www.0x65.dev /blog/2019-12-14/the-architecture-of-a-large-scale-web-search-engine-circa-2019.html

# The Architecture of a Large-Scale Web Search Engine, circa 2019

42-53 minutes

---

real-time-search

kubernetes

ml-systems

cloud-native

oss

The Architecture of a Large-Scale Web Search Engine, circa 2019

Our Journey to Microservices, Kubernetes and beyond.

December 14th, 2019

In previous posts of this advent series, we have described some of the technologies that power our private search products. It is about time that we introduce the systems that bring everything together. It is important to understand that a web scale search engine is highly complex. It is a distributed system with strong constraints on performance and latency. On top of that it can easily become extremely costly to operate; both in human resource and, of course, in money.

This article explores the technology stack we employ today and some of our choices and decisions, which have been taken and iterated upon over the years, to cater both external and internal users.

The topic at hand is very broad and cannot be covered in a single sitting, but we hope to give you the gist of it.

We use a combination of prominent open source and cloud-native technologies wrapped with home grown tooling, which have been battle tested. Places where we haven't found a solution in the open source world or commercial efforts, we have been prepared to dive deep and write some core systems from scratch, which has worked well for us at our scale.

**Disclaimer:** We describe how our system is, as of today. Of course we did not start like this. We had multiple architectural overhauls throughout the years, always considering constraints like costs, traffic and data size. By no means, we would suggest that this is a recipe to build a search engine; it is what is working today, as wiser people said:

> "Premature optimization is the root of all evil" ~ Donald Knuth

And we agree wholeheartedly. As matter of fact, we really advise anyone, to never try to throw all the ingredients to the pot at once. But instead to add them one by one; slowly and incrementally adding complexity one step at a time.

Given the nature of this post, we want to provide an ordered outline of all topics covered:

- Cliqz search as a product and its system requirements.
- Web Search Systems: A near real-time and truly automated search system.
- Data Processing Platform: Facilitating near Real-time and Batch Indexing.
- How deployments were done in the past? The Pros and Cons of various approaches.
- Microservices Architecture: Orchestrating services involved to deliver content for a search engine result page.
- Our need for using containers and a container orchestration system (Kubernetes).

- Introduce our Kubernetes stack - How we deploy, run and manage Kubernetes and various add-ons and the problems they solve for us.
- Local Development on Kubernetes - An end to end use case.
- Optimizing on Costs.
- Machine Learning Systems.

## Our Search Experience—Dropdown & SERP

The search engine at Cliqz has two consumers with different requirements.

### Search-as-you-type

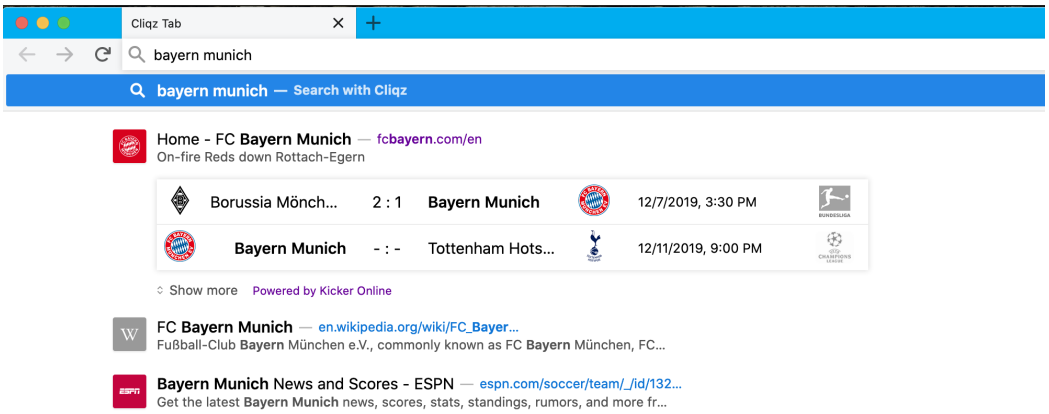


Figure 1: Cliqz Dropdown in the Browser

The search in the browser address bar[1], with results available on the dropdown. This type of search requires fewer results (typically 3) but is extremely latency sensitive (less than 150 ms); otherwise the user experience suffers.

### Search in SERP


Figure 2: Cliqz Search Engine Result Page beta.cliqz.com

Search on a web page, the typical search engine results page everybody knows. In here, the depth of the search is unbounded but it is less demanding on latency (less than 1000 ms) as compared to the dropdown version.

## Fully Automated and Near Real-time Search

Consider a query like *"bayern munich"*. Now, this may seem a very generic query, but when issued, it touches several services within our system. If we try to interpret the intent from the query, we will figure out that the user may be:

- Researching about the club (in which case a Wikipedia snippet would be relevant)
- Interested in booking tickets, buy merchandise, register as an official fan (Official Website)
- Interested in current news about the club:
    - Pre-Match news about the game
    - In-game information like: Live Scores, Live Updates or Commentary.
    - Post-match analysis about the game
    - Off-season Information about the inner workings of the club and activity during the transfer window, hiring new coaches etc.
- Searching for old web-pages and content, club history, record of past games, etc.

