# A Guidance Framework for Architecting Portable Cloud and Multicloud Applications

**Published:** 16 December 2016 **ID:** G00302796

**Analyst(s):** Eric Knipp, Traverse Clayton, Richard Watson

Technical professionals must be wary of locking their applications into a particular cloud provider's service. This report provides guidance on creating applications that are portable across cloud services, and on creating multicloud applications that span these services.

## Key Findings

- Cloud application portability is a worthwhile goal, but it must be tempered with pragmatism. Application architects must weigh the costs and benefits of portability on an application-by-application and service-by-service basis.
- An application or service has a multicloud architecture when components of the solution reside on multiple public cloud service providers.
- The trend in application architecture toward radical distribution of loosely coupled, highly cohesive components is a natural fit with multicloud deployment. (Such components include APIs, miniservices and microservices.)
- Cloud platforms offer tremendous benefits to modern applications architected for elasticity and horizontal scalability, but this value incurs the risk of provider lock-in.

## Recommendations

Technical professionals responsible for cloud application architecture and infrastructure:

- Identify your most important driver behind application portability along the following hierarchy: survivability, resiliency, performance, negotiation, and hybrid cloud and/or multicloud strategy.
- Where portability is a priority, architect contextually independent applications that have well-defined interfaces and few dependencies, which can be easily satisfied.
- Do not develop nonportable cloud applications by accident. Make nonportability an explicit choice for each application, so that it doesn't become a major problem after the fact.

- Explicitly decide whether your application architecture should be a multicloud one. Do not fall into the trap of relying on multiple cloud information services without understanding the implications they pose for your application architecture.

## Table of Contents

## List of Tables

## List of Figures

## Problem Statement

Organizations achieve substantial benefits — such as greater elasticity and faster delivery times — through the use of cloud-based applications, platforms and services. However, cloud computing brings challenges alongside its opportunities. First, cloud applications that were not initially designed to be moved between providers are not portable without substantial rearchitecting or refactoring. This isn't a problem if you decided not to prioritize portability and thus chose to make the application nonportable. However, in cases where you didn't explicitly choose to be locked in, you can end up with major headaches. In addition, because cloud application designs increasingly span multiple cloud providers' platforms or services, you must ensure that you are designing new applications to run effectively in such "multicloud" application architectures while avoiding related portability or vendor lock-in problems.

Unanticipated vendor lock-in is particularly critical because the very nature of the public cloud computing model makes the consumers of applications more vulnerable to dependencies. Literally every piece of your application or service is dependent on the platform components underneath it, and in the public cloud, these components may be entirely portable, entirely proprietary or somewhere in between. Exacerbating this challenge is the fact that cloud providers may be at risk of leaving the marketplace unexpectedly. Significant disruption is occurring in the cloud service arena, and providers must invest substantial sums just to keep up with the market. Not all providers can make such investments. At the same time, price wars for commodity services are fierce, and they will continue to drive some competitors out of the market. Because of this ongoing market volatility, customers of infrastructure as a service (IaaS), platform as a service (PaaS) or software as a service (SaaS) cloud providers may have to move before they planned to. This has already happened in several cases. For example:

- In 2013, the Nirvanix cloud storage service shut down with only a few weeks' notice.[1] Customers with many terabytes of data stored there would have been out of luck had other vendors, notably Amazon Web Services (AWS) and IBM, not stepped in to provide assistance with customer data offloading — onto their own platforms.
- In 2014, Code Spaces, a SaaS-based code-hosting site, became instantly unavailable when hackers wiped it clean, forcing the developers using it to scramble in search of backups.[2]
- In September 2014, CloudBees announced that it was leaving the application PaaS (aPaaS) market in order to focus on its core competency with Jenkins Enterprise.[3] All customers on CloudBees RUN@cloud had only a few months to move their applications elsewhere before the service shut down on 31 December 2014.
- In October 2015, HP announced that it was shutting down its HP Helion public cloud service, effective 31 January 2016.[4] HPE, the enterprise IT company created out of the HP split in November 2015, now prefers to work only on private cloud engagements. Customers were given a little over a month — until 1 March 2016 — to find another hosting arrangement for their applications and data.
- In February 2016, Verizon announced it was closing down its Verizon Public Cloud service.[5] Verizon told customers that they had two months to move their data before it would be "irrecoverably deleted" on 12 April.
- In early 2016, Facebook announced that it will discontinue Parse, its popular mobile back end as a service (MBaaS) platform.[6] Customers have until 28 January 2017 to move their products over to other platforms before the service shuts down. Because the service is an MBaaS one, moving applications involves considerable work for customers — as evidenced by Facebook's lengthy migration guide, which contains links to numerous open-source projects and related instructions.

Market challenges and the risk of vendors exiting the market are not the only reasons that contextual independence may be important to you. Other reasons to design applications for portability — or to design them for multicloud architectures — can include any of the following:

- The organization may want the flexibility to leave the provider by choice, whether due to dissatisfaction or simply the desire to improve negotiating leverage.
- The organization may require one of the following capabilities to ensure business continuity or to improve application resiliency:
  - The ability to shift an application to a backup provider or platform
  - The ability to run an application simultaneously on multiple provider platforms (in a redundant multicloud configuration)

- As part of a hybrid cloud or multicloud strategy, the organization may require portability between public cloud services and its own private cloud in the data center, or between multiple public cloud providers. It may need this portability for complete cloud applications or for only pieces of these applications (for example, microservices).

For all of these reasons, it is important for solution architects to adhere to architectural disciplines that promote portability and multicloud design. Organizations must avoid practices that unwittingly cement their cloud-native applications to a particular cloud provider's services, or that inhibit architects from using multicloud designs where appropriate.

This research provides a guide to creating portable and multicloud applications as part of a cloud strategy that:

- Minimizes dependence
- Avoids unwanted vendor lock-in
- Exploits the resiliency or functionality benefits that can be achieved through multicloud architectures

This guidance answers the following fundamental question:

> How can solution architects design and maintain cloud applications that are portable between different cloud service providers, or that operate effectively across providers in multicloud architectures?

## The Gartner Approach

Long-term digital business success requires a cloud application strategy that respects the potential need for portable cloud application deployments. In addition, to achieve resiliency benefits and exploit the lure of differentiated cloud services, solution architects must evaluate strategies for multicloud application deployment. For some applications, portability or multicloud concerns will be critical, while for others, they may matter less or not at all. You should prioritize the application portfolio for portability and multicloud needs, and plan accordingly.

Where you determine portability to be an important priority, you should design cloud applications to be *contextually independent.* As an end state, contextual independence is difficult — and sometimes impossible — to achieve. However, as a goal state, it is a worthy aim. Like perfection itself, the end result may be unattainable, but the process of pursuing perfection is worthwhile for the gains you make along the way. The three characteristics of contextual independence are:

- **Few dependencies:** Contextually independent systems can be deployed almost anywhere because they don't depend on much else to function. They are relatively self-contained.

- **Well-defined interfaces:** The means of interfacing with those systems are very well-defined and easily understood.
- **Easily satisfied dependencies:** What few dependencies the contextually independent systems have are also easy to satisfy.

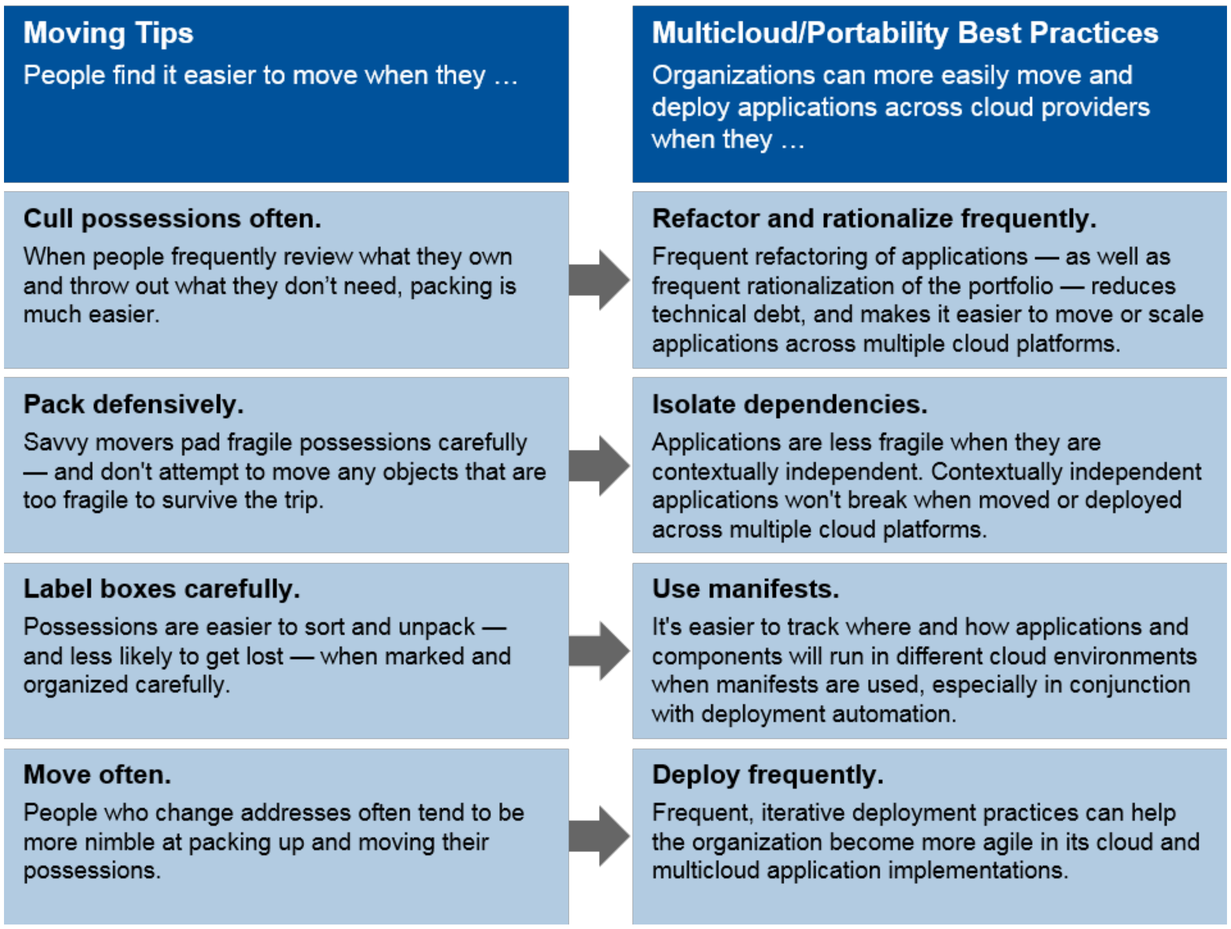
As a general rule, portability through contextual independence should be prioritized when building systematic, strategic and long-lived applications, and deprioritized when building opportunistic, tactical and short-lived applications. Applying the principles of contextual independence during the design and development of these applications will be critical. It will also be important to remain vigilant over time to ensure that this independence isn't undermined through the haphazard use of proprietary interfaces or services offered by the cloud service provider.

