There Should be One Obvious Way to Bring Python into Production

Sebastian Neubauer

sebastian.neubauer@blue-yonder.com
@sebineubauer

europython
9-16 JULY 2017 Rimini

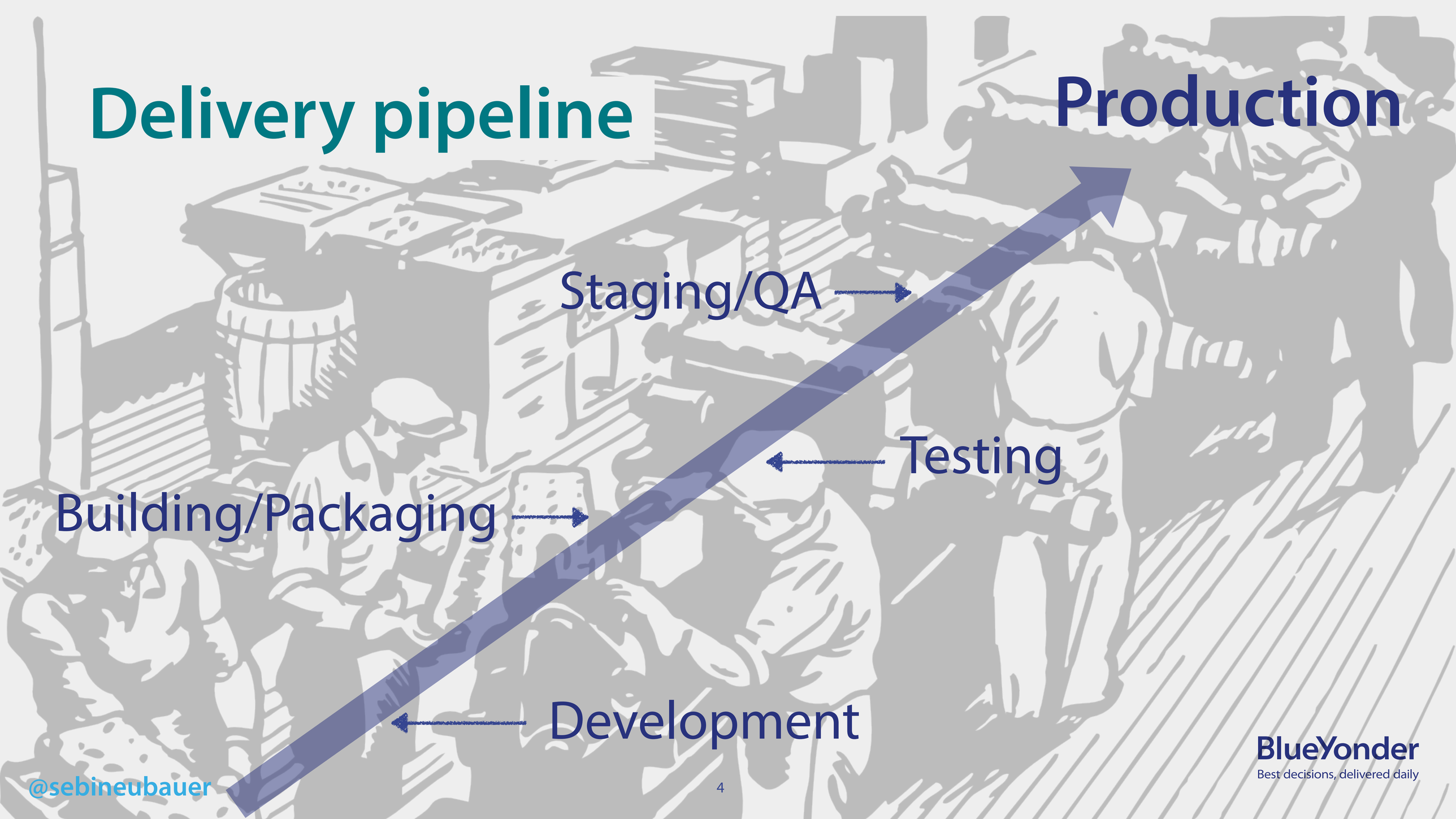
BlueYonder
Best decisions, delivered daily

Agenda

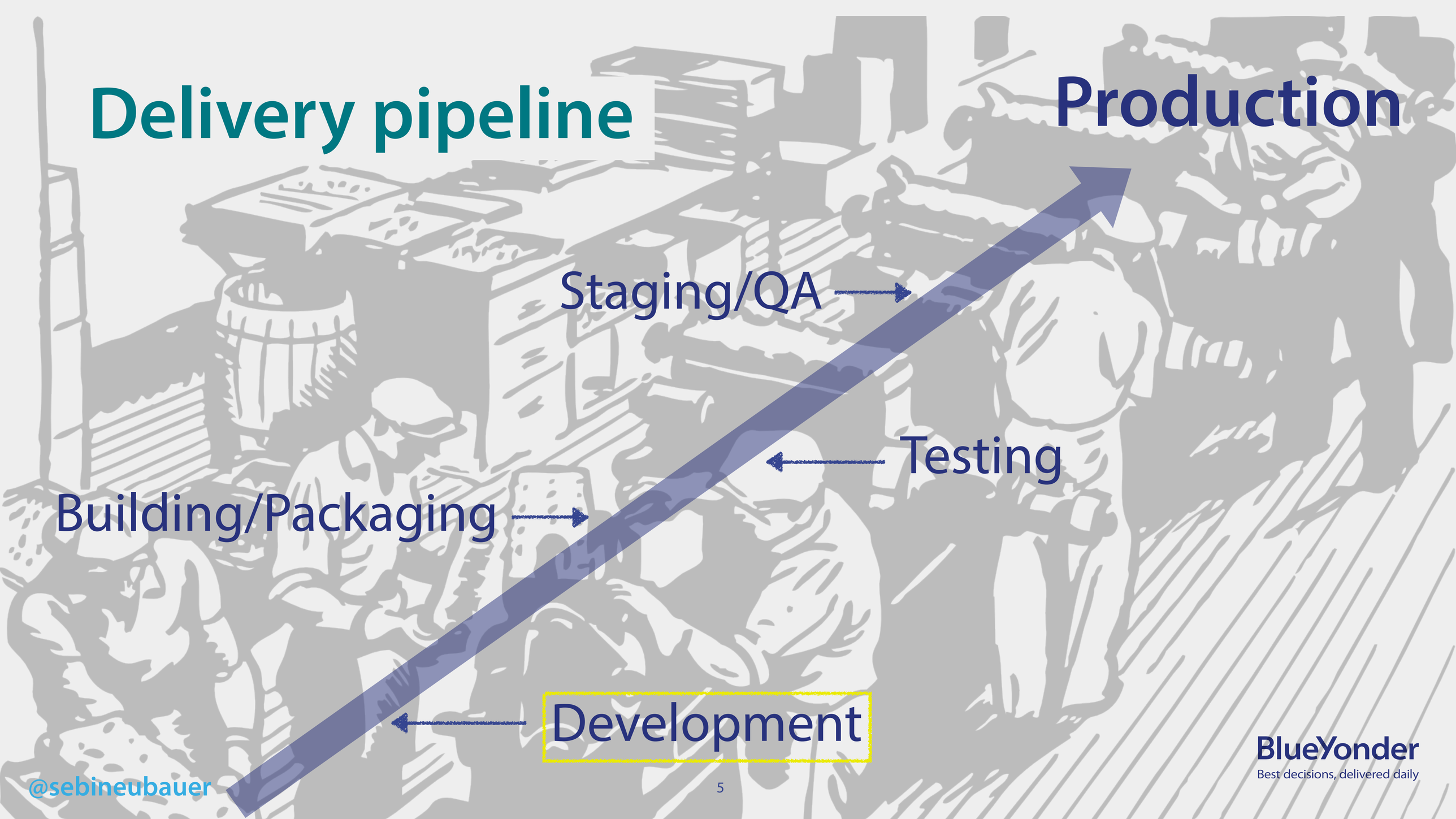
- What are we talking about and why?
  - Delivery pipeline
  - Dependencies
  - Packaging
- What is the current state?
  - A walk through the different possibilities
  - Summarizing all the pros and cons
- Can we find a better solution?
  - How does the future look like?
  - Discussion: what could the „one obvious way“ be?

What are we talking about and why?

