6 QUESTIONS TO 6 EXPERTS ON IMMUTABLE INFRASTRUCTURE

Brought to you by HighOps – http://highops.com

HIGHOPS

OPERATIONS
IDEAS
DELIVERED

Immutable Infrastructure is not a new concept [see Further Links] and while there are many examples of successful implementations it would be a lie if we said that the recent hype around containers in general and Docker in particular hasn't made the concept more widespread.

As it often happens in these cases, we see a lot of different interpretations of its meaning, benefits, challenges, and adoption paths. We asked 6 experts who have been thinking, writing and implementing Immutable Infrastructures to share their experience by answering 6 questions on the topic:

1) What does Immutable Infrastructure mean to you?
2) What's your position on Immutable Infrastructure and why?
3) What are the main benefits you see/care about?
4) Biggest adoption challenges/things that are not there yet in your opinion?
5+6) Starting from scratch is (relatively) easy. What about those with existing systems? any hints on how others could get started moving towards an Immutable Infrastructure?

Marco Abis
November 3rd, 2014

Experts Profiles

Kief Morris

Kief is a software delivery consultant with ThoughtWorks in London, specializing in tools, practices, and processes for the Continuous Delivery of software.

http://kief.com/ - https://twitter.com/kief

Andrew Phillips

Andrew heads up product management at XebiaLabs, building tools to support DevOps and Continuous Delivery.

http://blog.xebialabs.com/author/aphillips/

Florian Motlik

Florian is CTO and Founder of Codeship where he makes sure the System is up and running as well as getting it into as many hands as possible

http://blog.codeship.io/author/florianmotlik - https://twitter.com/flomotlik

Julian Dunn

Julian is an engineering manager at Chef, where he helps to build products on top of the core Chef product portfolio.

http://www.juliandunn.net/ - https://twitter.com/julian_dunn

Matthew Skelton

Matthew is a Continuous Delivery specialist, DevOps enthusiast, and an Operability nut. He set up and co-runs both LondonCD and PipelineConf. He is co-founder and Principal Consultant at Skelton Thatcher Consulting.

http://blog.matthewskelton.net/ - https://twitter.com/matthewpskelton

Ben Butler-Cole

Ben likes to build systems rather than software. He has spent the last twelve years looking for ways to avoid unnecessary work. When all else fails he likes to write code in languages that haven't been designed to hurt him. He currently works for Neo Technology.

http://www.bridesmere.com/

Highlights

The full transcript of all answers follows, grouped by question but here are the highlights:

1) What's an Immutable Infrastructure?

A pattern or strategy for managing services in which infrastructure is divided into "data" and "everything else". "Everything else" components are replaced at every deployment, with changes made only by modifying a versioned definition, rather than being updated in-place.

2) What's your position on Immutable Infrastructure and why?

Definitely a good idea that all components of a running system should be in a known state. Worth exploring as long as it doesn't become yet another silver bullet. We still have a lot to learn about how to make it work well and really need to analyze the problem we are trying to solve, which is minimizing errors in deployment, configuration, or runtime.

On one hand it's a myth, because it ignores the actual operating conditions of systems, because servers and network devices are changing all the time (e.g. RAM) but seen as one point on a continuum or plane of configuration options, it can be useful because incremental configuration management tools can never control everything; in particular it's impossible to be certain that things are absent.

3) What are the main benefits you see/care about?

It's a way to simplify change management: servers never rot and you can think of an application as a single deployable artifact, you can reason about it at a higher level.
It also promotes better understanding across the whole delivery chain making it more resilient, flexible, portable and with better safeguards because it forces you to expect failures and have a reliable process to rectify them simply.

4) Biggest adoption challenges/things that are not there yet in your opinion?

We don't know the best way to do it yet; not only does it require a bigger shift in mindset than things like Infrastructure as Code, but the publicly available technology is still very new. Tooling needs to evolve considerably and a lot of the plumbing (e.g. orchestration and resource allocation primitives) isn't there yet.
It also requires a high degree of maturity from organisations, processes and operational knowledge, advanced networking and storage.

Last but not least, creating servers from scratch is generally slower than updating them in-place and you need to wean yourself off existing tools which have been created to patch the old way of doing things.

5+6) Starting from scratch is (relatively) easy. What about those with existing systems? any hints on how others could get started moving towards an Immutable Infrastructure?

Determine if it makes sense: only consider it if you have a lot of pain in areas that it could address and stop once you have addressed the problems you had with that service - don't "boil the ocean".
Identify specific elements of your infrastructure to trial it, and work on it incrementally and iteratively. A good place to start is often the build & test environments but be prepared to need a lot more plumbing than just the AMI factory or Packer or Docker.
Or try it out on a smaller, green-field system first in order to iron out problems and properly understand the implications. Since it currently works well if you have a 12-factor, stateless SaaS app you can start with that.

