Desde que uso Pixi estoy tan Pichi

Entornos reproducibles para ROS 2 con Pixi, RoboStack y Prefix.dev

Francisco Martín Rico
fmrico@gmail.com

Esteve Fernández
esteve.fernandez@gmail.com

Agenda

1. Introducción: ¿por qué ROS 2 necesita algo mejor?
2. ¿Qué es Pixi?
3. Instalación y uso básico de Pixi
4. ¿Qué es RoboStack?
5. Uso básico de RoboStack para compilar un workspace ROS 2
6. Infraestructura de RoboStack

1 Introducción: ¿por qué ROS 2 necesita algo mejor?

2 ¿Qué es Pixi?

3 Instalación y uso básico de Pixi

4 ¿Qué es RoboStack?

5 Uso básico de RoboStack para compilar un workspace ROS 2

6 Infraestructura de RoboStack

7 Tu propia Buildfarm basada en RoboStack

Antes de empezar...

- ¿Cuántos habéis tenido un `rosdep install` que falla distinto en vuestro portátil que en el del compañero?
- ¿Cuántos tenéis un `Dockerfile` de ROS 2 de más de 150 líneas solo para fijar versiones?
- ¿Cuántos tenéis instalado más de un `/opt/ros/<distro>` en la misma máquina a la vez?

No sois los únicos. Este workshop va de eso.

El problema, en tres capas

Dependencias

ROS 2 vía apt está atado a una versión de Ubuntu concreta. Mezclar ROS con librerías de ML modernas (PyTorch, CUDA reciente) choca con lo que trae el repositorio del sistema.

Reproducibilidad

`apt install ros-jazzy-desktop` hoy y dentro de 6 meses puede instalar builds distintos. No hay lockfile. `rosdep` resuelve llamando a gestores distintos (`apt`, `pip`...) sin fijar versiones.

Overlays

El modelo underlay/overlay de `colcon` (`source /opt/ros/.../setup.bash` → `source install/setup.bash`) es manual y frágil.

¿Y Docker no resolvía esto?

- Docker aísla, pero **no fija automáticamente el contenido**: un `apt-get install` dentro del Dockerfile sigue sin estar reproducido a menos que se congele todo a mano.
- Overhead de build/caché por capas → iteración lenta en desarrollo.
- No es multiplataforma "gratis": desarrollar en macOS/Windows para un target Linux añade capas de virtualización.

Mensaje clave

Docker sigue siendo válido para el despliegue final, pero no debería ser la **única** fuente de verdad de las dependencias del proyecto.

La idea central de hoy

¿Y si tuviéramos el equivalente a un `package-lock.json`
o un `Cargo.lock`

para ROS 2?

Que funcione igual en tu portátil, en el del compañero y en CI.
Sin `sudo`. Sin depender de la versión de Ubuntu.
Sin tener que aprender Docker desde cero.

Las tres piezas

Pixi

Gestor de entornos reproducible: lockfile, tasks, multi-entorno.

RoboStack

ROS 2 empaquetado como paquetes conda, multiplataforma.

Prefix.dev

Dónde vive y se publica todo: canales conda, públicos y privados.

Hoy vamos a recorrer las tres, de teoría a práctica.

1 Introducción: ¿por qué ROS 2 necesita algo mejor?

2 ¿Qué es Pixi?

3 Instalación y uso básico de Pixi

4 ¿Qué es RoboStack?

5 Uso básico de RoboStack para compilar un workspace ROS 2

6 Infraestructura de RoboStack

7 Tu propia Buildfarm basada en RoboStack

¿Qué es Pixi?

- Gestor de entornos y paquetes **multiplataforma y multi-lenguaje**, escrito en Rust.
- Construido sobre **Rattler**: librerías Rust que reimplementan el ecosistema conda (resolución, instalación, canales). Rattler no es una CLI, es el motor; Pixi es la interfaz de usuario.
- Creado por el equipo de **prefix.dev** (los mismos autores de **mamba**).

No es "conda pero más rápido"

Repiensa el flujo de trabajo: sin entorno base global, workspace-centric, lockfile nativo de primera clase.

