Immutable Infrastructure Calls for Immutable Architecture: Deploying a Changeless Architecture in the Cloud

Anders Mikkelsen
Netcompany
Oslo, Norway
anderm@netcompany.com

Tor-Morten Grønli
Kristiania University College
Mobile Technology Lab
Department of Technology
Oslo, Norway
tor-morten.gronli@kristiania.no

Rick Kazman
University of Hawaii
Shidler College of Business
United States of America
kazman@hawaii.edu

Abstract

With the advent of cloud computing and the concept of immutable infrastructure, the scaling and deployment of applications has become significantly easier. This increases the possibility of "configuration drift" as an operations team manages this cluster of machines, both virtual and actual. In this paper we propose a revised view on configuration and architecture. We propose that software deployed on a public or private cloud should, to the furthest possible extent, be immutable *and source controlled. This reduces configuration drift and ensures no configuration problems in production as a result of updates or changes. We will show an example of a software project deployed on Amazon Web Services with an immutable Jenkins setup which manages updating the whole cluster and is self-regenerating. We will also discuss how this lends itself naturally to interoperability between clouds, because of the infrastructure-agnostic nature of this approach.*

1. Introduction

The thing about perfection is that it is unknowable, it's impossible, but it's also right in front of us, all the time! - Kevin Flynn

Traditional software systems are constructed with architectures designed for current times and planned with architectures for the near future. However, software development is a rapidly and ever-evolving industry and is continuously influenced by a huge number of factors such as changes in requirements, new versions of frameworks and third-party components, and technical debt [1], to name but a few. Our response to development and maintenance in such a changing environment is to no longer "go with the flow", but rather to reject uncertainty and embrace immutability.

Immutability is defined as an entity being unchangeable, or changeless. This makes such entities eternally knowable, as they will never change during their lifetime and, even when completely destroyed, the information about their composition is still perfectly valid. Mutable entities or, in the case of software architecture, systems, are in their essence unknowable. Unknowable systems are naturally difficult to debug, manage and maintain as you can never know with any certainty that what you are looking at is the correct base for implementing corrections. Furthermore, as you implement a correction to an obvious problem, that problem might actually be an accident due to an undocumented change in the architecture.

To develop software systems with a high degree of productivity and predictability, we are need to be able to rapidly identify how to solve any emergent problem and we need to be confident this change will not break some other functionality or system quality. Several approaches within the domain of software testing partially address this issue, but do not provide a solution. Further to this, how can we continuously and rapidly develop a software project, and ensure that the production version is immutable and thus knowable at all times?

The primary challenge here is mapping all variables and handling them accordingly. Current approaches focus on managing the architecture according to a known state and provisioning virtual machines (VMs) with necessary dependencies. This is achieved with Continuous Delivery where the deployment of the system to production is fully automated and continuous [2, 3]. We believe this existing approach does not solve the problem of the system being free from configuration dependencies, but it implements mutable tools that are constantly altering the system to achieve the closest approximation to the desired state based on changes to the tools themselves.

"Desired" is however neither known nor certain. Our vision and our approach revolves around continuously updating the entire architecture with new complete images with all dependencies whenever a change is

URI: https://hdl.handle.net/10125/60142
ISBN: 978-0-9981331-2-6

called for, including any tools for build, deployment or operations. This imaging is achieved by using containerization technology to ensure that the respective component of the architecture is agnostic with respect to the underlying infrastructure and its configuration. For this research we have employed the *Docker* container system as a contextualized base for implementation. Instead of attempting to manage state, the system will always be trackable to source-controlled images. In essence we *replace* the state instead of updating or modifying it. Containerization of the entire architecture is key to this approach and the architecture running in production is always knowable because it is always unchangeable in its current iteration. To the best of our knowledge this stateless approach to software architecture has not previously been addressed by research.

2. Related Work

Cloud computing has received considerable attention in the software engineering community and has been embraced by the industry as the preferred emerging platform for software hosting. Cloud computing utilizes three major distribution models; Software as a Service (Saas), Platform as a Service (PaaS) and Infrastructure as a Service (IaaS). Here we concentrate on SaaS with applications delivered as services over the Internet and the hardware and systems software in the data centres providing these services. The idea is built around an economy of scale, where the ultimate goal is to provide more resources, better scalability and flexibility for less money [4]. Cloud computing indicates a movement away from computing as a product that is owned and towards computing as a service [5]. Service in this context is a concept that deals with the utilisation of reusable fine-grained components across a vendor's network, and cloud-based services are usually billed using a pay-per-use model [6]. Further to this, cloud computing is facilitating a focus on the software development process and architectural optimization with is current features for infrastructure and deployment abstraction.

