**Adapting a Container Infrastructure for Autonomous Vehicle Development**

Yujing Wang

Department of Mechanical and Mechatronics Engineering

University of Waterloo

Waterloo, ON, Canada

yj9wang@edu.uwaterloo.ca

**Abstract**

In the field of Autonomous Vehicle (AV) development, having a robust yet flexible infrastructure enables code to be continuously integrated and deployed, which in turn accelerates the rapid prototyping process. The platform agnostic and scalable container infrastructure, often exploited by developers in the cloud domain, presents a viable solution addressing this need in AV development. Developers use tools such as Docker to setup containers and Kubernetes to orchestrate the container network. Depending on the needs, developers may also apply other third-party add-ons to enrich the container network. This paper presents a container infrastructure strategy for AV development. It then discusses the scenarios in which this strategy is useful and perform an analysis on resource and time overheads when using containers, and their impact on a Mix Critical System (MCS). This paper then proceeds to discuss an experiment conducted to compare the operation runtime and layering runtime of running on native machine, in new containers, in existing containers, and in nested containers. With these concerns in mind, the developers may be more informed when setting up the container infrastructure, and take full advantage of the new infrastructure while avoiding some common pitfalls.

**Introduction**

Agile, A new form of software development has been quickly winning cloud developers’ favor over the traditional waterfall model. In agile practice, Software developers write code, run it through CI/CD pipelines, integrate daily and deploy as soon as a new version can be released [1]. This gives developers chances to test newly developed prototypes in all kinds of scenarios much more frequently than the traditional waterfall model. As a result, more iterations can be performed, and more bugs can be discovered in the meantime; thus, increasing both the quality and speed of the software. Applying agile practice to AV development is much more challenging, however, software technologies used on an AV often come from a wide range of temporal criticality: from low-level safety-critical mechanical control, to embedded real-time system to high level training of perception model, as well as mid-level network applications. Developers often fear mess up the temporal separation in an MCS, and back away from implementing a unified infrastructure strategy. Furthermore, Industry’s practice of modularizing teams into specific functional unit also enhances the status quo mindset where things should be done as it is. As a result, each team must perform repetitive adaptation processes for each vehicle during each iteration. This greatly slows down the development and testing feedback loop.

A well architected container infrastructure removes these overheads and allows developers to build once and run on any platform. The accelerated build and test cycle makes continuous delivery of new features in response to ever-changing requirement possible [1]. The idea of a container first came from Linux name space which makes isolating resource set in each namespace for each running task possible [2]. Docker later came out to streamline the resource isolation process. Docker packages all software dependencies and running mechanism into an isolated environment called “image”. Each software dependency or each step of running mechanism is a layer in the image [3]. To update the image, docker updates the corresponding layer without making modification to the rest of the image. Then such image is deployed and ran in containers independent from other containers and underlying host machine. Docker daemon supplies the needed configuration from host environment to each container and stores no data inside the containers. This saves user from having to deploy large resource overhead needed to virtualize an entire OS as needed by Virtual machine infrastructure (Figure 1).

