

```
fastlane > Appfile
1  app_identifier("com.example.example")
2
```

Next, we’ll set up some basic actions in our Fastfile.

First, we’ll need to load in our App Store Connect API key. This is how we’ll authenticate with the [App Store Connect API](#) so we can download provisioning profiles, upload binaries to TestFlight, and more. We’ll go over how to create your API key and add it to GitHub’s encrypted secrets later on in this tutorial.

Add the `:load_asc_api_key` lane to your Fastfile:

```
platform :ios do
  desc "Load ASC API Key information to use in subsequent lanes"
  lane :load_asc_api_key do
    app_store_connect_api_key(
      key_id: ENV["ASC_KEY_ID"],
      issuer_id: ENV["ASC_ISSUER_ID"],
      key_content: ENV["ASC_KEY"],
      is_key_content_base64: true,
      in_house: false # detecting this via ASC private key not currently supported
    )
  end
end
```

Product

Integrations

Automations

Customers

Pricing

LOG IN

CONTACT SALES

GET STARTED

```

desc "Bump build number based on most recent TestFlight build number"
lane :fetch_and_increment_build_number do
  #fetch read your app identifier defined in your Appfile
  app_identifier = CredentialsManager::AppfileConfig.try_fetch_value(:app_identifier)
  api_key = lane_context[SharedValues::APP_STORE_CONNECT_API_KEY]

  current_version = get_version_number(
    target: "main-target" # replace with your main target, required if you have more than one non-test
  )
  latest_build_number = latest_testflight_build_number(
    api_key: api_key,
    version: current_version,
    app_identifier: app_identifier
  )
  increment_build_number(
    build_number: (latest_build_number + 1),
  )
end

desc "Installs signing certificate in the keychain and downloads provisioning profiles from App Store Connect"
lane :prepare_signing do |options|
  team_id = CredentialsManager::AppfileConfig.try_fetch_value(:team_id)
  api_key = lane_context[SharedValues::APP_STORE_CONNECT_API_KEY]

  keychain_name = "signing"
  keychain_password = "temp"

  delete_keychain(
    name: keychain_name
  ) if File.exist? File.expand_path("~/Library/Keychains/#{keychain_name}-db")

  create_keychain(
    name: keychain_name,
    password: keychain_password,
    default_keychain: true,
    unlock: true,
    timeout: 3600
  )

  import_certificate(
    certificate_path: ENV["SIGNING_KEY_FILE_PATH"],
    certificate_password: ENV["SIGNING_KEY_PASSWORD"],
    keychain_name: keychain_name,
    keychain_password: keychain_password
  )

  # fetches and installs provisioning profiles from ASC
  sigh(
    adhoc: options[:adhoc],
    api_key: api_key,
    readonly: true
  )
end

desc "Build the iOS app for release"
lane :build_release do |options|
  app_identifier = CredentialsManager::AppfileConfig.try_fetch_value(:app_identifier)

  profile_name = "App Provisioning Profile" # replace with the name of the profile to use for the build
  output_name = "example-iOS" # specify the name of the .ipa file to generate
  export_method = "app-store" # specify the export method

  # turn off automatic signing during build so correct code signing identity is guaranteed to be used
  update_code_signing_settings(
    use_automatic_signing: false,

    targets: ["main-target"], # specify which targets to update code signing settings for
  )
end

```

```
# build the app
gym(
  scheme: "example-scheme", # replace with name of your project's scheme
  output_name: output_name,
  configuration: "Release",
  export_options: {
    method: export_method,
    provisioningProfiles: {
      app_identifier => profile_name # here you can add any additional bundle identifiers and their
    }
  }
)
end

desc "Upload to TestFlight / ASC"
lane :upload_release do
  api_key = lane_context[SharedValues::APP_STORE_CONNECT_API_KEY]

  deliver(
    api_key: api_key,
    skip_screenshots: true,
    skip_metadata: true,
    skip_app_version_update: true,
    force: true, # skips verification of HTML preview file (since this will be run from a CI machin
    run_precheck_before_submit: false # not supported through ASC API yet
  )
end
```

Once you have these lanes set up, you can tie them all together in a single “build and upload” lane that we’ll call from the CI workflow:

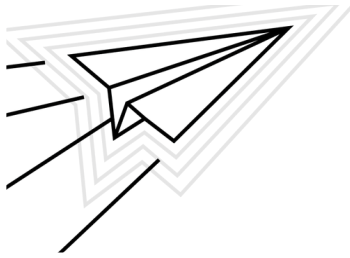
```
desc "Build and upload to TestFlight"
lane :build_upload_testflight do
  load_asc_api_key
  prepare_signing
  fetch_and_increment_build_number
  build_release
  upload_release
end
```

Let’s break it down. The first thing we do is load the ASC API key that is used in all subsequent fastlane actions so we can authenticate and use the ASC API. Then, we’ll call the “prepare_signing” lane to load the signing certificate into a keychain on the CI machine and download provisioning profiles from the ASC API.

We’ll then increment the build number based on the latest build number found on TestFlight for the version. Finally we’re ready to build the app. “build_release” will build the Xcode project in the “Release” configuration using the distribution provisioning profile & signing certificate, and the “app-store” distribution method.

Lastly, “upload_release” will take the .ipa file generated by the “build_release” action and upload it to TestFlight.

And that’s it! With these six actions, fastlane will handle bumping the build number, preparing everything needed for signing builds, building, and uploading binaries to App Store Connect. Now, let’s hook it up to GitHub Actions.

[Product](#)[Integrations](#)[Automations](#)[Customers](#)[Pricing](#)[LOG IN](#)[CONTACT SALES](#)[GET STARTED](#)

We'll share our perspectives on the mobile landscape, peeks into how other mobile teams and developers get things done, technical guides to optimizing your app for performance, and more. ([See a recent issue here](#))

[SIGN UP](#)

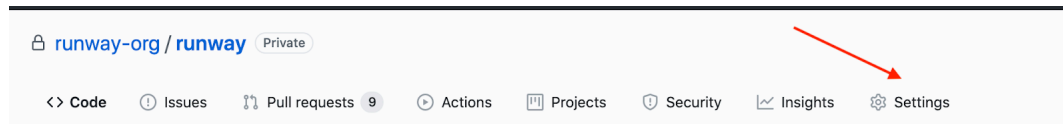
Storing your secrets

In order to authenticate with the ASC API, we'll need an API key along with a couple of other details. These are considered sensitive, which means we'll need to store them securely in a place where they can be accessed by our GitHub workflows. Enter GitHub's encrypted secrets: we'll be storing all our sensitive keys in repository secrets, making them automatically accessible to our GitHub Actions workflows.

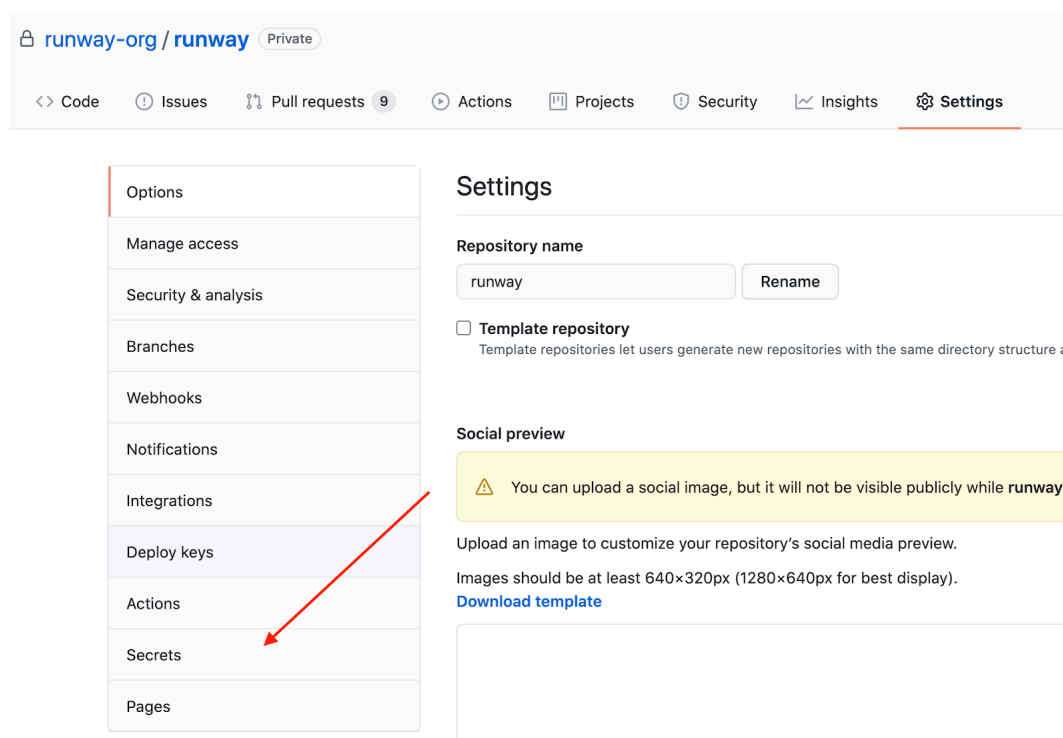
Creating & storing your App Store Connect API Key