As one might observe, this is much more than finding relevant pages. Not only should the information requested be semantically relevant, but it should be relevant w.r.t time. Recency or Temporal Sensitivity in search is a very important factor for the user experience.

To make this a coherent experience, the information must be made available from different sources and transformed into a search-able index in near real-time. We need to ensure that all models, indexes and assets are up-to-date (e.g. Loading of images must reflect the current events and keep the title and content up-to-date with respect to a developing story). **As hard as it may seem to execute successfully at scale, we strongly believe that our users should always be presented with up-to-date information, this intuition forms the basis of our overall system architecture.**

The data processing and serving platform at Cliqz follows a multi-tiered Lambda Architecture. It is composed of three tiers based on the recency of the content being indexed. They are:

1. **Near Real-time Indexing**

    - Fully automated and powered using Kafka (Producer, Consumers and Stream Processors), Cassandra, Granne[2] and RocksDB.
    - Cassandra stores the Index data in different tables. Records in different table have varying Time to Live (TTL), so we can free up the storage as the data is re-indexed in later stages.
    - This component is also responsible for our trending or popularity driven ranking features which help identify trends over a moving window of varying time sizes. We accomplish this via stream processing using KafkaStreams.
    - All this translates into product features including recent content within search results and top news.

2. **Weekly or Sliding-Window-Based Batch Indexing**

    - Based on content of past 60 days.
    - Re-indexed weekly (End to End automated pipelines of batch jobs on Jenkins).
    - Machine Learning and Data Pipelines are executed upon recent data resulting in higher quality of our search results.
    - A good framework to test and prototype new ML models and algorithmic changes using a subset of data thereby reducing costs of end to end experiments on entire data.
    - Map-Reduce and Spark based batch workflows managed through Luigi and retrospectively managed using Jenkins Pipelines.
    - Keyvi, Cassandra, qpick and Granne for serving.

3. **Full Batch Index**

    - Based on all available data
    - Re-Indexing: once every 2 months
    - MapReduce and Spark based batch workflows managed through Luigi
    - Used to train large scale Machine Learning Models over a large data-set. e.g.: Query and Word Embeddings, Approximate Nearest Neighbour Models, Language Models etc.
    - Keyvi, Cassandra, qpick and Granne for serving

What is important to note here is that, Near Real-time and Weekly Index is responsible for a large portion of search related content served on SERP. This is a similar behavior to other search engines which promote recent content over historical content about the topic. The batch index is handling time independent queries, long tail of queries and content which is rare, historical or tricky in the context of understanding a search query. The combination of the three gives us the necessary ammunition to build Cliqz search in its current form. All systems are capable of answering all the queries, the final results, however, is a mixture of the results of all indexes.

## Deployments—A Historical Context

> You haven't mastered a tool until you understand when it should not be used. ~ Kelsey Hightower

From the start, we have been focused on delivering our search services using a public cloud provider rather than managing infrastructure on-premises. In the last decade, that has become the norm across the industry, given the complexity and resources required to operate one's own data center(s) compared with the relative ease of hosted services and the ease to digest pay as you go model for startups. Amazon Web Services (AWS)[3] has been convenient for us as it allowed to abstract ourselves from managing our own machines and infrastructure. If not for AWS, it would have taken us a lot more effort to reach this stage. (But, they are as convenient as they are expensive. You will see in this article some tricks we came up with to reduce costs, but we advise you to be extremely careful using cloud services at scale.)

We typically strive to avoid a managed offering of a service that might be useful to us as costs can be unbearably high at the scale we typically operate at. To bring in some context let us start from the year 2014. A growing concern that we met early on was reliably provisioning resources and deploying applications on AWS.

We started with putting some effort into building our own infrastructure and configuration management systems on top of AWS. We focused on shipping a solution which was native to python to ease developer on-boarding. We wrapped the Fabric project and coupled it with Boto to provide nice interfaces to launch machines and configure one's application with few lines of code driven through a *deploy.py* file co-located to the service source. This was then wrapped into project template generators for easy onboarding of new projects. Back then, it was early days of docker and traditionally we shipped as python packages or plain python code which was challenging because of dependency management. Even though the project gained a lot of traction and was used by many services driving many products at Cliqz, there are certainly things a library driven approach to infrastructure and configuration management lacks. Global state management, central locking for infra changes, no central view of cloud resources utilized by a project or developer, reliance on external tools to cleanup orphaned resources, limited configuration management, limited observability into developer usage, context seep in from regular users of the tool are some of the things that created friction and which in turn led to an increased operational complexity.

