ADVANCED DEVOPS TOOLCHAIN

eMag Issue 28 - May 2015

InfoQ

ARTICLE
Docker: Present and Future

VIRTUAL PANEL
Immutable Infrastructure

ARTICLE
Service Discovery

Virtual Panel on Immutable Infrastructure

InfoQ reached out to experienced ops engineers to ask about their definition and borders of immutable infrastructure as well as its benefits and drawbacks, in particular when compared to "desired state" configuration management solutions. Is it a step forward or backwards in effective infrastructure management.

Docker: Present and Future

Chris Swan presents an overview of the Docker journey so far and where it is headed along with its growing ecosystem of tools for orchestration, composition and scaling. This article provides both a business and a technical point of view on Docker and separates the hype from the reality.

How I Built a Self-Service Automation Platform

The current buzz in the industry is how DevOps is changing the world and how the traditional role of operations is changing. Part of these discussions lead to building self-service automation tools which enable developers to actually deploy and support their own code. Brian Carpio describes his journey towards an infrastructure that development teams could deploy with the click of a mouse or an API call.

Managing Build Jobs for Continuous Delivery

The number of jobs in a continuous integration tool can range from a few to several thousand, all performing various functions. There is an approach to manage these jobs in a more efficient manner.

Service Discovery

Building microservices using immutable infrastructure means that your services need to be reconfigured with the location of the other services they need to connect to. This process is called Service Discovery. This article compares the approaches by several key contenders, from Zookeeper to Consul, Etcd, Eureka and "rolling your own".

FOLLOW US

facebook.com /InfoQ **@InfoQ** **google.com /+InfoQ** **linkedin.com company/infoq**

CONTACT US

GENERAL FEEDBACK feedback@infoq.com

ADVERTISING sales@infoq.com

EDITORIAL editors@infoq.com

Any Metric, Any Scale, Any Resolution

Modern monitoring platform using streaming analytics

Build Your Apps, Not Your Monitoring - With Real-time Metrics and Events at Web Scale

Focus on interpreting the data and finding application patterns or anomalies as they occur. SignalFx takes care of the infrastructure and plumbing to process, store, and run analytics against that data in real time for you at frequencies up to every second and at any scale.

Sleep Better with Meaningful Service Level Alerting

Build alerts on better signals, such as fleet-wide behavior in percentiles or service-level error rates, rather than individual systems or static thresholds. Define a rule once without having to change or refresh it as services scale, and get notifications within seconds of the raw data.

Be The Hero - Deliver Great User Experience and Catch Problems in Real Time

Apply custom analytics pipelines to your data as it comes in, choosing from a comprehensive library of functions like sums, percentiles, moving averages, and standard deviations. Provide multiple dimensions for any metric to enable easy filtering and aggregation. Append additional metadata on data already in SignalFx to enable more efficient searching.

"SignalFx is so powerful because it lets us look at aggregations like the 95th percentile and stream them alongside the same analytics from a day ago, week ago, or any time that makes sense."

- Tapjoy

"SignalFx has scaled to support our massive volume of high-frequency metrics, and has provided us with powerful tools to explore and find critical insights in our data."

- yelp

www.signalfx.com | @signalfx

signal fx

MANUEL PAIS is InfoQ's DevOps Lead Editor and an enthusiast of Continuous Delivery and Agile practices. Manuel Pais tweets @manupaisable

A LETTER FROM THE EDITOR

The edition follows in the footsteps of the previous "DevOps Toolchain for Beginners" eMag. Once an organization has acquired the essential skill set to provision and manage infrastructure, set up delivery pipelines and analyze logs and metrics, it will then gradually face more complex decisions. Decisions that depend much more on the type of systems architecture and organizational structure than on individual tool choices.

In this eMag we provide both implementation examples and comparisons of different possible approaches on a range of topics from immutable infrastructure to self-service ops platforms and service discovery. In addition, we talk about the Docker ecosystem and the different aspects to consider when moving to this increasingly popular system for shipping and running applications.

In the immutable infrastructure virtual panel we learn from both advocates and skeptics on the pros and cons of this "disposable" approach to infrastructure changes. Instead of trying to converge machines to a desired state, outdated machine are simply destroyed and replaced by a brand new machine with the new changes applied.

Chris Swan gives us a rundown of the increasingly popular Docker tool, outlining the strengths and weaknesses of containers when compared to the more established virtual machines. He also highlights the expanding ecosystem around Docker, including tools and libraries for networking, composition and orchestration of containers.

Many companies haven taken the route of automating delivery of traditional ops services such as provisioning infrastructure, deploying to production environments, adding metrics to monitor or creating test environments as similar as possible to production. This decision often comes from reaching a bottleneck in operations workload as development teams are empowered to deploy and own their applications through their entire lifecycle in a DevOps mindset. Brian Carpio describes his technical journey into creating such a self-service platform.

When teams reap the benefits of continuous delivery, they want more of it. On every project. Every component. The number of jobs in the continuous integration/delivery server explodes and a maintenance nightmare ensues if care is not taken. Automated job creation and update becomes crucial to keep pace and avoid development teams getting bogged down by build and deployment issues. Martin Peston provides an overview and practical examples on this topic.

When adopting immutable infrastructure the need to (re)connect services dynamically becomes a priority. This process is called Service Discovery. Microservice architectures further exacerbate the importance of reliably discovering and connecting an increasing number of standalone services. The Simplicity Itself team analyzed the main solutions available, as well as the pros and cons of (trying to) do this on your own.

Virtual Panel on Immutable Infrastructure

By Manuel Pais

THE PANELISTS

Chad Fowler is an internationally known software developer, trainer, manager, speaker, and musician. Over the past decade, he has worked with some of the world's largest companies and most admired software developers. Chad is CTO of 6 Wunderkinder. He is the author or co-author of a number of popular software books, including R*ails Recipes and The Passionate Programmer: Creating a Remarkable Career in Software Development*.

Mark Burgess is the CTO and founder of CFEngine, formerly professor of network and system administration at Oslo University College, and the principal author of the CFEngine software. He's the author of numerous books and papers on topics from physics to network and system administration to fiction.

Mitchell Hashimoto is best known as the creator of Vagrant and founder of HashiCorp. He is also an O'Reilly author and professional speaker. He is one of the top GitHub users by followers, activity, and contributions. "Automation obsessed", Mitchell strives to build elegant, powerful DevOps tools at HashiCorp that automate anything and everything. Mitchell is probably the only person in the world with deep knowledge of most virtualization hypervisors.

Servers accumulate change for better or worse over time, including: new applications, upgrades, configuration changes, scheduled tasks, and in the moment fixes for issues. One thing is certain, the longer a server has been provisioned and running the more likely it is in an unknown state. Immutable Servers solve the problem of being certain about server state by creating them anew for each change described previously. Immutable servers make infrastructure scalable and reliable in spite of change. However, this can require fundamentally new views of systems, patterns, deployments, application code and team structure, says panel member Chad Fowler.

Mitchell Hashimoto, another panelist, sees immutable infrastructure as an enabler for repeatable environments (through configuration management tools) and extremely fast deployments times (through pre-built static images) and service orchestration times (through decentralized orchestration tools).

This idea has been deemed utopian by Mark Burgess, our third panelist, due to the large number of external dependencies in any system today. Burgess criticizes the "abuse of the notion of immutability".

InfoQ reached out to them to ask about their definition and borders of immutable infrastructure as well as its benefits and drawbacks, in particular when compared to "desired state" configuration management solutions. Is it a step forward or backwards in effective infrastructure management?

InfoQ: Can you briefly explain your definition of immutable infrastructure?

Chad: I wrote about this on my blog several months ago in detail, but to me the high-level essence of immutable infrastructure shares the same qualities that immutable data structures in functional programming have. Infrastructure components, like in-memory data structures, are running components that can be accessed concurrently. So the same problems of shared state exist.

In my definition of immutable infrastructure, servers (or whatever) are deployed once and not changed. If they are changed for some reason, they are marked for garbage collection. Software is never upgraded on an existing server. Instead, the server is replaced with a new, functionally equivalent server.

Mitchell: Immutable infrastructure is treating the various components of your infrastructure as immutable, or unchangeable. Rather than changing any component, you create a new one with the changes and remove the old one. As such, you can be certain (or more confident) that if your infrastructure is already functioning, a change to config management, for example, won't break what's already not broken.

Immutable infrastructure is also sometimes referred to as "phoenix servers" but I find that term to be less general, since immutability can also apply at the service level, rather than just the server level.

Mark: The term «immutable» is really a misnomer (if infrastructure were really immutable, it would not be much use to anyone: it would be frozen and nothing would happen). What the term tries to capture is the idea that one should try to predetermine as many of the details as possible of a server configuration at the disk-image stage, so that «no configuration is needed». Then the idea supposes that this will make it fast and reliable to spin up machines. When something goes wrong, you just dispose of the machine and rebuild a new one from scratch. That is my understanding of what people are trying to say with this phrase.

I believe there are a number of things wrong with this argument though. The idea of what it means to fix the configuration is left very unclear and this makes the proposal unnecessarily contentious. First of all, predetermining everything about a host is not possible. Proper configuration management deals with dynamic as well as static host state. Take the IP address and networking config, for instance – this has to be set after the machine is spun up. What about executing new programs on demand? What if programs crash, or fail to start? Where do we draw the line between what can and can't be done after the machine has been spun up? Can we add a package? Should we allow changes from DHCP but not from CFEngine or Puppet? Why? In my view, this is

all change. So no machine can be immutable in any meaningful sense of the word.

The real issue we should focus on is what behaviours we want hosts to exhibit on a continuous basis. Or, in my language, what promises should the infrastructure be able to keep?

InfoQ: Can immutable infrastructure help prevent systems divergence while still coping with the need for regular configuration updates?

Chad: Absolutely. It's just a different model for updates. Rather than update an existing system, you replace it. Ultimately, I think this is a question of granularity of what you call a "component". Fifteen years ago, if I wanted to update a software component on a UNIX system, I upgraded the software package and its dependencies. Now I tend to view running server instances as components. If you need to upgrade the OS or some package on the system, just replace the server with one that's updated. If you need to upgrade your own application code, create new server instances with the new code and replace the old servers with it.

Mitchell: Actually, immutable infrastructure makes things a bit worse for divergence if you don't practice it properly. With mutable infrastructure, the idea is that configuration management constantly runs (on some interval) to keep the system in a convergent state. With immutable infrastructure, you run the risk of deploying different versions of immutable pieces, resulting in a highly divergent environment.

This mistake, however, is the result of not properly embracing or adopting immutable infrastructure. With immutable infrastructure, you should never be afraid of destroying a component, so when a new version is available, you should be able to eventually replace every component. Therefore, your infrastructure is highly convergent. However, this is mostly a discipline-and-process thing, which is sometimes difficult to enforce in an organization.

Mark: I don't believe immutable infrastructure helps prevent systems divergence. Trying to freeze configuration up front leads to a "microwave dinner" mentality. Just throw your prebaked package in the oven and suffer through it. It might be okay for some people, but then you have two problems: either you can't get exactly what you need or you have a new problem of making and managing sufficient variety of prepackaged stuff. The latter is a harder problem to solve than just using fast model-based configuration management because it's much harder to see into prepackaged images or "microwave dinners" to see what's in them. So you'd better get your packaging exactly right.

Moreover, what happens if there is something slightly wrong? Do you really want to go back to the factory and repackage everything just to follow the dream of the microwave meal? It is false that prepackaging is the only way to achieve consistency. Configuration tools have proven that. You don't need to destroy the entire machine to make small repeatable changes cheaply. Would you buy a new car because you have a flat tyre, or because it runs out of fuel?