Applications that lack contextual independence are more fragile, in the sense that they are more likely to "break" when moved between, or deployed across, cloud providers. Contextual independence isn't the only key priority, however, when it comes to architecting applications to be both portable and multicloud-ready. Other important considerations include:

- Revisiting and refactoring legacy applications frequently to reduce technical debt
- Using application manifests and automation to ease redeployment
- Gaining agility through continuous delivery practices

In this sense, the best practices associated with designing portable, multicloud applications are similar to the practices that make it easier for families to move to a new house or apartment. Figure 1 elaborates on this analogy.

Figure 1. Moving Tips and Related Multicloud/Portability Pointers



| **Moving Tips** People find it easier to move when they … | | **Multicloud/Portability Best Practices** Organizations can more easily move and deploy applications across cloud providers when they … |
|---|---|---|
| **Cull possessions often.** When people frequently review what they own and throw out what they don't need, packing is much easier. | → | **Refactor and rationalize frequently.** Frequent refactoring of applications — as well as frequent rationalization of the portfolio — reduces technical debt, and makes it easier to move or scale applications across multiple cloud platforms. |
| **Pack defensively.** Savvy movers pad fragile possessions carefully — and don't attempt to move any objects that are too fragile to survive the trip. | → | **Isolate dependencies.** Applications are less fragile when they are contextually independent. Contextually independent applications won't break when moved or deployed across multiple cloud platforms. |
| **Label boxes carefully.** Possessions are easier to sort and unpack — and less likely to get lost — when marked and organized carefully. | → | **Use manifests.** It's easier to track where and how applications and components will run in different cloud environments when manifests are used, especially in conjunction with deployment automation. |
| **Move often.** People who change addresses often tend to be more nimble at packing up and moving their possessions. | → | **Deploy frequently.** Frequent, iterative deployment practices can help the organization become more agile in its cloud and multicloud application implementations. |

Source: Gartner (December 2016)

## The Guidance Framework

To effectively plan for portable and multicloud application deployments, solution architects should follow the five steps offered in this guidance framework:

1. **Assess portability and multicloud priorities for your application portfolio:** Examine the nature of each application and service to determine whether portability should be a priority, and to what extent you're willing to trade off portability for productivity or capability gains. Conduct this assessment on an application-by-application and service-by-service basis. Portability and multicloud priorities will vary across your portfolio.

2. **Understand and avoid portability anti-patterns:** Although they offer a substantial time-to-market upside, high-productivity platforms, proprietary interface dependencies and proprietary value-added cloud services virtually guarantee platform lock-in. Know how to identify and avoid these anti-patterns when portability is a high priority.

3. **Select enabling technologies for portable and multicloud architectures:** Examples include OS containers, PaaS frameworks and container-orchestration-based approaches. Some of the mechanisms available, such as PaaS frameworks, offer considerable promise. However, others, such as multicloud toolkits and IaaS+ middleware approaches, are potential traps around which you should tread carefully.
4. **Implement application- or service-level contextual independence:** Factor code into highly cohesive, loosely coupled components, and implement cloud architecture patterns based on the following principles: latency-aware, instrumented, failure-aware, event-driven, secure, parallelizable, automated and resource-consumption-aware (LIFESPAR). See "How to Architect and Design Cloud-Native Applications" for more information on the LIFESPAR principles. Deploy your own versions of application middleware, such as data management and integration software, rather than depending on proprietary cloud services, to preserve contextual independence.
5. **Maintain vigilance in production:** Do not let proprietary services creep in at a later date and damage your portability or multicloud value proposition. The quick time to value offered by powerful new cloud services may be a good-enough reason to sacrifice portability, but your decision to do so should be explicit.

## Step 1: Assess Portability and Multicloud Priorities for Your Application Portfolio

The first important step is to determine the cloud applications and services for which portability and/or multicloud design considerations will be a high priority. Conduct this assessment on an application-by-application and service-by-service basis, because portability and multicloud priorities will vary across your portfolio.

The portability question is critical given the commitment and effort that will go into the process of maximizing applications' contextual independence. Expending this level of effort on all cloud applications would be unrealistic, expensive and unnecessary. Thus, you should undertake a structured process to identify those applications for which future-proofing is important, and those for which it is not a major concern. As part of this process, you should consider which aspects of portability are important to your organization and whether you are prepared to bear the costs and effort involved.

In addition, you'll need to consider which applications will require multicloud deployment and what type of multicloud architecture they'll need:

- Redundant multicloud architecture, in which complete applications are replicated across multiple cloud providers
- Composite multicloud architecture, in which components of a single application are distributed among multiple providers
- Redundant and composite multicloud architecture, in which components of a single application are replicated and distributed among multiple providers

Key priorities in Step 1 include:

- Consider the planned life span and strategic-versus-tactical nature of each application or service.
- Consider the costs or effort involved in making applications contextually independent.
- Determine what kind of portability and multicloud scenarios you intend to support. For example, these may include:
  - Portability from public IaaS to public PaaS
  - Portability from public PaaS to private PaaS
  - Multicloud scenarios in which different services within a composite application need to be portable across different cloud platforms or providers
- Use the "pyramid of portability and multicloud needs" to determine how critical the advantages of portable or multicloud application designs are for your organization, and where it makes sense to trade off these advantages for productivity or capability gains.

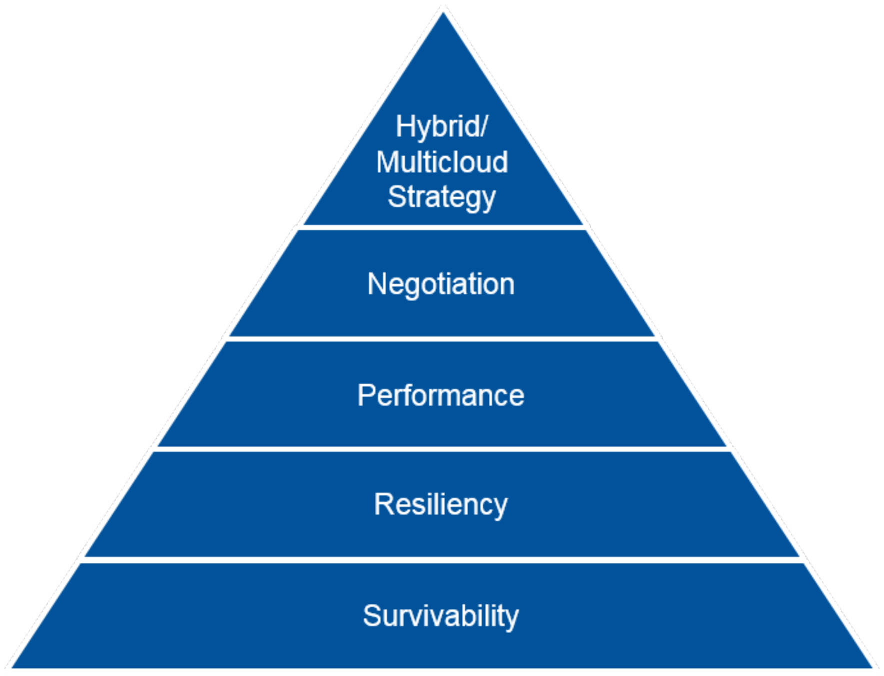
### Assess Cloud Applications in Terms of the Hierarchy of Portability and Multicloud Needs

Another important step is to weigh, for each application, the importance of portable or multicloud designs against the organization's various needs and motivations. A useful tool in this regard is the pyramid of portability and multicloud needs, which organizes the motivations for portable or multicloud designs into a hierarchy (see Figure 2). This hierarchy, which is similar to the hierarchy of needs developed by psychologist Abraham Maslow, ranges from the most basic motivations (ensuring the application survives) to the most strategic (supporting an overarching hybrid cloud/ multicloud strategy for the enterprise).