This led us to explore alternate external solutions as homegrown efforts had to be stopped due to limited resources. The alternative we eventually landed on is a combination of solutions from Hashicorp including Consul, Terraform and Packer and eventually configuration management tooling like Ansible and Salt.

**Terraform** presented an excellent declarative approach to infrastructure management which many of the current technologies in the cloud native space have leveraged. So we decided, after careful evaluation, to retire our fab-based deploy library for Terraform. Besides technical pros and cons, one always have to consider human factors. Some teams are slower to adopt changes than other, be it because of lack of resources or because the cost of transitions are not uniform. For us, it took a long time, about one year, to migrate.

Terraform certainly brought us some out of the box features which we were missing from our old deploy project, including:

- Central state management of infrastructure.
- Verbose plan, patch and apply support.
- Easy teardown of resources with minimal orphaned resources.
- Support for multiple clouds.

Meanwhile, we also faced some **challenges** in our journey with Terraform:

- Complex cloud specific DSL which is typically not DRY.
- Difficult to wrap it in other tools.
- Limited and sometimes complex templating support.
- No feedback on the health of services.
- No easy rollbacks.
- Missing crucial features implemented by third parties like terragrunt.

Terraform certainly has its place at Cliqz and these days we still use it to deploy most of our Kubernetes infrastructure.

## Intricacies of a Search System

Figure 3: Search Overview

Over the years, we have moved from a distributed architecture, with dozens of servers, to a monolithic architecture, to finally end up on a microservices architecture.

Each solution we believed in was the most convenient at that time, given the resources available; for instance, the monolith version was due to the fact that most of our latency came out of network IO among the machines of the cluster. At that time, AWS launched the X1 Instance, with a whopping 2 TB of RAM. A quick change of architecture allowed us to reduce latency quickly, but of course, costs went up. The next iteration on architecture we focused ourselves on cost. Step by step we tried to fix one variable without worsening the others. It might not look like a very fancy approach, but it worked well for us.

> The microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. ~Martin Fowler

The microservice definition by Martin Fowler is technically correct, but is somewhat abstract. To us it generally does not give enough of a description on how to exactly build and proportion your microservices, which are important concerns. The move to microservices brought us:

- Better modularity and autonomy of teams and separation of concerns.
- Horizontal scalability and workload partitioning.
- Fault isolation and better support for multiple languages.
- Multi-tenancy and generally a better security footprint.
- Better automation in operations.

Looking at a broad picture of the architecture and how we structure our microservices when a search query is issued to the backend, a number of services are triggered in the request path. Each service is considered a microservice in the sense it has separation of concern, is driven by a lightweight protocol (REST/ GRPC) and is horizontally scalable. Each service can be constituted of individual microservices and can have a persistence layer. The request path typically involves:

- **Web application firewall (WAF)** - Application firewall against common web exploits.
- **Load Balancers** - Request ingestion and load balancing.
- **Ingress proxies** - Routing, edge observability, discovery, policy enforcement.
- **Eagle** - Server side rendering for SERP.
- **Fuse** - API Gateway, results mixer, edge caching, authentication / authorization.
- **Suggest** - Query Suggestions[4].
- **Ranking**[5] - Serves search results using a near real-time index and pre-compiled batch index (Lambda Architecture).
- **Rich Results**[6] - Adds rich information like snippets for weather, live scores, other third party sources.
- **Knowledge Graph and Instant Answers** - To the point information related to a query.
- **Places** - Geo-Local based content recommendation.
- **News** - Real-time Content from reputable news sources.
- **Tracker** - Domain specific tracker Information via WhoTracks.me.
- **Images** - Image results relevant to user query.

All these services are orchestrated through a common API gateway responsible for handling search volume as well equipped with features like providing protection around traffic surges, Auto-scaling based on requests / cpu / memory / custom metrics usage, edge caching, traffic shadowing and splitting, A/B Testing[7], Blue-Green[8] deployments, Canary release[9], etc. The platform is also responsible for providing many of these functionalities to services in play.

## Docker Containers and Container Orchestration System

So far, we have described some of the requirements and specific details about our product offering. We described how we did deployments and what were the shortcomings with the solutions which we tried. Given these learnings, we chose Docker, as the fundamental building block for all of our services. We started delivering our code using docker containers, instead of using plain VMs with code and dependencies. With docker, code and its dependencies are shipped as docker images to a container registry (ECR).

But once the services started growing again, there was a need to manage them, specially when we wanted to scale them in production. It became apparent over the years that we had to introduce a container orchestration system. The pain points that led to the introduction were typically **wasted compute resources** and **complexities in infrastructure** and **configuration management.**

We are always short on people and on computing power, this is a situation shared by most start-ups with limited resources. Of course, to be effective, we must focus on solving the problems that we have, that cannot be solved by tools that already exists. But, we do not believe in reinventing the wheel (until it radically changes the existing landscape). We are avid users of open source software, where we found solutions to our critical business problems.

