Concept Study «Cloud-optimized OGC WMS Server»

Nr. NGDI N° INDG	20-02
Titel Titre	Concept Study «Cloud-optimized OGC WMS Server»
Antragsteller (Organisation) Demandeur (organisation)	Kanton Solothurn
Antragsteller (verantwortliche Person) Demandeur (personne responsable)	Andreas Neumann
Projektpartner (Organisation) Partenaire de projet (organisation)	Kanton Solothurn, Stadt Zürich, Ville d'Yverdon, Rudaz + Partner
Projektleiter Responsable de projet	Andreas Neumann
Projektkoordinator (PROK) Coordinateur de projet (PROK)	Pasquale DiDonato
Kontaktperson KOGIS Personne de contact COSIG	Christine Najar
Vertragssumme inkl. MWST Montant contractuel avec TVA	CHF 32'310.-
Vertragsende Fin du contrat	31.05.2021

Authors of the study: Marco Bernasocchi (OPENGIS.ch GmbH), Paul Blottiere (QCooperative), Andreas Neumann (Kanton Solothurn), Alessandro Pasotti (QCooperative), Robert Pupel (devs group GmbH), Vincent Trötschel (devs group GmbH).

1 Introduction

Motivation

OGC WMS servers have been traditionally deployed on relatively large dedicated bare-metal or virtual servers in the past. With the ongoing move of services into cloud infrastructures, OGC WMS Servers are increasingly migrated to container environments where containers run single services and should be more «scalable» when system load varies (increasing or decreasing numbers of requests). The move to containers infrastructures solves some issues regarding scalability, but introduces other challenges.

This study analyzes the problems when moving traditional OGC WMS servers into container environments and suggests potential strategies to overcome these challenges.

Container Orchestration - Why Kubernetes?

Containers provide us with high flexibility for running cloud-native applications on physical and virtual infrastructures. Containers are meant to package up the services and dependencies which are required to run a given executable program.

Because a running web application often needs external dependencies like database drivers or system wide configurations, the application needs to run inside an encapsulated environment. Containers are meant to provide such an environment. They allow us to start multiple instances of packaged running applications on a single machine, without caring about running the dependencies in parallel with different versions. Another significant advantage is the possibility to spread the containers around multiple machines while allowing communication in between. This concept allows us theoretically to scale systems without any limits of resources. To manage containers on machines, another level of abstraction is needed to have a source of control and a better overview of the whole system consisting of multiple containers.

This level of abstraction is often called orchestrator. There are multiple container orchestrators like Apache Mesos, Docker Swarm, HashiCorp's Nomad, or Kubernetes. These orchestrators are starting to resemble each other regarding features and functionality. Still, Kubernetes is becoming the **de-facto standard** due to its architecture, innovation, and the sizable open-source community around it.

Kubernetes allows us to configure applications using YAML files and deploy them securely to a cluster. It also comes with considerable advantages of already implemented functionalities to replicate containers, automatically scale their needed resources, load balance network-requests, or rolling out updates with the guarantee of uninterrupted deployments.

Glossary

Node:

A Node is a worker machine in Kubernetes and may be either a virtual or a physical machine, depending on the cluster. The Master manages each Node. A Node can have multiple pods, and the Kubernetes master automatically handles scheduling the pods across the Nodes in the cluster.

Cluster (Kubernetes):

A kubernetes cluster is a set of connected nodes.

Container:

Containers are a form of operating system virtualization. A single container might be used to run anything from a small microservice or software process to a larger application. Inside a container are all the necessary executables, binary code, libraries, and configuration files.

INGRESS / EGRESS:

Ingress is a kubernetes resource for incoming traffic to the pod, and egress is outgoing traffic from the pod. In Kubernetes network policy, you create ingress and egress “allow” rules independently (egress, ingress, or both).

IPVS (load balancing):

IPVS facilitates transport-layer load balancing implemented inside the linux kernel, relying on the Linux netfilter framework which is also utilized by iptables. It is a mature feature, in the mainline linux kernel since 2.4.x and used in highly scalable and highly available e-commerce and e-government applications. It can be configured in userland through the command line tool *ipvsadm*, it requires superuser privileges on the machine.

Load Balancing:

Load balancing refers to efficiently distributing incoming network traffic across a group of nodes or pods.

Pod:

Pods are the smallest, most basic deployable objects in Kubernetes. A Pod represents a single instance of a running process in your cluster. Pods contain one or more containers, such as Docker containers. When a Pod runs multiple containers, the containers are managed as a single entity and share the Pod's resources.

Proxy:

Proxy is a gateway between the requester and the destination of an incoming request.

This serves as a method to simplify or control the complexity of the request, or provide additional benefits such as load balancing, privacy, or security.

PV: A PersistentVolume (PV) is a piece of storage in the cluster that has been provisioned by an administrator or dynamically provisioned using Storage Classes. It is a resource in the cluster just like a node is a cluster resource. PVs are volume plugins like Volumes, but have a lifecycle independent of any individual Pod that uses the PV. This API object captures the details of the implementation of the storage, be that NFS, iSCSI, or a cloud-provider-specific storage system.

PVC: PersistentVolumeClaim (PVC) is a request for storage by a user. It is similar to a Pod. Pods consume node resources and PVCs consume PV resources. Pods can request specific levels of resources (CPU and Memory). Claims can request specific size and access modes (e.g., they can be mounted ReadWriteOnce, ReadOnlyMany or ReadWriteMany, see AccessModes).

While PersistentVolumeClaims allow a user to consume abstract storage resources, it is common that users need PersistentVolumes with varying properties, such as performance, for different problems. Cluster administrators need to be able to offer a variety of PersistentVolumes that differ in more ways than just size and access modes, without exposing users to the details of how those volumes are implemented. For these needs, there is the StorageClass resource.

Orchestrator:

In the field of system administration, orchestrator is the automated configuration, coordination, and management of computer systems and the running software.

ReplicaSet:

A Set of pods which are indistinguishable from one another (stateless). The kubernetes scheduler will ensure that the number of those pods is kept constant by starting new replicas whenever one pod stops. A ReplicaSet is usually employed through a *Deployment* this allows for Services to manage routing of requests to any pod in the ReplicaSet and providing a single target for requests to be sent to.

Service (Kubernetes):

A Service in Kubernetes is an abstraction which defines a logical set of pods and a policy by which to access them. Services enable a loose coupling between dependent pods and allow the access from Ingress to a specific pod.

Scheduler (Kubernetes):

The kube-scheduler is responsible for starting pods. It decides how many pods should be created and on which nodes. It takes into account the resources available to the cluster.

2 Analysis of the challenges when running OGC WMS servers in container environments

2.1 - Architectural Questions & Resource Management

Should the WMS process inside a container be single- or multithreaded?

The document “QGIS server Benchmarking report” (Reference 1) concludes that the best performance is achieved by running WMS processes in single threaded mode. The following paragraphs outline considerations on how to set up an architecture in kubernetes to most efficiently process multiple requests in parallel.

Docker containers are just an abstraction on various kernel namespaces, the runtime overhead versus running a process outside Docker is minimal. However spinning up a container with a WMS process takes longer and consumes more resources than the creation of a child process.

Hypothetical: One Container per Request

Spinning up one container for each individual request would require the preemptive creation of containers sitting idly until their services are requested.
This approach would be similar to a thread pool only with containers or, on a kubernetes structure, with pods. This is not a construct kubernetes was built to orchestrate, it’s scaling mechanism does not target the number of idle pods, but the average load on all pods.

To be able to flexibly react to highly fluctuating request loads, the number of currently idle containers would have to be reevaluated at high rates, in practice this rate would be limited by the container startup time. Therefore depending on it, even in conjunction with a warm caching solution (see chapter 2.2), the scaling response might be inappropriate for highly fluctuating demands. Furthermore, if other applications are running on the same cluster this approach would exhaust its resources faster since spare system resources must constantly be reserved, regardless of the load.

Recommendation: Multiple Requests per Container

To make the best use of kubernetes’ scaling features it is imperative for a container to have the ability to handle multiple requests at the same time. Otherwise kubernetes’ inherent scaling and load balancing patterns are less efficient, see the section “Resource Management - Scaling".

It is therefore advised for every container to start its own FCGI process for each request in order to allow for parallel request handling.

This is why it makes more sense to run WMS processes within a web server which is able to start a child process for each new incoming request. And run the WMS server in single threaded mode.

Still, it's highly recommended to run multiple containers of a service to have better flexibility for scaling, updating resource needs, and rolling out deployments. Later in this document, we describe VPA and HPA. These patterns are very resource-friendly as we can run only as many containers as needed in a given time and reserve only needed resources. This is not only environment friendly but can also save a lot of costs when services are running on a public cloud.

Should each WMS server container have a web server?

Running multiple processes in the same container is not only a bad practice, but also introduces its own problems. In Kubernetes with the concept of pods, we are able to run multiple containers inside one pod. Because the containers inside one pod are sharing the same NIC (network interface controller) the containers are accessible to each other over localhost. In this way, we are able to use a web server like nginx or apache with fcgi module enabled as a middleware to access the WMS server container. That means that we recommend to deploy the WMS server container and the web server container in one pod.