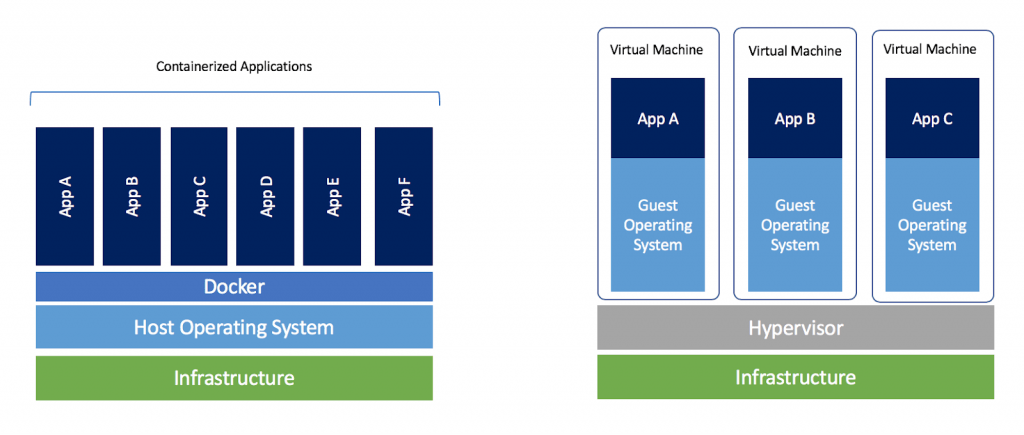


Figure 1 Containers vs Virtual Machine Infrastructure [4]

As the number of containers increases, the need for establishing an efficient container network becomes crucial especially for AV where the storage and computing power is highly constrained. Kubernetes is an open source container orchestration tool, that lets user manage a network of containers [2]. Kubernetes reads declarative configurations from yaml files, where the user specifies a desired state, to which Kubernetes will work its way to achieve such state [5]. Kubernetes automate the tedious process of spawning, updating and healing any number of containers. Moreover, it lets user provision CPU and memory resources for a specific task. The Kubernetes network is formed by a master node and many worker nodes [6]. The master node accepts command from the user, stores configuration, schedule pods and then realize actions by sending signals to worker nodes. Worker nodes connect to master through Kube-proxy and accepts the signal through Kubelet which performs the action.

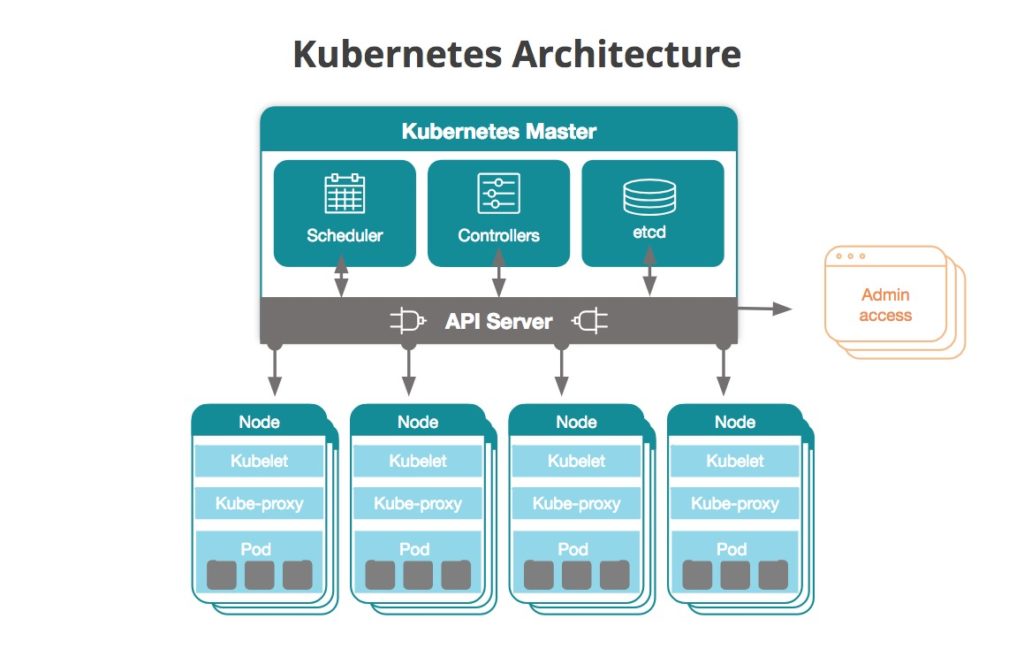


Figure 2 Architecture of Kubernetes [6]

This paper will present the scenarios in which a container infrastructure benefits AV development, examine the resource and time overhead each container layer adds.

