Design of a ROS2-Based Control and Simulation Architecture for the Kawasaki FS06N Robot

Bjørn Remseth
Email: la3lma@gmail.com
With much AI assistance

Abstract*—This document presents a comprehensive design for a Robot Operating System 2 (ROS 2) [1] Jazzy-based control architecture and simulation environment for the Kawasaki FS06N industrial robot manipulator. The FS06N is a 6-axis robot with 6kg payload capacity, 1102mm reach, and 0.05mm repeatability. The proposed system follows best practices in ROS2 and modern software engineering, enabling testing at multiple levels (unit, integration, simulation) with seamless transition from simulation to real hardware with Kawasaki D-series controller. The architecture leverages Gazebo Classic for physics simulation, ROS2 Control (ros2_control framework) with a JointTrajectoryController, and MoveIt 2 for motion planning. The robot's model, sensors, and controllers are designed to support both mock hardware interfaces for simulation and real hardware interfaces using Kawasaki's AS language for physical deployment. Key components include scene representation, wrist-mounted camera, and depth camera simulation. The design emphasizes modularity, testability, and consistency between simulation and reality, providing a robust foundation for Kawasaki robot integration with ROS2.** ***Note: This document was created with AI assistance; see "About This Document" and "Note on References and Verification" sections for details on methodology and verification processes.

***Index Terms*—ROS2, Kawasaki FS06N, AS language, robotics, motion planning, MoveIt2, Gazebo, ros2_control, hardware abstraction, simulation**

`version: 2025-11-16-22:01:03-CET-4c6acb0-main`

I. Introduction

Modern robotic systems require sophisticated control architectures that enable rapid development, comprehensive testing, and reliable deployment. This document proposes a ROS 2-based control architecture and simulation environment specifically designed for the **Kawasaki FS06N** industrial robot manipulator [2], with focus on enabling seamless transition from simulation to physical hardware.

A. Target Robot System

The Kawasaki FS06N is a 6-axis industrial robot featuring:

- **Payload capacity**: 6 kg
- **Reach**: 1102 mm
- **Repeatability**: ±0.05 mm
- **Applications**: Assembly, pick-and-place, machine tending, inspection

The robot is controlled using Kawasaki's proprietary **AS (Kawasaki Robot Language)** programming environment [3], [4], which requires translation of ROS2 control commands into AS monitor commands for hardware deployment.

B. Design Goals

The goal is to create a system that can be tested at multiple levels—unit, integration, and simulation—before deployment to real hardware. We employ Gazebo Classic for physics simulation, ROS2 Control (specifically the ros2_control framework [5]) with a JointTrajectoryController, and MoveIt 2 [6] for motion planning. The robot's model, sensors, and controllers are defined to support both a mock hardware interface (for simulation or testing) and a real hardware interface (translating commands to AS language [7] for the physical robot).

This design addresses several critical requirements:

- **Testability**: Enable comprehensive testing in simulation before hardware deployment
- **Modularity**: Separate concerns (planning, control, hardware) for easier debugging
- **Consistency**: Maintain identical interfaces between simulation and real hardware
- **Extensibility**: Support addition of sensors, actuators, and capabilities
- **Standards compliance**: Follow ROS2 best practices and established patterns

The document provides an overview of the architecture with UML diagrams, code sketches in C++/Python, and references to relevant ROS2 best practices. It is intended for software engineers with general programming background who may not have extensive robotics experience.

II. System Overview

At a high level, the system consists of four main subsystems that work together to enable motion planning and execution, as illustrated in Figure 1.

A. Motion Planning (MoveIt 2)

MoveIt 2 plans collision-free trajectories for the robot's manipulator. The planning pipeline runs in a ROS2 node (the Move Group), which sends trajectory commands to the controller. MoveIt's planning scene maintains knowledge of obstacles and the robot's current state, enabling sophisticated motion planning algorithms including Open Motion Planning Library (OMPL) planners [8].

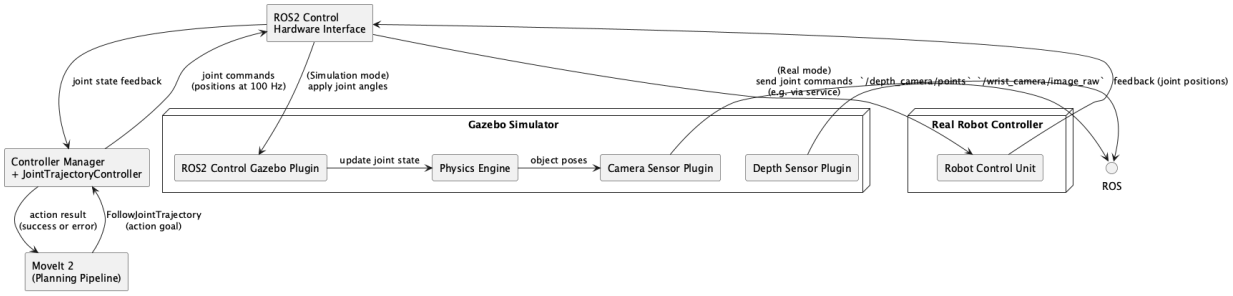

Fig. 1. *High-level architecture showing the main components: MoveIt 2 for planning, Controller Manager with JointTrajectoryController for execution, Hardware Interface abstraction, and either Gazebo simulation or real robot hardware as the backend.*

