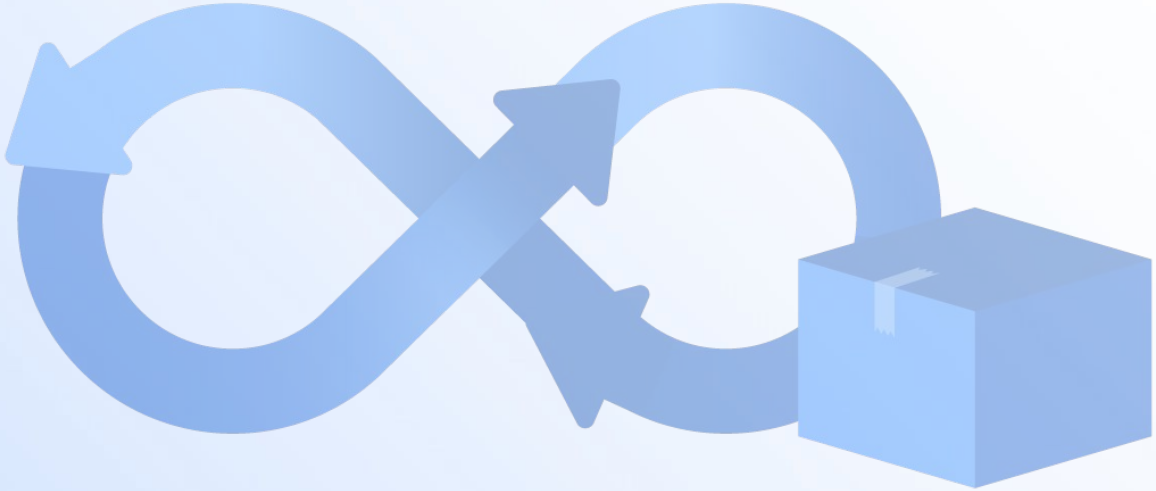
CI/CD



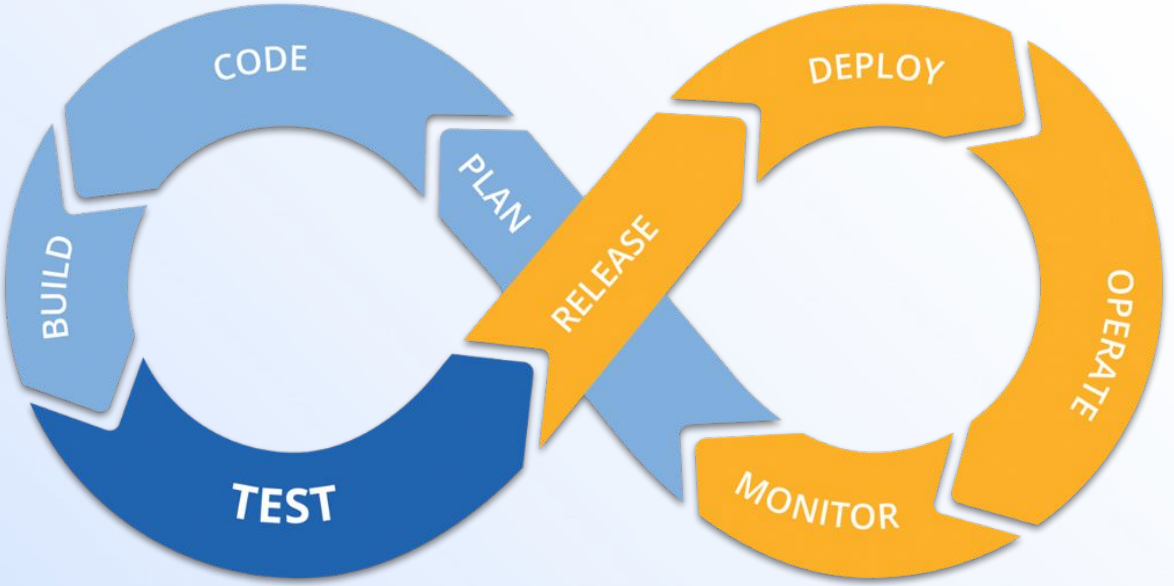
CI / CD - DevOps practices that allows developers to deploy software changes more often and more reliably, minimize errors, increase build rates and improve the quality of the product being developed.

What exactly CI/CD mean?

CI/CD - concept that is implemented as a pipeline, making it easy to merge newly committed code into the main codebase.

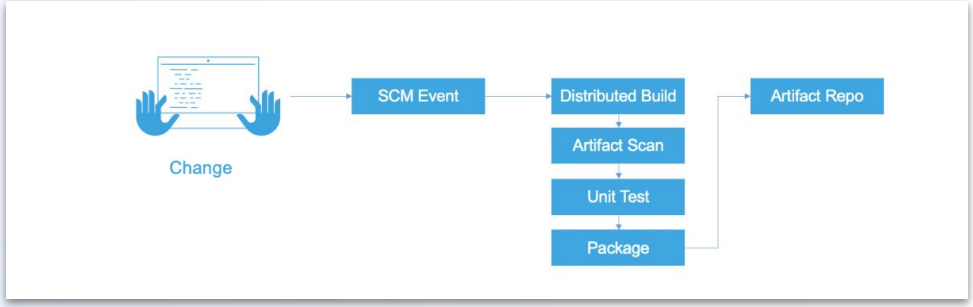
Allows you to run different types of tests at each stage (execution of the **integration** aspect) and complete it with the deployment of the committed code into the actual product that end users see (execution **of delivery**).

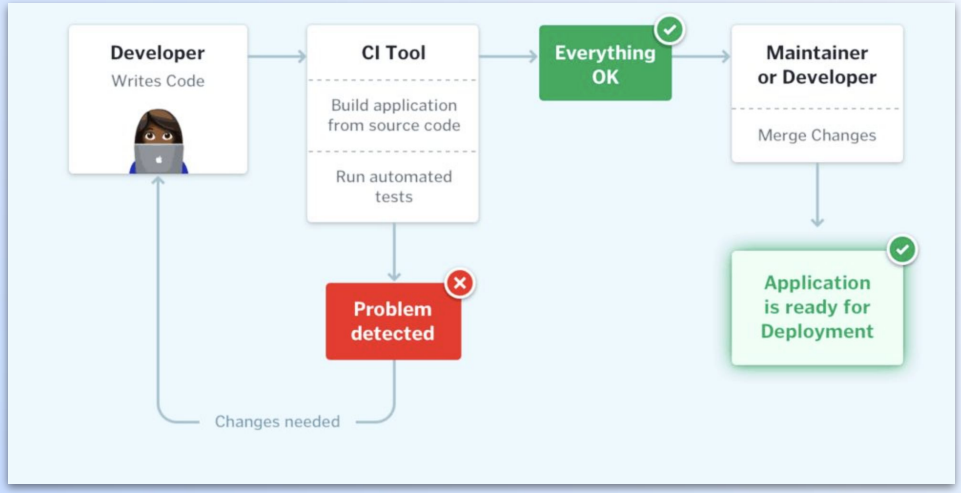
- **Continuous integration**
- **Continuous delivery**
- **Continuous deployment**



Continuous integration

Continuous Integration is automated builds that can be triggered by some sort of event, such as a code check-in, or merge, or on a regular schedule. **The end goal** of a build is to be deployed somewhere, and the main goal of Continuous Integration is to build and publish that deployable unit.





Continuous integration requires

- Developers to merge their changes to the main code branch many times per day.
- Each code merge to trigger an automated code build and test sequence. Developers ideally receive results in less than 10 minutes, so that they can stay focused on their work.

A release candidate - final form of an artifact to be deployed.

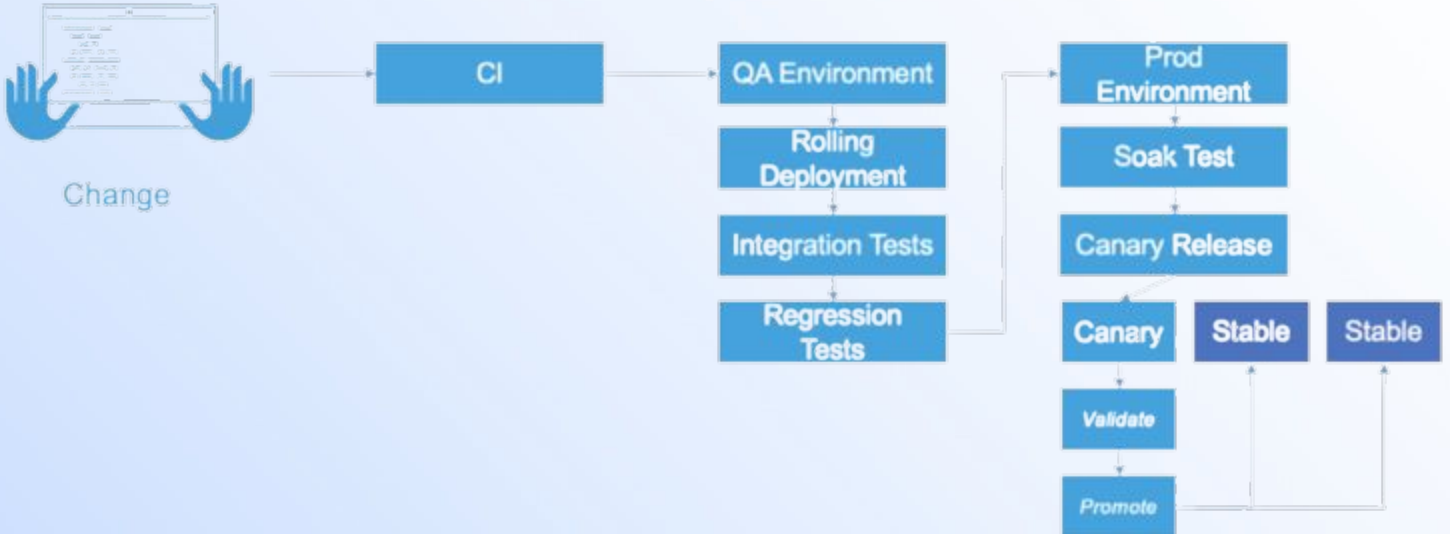
Three pillars that Continuous Integration solves for is having the builds be repeatable, consistent, and available.

