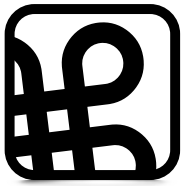


# Macoun

# CI/CD für iOS Projekte

Sebastian Messingfeld

# Hinweis

- Basiert auf eigenen Erfahrungen
- Änderung der Teamkonstellation während Projekt
- Arbeitsumfeld
  - Git als SCM
  - GitLab oder GitHub als Git Frontend
  - Buildserver / -service im Projekt vorhanden

# CI/CD!? Alles als Code!?

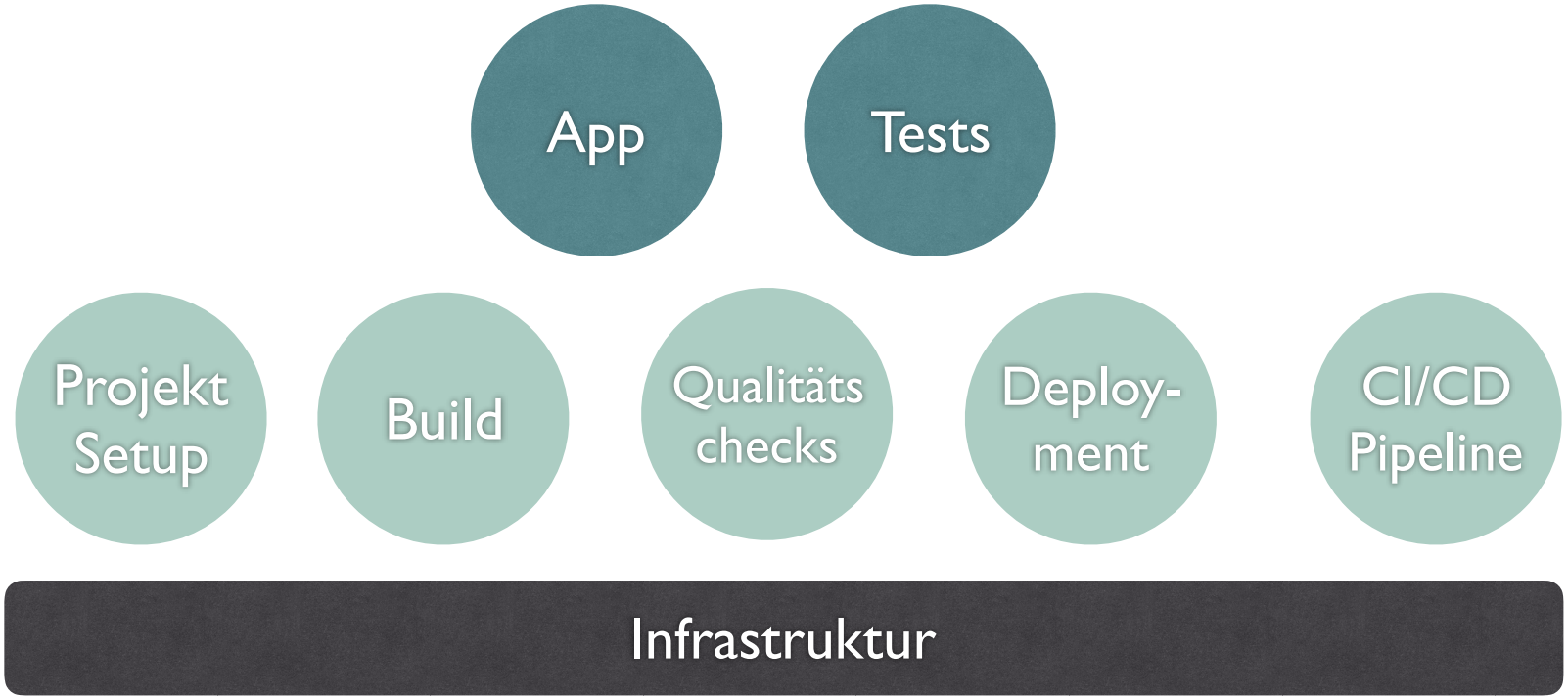
# CI? CD? CD?

