UNIVERSITÀ
DI TRENTO
Department of
Information Engineering and Computer Science

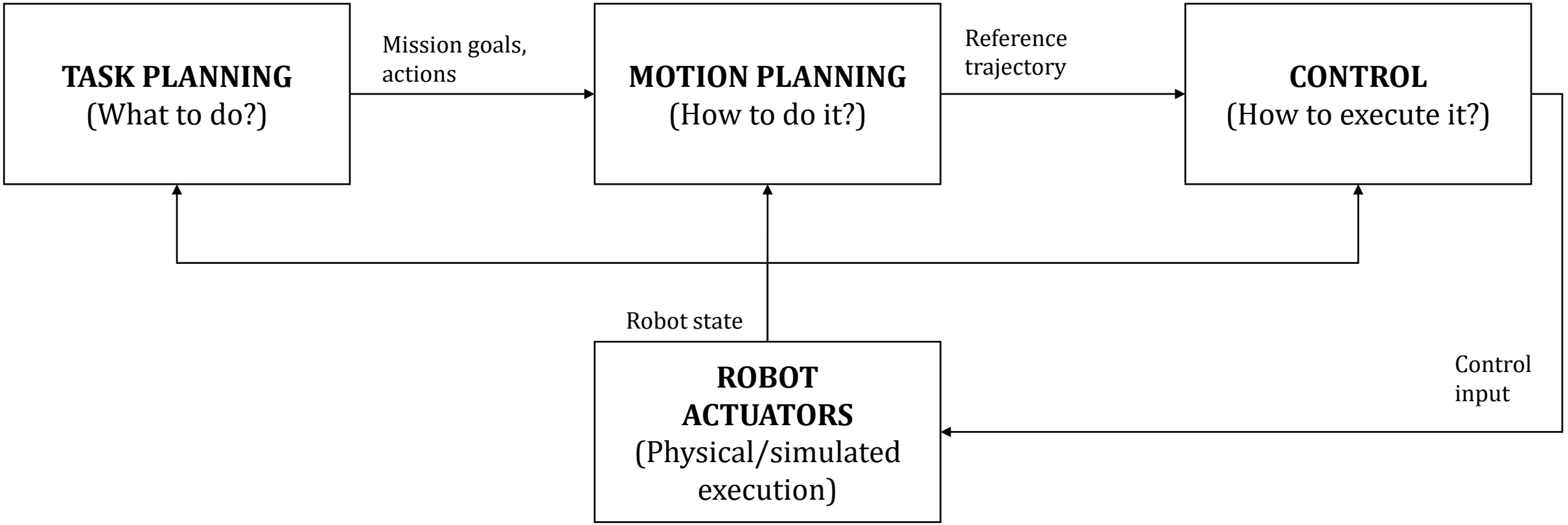
ROS 2 Motion and Task Planning

Edoardo Lamon

Software Development for Collaborative Robots

Academic Year 2025/26

High-level Framework for a Robotic Application

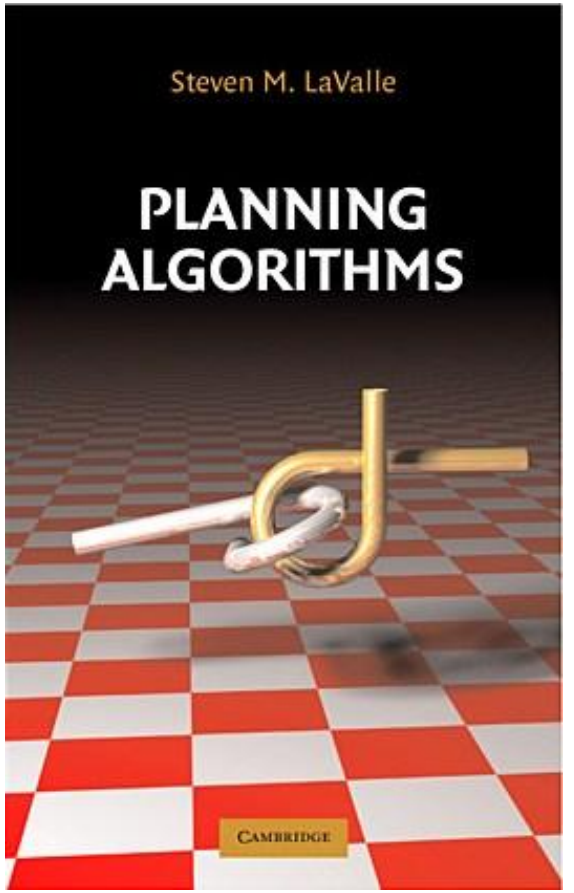


High-level Framework for a Robotic Application

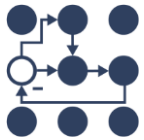
Layer	Purpose	Time Scale	Typical Input	Typical Output
Task Planning	Decides *what* to do: high-level goals and action sequences to achieve a mission.	Seconds → Minutes	Mission objective, world model, available actions	Sequence of actions or goals
Motion Planning	Decides *how* to move to reach each goal: finds a feasible path.	Milliseconds → Seconds	Start/goal states, environment map, robot kinematics	Cartesian of joint trajectory/path
Control	Executes *how to move:* converts planned motion into motor commands while reacting to sensors.	Milliseconds	Reference trajectory, sensor feedback	Low-level torque/ velocity/position commands

ROS 2 Tools

Motion Planners in ROS 2



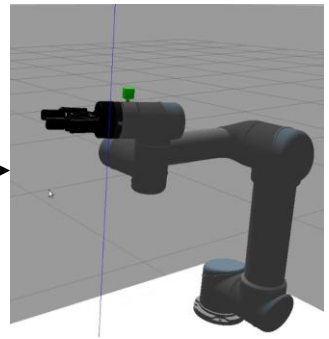
Robot Programming Pipeline



ros2_control

GAZEBO

CoppeliaSim

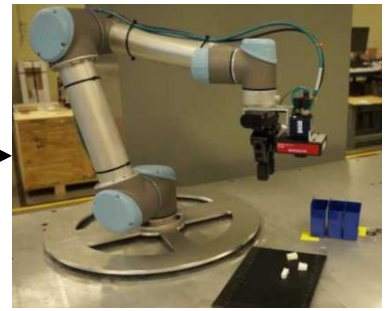


??