We started evaluating Kubernetes[10] (k8s) as soon as the 1.0 version was released and had production level workloads running by 1.4—once the project showed stability and maturity in tooling. During the same time, we evaluated other orchestration systems including Apache Mesos and Docker Swarm for our large scale projects like fetcher (web scale crawler). We eventually landed on running everything with Kubernetes, as it was quite evident that the project showed promise in having a cohesive approach to tackling orchestration and configuration management while others didn't, and also included a strong community support.

## Kubernetes - The Cliqz Stack

Figure 4: OSS at Cliqz

Open Source has won!

Cliqz relies heavily on a lot of Open Source Software (OSS) projects typically under the umbrella of Cloud Native Computing Foundation[11] to provide a cohesive cloud native experience. We try our best to contribute back to the community both in terms of code, blog posts and on other channels including slack. We will share details about the usage of some critical OSS projects which form the core of our stack:

**KOPS—Kubernetes Orchestration**

For container orchestration, we self manage multi-region Kubernetes clusters using KOPS and some homegrown tooling on top to manage cluster lifecycle and addon management. Shoutout to Justin Santa Barbara and kops maintainers for their awesome work, providing a well integrated experience on bringing up the k8s control plane and worker nodes in a conformant way. Currently we don't rely on a managed offering just because of the flexibility of KOPS and lack of maturity in EKS, an AWS managed k8s control plane.

Using KOPS and self managing a cluster meant that we can really set at our own pace, dive deeper during troubleshooting, activate features which were application requirements but available only in specific Kubernetes releases. If we had waited for a cloud offering, we would had to wait a considerably longer time to reach at the stage that we have now.

**Weave Net—Network Overlay**

Its important to note that Kubernetes helps to abstract all parts of the system. This not only includes compute and storage, but also networking. For our clusters which can grow to hundreds of nodes, we employ an overlay network which forms the backbone for providing flat networking and network policy enforcement capabilities for pods spawning over multiple nodes, availability zones and regions. Weave Net is the overlay of choice we landed on early because of easy manageability and route management through gossip. As we grow bigger we might shift to AWS VPC CNI and Calico as they mature to provide less network hops and more consistency in routing and traffic. Till now weave net has performed exemplary in our latency and throughput targets and there hasn't been a reason to switch.

**Helm / Helmfile—Package management and delivery**

We have relied on helm (v2) from the start for package and release management of Kubernetes manifests. Even though it has its pain points, we found it to be an excellent resource for release management and templating. We follow a single repository structure for helm charts of all of our services which are packaged and served using the chartmuseum project. Environment specific values are then kept in a separate repository to separate concerns. These are driven using the gitOps pattern through Helmfile, which provides a declarative approach to multiple helm charts release management and associated essential plugins like diff, tillerless and secret management at rest with SOPS. Changes to this repository are validated and deployed using CI / CD pipelines driven through Jenkins.

**Tilt / K9s—No stress local Kubernetes development**

One of the problems we faced early on was: How best to include k8s into the inner loop of a developer development life-cycle. Some requirements are quickly apparent. How to build and sync code into containers and do it as fast as possible. Initially we had a simple homegrown solution to tap into filesystem events on source changes and just rsync everything to containers. We also experimented with projects including Skaffold from Google and Draft from Microsoft trying to solve the same problems. Eventually what worked for us is Tilt from Windmill Engineering (shoutout to Daniel Bentley), who have an excellent product on their hands with a workflow driven by Tiltfile written using starlark language. It is able to watch files for edits, can apply changes automatically, build container images in real-time, make build faster with in cluster builds and skip registries and has a nice UI to see all information about your services in a single pane, etc. And when you want to dig a bit deeper we open the nitty gritty of k8s through an awesome CLI tool called K9s to interactively run k9s commands and simplify developer workflows. Today all of our workloads running in k8s are developed in cluster with a cohesive and fast experience across projects thanks to helm / tilt / k9s and anyone new joining in can easily onboard with a few commands.

**Prometheus, AlertManager, Jaeger, Grafana and Loki—Observability**

We rely heavily on the Prometheus monitoring solution and time series database (tsdb) to gather, aggregate and forward our metrics scrapped from individual services. Prometheus ships with an excellent querying language PromQl and alerting solution Alert Manager. Jaeger forms the backbone for our trace aggregation. Recently we started shifting from Graylog to Loki as our logging backend to provide a similar experience as Prometheus. This all is done to provide a single pane of glass where all observability requirements can be met and we intend to deliver that with Grafana which is our charting solution. To orchestrate all these services we rely heavily on the Prometheus Operator project with integrations on top to manage the life-cycle of many multi-tenant Prometheus deployments that we have today. At any given time we ingest hundreds of thousands of time series to gather insights about running infrastructure and services to aid when something fails.