A cloud application analysis at each of these levels can help you determine:

- How critical the advantages of portable or multicloud application designs are for your organization
- To what extent your organization is willing to trade off these advantages for productivity or capability gains

Figure 2. The Pyramid of Portability and Multicloud Needs



Source: Gartner (December 2016)

For each application, consider the following:

- **Survivability:** Is it important for the application to have the ability to be taken elsewhere, without modifications, if the provider closes its doors? Can the application survive if the provider exits the market? If so, how much effort will be necessary for it to survive? Every organization evaluating cloud platforms should establish survivability as a baseline requirement for any mission-critical cloud applications.
- **Resiliency:** Is it critical for the application to be available when something temporarily goes wrong with the provider? For example, if AWS is hosting the cloud application and its data center drops connectivity for a while, can your business survive in the interim? If not, portable or multicloud application designs become more important, because you'll need to temporarily redirect traffic to other instances of the application hosted on other clouds.
- **Performance:** Do user experience (UX) requirements drive a need for fast application response under heavy utilization? Are you certain that your primary cloud platform provider can spin up resources for your application quickly enough at peak times? If the answer is "no" or "I'm not sure," be forewarned: If the application isn't a portable or multicloud one, you may not be able to scale across multiple providers, or even push your application to the edges of the network, where customers can get the best UX.
- **Negotiation:** How important is it to have leverage to negotiate a better deal from the provider? The ability to switch providers, and to use applications across multiple clouds, will increase this leverage. Consider the example of Spanish bank Banco Bilbao Vizcaya Argentaria (BBVA),

which created a framework called Hydra (now available as open source). Hydra load balances front-end applications across multiple public cloud IaaS providers, enabling BBVA to drive traffic to the best-performing cloud platform in its supply chain at moments of peak utilization. Moreover, BBVA has successfully used Hydra's capability to move traffic "at the flip of a switch" as negotiating leverage with large cloud service providers. (For more information, see "BBVA Implements a Sophisticated Multicloud Vision.")

- **Hybrid cloud and/or multicloud strategy:** How important is it for your application to simultaneously run in one of the following configurations?
  - A hybrid configuration spanning both your public cloud provider and your private data centers (see "Hybrid Architectures for Cloud Computing")
  - A configuration spanning multiple cloud providers

  To achieve this pinnacle of portability, an enterprise strategy requires a firm commitment to contextual independence at every layer of the application. Although it is expensive to achieve, this goal is worthwhile for organizations that value maximizing deployment options on a moment-by-moment basis.

Assess where your organization's priorities lie on this hierarchy of needs — from the most basic to the most evolved. Then, evaluate how much those needs apply to each application under consideration.

> The applications that really need to run in a hybrid cloud and/or multicloud configuration will likely require the most time and effort in terms of ensuring portability, creating automation, isolating dependencies and minimizing technical debt.

By contrast, if basic survivability is all that really matters, the application won't require as much investment in portability or multicloud features to meet that goal.

### Determine Application Portability Priority

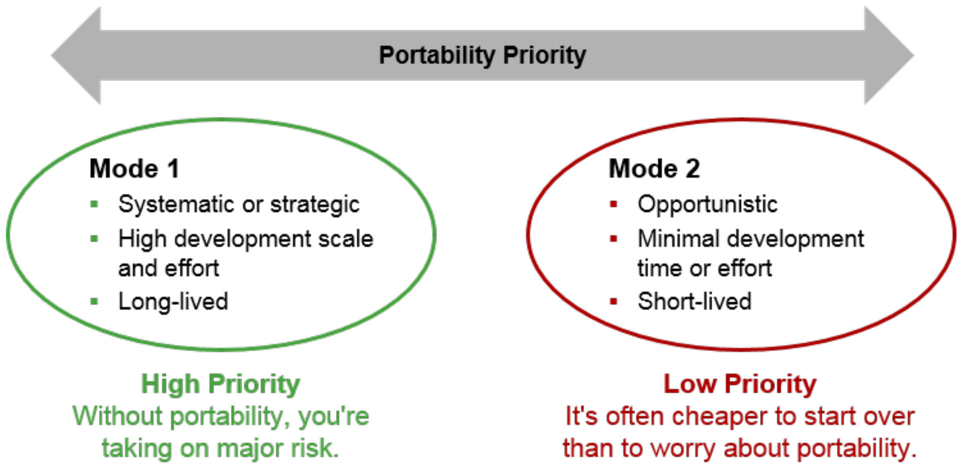
To determine your assessment of portability priority, consider the nature of each application in terms of how it will be used and how long it is likely to endure. "Bimodal IT: How to Be Digitally Agile Without Making a Mess" introduces the concept of dual-mode IT operation in the era of digital business. Mode 1 focuses on delivering reliability, while Mode 2 focuses on delivering agility. Mode 1 employs strictly governed practices to shepherd sustaining technologies and process innovations that must remain stable over long periods of time. By contrast, Mode 2 emphasizes results over processes, experimentation over stability, and disruptive innovation over the status quo. If your application is more suited to the Mode 1 delivery model, then portability should be front and center in your planning, even if you choose to implement portions of the application using agile development methodologies. Consider the following factors:

- **The high-level nature of the application:** Is the application a *systematic* or highly strategic one that will be critical to running the business or to accomplishing a long-term strategy? Or is it an *opportunistic* one that is intended to seize a fleeting opportunity to gain competitive advantage?
- **The development scale and effort:** Is the application a large-scale one that will require significant development effort? Or is it a simple one that can be created — or, if necessary, re-created — relatively quickly and easily?
- **The planned life span:** Is the application likely to be a short-lived one, or will it endure for many years to come?

These factors weigh for or against prioritizing cloud application portability as follows (see Figure 3):

- **Portability is a higher priority when** applications have a systematic or highly strategic nature, require a high degree of development effort, and/or will endure for an extended period. For these applications, the organization is taking on a huge risk if it doesn't make explicit choices about portability in its cloud application strategy.
- **Portability is a lower priority when** applications have a more opportunistic nature, require relatively minimal development time and effort, and/or will likely be short-lived. In these cases, "starting over" will likely be a cheaper and better option. Citizen-developer-oriented applications fall into this category (for more on citizen developers, see "Extend IT's Reach With Citizen Developers").

Figure 3. Prioritizing Portability Based on Application Characteristics


Portability Priority
Mode 1
Systematic or strategic
High development scale and effort
Long-lived
Mode 2
Opportunistic
Minimal development time or effort
Short-lived
High Priority
Without portability, you're taking on major risk.
Low Priority
It's often cheaper to start over than to worry about portability.

Source: Gartner (December 2016)

The systematic-versus-opportunistic level of this analysis relates directly to the considerations involved in Gartner's Pace-Layered Application Strategy. This is a framework for strategic application planning that enables IT to be responsive to differentiated business needs (see "How to Develop a Pace-Layered Application Strategy"). This approach categorizes applications into three groups:

- **Systems of record:** Applications that address capabilities that focus on standardization or operational efficiency. These applications are often subject to regulatory or compliance requirements.
- **Systems of differentiation:** Applications that enable unique company processes or industry-specific capabilities. These applications sustain the company's competitive advantage.
- **Systems of innovation:** New applications that are built on an ad hoc basis to address emerging business requirements or opportunities. Development requires an experimental environment to test new ideas and to identify the company's next competitive advantage.

In pace-layering terms, cloud applications are more likely to require portability if they are systems of record or systems of differentiation. By contrast, portability is likely to be a low-priority consideration for systems of innovation. Prioritize accordingly.

## Understand Multicloud Application Architecture

Portability isn't the only important architectural consideration associated with the various platforms and hosting options for cloud applications. A related concern involves multicloud architecture. Increasingly, organizations are interested in using multiple cloud providers within the architectural design of a single cloud application. Cloud application architects must consider the architectural options and trade-offs related to such multicloud application designs.

> The multicloud approach is an extreme form of cloud application portability. Applications are designed not just to move between different places in the cloud, but also to live in multiple places at the same time.

An application or service has a multicloud architecture when components of the solution reside on different public cloud service providers. Multicloud application architecture can come about as an explicit design choice, or as the implicit result of choosing differentiated cloud information services without regard for hosting location. The latter is incentivized by cloud information services like the Twilio voice communication API or the Box content collaboration platform. Either way, the use of multicloud applications is increasing rapidly and will continue to accelerate.

> A strategy that includes multiple IaaS and PaaS providers will become the common approach for 80% of enterprises by 2019, up from less than 10% in 2015.[7]

This means that the need to design for multicloud approaches will quickly become a reality for most technical professionals responsible for architecting cloud-native solutions. These professionals will need to clearly identify the drivers and risks associated with the use of multicloud application architectures for the solutions they design and deploy.

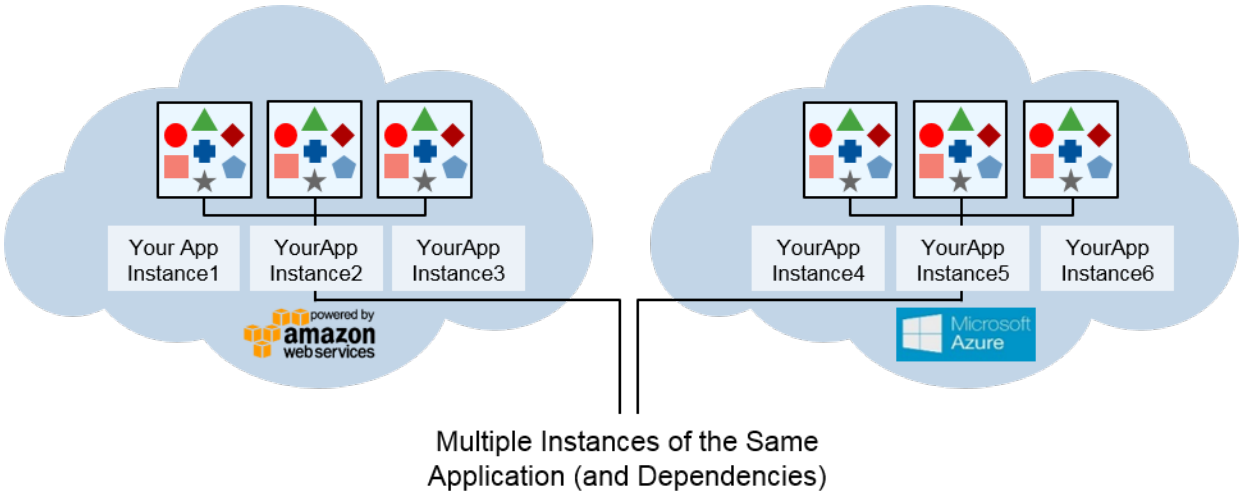
These drivers and risks will differ depending on the type of multicloud application architecture involved. There are three main types:

- Redundant multicloud application architecture
- Composite multicloud application architecture
- Redundant and composite multicloud application architecture

### Redundant Multicloud Application Architecture

In this approach, multiple identical instances of a complete cloud-native application are replicated within two different cloud providers (see Figure 4). This approach provides resiliency through redundancy: If a system failure or disruptive event occurs that causes the application to fail in one instance, traffic can be redirected to the redundant version to keep the system up and running. In addition, if load balancers are used, both applications may run simultaneously. For optimal performance, DNS can balance incoming traffic across the different cloud-hosted instances.

