INSTITUTO SUPERIOR
DE TECNOLOGIAS
AVANÇADAS DE LISBOA

Cloud provider independence using DevOps methodologies with Infrastructure-as-Code

Rui Manuel Ribeiro Pereira

Dissertação para obtenção do Grau de Mestre em

INFORMÁTICA

Júri

Presidente: Professora Dra. Andreia Cristina Teles Vieira

Arguente: Professor Dr. Arnaldo Emanuel de Almeida da Silveira Costeira

Orientador: Professor Dr. Pedro Ramos dos Santos Brandão

Aluno nº 99991905

Julho, 2021

Acknowledgments

I am extremely grateful to my wife Célia for her dedicated support and encouragement throughout the very intense academic years and also my son Rafael and daughter Daniela for their patience during the entire process. Thank you for all the unconditional support. It wouldn't have been possible without it.

I would also like to thank my supervisor Prof. Dr. Pedro Brandão for all the provided guidance and support as well as valuable insight during the development of this dissertation.

Resumo

Ao optar-se por infraestruturas de computação em nuvem para soluções de TI existe um risco associado de se ficar dependente de um fornecedor de serviço específico, do qual se torna difícil mudar caso se decida posteriormente movimentar toda essa infraestrutura para um outro fornecedor. Encontra-se disponível extensa documentação sobre como migrar infraestrutura já existente para modelos de computação em nuvem, de qualquer modo as soluções e os fornecedores de serviço não dispõem de formas ou metodologias claras que suportem os seus clientes em migrações para fora da nuvem, seja para outro fornecedor ou infraestrutura com semelhantes tipos de serviço, caso assim o desejem. Nestas circunstâncias torna-se difícil mudar de fornecedor de serviço não apenas pela complexidade técnica associada à criação de toda a infraestrutura de raiz e movimentação de todos os dados associados a esta mas também devido aos custos que envolve uma operação deste tipo. Uma possível solução é avaliar a utilização de linguagens para definição de infraestrutura como código ("Infrastructure-as-Code") em conjunção com metodologias e tecnologias "DevOps" de forma a criar um mecanismo que permita flexibilizar um processo de migração entre diferentes infraestruturas de computação em nuvem, especialmente se for contemplado desde o início de um projecto. Uma metodologia "DevOps" devidamente estruturada quando combinada com definição de infraestrutura como código pode permitir um controlo mais integrado de recursos na nuvem uma vez que estes podem ser definidos e controlados através de linguagens específicas e submetidos a processos de automação. Tais definições terão de ter em consideração o que existe disponível para suportar as necessárias operações através das "API's" das infraestruturas de computação em nuvem, procurando sempre garantir ao utilizador um elevado grau de controlo sobre a sua infraestrutura e um maior nível de preparação dos passos necessários para recriação ou migração da infraestrutura caso essa necessidade surja, integrando de certa forma os recursos de computação em nuvem como parte do modelo de desenvolvimento. Esta dissertação tem como objetivo a criação de um modelo de referência conceptual que identifique formas de migração de infraestruturas de computação procurando ao mesmo tempo uma maior independência do fornecedor de serviço com recurso a tais mecanismos, assim como identificar possíveis constrangimentos ou impedimentos nesta aproximação. Tal modelo poderá ser referenciado desde o início de um projecto de desenvolvimento caso seja necessário contemplar uma possível necessidade futura de alterações ao nível da infraestrutura ou de fornecedor, com base no que as "API's" disponibilizam, de modo a facilitar essa operação.

Palavras-Chave: Nuvem, Migração, Dependência, Desenvolvimento, Código, Referência.

Abstract

On choosing cloud computing infrastructure for IT needs there is a risk of becoming dependent and locked-in on a specific cloud provider from which it becomes difficult to switch should an entity decide to move all of the infrastructure resources into a different provider. There's widespread information available on how to migrate existing infrastructure to the cloud notwithstanding common cloud solutions and providers don't have any clear path or framework for supporting their tenants to migrate off the cloud into another provider or cloud infrastructure with similar service levels should they decide to do so. Under these circumstances it becomes difficult to switch from cloud provider not just because of the technical complexity of recreating the entire infrastructure from scratch and moving related data but also because of the cost it may involve. One possible solution is to evaluate the use of Infrastructure-as-Code languages for defining infrastructure ("Infrastructure-as-Code") combined with DevOps methodologies and technologies to create a mechanism that helps streamline the migration process between different cloud infrastructure especially if taken into account from the beginning of a project. A well-structured DevOps methodology combined with Infrastructure-as-Code may allow a more integrated control on cloud resources as those can be defined and controlled with specific languages and be submitted to automation processes. Such definitions must take into account what is currently available to support those operations under the chosen cloud infrastructure APIs, always seeking to guarantee the tenant an higher degree of control over its infrastructure and higher level of preparation of the necessary steps for the recreation or migration of such infrastructure should the need arise, somehow integrating cloud resources as part of a development model. The objective of this dissertation is to create a conceptual reference framework that can identify different forms for migration of IT infrastructure while always contemplating a higher provider independence by resorting to such mechanisms, as well as identify possible constraints or obstacles under this approach. Such a framework can be referenced from the beginning of a development project if foreseeable changes in infrastructure or provider are a possibility in the future, taking into account what the API's provide in order to make such transitions easier.

Keywords: Cloud, Migration, Lock-In, DevOps, Infrastructure-as-Code, Framework.

Index

List of Figures

List of Tables

Glossary

API	Application Programming Interface
CaaS	Container-as-a-Service
CD	Continuous Delivery / Continuous Deployment
CI	Continuous Integration
FaaS	Function-as-a-Service
IaaS	Infrastructure-as-a-Service
IaC	Infrastructure-as-Code
IT	Information Technology
JSON	JavaScript Object Notation
PaaS	Platform-as-a-Service
SaaS	Software-as-a-Service
SCM	Source Code Management
SDLC	Software Development Life Cycle
SOA	Service Oriented Architecture
TBD	To Be Defined
XaaS	Anything-as-a-Service
YAML	YAML Ain't Markup Language

Literature Review

Computing as Utility

The concept of computing as a utility was first coined by John McCarthy in 1961 around the idea that computational power could be charged by unit of consumption and it was expected that it would become the fifth utility after water, electricity, gas and communications [1]. Under that assumption, the possibility of having control over IT costs with cloud computing architectures instead of massive upfront investments for on-premises infrastructure along with its associated setup and maintenance has gained momentum in the last decade and cloud computing - named after the cloud image used in diagrams where anything over the internet is depicted - has evolved to become the preferred solution for most entities when it comes to their IT needs, making the shift to cloud-based infrastructure become the norm in recent years [2].

Cloud computing in its current form would not be achieved without several technological advancements especially in the field of virtualization technology combined with the evolution of processing capability and increased hardware density [3]. Virtualization technology has decades of existence and was first introduced in the 1960's by IBM on their mainframe systems, although only in recent decades has virtualization seen increased development and adoption under commodity hardware. In essence, virtualization allows for the sharing of available computational resources among systems or applications in a more efficient and flexible way [4].

Virtualization

Virtualization is implemented through the use of a virtual machine monitor, also known as hypervisor, positioned at a specific level depending on the type of virtualization, making the segmentation and partial allocation of resources possible. Virtualization techniques such as bare-metal virtualization consists of physical hardware segmentation with the hypervisor (known as type 1 hypervisor) being adjacent to the hardware level allowing the definition of sets of resources to be isolated and subsequently assigned to specific virtual environments, and hosted virtualization where the hypervisor (known as type 2 hypervisor) is located above or adjacent to the operating system layer, permitting the segmentation or sharing of resources from those available within the operating system where it resides [5]. Both types of virtualization can be hardware assisted by the use of specific hardware extensions and features developed for the purpose of improving its performance and efficiency [6]. Two other distinct forms of

virtualization are paravirtualization, a form of software virtualization based on specific operating system kernel functionalities and drivers to interact with its virtual environments in a more streamlined manner lowering its computational overhead, and operating system virtualization or containers based on the concept of duplicating the operating system environment or a subset of it but keeping the kernel layer common to all environments [7]. The several types of virtualization technologies are not necessarily self-contained and some can even be intertwined creating more complex and nested virtualization architectures [8]. Despite the underlying mechanism used for virtualization and type of hypervisor or virtual machine monitor implemented, virtualization techniques have extended to other layers of infrastructure besides virtual machines. The virtualization of other resources such as network devices enabling the creation of routers, firewalls or other networking equipment in their virtual equivalents as well as storage components through the creation of virtual disks and the encapsulation of their respective communication protocols is of critical importance for cloud infrastructure and combined with virtual machines comprises the core components for cloud-based infrastructure [9].

Cloud Computing

A cloud computing architecture, while depending on virtualization as its core engine, may encompass several different approaches and present different service models, with those models depending on what forms of virtualization are effectively available underneath. From a conceptual perspective, there are a variety of technical options and models for cloud deployments and solution requirements will define what kind of model best fits and in what form should cloud computing be adopted [10].

Characteristics

Regardless of the type of architecture or implementation, there are essential characteristics to what cloud computing provides that makes this type of technology more appealing for deploying current information technology solutions. The possibility to provision resources as needed or on-demand self-service, being able to dynamically create resources with a certain degree of abstraction on where those resources are physically located or where the pool of computing resources is available (commonly defined as resource pooling) providing scalable and flexible allocation and deallocation of resources enabling rapid elasticity as if the resource pool was unlimited, and a broad network access to those resources from anywhere through the internet,

are the characteristics that when ultimately combined with being charged only for resources consumed turning it into a fully measured service defined cloud computing, making it a standard approach for current IT architecture needs [11].

Deployment Models

The advantages of cloud computing are not exclusive through the use of cloud provider solutions since cloud infrastructure can be deployed locally and still provide most, if not all, of the previously mentioned features and benefits. Although cloud computing is usually seen from a provider perspective, there are different deployment models to choose from. Of those, three cloud infrastructure deployment models are commonly described. Private Cloud, where cloud infrastructure is provisioned for private use of an entity by deploying on-premises equipment or through colocation facilities with the operation and management being of sole responsibility of the entity itself, Public Cloud which is the most commonly adopted method of cloud computing through a cloud provider based on the renting of computational resources having therefore no responsibility on managing equipment or the infrastructure layer and benefiting from a certain service level, and Hybrid Cloud where a mixture of Private Cloud and Public Cloud resources are interconnected allowing for the expansion and scalability of IT infrastructure as needed but optionally keeping critical processes or data under more control [11]. A fourth type of cloud deployment model described in literature as Community Cloud is from a technical perspective a combination of the previously mentioned types of deployment, but shared among multiple entities which makes it more of a social aspect on cloud adoption rather than a technological type of implementation [12]. There are additional benefits brought by cloud computing independently of the deployment model. Of those, resource compartmentalization, detailed reporting on cloud resources consumption by compartment level (reporting not only on the resources consumed but also forecasting possible future trends), dedicated monitoring, high availability for increased resiliency, improved security through the use of encryption technologies for many of the processes and resources involved in the architecture as well as the communication or storing of information between them and strong access control measures are among the most common [13]. In a public cloud environment an increased security awareness can also be considered as an additional benefit by having dedicated security teams and processes providing reports on possible security issues or vulnerabilities concerning deployed assets, upon which action should be taken [14].

Service Models

A chosen architecture for cloud adoption will fit into a certain type of service model or possibly a combination of those depending on the solution requirements. The most common and traditionally known service models are Infrastructure-as-a-Service, Platform-as-a-Service and Software-as-a-Service, respectively known as “IaaS”, “PaaS” or “SaaS” models [11]. Infrastructure-as-a-Service resembles common virtualized IT infrastructure hugely based on and similar to a traditional virtualization approach, providing resources through a virtual abstraction layer and allowing the provisioning of compute, storage and network resources akin to physical equipment. Platform-as-a-Service pushes the abstraction layer one level up, providing resources that can be readily used for development and deployment of applications through a combination of database instances, application servers or any other type of middleware or software components for that purpose thereby abstracting the entire infrastructure layer and allowing the focus on development and management only of the application layer related assets. The upper layer of the cloud computing stack is commonly defined as Software-as-a-Service and relates to a software product ready for consumption over the internet where the essential characteristics still apply such as being charged only for consumed resources and having broad network access but still a layer of service completely dependent on cloud provider offerings and by definition a final product [12].

Alternative architectures

Other types of service models beyond the standard ones mentioned have surfaced over time in order to provide optional solutions to different problems and it is common to see a generalization of this trend defined as “XaaS” where “X” stands for something that is the object of becoming a service, such as “CaaS” for Container-as-a-Service or “FaaS” for Function-as-a-Service, this last one also commonly known as Serverless computing [15]. These types of service models allowed different approaches to cloud architectures bringing new concepts and alternative ways for cloud adoption and have been evolving rapidly with “CaaS” and “FaaS” already being supported on most cloud providers or available in software for cloud infrastructure due to such service models becoming mainstream for more advanced methods of cloud adoption [16].

Container-as-a-Service

The Container-as-a-Service or “CaaS” model is implemented on top of container-based virtualization and highly benefits from its advantages since this type of virtualization does not instantiate a new virtual machine or image of the operating system, instead creating an isolated runtime environment for the application which shares the same kernel with the hosting operating system and only partially duplicating some necessary system components for the application to run [17]. A container image incorporates the application or part of it with all the necessary dependencies for it to run within a given type of container-based virtual environment, being this runtime environment provided by the container-engine. The isolated runtime environment is only generated when the application is launched, allocating the necessary resources from those already available within the operating system and returning all those resources as it finishes execution, which can be short-lived or long-lived. This type of virtualization also benefits from better resource utilization since no new resources are allocated for the creation of the runtime environment besides those are already available on the operating system instance, becoming more lightweight, and achieving faster start-up times since the launching procedure does not have to boot an operating system image with all its associated hardware initialization and complex start-up processes, which is optimal for fast scalability [18]. It is desired that a container image can be deployed or run in different infrastructure and in order to provide a greater degree of independency, container engines assure certain degrees of compatibility. This type of service in a cloud infrastructure is usually provided with container engines being themselves deployed on top of virtual machines or other types of computational resources for that purpose. Resource allocation for those virtual machines must take in consideration the resource requirements of the applications to be run in such model [19].

Function-as-a-Service

The Function-as-a-Service or “FaaS” model, also known as Serverless computing, is in its essence an alternative method for taking advantage of the container-based virtualization features both from a technical and service level perspective [20]. It is applicable in a context where there is no need to instantiate an application component along with its dependencies which are commonly long-lived and conceptually deployed to stay running for longer periods of time [21]. In contrast, by invoking some code base that serves a very specific purpose, the function residing in a container image is immediately deployed and performs its specific function as requested for a given amount of time or specific number of invocations, preferably short-lived. From a service level perspective, charging can be done by the number of invocations or execution

time. This type of service has become an important feature in cloud architectures as it allows for the use of computing resources more efficiently without having to set up infrastructure in the traditional sense, relying instead on parts of code to be dynamically deployed when invoked on top of an already existing infrastructure awaiting such deployments, having at its core container engines similar to the Container-as-a-Service infrastructure, allocating the resources they need in response to certain events or triggers and releasing those resources when execution finishes [22].

Cloud Native Architectures

These newer and more recent service models brought different approaches to cloud computing, changing the nature of provisioning and deployment of services in a cloud infrastructure, which can also benefit from tools and methodologies already used in other areas of development. Commonly defined as Cloud Native architectures, these have gained widespread acceptance in recent years beyond the common service models and contributed to a different perspective not only on cloud adoption but also on how to envision cloud related development and deployment [23], [24]. In order to better take advantage of cloud computing capabilities like scalability and flexibility, a cloud native architecture consists of embracing cloud computing by developing and deploying applications more independently of traditional service models such as "IaaS" or "PaaS", instead choosing to adopt from the beginning of the development phase cloud models that rely on container-based virtualization highly leveraging its core features, using service models such as "CaaS" or "FaaS" and ultimately shifting from the traditional multi-tiered or monolithic architecture paradigm of development to a Microservices based one [25]. As a consequence, development under a cloud native methodology also requires adapting or switching to newer architectures and paradigms of software development in order to integrate and better take advantage of such cloud capabilities with container-based virtualization at its core, raising however the difficulty of migrating existing traditional IT solutions into this model.

Microservices

A Microservices architecture, one of the available options for application development under a cloud native approach, is built upon the premise that developing small software components with one specific functionality and making those software components with well-defined tasks interact among them preferably in a loosely-coupled manner in order to provide the desired outcomes as an alternative to a monolithic type of development, more appropriately fits

newer cloud computing models. This would result in faster development (given a lower complexity as a result of breaking down the code base for each component) and deployment, as well as higher resiliency and improved maintenance resulting from lower impact due to code changes, since only specific components are changed or updated instead of the entire application, without predictable impact on other working components assuming those are designed in a way that failure of one would not compromise the entire application [26]. Design and development of applications under this model requires understanding the different architectural styles and as well as its implications because they must not developed according to most of the traditional paradigms of software development and a certain degree of adaptation is required [27]. This type of architecture is also more cloud agnostic and can more easily be distributed even among different cloud providers, increasing its resiliency and tolerance to failure resulting in better service availability. A Microservices based architecture design builds on container-based virtualization solutions providing higher flexibility along with the possibility of very fast dynamic orchestration of those resources for extreme scalability. Microservices architectures are based on an event-driven model that relies on event-based communication mechanisms across the several components usually through event streaming solutions using publisher/subscriber models or alternatively Remote-Procedure Calls (RPC) or REST APIs as well as protocols for automating service discovery [28], optionally coupling with other modern cloud features such as functions or even with traditional "PaaS" offerings such as "DBaaS" (Database-as-a-Service) for persistence. By leveraging container-based virtualization which benefits from lower resource requirements, lower overhead and faster start-up time [7], as well as taking advantage of existing development tools and deployment models with high levels of automation both for testing and deployment, a Microservices based architecture can provide rapid scalability, higher resilience and higher tolerance to failure without the need for complex setup and management of servers or infrastructure, providing a more simplified path to take advantage of the benefits of the cloud computing model and achieve better levels of availability and dynamic scalability for large scale solutions [29]. Cloud native architectures also reduce (but do not completely eliminate) the need for large operational teams and take advantage of one of the most important cloud features which is being charged only by resources consumed in a more efficient way, since only when containers are launched or functions are invoked there is actual charge for resource consumption and when those are no longer needed are automatically deallocated making the charging process stop without any additional billing for those resources. Both containers and functions excel at efficiency for cloud resource allocation and reduced charging since it depends on the level of activity or requests made into the application [22]. All those combined features pushed cloud native architectures into an attractive model for cloud adoption and development.

Infrastructure-as-Code

The creation of computing resources under cloud computing service models is usually first done through intuitive user interfaces well suited for creating relatively simple architectures or to quickly deploy small infrastructure, but underneath those interfaces there are powerful application programming interfaces or "API's" that receive requests for the creation or change of those resources triggering the necessary actions and providing feedback on the result of such operations [30]. Understanding the power of those APIs is necessary for taking advantage of the real scalability and flexibility provided by the cloud computing model and whenever a certain degree of automation in resource management and control is desired. Infrastructure-as-Code has become a standard mechanism for defining and controlling resource creation and configuration in cloud environments, also providing a mechanism for performing such provisioning in an automated and orchestrated manner [31]. By using Infrastructure-as-Code it is possible to define or declare resources and artefacts to be created or changed in a cloud infrastructure, along with their respective characteristics as well as any necessary dependencies. Additionally, some degree of automation can be contemplated in such operations through the use of a specific languages for the purpose. As a result, several benefits such as improved configuration consistency and faster deployment times can be obtained [32]. Contrary to functional programming where one gives exact instructions on how computer operations should be performed, Infrastructure-as-Code typically uses, although not exclusively, declarative programming languages that state what final outcome is to be expected concerning the creation or change in cloud resources or artefacts [33]. The desired configuration is described, parsed and fed into the cloud infrastructure API for processing which subsequently operate on such resources accordingly. Different cloud providers are supported and most contribute to the development of such languages, with template definitions for resources according to their respective offerings or features. Most cloud providers also support existing third-party tools and provide well documented APIs for such operations. Although Infrastructure-as-Code is usually seen for provisioning of infrastructure, it may also encompass tools or frameworks for configuration management, most using their own domain-specific languages for the layout of such configurations [34]. As a result it is possible to completely define and automatically deploy an entire cloud solution with all its necessary dependencies, regardless of the desired type of resources, completely changing the landscape of provisioning and management of IT infrastructure. Due to the nature of Infrastructure-as-Code having configuration files akin to source code in software engineering, it is recommended to keep those in a repository using version control software with some of the principles and tools that are used

for software development being applicable to Infrastructure-as-Code. Resource definitions can be kept and managed using a centralized source code repository, taking advantage of several features brought by such tools, such as the ability for different personnel to work concurrently on those resource definitions in a distributed manner and use version control which is of utmost importance to understand changes in time that subsequently reflect changes in cloud resources or infrastructure. Having principles of software engineering applicable to Infrastructure-as-Code provides not only the aforementioned additional benefits but also brings a new paradigm for the provisioning and creation of IT infrastructure, at the cost of having to adapt and learn new languages and methodologies for the purpose [35].

