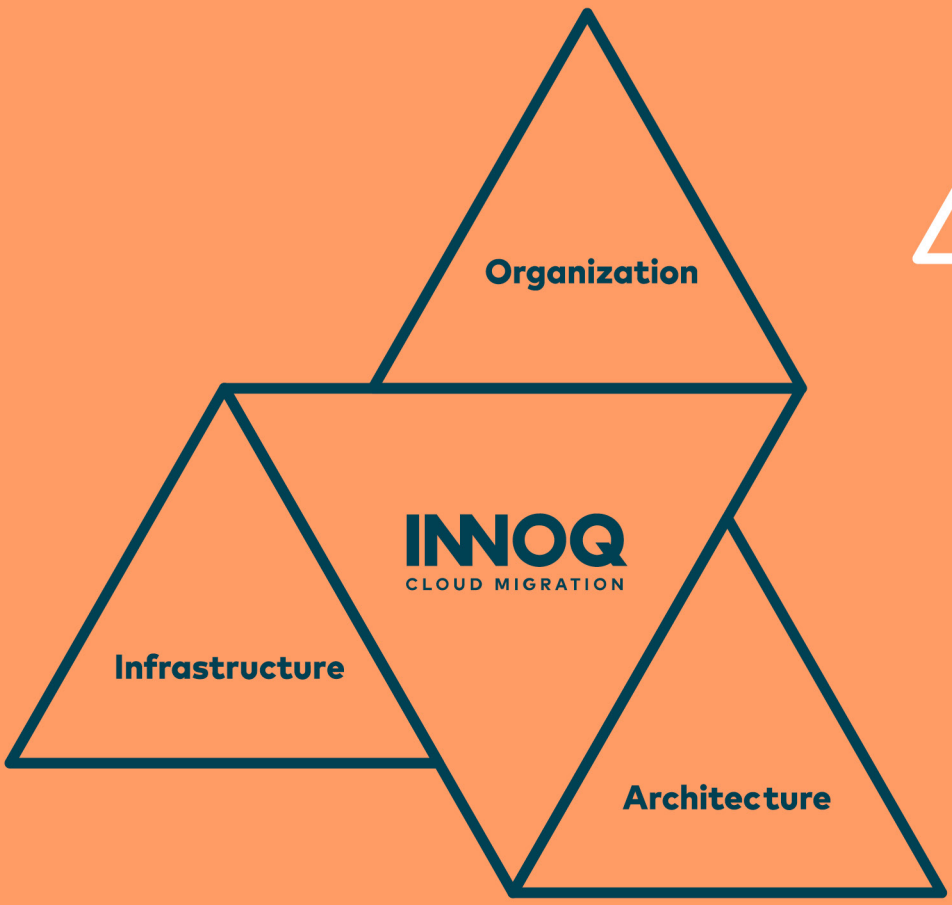
Christopher Schmidt · Sascha Selzer

Cloud Migration



INNOQ

Migrating to Cloud

From Legacy to Cloud Native Architectures

Christopher Schmidt
Sascha Selzer

ISBN —
innoQ Schweiz GmbH
Schutzengelstr. 57 · 6340 Baar · Switzerland
Phone (+41) 41 743 01 11 · WWW.INNOQ.COM

Layout: Tammo van Lessen with XeLaTeX
Design: Sonja Scheungrab
Print: Pinsker Druck und Medien GmbH, Mainburg, Germany

Migrating to Cloud – From Legacy to Cloud Native Architectures
Published by innoQ Schweiz GmbH
1st Edition · March 2022

Contents

1 Introduction

1.1 Executive Summary

Many companies are turning to the cloud expecting a lot of advantages and innovative solutions. But moving to the cloud is not a simple process. Various challenges have to be overcome. These are not only of a technical nature but also affect software architecture, the organization itself, and its culture.

This primer shows what needs to be done and what needs to be considered to make a planned cloud migration successful and for the benefits to be realized.

1.2 Cloud Migration

Cloud migration is the process of moving workloads, data, applications, and processes to a cloud environment. In most cases, resources are moved from an on-premises data center to a cloud vendor. Sometimes the shift is from one cloud provider to another.

The reasons for such a move are manifold. Some companies hope for reduced costs, some for more flexibility, and sometimes regulations force you to use more than one cloud provider. Whatever the reason, a simple lift and shift to the cloud is not always a promising approach, nor can it leverage the benefits and promise of the cloud. In addition, moving to the cloud may place new demands on the organization and require change. A change that is not always easy for long-established companies.

In this primer, we would like to show what the terms Cloud Native and Cloud Native Development actually mean and what consequences should be derived from them. We then point out the challenges of such a migration and give recommendations for a step-by-step approach. Of course, the security of the environments and data must not be ignored. We provide information on the security aspects

that need to be taken into account. Finally, we formulate migration patterns that describe best practices gained from many migration projects.

2 What Is Cloud Native?

When we migrate to the cloud, we can of course leave the application exactly as it was before. If it was a so-called monolith, the result is, in extreme cases, a single container in the cloud. Such a monolith does have some advantages. It is usually easier to develop (at least in the beginning) and does not require a network or messaging infrastructure for communication between components. However, when it comes to sustainable development speed with multiple teams or to scaling issues, Microservices are the architecture pattern of choice. That's because they enable just that: scalable applications (with more optimized use of resources) and scalable teams (size and number).

Microservices in the cloud have coined the term "cloud native." It describes the fact that cloud-native software is highly distributed (i.e., communicates with each other via a network), must exist in an environment that is constantly changing (number of worker nodes, for example), and is itself also constantly changing (continuous delivery). Cloud describes where we run our software and cloud native describes how we run it.

2.1 What Is Cloud Native Development?

Cloud-native development of Microservices is not very different from traditional software development. Except that we no longer have a local infrastructure but develop components for the cloud and with the cloud. However, more emphasis must be placed on some already known components and principles. These are:

- Containers and container orchestration
- Availability
- DevOps
- Microservices
- API management

2.1.1 Containers and Container Management

When we use containers, we use a generic shipment artifact that contains both the application and its runtime environment. This by itself places new demands on traditional software development departments. They have to deal not only with their applications themselves, but also with the components of a Linux distribution (as base images) or with programming a shell. Understanding what containers actually are, how they work, and what security implications this can have is essential for working software.

When multiple containers and multiple worker notes become necessary, there is usually also a need to use a container manager such as Kubernetes. It takes care of the automation of nonfunctional requirements such as scaling, rolling updates (initiated by continuous delivery), placement, and monitoring of containers. However, for such automation to be possible, containers must have some characteristics that are necessary. So the scaling takes place mainly over container instances and not over threads (threads can share memory, in contrast to processes). Also, the processes within a container must be resilient to change. This can be related to the inaccessibility of other components, to the change of the number of worker nodes, or the failure of networks, volumes, databases, etc.

2.1.2 Availability

The definition of availability is primarily a business decision. For example, for a travel site, the current weather report of a destination is certainly not as important as the booking funnel or the listing of offers. It follows that the weather report service may have a more relaxed availability requirement than the booking section. Of course, this does not only affect how we develop this part of software or how we operate it, but it also affects how we put new versions into operation. The deployment strategy Recreate is much simpler in its handling than for example Canary Release.

Once availability is defined as a whole, it has to be broken down to the individual services and their call chain. This is done by defining Service Level Indicators (i.e., the measurement points in the system) and specifying expectations about the value ranges in which these measurement values may move, the Service Level

Objectives. It is important that such values are collected and evaluated in the form of metrics. Ensuring the observability of components as well as the definition of alarms is an essential part of software development from the very beginning.

2.1.3 DevOps

In recent years, many DevOps departments have sprung up. However, DevOps is certainly not a department; it is not even a role, but rather a philosophy. Its purpose is to ensure that knowledge silos are reduced, that errors are accepted as something normal, that gradual change is implemented, that tools and automation is used, and that everything is measured. This philosophy is not something that applies only to individuals, but must apply equally to all members of the application teams.

