Containerized Development and Microservices for Self-Driving Vehicles: Experiences & Best Practices

Christian Berger[1], Björnborg Nguyen[2], and Ola Benderius[2]

***Abstract*— In this paper, experiences and best practices from using containerized software microservices for self-driving vehicles are shared. We applied the containerized software paradigm successfully to both the software development and deployment to turn our software architecture in the vehicles following the idea of microservices. Key enabling elements include *onboarding* of new developers, both researchers and students, traceable development and packaging, convenient and bare-bone deployment, and traceably archiving binary distributions of our quickly evolving software environment. In this paper, we share our experience from working one year with containerized development and deployment for our self-driving vehicles highlighting our reflections and application-specific shortcomings; our approach uses several components from the widely used Docker ecosystem, but the discussion in this paper generalizes these concepts. We conclude that the growingly complex automotive software systems in combination with their computational platforms should be rather understood as *datacenters on wheels* to design both, (a) the software development and deployment processes, and (b) the software architecture in such a way to enable continuous integration, continuous deployment, and continuous experimentation.**

I. INTRODUCTION

In the last decade, several large-scale demonstration events and competitions such as the 2007 DARPA Urban Challenge, the European ELROB event series, as well as the Grand Cooperative Driving Challenge facilitated the research and technology transfer to commercialize semi-automated systems, e.g., SAE level 2 and level 3. Reports about system and software architectures from teams participating in these challenges show similarities in terms of software component-oriented systems that focus on *pipes-and-filters*-based data processing [1], [2], [3], [4], [5], [6], [7], [8]. One of our previous studies [9] also unveiled similarities in terms of architectural design considerations.

Our recent research [10], [11] in our vehicle laboratory Revere, however, has shown that adopting a new software engineering paradigm based on containerized design principles unlocks advantages for accelerating software testing and to quickly onboard new developers. Such ideas, in turn, pave the road to even more recent ideas such as *DevOps* [12] or *microservices*. A microservice, considered as a specialization of the general service-oriented architecture (SOA) style, according to Fowler[1], is "an approach to designing software as a suite of small services, each running in its own process and communicating with lightweight mechanisms."

A. Research Goal

Our vision is to fundamentally restructure the software architecture powering autonomous systems and in particular self-driving vehicles to introduce microservices as the guiding design paradigm. Thereby, we want to completely decouple computational resources from the algorithms to dynamically restructure the data processing in a flexible and adaptive way to gain benefits such as enabling *continuous experimentation* for example.

The goal of this work is to summarize our experiences and best practices from introducing and adopting the technology enabling microservices for our *products* (the fleet of self-driving vehicles at various scales) as well as the *process* (the way of engineering the products by coordinating a fluctuating group of contributors).

B. Contributions

Our contribution is to share our experiences with the introduction and adoption of the microservice paradigm to our vehicle system development and to highlight the advantages and to point out potential pitfalls.

C. Structure

The rest of this article is structured as follows: Sec. II gives an overview of related work, Sec. III describes the overall microservices architecture that we have adopted for our vehicles, Sec. IV provides an overview of our containerized software development and deployment procedure, and our reflections about best practices and lessons learned are summarized in Sec. V.

II. RELATED WORK

In the development of software targeting the automotive industry, it is especially important to ensure the correctness of the software. In one study [13], the correctness of critical software when implemented as microservices was investigated, and how these services impact the computational performance. Furthermore, it is a great challenge to build such critical applications upon operating systems or kernels that might have integrated incorrect code, for example with a monolithic kernel such as Linux which has roughly 20 million lines of code. Therefore, it must be assumed that large systems

[1]Christian Berger is with the Department of Computer Science and Engineering, University of Gothenburg, Sweden christian.berger@gu.se

[2]Björnborg Nguyen and Ola Benderius are with the Department of Mechanics and Maritime Sciences, Chalmers University of Technology, Gothenburg, Sweden {bjornborg.nguyen,ola.benderius}@chalmers.se

[1]http://martinfowler.com/articles/microservices.html

978-1-5090-4793-2/17 $31.00 © 2017 IEEE
DOI 10.1109/ICSAW.2017.56

have vulnerable bugs, and it has been found that open-source software has in average 0.61 security-related defects per 1000 lines of code, and that commercial software may have a slightly higher such rate [14].

