
ROSCon ES
'25
BARCELONA

ROS 2: Una Guía Práctica de Supervivencia

Xavier Ruiz y Jesús Silva
Ingenieros de software en robótica
Ekumen

Índice

Introducción

- Docker
- Comandos frecuentes
- Motivación

Módulos

1. Herramientas de Análisis Estático
2. Unit Testing
3. ROS Unit Testing
4. Integration Testing
5. End-To-End Testing
6. Continuous Integration (CI)

Docker

* **Descarga** el repositorio.

```
git clone https://github.com/Ekumen-OS/ros2_testing_workshop_roscon_es_25.git
```

* **Descarga** la imagen de dockerhub.

```
docker pull ekumenlabs/ros2-testing-workshop-roscon-es-25:jazzy
```

Alternativamente, haz un **build** de la imagen docker localmente.

```
./docker/build.sh
```

* Haz un **run** del contenedor o abre una **nueva terminal** dentro del que ya existente.

```
./docker/run.sh
./docker/join.sh
```

Comandos frecuentes

- **Compila el código** del paquete indicado y sus dependencias necesarias.

```
colcon build --packages-up-to module_x --event-handlers console_direct+
```

- **Actualiza el entorno** con el código recién compilado.

```
source install/setup.bash
```

- **Ejecuta solo los tests** del paquete seleccionado.

```
colcon test --packages-select module_x --event-handlers console_direct+
```

- **Muestra los resultados detallados** de los tests (indica si los tests pasan o fallan).

```
colcon test-result --verbose
```

Motivación del Workshop

- Los proyectos ROS son **complejos**: combinan algoritmos, *drivers*, *middleware* e interfaces de hardware.
- ¿Efecto demo? Faltan tests.
- Las **tests** ofrecen **feedback rápido y repetible**, detectando errores antes y con menor coste.

BENEFICIOS
- Confianza en el cambio.
- Mejor diseño.
- Prevención de errores.
- Documentación viva.
- Colaboración segura.
- Sinergia con CI.

COSTES
- Desarrollo
- Mantenimiento

Módulo 1: Herramientas de Análisis Estático

Motivación

- Forzar un **estilo único** en el código.
- **Prevenir errores simples** como punteros nulos, que pueden aparecer por pequeños detalles.
- **Automatizar** la parte tediosa en la **revisión de código**, pudiendo centrarte en el diseño y la lógica.

Comparación de herramientas

- No todas las herramientas son iguales.
- Se dividen en **3 grupos**:
  - **Formateadores**: solucionan problema de estilo en el código. Por ej: *clang-format*, *uncrustify*.
  - **Linters**: señalan violaciones de estilo. Por ej: *cpplint*, *flake8*.
  - **Analizadores estáticos**: encuentran errores potenciales sin tener que ejecutar el código. Por ej: *cppcheck*, *clang-tidy*.

Integración con ROS 2

- Configurar varias de estas herramientas puede ser muy tedioso.
- Por suerte, la comunidad de ROS 2 hizo esto más fácil gracias a un paquete: *ament_lint_common*
- También ofrecen **paquetes individuales** para las distintas herramientas.
- Basta con añadirlos como **dependencia** para los tests.

NOTA IMPORTANTE

Los paquetes usados como herramientas de línea de comandos son aquellos con solo *ament_* de prefijo (*ament_clang_format*).

Sin embargo, aquellos usados para el chequeo automático usando tests son aquellos con *ament_cmake_* de prefijo (*ament_cmake_clang_format*).

Es crítico **no confundirlos**.

Problemas

- *ament_lint_common* provee **configuraciones por defecto** para las distintas herramientas.
- También es posible usar **tu propia configuración**.
- Hay ocasiones en las que pueden aparecer **falsos positivos**.

.clang-format

```
Language: Cpp
BasedOnStyle: Google
IndentWidth: 2
ColumnLimit: 100
```

```cpp
#include <vector>

void processMatrix() {
    // clang-format might try to wrap this or stack it
vertically,
    // making it hard to read as a matrix.

    // clang-format off
    const std::vector<std::vector<double>>
transformation_matrix = {
        { 1.0, 0.0, 0.0, 0.5 }, // Row 1: Translation X
        { 0.0, 1.0, 0.0, 0.2 }, // Row 2: Translation Y
        { 0.0, 0.0, 1.0, 0.0 }, // Row 3: Translation Z
        { 0.0, 0.0, 0.0, 1.0 }  // Row 4: Homogeneous
    };
    // clang-format on

    // ... code that uses the matrix ...
    // Clang-format will resume formatting normally
from here.
}
```

Recomendaciones

- Es importante elegir el **set de herramientas adecuado** para tu caso, para evitar conflictos.
- Para **C++**, por ejemplo:
  - Elegir un **formateador único**, el más usado es *clang-format*.
  - Combinar con un **linter** que te dé una **capa más de seguridad**, por ejemplo *cpplint*.
  - Si se está usando **código en producción**, añadir un **analizador estático** que ayude a detectar errores de forma preventiva, como *clang-tidy*.