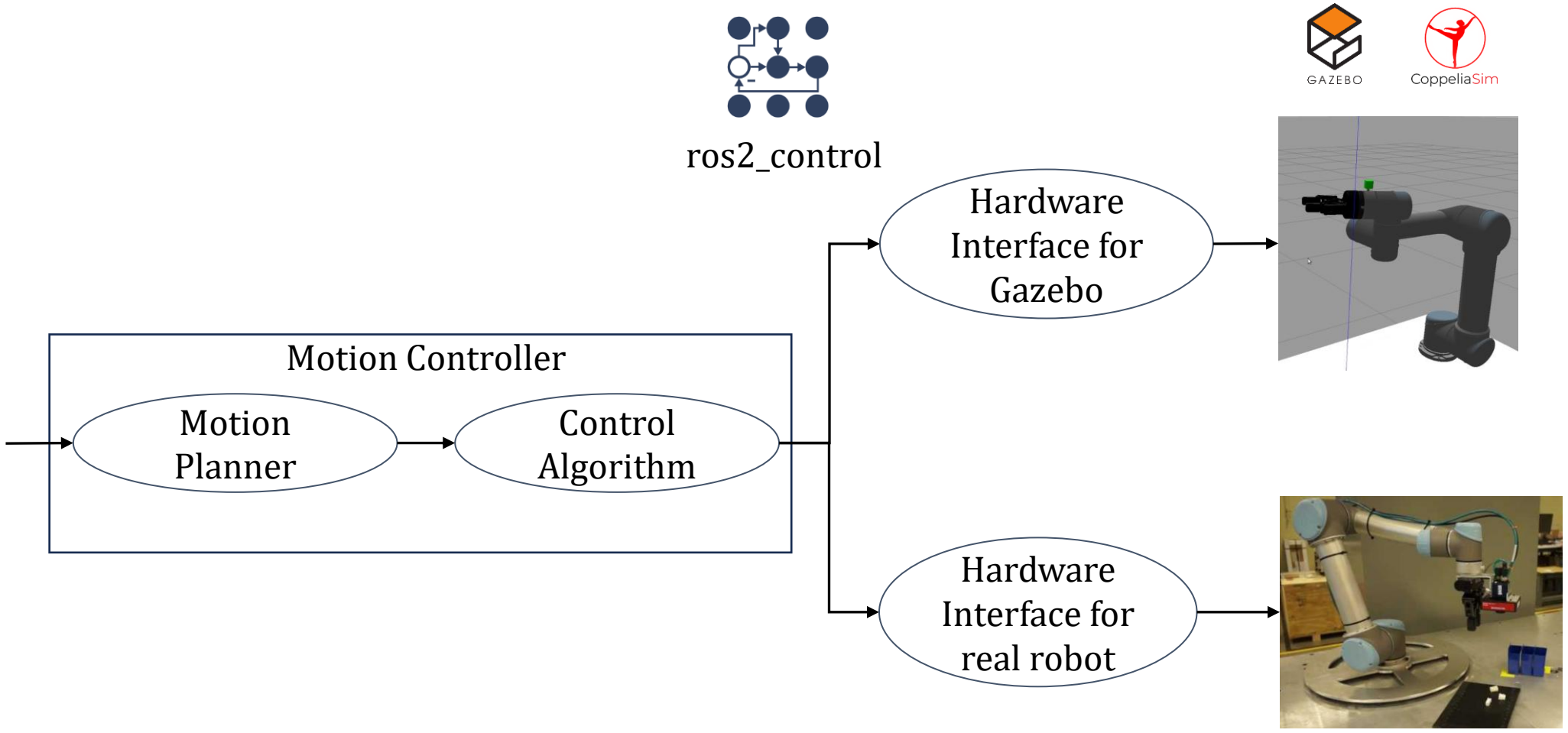
Control Algorithm

Hardware Interface for Gazebo

Hardware Interface for real robot



Robot Programming Pipeline

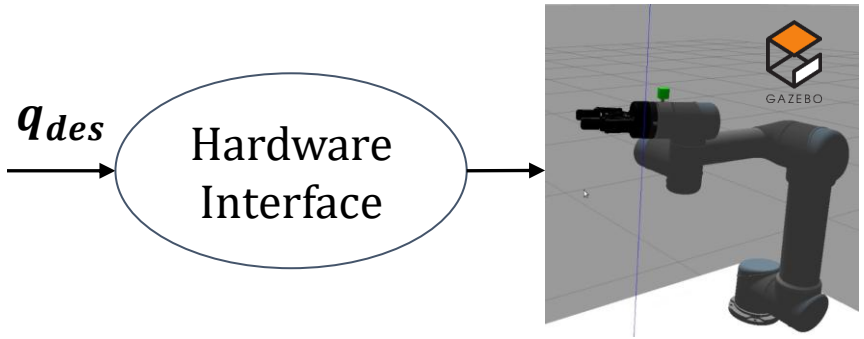


Motion Planning

- A control algorithm ensures that, given a predefined **reference** (either, position, velocity, acceleration, torque, force, etc.), the system is able to track the reference (low-level, usually in joint space and includes physical limits of the system);
- Each reference is assumed to be tracked in **short time** (one or few control loops, the reference is close to the actual value);
- Usually, the control references are generated by sampling higher level **trajectories (path+timings)** usually in Cartesian space (various control loops and depends on the distance of the actual position from the goal).
- Generally, within trajectories one can include **motion constraints** (self-collisions, obstacles etc.);
- Motion Planning and Control problems are really close and not always are handled separately. In general, one can refer to the Motion Control problem.