In one paper [15], microservices were used in automotive applications. The work describes a workflow and methodology targeting automotive projects in general, with the purpose of achieving higher efficiency and better code quality. The author identifies five key points of software development important to connected vehicles. Firstly, some components have higher resource demands or are more exploited thus the *scalability* of the software is critical for the efficiency. Furthermore, in a large complex software system, components most likely get outdated at a certain point and must, therefore, be able to be *exchanged* without disrupting the system as a whole. Additionally, projects with similar goals or requirements from previous work should be able to *reuse* existing framework or components. This implies that many software implementations will be used in many different projects thus *code quality* is essential on the basis of abstraction and generalization. Finally, a newly developed feature to the system must be available for the team so that the compatibility can be ensured through *continuous deployment*.

Our described workflow may be seen as one adapted form of the Rugby process model for continuous development and deployment. In one paper [16], the Rugby workflow was investigated and compared in different implementations of commercial and open source tools. Rugby highlights the continuous delivery aspect which makes the process model highly suitable for parallel project-driven organizations in addition promoting rapid releases and prototypes. The design of the Rugby workflow is to improve the communication between developers and users by using a feedback system while also improving the coordination among the developers through event based releases.

When using dedicated hardware in *Internet of things* (IoT) systems, the software may need to be statically deployed on specific nodes due to hardware constraints and may, therefore, be viewed as IoT gateways or edge cloud modules. One such solution is the Cloud4IoT [17], in which the higher layers may dynamically be deployed in any arbitrary node.

III. DESIGNING A MICROSERVICES-BASED ARCHITECTURE FOR SELF-DRIVING VEHICLES

Today's vehicles are becoming, similarly to many other IoTs, more automated, complex, and highly connected. They can be regarded as entities capable of sensing its environment and conveying processed data to the consumers, in particular, the driver as well other traffic participants. Besides being means of transportation, vehicles are now shouldering new functionalities and improvements to the consumer in both safety function as well comfort applications, thus facing new kinds of challenges and difficulties in the automotive industry.

A. Our Approach

To cope with changing user expectations, requirements, and the need to adjust software even after-sales, as seen in Tesla's Autopilot system, the vehicle software needs to be designed in a non-monolithic way and deployed in a flexible and traceable manner. Following Fowler's concept of microservices, we have designed our software system for the Revere vehicles in a strict microservices-oriented style as depicted in Fig. 1; the example depicts a schematic overview of our principle system architecture for our experimental vehicles.

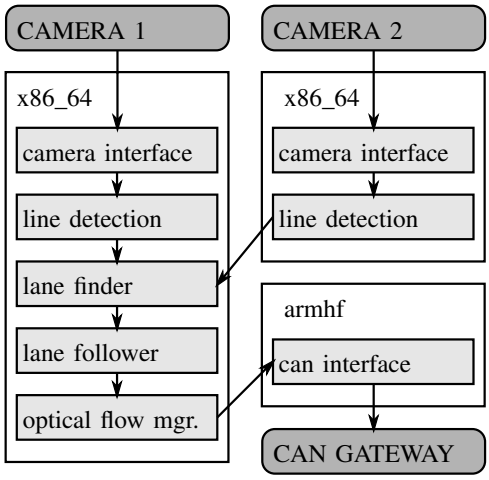

Fig. 1. An example of microservices (light gray) running on three nodes of two different platform types. Together, the services implement lane following using two cameras and a CAN gateway (dark gray).

