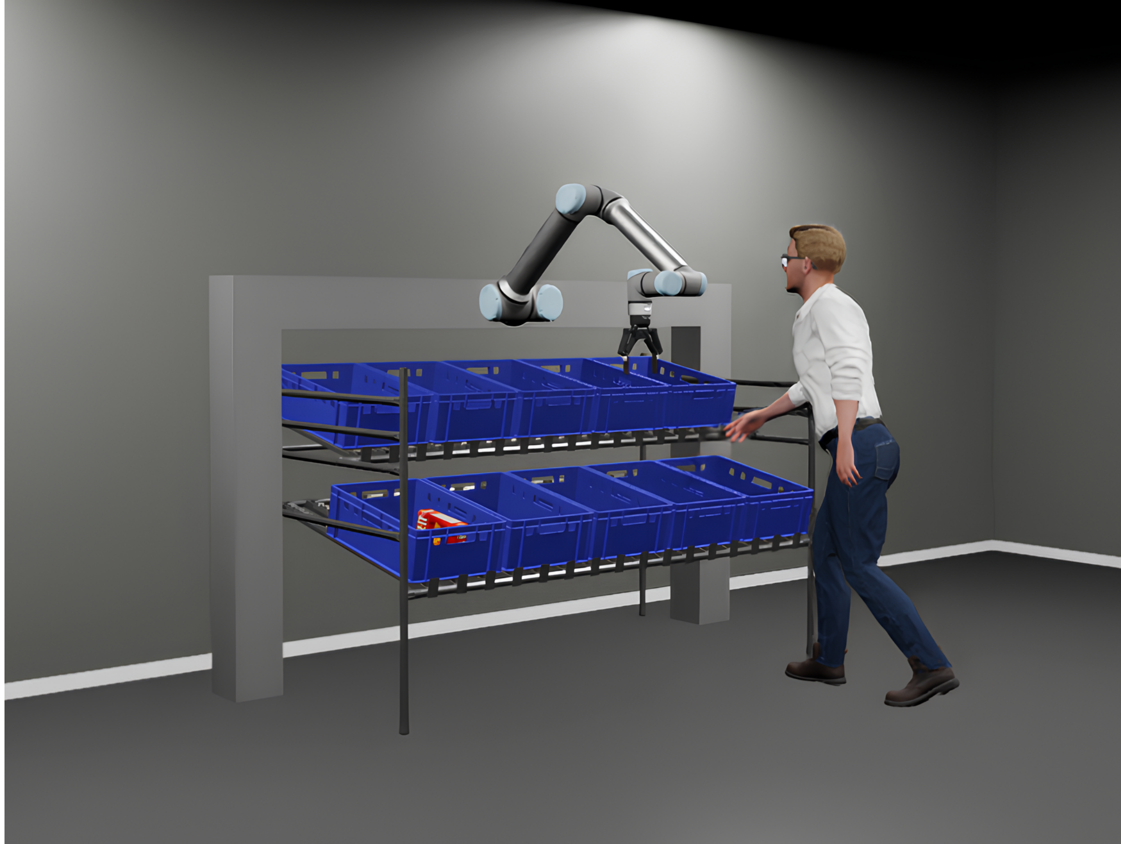


# Collaborative Robotic Arm and Humanoid Interaction for Kitting Tasks in Simulated Factory Environment

## A Simulation-Based Evaluation of Motion Planning Methods for Automated Kitting in Dynamic Industrial Environments

Master's thesis in Complex Adaptive Systems

MARCUS OLSSON
ELIAS WILSBORN

---

DEPARTMENT OF MECHANICS AND MARITIME SCIENCES
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2026
www.chalmers.se

Master's Thesis in Complex Adaptive Systems

# Collaborative Robotic Arm and Humanoid Interaction for Kitting Tasks in Simulated Factory Environment

A Simulation-Based Evaluation of Motion Planning Methods for Automated Kitting in Dynamic Industrial Environments

MARCUS OLSSON
ELIAS WILSBORN

Department of Mechanics and Maritime Sciences
Division of Vehicle Engineering and Autonomous Systems
Chalmers University of Technology
Gothenburg, Sweden 2026

Collaborative Robotic Arm and Humanoid Interaction for Kitting Tasks in Simulated Factory Environment
A Simulation-Based Evaluation of Motion Planning Methods for Automated Kitting in Dynamic Industrial Environments
MARCUS OLSSON
ELIAS WILSBORN

© MARCUS OLSSON, ELIAS WILSBORN, 2026.

Supervisor: Hamid Ebadi, PhD in Computer Science, Infotiv Technology Development
Examiner: Krister Wolff, Department of Mechanical Engineering

Master's Thesis 2026
Department of Mechanics and Maritime Sciences
Chalmers University of Technology
SE-412 96 Gothenburg
Sweden
Telephone +46 31 772 1000

Cover: Isaac Sim simulation scene showing a gantry-mounted UR10e manipulator with an attached gripper operating above a flow-rack kitting setup with blue storage crates and a humanoid actor positioned next to the rack.

Typeset in LaTeX
Gothenburg, Sweden 2026

Collaborative Robotic Arm and Humanoid Interaction for Kitting Tasks in Simulated Factory Environment
A Simulation-Based Evaluation of Motion Planning Methods for Automated Kitting in Dynamic Industrial Environments
MARCUS OLSSON
ELIAS WILSBORN
Department of Mechanics and Maritime Sciences
Division of Vehicle Engineering and Autonomous Systems
Chalmers University of Technology

# Abstract

Future industrial automation requires robotic systems that can operate in workspaces where objects, equipment, and humans may be present. This thesis presents the development and evaluation of a simulation-based system for motion planning in a kitting task. The system was built around a gantry-mounted UR10e robot arm with a Robotiq gripper in a simulated factory environment containing a flow rack, crates, static and dynamic obstacles.

The work integrated ROS 2, Isaac Sim, MoveIt 2, and several motion-planning frameworks in a containerised software architecture. Sampling-based planning with OMPL, GPU-accelerated planning with cuMotion and cuRobo, and a hybrid planner based on cuRobo MotionGen and MPC were implemented and evaluated. The planners were tested in simple motion cases, complete pick-and-place workflows, and a dynamic obstacle benchmark where the robot had to react to a newly introduced obstacle during execution.

The results showed that cuRobo provided the strongest overall balance between planning speed, success rate, and motion efficiency in the static benchmark cases. cuMotion also achieved high success rates, but generally required longer planning times. The OMPL planners were computationally cheap and could be fast in successful cases, but showed lower robustness in several scenarios. The hybrid planner was able to react to dynamic changes in the environment, but its success depended on how close the obstacle appeared to the robot. When enough clearance was available, the hybrid planner recovered reliably, while recovery became less likely when the obstacle was inserted very close to the robot.

The thesis shows that simulation is a useful tool for evaluating motion-planning methods for constrained industrial tasks before real-world deployment. It also shows that GPU-accelerated and hybrid planning methods are promising for robotic operation in dynamic environments, but that further work is needed before the system can fully represent realistic human-robot collaboration.

Keywords: robotics, motion planning, kitting, ROS 2, Isaac Sim, MoveIt 2, cuRobo, cuMotion, OMPL, human-robot collaboration.

# Preface

This report presents the outcome of our master's thesis project carried out at the Department of Mechanics and Maritime Sciences at Chalmers University of Technology during the spring of 2026. The thesis was done within the master's programme Complex Adaptive Systems and focuses on simulation-based motion planning for robotic kitting tasks in industrial environments.

The project was developed in collaboration with Infotiv Technology Development. The work allowed us to combine robotics, simulation, motion planning, and system integration in one larger project. A large part of the thesis involved building a complete software and simulation environment, integrating different planning methods, and evaluating how these methods behaved in both static and dynamic scenarios.

# Acknowledgements

We would like to thank our supervisor, Hamid Ebadi at Infotiv Technology Development, for his guidance, feedback, and support throughout the project. We would also like to thank our examiner, Krister Wolff at Chalmers University of Technology, for his valuable comments and academic guidance.
We would also like to thank Atieh Hanna, Researcher at Volvo Trucks Technology, for her helpful feedback and support during the thesis work. In addition, we would like to thank the rest of Infotiv for their support and for the trust they placed in us throughout the project.
Finally, we would like to thank everyone else who has supported us during the thesis work.

Marcus Olsson, Elias Wilsborn, Gothenburg, May 2026

# List of Acronyms

Below is the list of acronyms that have been used throughout this thesis listed in alphabetical order:

| | |
|---|---|
| CPU | Central Processing Unit |
| CSV | Comma-Separated Values |
| DOF | Degree of Freedom |
| GPU | Graphics Processing Unit |
| IK | Inverse Kinematics |
| JSON | JavaScript Object Notation |
| L-BFGS | Limited-memory Broyden–Fletcher–Goldfarb–Shanno |
| MPC | Model Predictive Control |
| MPPI | Model Predictive Path Integral |
| OMPL | Open Motion Planning Library |
| PRM | Probabilistic Roadmap |
| RRT | Rapidly-exploring Random Tree |
| TCP | Tool Center Point |
| TF | Transform |
| URDF | Unified Robot Description Format |
| USD | Universal Scene Description |
| XML | Extensible Markup Language |
| XRDF | XML Robot Description Format |

# Definitions

Below is the list of terms that have been used throughout this thesis.

| | |
|---|---|
| Motion | The physical movement of the robot from one state to another during execution, including changes in joint configuration and end-effector pose over time. |
| Trajectory | A time-parameterised sequence of robot configurations that specifies both the configurations the robot should pass through and when they should be reached. |
| Configuration | A complete specification of the robot's joint values, usually represented as a vector $q$, where each element corresponds to one joint variable. |
| Pose | The position and orientation of a body or end-effector in the workspace, often represented by a homogeneous transformation matrix. |
| Path | A continuous geometric curve through configuration space from a start configuration to a goal configuration, without specifying timing. |
| Plan | The output of a planning method, typically a collision-free path or trajectory that can be used to move the robot toward a goal. |
| Seed | An initial candidate path or trajectory used as a starting point for trajectory optimisation before refinement. |

# Nomenclature

Below is the nomenclature of indices, sets, parameters, and variables that have been used throughout this thesis.

## Indices

| | |
|---|---|
| $i$ | Joint and candidate-trajectory index |
| $j$ | Candidate-trajectory summation index |
| $k$ | Prediction-step index |
| $t$ | Trajectory time-step index |

## Sets

| | |
|---|---|
| $\mathcal{C}$ | Robot configuration space |
| $\mathcal{C}_{free}$ | Collision-free subset of the configuration space |
| $\mathcal{C}_{obs}$ | Obstacle subset of the configuration space |
| $\mathcal{X}$ | Set of allowed system states in MPC |
| $\mathcal{U}$ | Set of allowed control inputs in MPC |

## Parameters

| | |
|---|---|
| $n$ | Number of robot degrees of freedom |
| $N$ | Prediction horizon |
| $T$ | Trajectory horizon length |
| $K$ | Number of sampled candidate trajectories |
| $\alpha$ | Optimisation step length |
| $\beta$ | Temperature parameter for trajectory weighting |
| $\sigma$ | Sampling covariance used to generate candidate trajectories |

| | |
|---|---|
| $q_i^{min}$ | Lower joint limit of joint $i$ |
| $q_i^{max}$ | Upper joint limit of joint $i$ |

# Variables

| | |
|---|---|
| $q$ | Robot joint configuration vector |
| $q_i$ | Joint value of joint $i$ |
| $q_{start}$ | Start configuration of the robot |
| $q_{goal}$ | Goal configuration of the robot |
| $q_{retract}$ | Retract reference configuration |
| $\theta_t$ | Robot configuration at trajectory time step $t$ |
| $\theta_{[0,T]}$ | Complete discrete joint-space trajectory |
| $\tau$ | Continuous path in configuration space |
| $s$ | Path-progress parameter along a path |
| $T(q)$ | Forward-kinematics transformation for configuration $q$ |
| $x_k$ | System state at prediction step $k$ |
| $u_k$ | Control input at prediction step $k$ |

# Contents

**List of Acronyms** **ix**

**List of Definitions** **xi**

**Nomenclature** **xiii**

**List of Figures** **xix**

**List of Tables** **xxiii**

**1 Introduction** **1**
1.1 Background . . . . 1
1.2 Purpose . . . . 2
1.3 Goals . . . . 3
1.4 Scope and Limitations . . . . 3
1.5 Contributions . . . . 4

**2 Theory** **7**
2.1 Simulation in Robotics . . . . 7
2.2 Robot Kinematics and Configuration Space . . . . 7
2.2.1 Degrees of Freedom and Joint Representation . . . . 7
2.2.2 Forward and Inverse Kinematics . . . . 8
2.2.3 Configuration Space Representation . . . . 9
2.2.4 Obstacle Representation and Collision Detection . . . . 10
2.3 Motion Planning . . . . 11
2.3.1 Problem Formulation . . . . 11
2.3.2 Sampling-Based Planning . . . . 12
2.3.3 Trajectory-optimisation-based planning . . . . 14
2.3.4 Planning with Continuous Updating . . . . 18
2.3.4.1 Model Predictive Control . . . . 18
2.3.4.2 Model Predictive Path Integral . . . . 20

**3 Methods** **21**
3.1 Project Structure . . . . 21
3.2 ROS 2 Integration . . . . 24
3.3 Simulation . . . . 26
3.3.1 Simulation Environment . . . . 27

3.4 Path Planning . . . . . 29
3.4.1 Planning Group and Goal Representation . . . . . 30
3.4.2 Sampling-Based Planning with OMPL . . . . . 30
3.4.3 GPU-Accelerated Planning with cuMotion . . . . . 31
3.4.4 GPU-Accelerated Planning with cuRobo . . . . . 32
3.4.5 Hybrid Planning with cuRobo MotionGen and MPC . . . . . 34
3.5 Path Planners Comparison . . . . . 38
3.6 Test Automation . . . . . 38
3.7 Hardware and Software Setup . . . . . 40
3.8 Data Collection . . . . . 41
3.9 Dynamic Human Obstacle Integration . . . . . 43
3.9.1 Dual-Camera Motion Capture . . . . . 43
3.9.2 Character Rigging and Collision Modelling . . . . . 45

**4 Results** **49**
4.1 Simple Motion Benchmark . . . . . 49
4.1.1 Success Rate . . . . . 49
4.1.2 Planning Time . . . . . 50
4.1.3 Execution Time . . . . . 50
4.1.4 TCP Movement . . . . . 51
4.1.5 Joint Movements . . . . . 51
4.2 Pick-and-Place Benchmark . . . . . 52
4.2.1 Workflow Success Rate . . . . . 52
4.2.2 Workflow Time . . . . . 53
4.2.3 Per-Phase Planning Time . . . . . 54
4.2.4 Per-Phase Success Rate . . . . . 55
4.2.5 Joint and Gantry Motion . . . . . 55
4.3 Hybrid Dynamic Obstacle Benchmark . . . . . 57
4.4 Humanoid Obstacle Demonstration . . . . . 58
4.5 Resource Usage . . . . . 59

**5 Discussion** **61**
5.1 Overall Planner Performance . . . . . 61
5.2 Planning Speed Versus Robustness . . . . . 62
5.3 Motion Efficiency and Motion Distribution . . . . . 63
5.4 Full-Workflow Evaluation and Failure Modes . . . . . 65
5.5 Hybrid Planning for Dynamic Obstacles . . . . . 66
5.6 Cost of Reactive Planning . . . . . 68
5.7 Implications for Human-Robot Collaboration . . . . . 68
5.8 Computational Cost and Deployment Considerations . . . . . 70
5.9 Limitations . . . . . 71

**6 Conclusion** **73**
6.1 Future Work . . . . . 74

**Bibliography** **77**

**A Simple Motion Cases** I

**B Pick-and-Place Phases** III

**C Detailed Result Tables** VII

# List of Figures


3.1 Overview of the project software architecture. The system is divided into three runtime containers: a planner container for planning libraries and planning interfaces, a ROS 2 container for the `ros2_control` controllers, and an Isaac Sim container for simulation. Dashed boxes are container boundaries. Each solid box groups the processes launched in that devcontainer. Arrows are ROS 2 topics/actions/services between containers or between processes in the same container. . . . . 23
3.2 UR10e manipulator with the attached Robotiq 2F-140 gripper, isolated from the simulation scene. The image shows the robot model used in the project without the surrounding gantry, workspace, or dynamic obstacles, in order to highlight the arm and end-effector configuration. . . . . 24
3.3 Comparison between the original sourced rack and the rack after the geometric modifications had been made. . . . . 27
3.4 Final simulation setup showing the modified flow rack, the object placement table used during the kitting task, the overhead support frame with the mounted robot, and the crate arrangement in the simulated environment. Some crates are not shown to improve visualisation. . . . . 28
3.5 Visual integration of a Gaussian splatting model of the research environment in Isaac Sim. The model was combined with the simulated robot and generated assets, but was not used in the evaluation. . . . 29
3.6 Overview of the hybrid planning flow. The global planner uses cuRobo MotionGen to generate a reference trajectory, while the local planner samples waypoints from this trajectory and uses cuRobo MPC to produce short-horizon control commands. . . . . 35
3.7 Schematic illustration of the recommended dual-camera setup for motion capture in Rokoko Vision. The figure shows a capture area with a diameter of approximately 4 m and two RGB cameras placed around the subject at different viewing angles. The illustration represents the general setup principle rather than the exact camera positions used in the recordings. . . . . 44

3.8 Example from the dual-camera motion-capture workflow in Rokoko. The two synchronised camera views used for recording are shown on the right, while the reconstructed 3D skeletal character is shown on the left. The recorded motion represents a picking movement intended to imitate a human operator retrieving an object from the flow rack. . 45
3.9 Comparison between the animated humanoid model in Isaac Sim and the corresponding simplified collision model used for planning. The Isaac Sim view shows the visual character in the simulation environment, while the RViz view shows the simplified collision geometry generated from the same skeleton pose. . . . . . . . . . . . . . . . . . 46
3.10 Personalised Avaturn character used as the visual humanoid model in the Isaac Sim factory environment. . . . . . . . . . . . . . . . . . . . 47

4.1 Pick-and-place workflow timing. The stacked bars show the mean global planning time, execution/control time, and remaining overhead. The whiskers show one standard deviation of the total workflow time over successful runs. For the hybrid planner, global planning time is not shown because planning and control are interleaved during execution. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 53
4.2 Per-phase planning time in the pick-and-place benchmark. Bars show the mean planning time over successful phase executions, and whiskers show one standard deviation. . . . . . . . . . . . . . . . . . . . . . . . 54
4.3 Total arm joint motion in the pick-and-place benchmark. The markers show the mean over successful runs, and the whiskers show one standard deviation. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 56
4.4 Total gantry motion in the pick-and-place benchmark. The markers show the mean over successful runs, and the whiskers show one standard deviation. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 56
4.5 Success rate for the hybrid dynamic obstacle benchmark grouped by robot clearance at obstacle spawn. The number above each bar shows the number of runs in that clearance range. . . . . . . . . . . . . . . . 57
4.6 Frame from the humanoid obstacle demonstration video. The humanoid reaches into the lower crate during the home_to_lower_crate motion, causing the hybrid planner to stop the robot arm until the path becomes clear again. . . . . . . . . . . . . . . . . . . . . . . . . 58

A.1 Between lower to lower crate. . . . . . . . . . . . . . . . . . . . . . . I
A.2 Home to lower crate. . . . . . . . . . . . . . . . . . . . . . . . . . . . I
A.3 Lower to lower crate. . . . . . . . . . . . . . . . . . . . . . . . . . . II
A.4 Lower to upper crate. . . . . . . . . . . . . . . . . . . . . . . . . . . II
A.5 Move above crates. . . . . . . . . . . . . . . . . . . . . . . . . . . . II

B.1 Home position. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . III
B.2 Pre grasp position. . . . . . . . . . . . . . . . . . . . . . . . . . . . III
B.3 Grasp position. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . IV
B.4 Grasp lift position. . . . . . . . . . . . . . . . . . . . . . . . . . . . IV
B.5 Pre drop position. . . . . . . . . . . . . . . . . . . . . . . . . . . . . IV

B.6 Release position. . . . . . . . . . . . . . . . . . . . . . . . . . . . . V
B.7 Release retreat position. . . . . . . . . . . . . . . . . . . . . . . . . . V
B.8 Return home position. . . . . . . . . . . . . . . . . . . . . . . . . . . V

# List of Tables

3.1 Evaluation criteria used to compare the planning algorithms. . . . . . 38

4.1 Success rate for the simple motion benchmark. . . . . . . . . . . . . . . 49
4.2 Planning time for the simple motion benchmark, reported as mean ± standard deviation in seconds over successful runs. . . . . . . . . . . . 50
4.3 Execution time for the simple motion benchmark, reported as mean ± standard deviation in seconds over successful runs. . . . . . . . . . 51
4.4 TCP movement in meters for the simple motion benchmark, reported as mean ± standard deviation in metres over successful runs. . . . . . 51
4.5 Arm joint movement for the simple motion benchmark, reported as mean ± standard deviation in degrees over successful runs. . . . . . . 52
4.6 Gantry movement for the simple motion benchmark, reported as mean ± standard deviation in metres over successful runs. . . . . . . 52
4.7 Overall success rate and most common failed phase for the pick-and-place benchmark. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 53
4.8 Per-phase success rate for the pick-and-place benchmark. . . . . . . . 55
4.9 Resource usage for the evaluated planners. . . . . . . . . . . . . . . . 59

C.1 Pick-and-place workflow timing, reported as mean ± standard deviation in seconds over successful runs. . . . . . . . . . . . . . . . . . . . VII
C.2 Per-phase planning time in the pick-and-place benchmark, reported as mean ± standard deviation in seconds over successful phase executions. VII
C.3 Total arm joint motion in the pick-and-place benchmark, reported as mean ± standard deviation in degrees over successful runs. . . . . . . VII
C.4 Total gantry motion in the pick-and-place benchmark, reported as mean ± standard deviation in metres over successful runs. . . . . . . VIII
C.5 Success rate for the hybrid dynamic obstacle benchmark grouped by robot clearance at obstacle spawn. . . . . . . . . . . . . . . . . . . . . . VIII

# 1
# Introduction

## 1.1 Background

Industrial production is increasingly moving toward flexible automation, where robots are expected to operate in environments that are less structured and more changeable than traditional robot cells. In such environments, robots may need to work near storage systems, workstations, tools, objects, and human operators. This places higher demands on the robot's motion-planning system, since planned motions must be feasible not only for the robot itself, but also with respect to the surrounding environment.

One task where these challenges become relevant is kitting. Kitting refers to the process of collecting and organising components required for later assembly operations. The task typically involves retrieving parts from storage locations and placing them into structured containers or kit trays. Although the task may appear simple, it requires the robot to move through a constrained workspace, reach into storage regions, avoid surrounding objects, and execute repeated pick-and-place motions reliably. If the environment changes during execution, for example, because a human operator or another object enters the robot's workspace, the planning system must also be able to handle these changes safely.

Simulation-based environments provide a useful way to investigate these problems before testing on physical hardware. In simulation, robot models, workspaces, obstacles, and task scenarios can be created and modified in a controlled way. This makes it possible to evaluate different planning approaches under repeatable conditions and to study how the system behaves when the environment changes during execution. Another important advantage is that simulation allows potentially unsafe situations to be tested without risking damage to human operators, the robot, or objects in the workspace. This is particularly relevant for shared-workspace scenarios, where collision avoidance and proximity to moving obstacles must be studied carefully before physical deployment. At the same time, modern simulation tools are becoming increasingly realistic through improved physics engines, more detailed robot models, realistic rendering, and closer integration with robotic middleware. As a result, the sim-to-real gap is gradually reduced, making simulation a useful intermediate step between algorithm development and real-world implementation.

A challenge in this type of system is motion planning in the presence of obstacles.

The planner must generate motions that satisfy the robot's kinematic limits while avoiding collisions with the surrounding environment. In static environments, the planner can generate a complete path before execution. In dynamic environments, however, a previously valid path may become invalid if an obstacle moves into the robot's workspace. This creates a need for planning approaches that can either generate new trajectories quickly or adapt the motion during execution.

Several motion-planning approaches can be used for this purpose. Sampling-based planners, such as those available through OMPL in MoveIt 2, are widely used for finding collision-free paths in high-dimensional configuration spaces. More recent GPU-accelerated planners, such as cuRobo, use parallel computation and trajectory optimisation to generate motions more quickly. Hybrid planning approaches combine a global planner, which generates an initial path, with a local planner, which can adapt the motion during execution. Comparing these approaches in the same simulated workspace can give insight into their relative strengths and limitations for kitting-like robot tasks in changing environments.

## 1.2 Purpose

The purpose of this thesis is to develop a realistic simulation-based evaluation environment for a robotic arm performing kitting-related motions in an industrial setting. The environment should be close enough to real hardware and production-like workspaces to be useful as a step between algorithm development, simulation testing, and later testing on physical robot systems.

The work focuses on integrating ROS 2, Isaac Sim, MoveIt 2, and GPU-accelerated planning frameworks in a containerised system architecture. A simulated industrial workspace is built with a gantry-mounted robotic arm, a flow-rack-based kitting setup, collision-aware environment models, and dynamic obstacles that represent moving objects or humans. Using simulation makes it possible to evaluate motion-planning methods in a safe, reproducible, and cost-efficient way. It also gives access to ground-truth information and controlled test conditions that are difficult to achieve in physical experiments.

