Toward Reproducible Version-Controlled Perception Platforms: Embracing Simplicity in Autonomous Vehicle Dataset Acquisition

Ignacio Vizzo Benedikt Mersch Lucas Nunes
Louis Wiesmann Tiziano Guadagnino Cyrill Stachniss

***Abstract*— Building datasets for autonomous vehicles has become an essential element of robotics research. Numerous datasets were published, pushing the state-of-the-art forward. This article investigates the problem of building reliable perception platforms that can accurately capture and process sensor data, ensuring the integrity and quality of the datasets generated. We propose using version control systems to enhance dataset acquisition's efficiency, reliability, and scalability. We present a method that can launch the system and record data while logging the exact state of the system simultaneously, making the setup and the data generated with it more reliable and reproducible. The main contribution of this paper is a systematic method that applies to existing and operating perception platforms used to collect data. Our framework is based solely on standard tools and is independent of the chosen sensor suite or host system. Implementing our method for existing perception platforms is possible, and to facilitate this, we open-source all the software used to operate our perception platform at: `https://www.github.com/ipb-car`.**

I. Introduction

Building datasets for autonomous vehicles [1], [2], [3], [7], [14], [24], [30], [34] has pushed research forward and allows easy comparison between approaches. Properly recording such a dataset, especially if multiple sensors are used, is a significant effort in robotics. While various datasets have been developed to aid the development of algorithms, numerous challenges remain when building robust and scalable perception platforms. Software in research labs is rarely in the final state [5], [6], and researchers continuously test, use, discard, and deploy new developments, configurations, and setups. The tools used to record datasets often lack reproducibility due to scattered software, hardware descriptions, and documentation. This makes it difficult to determine the system's state for a specific recording. In addition, system documentation is often maintained separately and not as part of the main software, leading to potential errors in the recording phase due to outdated instructions.

One solution to these challenges is what we call *version-controlled perception platforms*. These platforms allow the creation of datasets by exploiting version control beyond the software, enabling tracking changes in the configuration, setup, sensor drivers, documentation, hardware description, and operating system configuration. This facilitates collaboration among team members and ensures the use of up-to-date software for the perception platform.

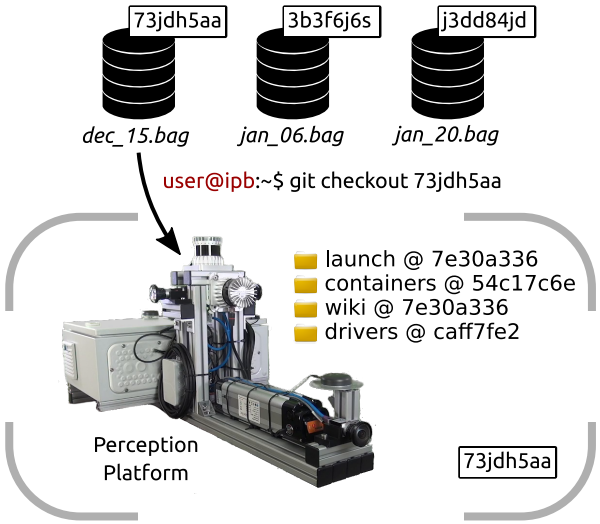

Fig. 1: After recording data at different points in time using different driver versions or configurations, our proposed version-controlled perception platform allows to checkout at a recording's unique hash describing the state of the complete system at that moment.

This system or best-practice paper focuses on improving perception platforms' reliability in robotics research. We propose to extend the usage of version-control systems, like Git, to implement version-controlled perception platforms. As illustrated in Fig. 1, we integrate the complete system description into a version-controlled environment, in which each recording generated with the platform is associated with a Git hash that captures the complete state of the system at the time of recording. This approach facilitates the creation of experimental branches for development and testing. We also explore the concept of a software-configurable sensor suite that provides flexibility and adaptability without requiring hardware modification. Furthermore, our methodology explores the use of a time-synchronization scheme that enables the synchronization of different sensors solely through network cables, eliminating the need for additional hardware triggering systems, enhancing the system's simplicity, and avoiding the complexities and costs associated with external triggering mechanisms.

All authors are with the University of Bonn, Germany. Cyrill Stachniss is additionally with the Department of Engineering Science at the University of Oxford, UK, and with the Lamarr Institute for Machine Learning and Artificial Intelligence, Germany.

This work has partially been funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy, EXC-2070 – 390732324 – PhenoRob, by the European Union's Horizon 2020 research and innovation programme under grant agreement No 101017008 (Harmony), and by the European Union's Horizon Europe research and innovation programme under grant agreement No 101070405 (DigiForest).

To implement a version-controlled perception platform, we utilize standard tools available in most modern operating systems, namely Git and Docker. We propose a method to install and launch the system on a new machine easily, reliably record data, and simultaneously log the exact state of the system. Furthermore, we introduce a sensor synchronization setup that minimizes hardware intervention using a single network cable, such as Ethernet or Thunderbolt. This unique network interface is sufficient to capture the complete sensor stream using multiple cameras and 3D LiDARs.

