Blogs & News

Power and Performance

Dynamically Scaling Video Inference at the Edge

Puneet Sethi

19 Dec, 2019

Dynamically Scaling Video Inference at the Edge using Kubernetes* and Clear Linux* OS

Authors: Ken Lu, Puneet Sethi

Introduction

This case study describes how to redesign a traditional video inference pipeline with a monolithic architecture into a cloud-native architecture. Our use case combines edge computing and video inference. We explore how the new cloud-native architecture allows us use Kubernetes* pod scaling for a complex edge inference solution.

The demo source code is published on GitHub. The software uses the Intel® Distribution of OpenVINO™ and container images based on Clear Linux* OS.

Background

Kubernetes has rapidly become a key ingredient in edge computing. With Kubernetes, companies can run containers at the edge in a way that maximizes resources, makes testing easier, and allows DevOps teams to move faster and more effectively as organizations consume and analyze more data in the field.

However, a challenge when moving to a container-based Kubernetes cluster is understanding how to evolve traditional software with monolithic architectures or service-oriented architectures (SOA) into cloud-native architectures that are more appropriate for microservices.

Take video inference for example: A traditional video inference pipeline includes fetching a frame from a video source, performing inference on the frame against a pre-loaded model, and outputting the result - all executed within a single thread/process. This traditional single thread/process approach has limitations: Scaling resources is complex and potentially expensive. This is problematic in a cloud environment where scaling in response to dynamic needs is necessary.

In comparison, a modular cloud-native architecture provides many benefits such as:

- Flexible deployment across cloud data centers and edge
- Independent scaling on diverse edge hardware, supporting a variety of workloads
- Easier operations for devops teams
- Reduced disruption of the end-user experience

Case Study

Video inference is at the heart of many of edge AI solutions such as self-driving systems, smart cities, and intelligent manufacturing. We use the smart retail use case to better understand the requirements of edge AI solutions with video inference . Smart retail is a term used to describe improving all aspects of the buyer and seller experience with technology. Some potential requirements for a smart retail system using video inference are:

- When a customer enters the store, a smart store should be able to authenticate the customer and open the door at the right time.

  Solution: Have one or more cameras to detect people and perform facial recognition.

- While a customer is browsing, a smart store should be able to recognize a customer's position in the store and present a customized shopping menu to the customer.

  Solution: Have one or more cameras to do body detection/recognition and perform behavior analysis.

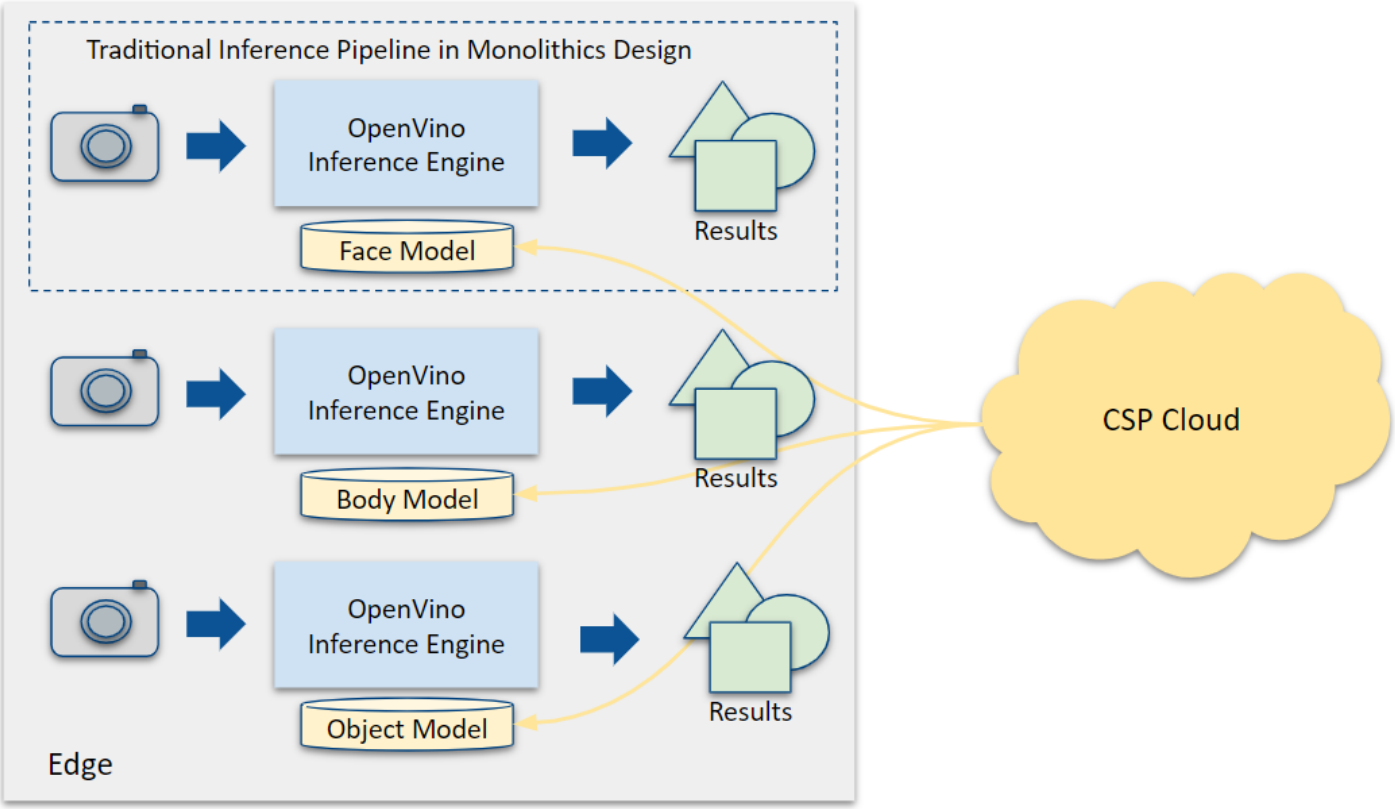
- When a customer exits the store, a smart store should be able to automatically charge payment based on the person exiting and the goods removed from the store.