Otras herramientas

- **sanitizers**
  - Encuentran errores mientras los tests corren.
  - 2 tipos más comunes: Address Sanitizers (*ASan*) y Thread Sanitizers (*TSan*).
- ***colcon lint***:
  - Revisa tu *CMakeLists.txt* y *package.xml*.
  - Señala dependencias explícitas que faltan.
- ***ros2 doctor***
  - Revisa la instalación de ROS 2 y todo el entorno.
  - Resuelve el típico "A mi me funciona".

PRE-COMMIT

En el día a día del desarrollador, aplicar todas estas herramientas puede llegar a ser muy **tedioso**.

Para aliviar esa carga, se puede configurar **pre-commit** para ejecutar **localmente** todos los chequeos que sean necesarios.

De esta forma, **cada commit introducido** es forzado con el estilo predefinido.

Ejercicio 1

Objetivo: Detectar y solucionar problemas en el código con las herramientas mostradas.

Tarea 1: Construye el paquete para el Módulo 1 y ejecuta los tests.

Tarea 2: Revisa los problemas mostrados en los tests, y solucionalos formateando el código o a mano.

Éxito: Los tests muestran que no hay **ningún error** en el código ni en el formato.

Seguir instrucciones en el **README**

Ejercicio 2

Objetivo: Identificar problemas de dependencias con *colcon lint*.

Tarea 1: Ejecuta *colcon lint* con el paquete del Módulo 1. ¿Qué indicaciones observas?

Tarea 2: Soluciona las indicaciones sugeridas por la herramienta.

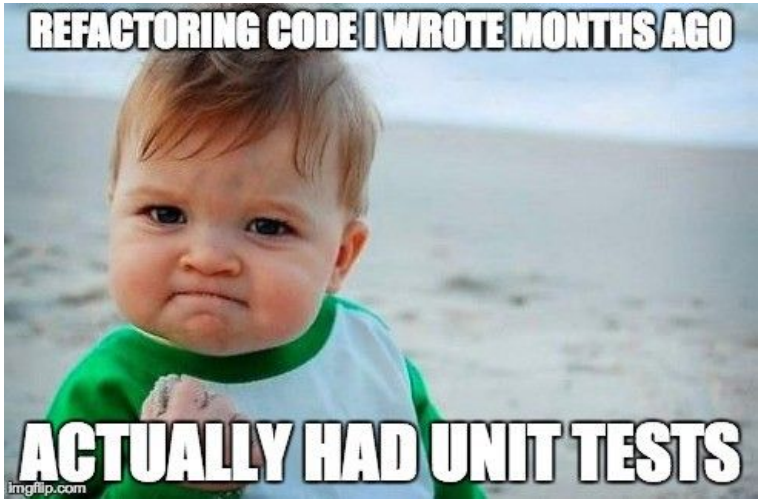
Éxito: La salida del comando de *colcon lint* no muestra **ninguna sugerencia**.

Seguir instrucciones en el **README**

Módulo 2: Unit Testing

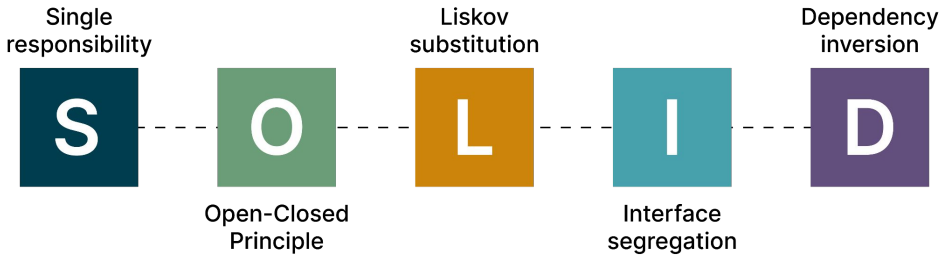
Motivación

- Los nodos **combinan lógica algorítmica** y **comunicación ROS** → complica los tests.
- Los **tests unitarios** aíslan la lógica del sistema y eliminan la dependencia del middleware: ejecución **rápida** y resultados **deterministas**.
- Permiten **refactorizar** con confianza y **detectar errores** antes de integrar.
- Permiten que los tests de **integración** se centren en la **comunicación entre nodos**.