The main contribution of this paper is a systematic and practical method that will improve the existing perception platforms used for data collection in most research labs. Our method implements version control across the entire system and uses Docker containers. By utilizing specific versions and referencing the commit hash in the logs, the configurations, software, and recording sessions of the datasets can be retrieved precisely. Our method not only simplifies the setup process but also increases the accessibility of the platform for users who may not have specialized equipment or knowledge. Although our focus has been on perception platforms for autonomous vehicles, the principles and techniques we propose can be adapted and applied to other robotic systems. All components needed to reproduce our system or build an own setup are open-source.

II. Related Work

Validation and testing algorithms' performance is a crucial step in robotics research [8], [28], [29], but it also ensures safety when deploying robotic systems in the real world. This requires the availability of large, diverse, and reliable datasets. Although various datasets are available in the context of autonomous vehicles, there is little detailed work on *how* to build a good platform for data collection.

In the robotics research community, some integration frameworks have been developed to facilitate the integration of components of the robotic system [20], [22], [23]. Within these frameworks, the Robotic Operating System (ROS) [23] became popular within the robotic community. For years, ROS has been the primary development platform for robotic systems, being a hub of algorithms in robotics, providing tools and architecture for communication between robot sensors. With the rapid expansion of robotics in enterprise environments, ROS 2 [18] was proposed as a more reliable, consistent, and robust platform. Although ROS 2 has been proposed as a more robust replacement for ROS 1, the robotics community still widely uses ROS 1 since years of research contributions integrated with this platform are not easily portable to ROS 2. Most robots today are a mixture of existing ROS modules, self-developed software, research code under development, and proprietary drivers. Similar observations hold for other middleware systems as well.

One of the first benchmarks for autonomous driving was the KITTI dataset [14], [15]. Their data collection platform comprised two colors, two grayscale cameras, a Velodyne LiDAR with 64 beams, and a GNSS/IMU localization unit with real-time kinematic correction (RTK). To synchronize the LiDAR and camera frames, the LiDAR is used as a reference to trigger the cameras after each rotation. For each LiDAR-triggered time, the closest GNSS/IMU measurement is taken since the localization system runs at a higher frequency than the other sensors.

Following KITTI, other datasets were proposed in the context of autonomous driving [2], [17], [19], [24], [30], [34]. Some datasets use a unimodal sensor setup that collects only image data [9], [13], [21], [25]. One such dataset is Cityscapes [9] which first focused on semantic segmentation in an urban context and contained only image data from a stereo camera. Later, the dataset was extended to 3D object detection using depth information from stereo cameras [13]. Other works [10], [25] took advantage of advances in simulation environments to generate synthetic datasets. More recently, many companies have pushed the field of publishing large-scale datasets from urban scenarios [4], [7], [26], [33]. These works and their proposed datasets helped to push the field of autonomous driving. However, since those works focused on the collected datasets, their data collection platform was often not discussed in detail, which kept the challenge of developing a system able to collect synchronized data continuously. To the best of our knowledge, solely the ApolloScape dataset [16] released an open-source software architecture for data collection. However, unlike our work, many specific hardware requirements must be met to collect the data properly. This paper mainly discusses our approach to managing our data collection platform, developing a portable, reliable, and consistent system capable of recording synchronized data on different hardware sets and operating systems in a *plug-and-play* fashion.

Moreover, in robotics research, an equally challenging aspect is the ability to reproduce software code, which is crucial for validating research findings and promoting collaboration within the community. Several studies have focused on this aspect [5], [6], [12], [27]. These studies have highlighted researchers' difficulties in effectively reusing code and proposed solutions to overcome these challenges. Additionally, working with ROS in Docker can be challenging, especially when network configurations must be performed [31], [32]. To cope with networking problems, Wendt et al. [31] proposed a method for containerized ROS deployments in distributed multi-host environments, including proxying ROS communications, enabling efficient deployment and communication in distributed settings. In contrast to the challenges mentioned above, our methodology offers a streamlined approach that minimizes the difficulties. By implementing version control and Docker containers, our method simplifies the setup process and enhances the accessibility of the perception platform. Additionally, we provide a straightforward and configurable networking setup by specifying the networking setup in a simple configuration file. This eliminates the need to manually configure complex network settings, making the deployment and communication of the perception platform more efficient and user-friendly and transforming the platform into a plug-and-play solution, as no manual intervention on the host machine is required.

III. The Meta-Workspace Concept

An essential component of our system is what we call the meta-workspace. It is a Git repository containing only two files sufficient to describe, build, and run the entire system: *.gitmodules* and *launch.yaml*. Our meta-workspace is similar to what is commonly known as the *src* directory in Catkin or Colcon workspaces. This implies that cloning it into an empty directory on the host system is sufficient to download all components, initiate the build process, and launch the system.

The file *launch.yaml* is a docker-compose description file, which specifies the network configuration of the host machine, the Docker services to run, the IP addresses of all sensors present in the network, and the network interface used to stream data. This description file also indicates where to mount the meta-workspace directory in the container that will run the setup. Encoding the network configuration in the *launch.yaml* file and the container specifications are crucial components that enable development on a host machine completely decoupled from the operating system used to run the sensor suite.

We utilize Git Submodules, a standard tool included in all Git distributions, to list all the necessary components required for the proper functioning of our system. Maintaining a separate Git repository for each component, enables us to have different configurations for continuous integration (CI), issue trackers, and potentially separate Git servers. On the contrary, using a monorepo to store all the software components is also a common technique. However, it does have certain drawbacks: one major drawback is the loss of individual version control for modules. This makes updating specific modules with newly released code from sensor vendors difficult. Additionally, a monorepo environment lacks the transparency to change individual module configurations. These limitations can hinder the flexibility and efficiency of managing and updating modules within the perception platform, especially when larger systems are built, and modern sensors are used. Therefore, we opted against a monorepo for our setup.