Full Answers

What does Immutable Infrastructure mean to you?

Kief: I see it as conceptually dividing your infrastructure into "data" and "everything else". Data is the stuff that's created and modified by the services you're providing. The elements of your infrastructure that aren't managed by the services can be treated as immutable: you build them, use them, but don't make changes to them. If you need to change an infrastructure element, you build a new one and replace the old one.

Andrew: To me, II describes a pattern or strategy for managing services in which changes (patches, application upgrades, configuration changes etc.) are made only by modifying a versioned service definition and updating running services instances to the new definition version. This as opposed to making changes to running instances directly.

Florian: Immutable infrastructure is comprised of immutable components that are replaced for every deployment, rather than being updated in-place. Those components are started from a common image that is built once per deployment and can be tested and validated. The common image can be built through automation, but doesn't have to be. Immutability is independent of any tool or workflow for building the images.
Its best use case is in a cloud or virtualized environment. While it's possible in non-virtualized environments, the benefit doesn't outweigh the effort.

Julian: Immutable infrastructure to me basically means, "build once, run once". After that, you never mutate the running system; changes require a rebuild. Although containerization (e.g. Docker) is an example of immutable infrastructure, it's not the only possibility. Companies like Netflix that deploy by creating fresh AMIs and blowing away machines running the old one can also be considered to be using "immutable infrastructure".

If you've ever been a Java application developer, the WAR or EAR that Maven generates for you can also be considered "immutable"; it's a runnable artifact. If you want to make changes to the application, you update the Java source code, recompile into classes, and get a new deployable artifact. That's the general idea.

Matthew: The drive behind Immutable Infrastructure is to configure and test infrastructure *before* it is deployed in place to a Production environment, rather than using in-place configuration. Think using Chef Zero or masterless Puppet to generate versioned templates or configuration sets which are then 'flowed down' the deployment pipeline. The goal is to increase test coverage up front and to try to avoid side-effects or accidental misconfigurations such as an in-place upgrade of Ruby or npm packages.

Ben: Server configuration is a one-off event when a server is first provisioned. After that the configuration is never updated -- the only thing that changes on the server is application data. When the configuration needs to change, the server is discarded and replaced with another freshly built one.

What's your position on immutable infrastructure and why?

Kief: Trial. We still have a lot to learn about how to make it work well, when to use it, when not to use. There is not a mature ecosystem of tools and practices.

Andrew: Definitely A Good Idea for all components of a running service that should be in a "known state" (i.e. not the data inside a transaction log, for example...but indeed the schema definition of the log file structure itself). Whether it makes sense in a given context depends highly on the particular "service definition" technology chosen: updating the running service to a new definition version must not be significantly more complex than making the equivalent change directly on the old instance. In order to catch problems early, the definition format should also be easy to create, modify, understand and verify.

Beyond what was said above, for a couple of key reasons (in no particular order):

* it promotes better understanding across the whole delivery chain: there is a much earlier opportunity for shared insight into what the running service will actually look like
* it is more resilient: regenerating an entire service landscape should be easy
* it is more flexible and portable: a service definition can be translated to other runtime platforms much more easily (of course, it's not trivial) than trying to clone running systems
* it allows for better safeguards: there is more opportunity to (automatically, if possible) verify that the proposed change to the service is safe

Florian: I wrote about it in this post http://blog.codeship.io/2014/07/22/immutable-infrastructure.html

Julian: The concept of immutable infrastructure is interesting, but it's not a panacea, which is unfortunately how many folks are approaching it today. They're starting with "immutable infrastructure" as being a desirable goal without really analyzing the problem they're trying to solve, which is minimizing errors in deployment, configuration, or the runtime. But if you have a lot of bugs in your application proper, then immutable infrastructure isn't going to help you; you're just isolating them.

Returning for a moment to my Java application analogy above: why did folks start creating WARs instead of littering class files, JARs and WEB-INFs all over the filesystem? Because it's

helpful to think of the application as a single, unchanging deployable package that you can move from Dev to QA to production without modification.

However, as a former developer and then operations person, I'd argue that most of the errors that occur in production are actually application (programming) errors. Although developers like to fixate on "configuration" or "environment" errors, if you measured where production faults occur, they are usually in the running software. An example of that is bad data in a database. Immutable infrastructure isn't going to save you from that, because some parts of your infrastructure, like your customer data, are not going to be immutable..

If you are truly 100% immutable, you're actually a dead company: nothing is changing, therefore, it means you don't have any customers and you can just go home.

Matthew: To some extent Immutable Infrastructure is a software development myth, ignoring the actual operating conditions of systems, because servers and network devices are changing all the time (logs, RAM allocation, etc.) - Mark Burgess of CFEngine has some wise words on this. I think if Immutable Infrastructure is seen as one point on a continuum or plane of configuration options, it can be useful. Certainly reducing the possibility for accidental misconfiguration of a complex system is a good thing.