The environment is used to study whether sampling-based, GPU-accelerated, and hybrid motion-planning methods are suitable for generating robot motions in constrained workspaces where the surroundings may change during execution. The evaluation looks at whether the planners can generate feasible and collision-aware motions, how quickly they respond to planning requests, how reliably they complete different kitting-related tasks, and how they behave when an obstacle blocks or invalidates the current path during execution.

Another reason for developing the simulation environment is that it can support future work on learning-based robotics methods. Since the environment contains controllable scenarios, repeatable experiments, dynamic obstacles, and simulation ground truth, it could later be used to generate training data, evaluate learned poli-

cies, and study sim-to-real transfer before testing on physical hardware.

The overall objective is to assess how suitable the evaluated planning methods are for robotic operation in dynamic industrial environments, where human presence or other moving obstacles must be considered. At the same time, the thesis gives a reusable simulation and evaluation platform that can support future development of both classical and learning-based methods for industrial robot motion generation.

## 1.3 Goals

The main goal of this thesis is to create a simulation-based system for evaluating motion-planning approaches for a robotic arm in a constrained kitting environment. The system should combine robot simulation, ROS 2-based control, motion planning, dynamic obstacle handling, and automated benchmarking in one integrated system.

More specifically, the goals are to:

- Construct a simulated kitting workspace containing a gantry-mounted UR10e robot arm, a Robotiq gripper, storage crates, a flow rack, and relevant collision geometry.
- Integrate the simulated robot with a ROS 2 control architecture so that joint commands, joint states, TF transforms, and planning information can be exchanged between Isaac Sim, MoveIt 2, and the planning components.
- Implement and configure several motion-planning approaches, including sampling-based planning with OMPL, GPU-accelerated planning with cuMotion and cuRobo, and hybrid planning using a global planner together with a local planner.
- Include static and dynamic obstacles in the planning environment, including simplified moving obstacles and a humanoid actor.
- Develop automated benchmark scenarios for evaluating planner performance across selected kitting-related motions, complete manipulation sequences, and dynamic changes in the planning environment.
- Collect performance data, including planning time, success rate, execution behaviour, trajectory movement, computational resource usage, and the ability of the system to react when new obstacles are introduced during execution.
- Compare the different planning approaches and discuss their suitability for robot operation in constrained environments where human presence or other dynamic obstacles must be considered.

## 1.4 Scope and Limitations

This thesis is limited to simulation-based development and evaluation. All experiments are performed in Isaac Sim, and no validation is carried out on a physical robot. The results, therefore, describe the behaviour of the implemented planning system in the simulated environment and should not be interpreted as a complete

verification of real-world robot performance.

The work focuses on motion planning, system integration, and benchmark-based evaluation. It does not attempt to solve the full task-level human-robot collaboration. Higher-level coordination strategies, such as assigning tasks between a human and a robot, negotiating shared workspace access, or dynamically reordering work between agents, are outside the scope of the thesis. Human presence is instead considered mainly as a dynamic obstacle that the robot planning system must take into account.

The humanoid actor used in the simulation is based on pre-recorded motion-capture animation and simplified collision geometry. The thesis does not include real-time human perception from cameras or other sensors, nor does it estimate human intention or predict future human motion. The humanoid model is therefore used to demonstrate how the planning system responds to a simulated human entering the workspace, rather than to represent a complete human-aware perception system.

The collision models used for planning and simulation are simplified. The robot, workspace objects, crates, flow rack, and humanoid actor are represented using approximated collision geometry such as boxes, spheres, and cylinders. This simplification was necessary to keep collision checking and simulation efficient, but it means that the collision representation does not perfectly match the visual geometry of all objects.

The planner comparison is limited to the selected planners and configurations implemented in the project. These include OMPL-based sampling planners, cuMotion, direct cuRobo planning, and the implemented hybrid planning setup. Other planning algorithms, parameter settings, and optimisation strategies are not investigated. The results should therefore be seen as a comparison of the implemented configurations rather than a general ranking of all available motion-planning methods.

The benchmark scenarios are designed to represent selected kitting-related motions and obstacle-interaction cases, but they do not cover all possible industrial kitting tasks. The evaluation focuses on specific aspects of planner performance, such as feasibility, responsiveness, execution behaviour, and robustness to environmental changes. The thesis does not evaluate complete production throughput, long-term reliability, hardware wear, operator ergonomics, or formal safety certification.

## 1.5 Contributions

This thesis contributes a simulation-based approach for evaluating reactive motion planning in robotic kitting tasks where the workspace can change during execution. The main contribution is the demonstration that a hybrid planner, combining global planning with local replanning, can allow a robot to recover when a previously

valid path becomes blocked by a dynamic obstacle. This provides practical insight into how reactive planning can support future industrial robot systems that need to operate near humans, objects, and equipment without relying on cages, fixed barriers, or fully separated workspaces. The work also provides an evaluation basis that can be reused to compare planning methods and further develop human-aware robot motion planning before physical robot deployment.

# 2
# Theory

## 2.1 Simulation in Robotics

Simulation plays a central role in modern robotics research and development. As robotic systems increase in complexity, the need for safe, scalable, and reproducible experimentation becomes more important. Simulation environments provide a controlled and configurable platform in which robotic systems can be designed, tested, and validated before deployment in physical hardware.
Simulation environments rely on mathematical models of rigid-body dynamics, kinematic chains, contact interactions, and collision detection. These models approximate the physical behaviour of robotic systems and their interaction with the environment. In motion planning, simulation serves as a geometric and dynamic ground truth that enables systematic evaluation of planning algorithms and collision avoidance strategies.
In this work, simulation is used as a representation of the robot and its environment. The simulation framework provides articulated robot models, collision geometry, and physics-based interaction, which together form the foundation for motion planning and collision checking.

## 2.2 Robot Kinematics and Configuration Space

This section introduces the concepts required to describe robot motion and formulate motion planning problems. Robot configurations are first defined through joint variables and degrees of freedom. This is followed by an overview of forward and inverse kinematics, which relate joint configurations to the end-effector's pose in the workspace. Finally, the concept of configuration space is introduced, giving a framework for expressing motion planning problems.

### 2.2.1 Degrees of Freedom and Joint Representation

A robot manipulator is composed of rigid links connected by joints. Each joint allows a specific type of motion and contributes one or more degrees of freedom (DoF) to the system. The total number of degrees of freedom determines the dimension of the robot's configuration space. [1]

The most common joint types in industrial manipulators are *revolute* and *prismatic* joints. A revolute joint allows rotational motion around a fixed axis and is typically

represented by an angle $\theta \in \mathbb{R}$, while a prismatic joint allows linear motion along an axis and is represented by a displacement $d \in \mathbb{R}$. Both of these joint types provide one degree of freedom. [1, 2]

The full configuration of a robot can be described by a set of joint variables, one for each degree of freedom. These variables are commonly grouped into a configuration vector $q \in \mathbb{R}^n$, where $n$ is the number of degrees of freedom of the robot. Each element of $q$ corresponds to the position of a single joint, such as an angle for a revolute joint or a linear displacement for a prismatic joint. Each joint may also have a joint limit which restrict the range of motion for each joint:

$$q_i^{\min} \leq q_i \leq q_i^{\max}.$$

[1, 2]

### 2.2.2 Forward and Inverse Kinematics

Kinematics describes the relationship between the robot's joint configuration and the position and orientation of its end-effector in the workspace [1, 2]. Forward kinematics computes the pose of the end-effector from a given joint configuration $q$. In this context, the pose consists of both position and orientation, and is commonly represented by a homogeneous transformation matrix (T)

$$T(q) \in SE(3),$$

whose rotational part describes orientation and whose translational part describes position in three-dimensional space [1, 2]. The rotational part is a rotation matrix, commonly denoted $R \in SO(3)$, which represents the orientation of the end-effector frame relative to a reference frame.

The forward kinematics of an open kinematic chain is obtained by chaining together the transformations associated with the joints and links of the robot. This can be expressed using homogeneous transformation matrices, for example, through the Denavit–Hartenberg convention, or through the product of exponentials formulation [1, 2].

In addition to rotation matrices, robot orientation can be represented using alternative parametrisations. A common three-parameter representation is given by Euler angles, in which the final orientation is obtained through a sequence of three successive rotations about specified axes [2]. A closely related representation is roll–pitch–yaw angles, which also describe orientation through ordered rotations about coordinate axes [2].

Another widely used representation in robotics and motion planning is the unit quaternion representation of orientation. A unit quaternion represents a rotation using four parameters

$$q = [q_0, q_1, q_2, q_3]^T,$$

subject to the constraint $\|q\| = 1$. It can be interpreted as a rotation of angle $\theta$ about a unit axis $\hat{\omega}$, where the quaternion is defined as

$$q = \begin{bmatrix} \cos(\theta/2) \\ \hat{\omega}\sin(\theta/2) \end{bmatrix}.$$

This representation avoids singularities that occur in three-parameter representations and provides a numerically stable way to represent orientation, at the cost of introducing one redundant parameter [1]. Regardless of the chosen representation, the full pose combines both position and orientation and can be expressed theoretically by a homogeneous transformation matrix in $SE(3)$ [1].

For an open kinematic chain, the position and orientation of the end-effector are uniquely determined by the joint configuration. The forward kinematics problem is therefore to determine the pose of the reference frame attached to the end-effector from the given joint values [1].

Inverse kinematics solves the opposite problem: given a desired pose of the end-effector, determine the joint configuration or configurations that achieve this pose. Unlike forward kinematics, inverse kinematics may have multiple solutions, a single solution, or no solution at all, depending on the robot geometry and the desired pose [2].

In practice, inverse kinematics is used to convert task-space goals, such as reaching a desired position and orientation, into joint-space targets for a motion planner or controller. Analytical solutions exist for some robot structures, while numerical methods are often used for more general or more complex manipulators. [1, 2]

### 2.2.3 Configuration Space Representation

The configuration space, denoted $\mathcal{C}$, is the set of all possible configurations that a robot can attain. Each point in $\mathcal{C}$ corresponds to a complete specification of the robot's configuration, typically given by the joint configuration vector $q \in \mathbb{R}^n$, where $n$ is the number of degrees of freedom. [1]

In general, the configuration space $\mathcal{C}$ does not behave as a standard Euclidean space. This is because some joint variables, such as rotational joints, are periodic, meaning that different numerical values can represent the same physical configuration (for example, angles differing by $2\pi$). As a result, the configuration space may have a curved or wrapped structure. For this reason, $\mathcal{C}$ is more accurately described as a manifold, which locally behaves like $\mathbb{R}^n$ but may have a different global structure. [3]

From a motion planning perspective, the configuration space can be viewed as the state space over which planning is performed. By representing the robot in this abstract space, complex geometric planning problems in the physical workspace can be transformed into problems of finding paths in $\mathcal{C}$. [1]

A key concept in motion planning is the partitioning of the configuration space into free space and obstacle space. The free space, denoted $\mathcal{C}_{\text{free}}$, consists of all configurations in which the robot does not intersect any obstacles in the environment. The obstacle space, denoted $\mathcal{C}_{\text{obs}}$, consists of all configurations that result in a collision. These two sets satisfy

$$\mathcal{C} = \mathcal{C}_{\text{free}} \cup \mathcal{C}_{\text{obs}}, \quad \mathcal{C}_{\text{free}} \cap \mathcal{C}_{\text{obs}} = \emptyset.$$

Motion planning is therefore restricted to finding solutions within $\mathcal{C}_{\text{free}}$. [3]

A path in configuration space is defined as a continuous function

$$\tau : [0, 1] \to \mathcal{C},$$

where $\tau$ denotes the path and $s \in [0, 1]$ is a scalar path parameter. For each value of $s$, the path returns a configuration

$$q = \tau(s).$$

The parameter $s$ can be interpreted as progress along the path, where $s = 0$ corresponds to the start configuration and $s = 1$ corresponds to the goal configuration. It is important to note that a path is a function rather than only a set of points, meaning that each point along the path is associated with a specific parameter value $s$. Continuity ensures that nearby values of $s$ correspond to nearby configurations, so that the path does not contain jumps or discontinuities. [3]

### 2.2.4 Obstacle Representation and Collision Detection

In motion planning, obstacles are typically defined in the workspace and then implicitly mapped into configuration space. A configuration belongs to the obstacle region $\mathcal{C}_{\text{obs}}$ if the robot, placed at that configuration, intersects with any obstacle in the environment [1]. The set of all such configurations forms the so-called *C-obstacles*. In practice, explicitly constructing $\mathcal{C}_{\text{obs}}$ is generally infeasible for high-dimensional systems, and motion planning algorithms instead rely on performing collision checks for individual configurations [1].

Collision detection, therefore, becomes an important operation in motion planning. Given a configuration $q$, a collision detection function determines whether $q \in \mathcal{C}_{\text{free}}$ or $q \in \mathcal{C}_{\text{obs}}$. Since this operation is executed repeatedly during planning, its computational efficiency significantly impacts overall performance [3].

To improve efficiency, it is common to approximate robot and obstacle geometries using simplified representations. For example, bounding volumes such as spheres or boxes can be used in an initial broad phase to quickly rule out non-colliding configurations before performing more detailed checks [3]. This introduces a trade-off between geometric accuracy and computational cost.

Modern motion generation frameworks extend this idea further by designing the entire collision checking pipeline around such simplified representations. In cuRobo, for instance, the robot geometry is approximated by a set of spheres. This allows collision checking to be reduced to efficient distance computations, such as sphere–sphere distances for self-collision and point-to-environment distance queries for obstacle avoidance. By avoiding expensive mesh-based collision checks, this approach enables fast and highly parallel collision evaluation on GPUs [4].

In addition to simplifying the robot geometry, modern systems also support multiple environment representations, such as bounding boxes, triangle meshes, or signed distance fields. These representations give different trade-offs between accuracy and computational efficiency, and can be selected based on the application. [4]

This formulation provides the foundation for motion planning, where the objective is to compute a path that connects a start and goal configuration while remaining within $\mathcal{C}_{\text{free}}$.

## 2.3 Motion Planning

Building on the configuration-space formulation introduced above, motion planning concerns the computation of robot motions that move the system from an initial configuration to a desired goal configuration while satisfying relevant constraints. These constraints include collision avoidance, joint limits, and, in more general settings, dynamic and control limitations.

The following section first presents the formal problem formulation and then introduces the main classes of planning methods considered in this work.

### 2.3.1 Problem Formulation

Given the configuration space $\mathcal{C}$ and its collision-free subset $\mathcal{C}_{\text{free}}$, the motion planning problem can be defined as follows. A start configuration $q_{\text{start}} \in \mathcal{C}_{\text{free}}$ and a goal configuration $q_{\text{goal}} \in \mathcal{C}_{\text{free}}$ are specified, and the objective is to determine a path

$$\tau : [0,1] \to \mathcal{C}_{\text{free}}$$

such that

$$\tau(0) = q_{\text{start}}, \quad \tau(1) = q_{\text{goal}}.$$

[3]

In this formulation, the problem is purely geometric and is commonly referred to as path planning. The goal is to find a continuous sequence of configurations that avoids collisions, without explicitly considering the time required to execute the motion or the dynamics of the robot [1]. In addition to avoiding collisions, the path must satisfy constraints imposed by the robot, such as joint limits and other kinematic or dynamic restrictions. As a result, valid solutions must lie entirely within

the feasible subset of $\mathcal{C}_{\text{free}}$ [3].

In practice, although paths are defined as continuous functions, computational methods operate on discretised representations. A trajectory, defined as a path together with a time scaling that specifies when each configuration is reached [1], is therefore typically approximated as a finite sequence of configurations:

$$\theta_{[0,T]} = [\theta_0, \theta_1, \ldots, \theta_T], \quad \theta_t \in \mathbb{R}^n,$$

where each $\theta_t$ corresponds to the robot configuration at a discrete time step. [4]

A distinction is often made between feasibility and optimality in motion planning. A feasible solution is any path that satisfies the constraints and remains collision-free. An optimal solution, on the other hand, minimises a cost function defined over trajectories. Common cost criteria include path length, smoothness, energy consumption, or execution time [1].

Motion planning problems can also be categorised based on how they are solved in practice. In an offline setting, the environment is assumed to be static, and the planner is allowed more time to compute a solution. In contrast, online planning requires fast replanning in response to changes in the environment or task, which places stronger requirements on computational efficiency [1].

Although paths are defined theoretically as continuous functions in configuration space [3], computational planning methods often approximate them using discrete waypoints or time-indexed configurations. This allows the planning problem to be handled numerically while approximating the underlying continuous motion [4].

### 2.3.2 Sampling-Based Planning

Sampling-based planning methods address the motion planning problem by avoiding an explicit construction of the obstacle region $\mathcal{C}_{\text{obs}}$ in the configuration space. Instead, they rely on sampling configurations and using collision checking to determine feasibility, thereby constructing a representation of the free space indirectly through samples and connections between them. [5]

More formally, these methods generate samples in the configuration space and keep those that lie in $\mathcal{C}_{\text{free}}$. A graph or tree structure is then incrementally constructed by connecting nearby collision-free samples. The resulting structure represents a set of feasible trajectories, from which a solution path can be extracted. [5]

A key distinction within sampling-based planning is between multi-query and single-query methods. Probabilistic Roadmaps (PRM) belong to the multi-query category and construct a reusable graph that captures the connectivity of the free space. In contrast, Rapidly-exploring Random Trees (RRT) are single-query methods that incrementally build a tree starting at the initial configuration, expanding toward

randomly sampled configurations until a feasible path to the goal is found. [5]

An important theoretical property of sampling-based planners is probabilistic completeness, meaning that if a feasible path exists, the probability of failure decreases to zero as the number of samples increases [5]. However, basic variants such as PRM and RRT do not guarantee optimal solutions. It has been shown that the cost of the solution returned by these algorithms converges to a non-optimal value as the number of samples increases [5].

### Probabilistic Roadmaps (PRM)

PRM constructs an undirected graph, referred to as a roadmap, that captures the connectivity of the free configuration space. The algorithm operates in two phases: a learning phase and a query phase [6].

During the learning phase, configurations are sampled from the configuration space, and those that lie in the free space are added as roadmap vertices. Each vertex is then connected to nearby vertices, and connections are retained only if they are collision-free. This results in a graph that approximates the connectivity of the free space [6].

In the query phase, the start and goal configurations are connected to the roadmap, after which a graph search algorithm is used to find a path between them. PRM is particularly well-suited for multi-query problems, where multiple planning queries will be solved in the same environment. As the number of samples increases, the roadmap provides a more accurate approximation of free-space and retains probabilistic completeness [6, 5].

### Rapidly-exploring Random Trees (RRT)

In contrast to PRM, RRT is an incremental, single-query algorithm that builds a tree rooted at the initial configuration. At each iteration, a random configuration is sampled, and the nearest node in the existing tree is identified. The tree is then extended toward the sampled configuration, typically by generating a new point in that direction using a local steering procedure. If the resulting motion is collision-free, the new configuration is added to the tree. [5, 7]

A key property of RRT is its exploration behaviour. The algorithm tends to expand toward previously unexplored regions of the configuration space, which allows it to rapidly cover large portions of the space in the early stages of planning [7]. The algorithm continues until a configuration is found that lies within the goal region or until a computational limit is reached. Like PRM, RRT is probabilistically complete, but it does not guarantee optimal solutions [5].

**Optimal Variants: PRM$^*$ and RRT$^*$**

Standard PRM and RRT are not asymptotically optimal, meaning that the solution does not necessarily converge to the optimal path as the number of samples increases. To address this limitation, asymptotically optimal variants, PRM$^*$ and RRT$^*$, have been introduced. [5]

In PRM$^*$, asymptotic optimality is achieved by modifying the connection strategy such that each sample is connected to nearby nodes within a radius that depends on the number of samples. Specifically, the connection radius scales proportionally to $(\log n/n)^{1/d}$, where $n$ is the number of samples and $d$ is the dimension of the configuration space. In RRT$^*$, the algorithm incrementally improves the solution by selecting the parent that yields the lowest path cost and by performing a rewiring step, in which nearby nodes are reconnected if a lower-cost path is found [5].

**Limitations**

Sampling-based planners are particularly effective in high-dimensional spaces, where explicit representations of the configuration space are very expensive. Their reliance on collision checking allows them to scale well with complex environments. [5]

However, the motions produced by grid-based and sampling-based planners are often jerky and may require post-processing or smoothing to improve their quality. This motivates the use of trajectory-optimisation-based methods, which instead aim to directly optimise trajectories with respect to a defined objective. [1]

### 2.3.3 Trajectory-optimisation-based planning

Trajectory optimisation formulates motion planning as a nonlinear optimisation problem, where the aim is to find a feasible motion that minimises a given objective while satisfying constraints such as system dynamics, collision avoidance, and boundary conditions. [1]
In this work, the robot motion is represented in joint space as a discrete sequence of configurations, and cuRobo is used as the trajectory-optimisation-based planner.

**Trajectory representation and objective**

A trajectory in joint space can be written as

$$\theta_{[0,T]} = [\theta_0, \theta_1, \ldots, \theta_T], \qquad \theta_t \in \mathbb{R}^n,$$

where $n$ is the number of robot joints, $\theta_t$ is the full joint configuration at time step $t$, $\theta_0$ is the start configuration, and $\theta_T$ is the final configuration. Thus, $\theta_{[0,T]}$ denotes the complete joint-space trajectory, while each $\theta_t$ denotes one robot state along that trajectory. [4]

The optimisation is typically carried out over the states after the fixed initial configuration, that is, over $\theta_{1:T}$. In general, trajectory optimisation can be written as

minimising an objective of the form

$$J(\theta_{1:T}) = C_{\text{task}} + C_{\text{smooth}} + C_{\text{limits}} + C_{\text{collision}},$$

where the different terms penalise different undesirable properties of the motion. The task cost encourages the robot to reach the desired goal, for example, by penalising the final end-effector pose error. The collision cost penalises self-collisions and collisions with the environment. The limit-related cost penalises violations of joint, velocity, acceleration, or jerk limits. The smoothness cost encourages gradual variation along the trajectory rather than abrupt changes between neighbouring states. [8] In practice, this is commonly achieved by penalising large velocities, accelerations, jerks, or finite differences between consecutive waypoints. A smooth trajectory is therefore one in which the motion changes gradually over time. [1]

In cuRobo, motion generation can be viewed as a time-discretised trajectory optimisation problem over a joint-space trajectory. In simplified form, this can be written as

$$\min_{\theta_{1:T}} \; C_{\text{task}}\Big(X(\theta_T), X_g\Big) + \sum_{t=1}^{T} C_{\text{smooth}}(\cdot), \tag{2.1}$$

where $\theta_{1:T}$ denotes the planned sequence of joint configurations and the initial state $\theta_0$ is treated as fixed. The term $X(\theta_T)$ denotes the end-effector pose obtained from forward kinematics at the final configuration, while $X_g$ denotes the desired goal pose in task space. The task cost penalises the difference between the final end-effector pose and the goal pose, while the smoothness term encourages a smooth trajectory, for example, with respect to velocity, acceleration, and jerk. The optimisation also accounts for joint position, velocity, acceleration, and jerk limits, as well as self-collision and world-collision avoidance [4].

### Seed trajectory generation

Trajectory optimisation is a local optimisation problem and therefore needs an initial guess from which the optimisation starts. Such an initial guess is called a *seed trajectory*. A seed trajectory is therefore an initial full joint-space motion, from the start state toward the goal, which is later refined by the optimiser.
Each IK solution from the IK solver provides a candidate joint-space goal configuration $q_{\text{goal}}$. From these candidate goals, seed trajectories can be constructed, which serve as starting points for later optimisation. [9]
Several strategies can be used to generate such seeds.

- **Straight-line joint interpolation:** A simple seed is obtained by linearly interpolating in joint space from $q_{\text{start}}$ to $q_{\text{goal}}$ over $T$ waypoints. [10]

- **Interpolation via a retract configuration:** A seed can also be generated by first moving to a predefined safe or convenient posture $q_{\text{retract}}$, and then from there to the goal:

$$q_{\text{start}} \to q_{\text{retract}} \to q_{\text{goal}}.$$

This can help avoid poor direct initialisations in cluttered scenes. [10]

- **Geometric planner seed:** A seed can also be constructed using a parallel geometric planner in joint space. First, a simple heuristic connection is attempted, such as direct start-to-goal interpolation or motion via a retract configuration. If these fail, a sample collision-free joint configuration connects nearby nodes and searches for a collision-free path through the resulting graph. A shortcutting step is then applied to remove unnecessary waypoints and produce a better initial path. [4]

**Particle-based refinement of seeds**

Optimisation-based motion planning methods are generally sensitive to the initial guess. As noted in *Modern Robotics* [1], such methods can generate smooth near-optimal motions, but they may also become trapped in local minima in cluttered configuration spaces, which makes good initialisation important.

Different trajectory-optimisation methods address this issue in different ways. For example, TrajOpt [8] uses primarily local optimisation methods and therefore depends strongly on the quality of the initial trajectory, while stochastic approaches such as STOMP [11] use sampling to improve robustness to poor initialisation and local minima.