A significant benefit of our system is that the data recording process is simplified by decoupling the host machine from the software used to operate the perception platform. This means that, unlike other robotics platforms with more specific requirements, our platform only needs a computer running any GNU/Linux distribution and a network interface to connect to the setup. The meta-workspace handles all other dependencies and configurations. This approach not only simplifies the setup process but also increases the accessibility of the platform for users who may not have specialized equipment or knowledge.

Employing Git for all components in our perception platform brings the inherent benefits of version control, such as checking out, tracing back changes (blaming), branching, and others. This ensures the reliability of the datasets created with the system since the perception platform state at the time of dataset recording is preserved through the associated Git hash. As a result, users can have confidence in the datasets knowing that all components' states can be accurately recovered. Furthermore, reproducing the platform state is simple by checking the hash associated with a specific recording, as illustrated in Fig. 1.

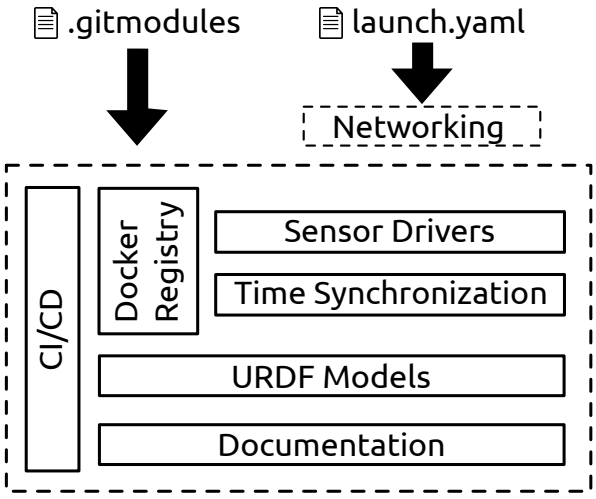

Fig. 2: Main components of a version-controlled perception platform. Our meta-workspace is defined by two files, *.gitmodules* and *launch.yaml*. As shown in this illustration, the *launch.yaml* file specifies all the necessary network configurations to operate the system. The *.gitmodules* file, instead, specifies all the necessary components to build and run the system.

IV. Building Blocks of a Version-Controlled Perception Platform

In the following sections, we outline the essential components required to realize a version-controlled perception platform using standard tools, regardless of the chosen sensor suite or the host machine used for recording. Our approach is not tied to a specific platform, but we will give a practical example of how to implement the concepts discussed here in a real-world case study. Unless explicitly stated otherwise, each subsection refers to one or more Git repositories that are part of our meta-workspace. An illustration of the meta-workspace is given in Fig. 2.

A. Docker Registry

We containerize all software components required to operate the sensor suite to decouple the host system from the perception platform. Furthermore, we maintain a private Docker registry on our Git server that provides pre-built Docker images for all the containers required to run our system. The docker files used to build these containers are also included in our meta-workspace, allowing us to have a local copy of the files used to generate these containers in our registry. Additionally, this approach enables us to modify the containers locally, as necessary, for example, in the early stages of development or for debugging purposes.

B. Sensor Drivers

Our perception platform uses the Robot Operating System ROS to operate the sensors and log data. However, we do not

enforce a specific ROS version, and other robotic frameworks can also be used. We containerize all required sensor drivers, including LiDAR, camera, and custom nodes. Adding a new sensor to our setup is as simple as adding the sensor-specific driver to our Git submodules list. If a specific SDK or system package is required to operate the sensor, then the Dockerfile can be updated without affecting the development on the host machine.

An important benefit of this design choice is that the sensor suite can be operated independently with different ROS versions, such as ROS 1 or ROS 2, simply by changing the meta-workspace definition. A possible realization could be maintaining separate Git branches for each ROS version, with modified Git submodules and Dockerfiles. This enables switching between ROS versions without changing the host machine or its operating system, hardware, or sensor configuration.

C. Time Synchronization

Ensuring time-synchronized sensor data is a crucial requirement for modern perception platforms. It is important to distinguish between two concepts: *sensor time synchronization* and *synchronized sensor triggering*, which are often misunderstood and (wrongly) used interchangeably. We compare both setups in Fig. 3.

Sensor time synchronization refers to the use of a shared master clock by all sensors in a running system that allows them to report timestamps relative to the same clock, and often, it is possible to also trigger sensors via software, for example, to take images at a specific point in time. However, it does *not* necessarily mean that the sensors will capture data frames in perfect sync as a hardware trigger would do.

On the other hand, synchronized sensor triggering refers to the use of a particular signal, often generated by a hardware trigger or another sensor GPIO output, to capture the sensor data frames in a coordinated fashion. This approach is often used for stereo cameras or overlapping multi-camera systems. For example, in the KITTI odometry dataset [14], the Velodyne LiDAR triggers the cameras at the end of each laser sweep. Although this approach can ensure that data frames from different sensors can be associated, it has some limitations. For example, it enforces additional hardware modifications, such as wiring the GPIO pins from one sensor to another. This makes the platform setup challenging to reproduce in scenarios where hardware engineers are unavailable. It also limits the flexibility of triggering by the LIDAR hardware. Furthermore, some sensors, such as a 3D LiDAR, cannot be actively triggered. They can only generate a trigger output in specific configurations. Additionally, triggering does not scale well as the number of sensors in the suite grows.

