**Adapting a Container based 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 turns accelerates the rapid prototyping process. Container infrastructure strategy exploited by developers in the cloud domain for its platform agnosticism and flexible scalability presents a viable solution addressing this need. In cloud computing, developers use tools such as Docker to setup containers and then 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-based infrastructure strategy for AV development, testing and deployment with the use of Kubernetes and Docker that facilitates easy 2-way communication network between developers and vehicle. The aim of the infrastructure is to abstract the platform such that once a code is packaged, it may run anywhere.

**Introduction**

Agile, A new form of software development has been quickly wining cloud developers’ favor over the traditional waterfall model. In agile practice, Software developer develop code, run them through CI/CD pipelines, integrate them daily and deploy their prototype to test how it works in real life environment. Then feed discovered issues back into this loop and kick off the pipe again. Yet software technologies found on an AV often come from a wide range of temporal criticality: from low-level safety-critical mechanical control embedded real-time system to high level training of perception deep learning model, as well as mid-level network applications. Developers often fear mess up the temporal separation in Mixed Critical System (MCS), and thus back away from implementing a unified infrastructure strategy. The modularization of teams into specific functional unit also enhances the status quo mindset where things would 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 loop.

A properly architected Container-Infrastructure removes these overheads and allows developers to build once and run on any platform. The accelerated build and test cycle make continuous delivery of new features in response to ever-changing requirement possible. The idea of containers first come from Linux name space which offers the possibilities of isolated resources set in each namespace for each running task。 Docker later came out to streamline the process of image creation. Docker package all software dependencies and running mechanism into an isolated environment called images. Each of the dependency and mechanism is a layer in the image. To update the image, docker update 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. During running, docker daemon supplies the needed configuration from host environment to each container and stores no data inside the containers. This saves user the large resource overhead of virtualizing an entire OS inside each container as Virtual machine infrastructure needs.

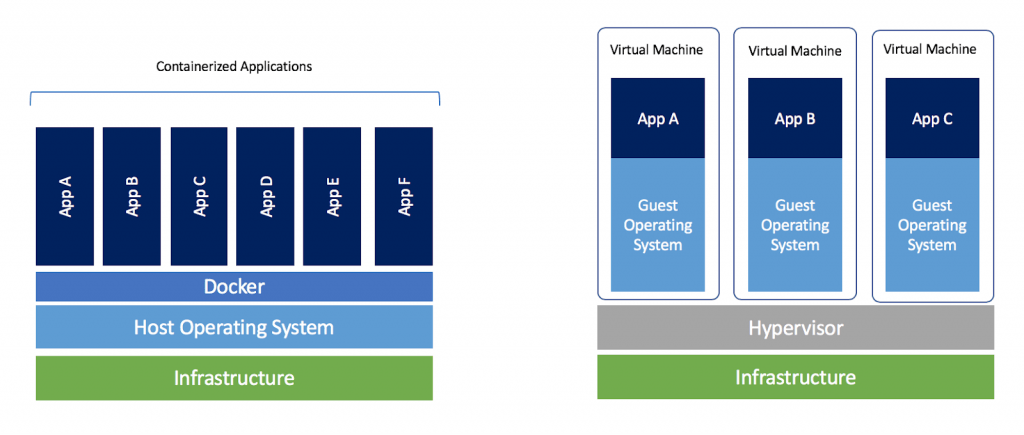
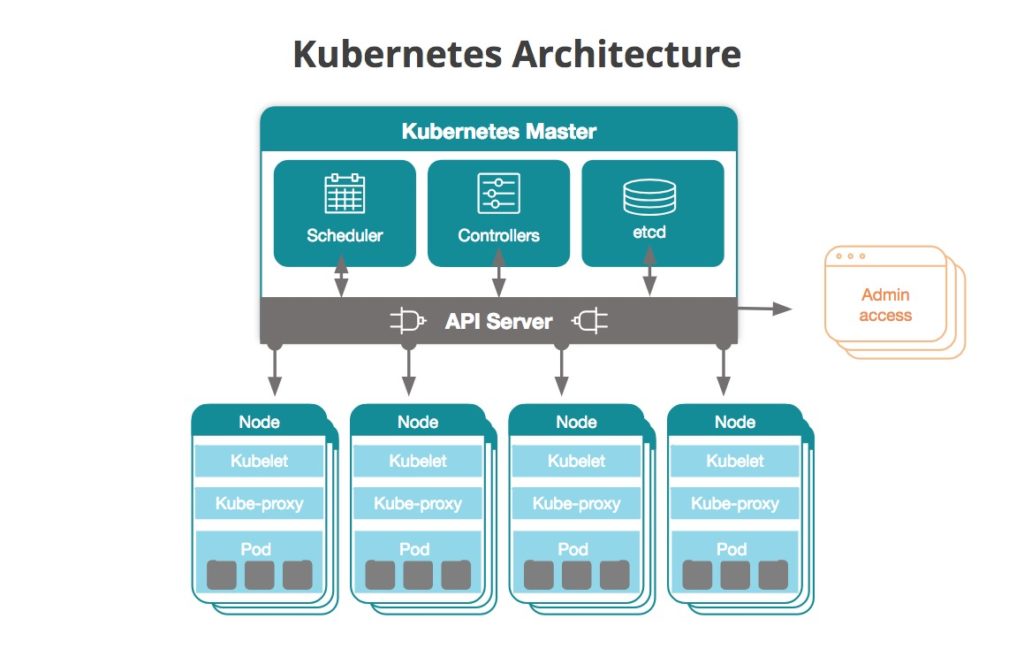


