Kubernetes & production. To be or not to be?

Hundreds of containers. Millions of external requests. Myriads of internal transactions. Monitoring and incident notifications. Simple scalability. 99.9% uptime! Deployment and rollback.

Kubernetes as a magic pill! To be, or not to be: that is the question!

Disclaimer

Although this article has been published, I had initially written it for personal use as a form of rubber duck debugging. Having played around with hipster technologies for more than two years, I feel that I should take a pause and reflect on how feasible those technologies have proven to be, and whether they would fit the needs of any of my upcoming projects. Nevertheless, I really hope that this material will find its reader and eventually help some of you decide whether to try and use Kubernetes or walk past it.

The article summarizes the experience that we've had at Lazada E-Logistics, the logistics wing of Lazada Group, which, in turn, is a subsidiary of Alibaba Group. Here, we build and maintain systems that help automate the entire supply and fulfillment chain in the 6 largest countries of Southeast Asia.

Background

One day, a representative of a worldwide cloud solutions company, asked me, 'What does "cloud" mean to you?' I hesitated a few seconds (and said to

myself, 'Hmmm…apparently, we're not going to talk about a dense visible mass of tiny water droplets…') and said, 'Well, they say a cloud is a fail-safe computer with unlimited resources and virtually no overhead of data transfer (network, disk, memory, and so on).' As if I had a personal laptop operating for the whole world and able to withstand such a load, and I could manage it on my own.'

But seriously, why do we need this magic cloud? It's easy! We strive to make life easier for developers, system administrators, DevOps, tech-managers—and a cloud, if made right, can help us with that! Not to mention that monomorphic systems that serve business needs have always been cheaper and always generated fewer risks. So we set out to find a simple, convenient and reliable private cloud platform for all our applications and suitable for all tech staff listed above. We did a little research: Docker, Puppet, Swarm, Mesos, Openshift + Kubernetes, Kubernetes—Openshift …. and stopped at the last one—Kubernetes with no add-ons whatsoever. The functionality described at the very first page was just what the doctor ordered for us and the company. A detailed study of the documentation, chatter with colleagues and a small experience of rapid testing—all of this gave us added confidence that the product was really capable of what it said it was, and that we would be able to get our hands on the magic cloud!

Buckle up and off we go…

Problems and solutions

3-layer architecture

Everything starts with the basics. In order to create a system that can live well in a Kubernetes cluster, you will have to think through the architecture and development processes, configure a bunch of mechanisms and delivery tools, learn to tolerate the limitations/concepts of the Docker universe and its isolated processes. As a result, we come to a conclusion that the ideology of the micro-service and service-oriented architecture unfortunately suits our needs. If you read the Martin Fowler's article on the subject, then you can imagine what herculean work has to be done before the first service comes to life. My checklist divides the infrastructure into three layers and roughly describes what you need to consider at each level when building a similar system.

The three layers in question are:

- Hardware—servers, physical networks
- Cluster—in our case, Kubernetes and the system services supporting it (flannel, etcd, confd, docker)
- Service—the process itself, packaged in Docker—a micro- or macro-service in its domain

In general, the idea of a 3-layer architecture and tasks associated with it deserves a whole new article. But it won't be released until the check-list is immaculately complete. This may never happen :)

Qualified engineers

In light of the growing interest in private cloud technologies, medium and large businesses face a surging need for architects, DevOps, developers, and

DBAs who would be qualified enough to work with that.

The reason is that when innovative technologies enter the market, they simply are 'too new' to have sufficient documentation, tutorials or say answers on 'Stack Overflow'. Despite this, however, technologies like Kubernetes have become extremely popular, which caused companies to rush to hire people with relevant experience, which caused shortage of staff in the field. The solution is simple—we need to nurture in-house engineers! Luckily for us, we already knew what Docker was and how to cook it—and we just had to catch up with the rest.

Continuous Delivery/Integration

Despite the beauty of the 'smart cloud cluster' technology, we needed tools to enable installation and communication of objects inside Kubernetes. Having worked our way up from a bash script and hundreds of branches of logic, we managed to create clear and readable recipes for Ansible. To fully transform Docker files into live objects, we used the following:

- A set of standard solutions:
  - Team City—for automated deployment
  - Ansible—to build templates and to deliver/install objects
  - Docker Registry—to store and deliver Docker images
- images-builder—a script to recursively search for Docker files, build images from these files, and subsequently push the images to a centralized registry

- Ansible Kubernetes Module—a module for installing objects with different strategies depending on the object (‘create or update’/ ‘create or replace’/ ‘create or skip’)

Among other things, we were looking at Kubernetes Helm. But we just could not find the killer-feature that would lure us in abandoning or replacing Ansible templates with the Helm charts. Because except for the charts, we didn’t see any other benefits of using Helm.

For example, how to check if an object has been successfully installed and we can continue to roll out the rest? Or how to fine-tune and install containers that are already working, when we just need to execute a couple of commands inside them? These and many other questions make us treat Helm like a simple template engine. But why use it…if Jinja2, which is part of Ansible, can easily outmatch any third-party solution?

