2019-XX-XXXX

Implementation Methodologies for Simulation as a Service (SaaS) to develop ADAS Applications

Author, co-author (Do NOT enter this information. It will be pulled from participant tab in MyTechZone)
Do NOT enter this information. It will be pulled from participant tab in MyTechZone

Abstract

Over the years, the complexity of autonomous vehicle development (and concomitantly the verification and validation) has grown tremendously in terms of component-, subsystem- and system-level interactions between autonomy and the human users. Simulation-based testing holds significant promise in helping to identify both problematic interactions between component-, subsystem-, and system-levels as well as overcoming delays typically introduced by the default full-scale on-road testing. Software in Loop (SiL) simulation is utilized as an intermediate step towards software deployment for autonomous vehicles (AV) to make them reliable. SiL efforts can help reduce the resources required for successful deployment by helping to validate the software for millions of road miles. A key enabler for accelerating SiL processes is the ability to use Simulation as a Service (SaaS) rather than just isolated instances of software. The primary benefits ensue from the in-parallel processing of multiple scenarios or tests using cloud or multiple cores especially to more systematically create "what-if analyses" thereby reducing both development time and cost. Here, we present the workflow of our utilization of SaaS methods (provisioned by Metamoto) and our explorations in this domain using exemplar ADAS scenarios. Additionally, we highlight our ability to perform parametric sweeps over variables such as environmental conditions, actors in the scene, etc. hence performing tests over a variety of scenarios including edge cases. The goal of our efforts is to examine viability and ease-of-use of SaaS (Metamoto in a co-simulation mode) to support Software-in-the-Loop co-development and functional reliability within MATLAB, ROS and Python frameworks.

Introduction

Interest in autonomous vehicles has grown steadily in the mobility industry owing both to its potential to mitigate the on-road fatalities and injuries as well as to maximize return of investment on the mobility infrastructure [1]. However, a stark contrast between autonomous vehicles relative to preliminary versions of vehicle automation or automation in fields such as robotics or aerospace, is the safety and reliability imperative. There are various functionalities such as perception, localization, planning and control (often provided by multiple vendors) that come together in an autonomous vehicle at the component-, subsystem- and system-level to permit it to drive without human intervention partially or fully (SAE L4-L5 operations).

Hence rigorous verification and validation is required to ensure coverage of various scenarios that excite and exercise the numerous interactions that support autonomous operations between these multiple functionalities occurring at the component-, subsystem- and system-levels. Quixotically, rigorous and vigorous pre-deployment testing poses a quandary because developers rely on the results from test-driving autonomous vehicles extensively in real traffic and analyze their performance so that the systems can be evaluated and improved [2]. Obviously, the notion that a vehicle needs to be tested for its self-driving for all possible scenarios before deployment is unreasonable given their great complexity and the diversity and unpredictability of conditions in which they need to operate.

In contrast, studies show that people's acceptance of autonomous vehicles will lower significantly if early deployment is achieved because people's trust will go down even when in total, the number of accidents reduce because humans tend to hold higher standards for automation than for themselves [3]. This leads to further skepticism with this approach since problems such as inclement weather and edge case scenarios pose challenges for autonomous vehicles (as well as for human drivers), and autonomous vehicles might perform worse than human drivers in some cases, particularly at initial stages in the scenario of an early roll out [4]. Companies such as Tesla [5] try to combat this issue through the use of shadow mode simulation which is the process in which the vehicle's computer continues to make mock decisions (that are not applied to vehicle operation) whilst on-road which is then compared by developers with the actual decision made by the driver for analysis and improvement [6]. This indeed serves as an effective alternative except for the fact that it is only advantageous and affordable to large automotive companies with a fleet of vehicles already operating on road which forces many organizations to seek out other forms of verification and validation.

There are various software development cycles each organization follows but most of them are derived from the V Model [7], which over the years has been modified to suit their own requirements and goals. The basic structure involves identification of overall functional requirements of the product that are then separated at a subsystem level into meaningful clusters of operation whose requirements are subsequently derived. Exemplar cases of these so called subsystems could be perception, localization etc. and the components/requirements would be the sensors and the processing algorithms that help extract information and provide context for actions from the incoming data. Once the necessary functionalities on a unit level are developed and tested for each individual subsystem (or sub-subsystem), the integration of those subsystems into one product is commenced post which validation is conducted at a product level. The time frame during this development-validation cycle is a crucial time to deliver a reliable product and there are various methods which are utilized to validate and accelerate this process. Autonomous vehicles, as stressed upon above, need to be tested extensively for their reliability in a more stringent way than conventional automobiles, considering the quantum of electronics, computing and software that goes on to it. This would be an extension of the validation process that is already being carried out on vehicles

considering the driver to be in the loop. Although existing verification methods for partial automation/ADAS might look tantalizingly close to what it would take to achieve complete autonomy, it is not the case since there is no fallback (driver) in case of failure [8]. According to several studies, in order to validate the catastrophic failure rate of a vehicle fleet one must conduct at least a billion hours of testing and possibly even test for repeatability of the vehicle's decisions multiple times to achieve statistical significance [9]. This further assumes that the testing environment is highly representative of real-world deployment, and that circumstances causing mishaps arrive in a random non-deterministic/natural manner. Building a fleet of physical vehicles big enough to run billions of hours in representative test environments without endangering the public is impractical. This is exactly a place where extensive simulation can help the project timeline and development cycle by providing the acceleration in the development process [10].

Simulation cuts the development time and cost of autonomous vehicles and allows for checking the behavior of autonomous vehicles in a huge number of scenarios varied by environments, system configurations and driver characteristics. It does not make proving ground tests obsolete, but it can help minimizing the amount of proving ground tests and limit it to only verify the simulation results and for certification measurements. Field tests will contribute with further validation insights, which derive from unexpected driving situations and retroactive effects under real driving conditions. Simulation plays an essential role in the development and testing of autonomous driving software without which the huge number of tests and verification procedures can not be managed. Because simulation is conducted in a virtual environment, it continues to be faster, less expensive and provides more insights into the underlying physics than physical prototyping. It is a practical way to analyze autonomous vehicles' performance over the billions of miles of test driving that is required as discussed above. As a result, vehicle developers can: a) accelerate time to market, b) reduce costs and c) improve product quality [11].

This virtual testing methodology is done by using simulated sensors, a vehicle dynamics model and simulated scenarios. The function modules are tested by Software in Loop (SiL), Hardware in the loop (HiL) or Vehicle in the Loop (VeHiL) methodologies. The accuracy of the results generated from these testing methodologies depend upon the accuracy of the sensor data being generated and the fidelity of the vehicle model etc.[12]. Over the years the software tools that have been developed have gotten more and more closer to an accurate representation of reality, for example certain software have good vehicle dynamic models, certain software are good at generating perception data for ground truth analysis of the perception systems etc.