? ¿Quién ha usado conda o mamba antes? ¿Qué es lo que más odiáis?

Pixi frente a conda: diferencias de fondo

- Sin "entorno base" tipo Miniconda/Anaconda: Pixi es un binario estático, nada que activar a nivel de sistema.
- **Workspace-centric**: todo vive en una carpeta con `pixi.toml` + `pixi.lock`, autocontenida, clonable, versionable en git.
- Lockfile nativo de primera clase (`pixi.lock`) — algo que conda nunca tuvo.
- Combina en el mismo manifiesto dependencias **conda** (C++, CUDA, binarios de sistema) y **PyPI** (resueltas con un motor basado en `uv`).

Pixi vs. el resto del ecosistema

Feature	Pixi	Conda	Pip	Poetry
Lockfile reproducible	✓	–	–	✓
Task runner integrado	✓	–	–	–
Multi-lenguaje (no solo Py)	✓	✓	–	–
Multi-entorno en un proyecto	✓	parcial	–	parcial

Pixi resuelve e instala un entorno desde cero ~3× más rápido que micromamba y >10× más rápido que conda clásico.

Estructura de un pixi.toml

```
[workspace]
name = "my_ros2_project"
version = "0.1.0"
channels = ["robostack-jazzy", "conda-forge"]
platforms = ["linux-64"]

[dependencies]
ros-jazzy-desktop = "*"
colcon-common-extensions = "*"

[tasks]
build = { cmd = "colcon build --symlink-install", inputs = ["src"] }
sim   = "ros2 run turtlesim turtlesim_node"

[activation]
scripts = ["install/setup.sh"]
```

Secciones del manifiesto

- `[workspace]` — metadatos + `channels` + `platforms`.
- `[dependencies]` (conda) / `[pypi-dependencies]` (PyPI, resuelto con `uv`) — se pueden mezclar.
- `[tasks]` — comandos del proyecto.
- `[activation]` — scripts/env vars que se cargan automáticamente al activar el entorno.
- `[feature.*]` + `[environments]` — varios entornos combinables en el mismo repo.
- `[target.<platform>.*]` — overrides específicos de plataforma.

También puede vivir embebido en `pyproject.toml` bajo `[tool.pixi.*]`.

Flujo de trabajo típico

```
pixi init                # crea el workspace (pixi.toml)
pixi add <paquete>        # añade dependencia, resuelve, instala
pixi install              # instala desde el lockfile existente
pixi run <comando>        # ejecuta dentro del entorno, sin "activate"
pixi shell                # abre un shell interactivo con el entorno activado
pixi task add / list       # gestiona tasks
pixi lock / pixi update     # gestiona el lockfile
pixi global install        # instala herramientas CLI globales, aisladas
```

Lockfiles: el porqué de la reproducibilidad

- `pixi.lock` fija versión exacta, build hash y **sha256** de cada paquete (conda y PyPI), **por cada plataforma declarada**.
- Se actualiza automáticamente con `pixi add` / `pixi install` / `pixi run`.
- `pixi install --locked` → falla si el lockfile no está sincronizado (ideal para CI).
- `pixi install --frozen` → instala exactamente lo que dice el lockfile.

Mensaje central

`pixi.lock` se commitea a git: todo el equipo, CI y hasta el propio robot instalan exactamente los mismos binarios, byte a byte.

Tasks y entornos multiplataforma (aperitivo)

- Las tasks pueden encadenarse (`depends-on`), cachear por `inputs`/`outputs` (no recompila si nada cambió), y tener parámetros con Jinja.
- `platforms = [...]` permite resolver dependencias distintas por SO/arquitectura en el mismo `pixi.lock` — incluso requisitos "ricos" como CUDA o versión mínima de glibc.

Todo esto se practica en el siguiente bloque.

Ventajas concretas para ROS 2

Problema	Cómo lo resuelve Pixi
ROS atado a versión de Ubuntu	Paquetes conda multiplataforma, sin depender de `apt`
Sin reproducibilidad real	`pixi.lock` fija todo, byte a byte, por plataforma
Overlays manuales y frágiles	`[activation]` automatiza el `source` en cada `pixi shell/run`
Varias distros ROS a la vez	`[environments]` — Humble y Jazzy en paralelo, mismo repo

Un dato honesto

RoboStack es hoy Tier 3

En la clasificación oficial de plataformas de ROS: soporte **comunitario**, no de Open Robotics.