I have seen too many organisations latch onto one tech fad or another, believing that 'Agile', 'Kanban', 'DevOps', or 'microservices' will fix all their delivery headaches. If people think that Immutable Infrastructure alone will solve their configuration problems, they are mistaken. If, however, they see aspects of Immutable Infrastructure as a useful part of managing infrastructure configuration (along with traceability, blame-free culture, etc.) then it could work well for their team.

Ben: Extremely positive. It's the only way that you can be absolutely sure what the configuration of your servers is. Incremental configuration management tools can never control everything -- in particular it's impossible to be certain that things are absent. With immutable servers the state is completely described by an initial, standard OS image and a set of scripts which are applied to the image.

What are the main benefits you see/care about?

Kief: Simplicity. Much of the cost and risk of managing infrastructure is due to the complexity of managing changes to it. Immutable infrastructure is a way to simplify change management.

Andrew: See previous answer :-)

Florian: :-) http://blog.codeship.io/2014/07/22/immutable-infrastructure.html

Julian: Immutable infrastructure is a powerful concept if you can think of your application as being one deployable artifact instead of the mess of all the tangled libraries, external dependencies, configuration, etc. that make up a piece of software. The fact that you can reason about an application at a higher level than just the deployable package containing the code (WAR/zip/MSI/whatever) is a useful concept.

Matthew: The main benefits of Immutable Infrastructure (done well) are probably increased transparency and ability to rectify faults simply: rather than layering additional changes onto an already-faulty system, with Immutable Infrastructure you can 'cut out' the faulty part and replace with a correct part. This does beg the question: "How much of my infrastructure should I rebuild if I find a fault?" which is clearly a matter of experience.

Ben: Servers never "rot".

You can replace or modify existing servers in a completely controlled and safe way. You can be absolutely certain that the replacement server is identical to the original, or that it differs only in exactly the ways you want it to.

You can test proposed configuration changes by standing up replica servers outside your production environment and be certain that the results of the test are accurate, because the replicas are identical.

Biggest adoption challenges/things that are not there yet in your opinion?

Kief: We just don't know the best way to do it yet. It's a huge shift in mindset for most people in IT Ops. The cloud-native, infrastructure as code approach to managing IT infrastructure is still not widely understood and used in the industry, and immutable infrastructure is a bigger leap yet from the "iron age" of infrastructure

Andrew: Adoption challenges:

* Technical maturity. A lot of the technology that is publicly available is still very new, with all kinds of kinks and unaddressed issues.
* Process maturity. Running large-scale services with this kind of architecture requires some process and operational knowledge that very few organizations have at this point.
* Skills and experience: It's very hard to find or hire people with both the technical skills and experience of the relevant tools and processes...so call HighOps ;-)

Missing pieces. Again, in no particular order:

* A workable service definition language. "Stack templates" like TOSCA/Cloud Formation/Heat tend to be pretty unwieldy, and "Container definitions" like Dockerfiles do not allow you to describe systems consisting of multiple components (although that is coming, of course). These definition templates also tend to focus highly on content ("what is in this thing") rather than policy ("how can this thing be run").
* Better support for verification of whether a new service definition is "good". This is basically saying that all the types of checks and processes (such as CI) we have devised for application code also apply to these definitions now, yet the tooling lags behind. FindBugs for Dockerfiles, anyone?
* Negotiation over hard requirements. In other words, what you can mainly do right now is say "I need all of this" and, if any one of these pieces is not available, fail. There is no notion of being able to *negotiate* with the underlying runtime platform and see how you could deal with less. Mesos is based precisely on this resource negotiation concept, but it's happening at a very low level (#CPUs vs. "I have a persistent store with these capabilities for you").

Florian: Tooling still needs to evolve and process needs to become more common for all different tools to support it and see it as default.

Julian: To me, the main gap is that a lot of the plumbing isn't there yet. It took Java developers years to figure out the benefits of having WARs/EARs, and for all the ancillary parts to be built like build tools, artifact repositories and development frameworks. Even today, many developers are still doing it wrong: e.g. deployment patterns where you have to crack open the WAR and monkey with text properties inside WEB-INF. At that point you don't really have immutability, just as if you fired up a Docker container and then messed around with it using "nsenter" or "docker exec".

I see a couple of other problems and misperceptions in the immutable infrastructure world:

- Folks that think every app can easily be made immutable and they can run large swaths of infrastructure in an immutable way. In practice, this is only true if you have a 12-factor SaaS app or microservice. What if you don't have an app that meets this profile?

- Orchestration and resource allocation primitives are not there yet (or at least there are many half-complete solutions). If your deployment model now amounts to "rip and replace", there is sophisticated cluster coordination & service discovery required to ensure that you can do this safely and that you don't lose customer data by doing this!

