Component based or monolithic development for large C and C++ projects: Why not both?

Diego Rodriguez-Losada
20-Nov-2025

NW C++ Northwest C++ Users' Group

JFrog | CONAN 2.0

C/C++ Package Manager

JFrog | CONAN
C/C++ Package Manager

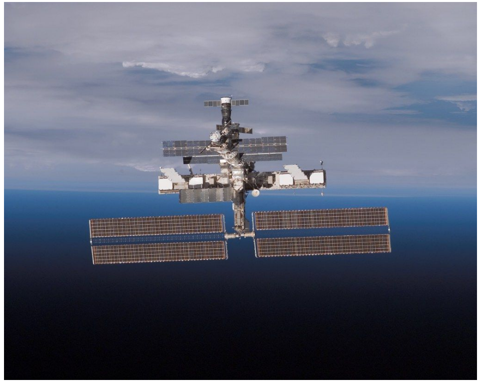
- Free and open source, MIT
- C and C++: static, shared, headers, linkage
- Universal, any OS, any build system
- Binary management with customizable binary model
- Extremely extensible and powerful, enterprise ready
  - Audit, SBOMs…
- Fully maintained by JFrog, 10 people team full time maintainers
- Free JFrog Artifactory CE
- Used in production by thousands of organizations, from startups to ~15% of Fortune500

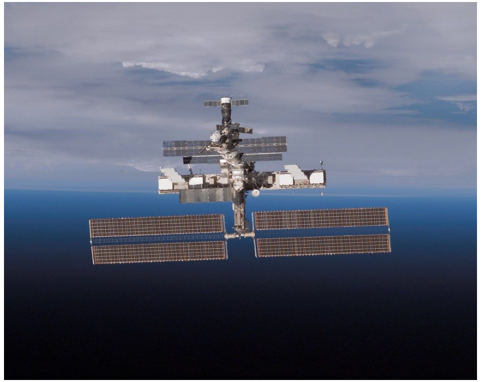
conan.io – github.com/conan-io/conan

Outline

- **Introduction: monorepo vs components**
- Challenges of component based development
- Continuous Integration at scale
- Simultaneous development of multiple packages
- Conclusions
- QA

RIGHT WAY
Right Way
Right Way

	Component based paradigm	Monorepo based paradigm
Seen by component based developers		
Seen by monorepo based developers		

	Component based paradigm	Monorepo based paradigm
Seen by component based developers		
Seen by monorepo based developers		

Conway’s law

> Organizations which design systems (in the broad sense used here) are constrained to produce designs which are copies of the communication structures of these organizations.
>
> —Melvin E. Conway, How Do Committees Invent?

The structure of any system designed by an organization is isomorphic to the structure of the organization

You can see the organization chart of a car company in the dashboard, and also see whether the steering wheel team hates the gear stick team.

Development paradigms

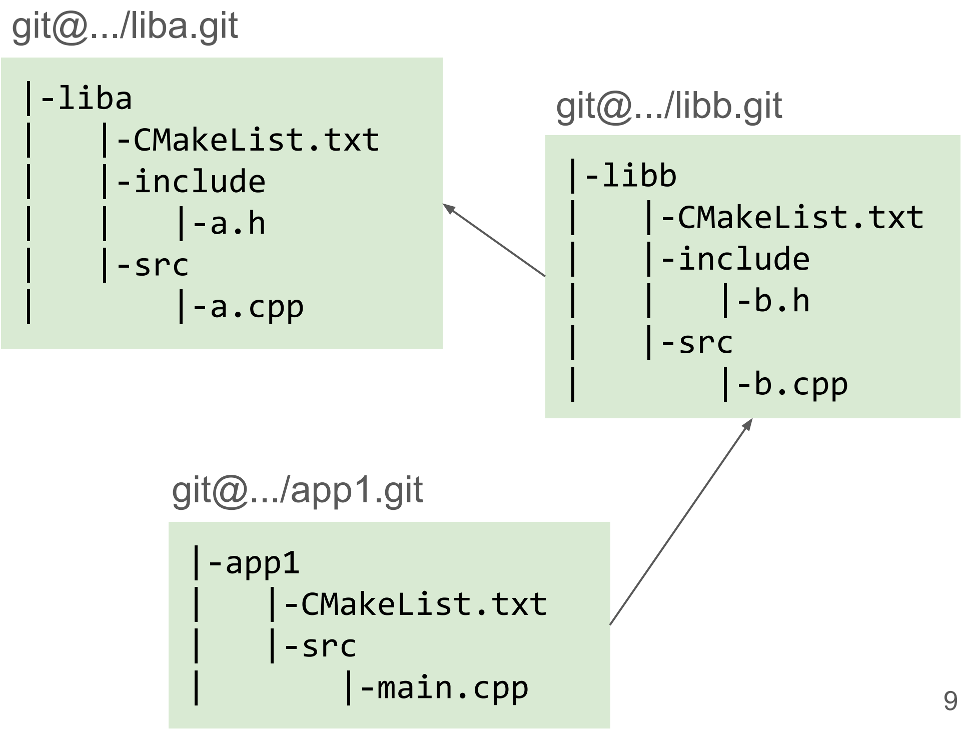
Mono-repo / monolithic build

git@.../monorepo.git

Could be a submodule

```
|-WORKSPACE
|-liba
|    |-BUILD
|    |-include
|    |    |-a.h
|    |-src
|         |-a.cpp
|-libb
|    |-BUILD
|    |-include
|    |    |-b.h
|    |-src
|         |-b.cpp
|-app1
|    |-BUILD
|    |-src
|         |-main.cpp
```

Multi-repo / component build



git@.../liba.git

```
|-liba
|    |-CMakeList.txt
|    |-include
|    |    |-a.h
|    |-src
|         |-a.cpp
```

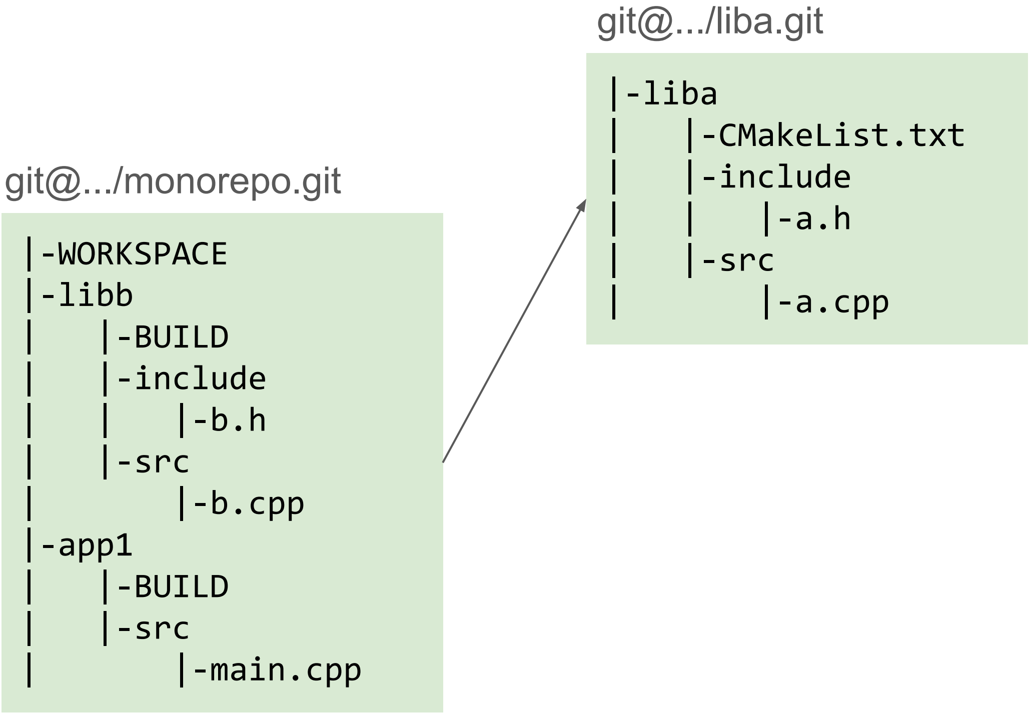
git@.../libb.git

```
|-libb
|    |-CMakeList.txt
|    |-include
|    |    |-b.h
|    |-src
|         |-b.cpp
```

git@.../app1.git

```
|-app1
|    |-CMakeList.txt
|    |-src
|         |-main.cpp
```

Development paradigms: hybrid



git@.../liba.git

```
|-liba
|    |-CMakeList.txt
|    |-include
|    |    |-a.h
|    |-src
|         |-a.cpp
```

git@.../monorepo.git

```
|-WORKSPACE
|-libb
|    |-BUILD
|    |-include
|    |    |-b.h
|    |-src
|         |-b.cpp
|-app1
|    |-BUILD
|    |-src
|         |-main.cpp
```

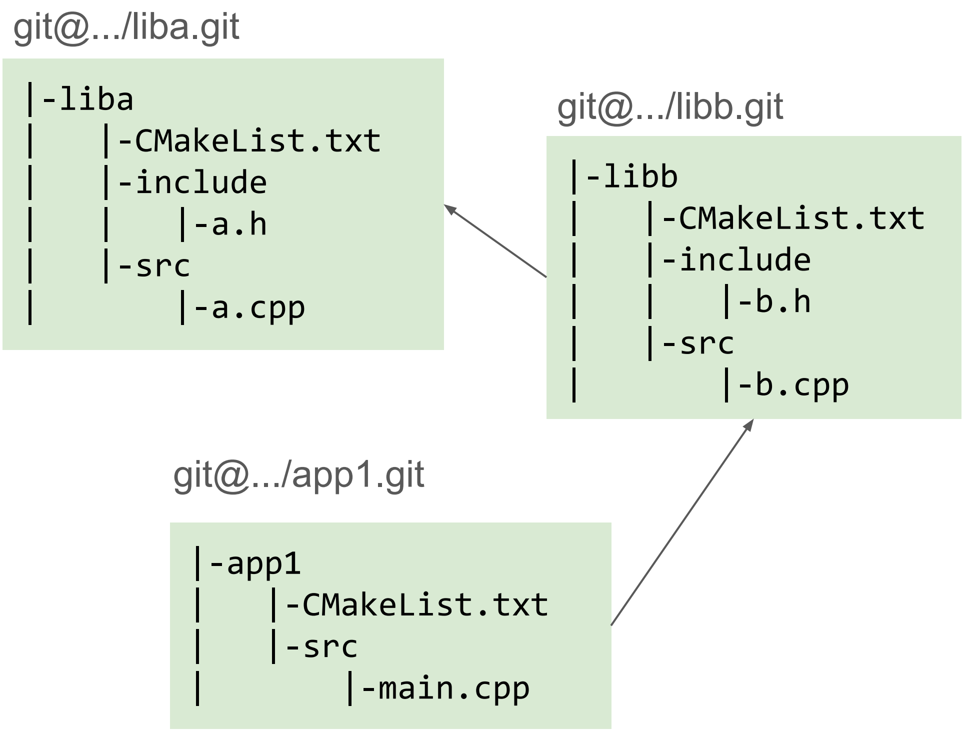
Mono repo

```
|-WORKSPACE
|-liba
|   |-BUILD
|   |-include
|   |   |-a.h
|   |-src
|       |-a.cpp
|-libb
|   |-BUILD
|   |-include
|   |   |-b.h
|   |-src
|       |-b.cpp
|-app1
|   |-BUILD
|   |-src
|       |-main.cpp
```

- Live at Head paradigm
    - (Titus Winters, Google)
- Tooling:
    - Bazel(blaze), Buck2, Visual
    - Heavy use of compilation caching
    - Very dedicated and optimized build infra
    - Tooling for git itself
- Pros:
    - No versioning
- Cons:
    - No versioning
    - Organizational challenges
    - Infra
    - Tools can be complex

Multi-repo/components

- Classic versioning paradigm
- Tooling:
  - CMake, Makefiles, MSBuild, Meson
  - Caching at the binary level (package management)
- Pros:
  - Per component development, versioning and releasing
- Cons:
  - Per component development, versioning and releasing



git@.../liba.git

```
|-liba
|    |-CMakeList.txt
|    |-include
|    |    |-a.h
|    |-src
|         |-a.cpp
```

git@.../libb.git

```
|-libb
|    |-CMakeList.txt
|    |-include
|    |    |-b.h
|    |-src
|         |-b.cpp
```

git@.../app1.git

```
|-app1
|    |-CMakeList.txt
|    |-src
|         |-main.cpp
```

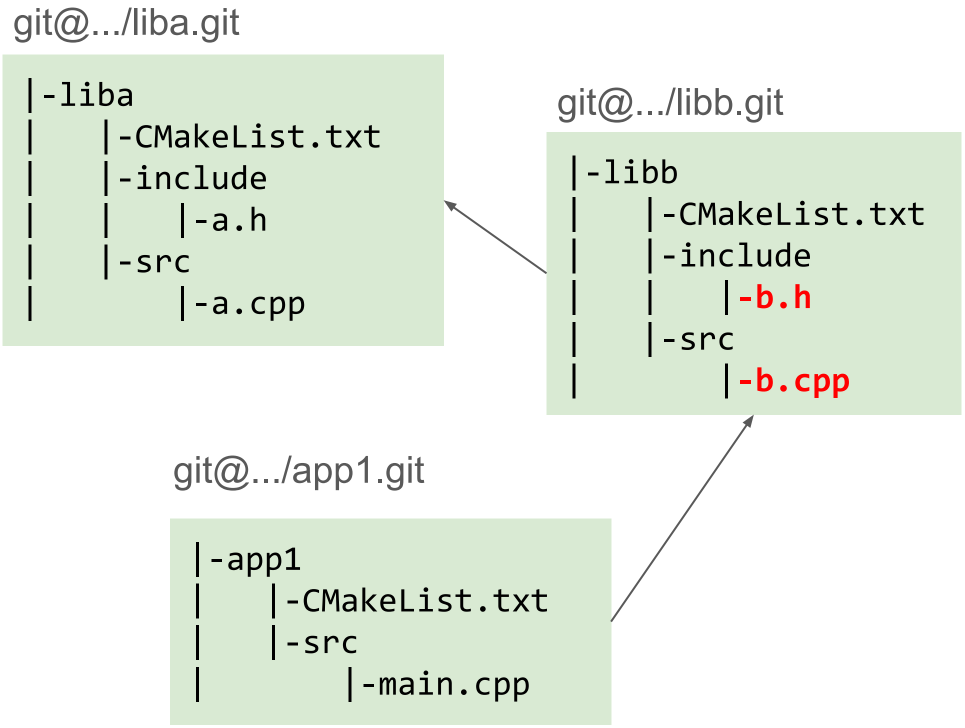
WHY NOT BOTH?