In the future, we plan to either integrate Thanos or Cortex project to address Prometheus scalability challenges and provide a global query view, high availability and data backup for historical analysis.

**Luigi and Jenkins—Automating Data-pipelines**

We use Luigi and Jenkins to orchestrate and automate our data pipelines. The batch jobs are submitted via steps to EMR and Luigi helps us to build highly complex batch workflows on top of these jobs. Jenkins is utilized to trigger a series of operations in course of an ETL process providing us control over automation and use of resources, drilled down to individual task.

We package and version our batch job code in versioned docker containers, to keep a consistent developer and production experience.

**Addon Projects**

We also include many other projects developed by the community, which are delivered as addons and maintained as part of the cluster life-cycle to provide value to the services deployed both in production and development environments. We discuss them briefly below:

- Argo Workflow and CD—Evaluating as an alternative to Jenkins for batch processing task and continuous deployment efforts.
- AWS IAM Authenticator—User identity management in k8s.
- ChartMuseum—Serves our remote helm charts.
- Cluster Autoscaler—Manages scaling of on-demand and spot fleets in our clusters.
- Vertical Pod Autoscaler—Vertically scales pods where necessary or based on custom metrics.
- Consul—State store for many projects, where we can we try to move to Custom Resource Definitions (CRDs).

- External DNS—Maps DNS records to Route53 for external and internal access.
- Kube Downscaler—Downscales deployments and statefulsets when they are no longer in use.
- Kube2IAM—Transparent proxy to restrict aws metadata access and role management for pods.
- Loki / Promtail—Log shipment and aggregation.
- Metrics Server—Metrics aggregation and interfacing for other consumers.
- Nginx Ingress—Ingress controller for internal and external services. We continue to evaluate other ingress controllers for extended API gateway functionalities including Gloo, Istio ingress gateway and Kong.
- Prometheus Operator—Prometheus operator stack to provision Grafana, Prometheus, AlertManager and Jaeger Deployments.
- RBAC Manager—An operator to easily manage Role Based Access Control for k8s resources.
- Spot Termination Handler—Handles spot termination gracefully by preemptively cordoning and draining nodes.
- Istio—We continue to evaluate Istio for its mesh, observability, traffic routing and shaping capabilities. For many of these features, we have built solutions in house, which have shown limitations over time, which we intend to fulfill with this excellent project.

Our experience with k8s coupled with excellent community tooling has enabled us to not only ship our core stateless services providing search, but also helped us run large and critical stateful workloads like Cassandra, Kafka, Memcached and RocksDB in multiple availability zones and clusters, for high availability and replication. We have developed additional tooling to manage and safely execute these workload for our scale within Kubernetes.

## Local Development with Tilt—An end to end use case.

So far we covered a lot of ground describing all the tools we use, but we would like to give a concrete example on how this tooling, well, parts of it, is affecting the typical day-to-day workflow of our developers.

Let us take the example of an engineer working on Ranking. Previously, the workflow was:

1. Start a spot instance with a custom OS image, tag instance and associated resources with ownership information.
2. Rsync application code to the instance and install application dependencies.
3. Figure out setting up other services like api-gateway and front-end, their dependencies and deployments.
4. Configure them to work well with one another.
5. Start working on the Ranking application.
6. At the end, once done, make sure to **terminate** the instance.

We can see that, one needs to follow a series of steps repeatedly, over and over again and every new engineer in the team will have to repeat it, which is a total waste of developer productivity. If the instance is lost we repeat. Also there is a stark difference between the production and local development workflow, leading to inconsistencies at some point of time. One might also argue the need for setting up other services like front-end along with ranking, but aim is to be generic here and in addition its always good to have complete visibility of the product. Additionally, as the team size grows, more cloud resources are created and more resources are under-utilized. Engineers let instances run, because they don't want to redo the setup process every day. If a team member leaves and has an orphan instance without sufficient tags, it is a challenge to identify whether or not it is safe to turn off the instance and delete said cloud resources.

What would be ideal, is to provide an engineer with a base template to set up local environment with his own full version of SERP along with other services responsible for ranking. We configure the base template generically, which tags resources created by the user with thier unique identifier names and allows them to control the life-cycle of their application. As K8s already abstract out need to launch instances and managing them (we centrally administer them using KOPS), we use the template to set defaults (auto downscaling during non-work hours) and hence tremendously reduce costs.

All the user now has to care about is the code written in his local editor and our tooling which is composed of a combination of Docker, Helm and Tilt on top of Kubernetes facilitates this said workflow, working behind the scenes like magic.

