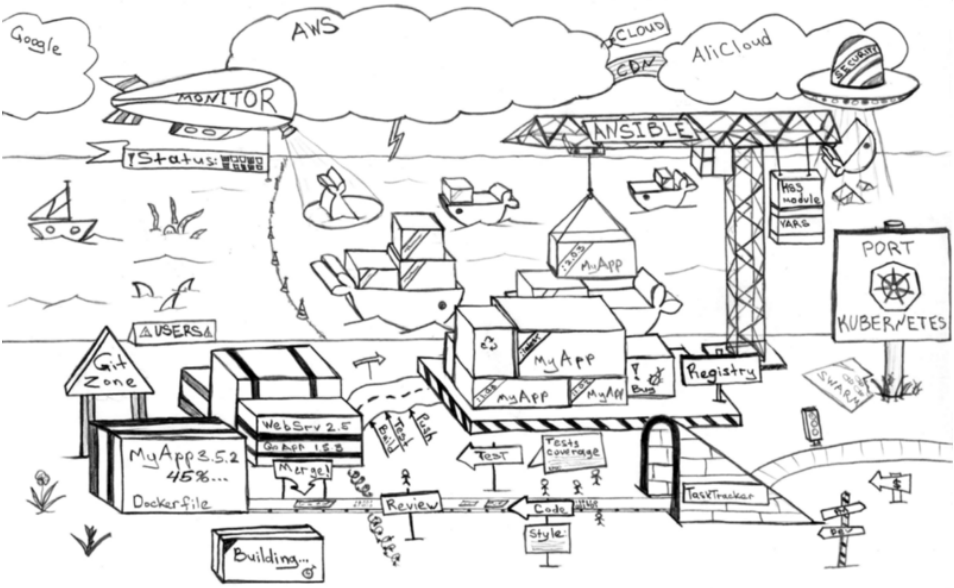
The best architecture with Docker and Kubernetes—myth or reality?



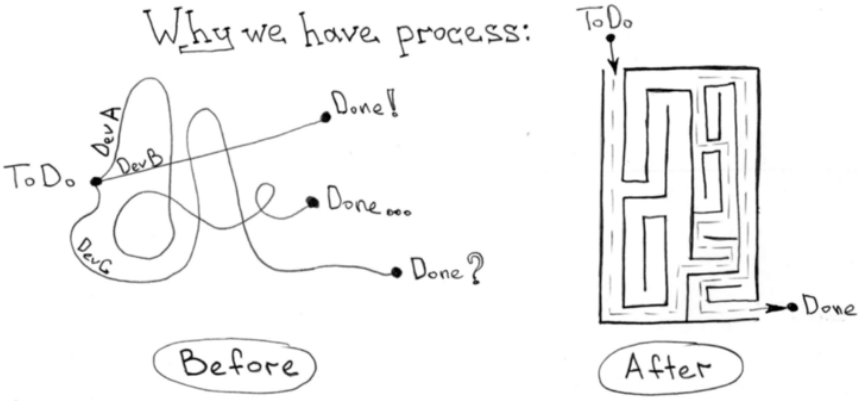
How has the world of software development changed in the era of Docker and Kubernetes? Is it possible to build an architecture once and for all using these technologies? Is it possible to unify the processes of development and integration when everything is “packed” in containers? What are the requirements for such decisions? What restrictions do they bring into play? Will they make life of developers easier or, instead, add unnecessary complications to it?

It's time to shed the light on these (and some other) questions! (In text and original illustrations.)

This article will take you on a journey from real life to development processes to architecture and back to real life, giving answers to the most important questions at each of these stops along the way. We will try to identify a number of components and principles that should become part of an architecture and demonstrate a few examples without going into the realms of their implementation.

The conclusion of the article may upset or please you. It all depends on your experience, your perception of this three-chapter story, and perhaps even your mood at the time of reading. Let me know what you think by posting comments or questions below!

From real life to development workflows



For the most part, all development processes that I have ever seen or been honored to set up served one simple goal—reduce the time between the birth of an idea and its delivery to production, while maintaining a certain degree of code quality.

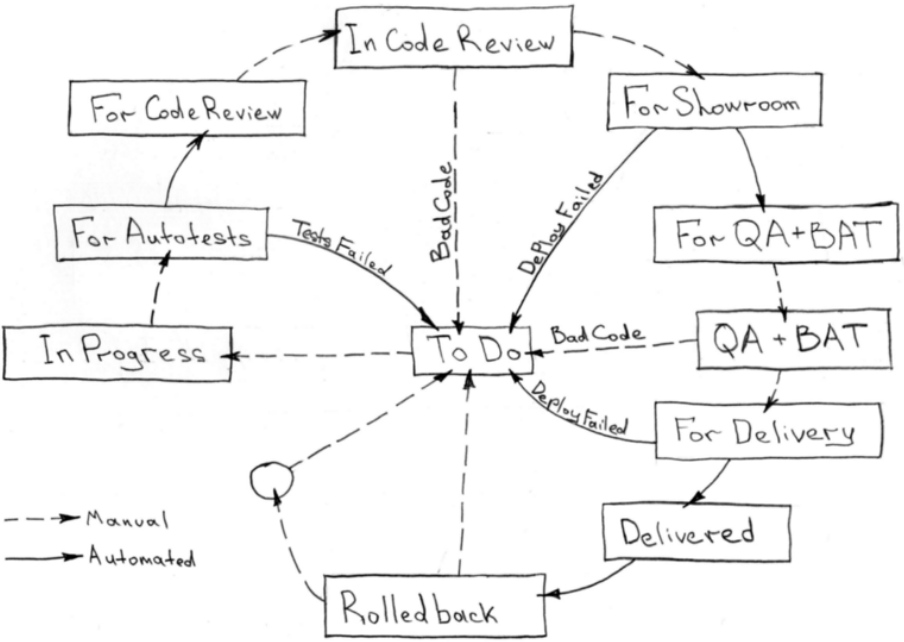
It doesn't matter whether the idea is good or bad. Bad ideas come and go fast —you just try them and turn them down to disintegrate. What's worth mentioning here is that rolling back from a bad idea falls on the shoulders of a robot which automates your workflow.