Current deployment and maintenance of a software architecture is managed in three phases. Build, Deployment and Operations. This rings true both for traditional management and the new paradigm of DevOps [2]. DevOps is an abbreviation of Development and Operations, which seeks a closer collaboration between the two camps, and more crucially, treating infrastructure and architecture as *code* to automate the previously time-consuming and manual provisioning process [2, 3].

Continuous Integration is often used as an umbrella for both CI and Continuous Deployment (CD). CI is, in its most basic implementation, the process of continuously building and testing the code of a software project with unit and/or integration tests [7]. Usually for the purpose of having a small iteration cycle from implementation to product, and having a broad suite of tests to verify expansions to the product to not break old features. In this sense it also becomes a programming paradigm in which no code is expected to be submitted without proper tests that allow regression testing in all subsequent builds. If the test suite is sufficiently broad and of a high quality, confidence is built in the team for pushing code changes and new implementations can be delivered quickly to the end user [8]. Test Driven Development (TDD), which is a programming paradigm for writing tests before producing the implementation code, is closely linked to the field of CI. As shown by Amrit and Meijberg [8], it is not a prerequisite because there is not consensus whether this approach increases or decreases performance, but it helps to explain the benefits of a test oriented approach to delivering a product. The typical implementation of CI is hosting the code of a project on a distributed repository like GitHub or BitBucket, running a CI server like Jenkins or Travis CI, and having it react to all changes on a dedicated release branch. CD is quite simply the process of having delivered every successful build in a CI process to production. For example automatically replacing running backend services or uploading a revision to Google Play Store or the App Store. The collective idea of CI and CD combined is to continuously improve a product, without breaking old functionality, and having an uptime of as close to 100% as possible [8].

The recognized challenges in maintenance of large scale architectures are to a great extent context dependent and there has been some recent research addressing the impact of continuous deployment together with DevOps on the architecture [9]. Keeling [10] highlights that this impact is dependent on the software design and has different magnitude for lightweight architectures and flexible designs. Keeling [10] highlights further that benefits of DevOps are achieved first when the typical architectural quality attributes security, availability, reliability etc. are consciously addressed. These issues present a renewed interest in fundamentals for software architecture and combined with cloud computing distribution, new issues to address come to light. Bellomo et al. [11] note that DevOps, continuous integration, and continuous deployment can not be achieved at scale without architectural support, and also identify a set of architectural tactics—recurring design primitives—to achieve these goals.

Many of the services needed to support this new paradigm are often included in public clouds. While having a low learning curve, and at times high configurability these services undeniably lead to a dependence on the cloud provider and have potentially long correction times if fundamental bugs are uncovered, because you do not have direct access to the infrastructure. Often, you can also experience issues with API reliability and reporting delays, as shown by Zhu et al. [12].

In the example by Zhu et al. [12], they discuss the pros and cons of *heavily* and *lightly baked* images for deploying software. Heavily baked is a clearly defined Amazon Machine Image, with only minor post instantiation actions while lightly baked includes recipes for post instantiation configuration (through the Chef framework). While this is a sound comparison for deployment practices using only virtualization, it fails to consider the architecture itself being containerized on top of a VM. This would possibly lead to a slightly longer build time compared to a lightly baked approach. This is due to the fact that all configuration must be included into the image, but post build configuration will be eliminated and deployment time will be much faster. In addition, you could avoid most of the API reliability issues by simply rerunning build and deployment as everything is immutable and can crash at any time. In this sense, we can get the best of both worlds by virtualizing the infrastructure and containerizing the architecture and configuration.

When you consider cloud technologies and containerization, you cannot escape talking about micro-services. It is also a difficult subject to define. One of the most concrete presentations outside of academic circles came from Lewis and Fowler in 2014 [13]. As understood by practitioners and academics alike the three main pillars of MSA are those of improving speed of change, small cohesive units and scalability [14, 15, 16, 13].

Whether Microservices Architecture (MSA) is a style or a constrained implementation of Service Oriented Architecture (SOA) is a contested point [15, 17]. Constraints in this context means putting less emphasis on SOA concepts such as the enterprise service bus. Some would say it is a best-practice approach to SOA, while others like Shadija et al. would say it is a separate paradigm [16, 18, 19, 15, 17]. There are also presentations among practitioners that embrace both ends of this spectrum, by defining multiple implementations of MSA. Some closely resemble SOA, in a more fine-grained form, others like Mulesoft, don't [14]. These differing opinions lead to there not being a clear definition of MSA or even microservices themselves in an academic context [19, 15, 16, 18], however there appears to be some universal constants concerning microservices that practitioners agree upon.