If you need to create a new ASC API key, [follow these steps](#). We'll be adding the Issuer ID, the Key ID, and the `p8` private key to GitHub's encrypted secrets.

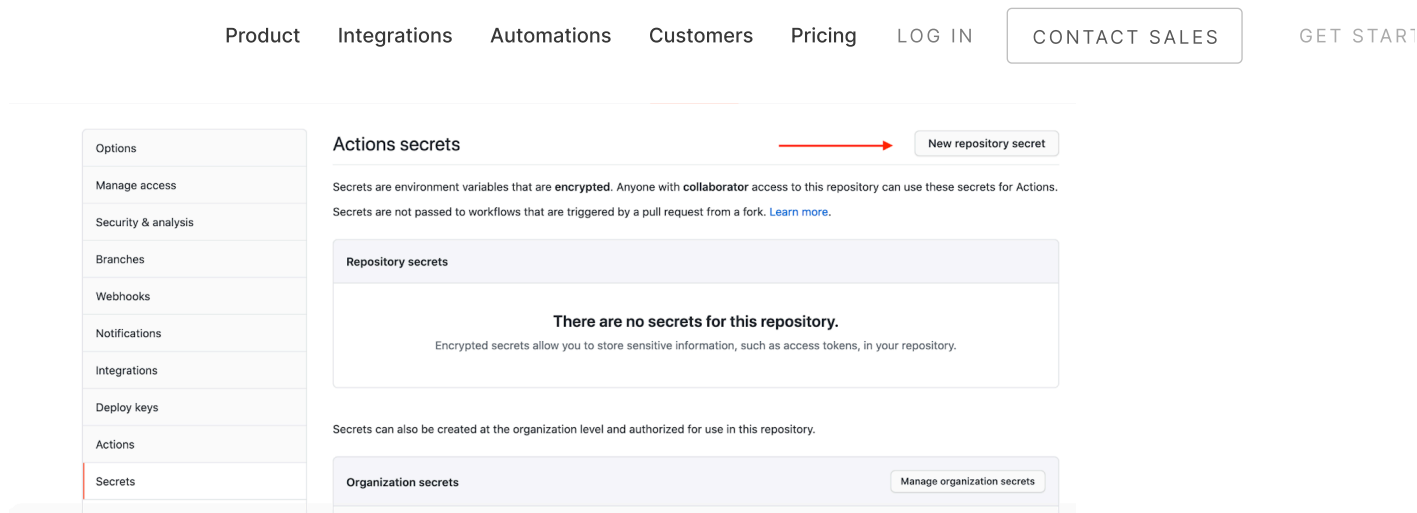
To add a new secret to GitHub's encrypted secrets, first navigate to the iOS repo which you'll be adding the GitHub workflow to. On the far right, click "Settings".



Then, click "Secrets" from the list in the left menu.



From here, click "New repository secret" to add a new secret:



When you click “New repository secret”, you’ll see a form that will prompt you to enter a name for your new secret, and the value.

Actions secrets / New secret

Name

Value

Add secret

GitHub secrets only accepts strings, so for certain credentials (the `.p12` signing certificate file for example), you’ll first need to convert the file to a base64-encoded string before adding it to GitHub secrets. You can do this from the command line:

```
base64 in_file_path | pbcopy
```

This copies the resulting string to your clipboard, so you can paste it directly into a new repository secret on GitHub.