DevOps

The use of declarative or procedural languages for defining cloud resources and configurations, providing more sophisticated means for cloud adoption and management, can benefit from already existing practices and methodologies in software engineering. The steps associated with software development (an iterative process by nature) and its life cycle have evolved and matured over the years, improving the quality of development and delivery of the final product. Tools and methodologies in the context of software development minimized manual intervention for repeatable processes and brought an increased level of consistency and automation not only into the mechanics for building the software according to requirements but also to perform all the necessary testing and assure it meets quality demands before it is released [36]. Regardless of this evolution, software development is usually seen as a distinct process from IT operations and each of these areas usually have their own teams with separate responsibilities. In recent years, there has been an increased interest in DevOps as a software development and IT management methodology, or set of practices, applicable to the realm of cloud computing taking full advantage of its service model. The term "DevOps" is a combination of the words Development and Operations and as a methodology it is based on the premise of improving the interaction between those usually differentiated teams, with the prime objective of combining their skills and responsibilities for increasing agility and flexibility in both aspects of the development processes and the operations associated with code deployment and release. DevOps leverages the use of specific languages within the context of provisioning cloud computing infrastructure and enables a more efficient software development lifecycle, combining provisioning and application development with highly automated deploy and test/release mechanisms. Besides the technical aspects of software development such as build, test automation and release/deployment, a DevOps methodology highly emphasizes the social aspect of

collaboration and communication between members of the development and operation teams [37]. The increasing adoption of Infrastructure-as-Code and other similar languages for configuration, automation and orchestration in the context of cloud computing, envisioning an increased role of developers in infrastructure, benefits from such improved interaction between those two commonly independent areas of IT which are becoming ever more dependent, with that interaction becoming of utmost importance in order to quickly adapt and respond to any incoming challenge or adversity [38]. From a technical standpoint, DevOps methodologies focus on the concepts of Continuous Integration and Continuous Delivery or CI/CD pipelines through which an automation and orchestration of the entire development and deployment cycle is implemented, providing the ability to quickly integrate fixes or changes to existing code (continuous integration) and immediately trigger a release and deployment of the newly finished and tested version combining those changes (continuous delivery). Having automated test cases that guarantee the necessary results before the release gives an enormous improvement to the development and deployment process thereby raising the quality of the final product or service while at the same time allowing the fixing of bugs or addition of new features and releasing those improved versions much faster without any predictable service disruption [39]. The concept of CI/CD pipelines are useful not only for the areas based on Infrastructure-as-Code but also for development methodologies applicable to loosely-coupled architectures such as cloud native, improving the software development and deployment in approaches such as Microservices [40]**.** The use of a source code repository that has a complete history of all changes is at the core of every development methodology and DevOps is no different in that respect with automated actions being triggered after updates or changes are made to the code in the repository. Cloud architectures when combined with DevOps methodologies and Infrastructure-as-Code can fully automate actions to be performed in order to streamline cloud infrastructure deployment, operations and management [41]. All those features combined make DevOps an interesting model for turning IT development and operations into more agile and streamlined processes with higher levels of consistency and adaptability to change, especially whenever cloud computing solutions are considered [42].

Cloud Migration

Due to the advantages of cloud computing, there is an ever increasing interest in migrating already existing IT solutions into the cloud. However, despite the evolution of cloud computing technologies, not only from the standpoint of modern methodologies for development and deployment but also in terms of management and operations, there are considerable obstacles

when choosing cloud computing as a solution for moving an existing IT infrastructure, as opposed to new deployments, raising several questions concerning the migration process and becoming a complex challenge with varying degrees of difficulty for success depending on the overall objective. Cloud migration consists of moving existing IT infrastructure or part of it from on-premises into the cloud, or even between clouds [43]. Migration processes are not as mature as some other technologies regarding cloud computing and no clear standards exist for such operations, which is understandable to some degree since the underlying technologies such as type of virtualization or solution architecture may become a limiting factor and some type of transformation may be necessary [44]. It is also important to identify other possible constraints or limiting factors as well as implications of such migration beyond the technical aspects, such as costs, staff expertise, security issues or levels of compliance [45]. Generic documentation is available to aid in such transition concerning cloud migration especially from on-premises to cloud with different methodologies and frameworks proposed to address both requirement analysis and steps involved in such process, at least to some compatible degree for the most common service models [46]. Entities willing to migrate their on-premises infrastructure or IT solutions to a public cloud also have a considerable amount of information available from cloud providers to help them achieve that objective in migrating to their cloud service offerings, mostly without any deep changes in architecture, in order to minimize the risk of such migrations. The effort and technical complexity for migrating into the cloud will also depend on the type of migration desired, which will be constrained by what is currently deployed at origin and what type of architecture is to be achieved at destination. If a cloud-native or Serverless architecture is to be achieved when having the original environments based on the traditional IT approach of client-server model running monolithic or multi-tiered applications, complexity becomes even more challenging as a complete rewrite may be necessary [47]. Beyond the common service models, several types of migration scenarios cannot really be described into a standard process as it involves specific technical knowledge about the existing infrastructure and the migration process may require contextualized development and rewriting or refactoring of applications into the new paradigm or architecture, at best having general recommendations on good practices for such transformation [48].

Under general recommendations or usual practices, there are some commonly accepted high-level strategies for cloud migration that describe how the movement of existing IT infrastructure or solution into the cloud should consist of, as well as the type of IT transformation that could result from such process. Six defined strategies, known as the 6 R's of migration, are designated as [44]:

1. Retain
2. Retire / Replace
3. Relocate / Rehost
4. Replatform / Refactor
5. Rebuild / Reuse
6. Rearchitect

Those six high-level strategies only present general guidelines without any in-depth detail on how the migration should be made, which is understandable since every environment will have its own technical complexities that will define what can or cannot be done. The resulting degree of transformation desired during the migration process will also influence the difficulty or effort involved in the process [49].

Vendor Lock-In

Despite the information and documentation available to assist in a cloud migration process, and their similarities at a conceptual level among all the different methodologies within the common service models, when moving existing IT infrastructure into the cloud, especially when public cloud is considered, there is a considerable risk of Vendor Lock-In [50]. This has become one of the greatest obstacles to cloud adoption since once IT assets are migrated into the cloud, moving those assets off the cloud back into a private cloud or into another cloud provider is not so well documented and poses a significant technical challenge to do so should the need arise [51]. Several obstacles have been identified as a likely cause for difficulties in migrating to another cloud infrastructure or cloud provider once migrated to the cloud, some of which are of special concern such as the recreation of the entire infrastructure and portability or interoperability issues across a different cloud stack. Such obstacles pose a significant hindrance whenever the need to migrate to another cloud provider or infrastructure is eventually necessary, and the lack of standardization for such processes make it technically difficult to switch, elevating some of the risks for cloud adoption [52]. Although all of the current top public cloud services have documented and made available well defined methodologies on how one should migrate to their cloud within the standard service models and have support services to help in such transition to a considerable degree, having the same level of support to export those configurations in order to

recreate a complete infrastructure in another provider or infrastructure is practically non-existent, and no standards exist. Different approaches have been proposed as possible solutions to this problem in order to lower the dependency on a single cloud provider. Switching to different forms of cloud adoption such as the previously described Cloud-Native architectures and combining those by deploying in multiple cloud providers is one option. Although such solutions lower the level of cloud provider dependency, they do not completely eliminate the problem [53] and represent a radical shift with a considerable effort necessary to make the adjustments or changes to the existing IT solution not only at an infrastructure level, but also on the software or application level as already described [54].

Multicloud

Cloud adoption based on using multiple cloud providers has gained acceptance as a possible solution or remediation to the problem of Lock-In. Commonly described as the Multicloud paradigm, in its essence consists of using more than one cloud provider to deploy a given service or architecture, distributing the components of the solution among those providers regardless of the service model involved. The Multicloud paradigm tries to guarantee that no disruption would occur should one provider become unavailable, this possibly resulting in the need to evaluate several migration patterns, or a combination among the ones available to choose from [55]. This approach lowers the risk of being locked into a specific provider, or at best minimizes such dependency. While it does address the problem of being dependent on a single cloud provider, it still does not completely solve the problem of cloud provider dependency since moving the components from one chosen provider onto another continues to be largely unsupported and several technical constraints should be taken into before choosing this paradigm, if the assumption that such movement of artefacts would be possible [56]. While Multicloud adoption may be similar to an hybrid cloud deployment model, from a conceptual perspective an hybrid model relates to the interconnection among different type of cloud implementations (private and public) while the Multicloud paradigm is based on interconnecting architectures or IT solutions between different clouds beyond the hybrid model, commonly within the same deployment model (private-private or public-public) [57].

Conclusion

The diversity of cloud computing infrastructure deployment models and architectural options regarding IT solutions deployed on such infrastructure can be associated with an increased level of complexity whenever a migration process involving such infrastructure is desired, regardless of that migration being from on-premises or traditional IT infrastructure into the cloud, or between clouds. Possible constraints such as portability and interoperability issues or specific contextualized technical difficulties for performing such migrations can also be identified from within an already deployed IT solution due to its architectural model, independently of its dimension. It is crucial to have a clear understanding of cloud deployment and architectural models as well as their technical details, in order to identify their main strengths and weaknesses concerning their viability as targets for migration. It is also imperative to ascertain some degree of compatibility whilst identifying any necessary changes to the original IT solution during the process, in order to improve the outcomes for a successful migration. New developments and deployments must also take into account that any choices made regarding infrastructure or architecture will impact its migration prospects in the future, eventually raising technical debt. Whether deploying a new IT solution or migrating an existing one, in order to minimize risks and assuming migration is also to be considered at later time, it is important to have a deep technical understanding of what is to be migrated and review alternative methodologies and tools to aid in such migrations beyond the already existing solutions, while at the same time evaluate the impact of architectural choices during the course of such migration and their implications on cloud provider independency.

Identified strategies

Despite the many advancements achieved in the field of cloud computing, the original concept and vision for cloud computing as a utility still lacks one major feature which is the ability to move or migrate between cloud providers or cloud infrastructure seamlessly whenever desired, akin to the simplicity of switching between internet service providers. Even with all the pervasiveness of cloud computing and the advancements made to comply with its main characteristics, the full concept of computing as a utility is yet to be achieved until such seamless movement between clouds is possible, at least up to some baseline service or other types of non-vendor-specific or standard service offerings.

Difficulties in cloud migration have been promptly identified since the inception of cloud computing. From the perspective of migrating from on-premises or Traditional IT into the cloud, extensive documentation and methodologies have been made available but those have never addressed the need for moving related resources between clouds after such initial migration, should it become necessary. Support and documentation for moving cloud resources between providers is scarce or very limited, partly due to the previously mentioned lack of standardization but also because it is not in the provider's interest to lose any customers. These factors have contributed to locking a customer on a specific provider with the option of moving between providers sometimes becoming more expensive than the current costs with existing infrastructure, resulting in a serious drawback that limits choice and freedom of movement for an IT solution running on cloud.

Some solutions have been proposed for this problem. However, even when considering existing solutions, such movement of IT architecture between cloud infrastructure is far from being achieved with ease, mainly because of portability constraints which lead to considerable technical modifications being needed when migrating to a different cloud, with portability and interoperability problems stemming as a result. A standardized baseline compatible service between different providers that can be migrated under such an ideal condition is difficult to find across the entire stack of cloud offerings, with most services or artefacts requiring a substantial technical adaptation effort if they are to be migrated. This also leads to any migration process being dependent on a detailed evaluation and approached on a case-by-case basis, having immense specificities to deal with, which could otherwise be made simpler through the existence of such standards for generic artefacts and their related movement between different clouds.

Although the lack of standardization for such migration processes between different

providers can be identified as one of the main causes of such difficulties, it is understandable that an all-encompassing type of standardization may not be possible due to the fact that some cloud service offerings are provider or vendor-specific and therefore some apparent portability constraints should not (at least conceptually) be treated as such, instead considering such offerings as a final product, similar to SaaS. Despite the fact that some of the components underlying cloud computing architecture already comply to well defined standards, such as virtual machine image formats, which is an important aspect to consider for an eventual standardization of migration processes, the standardization of migration techniques that could build upon those standardized artefacts is still lagging. A full standardization for every cloud service or artefact also having an associated standard migration method seems unlikely, but some baseline services or artefacts should definitely be supported and standardized for such seamless transition between clouds, a feature that would undoubtedly complement the original cloud computing vision.

From a theoretical perspective, several solutions or strategies have been proposed over time to partially address the problem of cloud migration and to more easily deal with the technical challenges associated with it. A diversity of cloud migration related topics and strategies are addressed among the evaluated papers and technical articles for this dissertation. Most of the strategies or topics on those documents concerning cloud migration are not concerned with achieving full independence from cloud providers from a practical standpoint, or to provide clear and detailed migration methods for achieving such independence, which would prove difficult if not almost impossible considering current service offerings, focusing instead in general constraints and difficulties foreseeable in such migrations while at the same time identifying other important and adjacent issues for reaching higher levels of portability or interoperability. Among the reviewed papers, several methodologies concerning migration were identified such as:

- ARTIST - Advanced seRvice provisioning and migraTIon of legacy Software
- CIM3 - Cloud Migration Maturity Model
- Cloud-RMM - Cloud Migration Reference Model
- Cloudstep - Cloud Migration Decision Process
- REMICS - REuse and Migration of Legacy Applications to Interoperable Cloud Services
- MDA - Model Driven Architecture
- mOSAIC - Open-source API and Platform for Multiple Clouds
- TOSCA - Topology and Orchestration Specification for Cloud Applications
- V-PAM - Variability-based, Pattern-driven Architecture Migration

Most of the reviewed papers and the aforementioned migration methodologies addressed adjacent issues to cloud migration such as: organizational aspects; risks, level of readiness and preparedness, compliance; strategic aspects; benefits, opportunities and threats; economic viability and technical feasibility for migration; migration evaluation; migration layers; required assessments; effort estimations; specific frameworks to aid in decision making as well as key factors for such decisions; migration planning; detailed procedures or tasks for migrating; identification of possible constraints; data portability; performance expectations; relevant reference architectures; development changes, among others. Other theoretical approaches on the reviewed literature are based on taxonomical and ontological definitions to cloud artefacts that could simplify the manageability of such artefacts by making them more easily transposable or interpreted between different providers at a more technical level, an important contribution to any standardization effort. The migration of legacy components into the cloud is also commonly addressed under the reviewed literature, mostly associated with an IaaS service model as this is the model that better adapts to the migration of legacy components or traditional IT without considerable effort, being easier to analyze and estimate from a migration perspective.

It can be seen reflected in literature that while cloud computing was becoming established over the years, the search for such seamless methods of migration between clouds was not easy to achieve or standardize due to the many different implementations, each providing their own features or specificities that would be difficult to adapt and conform to such migration process. As time progressed, some of the proposed ideas seen in literature became obsolete or no longer relevant from a practical standpoint due to other advancements in cloud technology that, in search for answers to other issues, somehow brought as a side effect alternative and more flexible solutions to the problem, at least partially.

In the search for more efficient use of cloud resources, recent migration techniques to cloud computing and cloud migration inadvertently addressed the issue of vendor Lock-In by leveraging technical features of already existing solutions to other problems and ingeniously adapt those, creating new paradigms for cloud computing in terms of migration, along with the development of new tools and methodologies that could help ease such dependency on a specific cloud provider or infrastructure. These recent tools and methodologies may help ease such constraints for cloud migration and offer a path (although still not a standardized one) for a higher independence.

Among all the methods reviewed in literature for addressing the problem on the subject

of migration, it is noticeable that the more recent approaches to cloud computing are the ones that more heavily contributed to the possibility of deploying IT solutions in cloud infrastructure in a way that could be more independent of the provider or the underlying infrastructure, not relying on the existence of formal standards but instead adapting and reusing existing technology for such advantage. Regardless of how such independence (even if partial) can be achieved, the important aspect to retain is that even though with some inherent complexity and still not as seamlessly as desired, it is now possible to achieve such movement between clouds though the use of technologies such as Infrastructure-as-Code, container-based virtualization technologies and DevOps methodologies for development and deployment of IT infrastructure and associated architecture, as discussed further on.

More recent approaches such as the non-standardized 6 R's of migration have been widely accepted as a general guideline and are used in practice, especially considering migration from on-premises to cloud. This method also provides an important contribution from a macro perspective on the subject of migrating into the cloud, hinting that some change of architecture in the process could give some benefits in terms of independence for future migrations.

The term "cloud provider" or "cloud infrastructure" may be used interchangeably in this dissertation as it may refer to a cloud service provided by an external vendor (public cloud) or a cloud infrastructure deployed under on-premises equipment (private cloud), or even an interconnection of both (hybrid cloud). Regardless of the type of deployment, it assumes running cloud infrastructure software complying with the common cloud reference model and definitions.

Assessment

Migrating to a cloud computing infrastructure or cloud provider encompasses a careful review and analysis of the characteristics of what is to be migrated into the cloud as well as its technical requirements, available options for migration, and identification of any potential drawbacks. The result of such analysis should be clearly detailed and understood before any technical decision is made, in order to minimize the possibility of constraints and other difficulties in future migrations, should they become necessary. Despite the several advantages of the cloud computing model, other critical aspects must be taken into consideration before migration, such as the level of transformation desired for the IT architecture during the course of migration, with this transformation having a direct impact on the level of independency that can be achieved from the provider. It is also crucial to identify if the existing environment has legacy applicational components that may be prone to other type of portability or interoperability issues. Adjacent topics such as costs, security, compliance and data integrity are also critical aspects to be considered whenever a cloud migration is to be planned but those are beyond the scope of this dissertation.

Before a migration process is initiated, an initial assessment should abide by some general guidelines mostly related to technical issues in order to not only document important aspects of the existing infrastructure or architecture but also, through careful and detailed planning, to be able to anticipate possible undesired consequences. The following general guidelines for a migration plan describe a structured and common approach for such assessment applicable to every layer of the IT architecture. The steps outlined are ordered and numbered for further reference:

General Assessment Guidelines

1	Evaluation of the infrastructure to be migrated and its architecture including all its components, their technical requirements and specifications
2	Identification of all dependencies and interconnections across the several components of the existing solution as well as their level of portability and interoperability
3	Choice of migration strategy and desired transformation, depending on constraints
4	Planning of necessary changes in component configurations or other specific changes during the course of migration, both for remaining and moved components
5	Measurement of the volume of data to be migrated, how to migrate and how to import it,

	as well as an estimation of the time needed
6	Development of a testing plan that should incorporate the necessary interoperability and functional testing for both migrated and with remaining components
7	Detailed description of necessary steps for partial activation of components as they are migrated, as well as expected downtime and measures to mitigate it
8	Development of a rollback strategy, if applicable

Table 1 General Assessment Guidelines

Each of these general steps will unfold different types of actions depending on the level of desired transformation in the IT architecture during the process of migrating to cloud infrastructure. Such transformation, even if partial, can also result not only as a requirement but also as consequence of the migration process, which in that case can set forth other general guidelines, eventually conditioning or partially defining the final architecture on cloud infrastructure after the migration process is completed.

Planning

The assessment must also take into account the desired migration strategy, which reflects what is expected as the final outcome in terms of IT architecture of the entire migration process and may imply some form of transformation. Depending on the type of strategy chosen for migration, the generic steps described previously under the general guidelines will have specific intermediate steps in order to reach the desired outcome, with the entire migration process having to adapt to (and therefore being impacted by) the chosen strategy. The following description of the most common strategies for migration usually seen not only in literature but also as current practice by public cloud providers are considered among the most valid methodologies. The first two are not relevant for this dissertation, they are however described for completeness. Higher complexity in migration is expected as more transformation is needed, with such complexity also depending on other specific technical properties of what is already deployed.

Common Strategies

Retain (As-Is)

Effort for migrating a given IT component or application will have a higher cost than retaining it

in its current model, either because it is scheduled to be decommissioned or because it is incompatible with current cloud solutions given the nature of the application or any other technical aspect of it.

Retire/Replace (SaaS)

Evaluate and terminate IT components that can be decommissioned. Some may have some cloud equivalent under a SaaS subscription model. Instead of planning for the migration of those, a proper evaluation of the SaaS solution should be done instead and if viable, be chosen as an alternative.

Relocate/Rehost (IaaS)

The current existing IT environment is viable to be migrated into an IaaS based cloud solution, similar to colocation or rehosting under the traditional IT but instead adapted to virtual infrastructure. Also commonly defined as Lift-and-Shift.

Replatform/Refactor (PaaS)

Existing IT components above the infrastructure layer may be good candidates for migrating into a PaaS cloud model benefiting from specific provider features and lower administration overhead, but some degree of reconfiguration of other components on the existing IT architecture may be necessary.

Rebuild/Reuse (CaaS / FaaS / XaaS)

Rebuilding specific IT components that are viable to be transformed in the context of a migration process to more cloud-specific features such as CaaS or FaaS models, lowering costs with infrastructure without the need for permanent servers or virtual machines and benefiting from higher scalability. Some development effort may be needed in order to transform legacy components as well as setting up the required testing.

Rearchitect (CaaS + FaaS + XaaS)

Completely transform existing infrastructure to a mixture of CaaS and FaaS/Serverless based cloud architecture, or any other XaaS model, even combining with PaaS or IaaS components whenever applicable or justifiable. This complete shift requires a substantial development and

testing effort because the entire infrastructure has to be evaluated to such transition as well as the associated impact of such changes.

IaC Templates

Defining cloud computing resources or artefacts using Infrastructure-as-Code (IaC) languages, classified as domain-specific languages developed for the purpose of defining infrastructure artefacts in the form of a parseable and verifiable template that must conform to specific rules instead of using any other non-programmatic way, made the provisioning of IT infrastructure into a descriptive format, transforming the management of infrastructure into an agile, consistent and repeatable process that can be automated with a high degree of reliability. The additional combination of IaC with tools and methodologies available for software development changed the management of such resources, which are usually under the domain of IT administrators or infrastructure architects, turning it into a process similar to software development. Those aspects pose a significant advantage for any migration planning since it is possible to define the required resources or artefacts upfront before the migration process begins, regardless of the chosen migration methodology as long as the types of cloud artefacts are supported. The use of IaC relates only to the creation of artefacts and has no influence on the volume of data or on how the data is transferred between the involved infrastructure.

