WHITE PAPER

Secure Inference at Scale in Kubernetes* Container Orchestration

Dariusz Trawinski, Intel Corporation

January 2019

Introduction

Inference as a service is seeing wide adoption in the cloud and in on-premise data centers. Accommodating various types of model servers like TensorFlow* Serving, Intel® Distribution of OpenVINO™ Toolkit or Seldon Core* in Kubernetes* is a great mechanism to achieve scalability and high-availability for such workloads. Nevertheless, the task of configuring the Kubernetes load balancer can be difficult. This article presents the most common challenges and recommended solutions.

One of the differences between inference and well-known load distribution methods is that inference typically uses a gRPC* (Google Remote Procedure Call) API instead of REST (Representational State Transfer) API. gRPC has great advantages over REST because of its low latency and efficiency in data serialization. Inference workload distribution presents several challenges, including potential load balancer bottlenecks, the optimal configuration of session affinity to pods, and the security of traffic between clients and endpoints (including proper access control).

This article describes recommended techniques for configuring load balancing in Kubernetes with a special focus on inference applications. It also describes how to protect served gRPC endpoints with Mutual Transport Layer Security (MTLS) authentication, both on the cluster and client side, and how inference scaling can be simplified with Inference Model Manager* for Kubernetes.

The examples covered here will use two model servers: TensorFlow* Serving and OpenVINO™ Model Server, with installation instructions included in the appendix.

Session Affinity to Kubernetes* Pods

AI applications are typically part of larger distributed systems and thus handle client requests both within and across data centers. One example is a swarm of client hosts submitting the requests to a gRPC endpoint. In other cases, a single host could generate a high volume of calls from a multithreaded program or a volume of independent processes.

In the latter scenario, Kubernetes may distribute the load in a suboptimal way and Kubernetes session affinity (and the implementation of cloud provided load balancers) could imperil scalability for gRPC inference requests. For stateful sessions, it is beneficial to route the sequential calls from a client to the same Kubernetes pod back end instance.

However, inference requests in such situations may all end up in a single node while other nodes remain idle.

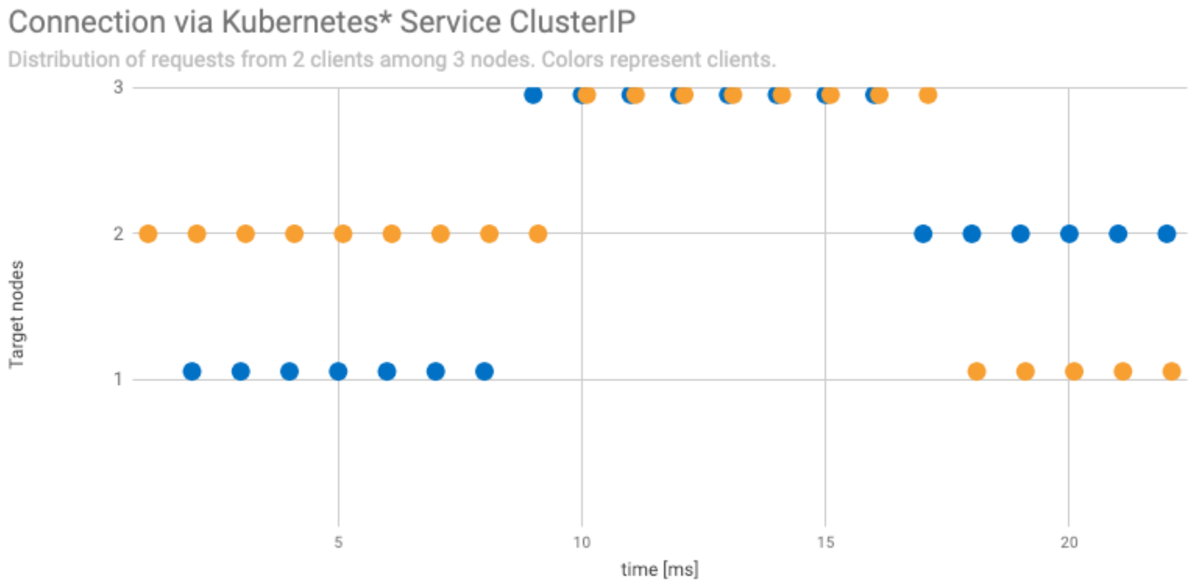
Typically, load balancers provides session affinity through:

- ClientIP - Routing remains static to a single backend instance till the configurable session timeout.
- Cookie - Connections from the same host can be distinguished using assigned session cookies so they can be distributed to multiple backed instances.
- None - No affinity should be similar to round-robin implementation, but for gRPC calls it preserves the routing as long as a connection is active.

Of particular relevance is how the gRPC client communicates with the server. It first establishes the connection channel. This is followed by sending a number of requests over this connection. gRPC requests to a service's virtual IP are routed by a kube-proxy component to a single target instance when the same stub is reused on the client.

Consider a front-end multithreaded microservice submitting inference requests to another Kubernetes service with inference execution nodes. In the case of a heavy load and a steady stream of requests, all the calls might be passed to a single Kubernetes pod.

The chart below demonstrates the request distribution from two clients connecting to a Kubernetes service with three pods running OpenVINO Model Server and executed in a GKE (Google Kubernetes Engine) cluster with Kubernetes version 1.10.7. The whole series of requests is routed to a single pod. Independent connections are routed randomly.



Similar behavior will be observed when a service is exposed by the Kubernetes cluster externally by cloud load balancers and service type is set to “LoadBalancer”. The service will get an external IP allocated but the same issues with routing will take place.

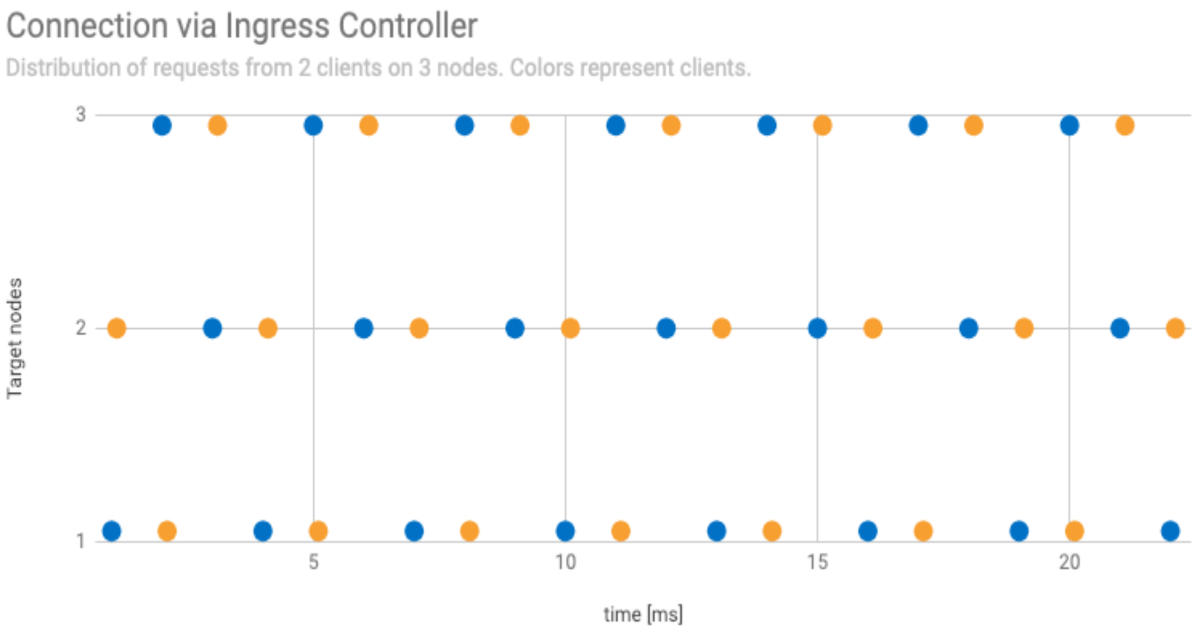
The reason for this is that gRPC is using the HTTP2 protocol where every request is a stream inside the same TCP connection. For that reason, L3/L4 load balancers will route all requests to a single target instance as long as the connection is established.