What prevents divergence of systems is having a clear model of the outcome you intend – not the way you package the starting point from which you diverge.

InfoQ: Is immutable infrastructure a better way to handle infrastructure than desired-state convergence? If yes, what are its main advantages? If not, what are its main disadvantages?

Chad: I think so. Of course, there are tradeoffs involved and you have to weigh the options in every scenario, but I think immutable infrastructure is a better default answer than desired-state convergence.

Immutable servers are easier to reason about. They hold up better in the face of concurrency. They are easier to audit. They are easier to reproduce, since the initial state is maintained.

Mitchell: It has its benefits and it has its downsides. Overall, I believe it to be a stronger choice and the right way forward, but it is important to understand it is no silver bullet, and it will introduce problems you didn't have before (while fixing others).

The advantages are deployment speed, running stability, development testability, versioning, and the ability to roll back.

With immutable, because everything is "precompiled" into an image, deployment is extremely fast. You launch a server, and it is running. There may be some basic configuration that happens afterwards but the slow parts are done: compiling software, installing packages, etc.

And because everything is immutable, once something is running, you can be confident that an external force won't be as likely to affect stability. For example, a broken configuration-management run cannot accidentally corrupt configuration.

Immutable infrastructure is incredibly easy to test, and the test results very accurately show what will actually happen at run time. An analogy I like to make

is that immutable infrastructure is to configuration management what a compiled application is to source code. You can unit-test your source code, but when you go to compile it, there is no guarantee that some library versions that could ruin your build didn't change. Likewise, with configuration management, you can run it over and over, but you can't guarantee that if it succeeds that it'll still succeed months down the road. But with a compiled application, or a prebuilt server, all the dependencies are already satisfied and baked in; the surface area of problems that can happen when you go to launch that server are much, much smaller.

Versioning is much simpler and clearer because you can tag a specific image with the configuration-management revision that is baked into it, the revision of an application, the versions of all dependencies, etc. With mutable servers, it's harder to be certain what versions or revisions of what exist on each server.

Finally, you get rollback capability! There are many people who think rollback is a lie, and at some point it is. But if you practice gradual, incremental changes to your infrastructure, rolling back with immutable infrastructure to a recently previous version is cheap and easy: you just replace the new servers with servers launched from a previous image. This has definitely saved us some serious problems a few times, and is very hard to achieve with desired-state configurations.

Mark: The way to get infrastructure right is to have a continuous knowledge relationship with system specifications (what I call promises in CFEngine language). To be fit for purpose, and to support business continuity, you must know that you can deliver continuous consistency. Changing the starting point cannot be the answer unless every transaction is designed to be carried out in separately built infrastructure. That's a political oversimplification, not a technical one, and it adds overhead.

I would say that the "immutable" paradigm is generally worse than one that that balances a planned start image with documented convergent adaptation. The disadvantage of a fixed image is a lack of transparency and immediacy. Advocates of it would probably argue that if they know what the disk image version is, they have a better idea of what the state of the machine is. The trouble with that is that system state is not just what you feed in at the start – it also depends on everything that happens to it after it is running. It promotes a naive view of state.

At a certain scale, pre-caching some decision logic as a fixed image might save you a few seconds in deployment, but you could easily lose those seconds (and a lot more business continuity) by having to redeploy machines instead of adapting and repairing simple issues. If there is something wrong with your car, it gets recalled for a patch; you don't get a new car, else the manufacturers would be out of business.

Caching data can certainly make sense for optimizing effort, as part of an economy of scale, but we should not turn this into a false dichotomy by claiming it is the only way. In general, an approach based on partial disk images, with configuration management for the "last mile" changes makes much more business sense.

In practice, "immutability" (again poor terminology) means disposability. Disposability emerges often in a society when resources seem plentiful. Eventually resources become less plentiful, and the need to handle the waste returns. At that stage, we start to discover that the disposable scheme was actually toxic to the environment (what are the side effects of all this waste?), and we wonder why we were not thinking ahead. We are currently in an age of apparent plenty, with cloud environments hiding the costs of waste, so that developers don't have to think about them. But I wonder when the margins will shrink to the point where we change our minds.

InfoQ: What are the borders for immutable infrastructure, i.e., for which kind of changes would it make sense to replace a server vs. updating its configuration/state?

Chad: If there are borders and some changes are done on an existing server, the server isn't immutable. Idealistically, I don't think we should allow borders. "Immutable" isn't a buzzword. It has meaning. We should either maintain the meaning or stop using the word. An in-between is dangerous and may provide a false sense of security in the perceived benefits of immutability.

That said, systems do need to be quickly fixable, and the methods we're currently using for replacing infrastructure are slower than hot-fixing an existing server. So there needs to be some kind of acceptable hybrid which maintains the benefits of immutability. My current plan for Wunderlist is to implement a hotfix with a self-destruct built in. So if you have to hot-fix a server, it gets marked to be replaced automatically. We haven't done it automatically yet, but we've manually done this and it works well. I see this as an ugly optimization rather than a good design approach.

Mitchell: The borders of immutable infrastructure for me break down to where you want to be able to change things rapidly: small configuration changes, application deploys, etc.

But wanting to be able to deploy an application on an immutable server doesn't make that server immutable. Instead, you should think of the immutability of a server like an onion: it has layers. The base layer of the server (the OS and some configuration and packages) is immutable. The application itself is its own immutable component: a hopefully precompiled binary being deployed to the server. So while you do perhaps have an arguably mutable component in your server, it itself is another versioned immutable component.

What you don't want to be doing for application deploys on immutable infrastructure is to be compiling live on an immutable server. The compilation might fail, breaking your application and perhaps the functionality of the server.

Mark: When a particular configuration reaches the end of its useful life, it should probably be replaced. That ought to be a business judgment, not a technical one. The judgment can be made on economic grounds, related to what would be lost and gained by making a change. But be careful of the hidden costs if your processes are not transparent and your applications are mission critical.

Anytime you have to bring down a system for some reason, it could be an opportunity to replace it entirely without unnecessary interruption, as long as you have a sufficient model of the requirements to replace it with minimum impact, and a hands-free approach to automation for creating that environment. Today, it is getting easy to deploy multiple versions in parallel as separate branches for some kinds of applications. But we have to remember that the whole world is not in the cloud. Planes, ships, rockets, mobile devices are all fragile to change and mission critical. There are thus still embedded devices that spend much of their time largely offline, or with low-rate communications. They cannot be re-imaged and replaced safely or conveniently, but they can be patched and managed by something like CFEngine that doesn't even need a network connection to function.

InfoQ: Is it possible to treat every server of a given infrastructure as immutable or are there some server roles that cannot be treated that way?

Chad: We have what I call «cheats» with immutable infrastructure. Relational databases are a good example. I think it's possible to work with them immutably, but so far it hasn't been worth the effort for us. If we were an infrastructure vendor, I would be applying some effort here, but since we're in the business of making applications for our customers, we have been content to outsource more and more to managed services such as Amazon RDS.

My goal is that our entire infrastructure consists of either pieces we don't manage directly or components that are completely replaceable. We're almost there and so far it's been a very positive experience.

Mitchell: It is possible, but there are roles that are much easier to treat as immutable. Stateless servers are extremely easy to make immutable. Stateful servers such as databases are much trickier, because if you destroy the server, you might be destroying the state as well, which is usually unacceptable.

Mark: Mission-critical servers running monolithic applications cannot generally be managed in this disposable manner. As I understand it, the principal argument for this pattern of working is one of trust. Some people would rather trust an image than a configuration engine. One would like to allow developers to manage and maintain their own infrastructure requirements increasingly. However, if you force everyone to make changes only through disk images, you are tying their hands with regard to making dynamic changes, such as adjusting the number of parallel instances of a server to handle latency and tuning other aspects of performance. Disregarding those concerns is a business decision. Developers often don't have the right experience to understand scalability and performance, and certainly not in advance of deployment.

InfoQ: What are the main conceptual differences between tooling for immutable infrastructure and desired-state convergence?

Chad: Desired-state convergence is in a different realm of complexity to implement. It's a fascinating idea, but at least for my use cases it's outdated. The longer a system lives, the more afraid of it I become. I can't be 100% sure that it is configured the way I want and that it has exactly the right software. Thousands upon thousands of developer hours have gone into solving this problem.

In the world of immutable infrastructure, I think of servers as replaceable building blocks, like parts in an automobile. You

don't update a part. You just replace it. The system is the sum of its parts. The parts are predictable since they don't change. It's conceptually very simple.

Mitchell: The tooling doesn't change much! Actually, all tools used for desired-state convergence are equally useful for immutable infrastructure. Instead, immutable infrastructure adds a compilation step to servers or applications that didn't exist before. For example, instead of launching a server and running Chef, you now use Packer as a compilation tool to launch a server, run Chef, and turn it into an image.

One thing that does change is a mindset difference: immutable tools know they can only run once to build an image whereas desired-state convergence tools expect that they can run multiple times to achieve convergence. In practice, this doesn't cause many problems because you can just run the desired-state convergence tool multiple times when building an image. However, the tools built for immutability tend to be much more reliable in achieving their intended purpose the first time.

Mark: If you want to freeze the configuration of a system to a predefined image, you have to have all the relevant information about its environment up front, and then you are saying that you won't try to adapt down the line. You will kill a host to repair the smallest headache. It's an overtly discontinuous approach to change, as opposed to one based on preservation and continuity. If you think of biology, it's like tissue, where you can lose a few cells from your skin because there are plenty more to do the job. It can only work if resources are plentiful and redundant.

With desired-state convergence, you can make a system completely predictable with repairs and even simple changes in real time and respond to business problems on the fly, at pretty much any scale, making only minimal interventions. This is like the role of cellular DNA in biology. There are repair processes ongoing because there is no redundancy at the intracellular level.

Bulk information is easier to manage from a desired-state model than from piles of bulk data because it exploits patterns to good advantage. You can easily track changes to the state (for compliance and auditing purposes) because a model defines your standard of measurement over many versions. Imagine compliance auditing like PCI or HIPPA. How do you prove to an auditor that your system is compliant? If you don't have a model with desired outcome, that becomes a process of digging around in files and looking for version strings. It's very costly and time wasting.

InfoQ: Is the choice and maturity of today's tools and the immutable-infrastructure pattern itself enough to become mainstream?

Chad: Probably not. The foundations are getting better and better with both hosted and internal cloud providers and frameworks, but creating a solid immutable architecture is not currently the path of least resistance. I think most of us will move in this direction over time, but it's currently far from mainstream.

Mitchell: Not yet, but they're leaps and bounds better than they were a year ago, and they're only going to continue to become more mature and solve the various problems early adopters of immutable infrastructure may be having.

All the tools are definitely mature enough to begin experimenting with and testing for some aspects of your infrastructure, though.

Mark: I don't believe it can become mainstream unless all software becomes written in a completely stateless way, which would then be fragile to communication faults in a distributed world. Even then, I don't think it is desirable. Do we really want to argue that it is better for the whole world to eat microwave dinners, or to force chefs to package things in plastic before eating it? If history has taught us anything, it is that people crave freedom. We have to understand that disposability is a large-scale economic strategy that is just not suitable at all scales.

InfoQ: Is immutable infrastructure applicable for organizations running their services on physical infrastructure (typically for performance reasons)? If so, how?

Chad: Sure. While perhaps the servers themselves would run longer and probably require in-place upgrades to run efficiently, with the many options available for virtualization, everything on top of that is fair game. I suppose it would be possible to take the approach further down the stack, but I haven't had to do it and I don't want to speculate.