- Continuous Integration
  - Integration von neuen Features in Haupt-Branch
- Continuous Delivery
  - Bauen der App mit neuen Features
- Continuous Deployment
  - App in Produktion bringen

# Workflow


IDE
Qualitäts checks
SCM Feature
Qualitäts checks
Code Review
SCM Master
Deploy-ment

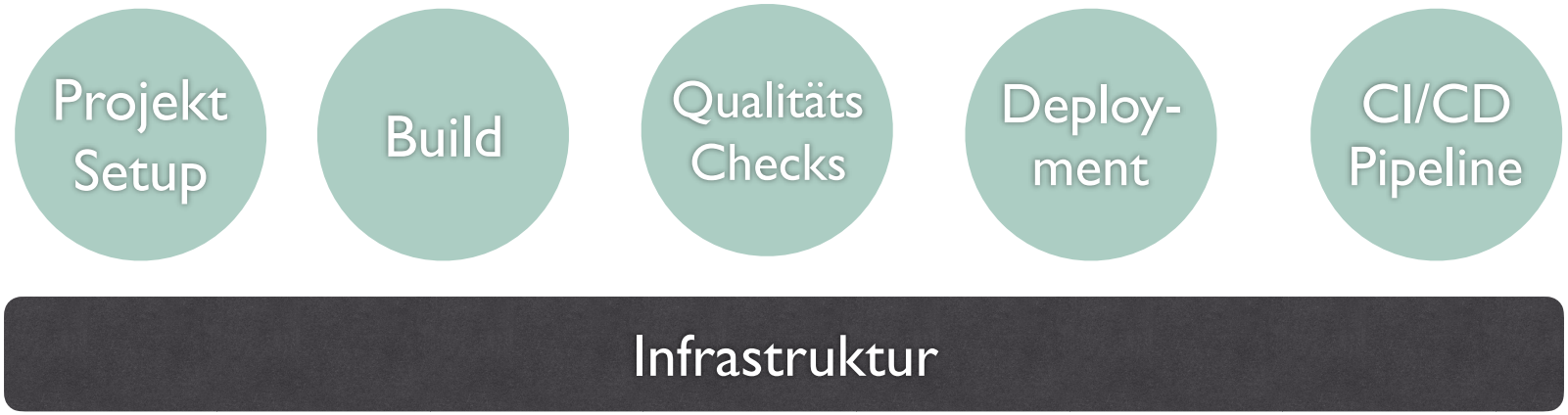
# Alles als Code



# Alles als Code!? Warum?

- Automatisierung von sonst manuellen Schritten
- Versionierter Zustand
  - Nachvollziehbarkeit
- Offen für Modifikationen

# Alles als Code



# Umsetzung

# Infrastruktur als Code

- iOS Buildserver
  - Xcode Installation nötig
  - CI/CD Komponenten: Jenkins, GitLab CI Runner
- Buildservice
  - Travis CI, Circle CI, Bitrise
  - Vorprovisionierte VM

# Build Services

- Travis CI, Circle CI, GitHub Actions, Bitrise
- Häufig auf GitHub / Bitbucket Lösungen beschränkt
  - Git Hook / API Integration
- Konfiguration als Datei im Projektrepository
  - `.travis.yml`, `./circleci/config.yml`

# Build Services

- Travis CI (.travis.yml)

```
language: objective-c
osx_image: xcode11
# …
jobs:
  include:
    - name: Pull Request
      if: type = pull_request
      script: fastlane ios test
```

# Eigene Hardware

- Selbstadministrierte Installation von benötigten Komponenten
  - Xcode
  - Homebrew
  - Ruby (RVM, rbenv, chruby)
- iOS Projekt enthält alle anderen Abhängigkeiten
- Lösung: Ansible

# Instrastruktur als Code - Ansible

- Software für zentralisierte Orchestrierung von verteilten Server
- Verbindung über SSH zu einem Build Server
  - ohne Client-Agent
- Vordefinierte oder eigene Befehle zum installieren von Komponenten
- Kann auch selbst über eine CI Pipeline ausgeführt werden