In cuRobo, an intermediate stochastic refinement stage is used before the gradient-based optimisation. Rather than applying local gradient-based updates directly to a raw seed trajectory, cuRobo first explores its neighbourhood using a stochastic particle-based optimiser. This allows multiple nearby candidate trajectories to be sampled and evaluated, making it possible to move the trajectory toward a more promising region of the objective landscape before local gradient-based refinement is applied. [12, 4]
For one seed, the optimiser maintains a mean trajectory $\mu$, initially set to the seed trajectory:

$$\mu \leftarrow Q_{\text{seed}}.$$

It then iteratively samples perturbed trajectories around this mean. [4]
In each iteration, the following steps are performed:

1. **Sample candidates:** from the current mean trajectory $\mu$ and covariance $\sigma$, generate $K$ candidate trajectories

$$Q^{(i)} = \mu + \sigma^{1/2}\epsilon^{(i)}, \qquad \epsilon^{(i)} \sim \mathcal{N}(0, I), \quad i = 1, \dots, K.$$

   Each particle therefore represents a complete perturbed trajectory, not just a single state. [4]
2. **Evaluate cost:** compute the trajectory cost $J(Q^{(i)})$ for each candidate. In cuRobo, this cost reflects the same general objective discussed above, including terms related to goal error, collisions, smoothness, and joint-limit behaviour. [4]

3. **Compute weights:** convert the costs into weights so that lower-cost trajectories receive higher influence:

$$w_i = \frac{\exp\Big(-J(Q^{(i)})/\beta\Big)}{\sum_{j=1}^{K} \exp(-J(Q^{(j)})/\beta)}.$$

[12, 4]

4. **Update the trajectory distribution:** update the mean trajectory and covariance using the weighted particles:

$$\mu \leftarrow (1 - k_\mu)\mu + k_\mu \sum_{i=1}^{K} w_i Q^{(i)},$$

while the covariance is also updated based on the weighted spread of the sampled trajectories. [4]

After a small number of iterations, this stage outputs a refined trajectory for each seed. [13, 14]

### Gradient-based refinement

Motion planning can be formulated as a nonlinear optimisation problem. Such problems may be solved using gradient-based methods, such as sequential quadratic programming (SQP) [15], or non-gradient methods, such as simulated annealing, Nelder–Mead optimisation [16], and genetic programming. For gradient-based methods, a locally optimal solution can often be obtained when the optimisation is started from a sufficiently good initial guess [1]. In cuRobo, this refinement is performed using L-BFGS, a gradient-based quasi-Newton method [17]. Unlike standard gradient descent, L-BFGS uses gradient evaluations together with a limited-memory approximation of curvature information to compute more effective search directions for local optimisation [4, 17].

For one candidate trajectory, the optimisation variables are all joint values across all time steps. If the trajectory has $T$ waypoints and the robot has $n$ joints, the trajectory can be flattened into a vector in $\mathbb{R}^{Tn}$. The optimiser therefore updates the whole trajectory jointly across time, rather than modifying one waypoint independently. [10, 18]

At the current iterate $x_k$, the trajectory cost and its gradient are evaluated:

$$g_k = \nabla f(x_k),$$

where $f(x_k)$ denotes the current trajectory objective. Because the objective contains task, collision, smoothness, and limit-related terms, the gradient reflects how all these factors change when the trajectory is perturbed. [4]

A basic gradient-descent method would move in the steepest descent direction

$$p_k = -g_k.$$

L-BFGS improves upon this by also using information from previous iterations. More specifically, it stores a limited history of

$$s_k = x_{k+1} - x_k, \qquad y_k = g_{k+1} - g_k,$$

which describes how the trajectory and gradient changed between consecutive iterations. From these vectors, L-BFGS builds an approximation of the local curvature and computes a search direction of the form

$$p_k \approx -H_k^{-1} g_k,$$

where $H_k^{-1}$ is an approximate inverse Hessian. [19]
After a search direction has been computed, a line search is used to determine how far to move along that direction. Instead of evaluating only one step length, several candidate step sizes are evaluated in parallel:

$$x_k(\alpha) = x_k + \alpha p_k.$$

The best acceptable step is then selected according to the line-search conditions. These step sizes can be evaluated in parallel using a GPU. [4]
The refinement then repeats the same pattern: evaluate cost and gradient, compute an L-BFGS direction, test candidate step lengths, accept an update, and continue until convergence or until the iteration budget is exhausted. [4]

**Post-optimisation trajectory selection**

After the final refinement stage, multiple optimised candidate trajectories may remain. These are then checked for feasibility and goal accuracy, and invalid candidates are discarded. In cuRobo, each candidate is interpolated and assigned an estimated timestep based on the joint velocity, acceleration, and jerk limits. [10]
The valid trajectories are then ranked according to criteria related to task-space accuracy, smoothness, and motion duration. The best-ranked trajectory is selected as the motion-planning result. In cuRobo, this selected trajectory may also undergo a final time-step refinement step, resulting in the final collision-free trajectory returned by the planner. [4, 9]

### 2.3.4 Planning with Continuous Updating

In dynamic environments, it is generally insufficient to compute a motion plan once and execute it unchanged over the full duration of a task. Changes in the environment, such as moving obstacles or varying workspace conditions, can render an initially feasible trajectory inefficient or infeasible during execution. The robot must therefore be able to adapt its motion online based on updated state information. To address this, planning approaches with continuous updating are employed. These methods repeatedly predict the future evolution of the system over a finite horizon, update the plan using the current state, and execute only the first part of the solution before replanning. This receding-horizon strategy forms the basis of Model Predictive Control and related approaches.

#### 2.3.4.1 Model Predictive Control

In a general discrete-time formulation, the system dynamics are written as

$$x_{k+1} = f(x_k, u_k), \tag{2.2}$$

where $x_k$ denotes the system state at prediction step $k$, $u_k$ the control input, and $f(\cdot)$ the model used to predict the future system evolution [20]. Based on the current state $x_0 = x_{\text{current}}$, MPC predicts the state trajectory over a finite horizon of length $N$. The controller then determines the control sequence

$$u_{0:N-1} = \{u_0, u_1, \ldots, u_{N-1}\} \tag{2.3}$$

that minimises a cost function of the form

$$J = \sum_{k=0}^{N-1} \ell(x_k, u_k) + \ell_f(x_N), \tag{2.4}$$

where $\ell(x_k, u_k)$ is the stage cost and $\ell_f(x_N)$ is the terminal cost. The stage cost typically penalises state deviation from a desired reference, control effort, or other undesired behaviour along the horizon, while the terminal cost accounts for the predicted state at the final step [20].

The optimisation is solved subject to the system dynamics and admissibility constraints,

$$x_{k+1} = f(x_k, u_k), \quad k = 0, \ldots, N-1, \tag{2.5}$$

$$x_k \in \mathcal{X}, \quad u_k \in \mathcal{U}, \tag{2.6}$$

$$x_0 = x_{\text{current}}, \tag{2.7}$$

where $\mathcal{X}$ and $\mathcal{U}$ denote the sets of admissible states and control inputs. In practice, these constraints may represent actuator limits, velocity or acceleration bounds, workspace boundaries, or other physical restrictions. The explicit inclusion of such constraints is one of the main strengths of MPC [21].

At each sampling instant, the current state is first measured or estimated, after which an optimization problem is solved to obtain the control sequence that minimizes the objective function while satisfying the imposed constraints. However, only the first control input $u_0^*$ is applied to the system. At the next sampling instant, the state is updated, the horizon is shifted forward, and the optimisation is solved again. In this way, MPC implements the receding-horizon principle through repeated online optimisation [22].

A key advantage of MPC is that it combines prediction and constraint-aware control within a unified framework. However, the repeated solution of the associated optimisation problem can become computationally demanding, particularly for nonlinear or high-dimensional systems [20]. This limitation becomes more pronounced in settings that require rapid online updates and continuous local re-optimisation during execution. For such cases, other predictive control formulations can be considered within the same receding-horizon principle.

#### 2.3.4.2 Model Predictive Path Integral

Model Predictive Path Integral (MPPI) can also be applied within an MPC setting. In cuRobo, this differs from its earlier use in particle-based seed refinement. While the underlying sampling-based update principle remains the same, the quantity being optimised is no longer a complete candidate trajectory intended for later selection. Instead, MPPI is applied to a finite-horizon control sequence within the MPC loop, where the horizon determines how many future steps are considered in each optimisation [14].
At each control step, the control sequence is updated using the sampling, cost-weighting, and weighted averaging procedure described in Section 2.3.3, but here it is applied over the finite prediction horizon used by MPC. Only the first control action of the optimised sequence is then applied, after which the optimisation is repeated from the updated state. In this way, MPPI is incorporated into a receding-horizon control framework rather than serving as an offline trajectory-refinement stage [12].

The use of MPPI in this setting is motivated by the need for repeated local re-optimisation during execution. By embedding the method within the MPC loop, the controller can continuously adapt the commanded motion to the current state and surrounding conditions without committing to a full trajectory in advance. MPPI thus represents one example of an alternative predictive control formulation within the same receding-horizon principle. This makes the approach well-suited for reactive, obstacle-aware control over a limited horizon [23].

# 3

# Methods

This chapter presents the approach used to design and implement the simulation-based robotic kitting system. The focus is on the overall system architecture, the integration between simulation, control, and motion planning, and the construction of the simulated environment that was used.

## 3.1 Project Structure

The project was implemented as a containerised software architecture built around ROS 2, Isaac Sim [24], and GPU-accelerated motion planning frameworks such as cuRobo. A container packages software and dependencies into an isolated runtime environment. The system was separated into multiple cooperating containers rather than a single large environment. This design choice was motivated by the partially conflicting dependencies of the involved frameworks. Isaac Sim requires a GPU-enabled runtime and graphical environment, whereas ROS 2 and MoveIt 2 rely on separate middleware and robotics libraries. While the OMPL-based planning baseline can run on the CPU, the complete simulation and GPU-accelerated planning setup requires an NVIDIA GPU with CUDA support, where CUDA is NVIDIA's GPU programming platform. Containerization was therefore used to isolate these environments, making sure it is reproducible and simplifying deployment across different computers.

In addition, the separation between the simulation layer and the control layer allows the simulation environment to be replaced by a real robotic system without requiring major changes to the overall architecture. This makes it possible to transfer the developed system from a simulated setup to a real-world deployment by replacing the simulation container with a hardware interface while keeping the remaining components unchanged.

The system consists of three primary runtime environments. The control layer is implemented in a ROS 2 container that provides the robot description, controllers, and motion-planning interfaces via MoveIt 2. This container acts as the central integration layer and is responsible for coordinating execution and communication between components. The simulation layer is implemented in a separate container running Isaac Sim, where the robot, workspace, and dynamic obstacles are defined. This environment exposes a ROS 2 bridge, which is the interface that translates simulation data to and from ROS 2 messages, making bidirectional communication

between the simulated world and the control system possible. The planning layer is implemented in a dedicated container, which provides access to multiple planning backends, including sampling-based planners and GPU-accelerated methods. Figure 3.1 shows the overall software architecture of the project. The dashed boxes represent the three runtime containers, while the solid boxes show the main processes and functional components launched inside each container. The arrows indicate the main ROS 2 communication paths between planning, control, simulation, and environment updates.

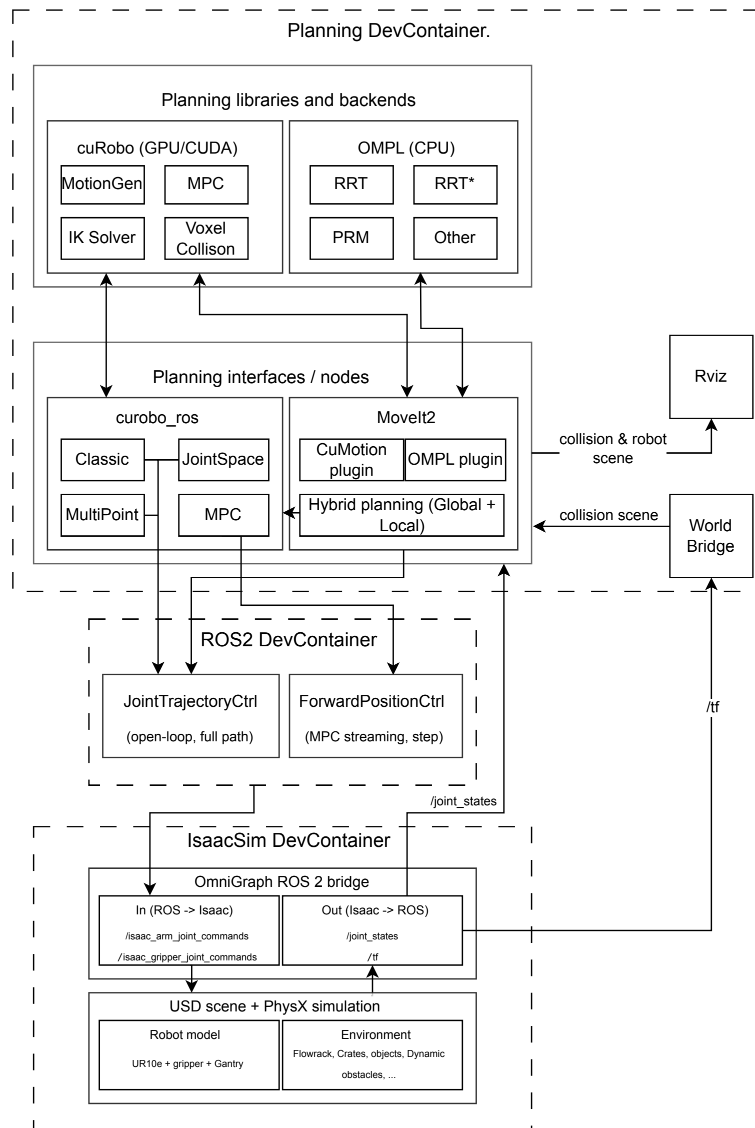



**Figure 3.1:** Overview of the project software architecture. The system is divided into three runtime containers: a planner container for planning libraries and planning interfaces, a ROS 2 container for the `ros2_control` controllers, and an Isaac Sim container for simulation. Dashed boxes are container boundaries. Each solid box groups the processes launched in that devcontainer. Arrows are ROS 2 topics/actions/services between containers or between processes in the same container.

All runtime containers share a common ROS 2 workspace and communicate through ROS 2 topics, services, and actions. Host networking enables communication between containers, allowing the system to act as a unified application.

The system was developed as an extension of the repository `ur10e_2f140_topic_based_ros2_control`, which provides an initial integration of

a UR10e robot arm, a Robotiq 2F-140 gripper, ROS 2 control, and Isaac Sim in a containerised setup [25]. The base repository supplied the robot model, the initial topic-based ROS 2 control interface, and an initial dual-container architecture. The robot model was described using URDF/Xacro files. URDF, the Unified Robot Description Format, is the standard robot description format used in ROS to define a robot's links, joints, visual geometry, collision geometry, and inertial properties. Xacro is an XML macro language used to generate URDF files from reusable components, which makes larger robot descriptions easier to maintain. In this project, this was useful because the complete robot model consisted of several connected parts, including the gantry, the UR10e arm and the Robotiq gripper. This project extends the base repository by introducing an additional planning container, integrating multiple motion planning backends, supporting hybrid planning strategies, and incorporating dynamic obstacle scenarios. The UR10e manipulator and attached Robotiq 2F-140 gripper are shown in Figure 3.2.

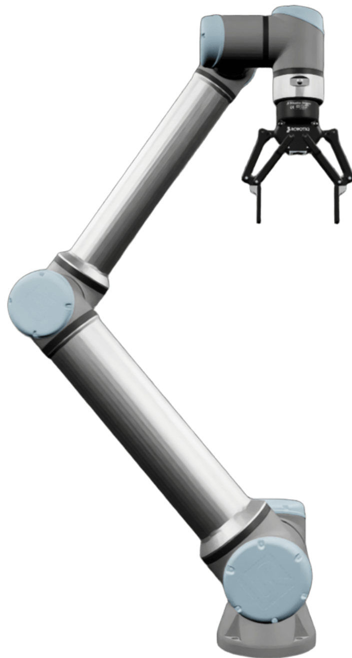


**Figure 3.2:** UR10e manipulator with the attached Robotiq 2F-140 gripper, isolated from the simulation scene. The image shows the robot model used in the project without the surrounding gantry, workspace, or dynamic obstacles, in order to highlight the arm and end-effector configuration.

## 3.2 ROS 2 Integration

Robot Operating System 2 communication was used across all three containers described in the previous section. This allowed the different components to work together as one integrated system, even though they were separated across different environments. ROS 2 provides a common communication structure for modular

robotic systems, allowing separate software components to exchange data and coordinate their operation as part of the same overall application [26]. Communication in ROS 2 is organised through a distributed architecture in which functionality is divided into separate processes, referred to as nodes. These nodes can exchange information through several communication patterns, most notably topics for publish-subscribe communication, services for request-response interactions, and actions for longer-running tasks with execution feedback [26]. In this project, ROS 2 was used for communication between simulation, planning, and control, allowing robot state information, commands, motion plans, and environment updates to be exchanged between the different components.

On the simulation side, Isaac Sim was connected to the ROS 2 network through the topic-based `ros2_control` hardware interface. Instead of communicating with physical robot hardware, this interface exchanged commands and state information through ROS 2 topics. Joint commands for the manipulator and gripper were sent from the control system to Isaac Sim, while Isaac Sim published the current simulated joint states back to the ROS 2 network. In this way, the simulated robot could be controlled through the same ROS 2-based control structure that would also be used for a physical system. This is important because it makes it possible to develop planners safely and reliably in simulation and then transfer the same code more easily to a real robot.

The controller layer was implemented through `ros2_control`, which ran inside the ROS 2 container and acted as the main interface between higher-level planning components and low-level robot actuation. Several controllers were configured for different execution modes. A joint trajectory controller was used for conventional trajectory execution, while a forward position controller was used for streaming commands when MPC was used. Separate controllers were also used for the gripper, together with a joint state broadcaster for publishing the current robot state.

MoveIt 2 was integrated into the ROS 2 network through the `move_group` node, which served as the main motion-planning interface for standard planning requests. ROS 2 was also used to publish collision information from the simulated environment into the MoveIt 2 planning scene, allowing the planner to take both static and exported environment geometry into account.

The GPU-based planning layer was implemented around a custom ROS 2 node referred to as the unified planner node. This node acted as the interface to the cuRobo-based planning backends and exposed its functionality through ROS 2 services and actions. Trajectory generation was requested through a service interface that accepted the current robot state together with a target pose or target joint configuration and returned a collision-free trajectory. The unified planner node subscribed to the current joint states from the ROS 2 network and published streaming joint commands to the active controller when MPC-based execution was used.

Overall, ROS 2 functioned as the communication backbone of the project. It pro-

vided the distributed runtime structure needed to connect simulation, planning, and control across multiple containers.

## 3.3 Simulation

Simulation was used since it allowed the robotic system to be tested in a controlled, safe and repeatable environment. In this project, NVIDIA Isaac Sim was used as the simulation engine and acted as the main representation of the robot and its surroundings during runtime. By maintaining the world model, Isaac Sim provided the system with a consistent description of the simulated environment for planning and control [27]. Isaac Sim is built on NVIDIA Omniverse, which provides the underlying framework for rendering, physics simulation, and extensible application development [27, 28, 29].

The simulated environment was represented using Universal Scene Description (USD), which is a format for describing 3D scenes, including objects, geometry, materials, transformations, and relationships between scene elements. A useful property of USD is its composition model, which allows a scene to be built from several referenced assets and layers instead of a single self-contained file. This made it possible to keep robot models, environment assets, and other scene elements separate and combine them into a complete simulation scene when needed [30]. Physics simulation in Isaac Sim is based on the NVIDIA PhysX engine, which provides real-time simulation of articulated robots, rigid bodies, and contacts between objects [31, 32, 33].

The simulation scene was built from a set of reusable USD assets. The main scene contained the UR10e robot, the Robotiq 2F-140 gripper, the gantry structure, and the surrounding workspace. Static environment objects such as the flow rack, crates, and structural elements were added as separate USD assets. The scene was built in a modular way, with the base robot asset kept separate and the surrounding environment added on top via USD composition. This made it easier to modify the workspace or create scene variants without changing the original robot asset. Several scene variants were used in the project, including a base environment without moving obstacles, a version with a dynamic cylinder obstacle, and a version with a humanoid obstacle.

The simulated robot was connected to the control system through a topic-based `ros2_control` interface. Instead of communicating with physical robot hardware, the simulator exchanged commands and state feedback through ROS 2 topics. Joint commands generated by the control and planning layers were sent to Isaac Sim, and Isaac Sim published the current joint states back to the ROS 2 network. In this way, the simulator acted as a substitute for the physical robot while still using the same overall control structure.

On the Isaac Sim side, the ROS 2 communication was implemented through an Action Graph, Isaac Sim's node-based visual scripting graph, configured in the simulation scene. This graph was evaluated at each simulation tick. At every update

step, it read the latest joint commands coming from the ROS 2 container and applied them to the simulated robot. After the robot state had been updated in the simulator, the graph published the resulting joint states back to the ROS 2 network. In addition, it published TF transforms, which are ROS 2 coordinate-frame transformations that describe the poses of robot links and objects relative to one another. This allowed the rest of the system to continuously receive updated state and pose information directly from the simulator during runtime.

### 3.3.1 Simulation Environment

The simulated environment was made using a combination of sourced and custom-made assets. The visual models of the crates and the flow rack were taken from GrabCAD [34, 35] with the creator's permission, while an overhead support beam for mounting the robot arm was designed and modelled in Fusion 360 [36]. The UR10e arm was inherited from the base repository, where it was based on the upstream description provided by Universal Robots, and was locally integrated with a Robotiq gripper model.

The flow-rack model was first imported into Fusion 360, where its geometry was modified to resemble an industrial kitting station better. In particular, the rack was extended in width, and the upper shelf was made shallower. These modifications were introduced to create clearance for the robot arm, which was mounted above the rack on the overhead support beam, while still allowing access to the lower crates. Figure 3.3 shows a comparison between the original sourced rack and the rack after the modifications had been made.

(a) Original sourced flow-rack model.

(b) Flow rack after the geometric modifications had been applied.

**Figure 3.3:** Comparison between the original sourced rack and the rack after the geometric modifications had been made.

After the modifications, the assets were imported into Blender [37] for collision modelling. A separation was made between visual and collision representations

throughout this stage. The CAD-derived meshes were used for visual rendering, while simplified box-based collision geometries were created for physical interaction and collision checking. For the flow rack, box-shaped collision primitives were used to approximate the structural elements and shelf regions. For the crates, several box colliders were combined to represent the bottom and side walls. This simplification reduced collision complexity and made the simulation more computationally efficient.

The final assets were then exported in USD format and imported into Isaac Sim. In the simulation environment, the flow rack and the overhead support frame were defined as static objects, while the crates were configured as dynamic bodies and assigned mass. A table was also imported and positioned in front of the flow rack to serve as the object placement area for the robot during kitting tasks. The robot was mounted on the overhead support frame, allowing it to reach both the flow rack and the table during pick-and-place execution. The friction properties of the shelves' collision surfaces were reduced to approximate the behaviour of a roller-based rack. This allowed the crates to slide more easily along the inclined shelf levels. The final simulation setup is shown in Figure 3.4.

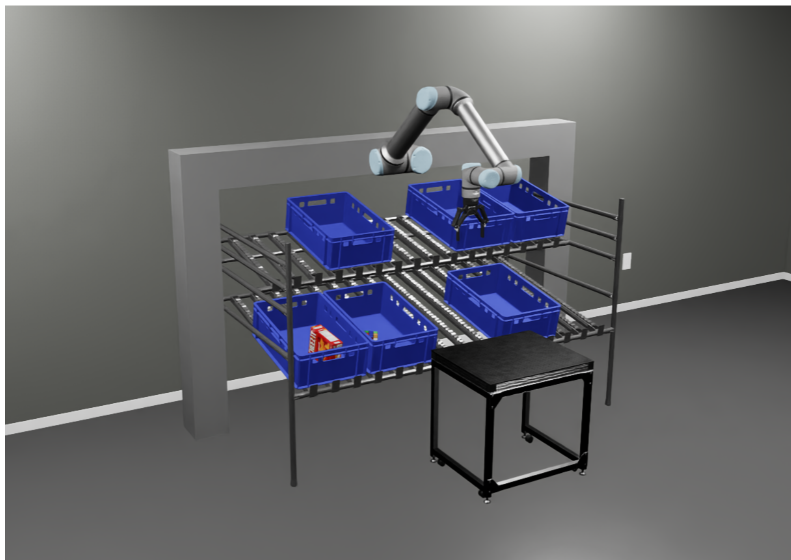


**Figure 3.4:** Final simulation setup showing the modified flow rack, the object placement table used during the kitting task, the overhead support frame with the mounted robot, and the crate arrangement in the simulated environment. Some crates are not shown to improve visualisation.

At a late stage of the project, an additional visual environment representation was also explored. This was based on a Gaussian splatting model of the real research environment inside Volvo CampX, developed by an ongoing bachelor's thesis group within the same research project. Gaussian splatting is a scene representation and rendering method that represents a 3D environment using a set of optimised 3D Gaussian primitives [38]. The splat was provided as a `.ply` Gaussian splatting file and was converted to a USDZ asset using NVIDIA 3DGRUT [39]. The converted

splat asset was then added to the USD scene, alongside the existing robot, crate, and rack assets.

This integration was only performed as a visual test and was not used for the evaluation presented in this thesis. The visualisation can be seen in Figure 3.5. All benchmark results and planner comparisons were instead obtained using the CAD-based simulation environment described above. However, the initial test showed that the Gaussian splatting representation could be used alongside the simulated robot and objects, which points to a possible direction for future work where more realistic environment representations are combined with robot simulation and motion planning.