After some research we found out that one of the best options is to use the "apache" web server with additional "mod_fcgid". With the apache mod_fcgid, any program assigned to the handler fcgid-script is processed using the FastCGI protocol. mod_fcgid starts a sufficient number of instances of the program to handle concurrent requests, and these programs remain running to handle further incoming requests. This is significantly faster than using the default mod_cgi or mod_cgid modules to launch the program upon each request. However, the programs invoked by mod_fcgid continue to consume resources. Therefore the administrator must weigh the impact of invoking a particular program once per request against the resources required to leave a sufficient number of instances running continuously. The benefit of this solution is that mod_fcgid is not only highly configurable (for e.g. by setting maximum number of processes, timeouts, limit of resources) but also because it terminates the running processes automatically after they are not needed anymore.

With nginx we need further work for postponing the application processes. To allow the same mechanism it's needed to use the fcgiwrap package. This is still not the best solution, because fcgiwrap runs the wrapped programs as CGI programs which is very slow because CGI is only able to run one process per request. A better choice for using nginx is to use spawn-fcgi which is a thin wrapper around a FCGI program and creates FCGI sockets in order to handle multiple requests. This solution needs some additional work because it is needed to install spawn-fcgi on the environment and write a systemd service for it. Luckily the widely used docker images docker-qgis by Oslandia and oq-qgis-server by the GEM foundation and OPENGIS.ch have this work already done. These images expose a FCGI TCP socket.

Choosing between Apache + mod_fcgid or NGINX + spawn-fcgi is likely based on a personal choice by the developers since both solutions provide a solid solution.

Static Files - Raster File Images

WMS servers often require static files to be able to serve all requests. Examples include raster imagery, svg symbols, fonts or additional vector data that are not stored in a DB. The most striking example is raster files for aerial imagery or base maps.

One principle to build scalable applications is that we should avoid local state and shared state on a disk by all means. For good scalability it's recommended to put static files to a separate service. In a 12-factor app this kind of service is called "backing service". A backing service is a service which is consumed by the app over the network as a part of its normal operation. In case of static files this service should be a binary asset service like for e.g. Amazon S3, MinIO or Seaweedfs. To make this possible it's necessary to add either a layer to the server which communicates with such a backing service to save and load static files or possibly make use of COGEO (Cloud optimized geotiff) format (Reference 6).

Because the WMS server is only reading files, the usage of a so-called shared volume where all pods are mounted to the same persistent volume path, would be also feasible and the pods could stay scalable without race conditions. This would be an approach which works with less effort, but it's not a best practice cloud native approach.

Because further work is required to make use of a "backing service" to handle the files, we recommend to make use of the standard kubernetes shared volume approach.

Resource Management - Scaling

Following the kubernetes design principle one containerized WMS would be deployed in one pod. Therefore the question of resource allocation per container becomes a question of allocation per pod in a kubernetes context.

The CPU resources this pod has access to, can be set in terms of millicores and the memory in MiB RAM.

Application Scalability is achieved by controlling either the number of pods between which the load is balanced (horizontal scaling) or by manipulating the resources a pod can access (vertical scaling) or in certain cases both. To be able to assess current load per pod kubernetes includes the Metrics API, this API constitutes a cluster wide access to the busyness of the pods.

For the sake of completeness it should be mentioned that kubernetes can also scale on the infrastructure level by adding physical nodes to ensure the cluster does not run out of resources. By employing the Cluster Autoscaler this process can be automated. Since this research does not target the specific questions of resource allocation per pod, this will not be covered in more depth in the scope of this document. But it can be a solution to increased

cost-efficiency if the cluster is running on a commercial public cloud.

The different approaches are introduced and their applicability discussed afterwards.

Horizontal Pod Scaling

While scaling the number of pods can easily be achieved manually, this approach produces a considerable workload for supervising the resources. To facilitate this process kubernetes allows for the employment of a Horizontal Pod Autoscaler (HPA).

It allows for the declaration of a limit of a certain metric at which to spin up an additional pod automatically. By default the monitored metric is the CPU load but custom metrics and external metrics (by third party applications) can be utilized.

To supply all autoscalers with the memory & CPU usage data for each individual pod, kubernetes includes the Metrics API that can be utilized by deploying the Kubernetes Metrics Server to run on top of it. This project is maintained as part of the kubernetes infrastructure by the kubernetes developers. For different metrics the custom or external metrics APIs have to be utilized.

The number of replicas ensured by a HPA are calculated similarly (neglecting some constraints) to:

desiredReplicas = ceil[currentReplicas * (currentCPULoad / desiredCPULoad)]

To make the best use of horizontal pod autoscaling, an educated decision must be made to assign appropriate maximum resource limits that a single pod can access.

While lower resource limits on pods allow for a high amount of horizontal control, leading to a more efficient allocation of cluster resources, higher limits impose less overhead.

Scaling - Vertical Pod Scaling

Adjusting the resources requested by a pod is called vertical scaling. The tool to automate this task is called the Vertical Pod Autoscaler (VPA) (Reference 5). VPA is an infrastructure service that automatically sets resource requirements of pods and dynamically adjusts them at runtime, based on analysis of historical resource utilization, amount of resources available in the cluster, and real-time events, such as out of memory (OOM).

At its heart it creates a recommendation model that can be reviewed to manually set resource requests or be applied automatically via the kubernetes admission controller. For this to happen a pod will be evicted and recreated under the hood, so it can only work with multiple pods so as to not disturb current operations.

To create a recommendation model the metrics are polled at a configurable frequency

(depending on how spikey the workload). It treats memory and CPU consumption as independent random variables with distribution equal to the one observed in the last 8 days (to capture weekly periodicity) and calculates resource limits depending on the metrics to optimize for. VPA depends on prometheus as a storage for this history data.

For CPU the objective is to keep the fraction of time when the container usage exceeds a high percentage (e.g. 95%) of request below a certain threshold (e.g. 1% of time).

For memory the objective is to keep the probability of the container usage exceeding the request in a specific time window below a certain threshold (e.g. below 1% in 24h).

This process can be regulated by declaring a policy. These policies control how the autoscaler computes recommended resources and can be adjusted at runtime.

The resource policy may be used to set constraints on the recommendations for individual containers. If not specified, the autoscaler computes recommended resources for all containers in the pod, without additional constraints.

Scaling - Multidimensional Pod Scaling

At this moment a concurrent deployment of HPA & VPA on Memory and CPU is not possible. This is due to entanglement of responsibility and the lack of the replica count in the recommendation model of the VPA.

It is however possible to use the HPA on the custom metrics API and optimize for metrics such as network performance or IO.

Scaling - Applicability to OGC WMS Services

Separating application and state is a central paradigm of a cloud native infrastructure. With it comes the ability to scale. Therefore implementing a shared cache is imperative to achieve a scalable WMS application. Without it, the current long startup times would delay pod creation. To still guarantee pod availability as soon as the load requires it, pod creation would have to be triggered earlier, at lesser load. This in turn would produce a higher probability of a *scaling miss*, meaning the improper allocation of resources due to imprecise scaling.
This way a warm caching solution is beneficial to resource efficiency.

A detailed discussion about concrete caching implementations can be found in chapter 2.2.

Without a warm cache implementation or with otherwise somehow long pod bootup durations it should be taken into account that race conditions can occur when the HPA schedules pods to spin up and polls the average load before these are up and running, causing more pods to be scheduled for creation. This has to be worked around by setting delays after scaling events.

In the following discussion a proper shared cache solution will be assumed.

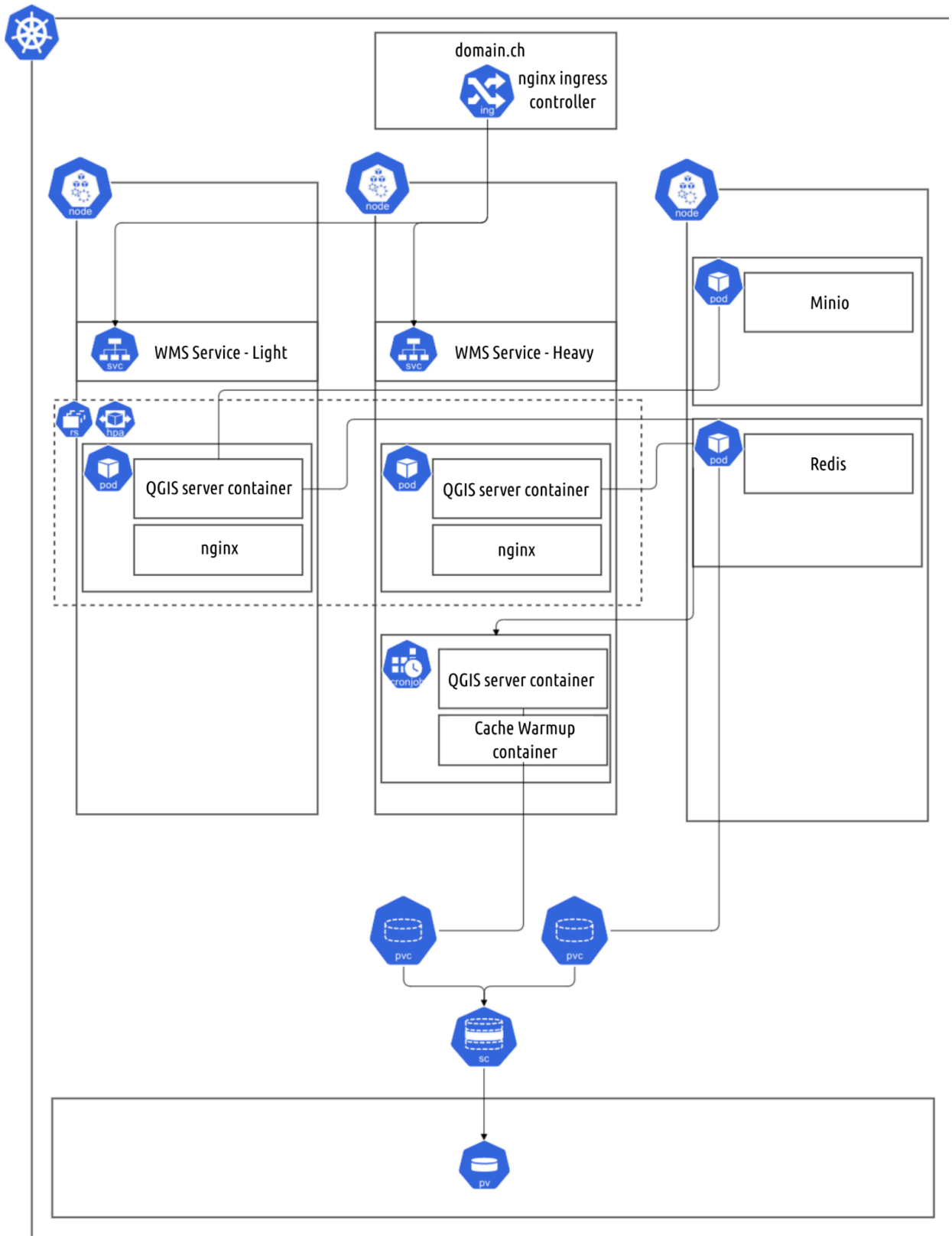
Taking into consideration the division-of-labor approach described in the next paragraph, different auto-scaling options can be employed for the different ReplicaSets.

