http://eprints.gla.ac.uk/279056/

Deposited on 9 September 2022

A Systematic Comparison of Simulation Software for Robotic Arm Manipulation using ROS2

Florent P. Audonnet[1*], Andrew Hamilton[2] and Gerardo Aragon-Camarasa[1,2]

[1] School of Computing Science, University of Glasgow
G12 8QQ, Scotland, United Kingdom (f.audonnet.1@research.gla.ac.uk) * Corresponding author
(gerardo.aragoncamarasa@glasgow.ac.uk)
[2] National Manufacturing Institute in Scotland
Scotland, United Kingdom (andrew.w.hamilton@strath.ac.uk)

Abstract: Simulation software is a powerful tool for robotics research, allowing the virtual representation of the real world. However, with the rise of the Robot Operating System (ROS), there are new simulation software packages that have not been compared within the literature. This paper proposes a systematic review of simulation software compatible with ROS version 2. The focus is research in robotics arm manipulation as it represents the most often used robotic application in industry and their future applicability to digital twins. For this, we thus benchmark simulation software under similar parameters, tasks and scenarios and evaluate them in terms of their capability for long-term operations, success at completing a task, repeatability and resource usage. We find that there is no best simulation software overall. Still, two simulation packages (Ignition and Webots) have higher stability than others while, in terms of resource usage, PyBullet and Coppeliasim consume less than their competitors.

Keywords: Simulation, Benchmarking, Digital Twins, Robotic

1. INTRODUCTION

With the advent of deep learning technologies, current research efforts have been focused on teaching robots how to perform various tasks autonomously. However, a data-driven approach is required to acquire and process the vast amount of data to effectively teach a robot how to perform an unfeasible task using a real robotic testbed. For this, robot simulation software [5, 6, 17, 18, 22] have been used to overcome the shortcomings of data-hungry AI approaches and to allow the developer to obtain a constant environment [?]. The world can be controlled in a simulated environment, including aspects that would be impractical in reality. There is also no risk of damaging the robot or human operators, and simulations allow time control, increasing the data collection speed.

Simulations are the gateway for Digital Twins, a high-fidelity representation of the physical world [11], and can allow manufacturing to increase production and flexibility of supply chains. Therefore, digital twinning consists of interconnecting a simulation software to a real autonomous robotic system to reduce the implementation time of the manufacturing process when changing a production line. A recent example of a digital twin solution for a robotic arm can be found in [21] where the authors used ROS (Robot Operating System) [19] to achieve seamless operation between the real and digital world. However, simulation software is imperfect because its physics engines do not accurately represent the real world. Furthermore, simulations allow for perfect data capture with no noise, which has powered research in deep learning approaches for robotics.

In this paper, we propose to carry out a systematic benchmark of current simulation software (Figure 1)

Name	Logo	Physics Engine	Headless Support	Open Source	ROS2 Support	ML Support
Gazebo		Bullet, DART, ODE, Simbody	Full	Yes	Yes	External
Ignition		DART	Full	Yes	Yes	External
Webots		ODE	Partial	Yes	Yes	External
Isaac Sim		PhysX	Full	No	Yes	Integrated
Unity		Havok, PhysX, RaiSim	Full	No	No	External
PyBullet		Bullet	Full	Yes	No	External
CoppeliaSim (Vrep)		Bullet, Newton, ODE, Vortex Dynamics	Full	No	Yes	External
Mujoco		Mujoco	Full	Yes	No	External

Fig. 1. Overview of the Simulation Software and Their Capabilities

to investigate their performance and suitability to perform different robotic manipulation tasks using the ROS2 (Robot Operating System version 2). ROS has become the de facto communication platform for modern robotic systems. We choose ROS2 because it supports a wide array of devices (e.g. micro-controllers), enabling the Internet of Things (IoT) integration. The latter is the main requirement for developing a working digital twin system. ROS2 can also be used to bridge the gap between AI-enabled robots and real-world robot control. We choose robotic arms in this paper as they are prevalent in automated manufacturing operations.

