Exploring Function-as-a-Service For Emerging Latency Sensitive Applications

Aidan Parsa
Mentored by Anirudh Sarma

Introduction and Background

Even though the latency of sending a request and receiving a response from the Cloud is suitable for most human internet activities, emerging applications such as autonomous vehicle and drone navigation systems only have machines in the loop, requiring decisions to be made on a smaller timescale. This justifies lowering the network latency by placing servers in proximity to the data source (e.g vehicle or drone). This paradigm of computing is referred to as “Edge computing” because the compute resources are at the edge of the network [4]. However, unlike the Cloud, the Edge does not have abundant resources at its disposal. Careful and judicious use of resources requires rethinking the conventional client-server paradigm when running emerging, futuristic applications.

Function-as-a-Service (FaaS) is a promising approach to conserve resources and aid multi-tenancy in the presence of user traffic unpredictability. Rather than continuously run an application and have it take up resources on the Edge, the FaaS compute paradigm allows the service provider to spawn application components within containers on the Edge only when a particular function is invoked [3]. While the function executes it consumes resources, but once it finishes, the container is removed and no longer takes up precious resources on the Edge server. In this way, Edge resources are allocated in proportion to the current demands of the application instead of requiring the developer to reserve resources for peak traffic in advance. This transient allocation of resources to applications is appealing from the application developers perspective because they only pay on a per function invocation basis rather than having to partition resources ahead of time to accommodate for peak traffic at uncertain times. By converting traditional services into FaaS services, the application can also achieve ease of maintenance and increased flexibility [5]. This serverless paradigm has a number of production offerings including OpenFaaS [2] and OpenWhisk [6] which allows for this type of interaction between client and server.

Emerging applications like autonomous driving and drone navigation explore the role of Edge nodes as a way to enhance perception capabilities and cope with, in many cases, limited onboard resources. A canonical use case of the Edge in a futuristic application is offloading an object detection computer vision model to the Edge server so that instead of the model executing onboard, it relies on the Edge for this functionality. Running the model on the Edge can potentially reduce inference times while enabling the multiplexing of expensive compute resources across vehicles.

Prior work [1, 7] on these applications have envisioned Edge node support as long running and monolithic services, and their efficacy and suitability for the FaaS paradigm is unexplored. This research effort focuses on characterizing the functions used by emergent applications and evaluating the suitability of porting such applications to be FaaS compliant. Implementing the FaaS equivalent of a monolithic application and evaluating its performance serves several purposes. First, it provides a way to understand the viability of the FaaS paradigm for this new class of applications. By comparing the same application structured for two different paradigms, we can directly explore the merit of the FaaS paradigm in the context of emerging applications. Second, we can illustrate the overall process of converting applications to the FaaS paradigm and the steps required. Third, it provides a way to explore potential barriers of the FaaS paradigm for being adopted by futuristic applications.

Implementation:

To study the efficacy of the FaaS paradigm for futuristic applications, we chose to port a work called "EMP: Edge-assisted Multi-vehicle Perception" [1]. EMP is a continuously running Edge service designed to support the perception of Connected and Autonomous Vehicles (CAVs). EMP reduces CAVs vulnerability to occlusion by enabling them to collaboratively share their individual perspectives of the surrounding environment. Participating vehicles partition the point clouds captured from their onboard LiDAR sensors into chunks and transmit them to the Edge service. The Edge service synthesizes the chunks of each individual vehicle into a global view of the environment, runs an object detection algorithm on the combined point cloud, and relays detections back to each vehicle. The goal of EMP is to improve road safety by enhancing the visibility of each vehicle.

EMP relies on Edge servers to synthesize vehicle point cloud chunks because it improves end-to-end latency compared to the traditional vehicle-to-vehicle sharing approach. For every point cloud chunk received by EMP, a Perception Module pipeline is executed sequentially. First, a decode function is run using the Draco decompression library [11] because vehicles compress point cloud chunks before transmitting them in order to reduce network latency. Next, a view merging function executes which merges the chunk into the accumulating view of the environment for the current timestep. This is the global point cloud that the EMP system would run object detection on. Next, the object detection function is run on this synthesized point cloud. Note that the EMP implementation does not actually run object detection online so the pipeline effectively ends at the merge step.

