Bernhard Meindl
Gregor de Cilla
Alexander Kowarik
Statistics Austria

Vienna
December 2020

DevOps and R

Experiences at Statistics Austria

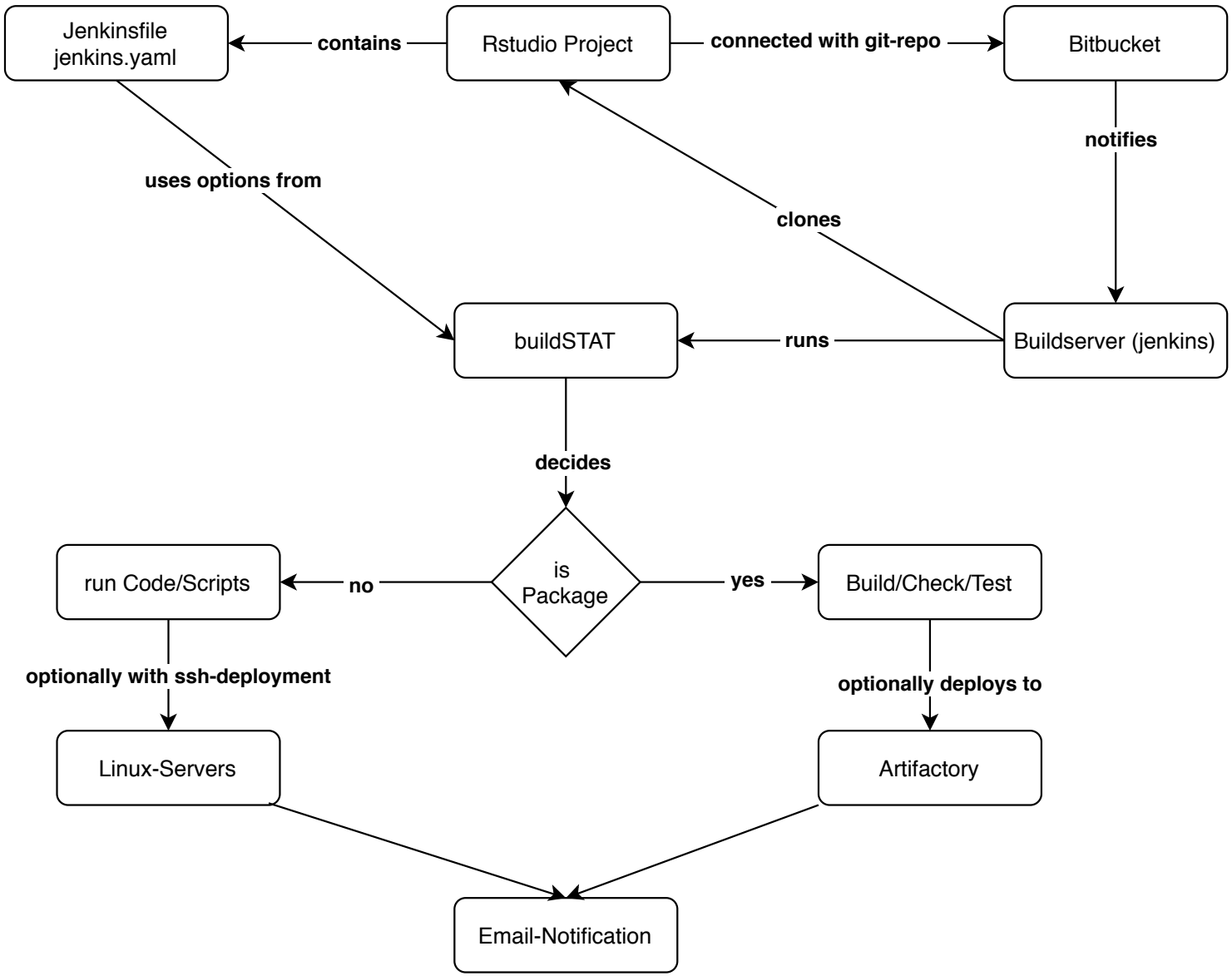
Outline

- **Goal**:
  - outlining the conceptual framework we have implemented at **Statistics Austria** to
    - facilitate the automated building
    - deployment of `R` projects and packages
- Our **environment** consists of:
  - a Linux-Server (`Ubuntu`)
  - `R`-Server (`Rstudio-Server Pro`)
  - a Git-server (`Bitbucket`)
  - a Build-server (`Jenkins`)
  - an Artifact repository manager (`JFrog Artifactory`)