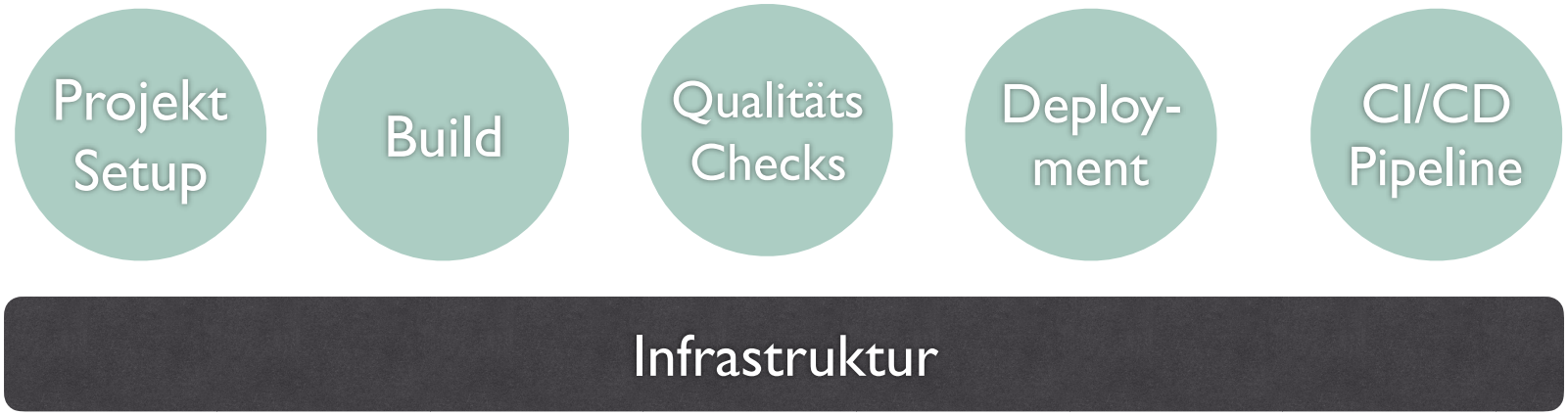
# Ansible - Beispiel

```
- name: Install macOS updates
  command: softwareupdate -i -a --restart

- name: Disable computer sleep
  command: systemsetup -setcomputersleep Never

- name: Install Homebrew
  command: bash -c "\curl -fsSL https://raw.githubusercontent.com/
Homebrew/install/master/install | ruby"
```

# Alles als Code



# Projekt Setup

- Installiere alle Abhängigkeiten des Projekts
  - Homebrew Packages, Ruby Gems installieren
  - Carthage / Cocoapods ausführen
- Sollte für (neue) Teammitglieder und dem Buildserver funktionieren
  - Der Buildserver kann die README.md Datei nicht lesen
- Sollte aus einem Befehl bestehen

# Projekt Setup

- `make` mit einem `Makefile` als Lösung
  - `make Targets` drücken Build-Schritte aus
  - Einzelne Tasks können als `Targets` abgebildet werden
  - Reihenfolge für die Ausführung kann festgelegt werden
  - Tasks können auch gezielt ausgeführt werden

# make - Beispiel

```
# Install Cocopods dependencies
install_cocoapods:
    $(info Install Cocoapods ...)
    bundle exec pod install
```

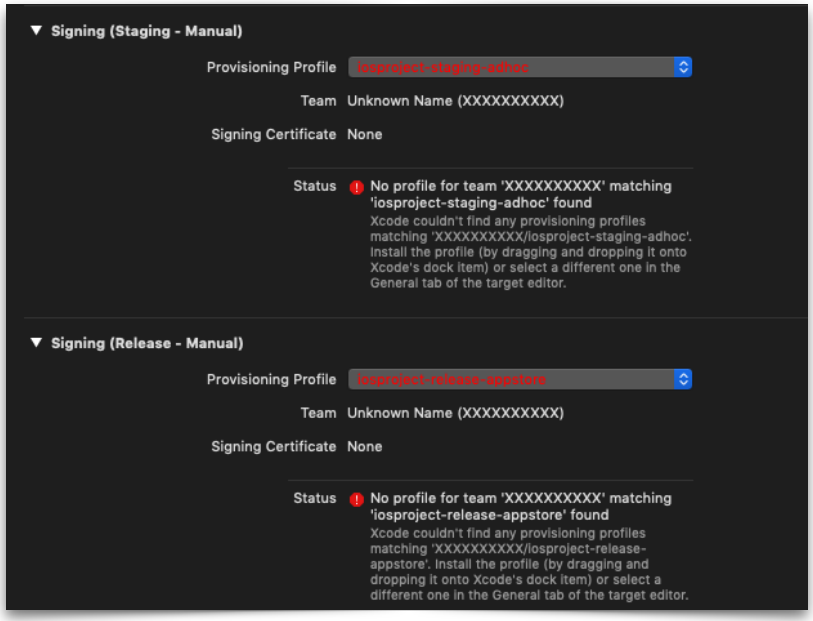
`make install_cocoapods`

```
# Makefile
default: bootstrap
bootstrap: \
    check_for_rvm \
    check_for_homebrew \
    update_homebrew \
    install_bundler_gem \
    install_ruby_gems \
    install_cocoapods
```