**Idea and Motivation**

In an AV, complex tasks such as lane changing, parking, and merging/yielding actions relies on a line of agents operating on data: data are collected, analyzed and according to which actions are executed. Most actions are performed by local agents, some are performed by remote agents hosted on the cloud and connect to each vehicle via internet. By utilizing cloud computing, vehicle can perform much more powerful data analyzation since local resource is limited. This kind of local and remote agents mix makes up a Cyber Physical System (CPS) [7]. Figure 3 shows the data processing line:

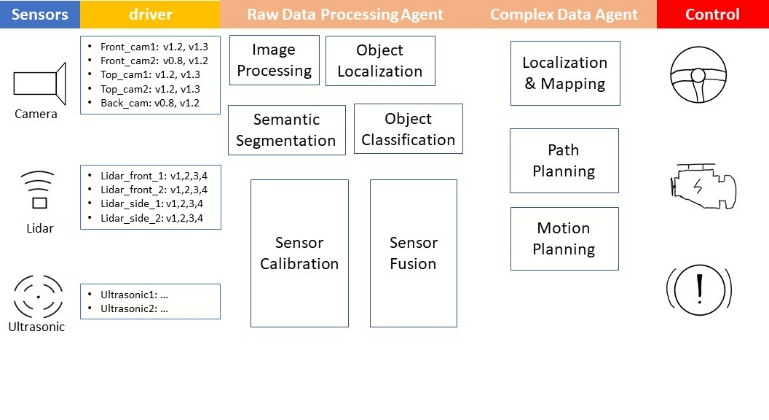


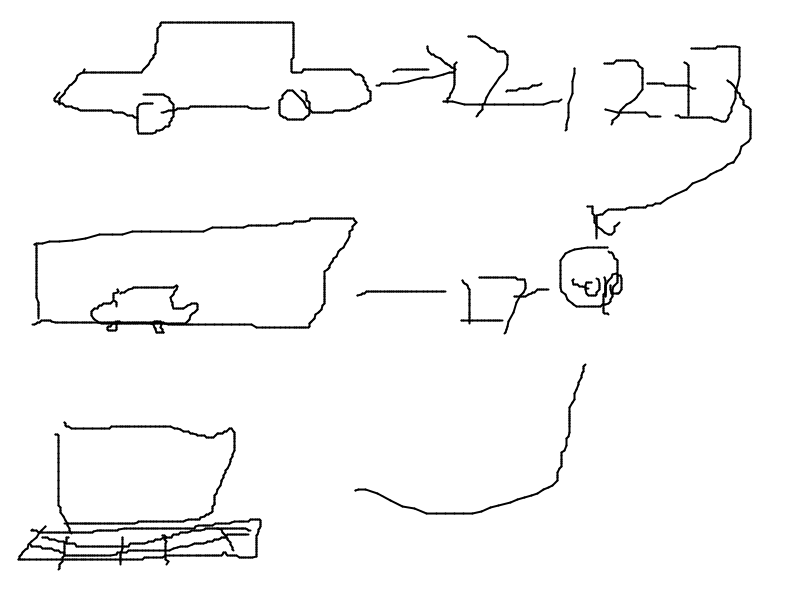
Figure 3 Line of Agents Operating on Data

Located at the very upstream of the pipeline are the data collectors. These are sensors such as Lidars, Cameras. On top of each sensors are their respective drivers. There may be multiple sensors of the same type mounted on the car, that are highly similar but not of the exact same hardware. In the case of camera for example, the front\_cam1 used for traffic observation may be using a different driver than back\_cam used for rear end approach check. Having to manage different versions and variation of camera drivers is tedious, especially when one needs to perform A/B split testing to see which version performs better []. In a container infrastructure, the user packages each revisions and versions in Docker images, then specify `sensor\_type:variation\_version`. For example, one may name the front left camera driver’s version 1.2 as: `camera\_driver:frontLeft\_v1.1`. Using Kubernetes’ replicable and self-healable deployment object, one can write a helm template manifest (via built in Helm Templating Engine) for shelving the sensor driver containers [7]. Then specify the correlation between physical sensors and containers in a key-value file.