In the example shown in Fig. 1, eight separate microservices are shown targeting different aspects from the *pipes-and-filters* data processing chain: Three modules directly interface with the respective hardware–software interfaces (camera and CAN) while the other five modules simply process the incoming data flow and emit new messages. The design of the system is partially time-triggered (camera and CAN) to query or serve the various interfaces to the hardware to ensure interfacing with the actuators in the prescribed schedule or to grab images from the cameras in the desired frame rate, but predominantly data-triggered to let the components react on incoming messages.

B. Design Decisions

Following Fowler's description, our individual components are separate processes running as Docker containers and communicating using our realtime-capable middleware OpenDaVINCI[2] for publish–subscribe communication. At first glance, we simply have a distributed system; however, we have adopted several deployment concepts from the Docker ecosystem to simplify the execution of the set of microservices.

Formalizing self-contained software bundles: The resulting software bundles as described in Sec. IV contain the executable binaries next to all library dependencies. However, all our individual microservices are highly configurable to provide, for example, information what CAN interface or

[2] `code.opendavinci.org`

camera to connect to, thresholds for detection algorithms, and so forth. Instead of maintaining separate configuration files per microservice, we have a centralized *dynamic module configuration protocol* inspired by DHCP that is providing the right subset of configuration settings from a central life-cycle management microservice during a microservice's startup phase. Using this approach, we just need to maintain one central configuration valid for all microservices distributed over a set of nodes.

As these settings alongside with, for instance, camera calibration matrices are typically not part of a Docker image as they might change very frequently. Furthermore, creating a Docker container from a Docker image typically involves many parameters such as definition of (virtual) network access, specification of using inter-process communication (IPC), and the mapping of volumes. As these settings are error-prone and time-consuming, we have adopted `docker-compose` to help building, starting, monitoring, and stopping our microservices. This formalization also simplified the use of our software environment for various use cases, for instance just recording data from the sensors or defining a specific demonstration show case[3].

Publish–subscribe message exchange using broadcast communication: To enable a truly seamless migration of microservices among nodes requires a low coupling with concrete other microservices in terms of *named* communication links such as TCP where the sender and receiver know each other. Therefore, publish–subscribe where microservices can subscribe to a set of message identifiers they are interested in without knowing their producers enables decoupling in terms of startup order and deployment node. To lower the network utilization, we are using UDP multicast to broadcast messages in our vehicle networks.

Standardization of message sets: Another design goal for our software environment is to enable reuse and exploit synergies among the different vehicle projects. Therefore, we gather all hardware–software interfaces in an open source Git repository[4] to encourage adoption from our different research projects. As we are supporting different vendors for the various sensors like Velodyne and SICK for lidar, or Axis and Logitech for cameras, we aim to avoid propagating sensor-specific information and properties to the data processing algorithms to also preserve reusability. However, this design decision requires an agreement to a standardized message set as the *greatest common divisor* for a specific category of sensors. Images are easy to standardize; for our lidar devices or position sensors, we defined generic abstractions to facilitate the design and implementation of algorithms that could cope with, for instance, a 16 or 32 layer lidar similarly as with a single layer laser scanner.

[3]`https://github.com/chalmers-revere/opendlv.core/tree/master/usecases`

[4]`https://github.com/chalmers-revere/opendlv.core`

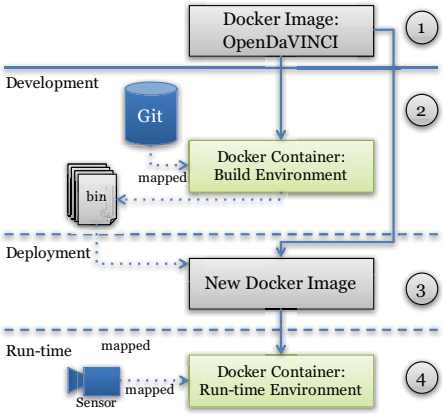

Fig. 2. A unifying software engineering process for a microservices-enabled architecture (based on [11]).