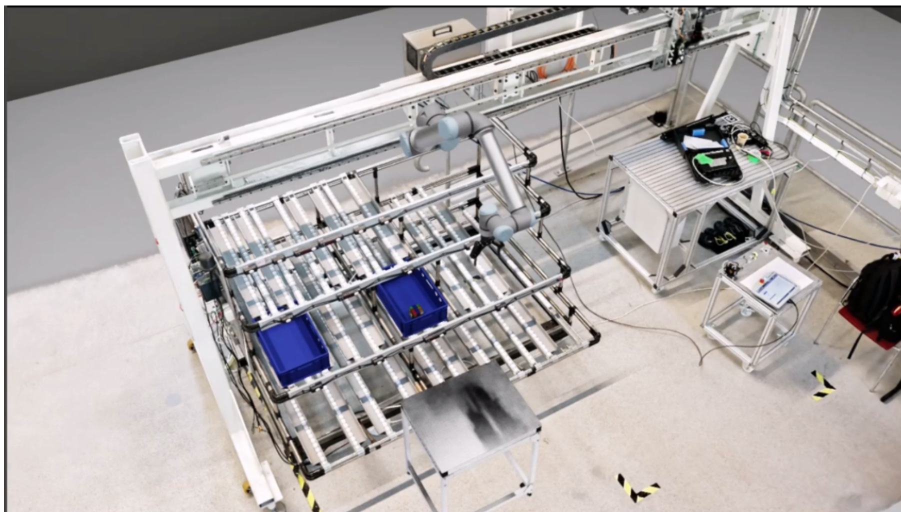


**Figure 3.5:** Visual integration of a Gaussian splatting model of the research environment in Isaac Sim. The model was combined with the simulated robot and generated assets, but was not used in the evaluation.

## 3.4 Path Planning

The path planning was developed in several steps. OMPL was used at first as a conventional MoveIt 2 baseline, since it is a common and easily available approach for robot motion planning in ROS 2. This gave a baseline that used the standard MoveIt 2 planning scene, collision checking, and trajectory execution interfaces.

After this baseline had been set up, GPU-accelerated planners were tested to see whether planning speed and reliability could be improved for the same robot and environment. cuMotion was tested first because it could be added as a MoveIt 2 planning plugin. This meant that the existing MoveIt 2 planning and execution pipeline could still be used. A direct cuRobo interface was then implemented through a separate ROS 2 interface, so that cuRobo could also be evaluated outside the MoveIt 2 planner plugin structure.

Based on the planner comparison, the direct cuRobo approach was chosen as the basis for the later hybrid planner. In this hybrid setup, cuRobo MotionGen was used to generate a complete global trajectory, while cuRobo MPC was used for local reactive control during execution. This made the hybrid planner suitable for the dynamic obstacle experiments, where the aim was to test whether the robot could react to changes in the workspace while moving.

### 3.4.1 Planning Group and Goal Representation

The robot planning group used throughout the project was the seven-degree-of-freedom manipulator chain extending from the gantry base to the tool centre point. This consisted of one prismatic gantry axis together with the six revolute joints of the UR10e robot arm. As described earlier, the gripper was handled as a separate subsystem and was therefore not included in the main arm path-planning group. Planning goals were typically expressed either as end-effector pose targets or as direct joint-space targets, depending on the task.

### 3.4.2 Sampling-Based Planning with OMPL

As a baseline planning approach, MoveIt 2 was configured to use the Open Motion Planning Library (OMPL), which provides a collection of sampling-based motion planning algorithms [40]. OMPL was integrated through MoveIt 2's standard planning pipeline interface and used the same seven-degree-of-freedom planning group as the other planners in the project. This meant that OMPL planned for the gantry joint together with the six revolute joints of the UR10e arm, while the gripper was handled separately.

MoveIt 2 is a framework for robot manipulation and motion planning within the ROS 2 ecosystem [41]. It provides tools for representing the robot, performing kinematic computations, managing collision environments, and generating feasible robot motions while respecting geometric and kinematic constraints [42]. Motion planning in MoveIt 2 is organised around a planning scene, which contains the current robot state together with the relevant environment geometry. The planning scene is used as the main interface for collision checking and constraint checking during planning [43].

In this project, OMPL used the MoveIt 2 planning scene as its collision environment. The planning scene contained the robot model, the current robot state, and the collision geometry that had been added to the planning environment. Motions generated by OMPL were therefore checked against the same MoveIt 2 planning scene representation used by the standard MoveIt 2 planning pipeline. This made OMPL suitable as a baseline planner, since it used the conventional MoveIt 2 collision-checking and planning setup rather than the separate cuRobo collision-world representation used by the GPU-based planners.

Several planner configurations were available in OMPL, including RRT, RRT*, PRM, PRM*, and other variants. In this project, RRT and RRT* were selected as the two OMPL configurations used in the benchmark evaluation. These planners were chosen because they use different sampling-based planning behaviours. RRT is mainly designed to quickly find feasible collision-free paths, while RRT* can continue improving the solution quality as more samples are added, as described in Section 2.3.2. Including both planners, therefore, made it possible to test both a fast conventional sampling-based planner and a sampling-based planner that focuses more on path quality.

RRT was included because of its speed and relatively low computational cost. Since it is designed to stop once a feasible path has been found, it gives a good reference for evaluating how quickly a conventional sampling-based planner can find executable robot motions.

RRT* was included to provide a second CPU-based comparison with a different planning objective. Unlike RRT, RRT* is asymptotically optimal, which means that it can improve the path cost as more samples are added. It was therefore useful for evaluating how a planner that prioritises path improvement compares against the faster feasible-path behaviour of RRT.

For each planning request, MoveIt 2 was configured with a per-attempt time limit of 10 seconds and allowed to perform up to 10 planning attempts from the same start and goal. For RRT, each attempt typically terminated once a valid path had been found. The 10-second limit was therefore mainly used as a safety ceiling for difficult planning problems where a feasible solution was harder to find.

For RRT*, the same 10-second planning time limit was used. However, because RRT* can continue improving a solution after an initial valid path has been found, the planner could use the available planning time either to search for a feasible path or to improve the path cost of an existing solution.

After OMPL generated a geometric path in configuration space, the result was processed by the MoveIt 2 planning pipeline before execution. This included validation with respect to the planning scene, where the trajectory could be rejected if it violated collision constraints, joint limits, or other planning constraints. If the processed plan was valid, MoveIt 2 applied time parameterisation and sent the resulting joint trajectory to the joint trajectory controller, which executed the motion in Isaac Sim through the ROS 2 control interface.

### 3.4.3 GPU-Accelerated Planning with cuMotion

As a second planning approach, NVIDIA cuMotion was added through its MoveIt 2 integration [44]. cuMotion was used as a GPU-accelerated planning backend to evaluate whether faster trajectory generation and improved success rates could be achieved compared with CPU-based sampling planners in MoveIt 2. Planning re-

quests were still sent through the standard MoveIt 2 interfaces. However, instead of using OMPL as the internal planning backend, the MoveIt 2 planning pipeline was configured to use the cuMotion planner plugin. This made it possible to use the same planning group, task-level requests, and execution interface as the OMPL baseline, while replacing the internal planner with a GPU-accelerated trajectory optimisation method.

The cuMotion MoveIt 2 plugin is built on NVIDIA's cuRobo planning framework, which combines parallel seed generation, collision-aware trajectory optimisation, and trajectory refinement on the GPU [4, 45]. In contrast to OMPL, which operates through the standard MoveIt 2 planning scene and collision representation, cuRobo-based planners require a planner-specific robot and world representation [44, 46]. This includes additional information such as collision spheres, acceleration limits, jerk limits, and default joint configurations. The robot geometry was therefore approximated using multiple collision spheres distributed along the arm, gantry, and gripper, thereby reducing collision checking to distance computations that can be evaluated efficiently on the GPU.

To make the MoveIt 2 planning scene compatible with this representation, a dedicated planning-scene processing node was implemented. This node transformed collision objects from the world frame into the robot base frame before planning, since this was the representation required internally by the cuRobo-based planner. In this way, environment information from the MoveIt 2 planning scene could still be used during planning, but it had to be converted into the format expected by the GPU planner.

Planner parameters were selected based on the available GPU memory, allowing the number of seeds and optimisation iterations to scale with the hardware. This made it possible to use a more computationally intensive planning configuration on GPUs with more available memory, while still keeping the planner usable on more constrained systems.

### 3.4.4 GPU-Accelerated Planning with cuRobo

In addition to the MoveIt 2-based cuMotion integration, a separate cuRobo-based planning interface was included through a ROS 2 wrapper referred to as `curobo_ros`. This wrapper was not developed from scratch in this project, but was based on the open-source `curobo_ros` repository from Lab-CORO [47]. The wrapper was adapted and extended to fit the project architecture, robot model, controller setup, and simulation environment. Two main cuRobo modes were used: MotionGen, which generates complete trajectories before execution, and MPC, which computes short-horizon commands repeatedly during execution [46].

The cuRobo integration was implemented around a unified planner node. This node acted as the interface between the ROS 2 system and the cuRobo API. Planning requests were sent to the node through ROS 2 services, while generated trajectories

or streaming control commands were sent onward to the `ros2_control` controller. The unified planner node subscribed to the current joint states of the robot and converted these into cuRobo data structures before each planning or control step. In this way, cuRobo could be used as an alternative planning backend while still remaining part of the same ROS 2-based control architecture described earlier.

As described for cuMotion, cuRobo requires a planner-specific robot representation rather than relying directly on the standard MoveIt 2 robot model [46]. For the GPU-based planners, the URDF/Xacro description alone was not sufficient. URDF defined the robot's kinematic structure and geometry, but cuRobo and cuMotion also required additional planning-specific information. This information was provided through an XRDF file, which stands for Extended Robot Description Format. The XRDF gave the URDF extra information used by the GPU planner, such as the planning joint order, joint limits, acceleration and jerk limits, and null-space weights, which are weights set to influence which joint positions and motions are preferred.

MotionGen was used as the global trajectory generation method in the direct cuRobo integration [46]. The MotionGen solver was constructed from the cuRobo robot configuration, the current collision world, and a set of planning parameters. These parameters controlled the number of trajectory optimisation time steps, the number of optimisation seeds and the number of graph seeds. The MotionGen configuration was adapted to the available GPU memory, allowing more computationally intensive settings to be used on GPUs with more available memory while keeping the planner usable on more constrained hardware. The MotionGen solver was created and warmed up once during node initialisation, which was for initial GPU setup and memory allocation before the first request. After warmup, the same solver was reused for all planning requests until shutdown. This was important because cuRobo relies on GPU memory allocation, and repeated reinitialisation would introduce unnecessary overhead.

The second cuRobo mode used in the project was MPC. Unlike MotionGen, MPC did not generate a complete trajectory before execution. Instead, it repeatedly solved a short-horizon optimisation problem from the current robot state and produced the next joint command to execute. The MPC solver was configured with a control time step of 0.03 s and a prediction horizon of 40 steps. During execution, the planner repeatedly read the current robot state from the ROS 2 joint state topic and converted it into a cuRobo `JointState`. The current goal was stored in a cuRobo goal buffer and passed to the MPC solver. At each control step, the solver computed an action using `mpc.step`. The first valid joint command was extracted from the result, scaled by a command-speed factor, and published to the forward position controller. This process was repeated until the target was reached or the MPC solver failed for too many consecutive iterations.

Because MPC produces commands continuously rather than a complete time-parameterised trajectory, it used a different execution interface from MotionGen. Before MPC ex-

ecution, the active controller was switched from the joint trajectory controller to the forward position controller. The MPC planner then streamed joint position commands directly to the forward position controller. When MPC execution finished, the current joint position was published as a hold command to prevent the robot from drifting. This made the MPC mode suitable for reactive control, where the commanded motion or the collision world could be updated during execution rather than being fixed when the plan was first generated.

The direct cuRobo planners used a separate cuRobo collision-world representation from the MoveIt 2 planning scene [46]. Environment objects from Isaac Sim were converted into cuRobo collision geometry and inserted into the unified planner's world representation at runtime. Before a new world configuration was applied to a solver, the corresponding collision-checking cache was cleared. MotionGen and MPC used the same obstacle source, but maintained separate collision-checking instances because their GPU buffer requirements differed. This allowed both planners to use updated environment information while avoiding conflicts between their internal GPU collision-checking buffers.

This integration provided a GPU-accelerated planning interface alongside the MoveIt 2-based planners. In the planner comparison, it was mainly used through MotionGen, which generated complete trajectories before execution. The same ROS 2-based structure also made it possible to access cuRobo's MPC functionality. Although MPC was not used as a separate standalone planner in the main benchmark comparison, this functionality became important for the later hybrid planner, where it was used for local reactive control during execution.

### 3.4.5 Hybrid Planning with cuRobo MotionGen and MPC

To use trajectory generation and reactive control in the same planning pipeline, a hybrid planning approach was implemented using MoveIt 2's Hybrid Planning framework [48]. The hybrid planner uses a combination of global and local planning. The global planner is responsible for generating a complete trajectory from the start state to the goal, while the local planner is responsible for following and adapting this trajectory during execution based on the current state of the robot and environment. Based on the preceding planner comparison, cuRobo was selected over the MoveIt 2-based planner alternatives because it gave the most suitable combination of planning speed, execution reliability, and trajectory quality for this project. It was therefore used as the basis for both the global and local planning components in the dynamic obstacle experiments.

In this setup, cuRobo MotionGen was used as the global planner to generate an initial reference trajectory, while cuRobo MPC was used as the local planner to track and adapt this trajectory during execution. The purpose of this integration was to combine the global planner's ability to find a complete collision-free path with the local planner's ability to react to changes in the robot's environment.

The hybrid planner was implemented using the standard component structure provided by MoveIt 2 Hybrid Planning. This structure consists of a global planning component, a local planning component, and a hybrid planning manager [48]. The high-level coordination and replanning behaviour were therefore not implemented from scratch. Instead, this project added custom adapters that connected MoveIt 2 Hybrid Planning to the cuRobo-based planning services described in the previous section. This kept the standard MoveIt 2 Hybrid Planning architecture, while using cuRobo MotionGen for global planning and cuRobo MPC for local planning. Figure 3.6 shows the main data flow in the hybrid planning implementation.

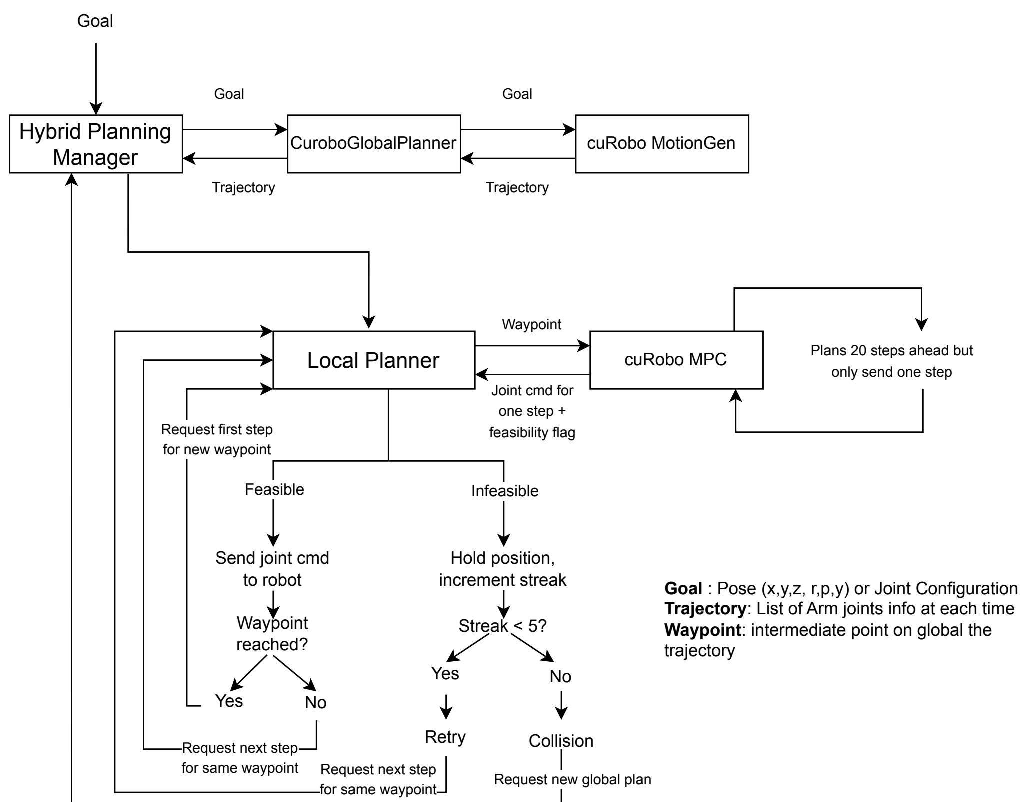



**Figure 3.6:** Overview of the hybrid planning flow. The global planner uses cuRobo MotionGen to generate a reference trajectory, while the local planner samples waypoints from this trajectory and uses cuRobo MPC to produce short-horizon control commands.

A new package, `curobo_hybrid_planning_plugins`, was implemented for this purpose. This package contained three plugins. The first plugin, `CuroboGlobalPlanner`, implements the global planning interface used by MoveIt 2 Hybrid Planning [49]. The second plugin, `LookaheadSampler`, implements the trajectory operator used by the local planner [50]. The third plugin, `CuroboMpcLocalSolver`, implements the local constraint solver interface, which is the component that computes the next local motion command [51]. Together, these plugins made it possible to connect the

MoveIt 2 hybrid planning pipeline to the existing `curobo_ros` unified planner node, without changing the MoveIt 2 Hybrid Planning framework.

The global planner plugin was used for converting MoveIt 2 global planning requests into cuRobo MotionGen requests. When a global planning request was received, `CuroboGlobalPlanner` extracted the current planning group, start state, and goal constraints from the MoveIt 2 request. It then calls the `curobo_ros` services for selecting the appropriate planner and generating a trajectory. The plugin could send either a joint-space target or a Cartesian pose target to cuRobo. If the MoveIt 2 goal contained complete joint constraints, a joint-space MotionGen request was used. For pose goals, the plugin first attempted to convert the pose into a joint-space target using MoveIt 2 inverse kinematics. If this was not possible, the goal was instead sent as a Cartesian pose request to MotionGen. The trajectory returned by cuRobo was then converted back into a MoveIt 2 motion plan response.

One important detail in the global planner implementation was the handling of the start state during replanning. In hybrid planning, replanning may be triggered while the robot is already moving. The start state stored in the original planning request can therefore become outdated. To reduce this problem, the global planner subscribed to the current joint states used by MoveIt 2 and allowed these live values to override old start-state values when a new global plan was generated. This helped make sure that replanning started from the current robot configuration rather than from an earlier state.

The local planner used the global trajectory as a reference rather than executing it directly. The `LookaheadSampler` plugin was responsible for selecting a local target from the global trajectory. During execution, it compared the current robot state with the stored global trajectory, estimated the current progress along the path, and selected a waypoint ahead of the robot as the next local target. This made the local planner track the trajectory step by step, instead of always aiming directly for the final goal. In addition to the local target, the sampler extracted a short window of future waypoints from the global trajectory. This window was used to check whether the upcoming part of the global trajectory was still collision-free.

The `CuroboMpcLocalSolver` plugin connected the MoveIt 2 local planner to the cuRobo MPC solver. At each local planning cycle, the solver reads the current robot state from the planning scene, receives the target selected by the lookahead sampler, and converts this information into a request for the `/unified_planner/mpc_step` service. The request contained the current robot state, the local target, and the future waypoints used for checking the upcoming path. The MPC computation itself was performed inside the `curobo_ros` unified planner node. When a valid MPC response was returned, it was converted into a single-step joint trajectory command for the local planner. The global trajectory, therefore, acted as a guide path, while the actual executed motion was produced step by step by the MPC local solver.

The MPC configuration used in the hybrid planner was slightly different from the

standalone MPC mode described in the previous section. The same control time step of 0.03 s was used, but the prediction horizon was reduced from 40 steps to 20 steps. This was possible because the MPC solver did not have to plan far ahead on its own in the hybrid setup. Instead, it used the global trajectory from MotionGen as a guide and only had to compute short local commands for tracking and adaptation. This reduced the amount of short-horizon optimisation required at each local planning step while still allowing the robot to react to changes during execution.

The hybrid planner used MoveIt 2's replanning manager to handle invalidated trajectories [48]. If the local solver detected that the current trajectory was no longer valid, it returned feedback indicating that there was a collision ahead. This caused the hybrid planning manager, configured with the `ReplanInvalidatedTrajectory` logic, to request a new global trajectory. While waiting for the new global trajectory, the local solver commanded the robot to hold its current joint position instead of continuing to follow the invalidated path, and resumed tracking only once an updated global trajectory had been received.

Tracking and collision checking operated over two distinct horizons. At each local planning cycle (30 Hz), the lookahead sampler first estimated how far the robot had progressed along the global trajectory, then selected a tracking target a fixed number of waypoints ahead of that progress point (10 waypoints in the configuration used). To keep the short-horizon MPC problem well conditioned, this target was pulled back toward the robot whenever its joint-space distance from the current state exceeded 0.45 radians, so the MPC always tracked a reachable nearby pose rather than a distant one. Separately, the sampler attached a forward *sentinel window* of up to 40 upcoming waypoints, bounded to at most 1.2 s of trajectory time ahead and 0.8 radians of joint-space travel. These waypoints were not tracking targets, they were forwarded to the MPC layer purely as collision probes for the path-ahead check.

The invalidation signal could originate from several conditions in the cuRobo MPC layer. The future-waypoint, or sentinel, checks each upcoming global waypoint for collisions against the current cuRobo world model and flags the path as blocked as soon as one waypoint is in collision with an obstacle. The path was also invalidated when the MPC produced a streak of repeated infeasible steps, defined as 10 consecutive solves, or when it failed to make progress toward the goal. This was defined as 25 steps without the goal error improving by more than a small threshold, while not yet near the goal. Any of these conditions, provided the goal had not already been reached, caused the local solver to emit a `COLLISION_AHEAD` feedback signal to trigger global replanning.

The hybrid planner combined the two cuRobo modes into a single planning pipeline. MotionGen provided a global reference trajectory, the lookahead sampler converted this trajectory into local tracking targets and sentinel probes, and MPC generated short-horizon commands, with 0.03 s per step, during execution. The standard MoveIt 2 Hybrid Planning framework provided the structure and replanning coordination, while the project-specific plugins connected this framework to the cuRobo-based planning and control services.

## 3.5 Path Planners Comparison

To compare the planning algorithms, both the quality of the generated motion plans and their usefulness during execution are evaluated. The selected criteria cover speed, reliability, trajectory quality, accuracy, robustness, and computational cost. Together, these metrics give a broad view of planner performance.

To ensure a fair comparison, all planners are tested using the same set of start states, goal states, objects, and environment configurations. Each scenario is repeated multiple times in order to capture average performance and variability across runs.

Table 3.1 presents the evaluation criteria, including what is measured and how it is measured.

**Table 3.1:** Evaluation criteria used to compare the planning algorithms.

| **Criterion** | **What is measured** |
|---|---|
| Planning Time | Time from submission of a planning request until a valid plan is returned or failure is reported |
| Success Rate | Fraction of planning requests that return a valid trajectory |
| Trajectory Duration | Total execution time of the planned motion |
| Path Length in Joint Space | Total accumulated joint motion across all joints |
| Cartesian Path Length of TCP | Distance travelled by the tool centre point along the trajectory |
| GPU Memory (VRAM) Usage | Peak and average GPU memory usage during planning and execution |
| CPU Usage | Processor load during planning and execution |

## 3.6 Test Automation

To evaluate the different planners, the testing was automated so that larger batches of simulation experiments could be run in a consistent way. The purpose was not only to simplify execution, but also to make it possible to test many scenario-planner combinations under repeatable conditions. This was important because the evaluation involved several different planner configurations, multiple task scenarios, and repeated runs of the same setup. Manual execution would therefore have made the testing process slower, less consistent, and more difficult to reproduce.

To enable this, Isaac Sim had to run without manual interaction through the graphi-

cal interface. This was done through Isaac Sim's standalone Python interface, using the `SimulationApp` API in headless mode. In this mode, the simulator can be launched without the graphical interface, while still allowing control of the simulation. The automated tests could therefore start and stop the simulation, reset the scene, and modify objects such as robot or obstacle poses between runs. This made it possible to start each test from a known simulation state and to run experiments without relying on manual setup inside the simulator.

The benchmark automation was organised around a master benchmark script. This script selected which benchmark suite to run, created a timestamped output directory for the session, and wrote a file describing the selected suites, planner configurations, test cases, and number of repetitions. Each benchmark suite then stored its results and logs in a separate subdirectory. This structure made it possible to keep the results from different benchmark sessions separated and to trace each recorded result back to the planner, scenario, and repetition that produced it.

The benchmark suite was launched through a shell interface that allowed either a single suite or all suites to be executed. The full benchmark mode ran three test groups in sequence: a simple motion benchmark, a hybrid planner benchmark, and a pick-and-place benchmark. For each run, the automation reset the simulation and ROS 2 runtime, stopped old launch processes, started the selected planner, played the Isaac Sim simulation, executed the selected scenario, and recorded the result. This made sure that each planner was evaluated from a clean and comparable start state.

The simple motion benchmark was used to test basic planner performance on isolated robot motion goals. The same set of motion cases was run across several planner configurations, including direct cuRobo, cuMotion, OMPL with RRT, OMPL with RRT*, and the hybrid planner. These cases tested motions between different regions of the workspace, such as simply moving above the crates, moving from the home position into a lower crate, moving from inside a lower crate to inside a different lower crate, moving from a lower crate to an upper crate, and moving through narrow gaps between lower crates. These tests were made to evaluate whether each planner could solve harder and harder point-to-point motions in the simulated workspace, and to record timing and failure information for each case. A visual representation of each simple motion case is provided in Appendix A.