We consider 2 tasks for the robot arm to perform. The first task is about picking and placing an object which is a common operation in the industry. The second task
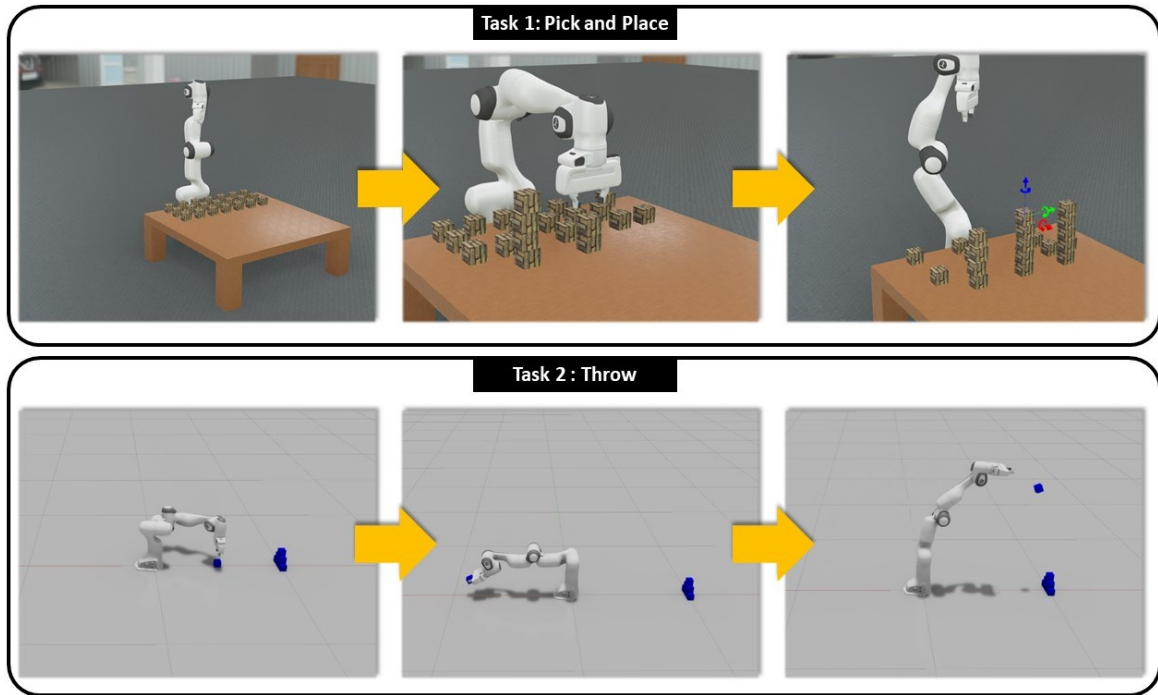
Fig. 2. Simulation Tasks Progression over Time. Task 1 (top), is a Pick and Place task where the goal is to stack 3 columns of 5 cubes. Task 2 (bottom) is a Throw task where the goal is to collapse a pyramid of 6 cubes by throwing a cube at it.

consists of throwing a cube into a pyramid. We chose this throwing task as we aim to test the accuracy and repeatability of the simulation software to decide its potential suitability for building digital twins. Figure 2 shows an overview of the tasks. We record the resource usage of each simulation considered in this paper while performing each task in both a headless and a graphical version. Our contributions include proposing a systematic comparison of state-of-the-art robotic arm simulation software using ROS2, the state-of-the-art version of the robot operating system. Furthermore, we develop an experimental methodology to evaluate robot simulation software for long-term operations and their success at completing a task. We also devised an experimental validation system to evaluate the stability of robot simulation software and its capacity to repeat a given task.

2. BACKGROUND

Benchmarking robotic simulation software can trace its origins to Oreback, and Christensen [14]. They were the first to propose a methodology to test robotic simulations. Their approach consisted of summarising the capabilities of 3 simulators, considering quantitative values such as supported OS or programming languages and qualitative opinions such as the learning curve or the difficulty of installation. They also recorded the amount of RAM used to control a real robot using the simulation software. Kramer and Scheutz [9] extended [14] and developed a comprehensive testing suite for open-source robot simulation software. They devised a set of criteria based on the software-development process. They cre-

ated a feature score based on different properties such as usability, supported features (path planning, voice recognition, etc.) and faults handling. They performed a low-level task implementation on a real robot and recorded the resource usage. However, the task is scarcely described and was only repeated three times.

Before ROS [19], roboticists used the simulation software as a middleware to send data and commands to the robot, e.g. [9]. Staranowicz and Mariottini [20] provided the first comparison of simulation software that used ROS as the communication framework to control a robot. They compared the properties of three open source and commercial simulations. They then demonstrated the capabilities of Gazebo [8], a popular simulation software with a real robot, effectively creating a Digital Twin. However, they neither recorded the resources usage nor tried a different simulator for the real-world task. Their work was then extended by Nogueira [13], who compared 2 simulators and their integration with ROS, the ease of building the world and the CPU usage.