There are two vital steps as part of the validation cycle, namely Software in the Loop (SiL) and Hardware in the Loop (HiL). In SiL, the software is tested against an analytical model that represents the system and the subsequent data required such as sensor feed and environment is attained from a co-simulated environment in a synchronous manner. This way we can validate the functioning of our algorithm using the synthetic data generated and how our controller performs in a holistic manner. In HiL we have the analytical/mathematical model that represents the system (typically high fidelity) and the controller running on real time systems connected via appropriate interfaces which is then used to prove the merit of the software through it's real-time reliability and performance metrics. Here the data in terms of sensors and environment is often either obtained using a highly realistic environment simulator running in a soft-real time mode in conjecture with the vehicle model running real time or real world data that has been collected during test drives which are then recreated for virtual testing.

Software development for autonomous vehicles is a multi-step process and it is difficult to test each iteration physically on hardware. Software in Loop (SiL) simulation is utilized to accelerate the software development deployment and in the process making the software reliable. SiL efforts can reduce the development hours, time and money but at the same time validate the software for millions of road miles in multiple scenarios thereby providing the much-needed confidence to deploy it on an actual vehicle. To perform SiL testing, organizations use software simulators to train and test the autonomous vehicles. Simulators serves as a digital twin to emulate real-world scenarios and components that are equipped on the vehicles such as drive-by-wire systems and sensors. In any simulation-based verification and validation framework, the performance (fidelity, accuracy, reliability and robustness) of two classes of entities are critical for success:

- **Environment** – The dynamic environment within which the ego vehicle operates, containing physical entities (pedestrians, roads, buildings, traffic elements etc.) together with representative physical behaviors to create an adequate similitude of reality

- **Ego Vehicle** – The digital twin of the relevant vehicle under study/system-under-test whose physical behaviors under the influence of the sensor-based functional interaction-response algorithms are being simulated and evaluated.

As discussed earlier, simulation provides us with the ability to test across a plethora of variability (simulation parameters) amongst the two relevant factors mentioned above. However, testing scenarios a single instance at a time varied by the simulation parameters in each instance, is still an incredibly inefficient way to test autonomous vehicle software and modern technologies such as cloud computing and parallel processing solve this problem for us with relative ease [13]. This gives rise to the concept of utilizing Simulation as a Service (SaaS) which accelerates the software development process exponentially. This is a method to architect simulation resources in the cloud, so that a plurality of users can access them as and when required [14]. The resources are managed in the cloud by a service provider who oversees the automatic, elastic (ad-hoc), and reliable provisioning of the computing resources demanded by the user(s). These characteristics of SaaS enables efficient management and utilization of resources, hence, achieving cost reduction [15]. The purpose of this study is to introduce the reader to the relevance of SaaS in an autonomous vehicle software development process and drive the point further using an exemplar implementation of the same.

Here on out, the study is divided into the following sections: Section 3: Simulation frameworks for autonomous vehicles, provides a comprehensive view on the requirements of a simulator and also serves as a literature review for the current state of the art in simulation software. It further goes on to elaborate on the importance of SaaS and its applications. Section 4: Motivation, is to educate the reader on the intent behind the study which is presented. A little information about the state of the art research instrument at our disposal is also presented. Section 5: Tools of Study, provides a review of the tools utilized, such as Metamoto, Docker, Robot Operating System (ROS) and MATLAB/Simulink. This information is meant to provide context to the reader regarding the advantages each tool presents and the reasoning behind their utilization. Section 6: Application Programming Interfaces, is presented to provide understanding regarding the workings of the connectivity bridges between the tools utilized and their underlying operational complexity. Section 7: Methodology/Deployment, is the heart of the study, where the working of the SaaS workflow adopted has been presented. The scenarios used to deploy the an ADAS application example using the chosen workflow have also been explained. The results of the simulation and the power of using SaaS have been quantified in Section 8: Results. Section 9: Summary, succinctly paraphrases the study into one section and in the final section, Section 10: Future Work, the possible directions to take this work further have been summarised.

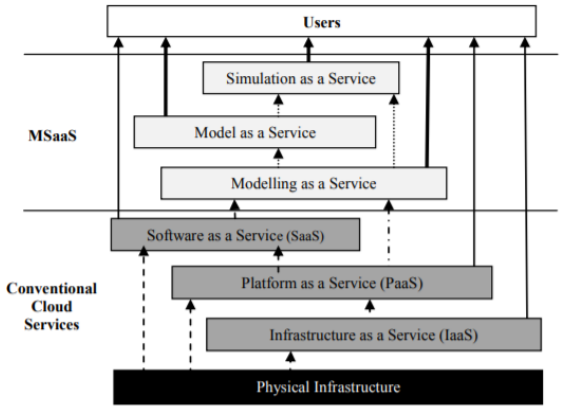

Figure 1: Developed Workflow [14]

Simulation Frameworks for Autonomous Vehicles

In the previous sections we discussed the merit of simulation and its role in the development of autonomous vehicles. The discussion in this section centers around the ideal qualities of a simulation framework/simulator that is desirable to serve appropriately to best bring out the qualities enlisted in the previous sections. This section is also intended to serve as a brief review of the state of the art in terms of the widely used and appreciated simulators in this field. In any simulator, there are certain features that play a very important role in its functionality and versatility:

1. **Time** – Time stands as an essential independent variable while describing the dynamical systems in any simulator. The time in a simulator can either be defined as a continuous model or a discrete model [16]. Generally, the real-world scenarios are defined as a discrete model where the states change at points in time, rather than a continuous model.

2. **Level of detail** – Depending upon the computation ability of a simulator, the level of detail can be categorized as macroscopic, mesoscopic or microscopic. A macroscopic model describes objects and their activities and interactions at a low level of detail. A mesoscopic model generally represents most objects at a high level of detail but describes their activities and interactions at a lower level. Whereas a microscopic model describes most objects and their interactions at a high level of detail. [16]

3. **Probabilistic model** – Probabilistic model plays a major role while reproducing specific conditions or events. The probabilistic model can further break into deterministic model or stochastic model [16]. Simulators' ability to reproduce both these models with ease determine their flexibility to reproduce various complex events.

4. **Design methodology** – The simulators ability is also checked by the ease of designing scenarios. Macroscopic simulators implement models that can be programmed in a progressive way, while microscopic simulators can be designed according to object-oriented standards [16]. In some simulators agent-oriented method of designing can also be applied [17].

Simulator State of the Art

The following literature review talks about some state of the art autonomous vehicle simulation platforms available currently (commercially or open-source) and is used by corporations and institutes around the world for software validation.