The definition and creation or modification of such resources through these languages, when also combined with source code repositories, allows for a higher degree of control and visibility of every resource or configuration created or changed throughout their lifetime. Keeping all resource definitions and modifications within a repository is important for later reference should any change or migration of the infrastructure be necessary. Through the use of a repository it is possible to have a history of the evolution of the IT solution and its respective resources, as well as changes in its associated architecture, in a self-documenting manner. Abiding to those practices gives the possibility to repeat any previous action with consistency and provides a manageable path for further migrations. Besides all cloud infrastructure related configurations, any other complementary environment necessary to the IT solution such as storage and networking components must also be done programmatically and kept within the repository whenever possible. It is therefore crucial to combine a configuration repository with IaC definitions or templates in order to achieve such degree of overview and control over the cloud infrastructure, and it is of utmost importance that every resource definition or change is registered within it.

Cloud providers and cloud infrastructure frameworks support the use of IaC languages and have well-structured and documented API's for their use, in order to allow an extensive level of operations to be done on cloud resources besides their creation, change or destruction. Cloud providers also contribute with language updates and changes upstream, according to their own infrastructure features and offerings. Terraform is one of the available and most supported languages for declaring Infrastructure-as-Code, especially for IaaS cloud architectures, and will be used and referenced throughout examples in this dissertation. The concepts applied through Terraform are however extensible to any other IaC language for the same purpose, always depending on which features the IaC language provides according to their underlying cloud infrastructure. Although Terraform is fairly common among IaaS deployments, other types of cloud architectures may have more appropriate IaC languages.

As a declarative language, Terraform configuration files define a state to be achieved in terms of cloud resources, contrary to procedural languages which instruct exactly what should be done. Terraform allows for the management of cloud artefacts of almost any nature regardless of the service model being IaaS, PaaS, XaaS or Cloud-Native environments, as long as the necessary artefacts are supported by the provider. Terraform is idempotent when invoked, ensuring that changes and actions are not applied more than once. Although Terraform is cloud-agnostic, differences exist in the definition of cloud artefacts between different cloud implementations, despite sharing common aspects at a conceptual level. Defining artefacts to one particular cloud infrastructure has to adhere to what is permitted by its respective model and will not work for other cloud implementations without adjustments. Terraform uses the concept of providers for addressing such differences, having specific providers describing the available resources according to the underlying cloud framework being considered. Terraform works by defining such resources in configuration files, formally describing artefacts using its own HCL syntax (Hashicorp Configuration Language) more suitable for humans, or optionally JSON syntax, more appropriate for machine processing, relatively to what is possible under the cloud infrastructure and always depending on the resources made available through the provider. Terraform automatically manages any necessary dependencies when creating or destroying artefacts, contemplating their dependencies in proper sequence. Since those provider modules are usually supported, there is a certain guarantee in their functionality and consulting the documentation on how to use them is necessary, not only because of some considerable complexity but also due to the evolution of language features or newer cloud offerings over time. Terraform also allows importing infrastructure artefacts previously created by other means such as web interfaces into

an IaC template, as long as it is supported by the provider as an exportable artefact, translating such objects into IaC declarations and giving the possibility to begin managing those resources with IaC as well. Another additional benefit of Terraform, due to the fact that everything is registered in a source file that can be parsed and interpreted in various forms, is the possibility to use that information in the context of a self-documenting infrastructure. Developing infrastructure with Terraform can also take advantage from many of the functionalities seen in common programming languages such as variables (which can have values defined from the working environment for higher flexibility), parameters, functions, loops, conditionals and expressions, allowing for the creation of more complex forms of declaring and changing infrastructure in terms of provisioning, configuration and also management.

From a technical perspective, assuming that a Terraform workspace environment is initialized with all necessary authentication configurations against the selected cloud infrastructure, the desired resources can begin to be defined in their respective configuration files within that workspace, having therefore the minimal setup to start defining infrastructure or cloud artefacts in place. When invoked, Terraform performs a validation process on resource definition files by parsing them for structure and syntax validation as permitted by the selected provider, identifying any possible errors or constraints. Assuming a successful validation, a complete report on what resources will be created, changed or destroyed is displayed for analysis before effectively applying the desired configuration against the selected cloud infrastructure. By accepting reported changes, Terraform will trigger associated actions through the provider that communicates with the cloud infrastructure API, displaying a final report on what was successful and what has failed. Failed resource changes will not only be reported but also kept in a state file for post-processing. Due to the asynchronous nature of Terraform, applying changes may not have immediate reflection on the cloud infrastructure, but some form of confirmation, even if not synchronous, is expected. Most changes made outside the control of Terraform not reflected on the resource definition files will also be detected and reported for analysis. Diagram 1 describes an example of the common workflow for configuring infrastructure using Terraform followed by its description:

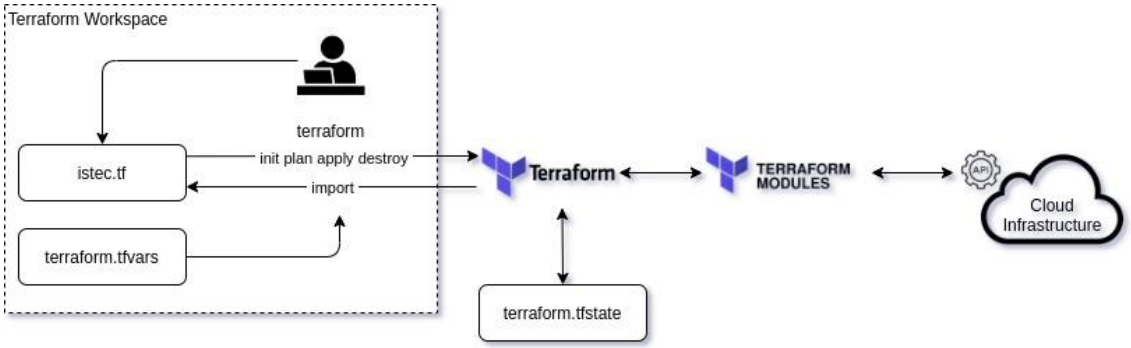

Diagram 1 Generic Terraform Workflow

After properly initializing a Terraform Workspace with *init*, the example file *istec.tf* would contain the resource definitions which would consequently define actions respective to the creation, modification or destruction of the declared artefacts. If necessary, resources already created by other means on the cloud infrastructure could also be imported and reflected into some *.tf* resource definition file. Only one file is given as example but multiple *.tf* files can exist, optionally named in some relatable way with the infrastructure they describe and, if desired, placed in different directories for applying logical segregation of resource definitions. Specific environment configuration variables are defined within the file *terraform.tfvars* and are evaluated whenever a Terraform action is invoked. Values for some of those variables can optionally be inherited from the working environment for greater flexibility. Amid the common Terraform actions, *plan* parses the *.tf* files for checking their consistency and identifying the provider needed, also reporting on exactly what actions are to be taken for both informational purposes and aid in decision making. Should the reported changes be accepted, *apply* effectively makes Terraform invoke the associated provider and establish communication with the cloud infrastructure in order to perform the respective cloud resource changes. Terraform is also capable of releasing any resources defined on the *istec.tf* file using *destroy* as well as import into a *.tf* file any resource already created on the cloud infrastructure but not yet referenced within the configuration file using *import*. A *terraform.tfstate* file is created and managed by Terraform for keeping a registration of pending or failed actions. Terraform can also detect any configuration drift or changes done outside Terraform by referencing the *terraform.tfstate* file. More complex operations can be done with other available options in Terraform, and will be referenced within their appropriate contexts whenever needed.

There are several other IaC languages depending on the chosen technological implementation, but regardless of the chosen technology, IaC is indissociably from cloud computing for describing infrastructure under the cloud computing model.

Code Repositories

Using a source code repository is necessary for having a centralized management and overview of all IaC related resource configurations, adopting a similar mechanism to software development practices in terms of code management. The concepts of source code management are traversal to most implementations, aiming to provide the same features and functionalities, although differing in the inner workings of their associated tools. Any source code management tool can be used with IaC as long as it provides a set of basic but essential features. Other advanced features of source code management tools and frameworks are also relevant not only to the realm of IaC but also for more complex forms of combined development and deployment or for Cloud-Native architectures. Crucial when application development and delivery becomes integrated with the deployment of its underlying infrastructure, or whenever incorporating toolchains for additional features such as monitoring, automation or orchestration is necessary.

Among the several features provided by source code repositories and management tools, versioning ranks as one of the most important for the realm of IaC, for keeping several versions of configuration files and to be able to track their changes across time, providing a detailed perspective on any resource or configuration change in a reliable manner and ensuring a complete and detailed record of every change applied. Changes are registered through the use of a unique identifier for each change or transaction committed into the repository. Source code repositories also provide the ability to have multiple developers working on the same set of source code files, enhancing collaboration without compromising the work of others even when inconsistencies in code arise, having full accountability on every change applied. Collaboration under this premise has to adhere to the chosen language constraints in terms of centralized repository usage and its level of compatibility. Other relevant features are branching and merging, for creating independent development branches of code, with the possibility of later integrating development done on those different branches back into the main branch, cloning or forking for performing a point-in-time local or remote repository copy of an already existing repository, as well as the indispensable security related features such as access control. Source code management tools maintain and guarantee the integrity and consistency of the repository during any operation. Git has become the most prevalent source code management software in the last decade, initially used as a repository in the development of the Linux Kernel and in other areas more related with software development, but lately being widely adopted across different areas of IT related with infrastructure and configuration management, nowadays widely used in the context of cloud

computing as a repository for IaC resource definitions or configuration files and other more advanced forms of cloud architectures.

Storing Terraform or IaC configuration files with a configuration repository is common practice in more massive or declarative forms of cloud deployments and a Terraform workspace environment can be coupled and synchronized with a Git repository, adding to the aforementioned features the possibility of having automated computational processes or any other authorized parties to access resource definitions. Git uses the concept of local copy for anyone that has pulled or cloned a repository contents into its local machine or working area, allowing for the modification of contents locally and just committing such changes to the repository when appropriate. The ability to work on local copies of the repository without needing any network access, that becoming necessary only when synchronization with the main repository is to take place, gives enormous flexibility to developers or anyone working on such code or resource definitions. Assuming an available Git repository for centralizing IaC resource definitions is available and authorizations are in place, an appropriate workflow for the Terraform configurations previously outlined can be augmented with Git as described in Diagram 2:

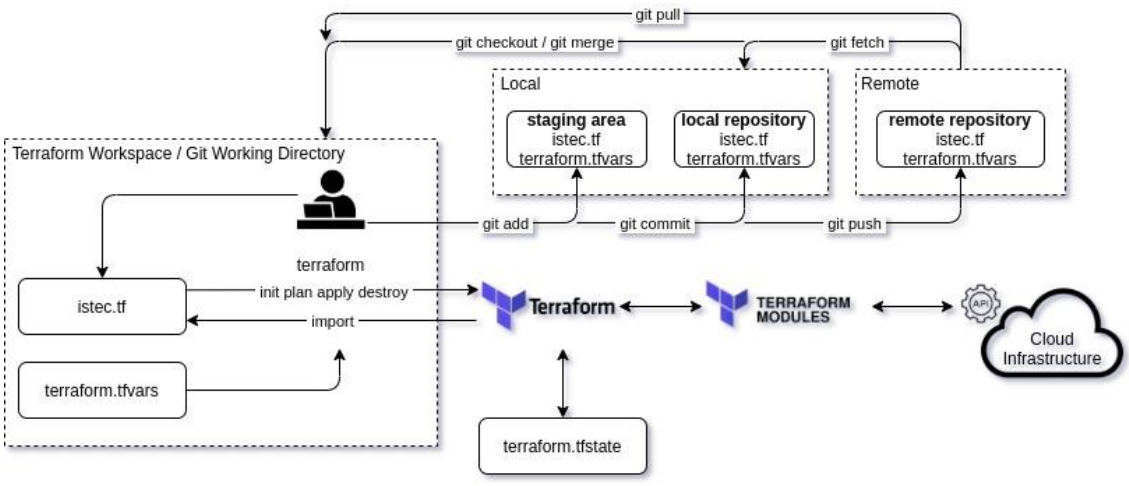

Diagram 2 Using Terraform with Git repository

After initializing a Git working directory on the same location as the Terraform workspace and editing the *istec.tf* resources file with the desired resource definitions according to selected provider, independently of triggering terraform actions, the *istec.tf* and *terraform.tfvars* files (or any other files under the working directory for that purpose) can be kept on the repository by invoking *git add* specifying which files should be put on the local staging area, with those becoming from that moment under local Git supervision, marking them to be tracked for changes.

Concerning Terraform, the *terraform.tfstate* file must be excluded for technical reasons as it preserves state related with local Terraform invocations and should not be shared among different Terraform instances or developers, serving as a concrete example that not everything that is local data is to be put on the repository or under revision control, but possibly under an ignore category. After all desired changes have been done on the *istec.tf* file, a *git commit* operation effectively saves the file under the local repository, awaiting eventual synchronization with the remote repository. Having files in the local repository is akin to a local cache or index of the remote repository, and any discrepancies between the local repository and the remote repository can be verified and operated on. Files under the local repository can be synchronized with the remote repository with a *git push* operation and should any conflicts arise, manual intervention for merging contents may be necessary after careful inspection of changes and differences. A *git pull* operation would retrieve any files existing on the repository but not on the local cache, possibly created by other developers or processes, making them available locally. Git controls files through their checksum instead of their contents, which is calculated whenever a file within the staging area is added to the repository and put under Git control. Git detects any changes done to files within its staging area by recalculating their checksums and by comparing those checksums with previous commits. Should any differences in the checksum arise, several operations are possible such as displaying differences between files or merging their contents. Git will also warn if any changes to files being tracked have been made without updating those into the repository. Since any changes to the *istec.tf* resource configurations will have their previous versions of the file kept as new changes are synchronized with the repository, this pattern of operation provides a complete history of any infrastructure modifications across time. The possibility to rollback committed operations, a feature not natively provided by Terraform due to its idempotent philosophy, also becomes possible with the use of source code management tools.

Combined Practices

The previously mentioned technologies for defining cloud resources in a programmatic manner (IaC) and for keeping such definitions and configurations under a centralized repository (SCM) are crucial for achieving a higher independence from the cloud regardless of the chosen provider and independently of the desired architecture. Although a seamless transition of artefacts between cloud providers as it would be desired is still far from being achievable until there are defined standards for such transition, using these technologies gives possibility to repeat the creation of artefacts in a consistent manner even if some degree of changes are necessary. The combination of Infrastructure-as-Code, a more recent technological practice, with Source Code

Management, a practice that has decades of existence, is a clear example on how it is possible to combine tools and methodologies for a common purpose and provide new and innovative ways to deal with existing hurdles. Both these technologies are indispensable for aiding in cloud provider independence, as it will be described. The following case scenarios assume a traditional IT infrastructure based on virtual environments with common IT solutions, since it would be uncommon, although not impossible, to migrate from a container-based Cloud-Native or Microservices architecture into an IaaS or PaaS type of cloud infrastructure.

IaaS Migration

The IaaS model for cloud infrastructure allows for the creation of virtual machine environments with their associated network contexts and storage elements, analogous to traditional IT architectures but entirely in virtualized form. By migrating already existing virtual machine images into such infrastructure, those resources can start benefiting from the cloud computing model characteristics. The IaaS model is appropriate for a relocate or rehost type of migration into the cloud, as described under the common strategies for migration. Regardless of creating new environments or migrating existing ones, adjacent network and storage virtual elements have to be created or defined within the context of the cloud infrastructure as well.

Migration to an IaaS cloud has been made easier due to the number of already existing virtual machines that resulted from a transformation with physical to virtual (p2v) methodologies in the previous era of computing, when virtualization was becoming mainstream, or that have already been created on top of virtualization solutions that became ubiquitous in the last decades such as VMware, Hyper-V, Xen or KVM. This has made the transition to cloud infrastructure somewhat similar to the movement of virtual machines between hypervisors, a process commonly known as virtual to virtual (v2v).

Assuming that the existing baseline infrastructure is in a virtualized state, the migration of such virtual elements into an IaaS based cloud model depends on the type of virtualization technology underlying the existing environments and the one supported by the cloud provider, since different types of hypervisors have their own incompatibilities. Some virtualization solutions may be supported for direct relocation of virtual machines, similar to what is known as collocation in traditional IT. Usually, a compatibility matrix from the cloud provider describes the virtualization technology used as well as the type of virtual machine images supported for migrationm and if some level of transformation or conversion of the original image is necessary

in the process. Virtual machine images are usually based on common supported formats such as: Open Virtualization Archive (OVA); Open Virtualization Format (OVF); Virtual Machine Disk (VMDK); Virtual Hard Disk (VHD/VHDX); XenServer Virtual Appliance File (XVA); Virtual Disk Image (VDI); RAW. It is possible, with appropriate tools, to convert between those image formats in order to comply with hypervisor specifications in terms of supported images. Before booting such images other technical details such as the type of boot, depending on the use of EFI or legacy BIOS, as well as the use of paravirtualized drivers must be taken in consideration. Additional specific compatibility requirements such as type of operating system and its version are also commonly identified, including networking and storage specifications necessary for compatibility and interoperability. If importing already existing virtual machine images is not possible, a new installation of the operating system and subsequent reconfiguration may be necessary. Under this scenario, having a configuration management solution such as Ansible is useful for reapplying operating system or applicational configurations, possibly with some adaptation. Other specific constraints may exist such as portability of the software running on existing virtual machines, but those are beyond the scope of this dissertation.

Despite those identified constraints, assuming an assessment has been made according to the premises described in the general assessment guidelines for a migration plan and the compatibility matrix is satisfied, the necessary steps to perform the migration from an infrastructure perspective are usually straightforward and well documented by providers, sharing common characteristics even across different implementations. This type of migration can present different options during its execution in-between the described general guidelines for IaaS migration. The procedure tries to be as cloud-vendor neutral as possible and resorting to IaC whenever possible or appropriate, having steps numbered for further reference:

General Guidelines for IaaS Migration

1	Creation of necessary network contexts and storage elements within the provider infrastructure according to requirements or specifications using IaC
2	Definition of virtual machine characteristics and associated resources in their respective network contexts using IaC
3	Evaluation of additional necessary configurations for interconnecting artefacts between migrated and non-migrated environments
4	Migration of virtual machine images by exporting and transferring their images for

	subsequent import, possibly subject to minimal adjustments or to some transformation process to conform to the compatibility matrix of the cloud implementation
4.1	Alternatively, creation of new virtual machines with approximate characteristics and separately migrate or recreate existing OS image related data and configurations
5	Attachment or configuration of any additional necessary storage or network elements
6	Transfer of any related applicational data into storage within the cloud infrastructure to the newly created/migrated environments

Table 2 General Guidelines for IaaS Migration

All actions regarding the preparation and configuration of infrastructure for migration into an IaaS cloud model, besides having to adhere to the general guidelines outlined for IaaS migration, must be done by resorting to IaC languages such as Terraform and such resource definitions must be kept in a source code repository like Git. Both these technologies were previously described.

Regardless of migrating already existing virtual machine images or creating new ones, the adjacent environment in terms of networking and storage components also have to be defined on the cloud provider infrastructure and should be done in a programmatically manner, as well as kept within the repository since such definitions will not be made available or exported by the provider in any other format later that would allow importing into another cloud infrastructure. Those resource definitions will be handy for a future migration, even if subject to any adaptation. The creation of such artefacts, the migration or creation of virtual machines and the interconnection of all those resources will comprise the resulting implementation in terms of compute, network and storage components, also known as an IaaS cloud architecture. The migration or deployment and configuration of applicational components followed by subsequent testing on such infrastructure would come next, but such actions are outside the scope of this dissertation.

The migration of a standard on-premises infrastructure as exemplified in Diagram 3 into an IaaS cloud model can have its equivalent or similar resource definitions declared in Terraform as exemplified in Annex 1. Diagram 3 describes such infrastructure in a simplified manner.

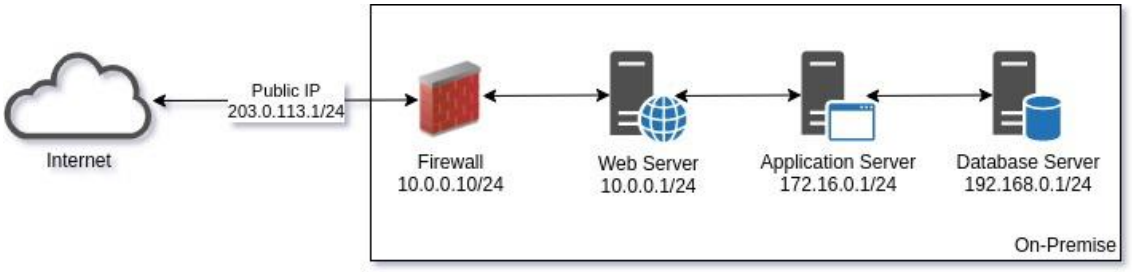

Diagram 3 Simplified On-Premises Infrastructure

Annex 1 exemplifies Terraform resource configuration files with resource descriptions reflecting the IT components of Diagram 3 in a non-rigorous manner, having the filename reflecting some relatable context of these resources. After creation of the Terraform resource files exemplified in Annex 1 with the respective resource definitions, those files are to be kept in the source code repository using Git, before invoking Terraform for the creation of such resources in the cloud. When Terraform is invoked with *plan* it will report on what resources or artefacts are to be created or changed with its output in Annex 2 of this document. If *apply* is chosen, such resources are subsequently created on the selected cloud provider as described. Resources will be put in their respective contexts with virtual machines in their associated network subnets and also with their individual storage components attached, as described in Diagram 4. Minimum security rules are also implemented with those depending on the default security policy according to the provider. This would comply with steps 1 and 2 of the general guidelines for IaaS migration.