- They are strongly tied to the practice of Continuous Integration (CI) [8]
- They should all be logically small (High Cohesion in Features) [15, 13]
- They each run in their own process [13]
- They communicate with each other via a lightweight interface, often a REST API [16, 18]

Kang et al. [20] discuss challenges with containerizing micro-services in a cloud environment and summarizes the three primary challenges as minimizing cross-configuration of services, maintaining state of running services and providing safe access to host resources.

Lenk et al. [21] have done a survey, testing virtual machine interoperability on an operating system and application level. They introduce a testing method called Testing Interoperability on Operating System and Application (TIOSA). They continue with a detailed execution process for testing and verifying interoperability with meticulous testing on both the OS and application level. The results and conclusions of this paper shows that VM interoperability is hard, and more importantly that it is not a reality in clouds yet. Mei et al. [22] compare cloud computing to service computing and pervasive computing. From this comparison we can see that clouds vary on a fundamental level by being abstract in both location, context and granularity. Being that the cloud is such an undefinable environment, we should ensure that our architecture is as agnostic to the cloud as possible, including any services it provides.

3. Design and implementation

The system being analyzed as an example for this paper is a backend system for a multi-platform mobile app used for streaming digital content from multiple providers. The system is designed to automatically build and redeploy a multitude of different apps based on the content and configuration of individual users of the different providers, in a single aggregated format. The various services of the backend architecture is constructed with Vert.x [23]. All services are clustered using the default clustering framework for *Vert.x*, *Hazelcast*. A simple content management system is created in *Ruby on Rails* and continuous delivery is managed by a Jenkins build server. All application components are containerized using Docker. The main focus of the example is the continuous delivery component, but the entire architecture is designed with immutability in mind.

The architecture is deployed on Amazon Web Services (AWS) in autoscaling groups. An AWS Auto Scaling grouping is a set of VMs that are all created from the same launch configuration and can be scaled up and down in instance count, based on a variety of metrics. All services have their own dedicated cluster of autoscalable virtual machines (VMs) spread across three availability zones. Data storage is handled by *Amazon DynamoDB* [24]. The Key/Value store is handled by *Amazon Elasticache*. All connections and configurations of these services are handled in code.

3.1. Prototype architecture

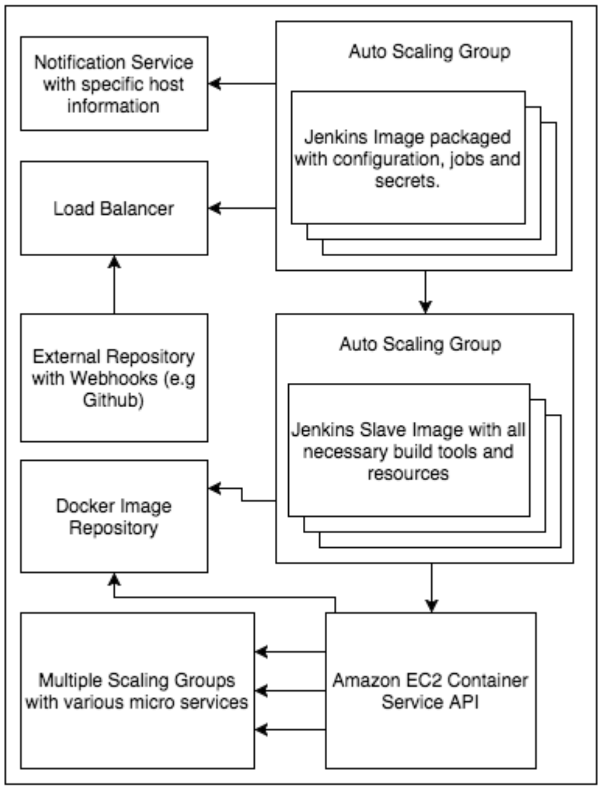
Notification Service with specific host information
Auto Scaling Group
Jenkins Image packaged with configuration, jobs and secrets.
Load Balancer
External Repository with Webhooks (e.g Github)
Auto Scaling Group
Jenkins Slave Image with all necessary build tools and resources
Docker Image Repository
Multiple Scaling Groups with various micro services
Amazon EC2 Container Service API

Figure 1. Architecture of implementation

Figure 1 shows the relationships of the logical architectural components for this implementation of an immutable architecture. It is primarily deployed in an AWS environment, supported by source code repositories hosted on GitHub, and Slack for notifications and messaging. Each box in the diagram represents a component, such as Amazon EC2, or a logical service component such as a load balancer or a notification service. As we can see, our build servers are running Jenkins master nodes inside an AWS auto scaling group. We are also running Jenkins slaves within a scaling group.