Pitonakova *et al.* [15] adopted the methodology in [13]. They compared three simulators and then ran extensive tests to record each simulator's performance on tasks involving multiple robotic arms. They recorded memory, CPU usage, and the real-time factor, meaning the speed at which the simulation runs. This is vital for Digital Twining. It is also vital for machine learning, as the faster the simulation runs without compromising the physics, the faster the training of a machine learning model would be. They performed each test with and without a Graphical User Interface (GUI) and then compared the impact of rendering the simulation to a screen. Ayala *et al.* [1] and Korber *et al.* [10] followed the idea of

recording resources usage during the running of the experiment. After recapitulating the properties of each simulator, they coded tasks and recorded memory and CPU usage. Korber *et al.* compared four simulation software on robotic manipulation tasks while Ayala *et al.* only compared 3 for humanoid robot scenarios.

In this paper, we initially consider eight robot simulation software but narrow our benchmark to five that support ROS2, including two simulation software that have not been considered in the literature. We also propose to implement pick and place and throwing tasks to investigate the advantages and limitations of each simulation software, their performance and, ultimately, their suitability for Digital Twins.

3. MATERIALS AND METHODS

To evaluate and compare robotic simulation software, we develop our methodology and experiments guided by the following research questions:

RQ1 How does simulation software compare in terms of supporting long-term operations while still succeeding at completing a given task?

RQ2 How repeatable is the simulation software under the same scene and task constraints?

RQ3 Which simulation software would be more suitable for machine learning research in terms of resource usage and idle time?

3.1 Simulation Software

The above research questions inform our choice of the simulation software investigated in this paper, as shown in Figure 1. Not all simulation software has support for ROS2. For this paper, we have attempted to implement our ROS2 bridge but with limited success due to the rapid development cycle of ROS2. For completeness, we describe our experience while implementing the ROS2 bridge for the simulations we do not use in this paper. Unity's existing bridge is limited as it does not support asynchronous communications, which are the underlying communication paradigm in ROS2. Mujuco conflicts with ROS2 because the ROS2 multithreaded controller is incompatible with Mujuco's single threading nature. Finally, we had to drop Gazebo because development efforts have turned to Ignition, and there is currently an implementation error in the controller code, causing our robot to move erratically[1].

We also consider simulations that feature a headless mode. This is because a headless mode is critical in a machine learning context (ref. **RQ3**). Therefore, we analyse the GUI's impact on resource usage. The robot simulation software examined in this paper are:

1) Ignition [17] is a set of open source software libraries which are arranged as multiple modular plugins written in Ruby and C++. Open Robotics has developed them since 2019. It has a similar communication princi-

ple to ROS2. We chose this simulator as it is the successor of Gazebo.

2) Webots [22] has been developed since 1998 by Cyberbotics, a spin-off company from EPFL (Swiss Federal Institute of Technology Lausanne). It supports a wide range of sensors and robot controllers out of the box, is well documented and includes several examples files. Figure 1 has partial headless support because it only disables the simulation rendering. There is still a GUI visible. We considered it because it is one of the oldest simulation software still actively developing.

3) Isaac Sim [5] is a recent, Linux-only, simulation environment developed by Nvidia which runs on the PhysX engine and can be programmed in Python or C. By default, it integrates machine learning capabilities and has in-built plugins to generate synthetic data for domain adaptation and transfer learning. The latter is possible because of its ray tracing capabilities, allowing for a visual simulation as close to reality. While it can be run in headless mode, this is not possible while using their ROS2 extension since there is an issue with the ROS2 plugin not loading when launched from a python script instead of from the terminal.

4) PyBullet [6] is a Python-based simulation environment based on the Bullet physics engine, which has been in development since 2016. It is popular for machine learning research as it is lightweight and easy to use. For this paper, we implemented a ROS2-compatible plugin since there is no official ROS2 support.

5) Coppeliasim [18], previously known as V-REP, is a partially closed source simulation environment developed since 2010. It can be programmed directly in Lua or external controllers in 6 other languages. We decided to include it in our research as it has been compared in previous simulation software reviews, e.g. [1, 7, 12, 13, 15].

3.2 Data Capturing

For data capturing and recording, we adopt the metrics mentioned in Section 2.namely, the processor usage (CPU), the memory usage (RAM) and the execution time (ref. **RQ1 & 3**). We also record task-specific data, such as the number of cubes placed or moved (ref. **RQ2 & 3**). The execution time is not mentioned in the literature but was added as a metric for machine learning, in which the running time can have major impact, as training a model involves several iterations of the simulation. A delay of tens of seconds for one iteration can turn into long training sessions. To accurately record each simulation, we start recording 5 seconds before the simulation starts and ends the recording 60 seconds after the task has ended. We record processes related to the simulation while discarding the rest, such as OS-specific processes.