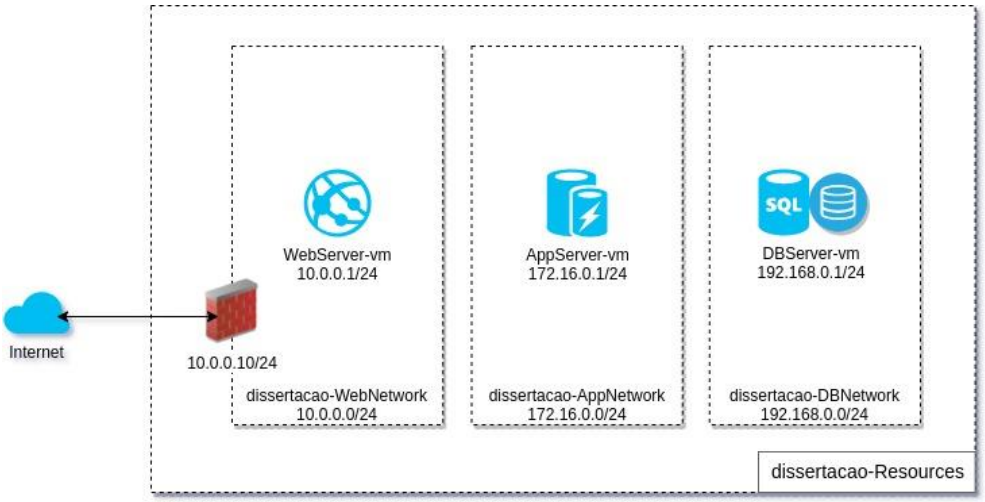

Diagram 4 IaaS deployed using IaC

After defining the new cloud infrastructure reflecting the existing on-premises infrastructure as closely as possible by using IaC, further changes should continue to be performed

inside the realm of IaC and synchronized with the repository. Code within the repository can not only be referenced and adapted later for applying similar configuration objects on another cloud provider, but also to expand IT infrastructure if necessary, leveraging a higher degree of independence. The configurations outlined on step 4.1 of general guidelines for IaaS migration are always specific within the context of migration due to applicational constraints or requirements and subjective to an analysis on a case-by-case basis, which could even lead to a migration from an applicational perspective.

The use of a configuration management solution is beyond the scope of this dissertation but as previously mentioned, it is an important asset on any IT infrastructure in any migration scenario for restoring or performing a rollback on any configuration change, or for reapplying previously existing configurations on cleanly installed environments in a reliable manner. Having a process for keeping configuration changes in a centralized repository is therefore crucial if it becomes necessary to reapply configurations in such newly deployed environments, assuming equal or compatible versions of operating system or applicational software. Steps 5 and 6 generally describing additional configurations and the movement of related data are also specific to the context of migration and will depend on external factors such as network connectivity and bandwidth. The main aspects of migration and its related infrastructure is however highly simplified with IaC, in a somewhat portable manner.

The migration to an IaaS cloud infrastructure is one of the most straightforward and well documented forms of migration. By planning a migration to an IaaS type of cloud infrastructure resorting to IaC it is possible to improve planning through detailing of necessary artefacts and their relationships as well as benefit from the features of IaC such as a higher level of consistency and faster deployment. Using IaC also provides more alternatives for migration or expansion in the future, by referencing and reprocessing configuration information about the existing infrastructure in the repository, enabling a higher chance of independence from any cloud provider. Since deployment of an IaaS type of cloud infrastructure can also be a pre-requisite for other types of cloud deployments that can depend on the provisioning of virtual machines through automated methods, such as those for use with container-based virtualization, IaC also provides a mechanism for orchestrated deployments according to demand.

While migrating to an IaaS based cloud infrastructure is a relatively simple process, once migrated the infrastructure is not portable between different providers, a major drawback in terms of future migrations if they become necessary. Even with the limited options available to migrate

into another provider or to rollback to on-premises infrastructure, some cloud providers have already developed and made available options to export virtual machine images existing within their infrastructure according to some of the standard formats mentioned previously, making it possible to somehow do the process in reverse. Still, all other artefacts associated with the IT solution such as storage or network elements would have to be recreated from scratch, a task that can be made easier when all resource definitions were previously done by resorting to IaC and kept in a source code repository for reference, even if some degree of adaptation is necessary. Despite not being a perfect solution, by resorting to IaC in the context of migration to create an IaaS based cloud infrastructure it becomes possible to, although not as seamlessly as desired, migrate into another cloud infrastructure or provider by referencing the entire configuration and readapting/recreating the necessary artefacts into the new infrastructure somewhat consistently, making IaC indispensable for such tasks, beyond its initial purpose.

PaaS Migration

The PaaS model is based on deploying ready-to-use instances for database, middleware or other similar components resembling a traditional multi-tiered model of IT architecture (although not exclusively in that context) abstracting the entire underlying infrastructure layer. The PaaS model is appropriate for a replatform or refactor type of migration into the cloud as described under the common strategies for migration. Migrating to a PaaS model may not be as straightforward as migrating to IaaS since configuration changes to applications or databases may be needed, possibly including some code refactoring, although at a manageable level.

This cloud model allows control over application design, but not control over the underlying physical infrastructure, shifting the development of IT solutions by concentrating on database, middleware or other IT components available under PaaS offerings as well as their associated development tools, instead of infrastructure components, providing a layer of hosting for cloud applications. Multiple constraints or drawbacks in deploying database instances or middleware components under the traditional on-premises IT infrastructure make this service model attractive. Usual hindrances such as time for deployment and configuration, administration and management with complicated patching matrixes, the need for additional infrastructure to provide backups and disaster recovery, along with limited scalability, are among the most common issues. Those contribute to rising costs even when the infrastructure is in an already virtualized form. A PaaS type of deployment is preferable for some architectures or solutions that may require such dedicated instances, possibly even as a complement for some architectures

already running on cloud.

Several products based on a PaaS service model may exist beyond database or middleware instances, depending on the cloud provider. The PaaS model can provide more flexible management and scalability of instances, with providers usually offering advanced features such as dynamic increase of available computational resources depending on the database or application load (or upon request) for providing rapid scalability, typically without any service disruption, and features in terms of management, like automated patching and backups, commonly associated with their own proprietary components or offerings. Depending on the provider or cloud infrastructure, and limiting the example to database or application server components, a compatibility matrix usually identifies what type of source database or middleware components are supported for migration, with the process usually well documented by cloud providers especially if the existing implementation is from the same vendor.

The PaaS model of cloud computing is more prone to a vendor Lock-In due to the fact that most cloud offerings for this type of service are usually vendor-specific and tend to perpetuate the dependency on a specific product even when running on cloud infrastructure. Despite the advantages presented by this model, there are substantial differences between cloud provider offerings and Vendor Lock-In may already exist before the migration, since each database or middleware component has its own specificities depending on the vendor providing it, if not based on an open source product. Careful analysis should be done before choosing a solution in terms of database instances (commonly defined as DBaaS) or application servers that rely on proprietary schemas or programming languages, thereby transposing the existing Lock-In into the cloud since once data is imported into a vendor-based or proprietary solution a substantial transformation or refactoring process has to be done to export such data or applications into another type or model. Some PaaS offerings may be provider specific and should therefore be treated as a final product similarly to SaaS.

If a PaaS service model is to be chosen, in order to achieve a higher degree of independency from the cloud provider, preferably an open-source based database engine or middleware platform should be selected. If technically feasible, a conversion process for open-source based database solutions should always be considered whenever the cloud provider instances are not compatible with the originally exported schema (meaning that some refactoring already has to be done) or if the schema of the original data is based on a proprietary solution, lowering technical debt if such refactoring is done earlier. By converting to open-source based

solutions, future migrations may become easier if they ever become necessary.

A migration to a PaaS service model is usually done to partial components of an existing IT infrastructure. Assuming an assessment has been made for those components according to the premises described in the general assessment guidelines, especially concerning steps 2 and 3, and the compatibility matrix is satisfied, the migration process consists of exporting data or middleware artefacts (commonly database exports or application archives) from instances to be migrated, transfer such data or artefacts into the cloud infrastructure, and subsequently import it within their respective contexts on the newly created PaaS instances. These operations may be easier if the vendor currently providing those components for on-premises solutions is the same provider for cloud infrastructure, since there is a decreased risk in compatibility problems. If technically supported, data can also be migrated by resorting to a synchronization process between the existing database instances and the instances in the cloud in order to minimize downtime. These types of solutions are typically proprietary and would have to be analyzed on a case-by-case basis. Similarly to an IaaS migration, a reconfiguration of other existing IT solution elements may be necessary in order to connect those to the newly created instances, depending on network configurations and application requirements. The following general guidelines for a PaaS type of migration are numbered for further reference:

Generic Guidelines for PaaS Migration

1	Definition of database or other middleware components on cloud infrastructure according to specifications or requirements using IaC, if available
1.1	Optionally, definition of instances using appropriate cloud scripting tools, if IaC is not supported
2	Placement of aforementioned instances in appropriate network context using IaC or scripting if IaC is not supported
3	Configuration for interconnecting existing IT components to new instances
4	Migration of data or applicational components by exporting and transferring their contents for subsequent importation, possibly subject to some transformation or refactoring process to conform to the compatibility matrix of the cloud implementation
4.1	Optionally, if applicable, migrate data by configuring replication process between existing database instances and instances running on cloud
5	Planning and reconfiguration of existing IT infrastructure for connecting to new

	instances

Table 3 Generic Guidelines for PaaS Migration

The migration of IT components into a PaaS model has similar aspects in terms of procedures with the previously described migration into an IaaS model, but at its core consists of exporting and importing data, abstracting the underlying infrastructure in terms of compute, storage and network elements. A PaaS model rarely comprises the entire IT solution, usually being only a part of it, coupled with other types of cloud artefacts under other cloud models or even with on-premises infrastructure in a hybrid-cloud model.

On some cloud providers, PaaS database and middleware components can be defined by resorting to IaC templates similarly to the IaaS migration model, and should preferably be done in such programmatically manner with the objective of making any future migrations or recreation of artefacts simpler under the same reasons already mentioned for IaaS migration. Support for the creation of PaaS components through the use of IaC may not be available, optionally creating those with the provider's command line tools or shell equivalents, which still makes it possible to create such resources programmatically but at the expense of portability since such code is specific to the cloud provider and therefore not directly applicable for recreating resources on another provider whenever needed, without readjustments. Still, despite the fact that the use of specific provider command line or shell features to communicate with its API is not optimal in respect to portability, by storing those configuration scripts and have a history of their changes through the use of a repository, a history on the creation or modification of such resources is available. Such information can be useful for future recreation of those resources if necessary, even though such definitions may have to be translated into another provider language. Other features usually present in IaC are also not available when scripting is used without additional development, such as having information like the one provided by Terraform about resources, reporting on what changes are to be done on them. Steps 3, 4 and 5 concerning the configuration and migration of data are always dependent on the context, just like in IaaS contexts, and must be analyzed on a case-by-case basis.

The migration of database and applicational components (data and application archives respectively) residing on the virtual machines of the standard on-premises infrastructure exemplified in Diagram 3 into a PaaS cloud model can have those PaaS components previously created through IaC resource definitions declared in Terraform as exemplified in Annex 3, or optionally through any other IaC language supported by the cloud provider for that effect,

reflecting what is generically described in Diagram 5. Alternatively, if IaC is not supported, some equivalent scripting language supported by the provider can be used. Regardless of the method, these configurations must also be kept within the repository for later reference.

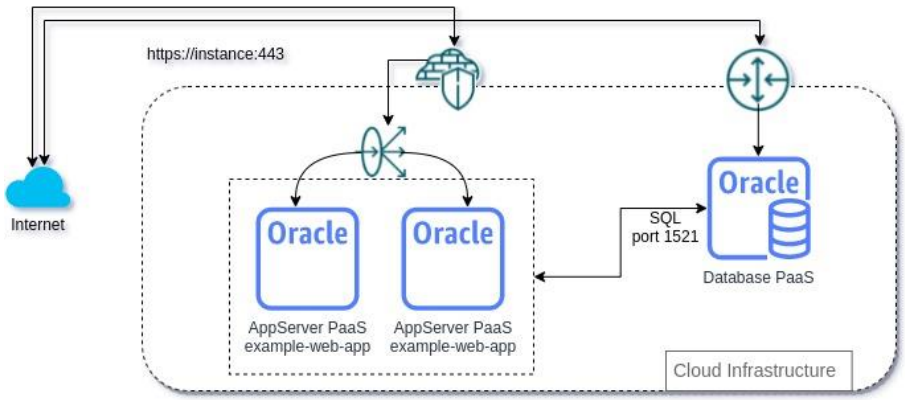

Diagram 5 PaaS deployed using IaC

Under the PaaS model for cloud computing, it is difficult to achieve a true provider independence since most service offerings are specific to the provider and therefore it may become a dead-end for further migrations without substantial efforts with refactoring. Additionally, most offerings may incur in extra costs beyond the PaaS instance, such as costs for backups usually into cloud storage, assuming information on the database is not for ephemeral purposes, or even for transferring those backups to local storage if necessary. Regardless of those constraints, as was the case with IaaS, a higher degree of independency can be attained by doing every configuration and keeping them in a code repository for later reference, using IaC whenever possible, and only if IaC is not available through the provider's own scripting languages for manipulating artefacts through their API. Although recreation of infrastructure artefacts may become easier by referencing such information, the PaaS model is usually the most limiting for migration when code or data related with PaaS instances is from a proprietary nature and therefore not possible to directly export and import into a newly created instance in another provider.

IaaS and PaaS Limitations

The previous migration methods that leverage cloud computing under the IaaS and PaaS service models are among the most common and straightforward methods of cloud migration, however they possess some limitations in terms of flexibility for future migrations and don't fully exploit other features and capabilities that became available in the cloud computing model.

Regardless of improving the outcomes of future migrations by using IaC and keeping every configuration done on the code repositories, any future migrations under these two service models will always have issues to contend with, such as incompatible virtual machine images in the case of IaaS that make the transition of those images between different cloud providers difficult or cumbersome, possibly always subject to a conversion process, and the adjacent storage and network configurations are also not directly transposable between providers in an IaaS model with some level of adaptation needed. The migration is also subject to additional requirements such as having to comply with specific operating system versions to satisfy the compatibility matrix of the provider, which may consequently cause portability constraints on the applications running on them. Although not directly related with migration aspects, the need to have fully dedicated resources on the running virtual machines is not optimal since charging is done for the entire set of resources even in periods of low activity, unnecessarily raising the costs with infrastructure.

The PaaS model overcomes some of those disadvantages but goes contrary to the idea of computing as utility due to the fact that migrating from a PaaS cloud implementation later may reveal to be even more difficult than from an IaaS one, especially if refactoring of data or applicational components is needed, which is likely to happen under proprietary offerings, seriously hampering any future migration. Although it provides better resource utilization and improvements in terms of administration and manageability, the drawback is becoming indefinitely tied to a given provider since most PaaS solutions are usually provider specific, somewhat like a final product.

CaaS Migration

Container virtualization is based on the principle of having an abstraction layer not only over the entire underlying infrastructure or any of its components, as that would be similar to any standard virtualization, but also completely abstracting the operating system layer. Herein lies one of the biggest differences, as the operating system and its associated compatibility issues affecting portability and interoperability can be overcome using container technology. Code or applications are developed and packaged specifically to run on top of such abstraction layer. The layer of container virtualization becomes similar to a PaaS model, but contrary to the PaaS model which runs specific instances of specific applications, mostly proprietary, containers can run anything, providing an open model for development and deployment. Based on container virtualization and under such premises, a CaaS service model has more advantages for any

migration between cloud providers, eliminating or minimizing portability and interoperability constraints between different implementations of CaaS infrastructure, as long as minimal compatibility requirements are met. Since a container-image can run without change in different cloud providers, assuming a compatible implementation of container technology, this enables a considerable degree of cloud provider independence, more easily attainable than in previous forms of cloud deployment. Consequently, as a result of this architectural change, a CaaS model also brought a different approach for developing, deploying and running applications on cloud infrastructure that could take better advantage of the cloud computing model, providing faster deployment and higher flexibility in terms of resource allocation, resulting in better scalability and lower expenses with infrastructure. Development under these new paradigms is not mandatory and components of an existing N-tier or multi-tiered application may also be suitable for transposing into this type of service model, which classifies as a rebuild/reuse under the common strategies for migration.

Applications running on container virtualization have their code assembled and packaged with all its dependencies together, in what is defined as a container-image. A container-image may be a complete application, or just part of one large application distributed across several container images. The latter commonly defines a Microservices architecture, or possibly some related form of a distributed architecture. The CaaS model resorts to deploying those ready-to-run container images on a container-based virtualization platform, which can even be built on container-native IaaS. Such platform runs container engines, responsible for code execution as well as resolving any dependencies related with the infrastructure layer, on top of an operating system and fully abstracting it. The container-based virtualization platform that supports the container engines is itself usually comprised of virtual machines running on the underlying infrastructure, having to comply with a specific version of an operating system that has the sole purpose of running and supporting a specific implementation of a container-runtime engine. This container-runtime engine is deployed with the single purpose of supporting the launch and execution of those ready-to-run self-contained images. This type of architecture, from a virtualization perspective, may classify as a form of nested virtualization.

Assuring portability and interoperability is the responsibility of the container-runtime engine, with applications bundled on container images not being dependent on the operating system running on the virtual machine, instead relying on the container-virtualization technology being used. If a container-runtime engine is supported between different operating system versions, then container images running on top of such runtime engine should run seamlessly

without any constraints when moved from one infrastructure into another. This is a crucial aspect to consider from the perspective of migration and cloud provider independence.

Containers take up less space than virtual machines and are launched or booted much faster. Multiple containers can run on the same container-engine implementation, sharing the OS kernel and resources with other containers. When a container-image is instantiated or launched, each container runs as an isolated processes in user space. This characteristic can also influence migration options, and may set forth some specific requirements in terms of orchestration or scheduling of those containers under more massive deployments, due to the fact that resources are shared. Under this model, instead of having to provision and launch additional VM's, which is suboptimal in terms of speed and resource allocation when compared with containers since VM's take longer to boot and continue to consume resources even in periods of lower activity, additional containers can be launched and terminated much faster. It is also possible to preemptively launch more virtual machines for supporting and running an additional number containers only when needed, keeping resource allocation to a minimum. The necessary provisioning of underlying virtual machines to support a container infrastructure and providing such scalability can be achieved by resorting to IaC templates and automation, similarly to what was described previously for IaaS. Some cloud implementations and providers have already extended service offerings, providing container-based infrastructure for direct deployment and running of containers, scaling out automatically as containers are deployed without any provisioning of infrastructure necessary by the client.

In order to migrate to this type of architecture, considerable changes in the overall architecture of an existing IT solution under a client-server model, N-tier or monolithic architecture may be necessary. Under a migration scenario, and assuming the baseline architecture is not already in a containerized form, the switch to this paradigm of computing for migration has a higher complexity and possibly higher cost initially due to the refactoring and transformation of existing applicational components in order to comply with this model. Choosing this type of migration may lower technical debt, as any future movements between different cloud providers can more easily be achieved, and even take advantage of new cloud computing paradigms such as Multicloud deployments. If existing IT components on the baseline architecture are already in containerized form, it could result in an easier migration, somewhat similar to a scale-out operation to another cloud. As a rough comparison, v2v was to the movement of virtual machines between different hypervisors as moving a container is between different container-runtime engines.

For a CaaS type of migration, steps 1 to 3 under the general assessment guidelines are not as straightforward as for previous migration methods, and unfold into a deeper analysis that involves changes not just from an infrastructure perspective but from an architectural and development one. Candidate applications or IT solutions to migrate to this type of cloud computing model may reveal to be relatively easy or somewhat complex to migrate, depending on several technical factors and knowledge about existing IT infrastructure, ultimately affecting how existing components adapt to this model. The applicational components running on the candidate infrastructure to be migrated must be decomposed in a way that would fit one or more container images and adapt to a container model. This is oversimplifying, since migration scenarios may require that different paradigms of development and architecture must be observed and taken into account under such analysis, in order to understand the tradeoffs and ultimately conclude if such transformation is viable or even desired. This process may grow in complexity as it may go beyond the simple transposition of the applicational component into a container-image. Under more complex migration scenarios, components are most likely to need some deeper refactoring process. Refactoring implies code changes to adapt to some different form of computing, without changing the final outcome or behavior of the component and presumably without discarding the existing code base, minimizing risk. This refactoring process may consume many available resources and its associated effort must be carefully evaluated. It is possible to use container virtualization with monolithic or N-tier architectures based on the client-server model of computing, but that is suboptimal.

The following guidelines for CaaS migration augment, from a generic perspective, the general assessment guidelines and identify aspects that may have to be addressed before planning the migration, regardless of the desired architectural pattern being based on a traditional approach and continuing to be based on a client-server model or monolith even when running in containers, or based on a Microservices approach or some other form of distributed architecture.

Evaluation Guidelines for CaaS Migration

1	Evaluate technical viability of candidate applications for transposing to container-based virtualization and migrating into CaaS
1.1	If applicable, also evaluate necessary changes in existing IT architecture to support conversion of applicational components to CaaS
2	Identify and measure development and refactoring efforts needed for transposing such

	components to comply with CaaS architecture
3	Deploy or subscribe to necessary container-virtualization infrastructure and configure container-image repositories
3.1	Optionally, assure readiness of required container-virtualization platform according to provider including storage and network requirements
4	Refactor and deploy code to container-image repositories
5	Deploy container images

Table 4 Evaluation Guidelines for CaaS Migration

Despite the selected container technology, the use of IaC and code repositories are implicit in a CaaS based approach. Each technology may have its own implementation of IaC for declaring artefacts, with some even supporting the use of generic solutions such as Terraform. Regardless of the language or implementation used, the same development principles presented in other forms of migration continue to apply. In addition to repositories for keeping code related with IaC declarations, as exemplified in previous migration methods, a repository to store container images is also necessary. The type of repository for IaC declarations or code is not the same as the one for keeping container images, with the latter depending on the technology used for container-based virtualization.

