End-to-End Kubernetes

Moshe Zadka – https://cobordism.com

Acknowledgement of Country

Belmont (in San Francisco Bay Area Peninsula)
Ancestral homeland of the Ramaytush Ohlone

0.1 Introduction

Kubernetes is an interesting solution. It is a container orchestration, that has managed to become the standard in orchestration. It is not the only solution, but it is by far the best supported one, especially across environments.

One option is to treat Kubernetes as the "target". Develop and test somewhere, and then deploy to Kubernetes. That would not be bad, but Kubernetes is capable of so much more.

The other option is to lean in to Kubernetes as hard as possible. Fully commit. Kubernetes can be the local development environment, the way to run unit tests, they way to build, and the way to run functional tests.

In order to understand how this could work, concrete examples are useful. Examples are "caricatures", in the most honest sense. They exaggerate some characteristics, and ignore others. Just like in a caricature, the goal is to have something that is "true to life" without having all the messy complexities of real life.

All of the examples here are fully, and unashamedly, caricatures.

0.1.1 Examples as caricatures

Examples as Caricatures

Realistic exaggeration

0.1.2 Example application

So what better way to start then with a caricature? Kubernetes is a way to run applications, and it is useful to have an application in mind. The following is a silly REST-ful service that adds two numbers.

A Caricature

```
# e2e_k8s/demo.py
from pyramid import response, config, view
@view.view_config(route_name='add')
def add(request):
    x, y = (int(request.matchdict[c]) for c in "xy")
    return response.Response(str(x+y))
with config.Configurator() as _cfg:
    _cfg.add_route("add", "/add/{x}/{y}")
    _cfg.scan("e2e_k8s")
    application = _cfg.make_wsgi_app()
```

With a few more lines of Dockerfile, which are not necessary right now, it is possible to containerize the application. Though this application is short, it could still have useful unit tests, and, as it is currently written, would fail a typical Python linter.

This serves well for the caricature: this application needs testing and linting. The container build can use a multistage build to have a "dev" container that can run the tests and linter.

This means that the first "easy" step is finished. This is all running in containers. The next question is: can we run this on Kubernetes... End-to-End?

0.1.3 Software as a pipeline

Before talking about how to do this, what is an "end"? The end-to-end metaphor assumes software development is like pipes.

Code comes in at one end. This is the job of the software developer: to put in good code. On the other side, customer value is created. This is how the software developer salary is paid.

This is not to say the pipeline is simple. On the contrary, its complexity, and how to reduce it, is the focus of the talk. But nothing changes the *ends* of the pipeline: those are defined by the business. Code comes in, customer value comes out.

Software as a Pipeline

Code → Customer value

0.1.4 Software as a leaky pipeline

Sitting with the pipeline metaphor a bit more, this is a complicated pipeline. Code has bugs, missing features, security issues, tech debt, and more. "Good code" means "code that gives good customer value". How do you know if code is good when you write it?

One option is to throw the code directly in the customer's face. The problem is that typical first drafts are really bad. The goal of the "pipeline" is to get feedback about the code before the customer offers it.

The pipeline is leaky, as anyone who had to fix a bug that was reported by a customer knows. The pipeline is leaky, as anyone who has had to push the "rerun" button on the CI to make a run green knows.

Steps earlier in the pipeline offer faster, but less accurate, feedback. A "leak" is when the feedback does not match what would have been the "real" feedback.

Leaky Pipeline

Leak:

Problem happening at one stage but not the previous one

Now is the time for another caricature: the one of the pipeline. The code runs unit tests, which can leak. After passing them, it can run local functional

test. Then it is moved through progressively "close to production" environment, until it is finally deployed to production.

Pipeline Caricature

Code
Unit tests
Running locally
"Simulation" environments
Production
Customer

0.1.5 Goal: fixing leaks

The reason End-to-End Kubernetes is useful is because *it fixes leaks*. It does not fix all of the leaks! But it does solve leaks that are caused by differences in the structure of the environment.