Outline

- Introduction: monorepo vs components
- **Challenges of component based development**
- Continuous Integration at scale
- Simultaneous development of multiple packages
- Conclusions
- QA

Components evolve

- One package gets changes
- Build those changes
- Down to our applications (integrate)
- Efficient and safe way



git@.../liba.git

```
|-liba
|    |-CMakeList.txt
|    |-include
|    |    |-a.h
|    |-src
|         |-a.cpp
```

git@.../libb.git

```
|-libb
|    |-CMakeList.txt
|    |-include
|    |    |-b.h
|    |-src
|         |-b.cpp
```

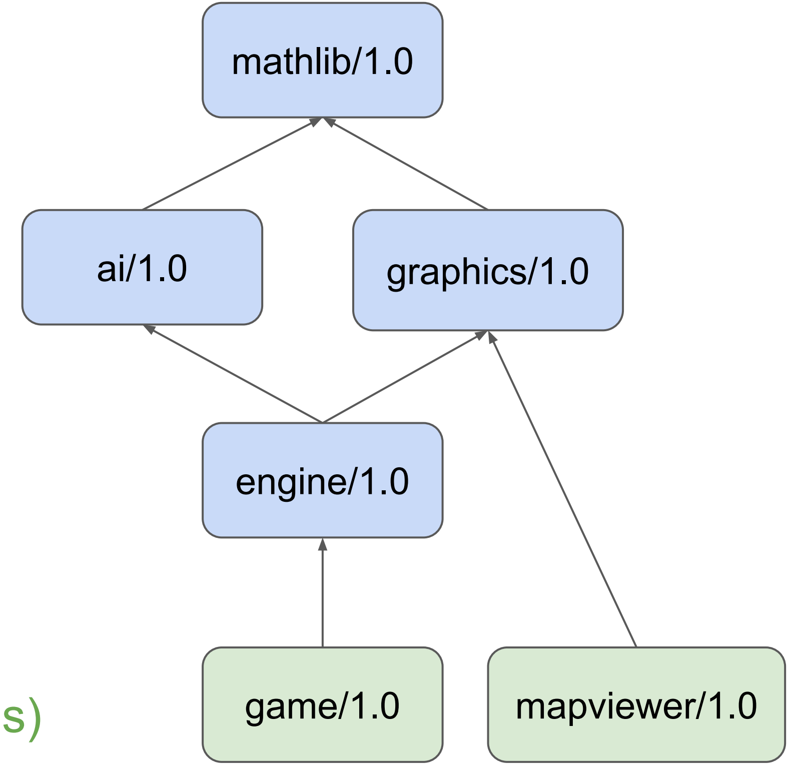
git@.../app1.git

```
|-app1
|    |-CMakeList.txt
|    |-src
|         |-main.cpp
```

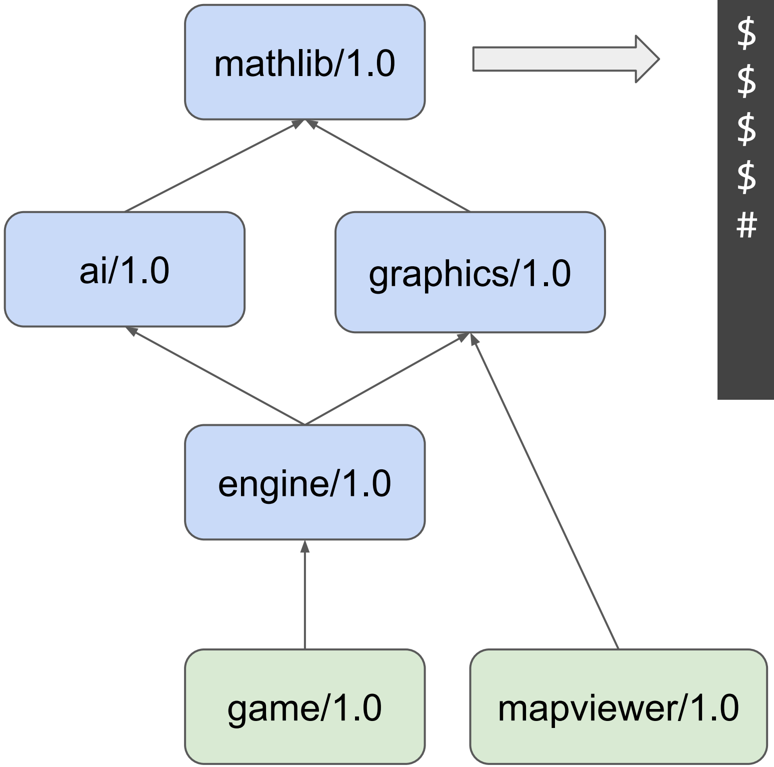
Example project

Libraries (static)

Applications (exes)

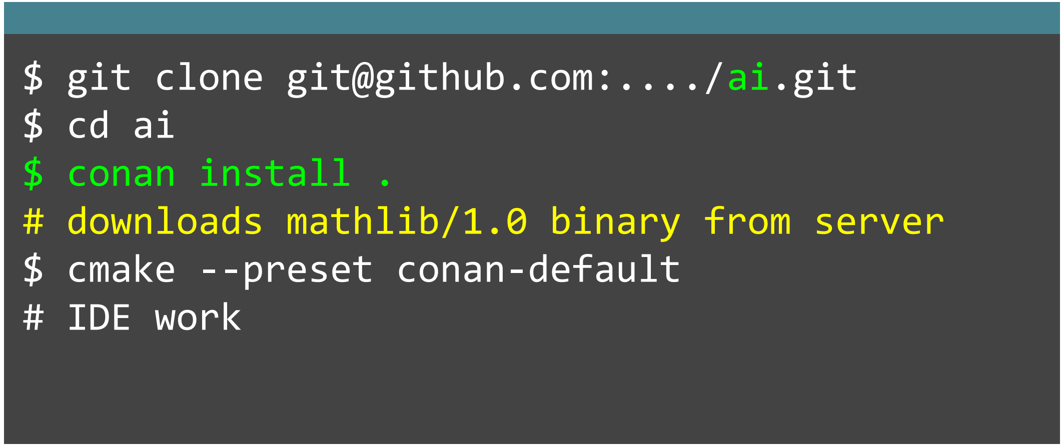


Example project: multi-repository

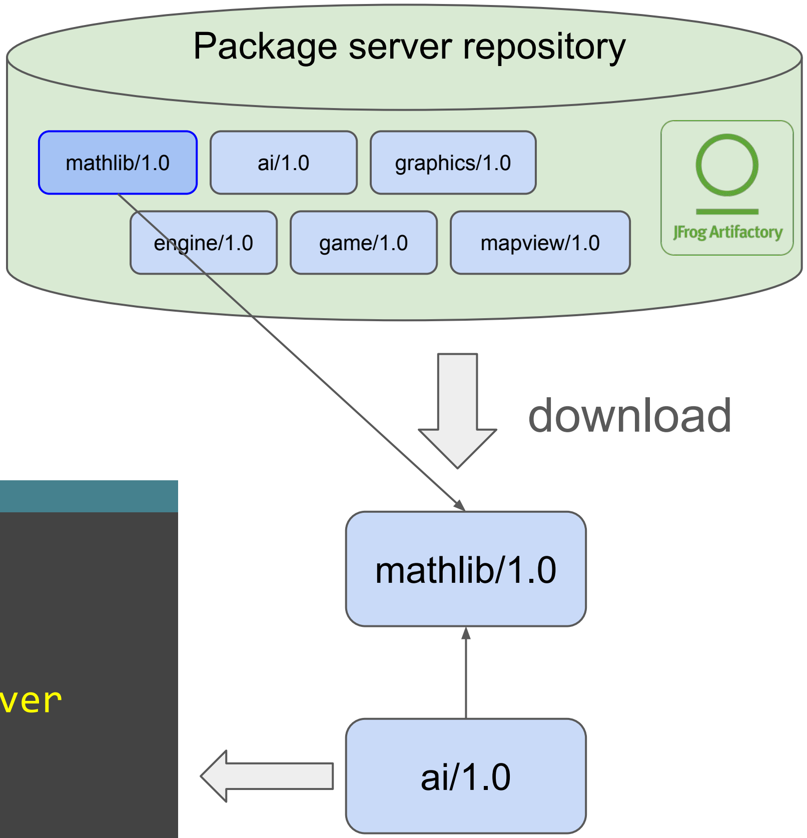


```
$ git clone git@github.com:..../mathlib.git
$ cd mathlib
$ conan install .
$ cmake --preset conan-default
# IDE work
```

Assumptions:
package and dependency management



```
$ git clone git@github.com:..../ai.git
$ cd ai
$ conan install .
# downloads mathlib/1.0 binary from server
$ cmake --preset conan-default
# IDE work
```



Package management 101

```
$ conan install
```

- Install dependencies of current project

```
$ conan build
```

- = conan install + build()
- Install dependencies of current project
- Executes “cmake” configure and “cmake” build steps

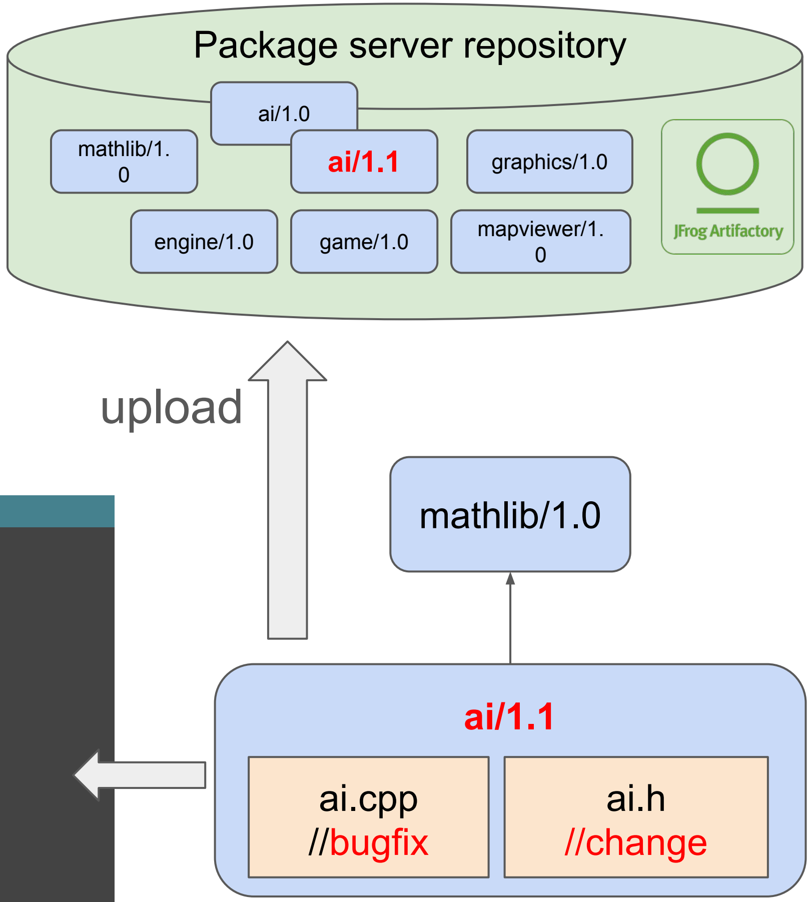
```
$ conan create
```

- Install dependencies of current project
- Builds from source:
  - cmake .
  - cmake --build
- Packages:
  - cmake --install

conanfile.py

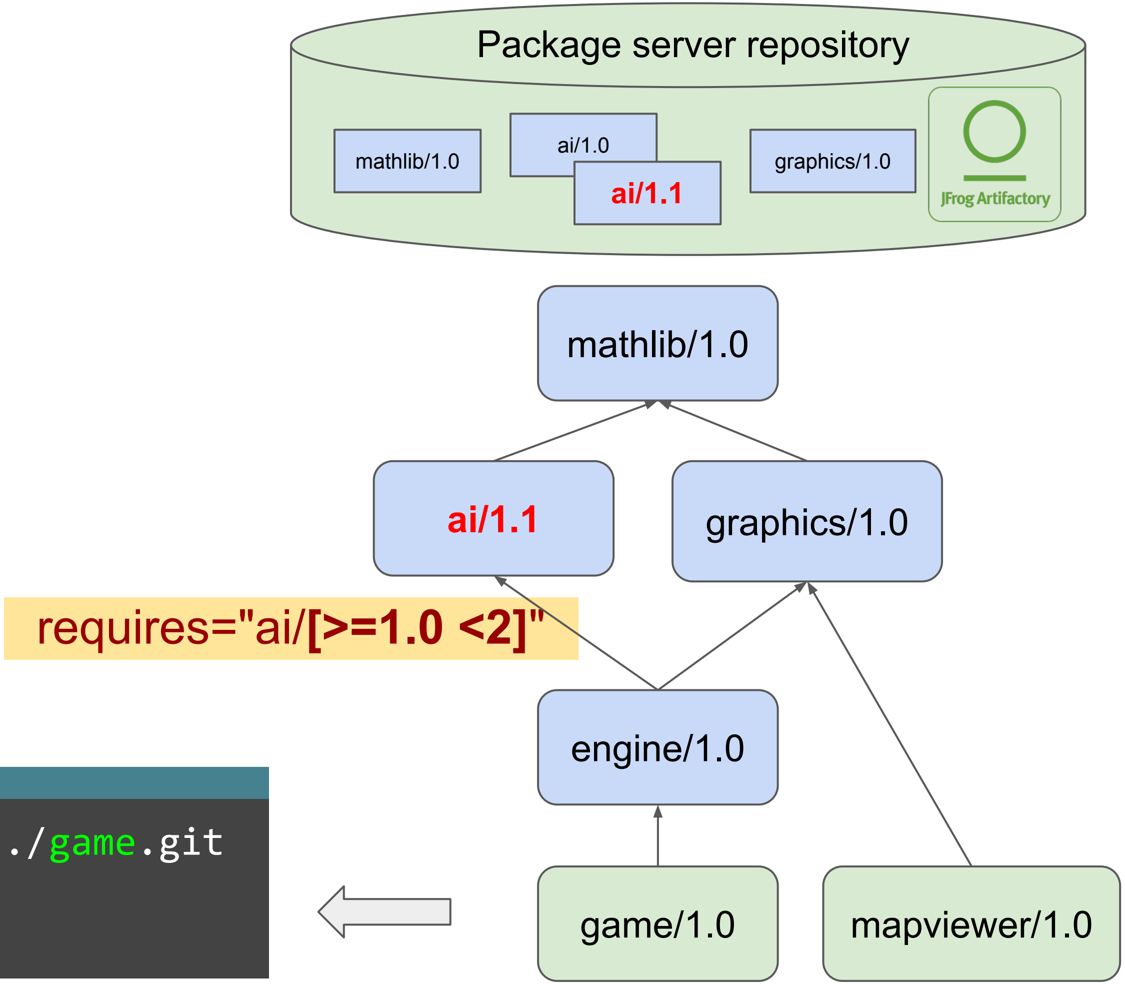
```python
class aiRecipe(ConanFile):
    name = "ai"
    version = "1.0"
    requires = "mathlib/[>=1.0 <2]"

    # Binary configuration
    settings = "os", "compiler", "build_type", "arch"
    package_type = "static-library"

    def export(self):
        git = Git(self, self.recipe_folder)
        git.coordinates_to_conandata()

    def generate(self):
        tc = CMakeToolchain(self)
        tc.preprocessor_definitions["PKG_VERSION"] = f'"{self.version}"'
        tc.generate()
        deps = CMakeDeps(self)
        deps.generate()

    def build(self):
        cmake = CMake(self)
        cmake.configure()
        cmake.build()

    def package(self):
        cmake = CMake(self)
        cmake.install()

    def package_info(self):
        self.cpp_info.libs = ["ai"]
```