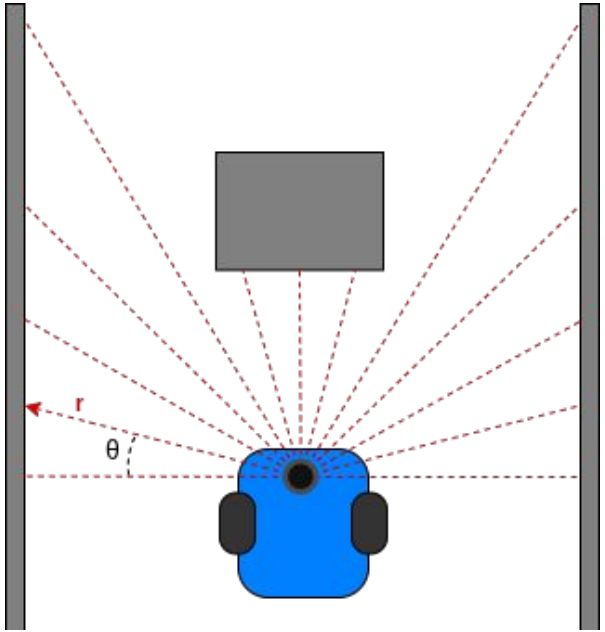
Diseño Testable

- Código bien **aislado** con **entradas y salidas bien definidas** y **mínimas dependencias** ocultas.
- **No requiere** que el **sistema completo** esté en ejecución para ser validado.
- Principios **SOLID** (especialmente S y D):



Ejercicio 1

* Un **LiDAR** escanea el entorno midiendo **distancias** con haces láser en distintos **ángulos**.
* Los valores obtenidos forman un mapa 2D del entorno, útil para **detectar obstáculos** y **planificar movimientos seguros**.



Ejercicio 1

Objetivo: Analizar el nodo *bad_laser_detector.cpp* y evaluar su testabilidad.

Tarea 1: Examinar el código y comprender su funcionamiento.

Tarea 2: Identificar los elementos que dificultan los tests y pensar qué cambios permitirían hacerlo testable. ¿Sigue los principios SOLID?

Éxito: Se puede explicar con claridad por qué el archivo es o no es testable e identificar los puntos que dificultan hacer tests unitarios.

Seguir instrucciones en el **README**

GoogleTest

- **Framework estándar** de C++ para definir tests de forma clara y expresiva.
- Se **integra** fácilmente con **ROS 2** y con sistemas de **CI** (Continuous Integration).
- Usa **macros** como *TEST*, *EXPECT_EQ*, *ASSERT_TRUE* para definir **verificaciones**.

CONCEPTOS CLAVE

EXPECT vs ***ASSERT***

- *EXPECT_** registra un fallo pero continúa el test.
- *ASSERT_** detiene el test si la condición falla.

Fixtures: permiten reutilizar código entre múltiples tests (*TEST_F*).

Parameterized tests: ejecutan la misma lógica con diferentes entradas (*TEST_P*).

GoogleTest

* Hay muchas **Macros** y **Assertions** que se pueden usar según convenga. Ejemplos sencillos:

```
#include <gtest/gtest.h>

TEST(TestBasicMath, Addition)
{
  ASSERT_EQ(4, 2 + 2);
}
```

```
std::vector<int> vec = get_vector();
// stop the test if size != 3
ASSERT_EQ(3u, vec.size());
// Now it is safe to check contents
EXPECT_EQ(10, vec[0]);
EXPECT_EQ(20, vec[1]);
EXPECT_EQ(30, vec[2]);
```

GoogleMock

- Framework que complementa GoogleTest para crear **mocks** (objetos simulados).
- Permite probar la lógica del algoritmo de forma **aislada inyectando entradas** controladas (simulando sensores, errores, etc.).
- Facilita la ejecución de tests unitarios **sin depender de hardware** o infraestructura física, acelerando el desarrollo.

Nota: Ejemplos y ejercicios con gmock quedan fuera del scope del workshop.

Integración con ament

- ROS 2 integra **GoogleTest** y **GoogleMock** a través de funciones **CMake** específicas, lo que permite compilar y ejecutar las pruebas fácilmente con *colcon test*.

package.xml

```
<test_depend>ament_cmake_gtest</test_depend>
<!-- add ament_cmake_gmock only if using gmock -->
<test_depend>ament_cmake_gmock</test_depend>
```

colcon

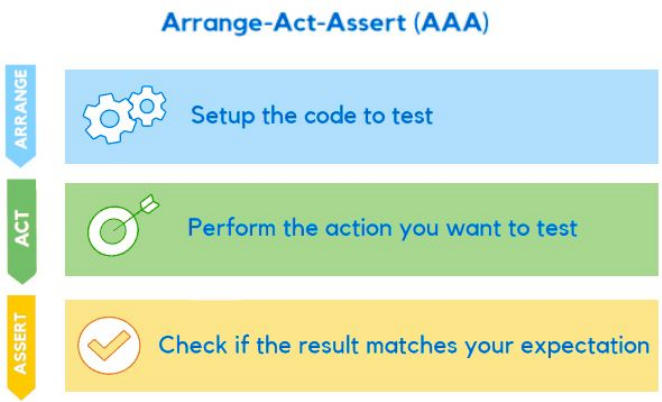
```
colcon build --packages-select module_2
colcon test  --packages-select module_2
colcon test-result --verbose
```

CMakeLists.txt

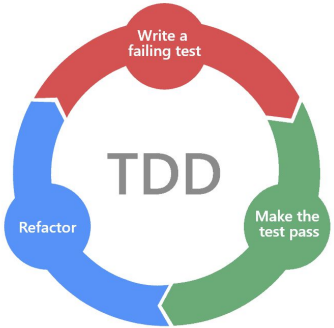
```
if(BUILD_TESTING)
  find_package(ament_cmake_gtest REQUIRED)
  # Define test executable
  ament_add_gtest(test_module_2_algorithm test/test_algorithm.cpp)
  # Link against internal libraries
  target_link_libraries(test_module_2_algorithm algorithm)
  # Link against external libraries
  ament_target_dependencies(test_module_2_algorithm Eigen3)
  # Same for mocks
endif()
```

¿Cómo escribir tests?