For every timestep, the EMP system also runs an Adaption Module pipeline which uses vehicle bandwidth estimations and vehicle positions to compute which regions of the environment should be uploaded by each vehicle. This function is called Region-based-Edge-Assisted-Partitioning (REAP) and computes a bandwidth-aware Voronoi partitioning of the environment and sends region assignments to each vehicle. The REAP function takes as input the vehicle locations and estimated bandwidth conditions for each vehicle. The purpose of REAP is to prevent redundant chunks of the environment from being sent, and to assign regions of the environment to vehicles that have the clearest view of that area.

In order to port the EMP application to the FaaS paradigm we needed to identify candidate functions. Originally, we hoped that the entire EMP service could be rewritten as a FaaS function chain. This would've provided the best opportunity to explore the performance of a FaaS service compared to its monolithic equivalent. Unfortunately, this wasn't possible because the EMP application builds up state in a large number of data structures. However, stateful functions are not amenable to the FaaS programming model in which functions should be stateless.

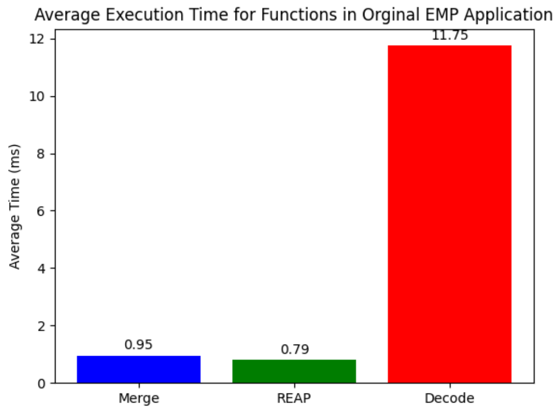
The individual functions which stood out as FaaS compatible were decode and merge from the Perception Module pipeline and reap from the Adaption Module. After selecting these functions we created a hierarchical tree of function calls in EMP to try and obtain more coarsely grained functions. We found that creating this tree where functions are nodes and function invocation establishes a parent-child relationship between nodes was helpful to try to widen the purview of a FaaS function. We tried to increase the functionality of each FaaS function in order to have more EMP code in the FaaS paradigm and also to amortize the overheads of each FaaS function call. For example, the function **updatePartitioningDecisions()** invokes the **reap()** function which means these functions could potentially be combined into a single FaaS function. However, we found that when we tried to combine functions, one of them would either modify data structures or read from data structures. This was the case in the above example where **updatePartitioningDecisions()** would access data structures to read the latest location and bandwidth estimations for each vehicle, pass them as arguments to the **reap()** function, and then use the partitioning decisions returned by **reap()** to further bookeep which regions will be uploaded by certain vehicles.

It's apparent that a major benefit of a monolithic application organization is that it is easy for multiple functions in an application to share state. In contrast, each FaaS function invocation is independent and stateless so there is no way to accumulate state without relying on an external store which would introduce overheads. Ultimately, we selected the REAP, decode, and merge functions to port to FaaS since they are purely CPU bound.

Rather than deploy these functions on a FaaS provider such as OpenFaaS [2], we simulate FaaS by Dockerizing the functions and running them as an isolated and standalone component similar to the microservices approach. The core EMP system which maintains state invokes the functions over HTTP. In this way, the EMP system has been split into a FaaS and monolithic hybrid approach where one core monolithic and stateful component invokes stateless FaaS functions. With this structure, we can still measure the suitability of FaaS for emerging applications.

In order to evaluate the performance of the EMP system after introducing FaaS functions, we conducted timing measurements of the original EMP application and the EMP application with FaaS functions. We used the same point cloud dataset that the EMP authors used in their work. The dataset contains 98 timesteps and we ran the evaluation using 4 client vehicles. To compare the two application architectures, we measured some performance metrics. We consider the end to end latency for an application run of 98 point cloud frames and the average time taken per frame. For the FaaS version of the application, we use further metrics to understand the specific overheads introduced. We measure the overheads of network, serialization, and deserialization both at the core EMP application and also at the FaaS functions.