`make bootstrap`

# Projekt / Build Setup

- Konfiguration des Buildartefakts im Projekt
  - Zertifikat & Profile verschlüsselt im Repo
- Unabhängig vom Buildservice & -tool
- “Buildtool” konfiguriert Buildumgebung



# Buildtool

- fastlane (https://fastlane.tools/) als Buildtool
- Ruby Frontend für `xcodebuild` als "DSL"
- Aktionen für typische, wiederkehrende Tasks in `Fastfile` in Repo
  - Individualisierung der App: Badge über App Icon (Version)
  - Verteilung der App: App Store Connect, MS AppCenter
- Erweiterbar über Plugins oder eigenen Ruby Code

# fastlane - Fastfile

```
# Fastfile
  lane :test do
    run_tests(scheme: 'iOSProject')
  end

  lane :release do
    build_ios_app(
      scheme: 'iOSProject',
      export_method: 'app-store',
      configuration: 'Release'
    )
    upload_to_testflight(
     skip_submission: true
    )
  end
```

Ausführung:

```
$ fastlane test
$ fastlane release
```

# fastlane - Anwendungsfälle

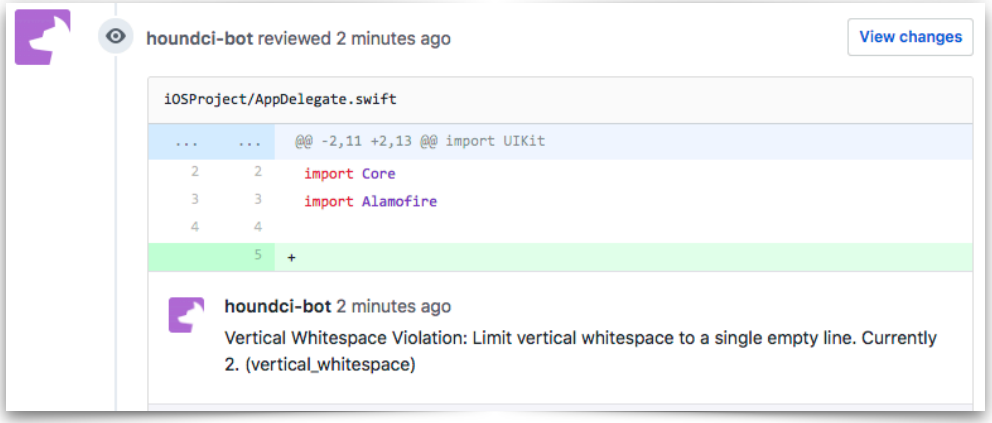
- Erstellt Buildumgebung auf Buildserver / -service
  - Entschlüsseln von Zertifikat & Profile
  - Erstellt Keychain und kopiert Profile
- Individualisierung der App
- Auswahl der Buildkonfiguration (Dev-, Stage-, Prod-Umgebung)
  - mit Xcode Konfiguration

# Qualitätschecks

- Vordefinierte Team-Regeln
  - Code-Style, Testabdeckung, Commit-Regeln
- Lokal & Pull Request
  - Lokal: Xcode “Run Script Phase”
  - Buildserver: fastlane
- Ziel: nervtötende Diskussionen vom Team fernhalten