Figure 4. Redundant Multicloud Application Architecture



Source: Gartner (December 2016)

The redundant multicloud architecture is becoming more common — particularly for mission-critical applications. High performance and high availability are paramount for mission-critical applications, where downtime could cause major business disruptions or revenue losses. However, this architectural approach poses challenges in a few areas, such as:

- **Deployment:** The application and its updates must be implemented simultaneously across multiple cloud environments managed by multiple providers, increasing complexity.

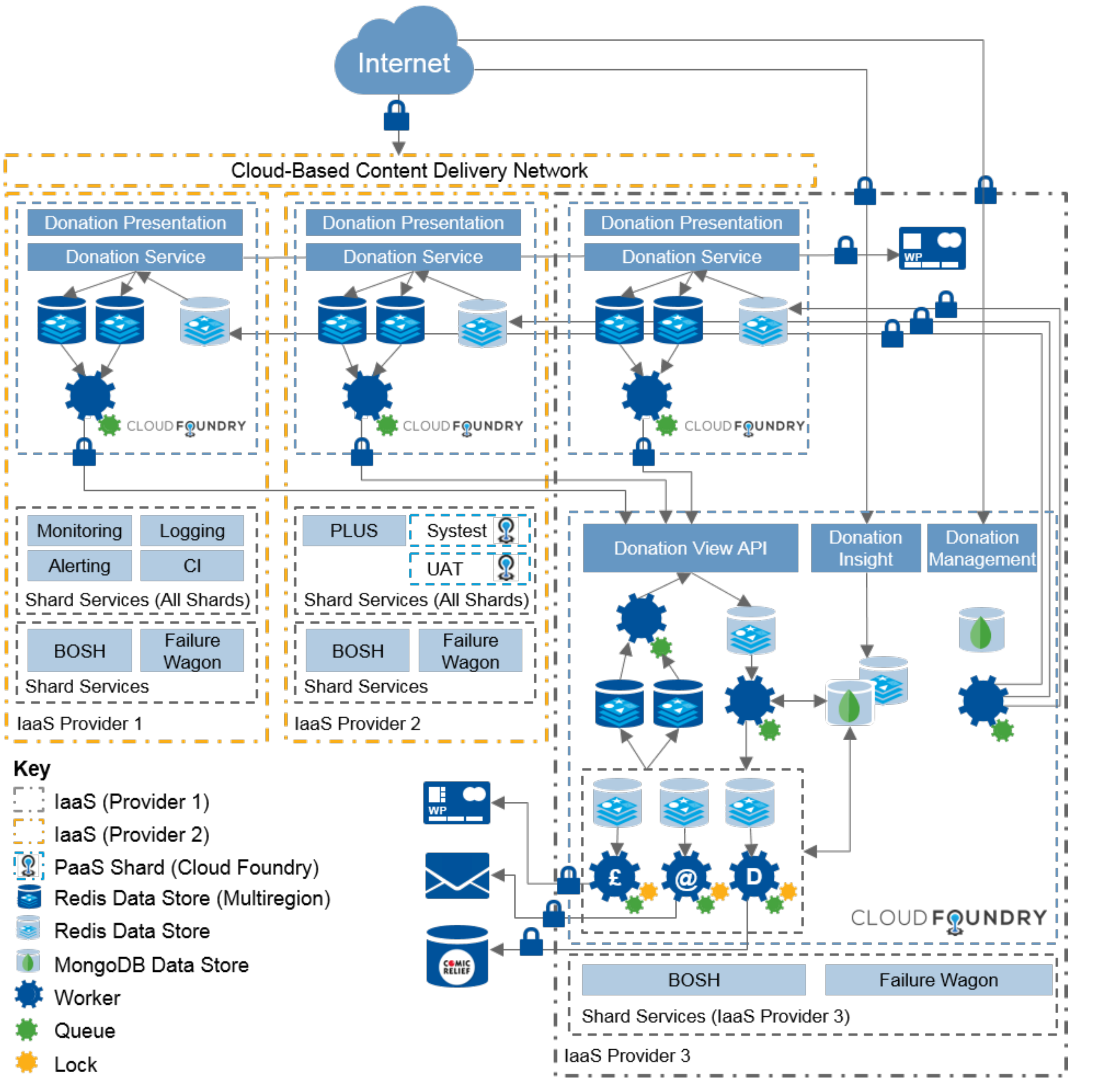
- **Monitoring and management:** The multiple instances of the application must be monitored and managed across multiple providers as well.
- **Consistency:** Maintaining consistency across nodes can be a challenge. If the two instances are running simultaneously, data may need to be frequently updated and replicated across both environments.

#### Case Study Example: Achieving High Availability With a Redundant Multicloud Architecture

One example of a successful multicloud deployment is the donation-processing platform operated by Comic Relief, a U.K.-based charity that collects donations to fight poverty in the U.K. and elsewhere. Comic Relief's donation platform tends to have huge spikes in donation volume centered around infrequent, high-profile fundraising events — such as the organization's highly popular "Red Nose Day," which is held once every two years. A major portion of the organization's contributed funding pours in during these biannual donation periods, which last only a few hours. Thus, if any system failure or performance bottleneck were to obstruct the inflow and processing of these donation payments, the financial repercussions would be severe for Comic Relief and the causes it supports.

To provide the scalability and resiliency needed to handle these spikes in donation volume, a cloud-based donation-processing platform was created for Comic Relief by Armakuni, a U.K.-based application development and consulting firm. This platform uses a redundant multicloud design, wherein redundant instances of the application/server stack are replicated and connected across multiple IaaS platforms (see Figure 5).

Figure 5. Comic Relief's Multicloud Donation-Processing Application



CI = continuous integration; D= database; PLUS = postcode look-up service; Systest = system test environment; UAT = user acceptance test environment; WP = web payment

Source: Armakuni

The application was created within the Cloud Foundry framework, which provides the necessary portability. Cloud Foundry enables the donation application to deploy across multiple providers' locations, including, at the time of this research, a private cloud vSphere deployment with a managed hosting provider and multiple AWS IaaS nodes. To maintain high availability and resiliency, the design uses DNS load balancing to drive incoming donation traffic across the different nodes. In addition, it employs asynchronous messaging to collect and forward data to a separate insight layer, which could be used for analysis and reporting purposes. Finally, by load balancing across three different payment service providers, including Braintree, Stripe and Worldpay, the design enables Comic Relief to obtain additional capacity, redundancy and resilience.

This application design enables Comic Relief to successfully process surges in donation volume without suffering any system failures or downtime. These volume surges can be up to 6,000% greater than the application's standard, steady-state volume. Comic Relief has booked record donation revenue each year it has used this platform.

As a testament to the portability engineered into multicloud architecture, Comic Relief moves the entire site to the IBM Bluemix platform during off-peak times to reduce complexity. In addition, for Comic Relief's upcoming 24 March 2017 Red Nose Day, the site will be hosted atop the open-source Cloud Foundry on AWS and atop Pivotal Cloud Foundry on Google Cloud Platform. In each case, the architects and developers responsible for the site need make only minimal changes, amounting to pointing the deployment endpoint to a different location.

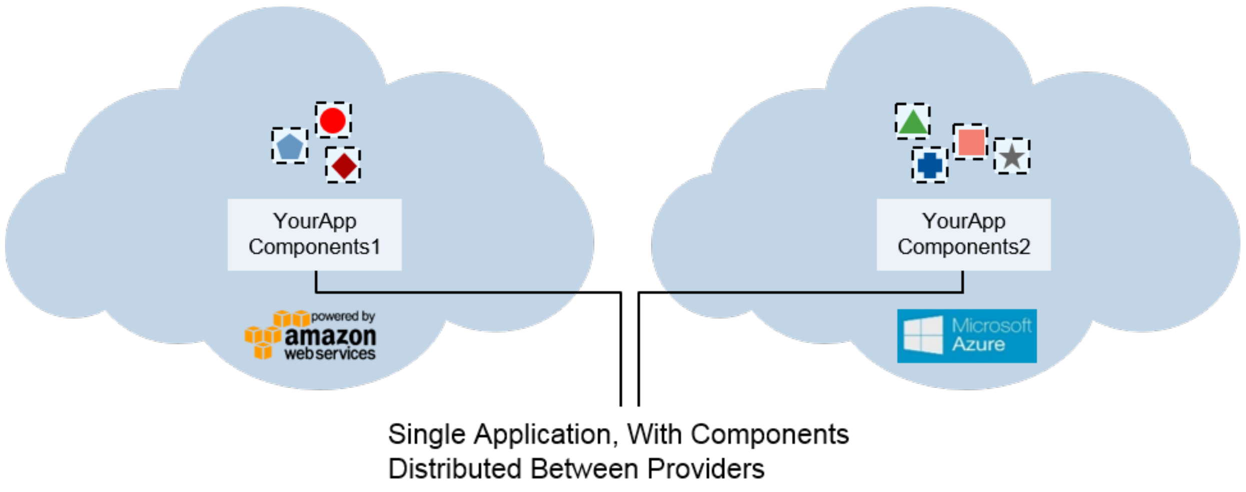
> This is the true benefit of a portable, multicloud application architecture: It keeps options open, granting architects the power to change hosting arrangements as easily as they can change a line of code.

**Composite Multicloud Application Architecture**

In this approach, different components or services that comprise a single application are distributed among multiple cloud providers (see Figure 6). Examples include:

- Applications that use multiple SaaS services exposed by APIs.
- Applications based on a miniservice architecture or a microservice architecture (MSA). In such cases, the myriad services used in a single application span more than one cloud provider's platform.