- **AAA**: Patrón para escribir tests de forma clara y concisa.



- **TDD**: workflow de desarrollo donde los tests se escriben antes que el código. Proceso cíclico.



Takeaway: La testabilidad debe guiar el diseño del código, sin importar si haces los tests antes o después.

Ejercicio 2

Objetivo: Completar la versión refactorizada del Ejercicio 1 para que el algoritmo sea testable.

Tarea 1: Analizar cómo la clase *LaserDetector*, definida en *laser_detector.cpp*, resuelve los problemas de testabilidad detectados en el Ejercicio 1.

Tarea 2: Completar los bloques *BEGIN EDIT / END EDIT de* la clase *LaserDetector* para que el algoritmo testable pase los tests definidos en *test_laser_detector.cpp* (enfoque TDD). Concretamente:

- Constructor
- *points_inside_footprint*
- *detect_obstacle*

Éxito: Los test definidos en *test_laser_detector.cpp* pasan mostrando **0 errores**.

Módulo 3: ROS Unit Testing

Motivación

- El algoritmo ya ha sido validado con **tests unitarios** (módulo anterior).
- Siguiente paso: crear un **nodo ROS 2** que lo envuelva y lo conecte con otros componentes en el sistema.
- **ROS unit tests** validan la interfaz del nodo (por ejemplo suscripción y publicación de mensajes)
- Sirve de **puente entre validación del algoritmo e integración**.

Test Fixtures

- Un **test fixture** (GoogleTest) define un **entorno común** compartido por varios tests.
- Gestiona la **preparación (setup)** y la **limpieza (teardown)** antes y después de cada test.
- Se crea derivando una clase de *testing::Test*.

Gestión del **ciclo de vida:**

- ***SetUpTestCase / TearDownTestCase***: Se ejecutan una vez por suite (ideales para *rclcpp::init()* y *rclcpp::shutdown()*).
- ***SetUp / TearDown***: Se ejecutan antes y después de cada test, para crear o destruir recursos como nodos, publicadores, ...

```
class TestMyClass : public ::testing::Test
{
public:
  static void SetUpTestCase()
  {
    rclcpp::init(0, nullptr);
  }

  static void TearDownTestCase()
  {
    rclcpp::shutdown();
  }

protected:
  const rclcpp::NodeOptions default_node_options;
};
```

Parámetros

- Los **parámetros** configuran el nodo y pueden probarse directamente con *rclcpp::NodeOptions*.
- Sobreescribir dichos parámetros nos permite verificar que estén **declarados, propagados e inicializados correctamente** antes de que el nodo se ejecute.

> **TIP:** Definir los nombres y valores por defecto de los parámetros como **constantes** dentro de la clase ROS 2.

```
TEST_F(TestMyClass, ParameterOverride)
{
  rclcpp::NodeOptions custom_node_options;
  custom_node_options.append_parameter_override("my_param", false);

  const MyClass dut(custom_node_options);

  ASSERT_TRUE(dut.has_parameter("my_param"));
  ASSERT_EQ(false, dut.get_parameter("my_param").as_bool());
}
```

Tópicos

- Los **publicadores** y **suscriptores** pueden verificarse con *get_topic_names_and_types()*, confirmando que los **tópicos estén declarados y registrados correctamente** en el grafo de ROS.
- Esto asegura que el nodo **asocia cada tópico con el tipo de mensaje esperado**, garantizando una **interfaz bien definida**.

```
TEST_F(TestMyClass, TopicRegistration)
{
  const MyClass dut(default_node_options);

  const auto topic_map = dut.get_topic_names_and_types();
  const std::string topic_name = "topic_name";
  const std::string expected_type = "std_msgs/msg/String";

  ASSERT_TRUE(topic_map.find(topic_name) != topic_map.end());
  ASSERT_FALSE(topic_map.at(topic_name).empty());
  ASSERT_EQ(expected_type, topic_map.at(topic_name)[0]);
}
```

Servicios

- De forma similar a los tópicos, los **servicios** pueden verificarse con *get_service_names_and_types()*, comprobando que el nodo los **registre con los nombres y tipos esperados** en el grafo de ROS.

```
TEST_F(TestMyClass, ServiceRegistration)
{
  const MyClass dut(default_node_options);

  const auto service_map = dut.get_service_names_and_types();
  const std::string service_name = "reset_service";
  const std::string expected_type = "std_srvs/srv/Empty";

  ASSERT_TRUE(service_map.find(service_name) != service_map.end());
  ASSERT_FALSE(service_map.at(service_name).empty());
  ASSERT_EQ(expected_type, service_map.at(service_name)[0]);
}
```

Test Pipeline de un Nodo

- Las **pruebas de pipeline** verifican que los **datos fluyan correctamente** desde la suscripción de entrada hasta la publicación de salida.
- Simulan el comportamiento completo del nodo **sin depender de procesos externos**.