3.3 Robotic Tasks

We consider 2 tasks, each divided into 2 sub-tasks, to evaluate each simulator considered here. A sub-task is repeated 20 times to reduce the variance during data recording and to obtain an accurate statistical characterisation of a simulation. In practice, we found that more

---

Table 1. Sub-Task overview and Design rationale

Name	Design	Features Tested	Data Recorded and Rationale
Task 1-A: Pick, and Place	A robotic arm randomly takes 5 cm cubes from a table with 21 cubes arranged in a 7×3 grid. The task is to stack them into 3 towers of 5 cubes, as seen in Figure 2. We consider 3 stacks to leave cubes on the table and to allow for more diversity in each repetition. We set the limit to 5 stacked cubes due to the table's height and the robot's capabilities.	Friction, Gravity, Inertia	This experiment addresses **RQ2** which analyses the numbers of cubes correctly placed. It will also test the stability and suitability of the simulation for long operations, as recorded by the number of cubes still in place at the end (ref. **RQ1**). The idea of stacking cubes to analyse performance is motivated from [10]
Task 1-B: Pick and Place Headless	We use the same setup as Task 1-A but without a GUI. This was chosen as in a machine learning setting; experiments need to be restarted multiple times and often run on a server with no screen (ref. **RQ3**).		
Task 2-A: Throwing	A robotic arm will pick up a cube and throw it towards a pyramid of 6 cubes. The arm goes as far back as mechanically possible and performs a throwing motion towards the pyramid in front of it. Figure 2 shows the trajectory taken by the robot during this task. The cube is released at 50% of the trajectory. The pyramid is placed such that a successful throw at full power will collapse it.	Friction, Gravity, Inertia, Kinetic	This task benchmarks the accuracy and repeatability of the simulation software and addresses **RQ2**. The latter is carried out by recording the number of cubes displaced from their original position. This idea has been inspired by a contributor to Ignition[2] demonstrating how to interface ROS2 and Ignition.
Task 2-B: Throwing Headless	We follow the same design as Task 2-A, except without a GUI (ref. **RQ3**).		

than 20 repetitions do not result in a statistically significant difference. The 2 tasks and their rationale are summarised in Table 1. The task execution logic is the same for all simulations. We must note that we use the default simulation parameters to set up these tasks. This is to remove bias while implementing the tasks and avoid tuning simulator-specific parameters in order to obtain an objective evaluation for each simulation software.

3.4 Robot Control

There are 3 methods to control a robot using ROS2: the joint controller, the joint trajectory follower and the general purpose ROS controller. The joint controller sets the position of the joints to a given joint angle using hardware-specific interfaces of a robot. This is the simplest method as it provides no feedback to the controller. The joint trajectory follower uses a ROS action client/server combination in which the client sends the joint position for a given trajectory as a list along with a delay. Then, the server continuously sends the current value of the joints as a feedback mechanism until the trajectory is completed. This method works well in practice and we have implemented it for Coppeliasim, PyBullet and Isaac Sim. For the Ignition and Webots, we use the general purpose ROS controller (`ros_control`) [3],

which is not implemented for the other simulations. It provides a wrapper for the joint trajectory follower described above and different methods of control such as a velocity, effort or gripper controller.

4. EXPERIMENTS

4.1 Methodology

We use a docker container with Nvidia Ubuntu 20.04 `cudagl` image for all simulators except for Isaac Sim that cannot access the base graphics driver API when using docker. Isaac Sim is thus executed in the base system from where we run all experiments. ROS2 Foxy has been installed, along with simulator-specific packages. Docker has been used to easily provide an image with all the necessary packages installed without conflict between different simulations. It also allows for reproducibility of these experiments by providing the same setup every time. The base system runs an Intel I7 10700 with 32GB of RAM and an Nvidia GeForce RTX 3060 with Ubuntu 20.04. We used `psutil`[3] which is a python package that records the CPU and RAM usage. Each process was monitored at 10 Hz to minimise the resource impact. We

used the recommended time step for each simulator, and we fixed all simulators to run in real-time. We use the Franka Panda robot and its model and configuration files provided by MoveIt 2 [4]. The project repository can be found at `https://github.com/09ubberboy90/ros2_sim_comp`.

4.2 Implementation

The implementation comprises 4 components as shown in Figure 3 and as noted below.