Figure 6. Composite Multicloud Application Architecture


YourApp
Components1
powered by amazon webservices
YourApp
Components2
Microsoft Azure
Single Application, With Components
Distributed Between Providers

Source: Gartner (December 2016)

Architects construct applications of this type when they wish to exploit the capabilities of multiple unique cloud services or SaaS applications within a single application. This architecture results when you build applications on top of a SaaS provider's APIs, and pull in other cloud-hosted services. For example, you might use the Twilio cloud-based telephony offering to add communications capabilities to a composite application. In the case of miniservice- or MSA-based solutions, a composite multicloud architecture may naturally result when individual application components can be more profitably scaled out across multiple providers.

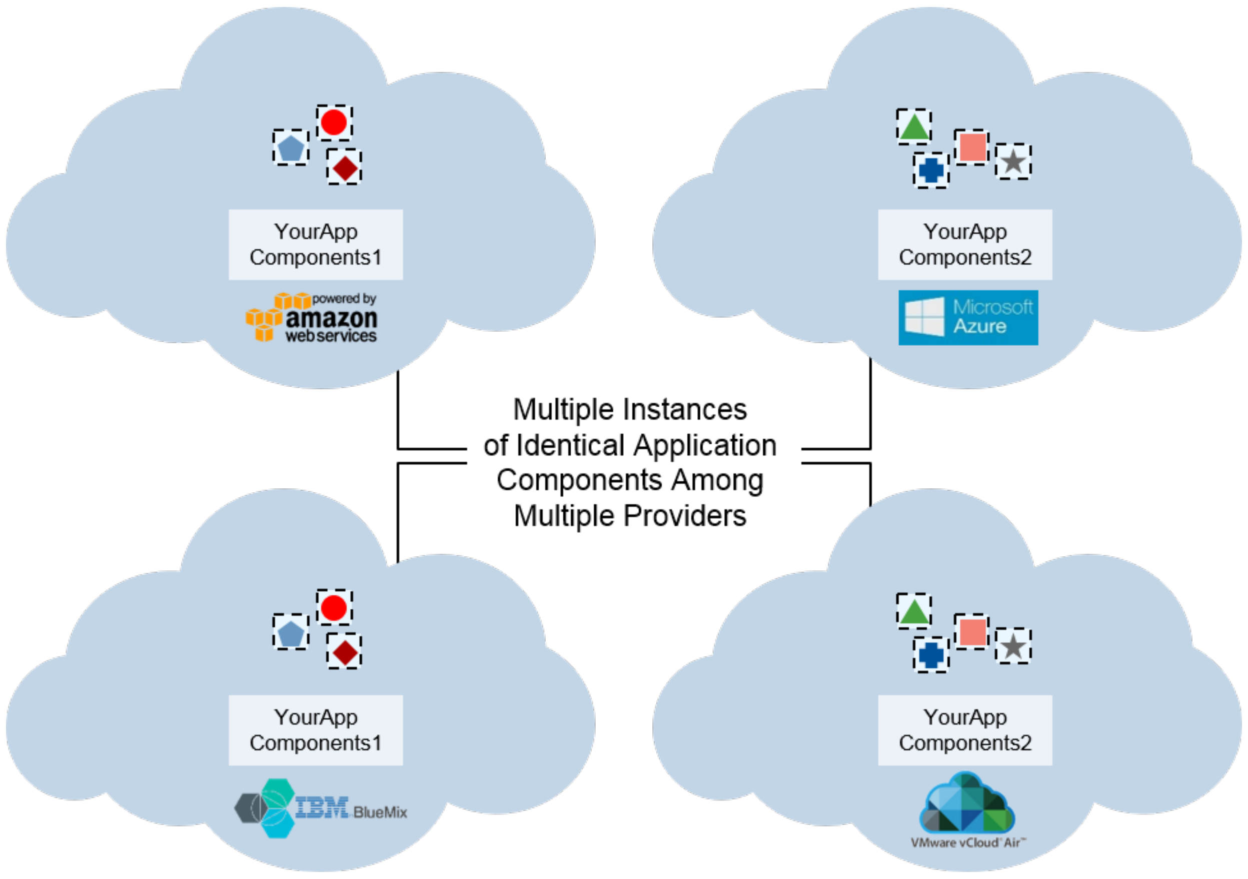
The key challenge associated with this approach is that it brings another level of complexity and effort to the application creation process, due to the advanced application packaging and automated deployment requirements involved. The specific challenges relate to:

- Deployment complexity that goes beyond that of redundant cloud architecture because individual component life cycles must be managed across multiple providers
- Management complexity involved in using different capabilities from different providers, each with its own APIs, cost models and other idiosyncrasies
- Monitoring complexity involved in assessing both the live performance of individual microservices and other application components, and the impact of those components on the overall user experience of the solution (although this can be mitigated with application performance monitoring [APM] technologies — see "Five Developer Use Cases for Application Performance Monitoring Tools")

### Redundant and Composite Multicloud Application Architecture

This multicloud architecture — the most extreme in terms of both capability and complexity — combines both of the alternatives discussed above into a single solution architecture. In this approach, a compound application, whose components are distributed across multiple providers, is also duplicated, in its entirety, on replicated instances hosted on one or more other cloud providers (see Figure 7).

Figure 7. Redundant and Composite Multicloud Application



Source: Gartner (December 2016)

This type of multicloud application architecture is the most advanced and complex of the three types discussed here, and it is unlikely to be used by most enterprises today. In theory, it could be used to simultaneously achieve the benefits of both the redundant and the composite approaches — such as achieving high resiliency while also exploiting the capabilities of multiple services from different providers, within a single application. However, it would also combine — and, thereby, compound — the challenges of both approaches, including the need to maintain cross-provider consistency across redundant application instances, and the need to use advanced application packaging and deployment disciplines. Given these compounded challenges, the redundant and composite multicloud application architecture represents a future state. To make this type of design and deployment practical on a broad scale, the maturity of organizations' multicloud capabilities, along with the maturity of deployment, management and monitoring tooling, must advance further.

**Multicloud Application Architecture Considerations**

Solution architects should expect multicloud applications to become an increasing reality in their environments, due to either the need for redundancy or the desire to exploit differentiated cloud information services. Increasingly, new applications will be compositions of existing capabilities — and, over time, more of those capabilities will themselves be offered as pay-per-use cloud information services.

Solution architects designing multicloud applications today will use either redundant multicloud designs or composite multicloud designs — rather than combining the two. Thus, you will need to weigh the advantages and trade-offs of each approach against your goals for each application to assess whether either approach is appropriate. As a general rule:

- **You are likely to use redundant multicloud application architecture when** high availability and performance are critical, and where an application needs to perform well across different geographic regions. This scenario is also more appropriate when the organization can benefit from the market dynamics of having multiple providers capable of hosting an application. This situation enhances competition among providers for the organization's business.
- **You are likely to prefer composite multicloud application architecture when** the organization wants to exploit the differentiation benefits offered by various cloud services. For example, the organization may be able to obtain superior or less-expensive compute services for its cloud application from one provider, less-expensive storage from another provider, and unique business process benefits from yet another proprietary service. This approach will come at the cost of higher complexity and effort for you, the architect.

Table 1 summarizes some of the prioritization considerations for these two multicloud approaches.

Table 1. Comparing Redundant Versus Composite Multicloud Approaches

| Use Redundant Multicloud Designs ... | Use Composite Multicloud Designs ... |
|---|---|
| For high availability and performance | For cost and differentiation benefits |
| For coarse-grained application architectures | For fine-grained application architectures (e.g., MSA) |
| For simplicity of application packaging and deployment | When the organization can handle the added complexity of the advanced application packaging and deployment required |

Source: Gartner (December 2016)

## Consider the Portability and Multicloud Scenarios for Each Application

Simply considering *whether* an application can be moved isn't enough. It is also important to consider *where* you may want to move it to. In other words, what type of platform or architecture will initially host the application, and which type do you wish to be able to move it to? Considering the multiple options involved on both sides of this question — from public cloud infrastructures and platforms to privately hosted servers — the possibilities are numerous. Potential scenarios include:

- **Portability from one public IaaS provider to another public IaaS provider:** If you host a cloud application on one IaaS service, such as AWS, you may desire the ability to move that application to Google Compute Engine or Microsoft Azure Infrastructure Services, and to run it there with minimal changes. In addition to enabling the application to move from one provider to another, this type of portability also supports redundant multicloud application architecture by allowing identical instances of an application to run simultaneously on multiple IaaS providers' platforms.
- **Portability from a public PaaS provider to a private PaaS environment:** For example, an application running on the Google App Engine platform may need to be run, with minimal changes, on a private PaaS infrastructure, such as the one offered by the open-source Cloud Foundry Foundation.
- **Portability of application components or services between multiple providers:** In what could be considered an "extreme portability" scenario, it may be important to design individual pieces of a composite multicloud application for portability, so that these pieces aren't locked into specific cloud provider platforms. For example, the individual microservices within an MSA-based application may need to be designed for portability between different PaaS providers.

These are just three examples of the many scenarios that you may consider among your portability options. Because these choices will influence the trade-offs considered for each application, they are an important component of the prioritization process.

## Step 2: Understand and Avoid the Portability Anti-patterns

Despite the best intentions to keep portability-prioritized applications contextually independent, temptations will continually lure decision makers into choices that may lock the organization's cloud applications into providers' services or environments. Several common issues may lead to choices that unintentionally violate the contextual-independence principles and practices laid out in this guidance framework.

An important early step is to learn about these issues, or "anti-patterns," in order to better recognize and steer clear of them in cloud application design. To help you remain vigilant to such dangers and counteract them when they arise, this section highlights three common portability-inhibiting anti-patterns that often lead to cloud provider lock-in, and offers guidance on how to avoid each.