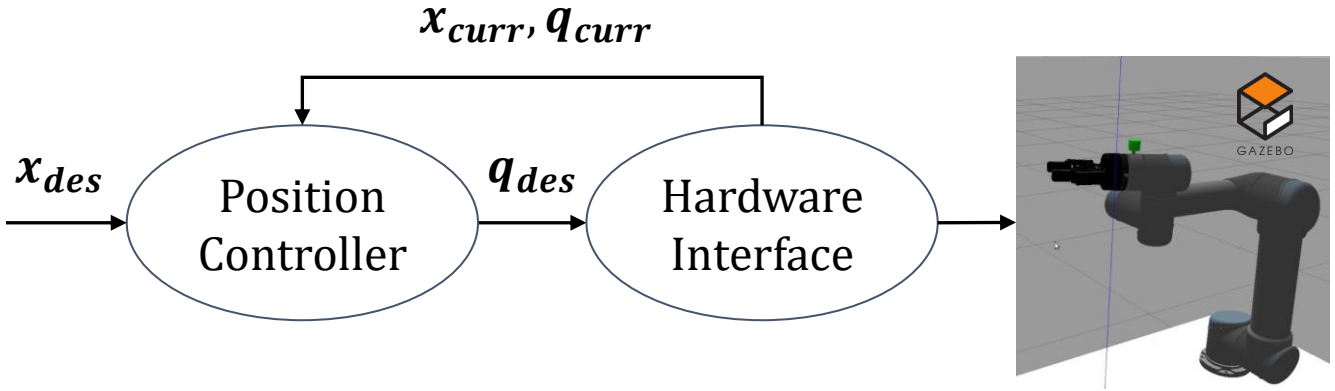
Motion Control example

Let's suppose to have an UR5 with a hardware interface which exposes joint position → We can set the target joint configuration $\boldsymbol{q_{des}}$.



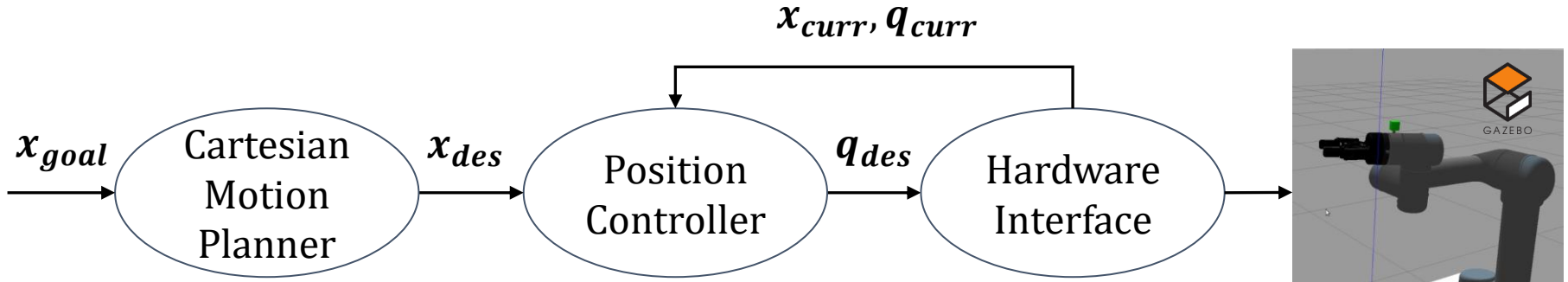
Motion Control example

A Cartesian position controller can receive a desired pose in the Cartesian space $\boldsymbol{x_{des}}$, read the current pose $\boldsymbol{x_{curr}}$ and configuration $\boldsymbol{q_{curr}}$ and map it to a desired configuration $\boldsymbol{q_{des}}$ (for example through inverse kinematics).



Motion Control example

A Cartesian motion planner can receive a goal pose $\boldsymbol{x_{goal}}$ which can be arbitrarily far from the current pose $\boldsymbol{x_{curr}}$, generate a trajectory with some patterns of motion (linear, curved, etc.), including also environmental constraints such as obstacles in the path. The trajectory is then sampled with step **dt** (control loop rate) to generate $\boldsymbol{x_{des}}$ and changes the position controller reference every **dt**.



OPEN CHAIN
(the current state is read only once, no feedback)

FEEDBACK LOOP
(every loop the current state is updated)

ROS2 Motion Planning

ROS 2 integrates two main frameworks for motion planning:

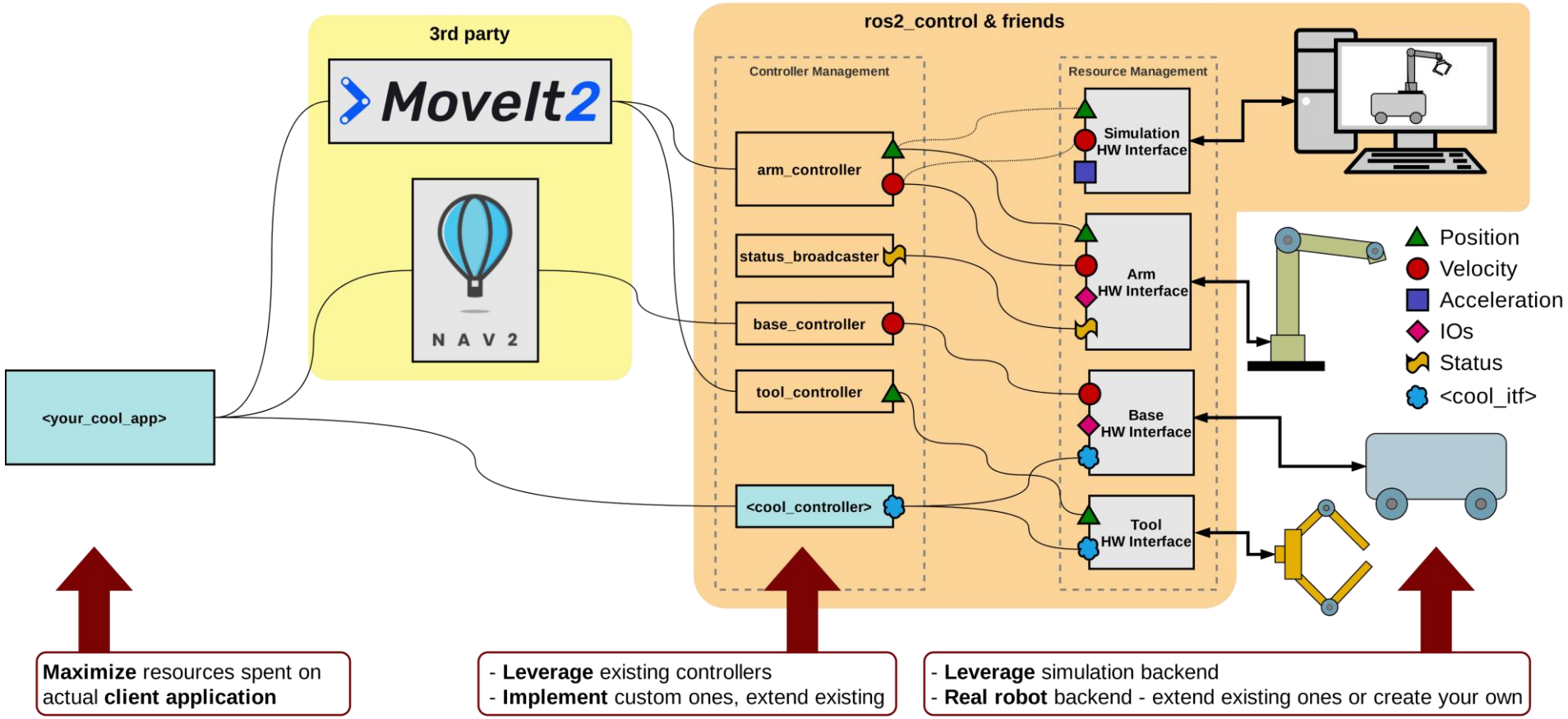
- MoveIt for manipulation (robotic arms);
- Nav2 for locomotion (mobile robots).

Frameworks are convenient, but often require long setup times and it is not always convenient to use them (depends on the features that one requires).

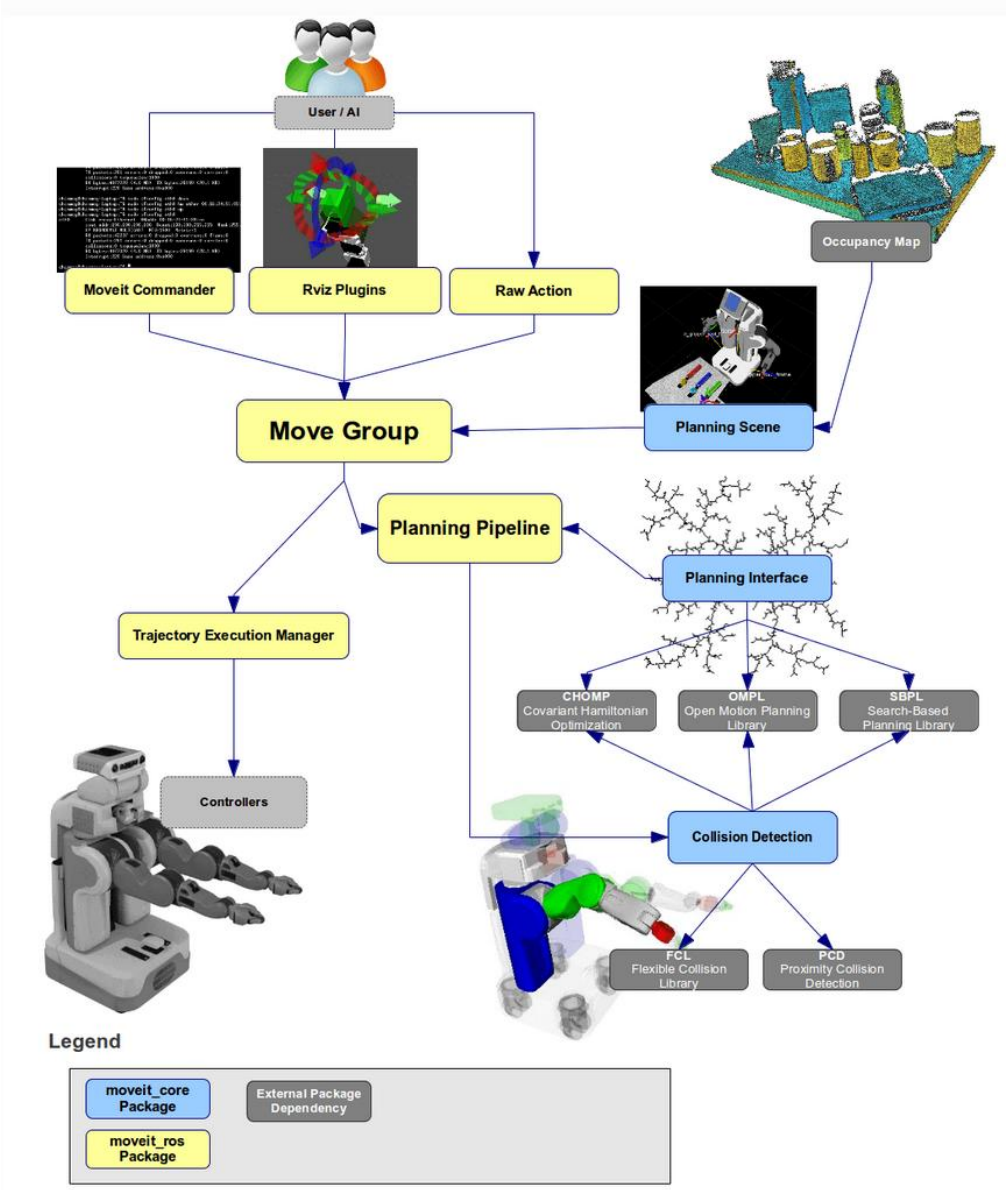
Othe options:

- Curobo from Nvidia;

ROS2 Motion Planning



MoveIt2


User / AI
Moveit Commander
Rviz Plugins
Raw Action
Occupancy Map
Planning Scene
Move Group
Planning Pipeline
Planning Interface
Trajectory Execution Manager
CHOMP
Covariant Hamiltonian Optimization
OMPL
Open Motion Planning Library
SBPL
Search-Based Planning Library
Controllers
Collision Detection
FCL
Flexible Collision Library
PCD
Proximity Collision Detection
Legend
moveit_core Package
moveit_ros Package
External Package Dependency

MoveIt2

MoveIt2 Component	Role / Function
Planning Scene	Represents the **environment** where the robot operates. Includes static geometry (meshes, polygons) and **dynamic obstacles** from sensors (lidar, cameras).
Planning Pipeline	Computes **collision-free trajectories** using robot kinematics and the planning scene. Supports **hybrid planning** combining global and local planners.
Move Group	The main **user interface node**. Receives planning requests, coordinates data collection, calls the planning pipeline, and sends the trajectory to execution. A robot may have multiple move groups (e.g., arm, gripper).
Trajectory Execution Manager	Sends planned trajectories to **controllers** via **ROS 2 actions**, monitors execution, and updates paths if the environment changes.
User Interfaces	- **RViz plugin:** visualize and interact with the robot and generated trajectories. - **CLI tools:** send planning or execution commands from the terminal.

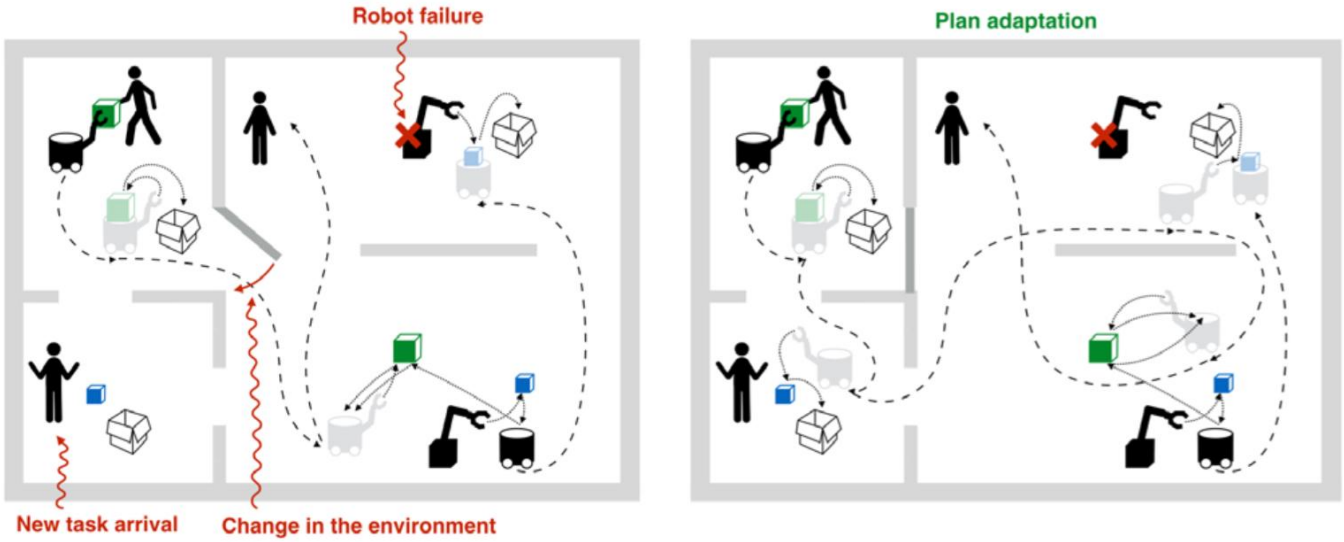
MoveIt2

MoveIt provides ROS 2 packages to plan manipulation tasks and integrates several libraries:

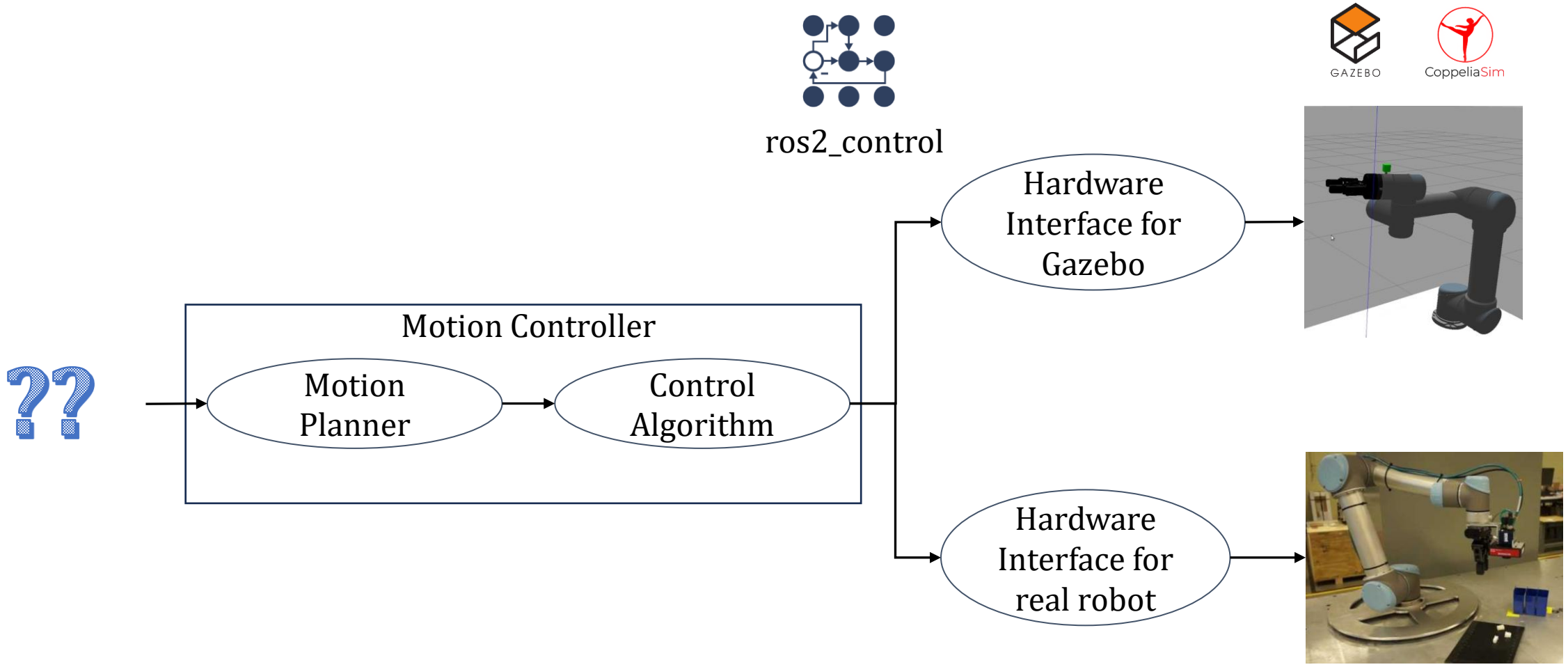
- The Kinematics and Dynamics Library (KDL) for modelling and computation of kinematic chains.
- Inverse Kinematics solvers like IKfast from OPENRAVE, or TrakIK;
- The Open Motion Planning Library (OMPL), a collection of state-of-the-art sampling-based motion planning algorithms.
- The Stochastic Trajectory Optimization for Motion Planning (STOMP), a probabilistic optimization framework for collision-free paths;
- The Trajectory Optimization Motion Planner (TrajOpt), a sequential convex optimization algorithm for motion planning problems.
- And others (SRMP ...)