Mitchell: Yes, but it does require a bit more disciplinary work. The organization needs to have in place some sort of well automated process for disposing of and re-imaging physical machines. Unfortunately, many organizations do not have this, which is somewhat of a prerequisite to making immutable infrastructure very useful.

For example, what you really want is something like Mesos or Borg for physical hardware.

Mark: The immutable-infrastructure idea is not tied specifically to virtualization. The same method could be applied to physical infrastructure, but the level of service discontinuity would be larger.

Today, immutability is often being mixed up with arguments for continuous delivery in the cloud, but I believe that disposable computing could easily be contrary to the goals of continuous delivery because it adds additional hoops to jump through to deploy change, and makes the result less transparent.

InfoQ: With respect to an application's architecture and implementation, what factors need to be taken into account when targeting an immutable infrastructure?

Chad: Infrastructure and services need to be discoverable. Whatever you're using to register services needs to be programmable via an API. You need to have intelligent monitoring and measuring in place, and your monitoring needs to be focused less on the raw infrastructure than on the end purpose of the service than you're probably used to.

Everything needs to be stateless where possible.

As I mentioned previously, we have cheats like managed database systems. I'm not going to comment on how you have to change architecture to have your own immutable, disposable database systems since it's thankfully not a problem I've needed or wanted to solve yet.

Mitchell: Nothing has to change in order to target immutable infrastructure, but some parts of architecting and developing an application become much easier. Developers in an immutable infrastructure have to constantly keep in mind that any service they talk to can die at any moment (and hopefully can be replaced rather quickly). This mindset alone results in developers generally building much more robust and failure-friendly applications.

With strongly coupled, mutable infrastructures, it isn't uncommon to interrupt a dependent service of an application and have that application be completely broken until it is restarted with the dependent service up.

While keeping immutable infrastructure in mind, applications are much more resilient. As an example from our own infrastructure managing Vagrant Cloud, we were able to replace and upgrade every server (our entire infrastructure) without any perceivable downtime and without touching the web front ends during the replacement process. The web applications just retried some connects over time and eventually came back online. The only negative experience was that for some people their requests were queued a bit longer than usual!

Mark: The aim should not be to make applications work around an immutable infrastructure. You don't pick the job to fit the tools. The immutable infrastructure is usually motivated as a way of working around the needs of application developers. The key question is how do you optimize a continuous-delivery pipeline?

InfoQ: Does immutable infrastructure promote weak root-cause analysis as it becomes so easy to trash a broken server to repair service instead of fixing it? If so, is that a problem or just a new modus operandi?

Chad: I don't think so. In the worst case, it doesn't change how we do root-cause analysis, because re-

> One thing is certain, the longer a server has been provisioned and running the more likely it is in an unknown state. Immutable Servers solve the problem of being certain about server state by creating them anew for each change described previously

placing and rebooting are sort of the same last-ditch effort when something is wrong. In the best case, it makes it easier to experiment with possible solutions, tweak variables one at a time, bring up temporary test servers, swap production servers in and out, etc.

I see the point you're hinting at in the question, though. There may be a class of problems that is all but eliminated (read: obscured to the point of essentially not existing) if the average lifespan of a server is less than one day. It may also be harder to pinpoint these problems if they do occasionally pop up without knowing to try server longevity as a variable.

Maybe that's okay.

Mitchell: Because it is very likely that the server you're replacing it with will one day see that same issue, immutable infrastructure doesn't promote any weaker root-cause analysis. It may be easier to ignore for a longer period of time, but most engineering organizations will care to fix it properly at some point.

Actually, I would say the root cause analysis becomes much stronger. Since the component is immutable and likely to exhibit the same problems under the same conditions, it is easier to reproduce, identify, fix, and finally to deploy your change across your entire infrastructure.

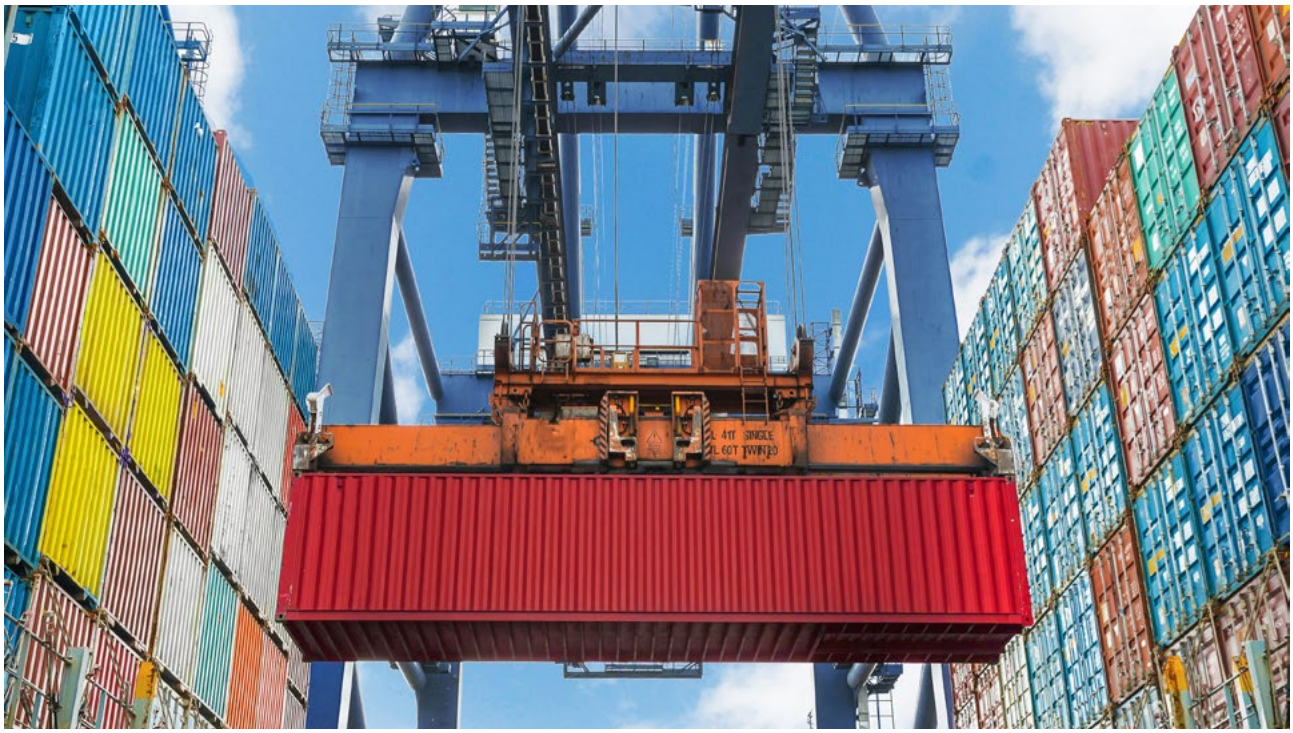
Additionally, desired-state configuration has a high chance of making the problem worse: a scheduled run of the configuration management system may mask the real underlying issue, causing the ops team to spend more time trying to find it or even to just detect it.

Mark: Disposal of causal evidence potentially makes understanding the environment harder. Without model-based configuration, a lot of decisions get pushed down into inscrutable scripts and pre-templated files, where the reason for the decisions only lives in some developer's head. That process might work to some extent if the developer is the only one responsible for making it, but it makes reproducibility very challenging. What happens when the person with that knowledge leaves the organization?

In psychology, one knows that humans cannot remember more than a small number of things without assistance. The question is how do you create a knowledge-oriented framework where intent and outcome are transparent and quickly reproducible with a minimum of repeated effort. This is what configuration management was designed for and I still believe that it is the best approach to managing large parts of the infrastructure configuration. The key to managing infrastructure is in separating and adapting different behaviours at relevant scales in time and space. I have written a lot about this in my book *In Search of Certainty: The Science of Our Information Infrastructure*.

Immutable servers are easier to reason about. They hold up better in the face of concurrency. They are easier to audit. They are easier to reproduce, since the initial state is maintained.

Docker: Present and Future



Chris Swan is CTO at CohesiveFT, a provider of cloud networking software. Previously, he spent a dozen years in financial services as a technologist to bankers and banker to technologists. He spent most of that time at large Swiss banks wrangling infrastructure for app servers, compute grids, security, mobile, and cloud. Chris also enjoys tinkering with the Internet of Things, including a number of Raspberry Pi projects.

Docker is a toolset for Linux containers designed to build, ship, and run distributed applications, first released as an open-source project by dotCloud in March 2013. The project quickly became popular, leading dotCloud to rebrand as Docker, the company (and ultimately selling off their original PaaS business). The company released Docker 1.0 in June 2014 and has since sustained the monthly release cadence that led up to the June release.

The 1.0 release marked the point where the company considered the platform sufficiently mature for use in production (with the company and partners selling support options). The monthly release of point updates shows that the project is still evolving quickly, adding new features, and addressing issues as they are found. The project has successfully decoupled "ship" from "run", so images sourced from any version of Docker can be used with any other version (with both forward and backward compatibility), something that provides a stable foundation for Docker use despite rapid change.

The growth of Docker into one of the most popular open-source projects could be perceived as hype, but there is a great deal of substance. Docker has attracted support from the industry, including Amazon, Google, and Microsoft. It is almost ubiquitously available wherever Linux can be found. In addition to the big names, many startups are growing up around Docker or changing direction to better align with Docker. Those partnerships (large and small) are helping to drive rapid evolution of the core project and its surrounding ecosystem.

A brief technical overview of Docker

Docker makes use of Linux-kernel facilities such as cgroups, namespaces, and SELinux to provide isolation between containers. At first, Docker was a front end for the LXC container-management subsystem, but release 0.9 introduced libcontainer, which is a native Go-language library that provides the interface between user space and the kernel.

Containers sit on top of a union file system, such as AUFS, which allows for the sharing of components such as operating-system images and installed libraries across multiple containers. The layering approach in the file system is also exploited by the Dockerfile DevOps tool, which is able to cache operations that have already successfully completed. This can greatly speed up test cycles by eliminating the wait time usually taken to install operating systems and application dependencies. Shared libraries between containers can also reduce RAM footprint.

A container is started from an image, which may be locally created, cached locally, or downloaded from a registry. Docker, Inc. operates the Docker Hub Registry, which hosts official repositories for a variety of operating systems, middleware, and databases. Organisations and individuals can host public repositories for images at Docker Hub, and there are also subscription services for hosting private repositories. Since an uploaded image could contain almost anything, the Docker Hub Registry provides an automated build facility (previously called "trusted build") where images are constructed from a Dockerfile that serves as a manifest for the contents of the image.

Containers versus VMs

Containers are potentially much more efficient than VMs because they're able to share a single kernel and application libraries. This can lead to substantially smaller RAM footprints even when compared to virtualisation systems that can make use of RAM over-commitment. Storage footprints can also be reduced when deployed containers share underlying image layers. IBM's Boden Russel has benchmarked these differences.

Containers also present a lower systems overhead than VMs, so the performance of an application inside a container will generally be the same or better than the same application running within a VM. A team of IBM researchers have published a performance comparison of virtual machines and Linux containers.

One area where containers are weaker than VMs is isolation. VMs can take advantage of ring -1 hardware isolation such as that provided by Intel's VT-d and VT-x technologies. Such isolation prevents VMs from breaking out and interfering with each other. Containers don't yet have any form of hardware isolation, which makes them susceptible to exploits. A proof-of-concept attack named Shocker showed that Docker versions prior to 1.0 were vulnerable. Although Docker 1.0 fixed the particular issue exploited by Shocker, Docker CTO Solomon Hykes said, "When we feel comfortable saying that Docker out of the box can safely contain untrusted uid0 programs, we will say so clearly." Hykes's statement acknowledges that other exploits and associated risks remain, and that more work will need to be done before containers can become trustworthy.