Continuous delivery

The goal of continuous delivery is to deliver a packaged artifact into a production environment. CD automates the entire delivery process, including the deployment process.

Continuous Delivery (CD) is a practice of automating the entire software release process. The idea is to do CI, **plus** automatically prepare and track a release to production. The desired outcome is that anyone with sufficient privileges to deploy a new release can do so at any time in one or a few clicks. By eliminating nearly all manual tasks, developers become more productive.

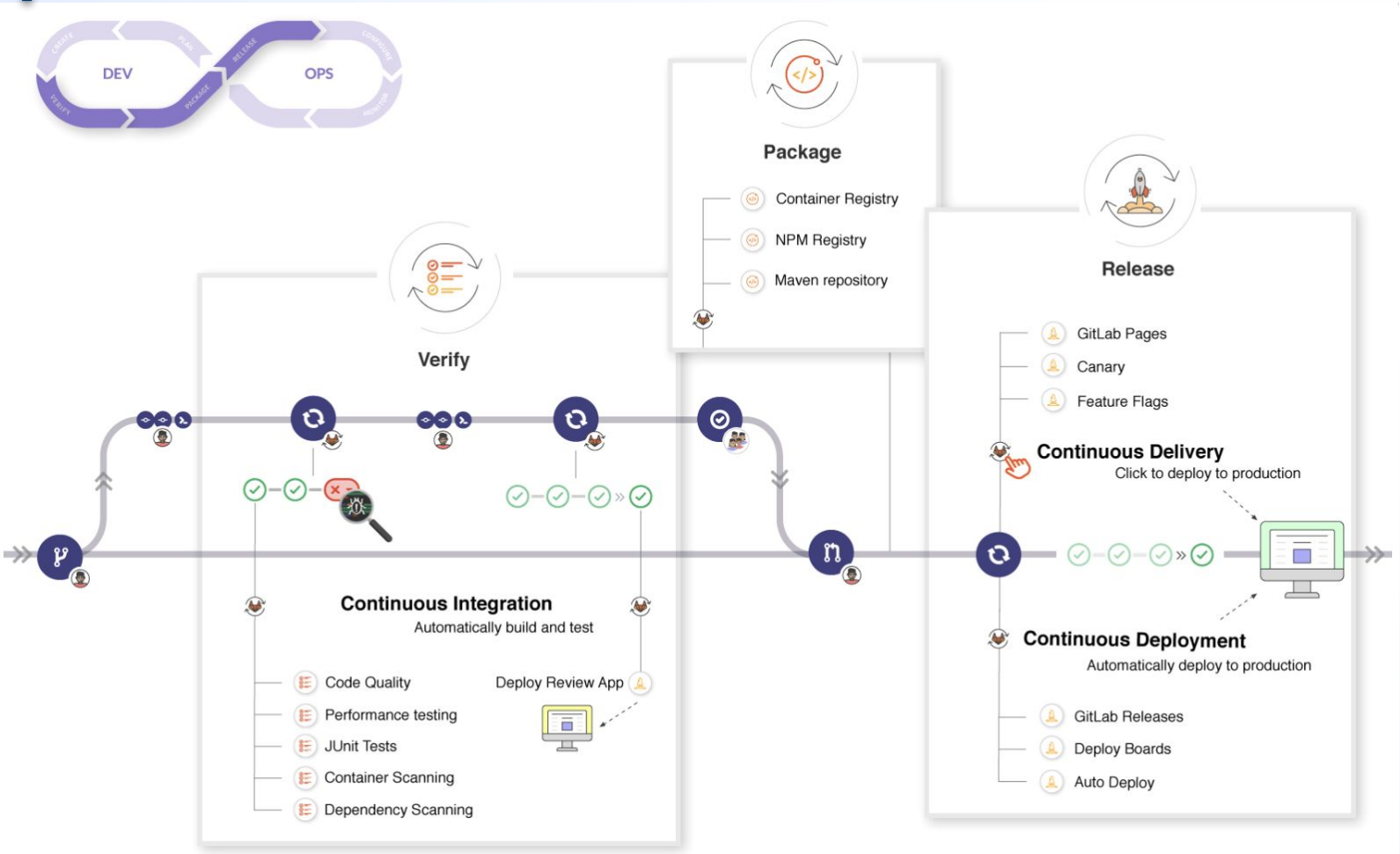
-Continuous delivery is the process of automating steps to safely make changes to the production environment
-Continuous delivery is the practice of ensuring that software is always ready for deployment



Continuous deployment

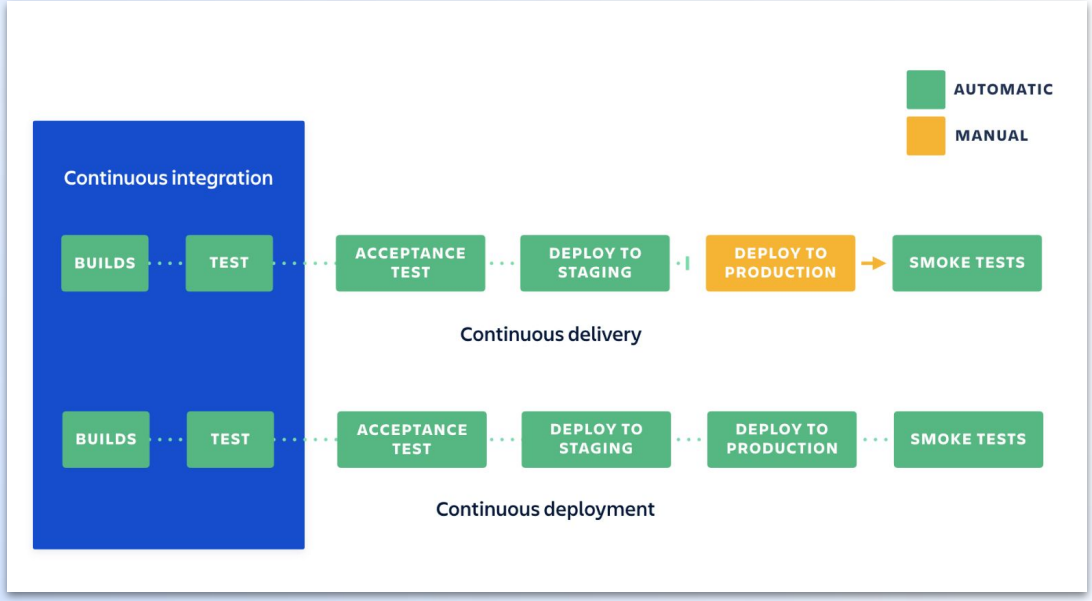
Continuous Deployment is another step beyond Continuous Integration, similar to Continuous Delivery. The **difference is that instead of deploying your application manually, you set it to be deployed automatically**. Human intervention is not required.

Deeper look into the CI/CD workflow



Quick summary and final thoughts

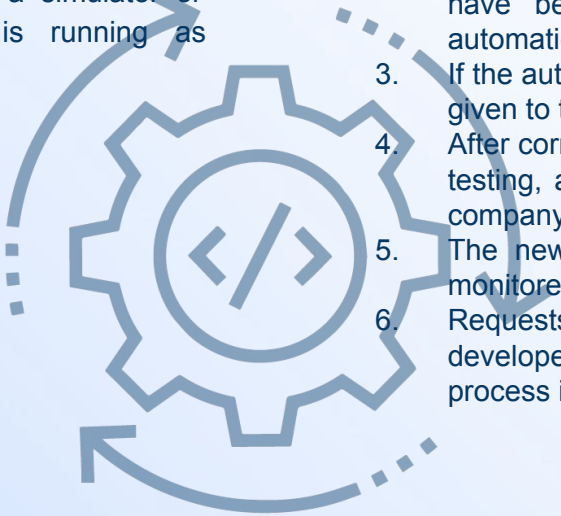
- **Continuous Integration (CI)**: short-lived feature branches, team is merging to master branch multiple times per day, fully automated build and test process which gives feedback within 10 minutes; deployment is manual.
- **Continuous Delivery (CD)**: CI + the entire software release process is automated, it may be composed of multiple stages, and deployment to production is manual.
- **Continuous Deployment**: CI + CD + fully automated deployment to production.