  Solution: Have one or more cameras to do object detection and perform facial recognition.

In practice, many cameras must be in place to meet the various inference requirements such as facial recognition, people/body recognition, and object recognition. Inference engines can run in the cloud, but increasingly are also running on local servers at the edge for real-time decision making. The need for diverse inference models that can be deployed from cloud to edge has been addressed by many cloud service providers (CSP) with solutions like Azure* Stream Analytics and AWS* Greengrass.

When developing an inference service, open source projects like OpenVINO can be used for flexible vertical scaling on heterogeneous edge computation hardware and accelerators, including CPU, GPU, VPU (Intel® Movidius™), and FPGA.

Challenges with inference on Kubernetes



In the OpenVINO documentation examples, each inference engine fetches frames directly from a camera input stream, executes model detection, and reports results. This traditional monolithic design is common for edge AI devices with low-end computation resources where scaling is limited. However, in cases where more processing power is required, a powerful edge server may be needed to handle diverse workloads on a consolidated platform. This is where scaling becomes relevant and poses unique challenges. For example:

- **Different inference models with different computation requirements.**
  For example, the standard pre-trained facial detection model requires 2.835 GFlops computation, while the pre-trained person detection model requires 12.427 GFlops computation. It would not be fair to allocate the same resources for both inference services.