For many use cases, the choice of containers or VMs is a false dichotomy. Docker works well within a VM, which allows it to be used on existing virtual infrastructure, private clouds, and public clouds. It's also possible to run VMs inside containers, which is something that Google uses as part of its cloud platform. Given the widespread availability of infrastructure as a service (IaaS) that provides VMs on demand, it's reasonable to expect that containers and VMs will be used together for years to come. It's also possible that container management and virtualisation technologies might be brought together to provide a best-of-both-worlds approach; so a hardware trust-anchored micro-virtualisation implementation behind libcontainer could integrate with the Docker tool chain and ecosystem at the front end, but use a different back end that provides better isolation. Micro virtualisation (such as Bromium vSentry and VMware's Project Fargo) is already used in desktop environments to provide hardware-based isolation between applications, so similar approaches could be used along with libcontainer as an alternative to the container mechanisms in the Linux kernel.

"Dockerizing" applications

Pretty much any Linux application can run inside a Docker container. There are no limitations on choice of languages or frameworks. The only practical limitation is what a container is allowed to do from an operating-system perspective. Even that bar can be lowered by running containers in privileged mode, which substantially reduces controls (and correspondingly increases risk of the containerised application being able to cause damage to the host operating system).

Containers are started from images, and images can be made from running containers. There are essentially two ways to get applications into containers: manually and Dockerfile.

Manual builds

A manual build starts by launching a container with a base operating-system image. An interactive terminal can then be used to install applications and dependencies using the package manager offered by the chosen flavour of Linux. Zef Hemel provides a walk through of the process in his InfoQ article "Docker: Using Linux Containers to Support Portable Application Deployment". Once the application is installed, the container can be pushed to a registry (such as Docker Hub) or exported into a tar file.

Dockerfile

Dockerfile is a system for scripting the construction of Docker containers. Each Dockerfile specifies the base image to start from and then a series of commands that are run in the container and/or files that are added to the container. The Dockerfile can also specify ports to be exposed, the working directory when a container is started, and the default command on start up. Containers built with Dockerfiles can be pushed or exported just like manual builds. Dockerfiles can also be used in Docker Hub's automated build system so that images are built from scratch in a system under the control of Docker, Inc. with the source of that image visible to anybody that might use it.

One process?

Whether images are built manually or with Dockerfile, a key consideration is that only a single process is invoked when the container is launched. For a container serving a single purpose, such as running an application server, running a single process isn't an issue (and some argue that containers should only have a single process). For situations where it's desirable to have multiple processes running inside a container, a supervisor process must be launched that can then spawn the other desired processes. There is no init system within containers, so anything that relies on systemd, upstart, or similar won't work without modification.

Containers and microservices

A full description of the philosophy and benefits of using a microservices architecture is beyond the scope of this article (and is well covered in the InfoQ e-mag Microservices). Containers are however a convenient way to bundle and deploy instances of microservices.

Whilst most practical examples of large-scale microservices deployments to date have been on top of (large numbers of) VMs, containers offer the opportunity to deploy at a smaller scale. The ability for containers to have a shared RAM and disk footprint for operating systems and libraries and common application code also means that deploying multiple versions of services side by side can be made very efficient.

Connecting containers

Small applications will fit inside a single container, but in many cases an application will be spread across multiple containers. Docker's success has spawned a flurry of new application compositing tools, orchestration tools, and platform as a service (PaaS) implementations. Behind most of these efforts is a desire to simplify the process of constructing an application from a set of interconnected containers. Many tools also help with scaling, fault tolerance, performance management, and version control of deployed assets.

Connectivity

Docker's networking capabilities are fairly primitive. Services within containers can be made accessible to other containers on the same host, and Docker can also map ports onto the host operating system to make services available across a network. The officially sponsored approach to connectivity is libchan, which is a library that provides Go-like channels over the network. Until libchan finds its way into applications, there's room for third parties to provide complementary network services. For example, Flocker has taken a proxy-based approach to make services portable across hosts (along with their underlying storage).

Compositing

Docker has native mechanisms for linking containers together by which metadata about a dependency can be passed into the dependent container and consumed within as environment variables and hosts entries. Application compositing tools like Fig and geard express the dependency graph inside a single file so that multiple containers can be brought together into a coherent system. CenturyLink's Panamax compositing tool takes a similar underlying approach to Fig and geard, but adds a web-based user interface, and integrates directly with GitHub so that applications can be shared.

Orchestration

Orchestration systems like decking, New Relic's Centurion and Google's Kubernetes all aim to help with the deployment and lifecycle management of containers. There are also numerous examples (such as Mesosphere) of Apache Mesos (and particularly its Marathon framework for long-running applications)

being used along with Docker. By providing an abstraction between the application needs (e.g. expressed as a requirement for CPU cores and memory) and underlying infrastructure, the orchestration tools provide decoupling that's designed to simplify both application development and datacentre operations. There is such a variety of orchestration systems because many have emerged from internal systems previously developed to manage large-scale deployments of containers; for example Kubernetes is based on Google's Omega system that's used to manage containers across the Google estate.

Whilst there is some degree of functional overlap between the compositing tools and the orchestration tools there are also ways that they can complement each other. For example, Fig might be used to describe how containers interact functionally whilst Kubernetes pods might be used to provide monitoring and scaling.

Platforms (as a service)

A number of Docker native PaaS implementations such as Deis and Flynn have emerged to take advantage of the fact that Linux containers provide a great degree of developer flexibility (rather than being "opinionated" about a given set of languages and frameworks). Other platforms such as Cloud Foundry, OpenShift, and Apcera's Continuum have taken the route of integrating Docker-based functionality into their existing systems, so that applications based on Docker images (or the Dockerfiles that make them) can be deployed and managed alongside apps using previously supported languages and frameworks.

All the clouds

Since Docker can run in any Linux VM with a reasonably up-to-date kernel, it can run in pretty much every cloud that offers IaaS. Many of the major cloud providers have announced additional support for Docker and its ecosystem.

Amazon has introduced Docker in its Elastic Beanstalk system (which is an orchestration service over underlying IaaS). Google has Docker-enabled managed VMs, which provide a halfway house between the PaaS of App Engine and the IaaS of Compute Engine. Microsoft and IBM have both announced services based on Kubernetes so that multi-container applications can be deployed and managed on their clouds.

To provide a consistent interface to the wide variety of back ends now available, the Docker team introduced libswarm (now Docker Swarm) to integrate with a multitude of clouds and resource-management systems. One of the stated aims of libswarm is to "avoid vendor lock-in by swapping any service out with another". This is accomplished by presenting a consistent set of services (with associated APIs) that attach to implementation-specific back ends. For example, the Docker server service presents the Docker remote API to a local Docker command-line tool so that containers can be managed on an array of service providers.

New service types based on Docker are still in their infancy. London-based Orchard offered a Docker hosting service, but Docker, Inc. said that the service wouldn't be a priority after acquiring Orchard. Docker, Inc. has also sold its previous dotCloud PaaS business to cloudControl. Services based on older container-management systems such as OpenVZ are already commonplace, so to a certain extent Docker needs to prove its worth to hosting providers.

Docker and the distros

Docker has already become a standard feature of major Linux distributions like Ubuntu, Red Hat Enterprise Linux (RHEL), and CentOS. Unfortunately, the distributions move at a different pace to the Docker project, so the versions found in a distribution can be well behind the latest available. For example, Ubuntu 14.04 was released with Docker 0.9.1, and that didn't change on the point release upgrade to Ubuntu 14.04.1 (by which time Docker was at 1.1.2). There are also namespace issues in official repositories since Docker was also the name of a KDE system tray – so with Ubuntu 14.04 the package name and command-line tool are both "docker.io".

Things aren't much different in the enterprise Linux world. CentOS 7 comes with Docker 0.11.1, a development release that precedes Docker, Inc.'s announcement of production readiness with Docker 1.0. Linux distribution users that want the latest version for promised stability, performance, and security would be better off following the installation instructions and using repositories hosted by Docker, Inc. rather than taking the version included in their distribution.

The arrival of Docker has spawned new Linux distributions such as CoreOS and Red Hat's Project Atomic that are designed to be a minimal environment for running containers. These distributions come with newer kernels and Docker versions than the traditional distributions. They also have lower memory and disk footprints. The new distributions also come with new tools for managing large-scale deployments such as fleet, "a distributed init system" and etcd for metadata management. There are also new mechanisms for updating the distribution itself

so that the latest versions of the kernel and Docker can be used. This acknowledges that one of the effects of using Docker is that it pushes attention away from the distribution and its package-management solution, making the Linux kernel (and the Docker subsystem using it) more important.

New distributions might be the best way to run Docker, but traditional distributions and their package managers remain very important within containers. Docker Hub hosts official images for Debian, Ubuntu, and CentOS. There's also a semi-official repository for Fedora images. RHEL images aren't available in Docker Hub, as they're distributed directly from Red Hat. This means that the automated build mechanism on Docker Hub is only available to those using pure open-source distributions (and willing to trust the provenance of the base images curated by the Docker team).

Whilst Docker Hub integrates with source-control systems such as GitHub and Bitbucket for automated builds, the package managers used during the build process create a complex relationship between a build specification (in a Dockerfile) and the image resulting from a build. Non-deterministic results from the build process aren't specifically a Docker problem – it's a result of how package managers work. A build done one day will get a given version, and a build done another time may get a later version, which is why package managers have upgrade facilities. The container abstraction (caring less about the contents of a container) along with container proliferation (because of lightweight resource utilisation) is, however, likely to make this a pain point that gets associated with Docker.

The future of Docker

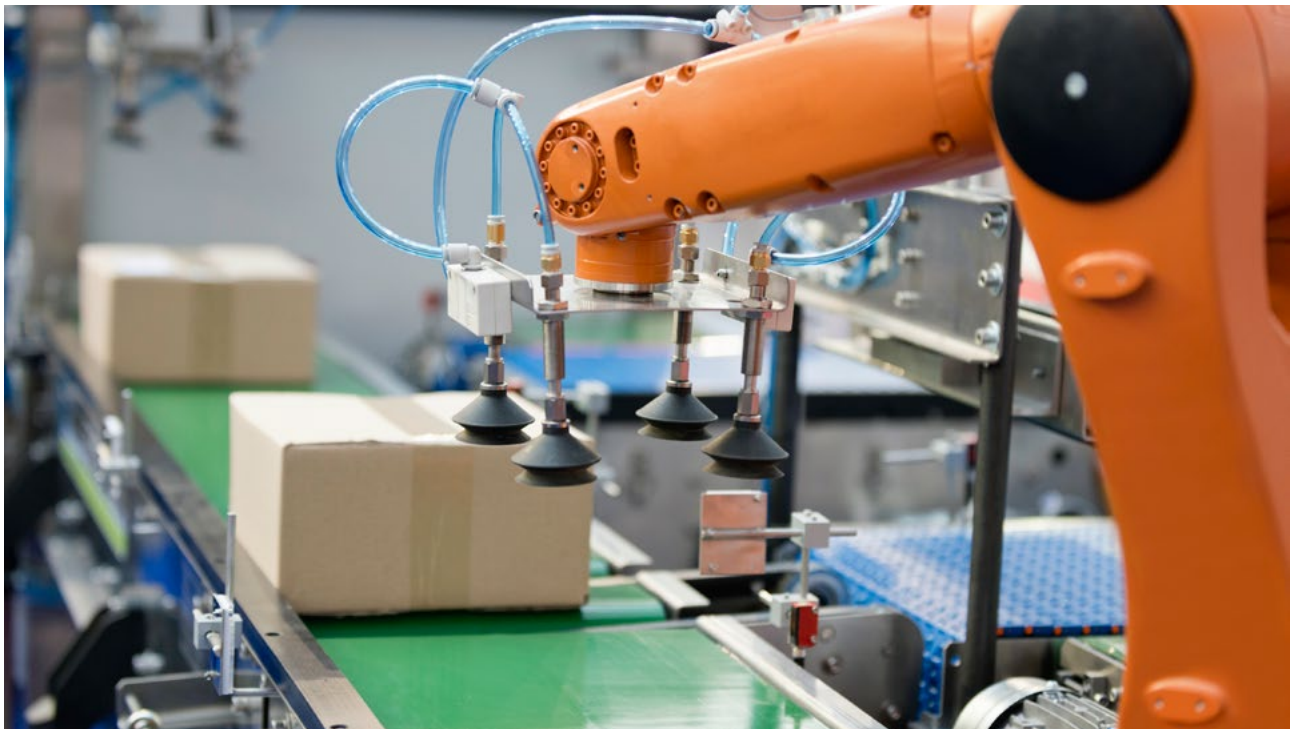
Docker, Inc. has set a clear path to the development of core capabilities (libcontainer), cross-service management (libswarm), and messaging between containers (libchan). Meanwhile, the company has already shown a willingness to consume its own ecosystem with the Orchard acquisition. There is however more to Docker than Docker, Inc., with contributions to the project coming from big names like Google, IBM, and Red Hat. With a benevolent dictator in the shape of CTO Solomon Hykes at the helm, there is a clear nexus of technical leadership for both the company and the project. Over its first 18 months, the project has shown an ability to move fast by using its own output, and there are no signs of that abating.