Configuración típica:

- Nodo bajo prueba (**DUT**).
- **Nodo de prueba** que publica y suscribe.
- **Executor** que gestiona ambos nodos.
- Utilidad de **wait/spin** determinista.

Determinismo es clave. Buenas prácticas:

- **Evitar sleeps arbitrarios**. Preferible esperas basadas en eventos.
- Controlar executor con *spin_some()* hasta cumplir una condición.
- Usar *spin_until_future_complete()* en operaciones bloqueantes.
- Activar *use_sim_time=true* y publicar */clock* para control temporal.

El **ejercicio** de este módulo sirve como ejemplo de un test pipeline.

Alternativa: Rtest

- **Rtest** es un framework emergente que permite ejecutar tests **deterministas** en ROS 2.
- Permite **mockear e inspeccionar** entidades ROS 2 (*publishers*, *subscribers*, *services*, *timers*, *actions*), **inyectar mensajes**, **disparar timers** y **controlar el tiempo simulado** sin usar un *executor*.
- **Limitación:** sólo puede probar componentes cuyo **código fuente esté en el workspace** (no paquetes precompilados).
- Aun así, tiene **gran potencial** como capa unificada y determinista que complementa a GoogleTest.

Ejercicio 1

Objetivo: Corregir y completar el nodo *LaserDetectorNode* y sus tests para que compile, se ejecute de forma determinista y pase todos los tests con éxito.

Tarea 1: Revisar el código fuente y corregir errores en parámetros o tópicos del nodo.

Tarea 2: Completar los tests en los bloques *BEGIN EDIT / END EDIT* con las aserciones correctas.

Tarea 3: Usar las funciones *spin_until* y *make_scan* para una ejecución determinista.

Éxito: Todos los tests pasan (**0 errores**), validando parámetros, tópicos y detección correcta de obstáculos.

Tests Aislados

- En tests unitarios ROS 2, pueden ocurrir **interferencias entre tests** (*cross-talk*).
- Por defecto, los tests se ejecutan **en paralelo**, y si varios usan **los mismos nombres de tópicos**, los nodos pueden **cruzar mensajes entre sí**, provocando resultados **no deterministas o fallos inesperados**.
- Esto ocurre porque todos los nodos comparten el mismo ***ROS_DOMAIN_ID = 0*** dentro del espacio DDS.

¿Cómo podemos evitar esta interferencia entre tests y aislarlos?

Tests Aislados - Solución

- Ejecución **secuencial** usando *colcon test --executor sequential*. Evita interferencias, pero **incrementa el tiempo total**.
- **Aislar el dominio** asignando un ***ROS_DOMAIN_ID* único** a cada test, aislando la comunicación y garantizando resultados deterministas. Activar con *ament_cmake_ros*:

package.xml

```
<test_depend>ament_cmake_ros</test_depend>
```

CMakeLists.txt

```
find_package(ament_cmake_ros REQUIRED)
ament_add_ros_isolated_gtest(test_my_node test/test_my_node.cpp
```

Ejercicio 2 (Opcional)

Objetivo: Configurar el paquete para que sus **tests se ejecuten de forma aislada**, evitando interferencias con otros nodos o tests.

Tarea 1: Actualizar los archivos *CMakeLists.txt* y *package.xml* para habilitar el aislamiento de tests.

Tarea 2: Volver a compilar y ejecutar los tests.

Éxito: Los tests se ejecutan correctamente y la consola muestra:

```
Running with ROS_DOMAIN_ID X
```

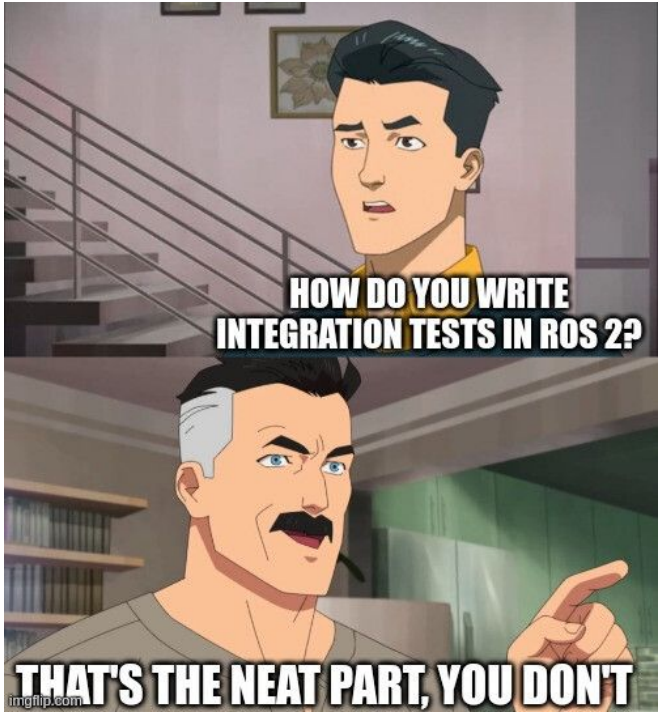
Confirmando que cada test se ejecuta **en aislamiento** con un dominio único.