Under CaaS, the mechanism for deployment becomes interrelated with applicational development. The cycle of development encompasses the entire process from code development to its packaging and deployment on container registries for subsequent deployment on ready-to-run infrastructure. As mentioned, this type of deployment is expected to be independent of the provisioning of the underlying infrastructure, contrary to the traditional paradigm of IT on IaaS or PaaS cloud models, or at the very least providing total abstraction on how it is provisioned, as long as the cloud provider or cloud infrastructure solution supports the chosen contained-based virtualization technology. From a development perspective, after code is staged and appropriate testing is done, assuming the results are successful, components are packed into a container-image and committed into the image repository after the packaging process has finished. This step of deploying code into a container registry becomes associated with part of the development process, being the last action of it, commonly identified as the integration phase. Subsequent deployment from the container registry into the container-based virtualization infrastructure for the code to run is part of the deployment phase, when an applicational container-image is pulled into a container-engine which sets up its associated environment, followed by the instantiation of the image processes.

Diagram 6 depicts a generic development and deployment workflow for CaaS using Docker, a popular container virtualization solution and one of the most used for container virtualization which will be used in the examples on this dissertation. It is important to understand such workflow in order to best understand a migration process into this model. The workflow describes container-image creation and deployment with subsequent instantiation of the container-image into memory, running as a container instance, considering an application that fits the requirements for being transposed into a container model and that can effectively be converted into it.

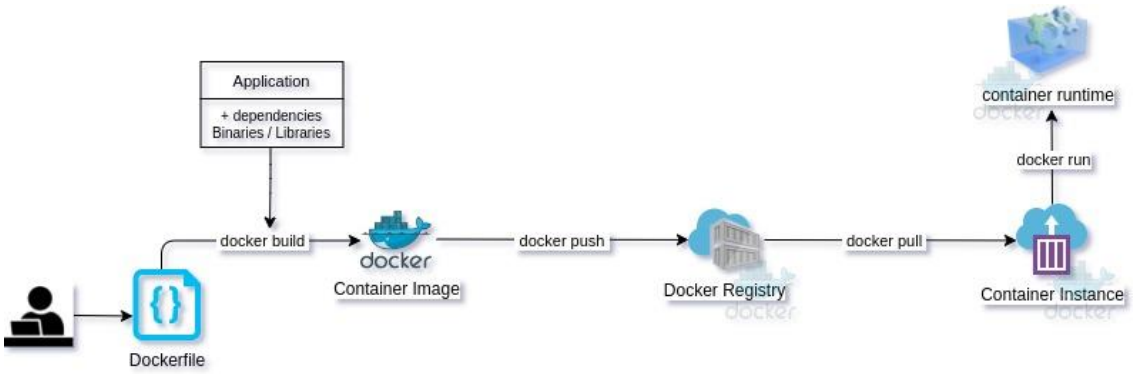

Diagram 6 CaaS using Docker

Within an implementation of container virtualization using Docker, the contents of the container-image will be described using its own IaC syntax within a file known as *Dockerfile*. This file will describe what will comprise the final container-image in terms of not only what it contains in terms of applicational components but also additional content-related metadata. Other necessary actions associated with the buildup of the container-image during the image creation process and its build context, or during container instantiation when launched from the container-engine are also described in the *Dockerfile*. All *Dockerfile* configurations, similarly to what was previously done in other migration scenarios, are to be kept within a repository such as Git for versioning and later reference, should it become necessary. This will also become important for more complex types of development and deployment requiring automation across all stages of development, testing and deployment. Both networking and storage layers are abstracted from the example and beyond the topic under investigation and involves understanding adjacent concepts related with container-based virtualization such as bridging, overlay networks, data volumes, among others.

Just as the previous migration methods, any applied configurations are expected to be kept in a configuration repository for later reference, which in the previous example is done through the Dockerfile. This is especially relevant from a migration perspective under this model for the recreationg (if necessary) and redeployment of containers on a different cloud infrastructure in the future, assuming the same type of container-engine implementation. Similarly to previous migration scenarios on other type of cloud models, cloud migration can be made simpler by reusing such IaC configurations kept on the Git repository. Under a CaaS migration it becomes even more straightforward by simply reprocessing and redeploying into new container registries associated with the new provider. It is also possible to redeploy or pull images onto a new cloud infrastructure assuming a connection between the new cloud infrastructure and the existing one can be implemented, similarly to a hybrid-cloud model, instead of redeploying new container images on a new repository, at the expense of having to keep older repositories under the existing infrastructure.

This type of migration it is more appropriate for improving provider independency, and it is recommended to keep the container-image repository under local control under a hybrid cloud implementation, permitting access from the container engines of any new provider to the container-image repository under local infrastructure is optimal for all migration or scale-out scenarios involving CaaS cloud model. This type of solution for the container repositories is compelling not only from an independence perspective, by keeping information related with the infrastructure for any future recreation under local control, but also to take advantage of other features in the types of migration or implementations such as Multicloud, described later.

It can be seen that the CaaS cloud model is one of the cloud deployment models that can effectively provide a high degree cloud provider independence by taking advantage of the container virtualization model, abstracting the entire underlying infrastructure, but at the cost of having to transform the existing IT solution or IT components to comply with such model. This type of cloud model implies the use of IaC and methodologies for both development and operational management of adjacent infrastructure, in which DevOps becomes a relevant, although loosely defined, and well-established methodology.

Microservices under CaaS

Some of the identified constraints under the IaaS or PaaS cloud models can be overcome by switching to CaaS. However, when it comes to migration, not all IT solutions or components

are suitable candidates for this type of cloud model, due to the associated transformation or refactoring needed on the original solution, and the nature of the original application to put under this model may not be adaptable for such changes. A detailed procedure for evaluating and decomposing the existing IT infrastructure or application in order to fit a container-based approach is beyond the scope of this dissertation and only relevant aspects in terms of migration from a macro perspective will be mentioned. Hence, a deeper evaluation must be made before choosing this type of architectural model when compared to the previous migration methods, as mentioned in the general guidelines for CaaS migration. A Microservices based architecture is one possible approach and is the one briefly described. A migration into this model complies with a rebuild/reuse or even rewrite type of migration described under the common strategies.

On planning to migrate existing IT solutions to a CaaS model, a proven architectural model should be taken in consideration, especially when refactoring is involved. A Microservices pattern or architectural style is among the most common for breaking a monolith application or IT solution into multiple independent components. A Microservices architecture is based on small and independent modules or services, each having a smaller code base with a specific functionality on the overall architecture, utilizing some form of messaging model to establish communication and synchronization among those independent components or optionally using specific APIs for such communication. Components should be developed or refactored in a loosely-coupled way so that failure of one would not compromise the entire solution and be deployable independently so that they can run in a distributed architecture that could optionally scale-out to more than one cloud infrastructure, contributing to a higher degree of provider independence. Each component can be developed and deployed independently, with its own test suites and data. A Microservices architecture also introduces some different patterns in terms of networking and storage configuration and management, with persistence in storage not being so prevalent. By implying a different form of development and management of cloud artefacts, this computing paradigm also imposes overcoming a considerable learning curve in order to adapt to it.

FaaS / Serverless Migration

The FaaS type of cloud deployment, commonly defined as Serverless computing, is in itself a special form of CaaS since most implementations of FaaS rely on the same container virtualization principles but apply it differently for a very specific domain within a cloud infrastructure. The FaaS model does not use or depend on IaC templates or other mechanisms to

describe infrastructure since there is conceptually none, instead assuming there is one already deployed ready to execute code. The underlying infrastructure is similar to the one used on a CaaS model since it also depends on container-engines running on top of virtual machines.

One of the main objectives of the Serverless approach was to eliminate the need for provisioning of infrastructure or having any other concerns associated with it, even at the container level, completely shifting focus to development efforts and having code or applicational artefacts running directly on top of a cloud infrastructure, completely abstracted of any infrastructure components underneath. This is conceptually similar to the CaaS model, but under FaaS there is no perception of instancing a container in the traditional sense, with that happening under a different set of conditions in an abstract and completely transparent manner to the end user. Code developed and deployed to work under this model can be triggered to run under a specific event or invoked directly using an API, without any necessary action concerning infrastructure layers. This type of approach is appropriate for executing specific chunks of code that perform specific tasks for a given set of events or invocations, or for a given amount of time, with the execution time possibly being limited by the implementation. This execution model also contrasts with CaaS on the perspective that under CaaS containers are deployed and supposed to be running for longer periods of time whereas under FaaS the container that encapsulates the code to be run is terminated as soon as execution finishes. Migrating IT components into this model also classifies as a rebuild/reuse type of migration under the common strategies, but rewriting code may become necessary. This type of service model also contributed to a further transformation on the development, deployment and execution of IT components running on cloud, resulting in a deeper form of abstraction and more efficient charging for resource consumption.

Even though it is based on the same technology used for CaaS, or container-based virtualization, under the FaaS model infrastructure components on top of virtual machines are comprised of container engines that run container images for a specific type of FaaS implementation. From an infrastructure perspective, a container-image is deployed for running a specially crafted container-instance, which in turn supports interpreting and running code elements of some programming language depending on the ones which are supported for the specific FaaS implementation in use. Code deployed to run under FaaS is launched not with the objective of running a general purpose long-running application container, which is common under the CaaS model, instead being typically short-lived and expected to last only for a specific timeframe, or for executing some well-defined task a given amount of times, with the instance

terminating as soon as execution finishes. Charging is calculated by some metric associated with running time or number of invocations, reducing resource consumption and costs even further when compared with the CaaS model. This represents a different use on the layer of container virtualization, since under most FaaS implementations the cloud infrastructure is itself running container engines just for the specific purpose of supporting a given FaaS implementation and programming language, this becoming the final layer of service.

Most cloud providers already made available FaaS or Serverless solutions for a given number of supported languages, with each provider having setup a complete infrastructure based on some FaaS implementation ready for consumption and to run code. Additionally, some providers have also made available their own specific functions or chunks of code to perform generic tasks through their appropriate API using a FaaS approach, as a complementary service to their cloud offerings. Using such provider functions however may be prone to Lock-In unless the source code is made available and completely based on an open-source implementation, therefore avoiding proprietary languages is recommended. This is an important aspect to consider in terms of migration that can affect cloud independence. Just as it was the case with CaaS, a FaaS model will not be appropriate for any existing application or IT component, being even more restrictive in what should be transposed or migrated into this type of computing model.

Beyond proprietary solutions, most providers also support open-source languages in their FaaS implementations for developing and executing code, especially the most popular ones such as Python, Perl, Go, JavaScript and others, as long as the code complies with the specific implementation of FaaS and its guidelines for development under this model. Choosing an open language is not sufficient to have complete independence as the FaaS implementations can have their own idiosyncrasies even for the same programming language, akin to what happens with CaaS in terms of container-engine implementations and their differences. This can impact migration, and from an independence perspective code should be developed by resorting to open languages under an open implementation of FaaS such as OpenFaaS or Fn-Project, guaranteeing that code can run on any other cloud that supports the same open implementation. The general guidelines associated with a CaaS migration partly resemble the ones for FaaS, however under FaaS deeper refactoring or rewriting efforts are needed and their impact on migration should be evaluated in a thorough manner.

Guidelines for Implementing FaaS

1	Evaluate technical viability of candidate components for refactoring or rewriting to container-based virtualization based on FaaS
1.1	If applicable, also evaluate necessary changes in IT architecture to support and integrate with refactored or rewritten components under FaaS
2	Identify and measure development efforts needed for refactoring or rewriting of such components to comply with FaaS architecture
3	Deploy or subscribe to necessary container-virtualization infrastructure and configure container-image repositories
3.1	Optionally, assure readiness of required container-virtualization platform suitable to the chosen FaaS implementation according to provider
4	Refactor and deploy code to container-image repositories
5	Deploy container images for subsequent invocation

Table 5 Additional Guidelines for Implementing FaaS

Diagram 5 exemplifies a generic implementation of FaaS or Serverless using the Fn-Project FaaS implementation, not only describing the necessary infrastructure but also the common flow of development and deployment under such model.

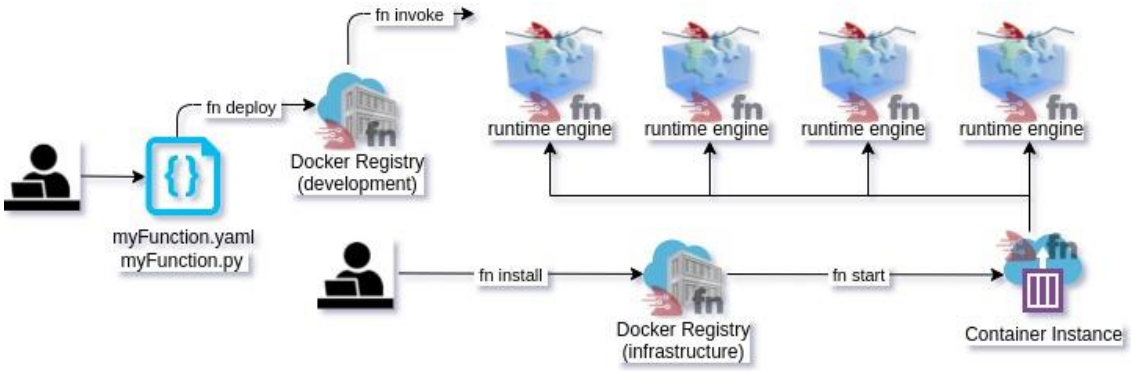

Diagram 7 FaaS using Fn-Project

From an infrastructure perspective, even though it is totally abstracted from the developer, the Fn-Project container-engines can be deployed using a CaaS implementation such as Docker, with a specific container-image residing on the “infrastructure” Docker registry that has the runtime components for supporting and providing a runtime environment for a given language when instantiated. After instantiation, the container-engines become ready for having code deployed onto them for processing. In the Fn-Project implementation of FaaS, code is defined in its respective source code file, named in this example *myFunction.py* for a source code

file having Python code, with additional metadata about the program in the file *myFunction.yaml* having to obey to a specific YAML structure with specific tags for describing additional information necessary for deployment. Code developed to work under this model may also have specific requirements and constraints in terms of core development, such as specific input/output processing. As development is finished, code is packed and deployed as a container image into an appropriate repository, described as the "development" Docker registry in the diagram. Having separate repositories is not mandatory and development could use the same repository as the one used for infrastructure container-images of the FaaS implementation, but separation of development and infrastructure components is considered a good practice. Code can be invoked in various forms. The container-image on the "development" repository is pulled whenever some specific trigger or event takes place, optionally through an API, executing the code within it on top of the FaaS container-engines and exiting as soon as execution finishes or some timer expires. Charging may be done on execution time, number of invocations or some other metric about code execution and result.

A deeper understanding of the details about the chosen FaaS implementation and what it supports is crucial before any refactoring or rewriting process, as well as having a deep understanding of how the chosen components to be put under this model work, with this being beyond the scope of this dissertation. Extensive documentation is available and converting or adapting any existing IT component into this model will have to fit such programming paradigm as well as follow the structure and rules for the implementation in use. A migration process under FaaS should also be done under the same principles of development in terms of repository usage, and just like the previous migrations scenarios all code should be kept in a centralized repository such as Git.

Although similar to the CaaS implementation, a FaaS approach may not be as good as CaaS in terms of cloud independence, especially if some proprietary or opaque form of FaaS based on a specific provider offering is used. Some providers also have extensive libraries of functions for different tasks ready to use, with such functions becoming attractive from a developer's perspective since not having to develop some specific functionality that may already be available lowers development efforts, limiting however the movement or migration of such functions or artefacts into another provider. Still, by applying the same principles used for previous migration scenarios, keeping all code changes in a locally controlled source code repository, with additional refactoring or rewriting of such functions these can possibly be transposed to another provider, but never without substantial effort.

Rearchitecting to Cloud-Native

Choosing cloud computing for the deployment of new IT architectures or to migrate existing IT solutions can have different approaches as it was described in the previously described migration scenarios, with some resembling traditional IT infrastructure such as in IaaS or even PaaS, with the latter simply delegating the management of infrastructure to some third-party. Other approaches such as CaaS, FaaS or similar types of cloud architectures based on container virtualization are more abstract on what constitutes the underlying infrastructure and more closely related with the cloud vision in terms of what it should represent as a commodity, due to the possibility of a higher level of independency from a given provider. All forms for deployment of cloud infrastructure and their related IT architectures have their own pros and cons, and when it comes to migration, depending on the context and the result of a thorough evaluation of each, some may reveal to be more appropriate than others.

A Cloud-Native architecture, although not being directly related with cloud migration practices, since new developments under a Cloud-Native approach are not necessarily concerned with the migration of existing IT solutions or artefacts, cannot be dissociated from any migration process due to the fact that whenever refactoring or rebuilding components is necessary, it is currently one of the most well-accepted and viable methodologies to take into account, being the preferred choice for new cloud-based IT solutions.

Migrating an existing IT solution, regardless of its origin, into some form of cloud computing model approaching a Cloud-Native architecture classifies as rearchitect under the common strategies, because the solution relies on transposing existing components possibly into more than one service model, essentially based on container virtualization such as CaaS or FaaS. This usually requires a substantial development effort, possibly even a partial or complete rewrite of applicational components. Some Cloud-Native proponents regard IaaS and PaaS or similar service models not as part of a Cloud-Native approach due to some disadvantages under those service models regarding their potential for Lock-In and how they allocate resources, due to their long time for provisioning and instancing when compared with CaaS or FaaS, therefore not providing the same flexibility and elasticity as the container-based models and also having a more rigid configuration mechanism. Some FaaS implementations are also prone to Lock-In depending on the underlying technology used to implement it, so it is debatable whether those should be considered part of a Cloud-Native approach. Others are more open to the use any of the available

service models under very specific cases or circumstances, depending on solution requirements.

Understanding the principles of development and deployment of IT solutions under a Cloud-Native architecture is crucial to understand its implications on migrating or transposing an existing architecture into this model. Adopting Cloud-Native approach for a given IT solution or architecture implies embracing a different paradigm of computing concerning its components, somewhat similar to a SOA architecture. From a development perspective, a Cloud-Native approach imposes breaking down the various parts of an application and its components and implementing such functionality based on a Microservices architectural pattern, with the objective of fully exploiting the characteristics of the cloud computing model from its inception on top of a container-based virtualization infrastructure, mostly CaaS. The service model can be combined with other types of service models, such as FaaS/Serverless, if more appropriate for any component of the solution.

As a development pattern or architectural style, contrary to a traditional IT approach of monolithic applications based on a N-iter architecture or client-server model, a Cloud-Native solution based on a Microservices architecture consists on decoupling functionality into such multiple small components with each having a well-defined task, distributed across the infrastructure that provides an established layer for allowing communication between them, which can be based on push/pull mechanisms. The components are written in a manner that failure is expected without compromising the entire solution, providing higher resiliency, and be able to be updated or deployed independently. Synchronization of the various components can also be done by resorting to event messaging mechanisms, typically asynchronous by nature, or through some component specific API endpoint using RESTful mechanisms. This pattern of development poses some difficulties concerning the overall state of the solution, sometimes using some intermediate solution to persistently store component state or any other relevant data that might have to be shared between several components, which are preferably stateless by design. By having small independent components with well-defined tasks, this also results in a smaller code base for each. Components can be managed by different independent development teams, having their own build and deploy methodologies and with each component also having their own test suites. This enables a more agile response from development, faster verification and resulting deployment. A higher scalability also results from this approach, benefiting from the apparent unlimited resources of the cloud computing model.

From an operational perspective, this paradigm also provides a better observability and

monitoring per component and a higher perception of its performance through individual measurement. From an infrastructure standpoint, coordination with development is crucial for choosing among the several service models based on container virtualization the one that best fits for a given IT architecture in terms of its deployment given the nature of the application, subsequently adopting deployment and operational practices best suited to such service model.

Cloud-Native architectures are not without their own drawbacks. One of the major drawbacks is a higher complexity due to the nature of the solution having multiple components that have to be kept well-orchestrated, resulting in a distributed architecture. Distributed architectures by themselves have their own peculiarities already known in other realms beyond cloud computing, presenting difficulties in choosing the right model for sharing data or state between its components, as data tends to be decentralized, possible network performance constraints such as high latency and complex dependency resolution between components when it is not possible to have them completely independent, are just some of the difficulties presented by this model. From an infrastructure and operational perspective, additional learning efforts to understand concepts such as overlay networks and layered storage, among many others, are also required.

Possessing a deep understanding of how the application to be decoupled works is crucial for any migration plan under a Cloud-Native approach. It is necessary to involve development teams as this type of migration cannot be undertaken just from an infrastructure perspective, contrary to other service models or IT architectures. Diagram 8 depicts the breaking down of a traditional application into a Microservices based pattern built on CaaS and FaaS.