Considering the apparent better performance of multiple single threaded WMS instances rather than fewer multi-threaded instances a larger number of horizontally scaled small pods seems to be preferable. For requests that require a heavier load on a single thread (GetPrint..) a vertically scaled approach can be considered.

Vertical scaling can be a more cost-efficient solution if a cluster-autoscaler is employed on a public cloud, because it can fit pods better into the resources of the available nodes without falling back to adding a new node to the cluster. Without the ability to scale nodes on a resource limited cluster, a VPA might result in a more efficient usage of resources.

However a more traditional, **HPA-only solution** is also feasible. It would lack the precision of a statistically backed recommendation model and will incur some overhead in terms of efficient resource allocation. On the other hand it is the more battle tested approach that is known to integrate well with other technologies that might be incorporated.

Architecture - An Outline



The above picture outlines a draft of how to approach a WMS Server application deployment into a kubernetes cluster (outermost box). Inside it the ingress is responsible for accepting

incoming traffic and routes it via kubernetes Services to the backend pods that are running the WMS Servers.

A central concept, that is unique to this specific application, is a division of labour. To achieve reliably fast responses for light requests in a situation of resource scarcity heavy load requests are meant to take the hit, since speed shortcomings are expected to be more accepted by the users this way.

Since all backend pods are indistinguishable the distinction is an architectural one. Pods are grouped per their function into two ReplicaSets and therefore scalable independently of one another (within their individual resource limits). Consequently two different autoscalers will be responsible for this.

To make sure requests find their appropriate target, both ReplicaSets will sit behind individual Services that assume the load balancing functions and receive their requests from the ingress.

2.2 - Slow startups and reading of configuration files

One issue that is inherent with some OGC WMS implementations is the fact that startup and initial readings of the WMS server instances are taking a long time. As an example, the initial reading of a large QGIS Server project in the Kanton of Solothurn with approx. 1000 WMS layers can easily take 2 minutes to load and parse. With the isolation of single CPU WMS server instances, each running in it's own container and with no sharing of configuration information between the instances, this has several drawbacks:

1. Server instances cannot start fast and ghus quickly react to an increased or decreased demand
2. As a follow-up issue to 1, instances need to remain idle in times of low demand, because they need to remain active when the demand increases again. In a dynamic environment, such as a deployment in a public cloud infrastructure, this generates additional costs with no benefits.
3. Rereading a new configuration takes a very long time, when many instances have to re-read the new configuration in parallel, and it creates a peak demand on database and file resources that have to be read and investigated during the initialization phase of projects and layers. Every instance is isolated on it's own without sharing the information or efforts.

Scaling horizontally is thus difficult … because of the above mentioned issues.

Possible solutions to improve startup time and better share configuration information can be introduced by

- an analysis of the startup process to see if it can be accelerated.
- an analysis of the parsing of project and layer configurations, with the goal of finding potential areas for performance improvements
- an establishment of a centralized project and layer cache.

Analysis of the startup process of a WMS server

By example of QGIS server, the startup process of a WMS server has been analyzed. The process can be broken down into 3 main steps:

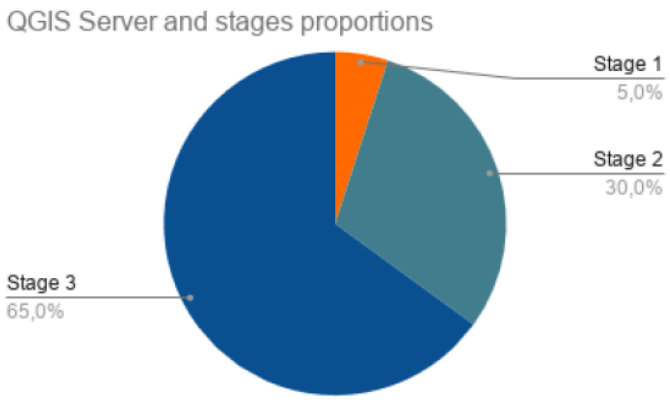
Stage 1: startup of the runtime environment (loading of executables, libraries, plugins, . . .)
Stage 2: reading a project configuration
Stage 3: generating the initial service response (GetCapabilities response)

The startup process has been profiled with a large project (approximately 800 Postgis layers) and produced the following results:

Starting the FastCGI process (stage 1) takes about 600 ms while the stage 2 and 3 take about 26 seconds. Thanks to the Hotspot tool, we can have a clearer idea of the situation for the two last steps:

Flamegraph (hotspot profiling tool) of the QGIS-Server startup process with a large project.

Considering that we're not interested in the exit stage, the situation of the proportions of the three stages during startup can be summarised with the chart below.



A detailed analysis of the three stages can be found in a separate document titled “QGIS Server and startup time in a cloud environment”. In addition to the profiling and study of the startup process, along with suggestions on how to improve the performance in future releases, this study also compared various data providers and their effect on startup time.

Stage 1: Startup of the process

This stage is already the shortest of all stages. With around 5% (600ms) there is not a lot of room for improvements. The above mentioned study lists a few optimization strategies around library, data provider and authentication method loading that can help us gain a few milliseconds during this stage.

Stage 2: Reading a project

This stage consists mainly of 2 subtasks:

1. Parsing the configuration document
2. Creating internal objects (map layers, project, attribute field names and properties, etc)

Concerning step 1 it is a fact that the configuration file format and performance of the file format parser used by the OGC WMS server is relevant for the performance of reading project and layer information. In the case of QGIS Server the configuration file format is based on XML. Parsing XML can be relatively slow, compared to other file format alternatives, and the QtXML parser seems to be slower than alternative XML parsers. UMN Map server uses a text-based configuration file using sections and indentations. The QGIS file format also supports a zipped version of the project file format (.qgz). It was discovered that zipping the file has a negative performance impact on the project loading. Replacing the XML based QGIS project file format with a binary format (as an alternative to the XML format and not as a replacement) could potentially bring a substantial performance boost in this stage 2 (reading the project).

In a separate parallel project, the QGIS code base is being examined for potential performance improvements - and mostly already fixed. In the project reading phase there is still some potential for skipping more project configuration settings that are irrelevant to the QGIS Server (e.g. settings for editing and snapping, styling backup copies, etc.). The QGIS 2.x server generation had a separate project file parser dedicated to server only. In the QGIS 3.x. series, this has been replaced by a general purpose project parser that handles all cases, but was less optimized for the QGIS server use case. While this improved code maintenance, the performance of the QGIS server startup stage suffered from this move. As a consequence, there are now code blocks that are skipped in cases where the part of project configuration is of no interest for the server use case.

Another area for potential improvements is the replacement of Qt DOM API calls with more efficient alternatives, as the Qt library seems not to be fully optimized for performance in this part of the Qt code.