Challenge: new version!



```
$ git clone git@github.com:..../ai.git
$ cd ai
$ conan install .
$ cmake --preset conan-default
# IDE work, bump version 1.0=>1.1
$ conan create .
$ conan upload "ai/*" -r=myremote -c
```

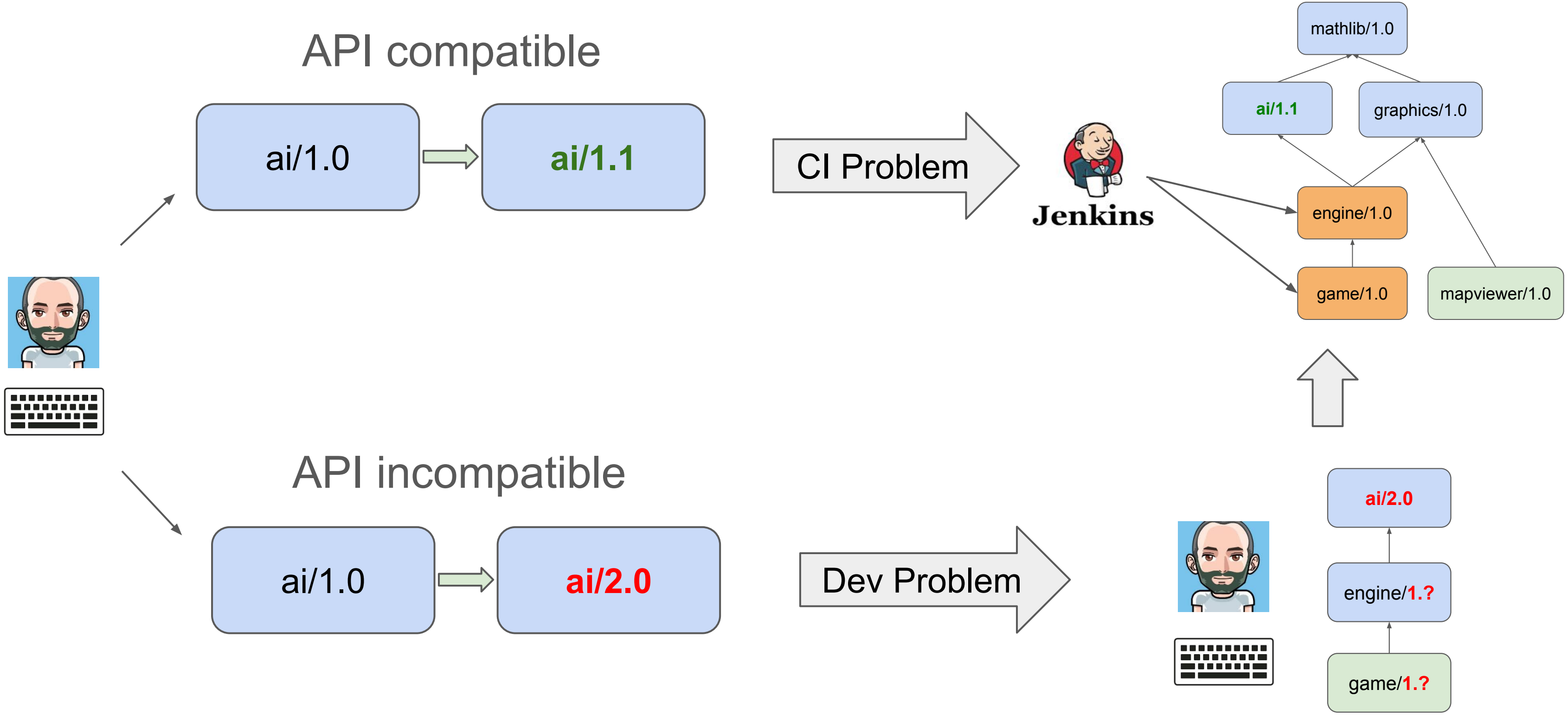
Problem statement: version-ranges



requires="ai/**[>=1.0 <2]**"

```
$ git clone git@github.com:..../game.git
$ cd game
$ conan install .
```

Two different scenarios



API compatible

ai/1.0 → ai/1.1

CI Problem

Jenkins

mathlib/1.0

ai/1.1

graphics/1.0

engine/1.0

game/1.0

mapviewer/1.0

API incompatible

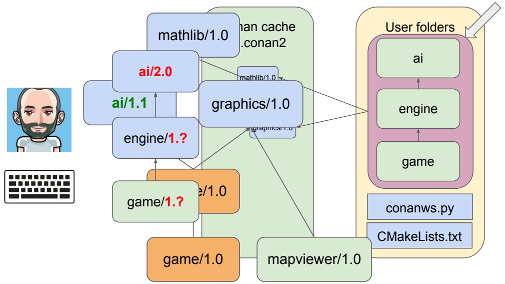
ai/1.0 → ai/2.0

Dev Problem

ai/2.0

engine/1.?

game/1.?

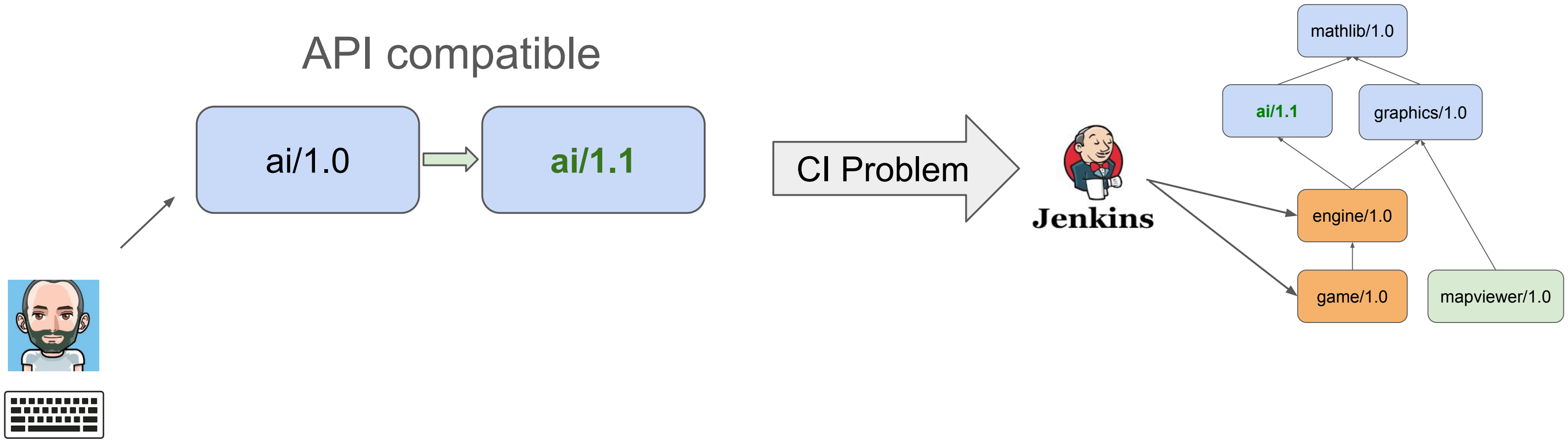


```python
class Ws(Workspace):
    def root_conanfile(self):
        return MyWs


class MyWs(ConanFile):
    settings = "os", "compiler", "build_type", "arch"

    def generate(self):
        deps = CMakeDeps(self)
        deps.generate()
        tc = CMakeToolchain(self)
        tc.preprocessor_definitions["PKG_VERSION"] = '"WS_0.1"'
        tc.generate()
```

Outline

- Introduction: monorepo vs components
- Challenges of component based development
- **Continuous Integration at scale**
- Simultaneous development of multiple packages
- Conclusions
- QA

The CI problem



API compatible

ai/1.0

ai/1.1

CI Problem

Jenkins

mathlib/1.0

ai/1.1

graphics/1.0

engine/1.0

game/1.0

mapviewer/1.0

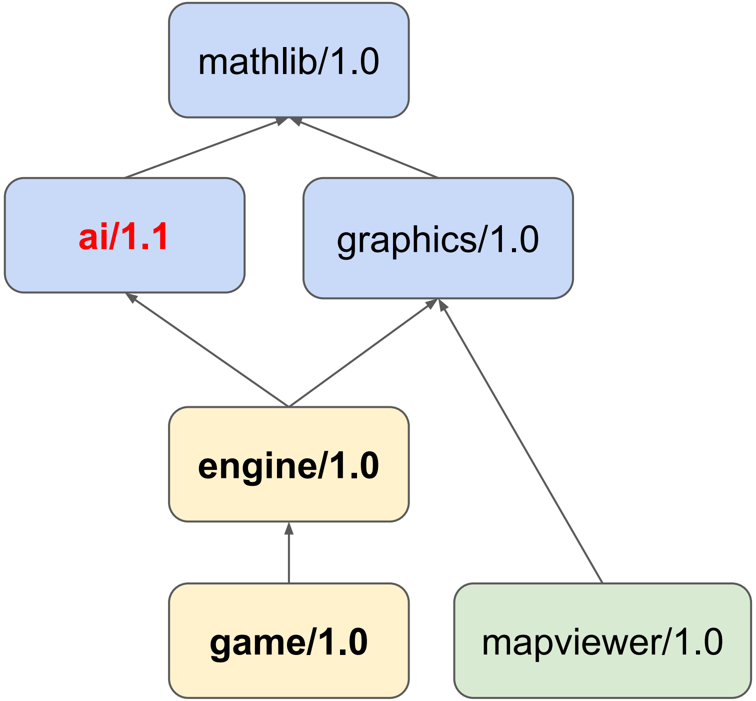
CI Problem statement

Given an API compatible new version of a package:

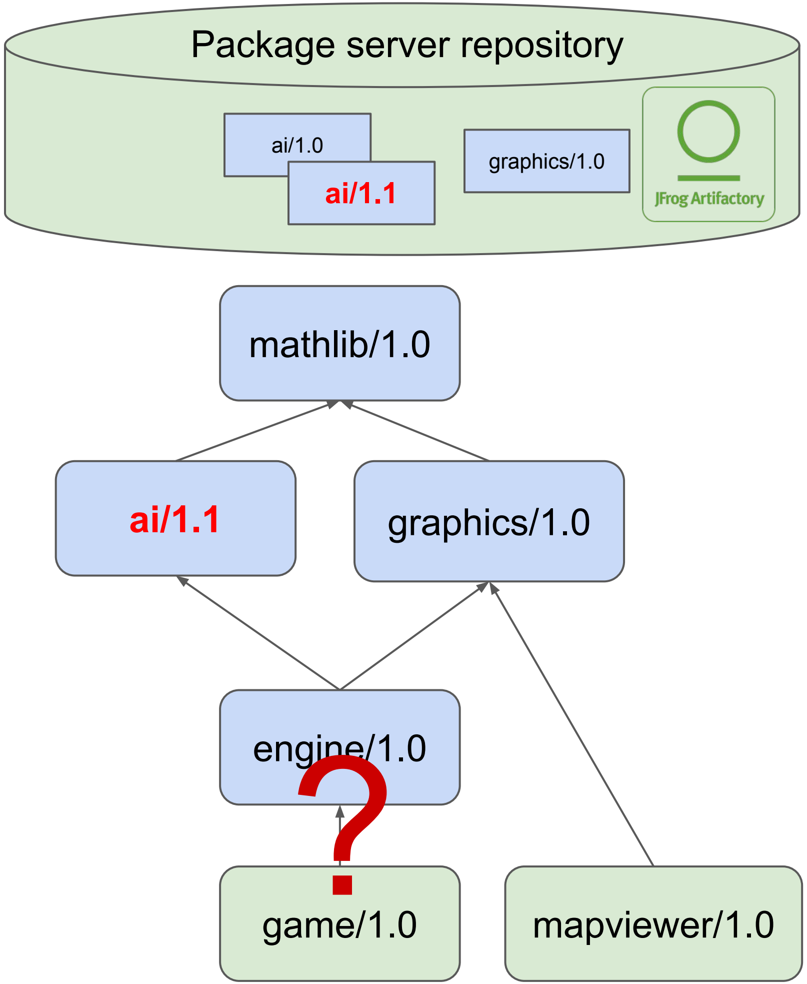
- Build and test the necessary packages for the supported platforms, in the right order down to my organization “products”

Conditions:

- Efficiently: do not build more than necessary
- Fast: build in parallel whenever possible
- Safely: do not break the build or disrupt other development and release processes

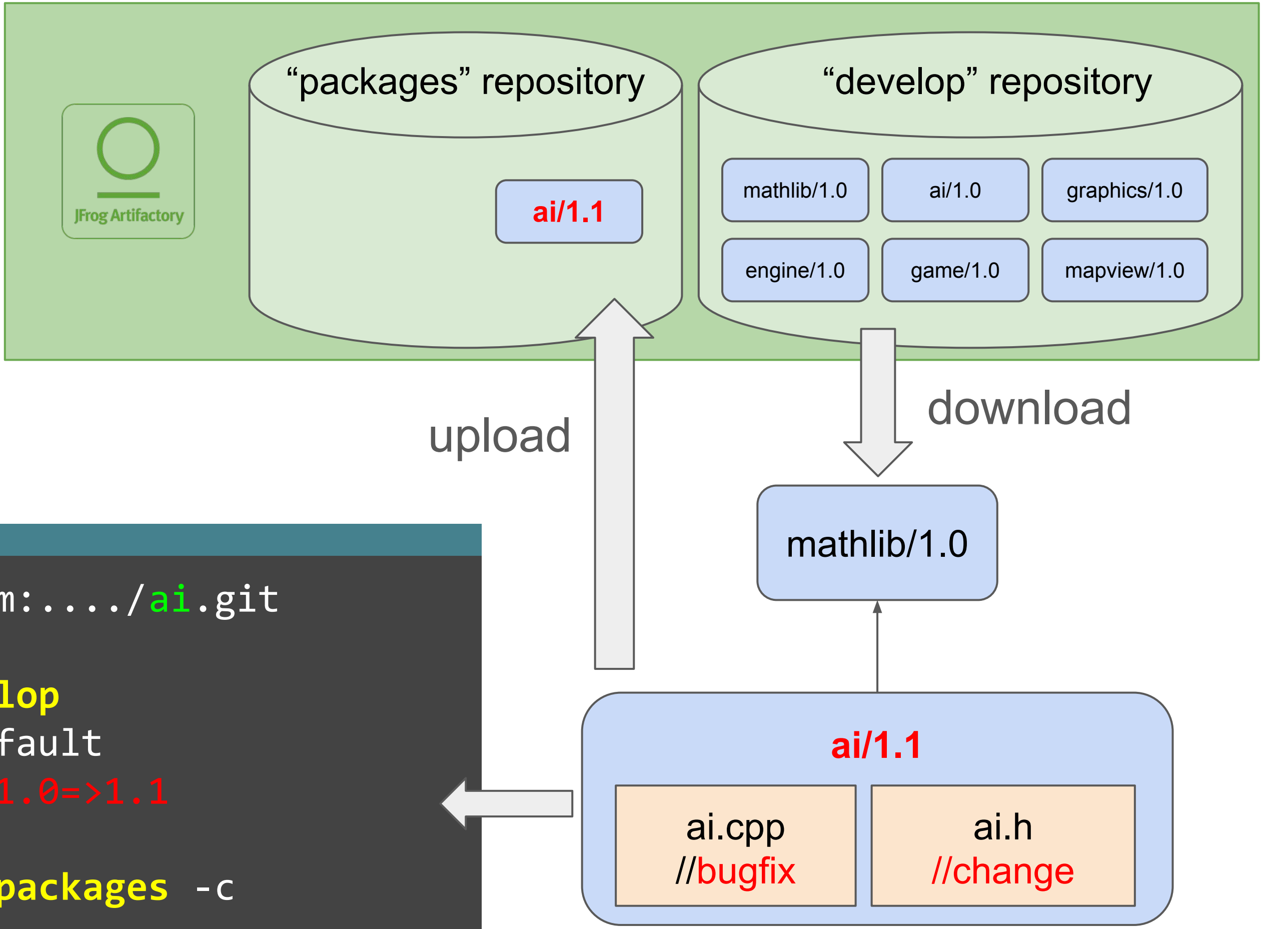


Principles: “don’t break the build”