Simulation was previously used in the automotive industry, specifically for vehicle dynamics. Some famous examples are: CarMaker [18], CarSim [19], and ADAMS [20]. Autonomous driving requires more than just vehicle dynamics, and factors such as complex environment settings, sensor configurations, and mixed-traffic simulation, must also be considered. The following list is intended to be a compilation of the most prominent simulators currently:

1. **CARCRAFT** – Developed by Waymo [21] for their in house simulation purposes, it was initially used to develop scenarios replicating situations where the driver had to take actions whilst on road testing and to analyze and rectify the software. This process however became multifaceted and grew into developing virtual models of various cities. The strengths are that it a) has a detailed map of the world, along with a physics model of different objects including tires and road and b) gives an option to provide a sensible range of values for the behavior of the other agents on scene using which the software creates a permutation & combination of them, thereby creating different scenarios that replicate difficult situations that tests the car. Its drawbacks lies in the fact that its simulation environment does not cover the perception problem (i.e. it does not use realistic graphics) [22].

2. **CARLA** – CARLA (Car Learning to Act) is an open source simulator for self-driving cars developed in partnership between Intel Labs[23], Toyota Research Institute [24] and the Computer Vision Center, Barcelona [25]. CARLA is a service oriented, high fidelity, realistic graphics environment built on top of Unreal Engine. The simulation environment allows testing from perception to vehicle control. The high points of CARLA are that it has a) a classic integrated pipeline that comprises a vision-based perception module, a rule-based planner, and a maneuver controller b) a deep neural network that maps sensor inputs to driving commands, trained end-to-end via imitation learning and c) provides photo realistic rendering with the help of game engines such as Unity [26] and Unreal Engine [27]. The drawback of this simulator is that it does not utilize HD Maps as a foundation for simulated arenas [28].

3. **Autonovi-Sim by UNC Gamma** – A novel high-fidelity simulation platform for automated driving data generation and driving strategy testing. Its strengths are that it a) enables creation of dynamic traffic scenario, numerous vehicles with unique dynamics b) allows for alteration of environmental conditions, other actor's behaviors (such as cyclists), etc. Its weakness are that it lacks calibration information to duplicate specific sensors and sensor configurations and its present driver models are limited to hierarchical, rule-based driving approaches [29].

4. **Apollo** – It is a software platform to develop automated driving software that includes a simulation framework. Strengths include a) having an HD map & localization (based on GPS, IMU, maps), b) accurate perception sensor data of the surrounding based on Baidu's [30] big data and large data base of real world labeled driving data, c) a planning system containing prediction, behavior, motion logic and d) an open data platform consisting of driving source codes. Despite its exhaustive capabilities, it's weakness lies in the fact that it relies completely on real world data and the simulation aspect is more of a visualizer [31].

5. **TASS-PreScan** - A platform consisting of a GUI-based preprocessor to define scenarios and a run-time environment to execute them. Strengths such as a) interface to link with MATLAB and Simulink to evaluate the developed algorithms, b) flexible interface to other vehicle dynamic model software (e.g. Carsim) and HIL simulators and hardware providers (e.g. dSPACE) whereas weaknesses such as a low computational performance even at low update frequencies[32].

6. **Metamoto** - Metamoto is a simulation software which can be used to generate synthetic data for training purposes as well as to integrate our sensor and control models with pre-existing models in the software. It has several strengths such as a) wide variety of commercially available car models, b) configurable sensor parameters for camera, RADAR, LiDAR with a provision to add

noises and distortion to the signal, c) easy to use GUI and well structure workflow, d) ability to interface with external codes via Docker containers, thereby it can be interfaced with ROS, Python codes, etc. and e) cloud based simulation platform. Its shortcomings lie in the fact that it has a low fidelity vehicle dynamics model [33].

7. **COGNATA** - Cognata simulator specializes in testing and evaluation of self-driving cars through high fidelity, realistic graphics simulation. Their platform uses artificial intelligence, deep learning, and computer vision to provide the only solution capable of validating automated vehicles with unlimited scalability. The strengths of this simulator are a) connectivity with MATLAB/Simulink and ROS for algorithm development, test, and analysis, b) use patented computer vision and deep learning algorithms to create an entire city-simulator including buildings, roads, lane marks, traffic signs, etc., and c) its real-time SDK enables simple integration between the simulator and customers' AV software stacks. However, over reliance on artificial intelligence for new sensor and map creation can lead to unforeseeable errors that might arise due obscurity of the machine learning algorithms[34].

Importance & Uses of SaaS

Automotive simulation is a highly data intensive enterprise. There are a multitude of sensors such as ultrasonic, GPS, IMU, RADAR, cameras and LiDAR, which need to be simulated. The environment in which the ego vehicle is placed in also has many variable parameters including but not limited to traffic and pedestrian flows, traffic lights, road conditions, weather conditions, etc. Creating a test matrix according to our application and then deploying those tests sequentially can take days to complete [35]. SaaS provides the avenue of parallelization and automatic scheduling of the test runs by allotting compute resources on the go (elastic allocation), thereby saving precious development time. It can become an indispensable tool in any domain which is data intensive like crowd simulations, mining robots, warehousing robots, autonomous farming and naturally the automotive industry.

Particularly, for the target application of autonomous vehicles, SaaS brings the following features as an arsenal which helps reduce the time between development and deployment dramatically:

1. **Scalability**: The ability to run millions of tests in a single cycle, ultimately driving billions of virtual test miles, to identify isolated outcomes, performance boundaries and system tolerances.

2. **Parameterization**: The ability to execute simulations across a spectrum of environmental (weather, traffic, road conditions, pedestrians, etc.) and hardware (vehicle properties, sensor placement, latency, sensor settings, etc.) parameters also called a parameter sweep.

3. **Continuous test and integration**: The ability to allow for seamless regression testing, agile workflows, version control, data-tagging and more, every time there is a change in vehicle software, sensors and/or infrastructure.

Motivation

The Open Connected-Automated Vehicle (OpenCAV) [36] project at Clemson University International Center for Automotive Research [37] is developing a novel modular, open-architecture, open-interface, and open-source-software based research instrument comprised of: a) augmented reality (AR) based simulation to facilitate Simulation- based-Design (SBD) of hardware (e.g. sensor-suites) and software (e.g. algorithms) for a range of Connected and Automated Vehicle (CAV) applications; and b) physical real-time hardware-in-the-loop validation on a full-scale vehicle retrofitted with advanced sensing, drive-by-wire, perception, connectivity, computation, and control modules. We have an autonomous driving capable Chrysler Pacifica car equipped with an array of sensors including cameras, RADAR, LiDAR, high accuracy GNSS system with RTK correction , a complete drive by wire package and a high capacity computing platform that makes it an ideal platform to develop and validate algorithms for autonomous driving.