Delivery pipeline



Delivery pipeline



Development

Required:

- Fast iteration cycles, fast changes
- Automated tests can be executed

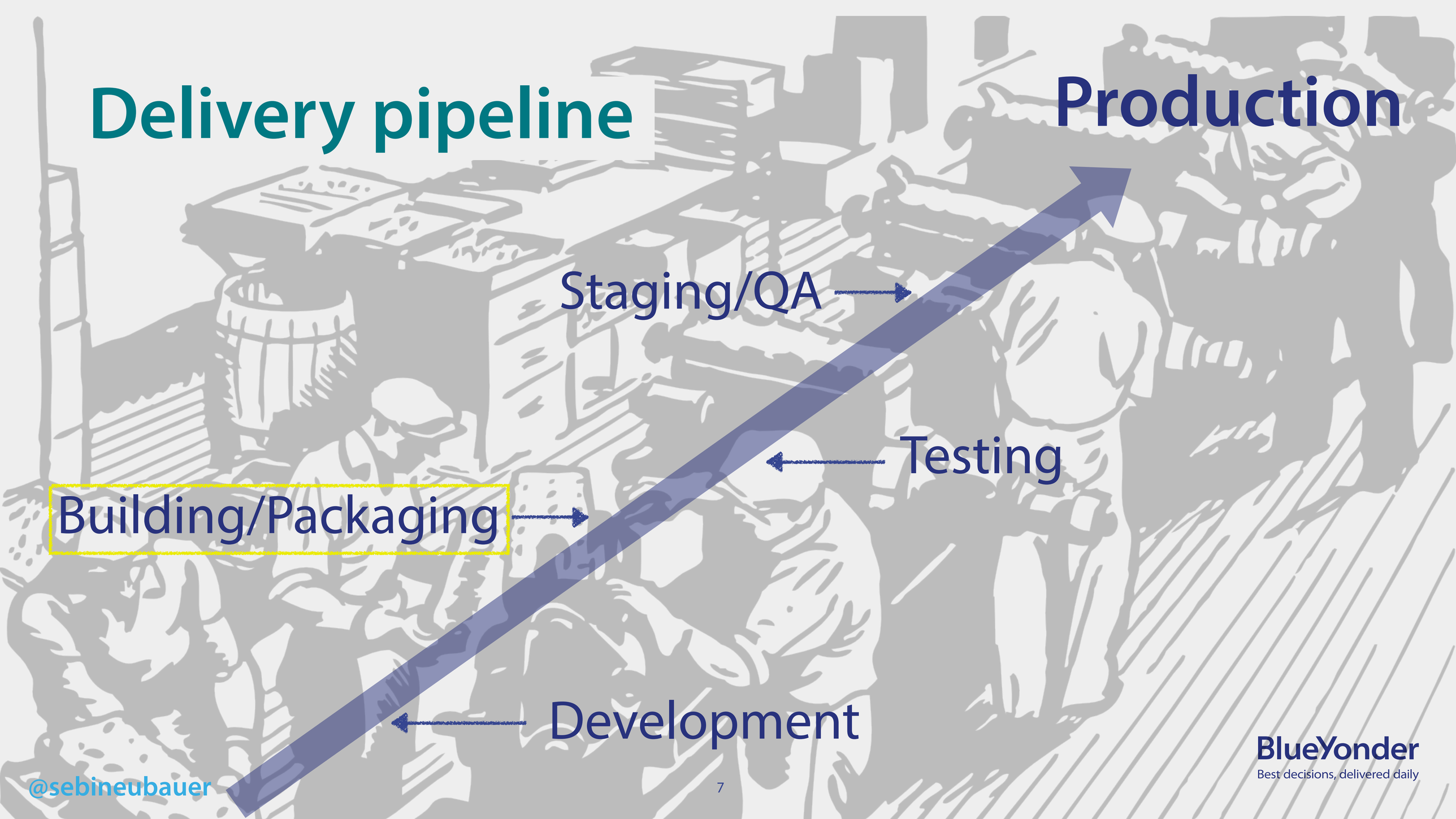
Nice to have:

- Production like local environment

Risks:

- „Works on my machine!“
- Dirty working directory

Delivery pipeline



Building/Packaging

Required:

- Build once, use everywhere
- Possibility to compile for the target systems
- Build uniquely versioned, signed packages

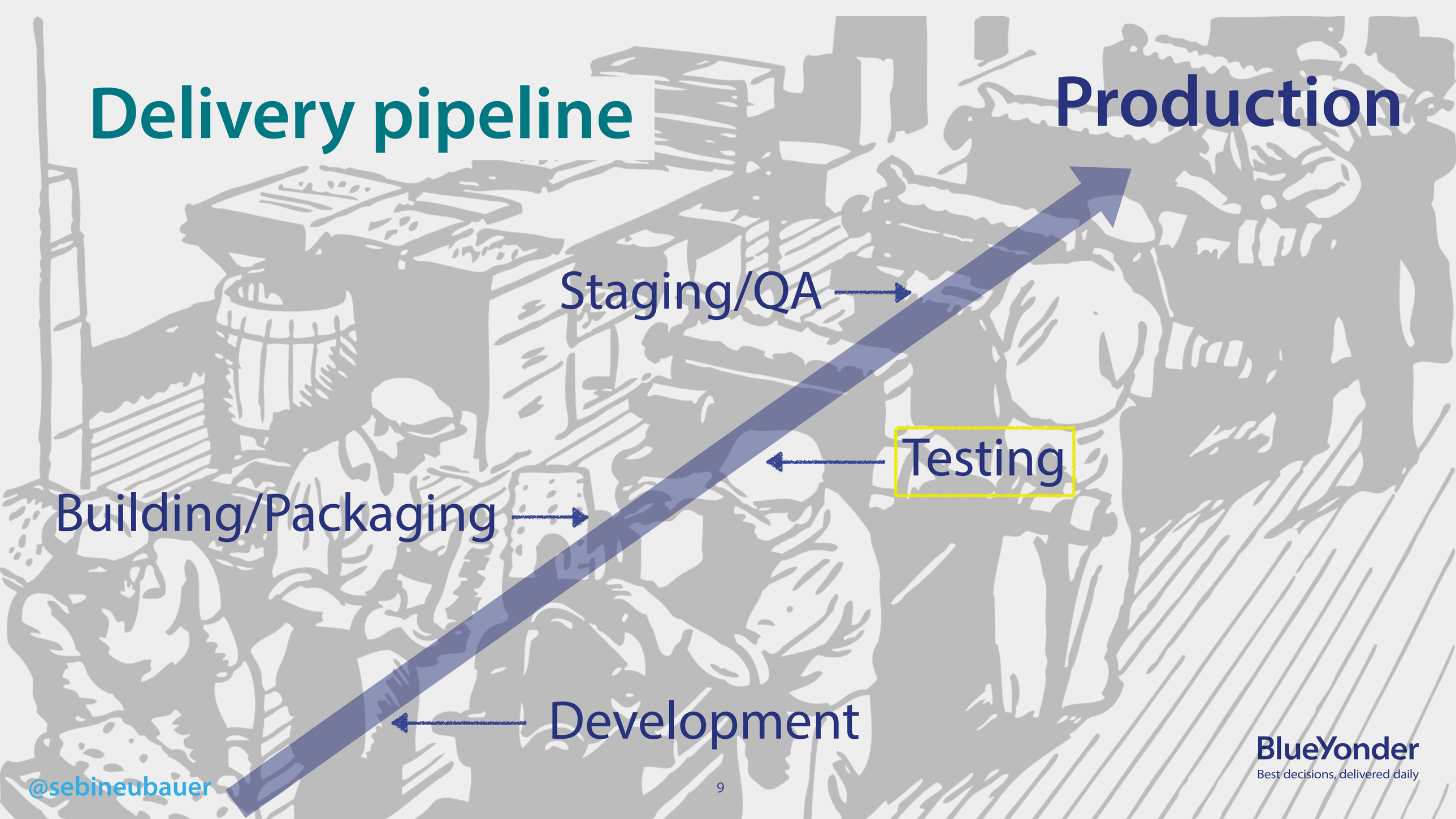
Nice to have:

- Upload to an artifact repository

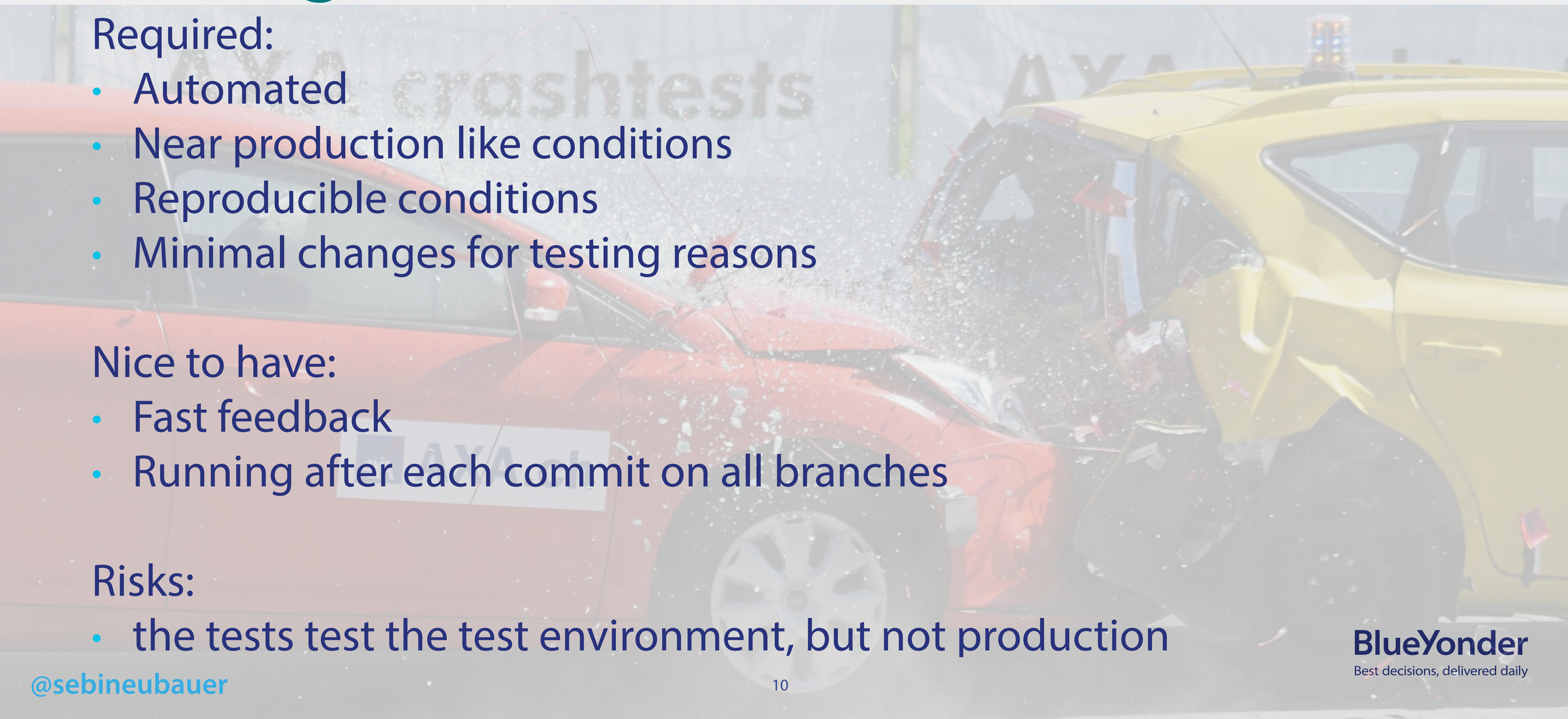
Risks:

- Misconfiguration of the build environment

Delivery pipeline



Testing



Required:

- Automated
- Near production like conditions
- Reproducible conditions
- Minimal changes for testing reasons

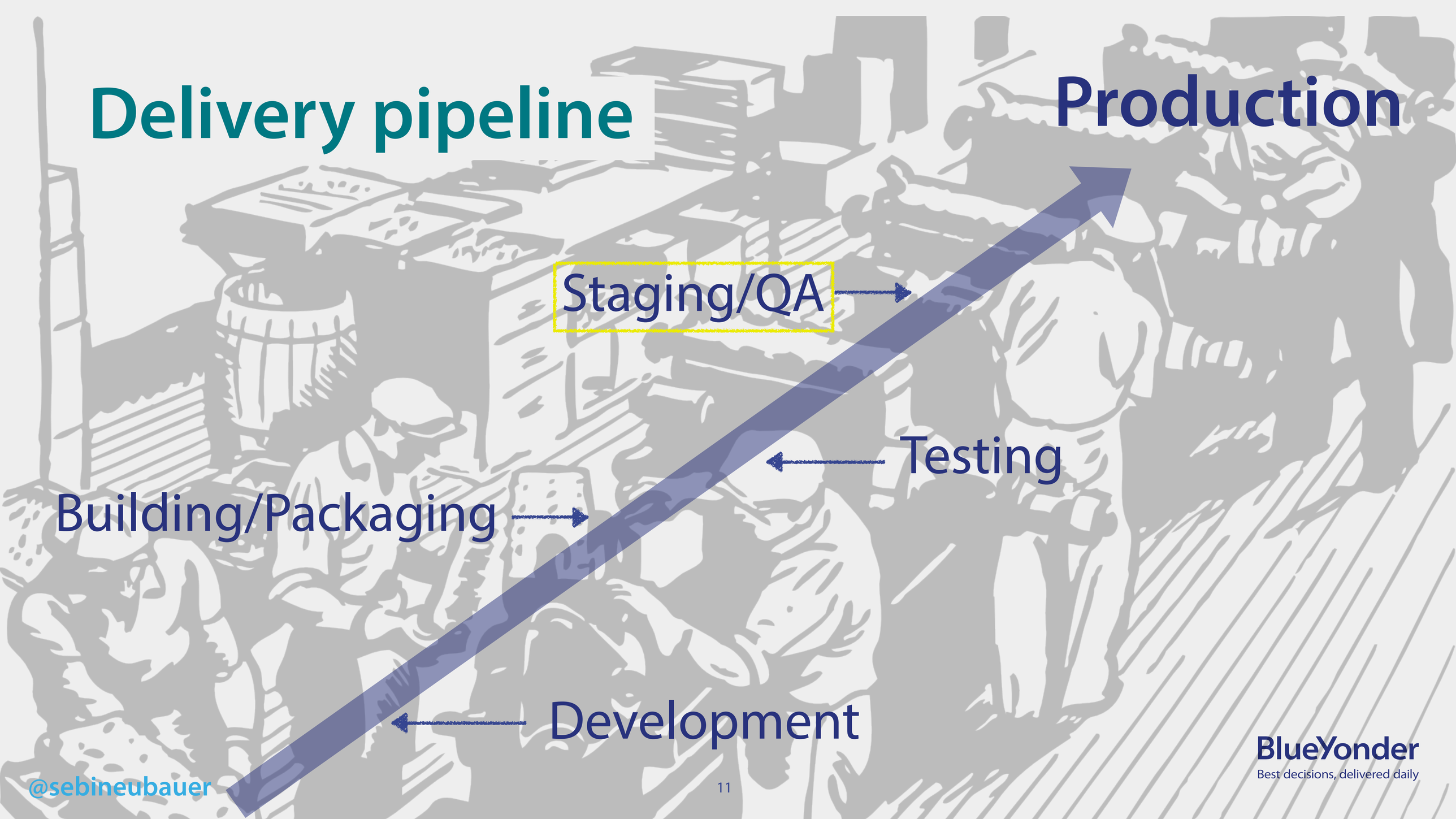
Nice to have:

- Fast feedback
- Running after each commit on all branches

Risks:

- the tests test the test environment, but not production

Delivery pipeline



Staging/QA

Requirement:

- Automated deploy in production like environment
- Nearly no changes for testing purposes