Comparison of deployment cycles

Manual

1. Check out source code from some remote repository (GitHub, Bitbucket, GitLab, etc.).
2. Do some static analysis of the code using SwiftLint, for example.
3. Compile and build your iOS app in a simulator or device to check that everything is running as expected.
4. Run unit, UI or other kinds of tests.
5. Code signing and archive your app.
6. Submit your app to iTunes Connect.
7. Notify users about your release.



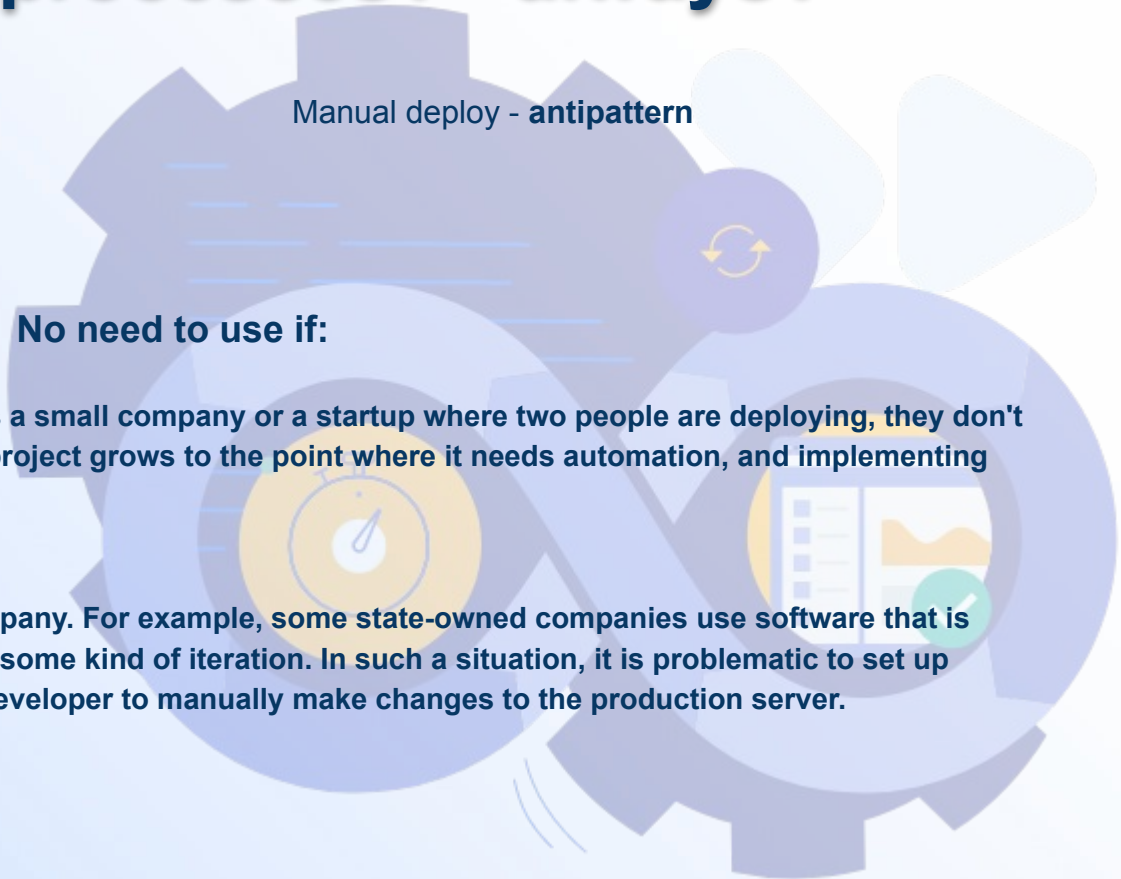
CI/CD

1. The developer writes the code, conducts initial testing to ensure there are no bugs, and commits the changes to their work branch. It then merges the modified code with the working code from upstream.
2. The system selected by the CI tool notices that there have been changes in the code and starts an automatic build and automatic testing of the program.
3. If the automatic testing was successful, the software is given to the test team for manual testing.
4. After correcting the shortcomings found during manual testing, automated installation of the software on the company's servers is started.
5. The new version of the program is supported and monitored.
6. Requests for fixing defects and bugs are collected, the developer makes changes to the code, and the process is repeated.

When to use ci/cd processes? - always?

Deploy - **routine task**

Manual deploy - **antipattern**



No need to use if:

- **When there is no need for automation. If it's a small company or a startup where two people are deploying, they don't need CI/CD yet. It's worth waiting until the project grows to the point where it needs automation, and implementing CI/CD will bring more pros than cons.**

- **When there are archaic methods in the company. For example, some state-owned companies use software that is more than 10-15 years old and is already in some kind of iteration. In such a situation, it is problematic to set up automatic delivery, and it is easier for the developer to manually make changes to the production server.**

The top benefits of CI/CD for iOS

1. Build faster

A continuous and automated development cycle helps mobile development teams make high-value releases faster.

2. Detect problems early

A continuous integration pipeline can run test suites on every code change. Thus, problems can be spotted before they can reach production.

3. Provide effective feedback

In a continuous integration workflow, the entire team can view and monitor source code activity. When new problems appear, the team members that introduced the failure are automatically notified.

4. Automate the testing process

Manually testing iOS apps on a simulator or multiple devices is a time-consuming process. In a CI/CD workflow, the process is automatic and straightforward. Simply configure the devices and simulators into the server workflow and let your pipeline do the work for you.

5. Build deployment logs

In a CI/CD pipeline, all build deployments are logged. This means that at any given time, you can identify the exact point where your app failed.

Best practices to remember for iOS CI/CD

1. Keep your Pipeline Fast

Always improve the quality of your pipeline by inspecting its configuration, improving build times, removing any unnecessary code or assets, configuring the right environments, and removing any bottlenecks causing issues on the deploy process.

2. Run Code Analysis Early

Fail fast: Include unit tests and linters early in the pipeline.

3. Separate Build Configurations

Use profiles to separate build configurations. Internal and testing builds are different from the builds for your users. By default, an iOS app includes Debug and Release build configurations. It makes sense to add Test, Staging and Production builds.

4. Automate Your Builds

Compiling and building an iOS app takes a great deal of time. Using CI/CD workflow you can automate the process and save time.

5. Deploy Only Through the Pipeline

The pipeline is configured to help and enforce best practices of CI/CD for your development cycle. Avoid at all cost manual deploys.

Making your code run through your pipeline requires each change to conform with your code standards and styles. This mechanism safeguards your codebase from untrusted code.

6. Maintain Similar Environments

If possible, try to maintain similar pipelines in all your environments. Changes that pass the requirements of one environment should pass in another. This also includes feature branches.

7. Monitor the Pipeline

Apps are unpredictable. No matter how cool you think your code is, the app can fail for a lot of reasons. Therefore, you'll need to proactively monitor your deploys throughout the pipeline so you can catch problems before they reach the App Store.

Some Tips:

To make CI/CD a reality, you need to automate everything that you can in the software delivery process and run it in a CI/CD pipeline.

- **Architect the system in a way that supports iterative releases**. Avoid tight coupling between components. Implement metrics that help detect issues in real-time.
- **Practice test-driven development to always keep the code in a deployable state**. Maintain a comprehensive and healthy automated test suite. Build in monitoring, logging, and fault-tolerance by design.
- **Work in small iterations**. For example, if you develop in feature branches, they should live no longer than a day. When you need more time to develop new features, use feature flags.
- Developers can **push the code into production-like staging environments**. This ensures that the new version of the software will work when it gets in the hands of users.
- **Anyone can deploy any version** of the software to any environment on demand, **at a push of a button**. If you need to consult a wiki on how to deploy, it's game over.
- **If you build it, you run it**. Autonomous engineering teams should be responsible for the quality and stability of the software they build. This breaks down the silos between traditional developers and operations groups, as they work together to achieve high-level goals.

Main advantages / disadvantages of CI/CD

- high speed of output of new functionality from the client's request to commissioning;
- the possibility of choosing the best option due to operational testing and a larger number of iterations;
- the quality of the final result is higher: autotesting covers all aspects of the product, which is difficult to implement with a standard release approach;
- providing quick feedback between developers and customers;
- ensuring cooperation between product teams of one organization;
- reduction of technical problems in the code;
- fast detection and elimination of defects;
- reducing manual testing and providing the ability to return to a previous version of the product in case of problems with the current build.