Similarly, data processing agents down the data pipeline can be broken down in a similar structure using the aforementioned strategy. Each agent stands alone in one container. It receives input from upstream agents, process it, and subsequently sends results to downstream agents.

The reverse is also true: the same functional units can be packaged and employed in different scenarios. This enables application that handles one specific task to be repetitively deployed on different devices when a change in environment only affected its upstream or downstream agents. For example, running simulation is a very common practice to train vehicle’s perception module. The same perception module is coupled with different agents in the following scenarios (Figure 4) [8]:

1. Running the vehicle on real world roads
2. Running the vehicle in a simulated environment
3. Running perception core on a computer simulated model.

Figure 4

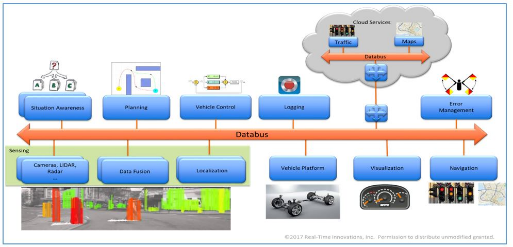
In different scenarios different setups are involved but the same perception logic remains in

1. a full fledge vehicle is used.
2. only the camera hardware involving perception logic.
3. no hardware set up is involved.

To maintain testing consistency across different testing setups, one perception model would be deployed repetitively in all scenarios.

**Resource Overhead and Boundary for Containers**

Overhead is the pain point of using containers. Virtualizing applications in containers inevitably consumes extra system resources and takes longer run-time and causes communication delay crossing containers’ boundary. In cloud computing, the limitations on memory and computation resource is close to negligible: developers can add more machines and disks to the container network and scale the cluster horizontally to accommodate for increased usage. In AV development, the physical space on the vehicle for housing machines are limited. Fog Architecture proposes a way to best utilize cloud computing’s power and accommodate for the limited space on vehicle by having vehicles upstreaming resource intensive computation logic to the cloud [9]. Modules on the cloud will enable vehicles to navigate through more complex situations such as driving in a chaotic urban environment where pedestrian and cars may cross path at irregular intervals and random locations outside the bounds of general traffic rules. The vehicle itself, on the other hand, hosts a complete ecosystem of data processing agent to navigate through places where network connections are weak to non-existent and the traffic is more predictable such as driving on a highway in countryside. The architecture of such infrastructure is a specific application of case 2 single model multi scenarios. Each vehicle joins the container network as less powerful nodes. Each module is deployed repetitively in each vehicle nodes and cloud nodes. Container flavor manager manages which tag of the image will be deployed given the specs of the local node. Though not the focus of this paper, containers infrastructure allows AVs to tap into the computing power of cloud machines and henceforth circumventing partially the limitation on physical space constraints.



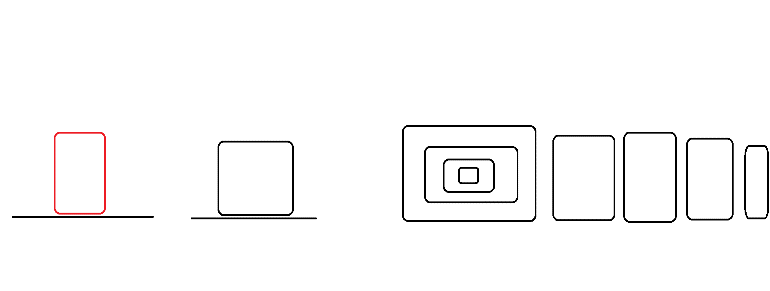
**Overhead Analysis and Real Time Scheduling Analysis**