For example:

```
base64 signing-certificate.p12 | pbcopy
```

Let's create new repository secrets as follows:

ASC_KEY_ID	the Key ID used for ASC authentication
ASC_PRIVATE_KEY	the private key (.p8) used for ASC authentication. Note that this key does not need to be base64 encoded – you can paste the contents of the key directly into the “value” field on the New secret form

Please note that you should store a backup copy of your secrets securely in another location (somewhere that is not GitHub encrypted secrets), as you won't be able to export or access the credentials again from GitHub after you've added them.

With these secrets added to GitHub's repository secrets, we can authenticate with the ASC API within the GitHub workflow we'll create to run our builds.

Storing your App Store distribution certificate & private key

In order to properly sign App Store distribution builds on CI, the workflow will need access to a [valid App Store distribution certificate](#) and private key pair. If you already use “fastlane match” to manage your App Store signing certificates, then you can skip this step and call fastlane match from your Fastfile instead. If you still manage your signing certificates manually, then you'll need to add the App Store distribution signing certificate & private key (.p12) to your repository secrets:

- IOS_DIST_SIGNING_KEY - the base64-encoded .p12 distribution certificate
- IOS_DIST_SIGNING_KEY_PASSWORD - the password used during export of the certificate

With these secrets added to GitHub's repository secrets, we're ready to set up our GitHub Actions workflow to run our builds.

Set up your GitHub Actions workflow .yml file

Let's set up our iOS GitHub actions workflow .yml file – it'll define the steps we'll run as part of our workflow. Within these steps, we'll call our fastlane lanes.

First let's create the necessary folders. From your project's root directory call:

```
mkdir .github && cd .github && mkdir workflows && cd workflows && touch build-upload-ios.yml
```

Then, paste the following code into your newly created build-upload-ios.yml file:

```
name: iOS binary build & upload

on:
  workflow_dispatch:

jobs:
  deploy:
    runs-on: macos-latest
    steps:
      - uses: actions/checkout@v2

      - name: Set up ruby env
        uses: ruby/setup-ruby@v1.138.0
        with:
          ruby-version: 3.2.1
```

Product

Integrations

Automations

Customers

Pricing

LOG IN

CONTACT SALES

GET STARTED

```

run: |
  echo $CERTIFICATE_BASE64 | base64 --decode > signing-cert.p12

- name: Build & upload iOS binary
  run: bundle exec fastlane ios build_upload_testflight
  env:
    ASC_KEY_ID: ${ secrets.ASC_KEY_ID }
    ASC_ISSUER_ID: ${ secrets.ASC_ISSUER_ID }
    ASC_KEY: ${ secrets.ASC_PRIVATE_KEY }
    SIGNING_KEY_PASSWORD: ${ secrets.IOS_DIST_SIGNING_KEY_PASSWORD }
    SIGNING_KEY_FILE_PATH: signing-cert.p12

- name: Upload app-store ipa and dsyms to artifacts
  uses: actions/upload-artifact@v2
  with:
    name: app-store ipa & dsyms
    path: |
      ${ github.workspace }/example-iOS.ipa
      ${ github.workspace }/*.app.dSYM.zip

```

Let's break down what this workflow is doing:

```

on:
  workflow_dispatch:

```

This line defines the trigger for the workflow as a manual workflow trigger. GitHub Actions workflows **support a number of triggers**; for example, if you want the workflow to run any time a specific branch has code pushed up to it, you would define it like so:

```

on:
  push:
    branches:
      - release

```

Our workflow defines a single job, called “deploy”, which runs on the latest macOS machine version supported by GitHub workflows. The job has a series of steps, marked with a dash “-”. Some of these steps will use predefined “actions”, which can either be provided by GitHub (e.g. `actions/checkout@v2`), or available as open source actions created and supported by the community through [GitHub's Marketplace](#).

```

jobs:
  deploy:
    runs-on: macos-latest
    steps:

```

The first step is checking out the codebase using GitHub's `checkout` action. Then, we install our dependencies. Since we're using fastlane, which is a Ruby gem, we'll define a Ruby environment with the option `bundler-cache: true` , which will **automatically run** `bundle install` and cache installed gems. `bundle install` will install fastlane and any other Ruby dependencies in your project.