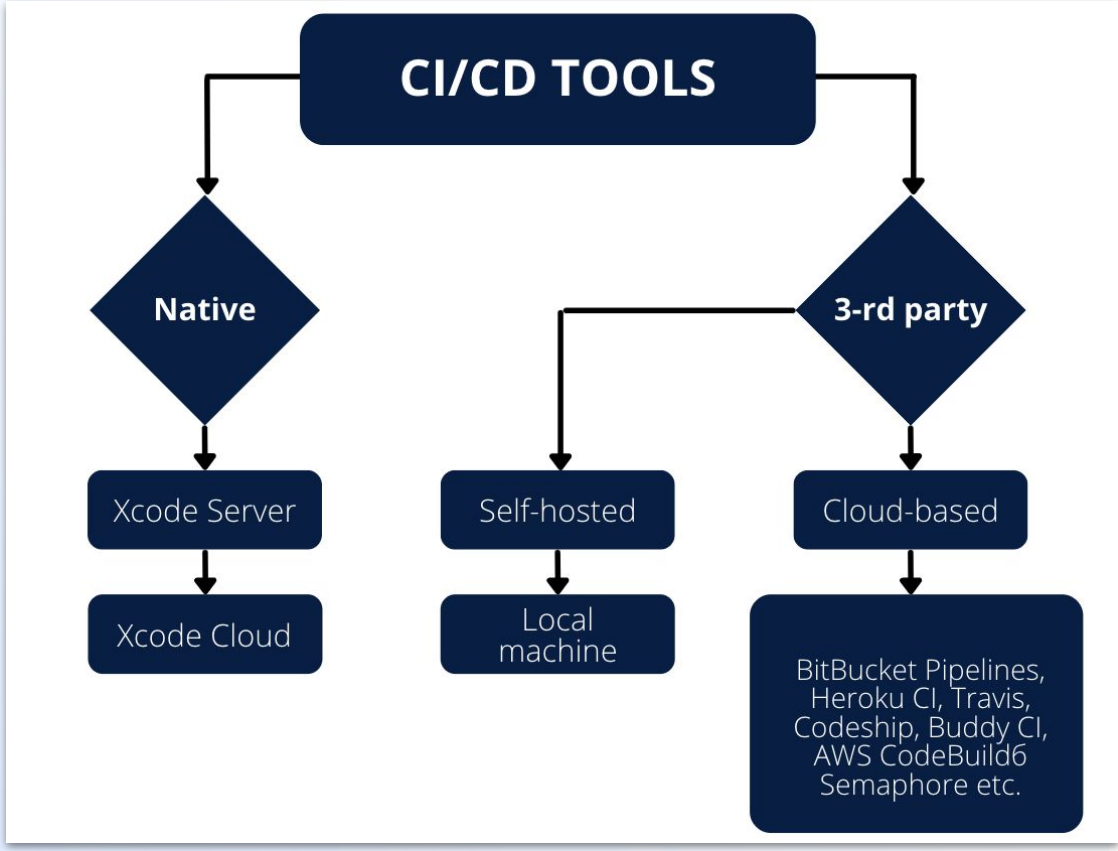
- maintenance of CI/CD tools is an expensive investment;
- changes create a domino effect.
- One small change with CI/CD can affect several different interactions in the code. That is, it can cause problems in service chains. Thus, we need an additional ability to roll back changes if necessary.
- continuous change requires constant monitoring and reporting
- CI/CD changes affect the platform on which they are deployed. Real-time monitoring and reporting is essential to understanding and quickly resolving any issues. If the change leads to incorrect behavior, you need to know about it before the problems affect other services.
- mandatory availability of automated testing, without which CI is basically impossible
- the need to further coordinate the delivery of changes
- CI/CD assumes that any changes are communicated to users as quickly as possible. In this case, for example, the support service is practically not notified of upcoming changes and may not be able to cope with customer questions and complaints.

1.
2.

Platform selection

- **Self-hosted CI/CD solutions** require owning hardware and setting up infrastructure. Maintenance becomes a big endeavor.

- **Cloud-based CI/CD tools** are easy to set up and all the hardware and software maintenance is managed by the service provider.



Platform selection

- **Atlassian Bamboo**, **GitHub Actions** and **GitLab CI** are not as mobile friendly as **Bitrise**, **Codemagic, TeamCity** and **Visual Studio App Center**.
- **Buddy** is fast and offers many useful features at an affordable price, but does not fully support iOS yet.
- If your team uses **Atlassian** products like Confluence, Jira or Trello, **Bamboo** can complement your mobile development workflow.
- If you have one or more open source projects on GitHub, it might be worth checking out **GitHub Actions**. It will be free and will not require a new account.
- If you're already on the GitLab Enterprise plan, you can get started with **GitLab CI** right away without much extra effort.
- **Jenkins and Buildkite** allow you to run builds on your own hardware. Jenkins is a great choice if you need a lot of customization and have experienced DevOps engineers on your team to maintain it, while Buildkite offers a friendlier interface, less maintenance overhead, and little curation of their plugin library.
- **Xcode Cloud** is of great value for lone developers or small teams working only on iOS and not needing some of the more complex configuration options provided by some "installed" services.

Announcement

Xcode Cloud is a CI/CD system that uses Git for source code management and provides you with an integrated system to ensure the quality and stability of your codebase. It also helps you publish apps efficiently. By combining Xcode with Apple's cloud-based code building and testing infrastructure, as well as TestFlight and App Store Connect, Xcode Cloud makes it easy to:

Generate and test your code automatically.

- Test your app on Apple devices in the Simulator automatically and frequently.
- Get notifications from Xcode Cloud to catch bugs before they become serious problems.
- Distribute new versions of your application to team members and testers using TestFlight.
- Providing new versions of your application for review before publishing them to the App Store.
- Collaborate on your software development using Xcode and Apple's cloud infrastructure.

Announcement

Fastlane + Bitrise

Fastlane is a powerful tool used for automated deployment to simplify deployments for mobile platforms. It is simple and easy to use but brings amazing value to your regular ios deployment workflows.

Allows automating every aspect of the build packaging and distribution process and it's an Open-source published under MIT License.

Supports all major CI Platforms like Bitrise, Jenkins, CircleCI, Travis, GitLab CI, Azure DevOps, etc.

Fastlane is a ruby gem, hence runs on the local machine, no server needed https://rubygems.org/gems/fastlane

Fastlane

```
lane :beta do
  increment_build_number
  build_app
  upload_to_testflight
end

lane :release do
  capture_screenshots
  build_app
  upload_to_app_store       # Upload the screenshots and the binary to iTunes
  slack                     # Let your team-mates know the new version is live
end
```

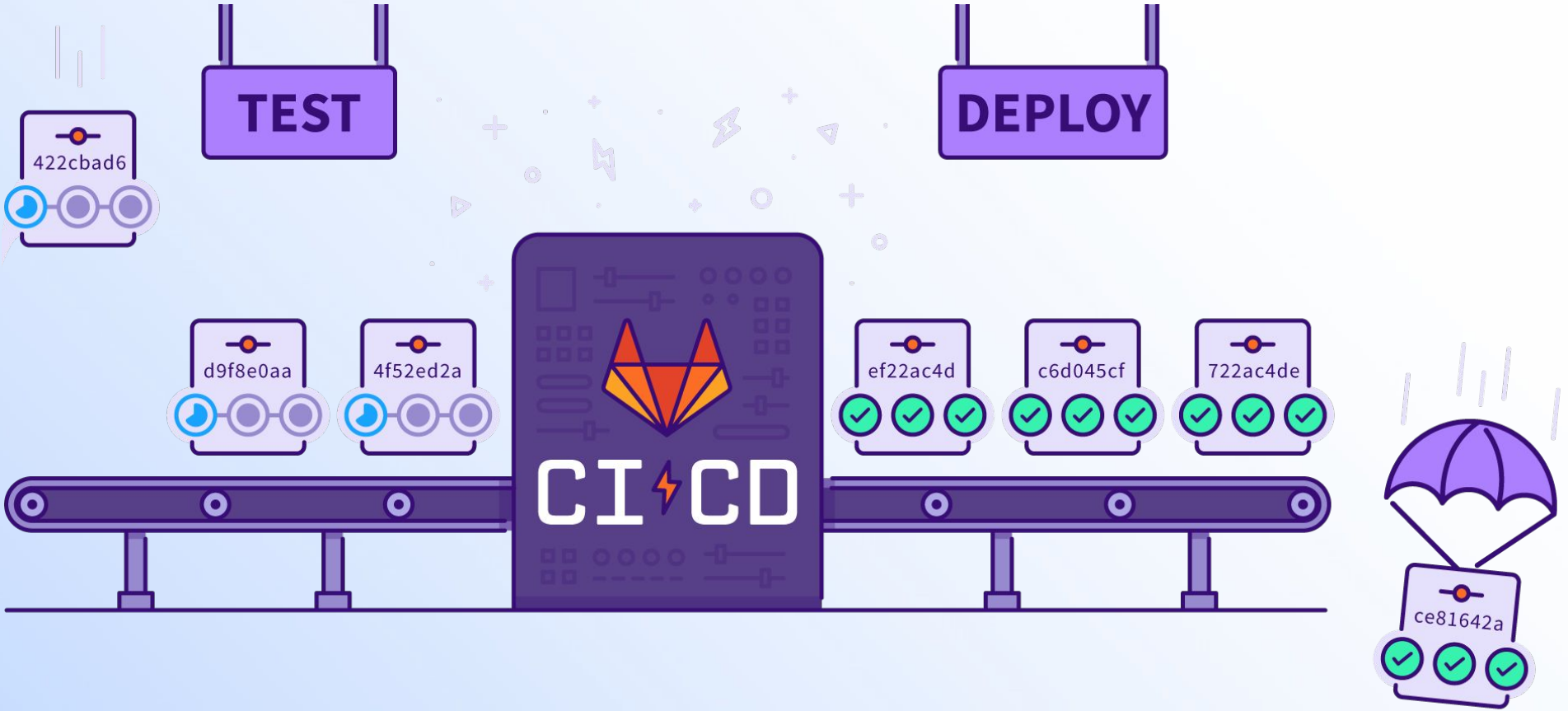
```
  desc "Running tests, returns code coverage report to #{@output_directory}"
  lane :tests do
    scan(
      clean: true,
      code_coverage: true,
      scheme: "YOUR_SCHEME_NAME",
      output_directory: @output_directory,
      skip_slack: true,
    )
  end
```

```
desc "Incrementing build number"
desc "Building"
desc "Archives"
desc "Upload to TestFlight"
desc "Send slack notification"
lane :testflight_upload do
  cert                                              # Working with signing certificate
  sigh                                              # Working with provisioning certificate

  build_number = latest_testflight_build_number + 1 # Get build number
  increment_build_number({                          # Increment build number
      build_number: build_number
  })

  gym(                                              # Build .ipa file
      export_xcargs: "-allowProvisioningUpdates"
  )

  testflight(                                       # Upload to TestFlight
    skip_submission: true
  )
  clean_build_artifacts                             # Clean folders
  slack(                                            # Send new message to channel
    message: "🎉 App successfully uploaded!",
    payload: {
      "📅 Build Date" => Time.new.strftime("%Y/%m/%d %T"),
      "🔔 Build Number" => build_number,
      "🍦 Version" => "v#{get_version_number}"
    },
    default_payloads: [:git_branch],
    slack_url: @slack_url
  )
end
```