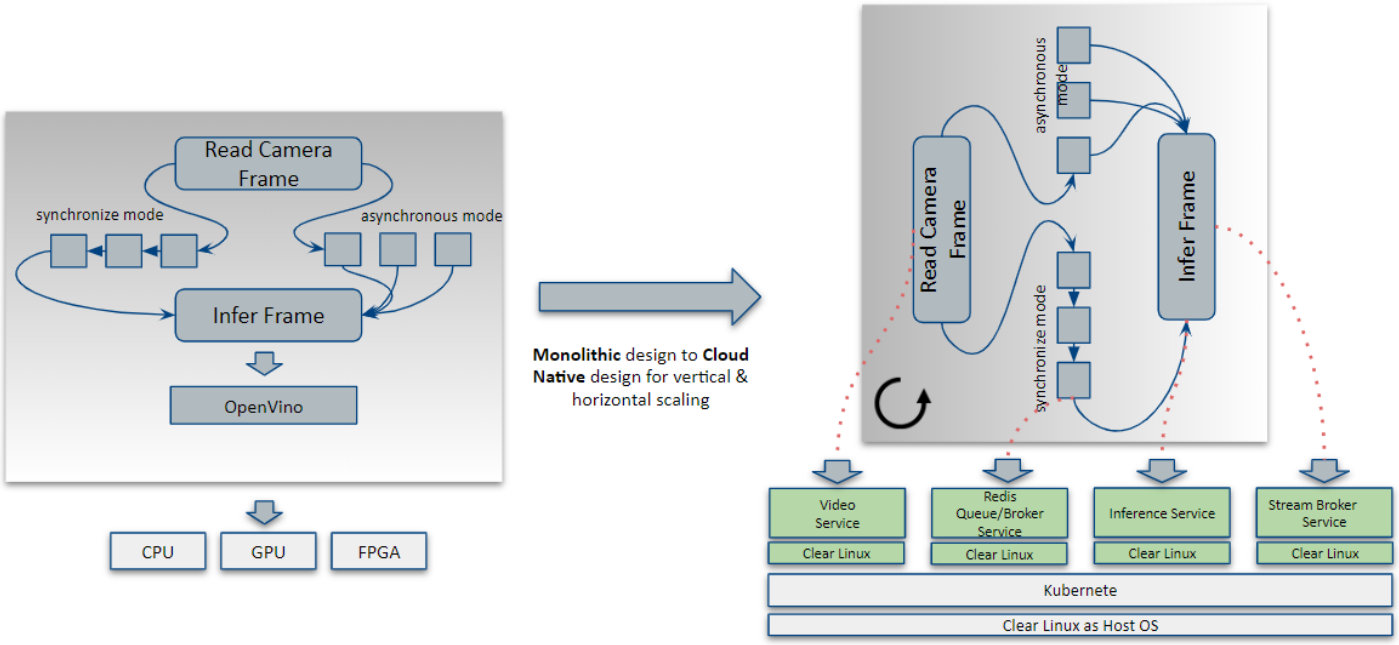
- **Multiple camera inputs for inference models that require more computation resources.**
  It would not be resource-effective to create separate instances of an inference service for each camera stream.

- **Multiple inference workloads.**
  OpenVINO provides an asynchronous API to do inference on multiple CPU threads/cores in

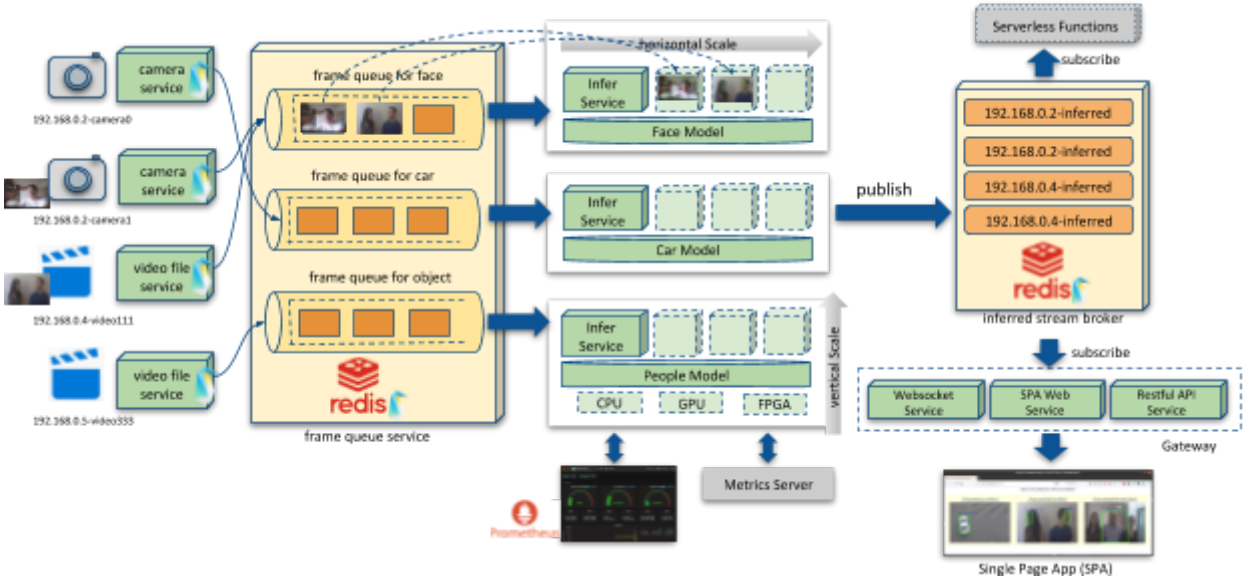
parallel on a single node in order to improve overall frame-rate throughput. This is great for a single inference workload, but is too rigid for a Kubernetes environment managing a cluster of resources.