Principles: “multi-repository”

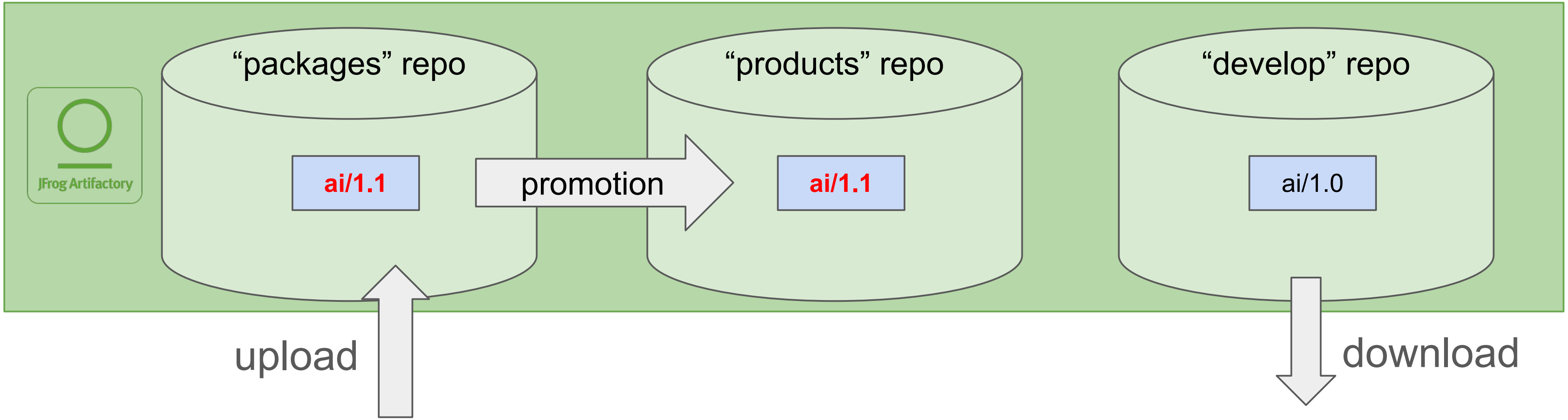
⇔ multi-branch in source



```
$ git clone git@github.com:..../ai.git
$ cd ai
$ conan install . -r=develop
$ cmake --preset conan-default
# IDE work, bump version 1.0=>1.1
$ conan create .
$ conan upload "ai/*" -r=packages -c
```

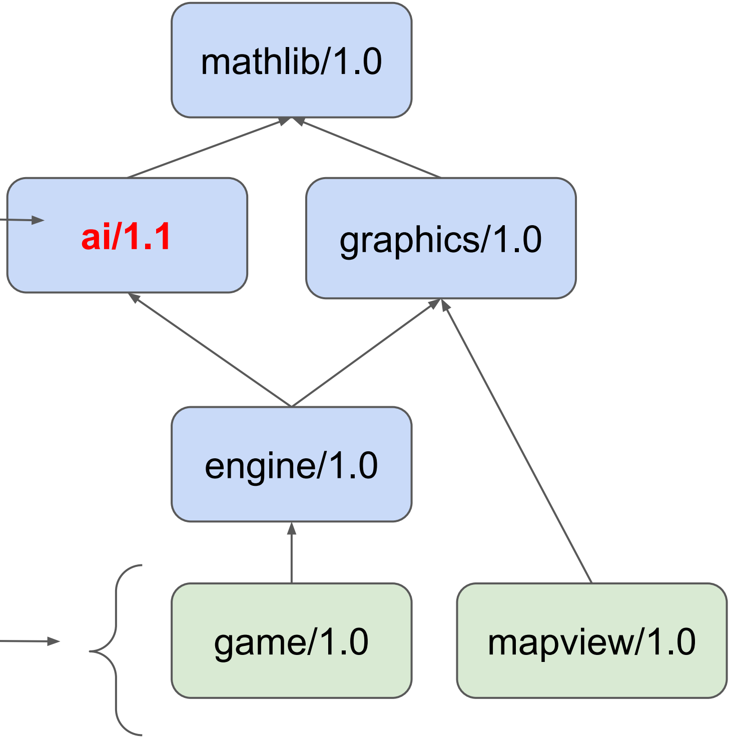
Principles: “package promotions”

⇔ merge in source

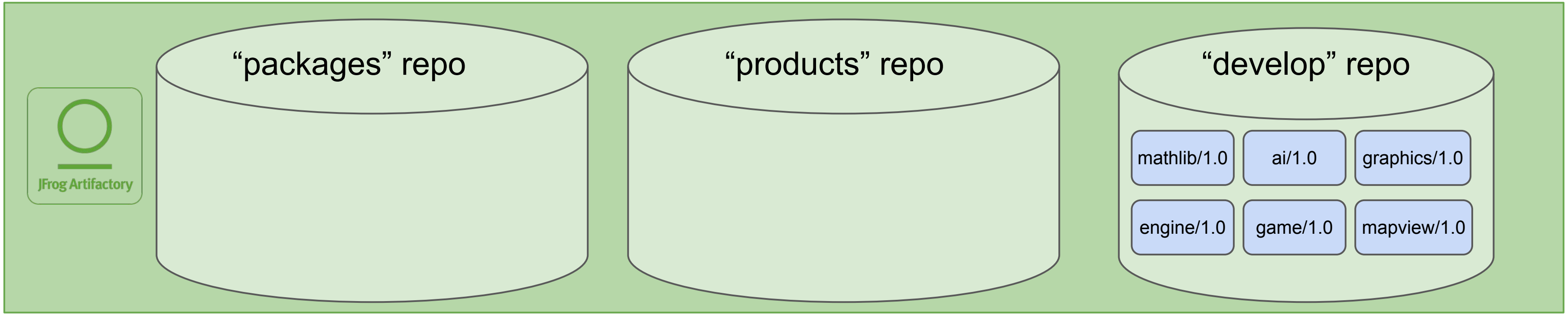


Principles: “packages” and “products” CI pipelines

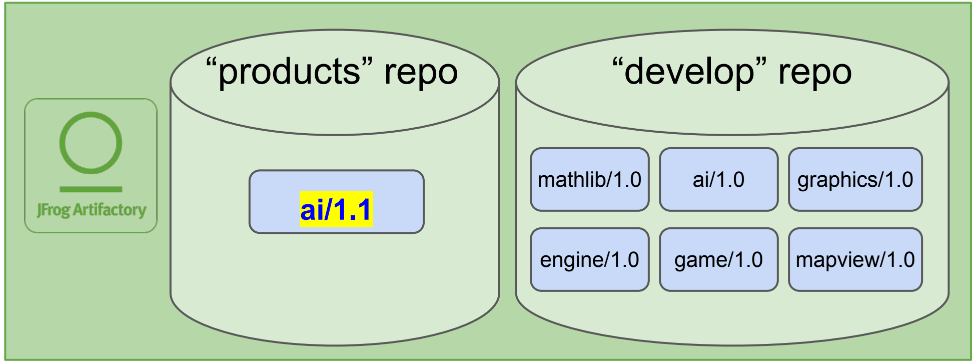
- “**Packages**”: Classic, build **ai/1.1**
- “**Products**”: Build **game/1.0** and **mapview/1.0** (and all other necessary intermediate packages) against new **ai/1.1**

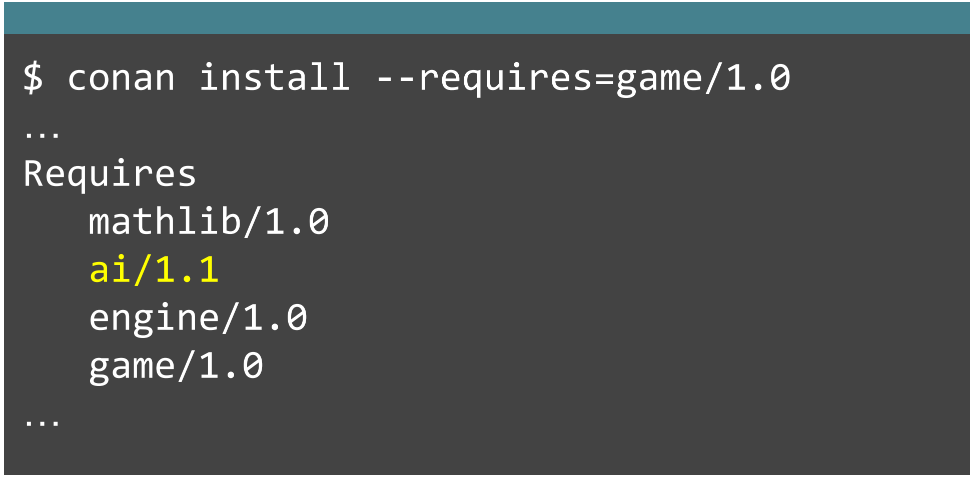


Project setup

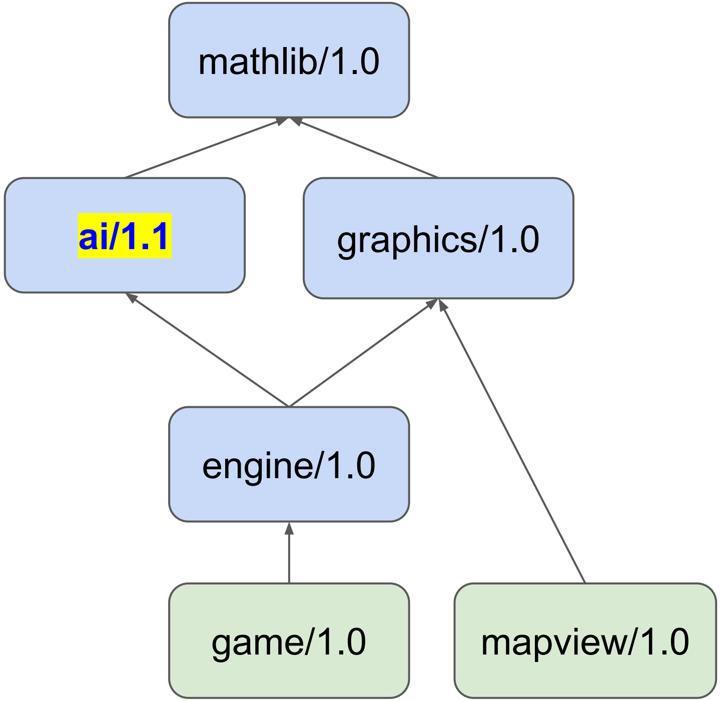


Product pipeline: game/1.0

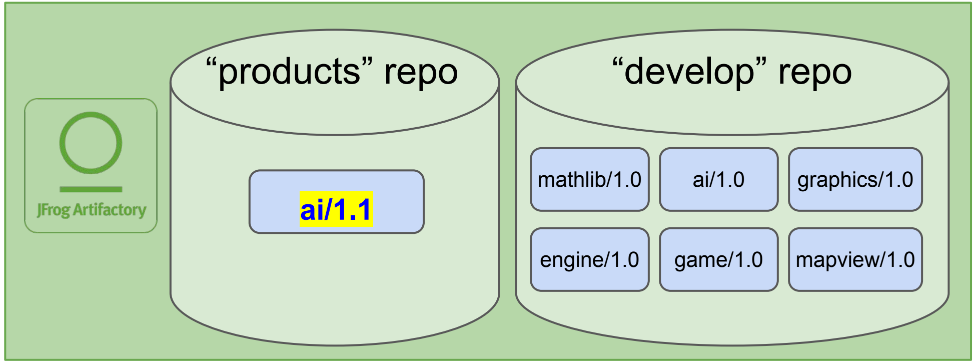




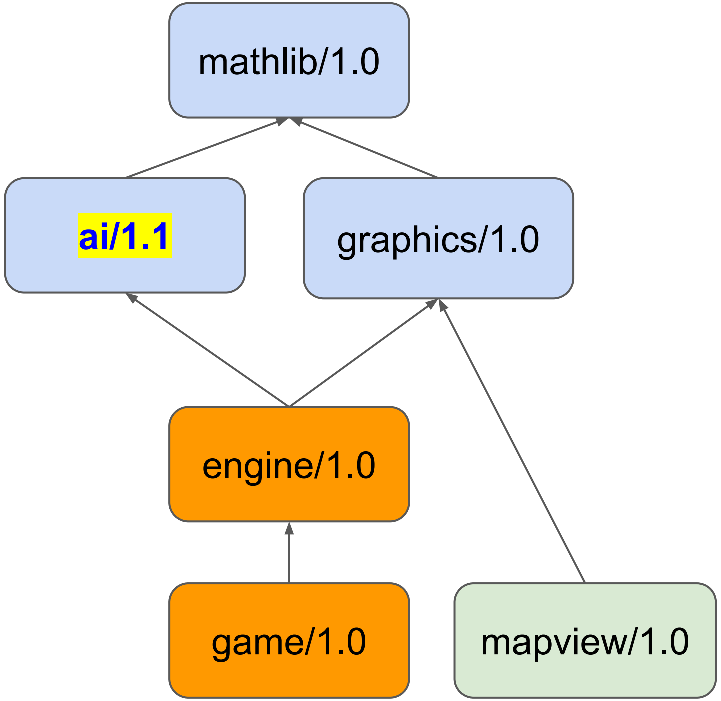
```
$ conan install --requires=game/1.0
…
Requires
   mathlib/1.0
   ai/1.1
   engine/1.0
   game/1.0
…
```