Continuous integration and delivery seem like a life saver in the world of software development. What can be simpler than that, after all? You have an idea, you have the code, so go for it! It would've been flawless if not for a slight problem—the process of integration and delivery is rather difficult to formalize in isolation from the technology and business processes that are specific to your company.

However, despite the seeming complexity of the task, life constantly throws in excellent ideas that bring us (well, myself for sure) a little closer to building a flawless mechanism that can be useful in almost any occasion. The most recent step to such mechanism for me has been Docker and Kubernetes, whose level of abstraction and the ideological approach made me think that 80% of issues can now be solved using practically the same methods.

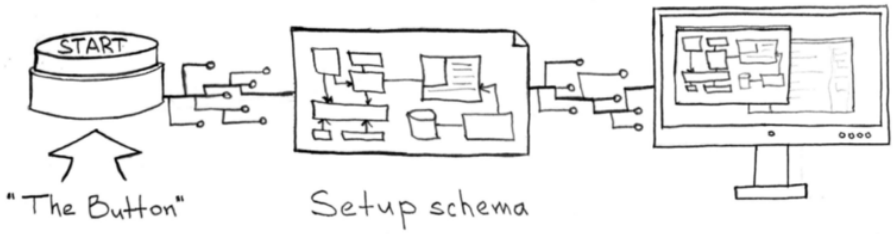
The remaining 20% obviously didn't go anywhere. But this is exactly where you can focus your inner creative genius on interesting work, rather than deal with the repetitive routine tasks. Taking care of the "architectural framework" just once will let you forget about the 80% of solved problems.

What does all of this mean, and how exactly does Docker solve the problems of the development workflow? Let's look at a simple process, which also happens to be sufficient for a majority of work environments:



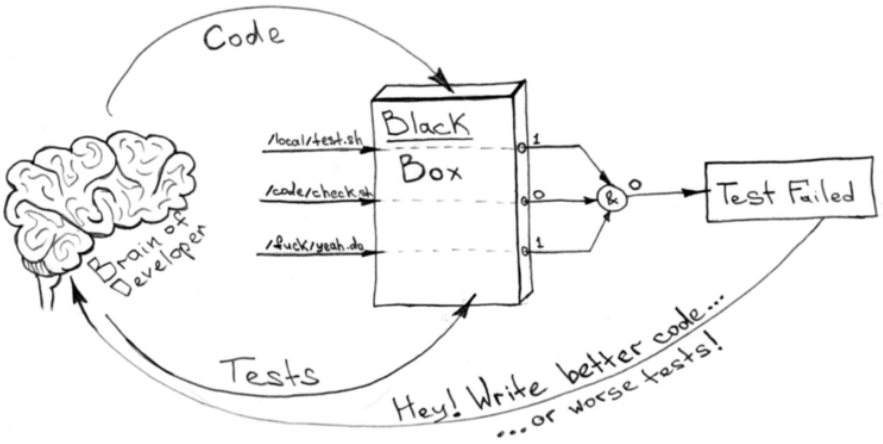
With due approach, you can automate and unify everything from the sequence below, and forget about it for months to come.

Setting up a development environment



A project should contain a `docker-compose.yml` file, which can spare you a trouble of thinking about what and how you need to do to run the application/service on the local machine. A simple `docker-compose up` command should start up your application with all its dependencies, populate the database with fixtures, upload the local code inside the container, enable code tracing for compilation on the fly, and eventually start responding at the expected port. Even when setting up a new service, you needn't worry about how to start, where to commit changes or which frameworks to use. All of this should be described in advance in the standard instructions and dictated by the service templates for different setups: `frontend`, `backend`, and `worker`.

Automated testing

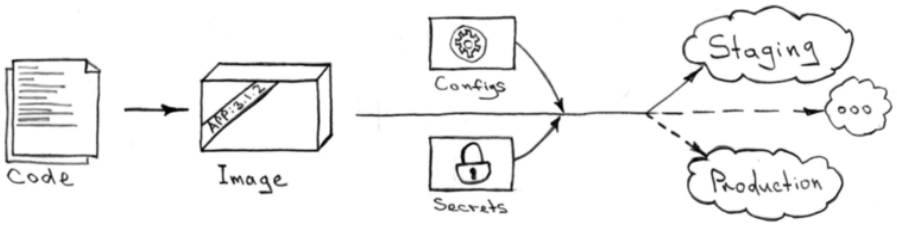


All you want to know about the "black box" (*more about why I call container this will follow later in text*) is that everything's all right inside it. Yes or no. 1 or 0. Having a finite number of commands that can be executed inside the container, and `docker-compose.yml` describing all of its dependencies, you can easily automate and unify testing without focusing too much on the implementation details.

For example, like this!

Here testing means not only and not so much unit testing, but also functional testing, integration testing, testing of (`code style`) and duplication, checking for outdated dependencies, violation of licenses for used packages, and many other things. The point is all of this should be encapsulated inside your Docker image.

Systems delivery



It doesn't matter when and where you want to install your project. The result, just like the installation process, should always be the same. There is also no difference as to which part of the whole ecosystem you're going to be installing or which git repo you'll be getting it from. The most important component here is idempotence. The only thing that you should specify is the variables that control the installation.