Here is an example Tiltfile, which describes the services and other dependent services required to setup a mini-SERP version. For launching these services in development mode, all the user has to do is run:
`tilt up`

```
# -*- mode: Python -*-

"""
This Tiltfile manages 1 primary service which depends on a number of other
micro services.
Also, it makes it easier to launch some extra ancilliary services which
may be
useful during development.
Here's a quick rundown of these services and their properties:
* ranking: Handles ranking
* api-gateway: API Gateway for frontend
* frontend: Server Side Rendering for SERP

"""

####################
# Project defaults #
####################

project = "some-project"
namespace = "some-namespace"
chart_name = "some-project-chart"
deploy_path = "../../deploy"
charts_path = "{}/charts".format(deploy_path)
chart_path = "{}/{}".format(charts_path, chart_name)
values_path = "{}/some-
project/services/development.yaml".format(deploy_path)
secrets_path = "{}/some-project/services/secrets.yaml".format(deploy_path)
secrets_dec_path = "{}/some-
project/services/secrets.yaml.dec".format(deploy_path)
chart_version = "X.X.X"

# Load tiltfile library
load("../../libs/tilt/Tiltfile", "validate_environment")
env = validate_environment(project, namespace)

# Docker repository path for components
serving_image = env["docker_registry"] + "/some-repo/services/some-
project/serving"

####################################
# Build services and deploy to k8s #
####################################

# Watch development values file for helm chart to re-execute Tiltfile in
case of changes
watch_file(values_path)

# Build docker images
# Uncomment the live_update part if you wish to use the live_update
function
# i.e., no container restarts while developing. Ex: Using Python debugging
docker_build(serving_image, "serving", dockerfile="./serving/Dockerfile",
build_args={"PIP_INDEX_URL": env["pip_index_url"], "AWS_REGION":
env["region"]} #, live_update=[sync('serving/src/', '/some-project/'),]
)

# Update local helm repos list
local("helm repo update")

# Remove old download chart in case of changes
local("rm -rf {}".format(chart_path))

# Decrypt secrets
local("export HELM_TILLER_SILENT=true && helm tiller run {} -- helm
```

```
secrets dec {}".format(namespace, secrets_path))

# Convert helm chart to standard k8s manifests
template_script = "helm fetch {}/{} --version {} --untar --untardir {} &&
helm template {} --namespace {} --name {} -f {} -f
{}".format(env["chart_repo"], chart_name, chart_version, charts_path,
chart_path, namespace, env["release_name"], values_path, secrets_dec_path)
yaml_blob = local(template_script)

# Clean secrets file
local("rm {}".format(secrets_dec_path))

# Deploy k8s manifests
k8s_yaml(yaml_blob)

dev_config = read_yaml(values_path)

# Port-forward specific resources
k8s_resource('{}-{}'.format(env["release_name"], 'ranking'),
port_forwards=['XXXX:XXXX'], new_name="short-name-1")
k8s_resource('{}-{}'.format(env["release_name"], 'some-project-2'),
new_name="short-name-2")

if dev_config.get('api-gateway', {}).get('enabled', False):
  k8s_resource('{}-{}'.format(env["release_name"], 'some-project-3'),
port_forwards=['XXXX:XXXX'], new_name="short-name-3")

if dev_config.get('frontend', {}).get('enabled', False):
  k8s_resource('{}-{}'.format(env["release_name"], 'some-project-4-1'),
port_forwards=['XXXX:XXXX'], new_name="short-name-4-1")
  k8s_resource('{}-{}'.format(env["release_name"], 'some-project-4-2'),
new_name="short-name-4-2")
```

Notes:

1. Helm charts are primarily responsible for the application packaging and managing their release lifecyle. We use helm templating and the use of customizable yamls for providing values to these templates. This allows us to configure releases to their very core. This includes allocating resources to containers, allowing an easy way to configure which services they connect to, the ports on which they can run, etc.

2. Tilt is used to setup the local k8s development environment with the provided helm charts and mapping local code to the services described in the charts. It provides functionality where we can continuously build docker container and deploy application to k8s or perform a local update (rsync local changes into a running container). One can also port forward the application to his local instance to access the service end-point while developing. In our case, we deploy using the k8s manifest by extracting the rendered template from the helm chart. This was due to our complex chart requirements where we are unable to fully utilize the helm functions provided with Tilt.

3. If the application endpoint needs to be shared with other team members, the charts provide a uniform mechanism to create internal ingress endpoints.

4. Our charts are served via the common helm charts repository, thus in production and development, we are using the same base code (versioned docker images), same chart template, but different values files for templating them according to our needs (deployment names, endpoint names, resources, replication, etc.)

5. We keep this practice consistent across each and every project providing a much easier onboarding experience and improved manageability and control over cloud resources.

   Any sufficiently advanced technology is indistinguishable from magic. ~ Arthur C. Clarke.

One caveat though, Magic, however, has its short-comings. It increases productivity, reliability and reduces costs by means of more efficient resource sharing. But, when something break, people are blind to what they issue must be and tracking down the problem become a difficult task, specially because things always tend to fail at the most inconvenient moment. So, as much as we are proud of our work, we stay humble knowing this fact.

