

Dependency Management Application vs Package

Application

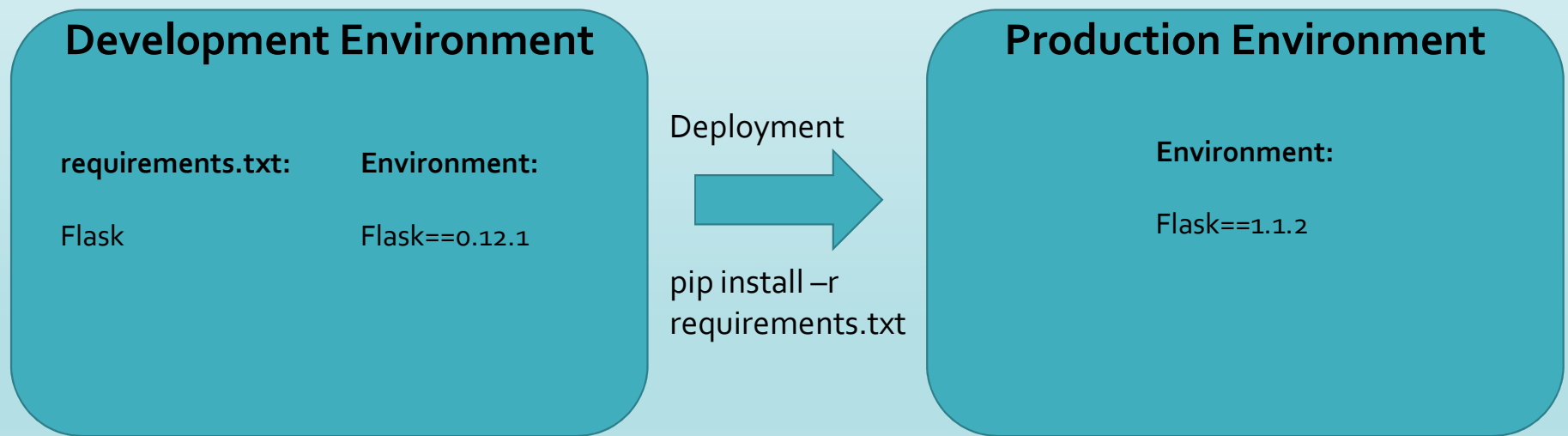
- requirements.txt
- requirements for a complete Python environment
- often exhaustive listing of pinned versions for repeatable installations of complete environments

Package

- setup.py
- minimal requirements for a single project to run correctly
- use of specific versions -> NOT best practice

Application dependency management with requirements.txt

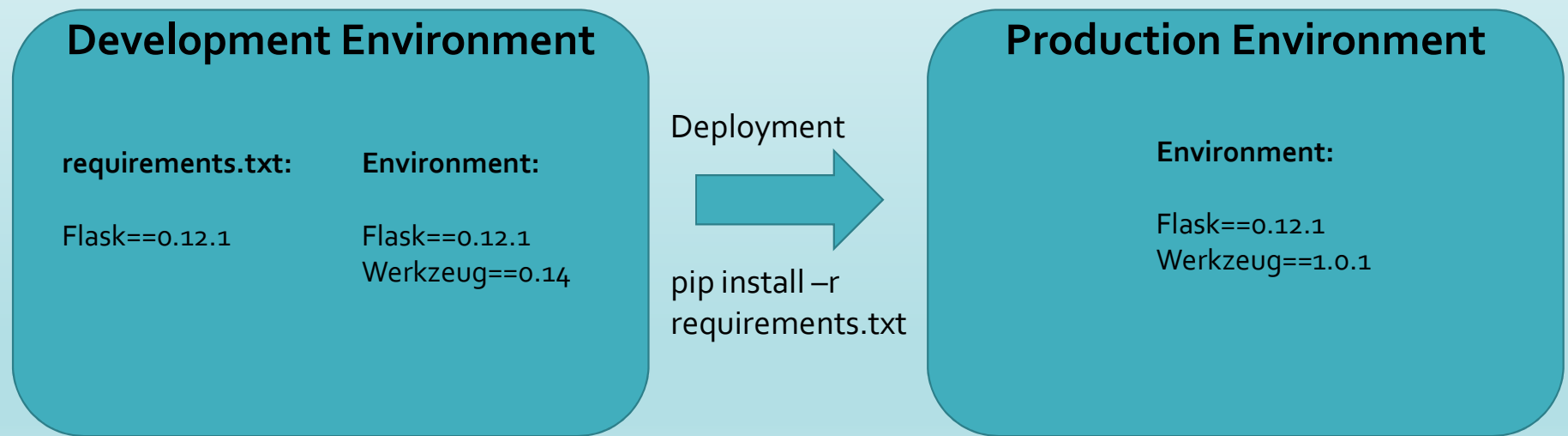
Unspecific packages



→ The build isn't deterministic for required dependencies.

Application dependency management with requirements.txt

Pinned dependencies + sub dependencies



→ The build isn't deterministic for required sub-dependencies.

Resolving dependencies of sub-packages with requirements.txt

requirements.txt:

package_a
package_b

dependencies:

package_a → package_c >= 1.0
package_b → package_c <= 2.0



requirements.txt:

package_c >= 1.0, <= 2.0
package_a
package_b

dependencies:

package_a → package_c >= 1.0
package_b → package_c <= 2.0

→ Responsibility for updating sup-dependencies → Security Risk

Unwanted packages in production

pip freeze

Development Environment

pip freeze ->

requirements.txt:

Environment:

Flask==0.12.1

Flask==0.12.1

Werkzeug==0.14

Werkzeug==0.14

Pytest==6.2.3

Pytest==6.2.3

Pytest not in production ->
requirements.txt, dev-requirements.txt

Deployment



pip install -r
requirements.txt

Production Environment

Environment:

Flask==1.12.1

Werkzeug==0.14

→ Responsibility for updating all (sub) dependencies -> Security Risk

How do you allow for deterministic
builds for your Python project
without gaining the responsibility
of updating version of sub-dependencies?

Pipenv -> Single tool for virtual environment and package management