Stage 3: generating the initial service response (GetCapabilities response)

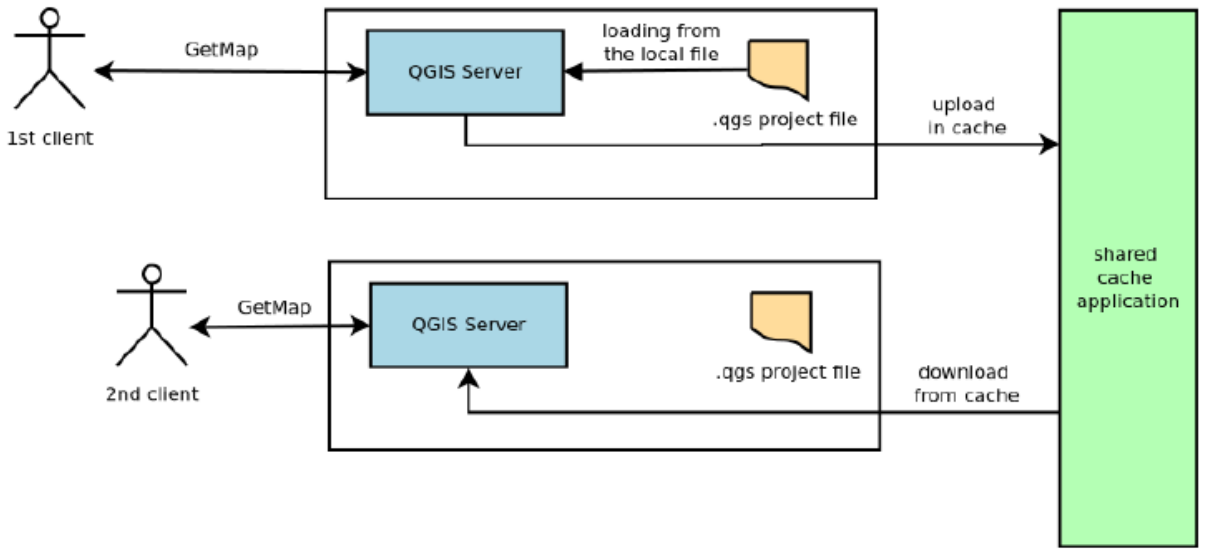
In this final stage, there is some room for improvement mainly in the following aspects:

- WGS84 reprojections (instead of recalculating layer bounding box coordinates for each request, the values could be cached in the project configuration). In the large project from Kanton Solothurn, this decreased the time being spent on the GetCapabilities response creation by 6%.
- Layer extent calculations (skip unnecessary feature counts)
- Skipping uniqueness checks and other constraints (trust metadata). Already implemented.
- Data provider influence: using PostgreSQL as a data provider might have some room for improvement, because it could be benchmarked that the project loading and GetCapabilities phase is substantially faster for Shapefiles than it is for PostgreSQL data sources.

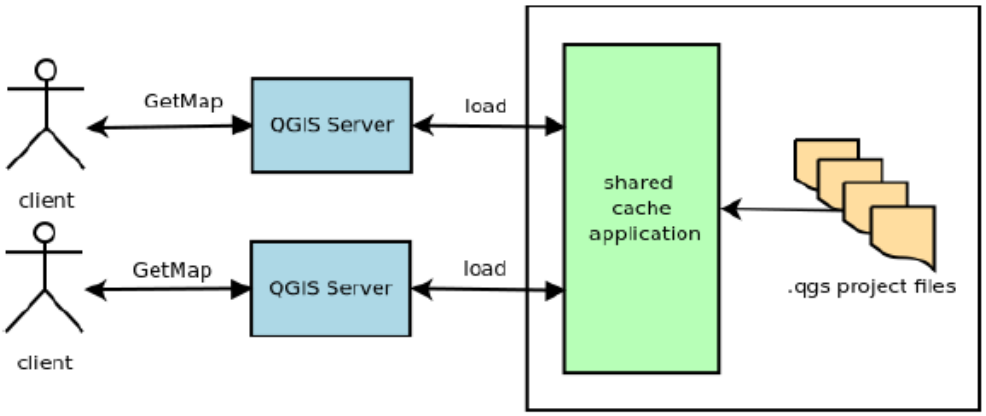
Implementation of a Shared Cache

As outlined above, stages 2 and 3 are mainly responsible for the slow startup of a newly spawned instance of a WMS server. All stages have to be passed through by all instances, because they do not share any resources, regarding the project and layer configuration. A shared cache based on noSQL databases, such as Memcached or Redis would be able to delegate the project loading, parsing and checking to either the first instance in a GDI or a separate application that would then store already digested configuration and GetCapabilities response centrally. A newly spawned instance of a WMS server would then be able to skip stages 2 and 3 and get all the information from the central cache. Once the centrally cached information has been retrieved, it is stored locally in the separate instances, as it is much faster to retrieve information from local RAM rather than through the network from a central cache.

There would be a decentralized and a centralized way to implement the sharing of the project and layer configuration:



Decentralized solution: every *first* time a project is required from any of the WMS Server instances the project is loaded from the disk, the project is then sent to the cache from the WMS Server instance that loaded it, all subsequent project requests from other WMS Server instances will retrieve the project from the shared cache.

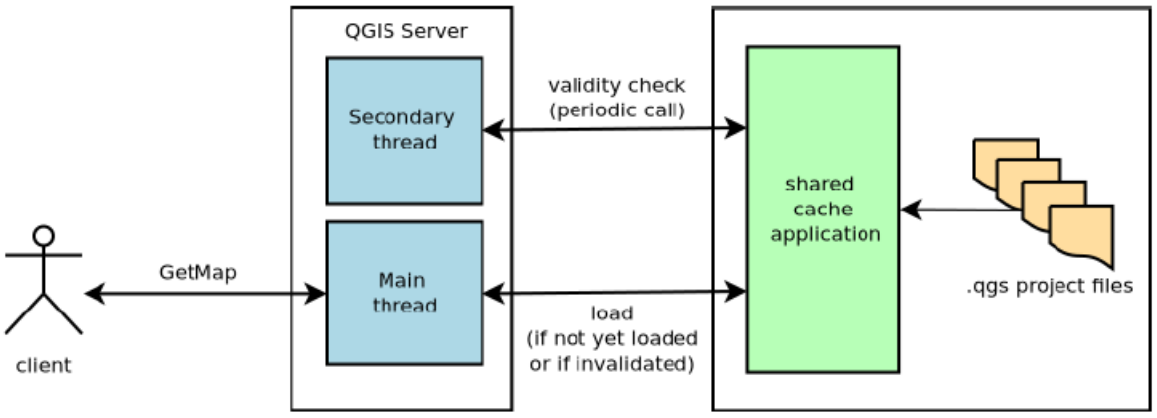


Centralized solution: projects may be stored on a single location/machine. In this case, the shared cache may be preloaded with all projects that will be served by the WMS Server instances.

Shared cache implementation challenges

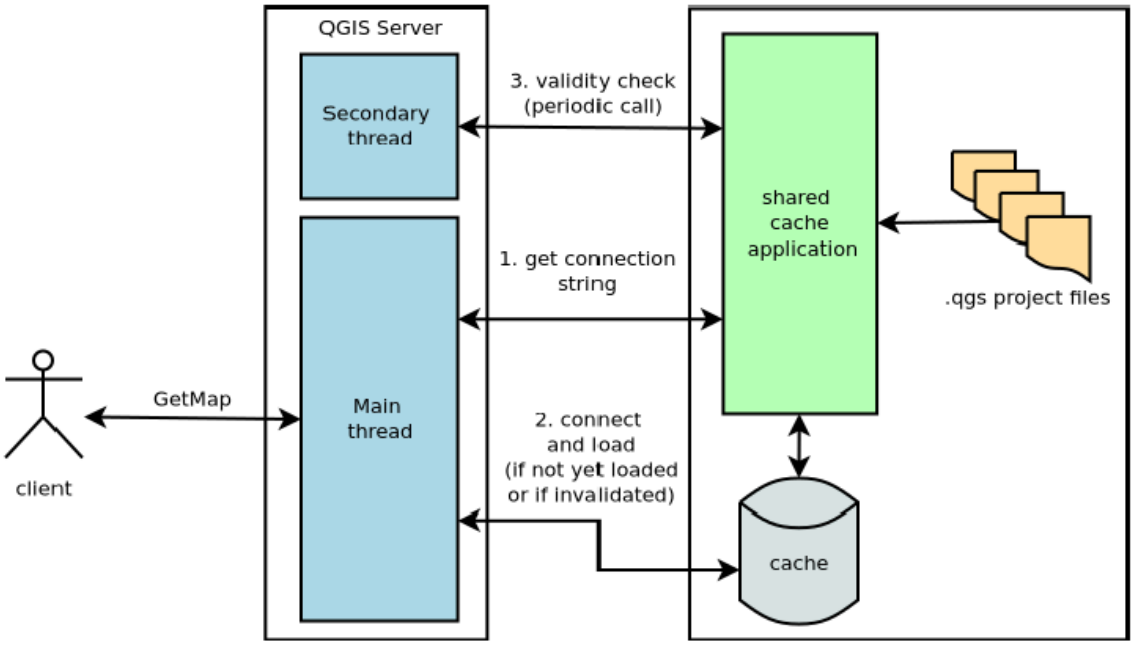
The above mentioned solutions were designed around an unidirectional communication channel. WMS server instances can pull information from the shared cache, but the shared cache cannot notify the instances in cases where all or parts of the cache is replaced.

A bidirectional communication, where the shared cache can inform WMS server instances, can be difficult to implement, especially in cases where the server instances are operated behind reverse proxies and load balancers. To overcome this problem, a polling mechanism in a secondary thread can check in regular intervals if their local cache is still up-to-date or if it needs to be replaced. Using this mechanism, a real bidirectional communication channel can be avoided.



Polling mechanism for cache validity checks

Out of several possible implementations for a shared caching architecture, the use of a Server controller application appears to be the most promising solution. This application could integrate the shared cache or rely on a generic shared cache application (memcached or redis for example). In this case the WMS Server instances would query the WMS Server controller application to know the type of cache they need to use and the connection parameters. An abstract shared cache interface would enable the implementation of different cache connectors (memcached or redis for example). Such a generic cache connector could also be used for other caching scenarios, such as the GetCapabilities response cache and possibly also for caching plugin filters (available in a Python server interface).