B. Controller Manager and Joint Trajectory Controller

A ROS2 Control Controller Manager hosts a JointTrajectoryController to execute trajectories on the robot's joints. The controller exposes a standard `FollowJointTrajectory` action interface used by MoveIt. It receives trajectory messages and interpolates commands for the actuators at a fixed update rate (typically 100 Hz). A joint state broadcaster publishes the current joint states on the `/joint_states` topic, essential for MoveIt and ROS Visualization (RViz) to obtain feedback of the robot's state [5].

C. Hardware Interface (ROS2 Control)

The hardware interface provides an abstraction layer that translates between the controller's generic joint commands and the specific interface to the robot's actuators. In simulation, this is a virtual hardware plugin connecting to Gazebo's physics engine. In real hardware deployment, this is a custom hardware interface that communicates with the physical robot. The interface is designed to be swappable: the same controller works with either simulated or real hardware by loading a different plugin [9].

D. Simulation Environment (Gazebo)

The robot's Unified Robot Description Format (URDF) [10]/Xacro [11] model is loaded in Gazebo Classic [12], including physics properties and ROS2 Control configuration. Gazebo, with the `gazebo_ros2_control` plugin, acts as the robot's physics simulator. The environment includes simulated sensors (wrist-mounted RGB camera and depth camera) and static objects. Sensors use Gazebo's ROS plugins to publish data to ROS2 topics, appearing identical to data from real sensors [13].

E. Visualization (RViz 2)

RViz 2 is used for development and testing to visualize the robot model, joint states, planned paths, and sensor data. The MoveIt RViz plugin shows the planning scene and allows interactive motion planning.

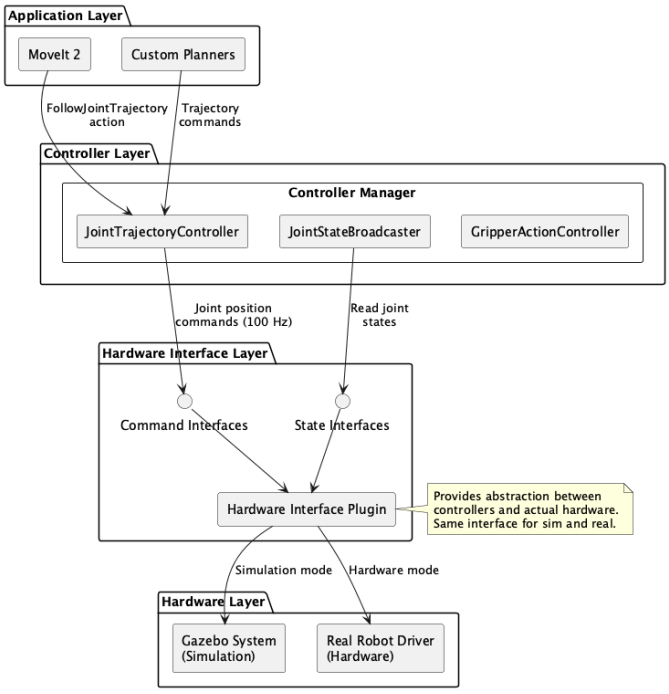

Fig. 2. *Layered architecture of the ros2_control framework, showing separation between application layer (MoveIt 2), controller layer (JointTrajectoryController, JointStateBroadcaster), hardware interface layer, and hardware layer (Gazebo or real robot).*

III. Control Architecture

A. ROS2 Control Layered Architecture

The system employs a layered architecture as shown in Figure 2, separating concerns between application logic, control algorithms, hardware abstraction, and physical hardware.

This layered approach provides several benefits:

- **Separation of concerns**: Planning, control, and hardware are independent
- **Reusability**: Controllers can be reused across different robots
- **Testability**: Each layer can be tested independently
- **Maintainability**: Changes to one layer don't affect others

B. Joint Trajectory Controller

We use the JointTrajectoryController from the `ros2_controllers` package [14] as the primary low-level controller. This controller accepts trajectory commands (type `trajectory_msgs/JointTrajectory`) and interpolates joint targets over time in a real-time loop.

The controller configuration is specified in YAML format:

Listing 1. Controller configuration in ros2_controllers.yaml

```yaml
controller_manager:
  ros__parameters:
    update_rate: 100  # Hz
    arm_controller:
      type: joint_trajectory_controller/
          JointTrajectoryController
    joint_state_broadcaster:
      type: joint_state_broadcaster/
          JointStateBroadcaster

arm_controller:
  ros__parameters:
    joints: [joint1, joint2, joint3, joint4,
        joint5, joint6]
    command_interfaces: [position]
    state_interfaces: [position]
    open_loop_control: false
    allow_integration_in_goal_trajectories: true
```

The controller manager loads these controllers at runtime, and the JointStateBroadcaster publishes joint angles on `/joint_states`, critical for RViz and MoveIt feedback.