1. *Simulator:* launches the simulation software and spawns the robot along with the gripper and arm controllers.
2. *Pathfinder:* launches Rviz (a ROS data visualisation software) and MoveIt 2.
3. *Controller:* chooses which object to grab from the list of collision objects advertised by MoveIt 2, shown in Figure 3 as blue arrows. Then, the pick and place or throwing task is executed accordingly.
4. *The Object-Handler:* spawns the objects and publishes their position to the planning scene of MoveIt 2 at 2 Hz. We choose 2 Hz because the scene's rate of change in real time does not deviate considerably. Higher rates consume more CPU usage, which impacts the performance of all simulations in this paper.

In our implementation, both the arm and the gripper are controlled using a position controller. We must note that the gripper controller available in ROS1 has not yet been ported to ROS2. The latter causes issues while grasping objects in the simulations (except Webots) as the amount of force is constant with no feedback. To mitigate this issue, we command the gripper to close its gripper beyond the optimal closing distance to ensure that the object is grasped correctly. Webots does not have this issue because it implements a PID controller for each simulated motor by default. These 4 components are launched using a single ROS2 launch file with ad hoc delays to ensure everything is started, as Parts 3 and 4 do not work unless part 2 is launched. For each simulation software, we are using the default physics time step. The physics engine is also the default, except for Coppeliasim, in which we use the *Newton physics engine* because the other supported physics engines cause the gripper to fail to grip the cube.

4.3 Experiment 1

Table 2 shows the result of task 1, which addresses **RQ1 & RQ2**. The reason the task times out (failure in Table 2) is because the ROS controller fails to start, or in the case of Coppeliasim, the client refuses to connect to the simulation server. The rest of the metrics only focus on successful attempts.

Ignition and PyBullet did not have timeouts; however, PyBullet performs significantly worse at stacking 5 towers than the other simulators as 15% of the cubes on average (i.e 3 cubes) were correctly positioned at the end of the simulation, and, therefore, the robot does not encounter scenarios where it collapses towers. Ignition and Webots are the best performing simulations for executing the task of stacking cubes and keeping the cubes in place. Coppeliasim and Isaac Sim, are carrying out the task well at placing the cube in the right place but, tend to have situations where the robot collapses the towers. Furthermore, while Coppeliasim achieves 92% success of placing cubes, we can observe that it often times out, and reduces its overall success. We can also observe in Table 2 that there is no statistically significant difference between headless and GUI modes. These results suggest that Ignition (headless and GUI) succeeds at completing the task more frequently using the default parameters (ref. **RQ1**) and has less variation over different attempts (ref. **RQ2**).

Table 3 shows that PyBullet headless consumes fewer resources overall, while Isaac Sim, is the most memory-intensive simulation as it consumes 10 times more RAM than the next simulator (Webots GUI). This is in line with the current trend of Pybullet being used in machine learning research (ref. **RQ3**). It is worth noting that Coppeliasim uses fewer resources with a GUI than headless. We speculate that this is because it was initially designed as a GUI application, with headless support only added at a later date, thus having received less development focus.

Figure 4 shows the spread of the start time and end time for each simulation (ref. **RQ3**). As mentioned in Section sec:implementation, Isaac Sim has to be started manually, thus the time that takes the simulation to start is not captured in the plot. Ignition takes the most time to load because it requires an additional movement to the start position that it has to do to start the effort controller. Webots finish the earliest with little variation. Combined with its relatively high success rate from Table 2, Webots appears to be ideal for prototyping robotic tasks and for machine learning experiments due to its relatively high success rate from Table 2 and finishing the task and simulation early with low variation. PyBullet, on the other hand, takes the most time and combined with its high failure rate (with the default parameters), it may not be suitable for machine learning as it would take more time to run a single experiment. Similarly, further parameter tuning would be required in order to obtain a stable simulation that succeeds at completing the task.