In contrast, we advocate building a sensor suite entirely configurable by software and not requiring additional hardware intervention. To achieve this, we use the IEEE 1588 Precise Time Protocol (PTP) [11], today a widely adopted standard for clock synchronization in networked measurement and control systems. PTP provides accurate clock

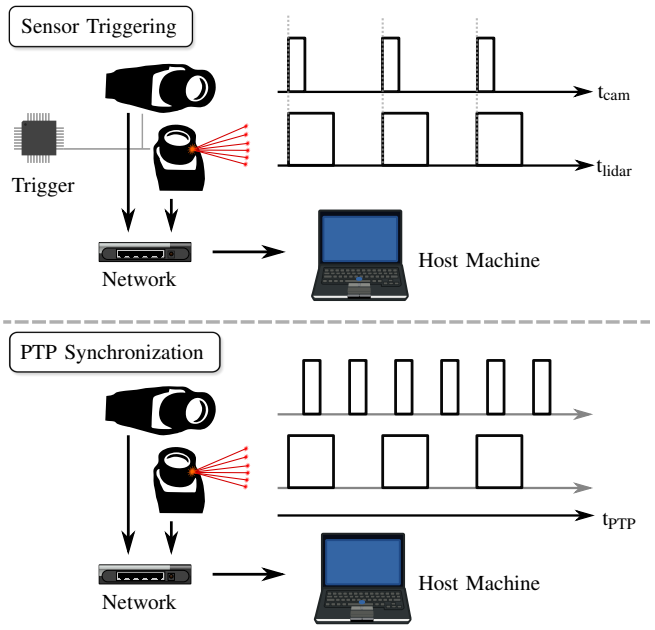

Fig. 3: Difference between synchronized sensor triggering *(top)* and time synchronization *(bottom)*. Traditional sensor hardware triggering relies on external hardware sources to capture data from sensors within the network. Our approach to time synchronization using PTP eliminates the need for additional hardware components. Instead, it synchronizes the internal clocks of the sensors themselves, ensuring precise and coordinated data acquisition without reliance on external triggers.

synchronization across multiple sensors and other devices, which is critical to obtaining reliable measurements. Note that the software-based PTP protocol can be established through standard plug-and-play network connectors, but it enforces the requirement that the sensor must come with PTP support from the vendor. Although our time synchronization setup drastically simplifies the hardware connections, stereo cameras or multi-sensor arrays with overlapping fields of views used for point triangulations may still require a hardware trigger if exactly synchronized exposure is essential.

PTP Grandmaster Clock. Our approach to network-synchronized sensors requires a precise Grandmaster node in the PTP network [11]. In some scenarios, any sensor or device on the network can act as this master clock. Nevertheless, we design and control the Grandmaster clock as part of the sensor suite to achieve reliability and absolute timestamps for each sensor. We propose using an onboard computing system like a Raspberry Pi or Intel NUC with additional GNSS hardware.

Although not strictly required for realizing a version-controlled system, we utilize a GPS-disciplined host to develop our synchronization setup. The GPS satellite time initially adjusts the Grandmaster's internal clock and is subsequently synchronized with a pulse-per-second (PPS) signal from the GPS sensor once per second. Once the Grandmaster clock is synchronized with the GPS satellites, all PTP-enabled sensors in the network will synchronize to

this Grandmaster clock, eliminating the need for user intervention or hardware configuration. Moreover, each sensor's timestamps will contain absolute timestamp information, as the PTP Grandmaster clock receives its time from the GPS satellites. Therefore, all datasets recorded with this synchronization setup will store the exact time a particular data frame was recorded.

This choice might introduce an additional vulnerability that could disrupt the functioning of the perception platform. Once the Grandmaster clock machine is set, it becomes crucial to ensure that any interventions or modifications to the system do not compromise its internal configuration. To address this weak point, we have developed a custom OS distribution for the Grandmaster clock computer, incorporating an additional monitoring layer[1]. Before launching our system, we conduct a sanity check to verify that the configuration of the Grandmaster clock computer aligns precisely with the specifications outlined in the current working environment. This enables us to identify any potential mismatches or discrepancies that may arise promptly.

D. Networking

Our methodology introduces a minimal requirement for the sensors, specifically an Ethernet connection with support for PTP, which is now commonly available. In contrast to the traditional approach in the robotics community, our system simplifies network configurations by specifying all essential details, such as IP addresses, within the *launch.yaml* file. Consequently, no modifications to the host machine are needed, resulting in a streamlined and portable setup. Another advantage is the ease of transferring and running our perception platform on another host machine without any changes or adjustments. Importantly, our setup does not override any local configuration on the host machine, making it a non-invasive approach.

Depending on the sensor configuration, it might also be necessary to ensure the network is 10 GbE instead of more traditional 1 GbE setups for fast and collision-free routing, an option we chose to handle multiple cameras and 3D LiDARs in our setup.