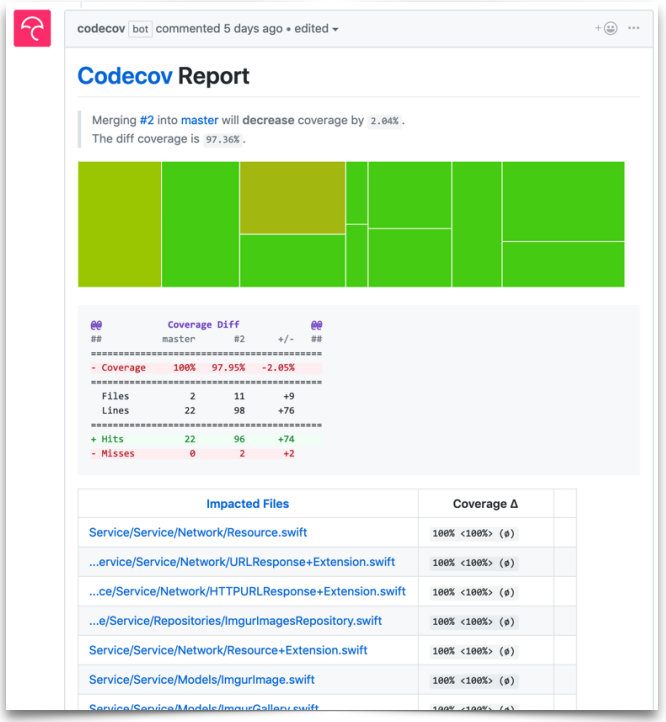
# Code Style - Swiftlint

- Verhindert Code Smells
  - force-unwarps
- Einheitlicher Code Style
- Regeln in `.swiftlint` Datei hinterlegt
- Lokal & Pull Requests

# Testabdeckung - Codecov

- Testabdeckung als “Qualitätsfaktor”
- Testabdeckung bei neuen Features
- Visualisierung um Testlücken zu erkennen
  - Codecov als Service für GH Repos



# Commit-Regeln - Danger

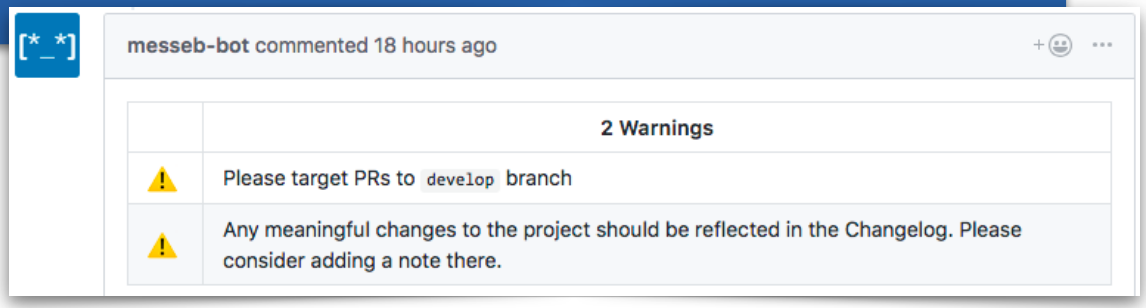
- “Linter” für Pull Request (https://danger.systems/)
- Eigenes Regelset (`Dangerfile`)
  - z.B. Eintrag in `CHANGELOG.md` Datei fehlt
- Integration über einen Bot-Nutzer in GitHub, Gitlab & Bitbucket
- Verschiedene Loglevel: Info, Warning, Error
  - Beeinflußt das Ergebnis des Buildjobs

# Commit-Regeln - Danger

```
# Dangerfile
# changes in the project folder
has_app_changes = !git.modified_files.grep(/iOSProject/).empty?

# Changelog entries are required for changes to library files.
no_changelog_entry = !git.modified_files.include?("CHANGELOG.md")

if has_app_changes && no_changelog_entry
  warn("Any meaningful changes to the project ....")
end
```