C. Hardware Interface Abstraction

The hardware interface bridges the JointTrajectoryController (issuing high-level commands) and the mechanism applying those commands (simulated model or physical motors). ROS2 Control uses a plugin system for hardware interfaces, implemented via `hardware_interface::SystemInterface` [5].

1) The SystemInterface API: The `SystemInterface` class provides the abstract interface that all hardware plugins must implement. It defines a standardized contract between controllers and hardware, enabling the same controller code to work with different robot hardware.

Key responsibilities of SystemInterface:

- **Lifecycle management**: Initialize, configure, activate, and deactivate hardware
- **Command interfaces**: Provide writable interfaces for sending commands (e.g., position, velocity, effort)
- **State interfaces**: Provide readable interfaces for getting feedback (e.g., actual position, velocity, effort)
- **Real-time communication**: Execute `read()` and `write()` in control loop at fixed rate (typically 100-1000 Hz)

Essential methods to implement:

Listing 2. SystemInterface key virtual methods

```cpp
class MyHardwareInterface : public
    hardware_interface::SystemInterface
{
public:
  // Lifecycle callbacks
  CallbackReturn on_init(const HardwareInfo&
      info) override;
  CallbackReturn on_configure(const State&
      previous_state) override;
  CallbackReturn on_activate(const State&
      previous_state) override;
  CallbackReturn on_deactivate(const State&
      previous_state) override;

  // Real-time loop methods (called at control
      rate)
  return_type read(const Time& time, const
      Duration& period) override;
  return_type write(const Time& time, const
      Duration& period) override;

  // Interface export
  std::vector<StateInterface>
      export_state_interfaces() override;
  std::vector<CommandInterface>
      export_command_interfaces() override;
};
```

Lifecycle states: The hardware interface follows a managed lifecycle:

1) **Unconfigured** → `on_init()`: Parse URDF parameters, allocate resources
2) **Inactive** → `on_configure()`: Connect to hardware, establish communication
3) **Active** → `on_activate()`: Enable motors, start control loop
4) **Inactive** → `on_deactivate()`: Disable motors, safe state

Command and State Interfaces: These define what controllers can read/write:

Listing 3. Example interface definitions

```cpp
std::vector<StateInterface>
    export_state_interfaces() {
  std::vector<StateInterface> state_interfaces;
  for (uint i = 0; i < info_.joints.size(); i++)
      {
    state_interfaces.emplace_back(
      info_.joints[i].name, "position", &
          joint_positions_[i]);
    state_interfaces.emplace_back(
      info_.joints[i].name, "velocity", &
          joint_velocities_[i]);
  }
  return state_interfaces;
}

std::vector<CommandInterface>
    export_command_interfaces() {
  std::vector<CommandInterface>
      command_interfaces;
  for (uint i = 0; i < info_.joints.size(); i++)
      {
    command_interfaces.emplace_back(
      info_.joints[i].name, "position", &
          joint_position_commands_[i]);
  }
  return command_interfaces;
}
```

The interfaces use pointers to member variables, enabling zero-copy access in real-time loops. Controllers read from state interface pointers and write to command interface pointers.

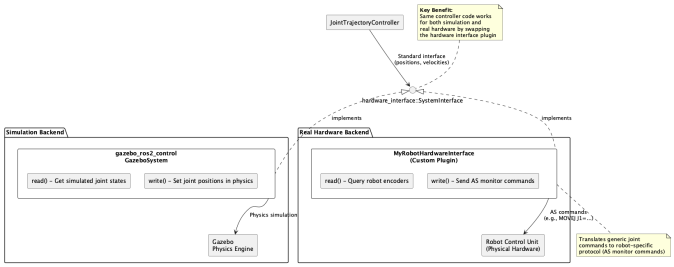

Fig. 3. *Hardware interface abstraction showing how the same JointTrajectoryController works with both Gazebo simulation (via GazeboSystem plugin) and real hardware (via custom hardware interface), providing consistent interfaces while adapting to different backends.*

Figure 3 illustrates how the same controller interfaces with different backends.

2) Simulation Hardware Interface: In simulation mode, we use the `gazebo_ros2_control` plugin, which provides a ready-made SystemInterface connecting to Gazebo's physics. When spawning the robot in Gazebo, the URDF includes a `<ros2_control>` tag specifying the Gazebo system plugin:

Listing 4. URDF ros2_control configuration for simulation

```
<ros2_control name="MyRobotSimulation" type="
    system">
  <hardware>
    <plugin>gazebo_ros2_control/GazeboSystem</
        plugin>
  </hardware>
  <joint name="joint1">
    <command_interface name="position"/>
    <state_interface name="position"/>
  </joint>
  <!-- repeat for all joints -->
</ros2_control>
```

This plugin reads joint commands from the controller and directly sets joint positions in the simulation, providing perfect feedback of joint states.