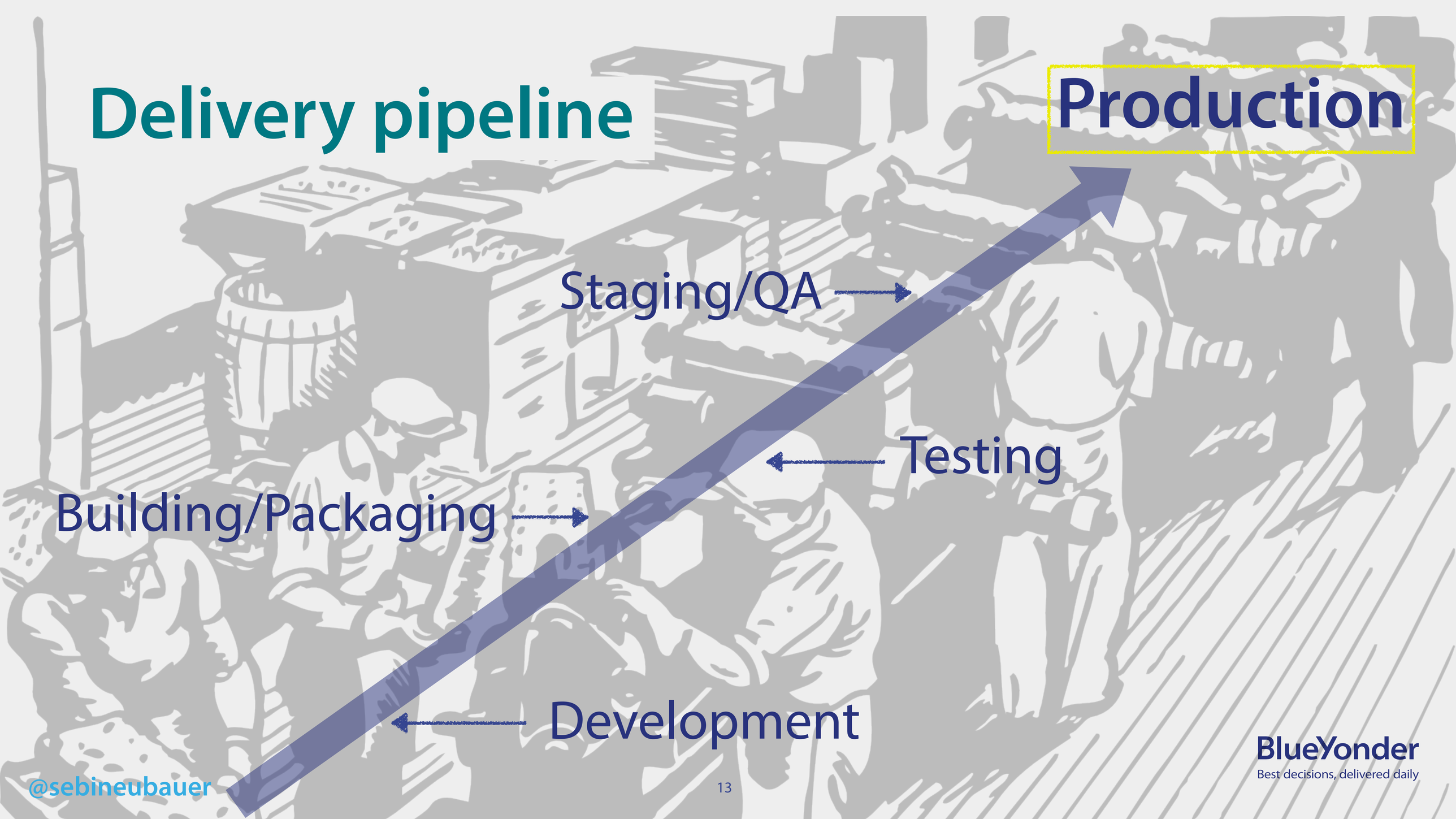
Nice to have

- A real clone of the production system
- Possibility to run A/B tests on that system

Risks:

- outdated, manually maintained setup

Delivery pipeline



Production

Required:

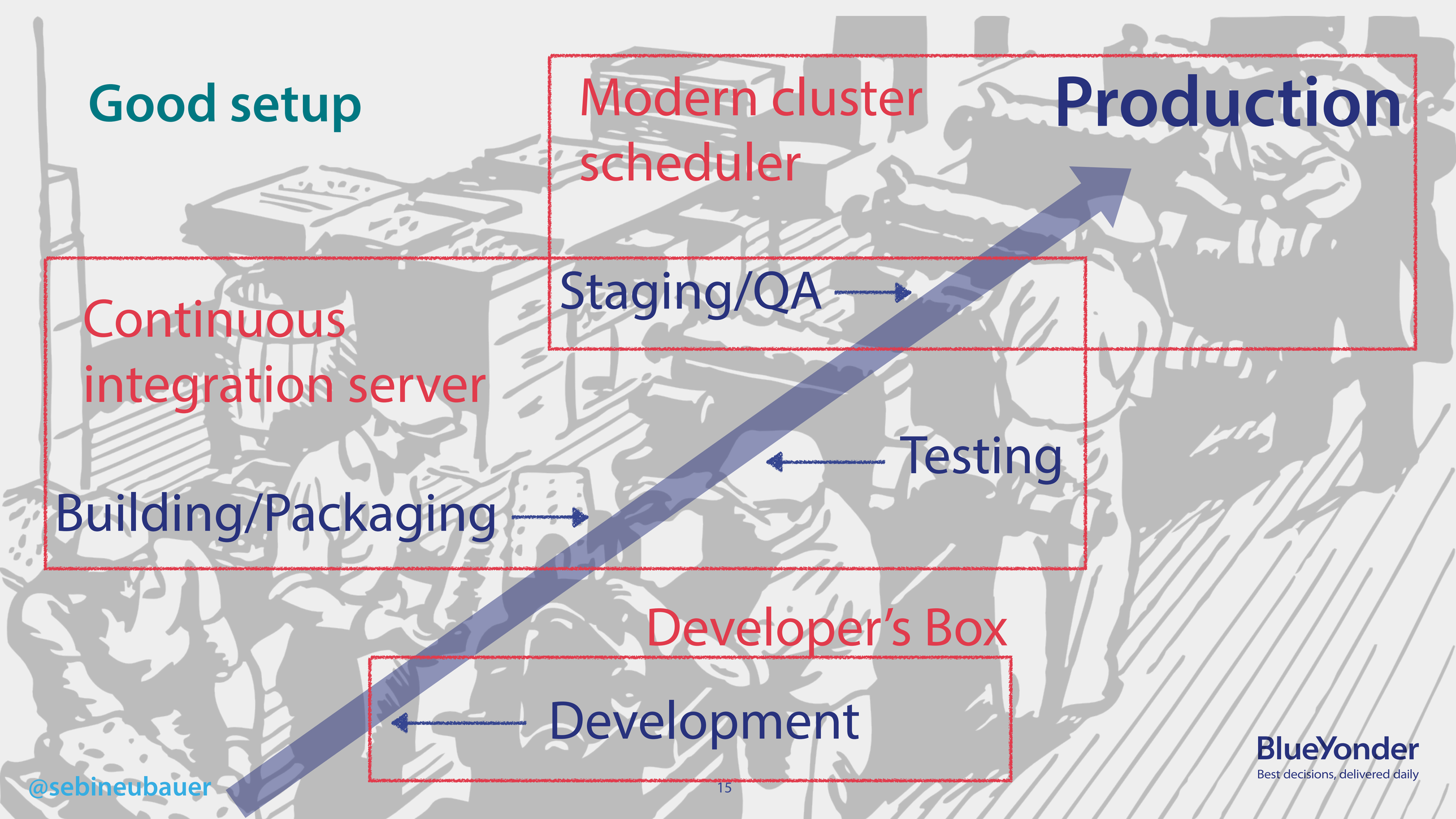
- No compiler
- No internet
- Health monitoring

Nice to have:

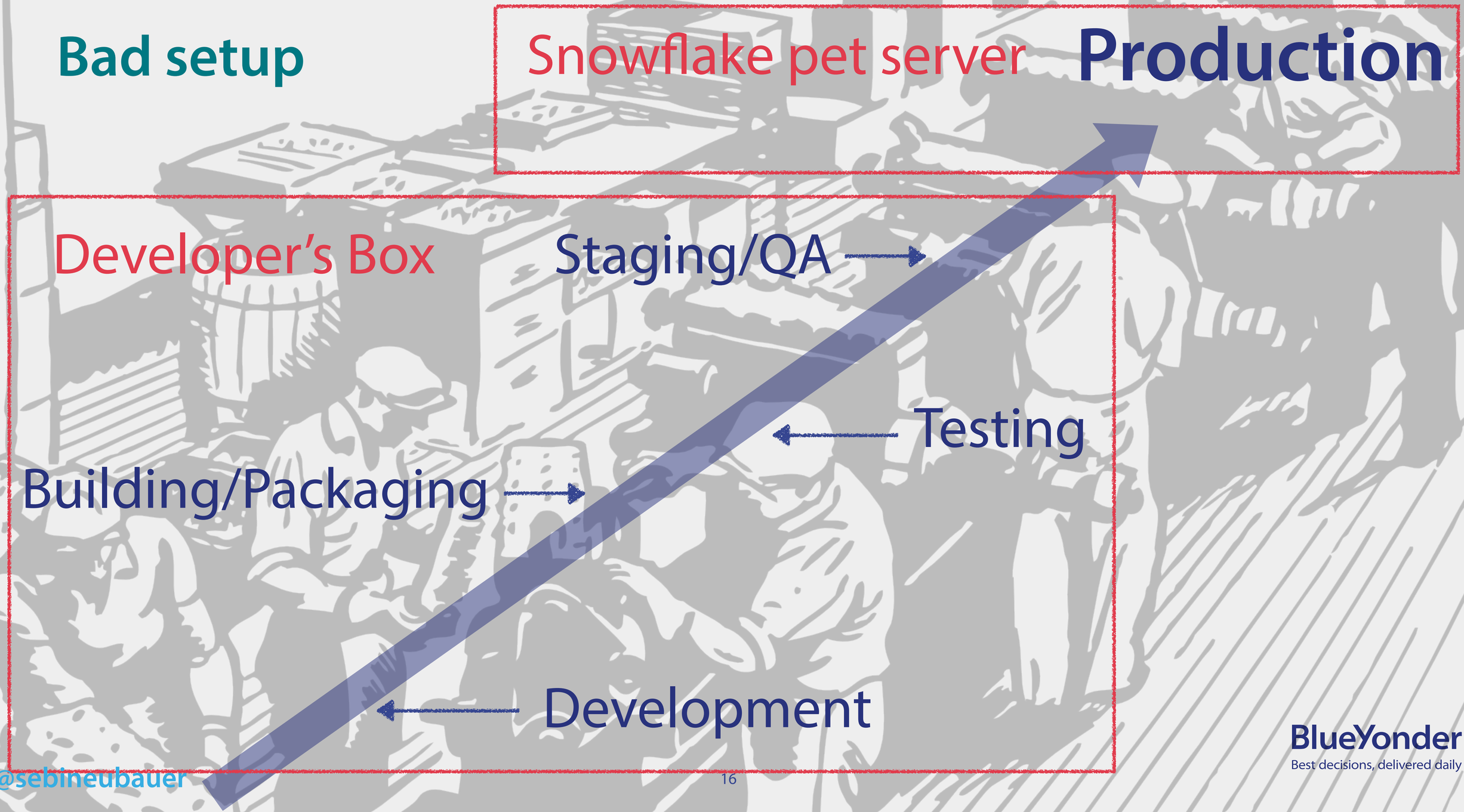
- Automated deploy
- Automatic monitoring
- Automatic self-healing
- Automatic rolling update and roll back