Here's the algorithm that seems to me quite effective at solving this problem:

1. Collect images from all of your `Dockerfiles` (for example, like this)
2. Using a meta-project, deliver these images to Kubernetes via Kube API. Initiating a delivery usually requires several input parameters:

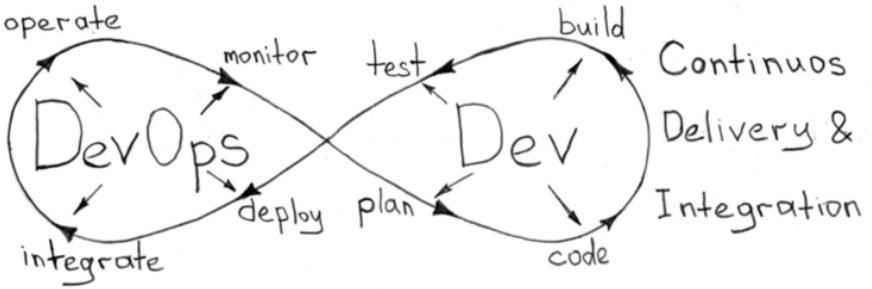
- Kube API endpoint
- an "secret" object that varies for different contexts (local/showroom/staging/production)
- the names of the systems to display and the tags of the Docker images for these systems (obtained at the previous step)

As an example of a meta-project that encompasses all systems and services (in other words, a project that describes how the ecosystem is arranged and how updates are delivered to it), I prefer to use Ansible `playbooks` with this module for integration with Kube API. However, sophisticated automators can refer to other options, and I'll dwell on my own choices later. However, you have to think of a centralized/unified way of managing the architecture. Having one will let you conveniently and uniformly manage all services/systems and neutralize any complications that the upcoming jungle of technologies and systems performing similar functions may throw at you.

Typically, an installation of the environment is required in:

- "ShowRoom"—for some manual checks or debugging of the system
- "Staging"—for near-live environments and integrations with external systems (usually located in the DMZ as opposed to `ShowRoom`)
- "Production"—the actual environment for the end user

Continuity in integration and delivery



If you have a unified way of testing Docker images—or "black boxes"—you can assume that the results of such tests would allow you to seamlessly (and with clear conscience) integrate `feature-branch` in the `upstream` or `master` branches of your git repository.

Perhaps, the only deal breaker here is the sequence of integration and delivery. When there is no releases, how do you prevent a "race condition" on one system with a set of parallel `feature-branches` ?

Therefore, this process should be started only when there's no competition, otherwise the "race condition" will keep haunting your mind:

1. Try to update the `feature-branch` to `upstream` (`git rebase/merge`)
2. Build images from `Dockerfiles`
3. Test all the built images
4. Start and wait until the systems with the images from step 2 are delivered
5. If the previous step failed, roll back the eco-system to the previous state
6. Merge `feature-branch` in `upstream` and send it to the repository

Any failure at any step should terminate the delivery process and return the task to the developer to fix the error, whether it's a failed test or a merge conflict.

You can use this process to work with more than one repository. Just do each step for all repositories at once (step 1 for repos A and B, step 2 for repos A

and B, and so on), instead of doing the whole process repeatedly for each individual repository (steps 1–6 for repo A, steps 1–6 for repo B, and so on).

In addition, Kubernetes allows you to roll out updates in parts for carrying out various AB tests and risk analysis. Kubernetes does it internally by separating services (access points) and applications. You can always balance the new and the old versions of a component in a desired proportion to facilitate problem analysis and make way for a potential rollback.

Rollback systems

One of the mandatory requirements to an architectural framework is an ability to reverse any deployment. This, in turn, entails a number of explicit and implicit nuances. Here are some of the most important of them:

- A service should be able to set up its environment as well as rollback changes. For example, database migration, RabbitMQ schema and so on.
- If it's not possible to rollback the environment, the service should be polymorphic and support both the old and the new versions of the code. For example: database migrations shouldn't disrupt the old versions of the service (usually, 2 or 3 past versions)
- Backwards compatibility of any service update. Usually, this is API compatibility, message formats and so on.

It is fairly simple to rollback states in a Kubernetes cluster (run `kubectl rollout undo deployment/some-deployment` and Kubernetes will restore the previous "snapshot"), but for this to work, your meta-project should contain information about this snapshot. More complex delivery rollback algorithms are highly discouraged, although they are sometimes necessary.

Here's what can trigger the rollback mechanism:

- High percentage of application errors after a release
- Signals from key monitoring points
- Failed `smoke` tests
- Manual mode—human factor

Ensuring information security and audit

There is no single workflow that can magically "build" bulletproof security and protect your ecosystem from both external and internal threats, so you need to make sure that your architectural framework is executed with an eye on the standards and security policies of the company at each level and in all subsystems.

I will address all three levels of the proposed solution later, in the section about monitoring and alerting, which themselves also happen to be critical to system integrity.

Kubernetes has a set of good built-in mechanisms for access control, network policies, audit of events, and other powerful tools related to information security, which can be used to build an excellent perimeter of protection that can resist and prevent attacks and data leaks.

From development workflows to architecture

An attempt to build a tight integration between the development workflows and the ecosystem should be taken seriously. Adding a requirement for such integration to the traditional set of requirements to an architecture (flexibility, scalability, availability, reliability, protection against threats, and so on), can greatly increase the value of your architectural framework. It is so crucial an aspect that it has resulted in the emergence of a concept called "DevOps" (Development Operation), which is a logical step towards the total automation and optimization of the infrastructure. However, granted a well-designed architecture and reliable subsystems, DevOps tasks can be minimized.

Micro-service architecture

There is no need to go into details of the benefits of a service oriented architecture—SOA, including why services should be "micro". I will only say that if you have decided to use Docker and Kubernetes, then you most likely understand (and accept) that it's difficult and even ideologically wrong to have a monolithic architecture. Designed to run a single process and work with persistence, Docker forces us to think within the DDD framework (Domain-Driven Development). In Docker, packed code is treated as a black box with some exposed ports.