The motivation for this study stems from our own efforts to reduce the development time and risk for the successful implementation of the software to be deployed on to the OpenCAV. This validation of our algorithm is set to be in a virtual environment where we have a digital twin of the car and scenario of our test circuit in Metamoto for our virtual validation process [38]. The intent was to utilize the SaaS capabilities of Metamoto in order to conduct mass simulation in a parallel mode on cloud. The study evolved to be a summary of our findings and intends to motivate researchers and professionals in this field towards using SaaS and serve as a starting point for the reader to pursue the same.

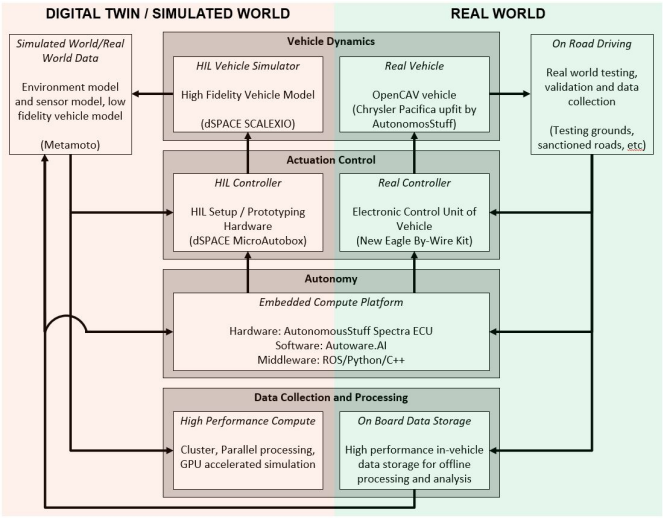

Figure 2: The overall software development pipeline of the OpenCAV project

Figure 3: The OpenCAV vehicle in comparison to its digital twin

Tools of Study

In this section, background is provided regarding the tools used for this study. Although we use particular tools and methodologies for our study, the motivation however is intended for it to serve as an introduction to the importance of simulation and particularly SaaS and its strengths in the autonomous vehicle software cycle. The details of the tools provided here only serve as examples that helps us to shape the direction of this study in a meaningful form.

Metamoto

Metamoto, out of the box comes with an inbuilt way-point controller and obstacle detection avoidance package. Different controllers and custom sensor models can be integrated with the simulation engine via the System Under Test (SUT) methodology.

SUT is an isolated environment that runs independently to interface with the simulation engine having its own operating system and packages that are essential for this purpose and is implemented using Docker images. This SUT establishes the interface between the simulation engine and our code via gRPC (Google Remote Procedural Call). This interface is responsible for subscribing to sensor data of the simulation and publishing the control commands back to the engine.The SUT runs our code synchronously with the simulation engine using the generated API. It is a server-client based setup where the SUT acts as server for control related activities and Simulator acts as server for data, map and reports.

Metamoto provides three different version of SUTs for running controllers:

- Native API through which we can develop controllers in Python and C++.
- ROS based SUT which interfaces with the ROS master, converting data in gRPC format to ROS topics and vice versa.
- Autoware based SUT which is an extension of ROS based SUT where the in built algorithms of Autoware.AI are used to perform various autonomous operations.

Deploying SaaS through Metamoto

To deploy the simulation workflow through the SaaS paradigm, Metamoto was used. It is based on a public cloud architecture, which means that services are provided to the end user over the internet. The front-end of the cloud is a browser-based interface, where one can create the test matrix for a given scenario and deploy the simulation(s). It also consists of Designer and Analyzer applications, which are used to create the actual simulation environment and to analyze and debug the end results, respectively. The back-end components (the actual cloud computing infrastructure and the software which provides the service with elasticity and reliability) is maintained and updated by Metamoto and is a black box to the customer. In these broad measures, it can be said that Metamoto achieves the three defining hall-marks of SaaS, which have been defined in section 3.2: Importance & Uses of SaaS.

The controller code, which is called a "System Under Test" is deployed to the simulation via Linux containers (Docker). These control algorithms can be scripted in languages like Python/C++. Furthermore, Metamoto provides functionalities to use ROS as a middleware, which again opens up a world of possibilities that can enable rapid & robust development of software, which proves ideal for a research setting like ours.

Google Remote Procedural Calls

Metamoto utilizes gRPC [39] – Remote Procedural Call developed by Google using their Google Protobuf architecture. Google-Protobuf [40] – a way to serialize data structures, which is language agnostic that can be used to communicate between different programming languages at the same time. gRPC acts as the link between the simulation engine and the user developed codes in Metamoto.

Docker

Docker [41] is an open platform for developing, shipping, and running applications. It provides the ability to package and run an application in a loosely isolated environment called a container. Containers run directly on the machine kernel and don't require any hypervisor which makes them lightweight. One can run many containers at once on their machine and have near native performance of the host machine [42]. These functionalities of Docker have made it very popular amongst developers as it aids in Continuous Integration and Continuous Development (CI/CD) cycles immensely. In the presented methodology, the SUT is developed and deployed as a container which is then pushed to the Metamoto registry to be used as a controller or sensor by the simulation engine.

Robot Operating System

Robot Operating System (ROS) [43] provides a flexible framework with numerous tools, libraries and conventions that aim to simplify the task of creating complex and robust robotics software. It acts as a middleware to communicate between different sub-systems of the robots to understand and solve the given tasks. At the lowest level, ROS offers a message passing interface that provides inter-process communication. ROS is an industry standard tool, with most commercial sensors already having ROS drivers. This gives an advantage of using the messages encapsulated for these commercial sensors and write control algorithms for deployment. ROS/ROS2 is a vital component of the robotics/AV industry. Tools such as Autoware.AI [44] and Autoware.Auto [45], which are integral in driving autonomy forward are based on ROS. Developing code to run on ROS, enables one to deploy the packages faster and eases the transition of software from simulation to reality.

MATLAB/Simulink

The ease in use of MATLAB/Simulink [46] and its plethora of toolboxes provided across multiple domains has made it a popular tool in engineering (especially research) applications. Recently, MathWorks developed a ROS Toolbox [47] which provides an easy and convenient GUI based approach to create ROS nodes. Their custom message type generation tool has been utilized to access certain data generated by the Metamoto engine, since Metamoto uses custom message types for few topics which are not part of the standard library of message types that come along with ROS. The control algorithm created as a Simulink model can act as a node connected to the ROS Master and publishes the vehicle control commands to it. This data is then utilized by the Simulation engine to propagate the simulation.

Application Programming Interface(API) / Middleware

For quick debugging, Metamoto has provided a utility called System Under Test Connector (SUT Connector) which enables the code to be run on a local system while connected to the simulation engine. Thus, a simulation instance happens between the local host and the engine running on the cloud. This facilitates quick development/validation of algorithms and once the debugging has been completed, this container is pushed to the cloud for running simulations under various scenarios.

