Reengineering of Simulation-based Digital Twin Toolchains for Automated Vehicles to Mitigate Technical Debt: A Case Study

Dennis Eller
German Aerospace Center (DLR)
Brunswick, Germany
University of Cologne
Cologne, Germany
dennis.eller@dlr.de

Niklas Först
German Aerospace Center (DLR)
Cologne, Germany
niklas.foerst@dlr.de

Ralf Stemmer
German Aerospace Center (DLR)
Oldenburg, Germany
ralf.stemmer@dlr.de

Tjark Koopmann
German Aerospace Center (DLR)
Oldeburg, Germany
tjark.koopmann@dlr.de

Alexander Weinert
German Aerospace Center (DLR)
Cologne, Germany
alexander.weinert@dlr.de

Philipp M. Fischer
German Aerospace Center (DLR)
Brunswick, Germany
philipp.fischer@dlr.de

Kim Grüttner
German Aerospace Center (DLR)
Oldenburg, Germany
kim.gruettner@dlr.de

Andreas Gerndt
German Aerospace Center (DLR)
Brunswick, Germany
andreas.gerndt@dlr.de

Michael Felderer
German Aerospace Center (DLR)
Oberpfaffenhofen, Germany
University of Cologne
Cologne, Germany
michael.felderer@dlr.de

Abstract

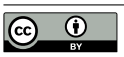
In early vehicle development, simulation-based Digital Twin toolchains enable simulated testing of automated driving functions prior to real-world testing. In a research setting, they are created by researchers and comprise research software prototypes, proprietary tools, and customized open-source software. Although this composition initially yields quick results, it accumulates technical debt and erodes reproducibility. We study a case of architectural technical debt in simulation-based Digital Twin toolchains for automated vehicles and a) identify symptoms of architectural technical debt present in the toolchains via analyses and interviews with researchers, b) investigate the underlying architectural technical debt causing these symptoms, c) relate them with categories in the field of architectural technical debt, d) reengineer the toolchains towards a containerized service-based architecture, and e) discuss the tradeoffs and lessons learned. Our results show that an intermediate implementation already substantially mitigates architectural technical debt while preserving performance metrics.

CCS Concepts

• **Computing methodologies** → **Simulation environments**; • **Software and its engineering** → *Software architectures*; • **Applied computing** → *Service-oriented architectures*.

Keywords

Digital Twin, Service-based architecture, Technical debt



MSSiS '26, Rio de Janeiro, Brazil

ACM ISBN 979-8-4007-2380-3/26/04
https://doi.org/10.1145/3786146.3788641

ACM Reference Format:
Dennis Eller, Niklas Först, Ralf Stemmer, Tjark Koopmann, Alexander Weinert, Philipp M. Fischer, Kim Grüttner, Andreas Gerndt, and Michael Felderer. 2026. Reengineering of Simulation-based Digital Twin Toolchains for Automated Vehicles to Mitigate Technical Debt: A Case Study. In *8th Workshop on Modeling and Simulation of Software-Intensive Systems (MSSiS '26), April 12–18, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3786146.3788641

1 Introduction

The development of vehicles is becoming increasingly complex, due to new regulations and more sophisticated software [36]. In particular, the demand for autonomous driving is increasing rapidly [14], while its verification remains a main challenge [37].

Modern Automatic Vehicles (AVs) implement Automatic Driving Functions (ADFs) to, e.g., keep the lane and speed limit, to overtake slower vehicles, or to prevent collisions. The German Aerospace Center (DLR) develops novel ADFs as part of researching future mobility and evaluates them by simulating their behavior in predefined scenarios. An automated monitor supervises each simulation for undesired behavior and renders a video of the ADF's behavior when controlling the AV (Figure 1).

Currently, it is infeasible to specify the expected behavior of an AV in all situations. Hence, Subject Matter Experts (in the following experts) manually evaluate the videos to determine whether or not the AV behaves as expected. Each simulation comprises multiple software tools in toolchains.

A Digital Twin (DT) is a virtual representation of an actual system that is continuously updated with real-time data [17] and can be used to simulate vehicles' behavior [7]. Such a twin can be created before the physical system, which is often referred as a Digital Twin Prototype (DTP) [19]. DTs are used in several domains, and their

Figure 1: A simulation of an automated vehicle (left) performing a driving scenario that gets monitored (right).

success is based on the simulation of physical aspects in a software domain [11]. This makes a DT essentially a software project.

When implementing new features in a software project, developers face trade-offs between quick and maintainable solutions [5]. Opting for the quick solution yields short-term benefits, but typically leads to overly complex implementations, so-called Technical Debt (TD). Accumulated TD decreases the velocity of development in the long run [12], and erodes its reproducibility. TD has been extensively researched, e.g., to evaluate its manifestation in software development [31], or to manage its impact on software projects [1].

Particularly, Architectural Technical Debt (ATD) has been identified as a sub-type of TD [10, 27]. ATD denotes short-term beneficial decisions that do not manifest in the source code, but instead in the overall architecture of the software system. While there exists extensive tool support to identify and quantify TD [6, 34], the quantification of ATD is a topic of ongoing research [24, 32].

Most existing work studies TD either in open source projects [26, 35, 42] or in industrial use cases [18, 38]. In contrast, little work exists on the impact of ATD on the quality of DTs in a research context. Software in a research context is developed with different goals, under different circumstances, and by different people than industrial software. Typically, requirements are not documented, verification and validation are done manually by experts, and the software architecture is strongly influenced by available hardware [22].

In this work, we study ATD in simulation-based DTs in a research context by investigating toolchains used for evaluating ADFs in AVs. For that, we a) identify symptoms of ATD present in the toolchains via analyses and interviews with researchers, b) investigate the underlying ATD causing these symptoms, c) relate them to categories in the field of ATD, d) reengineer the toolchain towards a containerized service-based architecture, and e) discuss tradeoffs and lessons learned.

Given the complexity of the software and the significant adaptations required to develop a fully service-based simulation-based DT, it is not feasible to pursue this within the scope of this work. Instead, we present a partially service-based architecture that already yields numerous benefits over the existing implementation, but only requires a fraction of the development effort. Furthermore, we focus on the virtual part of the DT, which can be seen as a DTP [19]. We believe the lessons learned to be transferable to the physical system, as well as other cyber-physical domains where DT-driven simulation pipelines are employed, e.g., smart-city or the simulation of manufacturing plants.