2.1.4 Microservice Architecture

Microservices have some obvious characteristics such as each microservice running in its own process or being small. In a large microservice project with multiple teams, however, it is much more important that each service is independently deployable and uses lightweight, standard communication mechanisms. This can be achieved on the one hand by reasoned cuts along business capabilities, on the other hand by a reasonable decision for or against messaging middleware. If this is done well, the rate at which new features are developed can be kept constant over a long period of time. In addition, microservices ensure better compliance with architectural concepts, less technical debt, better deployability, and easier technical scaling.

But not everything about Microservices is good – they are also significantly more complex. However, if the advantages outweigh the disadvantages, the question arises as to how such a zoo of services can be managed. For this purpose, a whole new category of tools has been created: Service Meshes. A Service Mesh is a decentralized infrastructure layer that implements observability, routing, resilience, and secure communication capabilities between microservices. They help one not to lose track and make it easier to ensure a secure and reliable system.

They do this not only in operation, but also by providing test support through failure or latency injection.

2.1.5 API Management

API management involves the design, monitoring, and evolution (API life cycle management) of APIs that can be used to connect applications or other organizations. This enables teams that create or use third-party APIs to monitor access and ensure that all requirements of the applications that use those APIs are met. Through API management, a team also ensures that APIs are compliant with defined policies.

2.2 What Are Cloud Native Architectures?

In addition to the generally known principles of modern microservice architectures, cloud-native architectures place particular emphasis on the following characteristics:

- Consideration of deployment strategies
- Resiliency and resilience patterns
- Elasticity
- Infrastructure abstraction
- Runtime isolation
- Container patterns
- Application decoupling
- Observability

2.2.1 Deployment Strategies

- How are schema updates done?
- How are versions updated without downtime and how can this be automated?
- How is the rollout of security updates on pods or on nodes automated?
- How complex is the required automation?

Deployment strategies are part of the application and architecture concept and application life cycle **because we want deployment to be as easy as possible**.

Whether we can deploy applications with or without downtime can make a difference. If the availability of an application is defined in such a way that we can use the deployment strategy Recreate, then security updates or even schema updates of a database are not a problem anymore. For highly available applications, things are more complicated and we have to use rolling updates or canary releases. Whatever we do, such working deployments are not only necessary to deliver a new version of our application. They are also explicitly important to roll out security updates to the systems quickly and without heavy technical processes. In addition, a problem-free deployment is only possible with resilient applications, which should also be able to cope well with unplanned failures.

2.2.2 Resiliency

- Supports deployment strategies
- Supports elasticity
- Supports automation (self-healing, container autoscaling, etc.)
- Supports fast security fixes
- Supports isolation / bulk heading

Resiliency is all about preventing voluntary changes or faults turning into failures **because we want to be available**.

As we know, Microservices are highly distributed. There are networks between the services whose bandwidth could be greatly reduced. In the same way, upstream services can also be unavailable for a certain time, for example due to updates or other errors in the system. Resiliency is the ability of such a system to prevent errors from becoming failures. In this context, failures are defined by the absence of availability (related to the business definition of availability).

But resiliency is not only about unplanned errors in the system. Irregularities in service access can also occur when we scale services up or down (automated

container autoscaling), when we change the number of worker nodes (as a cluster response to container autoscaling), or when we need to apply security fixes.

As we develop a service and it needs to communicate in some way with other upstream components over the network, we must always consider what should happen if that communication fails, the component returns an error, or is too slow (misses its service level objectives). This consideration is integral to any service development and must be backed up by the implementation of resilience patterns and metrics.

2.2.3 Elasticity

- Cost management automation and process
- Scale to zero if possible
- Downscale to a minimum depending on the load

> Cost-efficient response to high or low load **because we want to save money**.

Single virtual machines from a cloud provider are often not cheaper than on-premise hosting. The cost saving only comes from the fact that you only pay for what you really need. We have to make sure that our production environment only uses as many resources as the current load situation requires. The automated elasticity required for this is an important part of the cost management of any infrastructure. Cost management means automating scaling and monitoring costs.

2.2.4 Infrastructure

- Infrastructure abstraction (less vendor lock)
- Homogeneous access to infrastructure (limits complexity)
- Automation

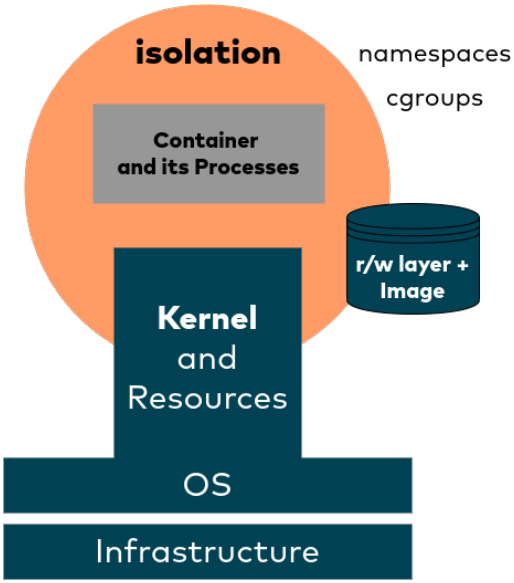
> Infrastructure is as abstracted and homogeneous as possible **because we don't want complexity to become complicated**.

The provisioning of infrastructure must be automated. This automation takes place not only during the initial setup of the infrastructure, but also to create elasticity during operation or during the dynamic generation of integration and test infrastructures. This automation can be complex. It is additionally made difficult by a large number of different APIs that we have to use for this purpose. A reasonable abstraction of the infrastructure and a homogeneous access to it is important to reduce the complexity as much as possible and to prevent an overly obstructive vendor lock.

2.2.5 Runtime Isolation

Technical isolation through containers **because we want to isolate life cycle, environment, and issues**.

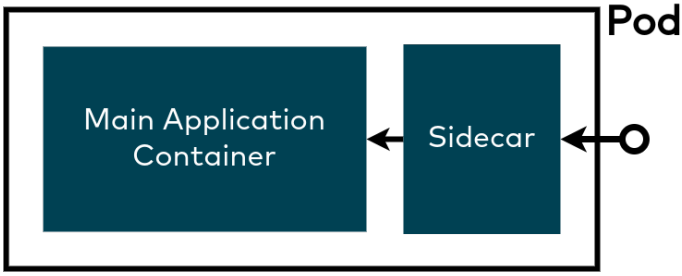
The processes of a container are only isolated from the rest of the system and not virtualized. All container processes share resources like the Linux kernel, networks, memory, and so on. This can entail certain security risks. For this reason, there are some container runtimes that increase security through further abstractions or virtualizations.



However, container images are also a standardized and generic shipment artifact. They are ideally suited to bring the benefits of microservice development to production.

2.2.6 Container Patterns

Patterns are also valid in the container world **because we want to separate concerns**.



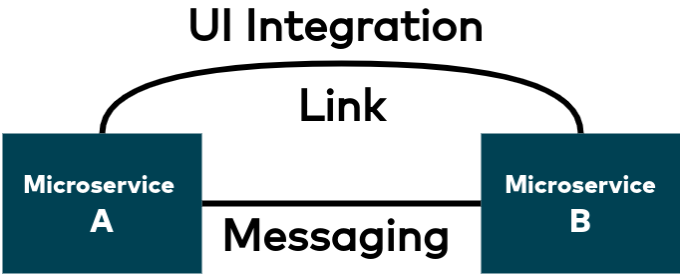
We already know patterns from object-oriented software development. Primary goals of patterns are to abstract away the low-level details, separation of concerns, or to simplify the reuse of software.

In the world of the containers these patterns exist likewise. So we can modularize some functionalities out of the main application container to a so-called sidecar or sidekick container. This functionality could for example be that it has a framework character (like a work queue pattern) or realizes certain nonfunctional requirements at runtime (e.g., TLS or rate limiting). Since the individual container images then have fewer use cases to satisfy, both development and testing are easier.

If we use Kubernetes we can use the pod abstraction for this. The pod description in the form of a YAML or JSON file is like a kind of functionality injection. With this, we can determine the sum of the functionality at deployment time.