Risks:

- your business is going down…


Good setup
Modern cluster scheduler
Production
Staging/QA
Continuous integration server
Testing
Building/Packaging
Developer's Box
Development

Bad setup


Snowflake pet server
Production
Developer's Box
Staging/QA
Testing
Building/Packaging
Development

Dependencies

„All shared software components that need to be present in the correct version so that the application works correctly“

fictitious definition

Dependency Hell

Problems:

- Transitive dependencies can have conflicting version requirements
- Python only knows application „global“ dependencies (javascript has local dependencies)
- Pip (still) doesn’t have proper dependency resolution (gh #988 open since 11 Jun 2013, but GSoC 2017 project, fingers crossed)
- System python dependencies interfere with application dependencies

Package management in python

package manager: pip
package format: wheel

- still much confusion around setuptools, distutils, eggs…
- many „best practices“ in stack overflow & co. outdated
- no standard templating for packages: see pyscaffold, versioneer…
- feels like lack of interest in the community…
- but: it has gotten way better in the last years:
  - setup.cfg
  - setuptools_scm

For details see: https://ep2017.europython.eu/conference/talks/python-packaging-current-state-and-overview by @webGandi

Package manager hell

System dependencies	Language dependencies
operating system, libraries	language specific libraries, frameworks
yum, apt-get, homebrew, vcpks…	pip, npm, conan, cpan, maven, composer, cargo, godep, gem,…
frequent security updates	almost no security updates
„operations“ take care	„developers“ take care
root/system wide	user space/virtualenv

Package manager hell

System dependencies	Language dependencies
operating system, librari…	language specific li…aries, frameworks
yum, apt-get, …omebrew, vcpks…	…ip, npm, conan, cpan, …aven, composer, cargo, godep, …em,…
frequent se…urity updates	alm… …security updates
„infrastructur…“ takes care	„developers“ take care
root/system wide	user space/virtualenv

Package manager hell

Where does it come from historically?

- disk space and bandwidth expensive
- separation between dev and ops
- single language environments
- rise of open source and sharing culture
- no package manager solved everything

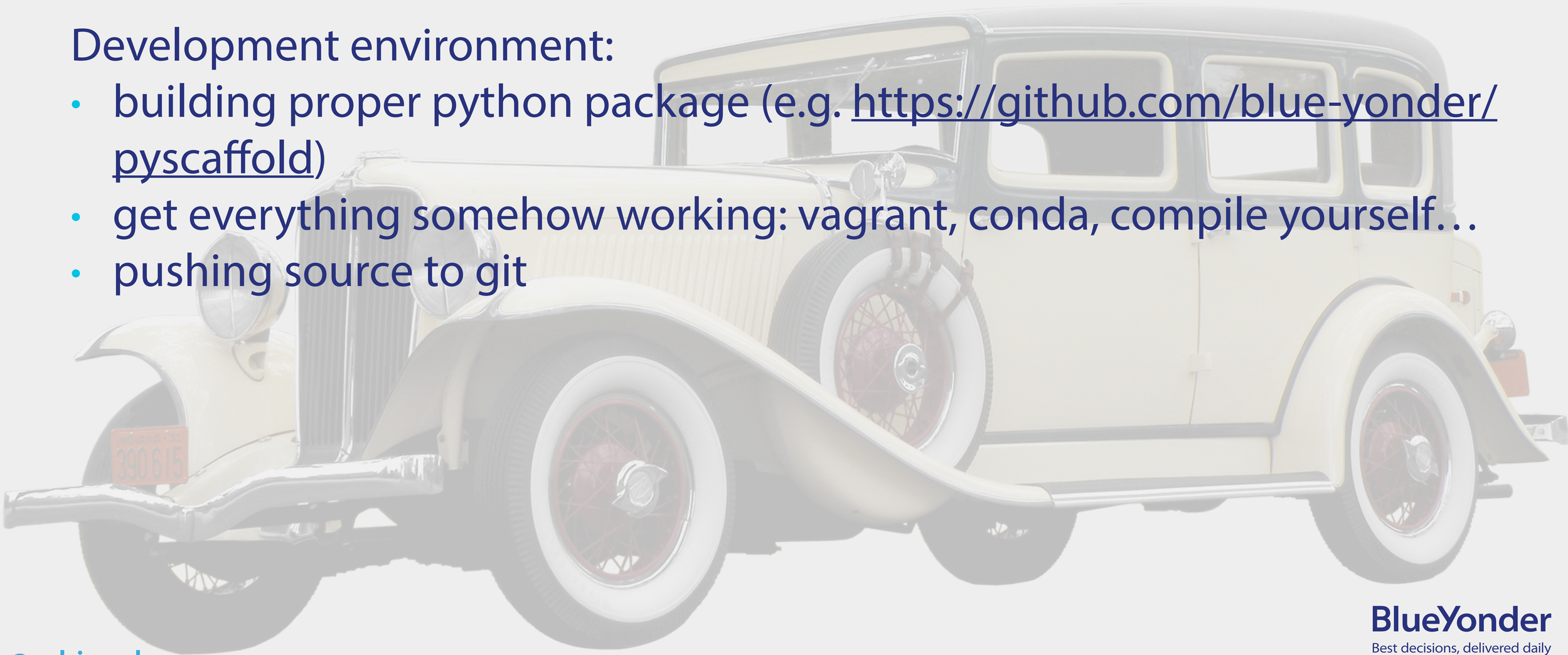
What is the current state?

a walk through the different possibilities

The classical approach

Development environment:

- building proper python package (e.g. https://github.com/blue-yonder/pyscaffold)
- get everything somehow working: vagrant, conda, compile yourself…
- pushing source to git



The classical approach

On Jenkins:

- building artifacts,
- testing,
- release: packaging (wheels) and publishing to an pypi compatible artifact repository (artifactory, devpi…)