Since all environments are K8s, that particular difference is eliminated. This also gives us a chance to reduce other differences: make sure that the K8s configurations in different environments are as close as possible.

Fixing Leaks

Less differences – less leaks

0.1.6 Talk goal: Mental model

Giving a complete solution is difficult. Each situation is a little bit different. There would also be so many details, this could easily be a 6-hour tutorial.

A more realistic goal would be a *mental model* of how kubernetes works, how it works in different environment, and how to take advantage of it. Using the mental model, it is possible to find good local solutions.

Model, Not Solutions

Hard problem to solve
Better mental model for local solutions

0.2 Kubernetes Crash Course

0.2.1 What is kubernetes?

Now it is time to define the second part of the title: Kubernetes. *Sigh* No, that would take too long. There needs to be some common ground, and Kubernetes is still new enough, that at least a quick tour of the highlight is in order.

First order of business: what problem does Kubernetes solve? The name is "orchestration". The thing being orchestrated here is which container runs on which compute resource.

Sounds simple? It would be unless it supports:

- Dealing with crashed containers
- Dealing with crashed compute resources
- Dealing with elastic needs for how many, and what kinds, of containers.
- Dealing with elastic amounts of available resources.

Oh yeah, and the original, static problem? It's NP complete. So, not so simple.

Kubernetes
Orchestration: Running containers on compute resources

0.2.2 Why is kubernetes popular?

Kubernetes is now supported in a wide range of environment. It's supported because people want to use it.

The first reason why it became popular now is because it builds on containers, and containers have enjoyed popularity. Previous computation orchestation solutions had to come up with their own alternative to containers, which meant developers had to understand a completely new target.

Kubernetes has a documented REST API. It is not *well* documented: the documentation is definitely so-so at parts. It is *thoroughly* documented. This is nice, and useful.

Finally, because of the REST API, and a few more nifty things, Kubernetes is extensible. Maybe not "easily" extensible, but at least "straightforwardly" extensible. Writing a new extension to Kubernetes can be done in one evening. (Ask me how I know.)

Why Kubernetes?

- Builds on containers
- Documented REST API
- Extensible

0.2.3 What is DevOps?

Kubernetes, and containers, are partially a solution to DevOps. Understanding DevOps is important to understanding how to use them well.

DevOps is not an engineering speciality, although the title "DevOps engineer" exists. It is not the idea that "developers should run their own code", although this is a common pattern. It is not the idea that operations is obsolete, and developers or automation can replace them, although reducing operational load is one of the goals.

It is also not ordered. It is not "Developers *telling* Operations what to do" or "Developers *report* to Operations".

What is DevOps (Not?

an engineering speciality

"developers should run their code"

"no need for operations"

"developers working for ops"

"ops working for developers"

So after talking about what it is not, what is it? The idea of DevOps is tied to the idea of the software pipeline, and leaks in it. The idea is that the *social* way to reduce the leaks in the pipeline is by having Developers, Operations, QA, Security, and everyone else working on the pipeline collaboratively.

A DevOps engineer is someone who can help build that collaboration. As part of the collaboration, the developers will often run their code in more environments. This can help reduce operational load. The idea is simple: developers are working *with* operations.

What is DevOps?

Developers, ops (and QA, and security, ...) collaborating on a pipeline

0.2.4 Why is DevOps good?

DevOps is a social convention, not a technical solution. The social convention, by Conway's law, does constrain the relevant technologies. One way that containers helped DevOps was by giving developers and operations a better way to collaborate by introducing the container as the "collaboration unit".

Kubernetes allows increasing the collaboration unit from the single container to the "container configuration". Not just which container to run, but also which containers to run together, how to tell when they are ready, and more. Kubernetes, in short, helps to work in a DevOps environment, or "do" DevOps.

How and Why DevOps

Reduce friction

Not just technical solution

Part of technical part: reduce differences between developer environment and production environment

0.3 Scope

0.3.1 Multi-cluster (beyond scope)

0.3.2 Database migration (beyond scope)

0.3.3 Stateful applications (beyond scope)

0.3.4 Advanced deployment (beyond scopre)

0.4 Software Development Lifecycle

0.4.1 What is an edit debug cycle?

0.4.2 Why edit-debug cycle latency matters

0.5 Environments

0.5.1 What is an environment?

Kubernetes is designed primarily around a microservice architecture. In this architecture, environment is a collection of services which talk to each other.