Results



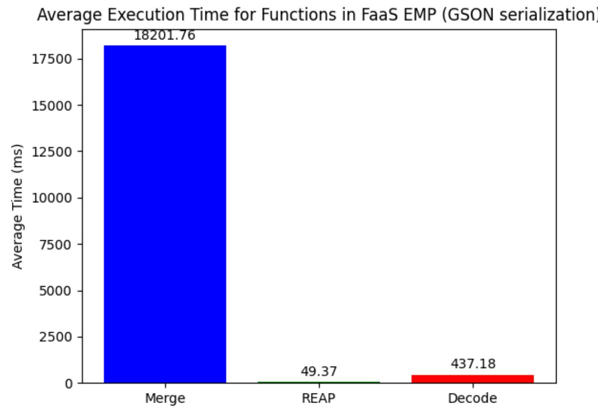
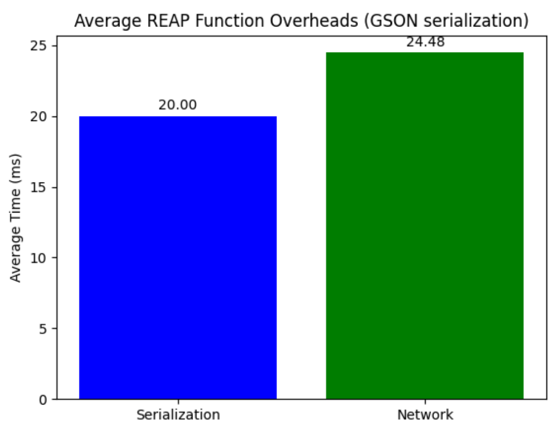


Figure 1 **Figure 2**



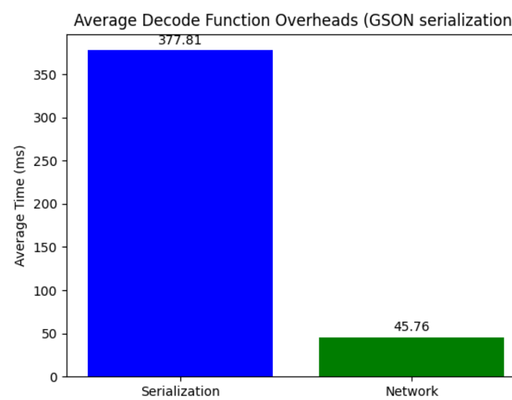
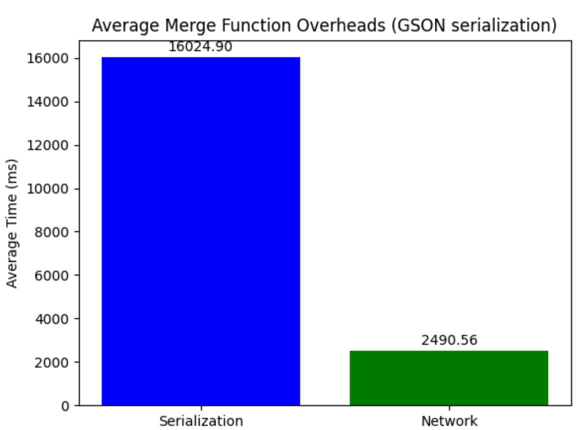


Figure 3 **Figure 4**



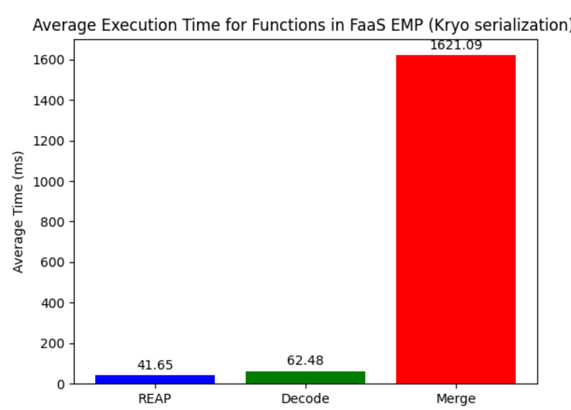
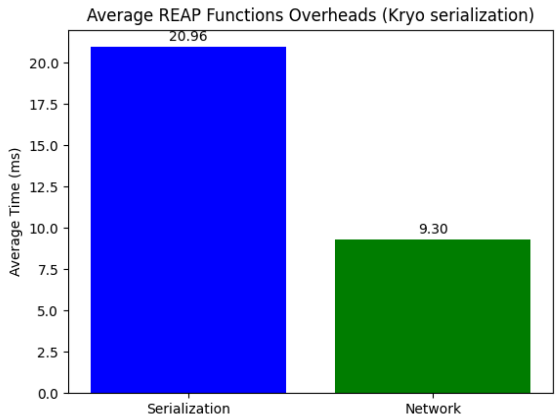