4.4 Experiment 2

As shown in Table 4, which focuses on **RQ1 & 3**, only Webots throws consistently. Isaac Sim consistently manages to throw the cube but fails to hit the pyramid as the motion behaviour is not consistent. We speculate that this is because we did not tune the simulation parameters and used the default values. Coppeliasim and PyBullet manage to hit the pyramid, but the behaviour is rare as the few times the arm manages to successfully perform the throwing motion, the throw is not always at the same distance nor perfectly aligned. Coppeliasim has a high timing out rate (failure column in Table 4) due to the reasons mentioned in Section 4.3. Finally, for Ignition, the success at hitting the pyramid is zero. We observe that, in most cases, the cube falls in transit, especially when the arm is as far back as possible and starts to move at full speed for
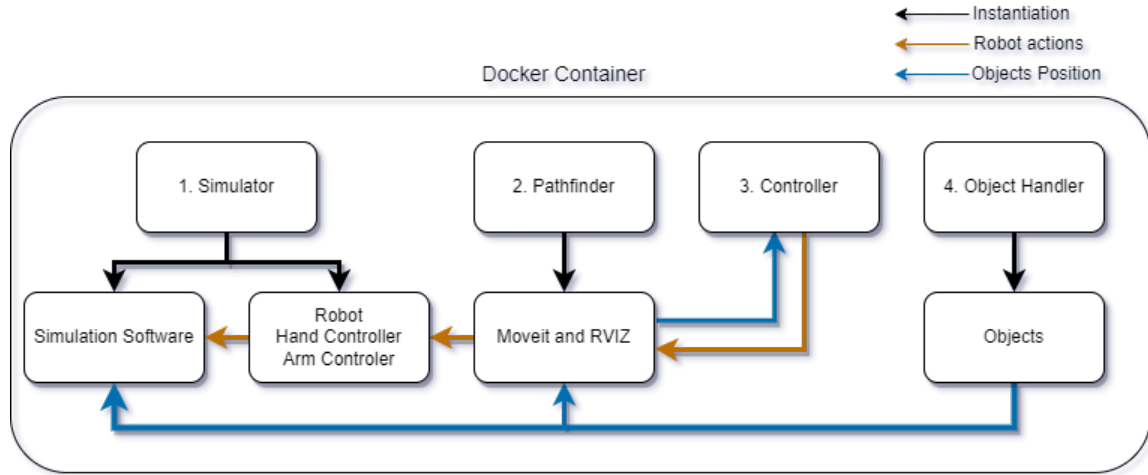
Fig. 3. Implementation Diagram

Table 2. Task 1: Results

Name	Failure (%)	Cubes Placed (%)			Cubes at the end of the task (%)		
		Min	Mean	Max	Min	Mean	Max
Ignition	0	60	91	100	47	82	100
Ignition GUI	0	80	94	100	67	88	100
Isaac Sim GUI	10	47	89	100	20	65	100
Pybullet	0	7	18	47	7	15	47
Pybullet GUI	0	7	11	40	7	11	40
Coppeliasim	30	67	92	100	47	79	100
Coppeliasim GUI	15	53	91	100	47	76	100
Webots	5	53	88	100	47	79	93
Webots GUI	5	73	91	100	40	80	100

Table 3. Task 1: Resources usage

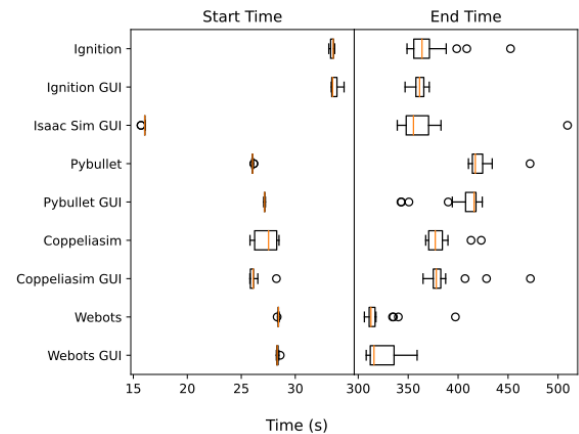| Name	CPU (%)	RAM (MB)
Ignition	202 ± 94	686 ± 313
Ignition GUI	205 ± 51	775 ± 159
Isaac Sim GUI	134 ± 25	10070 ± 1885
Pybullet	117 ± 41	**663 ± 224**
Pybullet GUI	140 ± 28	919 ± 168
Coppeliasim	135 ± 42	860 ± 262
Coppeliasim GUI	**100 ± 43**	850 ± 379
Webots	144 ± 72	1191 ± 550
Webots GUI	162 ± 61	1322 ± 410

Fig. 4. Task 1: Mean task start and end time

the throwing motion. At this point, the robot and the cube have the highest moment of inertia, and if the friction between the cube and the gripper is not enough, the cube falls. We must note that we fix the friction parameter to explore the default capabilities for each simulator. We also notice that there are instances when the robot manages to throw the cube but does not hit the pyramid. This is because the gripper controller had a delay in opening its gripper, changing the thrown cube landing spot.