Transformation to a cloud-native design

After identifying limitations of the traditional approach, we redesigned the inference pipeline into a cloud-native architecture. The redesign allowed us to enable dynamic, cluster-wide, horizontal scaling.



The key idea behind the transformation into cloud-native software was decoupling each component in the original single vertical inference stack into individual microservices that could scale independently. In our scenario there are five services:



1. **Camera/Video Service.**
   Responsible for capturing individual frames from a video source (such as camera devices or online media streams) and reporting them to the Frame Queue Service. The service is a Python application with image transformation algorithms, running inside an OpenVINO container image based on Clear Linux OS which includes an OpenCV* library with AVX2/AVX512 optimizations.

2. **Frame Queue Service.**
   Responsible for providing real-time frame queuing for all video/frame input for eventual processing by the Inference Service. Multiple dedicated frame queues are created for each type of model (for example a face detection queue, a body/people detection queue, or an object detection queue). The service is run inside of a redis container image based on Clear Linux OS.

3. **Inference Service.**
   Responsible for doing the actual video inference against trained models. It fetches the frame from the Frame Queue Service, performs the inference, and reports the resulting output to the Inferred Stream Broker Service. The service runs inside an OpenVINO container image based on Clear Linux OS.

   By default, the OpenVINO CPU plugin is used with VNNI (AVX-512 Vector Neural Network Instructions), with OpenVINO INT8 optimized models. OpenVINO also provides a

heterogeneous plugin to allow a single network of diverse hardware with vertical scaling capability. This means multiple inference service instances can be created to match the diverse hardware available on a Kubernetes node such as CPU, GPU, VPU (Movidius), and FPGA, with automatic workload placement.

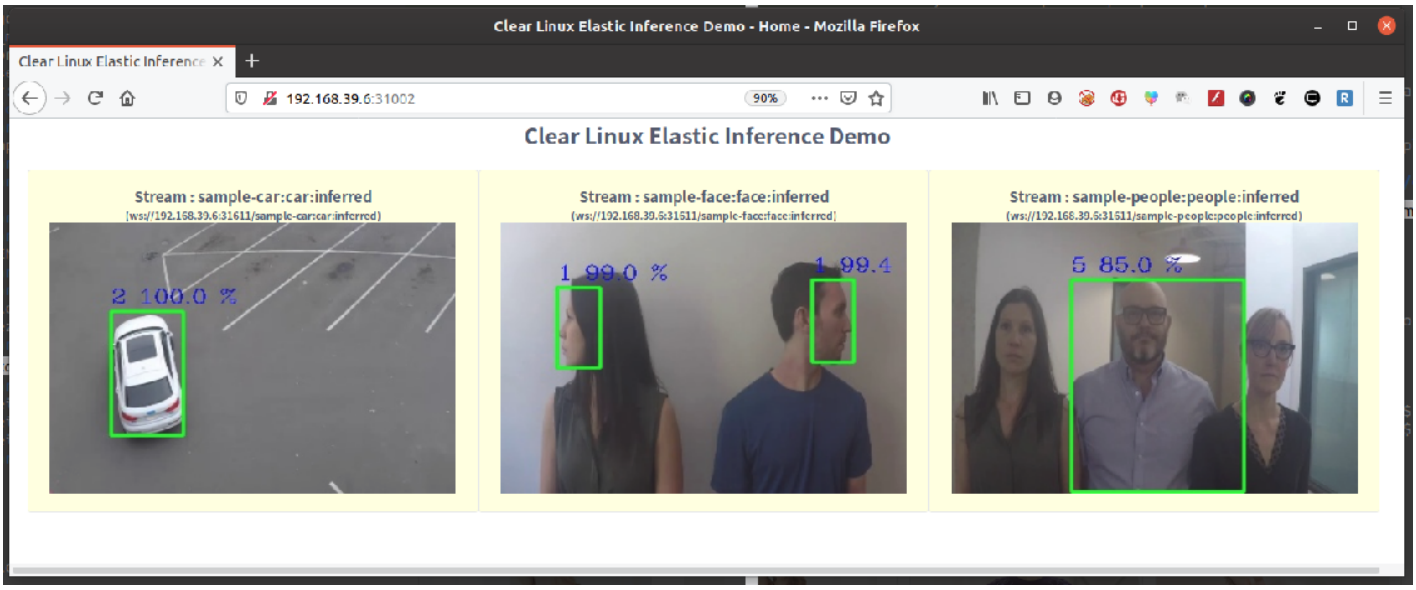
4. **Inferred Stream Broker Service.**
   Responsible for receiving the information from inferred frames and publishing the results for another application to use. The service runs inside a redis container image based on Clear Linux OS. Note that this can be the same redis instance as the Frame Queue Service.