A more optimal solution is to employ an ingress controller which will perform the routing using L7 load balancing with full support for the gRPC protocol. The following example is based on nginx-ingress, which was proven in our tests to be reliable and secure for gRPC traffic.

With the nginx ingress controller, we expose a single external IP address which is redirected to an ingress-nginx Kubernetes service with one or more controller pods. The nginx controller is load-balancing the traffic to the appropriate target backends according to the rules defined in ingress records.

The routing is based not only on a TCP connection in L3-L4 network transport layer as in earlier examples but also on http and http2 protocol in application layer L7 of the network OSI model.

The distribution from two clients among three inference nodes with ingress nginx controller as the load-balancer is depicted below in figure 2. Each request is reassigned to a different Kubernetes pod, for a more even and optimal load distribution compared to calling ClusterIP.



Ingress Mapping Rules with gRPC

In Kubernetes ingress records, it is possible to configure 2 types of rules assigning the incoming traffic to the backend services: rules based on host name or based on the URL path.

For inference requests over a gRPC interface, the rules based on URL path are not applicable because the gRPC protocol does not allow connections with the path as a parameter as in REST calls. gRPC clients connect to a specific target IP and port by just calling a given remote function name and passing its parameters.

What remains is the host name based rule in the ingress records. Often, we would like a Kubernetes cluster to serve a large volume of inference endpoints with a variety of AI models, so we end up with a large number of DNS names configured in the ingress rules. One could prepare multiple rules in a single ingress record or add multiple ingress records with a single rule. It is only a matter of preference, as long as the ingress annotations are identical. Generally, separate ingress records might give more flexibility.

A large number of inference endpoints might increase maintenance work because each of them might require adding a DNS record ‘A’ or ‘CNAME’ pointing to the external IP of the ingress controller service or potentially a dedicated transport layer security (TLS) certificate to encrypt the traffic.

There are 2 methods to mitigate the burden of DNS:

- Creating a DNS record with a wildcard '*' for example *.clustername.yourdomain.com. This way, you could create ingress endpoints using an arbitrary DNS name matching the regular expression and they would all point to the same IP address.
- Using an SNI feature in the TLS protocol, which allows gRPC client to add a special header with information about the target host. The gRPC client might connect to the ingress controller using an IP address or any DNS name, but the ingress controller will route the traffic based on the included header. This can be set on the client side using connection option `grpc.ssl_target_name_override` like here:

  ```
  opts = (('grpc.ssl_target_name_override', 'endpoint.clustername.yourdomain.com'),)
  ```

With that, you can easily configure in a Kubernetes cluster a high volume of inference endpoints and services with multiple AI models.

Security in Inference Endpoints

Security is an important aspect of an inference system. It should address a variety of threats and protect user assets such as inference endpoints serving AI models, input data, and inference results returned.

This requires user authentication and authorization just like traffic encryption between the clients and the inference system in the Kubernetes cluster.

Traffic encryption is straightforward to implement on the ingress controller, using TLS termination. The concept and example are described below.

User authentication and authorization could be employed via JWT (JSON Web Token) based mechanism or via a client certificate. Here, the first option will be presented based on Mutual Transport Layer Security (mTLS) authentication. It is convenient to control access to the inference endpoints for the applications, automation scripts, and other microservices.

Inference Endpoint Termination with TLS

Generally, the standard documentation on enabling TLS in the nginx ingress controller applies. You can refer to the following links covering the TLS termination feature:

- https://kubernetes.github.io/ingress-nginx/user-guide/tls/
- https://kubernetes.io/docs/concepts/services-networking/ingress/#tls

There are a few caveats which should be taken into consideration specifically for inference endpoints protected over gRPC.

We now describe how to create self-signed certificates which can be considered secure in the case of gRPC clients because it easy to establish the certificate's trust with the signing CA (Certificate Authority) certificate. For Web-based applications using the REST API, the preference would be certificates signed by third-party trusted Certificate Authorities.

The first step should be to create the CA (Certificate Authority) certificate which should be used to sign the gRPC server certificates. The usage of the CA certificate is not mandatory but strongly recommended.

Creating CA Certificates

Create a CA private key:

```
openssl genrsa -out ca/ca.key 4096
```

Or, to save the key in an encrypted format:

```
openssl genrsa -des3 -out ca/ca.key 4096
```

```
openssl req -new -x509 -days 3650 -key ca/ca.key -out ca/ca.pem
```

It will prompt for a passphrase in case the private key was encrypted which will be followed by questions related to the certificate:

- Country Name (2 letter code) [AU]:
- State or Province Name (full name) [Some-State]:
- Locality Name (eg, city) []:
- Organization Name (eg, company) [Internet Widgits Pty Ltd]:
- Organizational Unit Name (eg, section) []:
- Common Name (e.g. server FQDN or YOUR name) []:
- Email Address []:

The generated **ca.key** should be protected from unauthorized users while **ca.pem** can be shared with all clients as public to establish trust with the CA.

Create Server Endpoint Certificates

Create the server endpoint private key:

```
openssl genrsa -out server/server.key 4096
```

The next step is to generate a certificate request to be signed by CA:

```
openssl req -key server/server.key -new -out server/server.req
```

The generated certificate request (server.req from the example) should be passed to the CA admin to execute a command like:

For the first generated key:

```
openssl x509 -req -sha256
 -in server/server.req -CA ca/ca.pem -CAkey ca/ca.key -CAcreateserial -out server/server.pem
```

For the following instances, the existing serial file can be used:

```
openssl x509 -req -sha256
 -in server/server.req -CA ca/ca.pem -CAkey ca/ca.key -CAserial ca/file.srl -out server/server.crt
```

The generated server.crt file should be passed to the inference endpoint owner along with the server.key. It will be used to configure the endpoint TLS termination.

Configuring Ingress Endpoints

An assumption here is that Kubernetes is installed on the cluster with the nginx ingress controller. Examples described in this article were tested with Kubernetes version 1.10 and ingress-nginx version 0.17.1.