Caching architecture with a generic caching solution and abstract interface

2.3 - Load Balancing Problems

One major problem in load balancing is the unpredictable nature of the duration of processing a request. GetMap requests and requests for printouts with higher resolution are good examples, where the time it takes to process a request can easily vary between a few milliseconds and several seconds. The time it takes to process such requests depends on the zoom level, number of features, complexities of geometries, DPI, database queries, number of layers, complexity of symbology and labeling and many more. It is therefore impossible to predict, based on the input parameters, how long such a request will take to process.

Traditional “round robin” load balancing often fails to deliver satisfying queueing where a request that would be processed within a few milliseconds is queued behind a long-running GetMap or print request while in parallel other WMS servers would have spare resources because their processes are shorter than the long running process where the new short

incoming request is queued behind. The following section discusses several better alternatives than just simple random “round robin” load balancing.

Every scalable, stateless pod is part of a kubernetes *ReplicaSet* . Indistinguishable pods, so called replicas, share a common ReplicaSet. Transport of data to pods is handled by kubernetes *Services*, which expose ports in different scopes including external access. By associating a Service with a ReplicaSet, requests are automatically distributed between all replicas. First a node is chosen, by default this happens randomly, but other behaviour can be configured, then on the node a replica is chosen at random.

Every Service has a type that defines if ports will be exposed to the cluster only (type *ClusterIP*), on the physical node (type *NodePort*) or to a provided load balancer of a public cloud deployed in the cluster (type *LoadBalancer*).

In the following methods will be explored how load balancing can be achieved in a kubernetes cluster between the underlying nodes.

Inherent Load Balancing

A kubernetes Service is inherently load balancing between the pods it exposes.

In a kubernetes cluster pods can be created and evicted to adapt to resource requirements currently requested from the application. This process was summarized in the chapter (cross-reference) about autoscaling pods in a kubernetes cluster. Kubernetes abstracts away the nodes in this scenario. Responsible is the kubernetes Service that provides the routing of requests to any pod on any of the nodes. By default it uses iptables rules for this and chooses a target node at random.

Responsible for assigning (“binding”) a newly created pod to a node is the kubernetes scheduler, who filters nodes according to a set of rules and ranks them according to a scoring algorithm that incorporates certain metrics of the node, primarily available CPU & memory resources. In the chapter “Autoscaling” the resource limits of a pod were introduced but there is also a minimum amount. With *resources.request* a minimum of required resources can be set for a pod (defaults: 500 milli-cores CPU and 256 Mebibyte RAM). If resources on a node are not sufficient then it will be ignored by the kube-scheduler, if resources in the cluster are not sufficient then the pod will stay in a pending state until more resources are available. These minimum resources are guaranteed to the pod and only possibly changed by a VPA. This has the important implication that **pods never suffer from high load from other pods on the same node**.

When pods are started within a kubernetes Service the scheduler is incentivized to distribute the pods over all existing nodes to optimize network performance. Requests are then distributed randomly over all nodes per default as mentioned before.

Load Balancing via IPVS

It was mentioned that a kubernetes Service is responsible for choosing a target pod to handle a request and setup the corresponding routing.

This is possible because every kubernetes object that is not of type ExternalName is a REST object and has a DNS resolvable name inside the cluster. Kubernetes employs a central proxy that routes traffic to the appropriate objects (after DNS resolution with CoreDNS). How the kubernetes proxy arranges the routing depends on the mode it is running in.

By default it runs in *iptables-proxy-mode* and the assignment of requests to nodes is random. This is why the load balancing of a kubernetes Service happens at random.

However, when nodes are set up with the correct kernel modules for IPVS, the kube-proxy mode can be changed to IPVS-proxy-mode. This mode allows for limited load-balancing capabilities adhering to one of 6 rules that correspond to the underlying IPVS scheduling rules:

- rr - **round robin** : distributes jobs equally amongst the available real servers
- lc - **least connection** : assigns more jobs to real servers with fewer active jobs
- sh - **source hashing** : assigns jobs to servers through looking up a statically assigned hash table by their source IP addresses
- dh - **destination hashing** : assigns jobs to servers through looking up a statically assigned hash table by their destination IP addresses
- sed - **shortest expected delay** : assigns an incoming job to the server with the shortest expected delay. The expected delay that the job will experience is (Ci + 1) / Ui if sent to the ith server, in which Ci is the number of jobs on the ith server and Ui is the fixed service rate (weight) of the ith server.
  The service rate of the server is a fixed value assigned to each node by an IPVS administrator and defaults to 1.
- nq - **never queue** : assigns an incoming job to an idle server if there is, instead of waiting for a fast one; if all the servers are busy, it adopts the ShortestExpectedDelay policy to assign the job.

This method of proxying, as a side benefit, comes with decreased latency and CPU load, but requests to failed pods will not be retried. To mitigate this risk container readiness probes can be deployed to prevent routing to non-available pods.

This method achieves a little more control over the inherent load balancing mechanism and might mitigate performance inefficiencies in some edge cases random load balancing might produce.

Load Balancing via a provided Load Balancer

On a public cloud usually external network load balancers are provided on demand by setting the type of a service to LoadBalancer.

The available control depends on the configuration interface that such a load balancer exposes.

In theory and on bare metal it is possible to create a custom LoadBalancer implementation (i.e. with MetalLB). The overhead seems hardly justified though.

Load Balancing via Custom Load Balancer

A kubernetes Service of type NodePort provides the option of opening a port on the node it is running on. By default this port will be randomly chosen, but it can be set manually. The Service will automatically assess on which nodes pods exist that it targets and opens the same port on all of them. A fixed NodePort specification allows for load balancing solutions that kubernetes does not support natively yet.

An application could consume the prometheus api, that is described in more detail in the chapter "2.4 - Monitoring, Logging and Administration" and conduct load balancing with this information

Load Balancing - Recommended approach

The kubernetes architecture inherently mitigates problems of load balancing by the methods described in the chapter "Inherent Load Balancing". It is recommended to consider if the number of cases in which intelligent load balancing is needed is high enough to justify the increase in complexity and amount of customization that arises due to it.

Knowing the inherent method of load balancing of a kubernetes Service, an outline shall be drafted on how it would work for a WMS Server application.

WMS server pods should be grouped into the ones handling light requests (GetFeatureInfo and GetLegendGraphics) and the ones responsible for more heavy load requests (GetMap and GetPrint, ..), each group is assigned a HPA (or VPA). It should be assured that lighter requests should be less affected should resources ever become sparse. For this to be realized the HPAs allow for a maximum pod number to be set that limits the number of pods. The pod limits should be adjusted in a way that the heavy load pods never constrain additional light load pods to be created to meet higher demands. If heavy load pods are vertically scaled, resource limits should be assigned correspondingly to guarantee it.

Because services allocate resources across all nodes by default (meaning pod creation will be distributed across available nodes) the load will be balanced accordingly and therefore the default random request-to-node assignment should be sufficient in most cases. Because light

lifting containers have separated resources assigned and heavy lifting containers are constrained by their pod limits, they will never slow down lighter load containers whether they share a node or not.

When this is achieved load balancing becomes less of a problem. This is the recommended approach due to its simplicity and more cloud-like solution.

Load Balancing - Recommendation Limitations and Mitigations

In most situations the approach described above should handle request spikes fairly well, provided the general resources are adequate. To discuss edge cases, it is interesting to look at different ways that load will increase in the cluster. In a situation of resource scarcity it is understood that pods handling heavy requests will max out first, never constraining lightweight pod creation. This is assured by setting the pod number limit in the corresponding HPA (or scaling them via a VPA). Assuming this is done successfully, only the load of the lightweight pods are interesting. Now there are two scenarios, either the load rises symmetrically in all pods or not. In the first case the load is already balanced, so only the second one is of interest.

To create a situation that slows the handling of light requests, one has to conceive an asymmetry in distribution of load. The architecture is similar to the depiction described in chapter 2.2 architecture.

The image below pictures such a situation of an asymmetric distribution of load across 3 nodes, all of which run 1 pod handling heavy load requests at 75% utilization and 2 additional pods for lightweight requests. “Node 1” 2 lightweight pods using 90% of their allocated CPU resource limits, while “Node 2” and “Node 3” only run 2 lightweight pods each with a CPU utilization of 60%, the average CPU utilization of all lightweight pods is therefore 70%.

Such a situation could emerge from *GetMap* requests on complex layered maps that randomly were assigned to the same node “Node 1”. Say the HPA is configured to aim for an average CPU utilization of 65%, it will therefore trigger the creation of a new pod (currently 70%). The kubernetes scheduler will find pod numbers equally distributed over all nodes already and will therefore resort to scoring to find a match. It will see the most resources available at “Node 3” and schedule the new pod on it.