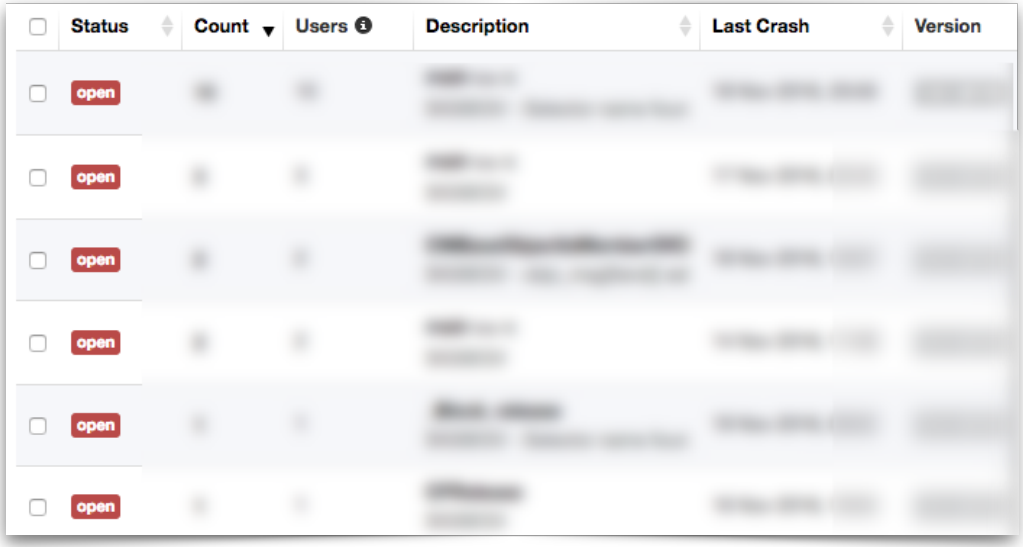


# Delivery / Deployment

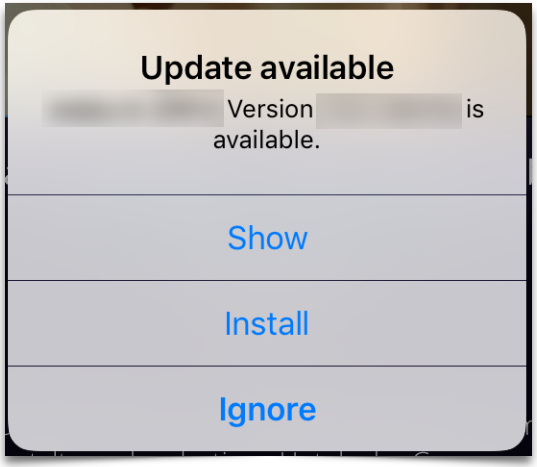
- In House
  - Während der Entwicklung mit Auto-Updates
  - Services: Crashlytics Beta, MS AppCenter
- App Store
  - Testflight
- fastlane Plugins für die Verteilung

# Delivery / Deployment

## Crash-Reports




## Auto-Updates



# Pipelines

- GitHub Actions, GitLab CI/CD, Travis CI
- Beschreibung des CI/CD Prozesses
  - Im App Repository
  - Nicht in Jenkins Eingabefeldern / visueller Workflow
- Führt gleichen Code aus, wie ein Entwickler
  - `Makefile` Abstraktion

# Pipelines - Beispiel

```
# .gitlab-ci.yml (GitLab CI/CD)
before_script:
  - make bootstrap

stages:
  - test
  - build

release_build:
  stage: build
  only:
    - /^release\/.+$/
  except:
    - branches
  script:
    - make release_build
```

```
# .travis.yml (Travis CI)
language: objective-c
osx_image: xcode11

install: make bootstrap

stages:
  - build

jobs:
  include:
    - name: Release
      if: tag IS present
      stage: build
      script: make release_build
```

# Pipelines - Regeln

- Feature Branch / Pull Request
  - Tests, Qualitätschecks
- Merge auf `develop` / `master` Branch
  - Bauen & Distribution der Staging-App
- Release Tag
  - Bauen & Distribution der Release-App

# Fazit

# Fazit

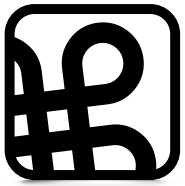
- IaaS: Build Services sind eine gute Lösung
- Projekt Setup: Konfiguriere das Projekt “build-ready”
- Build / Distribution: fastlane benutzen
- Qualitätschecks: während der Entwicklung automatisiert ausführen
- Pipeline: Sichtbarkeit für Entwickler & Ausführung des gleichen Codes

Fragen?

# Code

- iOS Project Template
  - https://github.com/messeb/ios-project-template
- Setup an iOS Project Environment
  - https://github.com/messeb/ios-project-env-setup
- Ansible iOS Build Server Provisioning
  - https://github.com/messeb/ansible-ios-build-server

Vielen Dank



# Macoun