2.2.7 Application Decoupling

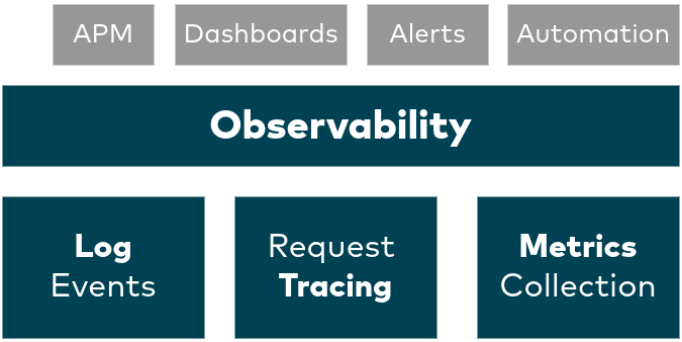
We are using Microservices, containers, and messaging **because we want to deploy services independently**.



In order to be able to deploy Microservices independently of each other, they must be decoupled. We can achieve this through the characteristics of Microservices and the isolation achieved by the container technology, or through messaging or other lightweight communication mechanisms.

2.2.8 Observability

We are observing our systems **because we want to know what's going on and automate everything**.



If we want to monitor availability or automate the infrastructure, then measuring and evaluating metrics is an important prerequisite. Metrics are a certain status

at a certain point in time. They can also be used to show trends before a problem occurs.

A log entry is an event that happened, presented in a very detailed and usually human-readable manner. Logs are mainly used to analyze an event after it has happened. Sooner or later logging will probably be replaced by tracing.

A trace describes a request as it propagates through the system. Traces combine the metrics of different services and allow a very detailed analysis. A brand-new class of tracing called profilers allows us to see not only into the communication between services but also into the service itself. This makes it possible to determine the need for optimization even in functions and methods of a service.

2.3 Cloud Provisioning

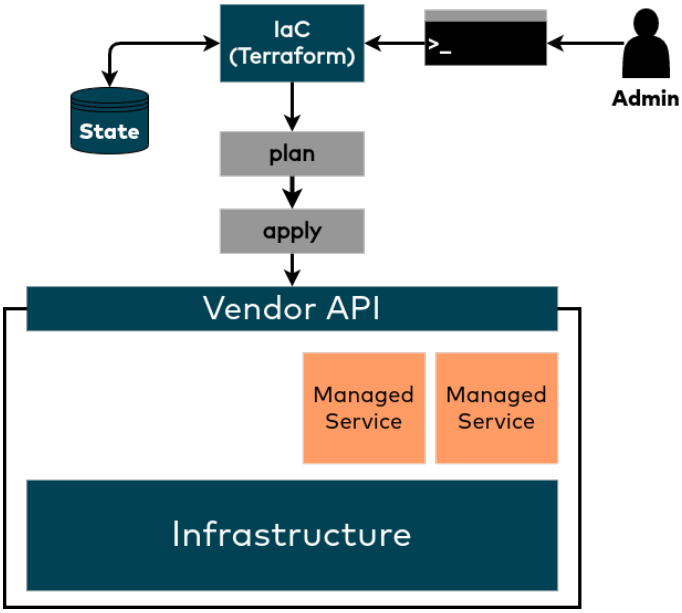
There are many ways to create infrastructure. The simplest one is to use the cloud vendor UI. This is OK for first exploration. However, when it comes to reliability and reproducibility of provisioning, infrastructure-as-code tools should be used. Because code can be versioned and shared, we can discuss about pull requests and so on.

Additionally, if we want to meet the requirements of the quite common hybrid cloud, we cannot escape the use of a cloud agnostic tool. Terraform, for example, has done a great job here. Nevertheless, a tool like Terraform cannot fix the semantic differences of cloud provider APIs or their differences in component abstraction. Thus, the reusability of the created infrastructure definitions will always be quite low.

That said, Terraform talks to the cloud API just like we (as developers) do. If we are cloud agnostic in terms of provisioning infrastructure, why should we accept cloud-vendor specifics (user credentials, managed services, scaling, updated, backup/restore, etc.)? However, since Kubernetes is found in many environments and has a well-defined interface, it is quite obvious to also manage the provisioning of the cloud infrastructure via the Kubernetes API. This procedure is usually referred to as Kubernetes first.

2.3.1 IoC Tools

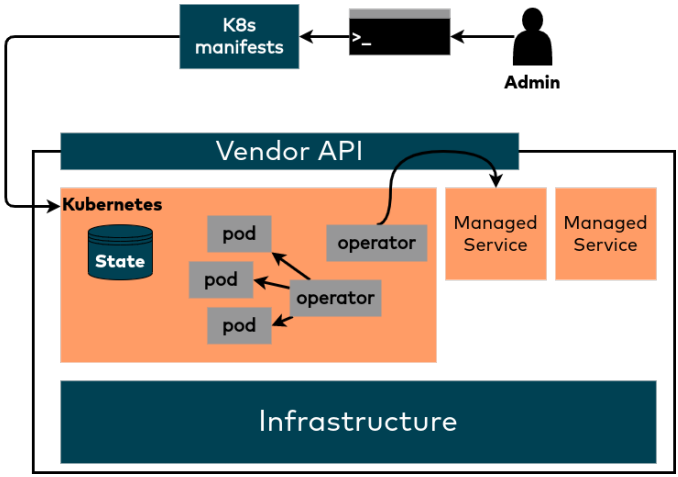
The following picture describes how classic infrastructure-as-code tools work. They are the current state of the art and are widely adapted.



Typical: State stored in S3 or git checked in, admin has credentials to execute Terraform apply and something like cloud admin on the cloud vendor API. Reprovisioning, adoptions, scaling, etc. has to be automated by executing scripts. If you are using Kubernetes for your production workloads there is a mismatch between what you are doing within Kubernetes (using the Kubernetes manifests/API) and as managed service (cloud vendor semantics). For example, if GitOps is to be used: Creating branches/PRs as Kubernetes namespaces and provisioning a dedicated database on vendor side always needs two tools (Helm and Terraform for instance) and a dedicated automation each.

2.3.2 Kubernetes First

If we follow the Kubernetes API first approach, everything has to be "re-mapped" to this API. To achieve this mapping we use operators[1]. Operators include all the know-how of operating a component (like PostgreSQL or own components) or of the use of a cloud provider API.



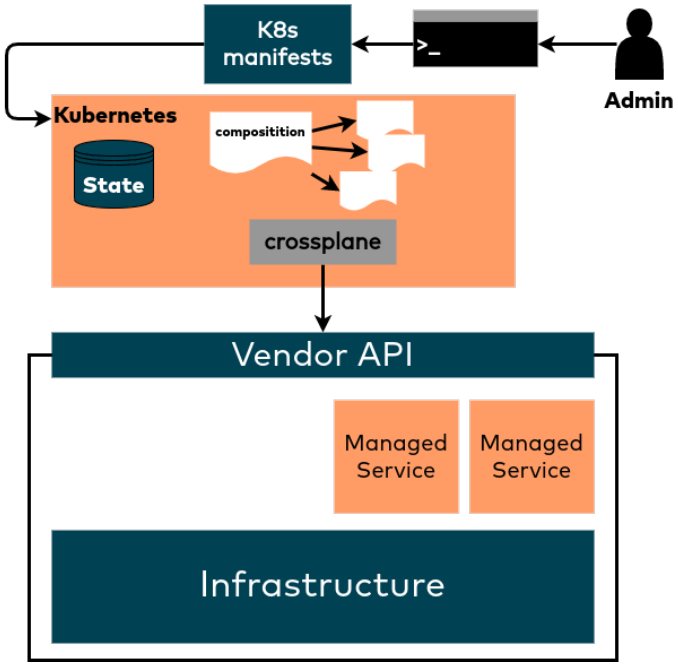
The Kubernetes API is the same, on the cloud vendor side, on-premises, and on the developer laptops. Therefore, this approach reduces the heterogeneity of APIs and also the necessary tools to manage them.

Of course, operators must be created or used that meet the quality standards we would have for managed services. This is somewhat of a challenge. However, it is to be expected that the mainstream tool providers have an interest in getting back in touch with a customer directly and not via a cloud provider. Thus, they will provide excellent operators [2], which will not be inferior to the available managed services.

[1] https://kubernetes.io/docs/concepts/extend-kubernetes/operator/
[2] List of tools based on the operator pattern https://operatorhub.io/

2.3.3 Self-Service Infrastructure Composition

However, even if you want to use managed services, the question arises as to how you want to provision them homogeneously via the Kubernetes API. There are some new approaches for this, such as Crossplane[3]. Crossplane allows the provisioning of managed services of a cloud provider via the Kubernetes API. In addition, groups of infrastructure objects can also be managed via so-called compositions. In this way, self-services adapted to the project can be created.