Seguir instrucciones en el **README**

Módulo 4: Integration Testing

Motivación

- Los **tests unitarios** prueban un **único nodo** en aislamiento como mucho.
- Pero, ¿qué pasa cuando queremos probar la **interacción con otros nodos**, pero sin llegar a usar todo nuestros componentes?
- Ahí es donde entran los **tests de integración**, probando la **comunicación y trabajo en equipo** de varios nodos.
- Son los eternos olvidados en la pirámide de tests.



Launch_testing

- El **framework base** usado para tests de integración.
- Usa *unittest* por defecto (también existe *launch_pytest*).
- El **concepto clave** es crear un script de Python que al mismo tiempo sea:
  - Un archivo launch de ROS 2.
  - Un script de test en Python.
- Tests en Python, pero es **posible testear cualquier nodo**.
- Los **nodos** se consideran **cajas negras**, sólo se chequean las entradas/salidas (topics, servicios, logs).

Componentes

* 2 componentes principales en estos tests: *generate_test_description()* y una clase que herede de *unittest.TestCase*.
* ***generate_test_description()***:
  - Sirve como **punto de entrada**, de la misma forma que *generate_launch_description()* en un archivo launch típico.
  - La última acción realizada debe ser *launch_testing.actions.ReadyToTest()*, para indicar al framework que todo está listo para lanzar los tests.
  - El resto de acciones puede ser muy variado.

Componentes

- ***unittest.TestCase:***
  - En esta clase es **donde los tests son añadidos**.
  - Distintas formas de comprobar el resultado del test:
    - La manera simple es usando logs generados por el nodo.
    - La manera más robusta es usando un nodo temporal de ROS, e interactuando con la API.

> **Nota:** *unittest* crea una instancia de la clase por cada método que encuentra con el prefijo *test_*.

Ejemplo

```python
import unittest
import launch
import launch_testing
from launch_ros.actions import Node

# The "launch" part: This generates the launch description.
def generate_test_description():

    # Launch the node you want to test
    talker_node = Node(
        package='my_package',
        executable='talker_node',
        name='my_talker'
    )

    # The "ReadyToTest" action signals that launch is complete
    # and the test "body" can start.
    return launch.LaunchDescription([
        talker_node,
        launch_testing.actions.ReadyToTest(),
    ])
```

```python
# The "testing" part: This is a standard Python unittest class.
class TestTalkerIntegration(unittest.TestCase):

    def test_talker_startup(self, proc_output):
        # This test checks the stdout of all launched processes.
        # We wait to see the "node started" log message.
        proc_output.assertWaitFor(
            b'Talker node has started.',
            timeout=5
        )

    # You can add more test_ methods here
    def test_node_publishes(self, proc_output):
        # In a real test, you might create a temporary
        # rclpy node here to subscribe and check for messages.
        pass
```

Integración en ROS 2

- Para añadir los tests, es necesario aplicar lo aprendido hasta ahora en el resto de módulos.

```
find_package(ament_cmake_ros REQUIRED)
find_package(launch_testing_ament_cmake REQUIRED)

# Function used to isolate the integration tests and avoid cross-
talk between them
function(add_ros_isolated_launch_test path)
    set(RUNNER "${ament_cmake_ros_DIR}/run_test_isolated.py")
    add_launch_test("${path}" RUNNER "${RUNNER}" ${ARGN})
endfunction()

add_ros_isolated_launch_test(./test_detection_launch.py)
```

```
<test_depend>ament_cmake_ros</test_depend>
<test_depend>launch</test_depend>
<test_depend>launch_ros</test_depend>
<test_depend>launch_testing</test_depend>
<test_depend>launch_testing_ament_cmake</test_depend>
```

Recomendaciones

- Con este procedimiento es muy fácil acabar con los conocidos como **“flaky tests”**.
- ¿Qué son estos tests? Son tests que funcionan... pero solo a veces.
- Esto se debe a que se generan **condiciones de carrera** entre las distintas partes del código.