TEST
DEPLOY
422cbad6
d9f8e0aa
4f52ed2a
CI/CD
ef22ac4d
c6d045cf
722ac4de
ce81642a

Gitlab CI Concepts

Jobs

Jobs are a fundamental element in a GitLab pipeline. A job defines a task that the pipeline must perform, such as compiling or testing code. Each job has a name and contains a script clause that defines what's to be done. If all jobs on a stage complete successfully, the pipeline moves on to the next stage.

Stages

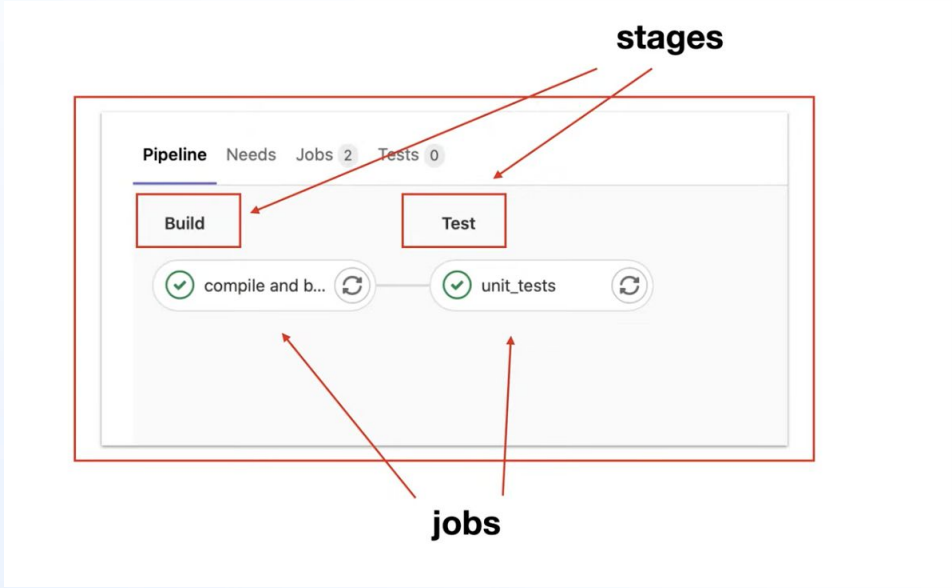
GitLab stages determine when to run jobs. A stage represents different steps in a CI/CD pipeline such as building, testing or deploying an application. Examples of stages include plan, create, verify, configure and release.

Pipeline

A pipeline in GitLab is a collection of stages, each containing one or more jobs. GitLab has different types of pipelines including basic, merge, parent-child and multi-project pipelines.

Commit

A commit is a record of a change, such as a code or a file change. For example, if you have changed a file stored in a repository, you commit the change to record the alteration.



Runners

Runners are agents or applications that run CI/CD pipeline jobs from GitLab. You can use GitLab Runners or use Linux, Windows or macOS runners. Runners process jobs on the machine where they are installed, but can also run in containers or Kubernetes clusters.

In general, pipelines are executed automatically and require no intervention once created. However, there are also times when you can manually interact with a pipeline. Learn about the types of CI/CD pipelines.

Pipeline Rules

1. Pipelines consist of **stages**, and they, in turn, consist of **jobs**

2. Each **job** can only have one **stage**

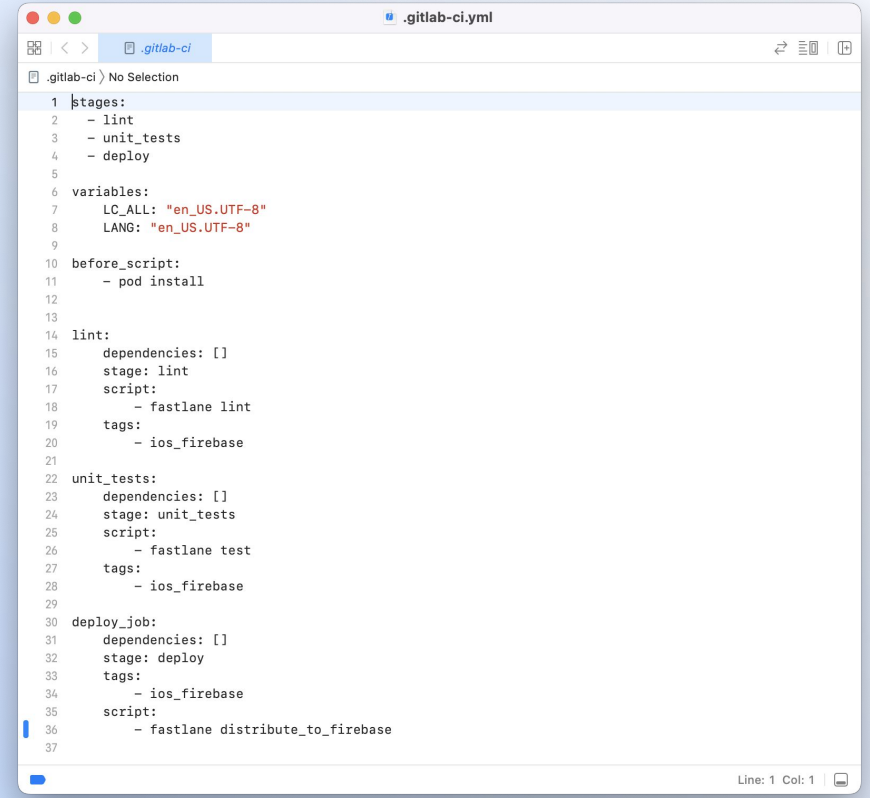
3. Each stage can contain several **jobs** that run sequentially and in parallel. So if you have multiple runners, then **test_job1** and **test_job2** would start almost at the same time without waiting for each other to complete

4. **Stages** go only sequentially, each waits for the completion of all **jobs** from the previous **stage** before starting

5. If the exit command was called with a non-zero error while executing the job, then the job fails: **exit 0** means that the job was successfully completed; any other number will indicate some kind of error

6. If one of the jobs in the stage fails without the **allow_failure: true** flag, then the next stage does not start. For **test_job3** this flag is enabled, but not for **test_job5**

Pipeline Rules



7. If there are several jobs in the stage and one of them fails, then the other jobs of the stage continue execution

8. You can write coverage in different jobs. For example, in job 1 we wrote coverage 0%, in job 2 we got 10%, and in job 4 we wrote 100%. Then the total coverage of the pipeline will be considered as the average among all jobs (0 + 10 + 100) / 3. This can be used, for example, for multi modularity.