BlueYonder
Best decisions, delivered daily

The classical approach

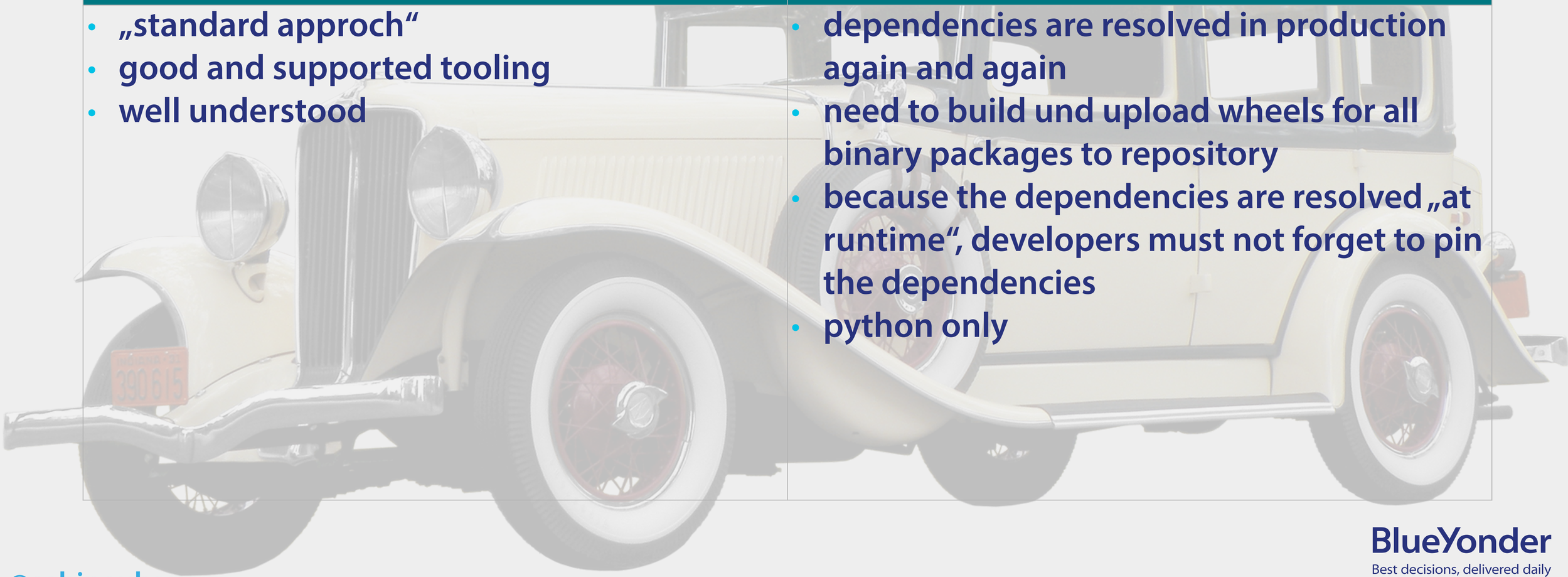
In production:

- standard virtualenv and pip
- application gets installed from repo together with dependencies
- OS and system dependencies are maintained separately

BlueYonder
Best decisions, delivered daily

The classical approach

Pro	Con
• „standard approch“ • good and supported tooling • well understood	• dependencies are resolved in production again and again • need to build und upload wheels for all binary packages to repository • because the dependencies are resolved „at runtime“, developers must not forget to pin the dependencies • python only



The „conserve virtualenv“ approach

- idea: build a virtualenv, then pack it, ship it and unpack on the target system
- several similar implementations:
  - platter: simple virtualenv and wheels
  - pex: new virtualenv implementation, includes executed command
  - dh-virtualenv: virtualenvs packaged in debian packages
- done once in build step

The „conserve virtualenv“ approach

Pro	Con

• no resolving of dependencies on target host
• no dependency to a pypi server
• „push or pull model“ possible, either you copy the archive to the target, or it pulls from a repo
• depending on the implementation (e.g. platter) it integrates well in „standard“ workflow with standard tools
	• system packages not included
	• need to compile for the exact target system
	• no standard repository:
	  • implement push infrastructure
	  • implement a repository (e.g. s3)
	• python only

The OS package approach

- idea: package the application as a standard OS package, e.g. debian package
- this way you can install the application with „`apt-get install`“ on the target machines
- deb package building is done once in the build step
- there are some few tools that help you:
  - stdeb: build deb packages with one command (can’t get it to work, last commit 2 years ago :face_with_rolling_eyes:)
  - dh-virtualenv
- for all dependencies, you either have to make deb packages too, or you bundle them up (see dh-virtualenv)

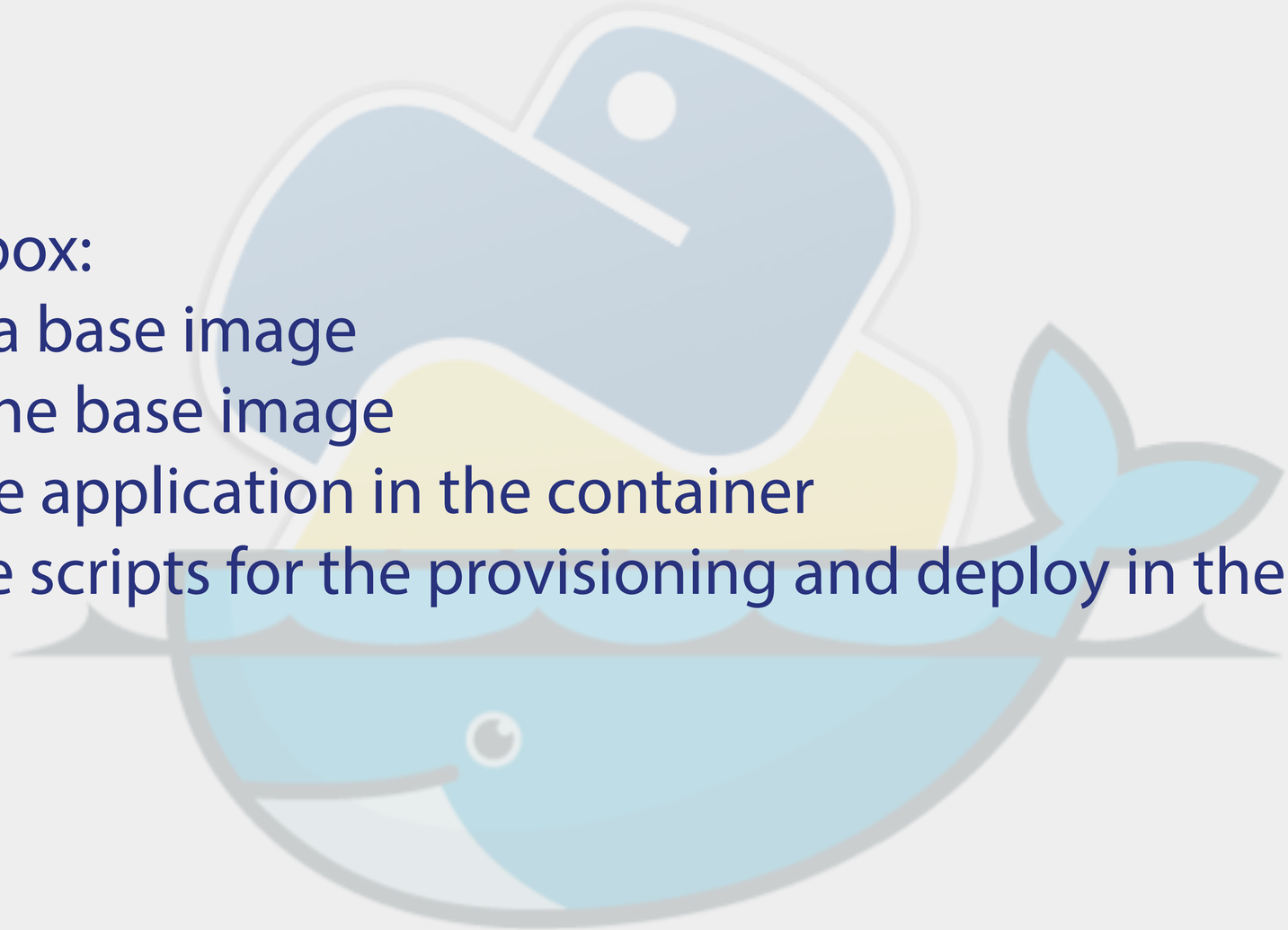
The OS package approach

Pro	Con
• integrates well with system maintenance • just one package manager needed • standard debian repository	• tooling seems to be very badly maintained • no tooling for dependency management, you have to create packages and declare the dependencies yourself (or use dh-virtualenv) • you need a debian repository • working with deb packages is often: globally installed by root, not always what one needs

The „container as PM“ approach

Developer’s box:

- download a base image
- provision the base image
- develop the application in the container
- commit the scripts for the provisioning and deploy in the container