The Master nodes have a unified Docker image while, conversely, slaves can be a multitude of different configurations built into separate Docker images. For this architecture it consists of one slave configuration for building Android apps, and a separate configuration for building java projects and the Jenkins master and slave configurations. The current Dockerfile for this prototype builds a generalized slave with preinstalled tools like Maven, Gradle, Java, Selenium, etc. For a more advanced production ready setup you could further build smaller images for specific build tools, or use other prebuilt images for these purposes. You could have a Node image, a Maven image, and a Gradle image f.ex. Then you could spin up a new container for each part of the build process and that way parallellize your builds without having to worry about race conditions, mutex etc. on slaves running multiple stages in parallell because in effect you have isolated systems building the different parts of your application.

The Android build slave will, based on input parameters, download and install a specific version of the Android SDK on the Docker image. This gives us the opportunity to store a wide range of versions as we go along. This eats up a lot of build time, but gives us startup times of maybe a couple of seconds to start an actual android build. We don't need to download particular packages on the fly during installation, as we spent that resource of time during build instead of runtime.

Because our Jenkins configuration is immutable we can simply scale up masters to handle large numbers of jobs because nothing stored on each master node is of any non replaceable importance. Job execution is handled by incoming webhooks that spread to any available master through the load balancer. Job status is pushed to an external notification service, in our case the industry standard Slack tool, but in theory any notification service responding to webhooks can be used. A possible issue here can be ”lost jobs” if a master tracking a particular job suddenly dies. This prototype does not handle that case, but the plan in a future iteration is having a service that oversees GitHub webhook activation and completion down the line. This can be a homegrown solution, or a cloud offering like AWS CodePipeline. CodePipeline is an offering from Amazon for automating various pipelines, most commonly Continuous Integration/Deployment scenarios.

On the application side of things the Amazon EC2 Container Service (ECS) is leveraged as a manager for the replacement of immutable architectural components as the entire configuration can be kept in source code, alongside the business logic for the services. Should a particular service require more resources it can be updated in source and then replaced as a new immutable configuration through the ECS. Services such as this with repository based configuration is critical for implementing immutable architectures. Further, the ECS will manage all containers of individual services. Replacing any faulty ones with fresh containers, scale up and down etc.

Multiple other cloud services are integrated with to manage the live cycle of a service, and all of these have their individual configuration in the source repository of their parent service. One such example is the use of AWS CloudWatch alarms for managing different parts of a service's life cycle based on internal AWS metrics and other external metrics. Having the service configuration in the source repository turns it into immutable configuration as the build job for these services would then redeploy any modified alarms up deployment of the parent service. So any commit in the repository will directly correlate to a particular version of an alarm. Preferable for complete history you would replace any dependent or supportive services like this alongside the parent service and tag it with a new version to keep as much as possible of the architecture immutable.

3.2. Implementation

Implementing this immutable configuration comes with a set of three major challenges. The obvious first challenge is to package services in such a way that no configuration needs to be done after instantiation of the image. For Jenkins this is easy as most configuration are possible to override during initialization. Where this is not possible, most often with third party plugins and manual configurations, we must use scripts and other tools to manually configure our image. This adds complexity but is also testable. We can run tests as part of the image build process and, in the spirit of DevOps, simply break the build whenever we encounter a state that is unexpected. In our development environment we can also easily start our images and do integration tests in a production-like environment. For this particular example we have used Docker Compose. With the Docker Compose tool we can set up multiple services and configure them to have whatever state we want, so we can emulate a mini cloud right on our development machine.

Docker is used as an example here but any other containerization technology may be used for this purpose. Most cloud providers have support for at least one type, most commonly Docker, as a main offering. Whatever abstraction level you would want to operate on, whether it is bare metal, virtual machines or something like the AWS Elastic Container Service which has the possibility of removing infrastructure configuration entirely, containers will allow you to always replace and never modify the architecture.