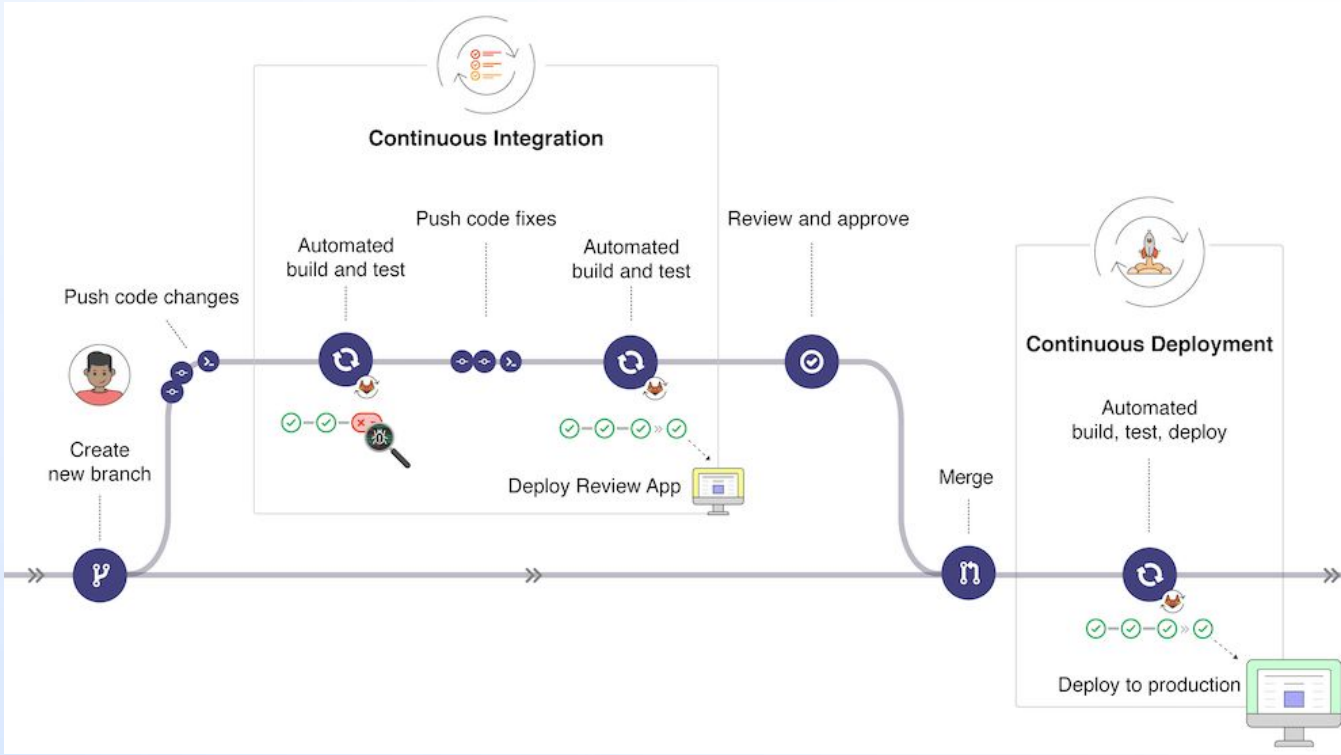
9. Coverage is taken into account even if the job fails

10. In job 8, the when: on_failure flag is set, which means that the job will be executed only if the pipeline fails. If the pipeline passed without problems, this job will not start. It is useful to set such a flag, for example, in jobs that notify about a failed pipeline

11. The order of the stages in the pipeline depends on their order in the CI file in the stages phase: those that are higher in order will be executed first. Jobs are the same: the higher you describe them in the file, the earlier they will be executed in the pipeline

Workflow

- Make changes to your codebase and commit to GitLab.
- GitLab recognizes that the codebase has changed.
- GitLab runs the build using the GitLab Runner you set up on your Mac.
- GitLab Runner executes the build and test process you specified in the .yml file script. (It depends entirely on your scenario)
- GitLab Runner reports its results back to the GitLab server, and Gitlab shows you the build result.



Gitlab Jobs

extends - needed to reuse basic things;

stage - indicate in which stage the job is involved. Within one stage, they are executed in parallel;

before script, script, after script - a set of Shell commands in the order in which they are executed;

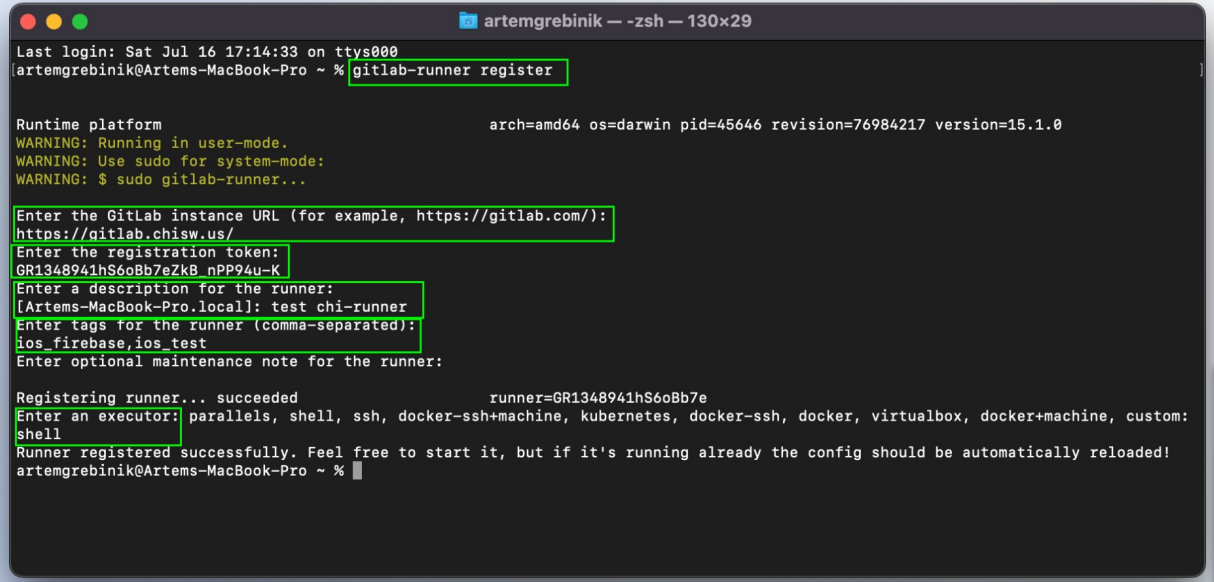
only/except - describe which rules the job falls / definitely does not fall under;

tags - tags with which we marked the runner when we registered it;

artifacts - we will use jobs to store artifacts.

allow failure - a property that allows / does not allow a faked job, valid exit codes can be specified here;

Gitlab-runner



Install gitlab-runner on mac:
brew install gitlab-runner

Making mac a CI runner:
gitlab-runner register

Set up a specific runner for a project

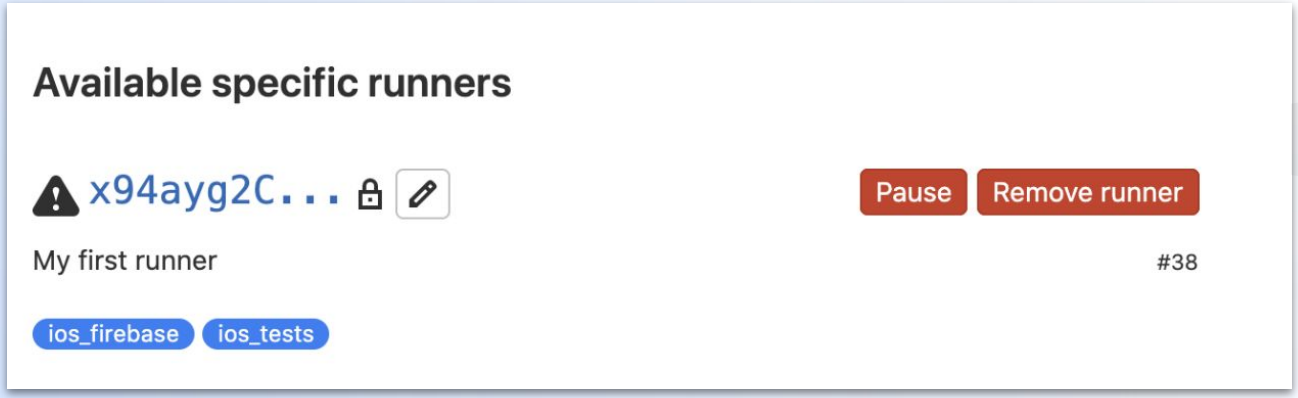
1. Install GitLab Runner and ensure it's running.
2. Register the runner with this URL:
   `https://gitlab.chisw.us/`