3) Real Hardware Interface: For the actual robot, we implement a custom hardware interface plugin. This component takes joint commands from ROS2 Control and sends them to the physical robot's controller. In the case of robots using proprietary command protocols (such as AS monitor commands), the hardware interface translates generic joint positions into robot-specific commands:

Listing 5. Pseudo-code for custom hardware interface write method

```
hardware_interface::return_type
MyRobotHardwareInterface::write(
    const rclcpp::Time&,
    const rclcpp::Duration&)
{
    // Get target positions for each joint
    std::vector<double> positions =
        joint_positions_;

    // Format robot-specific command
    std::string cmd = format_as_monitor_command(
        positions);

    // Send to robot (e.g., via service or
        network)
    bool success = send_command_to_robot(cmd);

    return success ?
        hardware_interface::return_type::OK :
        hardware_interface::return_type::ERROR;
}
```

The `read()` method similarly queries the robot's encoders for current joint positions, feeding back to ROS2 Control.

4) Plugin Registration and Loading: To make the custom hardware interface available to the controller manager, it must be registered as a pluginlib plugin. This requires two components:

1. Plugin export macro in source file:

Listing 6. Plugin registration macro

```
#include "pluginlib/class_list_macros.hpp"

PLUGINLIB_EXPORT_CLASS(
  kawasaki_hardware::
      KawasakiFS06NHardwareInterface,
  hardware_interface::SystemInterface)
```

2. Plugin description XML file (`kawasaki_hardware.xml`):

Listing 7. Plugin description XML

```
<library path="kawasaki_hardware">
  <class name="kawasaki_hardware/
      KawasakiFS06NHardwareInterface"
        type="kawasaki_hardware::
            KawasakiFS06NHardwareInterface"
        base_class_type="hardware_interface::
            SystemInterface">
    <description>
      Hardware interface for Kawasaki FS06N
          robot with D-series
      controller using AS language commands.
    </description>
  </class>
</library>
```

3. Export plugin in package.xml:

Listing 8. Package.xml export declaration

```
<export>
  <hardware_interface plugin="${prefix}/
      kawasaki_hardware.xml"/>
</export>
```

Loading the plugin: The URDF specifies which plugin to load:

Listing 9. URDF plugin specification for Kawasaki FS06N

```
<ros2_control name="KawasakiFS06N" type="system"
    >
  <hardware>
    <plugin>kawasaki_hardware/
        KawasakiFS06NHardwareInterface</plugin>
    <param name="robot_ip">192.168.1.10</param>
    <param name="robot_port">23</param>
    <param name="control_rate">100</param>
  </hardware>
  <joint name="joint1">
    <command_interface name="position"/>
    <state_interface name="position"/>
    <state_interface name="velocity"/>
  </joint>
  <!-- Repeat for joints 2-6 -->
</ros2_control>
```

The controller manager reads this URDF, loads the specified plugin via pluginlib, and manages its lifecycle. Custom parameters (IP address, port, rate) are accessible in `on_init()` via the `HardwareInfo` structure.

D. Sequence of Operations

Figure 4 shows the sequence of operations for trajectory execution in both simulation and real hardware modes.

From MoveIt's perspective, simulation and real hardware are indistinguishable—both provide the same action interface and adhere to the FollowJointTrajectory protocol. This design alignment with best practices ensures that 99% of the system remains identical between simulation and deployment.

IV. Robot Modeling and Simulation

A. URDF Model and ros2_control Setup

We create a detailed URDF (Unified Robot Description Format) model including:

- Links and joints with collision, visual, and inertial properties
- `<ros2_control>` elements specifying hardware interface plugins
- Transmission mappings between joints and actuators
- Joint limits (position, velocity) for safety and planning

Figure 5 illustrates how URDF and configuration files work together.

We use Xacro macros to maintain simulation and real variants, or use parameters to switch the hardware plugin. This maintains a single source of truth for robot kinematics while allowing hardware interface substitution.

B. Gazebo Classic Integration

Gazebo Classic simulates the robot in a virtual world. A Gazebo world file includes the robot (via URDF) and scene elements (table, objects, walls). The robot is spawned using ROS2 launch files with `gazebo_ros spawn_entity` functionality [15].

The `gazebo_ros2_control` plugin automatically bridges ROS2 Control interfaces into Gazebo. We run a separate controller manager in ROS2 to control the lifecycle of controllers explicitly:

Listing 10. Spawning controllers in launch file

```
ros2 run controller_manager spawner
    arm_controller
ros2 run controller_manager spawner
    joint_state_broadcaster
```

Gazebo simulates physics at high rate (e.g., 1000 Hz) while controllers run at 100 Hz, ensuring realistic timing.