Figure 5

Figure 6



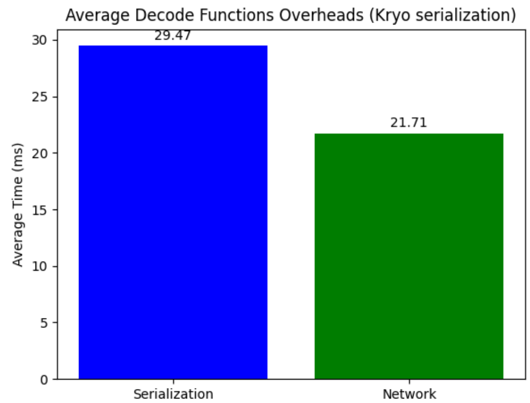
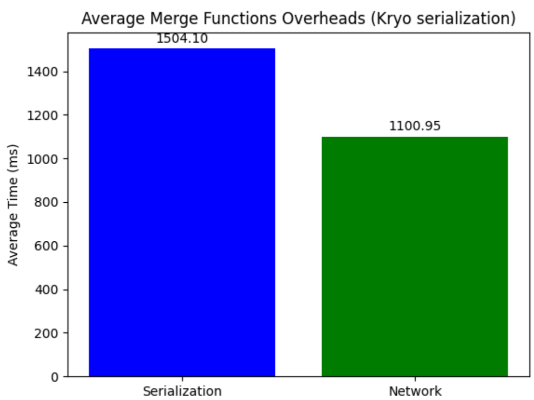