ROS 2 Tools

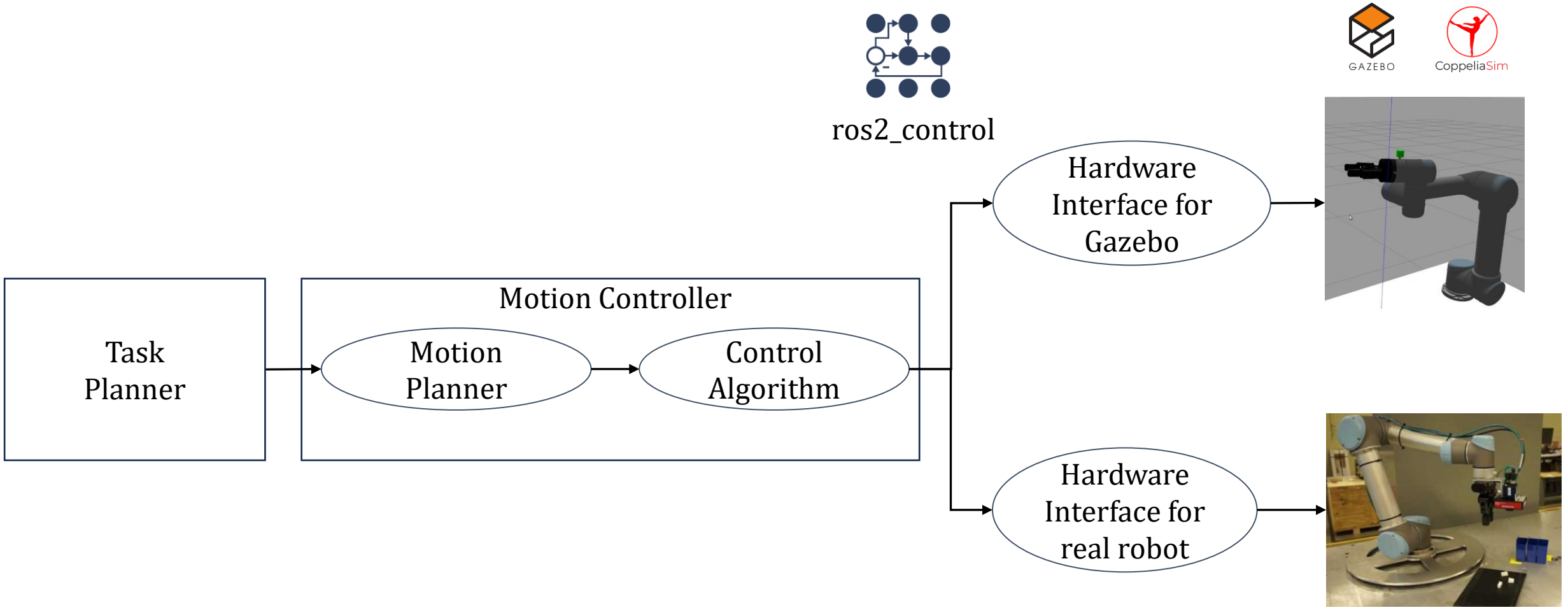
Task Planners in ROS 2



Robot Programming Pipeline



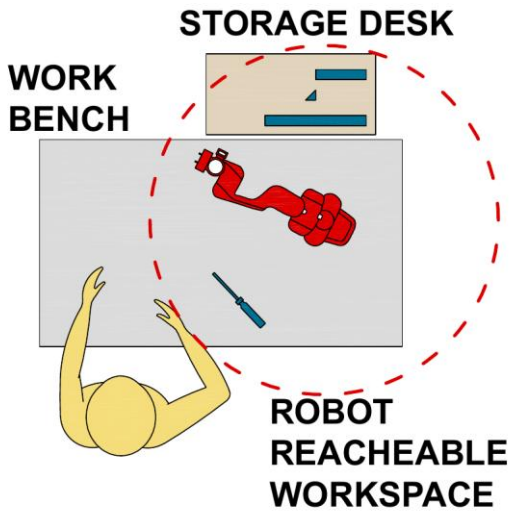
Robot Programming Pipeline



Task Planning example

- Suppose we have a manipulator in a world like the one in the picture. We would like our robot to perform a task such as *"take the objects from the storage desk and put it on the work bench"*.
- We could describe what the robot should do by breaking the solution down into individual *actions:*

  1. **Move** to the *desk*
  2. **Pick** up the *object*
  3. **Move** to the *work bench*
  4. **Place** the *object*



Task Planning example

- What if the task gets more complicated? Suppose we add more objects and locations to our simple world, and start describing increasingly complex goals like *“make sure all the screws are on the work bench and everything else is in the desk”*?
- Furthermore, what if we want to achieve this in some optimal manner like minimizing time, or number of total actions? Our reasoning quickly hits its limit as the problem grows, and perhaps we want to consider letting a computer do the work.

Task Planning

- Task planning refers to the autonomous reasoning about the state of the world using an internal *model* and coming up a sequence of *actions*, or a *plan*, to achieve a *goal*.

- It requires a model of the world and how an autonomous agent can interact with the world. This is usually described using *state* and *actions*. Given a particular state of the world, our robot can take some action to *transition* to another state.

- For example, in Finite State Machine (FSM), the goal is formulated as the final state.

Task Planning

- Task planning refers to the autonomous reasoning about the state of the world using an internal *model* and coming up a sequence of *actions*, or a *plan*, to achieve a *goal*.

- It requires a model of the world and how an autonomous agent can interact with the world. This is usually described using *state* and *actions*. Given a particular state of the world, our robot can take some action to *transition* to another state.

- For example, in Finite State Machine (FSM), the goal is formulated as the final state.

Task Planning features

- **Easy introspection**: figuring out what the robot is going to do given its state and inputs shouldn't be too complicated.
- **Maintainability**: the system should be easily modifiable.
- **Resilience**: the system should be capable of dealing with unexpected circumstances in the environment.
- **Modularity** and **scalability**: the components, and those actions composed by the interaction of multiple components, should be reusable not only within one system, but also in heterogeneous systems (for example in different robots).
- **Efficiency**: the organization should happen without significant computational overhead.
- **Separation of concerns**: each component behavior depends only on a limited amount of inputs → the logic controlling the execution flow of the machine needs to be separated from the actions implementation.

Task Planning in robotics

Task planning is an AI problem and robotics is only one of its main application field. Thus:

- Best task planners are agent and task agnostic;
- Actions are a more general class of interactions which include motions;
- Some task planners require a Knowledge Base (e.g., PDDL, Prolog, etc.);
- Most of task planners are not integrated with ROS2.

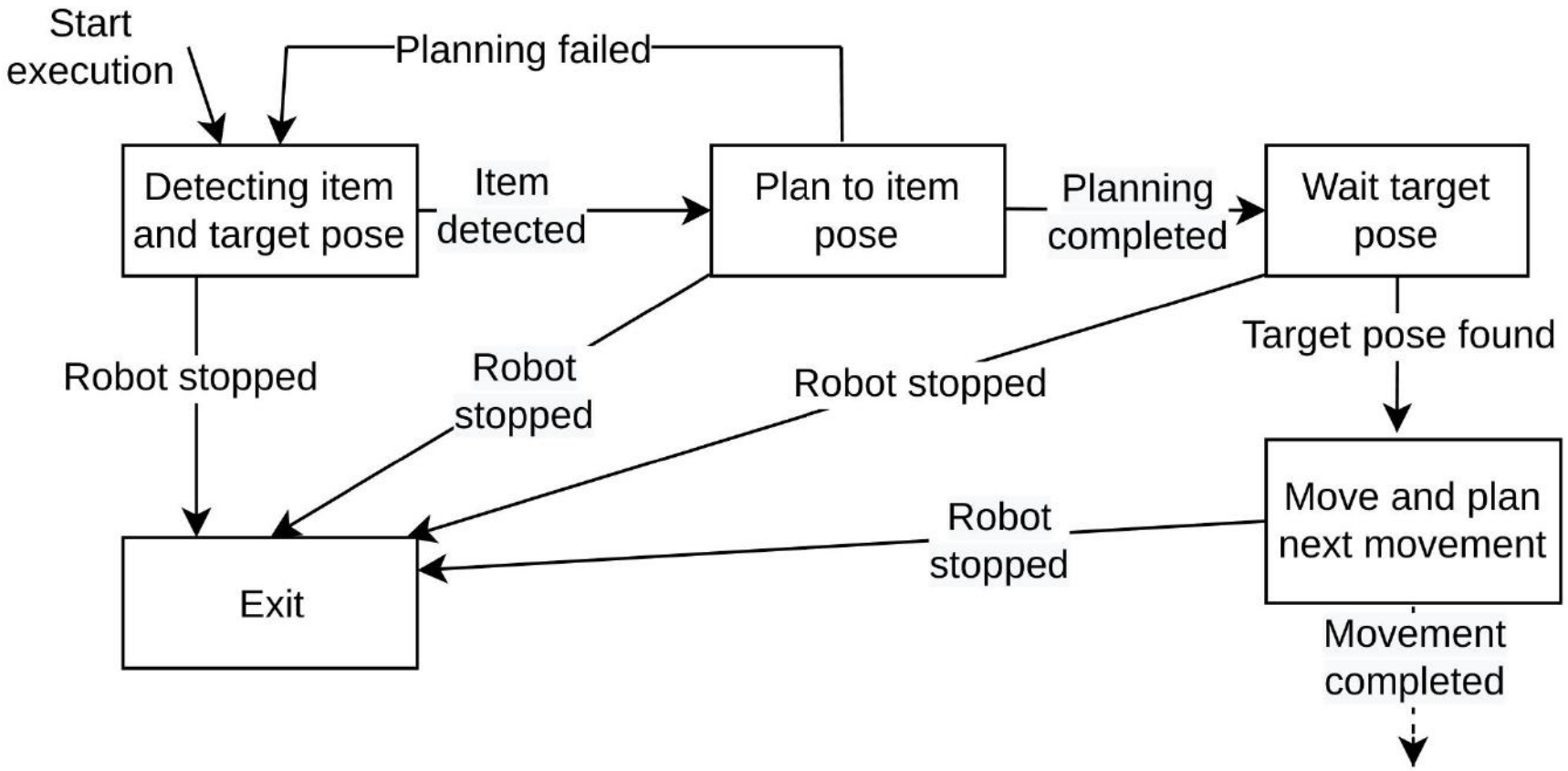
Finite State Machines (FSMs)