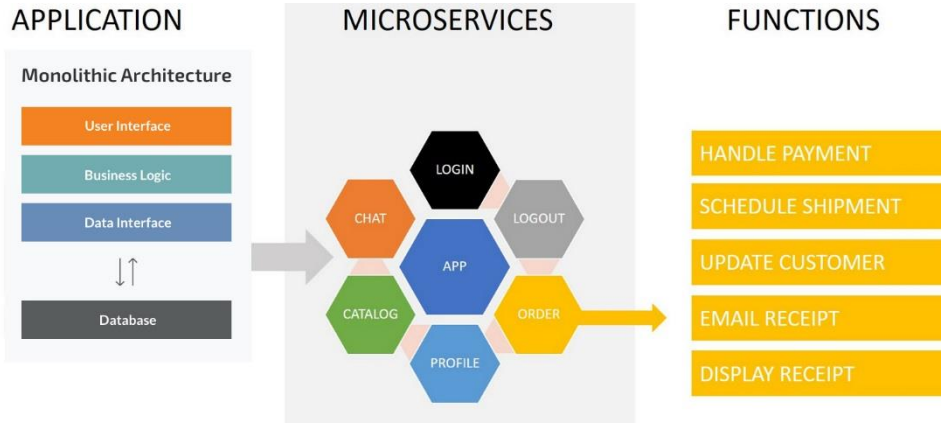

Diagram 8 Decoupling to Cloud-Native

Rearchitecting an existing application to fit into this model represents the most difficult type of migration as it involves refactoring and rewriting components, eventually turning a large share of the migration process akin to new developments. A complete successful migration of several IT components or artefacts in an existing IT infrastructure into CaaS or FaaS can eventually conform to a Cloud-Native architecture, assuming a successful migration of the entire set of candidate components into such models, according to the architectural definitions.

XaaS Migration

A migration that encompasses any type of service models such as IaaS or PaaS for some components or CaaS and FaaS for others, or whatever service models a cloud provider may have available for that matter, can qualify as XaaS or anything-as-a-Service. XaaS is a relatively recent description for any IT architecture running on cloud not being tied to one service model or to a category of those, being instead very unrestrained in terms of service models chosen for the architecture as long as those are more appropriate to a given type of problem and may fit the requirements for a solution without considerable constraints. Under the primary objective of moving everything to cloud first and optimize later, a XaaS architecture may help in a more rapid cloud adoption as components can possibly be migrated into cloud, always trying to find the right balance between relocating and rewriting, at the cost of not having the highest benefits of the cloud computing model in terms of flexibility, scalability and efficient resource allocation, but still reflecting on lower overall costs with the IT solution.

Rapid cloud adoption is usually one of the main reasons for approaching XaaS, and therefore viable for any migration. However, this type of approach may not take into account the possible Lock-In resulting of some of the choices made, according to those already presented under their respective models such as IaaS or PaaS, typically leaving such concerns for later analysis. A Cloud-First first and foremost approach to any new IT solution or development is usually combined with XaaS if the priority is to use cloud computing. Such approach can also combine a Cloud-Native architecture with Microservices development paradigms, if achievable. From an independence standpoint resorting to other service models such as IaaS or PaaS should only be done when absolutely required or justifiable.

Regardless of the chosen service models when adoption a XaaS approach, they should individually abide to the guidelines presented for each.

DevOps in Cloud Migration

Development under the cloud computing model requires learning new development methodologies as well as readapting old ones. At the same time, the management of the overall IT architecture and its underlying infrastructure under this paradigm of computing brought forward additional challenges not easily solvable under the traditional approaches for IT administration. The entire software development lifecycle and the infrastructure components to support it required more efficient mechanisms to streamline its management, from development to deployment, as the traditional style of software development and adjacent IT administration was ill-suited for this new paradigm of computing. Although not directly related with cloud migration, DevOps is indispensable for cloud adoption, especially under the most recent models such as CaaS and FaaS, with DevOps methodologies having an important contribution to cloud migration scenarios especially under these service models.

The definition of what DevOps means or what it represents is subject to different interpretations despite commonly agreed aspects on what it aims to achieve. Within the several interpretations around DevOps there are two main aspects to consider, the cultural aspect aiming for a more efficient interaction among development and operational teams, focusing on the importance of communication between them for increased agility and faster response to incidents, and how its technical implementation is done through the use of multiple tools depending on context. The technical implementation of DevOps may have different approaches and use different tools, depending on the expected outcomes and on the context of the IT solution.

Implementing DevOps

From a generic perspective, the typical model of software engineering encompasses the traditional phases of development, followed by system integration testing, user acceptance tests and subsequent deployment into production. It can be seen that these steps may be inadequate under the current development and deployment models for cloud computing when considering a Cloud-Native or Microservices paradigm, as the application or IT solution is architecturally different and broken down into multiple components, each being deployed independently, with dependencies among those components becoming more complex to work out.

Transposing the relevant phases of traditional software engineering into a functional model under the cloud computing paradigm is still necessary, since the phases of software development lifecycle continue to apply. Some of these phases, such as testing, require readapting

as the conventional testing strategy for any changed components may not work not without some rethinking on the overall testing mechanisms, possibly having to adapt to a Microservices based pattern, with individual components having dependencies from other components that may be short or long lived and located elsewhere when restarted, both in terms of infrastructure and network.

As the number of components that comprise a given solution increases, each having its own independent development cycles, it is imperative to adapt processes related with software development, testing and subsequent deployment with some kind of automation having carefully defined checks and constraints embedded into the automation process, so that all those steps from development to deployment have consistency and guarantee of success.

In terms of management of the adjacent infrastructure, contrary to the traditional management model of IT administration where components like virtual machines or services are commonly identified by some name or established nomenclature and managed individually, components or artefacts under these cloud architectures are no longer named or managed directly. The advent of programmable infrastructure using IaC under the cloud model brought rapid creation and destruction of cloud artefacts mostly by automated means without the need to manage those individually, with many of them even being ephemeral, especially when associated with IT solutions based on the Cloud-Native or Microservices architectures. All those characteristics when ultimately combined with the scalability provided by cloud computing, made standard deployment and management approaches somewhat obsolete.

The search for newer solutions and mechanisms to overcome such challenges made DevOps one of the preferred approaches for management under the cloud era for the complete software development lifecycle and its associated deployment on cloud infrastructure. Although not originally rooted in cloud computing, DevOps is currently one of the most accepted methodologies for management especially when under a Cloud-Native approach, fully integrating the development cycle of IT components with its adjacent infrastructure details from development to deployment. DevOps is rooted in agile methodologies and in a way, it can be said that in the cloud era, Cloud-Native architectures are to development as DevOps is to augment its management.

Although not directly related with cloud migration, DevOps methodologies must be referenced in that context since regardless of migrating to cloud or natively adopting it for a new

IT solution, DevOps is becoming ever more interconnected with deployment and management of any cloud architecture and it is an indispensable methodological approach whenever cloud computing is considered. Despite the fact that the main purpose of DevOps methodologies under a normal context are related with the entire SDLC in a cloud architecture, from development to deployment, it can also be applicable for more advanced migration scenarios, providing some mechanisms that can also be useful for streamlining cloud provider independency.

DevOps Pipelines

Two key aspects of DevOps are toolchains and automation. A technical implementation of DevOps can be used to automate from the simplest scenarios of plain deployment of an artefact into a repository after its build is done (build, deploy and run), or for complete integration and testing among the several components of an entire IT solution during its build process and subsequent deployment and replacement of running instance with a new one. Automation under DevOps permits the use of additional tools and mechanisms coupled to the stages of the SDLC, with multiple tools available for coupling at any stage and their implementation depending on the desired automation level. DevOps used the concept of pipeline, akin to a factory, where the several stages associated with development all the way to deployment and instancing of cloud artefacts take place and can have adjacent processes associated with each stage. Some implementations may desire that only part of the process is automated, others may implement full automation. The most common stages of the pipeline are generically defined as:

- CI – Continuous Integration
  Encompasses the development and build stages of the SDLC, including unit testing, and subsequently deploy the created artefact into a repository.

- CD – Continuous Delivery
  Previously created artefact which was deployed onto the repository after development is subject to the integration testing phase, according to some designated plan.

- CD – Continuous Deployment
  Depending on the result of the previous testing phase, artefact is subject to acceptance tests and depending on the result of those may be pulled into production for substituting the one currently running with a new instance.

Diagram 9 depicts the three typical generic pipelines under a DevOps methodology, describing the several levels of automation possible within the entire cycle from development to deployment. The three pipelines describe different incremental levels of automation starting with Continuous Integration (CI), Continuous Delivery (CD) and Continuous Deployment (CD), independently of what intermediate steps will take place and what tools are used in those steps for augmenting functionality.

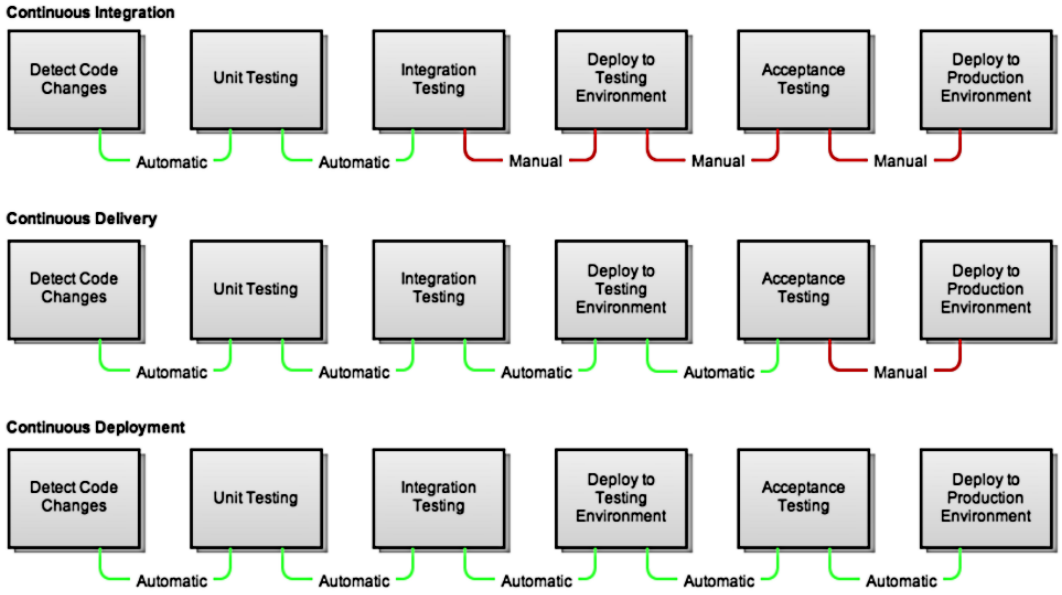

Diagram 9 DevOps Pipelines and Automation

As depicted, the desired automation level to be implemented on a DevOps pipeline can be complete or partial, with the latter whenever some manual confirmation or procedure is desired during the process.

DevOps Toolchains

Through the use of hooks throughout the pipeline, the coupling of actions with other external tools becomes possible, providing more advanced forms of configuration, deployment and control of cloud resources or IT components and enabling the creation of more complex workflows with additional degrees of automation and orchestration. Leveraging these mechanisms for invoking actions whenever any resource configuration is made on the various sections of the pipeline, triggering the subsequent launching of adjacent processes, permits the creation of advanced testing, integration, provisioning and delivery, laying the foundations for

more complex technical implementations of DevOps.

Diagram 10 describes a generic implementation with common example tools, namely Git, Python/Pytest, Jenkins and Selenium for additional processing along the several stages of the pipeline, with container technology being based on Docker:

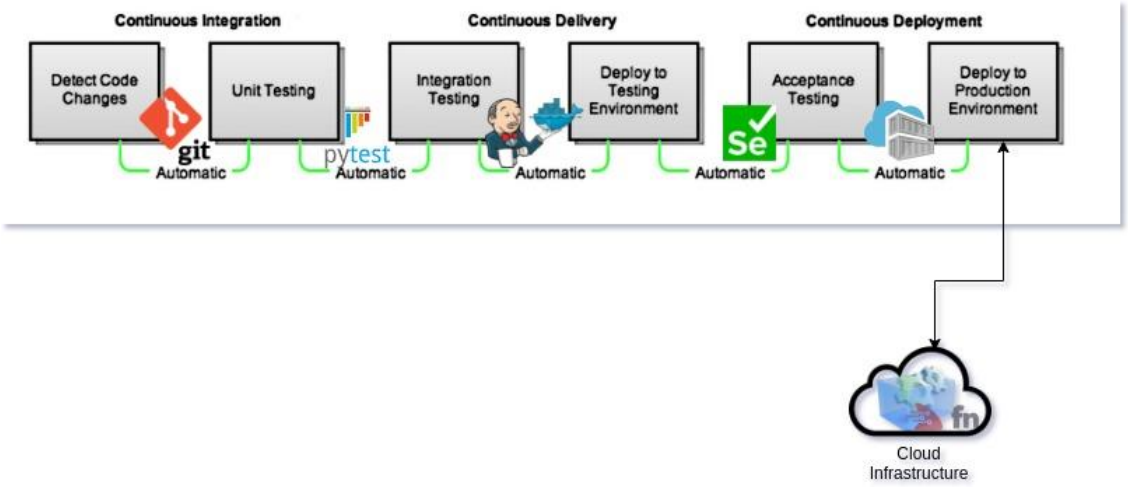

Diagram 10 DevOps and Tools

Details concerning the development and testing phases, namely Continuous Integration and Continuous Delivery, are not of much relevance to the topic of this dissertation as migration under an infrastructure perspective is more concerned with the Continuous Deployment phase. However, a brief description of the steps involved in the DevOps flow is in order since, although not likely, some of those steps can influence a migration process. The coupling of applications to perform specific actions in the several stages of the pipeline is given in the example diagram according to their context (development, testing or deployment). Given the flexibility of this methodology, coupling other types of applications for additional features such as automatic documentation or reporting is also common.

DevOps Flow

Whenever a commit is applied with new code into the development repository, a complete automated flow of events take place. A Git hook triggers the automatic unit testing using Pytest for such code, submitting it for integration testing depending on the result, finishing the Continuous Integration (CI) phase. Assuming no errors, Jenkins is invoked to perform all the necessary integration testing of the new code with the surrounding environment, and if there are

no identified issues automatically deploys a new artefact onto the repository, finishing the Continuous Delivery (CI) phase. Lastly, when a new deployment is done onto the repository a trigger generates the required usability testing using Selenium and should all tests end successfully a deployment of the new container-image is done onto the deployment repository. By entering a new image onto the repository an automated process of deployment takes place, replacing the running component with a new version of it. This exemplifies a completely automated DevOps pipeline. Of all the stages, the deployment phase, or Continuous Deployment (CD) is the most relevant phase in terms of cloud migration, since under a Cloud-Native or Microservices pattern of cloud computing, deployment can be done on different providers or on an on-premises private cloud infrastructure. As mentioned, first stages of the pipeline related with the development and testing phases under a migration scenario, at least from an infrastructure perspective, are not so relevant, unless specific development or testing actions may influence the chosen provider at the deployment stage. Under those circumstances these steps must also be taken in consideration.

Conceptually, the refactoring efforts necessary to adapt IaC code from a given cloud provider onto another can be implemented in an automated form by also using a DevOps pipeline or hooking into some part of it, similarly to the development process previously described. Such pipeline would need some form of standardized and provider-supported reference data describing the possible artefacts for a given cloud provider taxonomically classified. An automated process coupled to such pipeline could compare and refactor such code under some given parameters that could define origin and target provider. This could be useful for coupling some form of migration for artefacts especially under IaaS or PaaS service models, making it possible to automate the refactoring process of their associated IaC definitions and reapplying on the new cloud infrastructure after such refactoring.

As demonstrated, container-based virtualization technologies allow for a higher independence from a given cloud provider, with this being highly beneficial in Cloud-Native or Microservices architectures. Under this paradigm, it becomes possible to have a truly distributed architecture spawning more than one cloud infrastructure with a high degree of scalability and resiliency. The coupling of container-based virtualization features and DevOps methodologies brought the possibility of having automated deployment onto a different cloud provider or infrastructure from the deployment phase of a DevOps pipeline. This is especially relevant under a Multicloud deployment, having important implications not only in terms of provider choice, but consequently for any migration perspective, as described further under Multicloud Deployment.

Multicloud Deployment

Cloud service models based on container virtualization technologies solved the problem of interoperability and portability among different cloud providers, assuming a compatible implementation of container-based virtualization. This solved the problem for provider independency (although not completely) and contributed to the possibility of having an IT solution running on a different cloud provider or infrastructure without any refactoring being necessary, offering additional options for deploying IT solutions.

A Multicloud deployment is based on the paradigm of distributed architectures and implies having an IT solution under such assumptions, technically devised in a way that is supported within the realm of container-based virtualization. Given those technical characteristics, a Multicloud architecture is based on deploying an IT solution not just in one cloud provider, instead deploying and distributing its components in more than one or multiple cloud providers, improving the resiliency, flexibility and redundancy. Conceptually, a Multicloud deployment is not restrained to the CaaS or FaaS models as it can use a XaaS approach as long as the chosen service models fit the requirements and the identified constraints are taken into account, despite some service models having constraints upfront regarding their lower flexibility to move between clouds. As a result, a Multicloud approach can leverage any service model, but some of these models may hinder cloud provider independence and be prone to Lock-In, although they can still be valid even under a Multicloud architecture.

Additional challenges arise when choosing to deploy a given IT solution or architecture based on a Multicloud approach, such as interoperability issues when components deployed in different clouds need to establish communication among them, becoming necessary to adapt network configurations or any means of communication necessary between components, to such model of deployment. Those issues can be addressed with specific networking equipment or API gateways already tailored for this type of implementation, having configurations for rerouting traffic appropriately. Other solutions may also combine the use service discovery mechanisms through service registries where the components that are part of the solution register themselves whenever available, with status over their availability, combined with some publisher/subscriber models or through an API. Given the ephemeral nature of some components and the volatility of their network properties, these issues have to be carefully evaluated from a development and infrastructure perspective before choosing this type of cloud deployment or migration pattern.

Several other issues focusing on development aspects under a Multicloud architecture are however beyond the scope of this dissertation.

Assuming the necessary network configurations are in place and a totally independent Microservices-based application that can be fully deployed onto another provider without any dependencies, migration of a Cloud-Native architecture can be done by simply redeploying components into the new provider. Diagram 11 augments the previously described diagram exemplifying a completely automated pipeline with commonly used tools for some of the stages, and where components of the IT solution would be distributed between several cloud providers. The deployment phase can be adjusted whenever necessary to proceed with deployment for another chosen infrastructure, as exemplified on Diagram 11

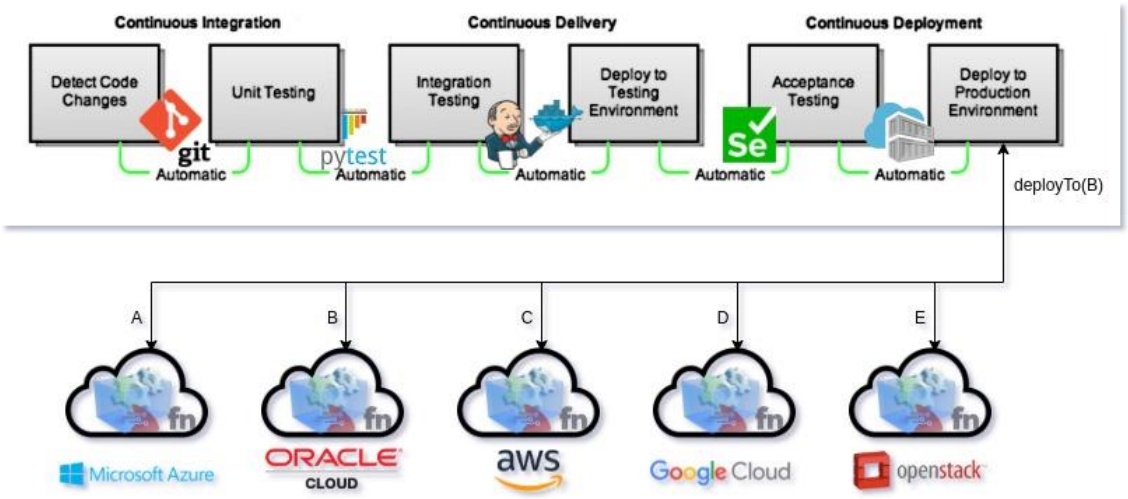

Diagram 11 Multicloud Deployment with CaaS/FaaS using DevOps

During the deployment phase, a mechanism for cloud provider selection can be implemented in order to deploy the solution or artefacts into the chosen provider infrastructure. Network configurations or any other aspects subject to necessary intervention due to the fact of deploying onto another provider could also be hooked in the same deployment stage of the pipeline, using an automated approach, or through some other alternative means. This in itself can be a methodological approach for cloud migration benefiting from provider independence, coupling the entire SDLC with choice of cloud provider. Although this could conceptually be done using other cloud service models such as IaaS or PaaS, as described in the DevOps flowm, it would not work without refactoring such components before the deployment phase in order to adapt the description of the artefacts and make them compliant with the chosen cloud provider.

This would provide additional difficulties and cumbersome to maintain.

An implementation of such DevOps pipeline under a hybrid-cloud model allows for a better control of IT resources, with core infrastructure for development purposes deployed On-Premises in order to keep critical development data under more control. This can also be a requisite from a compliance perspective in terms of critical or sensitive data. DevOps pipelines automate and orchestrate deployments to public or private cloud allocating the resources needed, as well as their respective teardown when no longer necessary, taking full advantage of the cloud computing model in terms of its scalability and flexibility, with migration from one provider into another also becoming easier by just adapting and redeploying. The combination of these technologies shifts the focus of requirements from infrastructure into development, as the infrastructure layer can be completely abstracted and no longer a concern for the developers or "Cloud Programmers" within the whole solution. Additionally, within this context, a migration scenario can be highly simplified and completely controlled through programmatic means

Migration Framework

The following migration framework aggregates and correlates the cases described under this dissertation, with the objective of providing a summarized overall perspective of the various migration scenarios possible along with their required level of effort, always with the objective of achieving a higher level of cloud provider independence using specific practices. Migrating to cloud computing should be subject not only to a technical evaluation, but also backed by a strong business case supporting such decision, this is however beyond the scope of this framework. Some other aspects that must be taken in consideration regarding cloud migration which are not reflected into this framework are the specifics of necessary development efforts in refactoring or rewriting components, which should be done according to adequate methodologies and current best practices, difficult to quantify as it depends on the context of each IT architecture or solution. Additionally, no inference is made on which specific public cloud providers should be chosen despite their features.