Ingress-nginx can be installed using the procedure from https://github.com/kubernetes/ingress-nginx/blob/master/docs/deploy/index.md.
Specifically, you would need to apply the Kubernetes records from
https://raw.githubusercontent.com/kubernetes/ingress-nginx/master/deploy/mandatory.yaml
which include a dedicated Namespace, ConfigMaps, ServiceAccount, Role and RoleBinding, ClusterRole and ClusterRoleBinding followed by a Deployment config.
Service configuration and the external exposure depends on the hosting environment.

The ingress controller will need the certificates to be stored in a Kubernetes secret which can be added with a simple command:

```
kubectl create secret tls secret_name --key ca/server.key --cert ca/server.crt
```

It is important to add the TLS secret in the same namespace where the ingress record will be added and the service backend is configured.

While the certificates are prepared and saved in a Kubernetes secret, the remaining step is to create the ingress records for the inference services, as in the example below.

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  annotations:
    allowed-values: CN=client
    kubernetes.io/ingress.class: nginx
    nginx.ingress.kubernetes.io/auth-tls-pass-certificate-to-upstream: "false"
    nginx.ingress.kubernetes.io/auth-tls-secret: ca-cert-secret
    nginx.ingress.kubernetes.io/auth-tls-verify-client: "on"
    nginx.ingress.kubernetes.io/auth-tls-verify-depth: "1"
    nginx.ingress.kubernetes.io/grpc-backend: "true"
    nginx.ingress.kubernetes.io/proxy-body-size: 64m
  name: openvino-service
spec:
  rules:
  - host: example.kubernetes.domain.com
    http:
      paths:
      - backend:
          serviceName: openvino-model-server
          servicePort: 9001
  tls:
  - hosts:
    - example.kubernetes.domain.com
    secretName: tls-certs
```

The important elements of the ingress configuration which should be explained here are:

- nginx.ingress.kubernetes.io/grpc-backend - This specifies how nginx should connect to the backend. In version 0.20+ it is replaced by *nginx.ingress.kubernetes.io/backend-protocol* annotation.
- *nginx.ingress.kubernetes.io/proxy-body-size: 64m* - Often, inference requests contain considerable volumes of data, so the default 4m values should be increased.
- The TLS secret name has to match the secret including the endpoint server certificate and its private key.

Authentication-related annotations will be explained in the next section. They are related to the client certificate validation and authorization.

If the backend services have enabled TLS encryption on the internal interface, two extra annotations must be used:

```
ingress.kubernetes.io/secure-backends: "true"
ingress.kubernetes.io/secure-verify-ca-secret: "ca-secret"
```

The second annotation must include a secret name with the base64 encoded content of the CA certificate used to sign the internal TLS endpoints on the service instances.

The TensorFlow Serving and OpenVINO Model Server versions tested here do not include an option to encrypt the traffic, but it is possible that such an option may be included in the future.

Client Authentication and Authorization

Enabling client authentication and authorization requires generating and exchanging client certificates and configuring ingress annotations included in the example from the previous chapter. They will be explained below.

Creating Client Certificates

To get started, we create a CA certificate which will be used to sign the client certificates. The CA certificate (without the private key) will be made available to the ingress controller via the nginx.ingress.kubernetes.io/auth-tls-secret option, so ingress will trust the incoming certificate.

The CA certificate for signing the client certificates could be created in the same process, like for the ingress endpoints presented earlier.

Next, the end user or the client owner generates the client key:

```
openssl genrsa -out client/client.key 4096
```

The following step is to generate certificate request to be signed by CA:

```
openssl req -key client/client.key -new -out client/client.req
```

The subject name (or the common name) of the certificate should be provided, which must match the name used in the allowed-values ingress annotation.

With that, the CA admin can issue the client certificate. For the following instances, the existing serial file can be used:

```
openssl x509 -req -sha256
 -in client/client.req -CA ca/ca.pem -CAkey ca/ca.key -CAserial ca/file.srl -out client/client.pem
```

Configuring Ingress for Client Authorization

The client certificate does not need to be added in Kubernetes secrets. It should be presented by the client establishing the connection with the inference system. Ingress will trust the client certificates based on the shared CA certificate used to generate the client certificate.

Every inference endpoint can authorize different clients, but it is also possible to share the certificate between endpoints and tenants as required.

You need to make sure the nginx-template includes the following lines in green to take advantage of client authentication by the certificate subject name:

```
set $ingress_name   "{{ $ing.Rule }}";
set $service_name   "{{ $ing.Service }}";
set $service_port   "{{ $location.Port }}";
set $location_path  "{{ $location.Path }}";
set $annotation_allowed  "{{ index $ing.Annotations "allowed-values" }}";

if ($ssl_client_s_dn != $annotation_allowed) {
 add_header X-SSL-Client-S-DN $ssl_client_s_dn always;
 return 403;
}
```

In the ingress records, there are additional annotations to be added:

```
   nginx.ingress.kubernetes.io/auth-tls-secret: ca-cert-secret
   nginx.ingress.kubernetes.io/auth-tls-verify-client: "on"
```

```
  nginx.ingress.kubernetes.io/auth-tls-verify-depth: "1"
  allowed-values: CN=client_cert_subject_name
```

They enable client certificate verification and configure in a Kubernetes secret the CA certificate which was used to sign client certificates.

```
kubectl create secret generic ca-secret --from-file=ca.crt
```

This way, ingress will trust the clients' certificates so the signing by the CA will ensure their validity.

The authorization itself is set via the ingress record annotation: allowed-values

With this setting, only the client using a certificate of the allowed subject name (CN) will be authorized to access the ingress endpoint. Other clients will be rejected and unauthorized on the ingress endpoint.

This can be tested using the gRPC client from the Appendix.

Note: In case the internal model server has TLS termination enabled for the service endpoint, two additional annotations are required in ingress records:

```
nginx.ingress.kubernetes.io/secure-backends=true
ingress.kubernetes.io/secure-verify-ca-secret: “internal-ca-secret”
```

The option secure-verify-ca-secret should include a secret including the CA certificate used to sign internal service endpoints on the Kubernetes pods. It can be created by:

```
kubectl create secret generic internal-ca-secret --from-file=ca.crt
```

Client Certificate Revocation

A client certificate can be revoked in case it has been compromised. This process allows recreating the client certificate with the same ‘common name’.

To make the CRL (Certificate Revocation List) feature work, it is necessary to update the nginx template to the include statement:

```
ssl_verify_client            {{ $server.CertificateAuth.VerifyClient }};
ssl_crl                      {{ $server.CertificateAuth.CAFileName }};
```

The commands to create an empty CRL are:

```
echo 01 > certserial

echo 01 > crlnumber

echo 01 > crlnumber

touch certindex

openssl ca -config ca.conf -gencrl -keyfile ca.key -cert ca.crt -out root.crl.pem
```

The command to revoke the example client certificate is:

```
openssl ca -config ca.conf -revoke client.crt -keyfile ca.key -cert ca.crt
```

In order to add a revoked certificate to CRL:

```
openssl ca -config ca.conf -gencrl -keyfile ca.key -cert ca.crt -out root.crl.pem
```

The command above will update the existing root.crl.pem file.

In order to add CRL support to ingress-nginx, append the content of root.crl.pem to the ca-cert-secret.crt file.

Remove the old CRL appended to ca-cert-secret.crt if it exists, and append new one with the following command:

```
cat ca-cert-secret.crt root.crl.pem >> temporary
```

Then, encode the created file and copy to the Kubernetes secret in place of ca.crt:

```
CA_CRT=$(base64 < "./temporary" | tr -d '\n')

kubectl get secrets ca-cert-secret -o json \

| jq '.data["ca.crt"] |= "$CA_CRT"' \

| kubectl apply -f -
```

Finally, restart the nginx controller pods:

```
kubectl patch deployment nginx-ingress -p \
 "{\"spec\":{\"template\":{\"metadata\":{\"labels\":{\"date\":\"`date +'%s'`\"}}}}}"
```

The above commands should result with 400 response for revoked clients.