### Lock-In to High-Productivity Application Platforms

High-productivity application platforms, such as Mendix, Microsoft PowerApps, OutSystems and Salesforce Force.com, offer the alluring prospect of rapid development and easy scalability. These platforms are the modern-day, cloud-based equivalents to the high-productivity fourth-generation languages (4GLs) and rapid application development (RAD) tools popular in the 1990s and early 2000s. Such offerings may be a good choice when portability isn't a priority for you — for example, when you're developing opportunistic or short-lived systems of innovation, or when you're working in Mode 2 of a bimodal application development strategy.

However, if portability is one of your major goals, you must recognize that these high-productivity cloud platforms carry a major downside risk: permanent lock-in. Any code written in them is not readily portable to another runtime environment. Granted, not all high-productivity platforms are the same: Force.com is available only as a public cloud service, while both OutSystems and Mendix license their products for deployment in a variety of public and private cloud hosting arrangements.

In the case of Mendix, the customer applications are deployed as Cloud Foundry buildpacks, conferring runtime portability between Cloud Foundry implementations. In addition, the models upon which Mendix applications are based are themselves exportable. OutSystems similarly enables customers to run generated applications separately from deployed instances of the development tools.

However, widely used SaaS offerings, such as Workday and Microsoft Dynamics AX, feature proprietary customization engines, whose use can lead to lock-in. Thus, these products are ill-suited to any strategic, long-lived and custom-made applications that are critical to the business — and for which potential portability to another provider or environment is a high priority.

**How to Avoid — Choose Control Over Productivity:** If portability is a high priority in a cloud application, opting for increased control — and harder work — over easy productivity gains is usually necessary. Thus, when it comes to your coding and platform choices, you'll need to trade off high-productivity environments, such as the ones described above, for programming environments such as Java, Ruby, Python or Node.js. In a high-productivity development environment, you can define everything easily with a few mouse clicks in a GUI, but you have less control over application design. Conversely, programming environments entail more work, but they offer greater control, enabling you to keep your application code contextually independent.

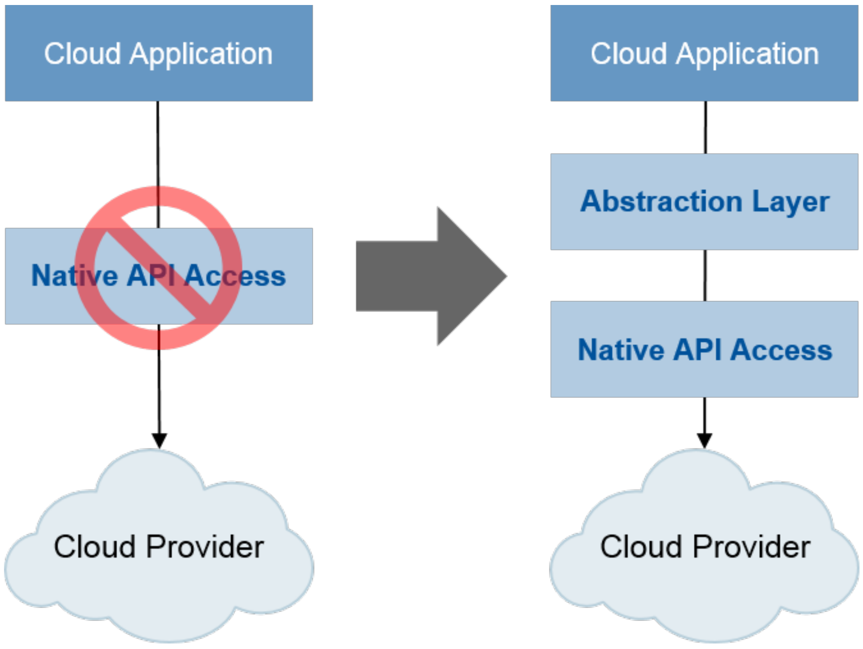
## Lock-In via Interface Dependencies

Cloud service providers offer APIs that customers can connect to their applications or to the dynamic configuration tools they build or buy. These proprietary, native APIs connect to a variety of useful external services, such as storage, mail, additional computing power or message queuing. Leading providers are devising and marketing more of these higher-level services — and attempting to bind customers to them through native, proprietary interfaces — in an effort to build their businesses, particularly given the narrow margins in today's market for lower-level, more-primitive infrastructure services.

These proprietary services offer some quick and tangible benefits, such as greater ease of use and faster scalability. However, by coding an application to access these services through native, proprietary APIs, you may be cementing that application to the provider's environment. The code will not be portable to another provider's environment without a substantial recoding and testing effort. In some cases, attempting to make such a move will be totally impractical due to the provider's own architectural decisions related to the interfaces used to access those services.

**How to Avoid — Eliminate Native-API Dependencies by Adding an Abstraction Layer:** For an application to remain portable, it is important to avoid native-API access to cloud provider resources. This means intermediating that access by adding an abstraction layer (see Figure 8). The best way you can establish such an abstraction layer is by using a PaaS framework (however, in a pinch, you can write your own abstraction layer or use an open-source multicloud toolkit). Another key to avoiding vulnerability to direct, proprietary interfaces is to adopt service-oriented architecture (SOA) principles, such as loose coupling. Alternatively, you can use microservice architecture to create independently deployable, purpose-built and cohesive components that limit the exposure to proprietary interfaces to the smallest possible footprint. In general, the higher up you can move the abstraction layer in a cloud application's architectural design (thereby insulating that application environment from dependencies), the more easily you can take that application to a new home.

Figure 8. Manage External Dependencies at Arm's Length With SOA Principles



Source: Gartner (December 2016)

## Lock-In to Proprietary Cloud Services

With increasing energy, leading cloud service providers are launching and marketing proprietary services to grow their margins and to differentiate themselves from competitors in what has become an increasingly commoditized and low-margin IaaS market. Examples of these services include:

- Amazon DynamoDB, a horizontally scalable database service
- Amazon Simple Queue Service (SQS), a messaging system
- Microsoft Azure App Service, a PaaS offering

These services are convenient and offer high productivity, but they are not conducive to portability. You cannot use these services if you want to create contextually independent applications, because you won't be able to find a compatible form of such services elsewhere. The more your cloud applications come to depend on such services, the more tightly they will be tied to the provider of those services.

**How to Avoid — Prioritize Commodity Services Over Proprietary Ones:** Because each use of proprietary services is a step away from contextual independence, you will increase portability by opting for commodity services instead. Commodity services format data and storage in a more standard way, enabling you to easily find alternative services elsewhere if you choose. Pursuing commodity services may mean opting for raw, lower-level storage or computing services at the provider level (such as blob storage and compute, which tend to be more portable than key-value storage and purpose-specific compute instances). It may also mean building or maintaining your own high-availability database, in-memory data grid (IMDG), mail server or load balancer on top of these low-level services, rather than relying on higher-level, proprietary services offered by the cloud provider. As with most portability-enhancing options, this approach will entail more work and effort on your part, but it will offer the reward of reduced lock-in. One positive development in this area is that portions of cloud infrastructure are becoming increasingly commoditized, and as a result, more services will likely emerge in the near future that don't cement customers to providers. For example, only a few years ago, there was virtually no portability at all among cloud providers — including for compute and storage resources. However, it is only natural that providers will focus most of their attention and investment on services that their competitors cannot easily replicate, so the presence of service lock-in will endure.

Before you choose to use a proprietary cloud option, ensure that you fully understand and accept the portability-reducing trade-offs involved. This advice applies to all of the provider-generated options described above — high-productivity application platforms, native API interfaces and proprietary services. Granted, each of these options can be a beneficial choice, rather than a "pitfall," if you are *intentionally* trading off contextual independence (for example, to gain speed or productivity benefits for an opportunistic, short-lived application). The key is to make these choices explicitly, rather than to allow reduced portability to happen by accident because you didn't consider or understand the implications when you made your decisions.

## Step 3: Select Enabling Technologies for Portable and Multicloud Architectures

After examining the relative importance of portability for each application, the next step is to consider the approach or mechanism you will use to achieve it. Most of the options we discuss focus on the lower levels of the cloud computing stack. Our reasoning is simple: Although business value increases as you travel up the stack from IaaS, through PaaS and to SaaS, the options for remaining portable simultaneously decrease.

### Understand How Cloud Delivery Models Impact Contextual Independence

The three key aspects of contextual independence — reduced dependencies, well-defined interfaces and easily satisfied dependencies — range from attainable at the IaaS level to unachievable at the SaaS level (see Figure 9):

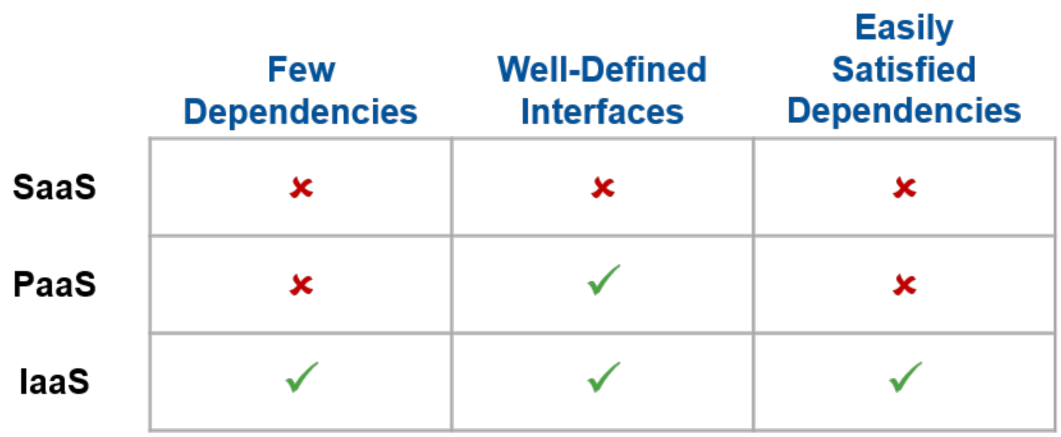
- **SaaS is basically nonportable:** The number of dependencies and the nature of the interfaces are unknown because the customer has no visibility into, or control over, how the software runs. The bottom line is that, if you choose SaaS, you have made the decision to forgo portability entirely, hopefully in the interest of substantial business value that cannot be achieved through other practical means.