Many investors are looking at the features matrix for VMware's ESX/vSphere platform from a decade ago and figuring out where the gaps (and opportunities) lie between enterprise expectations driven by the popularity of VMs and the existing Docker ecosystem. Areas like networking, storage, and fine-grained version management (for the contents of containers) are presently underserved by the existing Docker ecosystem, and provide opportunities for both startups and incumbents.

Over time, it's likely that the distinction between VMs and containers (the "run" part of Docker) will become less important, which will push attention to the build and ship aspects. The changes here will make the question of what happens to Docker much less important than what happens to the IT industry as a result of Docker.

For many use cases the choice of containers or VMs is a false dichotomy. Docker works well within a VM, which allows it to be used on existing virtual infrastructure, private clouds and public clouds.

How I Built a Self-Service Automation Platform



Brian Carpio is a technical leader in the IT space with over 15 years of experience Brian designed and supported solutions for multiple Fortune 500 and 100 companies. His specialty is in large scale environments where automation is a must. Brian loves working with new technology like NoSQL and AWS.

The current buzz in the industry is how DevOps is changing the world and how the traditional role of operations is changing. These discussions lead to building self-service automation tools that let developers deploy and support their own code. I want to discuss a little the work I did at a previous company and how it changed the way the company did business.

AWS architecture

We used AWS as the foundation of our platform, but using AWS isn't the be-all and end-all; if anything, it presented its own challenges to the self-service platform we ended up designing.

Cloud-agnostic

Amazon provides a wide range of services from basic EC2 features like instance creation to full blown DBaaS (database as a service). One of our most heated discussions was over what AWS services we actually wanted to support. I was a huge advocate of keeping it simple and sticking strictly to EC2-related services (compute, storage, load balancing, and public IPs). The reason was simple: I knew that eventually our AWS expense would grow so much that it might eventually make sense to build our own private cloud using OpenStack or other open standards. Initially, this wasn't a well-accepted viewpoint because many of our developers wanted to play with as many AWS services as they could, but this paid off in the long run.

Environments

AWS doesn't provide an environments concept, so the first architectural decision was on environments. Having spent the last 15 years working in operations, I

understood that it was critical to design an EC2 infrastructure that was identical among development, staging, and production. So we made the decision that we would create separate VPCs for each of these environments. Not only did the VPC design allow for network and routing segregation between each environment, it also allowed developers to have a full production-like environment for development purposes.

Security groups

The next decision was how security groups would work. I've seen complex designs where databases, webservers, product lines, etc. all have their own security groups, and while I don't fault anyone for taking such an approach, it didn't really lend itself to the type of automation platform I wanted to design. I wasn't willing to give development teams the ability to manage their own security groups so I decided there would only be two. I simply created a public security group for instances that lived in the public subnets and a private security group that lived in the private subnets.

This allowed for strict sets of rules regarding which ports (80/443) were open to the Internet and which ports would be open from the public to the private zone. I wanted to force all of the applications to listen on a specific port on the private side.

An amazing thing happens when you tell a developer, "You can push this button an get a Tomcat server, and if your app runs on port 8080 everything will work. However, if you want to run your app on a non-standard port, you need to get security approval, and wait for the network team to open the port." The amazing thing is that quickly every application in your environment begins to run on standard ports, creating an environment of consistency.

Puppet

We next decided which configuration-management system to use. This was easy, since I wrote the prototype application and no one else in my team existed at the time I picked Puppet. Puppet is mature and it allowed me to create a team of engineers who strictly wrote Puppet modules, which we ultimately made available via the self-service REST API.

Puppet ENC

At the time we built our platform, Hira was still a bolt-on product, and we needed a way to programmatically add nodes to Puppet as our self-service automation platform created them. Nothing really stood out as a solution, so we decided to roll our own Puppet ENC, which can be found here: http://www.briancarpio.com/2012/08/17/puppet-enc-with-mongodb-backend/

By using the ENC we created, which supported inheritance, we were able to define a default set of classes for all nodes and then allow specific nodes to have optional Puppet modules. For instance, if a developer wanted a NodeJS server, they could simply pass the NodeJS module, but by using the inheritance feature of our ENC, we were able to enforce all the other modules like monitoring, security, base, etc.

Puppet environments

We decided to use Puppet environments entirely differently than how Puppet Labs intended them to be used. We used them to tag specific instances to specific versions of our Puppet manifest. By using a branch strategy for Puppetfile and r10k, we were able to create stable releases of our Puppet manifest. For instance, we had a v3 branch of our Puppetfile, which represented a specific set of Puppet modules at a specific version; this allowed us to work on a v4 branch of Puppetfile that introduced breaking changes to the environment without impacting existing nodes. A node in our ENC would look like this:

```
classes:

  base: ‘’

  nptclient: ‘’

  nagiosclient: ‘’

  backupagent: ‘’

  mongodb: ‘’

environment: v3

parameters:
```

This allowed our development teams to decide when they actually wanted to promote existing nodes to our new Puppet manifest or simply redeploy nodes with the new features.

Command line to REST API

Our self-service platform started with my working with a single team who wanted to go to the cloud. Initially, I wrote a simple Python module that used boto and Fabric to do the following:

- Enforce a naming convention.
- Create an EC2 instance.
- Assign the instance to the correct environment (dev, stg, prd).
- Assign the correct security group.
- Assign the instance to a public or private subnet.
- Tag the instance with the application name and environment.

- Add the node and correct Puppet modules to the Puppet ENC.
- Add the node to Amazon Route 53.
- Register the node with Zabbix.
- Configure Logstash for Kibana.

By using Puppet, the node would spin up, connect to the Puppet master, and install the default set of classes plus the classes passed by our development teams. In the steps above, Puppet was responsible for configuring Zabbix, registering with the Zabbix server, and configuring Logstash for Kibana.

Rapid adoption

Once development teams were able to deploy systems quickly, bypassing traditional processes, adoption spread like wildfire. After about four different teams began using this simple Python module, it became clear that we needed to create a full-blown REST API.

The good thing about starting with a simple Python module was that the business logic for interacting with the system was defined for a team of real developers tasked with building a REST API. The prototype provided the information they needed to create the infrastructure across the environments, and interact with Zabbix, Kibana, Route 53, etc.

It wasn't long till we were able to implement a simple authentication system that used Crowd for authentication and provided team-level ownership of everything from instances to DNS records.

Features

Over time, the number of features grew. Here is the list of features that finally seemed to provide the most value across 70 development teams around the globe.

- Instance creation and management
- Database deployments (MongoDB, Cassandra, ElasticSearch, MySQL)
- Haproxy with A/B deployment capabilities using Consul
- Elastic load-balancer management
- Data encryption at rest
- DNS management
- CDN package uploading
- Multi-tenant user-authentication system

Conclusion

While the original idea was simple, it revolutionized the way my company delivered software to market. By centrally managing the entire infrastructure with Puppet, we had an easy way to push out patches and security updates. By creating an easy-to-use API, we let development teams deploy code quickly and consistently from development through production. We were able to maintain infrastructure best practices with backups, monitoring, etc., all built into the infrastructure that development teams could deploy with the click of a mouse or an API call.

Managing Build Jobs for Continuous Delivery

Martin Peston is a build and continuous-integration manager for Gamesys, the UK's leading online and mobile-gaming company, owners of the popular bingo brand Jackpotjoy and social gaming success Friendzys on Facebook.
Martin has 15 years of cross-sector IT experience, including work in aerospace, defence, health, and gambling. He has dedicated around half his career to build management and is passionate about bringing continuous-delivery best practices to software-development companies.
In his spare time, Martin is a keen amateur astronomer and is author of A User's Guide to the Meade LXD55 and LXD75 Telescopes, published by Springer in 2007.

Continuous delivery (CD) takes an evolving product from development through to production. Continuous integration (CI) plays an important part in CD and defines the software development at the beginning of the process.

There is a wealth of information describing CI processes but there is little information for the many different CI tools and build jobs that compile and test the code, which are central to the CI process.

In typical CI processes, developers manually build and test source code on their own machines. They then commit their modifications to a source-control-management system. A build tool will subsequently run jobs that compile and test the code. The built artifact is then uploaded to a central repository ready for further deployment and testing.

Hence, it is the role of jobs in the CI tool to manage the continuous modifications of the source code, run tests, and manage the logistics of artifact transportation through the deployment pipeline.

The number of jobs in a build tool can range from just a few to perhaps several thousand, all performing various functions. Fortunately, there is a way to manage all of these jobs in a more efficient manner.

Automatic creation of build jobs

So why should we set up a facility to automate the creation of other jobs?

Build-tool user guides describe how to create build jobs but they do not often explain how to manage them in any great detail, despite the tools' ability to do so.

Normally, the developer will know how the application is built and so takes sole responsibility for the creation and configuration of the build job. However, this process has some disadvantages:

1. **Software architectural constraints:** Applications are likely to be built differently from one another, depending upon architecture. One developer may interpret how an application is built differently from another developer and so the build configuration will slightly differ from one job to another. Hence, a large number of jobs become unmaintainable if they are all configured differently.
2. **Human factor:** Manually creating a job introduces the risk of making mistakes, especially if a new job is created by copying an existing job and then modifying that job.
3. **Job timeline control:** There is generally no record kept of job configuration for every build, so if a job configuration is changed, it could break previous builds.

Because of the points above, if developers are left to create build jobs without a consistent method for creating jobs, then they are going end up with a poorly maintained CI build system and a headache to manage. This defeats the purpose of a CI system: being lean and maintainable.

Most build tools can create, maintain, and back up build jobs automatically, usually via API scripts. However, these features, despite mention in the build-tool user manual, are often overlooked and not implemented.