Even more limited than resources are the response time in a time critical system. Signals traversing in container networks need to first exit its originating container and enter its destination container, crossing at least 2 layers of delay per container involved. The more containers involved the more layers signals need to cross and higher the delay stacks. The accumulation of delay is worsening when there are containers inside of containers or when signals need to traverse through multiple intermediate containers. Compounding layers of layers of boundary communication gives a significant delay in the signal relay process. The compounding delay also adds greater uncertainty in communication time thus breaking the deterministic trait in a real time system. [10] performed a runtime analysis on the temporal criticality of each container’s internal runtime in 4 different environments: ubuntu vanilla-native, vanilla-docker, RT-native and RT-docker. The result showed that runtime in Docker is approximately the same as running natively. Real-time enabled Linux Kernel performs more deterministically than Vanilla, which is overall faster in both empty and loaded context. [10] did not study the effect of crossing the container itself, which we intend to investigate in the following experiment to understand how container network should be orchestrated to avoid breaking deterministic traits.

**Experimental Setup**

To study the communication delay across containers, we decided to perform an experiment to see how container layers affects communication time. 4 scenarios are tested and juxtaposed:

* Running on native machine
* Running in newly initialized container
* Running in existing container
* Running in nested containers



In this experiment, we use an algorithm that perform iterative Gaussian Seidel Approximation on a strictly diagonally dominant matrix [11]. For any Matrix operation in the form

Ax = b

Where x is an Unknown vector, A is a known strictly diagonal matrix, and b is a known m x 1 vector. Decomposing Matrix A into lower triangular matrix L\* and strictly upper part U such that L\*+U = A, we get:

Isolating for one x and using forward substation:

This gives us an iterative solution:

This operation performs an iterative solution to calculate result. This effectively converts ordinary matrix solution with uncertain runtime to O(m) runtime during each iteration and storing O(m) variables in memory space. Knowing the size of the matrix, we have a constant runtime and constant memory, which are convenient for measurement purpose. Using an initial guess of 0,0,0,0,0, A total of 2500 iteration is performed for each call, and the output is the calculated result. For each call, it reads the default value and calling time as input, perform the calculation. Upon completion, output the result, and critical timestamps into a persistent storage hosted locally on the machine, then sends back the result, which is used as an initial guess in the next call. For each of the scenarios a total of 100 calls are performed. In addition, “run in newly initialized container”a single run is executed by running the function directly. Accessing existing container is achieved by wrapping the function inside a flask application that remains running. Each call is sent to the application without having to initialize the application. In our calculation the value used is: A, b and answer x are:

,

Only computations that output correct solution will be used in the overhead analysis. To determine the correctness each run, a 5th degree T-test on 99.95% confidence level is used based on formula 1, where is the answer being sampled, the mean is the “ground truth” calculated in II, n is the sample size, and   is standard deviation of sample results [12].

It is however noted that since each call perform 2500 iterations, it is very unlikely to have the result not fall into the 95% confidence level in standard deviation calculated in the in results of all input answers. In addition, since the result of previous stage is feed into the next stage, the accuracy is likely to increase over time.

In the first 2 scenarios, the function is exposed using “\_\_main\_\_.py”. When “python gausse.py” is executed, an instance is initiated either inside or out of the container. As soon as the results are returned, the instances are killed, and its corresponding memory space released. This help us study the overhead of instantiating a new container. In the latter 2 scenarios, the computation unit is wrapped in a flask, a python web framework that allows communication calls via HTTP calls. The application is only initiated once at the start, so we can measure the time for signals to traverse through the communication layer. To keep environment as consistent as possible, only one machine hosting Intel i7-8760 quad core process, 16 GiB of DDR3 memory running ubuntu 18.04 and with docker v19.3.0 and python 3.4 is used. When the experiment is finished, the average and standard deviation of Running duration and layering time will be determined using formula 2, and 3, where n is the number of elements, is average and is standard deviation:

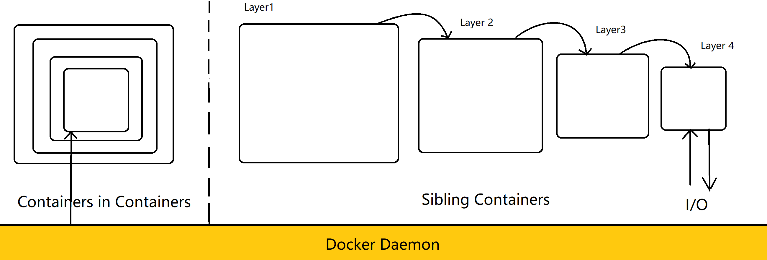
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Measurement** | **time** | **Native** | **New Container** | **Existing Container** | **5x nested container** |
| **Operation time (s)** |  | 0.009868 | 0.022465 | 0.033195 | 0.030283465 |
| **s** | 0.000867726 | 0.001595 | 1.62101E-05 | 0.003111 |
| **Layer Time(s)** |  | 4.01E-07 | 2.440119 | 0.001893 | 0.001209655 |
| **s** | 1.51E-07 | 0.45777 | 7.87E-07 | 0.000561769 |
| **Total(s)** |  | 9.87E-03 | 2.46 | 0.03508739 | 0.031493 |
| **s** | 4.34E-02 | 2.44 | 2.28913E-05 | 0.003606 |
| **Accepted result (%)** | **-** | 100% | 100% | 100% | 100% |

**Result Discovery and Finding**

The generated of logs and calculated data can be found on my Github repository at kamagawa/container\_benchmarking[x]. From accuracy perspective, all entries from all containers meet the accuracy requirement, due to strict diagonality of the matrix that guarantees convergence within 2500 iterations. Runtime (Figure 1) wise, executing the algorithm natively is much faster than any other options, then followed by running in new containers, container in container and existing containers. The standard deviation of running in existing container is the lowest by a large margin, making it the most deterministic option. Having been provisioned its own resource bundle, processes running inside Docker enjoys isolation from system noises as native processes are impacted. However, when placing containers inside containers five folds, the docker daemon’s scheduler can no longer provision the containers directly thus giving rise to layering time standard deviation. Operation runtime in 5-fold nested container is approximately the same as running in one-layer container.

Layering time wise, Native’s layering time is close to the time taken for measuring time itself, meaning that there is no layer between signal sent to being received when sending the signals to the code on the same machine from the same environment. Running code in new environment is the longest and the least deterministic option since the process of creating a new container and provision it resource takes a long time, however being about 2s initializing time, creating containers is still much faster than provisioning a VPC which would take significantly longer time. Sending signals across container layer to existing containers or existing containers in containers x5 takes approximate amount of time. However, sending signals to 1 layered container is much more deterministic than 5 layers.

Whether it’s operation time or layering time, containers in containers doesn’t take longer time than regular container. This is a little bit surprising for layering, as one would expect crossing 5 layers of the containers would take 5 times as long as crossing one layer of containers. This could be attributed to the architecture of containers creation in Docker Figure 1. When creating containers in containers, Docker Damon creates a sibling container that is linked to container that it is “in”, rather than directly hosting the containers directly in container [13]. However, its resources are shared from its Sibling, this could be the reason that both running application in multi layered containers and communicating with the container behave less deterministic than just one layer of container.



Understanding the runtime behavior of containers in different situation is crucial to the Autonomous Vehicle. For such system being able to know when a task will fire, and finish is more important than finishing it as quickly as possible. When one needs to nest a container, they must ask themselves, is this task time critical, and is it ok for it to share resource with its current containers? To achieve a higher level of temporal precision industry partners often uses a Realtime enabled Kernel (rt-kernel) and implement their own clock as part of their container networks.