- **In PaaS, portability may be achievable on a limited basis:** The provider operates the runtime, and the dependencies are under its control. However, the interfaces the application uses to interact with the PaaS — that is, the APIs or the command line interface (CLI) scripts — are published and should be clear. The customer won't be able to easily satisfy the dependencies unless one of the following is true:
  - The provider has built the platform using an open framework and has published any deviations from it.
  - The provider sells a "canned" version of the PaaS operating software.

  An example of this challenge exists with a platform like IBM Bluemix. Although IBM Bluemix is based on the Cloud Foundry framework, it's been embellished with substantial enhancements (such as Watson services). If used, these enhancements can increase an application's degree of lock-in to the Bluemix platform. Thus, just because a public cloud PaaS runs on a framework — either open-source or proprietary — does not mean that an identical platform can be made available on another provider's infrastructure. On the other hand, as the Comic Relief case shows, judicious avoidance of proprietary capabilities helps architects preserve contextual independence.
- **For IaaS, portability of simple virtual machines (VMs), OS containers and the data they contain is a base assumption:** Of course, the customer must stick to the IaaS options and avoid value-added PaaS features or application services that use proprietary APIs. Using IaaS requires the architect to select OS, middleware and runtime components, increasing the complexity of the development project. However, an organization's best shot at achieving perfect portability is to stick with pure IaaS. Bear the following in mind, however, if you choose to use scalable storage services like Amazon Simple Storage Service (S3): You may eventually achieve a level of data gravity that makes moving the application difficult (even though doing so may be theoretically possible from an application architecture perspective).

  The Open Container Initiative (OCI) provides an avenue for future OS-container-level portability between cloud service providers, and it is already supported by Microsoft, AWS and Google. Thus, if you can keep your application and all its dependencies within a container today, you can already guarantee portability between providers. It is worth noting that this is a relatively new phenomenon — even as recently as a year ago, no such mechanism existed.

Figure 9. Contextual Independence and the Cloud Computing Tiered-Service Model



| | Few Dependencies | Well-Defined Interfaces | Easily Satisfied Dependencies |
|---|---|---|---|
| SaaS | ✗ | ✗ | ✗ |
| PaaS | ✗ | ✓ | ✗ |
| IaaS | ✓ | ✓ | ✓ |

Source: Gartner (December 2016)

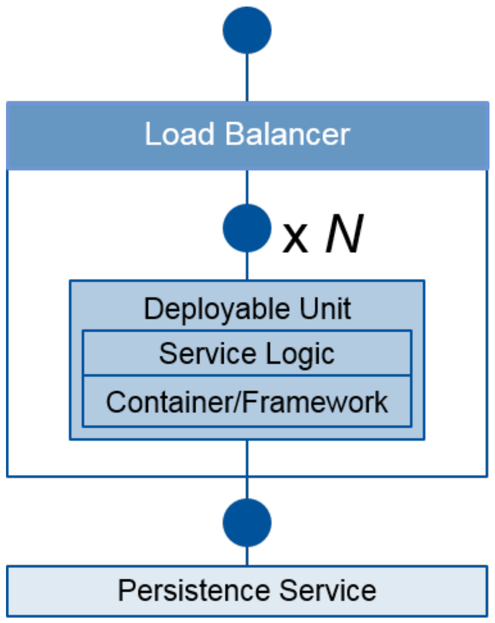
## Choose a Portable Outer Architecture to Gain Contextual Independence

Contextual independence is a result of sound application architecture decisions and implementation. Start your journey toward a contextually independent architecture by recognizing that every application or service includes both an inner and an outer architecture. We describe these perspectives as follows (see "Assessing Microservices for Cloud-Native Application Architecture and Delivery"):

- **Inner architecture:** The software architecture of an individual microservice and what it exposes to the outside world
- **Outer architecture:** The architecture of the operating environment and distributed management ecosystem in which your microservices will be built, deployed and executed

In other words, the inner architecture is the smallest individual deployable unit of your application. It includes both the business logic you've developed and the runtime containers or frameworks that support this logic (see Figure 10).

Figure 10. Component Inner Architecture

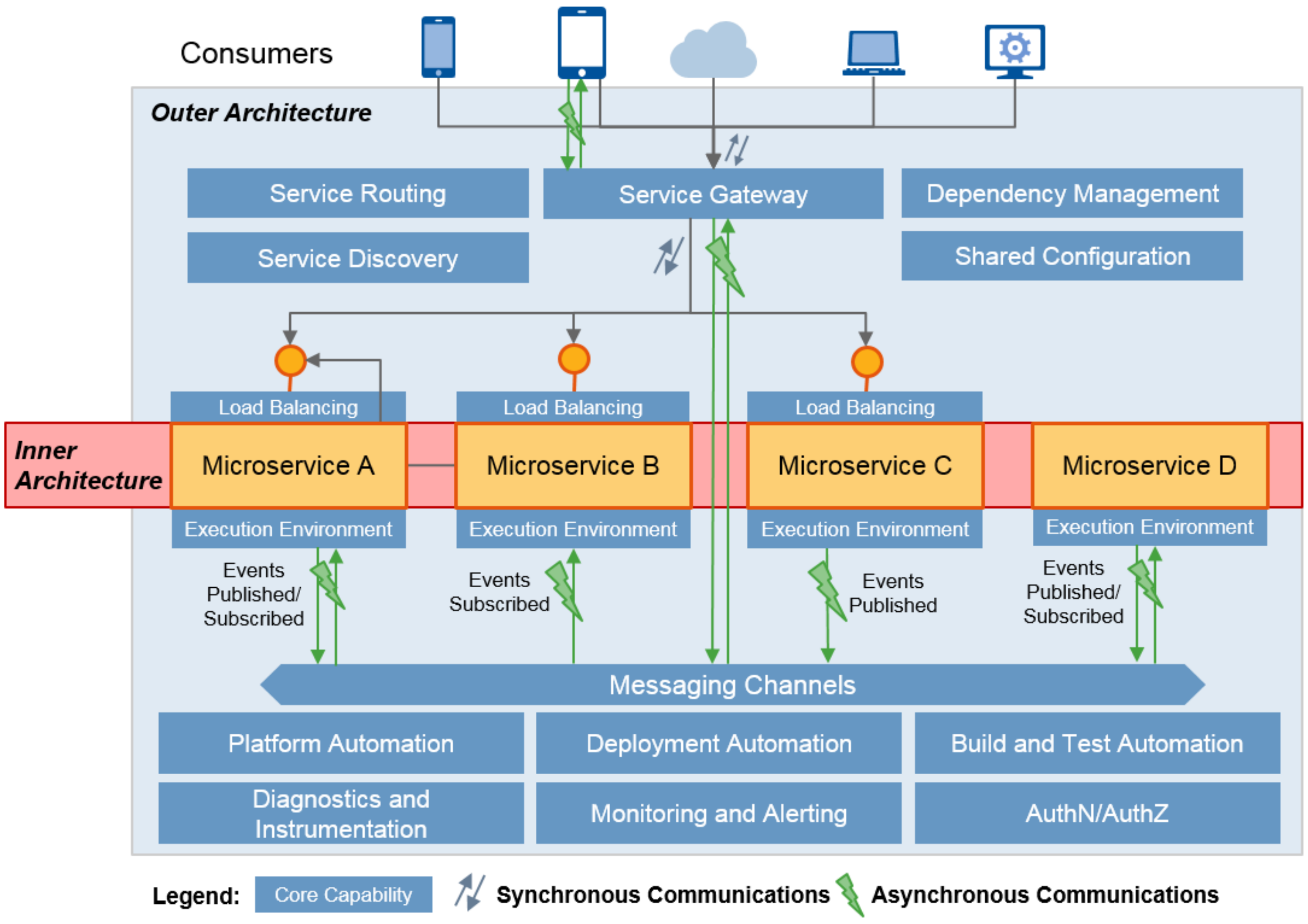


Source: Gartner (December 2016)

This lightweight inner architecture implies external dependencies, like the persistence service illustrated in the figure. Modern applications require all manner of external backing services, ranging from the aforementioned persistence, to messaging, to identity and access control. Do not make the mistake of assuming that, because your inner architecture is as simplistic as possible, you have automatically gained contextual independence and, therefore, have captured the benefit of portability. The external dependencies remain, and you must deal with them by engineering a portable outer architecture that encapsulates all of the infrastructure needed to support your applications and services. That way, when you take the outer architecture with you to a new hosting location, you can be assured that the various dependencies of the inner architecture can be satisfied.

Figure 11 provides an example of what this outer architecture might look like, specifically in the context of hosting an MSA-based application. Besides hosting the individual microservices themselves — and their associated inner architecture — the outer architecture includes a range of capabilities necessary for the runtime life cycle of each piece of the application.

Figure 11. Example of an Outer Architecture Hosting an MSA-Based Application



AuthN = authentication; AuthZ = authorization

Source: Gartner (December 2016)

All of the portability mechanisms discussed in this document focus on either the PaaS or the IaaS layer of the cloud stack. These mechanisms are listed in Table 2, along with a summary of our recommendations regarding their use. The sections that follow describe these five "pathways to portability" in detail, including their dependencies, interfaces, strengths and weaknesses.