The remainder of this work is organized as follows: We first review the existing work on ATD, software quality metrics, as well as microservices and containerization in Section 2. In Section 3, we describe the case study with the domain background and current implementation. Then, in Section 4 we describe the shortcomings of the original implementation as perceived by experts, as well as the underlying ATD causing these symptoms. Subsequently, we present the measures taken to reduce the ATD in Section 5 and report on the resulting mitigation in Section 6. After that, we discuss the results and lessons learned in Section 7 and close with a summary of our results and an overview of future work in Section 8.

2 Related Work

Architectural Technical Debt Categorization. In [39], a theory for Architectural Technical Debt was proposed, which connects causes, consequences, symptoms, and management styles of ATD. ATD is divided into the ATD types *MVP that Stuck*, *Workaround that Stayed*, *Re-inventing the Wheel*, *Source Code ATD*, *Architectural Lock-in* and *New Context*, *Old Architecture*. They also distinguish between different types of management strategies when dealing with ATD: active (allocate effort to reduce debt), reactive (delay effort until debt blocks progress), and passive (work around technical debt rather than fixing it).

Software Quality Metrics. One major metric for software quality is *ISO 25010*, an international standard that "defines a product quality model [for] software products" [21]. The standard identifies *Functional Suitability*, *Performance Efficiency*, *Compatibility*, *Interaction Capability*, *Reliability*, *Security*, *Maintainability*, *Flexibility*, and *Safety* as the main metrics of software quality.

This standard does not consider the unique challenges faced when developing software in a research context [22]. The *FAIR Principles* [43] are a well-established metric for evaluating the quality of scientific data and its management. Building on these principles, the *FAIR4RS Principles* [8] can be used to judge whether software is *Findable*, *Accessible*, *Interoperable*, and *(Re)Usable*. Both *ISO 25010* and the *FAIR4RS Principles* are abstract and prescribe no concrete guidance to developers. In contrast, the *Twelve-Factors* [41] represent a set of concrete recommendations that are designed to assist in the development of Software-as-a-Service (SaaS) applications. These recommendations include, e.g., *using a single code base under*

version control, *explicitly declared dependencies*, and *maintaining parity between development, staging, and production environments*.

All metrics discussed so far only serve to evaluate existing software products and their management. In contrast, Auer et al. [4] describe a set of metrics that serve to support the decision for or against reengineering a monolithic software system into a service-based one. Using this framework, software systems are evaluated with respect to their *Functional Suitability*, *Performance Efficiency*, *Reliability*, *Maintainability*, *Cost*, and *Process Related* metrics. Lenarduzzi et al. [25] show that migrating to microservices can reduce TD, but Villa et al. [40] show that migrating to microservices is not an easy task and can even increase TD.

Microservices and Containerization. In a microservice architecture, a software system consists of cohesive and independent modules that implement only a small set of functionalities, and are able to interact on their own with other modules via a defined interface [16]. Single modules and their dependencies can easily be exchanged or upgraded without affecting the whole software system. The independent nature of microservices also prevents a technology lock-in, as single modules can be implemented as needed [16, 30, 44]. Microservices are commonly implemented using containerization. Containerization is a virtualization technology where an application is packaged together into a standardized image with its runtime environment. This image can be run nearly everywhere. Containers are more lightweight, easier to manage, more scalable, and have faster startup than other virtualization methods like virtual machines [28, 33].

3 Case Study: Simulation-Based Testing of Automatic Driving Functions

Domain Background. The researchers develop ADFs for AVs and validate their behavior in simulations before testing them in the real world. For this, they use the workflow shown in Figure 2.

Researchers initially describe an abstract scenario involving a single AV. One example of such a scenario is a situation in which the vehicle is traveling along a two-lane road together with other vehicles. At some point, the current lane is blocked by another vehicle. The ADF then has to steer the vehicle around this obstacle. Using a *Concretizer*, the experts instantiate this abstract scenario into multiple concrete ones which can be used by the simulator. These scenarios differ, e.g., in the number of other vehicles involved, in the sizes and acceleration profiles of these vehicles, or in the lane widths. Each concrete scenario is then coupled with an ADF and executed in the *simulator*, which simulates the behavior of all vehicles in the scenario. The ADF governs the vehicle, while the scenario description governs all other vehicles. Inspecting each scenario manually for compliance would be infeasible for the experts. Thus, the simulations are automatically monitored for deviations from the desired behavior by a *Monitor*. If the *Monitor* flags a simulation as non-compliant, it is rendered as a video. An experts can then analyze the videos and use them to further improve the ADF.

Existing Implementation. Initial implementation of the toolchains was subject to unclear requirements, did not follow formal software engineering processes as they were perceived to inhibit research progress, and the software architecture was driven by the available hardware. These circumstances are quite typical for software in a research context [22]. The experts have partially implemented the workflow described in the previous paragraph. This development resulted in two toolchains, the Scenario Generation Toolchain (SGTC) and the Sceneratio Execution Toolchain (SETC). The former toolchain executes a fixed ADF in numerous generated scenarios, while the latter executes a fixed ADF in a number of pre-defined scenarios and monitors these executions for nominal behavior [20]. We illustrate the resulting implementation in Figure 3.

The SGTC generates variations of abstract scenarios, resulting in concrete scenarios. It builds on concepts described by Becker et al. [9]. The experts first specify abstract scenarios as one or more *Traffic Sequence Chart (TSC) files* [13]. These TSC files contain the scenario specification as well as a world model describing physical laws and properties. The abstract scenarios given as TSCs are concretized using the tool *TSC2OpenX* [23]. This tool produces variations of the TSC files as intermediate results, and outputs concrete scenarios as *OpenX* files. Each OpenX file comprises *OpenDRIVE* [2] and *OpenSCENARIO* [3] files. The concrete scenarios are executed by the simulator *CARLA* [15], which is an open-source simulator for autonomous driving research. A *Scenario Runner*, provided by the CARLA developers, reads the OpenX files and triggers the API of CARLA. The experts have constructed a Docker container comprising the original TSC files, TSC2OpenX, the Scenario Runner, and an instance of CARLA. The ADF to be evaluated is hard-coded into the configuration of CARLA.