## Optimizing Costs

Having a cheap infrastructure and a web scale search-engine do not go hand in hand, that said, there are ways by which you can save money. Let's discuss how we optimized for costs using our K8s based infrastructure:

1 . **Spot Instances**

- We heavily relied on AWS spot instances, as using them forced us to build the system for failures. But it was worth it, as they are a lot cheaper as compared to on-demand instances. Be wary though of not falling on the same self-inflicted damage we did. We were so used to those, that sometimes we outbid ourselves, happened more often than what we would have liked. Also, Don't exhaust high performance machines or you will get in a bidding war with other companies. Lastly, Never use spot GPU instances before a major NLP/ML conference.
- Mixed Instance Pools with Spot: Not only did we utilize spot instances for one-off jobs, but for service workloads. We came up with a neat strategy, where we create a node pool for running kubernetes resources using mixed instance types (but, similar configurations) which were distributed across multiple Availability Zones. This coupled with Spot Termination Handler allowed us to move our stateless workloads just in time to newly created or spare capacity spot nodes thus saving us from a potentially high downtime.

2 . **Sharing CPU and Memory**

As we are fully committed on Kubernetes, we discuss workload provisioning based on how much CPU or memory is necessary and how many replicas does one service need. In this, if the Request and Limits are equal we get guaranteed performance. Although, if the Request is low but Limit is high, which may be useful in case of sporadic workloads, we could over-provision and maximally utilize the resources on an instance (reduce idle resources on an instance).

3 . **Cluster Auto-scaler, Vertical and Horizontal Pod Autoscaler**

We deploy these to automate launching and downscaling of pods and in turn instances only when the need arises. This means when there is no workload, only the minimum set of instances are up and we don't need manual intervention to facilitate this.

4 . **Deployment downscalers in development environment**

We use deployment down-scalers to downscale pod replicas to 0 at specific times for all services in development setting. Using an annotation in our application's kubernetes manifest, we can specify an uptime schedule:

```
annotations:
    downscaler/uptime: Mon-Fri 08:00-19:30 Europe/Berlin
```

This means the deployment is downscaled to zero during non-working hours. And in turn the instances are automatically downscaled by the cluster autoscaler since there are no active workloads on the instance.

5 . **Cost Assessment and instance recommendations – Long term cost reduction**

In production, once we identify our resource utilization, we can select the instances which will be used heavily. Instead of on-demand, we could go towards a reserved instance pricing model which requires a minimum of 1 year upfront payment. In turn, the costs are significantly cheaper as compared to running instances on-demand.

For kubernetes, there are some solutions like kubecost, which monitors usage over time and based on that recommend additional ways to save costs. It also provides an estimated price for a workload, so one can get a decent idea of the overall cost for deploying a system. One is notified also of the resources which might not be of use anymore like ebs volumes, etc in one single interface.

All of these steps result in saving tens and thousands of euros for us annually. For larger organizations with high infrastructure bills, if executed properly, these strategies could easily save millions on average per year.

## Machine Learning Systems

Figure 5: Hidden Technical Debt in Machine Learning Systems — Sculley et al.[12]

It is interesting to note that our Kubernetes journey started in the most unexpected manner. We were exploring the idea to set up an infrastructure, which allows us to run distributed deep learning experiments using Tensorflow. At the time this idea was new and although Tensorflow had shipped distributed training a short while ago, apart from a few handful of very well resourced organizations, only few knew how to run and manage this workload End-to-End specially for large settings. It was also a time when there was no cloud offering which could help solve this problem.

We started out using a distributed setup deployed using Terraform, but we soon realized this solution has its limitations if we wish to scale it to all engineers of the organization. At the same time we found some community contributions, where plain Kubernetes manifests were generated via the use of smart jinja templating engine to create a distributed deployment of Deep Learning Training Application (Parameter Server & Worker Mode) This was our first contact with Kubernetes. In parallel, we started working on building our search to work near real-time, along-side experimenting with recency ranking. It was then, when kubernetes shone the brightest for us and we decided to get stuck in and dive deeper into it.

As part of our Machine Learning Systems journey, like with most of the infrastructure described above, our goal was to open it to the entire organization and make it easier for all developers to deploy applications on Kubernetes. We really wanted them to focus more on solving the problems instead of trying to solve infrastructure challenges associated with services.

But, from all the gains one gets from applying Machine Learning to their problems we quickly realize that maintaining machine learning systems is a real pain. It goes a lot deeper than just writing ML code or training models. Even for an organization at our scale, we need to address some of these issues. They are described in depth in the paper, “Hidden Technical Debt in Machine Learning Systems”[12:1]. It is a good read for anyone thinking of relying and running machine learning systems at scale in production. In our process, we looked at several solutions,e.g.:

- MLT[13]
- AWS SageMaker[14]
- Kubeflow[15]
- MLFlow[16]

Among all these, we found **Kubeflow** the most relevant, feature complete, cost effective and customizable for our needs.