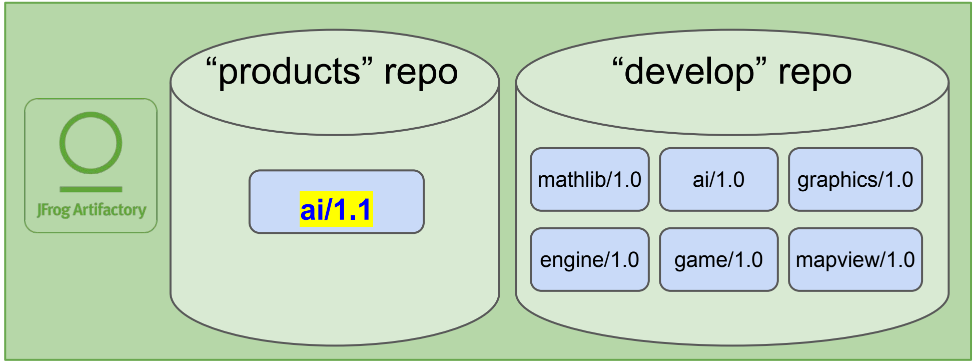
Product pipeline: game/1.0

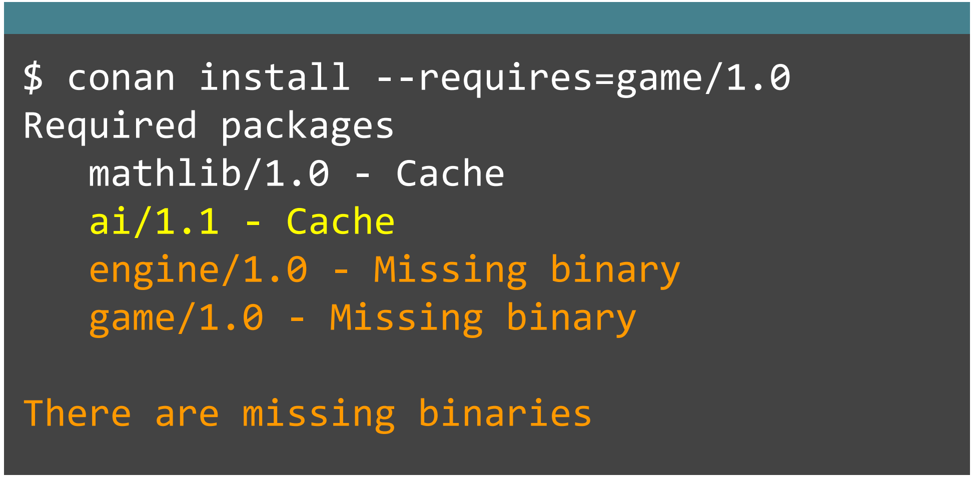


```
$ conan install --requires=game/1.0
Required packages
   mathlib/1.0 - Cache
   ai/1.1 - Cache
   engine/1.0 - Missing binary
   game/1.0 - Missing binary

There are missing binaries
```

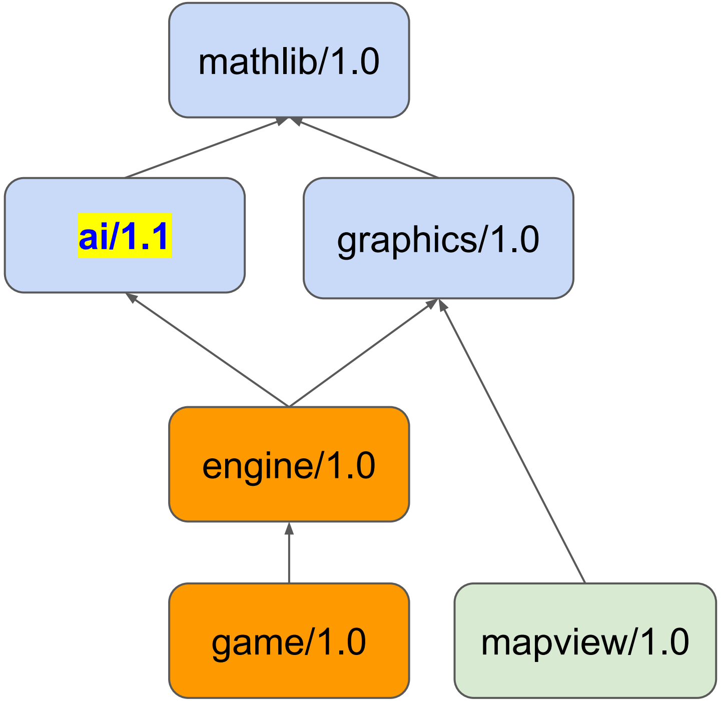


Product pipeline: game/1.0

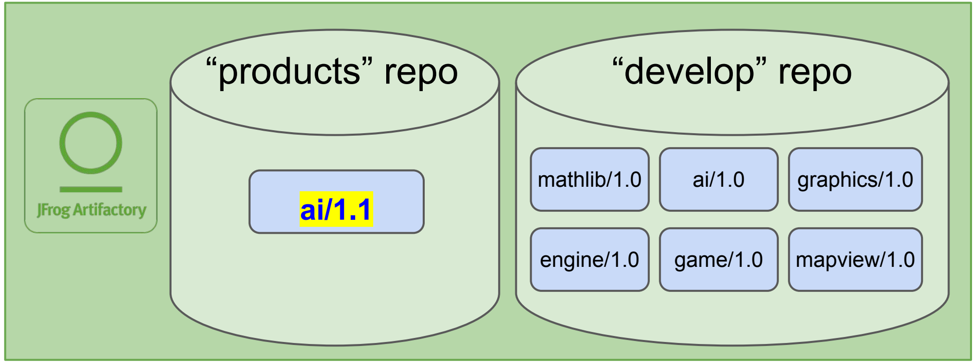




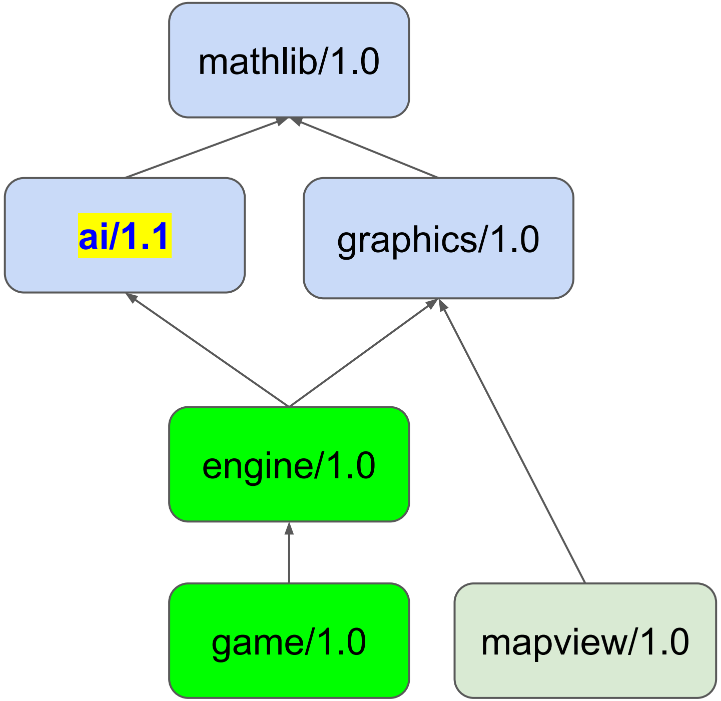
```
$ conan install --requires=game/1.0
Required packages
   mathlib/1.0 - Cache
   ai/1.1 - Cache
   engine/1.0 - Missing binary
   game/1.0 - Missing binary

There are missing binaries
```



Welcome “conan graph build-order”

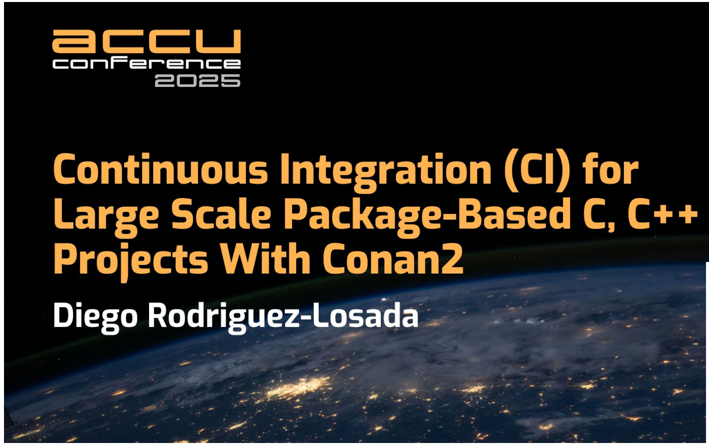


```
$ conan graph build-order
--requires=game/1.0 --build=missing >
game_build_order.json
```



graph_build_order.json

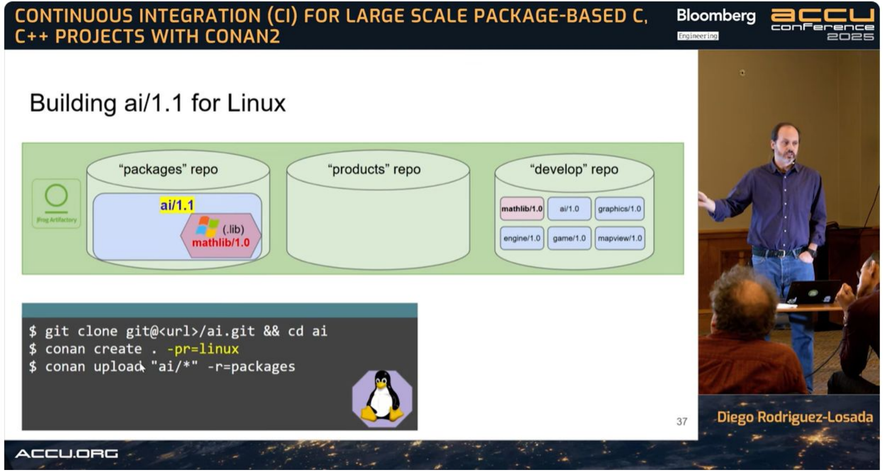
```
[
  {
    "ref": "engine/1.0",
    "packages": [[{
       "package_id": "de73..a765",
       "binary": "Build",
       "build_args": "--requires=engine/1.0 --build=engine/1.0",
    }]]
  }
],
[
  {
    "ref": "game/1.0",
    "depends": ["engine/1.0"],
    "packages": [[{
       "package_id": "bac7..9d4c",
       "binary": "Build",
       "build_args": "--requires=game/1.0 --build=game/1.0",
     }]]
  }
]
```



accu conference 2025

Continuous Integration (CI) for Large Scale Package-Based C, C++ Projects With Conan2

Diego Rodriguez-Losada



CONTINUOUS INTEGRATION (CI) FOR LARGE SCALE PACKAGE-BASED C, C++ PROJECTS WITH CONAN2

Bloomberg Engineering

accu conference 2025

Building ai/1.1 for Linux

```
$ git clone git@<url>/ai.git && cd ai
$ conan create . -pr=linux
$ conan upload "ai/*" -r=packages
```

37

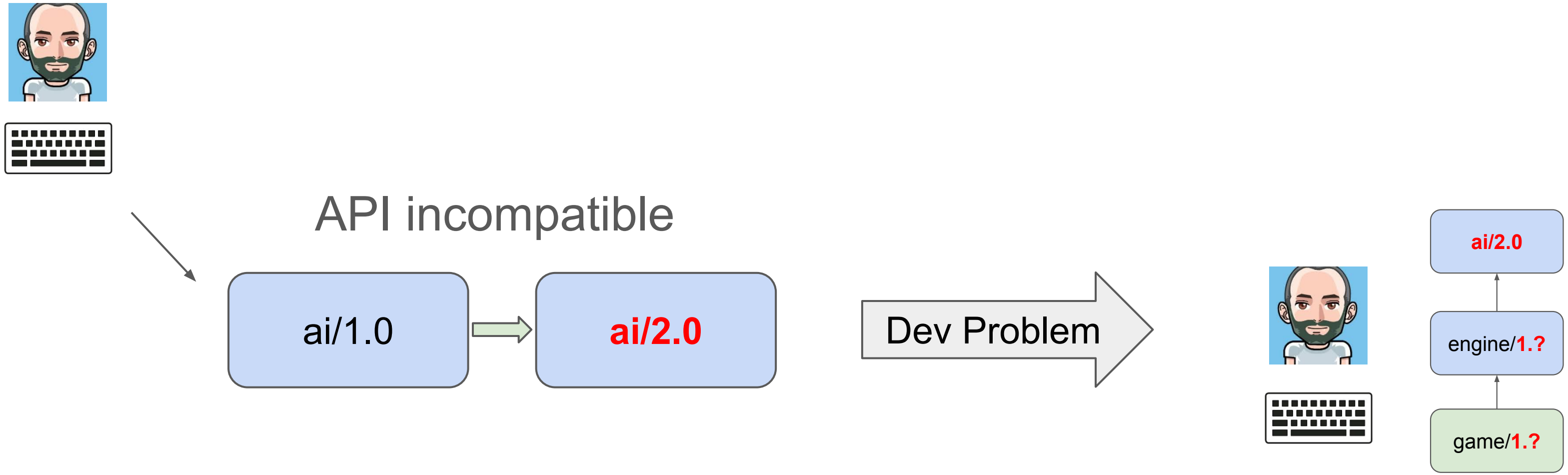
Diego Rodriguez-Losada

ACCU.ORG

https://**youtu.be**/A3X1MpvYTrM

https://docs.conan.io/2/**ci_tutorial**/tutorial.html

The development workspace problem



API incompatible

ai/1.0

ai/2.0

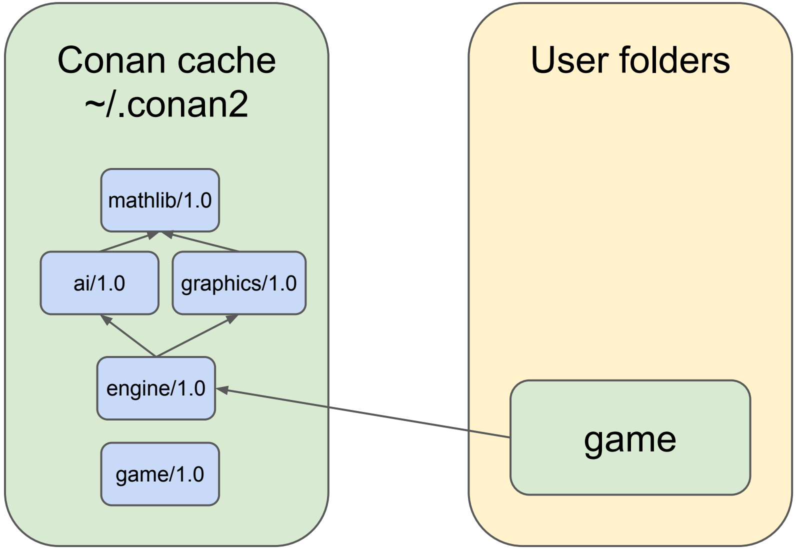
Dev Problem

ai/2.0

engine/1.?