IV. CONTAINERIZING SOFTWARE ENGINEERING AND INTEGRATION PROCESSES

The microservices architecture outlined in the previous section enables a seamless transition of individual microservices between the computational nodes; however, a given set of services at a certain point in time addresses only one concrete snapshot of a successfully deployed and started set of interacting software components. Complementary to the *product*-focus, the way how the complex software product is created, packaged, and deployed is similarly important focusing on the actual *process*.

A. Our Approach

Software-centric and software-driven industries like large social media providers have adopted containerization, i.e., the lean isolation of system resources among processes in combination with bundling all dependencies of a running process like third party libraries into a self-contained deployable artifact. We have adopted these principles as depicted in Fig. 2 using components from the widely used Docker ecosystem.

Fig. 2 depicts the way how we have incorporated containerization for our build and deployment process on the example of our software components, which interface with hardware like sensors. These components are based on our real-time capable middleware OpenDaVINCI, which allows independently acting data- or time-triggered software components exchanging data following the publish–subscribe pattern. The general idea is that every developer gets a build environment having everything included: We provide everything that is needed for a new team member to start building our open source software environment (referred to as step 1 in the figure). Therefore, the sources for the hardware–software interfaces in our example are mapped into a Docker-container based on the Docker-image (here: OpenDaVINCI) containing all prerequisites for building (step 2). The resulting artifacts from a build are stored outside of an encapsulating container for persistence. Once a build succeeded, the newly created binaries are added to the existing Docker image

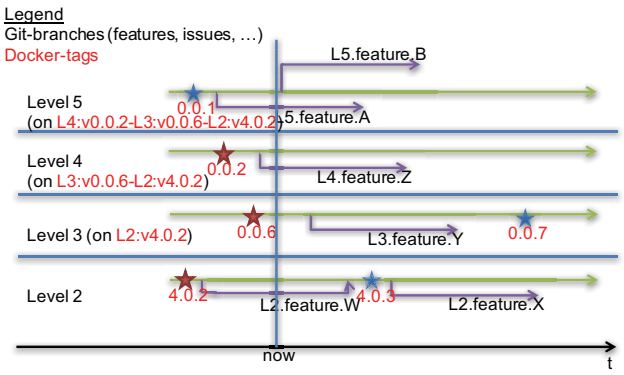

Fig. 3. Ensuring traceability and decoupling between our various stacked software layers using OpenDaVINCI and OpenDLV.

resulting in a new layer (step 3). This image is subsequently used to execute the software components comprising our microservices.

We combined the aforementioned workflow with the state-of-the-art, Git-based feature-branches and pull–approve distributed-development concept. Whenever a new set of features has been successfully integrated, we add a unique tag for traceability on Git that we simultaneously propagate to the Docker image containing these binaries. Thereby, we achieve traceability between a certain version of sources and deployable software bundles to be executed. Fig. 3 details our strategy for cross-layer traceability and preservation of decoupling between the separate software layers; for instance, level 2 contains our middleware OpenDaVINCI, level 3 contains the hardware–software interfaces in OpenDLV.core, level 4 provides our OpenDLV driving logic, and level 5 encapsulates all software that is bound to non-disclosure agreements. For example, level 3 uses the software bundle from its layer below labeled `v4.0.2`, which refers both, to a tag on Git and simultaneously to the same identifier on the corresponding Docker images. Thereby, level 2 can evolve (in our example to v4.0.3) and introduce new features or adjusting some interfaces without immediately breaking the layers above. The same strategy is applied to level 4 that is using level 3 Docker images to build its sources: here, the labels from level 3 and level 2 are combined to uniquely identify what software bundle was used to create the binaries in this layer.