However, none of the cloud provider's managed services will be available either on-premises or on your laptop.

[3] https://crossplane.io/

3 The Migration Challenge

Migrating to the cloud is, in many cases, not a simple undertaking. We are often not starting from a greenfield but with a huge legacy ecosystem and with organizational structures that have grown around supporting and operating it. A successful migration has to tackle several areas at once:

- The organizational structure and culture itself has to embrace the new possibilities and freedom teams can have through the simplified accessibility of underlying infrastructure up to the production environment
- The new responsibilities this access brings
- The architecture to make use of the possible fine-grained scalability
- The platform brings a new set of paradigms and tools

Additionally, migration does not happen with a big bang approach but is a continuous process since the availability of the service to the customer remains the priority. Therefore, a step-by-step approach is preferable. It is nevertheless still complex to decide which steps shall be taken first and how much planning is advance is required before starting.

There are several challenges you might experience along the way, some of which are listed and explained here.

3.1 The Hybrid Cloud Challenge

Depending on your business environment, you have to manage personalized data or data with additional protection needs which cannot be covered in the cloud. Many cloud providers don't have an easy out-of-the-box solution to manage this kind of data and if they do, they still must follow laws from their own country which can be in conflict with local rules and laws, e.g., the US Cloud Act. In these situations, a hybrid solution is a common approach to separate the processing of insensitive data in the public cloud and processing sensitive data on-premises. Hybrid clouds also cover other use cases which make them preferable, for example avoiding provider lock-in or the locality of the providers. It may be

that your preferred provider does not cover all markets so you have to use another provider.

The challenge arises from the different skills and toolsets needed to manage infrastructure in the cloud and on-premises. There is a conceptional break which may demand implementing different operational models and cooperation principles. The migration approach has to define a way to reduce the tool and operational gap or an option to get rid of the on-premises part completely.

3.2 The Skill Challenge

In traditional on-premise systems, you normally experience a culture of separation of concerns at the border between development and the aspects involved in bringing the software into production. This includes a clear separation of skills and tools on both sides and a handover process in between normally managed on an organizational level. The handover process usually contains a formal procedure described in the form of documents (manuals, test reports) and a sign-off by the receiving party. No matter what approach is taken for the final migration in terms of team structure, there is a shift of knowledge involved. This is true if you have a "you build it, you run it" plan in mind or a more conservative approach still with a central operational team.

In both scenarios, the operation or mixed teams need further skills focusing on managing and automating infrastructure as code with tooling already known to developers. On the other hand, development teams need additional skills in the operational constraints of their application in the sense of hardening, vulnerability management, and defining the resource needs since they are no longer just delivering the software but now also have to manage the deployment flow probably right through to production. Training and introduction of these concepts to the team must be a fundamental part of the migration and needs to be planned alongside the technical changes in the system.

3.3 The Tooling Excess (or Don't Follow the Latest Hype) Challenge

When starting to become familiar with the possibilities the different cloud providers and cloud-native tools offer it can be quite overwhelming. For example, reading the best practices created by the cloud provider on how to work with their offerings can quickly lead to overengineering. Before starting a migration process and starting to build up a tool selection, the focus shall always be on the goal of the migration. Is it for example flexibility to switch vendors any time, or a quicker time-to-market? Depending on your goal the technology can be very different and migration can have different paths.

So always start from the problem by asking yourself the following questions:

- What is the goal of the migration?
- When I select a tool or provider, does it help me to reach the goal?
- When is it good enough?

This challenge is ongoing as goals change over time as do tools and providers. The current path, tools, and decisions have to be regularly validated against the goal (the solution should follow needs, not vice versa). It is a part of the continuous feedback loop because the migration can only be measured as successful when the goal and the expected improvements are clear.

3.4 The Vendor Lock-In/Managed Services Challenge

Managed services are a tempting solution for developers' common needs such as persistence solutions or middleware as these are the complicated parts in a system to scale and maintain. Additionally, why should we not use specialized solutions by teams focusing on optimized solutions (persistence solution from the persistence experts, middleware from middleware experts)?

Besides the clear benefits, these ideas come with constraints that we have to keep in mind:

- Every provider I choose for a specialized managed service comes with its API semantic
- Every provider has its user management
- Every provider comes with its cost structures and operational constraints, e.g., how scalability works and how it has to be configured.

This means that we probably have to manage the automation and management with different tools and approaches. As the user management is quite complex, it could lead to a very small, highly privileged group with access to control all of that. There is a high likelihood that these teams cause bottlenecks and silos as the creation and update of any managed service has to go through that team.

If part of the migration is the idea of self-managed teams with self-responsibility for the needed infrastructure or also the idea of GitOps, this setup can make it quite complex or even impossible to achieve.

Additionally, it could lead us to the false impression that optimization is the work of someone else. But in reality, it means we have to understand all constraints and configuration options as whilst we may not operate it anymore, we have to define what optimal means to us.

The challenge is to define the balance between managed services in order to benefit from them without migrating the problems we have on-premises additionally into the cloud. This would cause inflexible demand management for new infrastructure needs. As a bare minimum, the constraints of the migration must be clear and well understood when the decision to go all-in for managed and specialized services is made because these decisions are difficult to change later.

3.5 The Complexity Challenge

Traditional applications prefer simple deployment strategies. One deployment artifact, one pipeline, one server. Fewer moving parts were the goal because setting up infrastructure is difficult and complicated when it implies real hardware setup and configuration. Every additional moving part with its release cycle also implies another delivery process and alignment between teams in the form of combined

release plannings. So fewer parts are much easier to handle and also easier to debug and monitor. In these scenarios, topics such as availability, scaling, and fault tolerance are typically pushed down to the operational level of the teams running the software and infrastructure. As the described operational needs are not covered by the software itself, they are covered by hardware capabilities. This can be uninterruptible power supplies or RAID systems to mitigate failures but can only avoid them up to a point. These systems focus fully on increasing the mean time between failures as much as possible because releases and bug fixes are expensive to roll out.

Architecture constraints of these kinds of applications focus on well-structured software within a deployment monolith that do not benefit directly from modern infrastructure. Lift-and-shift scenarios are more common than expected but have the reputation of ultimately combining inflexibility of monoliths and complexity of modern cloud infrastructure with its new tools and access management.

The challenge we have to face is how to evolve such an application over time from the traditional environment to an environment where infrastructure setup is cheap and simple and deployments can be self-managed by teams. During the migration, the main focus stays on the availability of the system toward the customer, so we have a long period of time in which legacy and new systems coexist.

A guide could be the Strangler Fig Application Pattern[1] as a possible evolution pattern and the SCS architecture approach[2] as an implementation possibility.

3.6 The Cultural Challenge

The traditional way in which an organization builds, delivers, and operates a system is sharpening its culture by defining the team structures, clear processes in between, and the split in responsibilities. These cultures are heavily driven by checks and balances and gate-keeping. The inherited team structure often follows the different phases of a delivery pipeline like planning, development, verification,

[1] https://martinfowler.com/bliki/StranglerFigApplication.html
[2] https://scs-architecture.org/

validation, releasing, and operation with the software flowing through each phase and being improved by each team on the way.

When moving to the cloud, these phases and clear hand-over processes become blurry, as do the responsibilities of the teams. Some of the tasks may be automated or all responsibilities for the delivery are handled by a single team for one part of the software. This is a paradigm shift that has to be handled from a cultural perspective as well. Just because we are moving to the cloud does not mean we must replace the whole organization with new people knowing the paradigms and ideas. The current organization with its people must transform to fit the new concepts.

If this transformation is not managed actively and is not seen as part of the cloud migration strategy, we could end up with teams trying to set up traditional processes on the new system to give them purpose and relevance. This is a challenge for the managers forming the logical organization structure as well as for the teams finding their spot in the new system in the form of improvements and support to ensure a continuous flow of value.