A separate benchmark suite was used for the hybrid planner. This suite was designed to test not only whether the planner could generate an initial trajectory, which was tested in the test just described, but also whether it could react when the environment changed during execution. During execution, an obstacle was inserted into the robot's path at different distances from the robot. These ranged from early insertion, where the TCP was around 60 cm from the inserted obstacle, to very late insertion, where the TCP was around 2 cm from the inserted obstacle. The benchmark first moved the robot to the start state, sent the motion goal, and then started a parallel obstacle-spawn process. This process waited for the initial global

trajectory, computed a suitable wall placement relative to the trajectory, monitored the robot's progress, and inserted the obstacle when the selected distance condition was reached.

For the hybrid benchmark, a run was only considered successful if several conditions were satisfied. The robot had to complete the motion, the obstacle had to be spawned, and the current path had to be invalidated at least once. This verified that the hybrid planner was actually using its replanning mechanism rather than simply completing the original path or avoiding the obstacle by chance.

The pick-and-place benchmark was used to test the complete pipeline. This benchmark was run across the main planner configurations, including OMPL, cuRobo, cuMotion, and the hybrid planner. In contrast to the simple motion benchmark, which only tested isolated planning requests. This tested a full task sequence involving object lookup, gripper control, motion planning, collision-object handling, object attachment, transport, release, retreat, and return to the home configuration. The task started by locating the target cube, opening the gripper, moving to a pre-grasp pose, moving down to the grasp pose, suppressing the cube as a world collision object, closing the gripper, and attaching the cube in the simulation and planning representations. The robot then lifted the cube, moved to the drop-off region, released the object, detached it from the robot model, restored it as a world collision object, retreated, and returned home. A visual representation of each state in the pick-and-place sequence is provided in Appendix B.

The testing also included failure handling during repeated runs. If a run exceeded the configured time limit, it was marked as failed and the relevant planner logs were saved for later inspection. The next configured run or scenario was then started instead of leaving the experiment stalled. This made it possible to run larger experiment batches with limited supervision, while still preserving failed attempts as part of the planner evaluation. After the batch was completed, the runtime environment was shut down in a controlled way.

This automation also provided the structure needed for data collection during the experiments. Since each run followed the same general sequence, measurements such as planning time, execution time, joint movement, success or failure, timeout occurrences, replanning events, and failure reasons could be recorded in a consistent way. The automated benchmark structure, therefore, formed the basis for the quantitative comparison of the different planning approaches.

## 3.7 Hardware and Software Setup

All experiments were run on the same computer to make the tests as consistent as possible. The computer had an AMD Ryzen 9 7950X3D 16-core processor, 32 GB of RAM, and an NVIDIA GeForce RTX 4080 GPU with 16 GB of VRAM. The operating system was Ubuntu 24.04.4 LTS with Linux kernel version 6.17.0. The NVIDIA driver version was 580.142

The hardware setup is important because some of the measured results depend on the computer that was used. This includes planning time, RAM usage, GPU memory usage, and CPU usage. Therefore, the reported timing and resource-usage results should be seen as results for this specific test setup, and not as values that are completely independent of the hardware.

## 3.8 Data Collection

Data collection was implemented as part of the automated test framework. The purpose was to store the relevant results from each scenario run in a consistent format so that the different planners could be compared after the experiments had been completed. The framework recorded selected runtime values needed for evaluation and wrote them to CSV files.

For each run, a recorder process was started before the scenario execution began and stopped after the run had completed or timed out. Each run was recorded independently, so that one execution of one scenario with one planner corresponded to one row in the CSV file. The recorder collected information during execution and produced a structured summary after the run. The summary was then converted into a single row in the CSV file by the test script.

The recorder measured planning time and execution time during runtime. When a phase-start event was received, the recorder called a monotonic runtime clock and stored the returned value as the start time of the phase. The recorder then monitored the action status messages from the trajectory execution interface. When the active action goal entered the executing state, the recorder called the same clock again and stored this value as the start of execution. The planning time for the phase was calculated as the difference between the execution-start time and the phase-start time. When the same action goal later reached a terminal state, such as succeeded, cancelled, or aborted, the recorder called the clock once more and calculated the execution time as the difference between this value and the execution-start time.

The TCP movement was estimated using transforms from the ROS 2 TF tree. These transforms were published from the simulated robot state, which was updated from the joint state information during execution. While a phase was active, the recorder repeatedly requested the latest available transform from the world frame to the TCP frame. The translation component of this transform gave the current Cartesian position of the TCP in the world frame. For each new valid position sample, the recorder calculated the Euclidean distance from the previous sample and added this distance to the accumulated TCP movement for the active phase. In this way, the TCP movement was calculated as the sum of the distances between consecutive sampled TCP positions. If a TF transform was temporarily unavailable, that sample was skipped.

Joint movement was recorded for each phase of the automated tests. Each phase was

defined by phase-start and phase-end messages published during the test run. The joint movement recorder used these messages to determine which part of the test was currently being measured. During each active phase, the recorder subscribed to the `/joint_states` topic, where the current joint positions of the robot were published. When a phase started, the latest available joint positions were stored as the first sample. Each following joint-state message was then compared with the previous sample. For each tracked joint, the absolute change in joint position was added to an accumulated movement value. This gave the total movement of each joint during the phase. The six revolute joints of the UR10e arm were reported in degrees, while the prismatic gantry joint was reported in metres.

RAM and GPU memory usage (VRAM) were recorded using a planner resource recorder that ran on the host system during each automated test run. The recorder first identified the running planning container and then used `docker top` to inspect the processes inside it. The relevant planner processes were selected based on the active planner configuration, for example `move_group` for OMPL, `curobo_trajectory_planner` for cuRobo, or the corresponding cuMotion and hybrid planning processes. For each selected process, the physical RAM usage reported by `docker top` was used as the process-level memory measurement. The values from all selected planner processes were summed and converted from KiB to MiB.

GPU memory usage was recorded by querying `nvidia-smi`, which reports the amount of GPU memory used by each compute process. The recorder matched these GPU-memory entries against the planner process IDs found from `docker top`. This meant that only the memory used by the selected planner processes was included, while other GPU users, such as Isaac Sim, were not counted. The recorder sampled these values at 1 Hz during the run. At the end of the run, the minimum, average, and maximum RAM and GPU memory usage were calculated and written to the CSV file.

To get the collision clearance for each motion phase, a separate recorder node monitored the live robot collision spheres together with the currently published obstacle geometry. The obstacle geometry was taken from the MoveIt planning scene and from the collision markers published by the simulated environment. During an active phase, each collision-sphere update provided the position and radius of the robot's collision spheres. The sphere centres were transformed to the world frame using TF, and each valid robot sphere was compared with each currently stored obstacle. For each robot sphere and obstacle pair, the recorder calculated a signed surface distance. For box-shaped obstacles, this was calculated as the distance from the center of the sphere to the surface of the obstacle, minus the radius of the sphere. For spherical obstacles, the distance between the centre of the robot sphere and the centre of the obstacle was calculated, and the robot sphere radius and obstacle radius were subtracted. The smallest sampled signed distance observed during the phase was stored as the minimum collision clearance. A positive value indicated free space, a value close to zero indicated contact, and a negative value indicated overlap.

For trajectory-variation analysis, the TCP pose was collected as discrete samples at 20 Hz and written to a JSON file for each movement phase. At every sampling step, the recorder queried the current TCP pose in the world frame from the ROS 2 transform tree, extracted the translational and rotational components of the transform, and stored them together with the elapsed monotonic time since the start of the phase. Each sample was saved as `x`, `y`, `z`, `qx`, `qy`, `qz`, and `qw`, producing a time-stamped Cartesian pose trace that could later be used to compare repeated runs and analyse variation in the executed TCP path under identical task conditions.

If a run exceeded the configured timeout, it was marked as failed. The available measurements were still written to the CSV file, and the first incomplete or unsuccessful phase was marked as failed. Relevant planner log output from failed runs was saved separately for later inspection.

## 3.9 Dynamic Human Obstacle Integration

To create a more realistic simulation scenario, human movement was introduced into the environment as a dynamic obstacle for the robot arm. This made it possible to include human motion during planning and execution and to study the robot's behaviour in a shared workspace.

### 3.9.1 Dual-Camera Motion Capture

To generate animations of a human operator picking objects from the flow rack, motion data was recorded using a dual-camera setup in Rokoko Vision [52], a markerless tool that reconstructs human motion in 3D from video recordings. Two RGB cameras were positioned at different angles around the recording area so that the subject could be captured from multiple viewpoints throughout the motion. A schematic illustration of the recommended dual-camera arrangement is shown in Figure 3.7. Rokoko recommends placing the cameras with an angular separation between 45 and 90 degrees around the capture area. In the recordings used in this work, the cameras were positioned closer to the lower end of this range. This made it possible for body movements that were less visible from one camera to still be observed by the other.

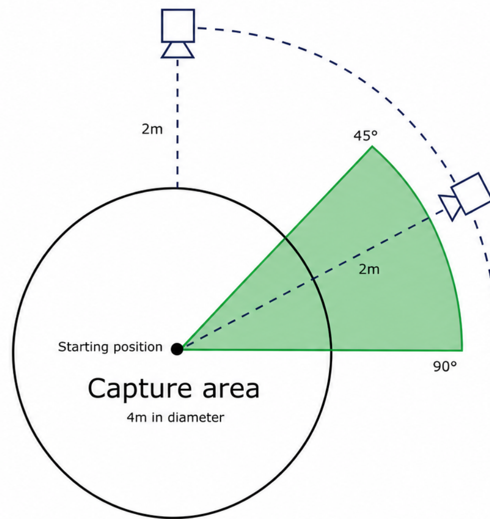



**Figure 3.7:** Schematic illustration of the recommended dual-camera setup for motion capture in Rokoko Vision. The figure shows a capture area with a diameter of approximately 4 m and two RGB cameras placed around the subject at different viewing angles. The illustration represents the general setup principle rather than the exact camera positions used in the recordings.

Before recording, the system was calibrated using Rokoko's checkerboard and floor calibration marker to define the camera setup and establish a shared 3D coordinate frame. Rokoko then estimated 2D body keypoints in each camera view and used the combined observations to reconstruct the motion as a 3D skeletal animation. Figure 3.8 shows an example from the recording workflow, where the two camera views are shown on the right and the reconstructed character is shown on the left. The resulting motion data was then exported as an FBX animation file for further processing in Blender.

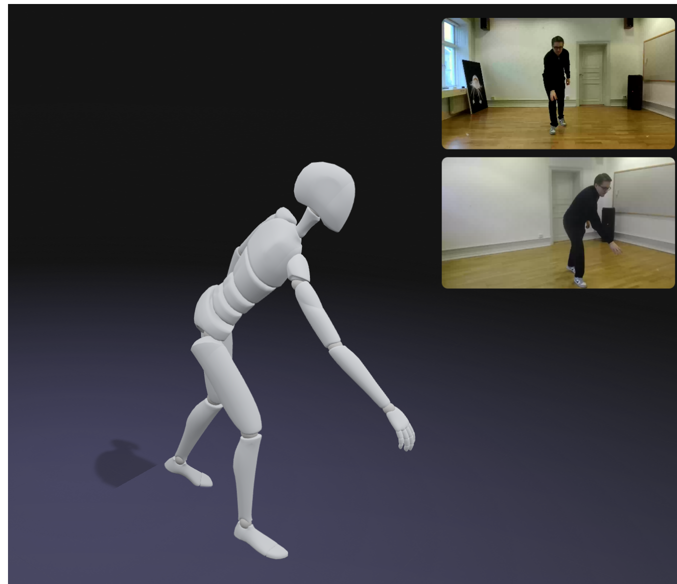


**Figure 3.8:** Example from the dual-camera motion-capture workflow in Rokoko. The two synchronised camera views used for recording are shown on the right, while the reconstructed 3D skeletal character is shown on the left. The recorded motion represents a picking movement intended to imitate a human operator retrieving an object from the flow rack.

### 3.9.2 Character Rigging and Collision Modelling

The recorded human motion from Rokoko Vision was exported as an FBX file and imported into Blender. FBX is a file format used for exchanging 3D content between software tools, including mesh geometry, skeletons with joint placements, and animation data, including joint positions at each timeframe. The FBX file contained the animated Rokoko skeleton character, which was used as the basis for the initial humanoid representation in Isaac Sim. Blender was used to export this animated character as a USD file, allowing the skeleton animation to be imported into the simulation environment.

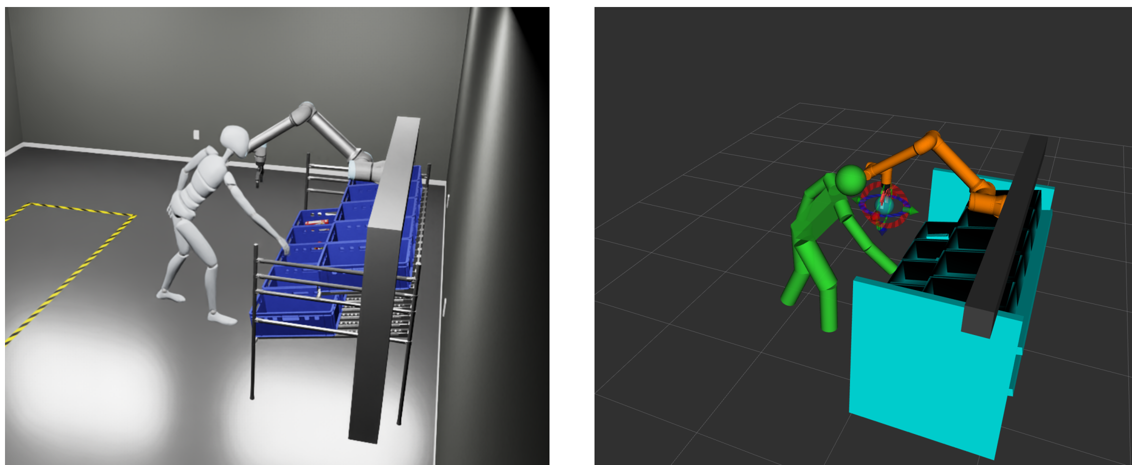
Inside Isaac Sim, the animated character skeleton was connected to a ROS 2 Action Graph to make selected skeleton joints available as TF frames. The Action Graph used a `ROS2 Publish Transform Tree` node, where the relevant skeleton prims were added as target prims. These target prims corresponded to the joints needed for the collision model, including the hips, thighs, shins, feet, shoulders, arms, forearms, hands, neck, and head. During simulation, the node published the transforms of these selected prims to the ROS 2 transform tree. This made the humanoid joint positions available to external ROS 2 nodes and provided the geometric reference for the collision approximation.

The human body was then represented using simplified collision shapes assigned to fixed pairs of skeleton frames. The limbs were modelled as cylinders between the corresponding frames for each body segment, for example between the shoulder and

elbow or between the hip and knee. For each cylinder, the length was determined by the distance between the two frames, and the orientation was aligned with the vector connecting them. The torso was approximated by a box whose centre was placed between the hip and shoulder frames. Its orientation was reconstructed from the current hip-to-shoulder direction and shoulder span, so that the torso collision object rotated with the upper body instead of remaining fixed in the world frame. The head was represented by a sphere placed at the head frame.

These simplified collision shapes were updated continuously at 20 Hz, keeping the collision model synchronised with the animated skeleton in real time. The resulting collision objects were published to the MoveIt planning scene and forwarded to the cuRobo world representation, allowing the planners to perform collision checking against the current human pose during planning and execution.

Figure 3.9 illustrates how the animated humanoid model was converted into the simplified collision representation used by the planners.



(a) Animated humanoid model in the Isaac Sim environment.

(b) Simplified humanoid collision model visualised in RViz.

**Figure 3.9:** Comparison between the animated humanoid model in Isaac Sim and the corresponding simplified collision model used for planning. The Isaac Sim view shows the visual character in the simulation environment, while the RViz view shows the simplified collision geometry generated from the same skeleton pose.

To make the simulated human representation more realistic, a personalised Avaturn character was later used to replace the simple white humanoid model. Avaturn is a web-based 3D avatar creator and avatar system for games, applications, and metaverse environments. It allows users to create realistic avatars that represent themselves and customise features such as body shape, clothes, accessories, hairstyles, shoes, and glasses [53]. In this work, a selfie was used as input to generate a human-like character with realistic skin texture and facial appearance. The generated avatar was imported into Blender, where the recorded Rokoko motion was retargeted to the Avaturn character using the Rokoko Blender add-on. The animated Avaturn

character was then exported as a USD file and imported into Isaac Sim.

Figure 3.10 shows the personalised Avaturn character standing in the final simulated factory environment in Isaac Sim.

**Figure 3.10:** Personalised Avaturn character used as the visual humanoid model in the Isaac Sim factory environment.

# 4

# Results

The results are grouped by benchmark. Each benchmark evaluates a different part of the planning system. The simple motion benchmark evaluates isolated motion-planning tasks, the pick-and-place benchmark evaluates the planners in a complete workflow, and the hybrid obstacle benchmark evaluates the ability of the hybrid planner to react to dynamic changes in the environment.

For each planner and test case, the reported metrics are success rate, planning time, execution time, TCP movement and joint movement. For the pick-and-place benchmark, phase-specific results are also reported to identify where delays or failures occur in the sequence.

Success rates are computed over all attempted runs. Timing and movement statistics are computed only over successful runs, since failed runs may terminate early, time out, or produce incomplete trajectories.

## 4.1 Simple Motion Benchmark

The simple motion benchmark consisted of a set of individual motion-planning problems. Each case defined a start and goal configuration or pose for the manipulator, and each planner was evaluated over repeated runs. The purpose of this benchmark was to compare the planners on isolated planning tasks before evaluating them in a full sequence.

### 4.1.1 Success Rate

Table 4.1 shows the success rate for each planner and simple motion case.

**Table 4.1:** Success rate for the simple motion benchmark.

| Case | cuRobo | cuMotion | RRT* | RRT | Hybrid |
|---|---|---|---|---|---|
| between_lower_to_lower_crate | 100.0% | 98.7% | 0.0% | 0.0% | 100.0% |
| home_to_lower_crate | 98.1% | 100.0% | 55.8% | 57.7% | 100.0% |
| lower_to_lower_crate | 100.0% | 100.0% | 0.0% | 0.0% | 100.0% |
| lower_to_upper_crate | 100.0% | 100.0% | 15.4% | 9.3% | 73.1% |
| move_above_crates | 100.0% | 100.0% | 100.0% | 98.1% | 100.0% |

The GPU-based planners achieved high success rates across most of the simple motion cases. cuRobo and cuMotion solved almost all tested runs, with only small reductions in success rate in one case each. The hybrid planner also performed well in most cases, but had a lower success rate for the lower-to-upper crate motion. The OMPL planners showed more variation between cases. In particular, several cases had a low or zero success rate for the OMPL planners, which shows that these scenarios were more difficult for the sampling-based planners.

### 4.1.2 Planning Time

Table 4.2 shows the planning time for each planner and simple motion case. The values show the time spent generating a valid motion plan before execution.

The hybrid planner is not included in this table because it does not work in the same way as the direct global planners. Instead of only generating one complete trajectory before execution, the hybrid planner combines global planning with local control during execution, and may also replan while the robot is moving.

**Table 4.2:** Planning time for the simple motion benchmark, reported as mean ± standard deviation in seconds over successful runs.

| Case | cuRobo | cuMotion | RRT* | RRT |
|---|---|---|---|---|
| between_lower_to_lower_crate | $0.54 \pm 0.04$ | $4.19 \pm 0.13$ | – | – |
| home_to_lower_crate | $0.52 \pm 0.06$ | $3.75 \pm 0.19$ | $10.03 \pm 0.03$ | $0.07 \pm 0.05$ |
| lower_crate_to_lower_crate | $0.54 \pm 0.05$ | $3.69 \pm 0.16$ | – | – |
| lower_crate_to_upper_crate | $0.49 \pm 0.01$ | $3.79 \pm 0.16$ | $5.31 \pm 4.07$ | $0.13 \pm 0.02$ |
| move_above_crates | $0.36 \pm 0.02$ | $3.11 \pm 0.12$ | $10.02 \pm 0.03$ | $0.05 \pm 0.03$ |

cuRobo produced the shortest planning times among the planners that consistently solved the simple motion cases. cuMotion also achieved high success rates, but required longer planning times than cuRobo. The OMPL timing results varied strongly between cases and should be viewed together with the success rates in Table 4.1. For cases with low success rates, the reported planning time is based on fewer successful runs and is therefore less representative of the planner's overall behaviour in that case. Overall, the results show that cuRobo provided the fastest trajectory generation among the planners evaluated in this table.

### 4.1.3 Execution Time

Table 4.3 shows the execution time for the simple motion benchmark. This is the time taken to execute the generated trajectory in the simulation. For the hybrid planner, execution time also includes the behaviour of the local control process during motion.

**Table 4.3:** Execution time for the simple motion benchmark, reported as mean ± standard deviation in seconds over successful runs.

| Case | cuRobo | cuMotion | RRT* | RRT | Hybrid |
|---|---|---|---|---|---|
| between_lower_to_lower_crate | $5.04 \pm 0.47$ | $3.95 \pm 0.05$ | – | – | $14.11 \pm 3.25$ |
| home_to_lower_crate | $4.60 \pm 0.26$ | $5.95 \pm 0.17$ | $3.01 \pm 0.05$ | $3.06 \pm 0.06$ | $18.80 \pm 2.04$ |
| lower_to_lower_crate | $3.42 \pm 0.39$ | $3.46 \pm 0.03$ | – | – | $8.22 \pm 1.57$ |
| lower_to_upper_crate | $4.68 \pm 0.49$ | $4.31 \pm 0.13$ | $6.03 \pm 2.56$ | $7.31 \pm 2.45$ | $21.44 \pm 1.85$ |
| move_above_crates | $4.09 \pm 0.22$ | $3.24 \pm 0.03$ | $5.55 \pm 0.07$ | $5.52 \pm 0.06$ | $9.46 \pm 0.58$ |

The execution times show that cuRobo and cuMotion both produced trajectories that could be executed quickly in the simple motion cases. cuMotion had shorter execution times in some cases, while cuRobo was faster in others. The hybrid planner had longer execution times overall. This is expected, since the hybrid planner executes through a local control process rather than only following a precomputed global trajectory. For OMPL cases with low success rates, the reported execution time is based on fewer successful runs and is therefore less representative of the planner's overall behaviour in those cases.

### 4.1.4 TCP Movement

Table 4.4 shows the total TCP movement for each planner and case. This metric shows the Cartesian path length followed by the end effector during execution.

**Table 4.4:** TCP movement in meters for the simple motion benchmark, reported as mean ± standard deviation in metres over successful runs.

| Case | cuRobo | cuMotion | RRT* | RRT | Hybrid |
|---|---|---|---|---|---|
| between_lower_to_lower_crate | $1.3 \pm 0.4$ | $1.5 \pm 0.0$ | – | – | $1.3 \pm 0.2$ |
| home_to_lower_crate | $0.9 \pm 0.0$ | $3.9 \pm 0.2$ | $0.9 \pm 0.0$ | $0.9 \pm 0.0$ | $1.3 \pm 0.2$ |
| lower_to_lower_crate | $1.1 \pm 0.2$ | $1.7 \pm 0.0$ | – | – | $1.0 \pm 0.1$ |
| lower_to_upper_crate | $2.0 \pm 0.7$ | $1.2 \pm 0.0$ | $3.8 \pm 0.8$ | $4.7 \pm 0.8$ | $1.8 \pm 0.1$ |
| move_above_crates | $1.2 \pm 0.0$ | $1.3 \pm 0.0$ | $1.2 \pm 0.0$ | $1.2 \pm 0.0$ | $1.2 \pm 0.0$ |

The TCP movement results give information about the path quality and effectiveness of the generated motions. Overall, cuRobo and the hybrid planner produced TCP movements of similar magnitude in several cases, while cuMotion sometimes produced longer Cartesian motion.

### 4.1.5 Joint Movements

Table 4.5 shows the total arm joint movement for each planner and simple motion case. This metric is the summed angular motion across the arm joints during execution.

**Table 4.5:** Arm joint movement for the simple motion benchmark, reported as mean ± standard deviation in degrees over successful runs.

| Case | cuRobo | cuMotion | RRT* | RRT | Hybrid |
|---|---|---|---|---|---|
| between_lower_to_lower_crate | $118.90 \pm 76.04$ | $122.94 \pm 5.67$ | – | – | $126.13 \pm 44.83$ |
| home_to_lower_crate | $313.29 \pm 7.38$ | $680.71 \pm 19.03$ | $294.60 \pm 1.76$ | $284.85 \pm 55.85$ | $369.52 \pm 28.59$ |
| lower_to_lower_crate | $85.36 \pm 63.93$ | $120.45 \pm 0.82$ | – | – | $71.31 \pm 25.04$ |
| lower_to_upper_crate | $419.29 \pm 77.05$ | $640.54 \pm 2.18$ | $893.23 \pm 306.37$ | $1162.69 \pm 209.37$ | $393.42 \pm 7.18$ |
| move_above_crates | $77.71 \pm 0.08$ | $507.88 \pm 1.54$ | $75.58 \pm 1.52$ | $77.65 \pm 2.12$ | $89.83 \pm 8.51$ |

The joint-movement results show clear differences in motion efficiency between planners. cuRobo and Hybrid generally required less arm-joint motion than cuMotion, while OMPL planners had larger joint motion in the more difficult lower-to-upper crate case. In particular, the OMPL planners produced substantially higher joint movement for the lower-to-upper crate. For move-above-crates, cuRobo and OMPL had similarly low joint motion, whereas cuMotion used much larger total joint movements.