```
CancellationError(code=StatusCode.CANCELLED, details="Received http2 header with status: 400")
```

Performance Analysis

Note: *T\The tests presented below were executed using TensorFlow Serving docker image from DockerHub* ‘tensorflow/serving:1.11.0’*
OpenVINO Model Server was built using the recipe from https://github.com/IntelAI/OpenVINO-model-server
The infrastructure was Google Compute Cloud using the GKE service with allocated nodes with Intel® Xeon® Scalable processors.*
All tests use the ResNet v1.50 model generated based on slim scripts from https://github.com/tensorflow/models/tree/master/research/slim with batch size 1 and float32 input data.*

Testing Ingress Overhead

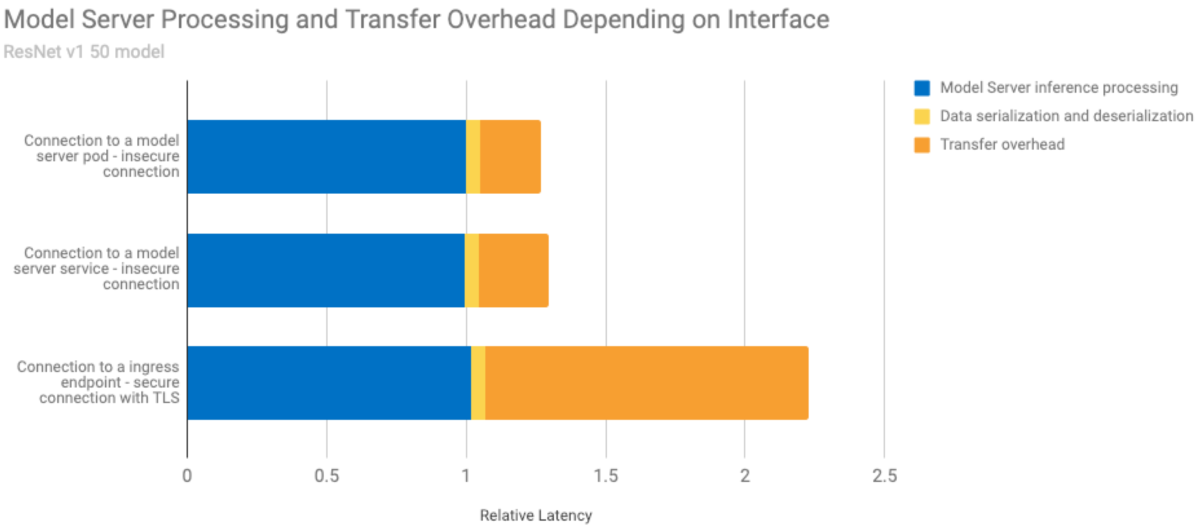
In our testing, we attempted to confirm if there are noticeable penalties from adding a load balancer between the client and the service backend instances.

To make the results realistic, the inference applications were executed using a gRPC client, submitting prediction requests to the OpenVINO Model Server. It was hosted on a pod with an Intel Xeon Scalable processor-based system with 16 virtual CPU cores and 32GB RAM, which should ensure the impact from transfer latency would be measurable compared with the inference execution time. Similar results could be captured with the TensorFlow Serving backend, but by using OpenVINO Model Server in the debug mode we could capture the time consumed on inference execution and data serialization.

The following conditions were tested:

- Connecting to the inference model server directly to the pod
- Connecting to the inference model server over the service internal Cluster IP
- Connecting to the inference model server via ingress over external IP

During the tests, the gRPC client was hosted on a pod in the same Kubernetes cluster, and the model server was serving the ResNet v1.50 model. Kubernetes infrastructure was set up in the GKE service. The gRPC client described in the appendix was sending sequential requests to inference endpoints and calculating response time statistics. There were 1000 requests in each sequence.



The latency was measured on the client so the transfer overhead includes the time needed to encrypt the inference input data using the TLS algorithm. It was observed that the load on the client side submitting the requests was 3 times higher while sending the data over an encrypted channel compared to an insecure one. The impact on the Kubernetes cluster infrastructure was negligible. The observed nginx ingress controller load from serving ~21MB/s (35 images/sec) was ~10% of a single CPU core. At the same time, the inference pod was consuming ~100% of 8 physical CPU cores (16 virtual CPU cores). This means the extra CPU load from the load-balancing is negligible, or on the level of 1-2%.