3.7 The Security/Secrets Challenge

When moving to the cloud, a key area of concern is how to handle secrets and security in and on a platform operated by someone else. There are concepts and tools available by cloud providers but also in the open-source domain. These concepts focus on splitting secrets management and application management and differ from on-premise concepts like handling encryption manually and reducing access to keys and secrets to a small group.

As it is a crucial part of the migration to decide how much trust I put into a provider and how much I want to do on my own, the secrets management concept has to be elaborate. This can go hand in hand with the benefits I expect from the rest of the migration.

3.8 The Regulated Organizations Challenge

Of course, it is relatively easy for a simple web store to make the switch to the cloud seamlessly. For companies in the insurance and banking industry, however, there are a number of constraints to consider when migrating to the cloud. In recent years, the regulatory authorities have created quite pragmatic and applicable criteria for outsourcing systems and workloads. Thus, it can be assumed that a cloud migration is absolutely permitted and possible, even toward the large hyperscaler. But even "smaller" providers have created services that enable the often-required rapid switch from one service provider to another. Likewise, moving the company's data outside its own borders is possible, as long as things like the least privilege principle, end-to-end encryption, or data segregation can be guaranteed.

Nevertheless, it is of course important to take notice of the relevant rules and to derive and implement appropriate measures from them.

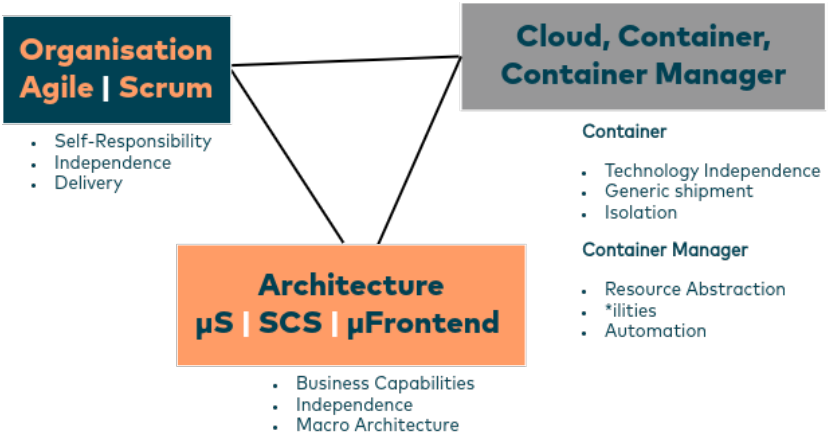
3.9 The Losing Sight Challenge

The combination of all these challenges at the beginning of the migration can lead to a lot of discussions and can stop us from going forward in any direction. Knowing all the challenges without having a solution from the start may paralyze us to probably take a wrong step or may lead us to too many goals.

To avoid long discussions and probably a big design up front, the direction and goal of our migration should stay in the focus of all actions we take. We do not need a full big picture of how the final result will look like but need a clear path to fulfill the goal. All we need in the beginning is a first step in the right direction to start the journey.

4 Migration to Cloud Native

We have seen that there are a lot of challenges in migrating to the cloud or cloud native. The challenges are not only technical. Introducing a tool such as Kubernetes or Docker does not make a product more flexible or better. It is important to understand that the migration to the cloud will be successful through three main components: the technical change of the production and development environment, a flexible and defined software architecture, and organizational changes. As with a Formula 1 car, success or failure is determined by the overall package and only those who consider all three elements together will be able to be more successful than before.



But what usually won't work either is a so called big-bang release. We will not be able to shut down our entire system on a Friday and put it online in a new, better version by Monday. Nor can we pull developer resources away from normal software development for years to develop a new version. So it is necessary to move step by step. A few aspects of this are shown below.

4.1 Re-Hosting

Porting an existing architecture as-is to run as a container.

Characteristics:

- At least a very simple first step
- Often called lift and shift
- Often does not take advantage of cloud capabilities (scalability, security, etc.)
- Can already use some managed services

Re-Hosting is a first simple step that we can do. We take our existing applications and put them, as is, into a container image and deploy them into a managed service from a cloud provider, such as Cloud Run or something similar. This approach is often called lift and shift. This is of course a viable first step because we can even use managed services from a cloud provider such as databases or messaging systems.

However, whether we can ensure points such as security or scalability is a completely different question. Not every legacy component can be scaled simply by increasing replicas. Instead, things like UI sessions or database locks have to be taken into account. Also, the security of our container does not increase just because we deploy it in the cloud. On the contrary, there are usually more attack vectors in the cloud than before.

Nevertheless, we can already learn and adapt some things here. For example, how we provision infrastructure, implement CI/CD, or how we use logs and monitoring.

Contributes to Cloud Native:

- Uses cloud vendor **infrastructure** (2.2.4 infrastructure)
- Application **isolation** (2.2.5 runtime isolation)
- Standard **observability** provided by the cloud vendor (2.2.8 observability)
- Course **elasticity** provided by the cloud vendor (2.2.3 elasticity)

4.2 Re-Platforming

Making slight adjustments to the existing application, like changing certain layers to start modernizing certain components.

Characteristics:

- Move NFRs to the platform
- Use the platform to perform routine tasks
- Automated scaling through container replicas
- Adding tools to improve platform functions (e.g., Service Meshes[1], CNI[2])

A platform is defined as services of a basic infrastructure that additionally provide capabilities for continuous integration, deployment, logging/monitoring, and so on. If we consider cloud providers or Kubernetes as such a platform, it makes sense to use their automation and realization capabilities of nonfunctional requirements. To do so we need to replace some layers of an application with the capabilities of the platform. This could be, for example, removing TLS termination, job functions, rolling updates, etc. from the application code and letting the platform take care of it. Similarly, we should redesign the applications to be scalable by simply increasing the replicas. In this case, the scaling itself is automated via the functions of the platform.

To enable this automation, we need to move toward cloud-native software. This will allow us to gain experience of more resilient software, more security through automated releases, and less dependency on human intervention.

Contributes to Cloud Native:

- Similar to Re-Hosting
- Better deployment using more specific **deployment strategies** (2.2.1 deployment strategies)
- Better **resiliency**, e.g., by using probes (2.2.2 resiliency)
- More specific/detailed **observability** (2.2.8 observability)
- Specifically automated **elasticity** (2.2.3 elasticity)
- Increased **security** due to regular and automated updates, through transparently encrypted communication (e.g., by using Service Meshes, CNI), etc.

[1] https://servicemesh.es/

[2] https://kubernetes.io/docs/concepts/extend-kubernetes/compute-storage-net/network-plugins/

4.3 Re-Tooling

Using tools that fit into the cloud-native paradigm.

It is not only our own software that must meet cloud-native criteria. It is equally important to use tools that support modern workflows in the development and operation of software. For example, if we want to create staging, test, or feature environments on demand, then the provisioning of databases and CD must also be more flexible. Ideally by GitOps.

Expected achievements:

- Using more cloud-native, resilient, and flexible tools for persistence, messaging that support things like automated scaling, self-healing, discovery, etc.
- Using more flexible tools for continuous delivery (CI/CD, progressive delivery [3], etc.)
- More dynamic environments, e.g., based on Git branches (GitOps [4])

4.4 Refactoring

Migrating applications to fit the new model and architecture. The new model should lead us to:

- More resiliency
- More performance and elasticity through scaling
- More deployment independence of modules
- More development speed through team scalability

If we look at a system from an architectural point of view, we have to distinguish between three different levels:

- **Micro architecture:** Local decisions about the internals of an application
- **Macro architecture:** Specifications on topics that are independent of the internal application structure but still need to be defined

[3] implements an automated gradual process of a rollout that is based on metrics

[4] developer-centric experience when operating infrastructure (https://www.gitops.tech/)

- **Domain architecture:** The functional modularization of the application landscape and the flow of information between applications

For a cloud migration, the micro architecture is initially less important. It is a team decision anyway and can only be such when we have technically independent Microservices. The macro architecture allows us to define things that are not defined in the micro architecture and that determine the interaction of the services with each other or the operating environment (including logging, monitoring, authorization, interfaces, protocols, etc.).

Nevertheless, the macro architecture requires the quantity and the functionalities of the services. The so-called domain architecture is used for this purpose. It describes the domain-oriented slices of an application and the data that must be exchanged between different domains. The better we define these slices, the less communication (both as quantity of data and as quantity of data types) is necessary, the more domain-specific the functions can be implemented, and the more independent the teams and service are. However, finding such slices is as difficult as it is important. There will also have to be some adjustments and iterations here over the course of a project.