E. Documentation

Traditionally, the documentation for perception platforms is stored in a separate repository or documentation system, independent of the system's main codebase. While this separation may have benefits, we have experienced that including the documentation repository in the main codebase of our system has several advantages. Firstly, it ensures that the documentation is always tightly coupled with the codebase. Second, it allows developers to search for source code and documentation consistently. Third, it allows one to check the documentation even when no internet connection is available. Lastly, it provides a comprehensive system overview by bringing together all relevant information in one place. We strongly recommend this approach, especially for university research labs with changing personnel.

[1]All the details of our custom OS can be found at `https://www.github.com/ipb-car/raspbian`

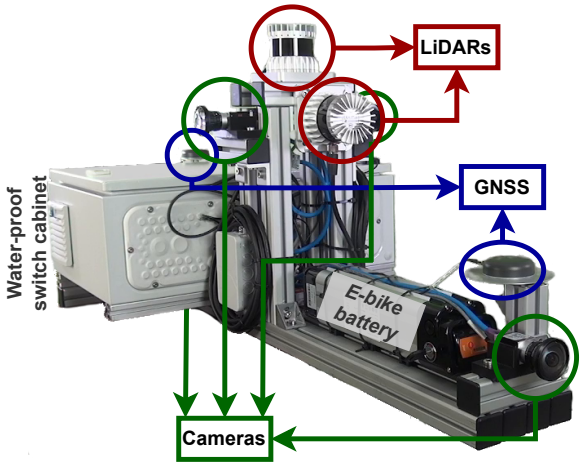

Fig. 4: The ipb-car platform: a practical example of a version-controlled approach.

F. URDF Models

In our meta-workspace, we also incorporate the calibration of the sensors utilized in our system. Our system's URDF (Unified Robot Description Format) models are included in our meta-workspace, which completes the description of the current working setup. By storing the calibration in the URDF Git submodule, any changes to the calibration parameters can be tracked and visually inspected when launching the system.

G. CI/CD

Continuous Integration/Continuous Deployment (CI/CD) is a software development practice that relies on a build server or similar infrastructure to automatically integrate code changes, build software artifacts, run tests, and deploy applications in a streamlined and continuous manner. The stability of our proposed system is highly dependent on our CI/CD infrastructure. All the sensor drivers and Docker images are built daily or when a request change is carried over on the repositories. Additionally, to keep our meta-workspace up to date, we use our CI/CD to pull new changes from the stable branches into the meta-workspace definition. In addition to keeping the meta-workspace updated, the CI/CD is in charge of building all the ROS nodes, docs, URDF models, and more on each commit of each Git submodule. We use the same containers to run the perception platform and the CI/CD infrastructure to ensure we can build the components locally. Furthermore, the CI/CD performs nightly builds of all the system components and deploys the Docker images our setup uses daily. This ensures that the Docker containers are up-to-date. Lastly, our CI/CD infrastructure builds our custom operating system for the Grandmaster clock computer, meaning that at any point in time, we can use the image generated by the CI to flash the OS and start from a new and known state.

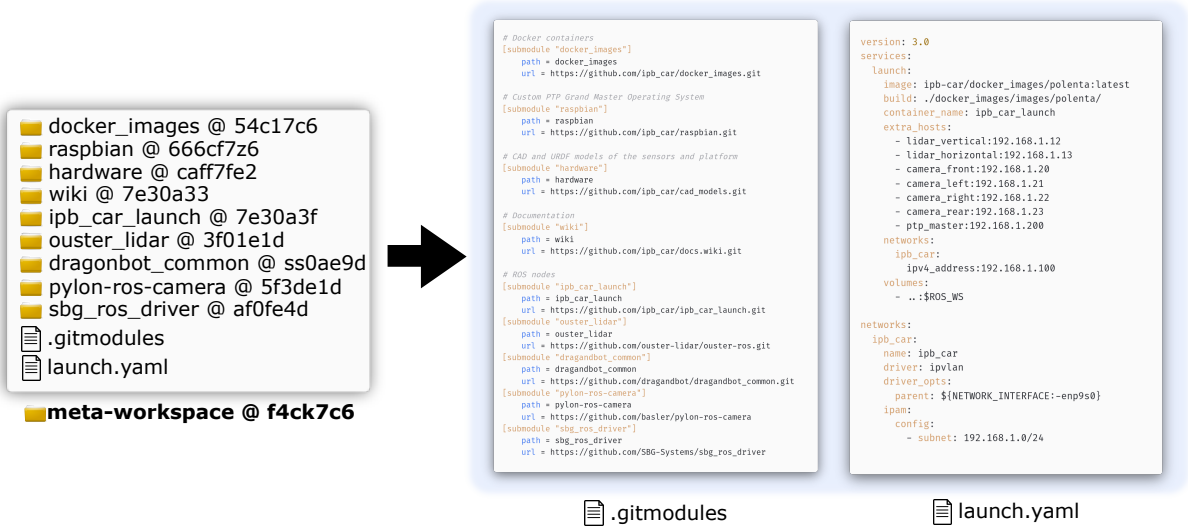

Fig. 5: An example of our ipb-car meta-workspace. All the necessary components are described within the *.gitmodules* file. Additionally, sensor IP addresses and additional network configurations are defined in the *launch.yaml* file on the right of the picture.