Figure 1 Container Infrastructure vs Virtual Machine Infrastructure

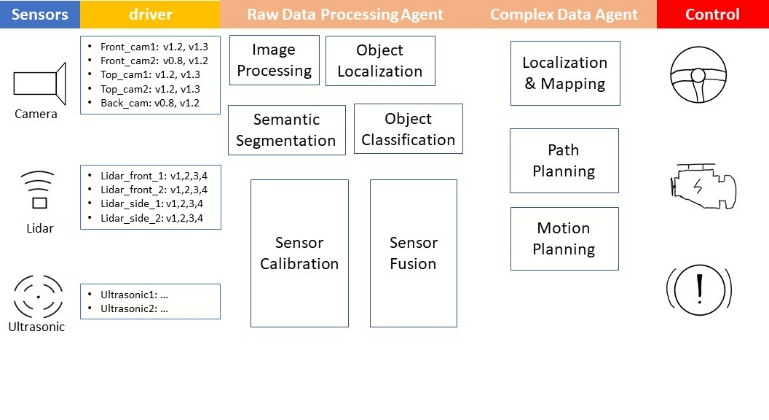
As the number of containers increases, the need for establishing an efficient container network becomes crucial especially for AV where the storage and computing power is highly constrained. Kubernetes is an open source container orchestration tool, that lets user manage a network of containers. Kubernetes reads declarative configurations from Yaml files, where the user specifies a desired state, to which Kubernetes will work its way to achieve such state. 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 worker nodes. 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.



This paper will present { the prerequisite nsetup for current application in section I, present the architecture of the container-based infrastructure for AV development in section II, provide an example use case of using such infrastructure in developing camera perception model in section III and eventually discuss short-comings and precautions developers need to consider in section IV.}

**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). Figure 1.1 shows the data processing line:



Located at the beginning 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 different versions and variation is of the camera driver tedious to manage especially when one needs to perform 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. Then specify the correlation between physical sensors and containers in a key-value file.

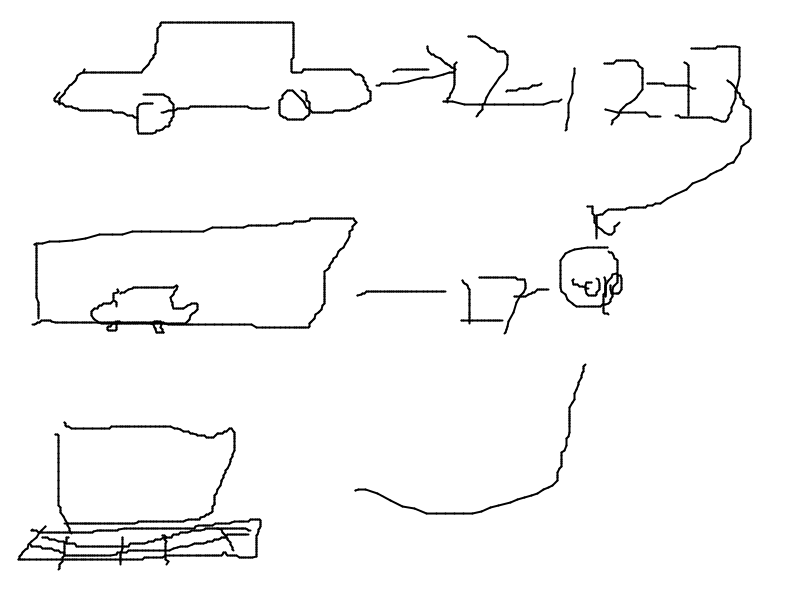
Similarly, data processing agents down the data pipeline can be broken down in a similar structure using the aforementioned strategy. Each agent stands alone in one container. It receives input from upstream agents, process it, and subsequently sends broadcast results to down stream agents.