Stateful services

As a comprehensive instrument for any type of services including stateful ones, Kubernetes comes with a set of drivers for network block devices. In case of AWS, this device being EBS.

But as you can see, the k8s tracker is full of EBS-related bugs, and they are being resolved quite slowly. As of today, we don’t have any troubles, except maybe one—sometimes it takes 20 minutes to create an object with a persistent storage. The EBS-k8s integration is of very, very, very questionable quality.

However, even if you use other storage solutions and don't have any issues, you still want to have reliable technologies for everything that can store data. So it took us quite long to fill in the gaps and identify trustworthy solutions for each of the cases:

- PostgreSQL cluster (see article on Medium)
- RabbitMQ cluster
- Redis cluster
- Backup script for PostgreSQL

What's more, Kubernetes as such and the Docker world in general sometimes make you resort to certain tricks and subtleties, which might seem obvious at first glance, but usually require an out-of-the-box solution.

A quick example. You can't collect logs inside a running Docker container. BUT a lot of systems and frameworks can't stream in 'STDOUT' as-is. It requires 'patching' and tweaking at the system level: writing in pipes, monitoring processes, etc. We filled this gap with Monolog Handler for 'php', that can stream logs in a way that Docker/k8s can understand them.

API gateway

As part of any micro-service and service-oriented architecture, you will most likely need a certain gateway. But this is for architecture, whereas here I want to focus your attention on why this is especially important for the cluster and the services inside it. Everything is pretty simple—we just need a single point of *(-failure-)* access to all of the services.

We had a few requirements in context of the Kubernetes cluster:

- **External access control and limitation of external requests**—as an example, a small LUA script sheds the light on the problem
- **Single point of authentication/authorization** for all services
- **Very few services that require HTTP access from the outside world**—it's harder to configure redundant connections on the server for each of such services, than to manage routing in nginx
- **Integration of Kubernetes-AWS** to work with AWS Load Balancer
- **Single point for monitoring HTTP statuses**—this also facilitates internal communication between the services
- **Dynamic routing** of requests to services or service versions, A/B tests (alternatively, this can be achieved by using various pods behind Kubernetes services)

A sophisticated Kubernetes user will quickly make a point about Kubernetes Ingress Resource, which was designed specifically for dealing with such problems. All right! But as you might have noticed, we needed a bit more 'features' for our API gateway than Ingress could offer. Moreover, Ingress is an nginx wrapper, and we already knew how to work with nginx.

Current state

Despite the myriad nuances and problems related to the installation, use and maintenance of the above solution, if you're persistent enough, you will most likely achieve the result and create something similar to what we have today.

So how does the platform look today? A few dry facts:

- Just **2–3 engineers** to maintain the entire platform
- **One repository** that stores data about the entire infrastructure
- 10–50 independent automated releases per day—**CI/CD** mode
- Ansible as a cluster **management tool**
- **Just a few hours** to build a live-like environment locally on minikube or on live servers
- **AWS-based architecture** provided by EC2 + EBS, CentOS, Flannel
- **500–1000 pods** in the system
- **Technology stack** wrapped in Docker/k8s: Go, PHP, JAVA/Spring FW/Apache Camel, Postgres/Pgpool/Repmgr, RabbitMQ, Redis, ElasticSearch/Kibana, FluentD, Prometheus, and more
- **No infrastructure outside the cluster** except for monitoring at the 'Hardware' level
- **Centralized log repository** based on ElasticSearch inside a Kubernetes cluster
- **Single point for metrics collation** & emergency notifications powered by Prometheus

This list reflects a lot of facts, but we deliberately didn't mention the obvious advantages and nice features of Kubernetes as a container management

system. You can find out more about those on the Kubernetes websiteor in the corresponding articles on Habrahabr (Russian) or Medium (English).

There's also an extensive list of our dreams and wishes, which are still being prototyped or cover too small an area of the system:

- **Profiling and tracing system** such as zipkin
- **Anomaly detection**—machine-learning algorithms for analyzing problems in hundreds of metrics, which helps a lot when you can't or don't want to understand what each metric/set of metrics means, but want to be alerted about the problems associated with them
- **Automatic capacity planning** and scaling of both the number of pods in a service and the number of servers in a cluster based on certain metrics
- **Intelligent backup management system** for any stateful services, primarily for databases
- **Network monitoring and connection visualization**—inside a cluster as well as between services and pods (a good example)
- **Federation mode**—managing several distributed clusters using single control plane

So, to be or not to be?

Most likely, an experienced reader already guessed that the article hardly gives any unambiguous answer to this seemingly simple question. A lot of details and trivialities can make your system insanely cool and productive. A lot of bugs and poor implementations can turn it into a disaster.

So it's up to you! But my resolution is: '***Yes! But be careful…***'

PS:

The original article was written in Russian, so thanks to my colleagues Sergey Rodin and Stuart Remphrey for help to translate it!