How do we get to such slices? Domain-driven Design (DDD) with its associated tools, such as event storming[5], has proven to be a good method for working out the so-called Bounded Contexts[6] with their ubiquitous language. Whether we have been developing a legacy system for years or are planning a brand-new project, the first step to a good/better domain architecture is always DDD.

Depending on where you are coming from, there are two ways to proceed as an example:

Greenfield:

Domain-driven Design with tools like event storming -> monolith first or Microservices

Microservices are there to enable the independence of teams. However, there is hardly any reason to start a new two-person project with 20 Microservices. That

[5] https://en.wikipedia.org/wiki/Event_storming
[6] https://martinfowler.com/bliki/BoundedContext.html

is why it is perfectly possible to start with a monolith. Because also a monolith may have quite an architecture – more exactly, a correct domain architecture (modulith [7]). Later, should it be necessary to scale the speed of the software development, domains can then be quite easily cut out as independent modules and each handed over to a team.

Legacy System:

Legacy monolith -> DDD -> continuous refactoring toward Bounded Contexts -> extract domain as Microservice

You can't always start from scratch. Very often there is a forest of legacy systems. The particular challenge here is to modernize the existing software without affecting running systems and their development or the development speed. One possibility is to work out a target picture of the domains through a DDD. In doing so, experiences of the already existing systems can be taken into account. Subsequently, a refactoring takes place over a certain period of time (a couple of sprints) with the help of the strangler patterns[8] to make at least a first domain visible in the monolith. Then this part can be transferred into its own independent module.

In all of these projects, however, it is very important to define and weight the goals of modernization. Not everyone is Twitter and needs to scale their service in large numbers. Not everyone is Google and has hundreds of employees working on the same project. At the same time, however, nontechnical factors such as employee motivation should not be forgotten.

4.4.1 System and Self-Contained System View

Some time ago, INNOQ, primarily for classic projects on the web, created the concept of self-contained systems. You can read more about it on the SCS website[9].

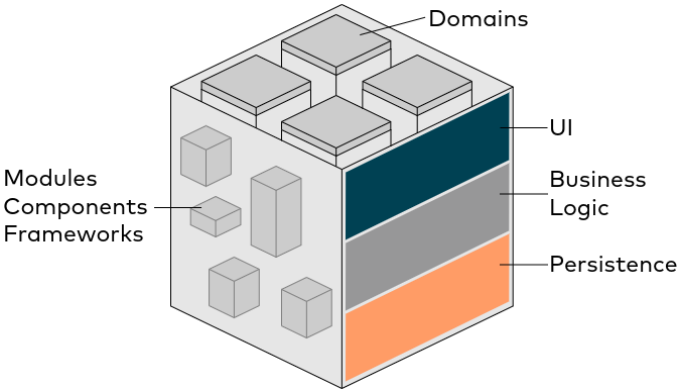
This concept serves as an example of how to modernize software to achieve the above goals. So a monolith ideally has the following characteristics:

[7] a modulith is a well-structured deployment monolith

[8] https://docs.microsoft.com/en-us/azure/architecture/patterns/strangler-fig

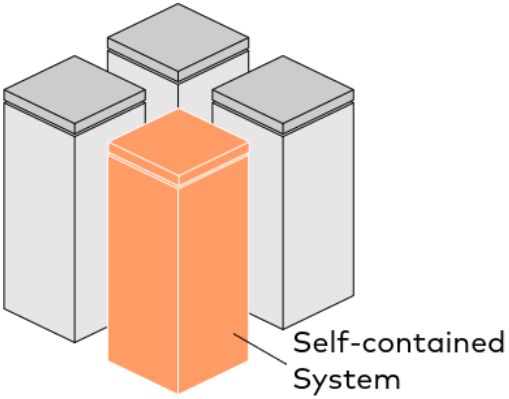
[9] https://scs-architecture.org

- Various domains (ideally, these have already been worked out)
- Layers like UI, business logic, persistence
- Uses a couple of frameworks
- Tends to grow



Once we have such a state or have worked it out (through refactoring), we can cut the monolith:

- Along domains
- Wrap domains in separate web applications
- Result: self-contained systems

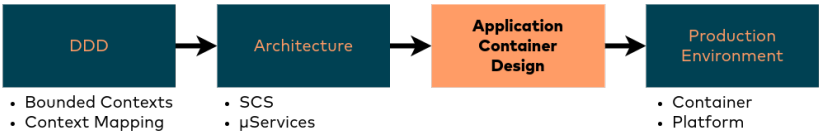


So the principle characteristics of the SCS architecture pattern are:

- An SCS realizes a business domain
- Each SCS is an autonomous web application
- All data, the logic to process that data, and all code to render the web interface is contained within the SCS
- An SCS has its own UI that is not shared
- Each SCS is owned by one team
- Organizational and technical decoupling is more important than sharing code/components/libraries

However, having such an autonomous web application does not mean that we can get by with a single container per SCS. This is possible, but sometimes it makes sense to modularize the functionality of such an application into several containers.

Such an approach is called application container design (for more information see this slide deck[10]).



We typically modularize in containers by considering the following:

- Architecture / business logic / domains
- Some required functionality has a framework characteristic (work queue pattern, etc.)
- Things that need to implement a nonfunctional requirement at runtime (resiliency pattern, encryption, protocol change, rate limiting, etc.)

Typically, we will get one group of containers per SCS. These can also subdivide into Microservices again, since an SCS can consist of several Microservices. Subsequently, we will try to put further nonfunctional requirements into separate containers. Not only to reuse them, but also to be able to reach standard products (such as the NGINX) and remove such things from the applications. The simpler a module becomes, the clearer it is and the better its testability.

[10] https://speakerdeck.com/fakod/application-container-design

4.5 Reorganization

We want to change the company's culture to agile and focus on continuous delivery and continuous feedback. But how do we get there?

Goals:

- More ownership and more freedom for teams to make decisions
- Small changes to the source code and less complex processes to bring these changes into production
- The team structure should not reflect the phases of software development, but the business domains
- Less documentation, fewer acceptance tests and checkpoints, more working software

Modernizing in-house applications based on cloud-native principles is as much about organization, processes, and governance as it is about the technology itself. The matching organizational principles should be more like those of agile software development. However, all these changes are not only structural. Major changes in cultural aspects are also necessary. Simply giving up control or removing control instances to allow more autonomy of the application teams can be a significant undertaking for companies with legacy systems and legacy employees.

But if we manage and establish this change, we gain more motivated employees, have new functions in production faster, and have more secure systems.

As we know that Conway's Law drives our architecture in a more likely direction mimicking the communication structure of the organization, this can be used as a tool to restructure the organization. The so-called reverse Conway maneuver helps to move an organization fitting the wanted architecture. So if our goal is to have autonomous teams, covering the whole process from development to deployment and being responsible for one part of the business domain, the focus has to be on cross-functional teams having the needed knowledge.

Nevertheless, having completely autonomous teams ends at the platform to be harmonized for all teams. The knowledge of how to operate a platform driving all other teams' velocity is normally bundled in a highly skilled platform team.

The main idea is to understand the platform as a product, helping all teams by simplifying the deployment and operation but also by improving the development experience. The platform team shall not be set up as a bottleneck for infrastructure demands but as an enabler for teams to set up their own infrastructure in a consistent and safe way. Infrastructure like databases or middleware shall not be "requested" from the platform team, but shall be set up by the requesting team with tooling provided by the platform, ensuring best practices and fail-safety.

As a last point, people with knowledge about operational patterns fitting to the cloud environment are rare. They are shared resources needed by all teams. They should not be seen as isolated consultants, but have to be involved in the daily development work. A good way is to follow practices described by the SRE book adapted to your conditions and organization. Not every company is Google but using it as a kind of blue book and understanding the basic ideas behind the SREs can help to make the migration smoother.

4.6 Surviving the Headwind

Stakeholders will doubt or simply deny your planned change. Many aspects of complex cloud initiatives allow different interpretations, such as the importance of moving to cloud at all or having more responsibilities as a software developer. Expect that some of your plans will contradict the opinions of others involved, and that these people may reject your project as a whole.