B. Design Decisions

The outlined strategy summarizes best practices for software engineering and deployment that are adopted and established in software-centric companies. For vehicular systems, though, further aspects need to be considered:

Separation of concerns: The workflow depicted in Fig. 2 captures one specific aspect of our software environment (hardware–software interfaces). These workflows, however, can also be stacked to separate various perspectives from each other; we are typically using at least five layers, where the first three are all open source: (1) basic middleware OpenDaVINCI, (2) OpenDLV.core providing all low-level hardware–software interfaces, (3) OpenDLV providing reusable self-driving algorithms, (4) project-specific hardware–software bindings that are sometimes covered by non-disclosure agreements (NDA), and (5) project-specific algorithms. While these layers preserve intellectual property, they mainly facilitate a decoupled development within the various layers: As one layer sitting on top of another layer must explicitly specify a concrete label from the layer below that is needed to build its own sources, the underlying layer can safely evolve over time and even release new versions as the higher layer simply sticks to a certain version and can deliberately decide when to move to a newer version for its lower layer.

Heterogeneous hardware platforms: In contrast to data centers where homogeneous platforms are used to allow simple replacements in case of hardware failures, our vehicles are powered by, for example, x86_64 and armhf platforms used for various purposes. To avoid that individual developers have to deal with hardware- or cross-compiler specifics, our stacked software layers simply depend on the compilers provided in the layer directly below the current layer. At the lowest layer, we provide two separate Docker images, where one is targeting x86_64 and the other armhf. All higher layers sitting on top of one of these two are automatically, using the same compiler, transparently creating the binaries for the correct platforms.

Dependencies to certain hardware: Our ultimate goal is to let our microservices transparently and seamlessly migrate between our computing nodes. This would allow a resource utilization-dependent allocation of a concrete microservice to a given node. However, contrary to data centers where such microservices can travel freely, computing nodes on our vehicles have certain capabilities that are needed by certain microservices. Therefore, a runtime deployment model needs to obey such dependencies to ensure proper functionality of the entire software system.

V. Lessons Learned & Best Practices

In the following, we are summarizing our reflections about experiences, lessons learned, and best practices.

A. Microservices Architecture

Stateless components: To let software components seamlessly start and migrate between computing nodes, they must exhibit a stateless behavior in terms of stopping and restarting. For instance, a lane-detection component should simply provide its extracted features to downstream microservices subscribing to its messages. However, sensor fusion algorithms typically improve their output over time by, e.g., using prediction models to realize a tracking system. As this seems to be in conflict with demanding stateless components, we suggest the use of *Quality-of-Service* annotations to messages to indicate the quality of the contained data.

Publish–subscribe communication: As described in Sec. III, our microservices rely on UDP multicast to lower the network utilization and to achieve broadcasting of messages and to avoid explicitly defining *a-priori known* communication

partners as in unicast communication. This strategy helps to transparently add more computing nodes to a vehicle cluster and also supports microservices to seamlessly migrate among them.

Current limitations: Our software development and deployment strategy, as well as the microservices architecture, heavily relies on Docker and its associated tools. As this technology is continuously growing, our laboratory also benefits from new features such as load balancing or bugfixes, for example. However, we also encounter a few limitations that need to be addressed either by finding a workaround, which would potentially violate some design decisions behind the Docker ecosystem or by adding the required functionality to Docker. For instance, to truly turn different computing nodes into a virtual computation node by using `docker-swarm`, a dedicated overlay network is required that belongs to the specified set of microservices. This design constraint makes sense as several microservices can coexist in parallel without interfering with each other. However, at the time of writing, the default Docker overlay network driver does not support UDP multicast[5] and third party libraries would be needed. As an alternative, the network for the set of microservices could be shared between the Docker containers and the host system using `-net=host` to enable UDP multicast. However, this workaround does not work with `docker-swarm` in combination with `docker-compose` where the use of `-net=host` is not possible and virtual networks as provided by, for example, the overlay network driver needs to be used.