Advantages for automating the creation of build jobs

There are several advantages of using a master build job to automatically create multiple jobs in order to facilitate the CI build process:

1. All build job configurations are created from a set of job templates or scripts. These can be placed under configuration control. Creation of a new job is then a simple matter of using the template or running the scripts. This can cut job-creation time significantly (many minutes to a few seconds). Job configuration changes are made easier; configuration changes to the build-job template ensure that all subsequent new jobs will inherit the changes.
2. Consistency with all existing build-job configurations implies that globally updating all configurations via tools or scripts is more straightforward than with a cluttered inconsistent system.
3. The developer does not necessary need detailed knowledge of the build tool to create a build job.
4. The ability to automatically create and tear down build jobs is part of an agile, CI-build lifecycle.

Let's discuss these points.

Point 1: Configuration control of job configuration

It is good practice that every component of a CI system be placed under configuration control. That includes the underlying build systems and the build jobs, not just the source code.

Most build tools store build-job configuration as a file on the master system. These files are set to a common XML format and are normally accessible via the front-end interface directly through a REST API of some kind. Some build tools have a built-in feature that can save job configuration as a file to any location.

Since the job configuration can be saved as a file, it can be stored in a configuration-management (CM) system. Any change in job configuration can be recorded either by modifying the file directly and then uploading the file to the build tool or by modifying the job configuration from the build tool, saving the file, and uploading it manually to the CM system. The method carried out depends on how easy it is to access the job-configuration file directly from the build tool.

Another important reason for storing job configuration in a CM system is that in the event of a catastrophic failure that loses all job configurations, the build system can recover relatively quickly and restore all jobs, build logs, and job history to their last known good states.

Point 2: Job maintenance

Maintaining a thousand jobs of different configurations would be a major headache, which is why it is so important to standardise job configurations. If a required change impacts a large number of similar jobs, then writing scripts or creating special tasks to modify these jobs would be straightforward.

However, having a thousand jobs in a build system is not the best strategy for CI. We will discuss this further under Point 4.

Point 3: Developer control

Although developers are encouraged to work autonomously, building their applications using a centralised build system, they need to work within guidelines outlined by the build process.

A developer will expect fast feedback on their code changes and will not want to waste time messing around with build tools and job configuration. Provid-

ing developers with a one-button self-service-style solution to automatically set up build jobs will help them achieve their development task objectives more quickly.

Point 4: A lean CI system

Although software-development teams may adopt agile methods, the build tools they work with are not necessarily used in the same way. A properly configured build tool should not contain thousands of long-lived build jobs. Ideally, jobs should be created on demand as part of a CI lifecycle. As long there is the means to recreate any build job from any historical job configuration and that build relics are retained (i.e. logs, artefacts, test reports, etc.), only a handful of jobs should exist in the build tool.

Of course, an on-demand job creation/teardown solution may not be an achievable goal, but the point is that the number of jobs in a build tool should be kept to a manageable level.

Day-to-day development

Build management governs the tools and processes necessary to assist developers in day-to-day activities with regards to build and CI. Utilities can be set up to help development teams to create build jobs and provide flexibility in job configuration but while remaining governed by software-development best practices.

The CI job suite

Consider typical tasks that developers carry out during day-to-day development:

- Build a baseline.
- Release the baseline.
- Build a release branch (often combined with releasing the baseline).
- Create a development branch and build the branch.
- Run integration tests

Best practices aside – i.e. develop on baseline as much as possible, hence few or no development branches – software-development teams generally follow the tasks outlined above by running a suite of build jobs in the build tool.

This is where automatic creation of jobs adds huge value to the software-development process. If a developer wanted to manually create four or five jobs to manage all of the development tasks in day-to-day operations, it would take a significant amount of time and effort. Compare this to automatically creating jobs in a fraction of the time at a push of a button.

When to implement a self-service job-creation solution

Implementing a self-service solution for creating build jobs is applicable in the following cases:

1 Initialisation of new greenfield projects (those that have never been built before).
2 Projects that already exist but never had any build jobs set up for them – i.e. building manually from a developer's own machine.

There are also cases where implementing self-service job-creation may not apply:

1 If a product only has a few jobs – such as with large, monolithic applications – then it may be considered not of value to the business to spend considerable time and effort to design, test, and roll out an automated create-job process for just those few projects. However, as the business expands, the product architecture will become more complex as more development streams are set up to manage the extra workload, hence the number

Consistent build job configuration leads to easier job maintenance and drives consistent development practices across a company

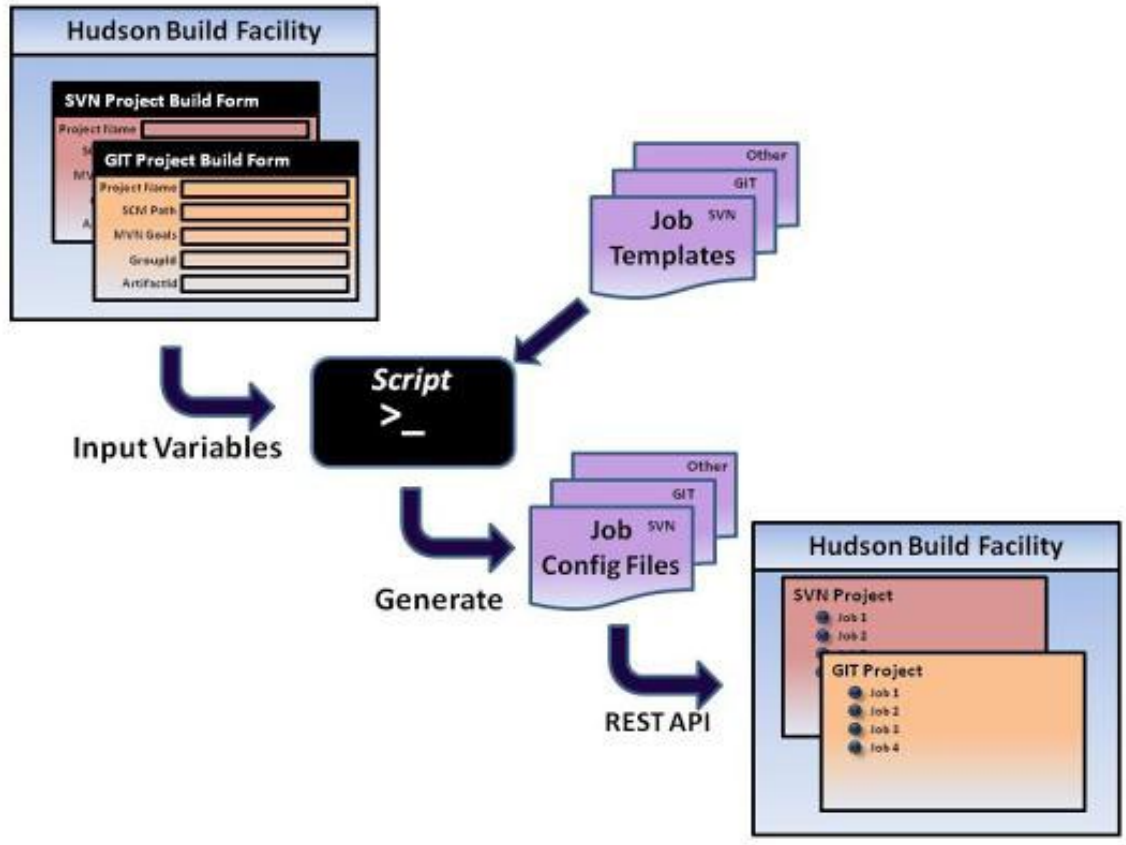


Diagram 1: Hudson job-creation workflow.

of jobs in the build tool will inevitably increase.

2 Projects with existing jobs may have issues. Consider two scenarios of dealing with existing jobs in the build tool:

- Delete the existing jobs and recreate them with the job-creation utility, thereby losing all historical build information and logs.
- Keep the existing jobs and attempt to modify their configuration to match the job configuration created by the job-creation utility.

In any case, there is always going to be some overhead with maintaining legacy build jobs.

Setting up self-service job creation in a build tool

We have talked about the advantages of automatically creating build jobs, so now let's discuss how we can implement such a feature in a build tool. The principle should be the same for building applications for any language, such as Java, .NET, etc.

Introducing the job-creation build form

The build form is a means to pass information from the build tool directly to another feature or back-end scripts that perform the job-creation tasks.

Ideally, as much project information should be provided up front. This will reduce any later effort required to add extra configurations to newly created build jobs. Information such as project title, location of source code, and specific build switches are normally required. A push of the "Build" button creates the jobs just a few moments later.

Build-tool implementation

Two tools with which I have set up automated job creation are Hudson and AnthillPro 3.

Hudson/Jenkins

For clarity, I will talk only about Hudson but the information is equally relevant for Jenkins.

The Hudson CI tool has become popular among build managers looking to minimise their development resource budget and who do not want to be tied down with expensive licences. Hudson defines its build job-configuration in XML format. Typically, this XML file is accessible from the top-level page of the job URL, i.e. http://hostname/job-name/config.xml

Most of the sections in the config file are self-explanatory. Sections are added by plugins in the config file only when they are used in the Hudson GUI.

These config files can be made into templates by replacing specific project information in them with tokens. Several templates can be used to carry out different tasks, e.g. checkout

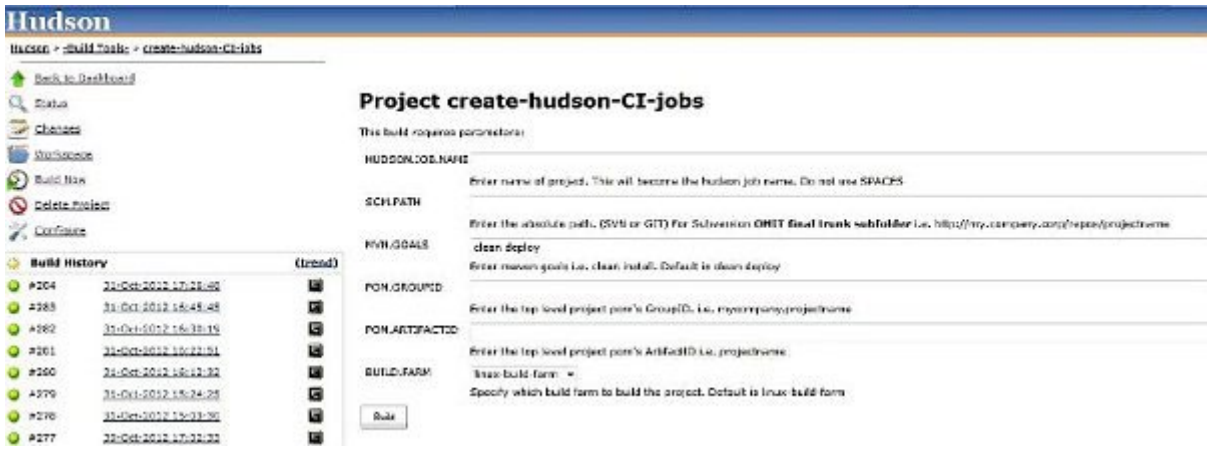


Figure 1: Hudson create-job form.