Environment
A collection of services which work together

0.5.2 Cross-talk

Environments can, and sometimes need to, communicate with each other. However, in general, environments do not talk *a lot*, and an environment will be *mostly* self-contained.

Caricatures are exaggerated. In a caricaturized environment, there is no cross-talk.

Environment Cross-Talk
Environments (mostly) don't cross-talk

0.5.3 Regional Separation

Some environments are separated by "region". This separation might be for pure geographical reason: distance, and network hops, cause latency. Sometimes it makes sense to do the entSomire computation, end-to-end, close to the client.

Another kind of regional separation is *jurisdictional*. This means that environments are separated for regulatory or legal reasons.

These are orthogonal, and sometimes contradictory, axis. For example, Ireland is part of the European Union, just like Germany. But Ireland is physically, and in network terms, close to Britain, which is not part ofthe European Union.

Depending on the needs, Ireland and England might share, or not share, an environment.

Sometimes regulatory needs mandate a different architecture. For example, maybe in some jurisdictions, choice of cloud provider is limited. Kubernetes shines here, as all major and minor providers support it.

Regional Environments
Jursidictional/Geographical

0.5.4 Maturity Separation

The more interesting separation is a "code maturity" separation. An environment is running code of a given maturity level.

"Production" is code that has been vetted as good enough to expose to real customers and real data. Other environments are usually less rigrously defined, and might serve different needs. Some environments are ad-hoc, Still more environments are "virtual": they will only run a subset of the services, and connect back to a different environment for the rest.

Environments do not have to be permanent. They can be ad-hoc: maybe running on a temporary EKS cluster, or running inside of a remote developemnt VM. The most ad-hoc of the environments is the one running in local development environments.

Maturity-Based Environments
Production
Stagingin, Testing....
Ad-hoc Remote
Local

0.6 Modern SDLC Caricature

An interesting thing about any environment that is not production, in general. It has *no value* except for feedbakc on code, and encouraging the code to be better.

All non-production environments are part of the software development lifecycle. Once again, here is a caricature of an SDLC. Code is written, reviewed by a peer, merged, and finally deployed.

Software Development Lifecycle (Caricature)
Develop
Review
Merge
Deploy

0.6.1 Develop

The development stage has the developer writing code. The developer tries to write code that accomplishes the goals it is set for. Sometimes this includes working from requirements, or doing research into what the requirements are.

The goal of the developer is write correct code quickly. Code on the developer laptpo is not useful. This step is done when the developer sends the code to a centralized source control, and asks for the code to be reviewed.

SDLC: Develop
Write code

0.6.2 Review

A colleague of the programmer who developed the code reviews the code changes. The reviewer checks if the code accomplishes the goal set for it. Goals include both fixing a bug or adding the original feature, as well as avoiding regressions and avoiding increasing tech debt.

If the reviewer thinks the code should be done better, the reviewer asks for changes. Otherwise the reviewer approves the change. This step ends when the reviewer approves.

SDLC: Review
Approve/Reject

0.6.3 Merge

The reviewer, the developer, or someone else, applies the change to the main branch. After merge, the CI system creates any artifacts needing to deploy the code.

SDLC: Merge
Integrate code into rest of product

0.6.4 Deploy

Finally, those artifacts run in the production environment. At this point, the benefit from the code changes starts accruing.

SDLC: Deploy
Run in production

0.6.5 No "test" stage

Curiously missing is a "test" stage. This is because each step includes a test, potentially – possibly more than one. When the developer develops the code locally, they get immediate feedback from unit tests, running the code in an ad-hoc way and seeing the results, or running local fnctional tests.

Before reviewing the code, the reviewer will check that the CI system has picked up the change and has run all relevant automated tests. The reviewer might also ask the original developer to test the artifacts in an environment, do it themselves, or ask a third-party.