C. Sensor Simulation

1) Wrist Camera: An RGB camera is attached to the end-effector link in the URDF. Using Gazebo's `gazebo_ros_camera` plugin, we define the sensor with parameters:

Listing 11. Wrist camera sensor definition

```
<sensor name="wrist_camera" type="camera">
  <camera>
    <horizontal_fov>1.047</horizontal_fov>
    <image>
      <width>640</width>
      <height>480</height>
    </image>
    <clip>
      <near>0.1</near>
      <far>5.0</far>
    </clip>
  </camera>
  <plugin name="wrist_camera_plugin"
          filename="libgazebo_ros_camera.so">
    <imageTopicName>/wrist_camera/image_raw</
        imageTopicName>
    <cameraInfoTopicName>/wrist_camera/
        camera_info</cameraInfoTopicName>
    <frameName>wrist_camera_frame</frameName>
  </plugin>
</sensor>
```

This publishes standard ROS2 messages (`sensor_msgs/Image` and `CameraInfo`) to topics matching what real cameras would use [16].

2) Depth Camera: A depth sensor (RGB-D camera like Kinect or RealSense) can be mounted on the robot or in the environment. Using Gazebo's `gazebo_ros_openni_kinect` plugin, we simulate both depth images and point clouds:

Listing 12. Depth camera plugin configuration

```
<plugin name="depth_camera_plugin"
        filename="libgazebo_ros_openni_kinect.so
            ">
  <depthImageTopicName>/depth_camera/depth/
      image_raw</depthImageTopicName>
  <pointCloudTopicName>/depth_camera/points</
      pointCloudTopicName>
  <cameraName>depth_camera</cameraName>
  <frameName>depth_camera_frame</frameName>
</plugin>
```

These sensor topics serve as placeholders for future perception integration, following standard conventions to ease later development.

D. RViz Visualization

RViz configurations visualize:

- Robot URDF model with current joint states
- Camera image streams (`/wrist_camera/image_raw`)
- Depth point clouds (`/depth_camera/points`)
- MoveIt planning scene and trajectories

The MoveIt RViz plugin enables interactive motion planning, picking target poses and visualizing planned trajectories.

V. Motion Planning with MoveIt 2

MoveIt 2 is configured using the MoveIt Setup Assistant [6], creating a `moveit_config` package containing:

- Semantic Robot Description Format (SRDF) [17] defining planning groups

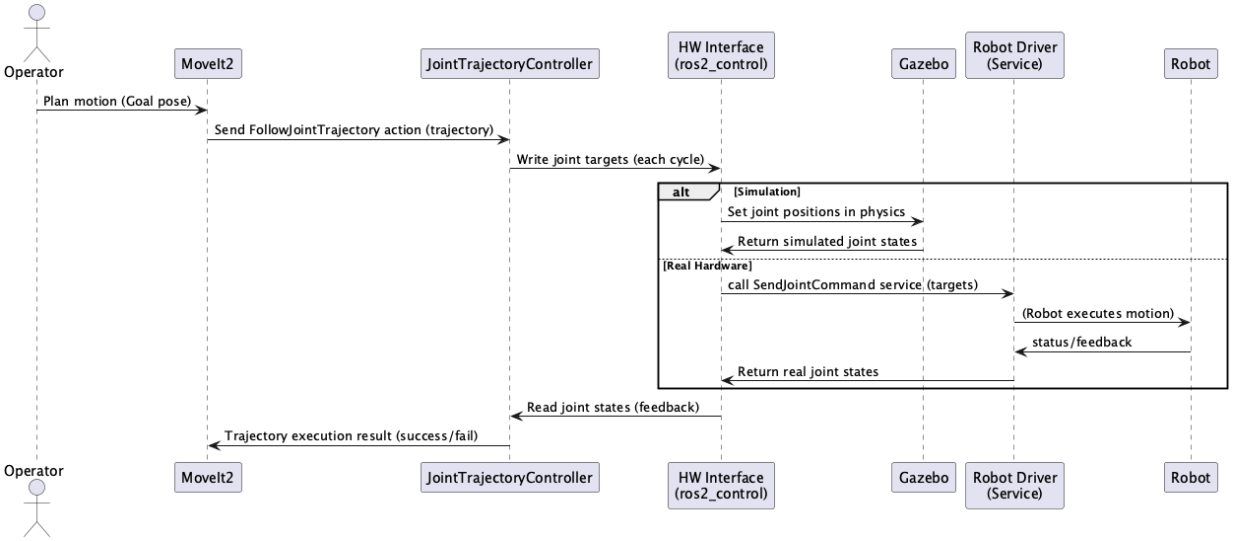

Fig. 4. *Sequence diagram showing how a trajectory command flows from MoveIt 2 through the JointTrajectoryController and hardware interface to either Gazebo simulation or real robot hardware, with feedback returning through the same path.*