The second challenge comes with handling the necessary secrets for these services, like ssh keys, username/password combinations, API keys etc. In this example with Jenkins, secrets are initialized on first startup if they do not already exist. The current solution is generating the secrets in a development environment and including these into the image. These secrets are available on the resulting instance of Jenkins either way so there will be no difference in security state compared to by natural initialization.

For production environments you could store these secrets in an external store like an encrypted AWS S3 bucket, or another store provided by your environment, but have the information about fetching said secrets clearly defined in the source. For example consider the object key for an item in a bucket on S3. Following the principles of immutable architecture (IA) any new key, would be a new object, which would need to be updated in any relevant services utilizing it. This way we retain the IA. A future iteration could also have a build step that automatically recreates secrets for every build to increase security above what would be provided by a standard Jenkins instance restarting.

The third major challenge with this approach is setting up connection parameters for the Jenkins instances. Each Jenkins instance must be able to communicate with any slaves it provisions and they must be able to communicate with the correct master. This needs to be built into the commands used to run the relevant slave. Solutions to this challenge will vary depending on the relevant cloud being used. On AWS metadata about the underlying instance is available by polling the instance itself. This is a brittle approach, as it is reliant on AWS. This works for the context of the scope of this paper, but more robust cross-cloud implementation would have a base image that is agnostic to its environment. We would then build on that by defining a global environment variable in the base image that we can easily override in an AWS image. We could do the same for any other provider.

If the provider does not have this information available in a fashion resembling this approach we would have to be more creative with scripts to collect the necessary metadata. In our specific case we also need to configure the individual IPs of hosts for the Slack plugin. This is needed so that we can display specific host information for debugging on any failed job. The complexity or impact on performance of this will vary greatly with the underlying infrastructure. In an on-premises configuration this metadata might not be easily accessible and might require port scanning or something similar to expose it. This will create a slight performance penalty in the deployment of the individual components, but won't impact otherwise.

4. Results

Describing the entire architecture in detail is beyond scope of this example, but suffice it to say that it is designed around the same principles. All services are clustered and intercommunicate freely, by using the same metadata for describing public and private IPs, cluster ports and so forth. When everything is packaged in this way, we can push a new version of our Jenkins master image on demand. This will rebuild itself, and further rebuild the entire clustered architecture to the new version, even during load as destruction and instantiation of new docker containers is trivial.

This short explanation shows how this approach increases initial development time, because we must handle all configurable variables at build time as the final image is meant to be changeless. The biggest payoff however, is that images are prebuilt and configured when instantiating new instances, thus greatly decreasing deployment time compared to post instantiation configuration tools like Chef [25], simply because there is no post instantiation configuration required.

4.1. Scenario 1: Adding or updating a job on the Jenkins Server

When adding, or updating, either a periodic job or a job for building and deploying a service/application in the architecture we would normally develop and test a new job, or update an existing one, on a development environment running a Jenkins server. Either on a continually running environment or on-demand. With this new approach we can simply modify the job, rebuild the image and redeploy it on our local machine emulating the complete production environment using Docker Compose. This way we shed the need for a continuously running development or stage environment with all its incumbent costs. Using Compose and other Docker images we can also recreate the environment as close to production as at all possible, by recreating services we depend on in sidecar containers that either mock that service or provide a local variant.

When we are satisfied with the performance of our new job and it has passed testing in the local production environment we can push the new version to source control and our live Jenkins server will rebuild itself and redeploy a new version of itself with the new job available, in exactly the same configuration which we were running in our local production environment. To ensure that no currently running jobs from any Jenkins Master are dropped, a production implementation of this architecture should manage http routing to the new environment and then drain the old environment before removing it entirely.

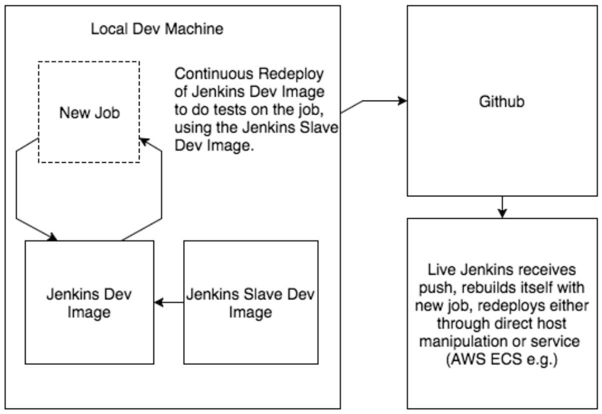
Local Dev Machine
New Job
Continuous Redeploy of Jenkins Dev Image to do tests on the job, using the Jenkins Slave Dev Image.
Github
Jenkins Dev Image
Jenkins Slave Dev Image
Live Jenkins receives push, rebuilds itself with new job, redeploys either through direct host manipulation or service (AWS ECS e.g.)