Furthermore, our `docker-compose` use cases employ the feature of dynamically building Docker images during the startup phase of a set of microservices. Thereby, we can add further files to the Docker image in use before starting the microservices that base on that image. However, `docker-swarm` in combination with `docker-compose` does not provide this feature and all Docker images must be present before starting the set of microservices. This strategy is understandable from the `docker-swarm` perspective to simply start and deploy a set of microservices based on a self-contained description. However, as a consequence, we would need to adapt our current strategy of using `docker-compose`.

The aforementioned topic is also related to a problem when certain microservices use data volumes mapped from a host into a running Docker container. This concept obviously violates the idea behind a `docker-swarm` where microservices can seamlessly migrate between physical nodes. Therefore, a microservices-based architecture for the data processing must be accompanied by a design and strategy for how to provide and store data from microservices that migrate dynamically. In this regard, established concepts like *network file system* (NFS) could be used but a proper configuration of the nodes in the cluster is necessary. A different concept to be used would be Flocker[6] that tries to let data volumes follow Docker containers when they migrate. However, in the context of self-driving vehicles where a lot of data in the range of up to 1GB/s[7] needs to be handled in realtime, the performance of such approaches needs to be further explored.

B. Containerized Software Development and Deployment Process

Deployability: As the automotive industry is increasingly emphasizing software engineering principles and techniques, it is utterly important to let design criteria regarding the engineering processes for software products be guided by deployability: How can a change in the sources be quickly and traceably bundled *and* delivered to the target platform?

Automation: As soon as deployability is in place, the deployment procedure itself must be optimized to avoid any manual interaction and focused on speed. A high degree of automation ensures repeatable software builds and quick onboarding of new developers. Furthermore, a highly automated process can be adjusted when, for instance, new compilers and libraries are required.

Traceability: While traceability for software sources is mandatory for any serious software development, the combined traceability between sources and binaries, as well as across layers is necessary to enable both, (a) decoupled development within one layer, and (b) accelerated locating of potential software bugs between two versions, where the older was working while the newer might not. The tight coupling between labeled binaries that can be quickly deployed to the target platform for validation helps to locate and isolate any changes to the software that unknowingly introduced the unexpected behavior.

Current limitations: Our software development and deployment process heavily depends on layered Docker images for our stacked software system. By design, we allow the various layers to change independently and our automated weekly release server is re-creating all Docker images from the current master branches. Furthermore, the fundamental base image, which is currently Ubuntu 16.04 LTS, that we use as the basis for our lowest layer, is updated weekly to include all security patches. For clients using our Docker images, this results in pulling regularly large amounts of data to get the newest version. Furthermore, when we are testing selected layers from our software system at the proving ground, we are constantly creating delta Docker images that are pushed and pulled to our private Docker registry. We noticed that it is important to accelerate the deployment procedure on site and therefore, having a private Docker registry as part of the vehicle computers is recommended.

VI. CONCLUSION AND FUTURE WORK

From the recent developments connected with autonomous and connected vehicles, it is clear that there is an increased need for formal and efficient software development and deployment strategies. As we observe in various demonstration cases from many major automotive OEMs, self-driving

[5] `https://github.com/vpac-innovations/rsa/issues/40`

[6] `https://clusterhq.com/flocker/introduction/`

[7] `http://www.kurzweilai.net/googles-self-driving-car-gathers-nearly-1-gbsec`

vehicles will potentially become similar to data centers on wheels. However, in contrast to real data centers that are static, specialized, and can be maintained with different strategies compared to vehicles, where the set of computing nodes is fixed and cannot be easily changed. Thus, specific strategies aiming at an automotive microservices architecture are needed.

In this paper, we present our approach and experiences from using containerized software development and deployment for our self-driving vehicles in our Revere laboratory. Our strategy simplified the way to onboard new developers to let them contribute to our software by providing ready-to-use images to build the system that contain all dependencies. Furthermore, we outlined our strategy to ensure traceability between sources and deployable software bundles while preserving decoupling between the various layers in our software system.