The SETC contains the *Scenario Manager*, which is a Python application developed by the experts. The scenarios and ADFs are hard-coded into the *Scenario Manager*. The *Scenario Manager* provides these to CARLA and to a *Monitor*. CARLA executes the scenario and provides virtual sensor data to the Scenario Monitor. The Scenario Monitor fuses the information about the executed scenario with the data received from the simulation and monitors the execution for undesired behavior. In this toolchain all three constituent components communicate asynchronously using ROS 2 [29]. The experts deploy the Scenario Manager and CARLA in a single Docker container. The Scenario Monitor is deployed as a non-containerized application on the same machine that executes the Docker container. In the remainder of this work, we do not consider the *Scenario Monitor* as part of the SETC. It is an application that only obtains data from the toolchain instead of being part of it.

4 Methodology and Technical Debt in the Existing Implementation

In the past, a passive management style regarding ATD was pursued. When problems arose, reactive management techniques such as opportunistic patching were used. In the scope of this work, we pursue a reactive and active management style by performing a major refactoring (reactive), systematically dedicating time to reduce technical debt (active), and building technical credit by proactively improving architectural elements (active). To identify ATD to be mitigated by the reengineering, we met with experts to gain insight into the existing implementation during a two-day workshop. We opted for an agile approach for the engineering, because we determined it would be infeasible to identify all ATD by interviewing experts and inspecting the current implementation.

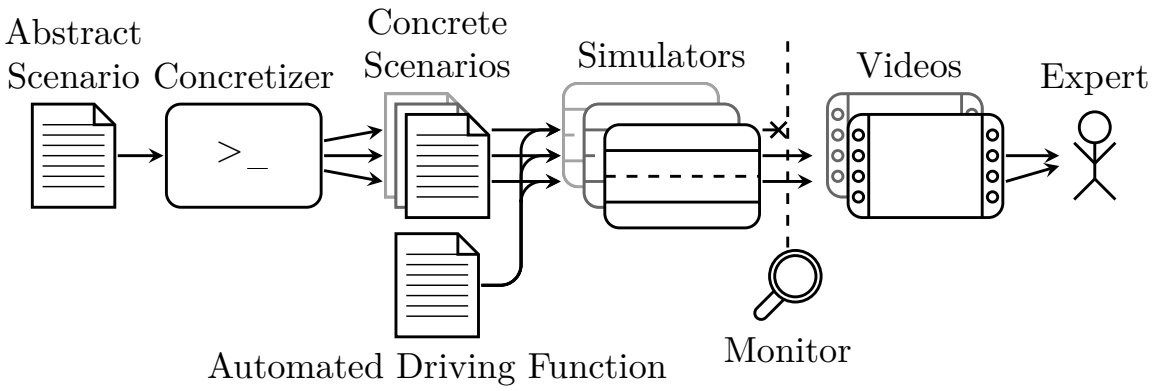


Figure 2: The conceptual workflow from an abstract scenario to analysis by an expert.

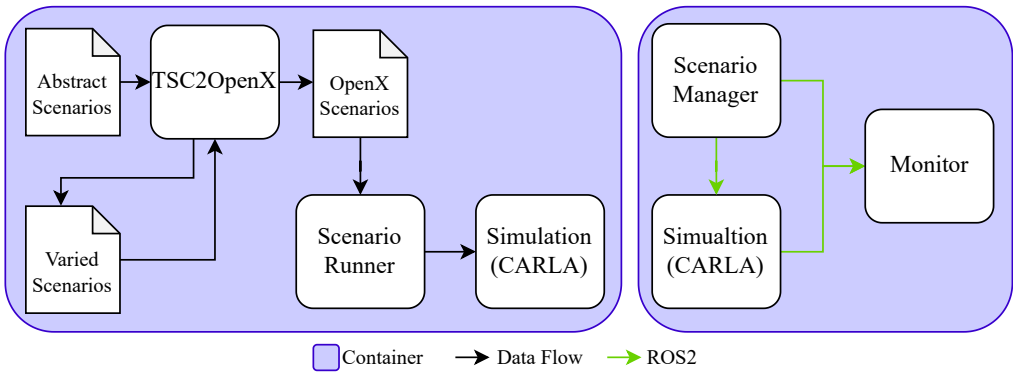


Figure 3: Current architecture of Sceneratio Generation Toolchain (left) and Scenario Execution Toolchain (right).

For that, two software engineers met with stakeholders (including the experts) in a sync meeting to gather symptoms of ATD. The software engineers then extended the existing implementation to mitigate the ATD. They consulted with experts on demand to answer questions. Afterwards, they again met with all stakeholders, presented their progress, and gathered new ATD based on the feedback given. This cycle was repeated for 3 months on a bi-weekly basis. While refining a target architecture, the two software engineers incrementally adjusted the existing implementation to conform to the new architecture. During the initial sync meeting with stakeholders, the software engineers observed that the main symptom of the ATD is the low software quality, which results in a low reproducibility of the toolchains and the results they create. We categorize the ATD using a framework that we derived from multiple sources of software quality and ATD types presented in Section 2. The categories of this framework are illustrated in Figure 4.

We base our initial categories on ISO 25010 [21] and FAIR4RS [8]. From ISO 25010, we omit the categories *Security*, *Interaction Capability*, *Performance Efficiency*, and *Safety*, because they do not contribute to reproducibility. From FAIR4RS, we omit the categories *Findable* and *Accessible*, as they are mainly affected by the deployment of the software, which is out of scope of this work. The FAIR4RS categories *Interoperability* and *Reusability* are subcategories of the ISO 25010 categories *Compatibility* and *Reusability*, respectively. Additionally, we augmented the ATD items with relevant ones from the Twelve-Factor App methodology [41]. We present the collected ATD items in Table 1.

We have collected 20 ATD items that contribute to the low reproducibility as the main drawback of the current toolchains. To increase the reproducibility, we propose a containerization service-based architecture, which we describe in the following section.