Pero con adopción creciente, y comunicación de que formará parte de la estrategia de accesibilidad de Open Robotics.

1 Introducción: ¿por qué ROS 2 necesita algo mejor?

2 ¿Qué es Pixi?

3 Instalación y uso básico de Pixi

4 ¿Qué es RoboStack?

5 Uso básico de RoboStack para compilar un workspace ROS 2

6 Infraestructura de RoboStack

7 Tu propia Buildfarm basada en RoboStack

PRÁCTICA

Instalación y uso básico de Pixi

30 minutos

Objetivo de este bloque

- Instalar Pixi y crear el primer workspace.
- Añadir dependencias, ejecutar comandos, definir tasks encadenadas con caché.
- Practicar múltiples entornos en el mismo proyecto.

Todavía sin ROS

Usamos ejemplos ligeros en Python para dominar la mecánica de Pixi sin mezclar curvas de aprendizaje. ROS llega en el bloque 5.

Ejercicio 1 — Instalación y primer workspace

```
curl -fsSL https://pixi.sh/install.sh | sh
# reiniciar la terminal
pixi --version
```

```
pixi init pixi-demo
cd pixi-demo
cat pixi.toml
ls -la
```

Checkpoint: `pixi.toml` recién generado con `[workspace]`, además de `.gitignore` y `.gitattributes`.

Ejercicio 2 — Dependencias y ejecución

```
pixi add python=3.11 numpy rich
pixi run python -c "import numpy; print(numpy.__version__)"
pixi list
pixi tree numpy
```

Checkpoint: `pixi list` muestra versiones exactas; `pixi tree` muestra dependencias transitivas. `pixi.lock` ha cambiado.

? ¿`pixi add --pypi requests` y `pixi add requests` instalan lo mismo?

Ejercicio 3 — Tasks encadenadas y con caché

```
[tasks]
hello = "python -c \"print('hola workshop')\""
build = { cmd = "python -m py_compile main.py",
          depends-on = ["hello"], inputs = ["main.py"] }
```

```
echo "print('soy main.py')" > main.py
pixi run build
pixi task list
```

Reto: ejecutar `pixi run build` dos veces sin tocar `main.py`. ¿Qué esperáis que cambie?

Ejercicio 4 — Multi-entorno con features

```
[feature.py310.dependencies]
python = "3.10.*"
[feature.py311.dependencies]
python = "3.11.*"

[environments]
py310 = ["py310"]
py311 = ["py311"]
```

```
pixi install
pixi run -e py310 python --version
pixi run -e py311 python --version
```

Checkpoint: dos versiones de Python distintas, mismo repo, sin virtualenvs.

Cierre del bloque

`init` `add` `run` `shell` `task` `environments`

Esto —dos versiones de Python conviviendo en el mismo proyecto— es exactamente el mecanismo que usaremos para tener ROS 2 Humble y Jazzy en el mismo repo sin pisarse.

1 Introducción: ¿por qué ROS 2 necesita algo mejor?

2 ¿Qué es Pixi?

3 Instalación y uso básico de Pixi

4 ¿Qué es RoboStack?

5 Uso básico de RoboStack para compilar un workspace ROS 2

6 Infraestructura de RoboStack

7 Tu propia Buildfarm basada en RoboStack

¿Qué es RoboStack?

- Empaqueta distribuciones completas de ROS 1/2 como **paquetes conda**.
- Convención de nombres: `ros-<distro>-<paquete>`, p. ej. `ros-jazzy-desktop`, `ros-jazzy-rclcpp`.
- Permite `pixi add ros-jazzy-desktop` igual que se añadiría `numpy`.