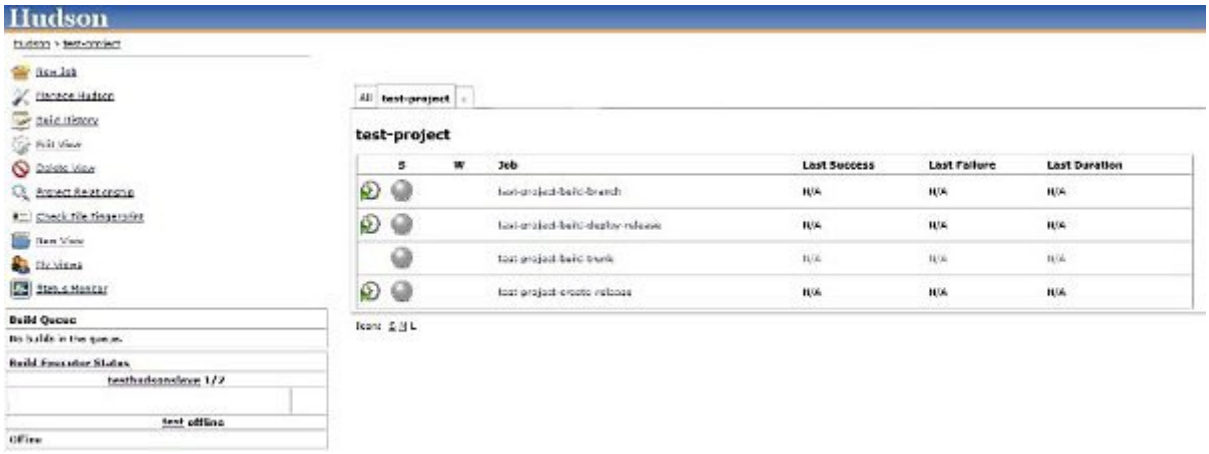


FIgure 2: Hudson job view.

Figure 3: AnthillPro job view.

of source code from different SCMs such as Subversion or Git.

Now we need to link the create-job build form to the scripts. Diagram 1 shows the workflow.

The build form passes the information to the scripts, which substitute the tokens in the templates with the relevant information. The config files are then uploaded to the Hudson tool via an API. More details about the API can be found in Hudson by adding the string "api" to the URL in the browser, i.e. http://hostname/job-name/api.

An example of a Hudson API create-job upload command is shown below:

curl –H Content-Type:application/xml –s –data @config.xml ${HUDSONURL}/createItem?name=hudsonjobname

(see Figure 1)

The jobs can be manually added to a new view. (see Figure 2)

There are a few points to note:

1 If the Hudson security section is configured, ensure that the "anonymous" user has "create job" privileges, otherwise the upload will fail authentication.

2 If Hudson tabs are used to view specific groups of jobs, it will not be possible to automatically place any of the newly created jobs under those tabs, except for the generic "all" tab (or any tab that uses a regex expression). The Hudson master-configuration file will require updating as well as a manual restart of the Hudson interface. Once the jobs are created, they can

```
<project name="createjob" default="createHudsonjobsConfigs">

<!-- Ant script to update hudson job config.xml templates to create new jobs in
  hudson -->

<!-- Get external properties from Hudson build form -->
  <property name="hudsonjobname" value="${HUDSON.JOB.NAME}" />
  <property name="scmpath" value="${SCM.PATH}" />
  <property name="mvngoals" value="${MVN.GOALS}" />

<!-- ...do same for rest of properties from the hudson form -->
...
...
  <property name="hudson.createItemUrl"
value="http://hudson.server.location/createItem?name=" />

<!-- Include ant task extensions-->
  <taskdef resource="net/sf/antcontrib/antlib.xml"/>

<target name = "createHudsonjobsConfigs" description="creates new config.xml file from
  input parameters">

<mkdir dir="${hudsonjobname}"/>

<!-- loop through each job template file replacing tokens in the job-templates with
  properties from Hudson -->
  <for list="CI-build-trunk,RC-build-branch" param="jobName">
      <sequential>
      <delete file="${hudsonjobname}/${configFile}" failonerror="false"></delete>
      <copy file="../job-templates/@{jobName}-config.xml"
  tofile="${hudsonjobname}/${configFile}"/>
      <replace file="${hudsonjobname}/${configFile}" token="$HUDSON.JOB.NAME"
  value="${hudsonjobname}"/>
      <replace file="${hudsonjobname}/${configFile}" token="$SCM.PATH"
  value="${scmpath}"/>
      <!-- ...do same for rest of tokens in the job template -->
      ...
      ...
      <antcall target="configXMLUpload">
        <param name="job" value="@{jobName}"></param>
      </antcall>
      </sequential>
  </for>
</target>

<!-- contruct the job config upload command -->
<target name="configXMLUpload">
  <echo>curl -H Content-Type:application/xml -s --data @config.xml ${hudson.
  createItemUrl}${hudsonjobname}-${job}</echo>
  <exec executable="curl" dir="${hudsonjobname}">
    <arg value="-H" />
    <arg value="Content-Type:application/xml"/>
    <arg value="-s" />
    <arg value="--data" />
    <arg value="@config.xml" />
    <arg value="${hudson.createItemUrl}${hudsonjobname}-${job}"/>
  </exec>
</target>
</project>
```

Code 1

```
001      <?xml version='1.0' encoding='UTF-8'?>
002      <project>
003        <actions/>
004        <description>Builds $POM.ARTIFACTID</description>
005      ...
006      ...
007        <scm class="hudson.scm.SubversionSCM">
008          <locations>
009            <hudson.scm.SubversionSCM_-ModuleLocation>
010            <remote>$SCM.PATH/trunk</remote>
011            <local>.</local>
012            <depthOption>infinity</depthOption>
013            <ignoreExternalsOption>false</ignoreExternalsOption>
014          </hudson.scm.SubversionSCM_-ModuleLocation>
015        </locations>
016      </scm>
017      <assignedNode>$BUILD.FARM</assignedNode>
018    ...
019    ...
020      <builders>
021        <hudson.tasks.Maven>
022          <targets>$MVN.GOALS</targets>
023          <mavenName>$MVN.VERSION</mavenName>
024          <usePrivateRepository>false</usePrivateRepository>
025        </hudson.tasks.Maven>
026      </builders>
027    ...
028    ...
029    </project>
```

Code 2: CI-build-trunk-config.xml Hudson job template.

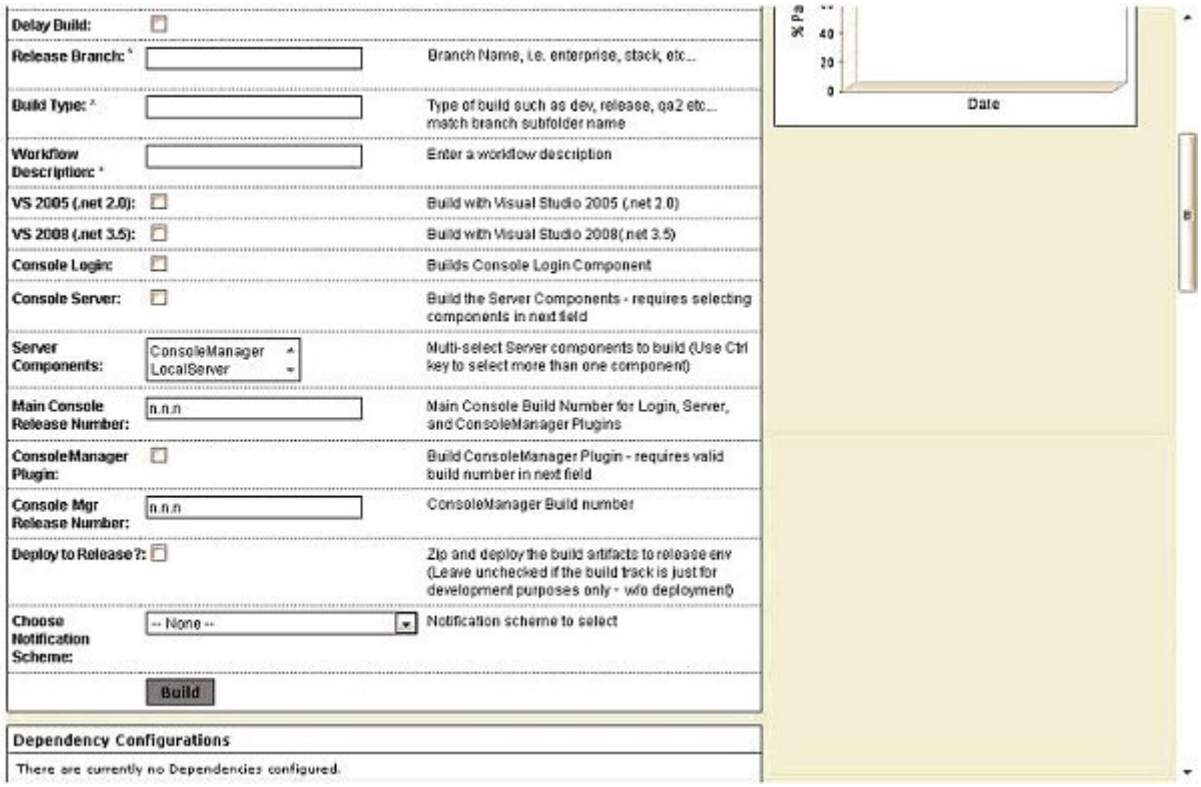


Figure 4: AnthillPro create-workflow form.

then be manually added to the relevant tab if so desired.

Scripts can be written in any language, such as Ant, Perl, Groovy, etc., so it is a relatively straightforward set of tasks to create scripts in order to carry out the steps outlined above.

A typical Ant script: Code 1

Code 2 is a typical tokenised Hudson job template:

AnthillPro

AnthillPro is a licensed build and CI tool from UrbanCode (now part of IBM) that provides the user almost complete custom control of build jobs via a GUI and an extensive API library.

The latest incarnation from UrbanCode is uBuild, which is essentially a cut-down version of AnthillPro and is geared primarily towards the CI building of products whereas its parent product was originally designed to manage an entire CD pipeline. uBuild does not have the API capability that AnthillPro has but it is possible to create a plugin that can do a similar thing.

Build jobs in AnthillPro are called workflows, which essentially are pipelines of individual job tasks executed in a specific order (in series or parallel).

AnthillPro's API-scripting language, BeanShell, is based on standard Java methods.

BeanShell scripts are stored within the AnthillPro graphic user interface and called from within a job task. The latest version of AnthillPro allows developers to custom-build plugins with any programming language.