We also provide our design decisions for a microservices-based software architecture that we are using on our self-driving vehicles. As key-enabling design drivers we list publish–subscribe communication and standardized messages to preserve dynamic reallocation of messages to different computing nodes at runtime. We summarize our lessons learned with using Docker for more than one year in our laboratory and point out some potential pitfalls to be considered when adopting containerized software development and deployment and microservices architectures.

Our future work entail further analysis of the specific realtime and data load requirements from adding more sensors and computing nodes to our Revere vehicles. In this regard, it is important to consider both, the online data processing, but also addressing a proper way to deal with data post-processing using microservices. Furthermore, runtime adaptability of the software stack, as well as service migration for safety-critical services that must follow a certain update frequency, needs to be further investigated.

ACKNOWLEDGMENT

The authors are grateful to Chalmers University of Technology for supporting the research. The Chalmers Revere laboratory is supported by Chalmers University of Technology, AB Volvo, Volvo Cars, and Region Västra Götaland.

This work has been supported by the COPPLAR (CampusShuttle cooperative perception and planning platform) project, funded by Vinnova FFI, Diarienr: 2015-04849.

REFERENCES

[1] J. Rojo, R. Rojas, K. Gunnarsson, M. Simon, F. Wiesel, F. Ruff, L. Wolter, F. Zilly, N. Santrač, T. Ganjineh, A. Sarkohi, F. Ulbrich, D. Latotzky, B. Jankovic, G. Hohl, T. Wisspeintner, S. May, K. Pervölz, W. Nowak, F. Maurelli, and D. Dröschel, "Spirit of Berlin: An Autonomous Car for the DARPA Urban Challenge - Hardware and Software Architecture," Freie Universität Berlin, Berlin, Germany, Tech. Rep., 2007.

[2] M. Montemerlo, J. Becker, S. Bhat, H. Dahlkamp, D. Dolgov, S. Ettinger, D. Haehnel, T. Hilden, G. Hoffmann, B. Huhnke, D. Johnston, S. Klumpp, D. Langer, A. Levandowski, J. Levinson, J. Marcil, D. Orenstein, J. Paefgen, I. Penny, A. Petrovskaya, M. Pflueger, G. Stanek, D. Stavens, A. Vogt, S. Thrun, S. Artificial, S. Cs, and D. Hähnel, "Junior: The Stanford Entry in the Urban Challenge," in *The DARPA Urban Challenge*, no. October 2005, 2009, pp. 91–123. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-03991-1_3

[3] A. L. Kornhauser, A. Atreya, B. Cattle, S. Momen, B. Collins, A. Downey, G. Franken, J. Glass, Z. Glass, J. Herbach, A. Saxe, I. Ashwash, C. Baldassano, W. Hu, U. Javed, J. Mayer, D. Benjamin, L. Gorman, and D. Yu, "DARPA Urban Challenge - Princeton University - Technical Paper," Department of Operations Research and Financial Engineering, Princeton University, Princeton, NJ, USA, Tech. Rep., 2007.

[4] S. Kammel, J. Ziegler, B. Pitzer, M. Werling, T. Gindele, D. Jagszent, J. Schröder, M. Thuy, M. Goebl, F. von Hundelshausen, O. Pink, C. Frese, and C. Stiller, "Team AnnieWAY's autonomous system for the 2007 DARPA Urban Challenge," *J. Field Robotics*, vol. 25, no. 9, pp. 615–639, 2008. [Online]. Available: http://dx.doi.org/10.1002/rob.20252

[5] J. Hurdus, A. Bacha, C. Bauman, S. Cacciola, R. Faruque, P. King, and C. Terwelp, "VictorTango Architecture for Autonomous Navigation in the DARPA Urban Challenge," *Journal of Aerospace Computing, Information, and Communication*, vol. 5, no. 12, pp. 506–529, 2008. [Online]. Available: http://dx.doi.org/10.2514/1.40547