5 Reducing the Technical Debt

Based on the ATD outlined in Section 4, we propose a service-based architecture that integrates the SGTC and the SETC. We describe this architecture in Section 5.1. However, during implementation, we encountered several challenges. This includes the adaptation of components to expose their functionality via an API, to consume APIs of other services, or to use databases instead of files for data exchange. Implementing these changes is out of the scope of this work. Therefore, in section 5.2 we present an intermediate software architecture that is feasible to implement and already reduces ATD.

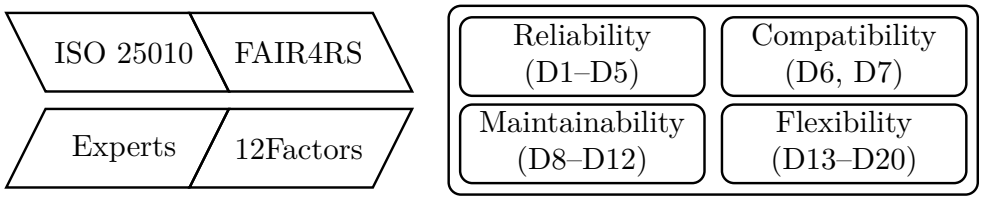


Figure 4: Our framework for categorizing Architectural Technical Debt.

Table 1: Gathered Architectural Technical Debt according to the proposed framework.

Reliability		ATD Type
D1	Toolchain components are tightly coupled.	Source Code ATD
D2	Failure of one toolchain component does affect other components.	Architectural Lock-in
D3	Toolchain components are dependent on specific service instances.	Architectural Lock-in
D4	Toolchain components are not disposable.	Source Code ATD
D5	Toolchain components are not scalable across multiple processes.	Source Code ATD
Compatibility		
D6	The toolchain interferes with other tools in its environment.	Architectural Lock-in
D7	Communication is not done via standards.	Architectural Lock-in
Maintainability		
D8	Toolchain components are not independently replaceable.	Architectural Lock-in
D9	Toolchain components are not independently modifiable.	Architectural Lock-in
D10	Toolchain components are not independently reusable.	Architectural Lock-in
D11	Toolchain dependencies and configurations are not explicit.	Architectural Lock-in
D12	Toolchain components implement already existing functionality.	Re-inventing the Wheel
Flexibility		
D13	Toolchain components are not independent of their environment.	Architectural Lock-in
D14	Host systems need to be reconfigured for each toolchain use case.	Source Code ATD
D15	Toolchain cannot be easily provisioned.	Source Code ATD
D16	Used services cannot be swapped out.	Architectural Lock-in
D17	Toolchain components are not self-contained.	Source Code ATD
D18	Scenarios are hardcoded into toolchain components.	The Workaround that Stayed
D19	Driving functions are hardcoded into toolchain components.	The Workaround that Stayed
D20	Toolchain components are all developed with the same technology.	Architectural Lock-in

5.1 Target Architecture

Following good practices for service-based architectures, we propose the target architecture in Figure 5.

The toolchain components are split into microservices to achieve operational independence. Instead of direct file access, our architecture stores data in databases to increase compatibility. To increase the flexibility and maintainability of the toolchain, the databases run in their own container, are attached as resources, and run independently of other services. We split *TSC2OpenX* into the separate services *TSC2SMT*, *SMT2Instance* and *Instance2OpenX* to break the cycle between the *Varied Scenarios* and *TSC2OpenX*. The *Scenario Runner* from the SGTC and the *Scenario Manager* from the SETC are combined into the *Scenario Executor* because they serve similar purposes. Since simulations dominate the toolchain's execution time, multiple instances of CARLA are used, which are managed by a load balancer. Cross-cutting services are centralized to de-duplicate functionality and simplify service access. The architecture includes an API gateway to provide a consistent interface to the services of the architecture. A logging service collects the logs from the services in a central location for debugging purposes. We include a monitoring service that collects events from all services in the architecture. Both the logging and the monitoring increase the maintainability of the architecture. We include a central event broker to enable asynchronous communication between the microservices, thus increasing the flexibility and maintainability of the architecture.

5.2 Intermediate Architecture

As a first step toward a service-based architecture, we develop an intermediate architecture shown in Figure 6.

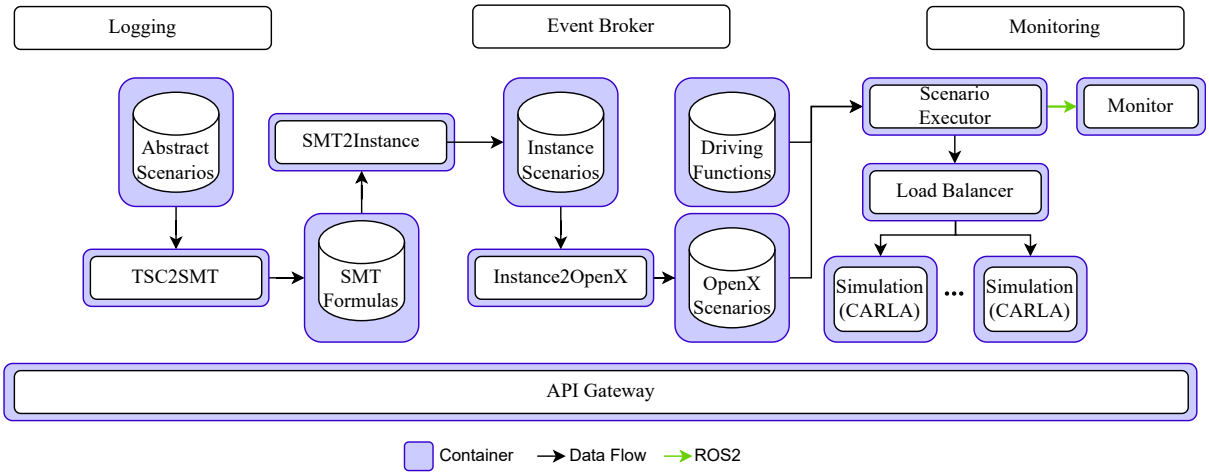


Figure 5: Target architecture for the reengineered toolchain.