Procedimiento **erróneo**:

1. Lanzar nodos.
2. Publicar un mensaje inmediatamente.
3. Revisar el resultado esperado

Procedimiento **correcto**:

1. Lanzar nodos.
2. Crear un publisher.
3. Esperar hasta que el subscriber se conecte a nuestro publisher.
4. Publicar el mensaje cuando el subscriber ya está listo.
5. Revisar el resultado esperado.

Ejercicio 1

Objetivo: Aprender cómo funcionan los tests de integración en ROS 2.

Tarea 1: El paquete del Módulo 4 contiene un *safety_light_node.cpp*, y del Módulo 3 ya hemos creado un *laser_detector_node.cpp*, así que es hora de probar si son capaces de trabajar juntos.

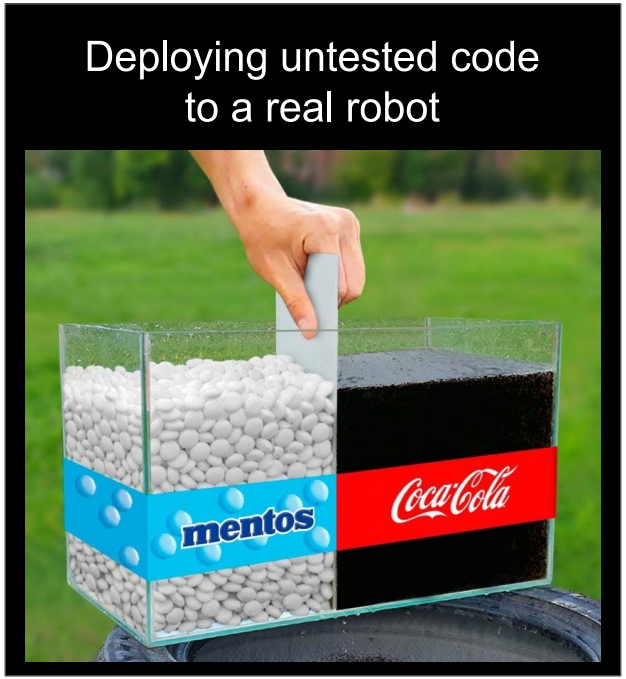
Tarea 2: En *test/test_detection_launch.py* tienes un test de integración que se encarga de lanzar los nodos, pero la lógica del test no está añadida, así que tu tarea es completarla para poder probar la comunicación entre nodos de forma robusta, evitando un "*flaky_test*".

Éxito: Se ha **completado y validado** la lógica del test de integración.

Módulo 5: End-To-End Testing

Motivación

- Responde a esa pregunta de: ¿estaré rompiendo algo con lo que he añadido?
- Usa **datos de entornos reales** (o simulados).
- Pone a prueba a **interacción entre todo el conjunto** de componentes que tiene la solución.



ROS 2 bag

- Herramienta clave para poder **recopilar información reproducible** de entornos reales.
- Funciona como **base de datos**, usando un archivo MCAP (antes de Iron, usaba sqlite3 por defecto).
- **3 comandos clave** para entender su funcionamiento:
  - *ros2 bag record*
  - *ros2 bag info*
  - *ros2 bag play*

```
# Get a summary of the bag file
ros2 bag info my_mission_bag/

# Example output:
# Files:             my_mission_bag.mcap
# Bag size:          15.8 MiB
# Storage id:        mcap
# Duration:          1m 10s
# Start:             Oct 17 2025 15:30:01.000
# End:               Oct 17 2025 15:31:11.000
# Messages:          3013
# Topic information:
   # Topic: /turtle1/cmd_vel | Type: geometry_msgs/msg/Twist |
Count: 9 | Serialization Format: cdr
   # Topic: /turtle1/pose | Type: turtlesim/msg/Pose | Count:
3004 | Serialization Format: cdr)
```

E2E Testing Manual

- Primer tipo de tests E2E que se suelen hacer.
- Un **usuario lanza el sistema**, ejecuta un rosbag y monitoriza que todo funcione como debe.
- Fases:
  1. Lanzar el sistema: *ros2 launch my_robot navigation.launch.py*
  2. Lanzar visualización: *ros2 run rviz2 rviz2 -d my_config.rviz*
  3. Proveer entradas al sistema: *ros2 bag play my_recorded_mission/*
  4. Observar y verificar.
- Muy **potente**, pero **nada automatizable**.

> **Nota:** *replay_testing* es una herramienta muy útil para tratar de automatizar este tipo de tests.

E2E Testing Con Simulación

- Segundo tipo de tests E2E, si los tests manuales fueron bien.
- En lugar de repetir datos fijos, usar un **simulador para generar nuevos datos**.
- Muy útil para:
  - Validar frente a **obstáculos dinámicos**.
  - **Probar casos extremos** (por ejemplo, el LiDAR deja de funcionar).
- Es manual todavía, pero se puede llegar a automatizar.

