Microservices and Continuous Delivery: a Pragmatic Perspective of three Common Dilemmas

Laurence Withers and Quirino Zagarese

Yoti Ltd. London, United Kingdom

Yoti is a digital identity provider offering a free-to-use mobile app which allows individuals to register existing identity documents, and a system with which organisations and individuals can request some proof of identity.

Early in our journey, we discovered that it was becoming difficult to build new features in our monolithic system as build and test times were excessive and there was significant coupling between components. It was clear that a modular solution was necessary.

We decided to adopt a microservice[3] architecture as, in addition to allowing shorter development cycles for the individual components, it allows for flexible horizontal scaling and Yoti aims to become a worldwide identity solution.

Our functionality is now split up into small, logical blocks, each of which is developed and deployed independently. Kubernetes[4] is used to orchestrate the deployment and execution of these services.

Implementing reliable Continuous Delivery (CD)[1] for such a system is not trivial. When each code change translates into a new release, reliability is achieved not only by enforcing good coding practices: it is matter of testing, deployment and communication strategies. Complexity is further compounded in a microservice architecture as a single deliverable change often impacts multiple services.

During our journey, we have highlighted three dilemmas that most organisations would encounter if they decided to migrate their systems towards a microservice architecture and CD.

Deploy one or deploy many. When a change to multiple microservices must be released, it is necessary to carefully control and synchronise the process to avoid incorrectly processing requests. We identify two categories of approach:

- Ensure each individual microservice to be deployed is compatible with the current service configuration, even as the configuration changes during a deployment (deploy one).
- Design the pipeline so that a set of services can be deployed together without service disruption (deploy many).

The main consideration of the first approach is that it forces committers to have a clear picture of the whole system. A simple example consists of a change where *service* A calls a new endpoint from *service* B: it is clear that B needs to be released before A, however this implies that whoever is working on A is coordinating with whoever is working on (and deploying) B.

There are variations on this approach. By ensuring that services are deployed in a consistent order, it is simpler to reason about the possible service configurations that may be encountered. In our experience this works well when the system does not exceed 15–20 services, but becomes impractical to maintain this level of coordination as the set of services grows or if the set of committers does not have a strong organisational relationship.

Another variation is to accept that the order of deployments is ill-defined, and to write services in such a manner that they produce the correct result in all reasonable service configurations. Examples of techniques used here are having transition code that produces requests/responses that will be interpreted correctly by both new and old services, or using version numbers on persistent objects, allowing services to reject operations on objects with larger version numbers (assuming some sort of “retry later” operation).

The second approach, a deploy-many pipeline, is more complex from a technical perspective. When new code is to be released, the old code is left running in parallel for a time. The orchestration layer ensures both that it only starts sending requests to the new instances once everything is deployed as well as ensuring that a single request is routed exclusively within only the old or the new set of services. In practice, we have realised that this is very difficult with any orchestration system and requires significant support from the application layer. It also does nothing to address requests that are left in a queue or database for later processing.

With either strategy, the problem is reduced in scope by reducing the amount of change to be introduced in any given release. The less outstanding change, the easier it is for the developers to reason about the possible combinations that may be encountered. In practice this means shortening the feedback loop between developers writing new code and deploying it to production: one of the goals of CD.

To Mock or not to Mock. Mocking can be invaluable in some circumstances, yet can be expensive to maintain and not necessarily effective. Our experience shows that mocking is necessary for a CD pipeline, but that its use should be carefully considered. The work to build and maintain the mock may exceed the work needed for the component being mocked. For more complex components, it is all too easy to end up writing tests that only exercise the mock itself, without adding any actual value to the testing process.

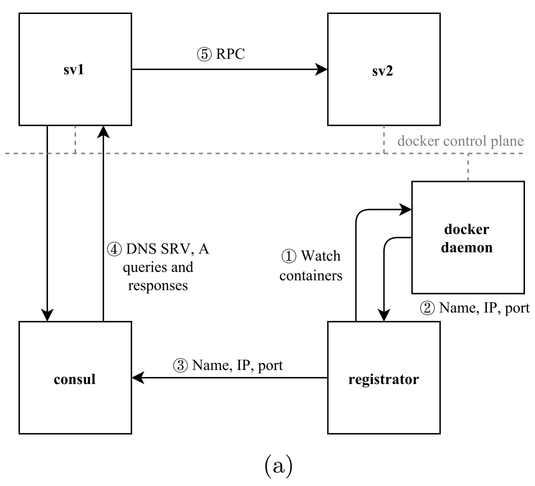
Mocking is necessary when an automated test covers a process which involves gathering (human) operator input, as they may not be available on a reasonable timescale, and are not guaranteed to return consistent results. Thus we mock interactions with our account onboarding system, which has human operators checking each new account is genuine.

Another relevant scenario concerns the dependency from external systems, where by “external” we mean any system that cannot be influenced by the team that is building the service under analysis. In this case, mocking can be a valuable solution, but only if the interface of such systems are stable. A good metric of stability can be the recent frequency of changes/releases. We usually mock all the calls to external APIs as we don’t want a downtime of such systems to have an impact on our CD pipeline.

To Tie or not to Tie to your Orchestration Layer. A microservice which has no knowledge of the orchestration layer on which it is to be run is a reusable, transportable component: it can be used for different purposes and in different environments.

One of several common tasks that requires interaction with the environment is service discovery: a microservice finding the address of some other service it needs to call. DNS provides a simple solution 1b, but does not provide port number information. This can be solved with the use of DNS SRV (service) records 1a, which also incorporate port number information, but this requires additional components and some supporting code in the in the calling microservices.

We have also developed a discovery method using native Kubernetes calls, where it is possible to “watch” for changes and efficiently maintain an address registry. While powerful, this mechanism ties the microservice implementation to Kubernetes and part of our testing pipeline runs outside of it, using only Docker[2], for much faster startup.



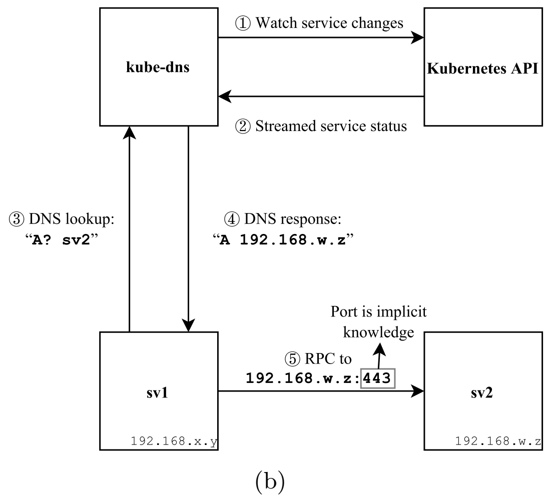


Figure 1: Discovery alternatives for using DNS SRV records (a) and using native DNS (b)

We believe that abstracting discovery is a very promising direction: a client side layer that supports both DNS SRV queries and the Watch API, so that services only need to know the name of the services they need to talk to. More importantly, such a solution allows for a flexible and reliable execution of a potentially heterogeneous CD pipeline.

References

[1] Jez Humble and David Farley. *Continuous delivery: reliable software releases through build, test, and deployment automation.* Pearson Education, Inc., 2010.

[2] Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014(239):2, 2014.

[3] Sam Newman. *Building microservices: designing fine-grained systems.* O'Reilly Media, Inc., 2015.

[4] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, Bordeaux, France, 2015.