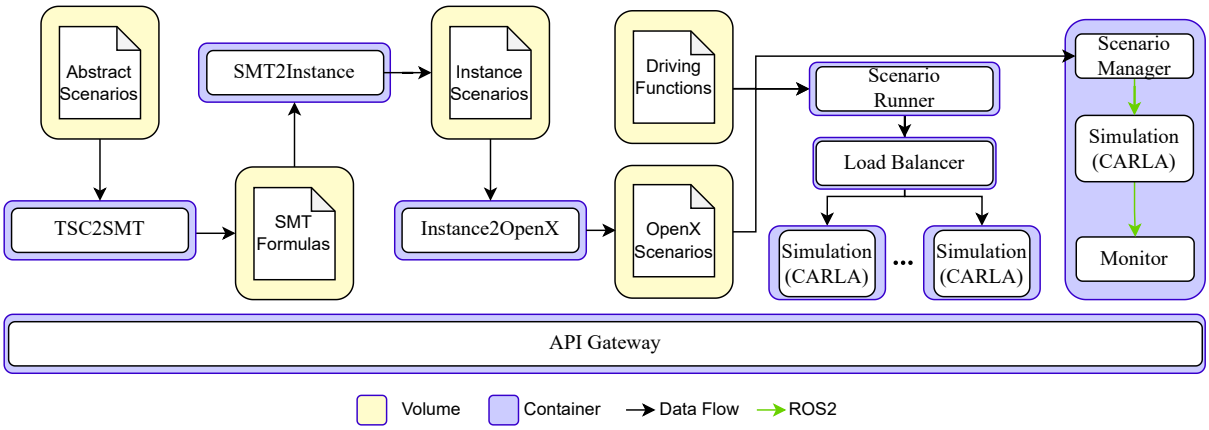


Figure 6: Intermediate architecture for the reengineered toolchain.

In contrast to the target architecture, the *Scenario Manager* and the *Scenario Runner* are individual components, as they are still an integral part of their respective toolchains. The separation of TSC2OpenX was successfully done. Databases are currently not used because the toolchain components work with files or have their inputs hardcoded. As an intermediate step, the access to the file system is virtualized using Docker volumes to avoid dependencies on files on the host machine. Docker volumes are a mechanism to store data outside of containers that can be shared between services. At the time of writing, there is no direct connection between the two toolchains. The scenarios generated in the SGTC have to be imported manually into the SETC. The proposed monitor is still under development, so the simulations are monitored manually. To facilitate the use of the toolchains, we use Docker Compose. This allows the containers to be set up and started in a reproducible way.

6 Evaluation of Technical Debt Mitigation

We evaluate the architectures presented in the previous section against the identified ATD. We show the results for the target architecture and the intermediate architecture in Table 2.

The target architecture mitigates all of the identified ATD. Reliability is achieved because the toolchain components run independently of each other (D1) and are independent of specific instances of supporting services (D3). The failure of one component does not affect other components (D2). It also makes parts of the toolchain easily disposable (D4), which is necessary to scale parts of the toolchains across multiple processes (D5). By separating the components into individual services, the individual services can be implemented with different technologies (D20). When components run as individual services, it is easier to independently replace (D8), modify (D9), and reuse (D10) parts of the toolchain. Furthermore, it reduces duplicate implementations by combining the Scenario Runner and the Scenario Manager into the Scenario Executor (D12). The usage of containers increases the maintainability and flexibility of the toolchains. Dependencies and configurations

Table 2: Mitigation of Architectural Technical Debt of the proposed architectures.

(a) Target Architecture

No.	Mitigated	No.	Mitigated
D1	✓	D11	✓
D2	✓	D12	✓
D3	✓	D13	✓
D4	✓	D14	✓
D5	✓	D15	✓
D6	✓	D16	✓
D7	✓	D17	✓
D8	✓	D18	✓
D9	✓	D19	✓
D10	✓	D20	✓

(b) Intermediate Architecture

No.	Mitigated	No.	Mitigated
D1	(✓)	D11	✓
D2	(✓)	D12	✗
D3	✗	D13	✓
D4	(✓)	D14	✓
D5	(✓)	D15	(✓)
D6	✓	D16	(✓)
D7	✓	D17	(✓)
D8	(✓)	D18	✗
D9	(✓)	D19	✗
D10	(✓)	D20	✓

are made explicit (D11), because a container packages everything that a microservice needs to run. As a result, it can share an environment with other tools (D6), which improves compatibility. It also makes the services as independent as possible from the underlying hardware and software (D13). The host system does not need to be reconfigured for each toolchain use case (D14) because the toolchains are isolated from each other by containers (D17). Because supporting services like databases are attached as resources, they can be swapped out, providing greater flexibility (D16). With the usage of databases, scenarios (D18) and driving function (D19) can be externalized and removed from the Scenario Executor. By using a container runtime, the toolchains can easily be deployed (D15). Communication between the services is done via standards, either synchronously via APIs or asynchronously via an event broker (D7).

The intermediate architecture mitigates 6 out of 20 ATD items completely and 10 items partially. It achieves limited reliability, because not all components can run independently of each other (D1) and are independent of specific service instances (D3). A failure of one component affects other components (D2) because some are tightly coupled. Parts of the toolchains are not easily disposable (D4), and cannot scale across using multiple processes (D5). Maintainability is limited, because not all components run as individual services. It is difficult to replace (D8), modify (D9), and reuse (D10) parts of the toolchain individually. But components that run as individual services can be implemented with different technologies (D20). The Scenario Manager and Scenario Runner implement similar functionality, which results in duplicate functionality (D12). Explicit declaration of dependencies and configurations increases the maintainability (D11). Therefore, the toolchains can share a common environment with other tools (D6). It also makes the services as independent as possible from the underlying hardware and software (D13). The host system does not need to be reconfigured for each toolchain use case (D14) because the toolchains are isolated from each other by containers (D17). Supporting services are not present because no databases are used. This architecture uses file access, via Docker volumes, which makes the file system swappable, which increases flexibility (D16). Using a container runtime, the toolchains can easily be deployed (D15). Communication between the toolchains is not done via standards, because they are currently not connected directly (D7). Scenarios (D18) and driving functions (D19) are partly hard-coded in the toolchains. To evaluate the performance implications of the intermediate architecture, the simulation execution time between the current and intermediate implementations are compared in Fig. 7.