Table 5 shows similar results to task 1. Coppeliasim uses the lowest amount of CPU while Ignition uses less memory. The CPU usage for all simulations observes less variation. This could be due to the simplicity of the world and the short time of execution. As mentioned in Section 4.3, Coppeliasim still uses fewer resources with a GUI than headless. Figure 5 shows similar start and end times for all simulations, observing lower variations compared to task 1. The reason for this is the relatively short time of execution and the low amount of path planning that can fail and delay the execution. For this scenario, consid-

Table 4. Task 2: Results

Name	Failure (%)	Cubes Moved (%)		
		Min	Mean	Max
Ignition	10	0	0	0
Ignition GUI	0	0	0	0
Isaac Sim GUI	0	0	18	83
Pybullet	0	0	0	0
Pybullet GUI	0	0	0	0
Coppeliasim	32	0	9	50
Coppeliasim GUI	20	0	0	0
Webots	5	0	11	50
Webots GUI	15	0	21	50

Table 5. Task 2: Resources usage

Name	CPU (%)	RAM (MB)
Ignition	152 ± 99	**524 ± 340**
Ignition GUI	137 ± 103	574 ± 371
Isaac Sim GUI	148 ± 48	11123 ± 1865
Pybullet	139 ± 32	730 ± 174
Pybullet GUI	136 ± 31	828 ± 205
Coppeliasim	115 ± 37	742 ± 185
Coppeliasim GUI	**98 ± 28**	838 ± 214
Webots	141 ± 74	1275 ± 261
Webots GUI	134 ± 72	1207 ± 198

ering only the time of execution will not have an impact on the choice for a machine learning approach as the difference between execution is minimal. If resource usage is important, then Coppeliasim should be considered for machine learning tasks. Otherwise, a more successful simulation should be considered such as Webots.

5. CONCLUSIONS & FUTURE WORK

In this paper, we have investigated current robot simulation software performance and their suitability to perform two different robotic manipulation tasks. We have also developed a methodology to systematically benchmark robot simulations under similar parameters, tasks and scenarios. Based on our experimental results, Webots appears to be the more suitable for long-term operations while still succeeding at completing a given task (ref. **RQ1**) and be able to replicate the same simulation conditions across attempts (ref. **RQ2**). Webots would only be suitable for machine learning if the execution time and resources are not a requirement while training machine learning models (ref. **RQ3**). Ignition, while comparable to Webots, is more suited to answer **RQ1 & RQ3**. **RQ2** is only satisfied if the task is slow moving and constant. We must note that Ignition is still in development and some of the challenges we encountered while implementing both tasks and carrying out our experiments may be mitigated in the future. Coppeliasim and PyBullet have less impact in terms of resource usage and are the most suited to
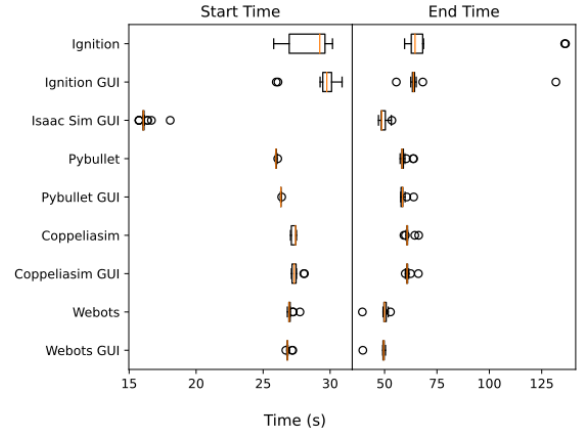

Fig. 5. Task 2: Mean task start and end time

answer **RQ3**. That is, Coppeliasim provides better stability for task success at the cost of timing out more often. Finally, Isaac Sim only satisfies **RQ1**, as the simulated scene was not repeatable across attempts.

From our review and experimental results, we found that current robot simulation software could not be used to develop a digital twin. This is because the simulators considered in this paper cannot maintain a repeatable simulated scene over time. We hypothesise that a continuous feedback mechanism is needed between the simulation and reality similar to [2] in order to maintain an accurate representation of the real environment. While this paper focused on benchmarking robot simulation software, future work consists of optimising each simulator to minimise failure rates and maximise task completion, and benchmark them accordingly. Additionally, the Unreal Engine plugin for ROS2 has recently seen more development and could potentially replace Unity in our original plan. We also aim to specifically benchmark each simulation in a machine learning context, such as in [16] with the view to developing a digital twin that can take advantage of a simulated environment to deploy AI solutions for autonomous robotic systems.

REFERENCES

[1] Angel Ayala, Francisco Cruz, Diego Campos, Rodrigo Rubio, Bruno Fernandes, and Richard Dazeley. A Comparison of Humanoid Robot Simulators: A Quantitative Approach. *arXiv:2008.04627 [cs]*, August 2020. arXiv: 2008.04627.