This framework does not contemplate migrating from more modern technologies back to older types such as from CaaS to IaaS, although possible, those migrations are marked as "Not Applicable", with an exception made from PaaS to IaaS or from FaaS to CaaS migrations, which can still make sense in very specific circumstances.

Some additional general migration guidelines should be taken in consideration, along with the initial general assessment guidelines reflected in Table 1, all in line with the proposed migration framework. Some of these guidelines are crucial for achieving a higher provider independency, regardless of the chosen service model:

- Use a phased approach for migration
- Keep all artefact definitions on a source code repository (eg: Git)
- State configuration for all artefacts using IaC if appropriate for service model (eg: Terraform)
- Use a centralized configuration management solution (eg: Ansible)
- Prefer cloud service models based on container virtualization for better portability
- Adapt IT components suitable to a Cloud-Native architecture by refactoring according to design patterns for SOA and cloud computing
- Avoid unnecessary containerization (eg: complex monolithic applications)
- Evaluate supported operations under provider API's
- Prefer widely deployed open-source based components
- Avoid proprietary cloud solutions

The following table summarizes the relationship between source and target architectures and attempt to provide an overall perspective on the several aspects concerning cloud migration under this dissertation. Two distinct contexts of cloud migration are considered, from On-Premises to Cloud (typically a first phase) and a Cloud-to-Cloud (when changing provider).

Migration Framework Reference

	IaaS	**PaaS**	**CaaS**	**CaaS / FaaS**
On-Premises Traditional IT	General evaluation - Table 1			
	Relocate/Rehost Table 2	Replatform/Refactor Table 3	Rebuild/Reuse Table 4	Rebuild/Rearchitect Table 5

	IaaS	**PaaS**	**CaaS**	**CaaS / FaaS**
	General evaluation - Table 1			
IaaS	Convert v2v Refactor IaC	Export / Import data Refactor / Redeploy code	Rebuild / Deploy to CaaS	Rebuild + Rearchitect to CaaS / FaaS
PaaS	Install VM with application Export / Import	Export / Import data Refactor / Redeploy code	Rebuild / Deploy to CaaS	Rebuild + Rearchitect to CaaS / FaaS
CaaS	N/A	N/A	Redeploy Redirect DevOps pipeline	Redeploy + Refactor Redirect DevOps pipeline for CaaS
FaaS	N/A	N/A	Refactor + Rebuild Redeploy Image	Refactor + Redeploy

Table 6 Migration Framework Reference

The steps involved in a cloud to cloud migration can be made easier if any previous migration from On-Premises has been done having all configurations kept for reference. Any of the described migration process do not imply that everything has to fit a certain service model, as components or applications can be distributed over several service models as described on the XaaS approach for cloud computing. However, the relevant steps for each individual service model continue to apply.

Conclusion

The lack of standards for cloud migration that could ease migration operations between providers for common service models is an important but missing part for the cloud computing vision in terms of how it was originally envisioned with its ubiquitous nature - as a utility. Such inexistence of standards results in the need for a careful analysis and anticipated planning whenever cloud computing is to be adopted under the common service models, with the aim of making migrations easier in the future should they become necessary. It is evident that choices made concerning cloud infrastructure and service models for new IT implementations can have an enormous impact on the ability to move them to another cloud provider or infrastructure later. Regardless of the decisions made for new projects, the lack of standards for those operations also impacts any currently existing cloud infrastructure independently of how it has been implemented, and under these circumstances the planning for any migration should be well thought out before any decision is made in line with reducing any future cloud provider dependency during the process.

Due to the different types of technologies and architectures involved as well as the constraints and challenges each pose in terms of their migration, it becomes difficult to have a standardized approach for those operations and highly complex to create one that could be applicable to all situations. Extensive literature has been created on this subject in order to better understand in which ways cloud migration could be made easier, attempting to provide solutions to the problem, clearly recognizing the existence of such difficulties and identifying several important aspects related with it and raising awareness to the Lock-In problem. Proposed solutions to the problem have been made in different contexts, although it is noticeable that much of the literature approaches the topic from a development perspective. Additionally, some of the reviewed literature appears to be somewhat outdated concerning the latest trends in cloud migration from an infrastructure perspective, not reflecting the latest solutions that have been developed to deal with the problem, some of them with high degrees of success and solving some of the most prominent issues such as portability, especially under the adoption of some specific service models.

While there is no existing framework that can be applicable to every scenario related with cloud migration and even in specific types of migration there is no standard or formally defined way for such transition, a more general framework such as the one presented under this dissertation can augment other existing literature which already provides important contributions to the topic from a development perspective. Due to the impossibility of migrating cloud artefacts

from a given cloud provider to another by just moving or exporting and importing them, with portability issues arising under some common models, the current framework resorts to the recreation of such artefacts on those, as migration could not be done otherwise, with some refactoring needed especially for IaaS or PaaS service models. It is debatable if this can be considered a migration in a strict sense, but understandably this seems a viable option when using appropriate tools and the suggested methodology, given the required level of consistency. Within this scenario, as noted, the importance of having a configuration repository with all the information on artefacts that have been created became evident as being of utmost importance for recreating everything on another cloud infrastructure, or for any other situation where analyzing the currently existing infrastructure through its associated code is relevant.

It can be seen that although all service models are valid for new cloud deployments, some are more prone to the Lock-In problem and portability issues arise especially with IaaS and PaaS making transition from cloud providers difficult. The advent of service models based on container-virtualization changed this, and the CaaS and FaaS service models have greatly solved the problem of Lock-In if planned and implemented according to some guidelines that take such issue into account, such as choice of container-based virtualization technology and keeping all configurations on a repository. It became evident as the investigation progressed that those newer service models, despite not being standardized, are commonly approached not only because of being more efficient in terms of resource allocation and flexibility, but also due to the easier movement of artefacts between cloud infrastructures, bringing the aforementioned benefits in terms of cloud migration. Although those service models are mainly approached because of their technical characteristics, their ability to provide a true cloud provider independency stood out as one major benefit. Consequently, these more recent approaches turned old ones almost obsolete in regards to provider independence and current trends on cloud adoption with architectures such as Cloud-Native, based on Microservices patterns of development, made concerns related with migration less relevant, due to their inherited cloud provider independence.

Nevertheless, considering the current state of technology for implementing an IT solution based on traditional service models, it can be concluded that a hybrid cloud model having on-premises configuration repositories, combined with CaaS/FaaS service models for IT solutions developed under a Cloud-Native approach using Multicloud deployment, provides a combination of methodologies and procedures for keeping investments on IT infrastructure in the cloud computing model under local control with a high degree of independence. This type of approach when associated with the management of such infrastructure using DevOps methodologies

provides a complete control and overview for deploying and managing IT infrastructure on cloud, with the possibility of customizing any cycle of the DevOps pipeline in order to make customizations for cloud transitions easier, using tools that can be adapted or created for such purposes in order to provide the needed functionality for these types of operations. The same type of approach for using infrastructure under a hybrid-cloud model combined with local configuration repositories continues to be a valid methodology when implementing cloud-based IT solutions that require having some (or even all) components under the traditional IaaS or PaaS service models, despite the identified constraints and required refactoring of those when migrating or moving them into another cloud eventually becomes necessary.

Additional Investigation

Due to the diversity of current cloud offerings, not necessarily related with the original service models of cloud computing, cloud providers are becoming a final product by themselves since most of their offerings are specific to the provider, similar to SaaS. Consequently, cloud providers are becoming differentiated at various service levels, sometimes being chosen by other services not directly related with the original cloud computing service models. The logic of "cloud migration" doesn't apply to most of those offerings, at least when taken from an infrastructure perspective or from the perspective of its related artefacts, if any. At most, migration from such service offerings has to be done from a development perspective.

Despite the growing service offerings by cloud providers, a standardization effort for the common implementations continues to make sense, at least to the traditional service models such as IaaS, which can form the basis of many other type of cloud service models or deployments. Having defined standards for seamlessly moving artefacts between cloud providers with baseline architectures such as IaaS would require a standardization effort for describing their respective compute, storage and network elements including any specifics concerning those, such as type of underlying virtualization technology, operating system flavor, type of virtual machine image and storage specifications concerning supported image types, data transfer protocols as well as network artefacts and their topologies along with any other details reflecting the interconnection among all those, in a parseable and interpretable manner according to standard definitions, for seamlessly reapplying such configurations on another provider. The advent of IaC languages can make this standardization easier but a considerable effort has to be done by standard-defining bodies in order to reach such objective. Since cloud infrastructure can be completely defined through code, cloud providers should make available complete metadata concerning the layout of infrastructure and any relevant characteristics in an exportable way to any defined cloud resources, at minimum for service levels such as IaaS. Without any standardization, such data has to be subject to transformation for reapplying on another provider just as it was demonstrated earlier, which may prove worthy for large deployments but probably not worth the effort for smaller implementations.

One possible approach for standardization would be to create a metamodel/metalanguage on top of any currently existing IaC language such as Terraform, that could provide a standardized taxonomical definition for non-proprietary cloud resources under a specific service model such as IaaS. Artefacts would become transposable to any service provider that would support and contribute to such standards. Conceptually, by choosing an existing IaC template having cloud

resources defined according to such standard, the parsing of such template using that metalanguage using options to specify the origin and destination provider could perform the required transformations in order to comply. Tools used to perform such transformation would reference some form of data supported and updated by cloud providers that assume to be in comply with the defined standard. Transposing related artefacts could be accomplished through the use of specific API's complying with such standards, also defining the transfer mechanisms and necessary operations between cloud providers for any recreation of artefacts or movement of related data.

A similar standardization effort similar to the one exemplified for IaaS could be developed for any other cloud artefacts under any service model whenever appropriate, but it is questionable if it justifies the effort, since some services models such as CaaS are already very cloud-agnostic due to their technical implementation. Alternatively (or complementarily) development of specific tools for hooking or coupling into specific stages of a DevOps pipeline with the objective of parsing IaC templates and converting them onto another provider under the same premises previously described could be subject to further investigation.

Besides a standardization effort for IaaS, standardization for other types of service models may be justifiable, however the evolution of cloud computing into several other areas of computing is turning many cloud service offerings proprietary in nature when considering public cloud providers and therefore moving away or migrating from such implementations requires rethinking or reimplementing those at a logical or development level, not in the same context as infrastructure or common service levels like IaaS or PaaS.

Any effort regarding the standardization or streamlining of cloud migration operations whether from on-premises to cloud or from cloud to cloud, independently of the service model or type of implementation, can also be empowered by any business model that can justify the investment, eventually contributing to the advancement of the technology or methodologies involved in such operations.

Methodology

The prime objective under this dissertation was to understand in which ways an higher level of independence from a cloud provider could be achieved, considering currently available options and methodologies related to cloud migration when complemented with other alternative methods and more recently available technologies, always within the scope and context of the research questions, resulting in a reference framework describing all findings.

In the attempt to understand and minimize the risks of becoming locked-in into a specific provider, some questions were raised. The following questions were considered of utmost importance for this research:

- Which frameworks are available to support the migration of an entire cloud infrastructure or solution into another cloud provider?
- Which tools or methodologies are available or recommended to address this need?
- In which steps should associated operations take place?

The need to acquire a deeper understanding of underlying technology in cloud computing and current migration practices through existing documentation on the subject was crucial to create the aforementioned framework. Comparing and contrasting existing or proposed solutions as well as identifying possible obstacles for their implementation from a practical perspective, whenever possible, was crucial to have a broader vision on the subject that could help address the research questions.

Research methodology is the systematic description of the procedures used in a theoretical analysis of the subject being studied, with the aim of presenting results of such study from a scientific perspective [58]. When the object under study allows for the formulation and subsequent testing of hypotheses, giving quantifiable statistical generalizations or other measurable results, it is defined as quantitative research [59]. If the object under study is not quantifiable but instead prone to different subjective interpretations due to the nature of the problem or the type of research question, it is defined as qualitative research [60].

Given the nature of the topic under investigation, a quantitative approach was not suitable since the research topic is not of a quantifiable nature and there is a considerable degree of subjectivity in the interpretation or applicability of some of the theoretical approaches. A quantitative approach was deemed more appropriate, therefore case study research was selected

for reviewing available articles, research papers and other related literature on the subject of cloud migration.

Due to the existence of several different implementations of cloud solutions and consequently different possibilities for migration, it was necessary to review the literature that addresses areas concerning migration, especially to a different cloud infrastructure or architecture, within the scope of the research question under this dissertation. The review of available articles, research papers and other related literature on the subject of cloud migration methodologies as well as the analysis of other existing solutions and procedures to address the issue, allowed for the comparison and contrasting of common patterns not only from a practical implementation perspective but also from a theoretical one.

After compiling and reviewing the relevant literature, a clear classification and categorization of different techniques and their possible shortcomings became possible, resulting in the identification of areas where alternative methods or tools could be suggested or eventually be augmented to existing practice and resulting in the creation of a reference table describing and classifying methods along with its key concepts and methodologies.

The identification of relevant steps involved in the migration processes which may include some form of transformation regarding the architecture were also identified for reference. Each of these steps was then independently analyzed and by contrasting such steps among the different sources in literature, made the identification of common approaches to cloud migration for specific types of architectures possible, and laid ground for other suggestions that combine or adapt their key aspects with more recent and compatible technical methodologies.

The suggested framework was based on transposing and combining steps for migration using an Infrastructure-as-Code based definition, taking advantage of cloud provider APIs, along with the use of DevOps methodologies when applicable, which allowed for the creation of a reference framework that not only represents an overall picture and understanding of current common practice including its related steps, but also an additional perspective on how such operations can be done or transposed in a way that diminishes the probability of being locked-in, should a migration or redeploy onto another cloud provider become necessary, even when there is a transformation of the cloud architecture in the process.

Bibliography

[1] I. Chana and T. Kaur, "Delivering IT as A Utility- A Systematic Review," *Int. J. Found. Comput. Sci. Technol.*, vol. 3, no. 3, pp. 11–30, 2013, doi: 10.5121/ijfcst.2013.3302.

[2] G. Petri *et al.*, "Predicts 2021 : Building on Cloud Computing as the New Normal," no. December 2020, pp. 1–15, 2020, [Online]. Available: https://www.gartner.com/document/3994453?ref=solrAll&refval=275213258.

[3] C. E. Leiserson *et al.*, "There's plenty of room at the top: What will drive computer performance after Moore's law?," *Science (80-.).*, vol. 368, no. 6495, 2020, doi: 10.1126/science.aam9744.

[4] W. Vogels, "Beyond Server," *Queue - Virtualization*, vol. 6, no. 1, February, pp. 20–26, 2008.

[5] J. Daniels, "Server virtualization architecture and implementation," *XRDS Crossroads, ACM Mag. Students*, vol. 16, no. 1, pp. 8–12, 2009, doi: 10.1145/1618588.1618592.

[6] M. Martonosi *et al.*, *Synthesis Lectures on Computer Architecture Editor Hardware and Software Support for Virtualization Datacenter Design and Management: A Computer Architect's Perspective A Primer on Compression in the Memory Hierarchy Analyzing Analytics Customizable Compu*. 2015.

[7] R. Morabito, J. Kjällman, and M. Komu, "Hypervisors vs. lightweight virtualization: A performance comparison," *Proc. - 2015 IEEE Int. Conf. Cloud Eng. IC2E 2015*, pp. 386–393, 2015, doi: 10.1109/IC2E.2015.74.

[8] M. Ben-Yehuda *et al.*, "The turtles project: Design and implementation of nested virtualization," *Proc. 9th USENIX Symp. Oper. Syst. Des. Implementation, OSDI 2010*, pp. 423–436, 2019.

[9] P. Case and K. Khajehei, "Role of virtualization in cloud computing," vol. 7782, no. Vm, pp. 15–23, 2014.

[10] Y. Rao Bhandayker, "A Study on the Research Challenges and Trends of Cloud Computing," no. April, 2016, doi: 10.5281/zenodo.2579238.

[11] P. Mell and T. Grance, "The NIST-National Institute of Standars and Technology-

Definition of Cloud Computing," *NIST Spec. Publ. 800-145*, p. 7, 2011.

[12] S. Goyal, "Public vs Private vs Hybrid vs Community - Cloud Computing: A Critical Review," *Int. J. Comput. Netw. Inf. Secur.*, vol. 6, no. 3, pp. 20–29, 2014, doi: 10.5815/ijcnis.2014.03.03.

[13] A. Rashid and A. Chaturvedi, "Cloud Computing Characteristics and Services A Brief Review," *Int. J. Comput. Sci. Eng.*, vol. 7, no. 2, pp. 421–426, 2019, doi: 10.26438/ijcse/v7i2.421426.

[14] U. M. Ismail, S. Islam, M. Ouedraogo, and E. Weippl, "A framework for security transparency in Cloud Computing," *Futur. Internet*, vol. 8, no. 1, 2016, doi: 10.3390/fi8010005.

[15] Y. Duan, G. Fu, N. Zhou, X. Sun, N. C. Narendra, and B. Hu, "Everything as a Service (XaaS) on the Cloud: Origins, Current and Future Trends," *Proc. - 2015 IEEE 8th Int. Conf. Cloud Comput. CLOUD 2015*, pp. 621–628, 2015, doi: 10.1109/CLOUD.2015.88.

[16] P. Debnath, V. S. Sharma, and V. Kaulgud, "A XaaS Savvy Automated Approach to Composite Applications," *Proc. - 2015 IEEE 8th Int. Conf. Cloud Comput. CLOUD 2015*, pp. 734–741, 2015, doi: 10.1109/CLOUD.2015.102.

[17] T. Siddiqui, S. A. Siddiqui, and N. A. Khan, "Comprehensive Analysis of Container Technology," *2019 4th Int. Conf. Inf. Syst. Comput. Networks, ISCON 2019*, pp. 218–223, 2019, doi: 10.1109/ISCON47742.2019.9036238.

[18] M. J. Scheepers, "Virtualization and Containerization of Application Infrastructure : A Comparison," *21st Twente Student Conf. IT*, pp. 1–7, 2014.

[19] M. K. Hussein, M. H. Mousa, and M. A. Alqarni, "A placement architecture for a container as a service (CaaS) in a cloud environment," *J. Cloud Comput.*, vol. 8, no. 1, pp. 1–15, 2019, doi: 10.1186/s13677-019-0131-1.

[20] M. Sewak, "Winning in the Era of Serverless Computing and Function as a Service - IEEE Conference Publication," *2018 3rd Int. Conf. Converg. Technol.*, pp. 1–5, 2018, [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8529465.

[21] R. A. P. Rajan, "Serverless Architecture - A Revolution in Cloud Computing," *2018 10th Int. Conf. Adv. Comput. ICoAC 2018*, pp. 88–93, 2018, doi:

10.1109/ICoAC44903.2018.8939081.

[22] A. Khan, “Key Characteristics of a Container Orchestration Platform to Enable a Modern Application,” *IEEE Cloud Comput.*, vol. 4, no. 5, pp. 42–48, 2017, doi: 10.1109/MCC.2017.4250933.

[23] C. H. Kao, S. T. Liu, and C. C. Lin, “Toward a cloud based framework for facilitating software development and testing tasks,” *Proc. - 2014 IEEE/ACM 7th Int. Conf. Util. Cloud Comput. UCC 2014*, pp. 491–492, 2014, doi: 10.1109/UCC.2014.66.

[24] K. Tang, J. M. Zhang, and C. H. Feng, “Application centric lifecycle framework in cloud,” *Proc. - 2011 8th IEEE Int. Conf. E-bus. Eng. ICEBE 2011*, pp. 329–334, 2011, doi: 10.1109/ICEBE.2011.32.

[25] D. Gannon, R. Barga, and N. Sundaresan, “Cloud-Native Applications,” *IEEE Cloud Comput.*, vol. 4, no. 5, pp. 16–21, 2017, doi: 10.1109/MCC.2017.4250939.

[26] R. V. O’Connor, P. Elger, and P. M. Clarke, “Continuous software engineering—A microservices architecture perspective,” *J. Softw. Evol. Process*, vol. 29, no. 11, pp. 1–12, 2017, doi: 10.1002/smr.1866.

[27] R. V. O’Connor, P. Elger, and P. M. Clarke, “Exploring the impact of situational context - A case study of a software development process for a microservices architecture,” *Proc. - Int. Conf. Softw. Syst. Process. ICSSP 2016*, pp. 6–10, 2016, doi: 10.1145/2904354.2904368.

[28] P. Kookarinrat and Y. Temtanapat, “Design and implementation of a decentralized message bus for microservices,” *2016 13th Int. Jt. Conf. Comput. Sci. Softw. Eng. JCSSE 2016*, 2016, doi: 10.1109/JCSSE.2016.7748869.

[29] G. Toffetti, S. Brunner, M. Blöchlinger, J. Spillner, and T. M. Bohnert, “Self-managing cloud-native applications: Design, implementation, and experience,” *Futur. Gener. Comput. Syst.*, vol. 72, pp. 165–179, 2017, doi: 10.1016/j.future.2016.09.002.

[30] J. Kirschnick, J. M. Alcaraz Calero, L. Wilcock, and N. Edwards, “Toward an architecture for the automated provisioning of cloud services,” *IEEE Commun. Mag.*, vol. 48, no. 12, pp. 124–131, 2010, doi: 10.1109/MCOM.2010.5673082.

[31] M. Guerriero, M. Garriga, D. A. Tamburri, and F. Palomba, “Adoption, Support, and

Challenges of Infrastructure-as-Code: Insights from Industry," *Proc. - 2019 IEEE Int. Conf. Softw. Maint. Evol. ICSME 2019*, pp. 580–589, 2019, doi: 10.1109/ICSME.2019.00092.