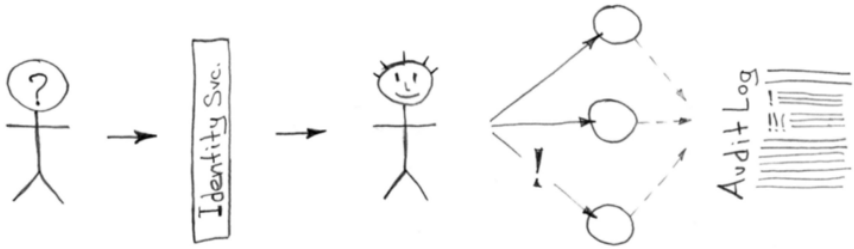
Critical components and solutions of the ecosystem

From my experience of designing systems with an increased availability and reliability, there are several components that are crucial to the operation of micro-services. I'll list and talk about each of these components later, and

even though I'll be referring to them in the context of a Kubernetes environment, you can refer to my list as a checklist for any other platform.

If you (like me) will have come to the conclusion that it'd be great to manage each of these components as a regular Kubernetes service, then I'd recommend you to run them in a separate cluster other than "production". For example, a "staging" cluster, because it can save your life when the production environment is unstable and you desperately need a source of its image, code, or monitoring tools. That solves the chicken and the egg problem, so to speak.

Identity service

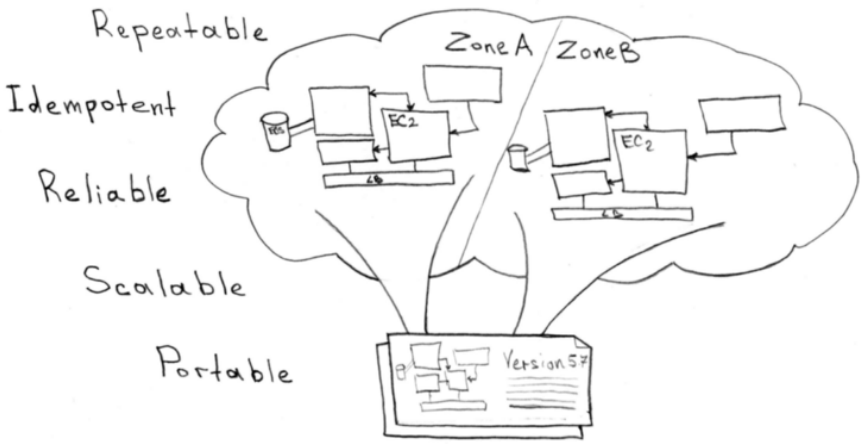


As usual, it all starts with access—to servers, virtual machines, applications, office mail, and so on. If you are or want to be a client of one of the major enterprise platforms (IBM, Google, Microsoft), the access issue will be handled by one of the vendor's services. However if you want to have your own solution, managed only by you and within your budget?

This list should help you decide on the appropriate solution and estimate the effort required to set up and maintain it. Of course, your choice must be

consistent with the company's security policy and approved by the information security department.

Automated service provisioning

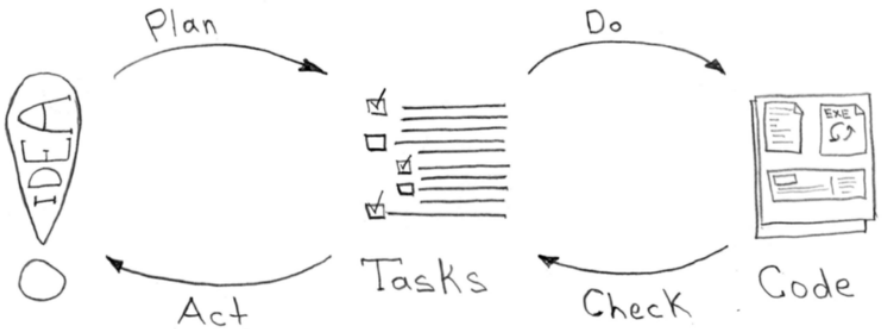


Although Kubernetes requires having only a handful components on physical machines/cloud VMs (docker, kubelet, kube proxy, etcd cluster), you still need to automate the addition of new machines and cluster management. Here is a few simple ways to do it:

- KOPS—this tool allows you to install a cluster on one of the two cloud providers—AWS or GCE
- Teraform—this lets you manage the infrastructure for any environment, and follows the ideology of IAC (Infrastructure as Code)
- Ansible—versatile tool for automation of any kind

Personally, I prefer the third option (with a little Kubernetes integration module), since it allows me to work with both servers and k8s objects and carry out any kind of automation. However, nothing stops you from using Teraform and its Kubernetes module. KOPS doesn't work well with the "bare metal" but it's still a great tool to use with AWS/GCE, too!

Git repository and a task tracker



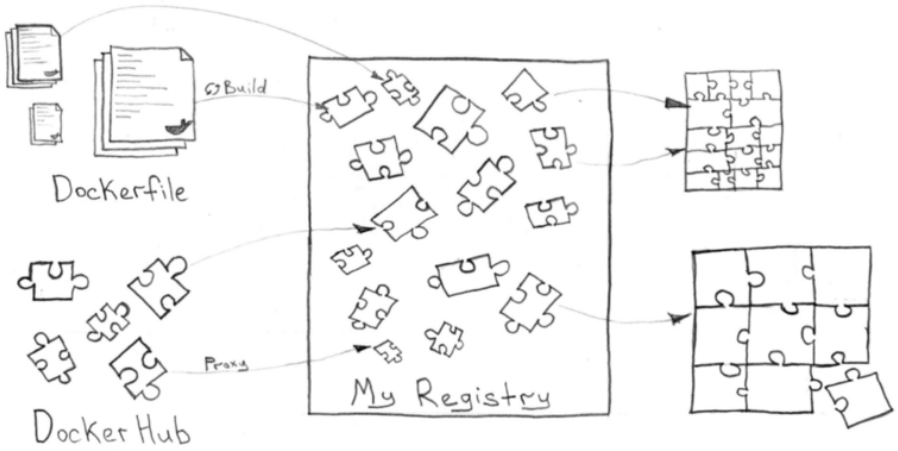
Needless to say that to provide for full-fledged work of developers and other related roles, you need to have a place for teamwork and code storage discussions. I would be hard pressed to determine which service is the best for this, but my personal gold standard for task tracking is redmine (free of charge) or Jira (paid), and for code repository—the "old school" [gerrit](https://www.gerritcodereview.com/) (free) or bitbucket (paid).