A few words of advice on this:

- This change also changes job descriptions of employees, company structure, organization, and culture. This is only possible with substantial top management support. Starting such a project as a submarine will not lead to success.
- Your initiative has to be based on facts, goals, and expectations which you must research and substantiate as carefully as possible
- Factual arguments alone will not be sufficient. People will be afraid of any change and new demands placed on them. Convince them that individual changes are a necessary part of a necessary larger whole. So a certain psychology and empathy is necessary here.

- Such an undertaking takes months rather than weeks, sometimes years. Don't lose patience and don't lose track of the initial goals you have been following. If the goals have been right, it is worth it in any case.

5 A Closer Look at Security

Security is not critical until it is. And then it is often too late to fix it.

Security in a cloud environment is an important task. There are different kinds of risks and vulnerabilities in the different layers of our environment, but also possible countermeasures to tackle and prevent them:

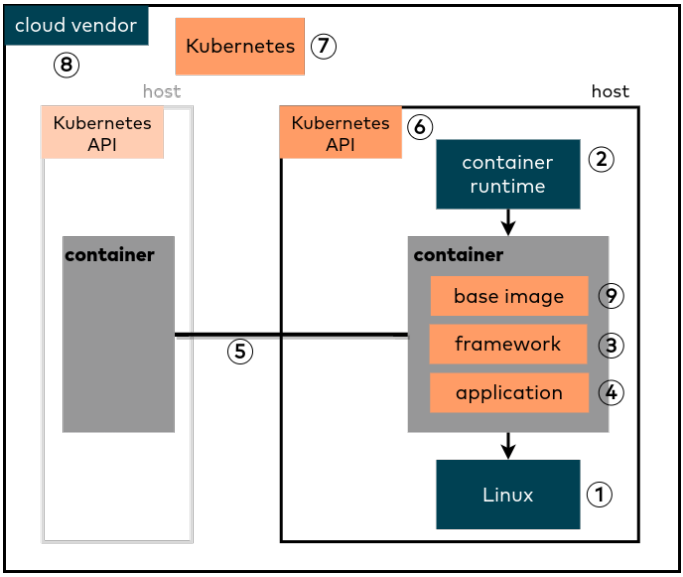


Figure 5.1: Areas of risk in a cloud environment

Provision of the Infrastructure (8)

- Internet gateways
- Private networks
- Firewalls
- Secrets management
- etc.

Host Operating System

- Operating system / Linux kernel (1)
- Container runtime (typically part of the operating system, 2)

Kubernetes

- Kubernetes processes (7)

Containers and Application

- Container base image (9)
- Application frameworks (3)
- Application code and configuration (4)
- Interservice communication (5)
- Authentication and authorization

Kubernetes API (6)

- Role-based access control (RBAC[1])
- Network policies (NP[2])
- Container security policies (e.g., OPA[3])

Ensuring security is not a one-time shot, but a continuous effort. Frameworks change, base images have to be updated, or firewall rules of the infrastructure have to be adapted, for example to new needs of the applications. Adapting to changes has to happen dynamically and, above all, automatically.

In many cases, you have to deal with a certain complexity of the system. This also makes it likely that you will lose track at some point. The more complex a system is, the higher the probability of errors. So you should always make sure you create structures that are as simple and homogeneous as possible. Because simplicity is security.

[1] https://kubernetes.io/docs/reference/access-authn-authz/rbac/
[2] https://kubernetes.io/docs/concepts/services-networking/network-policies/
[3] https://www.openpolicyagent.org/docs/latest/

5.1 Shared Responsibility Model

Security and compliance are the shared responsibility of the cloud provider and the user. This shared model helps reduce the user's operational risks as the cloud provider operates, manages, and controls components from the host operating system, virtualization layer, and managed services to the physical security of the facilities where the service is operated. It is critical to understand the shared responsibility model as well as the security tasks handled by the cloud provider and those handled by the organization itself.

5.2 Provision of Infrastructure

How infrastructure is provisioned depends on the used cloud provider. Depending on the cloud provider, certain best practices should be applied. In any case, it is important to configure the infrastructure so that it is as secure as possible without affecting the actual development of the application too much or making it inflexible. Pragmatism is key. Concerning user and permission management of the cloud provider API, these best practices can cover for example:

- Least privilege
- Separation of duties
- Named accounts
- Audit and event logging

But of course the infrastructure components used must be just as secure. So you'll want to think carefully about which networks need to be private and which don't. Which network segments are there and how are they allowed to communicate with each other? Some important questions also relate to data encryption, network encryption, and service identities.

Many companies use infrastructure-as-code tools like Terraform. Among other things, this enables infrastructure components to be created in a reproducible manner. Such tools are also relatively easy to automate and can usually be managed with Git. However, a Terraform script only lives while you run it. After it finishes, it does not actively monitor the system state and prevent and correct possible configuration drifts. To take this into account, new concepts are emerging

such as Crossplane or operators that can constantly monitor external infrastructure through components, for example in a so-called management cluster. This also has the advantage of not having to deal with two semantically different and complex APIs. Namely that of Kubernetes and that of a cloud provider. Uniform and automatable access (via an API) to a system reduces the probability of errors and increases security.

5.3 Host Operating System

For the host operating system, we usually use a Linux distribution. Part of these Linux distributions is usually also the container runtime, like containerd, CRI-O, Kata, or similar. Each distribution regularly provides new releases if critical vulnerabilities are found. For us, this means that we have to constantly check if there are updates available. Depending on the criticality of the problem, we also have to update our system as quickly as possible. If we wait for the next maintenance window, it means nothing else than that the risk of an intrusion exists for longer.

Typically, such updates are made by rebooting the host system. However, this also means that the overall application must be able to cope with the short-term unavailability of upstream services at any time. Resilience is therefore an inherent part of system design. The more resilient an application is, the easier it is to update it. And therefore the easier it is to guarantee more security through faster updates.

5.4 Kubernetes

Kubernetes consists of five to six different processes, depending on how you look at them. As with the host operating system, some updates need to be applied there. Issues with Kubernetes often affect “only” the Kubernetes API, which is typically not externally accessible, but not the application world. Also, it is typically possible to update Kubernetes without restarting the application containers. Nevertheless, changes to the API by newer Kubernetes versions might require minor rebuilds.

5.5 Containers and Application

Container images are the applications plus their runtime environment. In this respect, a container image contains a specific distribution, various frameworks, as well as the application code itself, and possibly also its configuration. Base images and frameworks used are typically external components whose need for updating must be monitored, preferably in an automated manner. However, the last Log4j vulnerability (Log4Shell) also showed that such measures alone are not always sufficient. In this respect, application protection is also necessary for other areas, such as controlling IP traffic through firewalls or network zones.

But even the application code can open up attack vectors. This is why it is necessary to run attack scenarios as part of a CI pipeline and to evaluate them automatically. For particularly critical applications, it also makes sense to perform explicit penetration tests occasionally.

Since our applications reside in many cases on a public cloud provider's infrastructure, we need to decide on whether or not we can/want to trust the provider's networks. If we don't, interservice communication must be encrypted. However, even this is not enough; in the end, we must use client certificates in addition to server certificates to build a mutual trust relationship between services. As certificates should have a reasonably short lifetime (24 hours), the constant generation and rollout of new certificates is a technical challenge and must be automated.

5.6 Kubernetes API

The Kubernetes API also offers a whole set of options for protecting both access to the API and the applications. Role-based access control (RBAC) allows the defining of access rights for operators and users in a very fine-grained way. On the network side, network policies allow isolating critical components, projects, teams, and environments from each other. Depending on the network components used, this can be done on OSI Layer 4 or OSI Layer 7. For example, it is quite common with Kubernetes namespaces to prevent all outbound and inbound traffic. For specific components like the ingress controller that require such access, it

is allowed again. Such explicit whitelisting of accesses would also have made the Log4Shell vulnerability, already at the Kubernetes level, less problematic.