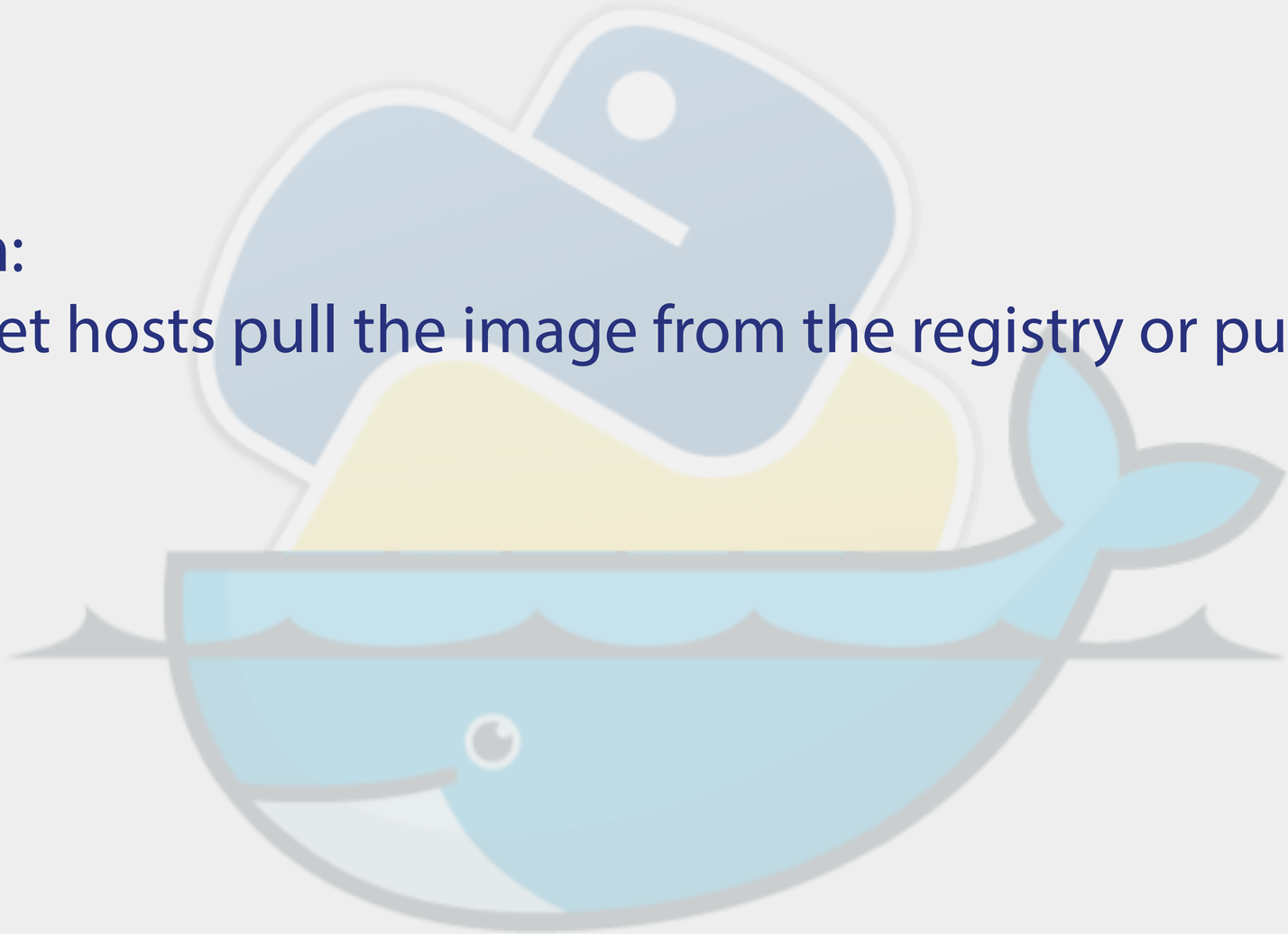


The „container as PM“ approach

On Jenkins:

- build the container image with the application baked in, using the scripts
- run the tests inside the container
- if all tests pass, upload the image to the registry (artifactory, docker registry…)

The „container as PM“ approach



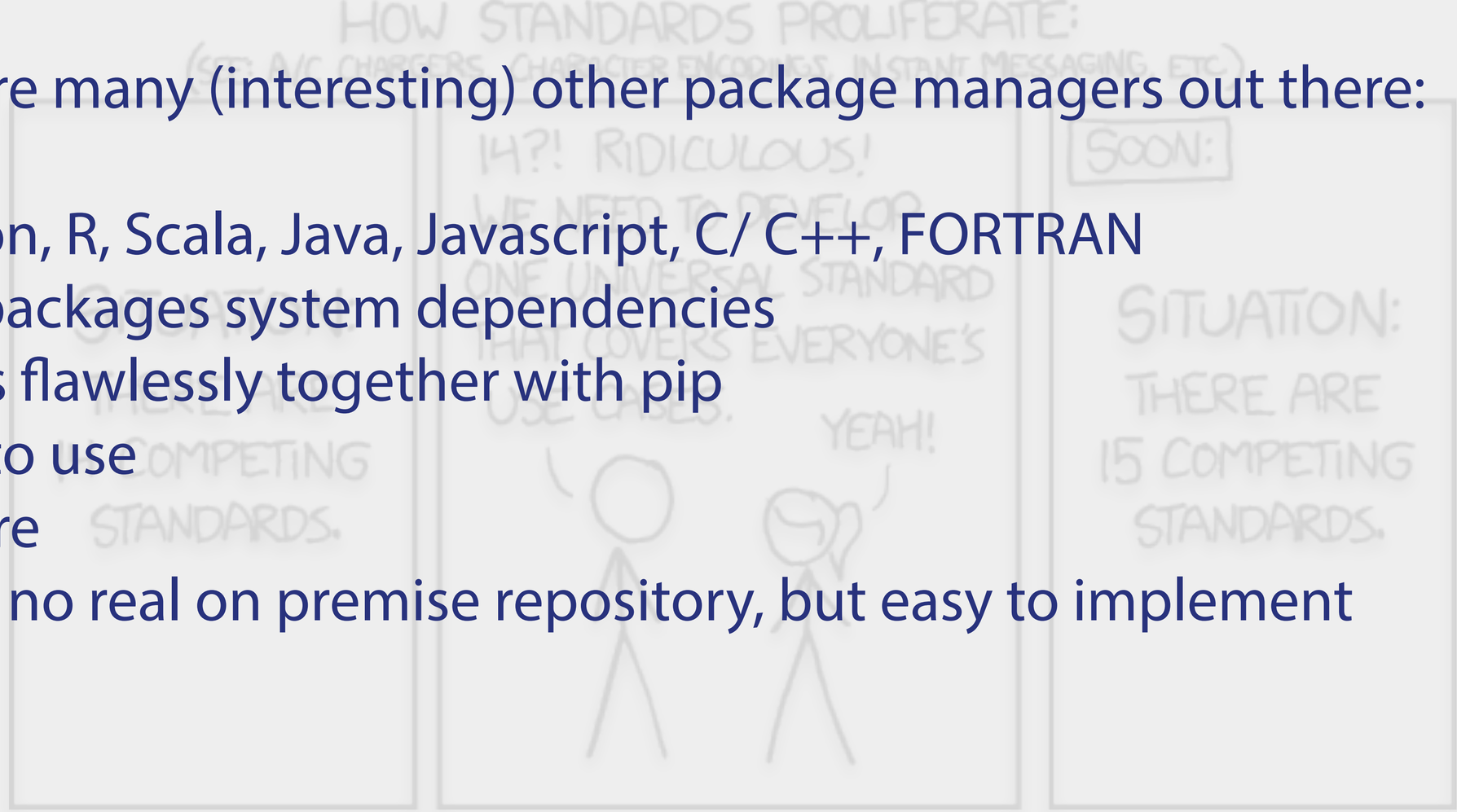
In production:

- let the target hosts pull the image from the registry or push it to the hosts
- start it

The „container as PM“ approach

Pro	Con
• good understood technology, de-facto industry standard: schedulers, repositories, monitoring • complete decoupling from host OS, windows, mac, jenkins or coreos in production, the application runs in the `==` same environment • complete environment+application is built once • everything in git • language independent	• security updates on host irrelevant for application • without proper processes and tooling, easy to do it wrong: unapproved software in production, heartbleed… • dependency resolution and pypi server still needed • chasm between system and language dependencies still exists, but now in a container • doesn’t really reduce complexity