[Product](#)[Integrations](#)[Automations](#)[Customers](#)[Pricing](#)[LOG IN](#)[CONTACT SALES](#)[GET STARTED](#)

```
ruby-version: 2.7.2 # omit if .ruby-version file exists in project, or replace with your target version
bundler-cache: true
```

Next, we'll need to load our distribution code signing certificate into a file so we can pass it into fastlane's `import_certificates` action. We'll do this using a simple bash command that takes our base64-encoded App Store distribution code signing certificate which is stored in GitHub's repository secrets, decodes them, and adds their contents to a new file in the current directory called `signing-cert.p12`. We'll pass this file path to fastlane via an environment variable, which will allow us to properly sign the build we'll be creating in the next step.

```
- name: Decode signing certificate into a file
  env:
    CERTIFICATE_BASE64: ${ secrets.IOS_DIST_SIGNING_KEY }
  run: |
    echo $CERTIFICATE_BASE64 | base64 --decode > signing-cert.p12
```

Finally, we can call our main fastlane action, "build_upload_testflight", passing in the required ASC API key information (also stored in GitHub's repository secrets), and the path of the signing certificate we decoded in the previous step. Our fastlane action will take care of downloading the necessary provisioning profiles, updating the build number, building the app, and uploading the resulting `.ipa` to TestFlight.

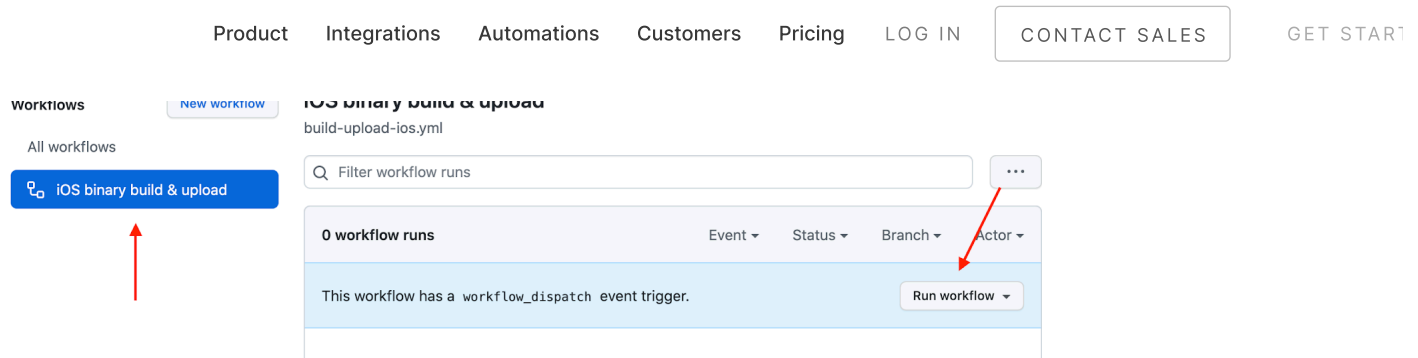
```
- name: Build & deploy iOS release
  run: bundle exec fastlane ios build_upload_testflight
  env:
    ASC_KEY_ID: ${ secrets.ASC_KEY_ID }
    ASC_ISSUER_ID: ${ secrets.ASC_ISSUER_ID }
    ASC_KEY: ${ secrets.ASC_PRIVATE_KEY }
    SIGNING_KEY_PASSWORD: ${ secrets.IOS_DIST_SIGNING_KEY_PASSWORD }
    SIGNING_KEY_FILE_PATH: signing-cert.p12
```

The last step in the workflow uploads the produced artifacts (the `.ipa` file and its associated dSYM files) so they can be viewed in GitHub as part of the workflow. If you don't have any use for the build's artifacts, you can remove this step from the workflow.