To sum up our testing, adding a L7 load balancer with TLS connection termination does have an impact on the latency and increased load on the client side. It will be proportional to the volume of data which should be passed to the server. On the other hand, it adds critical security features and efficient load distribution on all the service instances. As will be demonstrated in the following tests, it doesn't reduce the overall capacity of the inference system. It improves the system utilization; hence, the throughput goes up.

The conclusion is that connectivity over inference service cluster IP should be used when the clients are located inside the cluster, even if the distribution of the inference requests is not critical and there is no need to encrypt the traffic on internal interfaces. In all other cases, employing the ingress controller is the recommended technique of exposing the inference model servers.

Resource Allocation Analysis

In Kubernetes it is possible to configure pod resource allocation with an isolation between the containers or with shared resources.

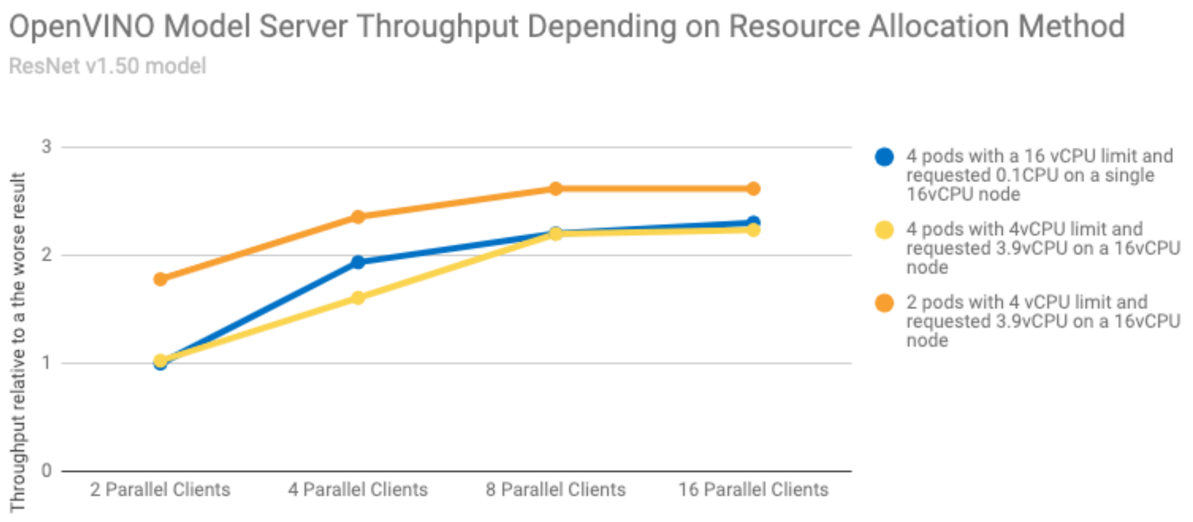
When we set the CPU request in the podspec and limit values as equal, we guarantee that Kubernetes will limit but also reserve the resources so it will be available when needed. The side effect might be that resources reservation might result in making them idle when they are not utilized.

Burstable (not setting limits but setting requests) is another alternative. This would allow CPU sharing to kick in and let the pod use the entire CPU if there are no other jobs. In some cases, it might cause competition from multiple processes over the same CPU cycles and in result degrade the performance.

The goal of the load testing was to verify this characteristic in inference execution using OpenVINO Model Server and TensorFlow Serving. The assumed load was coming from multiple gRPC clients sending sequential inference requests without intervals. The statistical results for the gRPC response time were verified under these conditions.

The tests were executed in Google Cloud GKE service using nodes equipped with 16 virtual cores of Intel Xeon Scalable processors and 32GB RAM.

OpenVINO™ Model Server


OpenVINO Model Server Throughput Depending on Resource Allocation Method

ResNet v1.50 model

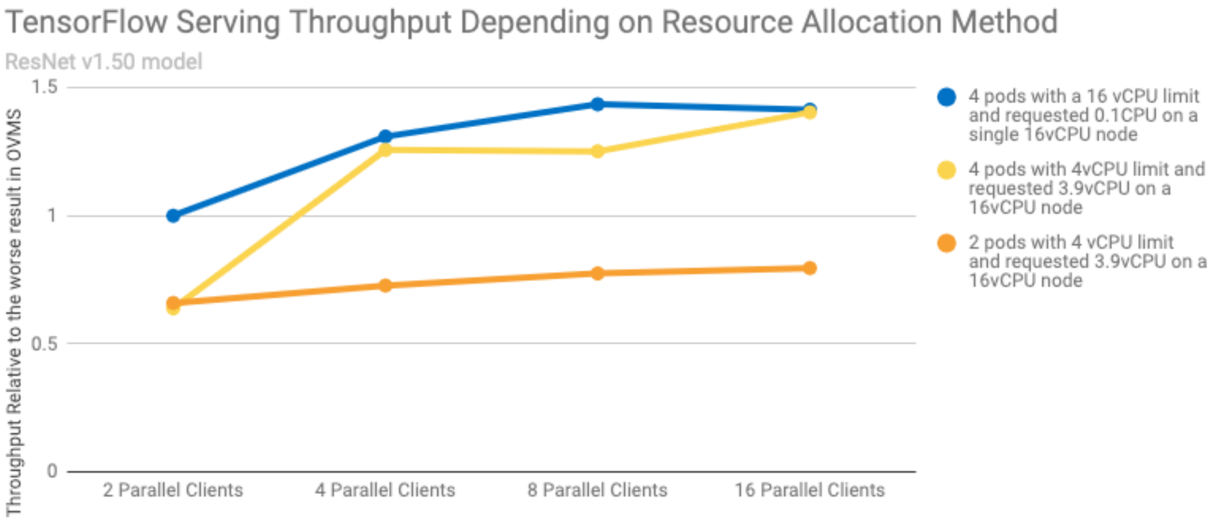
OpenVINO is generally recommended to be run without multithreading. With virtual cores we observed that allocating just half of the node capacity gives the best performance. Using multiple parallel connections gives the highest capacity. In this situation, containers are never idle, and data transfer is handled in parallel to inference processing from earlier requests.

When the containers have overcommitted capacity and the inference process needs to compete for resources, there is observed performance degradation, especially for a small number of parallel connections. This could leave some of the resources unutilized. When all the containers are utilized with multiple sessions, the overcommitment makes the difference in capacity about 10%. In most cases, it should be considered minor, given the benefits with the lower latency and better utilization of the irregular load and in shared servers.