Following subsections briefly explain about the functioning of these three bridges and the efforts required from the developer to use them for code deployment:

Native API

A base Docker image is made available to all users of Metamoto, which needs to be pulled from their registry and built. Building this installs all the utilities required to run simulations and also installs the SUT Connector utility. The requisite commands to launch said utility are provided by Metamoto. This establishes the connection between the local host and the cloud. This bridge supports the development of perception & control algorithms on C++ and Python using gRPC. Metamoto provides the code wrapper that let's the data in the gRPC format to be read in Python or C++ using the respective .proto files. This API also contains the code to initialize server on our controller side to pass the vehicle commands and data bus to receive data like sensor data, vehicle state & ground truth information that are being generated by the engine. This data exchange between the controller and engine takes place every simulation time step.

This bridge therefore let's users to create their functions in a high level programming language like Python or C++ and utilize the multitude of libraries and functions that have been developed in these languages. Developers need only to understand the code wrapper to start development of their algorithms.

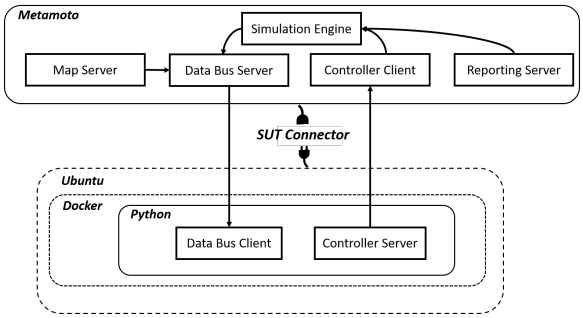

Figure 4: Native API Workflow

ROS API

Following the recent trend in development of software for autonomous systems in ROS, Metamoto has provided a way to interface ROS with the simulation engine. This bridge works in a similar way to the previously mentioned bridge. The API provided by Metamoto is in modular form developed as C++ utilities, each having separate functions such as creating the RPC connection between the code and the engine, initiating a ROS node, converting the data from gRPC to ROS topic format as well as vice versa. The SUT connector needs to be established in the same way as above. Once the connection is set, the executable for ROS API is launched. The simulation data is available in the form of ROS topics and users develop their nodes in the ROS ecosystem.

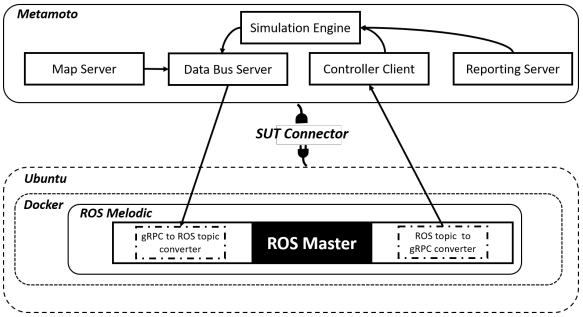

Figure 5: ROS SUT Workflow

Autoware API

This bridge is built upon the ROS API mentioned in section 6.2: ROS API, Metamoto provides the user with launch files that utilizes the packages present in Autoware.AI. The PCD (point cloud data) map of the environment needs to be provided by the user beforehand since the navigation packages in Autoware require them and Metamoto has created the PCD files for many of their simulation environments , which can be used straightway.

Methodology/Deployment

Metamoto requires the controller to be developed in Python or C++, either independently or as a ROS node. Therefore we can't interface MATLAB with Metamoto currently to utilize the capabilities of MATLAB to the fullest. On the other hand both Metamoto and MATLAB have an interface with ROS enivronments. Hence, ROS has been used as a middleware to combine Metamoto and MATLAB to create a seamless data and control flow. In this architecture, Metamoto takes care of the simulation; ROS acts as the bridge between Metamoto and MATLAB, converting data across two different formats and MATLAB executes the controller. This way one can even use the embedded converter of MATLAB such as MATLAB coder [48] to convert the model into a ROS executable C++ code.

The ROS-SUT provided by Metamoto, was used as the base upon which we built our workflow to integrate and utilize the features of MATLAB. The ROS SUT provided the means to convert the sensor data in the form of gRPC to ROS topics which made the simulation data accessible in that forms. Although the data is available in the form of ROS topics, it was present inside the Docker container, whereas MATLAB runs outside the container. Installing MATLAB inside the docker increases the image size and also requires extra steps to enable GUI based interface. In order to overcome these two constraints it was decided to run MATLAB on the local machine itself. The ROS master needs to have the same URI as the local machine so that MATLAB can connect to it. For this purpose the container was initiated with the host's network attributes. Thereby a bridge was established between Metamoto and MATLAB via ROS, bringing together the best of both worlds.

A large chunk of the data being transmitted from the simulation engine can be mapped directly to a ROS message type contained in the standard library but not all of it. The challenge was in converting the gRPC data which did not fit into any existing standard ROS message type into a valid ROS message. Therefore, few custom message types were created based on the need as shown in Table 1.

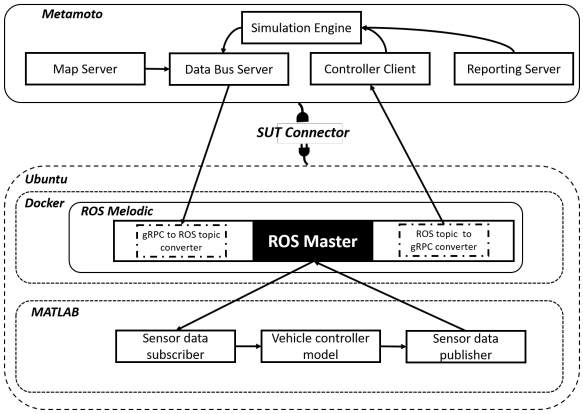

Figure 6: MATLAB-ROS SUT Workflow

In MATLAB the ROS node is initialized subscribing to the data in the main ROS master running in the Docker container. This helps in subscribing sensor data from Metamoto engine via ROS. The custom messages generated for this application were made accessible to MATLAB, using a utility of the ROS Toolbox, so that MATLAB could understand the data in those messages and parse them for further processing.

Information	Direction of Flow	ROS Topic Name	ROS Message Type	Metamoto Message Type
Radar	Metamoto to MATLAB	/radar_processed_array, /radar_processed	ROS custom message	radar.proto, radar_object.proto, enumeration.proto
Acceleration Throttle Steering Brake	MATLAB to Metamoto	ros_sut/ControlCommand	ROS custom message	vehicle_controls.proto, vehicles.proto

Table 1: ROS Metamoto Messages

Once the custom messages are ported, the sensor data was subscribed from the ROS master and used to write the MATLAB control command. The control commands that are published back to the ROS master are then sent to the Metamoto engine for the vehicle to perform the desired maneuvers. The structure of the MATLAB-ROS bridge (which was developed using the ROS Toolbox) that is used in this paper is as follows:

The following Simulink blocks were used to subscribe to the sensor data:

- **Subscribe Block**: The subscriber block is used to access the relevant data coming from the simulator via ROS.
- **Bus Selector Block**: Once the topic is accessed in the Simulink environment, it's contents are parsed using the bus selector block.