5. **Gateway Service.**
   An example output service for displaying the inferred streams in a HTML5-based single-page application (SPA). It includes the following components:

   - Restful API service to expose the stream information made available by the Inferred Stream Broker Service. It uses Python*'s Flask framework inside of a Python container image based on Clear Linux OS.

   - Websocket service to publish the stream via WebSocket protocol. The service is implemented as a I/O bound Python application and leverages Python's coroutine and asyncio to reduce CPU allocation. The service runs inside a Python container image based on Clear Linux OS.

   - SPA web service to publish the front-end HTML5 page using the VUE iview JavaScript framework. The SPA web service is an example of how the inferred stream data can be used. Inferred stream output is a simple queue that any application can pull from and then act on as desired. This is conducive to serverless functions.



Leveraging cloud scalability

With the components of the video inference solution split into independent services, it becomes easier to see how it can scale using standard cloud-native application scaling techniques. Beyond vertical scaling, the inference service can be scaled horizontally on Kubernetes, benefiting from its stateless design.

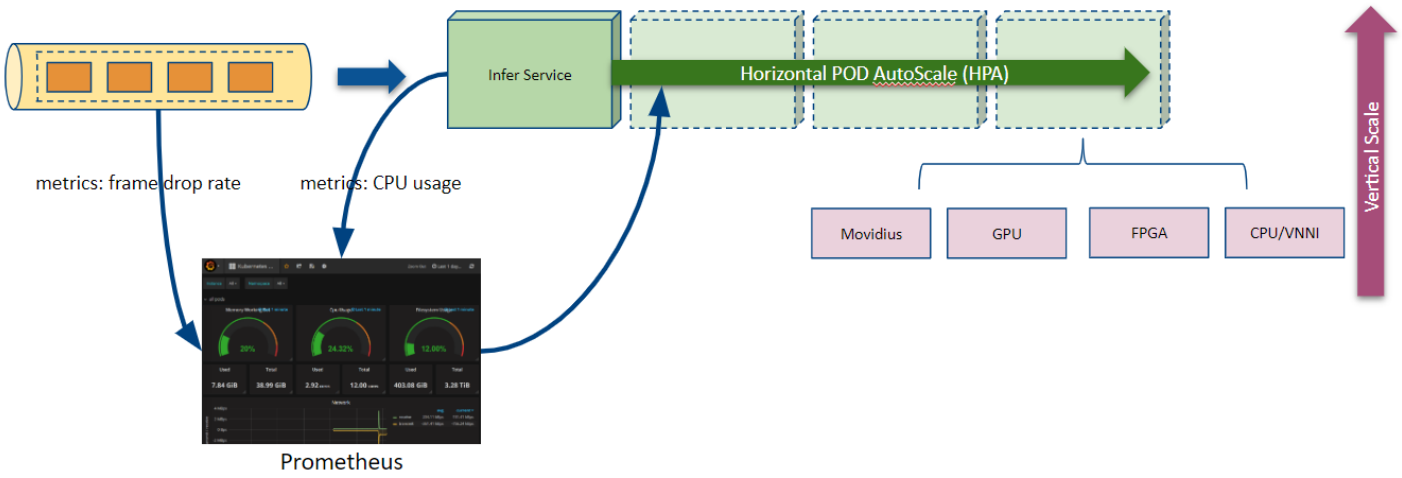
A custom Kubernetes Operator can be defined to automate additional scaling scenarios using more intelligent metrics such as CPU utilization of different models, changing the number of input streams, or changing the rate at which frames are received versus dropped from the frame queues. For example:

- The Kubernetes Horizontal Pod Autoscaler (HPA) can be enabled to schedule replicas of services dynamically. This means the replica count for each inference service on Kubernetes can be scaled according to the metrics above, not just CPU utilization.
- If more cameras are added for face detection, then a Kubernetes Operator can increase replicas of the face inference service.
- If the computation requirements of a new inference model are smaller than the previous model, the Kubernetes Operator can reduce the replicas of the inference service.