With one scenario, the execution time of both implementations is identical. With a scenario number from 2 to 4, the scaling of the intermediate implementation (one CARLA instance per scenario) yields a measurable performance benefit. With 5 or more scenarios, the intermediate implementation is slower. A likely cause is that 5 or more instances of CARLA exceed the available resources of the test system, or undiscovered bottlenecks are triggered.

7 Lessons Learned

The transition of a monolithic simulation-based Digital Twin to a containerized service-based architecture is not a one-time effort. This work demonstrates that such a transition is not a trivial task and should be considered as early as possible in the life cycle of the software. This section highlights good practices and insights that can be learned from this transition process.

The main aim of our work was to **improve the reproducibility of the toolchains**. This effort focuses both on the toolchains themselves and their comprising parts as well as on the results the toolchains generate. In the context of business applications, containers represent the primary technology to facilitate reproducibility. For the software in this work, **containers are necessary, but insufficient to ensure reproducibility**.

Firstly, while containers virtualize the software stack, **containers can still expose the underlying hardware** to the software running inside them. CARLA has strict hardware dependencies on NVIDIA graphics cards and utilizes ROS 2 for communication with other components of the toolchains. In our implementation, ROS 2 attempted to communicate between two containers using shared memory for efficiency. This failed, since by design, containers do not share a memory space. While containers can be configured to have shared memory, doing so breaks the isolation between the containers. In contrast, **breaking isolation may be necessary for efficiency**, as software in a research context typically produces and consumes large amounts of data. These data may not be efficiently transferred via industry-standard message-oriented interfaces. Also,

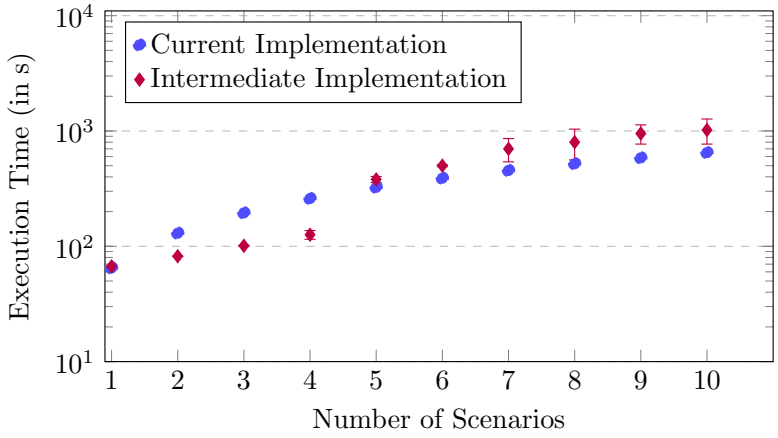


Figure 7: Execution time of the current implementation and the intermediate implementation. Measured on an Intel Core i9-13900K, 128 GB RAM, NVIDIA RTX A5000.

the benefit of scaling toolchain components has a limit and might even have a negative impact on performance.

Secondly, **containerization requires researchers to create and maintain additional software artifacts**, such as container build scripts. In one instance, we were unable to build a container image because dependencies became outdated or unavailable from public sources. The maintenance of container build scripts places an additional maintenance burden on the researchers. Moreover, containerizing an application involves explicitly declaring and managing all its dependencies. Dependency management is also a part of containerizing business software. But, **dependency management is particularly burdensome for research software** due to the reliance on other research software that is not consistently versioned and whose long-term stability is uncertain. However, once these dependencies are defined, **researchers can benefit from a reproducible and consistent build environment that reduces dependency issues and simplifies the deployment process**.

Finally, **containers still require the use of a container execution environment**, even if they are built and managed in a well-organized and reproducible way. This means that researchers must either administer and maintain the infrastructure for executing containers or rely on external vendors to provide such an execution environment. The former option can place an additional burden on researchers, while the latter can lead to vendor lock-in.

8 Conclusion and Outlook

In this paper, we worked towards the reengineering of simulation-based Digital Twin toolchains to a containerized service-based architecture. We first gathered ATD items via an agile approach, where we met with experts on a regular basis. To categorize the ATD items, we proposed a framework based on existing literature. We then developed a target architecture for the toolchains that mitigates the identified ATD. During development, it became clear that limitations in the existing toolchains prevented the target architecture from being realized. We developed an intermediate architecture that serves as a milestone on the way to the target architecture. We presented both of these architectures and evaluated them with respect to the collected ATD items, and conducted a performance evaluation of the intermediate architecture. Finally, we discussed lessons learned from implementing the intermediate architecture. Despite the challenges, the use of a service-based architecture and containerization provides significant benefits. It helps to enhance the reproducibility and simplifies the deployment of simulation-based Digital Twins.

In further work, we aim to also include the physical part of the Digital Twin and implement the target architecture. This will allow us to assess whether additional research toolchains can be migrated to a similar architecture. We will evaluate the architecture in other domains to extract general good practices. We believe that good practices will aid experts in various domains when transferring toolchains to service-based architectures and later lifecycles.

Acknowledgments

This work was supported by DLR as part of the projects V&V4NGC and CCES-Pre.

References

[1] Nicolli S.R. Alves, Thiago S. Mendes, Manoel G. de Mendonça, Rodrigo O. Spínola, Forrest Shull, and Carolyn Seaman. 2016. Identification and management of technical debt: A systematic mapping study. *Inf. and Soft. Tech.* 70 (Feb. 2016), 100–121. doi:10.1016/j.infsof.2015.10.008

[2] ASAM. 2024. OpenDRIVE. https://www.asam.net/standards/detail/opendrive Accessed July 17, 2024.

[3] ASAM. 2024. OpenSCENARIO XML. https://www.asam.net/standards/detail/openscenario-xml Accessed July 17, 2024.