It is worth paying attention to the two most consistent (although commercial) stacks for collaborative work in a corporate environment: Atlassian and Jetbrains. You can use either of them a standalone solution or combine various components of both.

To make the most out of a combination of a tracker and repository, think about their integration strategy. For example, a couple of tips to ensure the integrity of the code and related tasks (of course you can choose your own approach):

- The ability to “push” into a remote repository should only be enabled if a branch that one is trying to push to has the corresponding task number (`TASK-1` / `feature-34`)
- Any branch should be available to merge only after a certain number of positive code review iterations
- Any branch should be blocked and disabled for future updates if the corresponding task is not “In Progress” or a similar status
- Any steps intended for automation should not be directly available to developers
- Only authorized developers should be able to directly modify the `master` branch - everything else controlled by an automation robot
- A branch should not be available for merging if the corresponding task is in any status other than “For Delivery” or similar

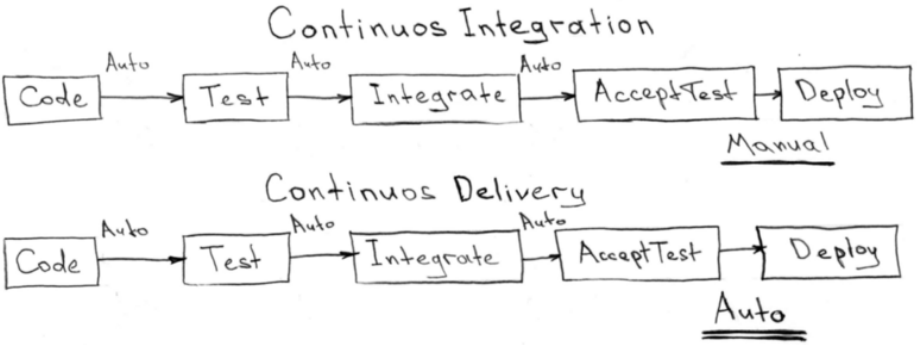
Docker Registry



Special attention should be given to a `Docker` image management system, as it is critically important for storing and delivering services. In addition, this system should support access for users and groups of users, be able to delete old and unnecessary images, provide a GUI and a RESTful API.

You can use a cloud solution (for example, hub.docker.com) or a privately hosted service, which can even be installed within your very Kubernetes cluster. Vmware Harbor, positioned as a corporate solution for Docker Registry, is a good example of the latter. Worst case, you can even use the good old Docker Registry if you just want to store images and have no need in a complex system.

CI/CD and services delivery system



None of the components we discussed earlier (git repo, task tracker, meta project with Ansible Playbooks, external dependencies) can function apart from one another as if suspended in a vacuum. What connects them is the continuous integration and delivery service.

CI—Continuous Integration CD—Continuous Delivery

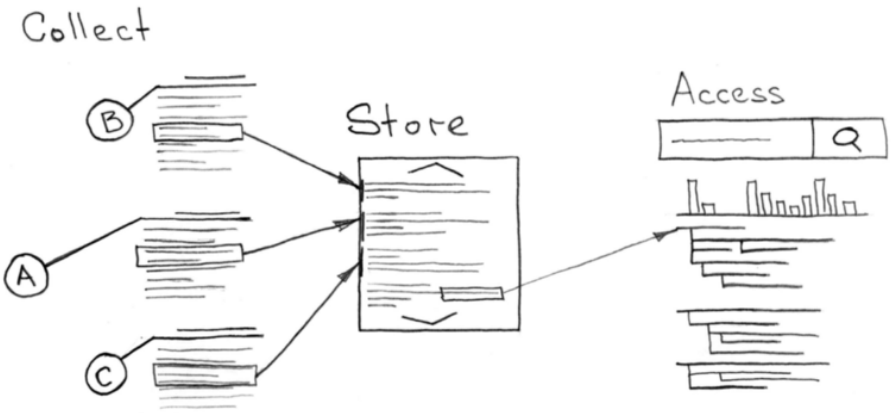
The service should be fairly simple and deprived of any logic related to the systems delivery or configuration. All that a CI/CD service should do is to react to events from the outside world (changes inthe git repository, moving tasks around the task tracker) and launch the actions described in the meta project. In addition, the servicecis a point of control of all repositories and a tool for managing them (branch merging, updates from `upstream/master`).

I've historically used a fairly powerful yet very simple tool from Jetbrains—TeamCity, but I don't see any problem if you decide to try something else, for example, the free Jenkins.