When you need to guarantee a specific latency and performance, it makes sense to isolate the resources for each container. In most of the scenarios, when there are multiple inference services hosted on the Kubernetes nodes and not all are fully utilized, there will be faster average responses with oversubscription.

In OpenVINO Model Server we ensure it is not consuming too many virtual cores by setting the additional variable OMP_NUM_THREADS to the number of desired physical cores to be consumed - for example, with Kubernetes CPU limit to 8 cores set OMP_NUM_THREADS=4. The performance tests described later will use this variable as set according to this rule. On servers without hyper-threading features, it wouldn't be needed.

TensorFlow* Serving



TensorFlow Serving has different characteristics depending on the resource allocation patterns. It automatically detects the hardware configuration and sets threading parameters. With half the virtual CPU allocation, the maximum capacity is 50%. Also, sharing the resources for multiple containers running inference requests has not resulted in a negative impact to overall capacity.

Overallocation of the resources is most effective and recommended unless someone needs to guarantee certain latency for services regardless of the other load in the cluster.

Horizontal Scalability

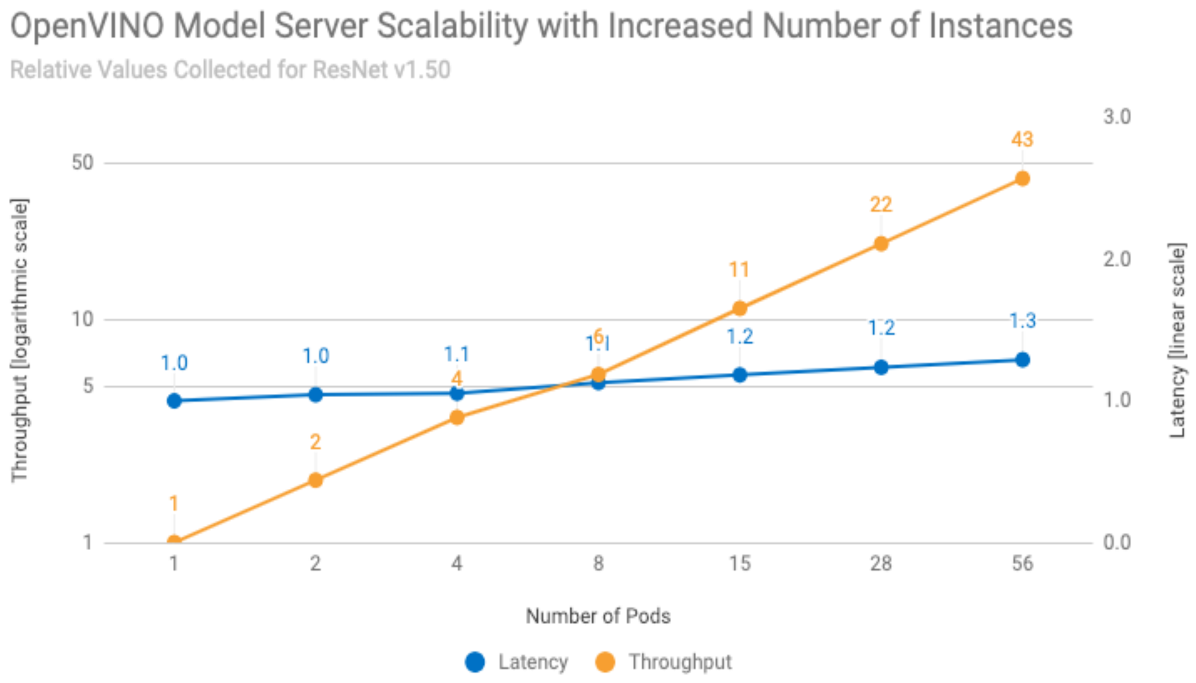
Horizontal scaling in Kubernetes and load balancing, in general, refers to increasing the system capacity by adding instances to the service in a form of nodes or Kubernetes pods. In the ideal situation, the capacity increase should be proportional to the number of created instances. In reality, there might be bottlenecks which at some point diminishes the returns. The goal of the load tests was to examine this characteristic for inference requests.

During the tests, the inference service hosted on Kubernetes with AI model servers in the backend was getting load from an increasing number of parallel clients, each sending sequential requests over gRPC interface. At the same time, the number of backend instances will be also increased, and we will capture statistics like average latency and throughput. Each instance of the service will be run as a pod with constrained and isolated CPU capacity of 4 cores.

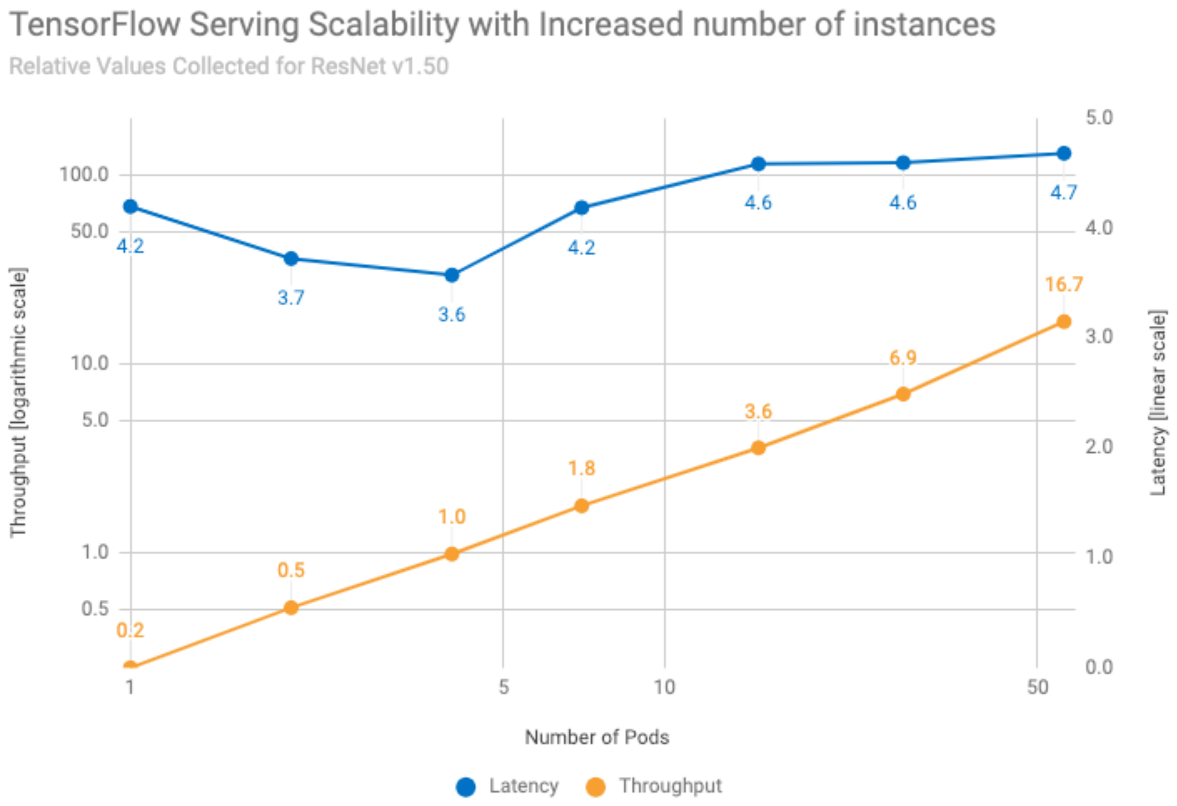
On the charts below, the left axis represents throughput in logarithmic scale while the right axis represents the average latency in linear scale. During the tests, the number of parallel clients was growing with the number of Kubernetes pods (2 clients per pod).

The tests were executed in Google Cloud GKE service using nodes equipped with 32 virtual cores of Intel Xeon Scalable processors and 64GB RAM.

OpenVINO Model Server



TensorFlow Serving