Origen y motivación

- Paper académico: *"A RoboStack Tutorial"*, Fischer et al., IEEE Robotics & Automation Magazine, 2021 (arXiv:2104.12910).
- Wolf Vollprecht (creador de mamba, cofundador de prefix.dev) y Tobias Fischer, entre otros.
- Motivación original: mezclar ROS con el ecosistema de ciencia de datos (PyTorch, Jupyter, OpenCV) sin pelearse con versiones fijas de `apt`, sin `sudo`, y en macOS/Windows además de Linux.

Canales conda de RoboStack

Canal	Para qué sirve
`robostack-<distro>`	**Recomendado para uso normal.** Uno por distro ROS.
`robostack-staging`	Interno/histórico, proceso de build de los mantenedores.
`conda-forge`	Dependencias no-ROS (compiladores, Boost, OpenCV, Python...).

Regla simple

Para uso normal, siempre `robostack-<vuestra-distro>` + `conda-forge`. Nada más.

Compatibilidad multiplataforma

- Linux: `linux-64`, `linux-aarch64`
- macOS: `osx-64`, `osx-arm64` (sí, ROS 2 en Apple Silicon)
- Windows: `win-64`

CI propio por plataforma en GitHub Actions.

Aviso importante

rosdep no funciona dentro de Pixi/RoboStack

Se salta la resolución de Pixi llamando directamente a `apt`/`pip` del sistema.

- Las dependencias de un paquete propio hay que traducirlas a mano: `pixi add ros-<distro>-<paquete>`.
- O usar la herramienta comunitaria **pixi-ros**, que lee los `package.xml` del workspace y genera el `pixi.toml`.

Relación con Pixi

RoboStack no sustituye a Pixi.
Es "solo" un conjunto de canales conda más.

Todo lo del bloque anterior (`add`, `run`, `shell`, tasks, `[environments]`, `[activation]`) se aplica sin cambios.

? Con lo que sabéis de Pixi, ¿cómo creéis que se instalaría ROS 2 Jazzy Desktop?

1 Introducción: ¿por qué ROS 2 necesita algo mejor?

2 ¿Qué es Pixi?

3 Instalación y uso básico de Pixi

4 ¿Qué es RoboStack?

5 Uso básico de RoboStack para compilar un workspace ROS 2

6 Infraestructura de RoboStack

7 Tu propia Buildfarm basada en RoboStack

PRÁCTICA

RoboStack + colcon

20 minutos

Ejercicio 1 — Entorno ROS 2 en dos comandos

```
pixi init taller_ros2 -c robostack-jazzy -c conda-forge
cd taller_ros2
pixi add ros-jazzy-desktop
pixi run ros2 topic list
```

```
# Terminal 1
pixi run ros2 run demo_nodes_cpp talker
# Terminal 2
pixi run ros2 run demo_nodes_cpp listener
```

Checkpoint: el listener recibe los mensajes del talker. Sin `sudo`, sin tocar el sistema.

Ejercicio 2 — Paquete propio + colcon

```
pixi add colcon-common-extensions "setuptools<=58.2.0"
mkdir src
pixi run ros2 pkg create --build-type ament_python \
  --destination-directory src --node-name talker_custom mi_paquete
pixi run colcon build --symlink-install
```

Checkpoint: aparecen `build/`, `install/`, `log/` junto a `src/`.

El overlay, automatizado

```
# añadir a pixi.toml
[activation]
scripts = ["install/setup.sh"]
```

```
pixi shell
echo $AMENT_PREFIX_PATH   # install/ del workspace, primero
ros2 run mi_paquete talker_custom
exit
```

Checkpoint: `$AMENT_PREFIX_PATH` lista `.../taller_ros2/install` como primera entrada. Overlay funcionando, sin `source install/setup.bash` a mano.

? ¿Qué pasaría si borro `install/` pero dejo el `[activation]` en el `pixi.toml`?

Ejercicio 3 — Tasks con caché encadenadas

```
[tasks]
build      = { cmd = "colcon build --symlink-install", inputs =
    ["src"] }
run_talker = { cmd = "ros2 run mi_paquete talker_custom",
               depends-on = ["build"] }
```

```
pixi run run_talker
pixi run run_talker   # 2a vez: ¿recompila?
# editar talker_custom.py y repetir
pixi run run_talker   # 3a vez: ¿y ahora?
```

Ejercicio 4 — Un workspace real: Nav2 con `pixi-ros`

- Hasta ahora, un paquete de juguete. ¿Y un workspace real y grande?
- Y de paso, resolvemos lo que dejamos abierto en el bloque anterior: si `rosdep` no funciona, ¿cómo relleno las dependencias de un `package.xml` real sin hacerlo a mano?