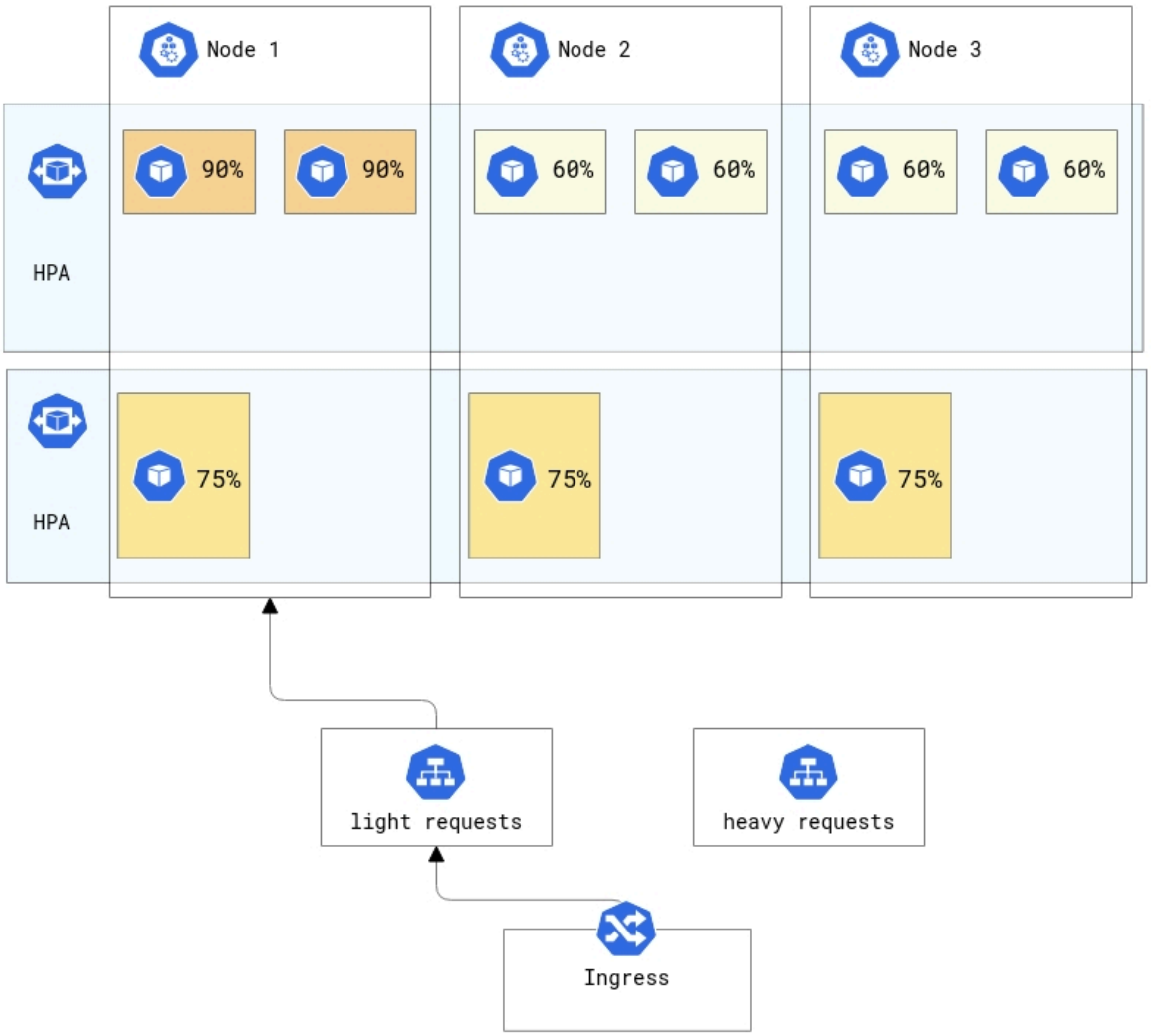
This will however still leave “Node 1” low on resources and, since requests are assigned randomly across nodes, it might get a new request to which it cannot respond as quickly as “Node 3” could.

Possible mitigation strategies are:

1. Have smaller pods:
   A higher number of smaller pods are balanced over all nodes and make it less probable

that all pods with high utilization will be on one node.

2. Move *GetMap and GetPrint* requests to heavy load pods:
   When pods only respond to requests with a guaranteed short response time, pod distribution and node randomization will be enough to prevent asymmetric load accumulation, thus removing the necessity for complex load balancing.
3. Employ proxying via IPVS:
   With a kube proxy running in IPVS mode the load-balancing strategy can be changed to one that considers the number of jobs running on the target node. *Nq*, *sed* or *lc* could prevent “Node 1” in the example from having to process more requests while its pods are running with a utilization of 90%.



2.4 - Monitoring, Logging and Administration

Health checks

“Ready”, “Live” and “Healthy”

A definition of **live** and **ready** can be understood as a concrete implementation and a broader general term. To facilitate the argumentation first the latter will be presented from a kubernetes point of view, then in terms of a WMSServer application and then an example implementation will be provided.

In the following section a pod that first responds appropriately to requests is to be considered **ready**. A pod that is **live** is one that stays ready during its runtime, namely didn’t get caught in a deadlock. The difference is manifested in how kubernetes handles the pod, i.e if it considers it for routing (ready) and if it schedules it for a restart (not live), as explained in the next paragraph. For the purposes of this document the additional differentiation of the status **healthy** is not deemed necessary.

In terms of a WMSserver application for a container to be ready or live the following three functionalities must be checked and ensured:

1. Data can be received from the database
2. A working value retrieval from the cache
3. A responsive WMSserver container

From a kubernetes standpoint these checks can be configured on a per pod (or per ReplicaSet) basis, the next paragraph will introduce the tooling that implements this natively.

Init Containers

The Kubernetes configuration of pods allows us to define a container which will be run before a specific pod is started. This concept in Kubernetes is called “Init Containers”.

By specifying a small sized container which is allowed to establish a connection to the database, a check can determine if the database is accessible before we start the WMSServer instance. Only when the init container returns an exit code 0 which means “success”, the WMSserver pod will be started.

Health checks - Kubernetes’ Liveness and Readiness Probes

Kubernetes Live- and Readiness probes integrate health checks into routing and pod scheduling. To visualize these solutions, first the problem should be described. Between pod creation and request handling there must be an event signaling a new receiver for requests.

Since kubernetes is agnostic as to which program a container executes, it considers a pod up and running (ready) with the successful start of its containers init processes with PID 1. Similarly a pod is not stuck (not live) before those processes are not defunct. This does not reflect the requirements of the application nor the definitions above very well. A container might not be able to handle requests while PID is 1 still intact.

With kube-proxy running in iptables mode (the default) the existence of these problems is not obvious because unhandled requests are retried with different pods. It still imposes unnecessary latency. With a kube proxy in IPVS mode, requests sent to unresponsive containers are even lost.

In short there are two problems:

1. When is a pod responsive first (ready)?
2. When is a pod stuck and should be restarted (not live)?

Those two problems are solved by so-called Readiness and Liveness Probes.
Both probes are deployed with their target pods and return the value of a custom command that is executed periodically, it can be a simple bash command or a HTTP request.

Unless the **Readiness Probe** returns a certain value to indicate success (0 for bash commands, 200-399 status codes for requests), the pod will not be considered ready to receive requests from its service.

Whenever the return value of the **Liveness Probe** indicates failure, its pod is considered stuck and will be restarted. It can be given an initial delay to prevent fresh containers to be restarted in case of a longer startup time.

Health checks - Recommendation for OGC WMS Applications

A solution is needed that checks a pod for the three criteria introduced at the beginning of the chapter while simultaneously not creating too much overhead.

It is advisable to deploy readiness and liveness probes with every Replica set and configure it to send a *GetCapabilities* request through its httpGet spec.

With knowledge of QGIS server specifics, there might be considerations if the request:

- Introduces more overhead than necessary (if only one big “main project” is used (such as in Kt. Solothurn), it might be advisable to also a include a small project only for the purpose of “health checks”)
- Doesn’t fully cover every functional aspect of the pod (db-connection, cache, …)
- The response status of *GetCapabilities* is not fully compliant with the definition:
    - Success: 200-399

- Failure: everything else

If those concerns are valid, a new route (i.e. “/healthy”) could be created to adhere fully to the specifications or optimize for performance.

Alternatively a custom executable can be developed that needs to be accessible by the pod, performs all the necessary checks and returns 0 on success, however there are few benefits with this approach while simultaneously deviating from production functionality more.

Monitoring

Monitoring - Prometheus

Prometheus is the second open source project to have joined the *Cloud Native Computing Foundation* after Kubernetes, it is therefore highly mature and well maintained. The functionality it provides is a monitoring approach that integrates exceptionally well with container environments like kubernetes.

Opposed to a more traditional solution of continuously pushing metric data to a central endpoint, Prometheus follows a pull approach. This is to reduce network overhead when a great number of microservices are monitored and would traditionally possibly overload a central monitoring system with network traffic. Additionally it reveals a failed service faster, since an unresponsive endpoint is a robust indication, while just missing log data (in case of a push approach) is not.

This pulling approach is realized by each service implementing a “/metrics” endpoint that exposes certain metric data about itself which are scraped by the Prometheus server periodically. The service has to be registered with Prometheus as a **target** and the metrics have to adhere to a certain plaintext format understood by Prometheus. Prometheus then accumulates it and stores it in a centralized, time series database.

Prometheus’ alertmanager can be configured to push notifications in certain situations which can speed up incident response or prevent incidents in the first place. Clients that are able to receive alerts are manifold and range from email over dedicated incident response tools to the slack-messenger. A typical application for an alert would be resource scarcity.

A **prometheus exporter** is responsible for exposing the data to the prometheus server. Some kubernetes components integrate exporters out of the box or after activation via a flag. Prometheus will even pick them up automatically by polling the KubernetesAPI for target auto-discovery. Some useful metrics are:

The **control plane metrics**. These contain a lot of metrics, including those of the kube-proxy, or the kubelet. The job of the latter is to be something like a kubernetes node operator. Useful metrics it provides are i.e. resource utilizations of the node in terms of CPU, memory, network,

etc.