[32] K. Advisory, "A reflection on the perceived benefits of Infrastructure as Code A concrete case study to reflect on the value," pp. 41–47.

[33] P. Anderson, "Programming the virtual infrastructure," ;*login Mag. USENIX SAGE*, vol. 34, no. 1, pp. 20–25, 2009.

[34] V. Shvetcova, O. Borisenko, and M. Polischuk, "Domain-Specific Language for Infrastructure as Code," *Proc. - 2019 Ivannikov Meml. Work. IVMEM 2019*, pp. 39–45, 2019, doi: 10.1109/IVMEM.2019.00012.

[35] K. Morris, *Infrastructure as Code: Managing Servers in the Cloud*, 1st ed. Newton, MA, USA: O´Reilly Media, 2016.

[36] G. Kumar and P. K. Bhatia, "Comparative analysis of software engineering models from traditional to modern methodologies," *Int. Conf. Adv. Comput. Commun. Technol. ACCT*, pp. 189–196, 2014, doi: 10.1109/ACCT.2014.73.

[37] W. de Kort, "What Is DevOps?," *DevOps on the Microsoft Stack*, pp. 3–8, 2016, doi: 10.1007/978-1-4842-1446-6_1.

[38] M. Senapathi, J. Buchan, and H. Osman, "DevOps capabilities, practices, and challenges: Insights from a case study," *ACM Int. Conf. Proceeding Ser.*, vol. Part F1377, 2018, doi: 10.1145/3210459.3210465.

[39] S. Garg and S. Garg, "Automated Cloud Infrastructure, Continuous Integration and Continuous Delivery using Docker with Robust Container Security," *Proc. - 2nd Int. Conf. Multimed. Inf. Process. Retrieval, MIPR 2019*, pp. 467–470, 2019, doi: 10.1109/MIPR.2019.00094.

[40] H. Kang, M. Le, and S. Tao, "Container and microservice driven design for cloud infrastructure DevOps," *Proc. - 2016 IEEE Int. Conf. Cloud Eng. IC2E 2016 Co-located with 1st IEEE Int. Conf. Internet-of-Things Des. Implementation, IoTDI 2016*, pp. 202–211, 2016, doi: 10.1109/IC2E.2016.26.

[41] N. Paez, "Versioning Strategy for DevOps Implementations," *Congr. Argentino Ciencias*

la Inform. y Desarro. Investig. CACIDI 2018, 2018, doi: 10.1109/CACIDI.2018.8584362.

[42] C. Lassenius, T. Dings??yr, and M. Paasivaara, "DevOps: A Definition and Perceived Adoption Impediments," *Lect. Notes Bus. Inf. Process.*, vol. 212, pp. 166–177, 2015, doi: 10.1007/978-3-319-18612-2.

[43] M. Bhopale, "Cloud Migration Benefits and Its Challenges Issue," *Iosrjournals.Org*, pp. 40–45, 2008, [Online]. Available: http://www.iosrjournals.org/iosr-jce/papers/sicete-volume1/8.pdf.

[44] N. Ahmad, Q. N. Naveed, and N. Hoda, "Strategy and procedures for Migration to the Cloud Computing," *2018 IEEE 5th Int. Conf. Eng. Technol. Appl. Sci. ICETAS 2018*, pp. 1–5, 2019, doi: 10.1109/ICETAS.2018.8629101.

[45] N. Khan and A. Al-Yasiri, "Framework for Cloud Computing Adoption: A Roadmap for Smes to Cloud Migration," *Int. J. Cloud Comput. Serv. Archit.*, vol. 5, no. 5/6, pp. 01–15, 2015, doi: 10.5121/ijccsa.2015.5601.

[46] H. reza Bazi, A. Hassanzadeh, and A. Moeini, "A comprehensive framework for cloud computing migration using Meta-synthesis approach," *J. Syst. Softw.*, vol. 128, pp. 87–105, 2017, doi: 10.1016/j.jss.2017.02.049.

[47] M. Mishra, S. Kunde, and M. Nambiar, "Cracking the monolith: Challenges in data transitioning to cloud native architectures," *ACM Int. Conf. Proceeding Ser.*, pp. 0–3, 2018, doi: 10.1145/3241403.3241440.

[48] J. F. Zhao and J. T. Zhou, "Strategies and methods for cloud migration," *Int. J. Autom. Comput.*, vol. 11, no. 2, pp. 143–152, 2014, doi: 10.1007/s11633-014-0776-7.

[49] M. Ahmed and N. Singh, "A framework for strategic cloud migration," *ACM Int. Conf. Proceeding Ser.*, pp. 160–163, 2019, doi: 10.1145/3330482.3330528.

[50] G. C. Silva, L. M. Rose, and R. Calinescu, "A systematic review of cloud lock-in solutions," *Proc. Int. Conf. Cloud Comput. Technol. Sci. CloudCom*, vol. 2, pp. 363–368, 2013, doi: 10.1109/CloudCom.2013.130.

[51] J. Opara-Martins, R. Sahandi, and F. Tian, "Critical review of vendor lock-in and its impact on adoption of cloud computing," *Int. Conf. Inf. Soc. i-Society 2014*, pp. 92–97, 2015, doi: 10.1109/i-Society.2014.7009018.

[52] G. A. Lewis, “Role of standards in cloud-computing interoperability,” *Proc. Annu. Hawaii Int. Conf. Syst. Sci.*, pp. 1652–1661, 2013, doi: 10.1109/HICSS.2013.470.

[53] D. Petcu, “Portability and interoperability between clouds: Challenges and case study (invited paper),” *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 6994 LNCS, no. Section 3, pp. 62–74, 2011, doi: 10.1007/978-3-642-24755-2_6.

[54] J. Miranda, J. M. Murillo, J. Guillén, and C. Canal, “Identifying adaptation needs to avoid the vendor lock-in effect in the deployment of cloud SBAs,” *ACM Int. Conf. Proceeding Ser.*, pp. 12–19, 2012, doi: 10.1145/2377836.2377841.

[55] S. C. and X. L. P. Jamshidi, C. Pahl, “Cloud Migration Patterns: A Multi-cloud Service Architecture Perspective,” *Springer Serv. Comput. - ICSOC 2014 Work.*, vol. LNCS 8954, pp. 6–19, 2014, doi: 10.1007/978-3-319-22885-3.

[56] J. Guillén, J. Miranda, J. M. Murillo, and C. Canal, “Developing migratable multicloud applications based on MDE and adaptation techniques,” *ACM Int. Conf. Proceeding Ser.*, pp. 30–37, 2013, doi: 10.1145/2513534.2513541.

[57] A. N. Toosi, R. N. Calheiros, and R. Buyya, “Interconnected Cloud Computing Environments,” *ACM Comput. Surv.*, vol. 47, no. 1, pp. 1–47, 2014, doi: 10.1145/2593512.

[58] Kothari, C.R. (2004) Research Methodology: Methods and Techniques. 2nd Edition, New Age International Publishers, New Delhi.

[59] Jackson, S. L. (2008). Research methods and statistics: A critical thinking approach. Australia: Heinle Cengage Learning.

[60] Yin, R.K. (2008) Case Study Research: Design and Methods. 4th Edition, Sage Publications, Thousand Oaks.

Appendices

Terraform for IaaS

 # provider.tf

```
terraform {
  required_providers {
    azurerm = {
      source = "hashicorp/azurerm"
      version = "~>2.0"
    }
  }
}

provider "azurerm" {
  features {}
}

resource "azurerm_resource_group" "dissertacao-RG" {
  name     = "dissertacao-Resources"
  location = "West Europe"
}
```

 # networks.tf

```
resource "azurerm_virtual_network" "WebServerNet" {
  name                = "dissertacao-WebNetwork"
  resource_group_name = "dissertacao-RG"
  address_space       = ["10.0.0.0/24"]
}

resource "azurerm_virtual_network" "AppServerNet" {
  name                = "dissertacao-AppNetwork"
  resource_group_name = "dissertacao-RG"
  address_space       = ["172.16.0.0/24"]
}

resource "azurerm_virtual_network" "DBServerNet" {
  name                = "dissertacao-DBNetwork"
  resource_group_name = "dissertacao-RG"
  address_space       = ["192.168.0.0/24"]
}
```

 # virtualmachines.tf

```
resource "azurerm_virtual_machine" "VMweb" {
  name                  = "WebServer-vm"
```

```
  resource_group_name   = "dissertacao-RG"
  vm_size               = "Standard_xxx"
}

storage_os_disk {
  name              = "VMweb-OS-Disk"
  caching           = "ReadWrite"
  managed_disk_type   = "Standard_xxx"
  create_option       = "FromImage"
}

resource "azurerm_virtual_machine" "VMapp" {
  name                = "AppServer-vm"
  resource_group_name   = "dissertacao-RG"
  vm_size               = "Standard_xxx"
}

storage_os_disk {
  name              = "VMapp-OS-Disk"
  caching           = "ReadWrite"
  managed_disk_type   = "Standard_xxx"
  create_option       = "FromImage"
}

resource "azurerm_virtual_machine" "VMdb" {
  name                = "DBServer-vm"
  resource_group_name   = "dissertacao-RG"
  vm_size               = "Standard_xxx"
}

storage_os_disk {
  name              = "VMdb-OS-Disk"
  caching           = "ReadWrite"
  managed_disk_type   = "Standard_LRS"
  create_option       = "FromImage"
}
```

 # storage.tf

```
storage_data_disk {
  name              = "VMdb-Data-Disk"
  disk_size_gb        = "100"
  managed_disk_type   = "Standard_xxx"
  create_option       = "Empty"
  lun               = 0
}
```

terraform init

Initializing the backend...

```
Initializing provider plugins...
- Finding hashicorp/azurerm versions matching "~> 2.0"...
- Installing hashicorp/azurerm v2.75.0...
- Installed hashicorp/azurerm v2.75.0 (signed by HashiCorp)

Terraform has created a lock file .terraform.lock.hcl to record the provider
selections it made above. Include this file in your version control repository
so that Terraform can guarantee to make the same selections by default when
you run "terraform init" in the future.

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
```

Terraform for PaaS

 # provider.tf

```
provider "oraclepaas" {
  user            = "..."
  password        = "..."
  identity_domain  = "..."
}
```

 # database.tf

```
resource "oraclepaas_database_service_instance" "default" {
  name        = "database-service-instance"
  description = "This is a description for an service instance"

  edition          = "EE"
  shape            = "oc1m"
  subscription_type = "HOURLY"
  version          = "12.2.0.1"
  vm_public_key     = "An ssh public key"

  database_configuration {
     admin_password     = "somepass"
     sid                = "BOTH"
     backup_destination = "NONE"
     usable_storage     = 15
```

```
  }

  backups {
     cloud_storage_container = "Storage-${var.domain}/database-service-instance-backup"
     auto_generate = true
  }
}
```

 # appserver.tf

```
resource "oraclepaas_application_container" "example-app" {
  name              = "ExampleWebApp"
  runtime           = "java"
  archive_url       = "my-accs-apps/example-web-app.zip"
  subscription_type  = "HOURLY"

  deployment {
    memory = "1G"
    instances = 2
  }
}
```

terraform init

```
Initializing the backend...

Initializing provider plugins...
- Finding latest version of hashicorp/oraclepaas...
- Installing hashicorp/oraclepaas v1.5.3...
- Installed hashicorp/oraclepaas v1.5.3 (signed by HashiCorp)

Terraform has created a lock file .terraform.lock.hcl to record the provider
selections it made above. Include this file in your version control repository
so that Terraform can guarantee to make the same selections by default when
you run "terraform init" in the future.

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
```

CaaS Deployment

cat Dockerfile
FROM ubuntu
RUN apt-get update
CMD [“echo”,”Image created”]

docker build -t ubuntu:istec -f Dockerfile .
Sending build context to Docker daemon 2.048kB
Step 1/3 : FROM ubuntu
latest: Pulling from library/ubuntu
7b1a6ab2e44d: Pull complete
Digest: sha256:626ffe58f6e7566e00254b638eb7e0f3b11d4da9675088f4781a50ae288f3322
Status: Downloaded newer image for ubuntu:latest
 ---> ba6acccedd29
Step 2/3 : RUN apt-get update
 ---> Running in fbf7c6a4d16a
Get:1 http://security.ubuntu.com/ubuntu focal-security InRelease [114 kB]
Get:2 http://archive.ubuntu.com/ubuntu focal InRelease [265 kB]
Get:3 http://archive.ubuntu.com/ubuntu focal-updates InRelease [114 kB]
Get:4 http://archive.ubuntu.com/ubuntu focal-backports InRelease [108 kB]
Get:5 http://security.ubuntu.com/ubuntu focal-security/restricted amd64 Packages [726 kB]
Get:6 http://security.ubuntu.com/ubuntu focal-security/multiverse amd64 Packages [30.1 kB]
Get:7 http://security.ubuntu.com/ubuntu focal-security/main amd64 Packages [1329 kB]
Get:8 http://security.ubuntu.com/ubuntu focal-security/universe amd64 Packages [825 kB]
Get:9 http://archive.ubuntu.com/ubuntu focal/main amd64 Packages [1275 kB]
Get:10 http://archive.ubuntu.com/ubuntu focal/universe amd64 Packages [11.3 MB]
Get:11 http://archive.ubuntu.com/ubuntu focal/restricted amd64 Packages [33.4 kB]
Get:12 http://archive.ubuntu.com/ubuntu focal/multiverse amd64 Packages [177 kB]
Get:13 http://archive.ubuntu.com/ubuntu focal-updates/multiverse amd64 Packages [33.6 kB]
Get:14 http://archive.ubuntu.com/ubuntu focal-updates/universe amd64 Packages [1104 kB]
Get:15 http://archive.ubuntu.com/ubuntu focal-updates/main amd64 Packages [1749 kB]
Get:16 http://archive.ubuntu.com/ubuntu focal-updates/restricted amd64 Packages [788 kB]
Get:17 http://archive.ubuntu.com/ubuntu focal-backports/universe amd64 Packages [21.7 kB]
Get:18 http://archive.ubuntu.com/ubuntu focal-backports/main amd64 Packages [50.0 kB]
Fetched 20.1 MB in 4s (5614 kB/s)
Reading package lists...
Removing intermediate container fbf7c6a4d16a
 ---> a0d722f9baa1
Step 3/3 : CMD [“echo”,”Image created”]
 ---> Running in 3127bb224c6f
Removing intermediate container 3127bb224c6f
 ---> a5005d92b741
Successfully built a5005d92b741
Successfully tagged ubuntu:istec

docker images
REPOSITORY TAG IMAGE ID CREATED SIZE
ubuntu istec a5005d92b741 3 minutes ago 105MB

docker container run -d -p 5000:5000 --name registry -v ~/docker/registry:/var/lib/registry registry

783f7122ac8e83fc034a4cb6e31b729beab753705253c25ab8c3317f34ed1cef

docker tag ubuntu localhost:5000/ubuntu:istec

docker push localhost:5000/ubuntu:istec
The push refers to repository [localhost:5000/ubuntu]
9f54eef41275: Pushed
istec: digest:
sha256:7cc0576c7c0ec2384de5cbf245f41567e922aab1b075f3e8ad565f508032df17 size: 529

docker pull repository:5000/ubuntu:istec
istec: Pulling from ubuntu
7b1a6ab2e44d: Pull complete
Digest: sha256:7cc0576c7c0ec2384de5cbf245f41567e922aab1b075f3e8ad565f508032df17
Status: Downloaded newer image for repository:5000/ubuntu:istec
repository:5000/ubuntu:istec

FaaS Deployment

 # fn --verbose deploy --app pythonapp --local
Deploying pythonfn to app: pythonapp
Bumped to version 0.0.2
Building image fndemouser/pythonfn:0.0.2
FN_REGISTRY: fndemouser
Current Context: default
Sending build context to Docker daemon 6.144kB
Step 1/13 : FROM fnproject/python:3.8-dev as build-stage
3.8-dev: Pulling from fnproject/python
7d63c13d9b9b: Pull complete
7c9d54bd144b: Pull complete
6c659176d5c8: Pull complete
31bfadeaf52b: Pull complete
2bb8ff279f62: Pull complete
e9789ac33c4c: Pull complete
Digest: sha256:e346404c37fbca72d400beb2ce8e6a9e4d91f8c5201823cea538308207062917
Status: Downloaded newer image for fnproject/python:3.8-dev
 ---> edb6774a8ff2
Step 2/13 : WORKDIR /function
 ---> Running in 0db718b87641
Removing intermediate container 0db718b87641
 ---> d5ab5b929f06
Step 3/13 : ADD requirements.txt /function/
 ---> 5d00d1874166
Step 4/13 : RUN pip3 install --target /python/ --no-cache --no-cache-dir -r requirements.txt &&
rm -fr ~/.cache/pip /tmp* requirements.txt func.yaml Dockerfile .venv &&
chmod -R o+r /python
 ---> Running in 861817a204b2
Collecting fdk>=0.1.39

```
  Downloading fdk-0.1.39-py3-none-any.whl (78 kB)
Collecting httptools>=0.1.1
  Downloading httptools-0.3.0-cp38-cp38-
manylinux_2_5_x86_64.manylinux1_x86_64.manylinux_2_12_x86_64.manylinux2010_x86_6
4.whl (441 kB)
Collecting iso8601==0.1.12
  Downloading iso8601-0.1.12-py3-none-any.whl (12 kB)
Collecting pytest-asyncio==0.12.0
  Downloading pytest-asyncio-0.12.0.tar.gz (13 kB)
Collecting pbr==5.4.5
  Downloading pbr-5.4.5-py2.py3-none-any.whl (110 kB)
Collecting pytest==5.4.3
  Downloading pytest-5.4.3-py3-none-any.whl (248 kB)
Collecting py>=1.5.0
  Downloading py-1.10.0-py2.py3-none-any.whl (97 kB)
Collecting more-itertools>=4.0.0
  Downloading more_itertools-8.10.0-py3-none-any.whl (51 kB)
Collecting pluggy<1.0,>=0.12
  Downloading pluggy-0.13.1-py2.py3-none-any.whl (18 kB)
Collecting packaging
  Downloading packaging-21.2-py3-none-any.whl (40 kB)
Collecting attrs>=17.4.0
  Downloading attrs-21.2.0-py2.py3-none-any.whl (53 kB)
Collecting wcwidth
  Downloading wcwidth-0.2.5-py2.py3-none-any.whl (30 kB)
Collecting pyparsing<3,>=2.0.2
  Downloading pyparsing-2.4.7-py2.py3-none-any.whl (67 kB)
Building wheels for collected packages: pytest-asyncio
  Building wheel for pytest-asyncio (setup.py): started
  Building wheel for pytest-asyncio (setup.py): finished with status 'done'
  Created wheel for pytest-asyncio: filename=pytest_asyncio-0.12.0-py3-none-any.whl
size=11664
sha256=0dc26fa3bdc0f07e290c368e5ad90ae723b8a55e7ba78ef1406ebf82b1296dd1
  Stored in directory: /tmp/pip-ephem-wheel-cache-
tn0bslyi/wheels/23/f6/f3/2afd8a859f174197bec92a0ce1403d1cab9385474a4750ede5
Successfully built pytest-asyncio
Installing collected packages: pyparsing, wcwidth, py, pluggy, packaging, more-itertools, attrs,
pytest, pytest-asyncio, pbr, iso8601, httptools, fdk
Successfully installed attrs-21.2.0 fdk-0.1.39 httptools-0.3.0 iso8601-0.1.12 more-itertools-
8.10.0 packaging-21.2 pbr-5.4.5 pluggy-0.13.1 py-1.10.0 pyparsing-2.4.7 pytest-5.4.3 pytest-
asyncio-0.12.0 wcwidth-0.2.5
Removing intermediate container 861817a204b2
 ---> 23f6e79e4c25
Step 5/13 : ADD . /function/
 ---> efa95a6209b5
Step 6/13 : RUN rm -fr /function/.pip_cache
 ---> Running in ed7433895077
Removing intermediate container ed7433895077
 ---> e1f7c7dc4f8a
Step 7/13 : FROM fnproject/python:3.8
3.8: Pulling from fnproject/python
7d63c13d9b9b: Already exists
7c9d54bd144b: Already exists
```

```
6c659176d5c8: Already exists
31bfadeaf52b: Already exists
2bb8ff279f62: Already exists
7c8eebdd2fab: Pull complete
d86952facb46: Pull complete
Digest: sha256:78e1ca1b09597a68d5269b1f6b2386c47badfcfe93d2c0e97074a228ab3f16e5
Status: Downloaded newer image for fnproject/python:3.8
 ---> f1c1f2dc8447
Step 8/13 : WORKDIR /function
 ---> Running in 1f61880d68bc
Removing intermediate container 1f61880d68bc
 ---> 49fe973e9683
Step 9/13 : COPY --from=build-stage /python /python
 ---> 58d08693b182
Step 10/13 : COPY --from=build-stage /function /function
 ---> ebcecaa1fa5d
Step 11/13 : RUN chmod -R o+r /function
 ---> Running in 6072e763612f
Removing intermediate container 6072e763612f
 ---> dc3e4e8ed95f
Step 12/13 : ENV PYTHONPATH=/function:/python
 ---> Running in 012c742b0ee5
Removing intermediate container 012c742b0ee5
 ---> 7cc367c3a708
Step 13/13 : ENTRYPOINT ["/python/bin/fdk", "/function/func.py", "handler"]
 ---> Running in 75941863b7d1
Removing intermediate container 75941863b7d1
 ---> 135117754e51
Successfully built 135117754e51
Successfully tagged fndemouser/pythonfn:0.0.2

Updating function pythonfn using image fndemouser/pythonfn:0.0.2...
Successfully created function: pythonfn with fndemouser/pythonfn:0.0.2
```