After merging, the CI runs again. It should rarely fail, unless the tests are unreliable. For example, setting the main branch to "require tests to pass and branch to be up to date before merging" will mean code is only merged when it was already shown to work in PR CI.

The output from this stage might be used for further testing, by different people in different environemtn. This is the post-merge test.

Finally, after deploying to production are monitored and atypical monitoring data alerts someone. Think of this as a "post-deploy" feedback.

SDLC: No Test?

Every stage contains testing:
Develop: unit/ad-hoc/local
Review: Continuous Integration testing
Merge: Continuous Integration testing
Deploy: Monitoring and alerting

0.7 Kubernetes across environments

With these environments participating in this SDLC, what can kubernetes do? To a first approximation, an environment is almost already like a k8s cluster. Little cross-talk, since inter-cluster communictions have more friction.

In many cases, environments can *be* Kubernetes clusters, all configured by a shared source.

K8s across Environments

Environments are like k8s clusters
Can they be the same?

0.7.1 Why

This is useful, because production is often already kubernetes. By setting up other environments as kubernetes clusters, it is possible to achieve 1:1 parity.

Clusters as Environments: Why

Production will be k8s probably
Less leaks!

0.7.2 Challenges

But this is not as easy as it sounds. Environments need to be of different sizes. Whether they are maturity or regionally separated, it sometimes makes sense to have them vary by size.

They might also use different platforms. If nothing else, the local environment will probably not be the cloud provider's managed platform.

Finally, they often need to run different versions of the code. The whole point of a staging environment, for example, is that the code running there is a newer version currently being tested.

Clusters as Environments: How
Different sizes
Different platforms
Different versions

0.7.3 Lima

Lima will run a VM that is container-enabled on a Mac. Some tricks are used to make the network interfaces seem seamless.

Local environments: Lima
Run a VM running containers on Mac

0.7.4 WSL2

On Windows, WSL2 can run a container-enabled VM.

Local environments: WSL2
Run a VM (that can runn containers) on Windows

0.7.5 Minikube

Minikube can run in several modes, but one of them is running on a container-enabled VM in order to run a one-node kubernetes cluster. This means that whether on Mac with Lima, Windows with WSL2, or Linux with Docker and Containerd installed, minikube can spin up a one-node K8s cluster.

This can be useful both locally and on remote "development machines".

Local environments: Minikube
Single-host kubernetes
Lima, WSL2, Local linux, Remote VM....

0.7.6 Cloud

Finally, all reasonable cloud providers provide a managed Kubernetes platform. It is likely to be better than manualy installing it.

Remote environments: Cloud native

Part of cloud offerings!

0.8 Customizing Kubernetes

When running Kubernetes on all of these diverse environmens, there is one big problem. One size, quite literally, does not fit all. Different environments need to have different sizes.

There are many different aspects to size. The most obvious one is the the number of pods in a Deployment or StatefulSet. There are also other, subtler, parameters.

The size of the container, and attached temporary or permanent storage, is the most literal one when talking about "size". For example, maximum amount of memory or how much compute power to assign the container.

Then, for slightly more metaphorical use of "size", there are configuration parameters that need to be different. Specific configuration passed into the container might need to be different.

For example, if using 12-Factor, the environment variables containing the URLs of databases will be different. Typically, databases do not run inside Kubernetes, and even when they do, it is often via a separate cluster.

Finally, if nothing else, the environment needs to know what it is for user-visibility reasons. When looking at the UI or observability output from an environment, knowing which one it came from is between extremely helpful and crucial.

Customizing Kubernetes

One size

literally

does not fit all environments.

The typical Kubernetes configuration is done via YAML files which are processed by kubectl and sent to the API server. Because of that, customization is often thought of in terms of "preparing YAML files".

0.8.1 Ad-hoc

YAML files, at the end of the day, are text files. They can be procesed with sed, or other text processing tools.

They can also be converted to JSON by something like yq and processed via jq in ways that understand the tree structure. Ad-hoc scripts like that sound "low tech", and maybe even ridiculous.