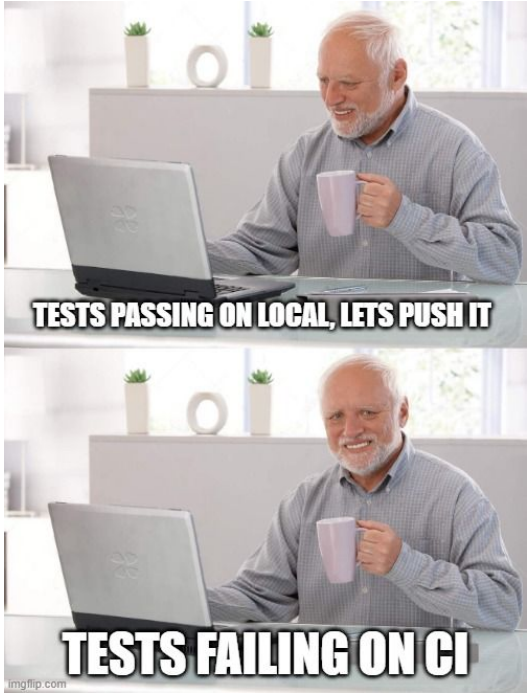
E2E Testing Automático

- Combina **todo lo que hemos aprendido** hasta ahora para generar casos que pueden ser ejecutados en un **servidor** con *colcon test*.
- También usa *launch_testing* para generar los tests, solo que esta vez combina el lanzamiento de nodos y también de otras herramientas como el simulador o el rosbag.
- En este caso, **el resultado no puede ser "observado"**, así que debe ser comprobado de forma programática (usando ASSERT).
- Es totalmente necesario alcanzar este punto si queremos integrar este tipo de tests en CI, como veremos a continuación.

Módulo 6: Continuous Integration (CI)

Motivación

- Hasta ahora, los **tests** se ejecutan manualmente en cada máquina.
- Esto puede funcionar en proyectos individuales, pero **no es fiable** en equipos o proyectos a largo plazo.
- ¿Qué pasa si...?
  - alguien **olvida ejecutar los tests** antes de subir código,
  - un cambio **falla en otro sistema operativo**,
  - o una nueva función **rompe código existente**.



¿Qué es Continuous Integration?

- Práctica de desarrollo donde los cambios se **fusionan frecuentemente** en un repositorio central.
- Tras cada cambio, se ejecuta automáticamente una **compilación y un conjunto de tests**.
- Beneficios:
  - **Garantiza calidad:** valida tests en cada *pull request*.
  - **Previene regresiones:** detecta errores y *"breaking changes"*.
  - **Automatiza tareas:** elimina pasos manuales y repetitivos.
  - **Fuente única de verdad:** el servidor CI decide si el código es integrable.

TIP: Usar *pre-commit* para optimizar el flujo de trabajo. Complementa CI y garantiza commits limpios y funcionales.

Nota: CD (Continuous Deployment) amplía la CI **automatizando el despliegue** del código validado.

Github Actions

- Plataforma CI/CD integrada en **GitHub**.
- Los *workflows* se definen en YAML dentro de *.github/workflows/*.
- Se ejecutan **automáticamente** al hacer *push*, abrir un *PR* o según una planificación.
- ROS 2 dispone de ***actions* preconfiguradas**, como [ros-tooling/action-ros-ci], que ejecuta: *colcon build*, *colcon lint*, y *colcon test* en un solo paso.

> **Nota:**
> Puedes usar *runners* propios para ejecutar la CI en tu infraestructura.
>
> Los comandos *ament_cmake* y *colcon* en CI se comportan **igual que en local**.

Ejemplo y Recomendaciones

* *.github/workflows/ci.yml*

Buenas prácticas:

- **Fijar versiones** de las *actions* para evitar riesgos de **seguridad**.
- Usar una *strategy matrix* para probar varias **distros**.
- Añadir **builds** programadas (diarias, semanales...).
- Habilitar caché (*ccache*) para acelerar compilaciones grandes.

```yaml
name: ROS 2 CI
on:
  pull_request: # Run on all pull requests
  push:
    branches: [ main ] # Also run on pushes to the main branch
jobs:
  build_and_test_ros2:
    runs-on: ubuntu-latest
    container:
      image: rostooling/setup-ros-docker:ubuntu-noble-latest
    steps:
      - name: Build and run tests
        uses: ros-tooling/action-ros-ci@v0.4
        with:
          package-name: |
            my_package_1
            ...
          target-ros2-distro: jazzy
          import-token: ${{ secrets.GITHUB_TOKEN }}
```

Visualización y Protección

- Resultados en PRs:
  -  Verde → todas las comprobaciones superadas.
  -  Rojo → fallos en *build* o *test* (ver “Details” para logs).
- Reglas de **protección de ramas**:
  - Obligan a que todas las verificaciones pasen antes de poder hacer *merge*.
  - Convierten la CI en un **filtro obligatorio de calidad**.
  - Configuración: *Settings → Branches → Add ruleset*

 A la hora de crear nuevos PRs, es recomendable diferenciar los cambios de lógica vs formato, con tal de facilitar el review.

EKUMEN

¿Preguntas?

Gracias!