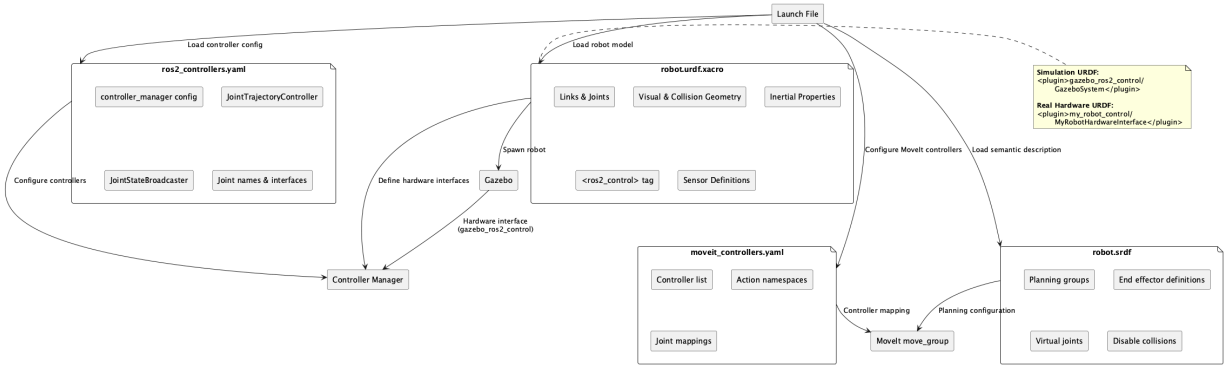

Fig. 5. *Configuration flow showing how URDF, controller YAML files, and MoveIt configuration files are loaded and connected during system startup.*

- Kinematics solver configuration (e.g., Track Inverse Kinematics (TRAC-IK))
- Controller configuration mapping to ROS2 Control
- Collision geometry from URDF

A. Controller Configuration

In MoveIt's `moveit_controllers.yaml`:

Listing 13. MoveIt controller configuration

```
controller_list:
  - name: arm_controller
    action_ns: follow_joint_trajectory
    type: FollowJointTrajectory
    joints:
      - joint1
      - joint2
      - joint3
      - joint4
      - joint5
      - joint6
```

We use the MoveItSimpleControllerManager to interface with controllers. When launching the move_group node, this configuration enables MoveIt to send trajectory commands via the action interface.

B. Integration Testing

In simulation, we verify that MoveIt can plan and execute motions (e.g., moving from "home" to target poses). The virtual robot moves in Gazebo while MoveIt RViz displays planned motion overlapping actual motion, proving correct pipeline integration.

Future enhancements can integrate perception into MoveIt's planning, using depth camera data to update the planning scene with detected obstacles.

VI. Testing Strategy and Development Process

Figure 6 illustrates the incremental development and testing approach.

A. Step 1: URDF and Basic Simulation

Build the URDF and spawn in Gazebo. Test joint movements manually, verify transforms are correct, ensure physics behaves plausibly.

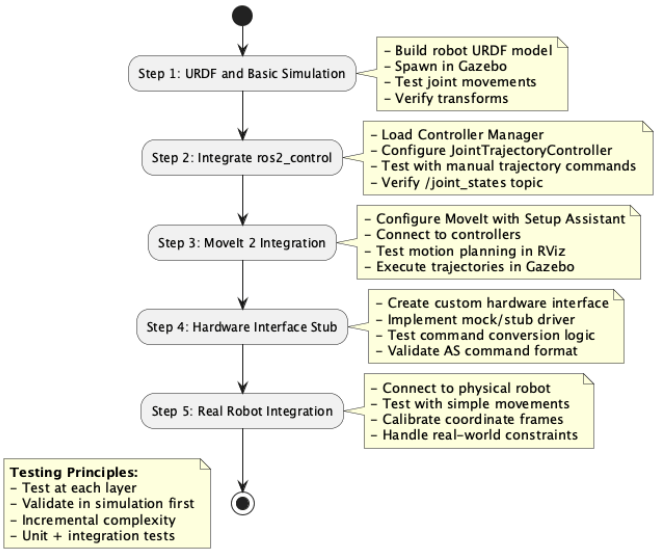

Fig. 6. *Incremental testing strategy showing five development steps from basic URDF simulation through real robot integration, with validation at each stage.*

B. Step 2: Integrate ros2_control

Load Controller Manager with JointTrajectoryController and joint state broadcaster. Validate by sending manual trajectory messages:

Listing 14. Testing controller with ros2 topic pub

```
ros2 topic pub /arm_controller/joint_trajectory
    \
    trajectory_msgs/JointTrajectory \
    '{...}' --once
```

Verify `/joint_states` publishes correctly. Use `ros2 control` CLI tools to inspect controllers and hardware interfaces.

C. Step 3: MoveIt 2 Integration

Start move_group node with RViz. Conduct motion planning tests: set target poses, let MoveIt plan and execute. Observe robot following trajectories in Gazebo. This provides system-level validation of the complete pipeline.

D. Step 4: Hardware Interface Stub Testing

Create custom hardware interface with stub/mock driver. Test command conversion logic without actual hardware:

- Verify AS command format generation
- Test joint limit handling
- Validate error handling

Unit tests capture and verify service calls match expected values.

E. Step 5: Real Robot Integration

Connect to physical robot cautiously (single joint, low speeds initially). Replace stub with actual driver implementing robot communication. Refine hardware interface for:

- Coordinate frame calibration
- Unit conversions (radians vs. degrees/encoder ticks)
- Latency and dynamics handling
- Safety constraints

Test progressively more complex trajectories. The validated simulation provides confidence that issues are hardware-specific rather than algorithmic.