V. Building a Version-Controlled Platform

In this section, we specify how to apply the concepts described in this paper to an existing perception platform in three steps. Although the concepts described in our work apply to virtually any hardware configuration, we illustrate how to follow these steps using our platform called *ipb-car*, the perception platform we use to record datasets for research on autonomous driving.

A. A Practical Use Case Example: The ipb-car Platform

The sensor setup of the ipb-car platform is shown in Fig. 4. It consists of the following sensors:

- 1 x 3D LiDAR Ouster OS1-128
- 1 x 3D LiDAR Ouster OS1-32
- 4 x industrial-grade cameras, Basler ACA2040-35GC
- 1 x navigation system (GNSS/IMU), SBG Ellipse-D
- 1 x PTP Grandmaster computer, based on a Raspberry Pi 4 with an additional GPS receiver
- 1 x QNAP WSW-M2108-2C 10 GbE network switch
- 1 x QNAP QNA-T310G1T 10 GbE, thunderbolt conv.

Our platform is designed with a single network cable to allow the seamless flow of the entire sensor stream from the platform to the host machine. For this, we use a 10 GbE setup. This requires that all sensors have available Ethernet ports, but not necessarily 10 GbE capable ones. For example, in our setup, all the cameras are 1 GbE. All the 1 GbE streams are automatically integrated into the 10 GbE network. We connect all Ethernet sensors to a network switch, which is then connected to an Ethernet-to-thunderbolt converter. As a result, a single thunderbolt cable is the only cable required to connect to the host machine to record data from the platform.

B. Step 1: Identify System Git Components

The most crucial aspect of version control of a new platform is creating Git repositories for each system component. This is a common practice in robotics nowadays. However, we push this technique further by creating repositories for non-software components, such as the URDF models describing the system, the documentation, and others. The reasoning behind this design choice is to enable us to take a snapshot of the entire system with a simple Git hash. Because of this design choice, the first step is identifying all the components required to run the perception platform and ensuring the selected components are accessible through the Git repositories.

C. Step 2: Containerize All Components

Once all components have been identified, a container must be provided for each component. There are two approaches to solving this step. The first is to create one container for each service, for example, one for the cameras, one for the LiDARs, and so on. The second approach is to create one unique container for all the services in the system. We chose the second option since we found that, in practice, it is easier to maintain and adapt to different types of sensor configurations.

D. Step 3: Create a Meta-Workspace Repository

Assuming all system components can be used in Docker containers and the information where each Git repository is already gathered, the next step is to create the meta-workspace repository, introduced in Sec. III. This is a regular Git repository that contains the two files sufficient to describe, build, and run the perception system: the files *.gitmodules* and the *launch.yaml*. The file *.gitmodules* contains all the necessary repositories that describe the perception platform. The containers identified in the previous stage should be added to the file *launch.yaml*. Additionally, all network configurations, such as the host machine's subnet, mask, and IP address, should be included in the *launch.yaml*. An example of how ours looks is shown in Fig. 5.

VI. Applications of Our Method

This section exemplifies how using our method simplifies everyday tasks encountered in the data collection process in robotics research labs.

A. How to Reliably Record Data

We can ensure consistency across multiple datasets by generating a Git tag for the meta-workspace when the system is known to be in a functional state. This tag represents the exact setup, configuration, and code for recording the datasets. Consequently, it is easy to reproduce the same conditions and maintain a reliable and consistent environment for dataset generation, even over long recording periods.

In practice, even small changes in the codebase can result in non-working system configurations. With our framework, as long as we can identify a functioning commit within the Git infrastructure, we can utilize standard tools like git-bisect to reproduce the problem, pinpoint the introduction of the bug, and finally eliminate it. It enables us to restore the last known working system state and effectively debug the issue. Furthermore, by using development branches, other team members can continue their work without disrupting the main recording setup. This allows for parallel development efforts while ensuring the stability and integrity of the system.

B. How to Retrieve System State From Data Recordings

Backtracing the state of the perception platform for a previously recorded dataset is usually challenging. Our approach allows us to check out the entire system at the hash associated with a recording to know precisely how data was recorded, for example, if any pre-processing or filtering was applied or what a specific sensor configuration was. This dramatically increases trust in the data and allows fair comparisons between different recordings.

C. How to Work on Different Hardware Configurations

When deployed in practice, our methodology facilitates using the perception platform on various types of robots, considering their specific constraints. For example, for a wheeled mobile robot that cannot accommodate a laptop for data recording, customization of the meta-workspace becomes necessary. If sensor drivers and the corresponding SDK associated with the cameras are removed, the build time and system footprint can be significantly reduced, allowing for the use of the system on a more resource-constrained platform. This adaptability allows the perception platform to be tailored to different robot configurations and requirements, ensuring its applicability across a wide range of robotic applications. It is also important to note that calibration procedures may need to be adjusted accordingly, which are also tracked when using our methodology.

D. How to Run the System on Different Machines

In practice, it is highly advantageous to have the ability to operate the system using different computers within a team. Sharing a single laptop can become a bottleneck, affecting productivity and flexibility. Our methodology enables developers to have individual copies of the meta-workspace, allowing them to build, launch, and test the system independently without relying on a shared machine or constant coordination. Furthermore, there are no specific requirements for the host operating system. This allows for the operation of the perception platform using an OS that may not support parts of the stack, such as ROS 1. As a result, team members can work independently on different requirements while maintaining a consistent and efficient development environment. This eliminates the challenges of sharing a single machine and allows for seamless collaboration and progress on different requirements within the perception platform. In the daily life of a research lab, this is a great support.