[2] Peng Chang and Taşkin Padif. Sim2Real2Sim: Bridging the Gap Between Simulation and Real-World in Flexible Object Manipulation. In *2020*

Fourth IEEE International Conference on Robotic Computing (IRC), pages 56–62, November 2020.

[3] Sachin Chitta, Eitan Marder-Eppstein, Wim Meeussen, Vijay Pradeep, Adolfo Rodríguez Tsouroukdissian, Jonathan Bohren, David Coleman, Bence Magyar, Gennaro Raiola, Mathias Lüdtke, and Enrique Fernandez Perdomo. ros_control: A generic and simple control framework for ROS. *Journal of Open Source Software*, 2(20):456, December 2017.

[4] T. Coleman David. Reducing the Barrier to Entry of Complex Robotic Software: a MoveIt! Case Study. 2014. Publisher: Università degli studi di Bergamo.

[5] NVIDIA Corporation. Nvidia isaac sim, 2022.

[6] Erwin Coumans and Yunfei Bai. Pybullet, a python module for physics simulation for games, robotics and machine learning. `http://pybullet.org`, 2016–2021.

[7] Serena Ivaldi, Vincent Padois, and Francesco Nori. Tools for dynamics simulation of robots: a survey based on user feedback. *arXiv:1402.7050 [cs]*, February 2014. arXiv: 1402.7050.

[8] N. Koenig and A. Howard. Design and use paradigms for Gazebo, an open-source multi-robot simulator. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566)*, volume 3, pages 2149–2154 vol.3, September 2004.

[9] James Kramer and Matthias Scheutz. Development environments for autonomous mobile robots: A survey. *Autonomous Robots*, 22(2):101–132, January 2007.

[10] Marian Körber, Johann Lange, Stephan Rediske, Simon Steinmann, and Roland Glück. Comparing Popular Simulation Environments in the Scope of Robotics and Reinforcement Learning. *arXiv:2103.04616 [cs]*, March 2021. arXiv: 2103.04616.

[11] Yuqian Lu, Chao Liu, Kevin I-Kai Wang, Huiyue Huang, and Xun Xu. Digital Twin-driven smart manufacturing: Connotation, reference model, applications and research issues. *Robotics and Computer-Integrated Manufacturing*, 61:101837, February 2020.

[12] M. Santos Pessoa de Melo, J. Gomes da Silva Neto, P. Jorge Lima da Silva, J. M. X. Natario Teixeira, and V. Teichrieb. Analysis and Comparison of Robotics 3D Simulators. In *2019 21st Symposium on Virtual and Augmented Reality (SVR)*, pages 242–251, October 2019.

[13] Lucas Nogueira. Comparative Analysis Between Gazebo and V-REP Robotic Simulators. December 2014.

[14] Anders Orebäck and Henrik I. Christensen. Evaluation of Architectures for Mobile Robotics. *Autonomous Robots*, 14(1):33–49, January 2003.

[15] Lenka Pitonakova, Manuel Giuliani, Anthony Pipe, and Alan Winfield. Feature and Performance Comparison of the V-REP, Gazebo and ARGoS Robot Simulators. In Manuel Giuliani, Tareq Assaf, and Maria Elena Giannaccini, editors, *Towards Autonomous Robotic Systems*, pages 357–368, Cham, 2018. Springer International Publishing.

[16] Nikos Pitsillos, Ameya Pore, Bjorn Sand Jensen, and Gerardo Aragon-Camarasa. Intrinsic Robotic Introspection: Learning Internal States From Neuron Activations. In *2021 IEEE International Conference on Development and Learning (ICDL)*, pages 1–7, Beijing, China, August 2021. IEEE.

[17] Open Robotics. https://ignitionrobotics.org/home, 2022.

[18] Eric Rohmer, Surya P. N. Singh, and Marc Freese. V-rep: A versatile and scalable robot simulation framework. In *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1321–1326, 2013.

[19] Stanford Artificial Intelligence Laboratory et al. Robotic operating system, 2018.

[20] Aaron Staranowicz and Gian Luca Mariottini. A survey and comparison of commercial and open-source robotic simulator software. In *Proceedings of the 4th International Conference on PErvasive Technologies Related to Assistive Environments*, PETRA '11, pages 1–8, New York, NY, USA, May 2011. Association for Computing Machinery.

[21] Pedro Tavares, Joao Silva, Pedro Costa, Germano Veiga, and A. Moreira. Flexible Work Cell Simulator Using Digital Twin Methodology for Highly Complex Systems in Industry 4.0. pages 541–552, November 2018.

[22] Webots. http://www.cyberbotics.com, 2022. Open-source Mobile Robot Simulation Software.