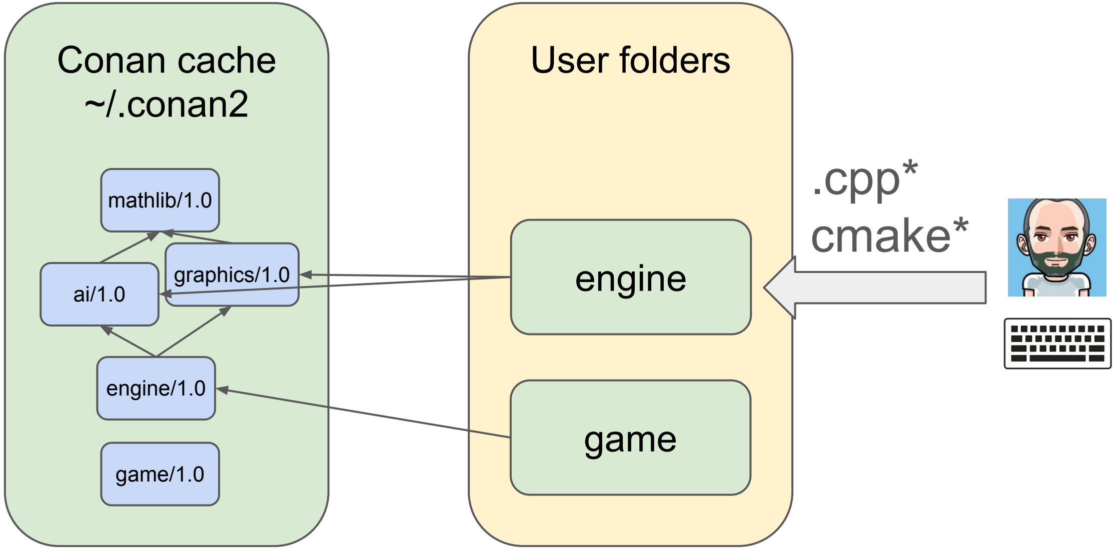
game/1.?

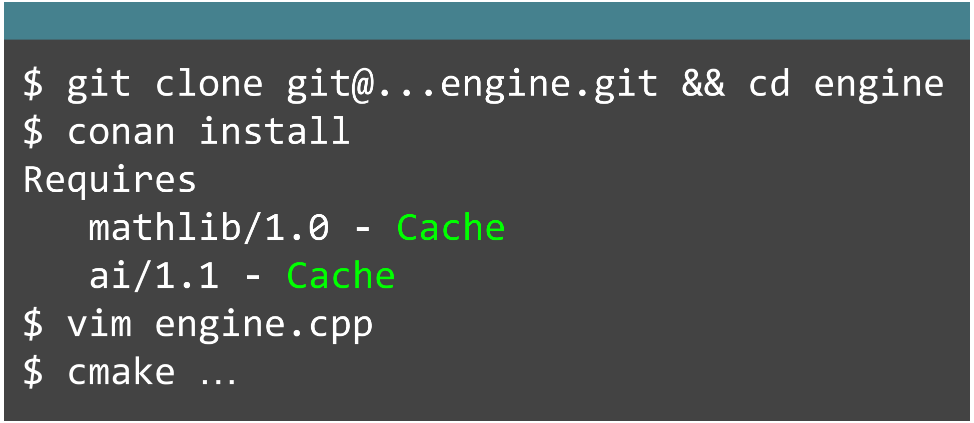
Working on multiple packages simultaneously



```
$ git clone git@...game.git && cd game
$ conan install
Requires
   mathlib/1.0 - Cache
   ai/1.1 - Cache
   engine/1.0 - Cache
```

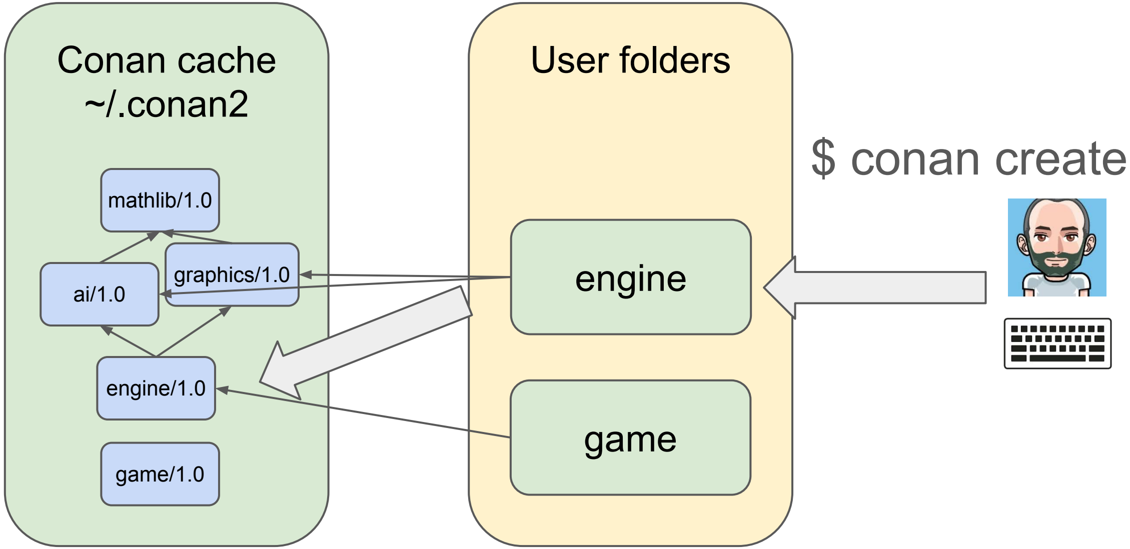
Working on multiple packages simultaneously



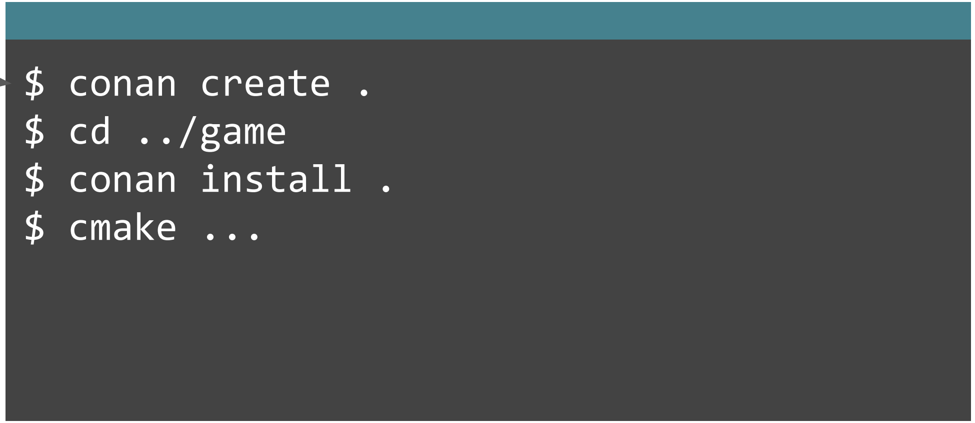


```
$ git clone git@...engine.git && cd engine
$ conan install
Requires
   mathlib/1.0 - Cache
   ai/1.1 - Cache
$ vim engine.cpp
$ cmake …
```

Working on multiple packages simultaneously

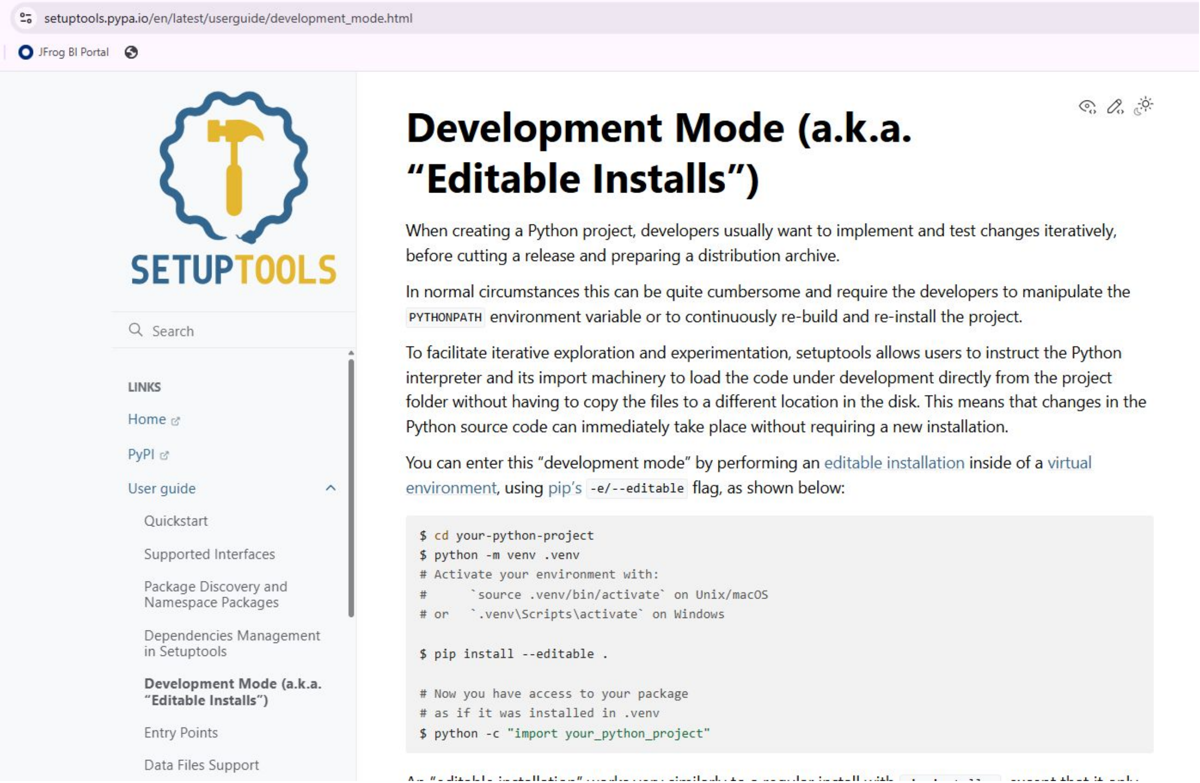


Full build, not incremental



```
$ conan create .
$ cd ../game
$ conan install .
$ cmake ...
```

Editable packages


setuptools.pypa.io/en/latest/userguide/development_mode.html
JFrog BI Portal
SETUPTOOLS
Search
LINKS
Home
PyPI
User guide
Quickstart
Supported Interfaces
Package Discovery and Namespace Packages
Dependencies Management in Setuptools
Development Mode (a.k.a. “Editable Installs”)
Entry Points
Data Files Support

Development Mode (a.k.a. “Editable Installs”)

When creating a Python project, developers usually want to implement and test changes iteratively, before cutting a release and preparing a distribution archive.

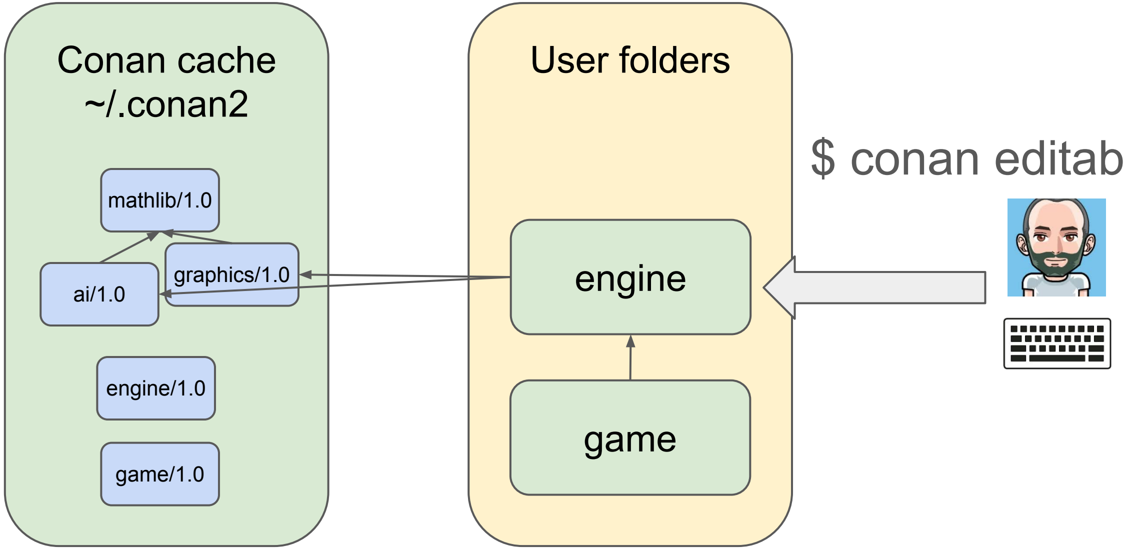
In normal circumstances this can be quite cumbersome and require the developers to manipulate the `PYTHONPATH` environment variable or to continuously re-build and re-install the project.

To facilitate iterative exploration and experimentation, setuptools allows users to instruct the Python interpreter and its import machinery to load the code under development directly from the project folder without having to copy the files to a different location in the disk. This means that changes in the Python source code can immediately take place without requiring a new installation.