Figure 7

Figure 8



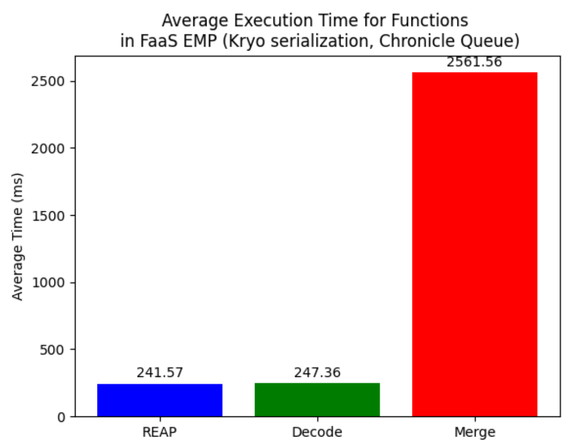


Figure 9

Figure 10

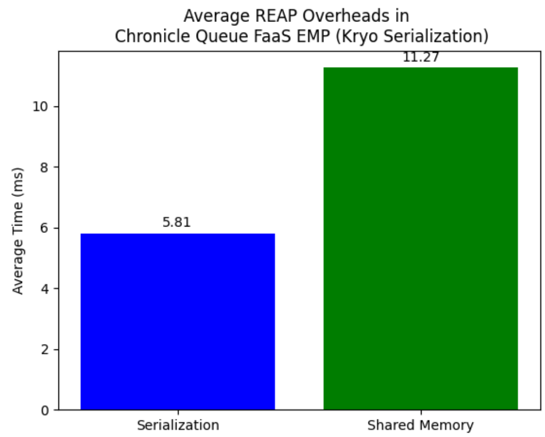


Figure 11

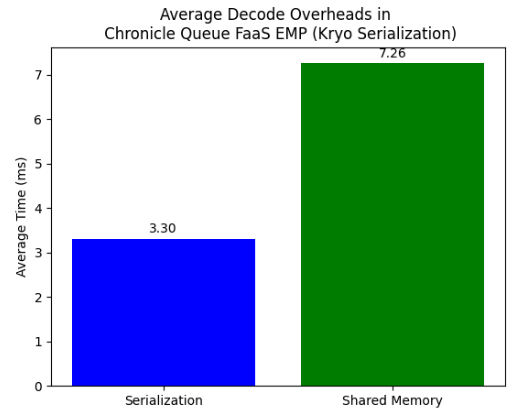


Figure 12

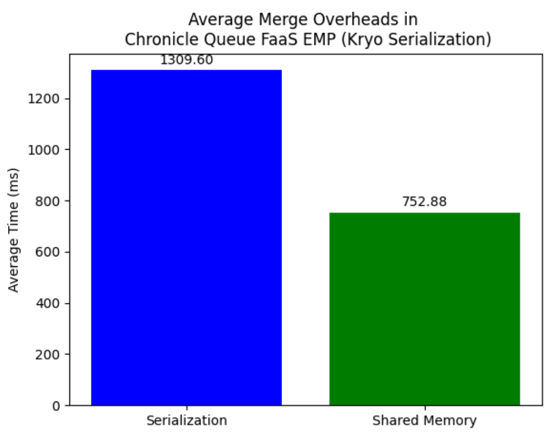


Figure 13

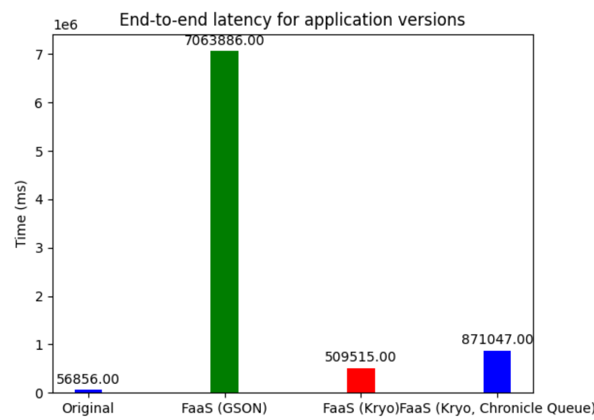


Figure 14

Discussion

Overall, our objective was to explore the fundamental drawbacks of porting a futuristic, latency sensitive application to the FaaS paradigm compared to its original monolithic design. Our experience with porting the EMP application highlights the challenges and implications of developing this class of applications for the FaaS paradigm. In this section we discuss our results for the EMP application and our lessons learned.

Figure 1 presents the average execution time for the reap, decode, and merge functions in the original EMP application. Figure 2 shows the end-to-end latency for the three functions in the FaaS EMP application using the GSON serialization library [8] for communication between the core EMP and the FaaS functions. Comparing these figures we can see that there is a significant slowdown compared to the original EMP application. Figure 14 presents the end-to-end latency across the four application configurations. End-to-end latency is defined as the time elapsed from when the first chunk of the 98 frames is sent to the Edge server to when the last chunk of those frames is merged. While the original EMP application took an end-to-end latency of 56 seconds, the FaaS version took 2 hours. This represents a slowdown of 120x. Figure 3 shows that the overheads due to serialization and network are relatively equal for the reap function. This is because the data for the reap function is small compared to the decode and merge functions. Figures 4 and 5 show that for the decode and merge functions, the overhead of serialization is more significant compared to network overheads. Given the slowdown of two orders of magnitude, this version of the application is clearly not a viable option.

In the EMP application, vehicles transmit raw point clouds from on board LiDAR sensors. Because each point cloud is around 2MB, the FaaS organization introduces four serialization or deserialization points per function invocation in addition to the network transmission overhead. It's interesting to note that sending raw sensor data is a decision of EMP. For autonomous driving systems which transmit processed data instead, the overheads may be smaller since serialization times are a function of the size of the data. Because Figures 4 and 5 indicate that serialization and deserialization is the dominant overhead in the FaaS version of the application, our next version of the application explored a different serialization library.

In this version of the application we use the Kryo serialization library [9] instead of the GSON serialization library. The Kryo serialization library differs from GSON because it converts the data into binary format rather than JSON format. With the Kryo serialization library, the end-to-end latency of the application was 8 minutes, a substantial improvement over the GSON version. Not only did the serialization and deserialization times decrease, but it also decreased the network overhead due to a more compact data representation. Although it is an improvement, an average frame processing time of 4.5 seconds is still not suitable, given that the original application was 400 milliseconds on our reference machine.

Looking at Figures 7-9, it appeared there was room to explore alternatives to the network for transmitting the data, particularly in the merge function which accounted for 1100.95 milliseconds of overhead. In our next scheme, we explored using the ChronicleQueue library [10] which allows for fast shared memory IPC between JVMs. The downside to this approach would be that the core of the EMP service and the FaaS functions would need to be co-located on the same machine. Although this presents a drawback, it may still allow some benefit of the FaaS paradigm to be realized as functions can be scaled on demand and resources need not be allocated up front. This can be beneficial since the REAP, decode, and merge functions do not execute at the same frequency. For example, decode and merge are run for every chunk of the frame whereas the reap function only runs once per frame. Therefore it may make sense to use FaaS to scale these instances of functions a flexible number of times.