Sometimes they are the most efficient way to change the configuration between environments. When using them, one interesting twist is that kubectl will take a YAML on its stdin, so it is possible to pipe the output directly into kubectl.

Customizing Kubernetes: Ad-hoc

Never underestimate a programmer with sed

0.8.2 Template

Another way to customize text has been carefully supported and optimized for decades: templating language. Regardless of your favorite templating language, Jinja2, mustache, or whatever else is already being used in the engineering organization, it can probably produce YAML as well as anything else.

Run the templating engine, spit out the YAML, and pass it to Kubernetes.

Since YAML is structured data, under this rubric you can also include any programming language that can serialize to YAML. The "template" is just code in, say, JavaScript or Python.

Customizing Kubernetes: Template

Use a generic templating language (e.g., Jinja2)

0.8.3 Kustomize

One programming language introduced by Kubernetes itself is kustomize. This is a DSL for modifying structured data, and expressed in YAML. You write "mini-programs" in the language (written in YAML) which express tree manipulation.

Those who are old enough to remember can see how the ideas behind XSL never die. If you don't remember XSL, that's probably a good thing though.

Customizing Kubernetes: Kustomize

Part of kubectl

YAML-based YAML-editing DSL

0.8.4 Helm

One templating language specifically invented for Kubernetes is "Helm". Helm comes with more sophisticated tooling, that understands concepts like "configuration repositories" or "configuration version".

It can be used directly with local templates, as well.

Customizing Kubernetes: Helm

Specialized templating system

0.8.5 Server-Side Apply

One option that works only for *specific* kinds of customization, and is kind of subtle to set up is server-side apply. With this introduction, you might wonder why this would be useful.

The biggest benefit of SSA is that it, after setting it up, it requires little on the *client* side. What server side apply is mark specific fields as controlled by specific "client identities".

For example, the replicas field can be specified by the "scalability" client, the image field can be specified by the "delivery" client, and all other fields by the "infrastructure-as-code" client.

In this case, the "infrastructure-as-code" client would run as part of the merge process into the "abstract-deployment-configurations" repository. The "scalability" client would run manually against the staging and local environments, and automatically against production. Finally, the "delivery" client would be run by a continuous delivery system, or some sort of version deployment infrastrucure.

Customizing Kubernetes: Server-Side Apply

Combine YAML from different sources

0.8.6 Operators

One way to make SSA even more useful is to use *operators*. Kubernetes Operators are a pattern which takes a custom object (standard−microservice.example.com, maybe) and create, delete, or update other objects based on that. It is possible to run operators as a deployment in the cluster, with a service account that has the right RBAC permissions.

The benefit of that is that standard−microservice can have separate fields that correspond to different clients, making SSA more useful. For example, instead of having an image field, it can have an image−base and a image−tag field, so that only the tag needs to be set by the delivery client.

Finally, operators are arbitrary code, which means that an operator can combine data from multiple objects. For example standard−microservice might describe the service configuration, microservice−version might only include the fields about which versions of containers to use with the microservice, and environment−size might include a field called factor to multiple all replica fields by. Then different objects can be set by different clients, with different permissions, without the need to set subtle SSA flags.

Customizing Kubernetes: Operators

Convert "abstract" description to "concrete" description

0.9 Kubernetes Application Architecture

Continuing with the theme of "caricatures", it is useful to have a caricature of how applications run on kubernetes.

Kubernetes Architecture Caricature

(Say that three times fast!)

0.9.1 Containers

The application starts with containers. From the point of view of the container builder, containers are a little like light-weight VM: the builder produces an (almost) complete Linux machine image.

From the point of view of the container *runner*, containers are a little like "heavy weight processes": there is little overhead, and a lot of observability into them. The different containers can be configured with resource limits, which is useful when running them.

Containers

"Light weight VMs"
"Heavy weight processes"

0.9.2 Pods

The containers are organized into Pods. Many applications, especially caricatures, will have only one-container-Pod. In almost all cases, a Pod will have one central container that does what it needs to. In some cases, there will also be a helper container or two.