A factor that is often forgotten is also the question of what containers are allowed to do on the corresponding host system and what they are not. The processes of a container are, depending on the container runtime used, simply normal Linux processes. So the question always arises: What capabilities are they allowed to assume? Are they allowed to mount the file system of the host or not? Are they allowed to run as a privileged container? The least-privilege principle also applies here. The fewer privileges a container process has the fewer the opportunities for an attack, and generally the more secure the overall system.

In the past, such pod or container security standards were ensured via the so-called pod security policies. However, these policies will have to be replaced in the future by a Pod Security admission controller or similar (OPA). Whatever one uses in the future, both defining container security standards and putting governance for them in place is a necessary feature of a container platform.

5.7 Data Categories

Handling data properly is an essential part of any security consideration. This applies not only to FINMA-regulated banks and insurance companies [4], for example. Therefore, it is important to categorize the data that resides in a system. Depending on the classification into these categories, measures must be taken to ensure confidentiality.

- **Synthetic data:** Synthetic data is data that is artificially generated and not based on real events. It is not related to customer data or patterns that can be traced back to individual customers. It is often created using algorithms and used for a variety of activities, such as test data for model or product validation and AI model training.

[4] As an example of regulatory requirements for outsourcing solutions by the Swiss Financial Market Supervisory Authority (FINMA): https://www.finma.ch/en/news/2017/12/20171205-mm-rs-outsourcing/

- **Anonymized data:** Anonymized data is data where personally identifiable information has been removed or changed. Customers can no longer be individually identified, but certain patterns can still be traced back to them.
- **Client identifying data:** Customer identification data is data that enables the identification of an individual customer.

Protecting data can be costly. In this respect, different measures can and must be taken depending on the categorization of the data. A number of questions need to be answered, such as: Where can my data be stored? Even outside my country? What scope of measures do I take for which data? Where is my data located in a hybrid cloud scenario and how do I access it?

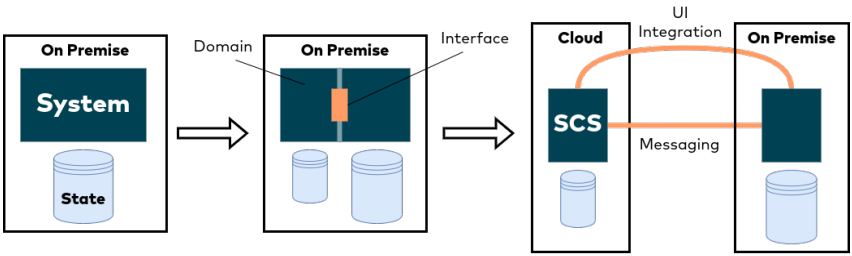
5.8 Simplicity Is Security

Having the security of a container platform in mind is not just a necessary evil, it is an absolute requirement for operation. However, its multilayered nature and complexity sometimes make one despair. Particularly when industry-specific regulations must also be taken into account. Nevertheless, there are very helpful tools for all levels that help to automate the task. Because only automation makes it possible to keep the complexity under control.

But the system design can also help to increase security. The fewer APIs there are, the more homogeneous the infrastructure, the fewer frameworks we use, and the simpler our applications are, the more security can ultimately be guaranteed.

6 Patterns

6.1 Data Migration Pattern

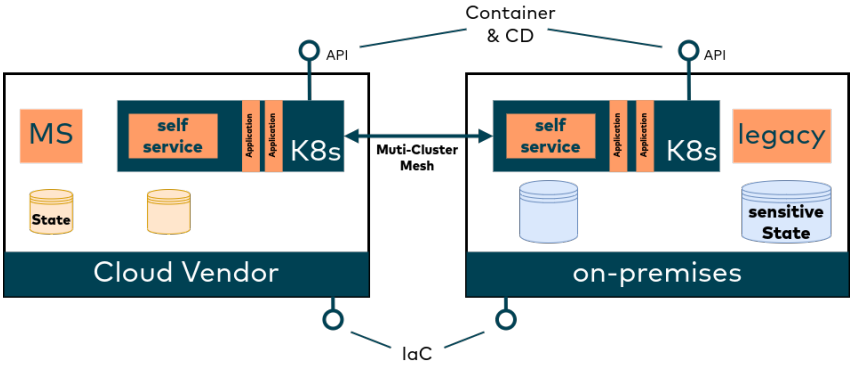


Context

Following the pattern of SCSs to split a legacy application into smaller web applications covering a part of the business domain, we also have to split the data accordingly. One way to do this is to first identify a part of the domain which shall be served by an SCS. The data belonging to this domain have to be separated from the original data source (e.g., tables, schema, etc.). The next step is to hide the corresponding frontend/API with a newly created SCS which interacts with the legacy system. The last step is to move the data into the cloud so no interaction is needed with the legacy system anymore.

Following this pattern, the data migration can happen step-by-step without breaking the availability.

6.2 Hybrid Cloud Pattern



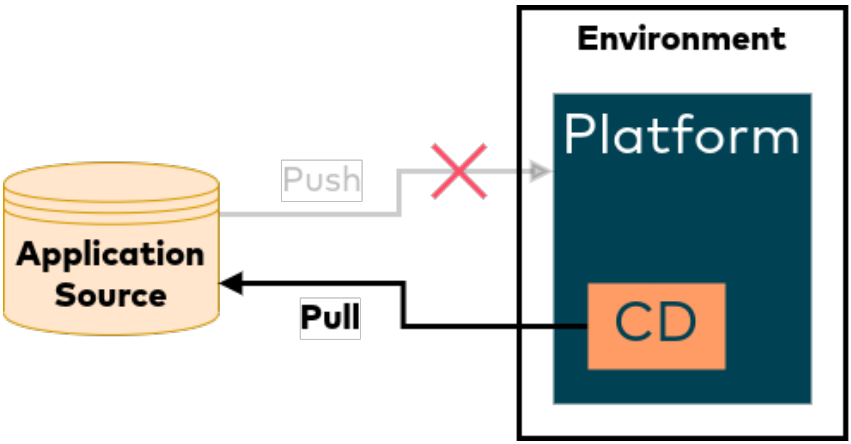
Context

In every larger system, you handle sensitive data which can probably not be migrated to the cloud so easily, due to laws, regulations, or other reasons. In these cases, you have to keep a part of your system in a secure environment, e.g., on-premise. To minimize the structural gap between cloud and on-premise, you should try to use similar tools and processes for both. Examples are:

- Use containerization to deliver your software artifacts
- Use common platforms like Kubernetes which can be operated on-premise and in the cloud
- Use common infrastructure-as-code tools to manage the platforms

The closer the tools and processes are, the easier it is to move parts of the system between the two platforms. This also includes not using overly vendor-specific tools.

6.3 Reverse Deployment Control Pattern



Context

When starting with the migration there is probably a possible CI/CD setup focusing on pushing changes into the system. It comes with security issues especially in the cloud, as the pipeline needs access to the production system. Also, other secrets are needed to configure the application properly. These issues can be avoided by reverting the control of deployment from push to pull. The pull approach as described by the GitOps paradigm moves the need to secure access management from the development environment to the production system, which by default already has the need to manage secrets. If following the GitOps approach, there are additional advantages such as:

- Auditing capability of production changes
- Disaster recovery possibility
- Full declarative definition of the production system

This approach is not only possible in the cloud but could also bring these benefits to the on-premise systems.

About the Authors

Christopher Schmidt

@fakod

Christopher is a senior consultant at innoQ Schweiz GmbH. He has been at home in software development for more than 20 years. During this time he has successfully brought numerous software and modernization projects into production in various roles. Christopher's focus is on current front / back-end technologies and highly scalable architectures. Kubernetes is his passion.

Sascha Selzer

@tommy1199

Sascha Selzer works as a senior consultant at innoQ Deutschland GmbH. He has many years of experience in development with JVM based languages and in software architecture. His current focus is on the design and implementation of backend architectures, as well as continuous delivery/deployment strategies. He also deals with cloud topics such as monitoring and tracing solutions as well as related architecture and development paradigms (microservices, devops).

Many companies are turning to the cloud, expecting a lot of advantages and innovative solutions. But moving to the cloud is not a simple process. Various challenges have to be overcome. These are not only of a technical nature but also affect software architecture, the organization itself, and its culture.

This primer shows what needs to be done and what needs to be considered to make a planned cloud migration successful and for the benefits to be realized.

innoq.com

INNOQ