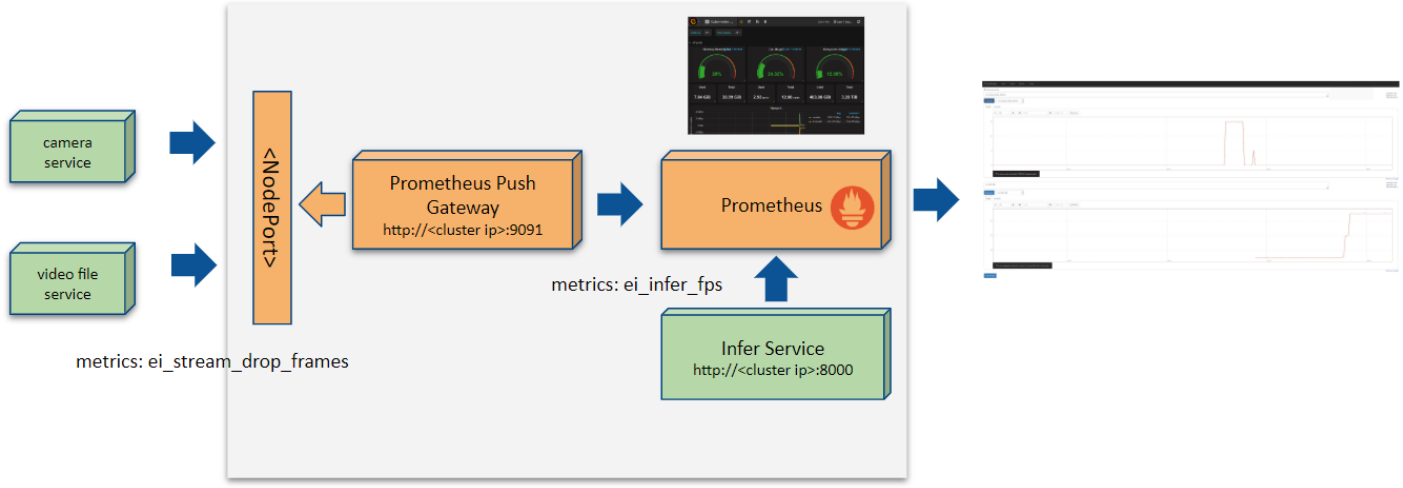
- If the rate at which frames of object detection/recognition are being dropped (FPS drop) is bigger than half of the capture speed (FPS), then a Kubernetes Operator can increase the replicas of the object detection service to improve the frame rate.

Customizing the Kubernetes Horizontal Pod Autoscaler

Kubernetes provides the Horizontal Pod Autoscaler feature, which automatically scales the number of pods in a replication controller, deployment, replica set, or stateful set based on observed CPU utilization.



To determine when to scale, CPU utilization is a good metric to start with. However, metrics closer to the actual running services will provide more accurate data for the video inference scenario to determine when to scale up or down. For example: frame capture speed from the camera service, frame drop speed from the frame queue service, or inferred frames speed reported from the inference service. One easy way to obtain these metrics is to use Prometheus via its push gateway service.



With relevant metrics being collected and accessible, the HPA could be enabled according to the Autoscaling on multiple metrics and custom metrics section of the Kubernetes documentation.

Conclusion

Based on growth trends of edge computing we built a video inference solution using open-source technologies such as Clear Linux OS, OpenVINO, and Kubernetes. We leveraged automated horizontal scaling mechanisms with improved metrics as follows:

- Identified challenges with the traditional monolithic application design used for video inference workloads.
- Redesigned software components with a cloud-native mindset. The inference pipeline was broken into several microservices based on Clear Linux OS containers, which can operate independently. Using Clear Linux OS allowed our developers to automatically adopt the hardware-specific software optimizations for greater performance.
- Deployed the redesigned software as services on a Kubernetes cluster. This allowed us to leverage the well-known horizontal scaling abilities of a cluster.
- Improved the Kubernetes Horizontal Pod Autoscaler to operate on metrics relevant to our application (FPS, dropped FPS, inferred FPS) in addition to the system-level (CPU, memory, storage) metrics.

While this case study focused on one scenario and problem, the techniques and tools described can be used to transform other software to performant cloud-native software.

We encourage you to review the source code on GitHub, check out the benefits of Clear Linux OS, and let us know what other cloud and edge problems you'd like to see solved!

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more complete information visit http://www.intel.com/benchmarks.