An AnthillPro job task is created using API calls that essentially construct all the necessary functions of that workflow. The script can be particularly long, as every aspect of the workflow configuration has to be populated with data. Just like Hudson, a build form collects build-job information and executes a master workflow, calling on the BeanShell scripts to create the relevant workflow. (Figure 3 & 4)

A typical Java BeanShell script: Code 2

Note: It is also possible to create a self-contained package of Java BeanShell scripts such as a jar file, which is placed in a folder on the AnthillPro application server. Direct calls to classes within the jar file can then be made from a shell task within the application. This works well, especially because the scripts can be uni-tested before being applied in a live development environment.

```
001 private static Project createProject(User user) throws Exception {
002          // get the values from the buildlife properties
003         String groupId = getAnthillProperty(“GROUPID”);
004         String artifactId = getAnthillProperty(“ARTIFACTID”);
005         String build_farm_env_name = “linux-build-farm”;
006
007         String branchName = “branchName”;
008
009         Workflow[] workflows = new Workflow[3];
010         // Set up Project
011         Project project = new Project(artifactId + “_” + branchName);
012
013         // determine whether the project already exists and is active
014         boolean isProjectActive;
015         try {
016             Project projectTest =
017 getProjectFactoryInstance(artifactId + “_” + branchName);
018         isProjectActive = projectTest.isActive();
019       } catch (Exception err) {
020           isProjectActive = false;
021       }
022
023       if (!isProjectActive) {
024           project.setFolder(getFolder(groupId, artifactId));
025           setLifeCycleModel(project);
026           setEnvironmentGroup(project);
027           setQuietConfiguration(project);
028
029           String applications = getAnthillProperty(“APPS”);
030           // create project properties
031           createProjectProperty(project, “artifactId”, artifactId, false, false);
032           // set the server group for the workflow and add environment properties
033           addPropertyToServerGroup(project, build_farm_env_name, applications);
034
035           project.store();
036
037 ...
038 ...
039
040     // Create the CI Build workflow
041             String workflowName = “CI-build”;
042             workflows[0] = createWorkflow(
043                     project,
044                     workflowName,
045                     perforceClientFor(branchName, groupId, artifactId, workflowName)
     + buildLifeId,
046                     perforceTemplateFor(branchName, groupId, artifactId,
     workflowName),
047                     “${stampContext:maven_version}”,
048                     build_farm_env_name,
049                     “CI-build Workflow Definition”
050                     );
051
052             // add trigger url to commit-build branch
053             addRepositoryTrigger(workflows[0]);
054
055     // Create the Branch workflow
056             workflowName = “Branch”;
057             workflows[1] = createWorkflow(
```

```
058                     project,
059                     workflowName,
060                     perforceClientFor(branchName, groupId, artifactId, workflowName)
    + buildLifeId,
061                     perforceTemplateFor(branchName, groupId, artifactId,
    workflowName),
062                     “From ${stampContext:maven_version}”,
063                     build_farm_env_name,
064                     “Branch Workflow Definition”
065                     );
066
067    // Create the Release workflow
068             workflowName = “Release”;
069             workflows[2] = createWorkflow(
070                     project,
071                     workflowName,
072                     perforceClientFor(branchName, groupId, artifactId, workflowName)
    + buildLifeId,
073                     perforceTemplateFor(branchName, groupId, artifactId,
    workflowName),
074                     “${stampContext:maven_release_version}”,
075                     build_farm_env_name,
076                     “Release Workflow Definition”
077                     );
078
079 ...
080 ...
081
082         } else {
083             // project already exixts
084             String msg = “Project “ + artifactId + “_” + branchName + “ already
    exists - check project name and pom”;
085             throw new RuntimeException (msg);
086         }
087         return project;
088     }
```

Code 3: Sample CreateProject Java BeanShell script.

Other tools

As long there exists the ability to create jobs via an API or a template feature, the implementation described here can be similarly applied to other build tools such as Go, TeamCity, and Bamboo to name but a few.

Ultimate build-tool agility: Implementing the lean build-tool philosophy

An advantage mentioned earlier in this article is automatically creating on-demand CI projects and tearing them down after completion. This would significantly reduce the amount of build jobs in the build tool.

I have yet to implement an agile, lean method to automatically create and tear down jobs on a build tool, so I cannot detail the technical challenges to implement such a task. However, everything that has been discussed here can lead to implementing such a solution. In fact, Atlassian's Bamboo build tool can detect branch creation from a main baseline and automatically create an appropriate job to build it.

Summary

We have discussed how to set up facilities to create other jobs that form part of the continuous delivery strategy for a business:

- An automated approach to creating jobs in a build tool should be adopted as part of everyday software development.
- Automatic creation of jobs saves time and allows the developer to get on with more important tasks.
- Consistent build-job configuration leads to easier job maintenance and drives consistent development practices across a company.
- Automatic job creation can ultimately lead to a true agile adoption of software development right down to the build-tool level.

Service Discovery

David Dawson is CEO of Simplicity Itself, a UK-based consultancy that specialises in understanding and building simple software that thrives on change. After leading Simplicity Itself in the building of several large microservices systems and applying DevOps in their construction and design, David now applies his experience to the development of Muon, a new open-source programming model for microservices.

When building microservices, you have to distribute your application around a network. It is almost always the case that you are building in a cloud environment, and often using immutable infrastructure. Ironically, this means that your virtual machines or containers are created and destroyed much more often than normal, as this immutable nature means that you don't maintain them.

These properties together mean that your services need to be reconfigured with the location of the other services they need to connect to. This reconfiguration needs to be able to happen on the fly, so that when a new service instance is created, the rest of the network can automatically find it and start to communication with it. This process is called service discovery.

This concept is one of the key underpinnings of microservice architecture. Attempting to create microservices without a service-discovery system will lead to pain and misery, as you will in effect be working as a manual replacement.

As well as the standalone solutions presented here, most platforms, whether full PaaS or the more minimal container managers, have some form of service discovery.

There are currently several key contenders to choose from: ZooKeeper, Consul, etcd, Eureka, and rolling your own solution.

ZooKeeper

ZooKeeper is an Apache project that provides a distributed, eventually consistent, hierarchical configuration store.

ZooKeeper originated in the world of Hadoop, where it was built to help in the maintenance of the various components in a Hadoop cluster. It is not a service-discovery system per se, but is instead a distributed configuration store that provides notifications to registered clients. With this, it is possible to build a service-discovery infrastructure although every service must explicitly register with ZooKeeper, and the clients must then check in the configuration.

Netflix has invested a lot of time and resources in ZooKeeper so a significant number of Netflix OSS projects have some ZooKeeper integration.

ZooKeeper is a well-understood clustered system. It is a consistent configuration store, so a network partition will cause the smaller side of the partition to shut down. For that reason, you must choose whether consistency or availability is more important to you.

If you do choose a consistent system such as ZooKeeper for service discovery, you need to understand the implications for your services. You have tied them to the lifecycle of the discovery system and have exposed them to any failure conditions it may have. You should not assume that "consistent" means free from failure. ZooKeeper is among the older cluster managers, and consensus (pun intended) is that its implementation of master selection is robust and well behaved.

If you do choose to use ZooKeeper, investigate the Netflix OSS projects, starting with Curator, and only use bare ZooKeeper if they don't fit your needs.

Since ZooKeeper is mature and established, there is a large ecosystem of good-quality (mostly!) clients and libraries to enrich your projects.

Consul

Consul is a peer-to-peer, strongly consistent data store that uses a gossip protocol to communicate and form dynamic clusters. It is based on the serf library.
It provides a hierarchical key-value store in which you can place data and register watches to be notified when something changes within a particular key space. In this, it is similar to ZooKeeper.

As opposed to ZooKeeper and etcd, however, Consul implements a full-service discovery system in the library, so you don't need to implement your own or use a third-party library. This includes health checks on both nodes and services.

It implements a DNS server interface, allowing you to perform service lookups using the DNS protocol. It also allows clients to run as independent processes and can register/monitor services on their behalf. This removes the need to add explicit Consul support to your applications. This is similar in concept to the Netflix OSS Sidecar concept that allows services with no ZooKeeper support to be registered and be discoverable in ZooKeeper.

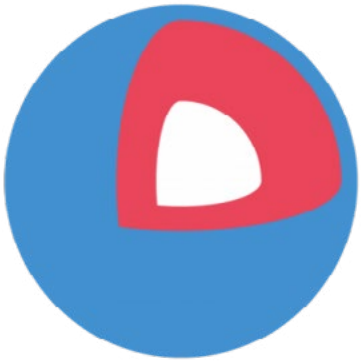
Deployment-wise, Consul agents are deployed on the systems that services are running on, and not in a centralised fashion.

This is a newer product and one that Simplicity Itself likes a lot. We recommend it if you are able to adopt it.

As with the other strongly consistent systems in the list, care must be taken that you understand the implications of adopting it, including understanding its potential failure modes.

If you would like something similar that chooses availability rather than consistency, investigate the related serf project. The serf library serves as the basis of Consul, but has chosen different data guarantees. It is nowhere near as fully featured, but can handily survive a split-brain scenario and reform afterwards without any ill effects.

etcd



As an HTTP-accessible key-value store, etcd is similar in concept to ZooKeeper and the key-value portion of Consul. It functions as a distributed, hierarchical configuration system, and can be used to build a service-discovery system.

It grew out of the CoreOS project, which maintains it. The tool recently achieved a stable major release.

If you are primarily using HTTP as your communication

mechanism, etcd can't be easily beaten. It provides a well-distributed, fast, HTTP-based system, and has query and push notifications on change via long polling.

Eureka

Eureka is a mid-tier load balancer built by Netflix and released as open source. It is designed to allow services to register with a Eureka server and then to locate each other via that server.

Eureka has several capabilities beyond the other solutions presented here. It contains a built-in load balancer that, although fairly simple, certainly does its job. Netflix states that they have a secondary load-balancer implementation internally that uses Eureka as a data source and is much more fully featured. This hasn't been released.

If you use Spring for your projects, Spring Cloud is an interesting project to look into. It automatically registers and resolves services in Eureka.

Like all Netflix OSS projects, Eureka was written to run on the AWS infrastructure first and foremost. While other Netflix OSS projects have been extended to run in other environments, Eureka does not appear to be moving in that direction.

We at Simplicity Itself very much like the relation between client and server in the Eureka design, as it leaves clients with the ability to continue working if the service-discovery infrastructure fails.

Server-wise, Eureka has also chosen availability rather than consistency. You must also be aware of the implications of this choice as it directly affects your application. Primarily, this manifests as a potentially stale or partial view of the full data set. This is discussed in the Eureka documentation.

Eureka is now the quickest to get started with Spring projects because of the investment the Spring team has made in adopting the Netflix OSS components via the Spring Cloud subprojects.

Roll your own

One major point to note in the above systems is that they all require some extra service-discovery infrastructure. If you can accommodate that, it is better to adopt one or more of the systems above for service discovery.

If you can't adapt to that, you will have to create your own discovery solution within your existing infrastructure.

The basis of this will be that:

- Services must be able to notify each other of their availability and supply connection information.
- Periodic updates to the records should strip out stale information.
- It must easily integrate with your application infrastructure, often using a standard protocol such as HTTP or DNS.
- It must notify you of services starting and stopping.

Building your own discovery service should not be taken lightly. If you need to do it, we at Simplicity Itself recommend building a system that values availability rather than consistency. These are significantly easier to build, and make it more likely that you will build something functional.

We recommend that you use some existing message infrastructure and broadcast notifications on service status. Each service caches the latest information from the broadcasts and uses that as a local set of service-discovery data. This has the potential to become stale, but we've found that this approach scales reasonably well and is easy to implement.

If you do require consistency, using some consistent data store could serve as the basis for a distributed configuration system that can be used to build service discovery. You will also want to emit notifications on status changes. You should realise, though, that building a consistent, distributed system is exceptionally hard to get right, and very easy to get subtly wrong.

Overall, we do not recommend this strategy but it is certainly possible.

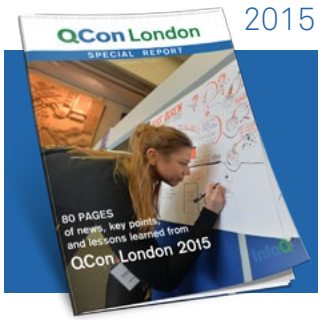
PREVIOUS ISSUES

27

Technical Debt and Software Craftsmanship

This e-mag brings together a number of authors who have addressed the topic on InfoQ and suggests ways to identify, monetize, and tackle technical debt in your products.

QCon London 2015 Report



26

This eMag provides a round-up of what we and some of our attendees felt were the most important sessions from QCon London 2015 including Microservices, Containers, Conway's Law, and properly implementing agile practises.

Next Generation HTML5 and JavaScript

25

Modern web developers have to juggle more constraints than ever before; standing still is not an option. We must drive forward and this eMag is a guide to getting on the right track. We'll hear from developers in the trenches building some of the most advanced web applications and how to apply what they've learned to our own work.

Mobile - Recently New Technology and Already a Commodity?

24

This eMag discusses some familiar and some not too familiar development approaches and hopefully will give you a helping hand while defining the technology stack for your next mobile application.