The reverse of that 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 changing the senario only the effected its upstream or downstream blocks. Running simulation is a very common practice to train vehicle’s perception, this exposes the perception unit to a number different senciarios as show in Figure 1.2:

1) Running the vehicle on real world roads

2) Running the vehicle in a simulated environment

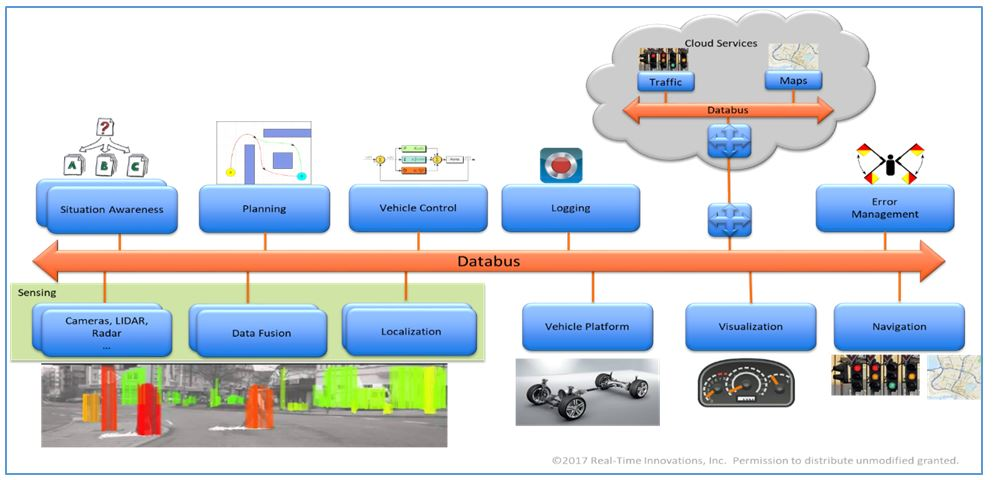
3) Running perception core on a computer simulated model.



In different scenarios different setups are involved but the same perception logic remains: in I) a full fledge vehicle is used. In II) only the camera hardware involving perception logic. In III) 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 extended boundary for Container Network**

Overhead is the pain point of using containers. Virtualization applications in containers inevitably consumes extra system resources and takes longer run-time and exponentially increases communication time across container edges. 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 up streaming resource intensive computation to cloud. 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 host 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 modules are 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 is the response time in a time critical system. Traversing in container networks signals need to exit its originating container and enter its destination container, meaning that at least 2 layers of communication delay is added per container involved. The more containers involved the more layers signals need to enter and exit. Some times there might even be containers inside of containers. Compounding layers by layers of boundary communication gives a significant delay in the signal relay process. The variation of communication time breaks the deterministic trait in a real time system. [Philip Masek’s paper] performed a runtime analysis on the temporal criticality of each container’s operation on 4 different environments in ubuntu vanila-native, vanila-docker, RT-native and RT-docker. The result was showed that time duration for performing operation in Docker is approximately the same as running natively. Real-time enabled Linux Kernel yields performs more deterministically than Vanilla, which however, is overall faster in both empty and loaded environment. 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 for juxtaposition: native run, horizontally across containers, and vertically into containers in containers, and container initiation overhead. In this calculation

Talk about his analysis on the deterministically of running container Talk about how This could add up, perform own analysis

**Useful Sources**

[1] Raeltime container for large scale mixed-critical system by Marcello Cinque

[2] Added clock to make stuff real time <https://roscon.ros.org/2018/presentations/ROSCon2018_LessonsLearnedSelfDriving.pdf>

**3.** <https://cloud.google.com/containers/>