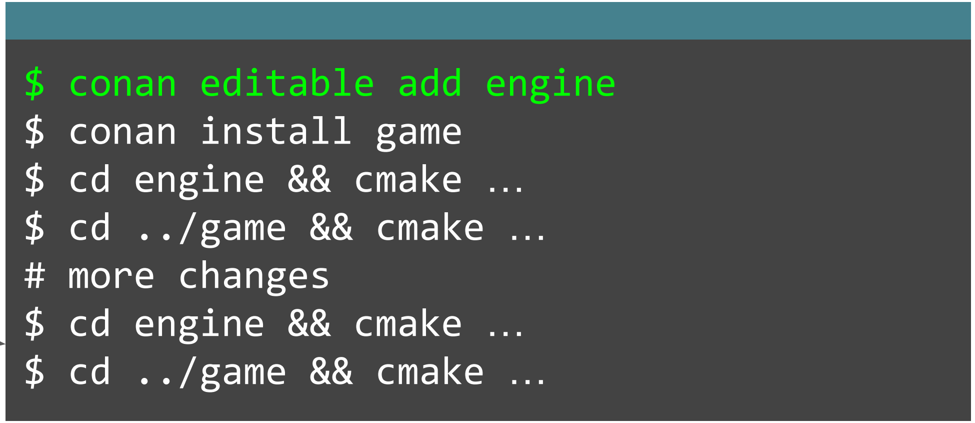
You can enter this “development mode” by performing an editable installation inside of a virtual environment, using pip’s `-e/--editable` flag, as shown below:

```
$ cd your-python-project
$ python -m venv .venv
# Activate your environment with:
#      `source .venv/bin/activate` on Unix/macOS
# or   `.venv\Scripts\activate` on Windows

$ pip install --editable .

# Now you have access to your package
# as if it was installed in .venv
$ python -c "import your_python_project"
```

Editable packages



$ conan editable

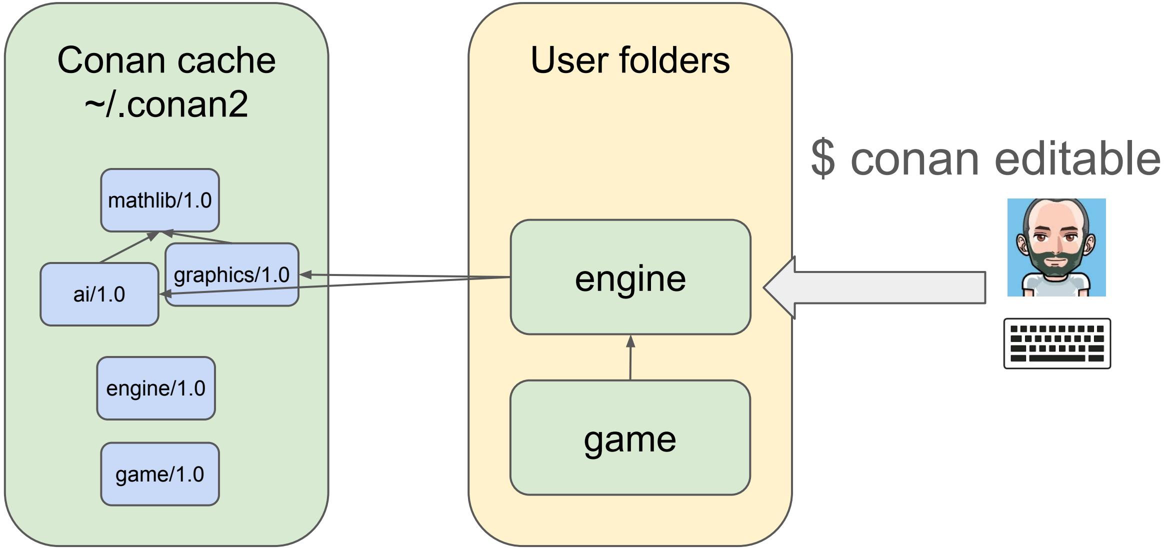


```
$ conan editable add engine
$ conan install game
$ cd engine && cmake …
$ cd ../game && cmake …
# more changes
$ cd engine && cmake …
$ cd ../game && cmake …
```

Incremental builds, much faster!

DEMO

Editable packages



```
$ conan editable add engine
$ conan install game
$ cd engine && cmake ...
$ cd ../game && cmake ...
# more changes
$ cd engine && cmake ...
$ cd ../game && cmake ...
```

Incremental builds, much faster!

WHY NOT BOTH?

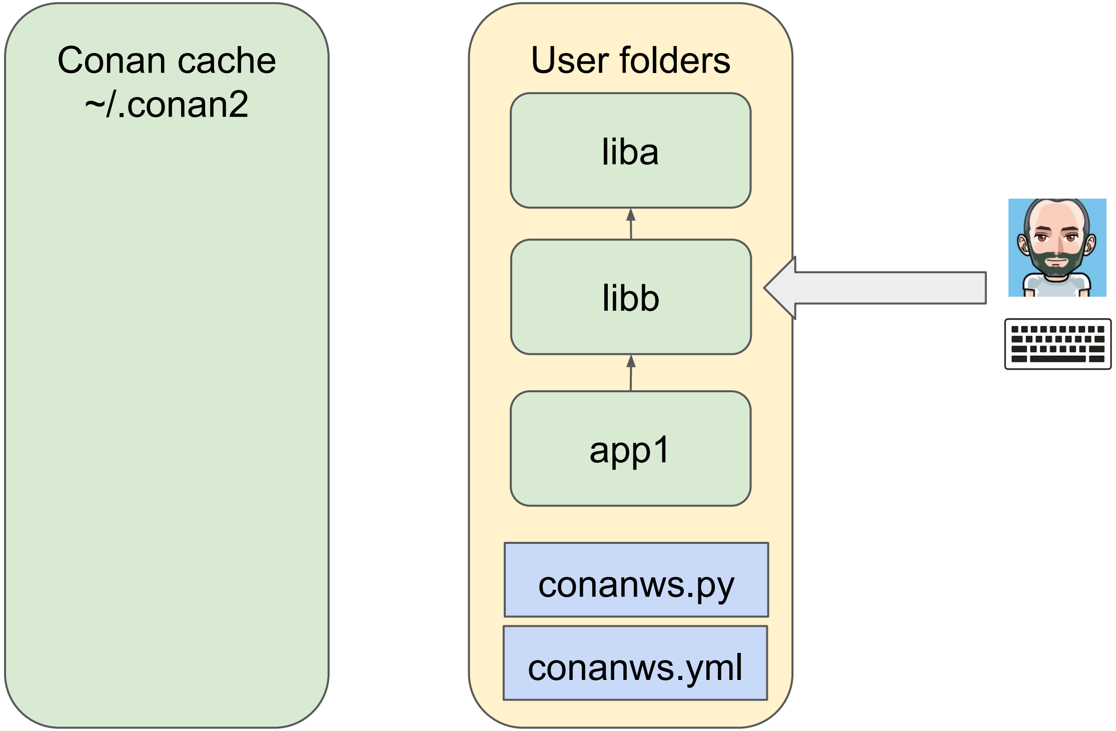
Workspaces!!!

- Definition
- Workspace open/add
  - SCM
- Workspace build (orchestrated)
- Workspace super-install (super-build monolithic)
  - CMakeLists.txt with FetchContent
- Workspace new template

Workspace

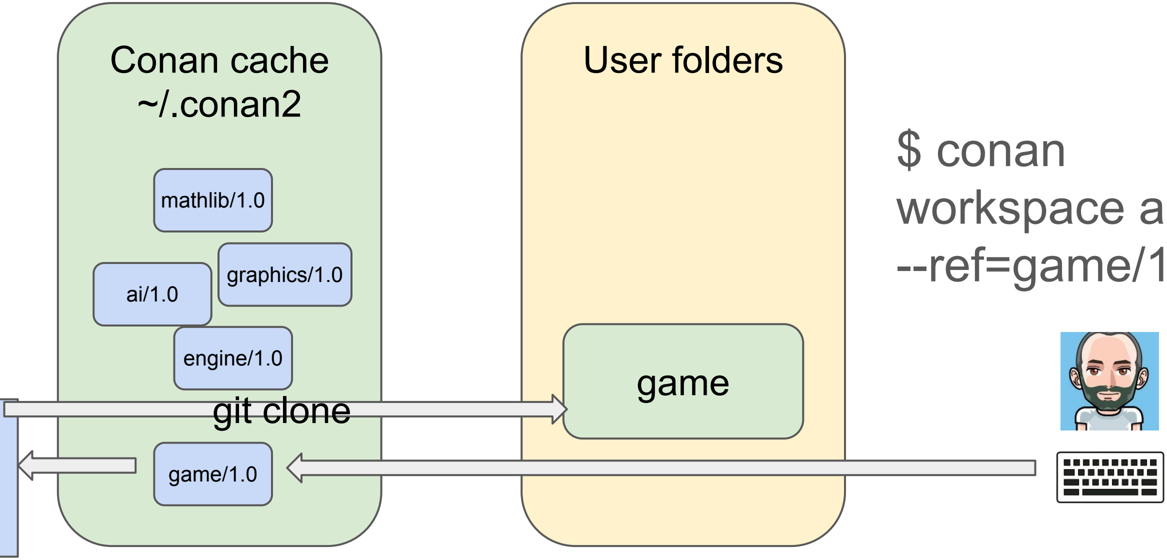
Definition: a dynamic and orchestrated set of locally editable packages:

- Editable definition not global
- Can add/remove packages
- Orchestrated:
  - Multi-repo
  - Mono-repo



DEMO

Dynamic: conan workspace open/add/remove



$ conan workspace add --ref=game/1.0

conandata.yml

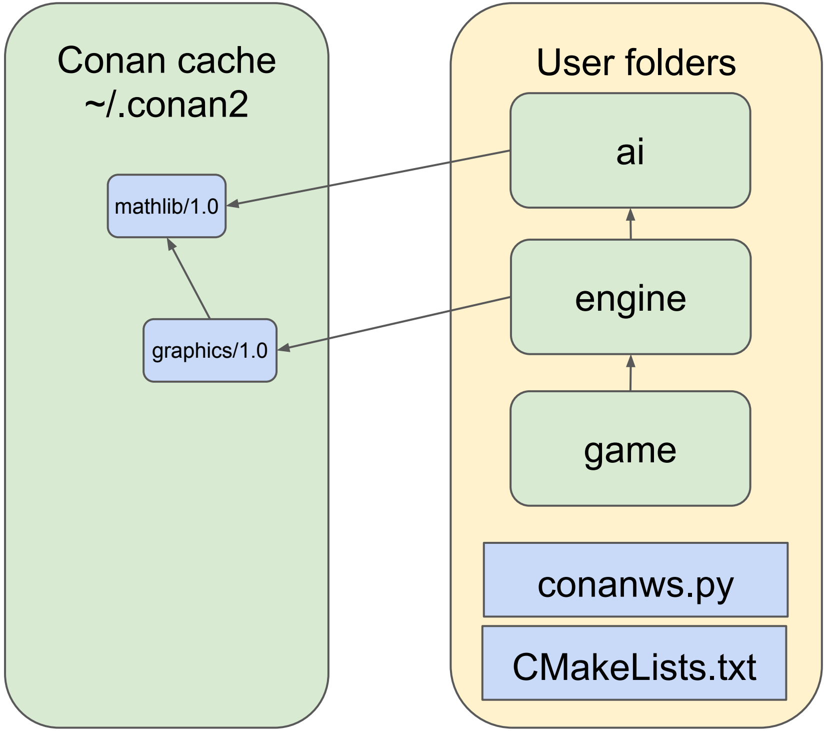
```
scm:
  url: git@github.com…/conanci_game.git
  commit: 0ab1c2…
```

```python
class aiRecipe(ConanFile):
    name = "ai"
    version = "1.0"

    def export(self):
        git = Git(self, self.recipe_folder)
        git.coordinates_to_conandata()
```

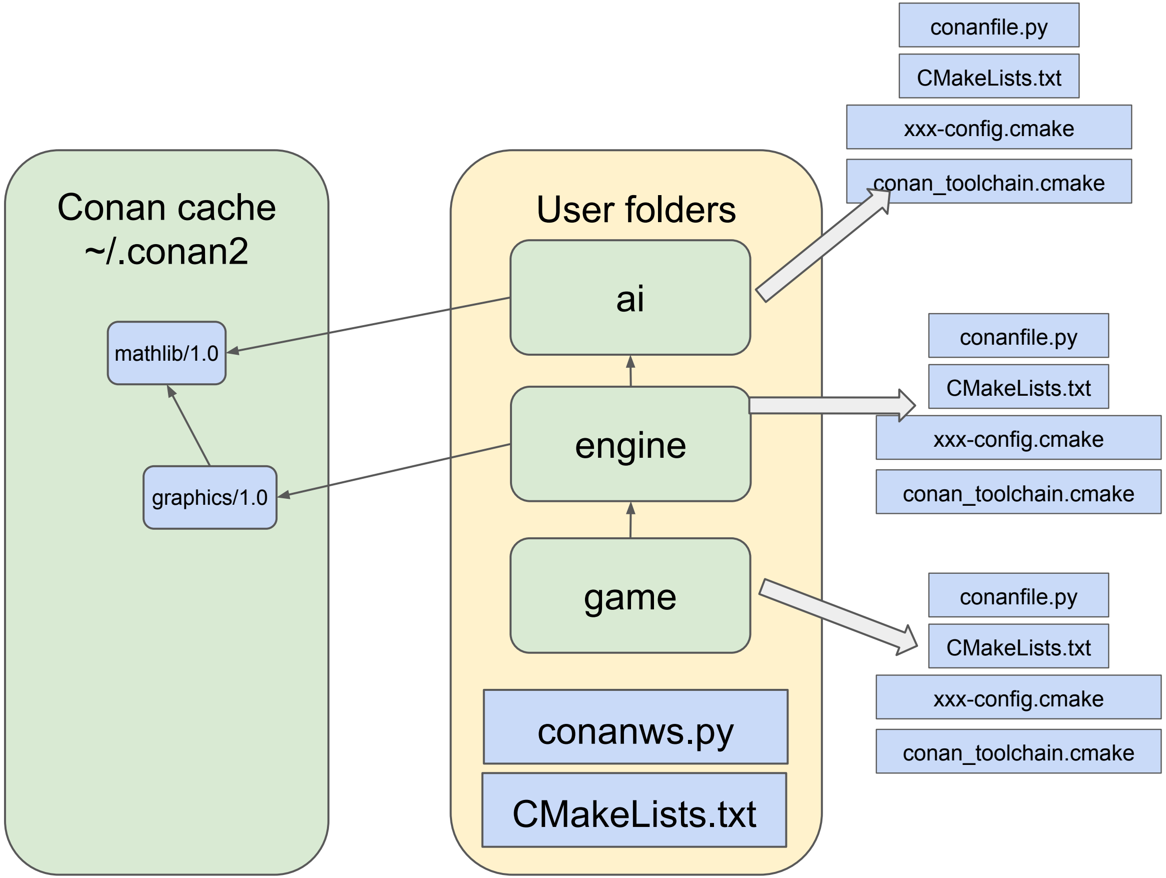
```
$ conan workspace add --ref=game=1.0
# Internally does git clone ...
# Then conan editable add game
```