E. How to Migrate between ROS 1 and ROS 2

Migrating to ROS 2 can be challenging, as existing systems may not work seamlessly, and installing ROS 1 and ROS 2 on the same operating system can be problematic. One possible solution is to dedicate a separate computer solely for the ROS 2 migration. Our framework offers a more elegant approach. By creating a new branch of the meta-workspace, developers can change the base image of the Docker containers and work towards achieving a functional ROS 2 setup. Once the migration goal is achieved, the team can decide whether to merge the ROS 2 branch into the main development branch or keep it separate. Importantly, even if the ROS 2 branch is merged, the older ROS 1-compatible branch remains accessible at any time through Git tags or releases, so previously used recording setups can be reproduced on demand easily. This allows developers to switch between ROS versions without changing the host operating system, computer, or other configurations. Our framework simplifies the migration process, reduces associated challenges, and allows us to easily revert to previous working configurations.

VII. Conclusion

This paper presented a novel methodology to develop version-controlled perception platforms that we use at the Photogrammetry and Robotics lab at the University of Bonn. Our approach to building a sensor suite enhances the reproducibility of the system's state and is easily adapted to any existing perception platform. Our methodology relies only on standard tools in any modern operating system, namely Git and Docker. It is, from our point of view, key when building a complex robot or sensing infrastructure. This allows us to successfully build a containerized software stack that can run on multiple host machines without setting the host to a particular state. All the software and tools used to develop our perception platform are open-source for the benefit of the community. We envision this work as a step forward in building more reliable perception platforms in the context of mobile robots that can accelerate the research of new algorithms for autonomous vehicles.

REFERENCES

[1] J. Behley, M. Garbade, A. Milioto, J. Quenzel, S. Behnke, C. Stachniss, and J. Gall. SemanticKITTI: A Dataset for Semantic Scene Understanding of LiDAR Sequences. In *Proc. of the IEEE/CVF Intl. Conf. on Computer Vision (ICCV)*, 2019.

[2] G. Brostow, J. Fauqueur, and R. Cipolla. Semantic Object Classes in Video: A High-Definition Ground Truth Database. *Pattern Recognition Letters*, 30(2):88–97, 2009.

[3] H. Caesar, V. Bankiti, A. Lang, S. Vora, V. Liong, Q. Xu, A. Krishnan, Y. Pan, G. Baldan, and O. Beijbom. nuScenes: A multimodal dataset for autonomous driving. *arXiv preprint*, arXiv:1903.11027, 2019.

[4] H. Caesar, V. Bankiti, A. Lang, S. Vora, V. Liong, Q. Xu, A. Krishnan, Y. Pan, G. Baldan, and O. Beijbom. nuScenes: A Multimodal Dataset for Autonomous Driving. In *Proc. of the IEEE/CVF Conf. on Computer Vision and Pattern Recognition (CVPR)*, 2020.

[5] E. Cervera. Try to start it! the challenge of reusing code in robotics research. *IEEE Robotics and Automation Letters (RA-L)*, 4(1):49–56, 2019.

[6] E. Cervera and A.P. Del Pobil. Roslab: sharing ros code interactively with docker and jupyterlab. *IEEE Robotics and Automation Magazine (RAM)*, 26(3):64–69, 2019.

[7] M. Chang, J. Lambert, P. Sangkloy, J. Singh, S. Bak, A. Hartnett, D. Wang, P. Carr, S. Lucey, D. Ramanan, and J. Hays. Argoverse: 3D Tracking and Forecasting with Rich Maps. In *Proc. of the IEEE/CVF Conf. on Computer Vision and Pattern Recognition (CVPR)*, 2019.

[8] M. Colosi, I. Aloise, T. Guadagnino, D. Schlegel, B. Corte, K. Arras, and G. Grisetti. Plug-And-Play SLAM A Unified SLAM Architecture for Modularity and Ease of Use. In *Proc. of the IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS)*, 2020.

[9] M. Cordts, S.M. Omran, Ramos, T. Rehfeld, M. Enzweiler, R. Benenson, U. Franke, S. Roth, and B. Schiele. The cityscapes dataset for semantic urban scene understanding. In *Proc. of the IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[10] P. Dellenbach, J. Deschaud, B. Jacquet, and F. Goulette. CT-ICP Real-Time Elastic LiDAR Odometry with Loop Closure. In *Proc. of the IEEE Intl. Conf. on Robotics & Automation (ICRA)*, 2022.

[11] J.C. Eidson, M. Fischer, and J. White. Ieee-1588™ standard for a precision clock synchronization protocol for networked measurement and control systems. In *Proc. of the Annual Precise Time and Time Interval Systems and Applications Meeting*, pages 243–254, 2002.

[12] T. Fischer, W. Vollprecht, S. Traversaro, S. Yen, C. Herrero, and M. Milford. A robostack tutorial: Using the robot operating system alongside the conda and jupyter data science ecosystems. *IEEE Robotics and Automation Magazine (RAM)*, 29(2):65–74, 2021.

[13] N. Gählert, N. Jourdan, M. Cordts, U. Franke, and J. Denzler. Cityscapes 3d: Dataset and benchmark for 9 dof vehicle detection. *arXiv preprint*, arxiv:2006.07864, 2020.