[6] Y.-L. Chen, V. Sundareswaran, C. Anderson, A. Broggi, P. Grisleri, P. P. Porta, P. Zani, and J. Beck, "TerraMax: Team Oshkosh Urban Robot," in *The DARPA Urban Challenge*, 2009, pp. 595–622. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-03991-1_14

[7] A. Bacha, C. Bauman, R. Faruque, M. Fleming, C. Terwelp, C. Reinholtz, D. Hong, A. Wicks, T. Alberi, D. Anderson, S. Cacciola, P. Currier, A. Dalton, J. Farmer, J. Hurdus, S. Kimmel, P. King, A. Taylor, D. V. Covern, and M. Webster, "Odin: Team VictorTango's Entry in the DARPA Urban Challenge," *Journal of Field Robotics*, vol. 25, no. 9, pp. 467–492, Sept. 2008.

[8] C. Berger and B. Rumpe, "Autonomous Driving - 5 Years after the Urban Challenge: The Anticipatory Vehicle as a Cyber-Physical System," in *Proceedings of the INFORMATIK 2012*, U. Goltz, M. Magnor, H.-J. Appelrath, H. K. Matthies, W.-T. Balke, and L. Wolf, Eds., Braunschweig, Germany, Sept. 2012, pp. 789–798. [Online]. Available: https://arxiv.org/pdf/1409.0413v1.pdf

[9] C. Berger and M. Dukaczewski, "Comparison of Architectural Design Decisions for Resource-Constrained Self-Driving Cars - A Multiple Case-Study," in *Proceedings of the INFORMATIK 2014*, E. Plödereder, L. Grunske, E. Schneider, and D. Ull, Eds. Stuttgart, Germany: Gesellschaft für Informatik e.V. (GI), Sept. 2014, pp. 2157–2168. [Online]. Available: http://subs.emis.de/LNI/Proceedings/Proceedings232/2157.pdf

[10] C. Berger, "Testing Continuous Deployment with Lightweight Multi-Platform Throw-Away Containers," in *Proceedings of the 41th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA)*, K.-E. Großpietsch and K. Kloeckner, Eds., Funchal, Madeira, Portugal, Aug. 2015.

[11] C. Berger and O. Benderius, "Encapsulated Software Build Environments on the Example of a Self-Driving Heavy Vehicle," in *Proceedings of the 42nd EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA)*, Sept. 2016.

[12] F. Giaimo, H. Yin, C. Berger, and I. Crnkovic, "Continuous Experimentation on Cyber-Physical Systems: Challenges and Opportunities," in *Proceedings of the Scientific Workshop Proceedings of XP2016*. Edinburgh, Scotland UK: ACM, May 2016. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2962695.2962709

[13] C. Fetzer, "Building Critical Applications Using Microservices," *IEEE Security & Privacy*, vol. 14, no. 6, pp. 86–89, 2016.

[14] Synopsys. (2015) Open source report 2014. [Online]. Available: http://go.coverity.com/rs/157-LQW-289/images/2014-Coverity-Scan-Report.pdf

[15] T. Schneider and A. Wolfsmantel, "Achieving Cloud Scalability with Microservices and DevOps in the Connected Car Domain," in *Software Engineering (Workshops)*, 2016, pp. 138–141.

[16] S. Taheritanjani, S. Krusche, and B. Bruegge, "Experience report: A comparison between commercial and open source reference implementations for the rugby process model." in *Software Engineering (Workshops)*, 2016, pp. 148–155.

[17] D. Pizzolli, G. Cossu, D. Santoro, L. Capra, C. Dupont, D. Charalampos, F. De Pellegrini, F. Antonelli, and S. Cretti, "Cloud4IoT: A Heterogeneous, Distributed and Autonomic Cloud Platform for the IoT," in *2016 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 2016, pp. 476–479.