Figure 2. Adding/Updating a Jenkins Job

4.2. Scenario 2: Updating configuration of architectural or application components

When updating components of the architecture itself like databases, configuration docs, caches or load balancers, or application components, the process will be exactly the same, as everything is containerized. Using databases as an example we could have this further separated into database containers and data containers which hold the actual data of the database. All the data will be shared into database containers. If we need to update or change the database, say from version 1.0 to 1.2 we can modify the Docker file for our database container to install the correct version and then rebuild this to a new image version.

We can then do testing of this on our local production environment before redeploying this into the live production environment. On this redeployment the old database containers will be trashed and replaced by the new image versions. In other words, we retain the immutability of our architecture as no modification has been done. Parts of the architecture have been destroyed and replaced. So instead of modifying the state we have entirely replaced it. To give an analogy: if this were a SQL database, the analogy would be that we do not adjust the schema of a running database to reflect a change in a model, we would replace the entire database with a new one with a new schema, that has no explicit connection to the old version. It would be as if the old one never existed. It is simply a reflection of a specific point in time represented by a commit in a repository.

Because we define all versions of an architecture as immutable we can run both versions concurrently, to do a green/blue deployment. Knowing that no change has been performed in runtime to adapt to changing conditions, we can be confident that the changes we introduce will not introduce unexpected bugs, because we have thoroughly tested the transition from an old to a new version locally already. This obviously creates a higher demand on resources in the interim period, but also facilitates a clean rollback when necessary. When a rollback is performed the new version will be removed and the old version is retained. Because all versions are immutable we can be confident that the end result of a rollback is always directly relatable to a specific commit in source code, and not a specific commit in source code plus a list of potentially idiosyncratic changes done at runtime. This mutable state introduced by runtime changes might not be correctly reset after such a rollback.

In this approach we can see how we move operations into the DevOps mindset. Now developers and operations will work in the same manner and cooperation on images will be much more natural than planning upgrades of running hosts before redeploying a container. The operations team can use the prepared application image whenever an architectural upgrade is ready.

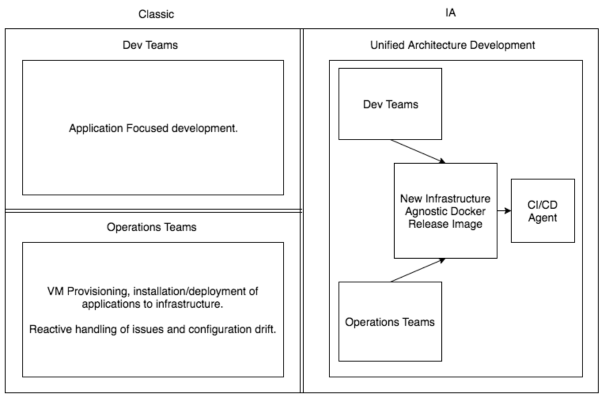
Classic
IA
Dev Teams
Application Focused development.
Operations Teams
VM Provisioning, installation/deployment of applications to infrastructure.
Reactive handling of issues and configuration drift.
Unified Architecture Development
Dev Teams
New Infrastructure Agnostic Docker Release Image
CI/CD Agent
Operations Teams

Figure 3. Classic / IA Workflow

5. Discussion

In this section we will discuss the implications of configuration drift and the benefits associated with containerizing and cloud interoperability. We will separately address the issues of eliminating configuration mutation, further discussing having image versions under version control, and what the impacts on interoperability are.

5.1. Avoiding configuration drift by eliminating configuration mutation

As shown in our Jenkins example we have eliminated configuration mutation by imaging the entire installation of Jenkins and only allowing modification in a development environment. This way we can thoroughly test new configuration options, modules and plugins and deploy a new state whenever the new version is ready for production. As mentioned by Ebert et al. [2] configuration tools are great enablers for DevOps as they reduce complexity and allow reproduction of the production system on development machines. With imaging the configuration tools in question we can achieve the same thing, just for the configuration tools themselves, and thereby the entire architecture. Any maintenance will also be done on a development environment and pushed to the Jenkins master cluster for regeneration. This way we can ensure that whatever is in source control is always what we will find in the live version. Depending on the options provided by the cloud provider one should always try to find the most optimal way of storing the Jenkins secrets.