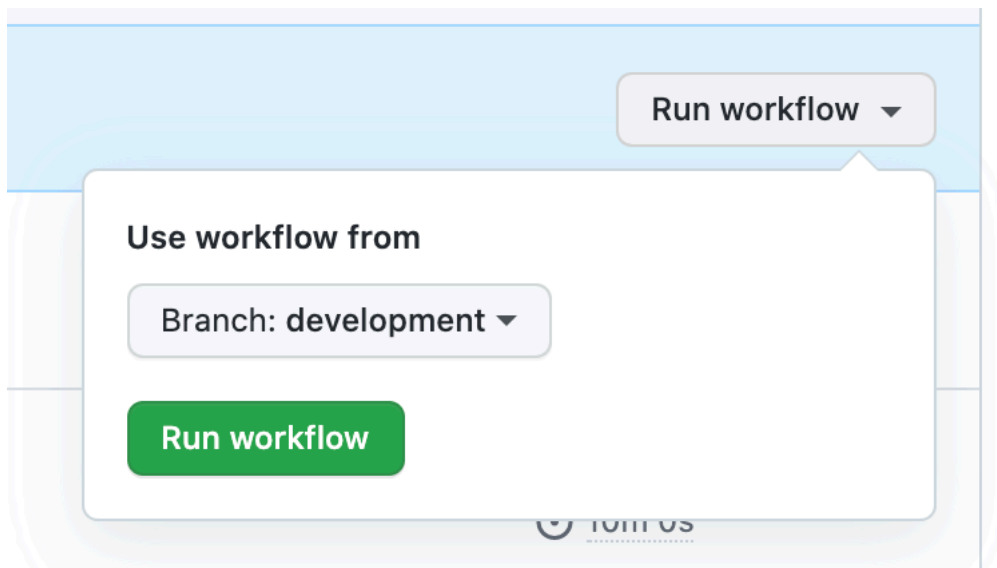
Now, you can commit and push up your newly created GitHub workflow file. It'll show up on your repo under the "Actions" tab.

Running your build

Once you've pushed up your GitHub workflow file, you'll be able to trigger your workflow directly from GitHub's UI. Simply find your workflow in the "Actions" tab, and click "Run workflow":



Choose a base branch to run the workflow from, and click “Run workflow” to try out your workflow!



And that's it!

Using fastlane combined with GitHub Actions, we've put together a simple CI pipeline that increments build numbers, creates a distribution `.ipa`, and uploads the `.ipa` to TestFlight. Anyone that has access to your GitHub repository can now trigger the workflow directly from GitHub, or you can set a workflow trigger to automatically kick off a workflow run any time pushes are made to certain branches. With a basic build & deploy CI pipeline now in place, anyone on the team is empowered to create and upload builds, removing a common bottleneck while also increasing reliability and consistency.

How to generate an iOS CI/CD pipeline automatically

If you're interested in skipping past all those steps above, you can autogenerate an end-to-end build-and-deploy workflow using our [Quickstart CI/CD wizard](#). The wizard will pull together all the necessary steps, configs, and secrets for you, then generate all the necessary files (GitHub Actions YAML, fastlane Fastfile, Ruby Gemfile, etc.) for you to merge straight into your repo or take with you wherever you like.

Questions or issues? [Get in touch!](#)


runway

Try Runway Quickstart CI/CD to quickly autogenerate an end-to-end workflow for major CI/CD providers.

GET STARTED

RELATED POSTS


How to set up and run a beta testing program for your mobile app



Richard Huffaker

SEPTEMBER 18, 2023


Key DevOps metrics and how to measure them for mobile teams



Isabel Barrera

NOVEMBER 30, 2022

The Runway year in review



Richard Huffaker

DECEMBER 21, 2023

SIGN UP FOR THE FLIGHT DECK, OUR MONTHLY NEWSLETTER

Email

SIGN UP

TRUSTED BY THE BEST MOBILE TEAMS



Product	Integrations	Resources
Mobile release management	Project management	Blog
End-to-end automation	• Jira	FAQ
Rollouts	• Linear	Documentation
Mobile insights	• Pivotal Tracker	Quickstart CI/CD
Build Distro	Version control	App review times
Automations	CI/CD	App Store Connect status page
Security	App stores	App hotfix leaderboard
Pricing	• App Store Connect	
What's new	• Google Play Console	
Explore sandbox	Slack	Company
	Monitoring	About us
	All integrations	Contact
Use cases		Terms of service
fastlane		Privacy policy
Release trains		Careers
Mobile DevOps		Status
Cross-platform		
• React Native		
• Flutter		