Based on our testing, we can conclude that both model servers are scalable horizontally in Kubernetes. It is quite easy to add capacity to the system by increasing the number of pods on a node or adding the nodes to the cluster.

The built-in capabilities in Kubernetes make it easy to adjust the capacity according to the dynamics of the load. Scaling up and down is quick and without any downtime for the inference service.

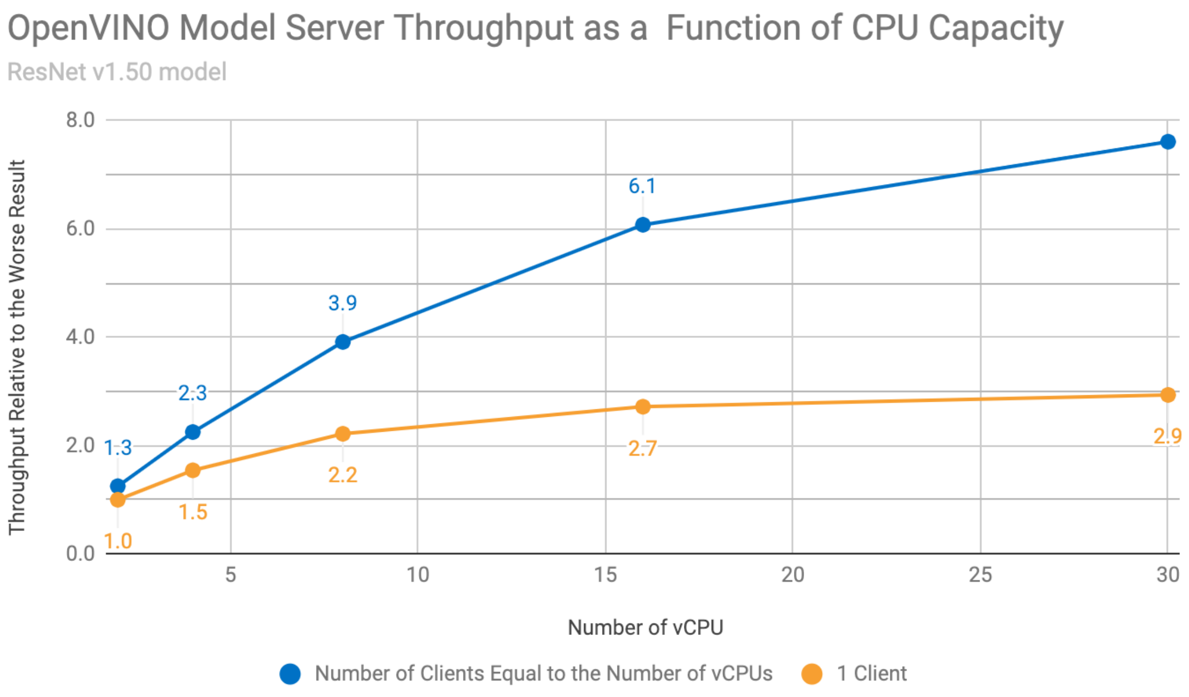
Besides scaling up the inference services, it is possible to scale the nginx ingress controller, but in the tests it was not required even with incoming traffic up to 2Gb/s. With increased traffic, a second pod was added to make sure it did not become a bottleneck.

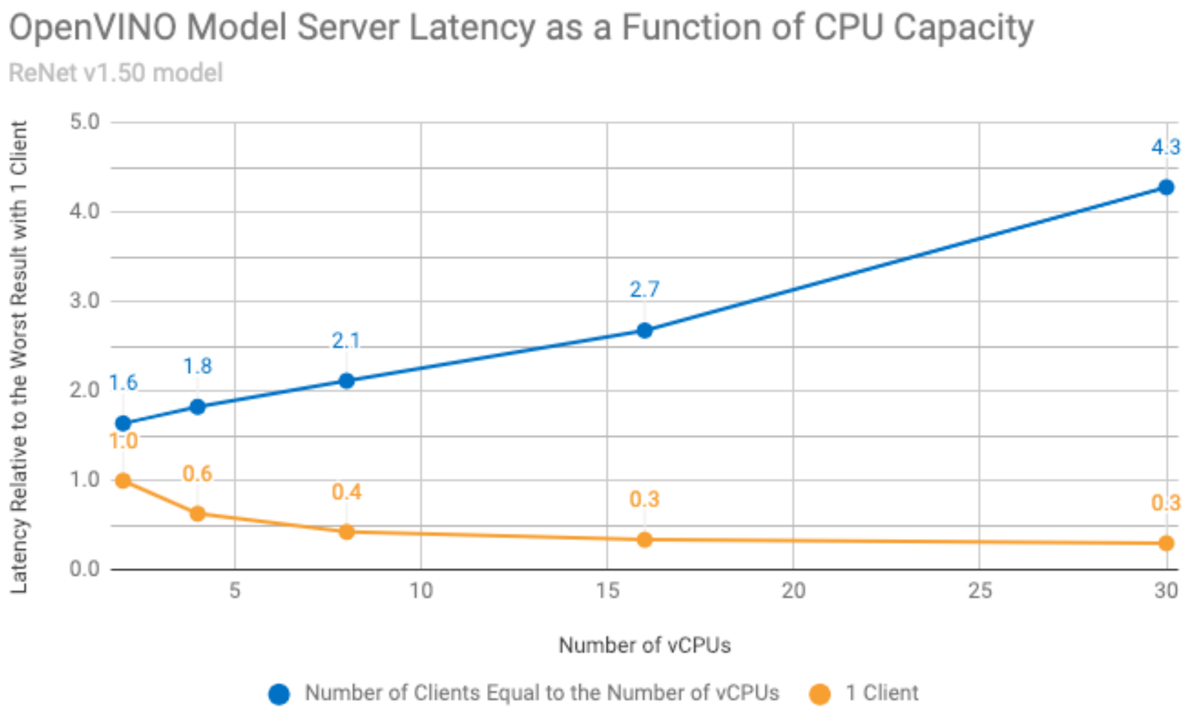
It should be also taken into account that some capacity needs to be reserved on the workers for Kubernetes services and host kernel processes so allocating complete node capacity to inference services will not increase the throughput. Leaving ~5% of the capacity unreserved might have the optimal results, especially if the inference requests include a big volume of data, as with visual computing.

Vertical scalability

Vertical scaling describes the process of increasing the capacity of individual server instances and keeping their number constant. Typically, it means replacing hardware with a faster CPU or inference accelerators. It could be also just allocating a larger number of CPU cores and processing units to a single instance. Below are load testing results for TensorFlow Serving and OpenVINO Model Server. A scenario was analyzed with a single client submitting sequential requests and multiple of such clients with the number proportional to the capacity.

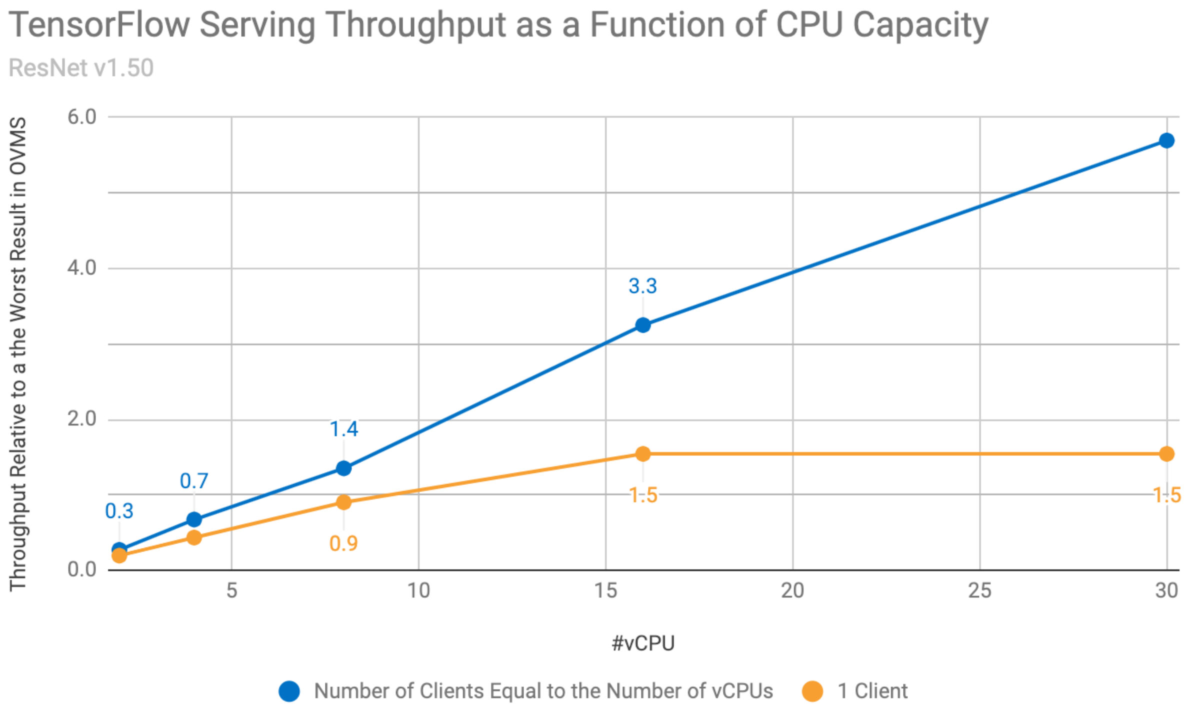
The tests were executed in Google Cloud GKE service using nodes with 32 virtual cores of Intel Xeon Scalable processors and 64GB RAM.

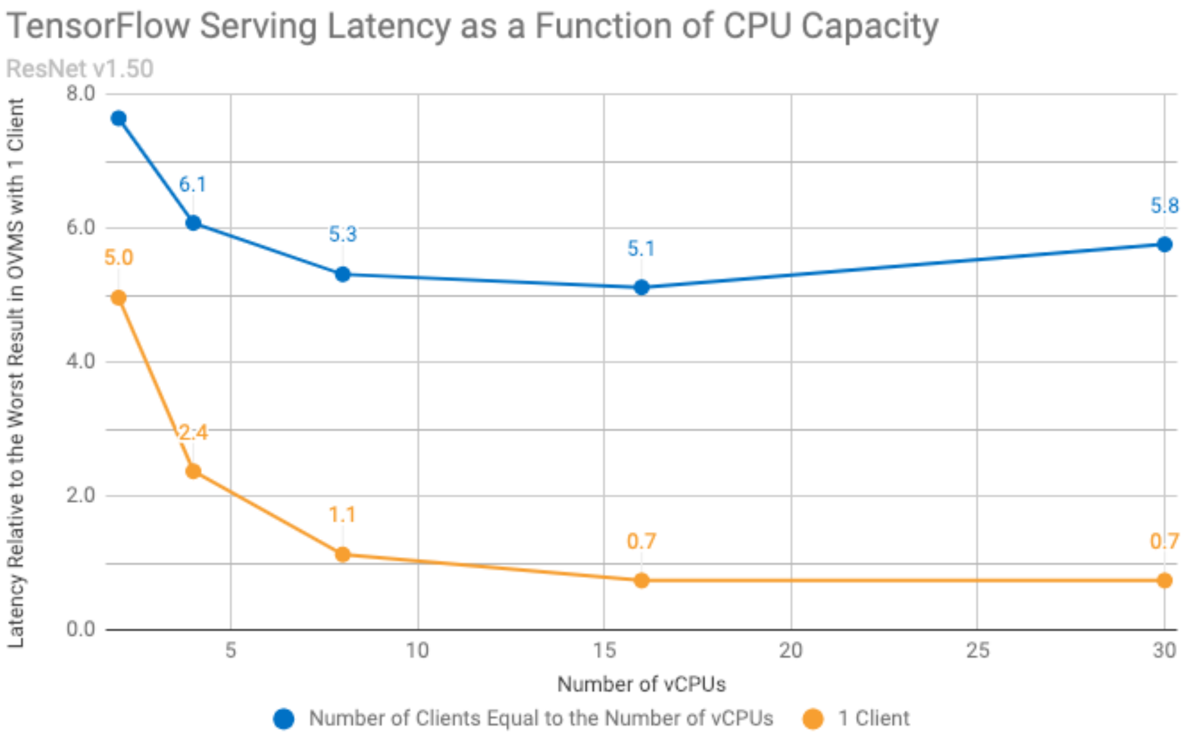




During the tests, the environment variable OMP_NUM_THREADS was applied equal to the half of assigned vCPUs. This ensures the inference service consumes resources up to defined limits. We can observe that in single user mode performance tests, the throughput scalability is limited, which is to a large extent related to the network transfer overhead and latency on the ingress load balancer. Because of that, a single client is not able to permanently consume all the CPU resources. We can, however, achieve higher throughput with multiple parallel clients. Nevertheless, the tests show that the most efficient instance configuration in terms of throughput per CPU core is with values up to 8. The Kubernetes cluster throughput can be increased by using more but smaller pods.

TensorFlow* Serving





TensorFlow Serving has a slightly different vertical scalability characteristic. Just like OpenVINO Model Server, it has limited scalability in single user mode. It has, however, a linear characteristic for bigger instance capacity with a multiuser mode. The best throughput per core was observed with the allocation of 16 vCPUs with negligible degradation with 32 vCPU. The difference in the characteristic is due to the TensorFlow feature of automatic tuning of parallelism in the computing operations based on the available capacity. The throughput can be attributed to this and is similar regardless of the allocated resources.