This approach of deploying configuration tooling will decrease the long tail and unreliability described by Zhu et al. [12] and potentially eliminate it if we expand this process to the entire architecture. If all the different dependencies for different services are actually just layered docker images we have a single point of reliability to worry about which is the repository API [12]. This means the deployment tool in question, be it Jenkins, Chef [25] or any other, will effectively just build and spin up predefined Docker images in a structured manner instead of installing and configuring anything. We can now test the Docker images at build time and just retry any builds that fail on a single host instead of managing and doing post instantiation processes on multiple hosts.

5.2. Containerizing everything and having all image versions in source control

In our opinion the challenges presented by Kang et al. [20] can be mitigated by taking containerization further. In the example of storage, they consider that an example might store user profiles on disk. This ”disk” might just be a docker volume running on a separate docker image as part of the same overall application. This volume will be persisted between startups by the docker engine. If the VM running the containers die, these containers can again have functionality for synchronizing from the cluster without considering the new VM. The same can be said for service discovery. They describe using a ”sidekick” process for handling registration and discovery, which increases the complexity and cross-configuration of the image. They should rather have a secondary docker image that handles service registration and discovery on behalf on the application image and link them. Creating an application interface between them will also make it possible to interchange service discovery frameworks if necessary, e.g. using a different cloud.

We want our application to be completely agnostic to any configuration outside of the container we control. This introduces challenges with managing secrets, environment variables and technologies that require host parameters like public/private IP. An example would be *Hazelcast* for our *Vert.x* cluster. These are solvable challenges but they nonetheless add to development time. The upside when the basics are configured is an image that is self-contained and agnostic to its hosting environment. Much like the light baking described by Zhu et al. [12] we can reduce the image as much as possible to limit configuration and we do not have to worry about installing packages needed for other containers running on the same host as these images are also self-contained. The key difference is that the baking is on the docker layer instead of the VM layer. This also ensures the immutability of the original VM, as it will only be defined by the VM image it is spawned from and never changed.

5.3. Using this technique for cloud interoperability

By isolating our software architecture as much as possible from the underlying infrastructure we facilitate for cloud interoperability. The reason for this is that we can construct basic docker images of all application services and dependencies, and then simply create cloud specific docker images on top of this to easily deploy anywhere [3]. For this to work we must containerize everything. From build, through deployment and even operations. The crux of interoperability and probably the most challenging aspect will be the containerization and scaling of the database(s) of choice and other stateful services across clouds [20].

The conclusions in the paper by Lenk et al. [21] are sobering and clearly show the great challenges that lie ahead for VM conversion between clouds. On the other hand, Kang et al. [20] quite thoroughly show how you can model your architecture with micro-services and containerization to alleviate the need to do migrations. With proper containerization and service discovery you can simply deploy your containers on another cloud with another VM, and not spend time figuring out VM conversion, because the underlying VM is, in the end, unimportant. Where our approach takes this concept even further is to containerize the entirety of the architecture; not just the application but also build, deployment and operations. By packaging all of this we can truly become cloud independent and can use resources from multiple clouds based on the specific requirements and cost constraints of our system. For example if we have exhausted storage capacities within our cost or performance specification within one cloud we might want to expand to another, as explained by Mei et al. in their second research question [22].

6. Conclusion

Mutable systems are in their essence unknowable. Immutable systems on the other hand, are eternally knowable due to the fact that they are changeless. With the introduction of Immutable Infrastructure a corresponding shift toward Immutable Architecture is natural, and in this paper we have highlighted the concept and provided an example of its use and some of the implementation considerations that this approach entails.

We have also discussed Immutable Architecture as a concept and noted that there appears to be little research into this approach to architecture and DevOps maintenance and evolution. The main contribution of this paper is to highlight the approach of containerizing all parts of the architecture and its environment.

Key concepts of Immutable Architecture:

1. Containerize everything, from operations to applications, to separate Architecture from Infrastructure as completely as possible.

2. Capacity to run a fully qualified production environment on developer machine, removing the need for a staging environment.

3. Live environment is fully knowable and directly documented through source code.

Just as Immutable Infrastructure allows us to transcend dependence on individual stateful pieces of infrastructure so should we develop Immutable Architecture to transcend dependence on the underlying cloud infrastructure. Only then can we truly experience cloud interoperability and simplify maintenance and evolution.

7. Future Research

Future research in this area should be devoted towards work on developing immutability and layering of (architectural) images. As a response to the current (isolated) focus on envisioning and planning future architectures through the lens of micro-services, research should look at the architecture itself as a composition of micro layers. We are now at the beginning of the era where we treat infrastructure as code. As such, we must not only treat it as scripts for configuration but as actual software.