A **Finite State Machine (FSM)** models a system as a **finite set of states** and **rules for transitioning** between them.

At any time:

- The system is in **one and only one state**.
- Transitions occur **only if defined conditions are met**.
- **Different actions** are executed depending on the current state or transition.

Finite State Machines (FSMs)



Finite State Machines (FSMs)

Issue	Description
Explosion of transitions	As the number of states grows, the number of possible transitions increases even faster, especially for error handling.
Hard to visualize	Large FSMs become complex to represent and maintain, losing their initial readability.
Limited reusability	Transitions and logic are often tied to local variables, making code reuse difficult.

Solution: **Hierarchical Finite State Machines (HFSMs)**

- Nested states: States can contain **sub-states**, forming modular sub–state machines.
- Grouped logic: Related behaviors are **encapsulated** inside macro-states (e.g., *Manipulation* contains *Plan*, *Move*, *Grasp*).
- Simpler transitions: Transitions can occur between sub-states and higher-level states, reducing redundancy
- More modular: Encourages reuse and easier maintenance for large robotic systems.

Behavior trees

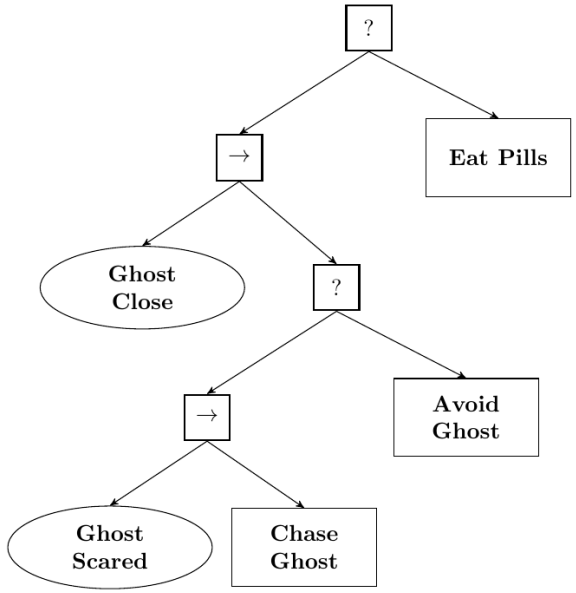
A **Behavior Tree** organizes tasks and conditions into a **hierarchical tree** traversed from **root → leaves** at each *tick*. Each **leaf node** represents an **action** or **condition**, while **control nodes** decide the execution flow.

Execution:

- The tree is **"ticked" periodically** or when an **event** occurs.
- The tick travels **from root to leaves**, returning *success*, *failure*, or *running*.
- Shared data between nodes is stored in a **blackboard**.

Behavior trees

Type of Node	Symbol	Success	Failure	Running
Sequence	$\rightarrow$	All children succeed	One child fails	One child returns Running
Fallback	?	One child succeeds	All children fail	One child returns Running
Decorator	$\diamond$	Custom	Custom	Custom
Parallel	$\rightrightarrows$	$\geq M$ children succeed	$> N - M$ children fail	else
Condition	$\bigcirc$	True	False	Never
Action	$\square$	Upon completion	Impossible to complete	During execution



Behavior trees

Reactivity

- Each task or condition quickly returns: ✅ *Success*, ❌ *Failure*, or ⏳ *Running*
- Enables **fast response** to environment or system changes
- Long-running tasks execute **asynchronously**, keeping the BT responsive

Modularity

- Branches can be **reused as subtrees**, promoting **hierarchical organization**
- **Add/remove behaviors** without redefining transitions
- Encourages **clean, maintainable, and scalable** designs

Readability

- **Graphical representation** makes robot logic easier to visualize and debug
- Once familiar with BT symbols, users can **interpret and modify** behavior intuitively
- Combines structure and clarity, even for complex systems

ROS2 Task Planning Libraries

Integration with ROS2:

- SMACC2 State Machine library, an event-driven, asynchronous, behavioral state machine library for real-time ROS 2 applications written in C++.
- Yasmin, implements robot behaviors using Finite State Machines (FSM). It is available for ROS 2, Python and C++.
- Behavior Trees, a tree-based planning system based on actions compositions (not state transitions). Both C++ implementation and ROS2 integration are available.

For generating autonomous behaviors without the need of a static program?

- FlexBE model behaviors as hierarchical state machines where states correspond to active actions and transitions describe the reaction to outcomes;
- Skiros2 implements an extension of the Behavior trees with parameters and conditions to form primitive and compound skills;
- PlanSys2 (inspired by ROSPlan), a PDDL(Planning Domain Definition Language)-based planning system implemented in ROS2.