[14] A. Geiger, P. Lenz, and R. Urtasun. Are we ready for Autonomous Driving? The KITTI Vision Benchmark Suite. In *Proc. of the IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, 2012.

[15] A. Geiger, P. Lenz, C. Stiller, and R. Urtasun. Vision meets Robotics: The KITTI Dataset. *Intl. Journal of Robotics Research (IJRR)*, 32(11), 2013.

[16] X. Huang, X. Cheng, Q. Geng, B. Cao, D. Zhou, P. Wang, Y. Lin, and R. Yang. The apolloscape dataset for autonomous driving. In *Proc. of the IEEE/CVF Conf. on Computer Vision and Pattern Recognition Workshops*, 2018.

[17] G. Kim, Y. Park, Y. Cho, J. Jeong, and A. Kim. Mulran: Multimodal range dataset for urban place recognition. In *Proc. of the IEEE Intl. Conf. on Robotics & Automation (ICRA)*, 2020.

[18] S. Macenski, T. Foote, B. Gerkey, C. Lalancette, and W. Woodall. Robot operating system 2: Design, architecture, and uses in the wild. *Science Robotics*, 7(66):eabm6074, 2022.

[19] W. Maddern, G. Pascoe, C. Linegar, and P. Newman. 1 year, 1000 km: The oxford robotcar dataset. *Intl. Journal of Robotics Research (IJRR)*, 36(1):3–15, 2017.

[20] M. Montemerlo, N. Roy, and S. Thrun. Perspectives on standardization in mobile robot programming: the carnegie mellon navigation (carmen) toolkit. In *Proc. of the IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS)*, volume 3, pages 2436–2441, 2003.

[21] G. Neuhold, T. Ollmann, S.R. Bulo, and P. Kontschieder. The Mapillary Vistas Dataset for Semantic Understanding of Street Scenes. In *Proc. of the IEEE Intl. Conf. on Computer Vision (ICCV)*, 2017.

[22] P. Newman. Moos - mission orientated operating suite, 2008.

[23] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Ng. Ros: an open-source robot operating system. In *ICRA Workshop on Open Source Software*, 2009.

[24] M. Ramezani, Y. Wang, M. Camurri, D. Wisth, M. Mattamala, and M. Fallon. The newer college dataset: Handheld lidar, inertial and vision with ground truth. In *Proc. of the IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS)*, 2020.

[25] G. Ros, L. Sellart, J. Materzynska, D. Vazquez, and A. Lopez. The synthia dataset: A large collection of synthetic images for semantic segmentation of urban scenes. In *Proc. of the IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[26] P. Sun, H. Kretzschmar, X. Dotiwalla, A. Chouard, V. Patnaik, P. Tsui, J. Guo, Y. Zhou, Y. Chai, B. Caine, et al. Scalability in perception for autonomous driving: Waymo open dataset. In *Proc. of the IEEE/CVF Conf. on Computer Vision and Pattern Recognition (CVPR)*, 2020.

[27] P. Triantafyllou, R. Afonso Rodrigues, S. Chaikunsaeng, D. Almeida, G. Deacon, J. Konstantinova, and G. Cotugno. A methodology for approaching the integration of complex robotics systems: Illustration through a bimanual manipulation case study. *IEEE Robotics and Automation Magazine (RAM)*, 28(2):88–100, 2021.

[28] I. Vizzo, T. Guadagnino, J. Behley, and C. Stachniss. VDBFusion: Flexible and Efficient TSDF Integration of Range Sensor Data. *Sensors*, 22(3), 2022.

[29] I. Vizzo, T. Guadagnino, B. Mersch, L. Wiesmann, J. Behley, and C. Stachniss. KISS-ICP: In Defense of Point-to-Point ICP – Simple, Accurate, and Robust Registration If Done the Right Way. *IEEE Robotics and Automation Letters (RA-L)*, 8(2):1029–1036, 2023.

[30] R. Wang, M. Schwörer, and D. Cremers. Stereo DSO: Large-Scale Direct Sparse Visual Odometry with Stereo Cameras. In *Proc. of the IEEE/CVF Intl. Conf. on Computer Vision (ICCV)*, October 2017.

[31] A. Wendt and T. Schüppstuhl. Proxying ros communications–enabling containerized ros deployments in distributed multi-host environments. In *Proc. of the Intl. Symp. on System Integration (SII)*, pages 265–270, 2022.

[32] R. White and H. Christensen. Ros and docker. *Robot Operating System (ROS) The Complete Reference (Volume 2)*, pages 285–307, 2017.

[33] B. Wilson, W. Qi, T. Agarwal, J. Lambert, J. Singh, S. Khandelwal, B. Pan, R. Kumar, A. Hartnett, J.K. Pontes, D. Ramanan, P. Carr, and J. Hays. Argoverse 2: Next generation datasets for self-driving perception and forecasting. In *Proc. of the Conf. on Neural Information Processing Systems (NeurIPS)*, 2021.

[34] F. Yu, H. Chen, X. Wang, W. Xian, Y. Chen, F. Liu, V. Madhavan, and T. Darrell. BDD100K: A Diverse Driving Dataset for Heterogeneous Multitask Learning. In *Proc. of the IEEE/CVF Conf. on Computer Vision and Pattern Recognition (CVPR)*, 2020.