I don't think many folks realize how much work it is to get this right, nor do they realize that they're owning a lot more (mutating) infrastructure to get the immutable components right. Again, look at Netflix that has been doing this for a while, and how many parts they've had to build (and have subsequently open-sourced) to manage their immutable AMIs.

Matthew: The biggest adoption challenge I see (based on the organisations I have worked with) is that Immutable Infrastructure requires a high degree of maturity from an organisation, coupled with the need for advanced or expensive networking and storage (in order to store and move the VM templates, for example). Small and medium-sized companies for which Amazon AWS/EC2 is too complicated or costly, are only now - in the second half of 2014 - seeing alternative PaaS options (such as Windows Azure) with enough flexibility to meet their integration needs *and* offer the configurability to support Immutable Infrastructure.

Ben: Creating servers from scratch is generally slower than updating them in-place. You have to accept that deployments will take longer.

You need a clear strategy for separating persistent data (for example application data or logs) from the immutable server.

To do this in the simplest way, you need to wean yourself off existing tools which have been created to patch the old way of doing things (for example ssh-in-a-for-loop tools like Fabric and configuration management tools like Puppet).

I don't know how practical it is to take this approach straight onto bare metal, using something like PXE. It may only be practical on (public or private) IaaS.

Starting from scratch is (relatively) easy. What about those with existing systems? any hints on how others could get started moving towards an Immutable Infrastructure?

Kief: Identify specific elements of your infrastructure to trial it, and work on it incrementally and iteratively.

Andrew: 1. Determine if it makes sense in the context of a particular system. What is currently the biggest pain in managing that system? Only consider II if you have a lot of pain in areas that II could address.

2. Identify a suitable "service definition" language or tool. For existing services, especially those that are not simply a single web process, a provisioning tool may be a better choice than moving straight to containers.

3. Start applying II to the biggest "problem children" for your particular service. Figure out a way to create new service instances even if not all the elements of that services are already II - look at "old-fashioned" things like VM snapshots or service virtualization tools for that purpose. If some heavyweight elements of your service are read-only, you can share instances of those between environments.

4. Stop once you have addressed the problems you had with this service - don't "boil the II ocean" if you don't have to. Getting the most out of II often means changing your service delivery model and your application architecture and development philosophy as well, which is not easily done with existing code. Of course, if you e.g. decide to break a big monolithic app into small chunks that are better suited to II...consider II from the get-go for those chunks!

Florian: Start pulling services out of your existing infrastructure and make those services immutable. One service after the other so in the end you have a service oriented architecture and immutable infrastructure.

Julian: As I said before, immutable infrastructure currently works well if you have a 12-factor, stateless SaaS app (or a microservice). So I'd start with that. Again, the objective isn't "immutable infrastructure" per se; the goal has got to be something larger. For example, if you're converting from a monolithic to a microservice-oriented architecture for other reasons (example: it's easier to deploy & manage individual services than one gigantic application) that might be a good time to investigate immutable infrastructure.

I wouldn't attempt to jam something that doesn't "quack like a duck" into immutable infrastructure. For example, here at Chef I've heard from a number of customers that they've tried to run the Chef Server inside a container. Well, why? What benefit does that give you? Also, the Chef Server is made up of a number of different services with process supervisors, and persists data to the disk – hardly a good use-case for containerization.

Finally, be prepared to need a lot more plumbing than just "the AMI factory" or "Packer" or "Docker". As with most things, prior planning prevents poor performance, so understanding the ecosystem of things you'll need before diving in is crucial.

Matthew: A good place to start with Immutable Infrastructure (as with any new configuration automation initiative) is with the build & test environments, as these can usually be changed rapidly and without too much process theatre. At one place I worked recently (a large online travel retailer in the UK) we moved all the 100+ build agent machines from hand-crafted 'snowflakes' to VMs that were built automatically from versioned templates. We also automatically destroyed and rebuilt the build agent VMs every few weeks to flush out any

'customisation' done by development teams, reducing surprises further down the pipeline. The techniques paved the way for the same technique to be used in the downstream environments.

Ben: This isn't something that I've tried. I can't see why it wouldn't be possible to do this one server-type at a time. But I would strongly advise the team tackling it to try out immutable servers on a smaller, green-field system first in order to iron out problems and properly understand the implications.

Further Links

- Immutable Server http://martinfowler.com/bliki/ImmutableServer.html
- Rethinking building on the cloud:
    - Layering the Cloud
    - Environments on the Cloud
    - Principles to maximize environments
    - Immutable Servers
- Why you should build an Immutable Infrastructure
- Virtual Panel on Immutable Infrastructure
- Trash Your Servers and Burn Your Code: Immutable Infrastructure and Disposable Components
- Immutable Infrastructure: Practical or Not?
- Food Fight Podcast: Immutable Infrastructure