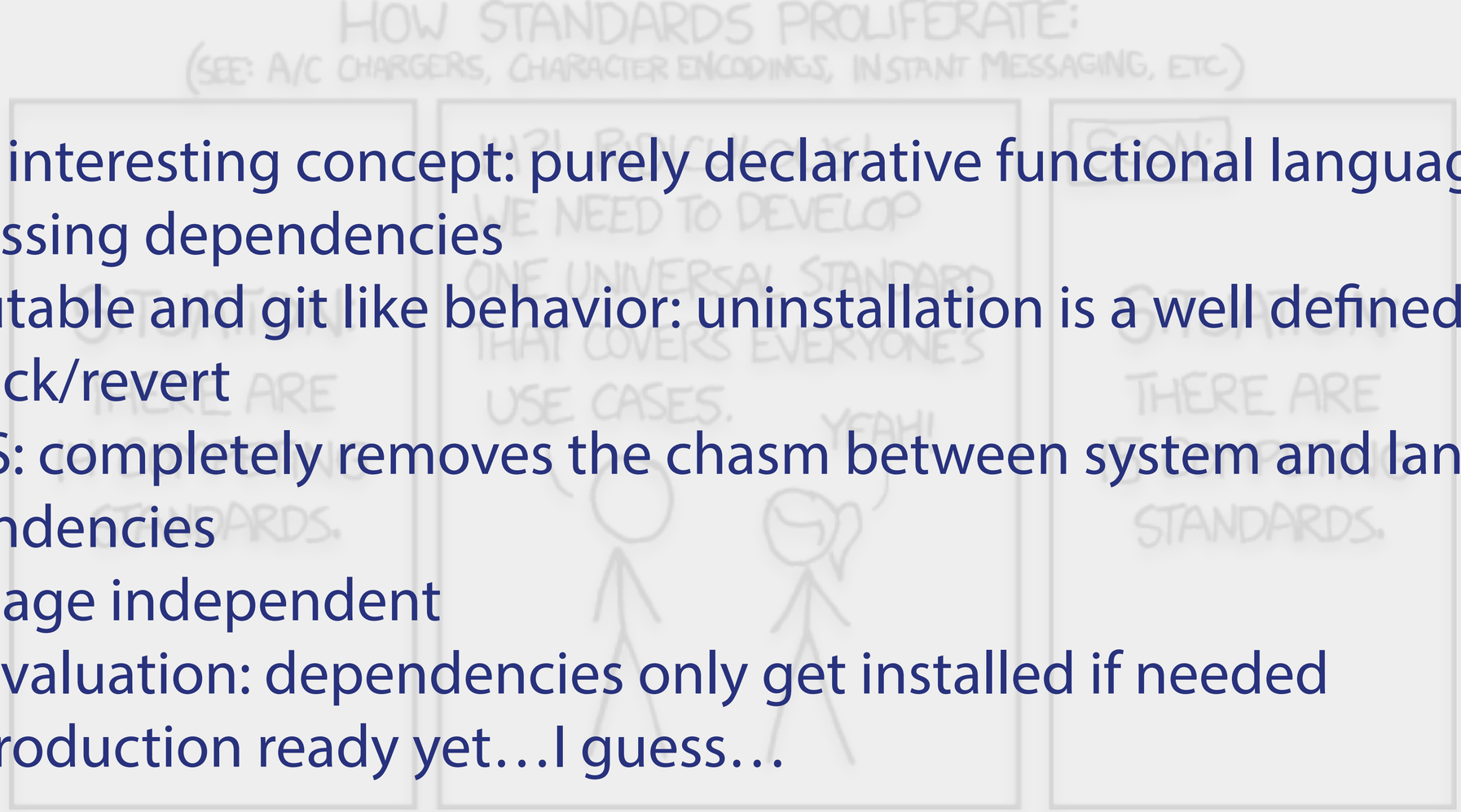
The „next packet manager“ approach



There are many (interesting) other package managers out there:
Conda:

- Python, R, Scala, Java, Javascript, C/ C++, FORTRAN
- also packages system dependencies
- works flawlessly together with pip
- easy to use
- mature
- so far no real on premise repository, but easy to implement

The „next packet manager“ approach



Nix:

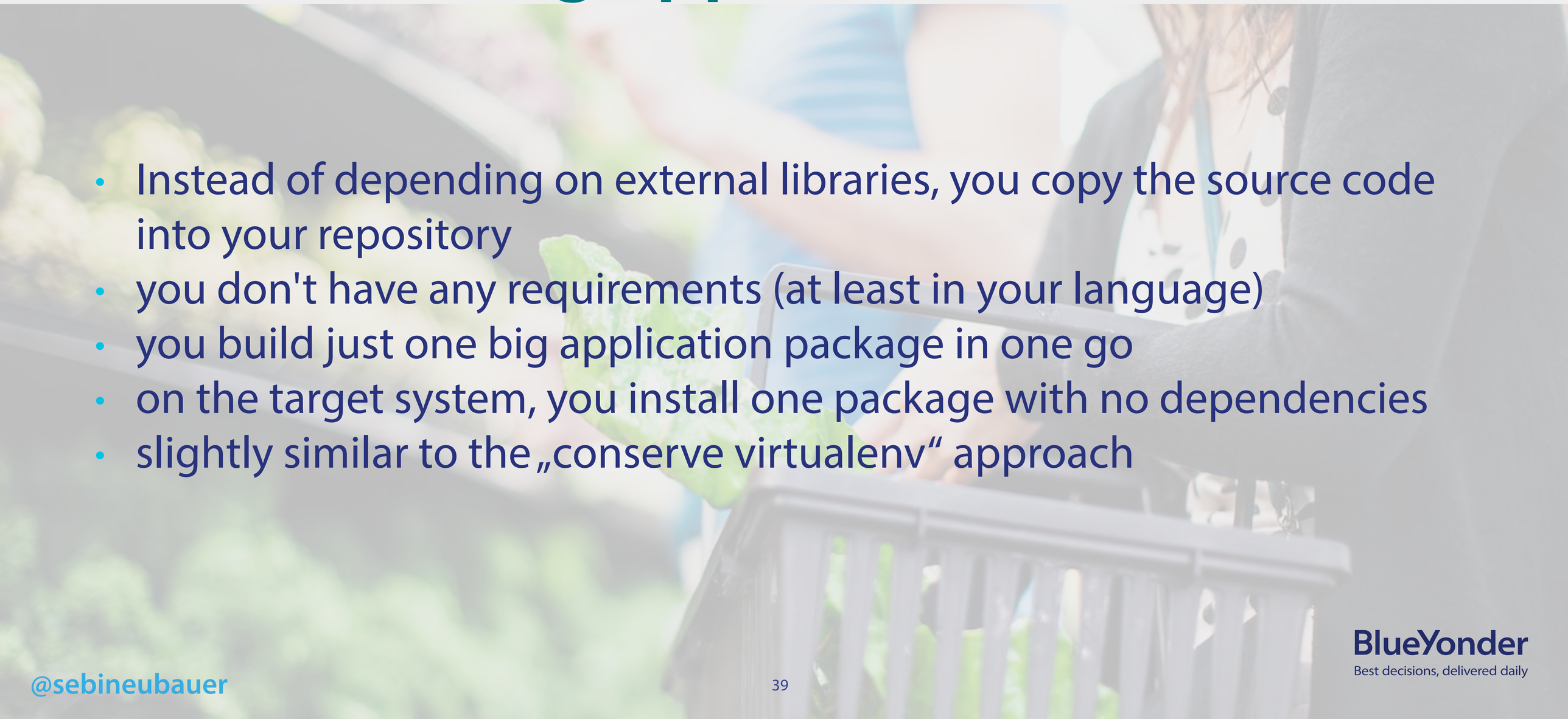
- really interesting concept: purely declarative functional language for expressing dependencies
- immutable and git like behavior: uninstallation is a well defined rollback/revert
- NixOS: completely removes the chasm between system and language dependencies
- language independent
- lazy evaluation: dependencies only get installed if needed
- not production ready yet…I guess…

The „next packet manager“ approach

Pro	Con

• in the end it is a „package manager“ problem, so maybe there is a „next package manager“ that solves most of the problems
• there are package managers that solve parts of the problems even today (e.g. get numeric python packages working on mac and on windows using conda…)
• a good end to end, language agnostic package management solution has a huge potential
	• package management is only part of the problem:
	  • security updates
	  • auditing
	  • same environment for development, testing and production
	• no end to end solution so far
	• very hard to get the critical mass needed that it is a holistic solution for the whole problem for all languages

The vendoring approach



- Instead of depending on external libraries, you copy the source code into your repository
- you don't have any requirements (at least in your language)
- you build just one big application package in one go
- on the target system, you install one package with no dependencies
- slightly similar to the „conserve virtualenv“ approach

The vendoring approach

Pro	Con
• no dependency resolution at all • easy IDE code discovery • no dependecy to external repositories • easy to patch third party libraries	• no dependency resolution at all, everything needs to be done manually • hard work to keep it up to date • easier to patch third party libraries instead of contributing and wait for release • dangerous licensing issues • useless for library development

Can we find a better solution?

How does the future look like?

Containers are here to stay, for many reasons

How does the future look like?

DevOps is the working mode

How does the future look like?

Polyglott: the right language for the job

How does the future look like?

Open source/sharing of code is increasing

How does the future look like?

Automation is a must

How does the future look like?

After that problem is solved, „serverless" becomes a thing

And now a short discussion!

Discussion

OS Package manager

Container

Conserve virtualenv

Classical/wheels

Vendoring

Next package manager

Other??

Q&A

Attributions

By BrokenSphere (Own work) [GFDL (http://www.gnu.org/copyleft/fdl.html) or CC BY-SA 3.0 (http://creativecommons.org/licenses/by-sa/3.0)], via Wikimedia Commons
By Rl (Own work) [GFDL (http://www.gnu.org/copyleft/fdl.html), CC-BY-SA-3.0 (http://creativecommons.org/licenses/by-sa/3.0/) or CC BY-SA 2.5-2.0-1.0 (http://creativecommons.org/licenses/by-sa/2.5-2.0-1.0)], via Wikimedia Commons