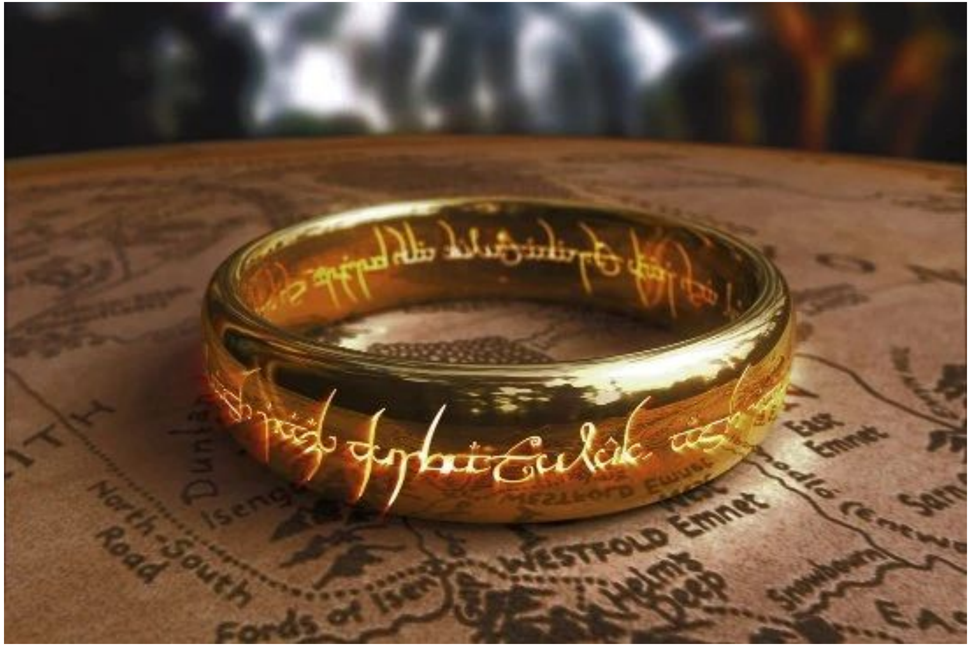
DEMO

Mono-repo like



Mono-repo like

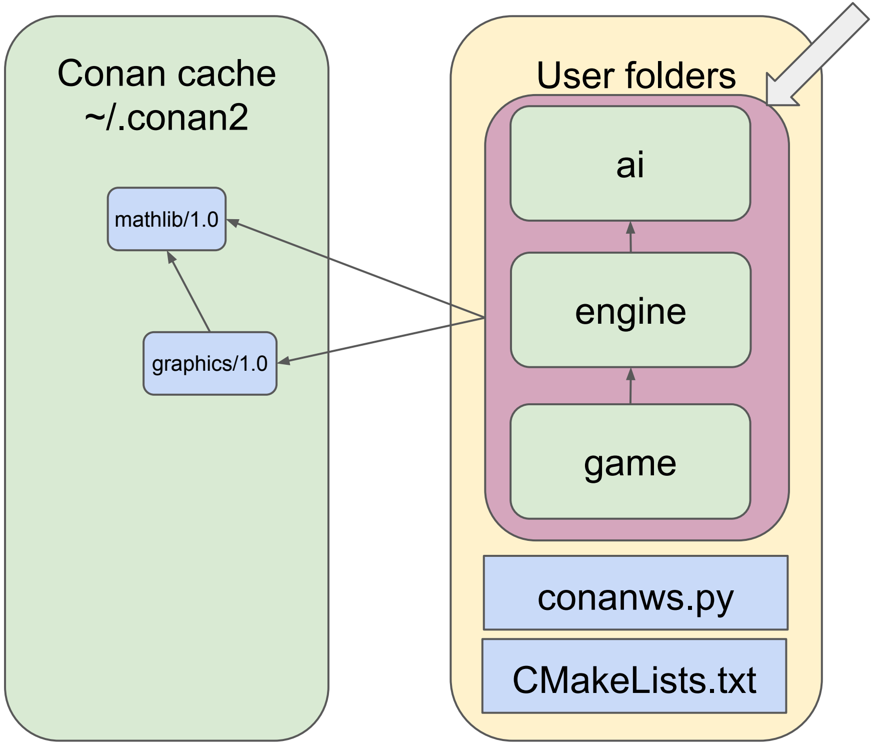




One CMakeLists.txt to rule them all

and one “conan_toolchain.cmake”, one install, 1 project in IDE

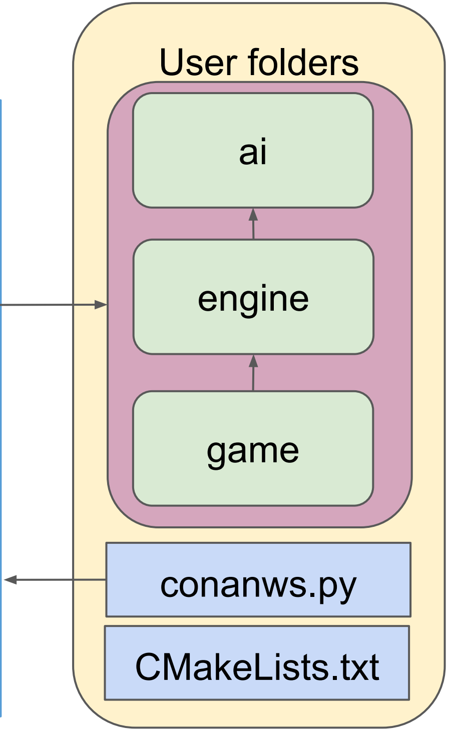
Workspace conanfile



Virtual collapsed node/pkg in the dependency graph

Workspace conanfile

```python
class Ws(Workspace):
    def root_conanfile(self):
        return MyWs


class MyWs(ConanFile):
    settings = "os", "compiler", "build_type", "arch"

    def generate(self):
        deps = CMakeDeps(self)
        deps.generate()
        tc = CMakeToolchain(self)
        tc.preprocessor_definitions["PKG_VERSION"] = '"WS_0.1"'
        tc.generate()

    def layout(self):
        cmake_layout(self)
```



Workspace CMakeLists.txt

```
cmake_minimum_required(VERSION 3.25)
project(myws CXX)

include(FetchContent)

function(add_project PACKAGE_NAME SUBFOLDER)
    FetchContent_Declare(
        ${PACKAGE_NAME}
        SOURCE_DIR ${CMAKE_CURRENT_LIST_DIR}/${SUBFOLDER}
        SYSTEM
        OVERRIDE_FIND_PACKAGE
    )
    FetchContent_MakeAvailable(${PACKAGE_NAME})
endfunction()

add_project(ai ai)
add_library(ai::ai ALIAS ai) # only necessary cause project didn’t
add_project(engine engine)
add_library(engine::engine ALIAS engine)
add_project(game game)
```

Dynamic CMakeLists.txt

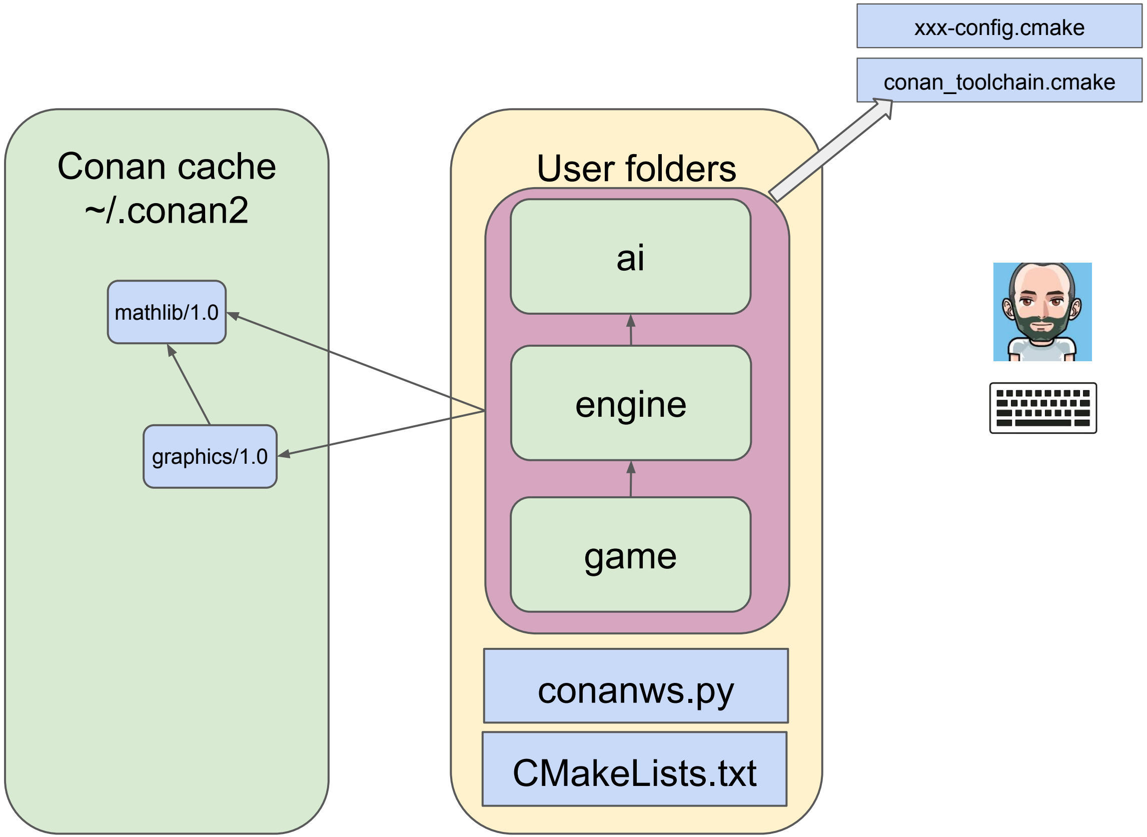
```
function(add_project PACKAGE_NAME SUBFOLDER)
    ...
endfunction()

add_project(ai ai)
add_library(ai::ai ALIAS ai)
add_project(engine engine)
add_library(engine::engine ALIAS engine)
add_project(game game)
```

```
function(add_project PACKAGE_NAME SUBFOLDER)
    ...
endfunction()

include(build/conanws_build_order.cmake)

foreach(pair ${CONAN_WS_BUILD_ORDER})
    string(FIND "${pair}" ":" pos)
    string(SUBSTRING "${pair}" 0 "${pos}" pkg)
    math(EXPR pos "${pos} + 1")  # Skip the separator
    string(SUBSTRING "${pair}" "${pos}" -1 folder)

    add_project(${pkg} ${folder})
    get_target_property(target_type ${pkg} TYPE)
    if (NOT target_type STREQUAL "EXECUTABLE")
        add_library(${pkg}::${pkg} ALIAS ${pkg})
    endif()
endforeach()
```

Dynamic conanws.py

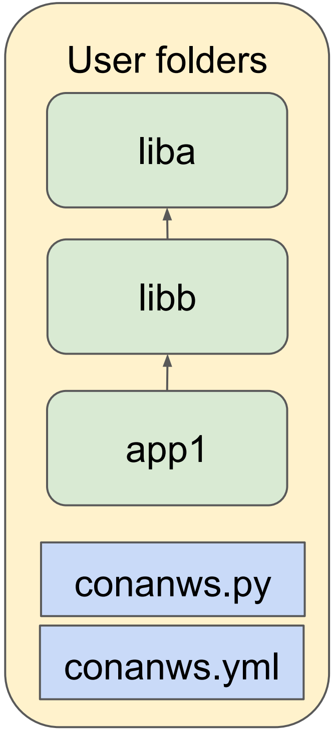
```
class Ws(Workspace):
    def root_conanfile(self):
        return MyWs

    def packages(self):
        result = []
        for f in os.listdir(self.folder):
            if os.path.isdir(os.path.join(self.folder, f)):
                if not os.path.isfile(os.path.join(self.folder, f, "conanfile.py")):
                    continue
                conanfile = self.load_conanfile(f)
                result.append({"path": f,
                               "ref": f"{conanfile.name}/{conanfile.version}"})
        return result

    def build_order(self, order):
        super().build_order(order)  # default behavior prints the build order
        pkglist = " ".join([f'{it["ref"].name}:{it["folder"]}' for level in order for it in level])
        save(self, "build/conanws_build_order.cmake", f"set(CONAN_WS_BUILD_ORDER {pkglist})")
```

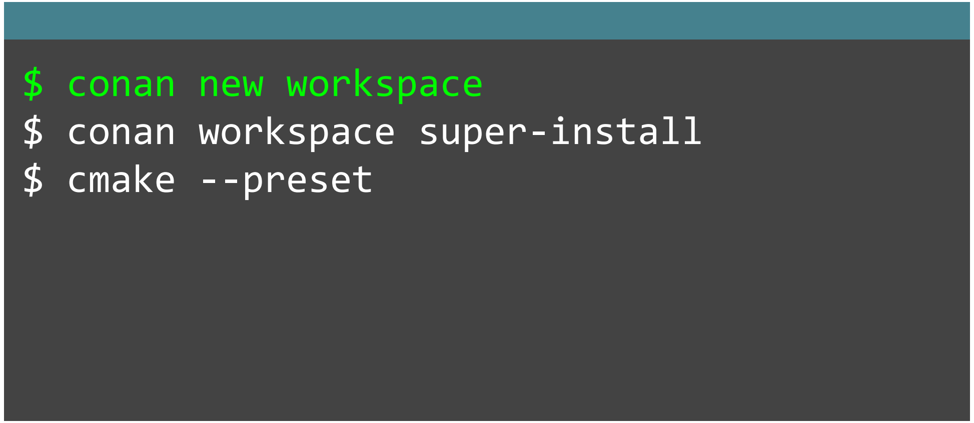
conan workspace super-install



DEMO

Want to experiment? “conan new workspace”





```
$ conan new workspace
$ conan workspace super-install
$ cmake --preset
```

Outline

- Introduction: monorepo vs components
- Challenges of component based development
- Continuous Integration at scale
- Simultaneous development of multiple packages
- **Conclusions**
- QA

Conclusions

- Both monorepo and component based development have their own challenges
- Component/package-based dev challenges:
  - CI at scale
  - Development UX to work on multiple packages
- CI at scale with Conan2
  - 200 lines of GH actions code: **simple!**
  - No extra scripting necessary
  - **Escalable**, for any graph size, any number of configurations (architectures, platforms), any number or products. **Without explicit model in CI!**
  - Jenkins or similar preferred for the products pipeline
- Workspaces: Developing multiple packages in a mono-repo project
  - **Simple**, standard and out of the box
  - 30 lines of CMakeLists + 50 lines of conanws.py

Conclusions

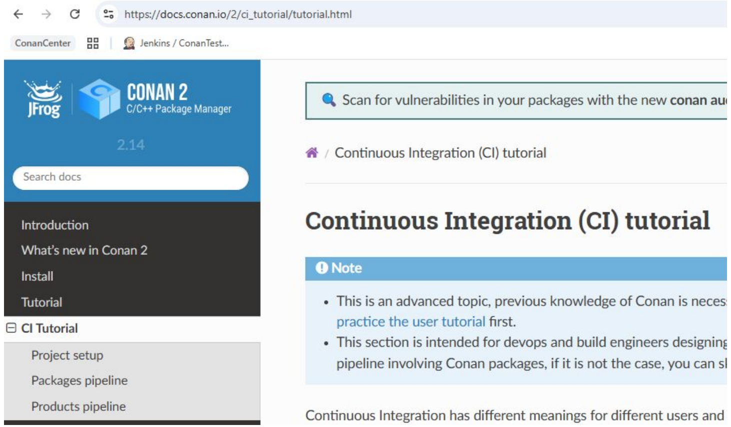
- **For the first time in C++ we have:**
  - Component/package based approach
  - A framework for scalable CI
  - Standard monorepo like development experience
  - With familiar and established tooling: CMake and Conan2
    - 30 lines of CMake + 50 lines of conanws.py
  - Extensible to MSBuild
- **An enterprise ready C, C++ tooling framework for dependencies, packaging, continuous integration and development**

Component based paradigm

Seen by component based developers

Thank you!

Source code: https://github.com/memsharded/conanci_*



https://docs.conan.io/2/ci_tutorial/tutorial.html

Scan for vulnerabilities in your packages with the new **conan au**

Continuous Integration (CI) tutorial

Continuous Integration (CI) tutorial

Note

- This is an advanced topic, previous knowledge of Conan is neces practice the user tutorial first.
- This section is intended for devops and build engineers designing pipeline involving Conan packages, if it is not the case, you can sl

Continuous Integration has different meanings for different users and

https://docs.conan.io

https://conan.io

https://github.com/conan-io/conan