**Table 4.6:** Gantry movement for the simple motion benchmark, reported as mean ± standard deviation in metres over successful runs.

| Case | cuRobo | cuMotion | RRT* | RRT | Hybrid |
|---|---|---|---|---|---|
| between_lower_to_lower_crate | $0.74 \pm 0.01$ | $0.67 \pm 0.00$ | – | – | $0.78 \pm 0.06$ |
| home_to_lower_crate | $0.57 \pm 0.01$ | $0.74 \pm 0.02$ | $0.54 \pm 0.01$ | $0.53 \pm 0.10$ | $0.61 \pm 0.09$ |
| lower_to_lower_crate | $0.75 \pm 0.03$ | $0.68 \pm 0.00$ | – | – | $0.75 \pm 0.02$ |
| lower_to_upper_crate | $0.45 \pm 0.11$ | $0.35 \pm 0.02$ | $0.84 \pm 0.46$ | $0.74 \pm 0.70$ | $0.42 \pm 0.03$ |
| move_above_crates | $1.21 \pm 0.00$ | $0.62 \pm 0.00$ | $1.20 \pm 0.02$ | $1.19 \pm 0.01$ | $1.20 \pm 0.01$ |

For move_above_crates, cuMotion had clearly lower gantry movement than the other planners, but it also had much higher total arm joint movement. This shows that lower movement in one part of the robot does not necessarily mean lower movement overall, since the motion can be distributed differently between the gantry and the arm joints.

## 4.2 Pick-and-Place Benchmark

The pick-and-place benchmark evaluated the planners in a complete manipulation workflow. The workflow consisted of several phases: moving to a pre-grasp pose, grasping the object, lifting the object, moving to the drop pose, releasing the object, retreating from the drop pose, and returning home. This benchmark is more representative of the full system behaviour than the simple motion benchmark, since a failure in any phase can cause the full task to fail.

### 4.2.1 Workflow Success Rate

Table 4.7 shows the overall success rate for the pick-and-place benchmark together with the most common failed phase for each planner.

**Table 4.7:** Overall success rate and most common failed phase for the pick-and-place benchmark.

| Planner | Success rate | Most common failed phase |
|---|---|---|
| cuMotion | 75.6% | grasp |
| cuRobo | 96.2% | post_grasp_lift |
| Hybrid | 80.9% | post_grasp_lift |
| RRT | 78.3% | return_home |
| RRT* | 62.0% | pre_drop |

cuRobo achieved the highest overall success rate in the pick-and-place benchmark, but did not complete all runs successfully. The most common failed phase for cuRobo was the post-grasp lift. The hybrid planner also had most failures during post-grasp lift, while cuMotion failed most often during the grasp phase. The OMPL planners had lower overall success rates in this benchmark, with RRT failing most often during return home and RRT* failing most often during pre-drop.

### 4.2.2 Workflow Time

Figure 4.1 shows the total workflow time for the pick-and-place benchmark, divided into global planning time, execution/control time, and remaining overhead. The whiskers show one standard deviation of the total workflow time over successful runs. The corresponding numerical values are provided in Appendix C, Table C.1.

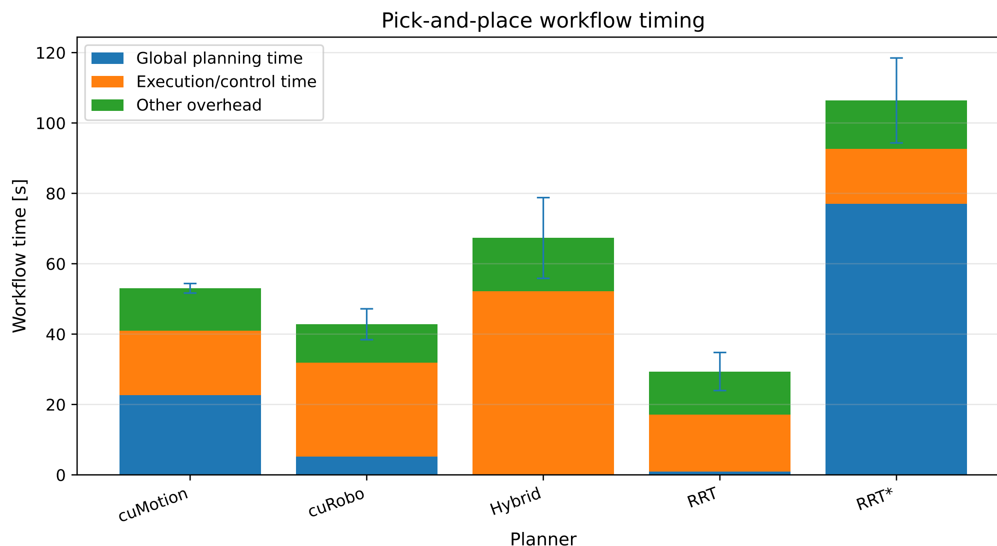



**Figure 4.1:** Pick-and-place workflow timing. The stacked bars show the mean global planning time, execution/control time, and remaining overhead. The whiskers show one standard deviation of the total workflow time over successful runs. For the hybrid planner, global planning time is not shown because planning and control are interleaved during execution.

RRT achieved the lowest total workflow time among the successful pick-and-place

runs, mainly due to its very short planning time. However, this should be interpreted together with its lower overall success rate shown in Table 4.7. cuRobo had a higher total time than RRT, but achieved the highest overall success rate and had substantially shorter global planning time than cuMotion and RRT*. RRT* had the longest total workflow time, primarily because of its much longer planning time. The hybrid planner had the longest execution/control time, which is expected since it relies on local control during execution instead of only following a precomputed trajectory.

### 4.2.3 Per-Phase Planning Time

Figure 4.2 shows the planning time for each phase of the pick-and-place workflow. The hybrid planner is not included because its planning process is not directly comparable to the other planners. For cuRobo, cuMotion, RRT and RRT*, planning time refers to the time required to generate a trajectory for a specific phase before that phase is executed. The corresponding numerical values are provided in Appendix C, Table C.2.

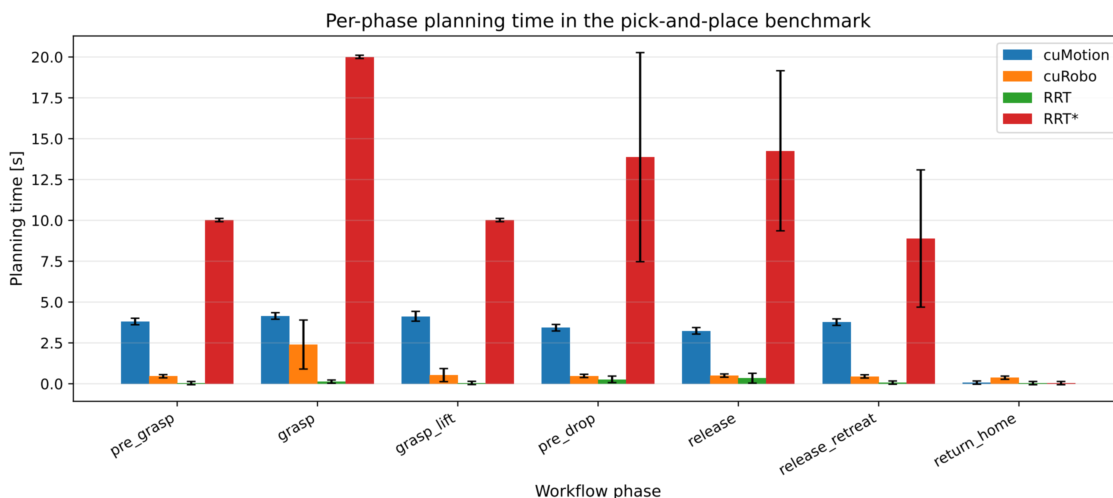



**Figure 4.2:** Per-phase planning time in the pick-and-place benchmark. Bars show the mean planning time over successful phase executions, and whiskers show one standard deviation.

The per-phase planning times show clear differences between the planners. RRT had the shortest planning times for most phases, but this should be interpreted together with its lower workflow success rate. cuRobo planned most phases in less than one second, with the grasp phase being the main exception. cuMotion required several seconds for most phases and showed relatively consistent planning times across the workflow. RRT* had the longest planning times overall, especially during the grasp phase.

These results show that cuRobo provided fast and reliable phase-level trajectory generation, while RRT was faster in successful attempts but less robust over the full workflow.

### 4.2.4 Per-Phase Success Rate

Table 4.8 shows the success rate for each phase of the pick-and-place workflow.

**Table 4.8:** Per-phase success rate for the pick-and-place benchmark.

| Planner | pre_grasp | grasp | grasp_lift | pre_drop | release | release_retreat | return_home |
|---|---|---|---|---|---|---|---|
| cuMotion | 99.5% | 88.0% | 100.0% | 100.0% | 100.0% | 100.0% | 86.4% |
| cuRobo | 100.0% | 100.0% | 96.2% | 100.0% | 100.0% | 100.0% | 100.0% |
| Hybrid | 100.0% | 96.8% | 83.5% | 100.0% | 100.0% | 100.0% | 100.0% |
| RRT | 100.0% | 100.0% | 100.0% | 93.3% | 94.6% | 98.1% | 90.4% |
| RRT* | 100.0% | 100.0% | 100.0% | 84.0% | 83.3% | 94.3% | 93.9% |

The per-phase success rates show where the failures occurred in the workflow. cuRobo completed most phases successfully, with failures only appearing in the grasp-lift phase. cuMotion had reduced success during grasp and return home, which explains its lower overall workflow success rate. The hybrid planner had its lowest phase success during grasp lift. RRT and RRT* both completed the early phases reliably, but their success rates decreased in later phases such as pre-drop, release, retreat, and return home.

### 4.2.5 Joint and Gantry Motion

The total arm joint motion and gantry motion were measured for the pick-and-place benchmark. These metrics give information about how much the robot configuration changed during the full workflow. The arm joint motion describes the accumulated motion of the six revolute arm joints, while the gantry motion describes the accumulated movement of the linear gantry axis.

Figure 4.3 shows the total arm joint motion for each planner. The corresponding numerical values are provided in Appendix C, Table C.3.

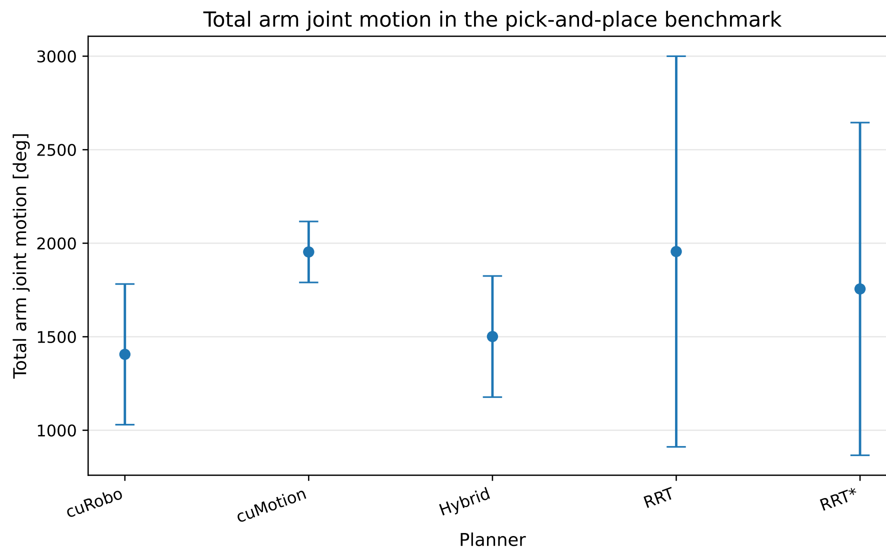



**Figure 4.3:** Total arm joint motion in the pick-and-place benchmark. The markers show the mean over successful runs, and the whiskers show one standard deviation.

The total arm joint motion shows that cuRobo produced the lowest accumulated arm motion over the full pick-and-place workflow. The hybrid planner also produced relatively low arm joint motion, while cuMotion and RRT required more arm movement. RRT and RRT* also had large standard deviations, which indicates greater variation between successful runs. Figure 4.4 shows the total gantry motion for each planner. The corresponding numerical values are provided in Appendix C, Table C.4.

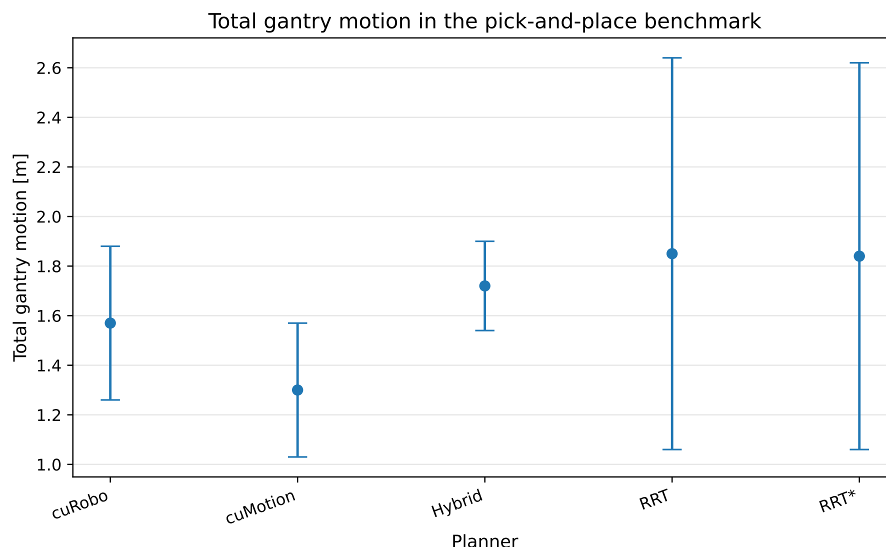



**Figure 4.4:** Total gantry motion in the pick-and-place benchmark. The markers show the mean over successful runs, and the whiskers show one standard deviation.

The total gantry motion shows a different pattern from the arm joint motion. cuMotion used the least gantry motion, while RRT and RRT* used the most. cuRobo

had the lowest total arm joint motion, but did not have the lowest gantry motion. This shows that the planners distributed motion differently between the arm joints and the gantry axis.

## 4.3 Hybrid Dynamic Obstacle Benchmark

The hybrid dynamic obstacle benchmark evaluated whether the hybrid planner could react to a new obstacle appearing during execution, and how close this obstacle could be spawned in front of the robot while the planner still managed to recover. The benchmark measured whether the robot reached the goal after the obstacle was inserted. All results in Figure 4.5 are based on the benchmark case move_above_crates. The number above each bar shows the number of runs in that clearance range, and the corresponding numerical values are provided in Appendix C, Table C.5.

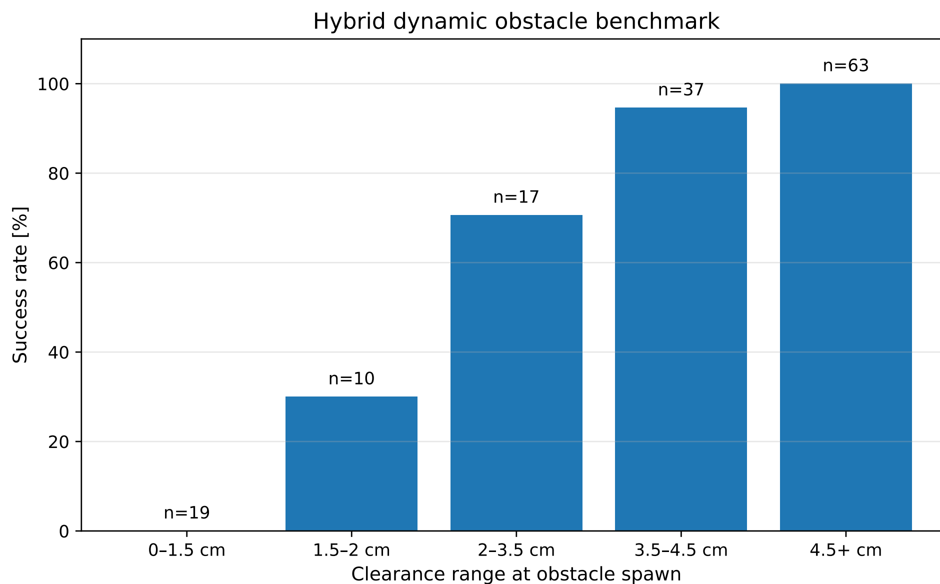



**Figure 4.5:** Success rate for the hybrid dynamic obstacle benchmark grouped by robot clearance at obstacle spawn. The number above each bar shows the number of runs in that clearance range.

The results show that the hybrid planner was able to handle the dynamic obstacle reliably when the obstacle was inserted with sufficient clearance from the robot. For clearance ranges of 4.5 cm and above, the planner reached the goal in all tested runs. The success rate decreased as the obstacle was spawned closer to the robot. In the 3.5–4.5 cm range, the planner still reached a high success rate, while the success rate dropped clearly for the 2–3.5 cm and 1.5–2 cm ranges. When the obstacle was inserted within 0–1.5 cm of the robot, no runs succeeded.

These results show that the hybrid planner can react to dynamic changes in the environment, but that there is a limit to how close an obstacle can appear before successful recovery becomes unlikely.

## 4.4 Humanoid Obstacle Demonstration

In addition to the repeatable dynamic obstacle benchmark, a separate demonstrative test was performed with the animated humanoid model described in Section 3.9. The purpose was to verify that the hybrid planner could also react to a more complex moving obstacle, rather than only to the simplified spawned obstacle used in the benchmark.

The demonstration used the home_to_lower_crate simple motion. During execution, the pre-recorded humanoid animation was started so that the humanoid's hand moved into the lower crate area and obstructed the robot arm's planned motion. When the humanoid occupied the planned path, the robot first attempted to adapt its motion around the obstacle. Since no feasible local motion around the humanoid was found, the robot stopped and waited. Once the humanoid moved away and the path became clear again, the robot continued toward the goal.

A video of the demonstration is available at `https://doi.org/10.5281/zenodo.20663252`. Figure 4.6 shows a frame from this video at the moment when the humanoid reaches into the lower crate and the robot arm has stopped.

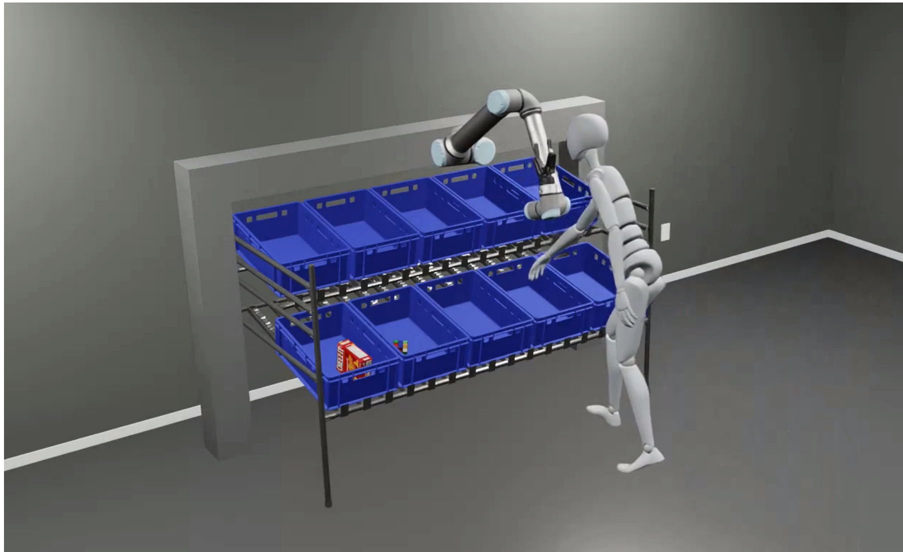


**Figure 4.6:** Frame from the humanoid obstacle demonstration video. The humanoid reaches into the lower crate during the home_to_lower_crate motion, causing the hybrid planner to stop the robot arm until the path becomes clear again.

This test was not included in the quantitative benchmark results as the humanoid animation could not be introduced with the same repeatability as the simplified dynamic obstacle. Small variations in planning time and execution behaviour could change the relative timing between the robot motion and the humanoid animation, making repeated runs difficult to compare reliably. The test should therefore be interpreted as a one-off demonstration based on observed behaviour, showing that the hybrid planner can react to a more realistic moving human obstacle.

## 4.5 Resource Usage

Resource usage was measured to compare the computational cost of the different planners. Table 4.9 shows average and maximum RAM usage, together with the average and maximum GPU memory usage.

**Table 4.9:** Resource usage for the evaluated planners.

| Planner | RAM avg [MiB] | RAM max [MiB] | GPU avg [MiB] | GPU max [MiB] |
|---|---|---|---|---|
| cuMotion | 2561 | 2734 | 5621 | 5972 |
| cuRobo | 2319 | 2351 | 5318 | 5345 |
| Hybrid | 2742 | 2811 | 5498 | 5876 |
| RRT | 212 | 368 | 0 | 0 |
| RRT* | 252 | 380 | 0 | 0 |

The resource usage results show a clear difference between the OMPL planners and the GPU-based planners. RRT and RRT* required only a few hundred MiB of RAM and did not use any GPU memory. This makes the OMPL planners computationally cheap and possible to run on much less powerful hardware.

In contrast, cuRobo, cuMotion, and the hybrid planner all required several gigabytes of GPU memory and more than 2 GiB of RAM on average. The differences between the GPU-based planners were relatively small compared with the difference between the GPU-based planners and OMPL.

# 5
# Discussion

The planner comparison gives insight into how the different planning approaches behaved in the simulated kitting setup. OMPL, cuMotion, and direct cuRobo were first evaluated in static motion-planning and pick-and-place tasks, where the results showed clear differences in planning speed, robustness, and motion efficiency. These differences laid the ground for the use of cuRobo as the basis for the hybrid planner.

The hybrid planner was then tested on the same static motion-planning and pick-and-place tasks, and then it was also tested on reactive behaviour during motion. This is relevant in a shared workspace, where the robot cannot assume that the environment will remain unchanged after a trajectory has been planned. The discussion, therefore, focuses both on the general trade-offs between the planners and on what the hybrid results show about reacting to dynamic changes in the workspace.

## 5.1 Overall Planner Performance

The comparison between the planners shows that the most important difference was not whether a planner achieved the best value in a single metric, but how well it balanced the different requirements of the task. For the simulated kitting environment, the planner needed to generate motions quickly, complete tasks reliably, and avoid unnecessarily inefficient movements. This means that the results cannot be interpreted by looking at planning time, execution time, or movement metrics separately. They must be looked at together with the success rate, since a fast or efficient result is less meaningful if it only represents the subset of runs where the planner managed to find a valid solution.

This is especially important when comparing planners with different strengths. A planner may appear favourable in one metric, for example, by producing a short planning time, but still be less useful for the system if it fails more often. In the same way, a planner with a higher planning time may still be preferable if it provides more consistent task completion. The evaluation, therefore, has to be seen as a trade-off between responsiveness, reliability, and the quality of the generated motion, rather than as a ranking based on a single result column.

When the benchmark results are viewed together, cuRobo provided the strongest overall balance. It combined high success rates with short planning times and generally efficient motion, without showing the same clear weakness as the other alter-

natives. cuMotion was often successful but slower, while the OMPL planners could be fast in successful cases but had lower success rates overall. This made cuRobo the most suitable planner for being used in the hybrid planner.

## 5.2 Planning Speed Versus Robustness

The results show that planning speed alone is not enough to determine which planner is most suitable for the full workflow. RRT is the clearest example of this. In the successful pick-and-place runs, RRT had the lowest total workflow time and a very short global planning time. However, its overall workflow success rate was lower than cuRobo. This means that RRT was very fast when it found a valid solution, but it was less dependable across repeated runs of the complete task. In a practical robotic system, this difference is important, since a failed plan can stop the workflow or require recovery behaviour that is not visible when only successful runs are compared.

cuRobo showed a better balance between speed and robustness. Its successful pick-and-place runs were slower than RRT, but it achieved the highest overall success rate while still keeping the global planning time low compared with cuMotion and RRT*. This suggests that a planner with slightly longer planning or execution time can still be preferable if it avoids failures more consistently. For a full manipulation task, reliability across all phases is often more important than achieving the shortest planning time in the subset of runs that succeed.

The same trade-off can also be seen in the simple motion benchmark. RRT had very short planning times in the cases where it succeeded, but several simple motion cases had low or zero success rates for the OMPL planners. This makes the timing values harder to interpret in isolation, since they only describe successful runs. A low planning time is therefore not necessarily a sign of good overall performance if the planner fails often in the same scenario.

RRT* showed a different limitation. In contrast to RRT, it did not provide very short planning times, since it was configured to spend more time searching for or improving a solution. However, this additional planning time did not lead to higher overall workflow success in this benchmark. This made RRT* the least favourable trade-off among the tested planners, since it combined long planning times with lower workflow robustness.

The hybrid planner is more difficult to compare directly against the global planners in terms of planning speed. Unlike cuRobo, cuMotion, RRT, and RRT*, the hybrid planner does not only generate one complete trajectory before execution. Instead, it combines a global trajectory with local control during execution and can also trigger replanning if the current path becomes invalid. Its longer execution and control time should therefore be interpreted as the cost of adding reactive behaviour, rather than only as slower trajectory execution. This makes the hybrid planner less attractive for static tasks where a single global plan is sufficient, but more relevant for dynamic

environments where the robot may need to react while moving.