[4] Florian Auer, Valentina Lenarduzzi, Michael Felderer, and Davide Taibi. 2021. From monolithic systems to Microservices: An assessment framework. *Inf. and Soft. Tech.* 137 (2021), 106600. doi:10.1016/j.infsof.2021.106600

[5] Paris Avgeriou, Ipek Ozkaya, Alexander Chatzigeorgiou, Marcus Ciolkowski, Neil A. Ernst, Ronald J. Koontz, Eltjo Poort, and Forrest Shull. 2023. Technical Debt Management: The Road Ahead for Successful Software Delivery. In *ICSE-FoSE '23*. IEEE, 15–30. doi:10.1109/icse-fose59343.2023.00007

[6] Paris C. Avgeriou, Davide Taibi, Apostolos Ampatzoglou, Francesca Arcelli Fontana, Terese Besker, Alexander Chatzigeorgiou, Valentina Lenarduzzi, Antonio Martini, Athanasia Moschou, Ilaria Pigazzini, Nyyti Saarimaki, Darius Daniel Sas, Saulo Soares de Toledo, and Angeliki Agathi Tsintzira. 2021. An

Overview and Comparison of Technical Debt Measurement Tools. *IEEE Soft.* 38, 3 (May 2021), 61–71. doi:10.1109/ms.2020.3024958

[7] Yash Bajaj and Seema Shah. 2019. Role of Digital Twin in Manufacturing Motor Vehicles. *International Journal of Research in Engineering, Science and Management* 2, 10 (Oct. 2019), 288–290.

[8] Michelle Barker, Neil P. Chue Hong, Daniel S. Katz, Anna-Lena Lamprecht, Carlos Martinez-Ortiz, Fotis Psomopoulos, Jennifer Harrow, Leyla Jael Castro, Morane Gruenpeter, Paula Andrea Martinez, and Tom Honeyman. 2022. Introducing the FAIR Principles for research software. *Sci. Data* 9, 1 (Oct. 2022). doi:10.1038/s41597-022-01710-x

[9] Jan Steffen Becker, Tjark Koopmann, Birte Neurohr, Christian Neurohr, Lukas Westhofen, Boris Wirtz, Eckard Böde, and Werner Damm. 2022. *Simulation of abstract scenarios: towards automated tooling in criticality analysis.* Zenodo, 42–51.

[10] Terese Besker, Antonio Martini, and Jan Bosch. [n. d.]. The Pricey Bill of Technical Debt: When and by Whom will it be Paid?. In *ICSME '17*. IEEE, 13–23. doi:10.1109/icsme.2017.42

[11] Noel Crespi, Adam T Drobot, and Roberto Minerva. 2023. The Digital Twin: What and Why? In *The Digital Twin*. Springer, 3–20.

[12] Ward Cunningham. 1992. The WyCash portfolio management system. *ACM SIGPLAN OOPS Messenger* 4, 2 (Dec. 1992), 29–30. doi:10.1145/157710.157715

[13] Werner Damm, Stephanie Kemper, Eike Möhlmann, Thomas Peikenkamp, and Astrid Rakow. 2017. *Traffic Sequence Charts–From Visualization to Semantics.* techreport. SFB/TR.

[14] Shutong Deng, Liang Ling, Caizhi Zhang, Congbo Li, Tao Zeng, Kaiqing Zhang, and Gang Guo. 2023. A systematic review on the current research of digital twin in automotive application. *Internet of Things and Cyber-Physical Systems* 3 (2023), 180–191. doi:10.1016/j.iotcps.2023.04.004

[15] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. 2017. CARLA: An Open Urban Driving Simulator. In *RL (PMLR, Vol. 78)*, Sergey Levine, Vincent Vanhoucke, and Ken Goldberg (Eds.). 1–16.

[16] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. 2017. *Present and Ulterior Software Engineering.* Springer, Chapter Microservices: yesterday, today, and tomorrow, 195–216. doi:10.1007/978-3-319-67425-4_12

[17] Romina Eramo, Francis Bordeleau, Benoit Combemale, M. Brand, Manuel Wimmer, and Andreas Wortmann. 2021. Conceptualizing Digital Twins. *IEEE Software* PP (11 2021). doi:10.1109/MS.2021.3130755

[18] Markus Finke, Thomas Neff, and Tobias Reichl. [n. d.]. How to introduce TD Management into a Software Development Process – A Practical Approach. In *TechDebt '23*. IEEE, 52–61. doi:10.1109/techdebt59074.2023.00013

[19] Michael Grieves and John Vickers. 2017. *Digital Twin: Mitigating Unpredictable, Undesirable Emergent Behavior in Complex Systems.* Springer International Publishing, Cham, 85–113. doi:10.1007/978-3-319-38756-7_4

[20] Dominik Grundt, Anna Köhne, Ishan Saxena, Ralf Stemmer, Bernd Westphal, and Eike Möhlmann. 2022. Towards Runtime Monitoring of Complex System Requirements for Autonomous Driving Functions. *arXiv preprint arXiv:2209.14032* (2022).

[21] ISO. 2023. ISO/IEC 25010:2023. https://www.iso.org/standard/78176.html

[22] Arne Johanson and Wilhelm Hasselbring. 2018. Software Engineering for Computational Science: Past, Present, Future. *Comp. in Sci. & Eng.* 20, 2 (March 2018), 90–109. doi:10.1109/mcse.2018.021651343

[23] Vincent Kalwa. 2021. *TSC2OpenX–Realisierung einer Werkzeugkette zur Simulation abstrakter Verkehrsszenarien.* Bachelor Thesis. Carl von Ossietzky Universität Oldenburg. doi:10.13140/RG.2.2.11502.33605

[24] Philippe Kruchten, Robert L. Nord, and Ipek Ozkaya. 2012. Technical Debt: From Metaphor to Theory and Practice. *IEEE Soft.* 29, 6 (Nov. 2012), 18–21. doi:10.1109/ms.2012.167

[25] Valentina Lenarduzzi, Francesco Lomio, Nyyti Saarimäki, and Davide Taibi. 2020. Does migrating a monolithic system to microservices decrease the technical debt? *J. of Sys. and Soft.* 169 (2020), 110710. doi:10.1016/j.jss.2020.110710

[26] Yikun Li, Mohamed Soliman, and Paris Avgeriou. [n. d.]. Automatically Identifying Relations Between Self-Admitted Technical Debt Across Different Sources. In *TechDebt '23*. IEEE, 11–21. doi:10.1109/techdebt59074.2023.00008

[27] Zengyang Li, Peng Liang, and Paris Avgeriou. 2014. *Architectural Debt Management in Value-Oriented Architecting.* Elsevier, Chapter Architectural Debt Management in Value-Oriented Architecting, 183–204. doi:10.1016/b978-0-12-410464-8.00009-x

[28] Guozhi Liu, Bi Huang, Zhihong Liang, Minmin Qin, Hua Zhou, and Zhang Li. 2020. Microservices: architecture, container, and challenges. In *QRS-C '20*. IEEE, 629–635.