In the scheme that we described above, the integration service is essentially responsible for launching the four main processes and an auxiliary one, which are as follows:

- Automatic service testing—typically, for a single repository, when a branch state has changed or when the status has been changed to “Awaiting Autotests” (or similar)
- Service delivery—usually, from a meta-project and for a number of services (a number of repositories, respectively), when the status has changed to “Awaiting Showroom” or “Awaiting Delivery” for for QA and production environment deployment, respectively
- Rollback—as a rule, from a meta project and for a particular part of a single service or an entire service, triggered by an external event or in case of an unsuccessful delivery
- Service removal—this is required to completely remove the entire ecosystem from a single test environment (`showroom`), when the `In QA` status has expired or the environment is no longer needed
- Image builder (the auxiliary process)—can be integrated in the service delivery process or used independently to compile `Docker` images and send them to Docker Registry. Often handles widely used images (DB, general services or services that do not require frequent changes)

Log collection and analysis system



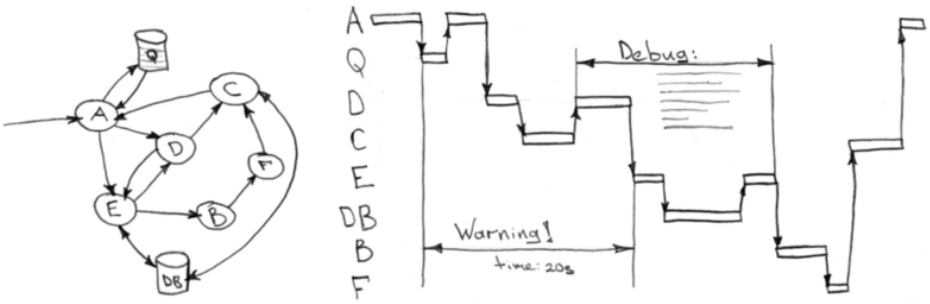
The only way for any Docker container to make its logs accessible is to write them to STDOUT or STDERR of the root process running in the container. Service developer doesn't really care what happens next with the logs data, the main thing is that they should be available when necessary and preferably contain records to a certain point in the past. All responsibility for fulfilling these expectations lies with Kubernetes and the engineers who support the ecosystem.

In the official documentation, you can find a description of the basic (and a good one) strategy for working with logs, which will help you choose a service for aggregation and storage of huge text data.

Among the recommended services for a logging system, the same documentation mentions fluentd for collecting data (when launched as an agent on each node of the cluster), and Elasticsearch for storing and indexing data. Even if the efficiency of either You may disagree with the efficiency of this solution, but it's reliable and easy to use so I think it's at least a good start.

Elasticsearch is a resource-intensive solution but it scales well and has ready-made Docker images to run both an individual node and a cluster of a required size.

Tracing system

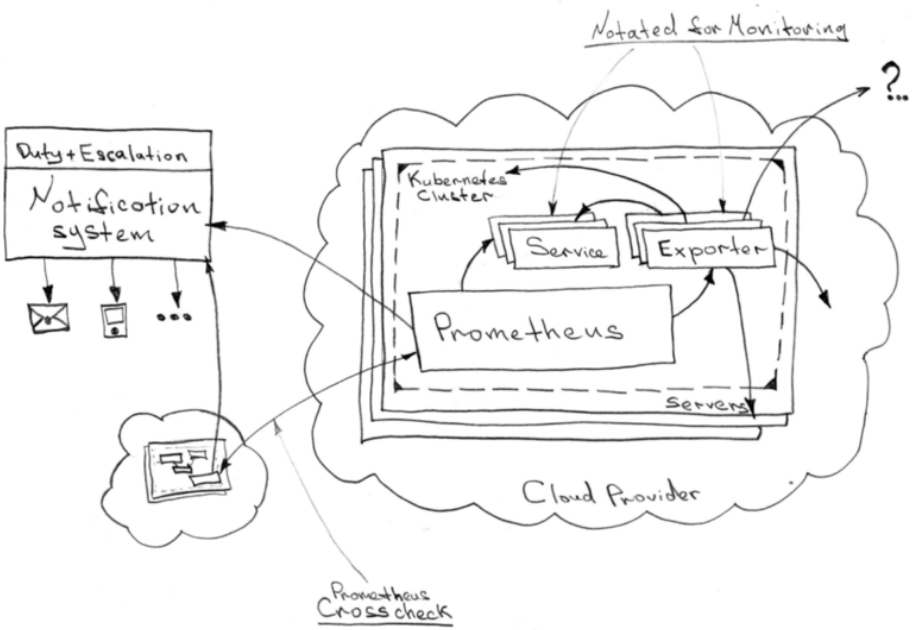


As perfect as your code can be, failures do happen and then you want to study them with a fine-tooth comb on production and try to understand “what the hell went wrong if everything worked fine on my local machine?”. Slow database queries, improper caching, slow disks or connectivity with an external resource, transactions in the ecosystem, bottlenecks and under-scaled computing services are some of the reasons why you will be forced to track and estimate the time spent executing your code under a real load.

Opentracing and Zipkin cope with this task for most modern programming languages and without adding any extra burden after instrumenting the code. Of course all the collected data should be stored in a suitable place, which is used as one of the components.

The complexities that arise when instrumenting the code and forwarding "Trace Id" through all the services, message queues, databases, and so on are solved by the above-mentioned development standards and service templates. The latter also take care of uniformity of the approach.

Monitoring and alerting



Prometheus has become the de facto standard in modern systems and, more importantly, it is supported in Kubernetes almost out of the box. You can refer to the official Kubernetes documentation to find out more about monitoring and alerting.

Monitoring is one of the few auxiliary systems that must be installed inside a cluster. And the cluster is an entity that is subject to monitoring. But monitoring of a monitoring system (pardon the tautology) can only be performed from the outside (for example, from the same “staging” environment). In this case, cross-checking comes in handy as a convenient solution for any distributed environment, which wouldn’t complicate the architecture of your highly unified ecosystem.