Conclusions and Recommendations

To summarize the tests, inference services can be enabled in Kubernetes in a highly scalable and secure way. It can be used to serve in a flexible manner a large number of models for a huge number of clients and requests.

There are obviously a lot of caveats to be aware of and take into account while deploying and managing Kubernetes infrastructure.

A variety of model servers can be adopted this way, such as TensorFlow Serving and OpenVINO Model Server. Each of them has its strengths, so consider using them depending on the model frameworks, topologies, and usage pattern. It's always the best practice to test the solutions in real-life scenarios. We hope this article will make the ramp up and initial deployment easier and more optimal.

If you are looking for simplified Kubernetes configuration for inference endpoints, Intel has recently released an open source component, **Inference Model Manager for Kubernetes**. It enables an easy-to-follow REST API for managing and controlling the endpoints in multi-tenant environments using OID (OpenID) token-based authentication. It abstracts the nuances of Kubernetes configuration away from the end users, so they can focus on data science challenges and easily deploy in production AI models and expose them at scale.

References

Presentation: gRPC Load Balancing on Kubernetes (Jan Tattermusch, Google)
Blog Post: Using Envoy to Load Balance gRPC Traffic (Mike White, Bugsnag)
gRPC Concepts: Overview
gRPC Concepts: Load Balancing
gRPC Client Code
Kubernetes Concepts: Services
Kubernetes Concepts: Ingress
Google Cloud: HTTP(S) Load Balancing Concepts
@bbengfort Development Journal: Secure gRPG with TLS/SSL
How to Set Up Mutual TLS Authentication to Protect Your Admin Console
Configuring TensorFlow Serving service in Kubernetes
Configuring OpenVINO Model Server service in Kubernetes
Nginx-controller deployment

Notices and Disclaimers

Intel technologies' features and benefits depend on system configuration and may require enabled hardware, software or service activation. Performance varies depending on system configuration. No computer system can be absolutely secure.

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.

Performance results are based on internal testing done on 27th November 2018 and may not reflect all publicly available security updates. See configuration disclosure for details. No product can be absolutely secure. Test configuration: Google Kubernetes Engine service using nodes equipped with Intel Xeon Scalable processors of variable cores, capacity, and RAM allocation.

Optimization Notice: Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.