The containers in a Pod share a network namespace: their 127.0.0.1 is the same. They can also share a process namespace, or storage, both ephemeral and durable.

Pods

Groups of containers
Share network namespace
Can share process namespace
Can share ephemeral storage
Can share durable storage

0.9.3 Deployments

Pods are important, but are rarely run directly by the end-user. The most caricatured of end-user objects is the "Deployment": a routable set of identity-less pods. By "identity-less", it means there is no coherent sense of "which one is the 5th container".

Deployment

Routable set of identity-less pods

0.9.4 Stateful Sets

Like every good forshadowing, the next object is the somewhat confusingly named "Stateful Set", which is not a set of stateful containers. Instead, it is a set of containers *with* identity. A container can know which one it is ("I am container number 5"), and it is possible to access a container by its identity.

StatefulSet
Set of identifiable pods

0.9.5 Naming

Kubernetes runs an internal DNS server. This server assigns names to different Kubernetes objects, and is the default DNS for all containers in the cluster.

Each Pod has a unique IP. This IP is assigned to a name that is based on the name of the Pod. Since Pods are often created dynamically for a Deployment or a StatefulSet, this name is often hard to predict.

Kubernetes routing: Pods
Unique IP
DNS depends on name

A more common name to use is that of a servie. Services' DNS names is based on the service name, which is explicitly set in the configuration.

A service is a "selector" that points to participating pods based on selection criteria. By default, services resolve to one IP, which is a "virtual IP" that routes each TCP connection to a Pod (which is ready at time of routing).

Services can also be headless, which means that the service will resolve to a set of DNS addresses for all ready pods which match. This allows the client to know, and log, which pod it connects to. It does require the client to re-resolve on disconnection, or it will never stop reconnecting to a down or dead Pod.

Kubernetes routing: Service
Select "participating" pods
Regular: route (usually TCP) to Pods
Headless: DNS to Pods

Assigning a service to a stateful set creates names which are based on the identity. Connecting to "pod number X in the service" will always resolve to a healthy pod.

Kubernetes routing: StatefulSet
With Service
Route to "name-number".suffix

0.10 Updating Containers

With the understanding of the Kubernetes application caricature, and environment caricature, you can set up multiple environments running the application. The simplest way to test new code is to build a new container from the code, and use whatever customization infrasture you use to set the version running in the environment to the new conainer.

Kubernetes-based Development: Basic

Build new container

Configure k8s with new contaier

Repeat

Simplest, but not fastest. There are ways to speed it up, though! Before trying any of the advanced techniques, start by optimizing the container build. Correctly taking advantage of caching options can bring a new container build to a matter of tens of seconds.

This is great! But it is still tens of seconds for any feedback. This needs to go faster

Kubernetes-based Development: Basic is Slow

Rebuild container (even with cache)

Redownload container

Restart container

Time for some subtle trade-offs! It is possible to update containers in place. Not recommended, perhaps, but sometimes it is the best option.

Kubernetes, fortunately, has all the tools needed for this. It does require some understanding of how containers work.

Kubernetes-based Development: Update Containers In-place

Quicker

less accurate

feedback

0.10.1 SSH to Pod

Common advice on the internet is "do not ssh into containers". This is, in general, good advice. But all good advice has its exceptions.

The one piece of that advice that is still relevant is to avoid ssh into the *application* container. Instead, the first step is to use the customization layer to add a container to the pod: one which is a dedicated SSH server. The SSH server can be configured with a public key to allow, so that only the local user can do bad things in the environment.

Kubernetes-based Development: SSH to Pod

With customization

Add container to pod

Running ssh server

0.10.2 Modifying files

The need for "custom" customization is not done! The next step is to configure the Pod to share the process namespace. The reason is kind of esoteric: using the process namespace, it is possible to "poke" files into another container's filesystem.

This means the SSH pod can run a command that checks the other Pod's files, and copies any relevant files over. These can be sources, for systems doing in-place source interpretation (like Python or Ruby), pre-processed artifacts (for example, post-processed JavaScript or byte-compiled Java), or full executables, if using a language like Rust and Go.