The blocks outlined below were used to publish ROS topics:

- **Bus Assignment Block**: Since the message type consists of multiple fields, the bus assignment block lets MATLAB select the commands that needs to be sent back to ROS master.
- **Blank Message Block**: The commands generated from the bus assignment block, are needed to be assigned to the correct message type to populate it correctly to the ROS environment.
- **Publisher Block**: The messages populated from the previous blocks are communicated to the ROS network via the publisher block as ROS topics. The ROS topics can either be custom generated or standard topics.

The trigger for publishing the control command is received along with the sensor data from the subscriber block, ensuring the synchronicity of communication similar to the case of a soft-real time application.

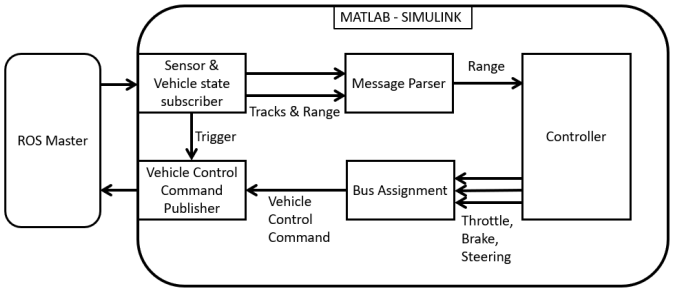

Figure 7: Schematic of the model

Test Scenarios

To test the algorithm, two scenarios were created. In the first scenario, collision avoidance with a stationary vehicle is performed. Whereas, in the second one, the vehicle is swapped out for a pedestrian walking towards the vehicle. The simulation environment in Metamoto is a complete replica of the neighborhood around the ICAR campus, which provides the unique advantage that the control algorithm developed is much more tailored and relevant to the actual testing performed in the real world.

Collision Avoidance with a stationary vehicle

In this scenario, the ego vehicle, which is the Chrysler Pacifica starts moving in a straight line, with a constant velocity. The labeled ground truth data that each sensor generates (RADAR sensor in this case) is relied upon, to detect the type of the obstacle and the distance to it. As soon as the sensor detects a vehicle and the distance between the Chrysler Pacifica and the actor vehicle falls below a certain threshold, a command is sent to the vehicle to execute a panic braking maneuver. Additionally, to preserve passenger comfort along with safety, the danger threshold was expanded and divided into two zones, wherein, in the first zone, gradual brakes are applied and when the vehicle enters the critical zone, hard brakes are applied. In this manner, the vehicle is bought to an early stop and maintain passenger comfort is maintained.

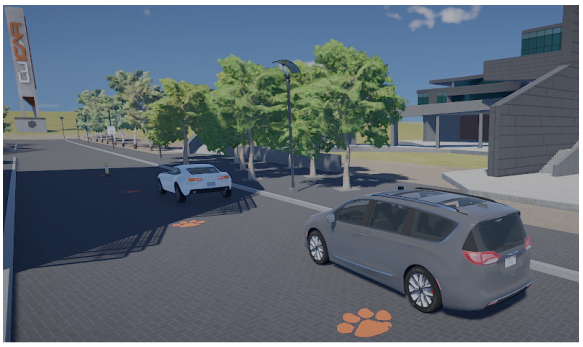

Figure 8: Simulation scenario with actor vehicle

Collision Avoidance with a pedestrian

In this case, the actor vehicle is removed and replaced with a pedestrian in the path of the Chrysler Pacifica vehicle. As mentioned in section 7.1.1: Collision Avoidance with a stationary vehicle, the object type and distance of the detected object is available to the user. Therefore, the condition was modified to look for a pedestrian instead of a vehicle and the behavior of the ego vehicle was left unchanged.

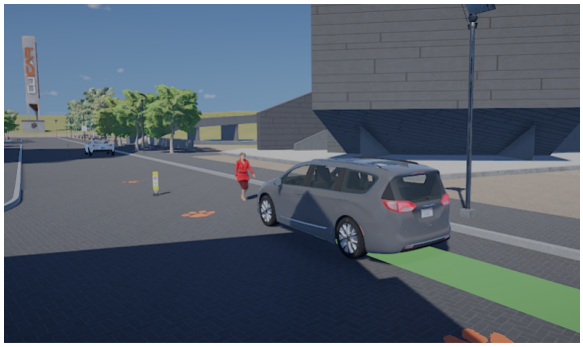

Figure 9: Simulation scenario with pedestrian

Results

As stated earlier in the abstract, the aim of this paper was to explore the domain of SaaS using Metamoto's workflow and also utilize industry standard tools such as ROS and MATLAB/Simulink to observe their effects in aiding the software development process. A rudimentary example of Emergency Collision Avoidance has been utilized to establish and demonstrate the workflow. It needs to be noted that one can deploy many control strategies using the different sensors at hand, to control the ego vehicle which need not be as simplistic as emergency braking or collision avoidance; using the presented workflow. Such controllers are not in the scope of this study and hence, are not discussed. A computationally complex algorithm on the MATLAB side hasn't been implemented since the aim of this study is to showcase the working of the novel workflow that has been devised.

To evaluate the workflow, the following metrics were considered:

- **Ease of programming**: This metric evaluates the ease of the development to deployment cycle of any control algorithm from scratch on to the simulation software.
- **Return on investment**: This indicates the ease of programming gained after learning how to deploy the workflow. This metric is about how much a user should learn from scratch to use the software and the bridge to develop their code. This includes things like Docker, gRPC etc. which are usually outside the scope for control algorithm developers/engineers. Thus, the term of "investment" for this metric relates to the investment of time from the developer,
- **Simulation Synchronicity**: Whenever, we add more and more modalities between the simulation engine and the code, there are chances of the simulation time steps going out of sync between the different layers of the simulation. This indicates whether the it is possible to execute a "blocking", synchronous simulation.
- **Effect on simulation run-time**: This metric assesses whether the addition of a different simulation modality affects the time it takes to perform one simulation run. In other words, the evaluation of latency whilst adding more layers to the workflow.
- **Ease of deployment**: This metric assesses the ease of deployment of the validated code on real-time systems, developed with a particular workflow.

To quantify these attributes, a scoring system was devised, wherein each workflow was given a score from 1 to 3. The higher the score, the better that particular workflow is in the corresponding attribute.

The scoring given to each of these attributes is given in the following table:

	Ease of Programming	**Return on investment**	**Simulation synchronicity**	**Effect on run-time**	**Ease of deployment**
ROS + MATLAB	3	3	3	3	3
ROS	1	1	3	3	2
Native API	2	2	3	3	1

Table 2: Some interesting results from our research.