   And this registration token:
   `GR1348941hS6oBb7eZkB_nPP94u-K`

Reset registration token

Show runner installation instructions

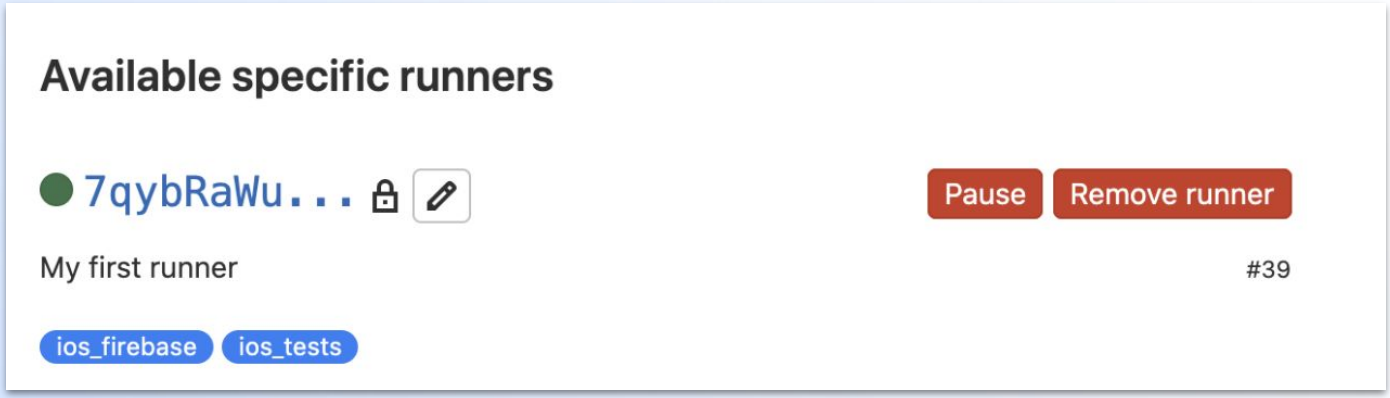
Gitlab-runner



Launching the runner:

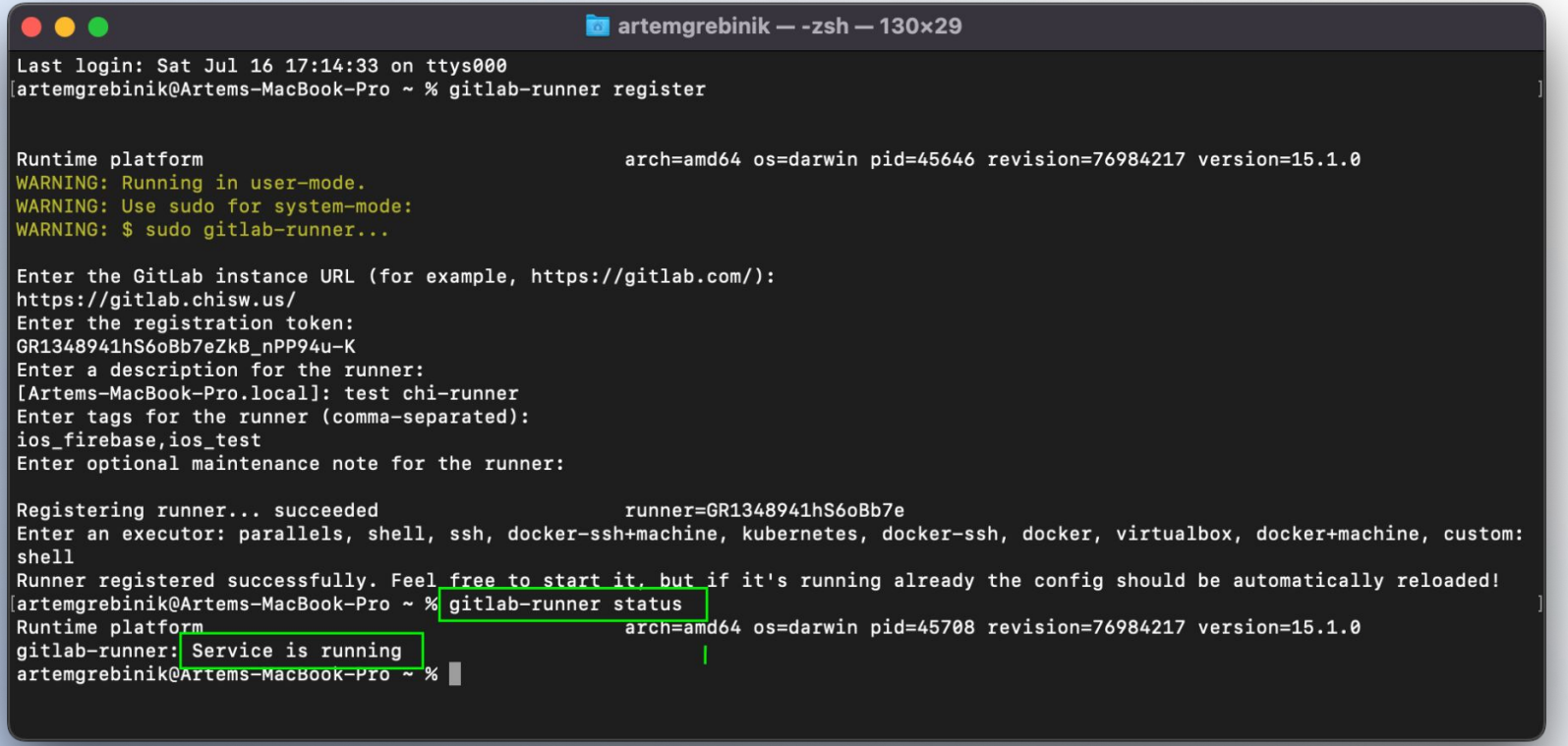
gitlab-runner install

gitlab-runner start

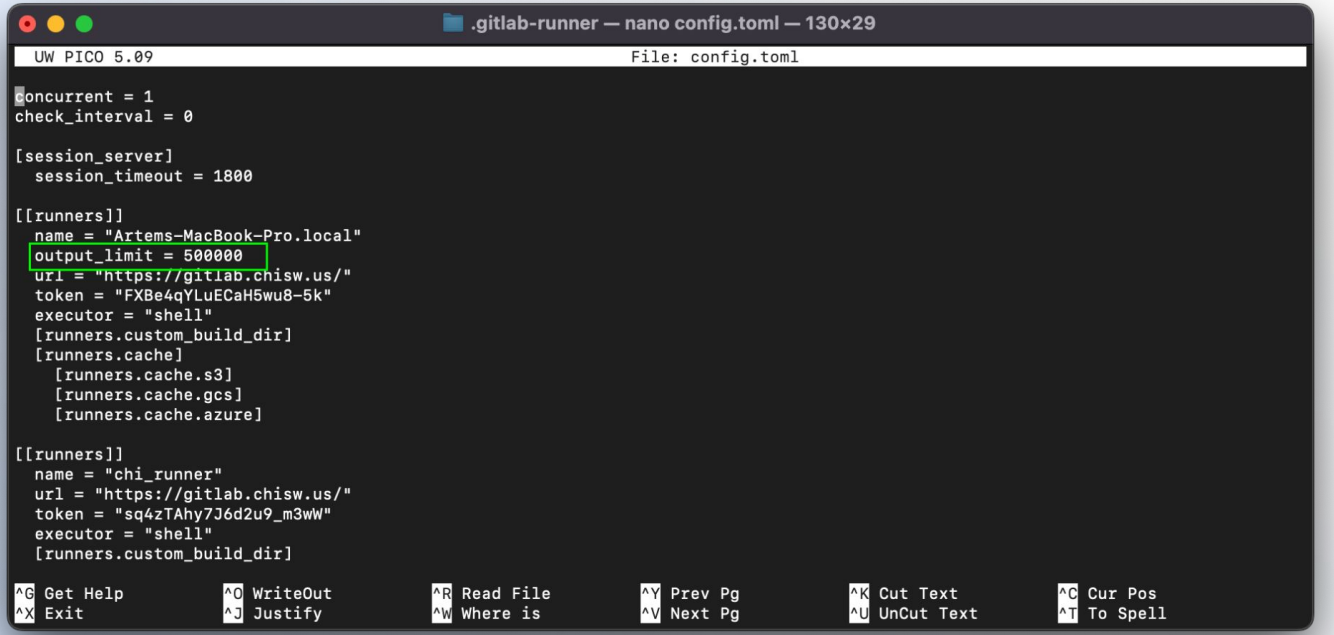


Gitlab-runner

Status:
gitlab-runner status



Gitlab-runner configuration



config.toml - all information about registered runners on this machine

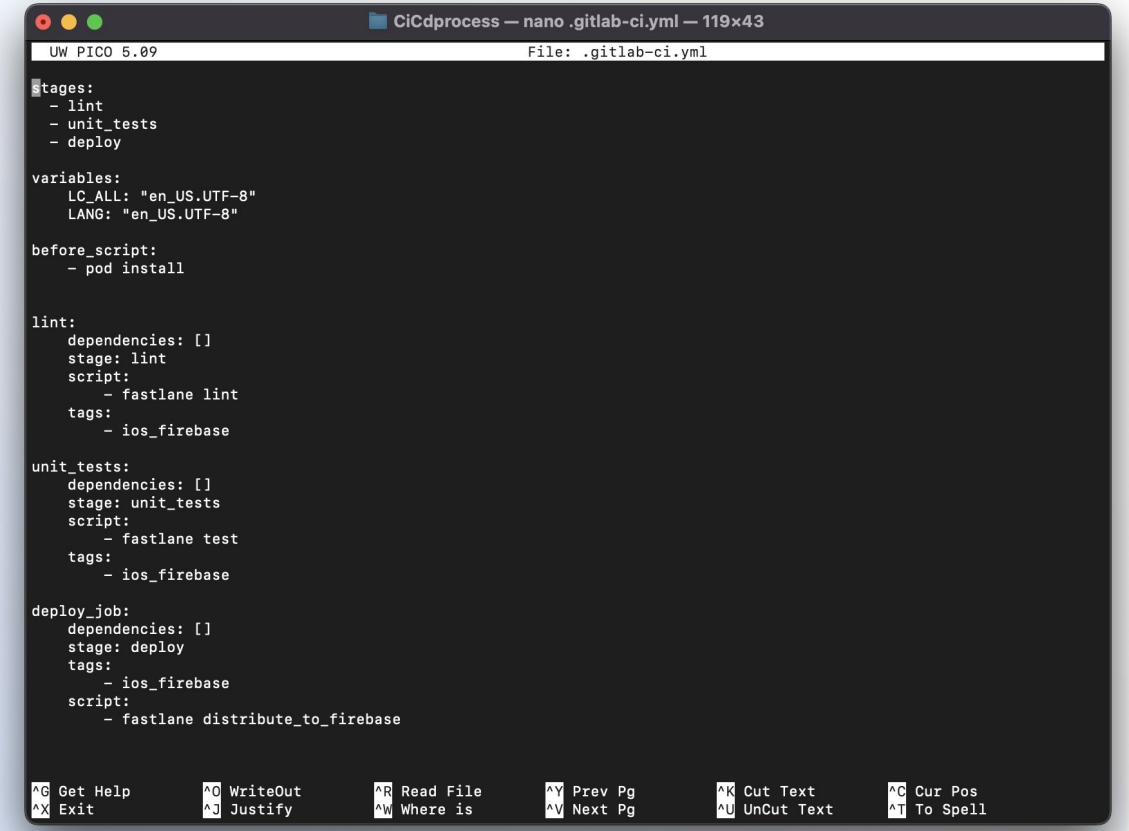
output_limit - maximum size in bytes of stored logs in jobs

! Anything that exceeds the output_limit threshold will not be logged and will not be displayed in job information.

Gitlab-ci.yml

Usage rules:

- The file name should always be .gitlab-ci.yml with a dot at the beginning.
- If you accidentally named the file incorrectly, then this file will not be recognized, and the runner will not execute the script.
- This file must be created in the root folder of the project.
- Tags are also defined in the .yml file. These tags correspond to the tags that were defined during the setup of the runners.
- After calling the yml file, the runner with the same tag as specified in the yml file will take the job and execute it through the pipeline.
- A pipeline is a set of tasks/stages defined in a yml file for execution.
- Each task/stage defined in the yml file is a job.
- A pipeline only succeeds if all the jobs associated with it succeed.
- If at least one job fails, the pipeline will not throw an error message in the Gitlab console.

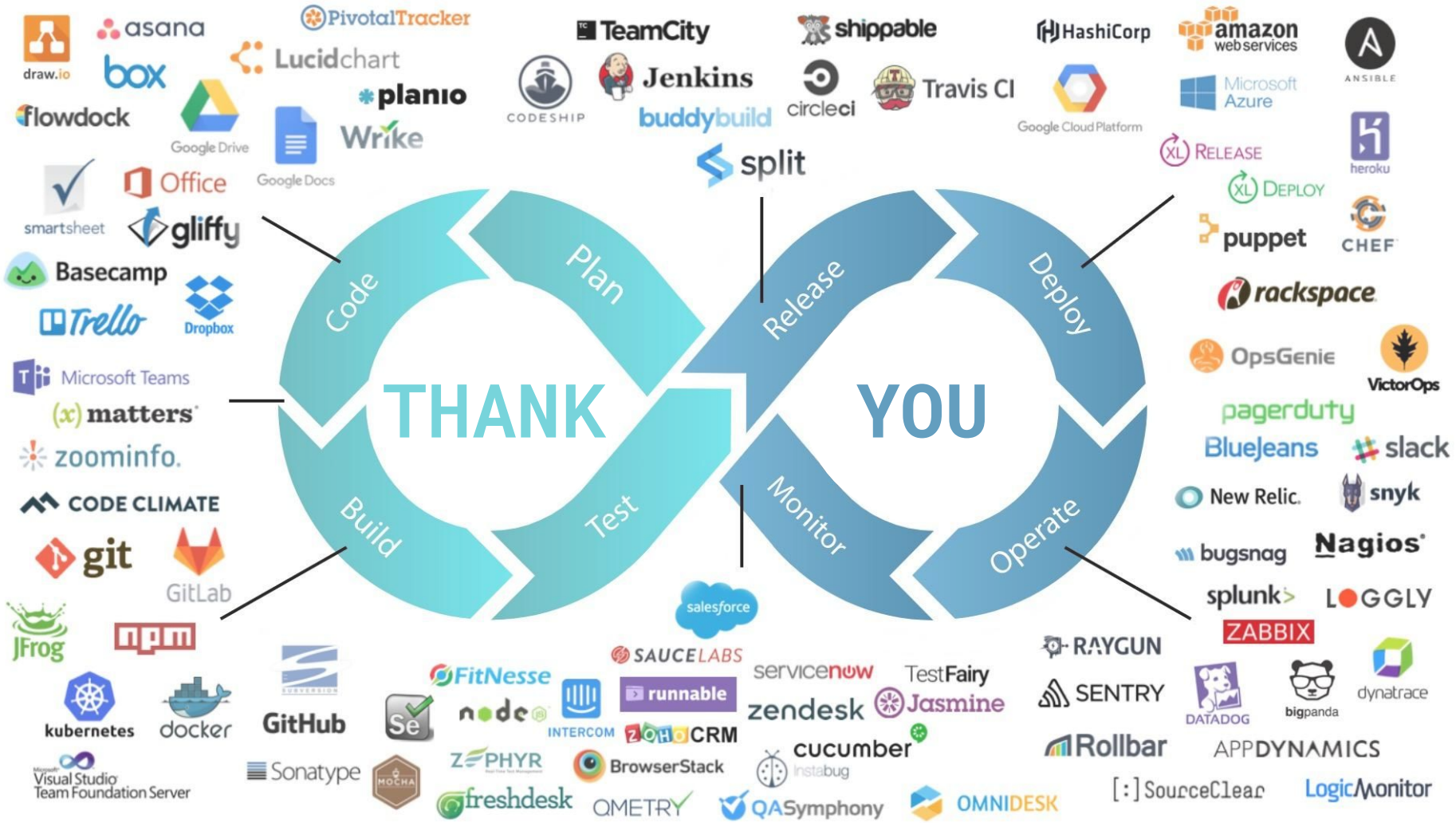


Resources

https://habr.com/ru/company/psb/blog/583532/
https://medium.com/maddevs-io/automatic-delivery-of-ios-applications-with-fastlane-and-gitlab ci-d06f6c2f96dc
https://blog.canopas.com/a-complete-guide-to-ios-app-auto-deployment-with-ci-cd-b5dc516ba4 1d
https://pac-pac.medium.com/how-to-auto-deploy-a-mobile-application-on-the-stores-with-gitlab-and-fastlane-608e44be3aac
https://github.com/boytpcm123/gitlab-ci-fastlane/blob/master/README.md
https://faun.pub/continuous-integration-in-ios-apps-using-gitlab-ci-11156f232087
https://medium.com/@spkdrxw/ios-ci-cd-experiments-gitlab-runner-fastlane-swiftlint-9c07a4522 d9e
https://medium.com/@MjCodingCamp/setup-ci-cd-on-gitlab-for-ios-project-complete-guide-3d1 24a0e51a7
https://phanquanghoang.medium.com/using-gitlab-ci-cd-fastlane-for-ios-project-part-3-f710b61 8da4a

Gitlab CI + Bitrise CLI
https://stevedao91.medium.com/gitlab-ci-bitrise-cli-lower-cost-more-flexibility-85a2b756417e

Firebase App Distribution
https://medium.com/@ryanisnhp/firebase-app-distribution-and-fastlane-5303c17b4395


THANK
YOU
Code
Plan
Release
Deploy
Build
Test
Monitor
Operate
draw.io
asana
PivotalTracker
TeamCity
shippable
HashiCorp
amazon web services
box
Lucidchart
CODESHIP
Jenkins
circleci
Travis CI
Microsoft Azure
ANSIBLE
flowdock
planio
buddybuild
Google Cloud Platform
Google Drive
Google Docs
Wrike
split
RELEASE
heroku
DEPLOY
Office
smartsheet
gliffy
puppet
CHEF
Basecamp
rackspace
Trello
Dropbox
OpsGenie
VictorOps
Microsoft Teams
pagerduty
(x) matters
BlueJeans
slack
zoominfo.
New Relic.
snyk
CODE CLIMATE
bugsnag
Nagios
git
GitLab
splunk>
LOGGLY
JFrog
npm
ZABBIX
salesforce
SAUCELABS
servicenow
TestFairy
RAYGUN
FitNesse
runnable
zendesk
Jasmine
SENTRY
DATADOG
bigpanda
dynatrace
kubernetes
docker
GitHub
INTERCOM
ZOHO CRM
cucumber
Rollbar
APPDYNAMICS
ZEPHYR
BrowserStack
Instabug
Sonatype
Visual Studio Team Foundation Server
freshdesk
QMETRY
QASymphony
OMNIDESK
[:]SourceClear
LogicMonitor