The end-to-end latency for the version of the EMP application using Chronicle Queue shared memory was 15 minutes. Surprisingly, it is slower than using the network. We believe this is because with 4 vehicles each sending MBs worth of point clouds per frame, the size of the queue is too large to be in memory and is swapped to disk. Despite the overhead of shared memory appearing lower for shared memory versus the network, we believe the cost of swapping is not accounted for in this measurement. This is supported by observations from the /proc/[pid]/io interface which shows the number of bytes written to and read from secondary storage during the process's lifetime. The version of EMP which didn't use Chronicle Queue read 24526848 bytes and wrote 999424 bytes. However, the version of EMP that did use Chronicle Queue read 48861184 bytes and wrote 10175934464 bytes. This represents a 2x increase in the number of bytes read from stable storage and a 10,000x increase in the number of writes. Further causes of slowdown are that we introduced locks into the REAP, decode, and merge functions in order for each thread on the core EMP to get access to a separate Chronicle Queue for communication with the function. Each Chronicle Queue represents a unidirectional communication medium, so the threads on the Core EMP must synchronize themselves to gain access to one of the Chronicle Queues to communicate with the respective FaaS function.

The recommended frame processing time for autonomous vehicles is 100ms per frame. Our experiments suggest that the overheads introduced by serialization, deserialization, network, and shared memory make the FaaS paradigm poorly matched for the EMP application and a class of applications characterized by a crucial need for low latency communication. This study underscores the inherent problems with using FaaS for settings in which low latency communication is not just a nice to have, but a crucial component of the system safety. In cases like this, it is not tolerable to increase the latency despite savings in monetary cost of increased flexability. This study shows that improvement in serialization, network, and IPC may make FaaS more viable for these futuristic applications.

References

[1] Xumiao Zhang, Anlan Zhang, Jiachen Sun, Xiao Zhu, Y. Ethan Guo, Feng Qian, and Z. Morley Mao. 2021. EMP: edge-assisted multi-vehicle perception. In Proceedings of the 27th Annual International Conference on Mobile Computing and Networking (MobiCom '21). Association for Computing Machinery, New York, NY, USA, 545–558. https://doi.org/10.1145/3447993.3483242
https://dl.acm.org/doi/pdf/10.1145/3447993.3483242

[2] Ltd, O. (n.d.). *Home*. https://www.openfaas.com/

[3] Tom Kuchler, Michael Giardino, Timothy Roscoe, and Ana Klimovic. 2023. Function as a Function. In Proceedings of the 2023 ACM Symposium on Cloud Computing (SoCC '23).

Association for Computing Machinery, New York, NY, USA, 81–92. https://doi.org/10.1145/3620678.3624648

[4] K. Cao, Y. Liu, G. Meng and Q. Sun, "An Overview on Edge Computing Research," in IEEE Access, vol. 8, pp. 85714-85728, 2020, doi: 10.1109/ACCESS.2020.2991734. keywords: {Cloud computing;Edge computing;Real-time systems;Internet of Things;Bandwidth;Security;Data privacy;Edge computing;cloud computing;Internet of Things}, https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9083958

[5] Mohammad Shahrad, Jonathan Balkind, and David Wentzlaff. 2019. Architectural Implications of Function-as-a-Service Computing. In Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '52). Association for Computing Machinery, New York, NY, USA, 1063–1075. https://doi.org/10.1145/3352460.3358296 https://parallel.princeton.edu/papers/micro19-shahrad.pdf

[6] *Open source serverless cloud platform*. Apache OpenWhisk is a serverless, open source cloud platform. (n.d.). https://openwhisk.apache.org/

[7] Shuyao Shi, Jiahe Cui, Zhehao Jiang, Zhenyu Yan, Guoliang Xing, Jianwei Niu, and Zhenchao Ouyang. 2022. VIPS: real-time perception fusion for infrastructure-assisted autonomous driving. In Proceedings of the 28th Annual International Conference on Mobile Computing And Networking (MobiCom '22). Association for Computing Machinery, New York, NY, USA, 133–146. https://doi.org/10.1145/3495243.3560539 https://dl.acm.org/doi/pdf/10.1145/3495243.3560539

[8] Google. (n.d.-b). *Google/gson: A java serialization/deserialization library to convert java objects into JSON and back*. GitHub. https://github.com/google/gson

[9] EsotericSoftware. (n.d.). *EsotericSoftware/Kryo: Java binary serialization and cloning: Fast, efficient, automatic*. GitHub. https://github.com/EsotericSoftware/kryo

[10] OpenHFT. (n.d.). *OpenHFT/Chronicle-Queue: Micro second messaging that stores everything to Disk*. GitHub. https://github.com/OpenHFT/Chronicle-Queue

[11] Google. (n.d.-a). *Google/Draco: Draco is a library for compressing and decompressing 3D geometric meshes and point clouds. it is intended to improve the storage and transmission of 3D graphics.* GitHub. https://github.com/google/draco