VII. Design Principles and Best Practices

A. Modularity

Each component (planning, control, hardware, sensors) is modular, enabling:

- Testing components in isolation
- Swapping implementations without affecting other parts
- Following Single Responsibility Principle

B. Consistency Between Simulation and Reality

Using identical ROS interfaces and common code reduces simulation-vs-hardware divergence. Only the hardware plugin differs; configuration remains the same. This ensures simulation tests are meaningful for real deployment, without conditionals for ”if sim / if real.”

C. Safety and Error Handling

Real hardware requires safety mechanisms absent in simulation:

- Stopping robot if commands fail or communication is lost
- Checking joint limits and velocities
- Handling motor faults and sensor noise
- Using ROS2 Control’s error handling (`return_type::ERROR`)
- Implementing diagnostics (`diagnostic_updater`)

D. Extensibility

The design supports future additions:

- Additional sensors (force/torque, tactile)
- Gripper control (GripperActionController)
- Vision processing pipelines
- Mobile base integration

Camera topics and standard interfaces enable perception integration without redesigning the system.

E. Documentation and Maintainability

Clear component separation and comprehensive diagrams help developers understand the system. This document explains concepts (ros2_control, JointTrajectoryController) without assuming deep robotics expertise, making the system accessible to software engineers.

VIII. Conclusion

This design establishes a robust and extensible control and simulation architecture for robotic manipulators using ROS2. The combination of ROS2 Control’s JointTrajectoryController and MoveIt 2 provides reliable motion planning and execution, while Gazebo simulation enables thorough testing without hardware risk.

The hardware interface abstraction allows smooth transition to real robots, translating commands to native formats while maintaining software interfaces. Following ROS2 best practices and a test-driven incremental approach ensures maintainability and component verification at multiple levels.

This approach significantly reduces surprises when moving from simulation to real hardware, as the only differences lie in low-level communication and physics, while control logic and command flow remain identical. The result is a fully working simulated robot in Gazebo/RViz with motion planning capabilities, and a clear path to controlling actual hardware using developed controllers and interfaces.

Key benefits of this architecture include:

- **Reduced development risk**: Comprehensive simulation testing before hardware
- **Faster iteration**: Quick testing in simulation accelerates development
- **Code reusability**: Controllers and planners work across platforms
- **Standards compliance**: Following ROS2 conventions ensures compatibility
- **Maintainability**: Clear separation of concerns simplifies debugging

Future work could extend this architecture to include:

- Advanced perception integration with MoveIt planning scene
- Force control and compliance behaviors
- Multi-robot coordination
- Real-time performance optimization
- Hardware-in-the-loop testing frameworks

ACKNOWLEDGMENTS

This document represents a design study generated with AI assistance. The architecture described follows established ROS2 patterns and best practices from the robotics community. All references have been verified for accuracy and accessibility.

About This Document

This document represents a design study—something I found technically interesting and decided to explore further through AI-assisted research and synthesis. My process for creating these technical notes is typically as follows:

1) **Discovery**: Identify an interesting technical problem or system—in this case, the challenge of integrating a Kawasaki FS06N industrial robot with modern ROS2 control architecture.
2) **Initial Exploration**: Formulate, either by typing or dictating, a set of requirements, questions, and design considerations. Then engage with AI systems (Claude, ChatGPT, and others) to expand on these ideas. Usually in a dialogue form: I pose design questions, the AI provides architectural options, I refine requirements, and so on.
3) **Synthesis**: At some point I collect the dialogue results, structure them as a LaTeX document following established academic conventions, and ask the AI to formulate it as coherent technical prose with proper citations.
4) **Visualization**: Request diagram generation (PlantUML, Python scripts) when it aids understanding. The information for illustrations comes from the text itself or the references.
5) **Verification**: Rigorously check all references. References must be real and accessible, not AI hallucinations. I verify that URLs work, that authors and publications exist, and that the cited sources actually support the claims made in the document.

The result is a technical note documenting a design study that helps me understand and remember the system architecture. If this helps others to understand ROS2 control architecture for industrial robots, then I am happy. If not, I would of course like to hear about possible improvements. I will take every suggestion seriously and try to implement every reasonable one.

Note on References and Verification

This document contains AI-generated content. All references have been subject to rigorous verification to ensure academic integrity.

Verification Process:

- All URLs were tested for accessibility using automated tools
- Only official documentation and verified sources are cited
- Publication venues were validated
- Content relevance was checked against citations
- Verification dates are included in bibliography notes

Verification Status in References: Each reference includes a `note` field indicating its verification status and accessibility. All references in this document have been verified as accessible and accurate as of 2025-11-16.

Important Notice: Due to the AI-assisted nature of this document's creation, readers should independently verify any references used for critical applications. This level of scrutiny is essential when working with AI-generated academic content.