`pixi-ros`

Herramienta de la comunidad Pixi: escanea los `package.xml` del workspace y rellena el `pixi.toml` con las dependencias ya resueltas a paquetes conda de RoboStack. El `rosdep install` moderno, pero reproducible.

Aviso de timing

Nav2 son más de 80 paquetes C++. El instructor lanza el build al empezar el bloque anterior — aquí mostramos el resultado en vivo.

Preparar el workspace

```
pixi global install pixi-ros
```

```
mkdir -p nav2_ws/src && cd nav2_ws
git clone -b jazzy https://github.com/ros-navigation/navigation2.git \
  src/navigation2
```

Rellenar dependencias automáticamente

```
pixi-ros init --distro jazzy --platform linux-64
```

- Escanea todos los `package.xml` (`nav2_bt_navigator`, `nav2_costmap_2d`, `nav2_planner`...).
- Resuelve cada dependencia por prioridad: workspace → ficheros de mapeo → índice de la distro ROS → conda-forge → canales extra.
- Muestra una tabla de validación con el origen de cada dependencia.

Checkpoint: el `pixi.toml` generado tiene decenas de `ros-jazzy-*` + herramientas de build, y un bloque `[tasks]` con `build`/`test` ya creados (`colcon` por debajo). Es idempotente.

Compilar y verificar

```
pixi install
pixi run build
```

```
pixi shell
ros2 pkg list | grep nav2
ros2 pkg executables nav2_costmap_2d
```

Checkpoint: decenas de paquetes `nav2_*`, compilados dentro de este entorno Pixi, sin tocar el sistema ni usar `rosdep`.

El cierre real: simulación + RViz2

`nav2_bringup` declara en su `package.xml`: `ros_gz_bridge`, `ros_gz_sim`, `xacro`, `slam_toolbox`, `nav2_minimal_tb3_sim`.

Todo ya resuelto por `pixi-ros` como binarios conda de RoboStack. No hace falta clonar ningún repo más.

```
ros2 launch nav2_bringup tb3_simulation_launch.py
```

Levanta **Gazebo Sim** (`gz sim`) con el robot spawneado, toda la pila de Nav2, y **RViz2** con la vista por defecto.

Mandar al robot a navegar, en RViz2

1. Botón **2D Pose Estimate**: click y arrastra sobre el mapa en la posición/orientación real del robot (localizarlo para AMCL).
2. Botón **Nav2 Goal**: click y arrastra en el punto de destino, con la orientación final deseada.
3. El robot planifica una ruta y se mueve solo hasta el objetivo, esquivando obstáculos.

Checkpoint: el robot navega autónomamente en la simulación — Gazebo, RViz2 y Nav2 completo, dentro del mismo entorno Pixi, sin `sudo`, sin `apt`, sin Docker.

Aviso honesto: GPU/OpenGL

`gz sim` necesita una pila gráfica funcional. En VMs/WSL2 sin passthrough de GPU puede fallar o ir lento — probadlo antes del taller. Fallback: `LIBGL_ALWAYS_SOFTWARE=1` delante del comando, o hacerlo como demo proyectada del instructor.

Ejercicio 5.5 (opcional) — Varias distros ROS 2 a la vez (1/2)

Mismo patrón que el ejercicio 3.4 (Python), pero cada distro RoboStack vive en su propio canal: no se pueden mezclar `robostack-jazzy` y `robostack-humble` en la misma resolución.

```
[feature.jazzy]
channels = ["robostack-jazzy", "conda-forge"]
[feature.jazzy.dependencies]
ros-jazzy-desktop = "*"

[feature.humble]
channels = ["robostack-humble", "conda-forge"]
[feature.humble.dependencies]
ros-humble-desktop = "*"

[environments]
jazzy  = ["jazzy"]
humble = ["humble"]
```

Ejercicio 5.5 (opcional) — Varias distros ROS 2 a la vez (2/2)

```
pixi install
pixi run -e jazzy  ros2 --version
pixi run -e humble ros2 --version
```

Checkpoint: cada comando reporta la versión de `ros2` de su propia distro — dos ROS 2 completos, en el mismo repositorio, sin pisarse.