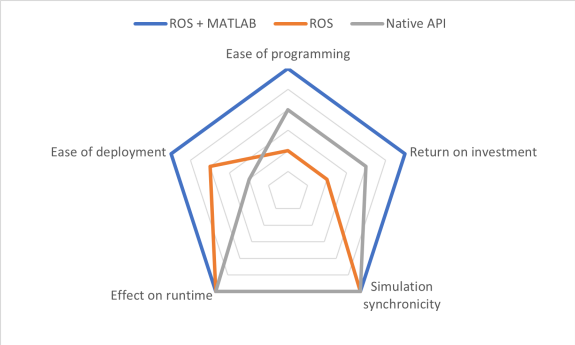

Figure 10: Comparison chart

Figure 10 is a visual representation of the scoring assigned to the performance metrics described.

As can be seen from figure 10, the ROS+MATLAB workflow is the best and has scored full points across the spectrum. The GUI based programming in Simulink, makes it very easy to develop code in MATLAB. The declarative nature of C++ (the language chosen to program in ROS), is an inefficient process in early stages of software prototyping and development and additionally introduces complexities that are only of concern post prototyping. Hence, it gets the least score in this regard. Similarly, the learning curve of MATLAB/Simulink is very shallow and the returns we get in speed of code development is manyfold when compared to the other two. All the given workflows do not impact the synchronicity of the simulation, nor do they impact the runtime of the simulation. Therefore, a score of 3 has been given to all the three in these two attributes. Irrespective of the SiL methodology used, it also has to be readily deployable on to HiL mode, otherwise, valuable time is lost in the overall development cycle. Using the Simulink Real-time (RT) code generator toolchain, one can wrap the control model in Simulink into a C++ code at the click of a button, which is then ready to be deployed to real-time systems for HiL simulation. Such functionality is not present in the other methods, especially using the Native API, which allows to develop a controller and validate it in software only, and thus, it has been scored as such.

Figure 11 shows the RQT Graph of the ROS + MATLAB methodology defined in section 7: Methodology/Deployment. This figure confirms the schematic shown in Figure 6.

Once, the workflow is established and understood, one can proceed to unlock the true power of SaaS, which is to deploy simulations at scale. A parameter sweep of the environment variables of the scenario shown in section 7.1: Test Scenarios.

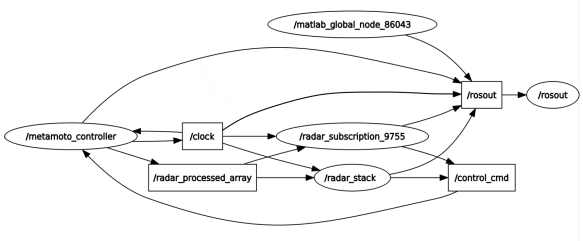

Figure 11: ROS RQT Graph

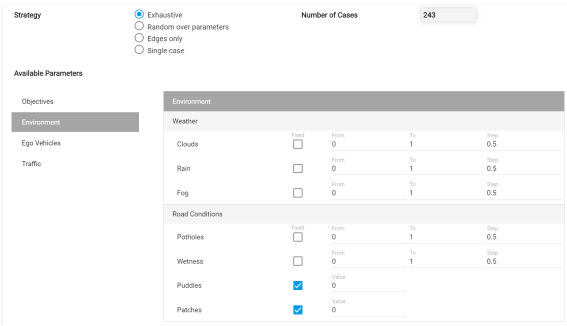

Figure 12: Simulation Test Matrix generation

Figure 12, shows that a parameter sweep to change the intensity of the weather such as fog, rain and clouds was included in the test matrix. Furthermore, road conditions like puddles and patches were also swept through. After creating this test matrix, a total of 243 unique cases were generated. The subsequent figure shows multiple simulations running at the same time.

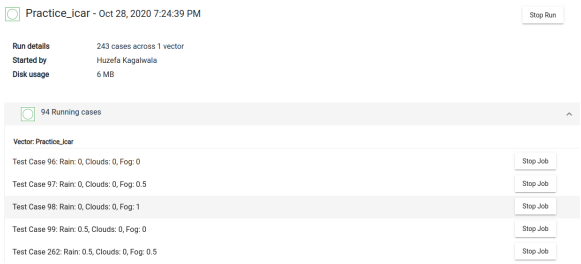

Figure 13: Multiple running simulation

In accordance to the space constraints we have refrained from showing the individual results of the 243 simulations and instead have included the final summary of the simulations has been given below.

Total cases	Passed cases	Failed cases
243	223	20

Table 3: Results of the parallel simulation run.

The total run time in executing these 243 simulations was only 3 hours.

Table 3 shows that 20 times out of 243 the vehicle failed to stop in time and incurred a collision with an actor agent. It was observed that these failures occur in heavy rain and fog scenarios or where the road is very wet. Since, the only sensing modality used in this scenario is RADAR, if at all the sensor is not placed in environments conducive to its reliable operation, the ADAS functionality is likely to fail. And that is what happened here. The RADAR sensor measurements got affected by the noise generated due to rain and fog and hence, reliable range information wasn't relayed to the algorithm issuing the control commands. The wet road conditions also made the braking difficult, which resulted in the crash. It is with such diverse simulations only, that the insights to improve upon the sensor suite and the control logic are afforded to the developer and SaaS in turn helps to deploy these massive tests quickly, efficiently and reliably.

Summary

In this paper, we presented SaaS as a viable solution, based on its ability to provide simulation capabilities at scale and also to accelerate the simulation workflow. We utilized Metatmoto to explore the domain of SaaS due to the rich feature set bought by the software. A basic emergency braking algorithm was developed and a scenario was created wherein obstacles such as a stationary vehicle and pedestrians were used as the triggers to deploy the emergency braking maneuver. The parameters described in section 8: Results were observed and scored and we found that employing MATLAB along with ROS gave us the best value in the development cycle. A batch simulation run was also deployed to show how efficiently and reliably we can deploy a large number of simulations, to get actionable insights on the performance of the algorithm and also on the autonomous architecture itself.

Future Work

In our efforts for further exploration of SaaS, we identified another important aspect of autonomous vehicle technology requiring the resources of SaaS for testing. This modality is the V2X communication aspect of AVs which usually require a large crowd simulation involving several cars. Naturally, this also means the amount of data being exchanged in each network of vehicles is incredibly large and is not suitable for any isolated instances of software requiring cloud based simulation. As soon as we introduce information exchange between vehicles it is also important to research cybersecurity since a vehicle open to the exchange of information is also a hotbed for hacking and cyberattacks. Therefore, in our future work, we will further expand on the other aspects of autonomy simulation such as injecting cyberattacks into the simulation to confuse the controller such as denial of service attacks, man in the middle attacks, etc. This direction is an important aspect that needs to be tested for intensively due to the fact that connected autonomous vehicles are vulnerable to such attacks and they could potentially cause catastrophes. We will also further explore HiL aspects which have been discussed in a shallow form in this study, but will be taking a deeper dive into in the future. Experimental setups such as connecting a steering wheel to the HiL setup in order to test steer-by-wire algorithms are in our purview and we believe the implementation details of the same would be of great value to the community.