[29] Steven Macenski, Tully Foote, Brian Gerkey, Chris Lalancette, and William Woodall. 2022. Robot Operating System 2: Design, architecture, and uses in the wild. *Sc. Rob.* 7, 66 (2022), eabm6074. doi:10.1126/scirobotics.abm6074

[30] Manuel Mazzara, Nicola Dragoni, Antonio Bucchiarone, Alberto Giaretta, Stephan T. Larsen, and Schahram Dustdar. 2021. Microservices: Migration of a Mission Critical System. *IEEE Trans. on Serv. Comp.* 14, 5 (2021), 1464–1477. doi:10.1109/TSC.2018.2889087

[31] Paula Rachow and Matthias Riebisch. 2022. An architecture smell knowledge base for managing architecture technical debt. In *TechDebt '22 (TechDebt '22)*. ACM. doi:10.1145/3524843.3528092

[32] Riccardo Roveda, Francesca Arcelli Fontana, Ilaria Pigazzini, and Marco Zanoni. 2018. Towards an Architectural Debt Index. In *SEAA '18*, Vol. 27. IEEE, 408–416. doi:10.1109/seaa.2018.00073

[33] Tasneem Salah, M Jamal Zemerly, Chan Yeob Yeun, Mahmoud Al-Qutayri, and Yousof Al-Hammadi. 2017. Performance comparison between container-based and VM-based services. In *ICIN '17*. IEEE, 185–190.

[34] Diego Saraiva, José Neto, Uirá Kulesza, Guilherme Freitas, Rodrigo Reboucas, and Roberta Coelho. [n. d.]. Technical Debt Tools: A Systematic Mapping Study. In *ICEIS '21*. SCITEPRESS, 88–98. doi:10.5220/0010459100880098

[35] Mohammad Sadegh Sheikhaei and Yuan Tian. [n. d.]. Automated Self-Admitted Technical Debt Tracking at Commit-Level: A Language-independent Approach. In *TechDebt '23*. IEEE, 22–26. doi:10.1109/techdebt59074.2023.00009

[36] Noriyuki Shikata, Satoru Goto, and Kiminori Gemba. 2019. Servitisation of the manufacturing industry in Japan. *Forum Scientiae Oeconomia* 7, 3 (2019), 19 – 30. doi:10.23762/FSO_VOL7_NO3_2 Cited by: 5.

[37] Jianbo Tao, Yihao Li, Franz Wotawa, Hermann Felbinger, and Mihai Nica. 2019. On the Industrial Application of Combinatorial Testing for Autonomous Driving Functions. In *2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 234–240. doi:10.1109/ICSTW.2019.00058

[38] Adam Tornhill and Markus Borg. [n. d.]. Code red: the business impact of code quality - a quantitative study of 39 proprietary production codebases. In *TechDebt '22 (TechDebt '22)*. ACM, 11–20. doi:10.1145/3524843.3528091

[39] Roberto Verdecchia, Philippe Kruchten, and Patricia Lago. [n. d.]. Architectural Technical Debt: A Grounded Theory. In *ECSA '20*. Springer, 202–219. doi:10.1007/978-3-030-58923-3_14

[40] Arturo Villa, Jorge Octavio Ocharán-Hernández, Juan Carlos Pérez-Arriaga, and Xavier Limón. 2022. A Systematic Mapping Study on Technical Debt in Microservices. In *2022 10th Int. Conf. in Software Engineering Research and Innovation (CONISOFT)*. 182–191. doi:10.1109/CONISOFT55708.2022.00032

[41] Adam Wiggins. [n. d.]. The 12 Factor App. https://12factor.net/. [Accessed 23-01-2024].

[42] Gregory Wilder, Riley Miyamoto, Samuel Watson, Rick Kazman, and Anthony Peruma. [n. d.]. An Exploratory Study on the Occurrence of Self-Admitted Technical Debt in Android Apps. In *TechDebt '23*. IEEE, 1–10. doi:10.1109/techdebt59074.2023.00007

[43] Mark D. Wilkinson, Michel Dumontier, IJsbrand Jan Aalbersberg, Gabrielle Appleton, Myles Axton, Arie Baak, Niklas Blomberg, Jan-Willem Boiten, Luiz Bonino da Silva Santos, Philip E. Bourne, Jildau Bouwman, Anthony J. Brookes, Tim Clark, Mercè Crosas, Ingrid Dillo, Olivier Dumon, Scott Edmunds, Chris T. Evelo, Richard Finkers, Alejandra Gonzalez-Beltran, Alasdair J.G. Gray, Paul Groth, Carole Goble, Jeffrey S. Grethe, Jaap Heringa, Peter A.C 't Hoen, Rob Hooft, Tobias Kuhn, Ruben Kok, Joost Kok, Scott J. Lusher, Maryann E. Martone, Albert Mons, Abel L. Packer, Bengt Persson, Philippe Rocca-Serra, Marco Roos, Rene van Schaik, Susanna-Assunta Sansone, Erik Schultes, Thierry Sengstag, Ted Slater, George Strawn, Morris A. Swertz, Mark Thompson, Johan van der Lei, Erik van Mulligen, Jan Velterop, Andra Waagmeester, Peter Wittenburg, Katherine Wolstencroft, Jun Zhao, and Barend Mons. 2016. The FAIR Guiding Principles for scientific data management and stewardship. *Sci. Data* 3, 1 (March 2016). doi:10.1038/sdata.2016.18

[44] Zhongxiang Xiao, Inji Wijegunaratne, and Xinjian Qiang. 2016. Reflections on SOA and Microservices. In *ES '16*. IEEE, 60–67.