Dos ROS a la vez, sin pisarse

Imposible o muy incómodo con `apt`.

Con Pixi, un `[environments]` más.

Exactamente el mismo mecanismo del bloque 3 con Python — solo cambia qué metemos dentro de cada *feature*.

1 Introducción: ¿por qué ROS 2 necesita algo mejor?

2 ¿Qué es Pixi?

3 Instalación y uso básico de Pixi

4 ¿Qué es RoboStack?

5 Uso básico de RoboStack para compilar un workspace ROS 2

6 Infraestructura de RoboStack

7 Tu propia Buildfarm basada en RoboStack

De `package.xml` a paquete conda

En el bloque anterior usamos paquetes ya construidos. ¿Cómo se construyen?

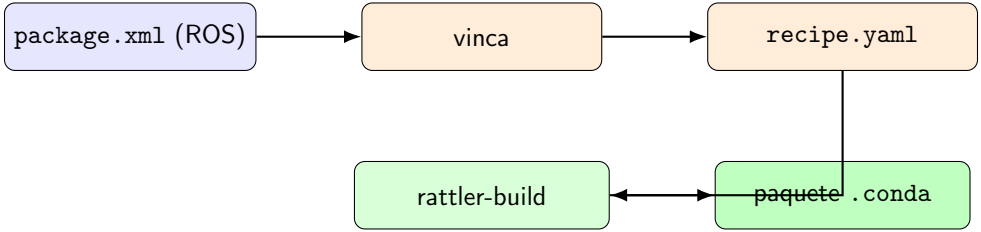
- **vinca**: lee el *rosdistro* oficial (metadata + `package.xml` de cada paquete) y un fichero `vinca.yaml`, y genera recetas en formato **rattler-build** (`recipe.yaml`, esquema `prefix-dev/recipe-format`).
- **rattler-build**: el motor que compila cada receta y produce el `.conda` final. Sucesor moderno de `conda-build/boa`.

Recetas y build farms

- Cada repo de distro (`RoboStack/ros-jazzy`, `ros-humble`, `ros-kilted`...) contiene la *configuración* (`vinca.yaml`, `robostack.yaml`, `patches/`, `conda_build_config.yaml`) — pero **no** recetas estáticas: se regeneran en cada build.
- CI en **GitHub Actions**, matriz por plataforma (`linux-64`, `linux-aarch64`, `osx-64`, `osx-arm64`, `win-64`), con caché de builds y generación dinámica de workflows.

Relación con conda-forge y contribución

- RoboStack se apoya en **conda-forge** para todas las dependencias no-ROS (compiladores, Boost, PCL, OpenCV, Python...).
- Contribuir un paquete nuevo, en general: PR de una línea en `vinca.yaml` → CI intenta construirlo en todas las plataformas → si falla, parche local guardado en `patches/` → rebuild para validar.



Este es exactamente el pipeline que vais a recorrer, en pequeño, en el próximo bloque práctico.

1 Introducción: ¿por qué ROS 2 necesita algo mejor?

2 ¿Qué es Pixi?

3 Instalación y uso básico de Pixi

4 ¿Qué es RoboStack?

5 Uso básico de RoboStack para compilar un workspace ROS 2

6 Infraestructura de RoboStack

7 Tu propia Buildfarm basada en RoboStack

PRÁCTICA

Tu propia Buildfarm

35 minutos

Objetivo y nota de diseño

- Construir un paquete conda propio con `rattler-build`.
- Publicarlo en un canal de prefix.dev.
- Instalarlo desde un `pixi.toml` propio.
- Integrarlo en un workspace ROS 2.

No usamos Vinca hoy

Generar recetas reales de ROS 2 completo tarda demasiado para 1h. Escribimos una receta `rattler-build` sencilla, con el mismo formato exacto que usa la buildfarm real.