Server Infrastructure

- R Server: two Ubuntu server (prod + dev)
  - RStudio professional
  - various R-installations with a well-maintained library of packages
- BitBucket: Enterprise-edition of Bitbucket server
  - has support for Webhooks and features a REST API
- Jenkins Server: also supports a REST API
- Jfrog Artifactory: used to host
  - CRAN-like repo for internal packages
  - cached versions of external CRAN-mirror

Workflow


Jenkinsfile
jenkins.yaml
contains
Rstudio Project
connected with git-repo
Bitbucket
notifies
uses options from
clones
buildSTAT
runs
Buildserver (jenkins)
decides
is
Package
no
yes
run Code/Scripts
Build/Check/Test
optionally with ssh-deployment
optionally deploys to
Linux-Servers
Artifactory
Email-Notification

Desired Workflow

- a RStudio-project is connected a Bitbucket repository (using git)
- Bitbucket notifies the Build-Server (Jenkins) if the repo was updated (via webhook)
- Jenkins runs a pipeline using special inputs contained in the repository
- the pipeline loads `R`-pkg `buildSTAT` that checks what should be built
  - **Case 1**: `R`-package:
    - dependencies are resolved; package is built, checked and tested
    - optionally uploaded to the artifactory repository (internal CRAN-repo)
    - the package can then be installed via `install.packages()`
  - **Case 2**: `R`-Project
    - scripts in the project are run
    - possible deployments via ssh

Specify build-behaviour via yaml-config (1)

- Repos that are built on Jenkins need to include two files
  - `Jenkinsfile`: defines the pipeline (and R-version) under which the project/package should be built
  - `jenkins.yaml`: defines options that are used during the build such as
    - installation of additional `R` or system-packages
    - who should be notified
    - how should be dealt with warnings, notes, ...
    - should the package (on a successful build) be pushed to the artifactory
    - should a pkgdown-site be created?

Specify build-behaviour via yaml-config (2)

- Some further possibilities:
  - caching of `R` packages and/or a `TinyTex` installation can be required to speed-up build-times
  - definition of (optional) scripts (`R`, `bash`, `python`, ...) that should run before/after building
- `yaml`-definition has sensible defaults (few settings required)

```
## file: Jenkinsfile
@Library("jenkins-r-shared-library") _
rPipelineYAML(
  docker_image: "library/r-base:4.0.3-stat-latest",
  config: "jenkins.yaml"
)

## file: jenkins.yaml
cache:
- r
mail:
- bernhard.meind@statistik.gv.at
```

Setup with REST API calls (1)

- quite a few components play together
- we wanted an easy way for our colleagues to get started
- **required resources**:
  - an RStudio project
  - a BitBucket repository
  - A Jenkins job
- **next steps**:
  - the Rstudio-Project needs to be linked with Bitbucket
  - Bitbucket needs to be linked with Jenkins
- **Idea:**
  - create a workflow package `useSTAT` similar to `usethis`
  - the required steps are facilitated by exploiting the available APIs of BitBucket and Jenkins

Setup with REST API calls (2)

- **required steps with `useSTAT`:**
  - open a new `R` session in Rstudio and run the following steps

```
# create a new RStudio project (package)
# from an internal template
useSTAT::create_stat_package("~/projects/newPackage")

# create a new BitBucket repo and links it
# to the RStudio project
useSTAT::use_git()
useSTAT::use_statbucket()

# creates a Jenkins job and links it with BitBucket
useSTAT::use_jenkins()
```

- we can now push changes and watch Jenkins go to work!

Pros/Cons of this approach

- **Pros**:
  - easy for people without DevOps experience to set up a new package
  - consistent package definitions (e.g adding the git-commit id to `DESCRIPTION`)
  - consistent quality checks across internal packages
- **Cons**:
  - API keys are required for the `useSTAT` $\longrightarrow$ managed with internal packages `authSTAT`, `apiSTAT`
  - Updating the yaml definition (new options) is tedious to test because all tests need to run in docker containers

Experience

- overall, few problems
- the cons of the initial setup outweigh the benefits (consistency)
- quite happy as new features (e.g secrets) in the pipeline can be implemented easily via `buildSTAT`