The **kube-state-metrics** contain information about deployments, pod metrics, resource reservations, etc. They provide “per pod” resources that can be analysed, which is beneficial for monitoring the state of the utilization of the applications as well as for setting triggers for alert notifications.

A common approach to metrics collection with Prometheus is to deploy a sidecar container alongside (namely the same pod) the container running the service providing the prometheus exporter. The reason for this is, that many services don’t produce metrics in a prometheus compatible format nor should that be their concern. The exporter either receives metrics emitted by the service or polls them for them, it then exposes them in the correct format for Prometheus server to pick them up. Many exporters are available to be integrated with a lot of commonly used containers, like nginx or PostgreSQL

Custom sidecar containers can be created with one of multiple prometheus client language bindings.

Monitoring - Grafana

Grafana is a visualization tool that was optimized to work on top of time series databases. It is commonly used for the visualization of data accumulated by Prometheus. It features a well-arranged dashboard that can display information gathered from multiple sources, this allows for the centralization of monitoring and logging information into one interface.

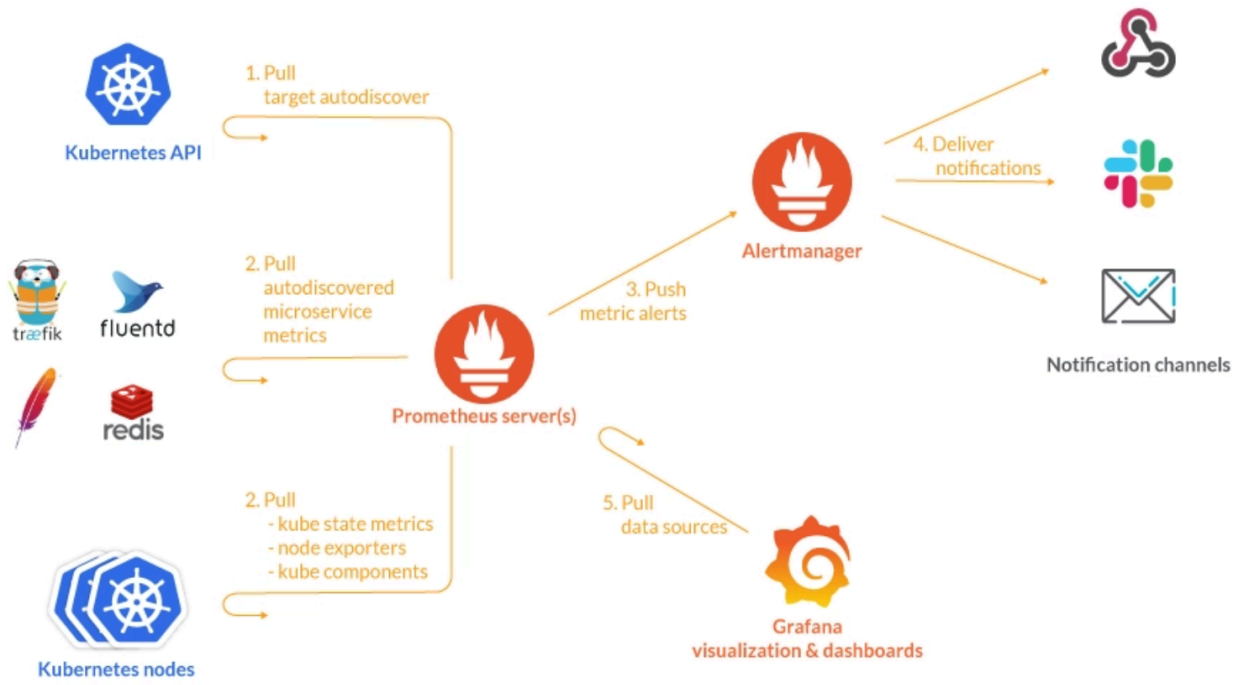
Grafana uses Prometheus’ query language promQL to pull data from any number of prometheus instances periodically.

Monitoring - Recommendations

It is advisable to deploy a monitoring of all the /metrics endpoints kubernetes exposes by default to be able to monitor resource allocation and visualize them in a grafana dashboard.

There might be metrics of the **application state** that are interesting, i.e. monitoring of the response time of different requests (i.e. *GetMap*), for instance to be able to implement a smarter load balancing solution from empirical data.

The most cloud native approach to implement this, if feasible, would be to build a server plugin that can pick up metrics from the running WMS server application and deploy it as a sidecar container alongside the main server container.
It would leverage the official prometheus client library for python to expose metrics to the Prometheus server.



Logging

Containerized applications create logs on the host they are running on, e.g. in the directory */var/log/docker*. Therefore log collection requires an aggregator to run on each node. This is a behaviour that can not be accomplished through the use of ReplicaSets that are used for the deployment of most services.

Assuring that an instance is running on each node is what a *DaemonSet* is for. A logging stack in kubernetes comprises of a log collector that is deployed to each node as a DaemonSet, an aggregator that centralizes the log data from the nodes and a tool for visualization.

Since Grafana is recommended for visualizing Prometheus' data anyways, a straightforward option would be to integrate logging into this dashboard. In the next paragraph such a setup will be described followed by an alternative which provides a more fine grained approach, albeit coupled to more overhead.

Logging - Grafana Loki

Loki is a log aggregation system that was built to work alongside Prometheus in a scalable cloud environment. It deploys a log collection agent (promtail) to every node (via a DaemonSet) that embeds prometheus service discovery libraries enabling it to work on the same configuration to associate the captured log streams with the same object prometheus monitors (i.e. pods with the same labels).

Loki runs in one of two modes to address a tradeoff of configuration complexity vs scalability. In the default mode (*single binary mode*) Loki operates in a monolithic fashion without

dependencies, collects logs via promtail on all nodes and streams it to grafana. Without the need for more functionality this approach should be scalable up to some 100 nodes. At any point the mode can be changed in a “scale-out” approach to include an object store like S3, which makes it suitable for large scale production environments.

Since Grafana is the common choice to visualize prometheus’ monitoring data, Loki allows to centralize logging data in conjunction with monitoring data in one dashboard with consistent labeling across the monitored components.

Loki’s approach is distinct from others in that it foremost strives for simplicity and a lean backend. This is expressed by their initial motivation to only provide “*log files plus grep*”, it only indexes kubernetes labels, not log files and relies on filters to provide functionality. These filters come in the form of a query language LogQL. Expressions that constitute a LogQL query are chained to a pipeline and comprise filters, parsers or formatters. However compared to Lucene on top of indexed log files the functionality is somewhat limited. The latter approach can be implemented with the elastic stack, which is introduced in the next chapter.

Logging - Elastic Stack (EFK)

An ELK (*Elasticsearch*, *Logstash*, *Kibana*) stack has been the most popular approach to provide data collection with enhanced analysis options before the time of cloud environments. Therefore it relishes a great deal of familiarity with developers and provides a wide range of mature options for every way of data processing including machine learning.

A traditional ELK stack lacks the capabilities to collect logs from every kubernetes node, however fluentd can be used to forward the data and fuel the stack. If needed, fluentd emits metadata via a /metrics endpoint for prometheus to connect.

In conjunction with Fluentd rather than Logstash the Elastic stack is abbreviated as **EFK**, meaning Elasticsearch, Fluentd and Kibana.

Fluentd

As another tool from the Cloud Native Computing Foundation, fluentd proves to deserve the attribute cloud native. It forwards contents of different inputs to a multitude of possible backends and can filter and route depending on its configuration. Together with elasticsearch it would take the job of a log collector.

Elasticsearch

Elasticsearch aggregates logs, tokenizes the lines and indexes them. This allows for fine grained manipulation of the data.

Logging - Recommendations

Which logging solution can be chosen depends on the need. Because EFK stack has a very

great search engine underneath, it’s possible to make very fine grained queries. Grafana Loki is a slim solution with the advantage of using the same dashboard like for monitoring, but it has fewer features and is more lightweight.

Conclusion

Recommendations

- Use Kubernetes or OpenShift for container orchestration
- Allow multiple parallel requests per container to fully utilize resources
- Use centralized shared cache instead of cache per container
- Warm up the cache every time new data is coming in or periodically
- Split the requested endpoints to two or three different groups in order to allow kubernetes make smart decision for horizontal pod auto scaling (e.g. light requests vs heavy requests)
- Start the containers only when database is accessible
- Don’t allow incoming requests to containers which don’t pass liveness and readiness checks
- Make use of prometheus and grafana for monitoring of cluster and application
- Make use of Elasticsearch, Logstash, Kibana stack for collecting logs and getting insights

Case study QGIS server

The document above highlights the general requirements and best practices for optimally deploying a cloud-native WMS server. In this chapter, we'll show what technical improvements are needed to allow implementing such a solution using QGIS as an example.

QGIS is a user-friendly Open Source Geographic Information System licensed under the GNU General Public License. QGIS is an official project of the Open Source Geospatial Foundation (OSGeo). It runs on Linux, Unix, Mac OSX, Windows and Android and supports numerous vector, raster, and database formats and functionalities.

QGIS not only provides a desktop GIS but also a server application, multiple web clients and mobile applications.