Fase A — Preparar el proyecto (1/3)

```
mkdir -p ~/taller-pixi/mi-paquete && cd ~/taller-pixi/mi-paquete
pixi init .
pixi add rattler-build rattler-index
mkdir -p src recipe
```

```
# src/pyproject.toml
[project]
name = "hola-pichi"
version = "1.0.0"
[project.scripts]
hola-pichi = "hola_pichi:main"
[build-system]
requires = ["setuptools"]
build-backend = "setuptools.build_meta"
```

Fase A — La receta `rattler-build` (2/3)

```
# recipe/recipe.yaml
package:
  name: hola-pichi
  version: "1.0.0"
source:
  path: ../src
build:
  number: 0
  noarch: python
  script: python -m pip install . --no-deps --no-build-isolation -vv
requirements:
  host: [python, pip]
  run: [python]
tests:
  - script: [hola-pichi]
```

Fase A — Construir el paquete (3/3)

```
pixi run rattler-build build --recipe ./recipe/recipe.yaml -c conda-forge
ls output/noarch/*.conda
```

Checkpoint: debe existir un fichero `output/noarch/hola-pichi-1.0.0-*.conda`.

Esta receta usa exactamente el mismo formato (`recipe-format` de prefix.dev) que genera Vinca automáticamente para cada paquete ROS 2.

Fase B — Publicar en prefix.dev

```
export PREFIX_API_KEY="pfx_..."

pixi run rattler-build upload prefix \
  --channel workshop-pichi-<nombre> \
  --skip-existing \
  output/noarch/*.conda
```

Checkpoint: el paquete `hola-pichi` aparece listado en el canal, en prefix.dev.

Canal público vs. privado

Plan free: 1 canal privado disponible. Usamos uno público hoy por simplicidad — la gestión de permisos de canales privados no se cubre en este taller.

Fase C — Instalar desde un pixi.toml propio

```
# pixi.toml
[workspace]
name = "consumidor_ws"
channels = ["https://prefix.dev/workshop-pichi-<nombre>", "conda-forge"]
platforms = ["linux-64"]

[dependencies]
hola-pichi = "*"
```

```
pixi install
pixi run hola-pichi
```

Troubleshooting: si no aparece (`No candidates found`) → `pixi clean cache --repodata -y`

Fase D — Integrarlo en un workspace ROS 2

```
[workspace]
channels = ["https://prefix.dev/workshop-pichi-<nombre>",
            "robostack-jazzy", "conda-forge"]
platforms = ["linux-64"]
[dependencies]
ros-jazzy-ros-base = "*"
colcon-common-extensions = "*"
hola-pichi = "*"
[tasks]
build = "colcon build --symlink-install"
[activation]
scripts = ["install/setup.sh"]
```

```
pixi install && pixi shell
hola-pichi
```

Habéis recorrido el mismo camino que una build farm real.

Receta → rattler-build → prefix.dev → `pixi.toml` → mezclado con ROS 2 oficial de RoboStack. Todo con lockfile, sin `sudo`, sin Docker, multiplataforma.

Y así lo hacemos de verdad, en producción

`IntelligentRoboticsLabs`, para distribuir
PlanSys2 y EasyNav como paquetes ROS reproducibles.

El patrón: forkear, no reinventar

`ros-rolling` y `ros-jazzy` son **forks directos** de `RoboStack/ros-rolling` y `RoboStack/ros-jazzy`.

Reutilizan el 100 % del motor (Pixi + Vinca + rattler-build + CI + patches) que acabáis de usar en pequeño. Lo único que añaden: **un fichero que dice qué construir**.

plansys2_subset/vinca.yaml

```yaml
ros_distro: rolling
conda_index:
  - robostack.yaml
  - packages-ignore.yaml

# Los paquetes ya en robostack-rolling se dan por existentes
skip_existing:
  - https://prefix.dev/robostack-rolling/

# Solo estos paquetes (sus dependencias se resuelven solas)
packages_select_by_deps:
  - plansys2_bringup
  - plansys2_core
  - plansys2_domain_expert
  - plansys2_executor
  # ... resto de paquetes plansys2_*
  - popf
  - rclcpp_cascade_lifecycle
```