REFERENCES

[1] Open Source Robotics Foundation, "Ros 2 documentation - jazzy," 2024, verified: Official ROS2 Jazzy distribution documentation. URL accessible 2025-11-16. Latest stable LTS release. [Online]. Available: https://docs.ros.org/en/jazzy/index.html

[2] Kawasaki Robotics, "Kawasaki fs-series industrial robots," 2024, verified: FS-series robot information including FS06N specifications. URL accessible 2025-11-16. 6kg payload, 1102mm reach, 0.05mm repeatability, 6-axis robot. [Online]. Available: https://usedrobotstrade.com/Kawasaki/FS06

[3] Kawasaki Heavy Industries, Ltd., *Kawasaki Robot Controller D Series AS Language Reference Manual*, Kawasaki Heavy Industries, Ltd., 2024, verified: AS Language Reference Manual for D series controllers. URL accessible 2025-11-16. Covers AS system programming, commands, and robot trajectory control. [Online]. Available: https://cncmanual.com/kawasaki-robot-as-language-reference-manual-d-series-controller/

[4] ——, *Kawasaki Robot Controller F Series AS Language Reference Manual*, Kawasaki Heavy Industries, Ltd., 2024, verified: Official AS Language programming manual for F series controllers. Document 90209-1025DEB. URL accessible 2025-11-16. Covers AS system outline, data types, trajectory control, and all commands. [Online]. Available: https://www.robowizard.ru/storage/app/media/october-webinar/90209-1025DEB_FControllerASLanguageReferenceManual.pdf

[5] ROS2 Control Contributors, "Welcome to the ros2_control documentation," 2025, verified: Official ROS2 Control framework documentation. URL accessible 2025-11-16. Covers hardware interfaces, controllers, and real-time control. [Online]. Available: https://control.ros.org/

[6] PickNik Robotics and MoveIt Contributors, "Moveit 2 documentation," 2024, verified: Official MoveIt2 documentation portal. URL accessible 2025-11-16. Covers motion planning, manipulation, and kinematics. [Online]. Available: https://moveit.picknik.ai/

[7] Kawasaki Robotics, "As language programming," 2024, verified: Official Kawasaki AS Language programming overview. URL accessible 2025-11-16. General AS language documentation for Kawasaki robot controllers. [Online]. Available: https://robotics.kawasaki.com/userassets1/productpdf/as_language.pdf

[8] PickNik Robotics and MoveIt Contributors, "Getting started - moveit documentation," 2024, verified: Official MoveIt2 getting started guide. URL accessible 2025-11-16. Covers controller integration. [Online]. Available: https://moveit.picknik.ai/main/doc/tutorials/getting_started/getting_started.html

[9] ROS2 Control Contributors, "Welcome to the ros2_control documentation - jazzy," 2025, verified: Official ROS2 Control documentation for Jazzy distribution. URL accessible 2025-11-16. Latest LTS release. [Online]. Available: https://control.ros.org/jazzy/index.html

[10] Open Source Robotics Foundation, "Urdf (unified robot description format)," 2024, verified: Official URDF specification and documentation. URL accessible 2025-11-16. XML format for representing robot models in ROS. [Online]. Available: https://wiki.ros.org/urdf

[11] ——, "Xacro - xml macros," 2024, verified: Official Xacro documentation. URL accessible 2025-11-16. XML macro language for simplifying URDF files. [Online]. Available: https://wiki.ros.org/xacro

[12] ——, "Gazebo: Tutorial - ros 2 overview," 2024, verified: Official Gazebo Classic ROS2 integration tutorial. URL accessible 2025-11-16. Covers gazebo_ros_pkgs and plugins. [Online]. Available: https://classic.gazebosim.org/tutorials?tut=ros2_overview

[13] ——, "Gazebo: Tutorial - gazebo plugins in ros," 2024, verified: Official Gazebo Classic plugins tutorial. URL accessible 2025-11-16. Covers camera and sensor plugins. [Online]. Available: https://classic.gazebosim.org/tutorials?tut=ros_gzplugins

[14] ROS2 Control Contributors, "ros2_controllers repository," 2025, verified: Official GitHub repository for ROS2 controllers. URL accessible 2025-11-16. Source code and examples. [Online]. Available: https://github.com/ros-controls/ros2_controllers

[15] Open Source Robotics Foundation, "Setting up a robot simulation (gazebo)," 2024, verified: Official ROS2 Gazebo simulation tutorial. URL accessible 2025-11-16. Part of ROS2 Humble documentation. [Online]. Available: https://docs.ros.org/en/humble/Tutorials/Advanced/Simulators/Gazebo/Gazebo.html

[16] Nav2 Contributors, "Setting up sensors - gazebo classic," 2024, verified: Nav2 documentation on Gazebo sensor setup. URL accessible 2025-11-16. Practical examples of lidar and depth cameras. [Online]. Available: https://docs.nav2.org/setup_guides/sensors/setup_sensors_gz_classic.html

[17] MoveIt Contributors, "Srdf (semantic robot description format)," 2024, verified: SRDF format documentation. URL accessible 2025-11-16. Semantic information for MoveIt motion planning. [Online]. Available: https://moveit.picknik.ai/main/doc/examples/urdf_srdf/urdf_srdf_tutorial.html