Kubernetes-based Development: Cross-Pod file-access

Share process namespace
Use proc filesystem
SSH pod can modify files in pod-friend

0.10.3 Synchornizing files

The next step is to make sure the SSH container has the files available. This can be done with rsync over SSH, systems like sshfs, or even ad-hoc scripts running scp. Together with the SSH Pod poking relevant files into the application container, this means that local files will be inserted into the application container.

Kubernetes-based Development: Sync files

Continuous sync
Over SSH
Over Pod

0.10.4 Auto-restarting

Some systems read files periodically: for example, CGI scripts. These are rare, because this is awkward and rarely a good idea. This means that for systems to *use* the files being poked into the application, some sort of restart is needed.

Luckily, customization can help here: the application container can be customized with a custom command or entry-point that has a restarter, or even an auto-restarter on file updates. This does mean the container needs to be built with this tool inside of it, even if it is dormant normally.

Sometimes this is not a problem. For "debug" builds, an extra dependency on watchmedo or a similar tool can be added. Potentially this is not something that needs to be done explicitly. For example, gunicorn can do this with a command-line switch.

If the container is a FROM scratch container with a statically-compiled language like Rust or Go, this becomes a bit more difficult. A small executable built in these languages which does a "if it shuts down, start again" loop can be inserted into debug builds. This means the SSH pod will have to be responsible for killing the process. Luckily, since they share a process namespace, this can be done by signaling the relevant process.

Kubernetes-based Development: Auto-restarting

Use customization

watchmedo and friends

0.11 Kubernetes-Oriented Development

0.11.1 Building containers

It is possible to build container images using Kubernetes. Buildkit, which is compatible with the common format of Dockerfile, can run in so-called "rootless daemonless" mode. This will allow it to run a build based on a Dockerfile.

It cannot save the image to the "local" image cache. The image needs to be sent to a registry. Depending on your needs, the usual registry that is being used locally might be fine, or you might want a custom one for different build set-ups.

It is possible to run a servicable registry using Kubernetes. This can be a useful option for local development.

Kubernetes Oriented Development: Build Images

Build container using rootless daemonless buildkit

Push to registry

0.11.2 Running tests and checks

With an SSH pod for synchronizing files, using kubectl to start a shell inside the container, and using customization to run a "development" container image rather than the usual runtime image, this can be used to run unit tests or lint inside of a container.

This can be useful to remove the need for a real "local" development environment completely, including unit tests and static file checks. Especially when the CI system, as is common, runs unit tests and other checks in a development container, this can be a good way to reduce discrepancies between the CI environment and the development environment.

Kubernetes Oriented Development: Local Integration

Unit tests, lint

Run in "development" container

0.12 Putting it all together

As mentioned in the beginning, this is not a solution! These are various pieces for a solution. Many of the pieces have several options. The best option to use depends on many things: compliance needs, technical stacks, and more.

End-to-End Kubernetes?

Put together the pieces

There were a lot of pieces to put together! It might be useful, now with the detailed explanations behind, to recap the pieces.

End-to-End Kubernetes Recap

Build environments

Implement customization

Build container images

Set up dynamic container updates

Set up local testing

Enjoy!

This is a lot of work. There better be a big benefit at the end of it. When this is all done, you can develop, review, merge, and deploy, using the same tooling, and synchronized configuration.

Kubernetes End-to-End: Benefits

Develop, Review, Merge, and Deploy

with synchronized Kubernetes configuration!

Many think of Kubernetes as useful as the "last step". Code first, then package it into containers, then explain to Kubernetes how to run it.

Kubernetes runs everywhere, so it is possible to reverse this order. Write Kubernetes configuration first for the new service, then build the containers that the configuration calls for, and only then start writing code. Sounds weird? No weirder than writing the function signature first and only then implementing the function. Kubernetes configuration are the "signature" for a microservice.

Kubernetes End-to-End: Start to Finish

It can run everywhere!

Less leakage in the SDLC pipe.