El atajo, y el flujo manual detrás

```
cd ros-rolling
pixi install
pixi run plansys2-all
```

Por debajo (idéntico al runbook genérico, apuntando al subset con `-d`):

```
pixi run vinca -d ./plansys2_subset --platform linux-64 -m
pixi run rattler-build build --recipe-dir ./recipes \
  -c https://prefix.dev/robostack-rolling --skip-existing
pixi run rattler-index fs ./output --force
pixi run rattler-build upload prefix --channel irl-rolling ...
```

Cómo lo consume alguien

```
[workspace]
name = "plansys2_ws"
channels = [
  "file:///.../ros-rolling/output",        # local, para iterar
  "https://prefix.dev/irl-rolling",         # canal propio
  "https://prefix.dev/robostack-rolling",   # ROS 2 oficial
  "https://prefix.dev/conda-forge",
]
platforms = ["linux-64"]

[dependencies]
ros-rolling-ros-base = "*"
ros-rolling-plansys2-bringup = "*"
ros-rolling-plansys2-core = "*"
ros-rolling-popf = "*"
```

El orden importa: local → canal propio → RoboStack oficial → conda-forge.

Esto funciona de verdad

- PlanSys2 + simulación de Nav2, 4 nodos coordinados (`rmw_zenohd`, `nav2_sim_node`, launch con árbol de comportamiento, controlador).
- Reproducible con `pixi install` + `pixi run build`.

Aviso honesto

La publicación a prefix.dev es **manual/local** — no está automatizada en ningún workflow de CI comprometido en el repo.

Para empresas e instituciones

Si tenéis software propio que queréis distribuir
como paquetes ROS reproducibles:

1. Fork del repo de RoboStack de vuestra distro.
2. Vuestro propio `<algo>_subset/vinca.yaml` seleccionando qué construir.
3. Reutilizad el resto de la infraestructura tal cual.

No hace falta reinventar nada de esto.

Reto final (si sobra tiempo)

- Tomad `mi_paquete` del bloque 5 (el nodo ROS 2 propio).
- Convertidlo en una receta `rattler-build` (dependiendo de `ros-jazzy-rclpy`).
- Publicadlo en vuestro mismo canal.

Cerrando el círculo completo con un paquete ROS real.

1. Introducción: ¿por qué ROS 2 necesita algo mejor?
2. ¿Qué es Pixi?
3. Instalación y uso básico de Pixi
4. ¿Qué es RoboStack?
5. Uso básico de RoboStack para compilar un workspace ROS 2
6. Infraestructura de RoboStack
7. Tu propia Buildfarm basada en RoboStack

Resumen de timing

Bloque	Duración	Tipo
1. Introducción	10 min	Teoría
2. ¿Qué es Pixi?	30 min	Teoría
3. Instalación y uso básico de Pixi	30 min	Práctica
4. ¿Qué es RoboStack?	15 min	Teoría
5. RoboStack + colcon	20 min*	Práctica
6. Infraestructura de RoboStack	15 min	Teoría
7. Tu propia Buildfarm	35 min	Práctica
Núcleo	**155 min**	
Total esperado	**~165-170 min (~3h)**	

* + 10-12 min del ejercicio Nav2 + `pixi-ros` (demo del instructor, build lanzado de antemano). Margen restante hasta las 3h: ejercicio de varias distros ROS 2 y reto final del bloque 7, ambos opcionales, más preguntas.

Recursos

- Pixi: `https://pixi.sh/` — `https://prefix.dev/`
- RoboStack: `https://robostack.github.io/` — `https://github.com/RoboStack`
- pixi-ros: `https://github.com/ruben-arts/pixi-ros` — Nav2: `https://github.com/ros-navigation/navigation2`
- Buildfarm propia: `https://github.com/IntelligentRoboticsLabs/ros-rolling` — `https://github.com/IntelligentRoboticsLabs/ros-jazzy`
- Paper original: Fischer et al., *"A RoboStack Tutorial"*, IEEE RAM 2021 (arXiv:2104.12910)

Preguntas