We have also penned down some of these reasons[17] on the official Kubeflow Blog a while back. Apart from providing us with custom resources like TfJob and PytorchJob to run our training code, one of the benefits of kubeflow has been its out of the box excellent notebook support.

**Kubeflow use-case @ Cliqz**

Many of the features responsible for our near-realtime search ranking utilized notebooks from Kubeflow. An engineer can spin up a notebook in the cluster and directly tap into our data infrastructure (batch and realtime streaming). It made it easier to share notebooks and work on different parts of the code together. Experimentation became easier for engineers as they don’t have to re-setup notebook servers, provide perimissions to access data infrastructure and look into the gory details of deployment, using a simple web interface one can choose what resources they needed for the notebook (cpu, memory and even gpu), allocate an ebs volume and start up a notebook server. Interestingly, some of the experiments, were done on notebooks with as low as 0.5 CPU and 1 GB of RAM. Usually this capacity was readily available in our cluster and we could easily facilitate spawning of such notebooks without starting newer instances. In a different setting, if two engineers from different teams were working, they most likely will start their own instances. This leads to increased costs and under-utilization of resources.

In addition, Jobs can be submitted which can then be used to train, validate and serve the model from within the notebook. An interesting project in this regard is Fairing.

Kubeflow itself is a very comprehensive initiative and we have only started scratching its surface. More recently, we have also started looking into projects like Katib (hyperparameter tuning for machine learning models), KFServing (Serverless inferencing of Machine Learning models on Kubernetes), Kubeflow Pipelines (End to End Machine Learning Pipelines for Kubeflow) and TFX (Create and manage a production ML pipeline). We already have some prototype around these projects and hope to release them to production in near future.

Given all these benefits, we would like to thank whole-heartedly the team behind Kubeflow for this amazing project.

…

As we grow and rely more on machine learning and its variants, we want the processes surrounding Machine Learning to be streamlined and be more reproducible in general. This is where things like model tracking, model management, data versioning and lineage becomes crucial.

To run things consistently at our scale where we apply periodic updates and assessments, we needed a solution around data management for serving models in production, which facilitates hot swapping of models and indexes in our live production services autonomously. To tackle this issue, we built a solution in-house “Hydra” which provides downstream services with the capability of performing a dataset pub-sub. It also ensures volume management for services in a Kubernetes cluster. We will describe this in detail in a future post.

## Final Words

> Once you’ve found success, your next goal should be helping others do the same. ~ Kelsey Hightower

Architecting Cliqz has been challenging and fun at the same time. We believe there is still a long road ahead of us. As with growing development in this space, there are several possibilities and routes to take.

Even though Cliqz employs 120+ people, the codebase is effectively developed by thousands of passionate open source developers, distributed globally, devoted to ship high quality code, and build safe and secure software systems for mankind. We would not have reached where we are without them. We would like to thank the open source community from the core of our hearts to be really generous and helpful in providing us with solutions, whenever we were stuck. Through this post, we wished to share our struggles, experiences and solutions for others who might have similar problems and are looking for answers. In the spirit of openness, we too are contributing back to the extent of our scarce resources here.

A fully private search offering from Cliqz is our contribution towards a privacy focused web, a daunting but not an impossible task. We invite you to try out our search and download our browsers on desktop and mobile to be part of this mission. And, if you enjoy working on such problems come and join us.

*Auf Wiedersehen (until we see each other again)*

## Remarks and references

---

1. https://0x65.dev/blog/2019-12-11/the-pivot-that-excited-mozilla-and-google.html ↩
2. https://0x65.dev/blog/2019-12-07/indexing-billions-of-text-vectors.html ↩
3. https://aws.amazon.com ↩
4. https://0x65.dev/blog/2019-12-08/how-do-you-spell-boscodictiasaur.html ↩
5. https://0x65.dev/blog/2019-12-06/building-a-search-engine-from-scratch.html ↩
6. https://0x65.dev/blog/2019-12-09/cliqz-rich-results.html ↩

7. https://en.wikipedia.org/wiki/A/B_testing ↩
8. https://martinfowler.com/bliki/BlueGreenDeployment.html ↩
9. https://martinfowler.com/bliki/CanaryRelease.html ↩
10. https://kubernetes.io ↩
11. https://www.cncf.io/ ↩
12. https://papers.nips.cc/paper/5656-hidden-technical-debt-in-machine-learning-systems.pdf ↩ ↩
13. https://www.intel.ai/mlt-the-keras-of-kubernetes/ ↩
14. https://aws.amazon.com/sagemaker ↩
15. https://www.kubeflow.org ↩
16. https://mlflow.org ↩
17. https://medium.com/kubeflow/why-kubeflow-in-your-infrastructure-56b8fabf1f3e ↩