The difference between cuRobo and cuMotion shows that planning time depends not only on the underlying planning algorithm, but also on how the planner is integrated into the system. Although both planners are based on the cuRobo motion generation stack, the direct cuRobo implementation used a more direct ROS 2 interface, where the CUDA-based solver setup could be initialised once and reused across later planning requests, reducing repeated setup overhead. In contrast, this behaviour was not exposed in the same way through the cuMotion MoveIt 2 plugin interface. The cuMotion integration included additional MoveIt 2 request handling, planning-scene processing, validation, and trajectory post-processing, while also providing less direct access to low-level cuRobo optimisation settings. Although this increased planning latency, it also provided the benefits of a standardised MoveIt 2 integration, including compatibility with established planning-scene management, action-based execution, and existing workflow components. This can be beneficial for maintainability and compatibility within the MoveIt 2 ecosystem, but it also reduces low-level configurability. The longer planning times observed for cuMotion should therefore be interpreted as a consequence of MoveIt 2 integration overhead and reduced low-level configurability, rather than as a direct limitation of the underlying GPU-based planner itself. For latency-critical applications, the results suggest that a more direct cuRobo interface may be preferable because it avoids part of this middleware overhead, provides greater control over planner configuration, and reduces planning latency.

Overall, the results show that planner performance must be viewed as a combination of speed and robustness. RRT was the fastest planner in successful attempts, but its lower success rate made it less reliable for the complete workflow. RRT* had long planning times without providing enough robustness to justify this cost. cuMotion was robust in many simple motion cases, but its MoveIt 2 integration introduced higher planning latency. Among the non-reactive planner configurations, cuRobo gave the best balance in this project, since it combined a high success rate with low planning time. This made it a suitable basis for the hybrid planner, where reactive capability was added at the cost of longer execution and control time.

## 5.3 Motion Efficiency and Motion Distribution

Motion efficiency is more complex in this system than simply measuring how far the TCP moved. Since the robot was mounted on a linear gantry, the planners could solve the same task by distributing motion differently between the gantry and the arm joints. This means that two trajectories with similar TCP movement could still represent different types of robot behaviour. One planner might move the gantry more and keep the arm posture relatively stable, while another might keep the gantry almost fixed and instead use larger arm-joint motions. Therefore, low TCP movement or low gantry movement does not necessarily mean that the full robot motion is efficient. A short TCP path can still require large changes in the arm joints, and low gantry movement can simply mean that the arm has to do

more of the work. The TCP movement, gantry movement, and arm-joint movement, therefore, need to be considered together.

The movement distribution is also relevant because the robot is intended to work in a shared environment. Motions that use large arm-joint changes while the gantry stays almost fixed can be harder for a nearby human to understand. Even if the TCP path is short, the robot may appear to change posture in a less expected way. A motion where the gantry and arm are used together can be easier to follow, since the whole robot moves more clearly toward the target. This is important for a collaborative robot, where predictable motion is part of making the system easier to work around.

The results show that the planners used the gantry and arm differently. cuMotion often reduced the gantry movement, but this was usually accompanied by higher arm-joint movement. This difference is likely caused by how cuRobo and cuMotion were configured in this project, rather than by a fundamentally different motion-generation method. In the direct cuRobo setup, the configuration allowed different weights to be assigned to individual joints in the trajectory optimisation objective. The gantry joint was given a lower movement cost than the main arm joints, especially the shoulder joint, which encouraged the planner to use the gantry for positioning and avoid large arm rotations. cuMotion was used through the MoveIt integration, where the same detailed per-joint weighting was not exposed in the same way. With more uniform joint costs, cuMotion had less optimisation pressure to prefer gantry motion over arm motion, so it could solve some motions by using more arm-joint movement while keeping the gantry displacement lower. This also helps explain why cuMotion sometimes had shorter execution times. The arm joints have higher velocity limits than the gantry, so a trajectory with more arm motion and less gantry motion can execute faster, even if it is not more efficient in terms of total joint movement. The hybrid planner followed a movement pattern close to cuRobo in several cases, which is expected because its global motion was generated with cuRobo before being adjusted during execution.

The OMPL planners had higher variation in their movement results than the other planners. This is likely because they are sampling-based planners, where the final path depends on which configurations are sampled and connected during each run. Therefore, the same start and goal can lead to different successful paths with different amounts of gantry and arm movement. However, the two OMPL planners did not behave identically. RRT* is also sampling-based, but it continues to improve the path while it searches by replacing parts of the path with lower-cost connections when possible. This can explain why RRT*, in some cases, had lower variation and lower joint movement than RRT. The effect was still smaller for cuRobo and cuMotion, because their motions are shaped more directly by trajectory optimisation. They do not only search for any valid path, but also optimise candidate motions according to similar constraints and cost terms each time. This makes the resulting motions more consistent when the start, goal, and environment are the same.

This shows that the planner choice affects not only whether a path is found, but

also how the robot reaches the target. For a gantry-mounted collaborative robot, this is important because the motion should be collision-free, consistent, and predictable for people working nearby. A planner that reduces one movement value by increasing another can give a misleading impression of efficiency. This is why the movement results support the same conclusion as the timing and success-rate results. cuRobo gave the most suitable overall behaviour because it balanced these trade-offs better than cuMotion and the OMPL planners.

## 5.4 Full-Workflow Evaluation and Failure Modes

The pick-and-place benchmark showed why it was important to evaluate the planners in a complete workflow and not only on isolated motion-planning problems. The simple motion benchmark tested whether a planner could solve individual point-to-point motions, which was useful for comparing basic planning capability. However, a full pick-and-place task depends on more than a single valid trajectory. The robot must move to the object, perform the grasp, update the planning scene, attach the object to the robot model, lift the object, move to the drop location, release the object, update the collision representation again, retreat, and return home. A failure in any of these phases causes the full task to fail, even if the planner is able to solve many of the individual motions.

This means that the pick-and-place benchmark evaluated the complete planning and execution pipeline rather than only the path planner itself. Some failures may therefore have been caused by the planner being unable to find a valid trajectory, while others may have been related to the surrounding task logic, such as grasp execution, object attachment, or adding and removing objects in the planning scene. This makes the benchmark more representative of a practical robotic system, where the planner has to work together with gripper control, collision-object handling, and execution logic.

An interesting result was that the OMPL planners performed better in the pick-and-place workflow than in several of the simple motion benchmark cases. In the simple benchmark, the motions required the planner to solve one larger point-to-point problem, for example, moving directly from inside one crate to inside another crate. In the pick-and-place workflow, the task was instead divided into several shorter phases with intermediate waypoints. This likely made the problem easier for the sampling-based planners, since each planning request covered a smaller part of the total motion. The results show that sampling-based planners can still be useful when the task is made into shorter and more guided segments, even if they struggle with some longer or more constrained point-to-point plans.

The phase-specific results also showed that the overall workflow success rate alone is not enough to understand the behaviour of each planner. Different planners failed in different parts of the task. cuRobo achieved the highest overall success rate, but its remaining failures occurred during the grasp-lift phase. The hybrid planner also had its lowest success rate during grasp lift. cuMotion mainly failed during grasp

and return home, while RRT and RRT* completed the early phases reliably but had reduced success in later phases such as pre-drop, release, retreat, and return home. These results show that the planners were not failing randomly, but were sensitive to different parts of the workflow.

The grasp and grasp-lift phases are especially sensitive because the robot is close to the object, the gripper, the crate, and the surrounding environment. In addition, the collision representation changes when the object is removed as a world object and attached to the robot. This can make the following lift motion more difficult, since the planner must now account for the grasped object as part of the robot. These phases also required several system-level updates to happen in the correct order, including gripper commands, object attachment in the simulation, removal of the object from the world collision representation, and updating the planning representation. As a result, some pick-and-place failures may not have been caused by the planner itself, but by timing or logic errors in the surrounding task execution pipeline. This shows that the pick-and-place benchmark evaluated the complete system integration, not only the motion-planning algorithm.

Later phases, such as pre-drop, release, retreat, and return home, introduced similar difficulties. At this point, the robot is moving from a state where the object has just been detached from the robot and restored as a collision object in the environment. These transitions again required several parts of the system to be updated consistently, and small errors in this sequence could affect the following planning request or execution phase.

Overall, the pick-and-place benchmark showed that planner performance should not only be tested from isolated planning requests. The full workflow shows interactions between planning, execution, grasping, and collision-scene updates that were not shown in the simple motion benchmark. It also showed that breaking a task into intermediate phases can improve the behaviour of sampling-based planners. For this reason, the full workflow benchmark was necessary for understanding how the planners behaved as part of the complete system.

## 5.5 Hybrid Planning for Dynamic Obstacles

The dynamic obstacle benchmark shows that the hybrid planner added a capability that the direct global planners could not provide in the same way. Instead of treating the planned trajectory as fixed after execution started, the hybrid approach allowed the system to react when the environment changed. This is important because the benchmark intentionally created a situation where the original path became invalid during motion. The relevant result is therefore not only that the planner could generate a path, but that it could recover after the planned motion was disturbed. This shows that reactive planning can improve the robot's ability to continue operating in a changing workspace, but only when the change is detected early enough for the system to respond.

The clearance results show that this recovery depended strongly on how much margin remained when the obstacle was inserted. Since every successful run required the obstacle to invalidate the current path, the difference between the clearance ranges is not whether the obstacle blocked the trajectory. The difference is how much space and time remained after the path was invalidated. In the 0–1.5 cm range, no runs succeeded. At this distance, the obstacle was inserted so close to the robot that there was almost no usable margin left. Even if the invalid path was detected, the system still needed time to interrupt the invalidated motion, update the command, and, when needed, request a new global trajectory. With this little clearance, that response came too late.

The 1.5–2 cm range shows the beginning of the recovery limit. The success rate increased to 30.0 percent, which means that recovery was possible in some runs, but still unreliable. This suggests that the system was operating close to the minimum clearance needed for the hybrid pipeline to react. In this range, small differences in the robot state, obstacle placement, and where the obstacle intersected the checked part of the path could affect whether recovery was possible.

In the 2–3.5 cm and 3.5–4.5 cm ranges, the success rate increased to 70.6 percent and 94.6 percent. These ranges show that the hybrid planner was not only detecting that the path was invalid, but also had enough margin in many runs to respond before the robot came too close to the obstacle. The improvement across these ranges indicates that the planner needed both spatial clearance and enough execution time for path checking, command updates, and possible replanning to take effect. The increase from 70.6 percent to 94.6 percent also shows that the transition was gradual rather than immediate. The planner did not suddenly become reliable as soon as some clearance was available. Instead, each increase in clearance made recovery more likely.

The clearance dependence is important because the transition to complete success happened at a very small distance. From 4.5 cm and above, all runs succeeded, which is a strong result for a moving obstacle inserted during execution. At this distance, the planner had only a limited physical margin, yet it was still able to detect that the current path was invalid, update the motion, and continue toward the goal. This shows that the hybrid planner was able to react effectively even when the obstacle appeared close to the robot. At the same time, 4.5 cm should not be interpreted as a general safety margin. If the robot and obstacle are moving relative to each other, a few centimetres can disappear quickly, and the available reaction time depends strongly on the current robot speed, obstacle speed, controller update rate, and detection latency. The result is therefore surprisingly good as a benchmark outcome, but it should still be understood within the controlled conditions of the simulation.

The results show both the strength and the practical limitation of the hybrid planner. The planner was able to recover from an invalidated path with very little

clearance, which shows that reactive planning can add meaningful execution-time adaptability. However, it cannot remove the need for margins around the robot. In this benchmark, the obstacle was inserted directly into the simulated planning scene, so the planner received a clean and immediate representation of the changed environment. In a real system, the obstacle would first have to be detected by sensors, its position would have to be estimated, and the planning scene would have to be updated before the planner could react. This would reduce the effective time and distance available for recovery.

## 5.6 Cost of Reactive Planning

The longer execution and control time of the hybrid planner should be interpreted as the cost of adding reactivity to the system. This was expected, since the hybrid planner does not only execute a precomputed trajectory, but also keeps evaluating whether the motion is still valid during execution. The important point is not that the hybrid planner was simply slower, but that the additional time was connected to the same mechanism that allowed it to recover in the dynamic obstacle benchmark.

This creates a clear trade-off. In a static environment, where the original trajectory remains valid, the reactive part of the hybrid planner mainly becomes overhead. In such cases, a direct planner such as cuRobo is more efficient because it can generate and execute the motion without the same execution-time checks. The hybrid planner is therefore not the best choice if the task is fully known in advance and the main objective is to minimise execution time.

In a dynamic workspace, however, the interpretation changes. If the robot is operating near humans or moving objects, it is not enough to execute a trajectory that was valid before motion started. The system also needs a way to react when that trajectory becomes invalid during execution. The extra execution time is therefore part of what makes this possible. Without this additional checking and response mechanism, the direct planners would continue to rely on a trajectory that was planned before the obstacle appeared, and therefore could not handle the newly introduced obstacle in this benchmark setup.

The cost of reactive planning should therefore be understood as a necessary cost for this type of behaviour. For static tasks, the added execution time is mostly unnecessary. For dynamic or collaborative tasks, it is required that the robot should be able to respond to humans or moving objects after execution has started. This makes the hybrid planner less efficient in simple static cases, but more suitable when the workspace can change during motion.

## 5.7 Implications for Human-Robot Collaboration

The dynamic obstacle results are especially relevant for human-robot coexistence because they show that reactive motion planning can allow the robot to continue

operating when the workspace changes during execution. In a shared workspace, the robot cannot assume that the environment will remain fixed after a trajectory has been planned. A human may reach into the workspace, move close to the robot, or enter the planned path while the robot is moving. The hybrid planner showed that the robot could respond to this type of change in simulation, and the fact that it reached complete success from a clearance of only 4.5 cm makes the result particularly relevant for dynamic workspaces.

The humanoid obstacle demonstration strengthens this result by showing that the reactive behaviour was not limited to an artificial benchmark object. In that test, the hybrid planner responded to a moving humanoid collision model during the home_to_lower_crate motion, stopped when the lower crate area was occupied, and continued once the path became free again. This is an important step toward more realistic human-robot coexistence, since the obstacle was no longer only a simple spawned shape but part of an animated humanoid model integrated into the simulated workspace. The demonstration therefore shows that the implemented pipeline can connect humanoid animation, collision representation, planning-scene updates, and reactive hybrid planning into one working system.

At the same time, these results should not be interpreted as a complete human-robot safety solution. Both the benchmark obstacle and the humanoid demonstration were performed in simulation, where the obstacle representation was available directly to the planning system. In a real collaborative workspace, the system would first need to detect the human or object with sensors, estimate its position, convert it into a suitable collision representation, and update the planning scene before the planner could react. This means that the available margin in a real system would depend not only on the hybrid planner, but also on the perception and scene-update pipeline around it.

This is especially important for human obstacles, where the relevant question is not only where the human is now, but where the human is likely to move next. If the system only reacts after a human has already entered the planned path, the available clearance may be too small for a real system, even if the planner itself can respond quickly in simulation. In physical deployment, additional delays and uncertainties could be introduced by sensor measurements, perception processing, planning-scene updates, controller response, and possible tracking errors in the robot joints. These effects may reduce the effective reaction margin compared with the idealised simulation case. Human-motion prediction could therefore be an important extension, since it may allow the robot to account for likely future human motion before the human has already entered the planned path.

Speed adaptation could also be added as an early response in a real collaborative system. If a human is detected at a larger distance, the robot could reduce its speed before the person reaches the planned path or enters the closest workspace region. This would give the system more time to update the planning scene and allow the reactive planner to respond with a larger remaining margin. The clearance results

support this, since the hybrid planner recovered more reliably when there was more distance available after the path became invalid.

## 5.8 Computational Cost and Deployment Considerations

The resource usage results show that computational cost is an important part of the planner comparison. The clearest difference was between the GPU-based planners and the OMPL-based planners. RRT and RRT* required only a few hundred MiB of RAM and did not use GPU memory, while cuRobo, cuMotion, and the hybrid planner all required more than 2 GiB of RAM on average and several GiB of GPU memory. This means that the GPU-based planners require more powerful hardware, while the OMPL planners can run on much simpler systems.

The differences between the GPU-based planners were relatively small compared with the difference between GPU-based planning and OMPL. cuRobo, cuMotion, and the hybrid planner all used a similar amount of GPU memory, with average GPU memory usage around 5–6 GiB. This suggests that the main deployment question is not whether cuRobo, cuMotion, or the hybrid planner is slightly more memory efficient, but whether the application can justify the use of a GPU-based planning system at all. For systems with limited hardware resources, the OMPL planners are easier to deploy and maintain, since they do not require GPU memory or CUDA-dependent planning components.

However, the OMPL planners were computationally cheap, but they also had lower success rates in several benchmark cases. RRT showed very short planning times when it succeeded, but its lower robustness makes it less suitable for difficult tasks where repeated failures would interrupt the workflow. In contrast, the GPU-based planners required more hardware resources, but provided higher success rates and more reliable behaviour in the more constrained planning problems. This creates a trade-off between hardware cost and task performance.

For practical deployment, the choice of planner should therefore depend on the difficulty of the task and the available hardware. In simpler or more structured environments, an OMPL-based planner may still be sufficient and attractive because of its low computational requirements. In more constrained or dynamic environments, the additional cost of GPU-based planning may be justified by improved planning reliability, faster trajectory generation, and the possibility of reactive behaviour. This is especially relevant for human-robot collaboration, where the robot may need to respond to changes in the workspace during execution rather than only execute a precomputed trajectory.

## 5.9 Limitations

The main limitation of this work is that all experiments were performed in simulation and used ground-truth information from the simulated environment. This was useful for comparing the planners under controlled conditions, since the same scenes, objects, robot states, and obstacle configurations could be reused across many runs. It also made it possible to isolate differences between planners without adding uncertainty from perception or hardware behaviour. However, this means that the results should be interpreted as simulation results rather than direct proof of real robot performance.

In a real system, the robot would not have perfect knowledge of the workspace. Objects, humans, and obstacles would have to be detected and tracked using sensors, and this information would then have to be converted into collision objects for the planners. This would introduce additional sources of uncertainty, such as sensor noise, calibration errors, delayed measurements, and simplified obstacle geometry. These effects could affect both the success rate of the planners and the ability of the hybrid planner to react in time to dynamic obstacles.
Real hardware would also introduce effects that were not fully captured in the simulation, such as controller tracking errors, joint behaviour, vibration and contact dynamics. Even though the simulation does a good job of simulating this, it is not perfect. Further testing on physical hardware would therefore be needed to evaluate how well the results transfer to a real human-robot collaboration setting.

This does not make the comparison invalid, but it limits what can be concluded from it. The simulation results are still useful for showing relative differences between the planners in the tested setup. For example, the results show which planners were faster, more successful, or produced more consistent motion under the same conditions. What they do not show is how much of this performance would transfer directly to a physical system. A planner that works well in simulation may still need additional tuning on hardware, especially when small clearances, grasping, and collision checking are involved.

The dynamic obstacle benchmark also had a limited scope. It tested one controlled type of obstacle insertion during a specific motion, which made it possible to study the effect of clearance in a clear way. This means that the results should not be generalised to all possible human movements. A person in a real workspace may approach from different directions, move at different speeds, stop unexpectedly, or only partially enter the robot workspace. The benchmark shows that the hybrid planner can recover in the tested dynamic scenario, but it does not cover the full range of situations that could occur in human-robot collaboration.

Another limitation is that the benchmark results are specific to the computer and software environment used in this project. Planning time, execution time, resource usage, and in some cases even success rate can be affected by the CPU, GPU, available memory, driver setup, and planner configuration. This is especially relevant

for the GPU-based planners, since their performance depends strongly on the available GPU resources. A computer with lower computational capacity could lead to longer planning times, slower replanning, higher risk of timeouts, or reduced success rates under the same benchmark settings. The measured values should therefore be interpreted as results for the tested hardware and software setup, not as fixed performance values for the planners in general. However, the results still provide a useful comparison within this setup, since all planners were evaluated on the same hardware and under the same benchmark conditions.

The planner settings also affect the interpretation of the results. Parameters such as planning time limits, trajectory optimisation weights, joint weights, collision margins, and replanning behaviour influence the balance between success rate, planning time, and motion efficiency. The results compare the planners as they were configured in this project. A different tuning strategy could change some of the numerical results, although it would likely also introduce new trade-offs. For example, increasing planning time may improve the success rate in some cases, while stricter collision settings may reduce the success or increase computation time.

These limitations mean that the results should be seen as a controlled comparison within the simulated kitting environment, rather than a final validation for deployment. The tests show clear trends in planner behaviour and demonstrate that reactive planning is possible in the tested setup. However, stronger conclusions about real collaborative operation would require hardware experiments, more varied dynamic-obstacle scenarios, and further tuning under real execution conditions.

# 6 Conclusion

This thesis developed and evaluated a simulation-based system for studying motion planning in robotic kitting tasks. The system combined ROS 2, Isaac Sim, MoveIt 2, and several motion-planning frameworks in a containerised architecture. A simulated industrial workspace was created with a gantry-mounted UR10e robot arm, a Robotiq gripper, a flow rack, crates, static collision objects, and dynamic obstacles. The result was an integrated platform where different planning approaches could be tested under repeatable conditions.

The work compared sampling-based planning with OMPL, GPU-accelerated planning with cuMotion and cuRobo, and a hybrid planning approach based on cuRobo MotionGen and MPC. The planners were evaluated using simple motion benchmarks, complete pick-and-place workflows, resource measurements, and a dynamic obstacle benchmark. This made it possible to study both isolated planning performance and the behaviour of the planners when they were used as part of a larger robotic task.

The results showed that direct cuRobo planning gave the best overall performance in the static benchmark cases. It achieved high success rates, short planning times, and efficient motions compared with the other tested planners. cuMotion also performed well in terms of success rate, but generally had longer planning times, likely because it was used through the MoveIt 2 planning pipeline. The OMPL-based planners were much cheaper in terms of computational resources and did not require GPU memory, but they were less robust in several of the tested cases. This shows that GPU-accelerated planning can be a strong option for constrained tasks, especially when fast planning and high reliability are important.

The pick-and-place benchmark showed that full workflow evaluation is important. Even if a planner performs well for individual motions, a complete manipulation task introduces additional difficulties. These include grasping, lifting, attaching and detaching objects, updating the collision scene, and moving from constrained configurations. Some failures were therefore not only related to the planning algorithm itself, but also to the interaction between planning, object handling, collision updates, and execution logic. This highlights the importance of testing planners in complete task workflows and not only in isolated motion-planning cases.

The hybrid dynamic obstacle benchmark showed that the implemented hybrid planner could react to changes in the environment during execution. This reactive

behaviour was also demonstrated with the animated humanoid model. When an obstacle appeared with enough clearance in front of the robot, the system was able to invalidate the current path, replan, and still reach the goal. However, when the obstacle appeared too close to the robot, the system had less time and space to recover, which reduced the success rate. This shows that hybrid planning is a promising approach for dynamic environments, but that its performance depends strongly on detection timing, available clearance, local control behaviour, and re-planning speed.

Overall, the thesis shows that simulation is a useful method for developing and evaluating robotic motion-planning systems before real-world deployment. The implemented platform made it possible to compare different planning methods, study their strengths and weaknesses, and test dynamic obstacle handling in a safe and repeatable way. The results suggest that GPU-accelerated planning, especially direct cuRobo planning, is well-suited for constrained kitting-related robot motions. The hybrid planner also showed potential for human-robot collaboration scenarios where the robot must react to changes during execution.

All experiments were performed in simulation, and the system was not validated on physical hardware. A simplification was that the system used ground-truth information from the simulation for the robot, obstacles, and environment state. This made it possible to evaluate the planning methods under controlled and repeatable conditions, but it does not represent the uncertainty and delay that would be present in a real system. In a physical setup, this information would instead need to come from perception systems, such as cameras, depth sensors, object detection, human tracking, and state estimation. The humanoid actor and collision models were also simplified, and the dynamic obstacle scenarios only represented selected cases. The results should therefore be seen as an evaluation of the implemented simulation system and planner configurations, rather than a complete proof of real-world performance.

## 6.1 Future Work

Future work should include more dynamic obstacle benchmarks so that the hybrid planner is tested under less controlled conditions. The current benchmark was useful because it isolated the effect of obstacle clearance and made the recovery behaviour easier to interpret. However, human-robot collaboration is not usually defined by one clean obstacle insertion. A person may move near the robot for a period of time, enter and leave the workspace, change direction, or move only part of the body into the robot's path. Testing the planner more with recorded human motions would therefore give a better understanding of whether the hybrid planner can handle continuous changes in the workspace, not only one isolated path invalidation. This would also make it possible to evaluate the planner in situations where the robot has to react more than once during the same task. In the current benchmark, the main question was whether the system could recover after the path became invalid.

In a more realistic collaborative scenario, the robot may need to adapt repeatedly during the same task.

A perception system would also be an important extension. In this project, the obstacle was inserted directly into the planning scene, which was useful for testing the planner's response itself. However, this removes one of the main difficulties in a real system. The robot would not automatically know the state of the surrounding environment. It would need to detect relevant objects, estimate their positions, and update the planning scene before the planner could react. Adding this to the simulation would make the benchmark more realistic and would show how much delay or uncertainty the hybrid planner can tolerate before the recovery performance starts to decrease.

Prediction could also improve the usefulness of the reactive planner. The current system reacts when the planned path becomes invalid, but in a collaborative workspace, it would be better if the robot could respond before the human is already close to the path. If the system can estimate where a person is moving, the robot could slow down, wait, or choose another path earlier. This would shift the behaviour from only reacting to a blocked path toward avoiding critical situations before they occur.