**Conclusion and Future Work**

This paper presented a container infrastructure for autonomous vehicle development, the scenarios in which it outperforms native development such as multi-scenario single core and one stage multi version. We presented an option for to extend the container network with the Fog network to tap into the power of cloud computing such that more powerful computation can be performed on cloud and less powerful computation perform locally. Then study the runtime overhead when running in container network compared to native machine and discovered that signals crossing into and out of containers experience a considerable delay compared to native but behaves much more deterministically. Nested container does not add extra overhead because architecturally they are linked as “sibling containers” rather than nested inside. However, their provisioned resources are shared making nested containers less deterministic than single layer container. Understanding the operation runtime and communication delay is important when designing a container network of a mixed critical system. To fulfill a higher degree of temporal precision, companies often uses rt-kernel and implement their own time control logic.

During the experiment, we face many obstacles that could potentially be inspiration for future work. One is streamlining the process of image creation such that when a non-critical line is changed in the code, it doesn’t rebuild the entire layer. Adding on to the container architecture, detailed pragmatic approach for utilizing Kubernetes to orchestrate the container network can be explored and utilizing such infrastructure for CI/CD of new features in AV development.

**Reference**

[1] M. Soni, "End to End Automation on Cloud with Build Pipeline: The Case for DevOps in Insurance Industry, Continuous Integration, Continuous Testing, and Continuous Delivery," 2015 IEEE International Conference on Cloud Computing in Emerging Markets (CCEM), Bangalore, 2015, pp. 85-89. doi:10.1109/CCEM.2015.29

[2] D. Bernstein, "Containers and Cloud: From LXC to Docker to Kubernetes," in IEEE Cloud Computing, vol. 1, no. 3, pp. 81-84, Sept. 2014. doi: 10.1109/MCC.2014.51

[3] “About images, containers, and storage drivers”, *docker.io*, 2019, [online], Available: https://docs.docker.com/v17.09/engine/userguide/storagedriver/imagesandcontainers/

[4] J.Fong, “Are Containers Replacing Virtual Machines?”, *docker.io*, 2019, [online], Available: www.docker.com/blog/containers-replacing-virtual-machines/

[5] “Declarative Management of Kubernetes Objects Using Configuration Files”, Kubernetes.io, 2019, [online], Available: https://kubernetes.io/docs/tasks/manage-kubernetes-objects/declarative-config/

[6] A. Gerrard, “What Is Kubernetes? An Introduction to the Wildly Popular Container Orchestration Platform”, blog.newrelic.com, 2019, [online], Available: https://blog.newrelic.com/engineering/what-is-kubernetes/

[7] W. Wolf, “Cyber-Physical Systems”. Computer, Vol. 42, No. 3, 88-89. 2009. doi:10.1109/MC.2009.81

[8] A. Kemeny, F. Panerai, "Evaluating perception in driving simulation experiments", Trends Cogn. Sci., vol. 7, no. 1, pp. 31-37, Jan. 2003. doi:[10.1016/S1364-6613(02)00011-6](https://doi.org/10.1016/S1364-6613(02)00011-6)

[9]  K. Katsaros, M. Dianati, A Conceptual 5G Vehicular Networking Architecture, Cham, Switzerland:Springer, pp. 595-623, 2017, doi:10.1007/978-3-319-34208-5\_22.

[10] P. Masek, M. Thulin, H. Andrade, C. Berger and O. Benderius, "Systematic evaluation of sandboxed software deployment for real-time software on the example of a self-driving heavy vehicle," 2016 IEEE 19th International Conference on Intelligent Transportation Systems (ITSC), Rio de Janeiro, 2016, pp. 2398-2403. doi:10.1109/ITSC.2016.7795942

[11] C. Gauss, “Werke”, Göttingen: Königlichen Gesellschaft der Wissenschaften, 1903, pp. 9

[12] STUDENT, “Probable Error of a Correlation Coefficient”, Biometrika, Vol. 6, Issue 2-3, September 1908, Pages 302–310, [doi:10.1093/biomet/6.2-3.302](https://doi.org/10.1093/biomet/6.2-3.302)

[13] A. Colangelo, “Sibling Docker Container”, Medium, Jul. 2019, [online], Available: <https://medium.com/@andreacolangelo/sibling-docker-container-2e664858f87a>