References

[1] Department of Transportation, Office of Operations, D. (2007, January). Systems Engineering for Intelligent Transportation Systems. https://ops.fhwa.dot.gov/publications/seitsguide/

[2] Kaleto, H. A., Winkelbauer, D., Havens, C. J. & Smith, M. (2001). Advancements in Testing Methodologies in Response to the FMVSS 201U Requirements for Curtain-Type Side Airbags. SAE Technical Paper Series. doi:10.4271/2001-01-0470

[3] Kalra, N. (2017, May 18). How to Realize Autonomous Vehicle Safety and Mobility Benefits. https://www.rand.org/pubs/testimonies/CT475.html

[4] Gomes, L. (2020, April 02). Hidden Obstacles for Google's Self-Driving Cars.https://www.technologyreview.com/2014/08/28/171520/hidden-obstacles-for-googles-self-driving-cars/

[5] Tesla, Electric Cars, Solar; Clean Energy. https://www.tesla.com/

[6] Templeton, B. (2019, April 30). Tesla's "Shadow" Testing Offers A Useful Advantage On The Biggest Problem In Robocars. https://www.forbes.com/sites/bradtempleton/2019/04/29/teslas-shadow-testing-offers-a-useful-advantage-on-the-biggest-problem-in-robocars/

[7] Department of Transportation, Office of Operations, D. (2007, January). Systems Engineering for Intelligent Transportation Systems. https://ops.fhwa.dot.gov/publications/seitsguide/

[8] Koopman, P. & Wagner, M. (2016). Challenges in Autonomous Vehicle Testing and Validation. SAE International Journal of Transportation Safety, 4(1), 15-24. doi:10.4271/2016-01-0128.

[9] Butler, R. & Finelli, G. (1993). The infeasibility of quantifying the reliability of life-critical real-time software. IEEE Transactions on Software Engineering, 19(1), 3-12. doi:10.1109/32.210303

[10] Huang, W., Wang, K., Lv, Y. & Zhu., F. (2016). Autonomous vehicles testing methods review. 2016 IEEE 19th International Conference on Intelligent Transportation Systems (ITSC). doi:10.1109/itsc.2016.7795548.

[11] Schöner, H. (2018). Simulation in development and testing of autonomous vehicles. Proceedings 18. Internationales Stuttgarter Symposium, 1083-1095. doi:10.1007/978-3-658-21194-3-82.

[12] Wang, F., Wang, X., Li, L., Mirchandani & Wang, Z. Digital and construction of a digital vehicle proving ground. IEEE IV2003 Intelligent Vehicles Symposium. Proceedings (Cat. No.03TH8683). doi:10.1109/ivs.2003.1212968.

[13] Prochazka, D. & Hodicky, J. (2017). Modelling and simulation as a service and concept development and experimentation. 2017 International Conference on Military Technologies (ICMT). doi:10.1109/miltechs.2017.7988851.

[14] Sixuan Wang, G. (1970, January 01). A simulation as a service methodology with application for crowd modeling, simulation and visualization - Sixuan Wang, Gabriel Wainer, 2015. Retrieved October 29, 2020, from https://journals.sagepub.com/doi/10.1177/0037549714562994

[15] Cayirci, E. (2013). Modeling and simulation as a cloud service: A survey. 2013 Winter Simulations Conference (WSC). doi:10.1109/wsc.2013.6721436

[16] Nair, V. G. A Study of driving simulation platforms for automated vehicles.

[17] Fadaie, J. G. The State of Modeling, Simulation, and Data Utilization within Industry - An Autonomous Vehicles Perspective.

[18] GmbH, I. (2020, August 25). CarMaker. https://ipg-automotive.com/products-services/simulation-software/carmaker/

[19] Mechanical Simulation. https://www.carsim.com/

[20] MSC Software Corporation. https://www.mscsoftware.com/product/adams-car

[21] Waymo. https://waymo.com/

[22] Story by Alexis C. Madrigal. (2018, December 03). Waymo Built a Secret World for Self-Driving Cars. https://www.theatlantic.com/technology/archive/2017/08/inside-waymos-secret-testing-and-simulation-facilities/537648/

[23] Intel Labs - Computer Science Research and Collaboration. https://www.intel.com/content/www/us/en/research/overview.html

[24] TRI,Innovating the Future of Mobility. https://www.tri.global/

[25] Barcelona Computer Vision Center.http://www.cvc.uab.es/

[26] Unity. https://unity.com/

[27] Unreal Engine https://www.unrealengine.com/en-US/

[28] CARLA. https://carla.org/

[29] AutoNavi. https://www.autonavi.com/

[30] Baidu ,https://www.baidu.com/

[31] Apollo, https://apollo.auto/

[32] PreScan. https://tass.plm.automation.siemens.com/prescan

[33] Metamoto. https://www.metamoto.com/

[34] Cognata: Autonomous and ADAS Vehicles Simulation Software. https://www.cognata.com/

[35] Metamoto - How Legacy Automotive Simulation Tools Differ From Metamoto's Scalable Cloud-Based Simulation. https://medium.com/@metamoto/how-legacy-automotive-simulation-tools-differ-from-metamotos-scalable-cloud-based-simulation-c9a1e07b9c2c

[36] OpenCAV.- Automation Robotics & Manufacturing Lab, Clemson University International Center for Automotive Research https://sites.google.com/view/opencav/

[37] Clemson University International Center for Automotive Research. https://cuicar.com/

[38] Clemson - Metamoto Presentation at Nvidia GTC 2020. https://www.youtube.com/watch?v=uACVoNThBdk&ab_channel=NVIDIA

[39] gRPC https://grpc.io/

[40] Google-Protobuf https://developers.google.com/protocol-buffers

[41] Docker https://www.docker.com/

[42] Xavier, M. G., Neves, M. V., Rossi, F. D., Ferreto, T. C., Lange, T. & Rose, C. A. (2013). Performance Evaluation of Container-Based Virtualization for High Performance Computing Environments. 2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing. doi:10.1109/pdp.2013.41.

[43] ROS http://wiki.ros.org/ROS/Introduction

[44] Autoware.AI. https://www.autoware.ai/

[45] Autoware.Auto. https://www.autoware.auto/

[46] MATLAB/Simulink https://www.mathworks.com/products/matlab.html

[47] ROS Toolbox - MATLAB https://www.mathworks.com/help/ros/getting-started-with-ros-toolbox.html

[48] MATLAB Coder - MATLAB code to C/C++ code generator https://www.mathworks.com/help/coder/