Real robot testing would be another important step, as it would show how much of the simulation behaviour transfers to the physical system. In simulation, the robot model, object positions, and collision geometry are idealised. On the hardware side, small calibration errors, controller-tracking differences, joint behaviour, and timing delays can affect how closely the executed motion follows the planned trajectory. This is especially relevant for the hybrid planner, since its recovery behaviour depends on reacting during execution. Hardware testing would therefore make it possible to evaluate whether the planner comparison remains valid when the robot is affected by real execution errors and physical system delays.

# Bibliography


[1] Kevin M. Lynch and Frank C. Park. (2017) *MODERN ROBOTICS MECHANICS, PLANNING, AND CONTROL*. Available at: https://hades.mech.northwestern.edu/images/7/7f/MR.pdf

[2] Mark W. Spong, Seth Hutchinson, and M. Vidyasagar. (2004) *Robot Dynamics and Control, Second Edition*. Available at: https://www.kramirez.net/Robotica/Tareas/Kinematics.pdf

[3] Steven M. LaValle. (2006) *Planning Algorithms*. Available at: https://lavalle.pl/planning/

[4] Sundaralingam, B., Hari, S. K. S., Fishman, A., Garrett, C., Van Wyk, K., Blukis, V., Millane, A., Oleynikova, H., Handa, A., Ramos, F., Ratliff, N., & Fox, D. (2023). *cuRobo: Parallelized Collision-Free Minimum-Jerk Robot Motion Generation*. arXiv:2310.17274 [cs.RO]. Available at: https://arxiv.org/abs/2310.17274

[5] Sertac Karaman, Emilio Frazzoli(2011). *Sampling-based Algorithms for Optimal Motion Planning*. Available at: https://arxiv.org/pdf/1105.1186

[6] Lydia E. Kavraki, Petr Švestka, Jean-Claude Latombe, and Mark H. Overmars (1996). *Probabilistic Roadmaps for Path Planning in High-Dimensional Configuration Spaces*. Available at: https://www.cs.cmu.edu/~motionplanning/papers/sbp_papers/PRM/prmbasic_01.pdf

[7] Steven M. LaValle (1998) *Rapidly-exploring random trees : a new tool for path planning*. Available at: https://msl.cs.illinois.edu/~lavalle/papers/Lav98c.pdf

[8] John Schulman, Yan Duan, Jonathan Ho, Alex Lee, Ibrahim Awwal, Henry Bradlow, Jia Pan, Sachin Patil, Ken Goldberg and Pieter Abbeel. (2014) *Motion planning with sequential convex optimization and convex collision checking*. Available at: https://escholarship.org/uc/item/6km506db

[9] NVIDIA. (2026). *motion_gen.py* in *NVlabs/curobo*. GitHub repository, commit 2fbffc3. Available at: https://github.com/NVlabs/curobo/blob/2fbffc3/src/curobo/wrap/reacher/motion_gen.py

[10] NVIDIA. (2026). *trajopt.py* in *NVlabs/curobo*. GitHub repository, commit 2fbffc3. Available at: https://github.com/NVlabs/curobo/blob/2fbffc3/src/curobo/wrap/reacher/trajopt.py

[11] Mrinal Kalakrishnan, Sachin Chitta, Evangelos Theodorou, Peter Pastor, Stefan Schaal. (2026). *STOMP: Stochastic Trajectory Optimization for Motion Planning*. Available at: https://ros.fei.edu.br/roswiki/attachments/Papers%282f%29ICRA2011_Kalakrishnan/kalakrishnan_icra2011.pdf

[12] Williams, G., Drews, P., Goldfain, B., Rehg, J. M., & Theodorou, E. A. (2016). *Aggressive driving with model predictive path integral control.* In *2016 IEEE International Conference on Robotics and Automation (ICRA)* (pp. 1433–1440). https://doi.org/10.1109/ICRA.2016.7487277

[13] NVIDIA. (2026). *particle_opt_base.py* in *NVlabs/curobo.* GitHub repository, commit 3690d28. Available at: https://github.com/NVlabs/curobo/blob/3690d28/src/curobo/opt/particle/particle_opt_base.py

[14] NVIDIA. (2026). *parallel_mppi.py* in *NVlabs/curobo.* GitHub repository, commit 0a50de1. Available at: https://github.com/NVlabs/curobo/blob/0a50de1/src/curobo/opt/particle/parallel_mppi.py

[15] Gill, P. E., & Wong, E. (2012). *Sequential quadratic programming methods.* In J. Lee & S. Leyffer (Eds.), *Mixed Integer Nonlinear Programming* (The IMA Volumes in Mathematics and its Applications, Vol. 154, pp. 147–224). Springer, New York, NY. https://doi.org/10.1007/978-1-4614-1927-3_6

[16] Lagarias, J. C., Reeds, J. A., Wright, M. H., & Wright, P. E. (1998). *Convergence properties of the Nelder–Mead simplex method in low dimensions. SIAM Journal on Optimization*, 9(1). https://doi.org/10.1137/S1052623496303470

[17] Liu, D. C., & Nocedal, J. (1989). *On the limited memory BFGS method for large scale optimisation. Mathematical Programming*, 45, 503–528. https://doi.org/10.1007/BF01589116

[18] NVIDIA. (2026). *newton_base.py* in *NVlabs/curobo.* GitHub repository, commit 2fbffc3. Available at: https://github.com/NVlabs/curobo/blob/2fbffc3/src/curobo/opt/newton/newton_base.py

[19] Florian Mannel, Hari Om Aggrawal, Jan Modersitzki. (2023) *A structured L-BFGS method and its application to inverse problems.* arXiv:2310.07296. Available at: https://arxiv.org/pdf/2310.07296

[20] Rawlings, J. B., Mayne, D. Q., & Diehl, M. M. (2017). *Model Predictive Control: Theory, Computation, and Design* (2nd ed.). Nob Hill Publishing. Available at: https://sites.engineering.ucsb.edu/~jbraw/mpc/MPC-book-2nd-edition-4th-printing.pdf

[21] Mayne, D. Q., Rawlings, J. B., Rao, C. V., & Scokaert, P. O. M. (2000). *Constrained model predictive control: Stability and optimality.* Automatica, 36(6), 789-814. Available at: https://doi.org/10.1016/S0005-1098(99)00214-9

[22] Garcia, C. E., Prett, D. M., & Morari, M. (1989). *Model predictive control: Theory and practice—a survey.* Automatica, 25(3), 335-348. Available at: https://doi.org/10.1016/0005-1098(89)90002-2

[23] Williams, G., Aldrich, A., & Theodorou, E. A. (2017). *Model Predictive Path Integral Control: From Theory to Parallel Computation.* Journal of Guidance, Control, and Dynamics, 40(2), 344–357. Available at: https://doi.org/10.2514/1.G001921

[24] NVIDIA. (2026) *Isaac Sim.* Available at: https://github.com/isaac-sim/IsaacSim

[25] Deyna, Q. (2025). *ur10e_2f140_topic_based_ros2_control: UR10e + Robotiq 2F140 Stack with Isaac Sim.* GitHub repository. Available at: https:

//github.com/qdeyna/ur10e_2f140_topic_based_ros2_control [Accessed: 28 May 2026].
[26] Brian Gerkey, Tully Foote, Chris Lalancette, William Woodall, and Michael Carroll. (2022) *ROS 2: Design, architecture, and uses in robotics*. Available at: https://arxiv.org/abs/2211.07752
[27] NVIDIA. (2025) *What Is Isaac Sim?*. Available at: https://docs.isaacsim.omniverse.nvidia.com/5.1.0/index.html
[28] NVIDIA. (2025) *Overview, Omniverse Kit*. Available at: https://docs.omniverse.nvidia.com/kit/docs/kit-manual/110.0.0/guide/kit_overview.html
[29] NVIDIA. (2025) *Extensions Overview, NVIDIA Omniverse*. Available at: https://docs.omniverse.nvidia.com/extensions/latest/index.html
[30] Pixar Animation Studios. (2025) *Introduction to USD*. Available at: https://openusd.org/dev/intro.html
[31] NVIDIA. (2025) *Physics, Isaac Sim Documentation*. Available at: https://docs.isaacsim.omniverse.nvidia.com/5.1.0/physics/index.html
[32] NVIDIA. (2023) *Rigid Body Dynamics, NVIDIA PhysX Documentation*. Available at: https://nvidia-omniverse.github.io/PhysX/physx/5.6.1/docs/RigidBodyDynamics.html
[33] NVIDIA. (2024) *Articulations, PhysX 5.5 Documentation*. Available at: https://nvidia-omniverse.github.io/PhysX/physx/5.5.0/docs/Articulations.html
[34] asai taspats. (2018) *E2 type stacking enclosure, GrabCAD*. Available at: https://grabcad.com/library/e2-type-stacking-enclosure-1
[35] Xynerva. (2025) *Flow Rack, GrabCAD*. Available at: https://grabcad.com/library/flow-rack-3
[36] Autodesk. (2026) *Fusion 360*. Available at: https://www.autodesk.com/se/products/fusion-360/overview [Accessed: 27 May 2026].
[37] Blender Foundation. (2026) *About Blender*. Available at: https://www.blender.org/about/ [Accessed: 27 May 2026].
[38] B. Kerbl, G. Kopanas, T. Leimkühler, and G. Drettakis (2023) *3D Gaussian Splatting for Real-Time Radiance Field Rendering* Available at: https://arxiv.org/abs/2308.04079
[39] NVIDIA Toronto AI Lab, *3DGRUT: Ray tracing and hybrid rasterization of Gaussian particles*, GitHub repository, 2025. [Online]. Available: https://github.com/nv-tlabs/3dgrut. Accessed: Jun. 3, 2026.
[40] Sucan, Ioan A. and Moll, Mark and Kavraki, Lydia E. (2012) *The Open Motion Planning Library IEEE Robotics & Automation Magazine*, vol. 19, no. 4, pp. 72–82. doi: https://doi.org/10.1109/MRA.2012.2205651.
[41] PickNik Inc. (2026) *MoveIt 2*. Available at: https://moveit.picknik.ai/main/index.html#
[42] Sachin Chitta. (2016) *MoveIt!: An Introduction*. In: Anis Koubaa (ed.), *Robot Operating System (ROS)*, Studies in Computational Intelligence, vol 625. Springer, Cham. Available at: https://doi.org/10.1007/978-3-319-26054-9_1

[43] MoveIt, *Planning Scene*. Available at: https://moveit.picknik.ai/main/doc/examples/planning_scene/planning_scene_tutorial.html [Accessed: 4 May 2026].

[44] NVIDIA, *Isaac ROS cuMotion*. Available at: https://nvidia-isaac-ros.github.io/repositories_and_packages/isaac_ros_cumotion/index.html [Accessed: 5 May 2026].

[45] NVIDIA, *cuRobo and cuMotion*. Available at: https://docs.isaacsim.omniverse.nvidia.com/5.1.0/manipulators/manipulators_curobo.html [Accessed: 5 May 2026].

[46] NVIDIA Research, *cuRobo Documentation*. Available at: https://curobo.org/ [Accessed: 5 May 2026].

[47] Lab-CORO, *curobo_ros*. Available at: https://github.com/Lab-CORO/curobo_ros [Accessed: 4 May 2026].

[48] MoveIt, *Hybrid Planning*. Available at: https://moveit.picknik.ai/humble/doc/examples/hybrid_planning/hybrid_planning_tutorial.html [Accessed: 5 May 2026].

[49] Infotiv Research. (2026) *curobo_global_planner.hpp* in *RITA-SIM*. GitHub repository. Available at: https://github.com/infotiv-research/RITA-SIM/blob/main/src/curobo_hybrid_planning_plugins/include/curobo_hybrid_planning_plugins/curobo_global_planner.hpp

[50] Infotiv Research. (2026) *lookahead_sampler.hpp* in *RITA-SIM*. GitHub repository. Available at: https://github.com/infotiv-research/RITA-SIM/blob/main/src/curobo_hybrid_planning_plugins/include/curobo_hybrid_planning_plugins/lookahead_sampler.hpp

[51] Infotiv Research. (2026) *curobo_mpc_local_solver.hpp* in *RITA-SIM*. GitHub repository. Available at: https://github.com/infotiv-research/RITA-SIM/blob/main/src/curobo_hybrid_planning_plugins/include/curobo_hybrid_planning_plugins/curobo_mpc_local_solver.hpp

[52] Rokoko. (2026) *Rokoko Vision*. Available at: https://www.rokoko.com/products/vision

[53] Avaturn. (2026). *What is Avaturn?* Avaturn Documentation. Available at: https://docs.avaturn.me/docs/what-is-avaturn/ [Accessed: 2 June 2026].

# A

# Simple Motion Cases

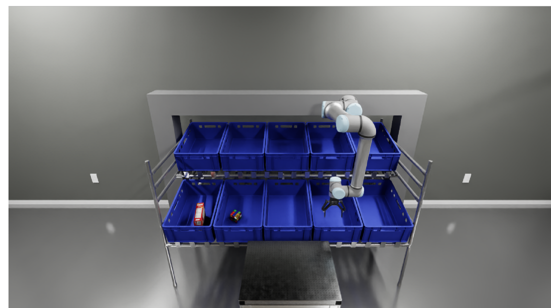
This appendix shows the start and goal positions used in the simple motion cases.

Start position

Goal position

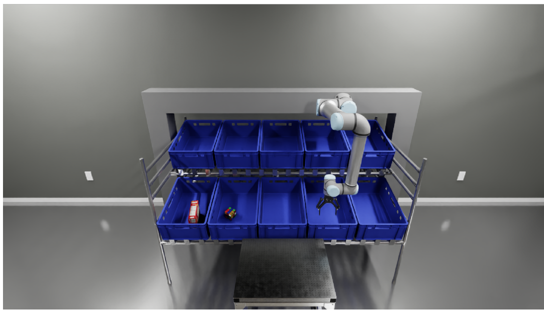
**Figure A.1:** Between lower to lower crate.

Start position

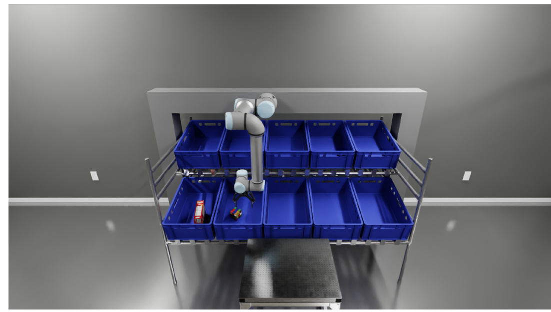


Goal position

**Figure A.2:** Home to lower crate.

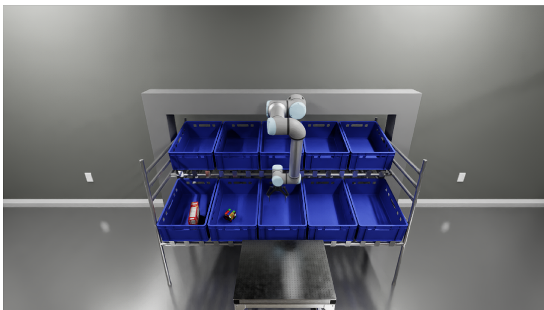


Start position



Goal position

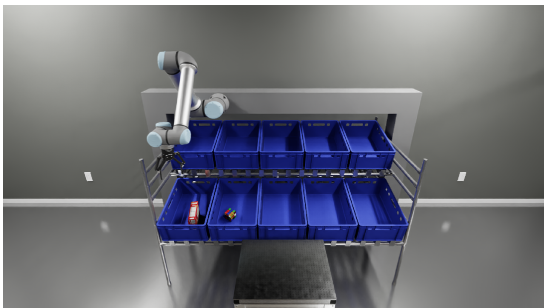
**Figure A.3:** Lower to lower crate.



Start position

Goal position

**Figure A.4:** Lower to upper crate.



Start position

Goal position

**Figure A.5:** Move above crates.

# B

# Pick-and-Place Phases

This appendix shows the robot positions used for the different phases of the pick-and-place benchmark.

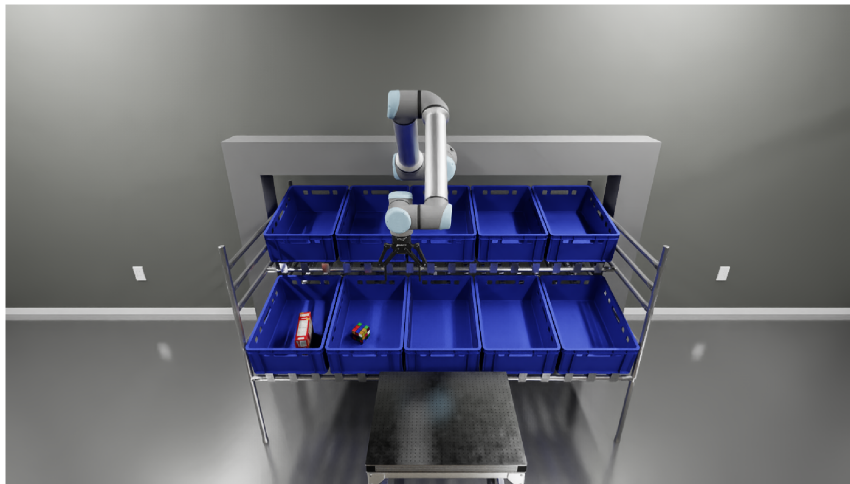


**Figure B.1:** Home position.

**Figure B.2:** Pre grasp position.

**Figure B.3:** Grasp position.

**Figure B.4:** Grasp lift position.

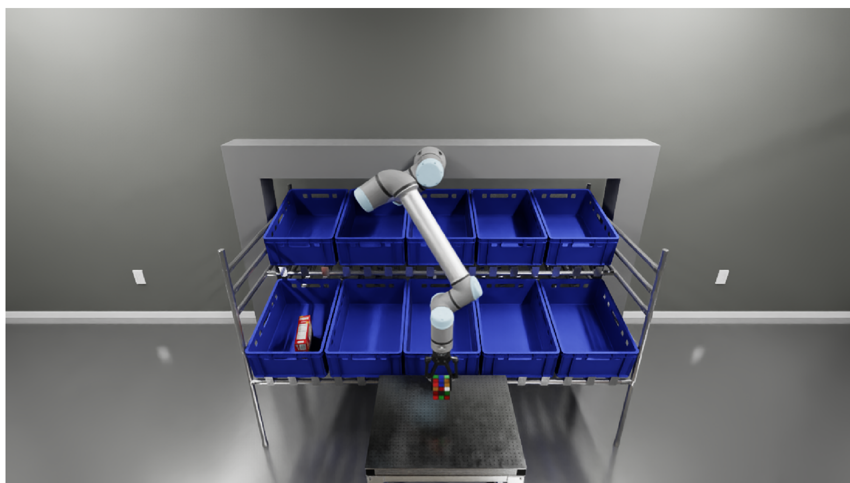


**Figure B.5:** Pre drop position.

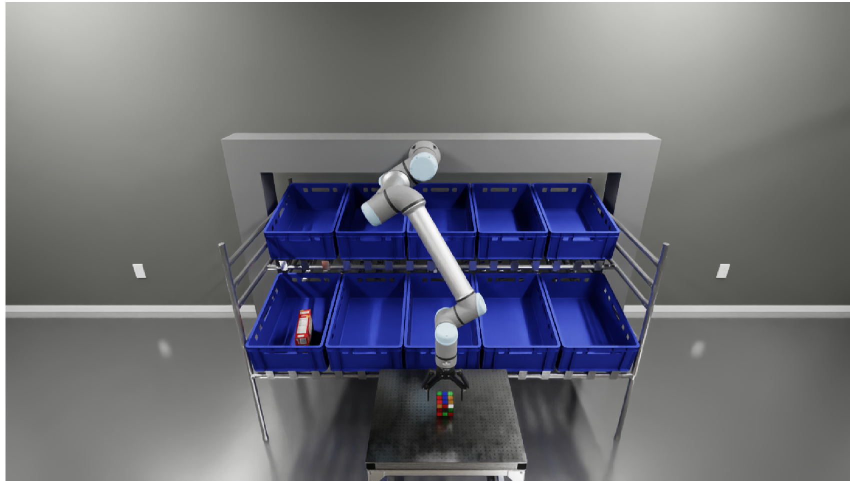


**Figure B.6:** Release position.

**Figure B.7:** Release retreat position.

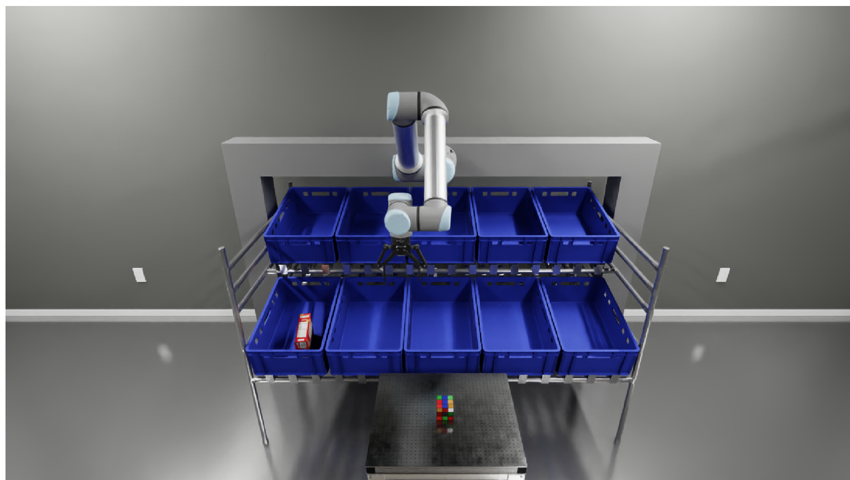


**Figure B.8:** Return home position.

# C

# Detailed Result Tables

**Table C.1:** Pick-and-place workflow timing, reported as mean ± standard deviation in seconds over successful runs.

| Planner | Total time | Global planning time | Execution/control time |
|---|---|---|---|
| cuMotion | $53.02 \pm 1.36$ | $22.63 \pm 0.71$ | $18.41 \pm 1.01$ |
| cuRobo | $42.80 \pm 4.39$ | $5.16 \pm 1.55$ | $26.80 \pm 4.01$ |
| Hybrid | $67.32 \pm 11.47$ | – | $52.27 \pm 10.91$ |
| RRT | $29.37 \pm 5.41$ | $0.91 \pm 0.36$ | $16.27 \pm 5.18$ |
| RRT* | $106.40 \pm 12.07$ | $77.13 \pm 8.73$ | $15.63 \pm 4.62$ |

**Table C.2:** Per-phase planning time in the pick-and-place benchmark, reported as mean ± standard deviation in seconds over successful phase executions.

| Planner | pre_grasp | grasp | grasp_lift | pre_drop | release | release_retreat | return_home |
|---|---|---|---|---|---|---|---|
| cuMotion | $3.81 \pm 0.2$ | $4.15 \pm 0.2$ | $4.13 \pm 0.3$ | $3.43 \pm 0.2$ | $3.24 \pm 0.2$ | $3.77 \pm 0.2$ | $0.07 \pm 0.1$ |
| cuRobo | $0.46 \pm 0.1$ | $2.40 \pm 1.5$ | $0.53 \pm 0.4$ | $0.48 \pm 0.1$ | $0.50 \pm 0.1$ | $0.45 \pm 0.1$ | $0.37 \pm 0.1$ |
| RRT | $0.04 \pm 0.1$ | $0.13 \pm 0.1$ | $0.05 \pm 0.1$ | $0.27 \pm 0.2$ | $0.34 \pm 0.3$ | $0.07 \pm 0.1$ | $0.04 \pm 0.1$ |
| RRT* | $10.02 \pm 0.1$ | $20.01 \pm 0.1$ | $10.02 \pm 0.1$ | $13.87 \pm 6.4$ | $14.26 \pm 4.9$ | $8.89 \pm 4.2$ | $0.04 \pm 0.1$ |

**Table C.3:** Total arm joint motion in the pick-and-place benchmark, reported as mean ± standard deviation in degrees over successful runs.

| Planner | Total arm joint motion [deg] |
|---|---|
| cuRobo | $1405.89 \pm 376.05$ |
| cuMotion | $1953.34 \pm 163.02$ |
| Hybrid | $1501.08 \pm 323.78$ |
| RRT | $1955.54 \pm 1044.32$ |
| RRT* | $1755.64 \pm 889.33$ |

**Table C.4:** Total gantry motion in the pick-and-place benchmark, reported as mean ± standard deviation in metres over successful runs.

| Planner | Total gantry motion [m] |
|---|---|
| cuRobo | $1.57 \pm 0.31$ |
| cuMotion | $1.30 \pm 0.27$ |
| Hybrid | $1.72 \pm 0.18$ |
| RRT | $1.85 \pm 0.79$ |
| RRT* | $1.84 \pm 0.78$ |

**Table C.5:** Success rate for the hybrid dynamic obstacle benchmark grouped by robot clearance at obstacle spawn.

| Clearance range | Runs | Success rate |
|---|---|---|
| 0–1.5 cm | 19 | 0.0% |
| 1.5–2 cm | 10 | 30.0% |
| 2–3.5 cm | 17 | 70.6% |
| 3.5–4.5 cm | 37 | 94.6% |
| 4.5 cm and above | 63 | 100.0% |

DEPARTMENT OF MECHANICS AND MARITIME SCIENCES
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2026
www.chalmers.se

CHALMERS
UNIVERSITY OF TECHNOLOGY