The whole range of monitoring is divided into three completely logically isolated levels. Here is what I think are the most important examples of tracking points at each level:

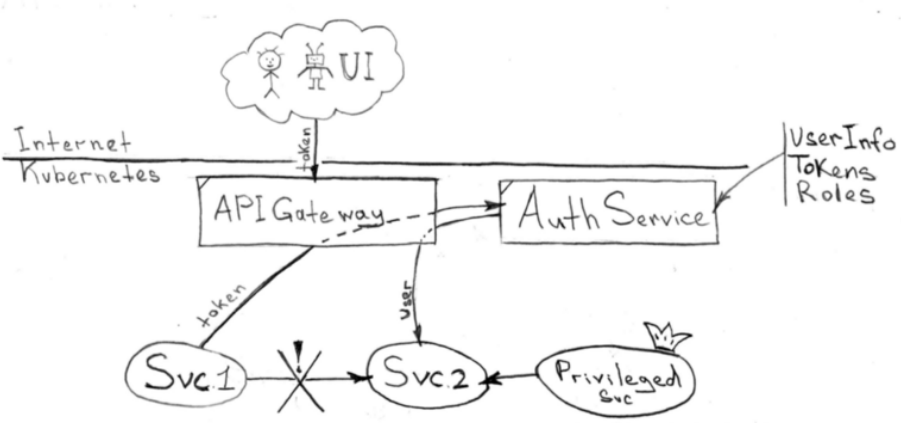
- Physical level:—Network resources and their availability—Disks (i/o, available space)—Basic resources of individual nodes (CPU, RAM, LA)
- Cluster level:—Availability of the main cluster systems on each node (kubelet, kubeAPI, DNS, etcd, and so on)—The number of free resources and their uniform distribution—Monitoring of permitted vs. actual resource consumption by services—Reloading of pods
- Service level:—Any kind of application monitoring—from database contents to a frequency of API calls—Number of HTTP errors on the API gateway—Size of the queues and the utilization of the workers—Multiple metrics for the database (replication lag, time and number of transactions, slow requests and more)—Error analysis for non-HTTP processes—Monitoring of requests sent to the log system (you can transform any requests into metrics)

As for the alert notifications at each level, I’d like to recommend using one of the countless external services that can send notifications to email, SMS or

make calls to a mobile number. I'll also mention another system—OpsGenie—which has a close integration with the Prometheus's alertmanager.

OpsGenie is a flexible tool for alerting which helps to deal with escalations, round-the-clock duty, notification channel selection and much more. It's also easy to distribute alerts among teams. For example, different levels of monitoring should send notifications to different teams/departments: physical—Infra + Devops, cluster—Devops, applications—each to a relevant team.

API gateway and Single Sign-on



To handle tasks such as authorization, authentication, user registration (external users-clients of the company) and other kinds of access control, you will need a highly reliable service that can maintain a flexible integration with your API gateway. No harm using the same solution as for the "Identity

service", however you may want to separate the two resources to achieve a different level of availability and reliability.

The inter-service integration should not be complicated and your services should not worry about authorization and authentication of users and each other. Instead, the architecture and the ecosystem should have a proxy service that handles all communications and HTTP traffic.

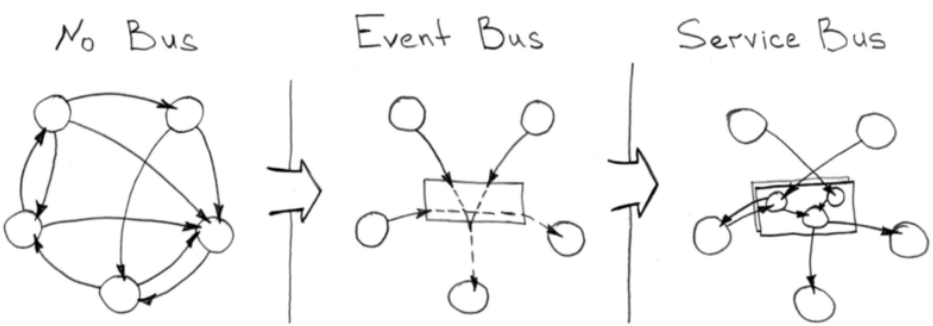
Let's consider the most suitable way of integration with the API gateway, hence with your entire ecosystem—tokens. This method is good for all three access scenarios: from the UI, from service to service, and from an external system. Then the task of receiving a token (based on the login and password) lies with the UI itself or with the service developer. It also makes sense to distinguish between the lifetime of tokens used in the UI (shorter TTL) and in other cases (longer and custom TTL).

Here are some of the problems that the API gateway resolves:

- Access to the ecosystem's services from outside and within (services do not communicate directly with each other)
- Integration with a Single Sign-on service:—Transformation of tokens and appending HTTPS requests with headers that containing user identification data (ID, roles, other details) for the requested service—Enabling/disabling access control to the requested service based on the roles received from the Single Sign-on service
- Single point of monitoring for HTTP traffic

- Combining API documentation from different services (for example, combining Swagger's json/yml files
- Ability to manage routing for the entire ecosystem based on the domains and requested URIs
- Single access point for external traffic, and integration with the access provider

Event bus and Enterprise Integration/Service bus



If your ecosystem contains hundreds of services that work in one macro domain, you will have to deal with thousands of possible ways in which the services can communicate. To streamline data flows, you should think of the ability to distribute messages over a large number of recipients upon the occurrence of certain events, regardless of the contexts of the events. In other words, you need an event bus to publish events based on a standard protocol and to subscribe to them.

As an event bus, you can use any system that can operate a so-called broker: RabbitMQ, Kafka, ActiveMQ, and others. In general, high availability and consistency of data are critical for the micro services, but you'll still have to sacrifice something to achieve a proper distribution and clustering of the bus, due to the CAP theorem.

Naturally, the event bus should be able to solve all sorts of problems of inter-service communication, but as the number of services grows from hundreds to thousands to tens of thousands, even the best event bus-based architecture will fail, and you will need to look for another solution. A good example would be an integration bus approach, which can extend the capabilities of the "Dumb pipe—Smart consumer" tactics described above.