The example in this paper can, in the future, be built upon to exemplify how Immutable Architecture should look, with the key elements being an absence of state and zero configuration after instantiation.

References

[1] R. Kazman, Y. Cai, R. Mo, Q. Feng, L. Xiao, S. Haziyevy, V. Fedaky, and A. Shapochkay, "A case study in locating the architectural roots of technical debt," in *Proc. 37th*, 2015.

[2] C. Ebert, G. Gallardo, J. Hernantes, and N. Serrano, "Devops," *IEEE Softw.*, vol. 33, no. 3, pp. 94–100, 2016.

[3] M. Callanan and A. Spillane, "Devops: Making it easy to do the right thing," *IEEE Softw.*, vol. 33, no. 3, pp. 53–59, 2016.

[4] C. Binnig, D. Kossmann, K. T., and S. Loesing, "the weather tomorrow?: towards a benchmark for the cloud," in *Second International Workshop on Testing Database Systems*, 2009.

[5] A. e. a. Khajeh-Hosseini, "The cloud adoption toolkit: supporting cloud adoption decisions in the enterprise," *Software: Practice and Experience*, vol. 42, no. 4, pp. 447–465, 2012.

[6] L. e. a. Vaquero, "A break in the clouds: towards a cloud definition," *ACM Computer Communication Review*, vol. 39, no. 1, pp. 50–55, 2008.

[7] M. Shahin, M. A. Babar, and L. Zhu, "Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices," *IEEE Access*, 2017.

[8] C. Amrit and Y. Meijberg, "Effectiveness of test driven development and continuous integration a case study," *IEEE Xplore: IT Professional*, 2017.

[9] M. Shahin, "Architecting for devops and continuous deployment," *Proceedings of 15th ASWEC*, 2015.

[10] M. Keeling, "Lightweight and flexible emerging trends in software architecture from the saturn conferences," 2008.

[11] S. Bellomo, R. Kazman, N. Ernst, and R. Nord, "Toward design decisions to enable deployability; empirical study of three projects reaching for the continuous-delivery holy grail," in *First International Workshop on Dependability and Security of System Operation (DSSO 2014)*, 2014.

[12] L. Zhu, D. Xu, A. Tran, X. Xu, L. Bass, I. Weber, and S. Dwarakanathan, "Achieving reliable high-frequency releases in cloud environments," *IEEE Softw.*, vol. 32, no. 2, pp. 73–80, 2015.

[13] J. Lewis and M. Fowler, "Microservices: A definition of this new architectural term," 2014. Online; accessed 9 Nov. 2017.

[14] Mulesoft, "The top 6 microservices patterns," tech. rep., Mulesoft, 2017.

[15] D. Shadija, M. Rezai, and R. Hill, "Towards an understanding of microservices," *23rd International Conference on Automation and Computing, University of Huddersfield, Huddersfield*, 2017.

[16] C. Pautasso, O. Zimmermann, M. Amundsen, J. Lewis, and N. Josuttis, "Microservices in practice, part 1 : Reality check and service design," *IEEE Software*, 2017.

[17] P. D. Francesco, "Architecting microservices," *IEEE International Conference on Software Architecture Workshops*, 2007.

[18] C. Pautasso, O. Zimmermann, M. Amundsen, J. Lewis, and N. Josuttis, "Microservices in practice, part 2 : Service integration and sustainability," *IEEE Software*, 2017.

[19] S. Hassan and R. Bahsoon, "Microservices and their design trade-offs: A self-adaptive roadmap," 2016.

[20] H. Kang, M. Le, and S. Tao, "Container and microservice driven design for cloud infrastructure devops," in *2016 IEEE International Conference on Cloud Engineering (IC2E)*, 2016.

[21] A. Lenk, G. Katsaros, M. Menzel, J. Revelant, R. Skipp, E. Leon, and G. V.P., "Tiosa: Testing vm interoperability at an os and application level – a hypervisor testing method and interoperability survey," 2014.

[22] L. Mei, W. Chan, and T. Tse, "A tale of clouds: Paradigm comparisons and some thoughts on research issues," *IEEE Softw.*, vol. 33, no. 3, pp. 7–11, 2016.

[23] "Vert.x." (Date last accessed 5-Jun-2018).

[24] "Dynamodb." (Date last accessed 14-Jun-2018).

[25] "Chef." (Date last accessed 7-Jun-2018).