QGIS Server is an open source WMS, WFS, OGC API for Features 1.0 (WFS3) and WCS implementation that, in addition, implements advanced cartographic features for thematic mapping. QGIS Server is a FastCGI/CGI (Common Gateway Interface) application written in C++ that works together with a web server (e.g., Apache, Nginx). It has Python plugin support allowing for fast and efficient development and deployment of new features.

QGIS Server uses QGIS as a back-end for the GIS logic and for map rendering. QGIS Server uses the QGIS project files generated by QGIS Desktop as a configuration language, the same files for Desktop and Server.

As QGIS desktop and QGIS Server use the same visualization libraries, the maps that are published as a service look identical to those created by QGIS Desktop. Recently, efforts have been made to analyze the rendering process more in depth and optimize the rendering code where it was possible with reasonable efforts and with the available resources at hand (Reference 2). The QGIS PerfSuite helps to monitor rendering speed over various QGIS Server versions and compare them, in order to avoid rendering speed regressions (Reference 4).

Missing elements

QGIS server already implements a large number of features that make it one of the leading WMS servers, broadly deployed across the spectrum. Furthermore, it is worth noting that QGIS server is certified and one of the OGC reference implementations for WMS 1.3.[1]

Nevertheless, we saw in the previous chapters that a very specific set of requirements is needed to allow a truly cloud-native deployment.

In QGIS' specific case the following functionalities and analysis have been identified as key points for future developments to enable cloud optimized deployments and thus an even broader acceptance.

[1] https://blog.qgis.org/2018/06/27/qgis-server-certified-as-official-ogc-reference-implementation/

- Implement a centralised cache as thoroughly described in chapter 2.2 and in the complete cache management study in annex 1
- Implement an automated Cache invalidation mechanism that works across networked filesystems to aloof QGIS server to update the project cache whenever the project file is updated
- An in-depth analysis or possible memory leaks and implementation of the identified fixes
- Fcgi zombies in-depth analysis or replacement of the fcgi progress by a HTTP native Internal server. QGIS already has a POC that needs further development to remove the fcgi requirement.
- Additional health checks so that orchestrators can be aware of the current status of a worker
- Addition of a /metrics route for centralised monitoring and implement QEP #193 “QGIS Server and monitoring” (Reference 3).
- More xml parsing optimisations. This point loses a lot of importance if the deployment has a shared cache and a dedicated “cache warming” container that is not exposed via ingress.

Future Technologies

We already observe a rapid rise in the growth of machine learning technologies applied to everyday tasks, these technologies offer new ways of engaging with existing challenges and could dramatically enhance performance if applied in specific areas of QGIS codebase.

One such example could be the usage of machine learning for estimating request processing times and thus being able to optimize load balancing of incoming requests to the appropriated pod types as described in chapter 2.3.

Situation with Other OpenSource OGC WMS Servers

UMN MapServer

UMN MapServer seems to be better suited for cloud deployment compared to QGIS Server in its current state (without improvements of startup time and shared cache as outlined before), because the startup time and the time necessary to load the configuration file seems to be substantially faster than in the case of QGIS Server.

When it comes to load balancing, monitoring and logging, the problems, challenges and potential solutions are pretty much the same as for QGIS Server.

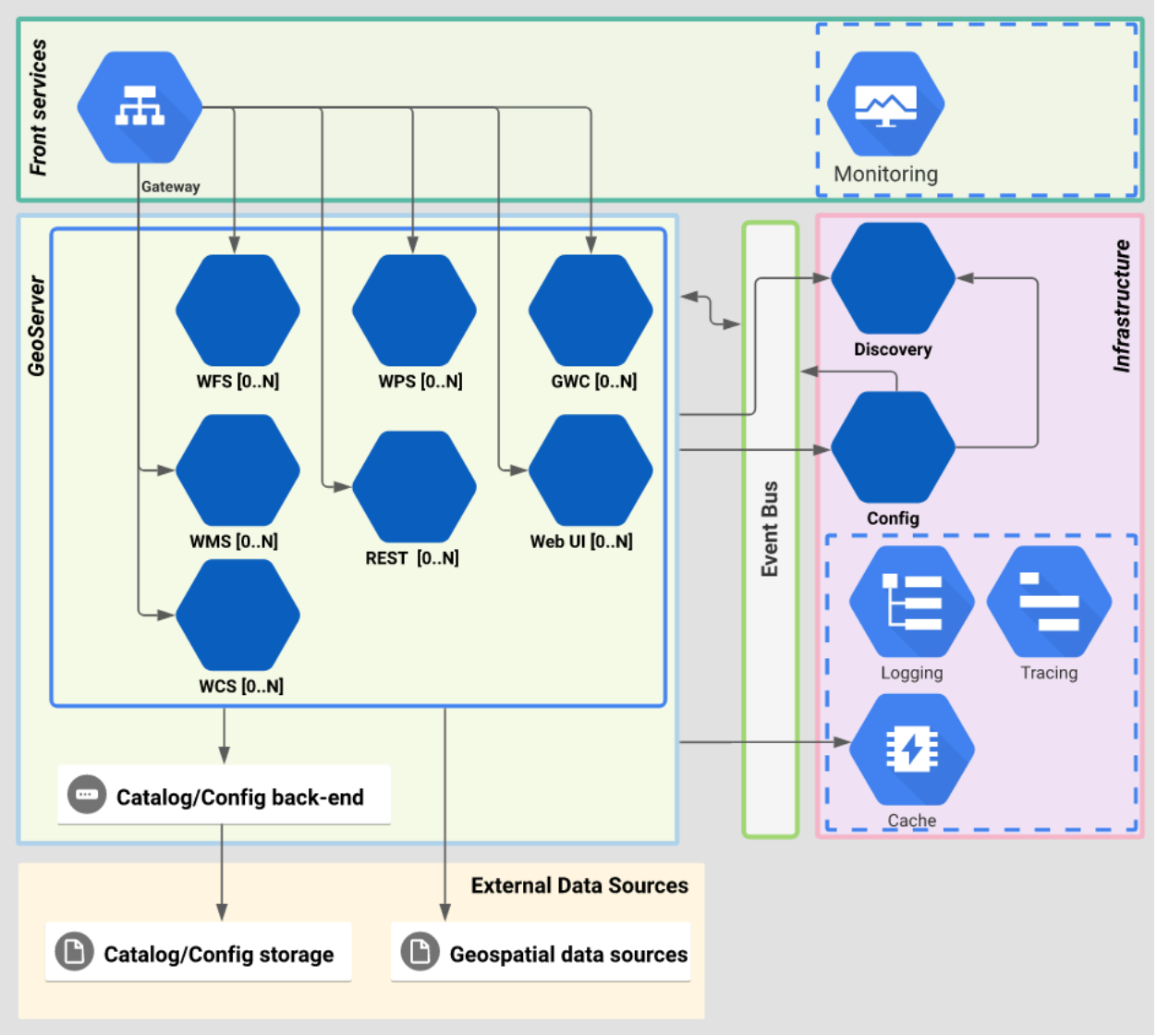
UMN MapServer has been successfully deployed as part of the Dutch NGDI (PDOK) in the public cloud with a kubernetes setup at Azure. (Reference 7)

UMN MapServer has also been successfully deployed as AWS Lambda function, a serverless application that works as “function as a service” (FaaS), suitable for microservice infrastructures. (Reference 8).

Geoserver

Geoserver has a separate project called “Cloud Native GeoServer” that aims to provide a “ready to use” Geoserver in the cloud through dockerized microservices (Reference 9). It allows provision of each business capability to be enabled, configured and deployed independently.

The following illustrations shows the general system architecture of this project (Source: http://geoserver.org/geoserver-cloud/#system-architecture-overview):



This seems to be a work in progress and not yet fully production ready (see comment at http://geoserver.org/geoserver-cloud/#project-status for open issues). This project can serve as a model for other OGC WMS Server deployments and their architecture. The catalog/config back-end seems to be a service that QGIS Server is still missing and should be implemented in order to allow faster startup of pods.

Annex

Annex 1 - QGIS Server and startup time in a cloud environment

https://drive.google.com/file/d/1K7wdH2Psrw6h-eGMHL6Z_68LNBtpOXrb/view?usp=sharing

References

Reference 1 - QGIS server Benchmarking report

https://docs.google.com/document/d/1zCUkcaHHGhxoq6rgR73hu_g4_JVVAplGbbGqBNvQEVI/edit?usp=sharing

Reference 2 - QGIS Server Rendering Speed

https://docs.google.com/document/d/1M-Z4mR3n3kp9mwSQ-H-4gkLxP6OBZldtpSlOcL8CCC8/edit?usp=sharing

Reference 3 - Monitoring QEP (QGIS Server)

https://github.com/qgis/QGIS-Enhancement-Proposals/issues/193

Reference 4 - QGIS PerfSuite

https://github.com/qgis/QGIS-Server-PerfSuite (Code) and http://test.qgis.org/perf_test/graffiti/ (results)

Reference 5 - Design proposal vertical pod autoscaler

https://github.com/kubernetes/community/blob/master/contributors/design-proposals/autoscaling/vertical-pod-autoscaler.md

Reference 6 - Cloud Optimized Geotiff in QGIS

https://www.cogeo.org/qgis-tutorial.html

Reference 7 - PDOK Github Repository

https://github.com/PDOK/

Reference 8 - MapServerless AWS Lambda Layer (FaaS)

https://github.com/bitner/mapserverless

Reference 9 - Cloud Native GeoServer

http://geoserver.org/geoserver-cloud/