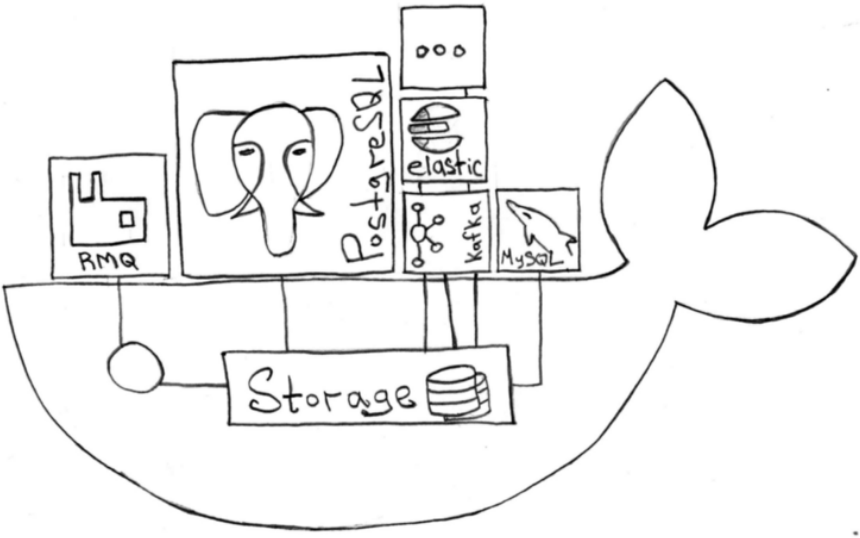
There are dozens of reasons for using the "Enterprise Integration/Service Bus" approach, which aims to reduce the complexities of a service-oriented architecture. Here's just a few of these reasons:

- Aggregation of multiple messages
- Splitting of one event into several events
- Synchronous/transactional analysis of the system response to an event
- Coordination of interfaces, which is especially important for integration with external systems
- Advanced logic of event routing
- Multiple integration with the same services (from the outside and within)

- Non-scalable centralization of the data bus

As an open-source software for an enterprise integration bus, you may want to consider Apache ServiceMix, which includes several components essential for design and development of this kind of SOA.

Databases and other stateful services



As well as Kubernetes, Docker has changed the rules of the game once and for all for services that require data persistence and work closely with the disk. Some say that the services should “live” the old way on physical servers or virtual machines. I respect this opinion and I won’t go into arguments about its pros and cons, but I’m fairly certain that such statements exist only because

of the temporary lack of knowledge, solutions, and experience in managing stateful services in a Docker environment.

I should also mention that a database often occupies the central place in the storage world, and therefore the solution you select should be fully prepared to work in a Kubernetes environment.

Based on my experience and the market situation, I can distinguish the following groups of stateful services along with examples of the most suitable Docker-oriented solutions for each of them:

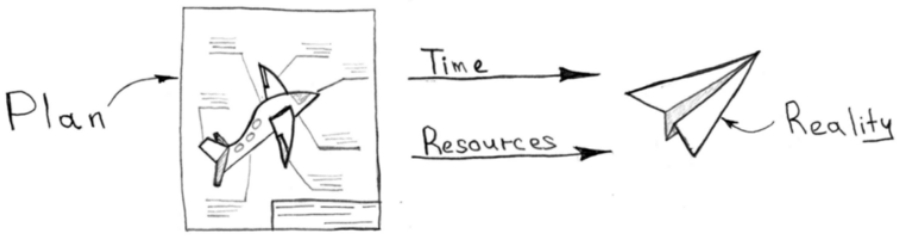
- Database management systems—PostDock is a simple and reliable solution for PostgreSQL inside any Docker environment
- Queue/message brokers—RabbitMQ is a classic software for building a message queue system and routing the messages. The `cluster_formation` parameter in the RabbitMQ's configuration is indispensable for a cluster setup
- Caching services—redis considered one of the most reliable and flexible data caching solutions
- Full text search—the Elasticsearch stack which I have already mentioned above, originally used for full-text search, but just as good at storing logs and for any kind of work with large amounts of text data
- File storage services—a generalizing group of services for any type of file storage and delivery (ftp, sftp, and so on)

Dependency mirrors

If you haven't yet encountered a situation where the packages or dependencies you need have been removed or made temporarily unavailable from a public server, don't think that this will never happen. To avoid any unwanted unavailability and provide security to the internal systems, make sure that neither building nor delivery of your services require an Internet connection. Configure mirroring and copying of all dependencies to the internal network: Docker images, rpm packages, source repositories, python/go/js/php modules.

Each of these and any other types of dependencies have their own solutions. The most common can be googled by the query "private dependency mirror for …"

From architecture to real life



Like it or not, sooner or later your entire architecture will be doomed to failure. It always happens: technologies become obsolete fast (1–5 years), methods and approaches—a bit slower (5–10 years), design principles and fundamentals—occasionally (10–20 years), yet inevitably.

Mindful of the obsolescence of technology, always try to keep your ecosystem at the peak of tech innovations, plan and roll out new services to meet the needs of developers, business and end users, promote new utilities to your stakeholders, deliver knowledge to move your team and the company forward.

Stay on top of the game by integrating into the professional community, reading relevant literature and socializing with colleagues. Be aware of your opportunities as well as the correct use of new trends in your project. Experiment and apply scientific methods to analyze the results of your research, or rely on the conclusions of other people that you trust and respect.

It is difficult to prepare yourself for fundamental changes, yet possible if you are an expert in your field. All of us will only witness a few major technological changes throughout our life, but it’s not the amount of

knowledge in our heads that makes us professionals and brings us to the top, it's the openness of to our ideas and the ability to accept metamorphosis.

Going back to the question from the title, "is it possible to build a better architecture?" No, not "once and for all", but be sure to strive for it and at some point, "for a short time", you will certainly succeed!

PS:

The original article was written in Russian, so thanks to my colleague in Lazada Sergey Rodin for awesome help to translate it!