Table 2. Five Portability- and Multicloud-Enabling Technologies

| Technology | Recommendation | Comments |
|---|---|---|
| OS containers | Recommended as a partial solution | Cloud application architects should be aware that containers are *not* a complete portability strategy. Containers address only the inner architecture of an application, service or component along with a manifest that defines what a container holds. |
| PaaS frameworks | Recommended | PaaS frameworks provide a portable outer architecture, with a few limitations. |
| Multicloud toolkits | Not recommended | These toolkits require a high level of developer expertise and effort. Interest in and adoption of these toolkits are waning. |
| IaaS+ middleware | Not recommended | IaaS+ middleware is not a true portability solution. |
| Container-orchestration-based approaches | Recommended with caution | These approaches require a high level of effort and expertise. However, with the right effort, expertise and commitment, these approaches provide the most flexible portable outer architecture available. |

Source: Gartner (December 2016)

## OS Containers

OS container technologies, such as the wildly popular Docker framework, provide the promise of portability for an application and its dependencies — provided that those dependencies can be packaged within a container. OS containers provide a relatively new type of application virtualization that depends on the advanced resource-sharing capabilities in modern server operating systems. "What You Need to Know About Docker and Containers" defines the technology as follows:

> "Shared OS virtualization is a technology that takes advantage of features of an OS kernel that isolate processes and provide control of CPU and memory resources to those processes. Those isolated processes are called 'OS containers' or 'containers.'"

Every commercially viable PaaS framework now fully embraces and implements the container metaphor. However, from a portability perspective, containers are only a partial solution. The vast majority of backing services — such as relational databases, messaging middleware and IMDGs — are not designed to run within an OS container. Thus, attempting to directly port these services creates risk. Eventually, container-native middleware will emerge to address this shortcoming. In the meantime, Docker and other OS container technologies provide a portability solution for inner architecture only. The Dockerfile[8] and the OCI Image Manifest Specification[9] also provide a manifest concept that improves the portability of individual containers. Although containers are critical enabling components for large-scale cloud application portability and multicloud application architectures, they do not yet fully address these needs by themselves.

> OS containers provide a strong foundation for portable inner architecture and are therefore a partial portability solution.

- **Dependencies:** A host OS or another framework or service capable of hosting OS containers
- **Interfaces:** Docker or rkt format container images, APIs and/or CLIs for creating, building, running and distributing containers
- **Strengths:** Simple packaging and distribution model for portable inner architecture
- **Weaknesses:** Not a complete portability solution for applications that require outer architecture components (such as MSA-based applications) and/or stateful backing services (such as virtually every enterprise application)

**PaaS Frameworks**

The PaaS framework approach is similar in concept to an application server and is particularly enticing for private cloud scenarios.

> The PaaS framework enables a complete runtime environment — including all orchestration and tooling needed to manage the scaling and health of the platform — which can be contained within a cluster of either physical servers or VMs.

Examples of PaaS frameworks include:

- Apprenda (commercial closed source)
- Cloud Foundry (available as open source and with several commercial distributions, such as Pivotal Cloud Foundry and HPE Helion Stackato)
- OpenShift (available as open source or as a commercial distribution from Red Hat)

Industry momentum has coalesced around OpenShift and Cloud Foundry, with the latter moving into a position of prominence, led by the Cloud Foundry Foundation, which boasts impressive industry membership.

Using a set of VMs, rather than a cluster of physical servers, to host a PaaS framework is a potentially effective option for providing portability. The VM-contained PaaS environment can be run on, and potentially moved between, different public cloud IaaS providers. Thus, a customer could implement one Cloud Foundry-based environment within its own data center, implement another one on a public IaaS provider, and then run cloud applications on — or move them between — either environment. In fact, PaaS frameworks must support APIs from multiple major cloud service providers and abstract these APIs from the user. Otherwise, these frameworks risk becoming irrelevant. This is why Cloud Foundry implements the Cloud Provider Interface (CPI), a uniform set of APIs that abstract away the underlying differences between cloud service providers. To date, a number of cloud service providers, including AWS, Microsoft and Google, have implemented their capabilities to Cloud Foundry's CPI specification. Both open-source and commercial cloud management platforms (CMPs), including OpenStack and VMware's vSphere, have also added CPI support.

PaaS frameworks like Cloud Foundry support their own sets of APIs and interfaces, which are portable to other implementations of the same PaaS framework. Portability is good when moving between identical implementations. However, if any services in the cloud application interact with the native APIs or proprietary features of the IaaS provider hosting the PaaS environment, contextual independence will be lost, and portability will be reduced.

- **Dependencies:**
  - Basic compute and storage provided by VMs on a supported CMP or IaaS platform
  - Alternatively, bare-metal physical compute and storage resources
- **Interfaces:** CLIs and REST APIs offered by the PaaS framework, such as those issuing commands to create new services or to push application code
- **Strengths:** Portability across VMs, IaaS platforms and bare metal (provided that the PaaS framework has been deployed there)
- **Weaknesses:**
  - Operational commitment, since you're still responsible for running, maintaining and updating the software
  - Opinionated architecture, meaning that you're limited to the choices that the framework authors have made for you

### Multicloud Toolkits

Multicloud toolkits provide a programmatic abstraction layer for interacting with CMP and IaaS-platform-native APIs and proprietary services. Examples include Dasein Cloud, Apache Libcloud and jclouds, and fog. These toolkits are developer-oriented libraries written in programming languages such as Java or Ruby. These libraries abstract an application's code or management scripting from a cloud provider's APIs. Such toolkits enable cloud application developers to code their applications to interact with an IaaS environment's storage or computing functionality while using abstracted interactions to hide the implementation underneath. In theory, this approach provides greater portability by enabling the cloud application to be moved from one IaaS or CMP environment to another, without being locked into the provider's APIs.

However, there are serious drawbacks to this approach. The multicloud toolkit ecosystem is fragmented and immature. Moreover, in recent years, use of these toolkits has not gained any traction. Instead, interest in this approach appears to be waning. Because these are open-source approaches, the low commitment rate is cause for concern. If you adopt one of these toolkits, you may ultimately be forced to support it yourself.

To date, almost all of these toolkits have been developed around the AWS feature set. Support for other IaaS platforms, while available, remains much more basic and limited. In addition, these toolkits involve a lot of effort and a considerable learning curve. Learning the APIs for a few important functions from several providers can actually be easier than investing the time to learn a multicloud toolkit.

> We do not recommend the use of multicloud toolkits as a practical portability mechanism.

- **Dependencies:** Supported CMPs and/or IaaS platforms (carefully evaluate provider and platform coverage for your needs)
- **Interfaces:** Native-language SDKs (for example, Java SDKs) that invoke APIs
- **Strengths:**
  - Portability across some CMP/IaaS platforms
  - A true abstraction layer for the supported provider functionality
- **Weaknesses:**
  - Low uptake and interest from developers
  - Additional developer effort and expertise required
  - Incomplete or inconsistent provider and platform coverage

### IaaS+ Middleware

This portability mechanism combines IaaS with legacy middleware, such as an application server. Two different approaches are common. The first leverages offerings that create a curated middleware stack for each role in an application (an application server, a database server, a load balancer and so on). Examples of these offerings include Engine Yard's application management platform and AWS's Elastic Beanstalk and CloudFormation services.

The second, more basic approach is to virtualize applications developed within a legacy middleware environment — such as IBM WebSphere, Oracle Fusion Middleware or Microsoft Windows Server — and run them "as is" within a VM on an IaaS platform. This "lift and shift" approach amounts to packing up whatever existed in the enterprise's server environment, placing it into a VM image and treating that virtualized environment like a hosted server.

Approaches like these provide good portability and the advantage of getting a cloud environment up and running quickly within the provider's hosting arrangement. However, they really amount more to advanced hosting than to true cloud computing. They will not provide opportunities for any cloud orchestration, for example. Customers remain in control of certain aspects of managing the operating environment, such as choosing the timing of updates to the underlying VM's operating systems, and configuring application health checks from the monitoring services available. Because customers maintain this control, and choose from a discrete set of OS and middleware options, they need to retain knowledge of these choices when the application is migrated elsewhere. For example, if a Java application is configured to use a specific version of Apache Tomcat 8 on Java 8, any migration target environment also needs to support that version of Tomcat 8 on a Java 8 runtime.

- **Dependencies:** A VM. You need an IaaS provider, CMP or virtual infrastructure of your own in which the application and its associated middleware stack can be deployed.
- **Interfaces:** Middleware-specific interfaces. Whatever APIs are exposed by the application server, the database server and any other middleware remain proprietary. They are portable only inasmuch as you can install the same middleware somewhere else and under the same configuration profile.
- **Strengths:** Portability across all VMs. Because you take ownership of the long-term management of the middleware, you are able to carry it with you. Provider automation that helps deploy your application and middleware stack will, of course, not be portable, nor will configuration metadata that describes, orchestrates and scales the virtual infrastructure.
- **Weaknesses:** More of an advanced hosting approach than a true cloud computing one.

### Container-Orchestration-Based Approaches

The rising interest in OS containers, particularly for hosting microservices, is driving a growing interest in container orchestration technologies. Container orchestrators are a class of runtime management technologies built to run and manage container workloads, such as Docker and rkt, on a cluster of host servers in public or private clouds, according to placement and service-level policies. (For more information, see "Orchestrating Docker Container-Based Cloud and Microservices Applications.")

> When combined with other open-source and commercial components, a container orchestrator like Apache Mesos, Docker Swarm or Kubernetes can indeed provide the backbone of a home-brewed PaaS framework.

Container orchestrators enable you to roll your own PaaS framework with ultimate flexibility over the composition of the platform. Besides a container orchestrator, you need to pick out other components of an overall outer architecture solution. You can choose from a wide variety of open-source pieces to mix and match a custom-made hosting environment (see "How to Succeed With Microservice Architecture Using DevOps Practices"). This "get your hands dirty" approach is attracting intense interest among end-user organizations as a way to gain total control over the platform. From a portability perspective, this approach can lend more independence to cloud applications, because it avoids lock-in to commercial PaaS or middleware providers' interfaces. Once your self-assembled platform is deployed on a cluster of VMs, it could be hosted on any number of public cloud platforms (or even in your own data center).

This do-it-yourself model is a good solution only for a very specific type of enterprise. The enterprise's business model must be able to justify the extra time, money and resources required to make this approach work. Examples of such business models include those that are purely digital and those that involve building and running PaaS environments for other customers. In these cases, you may be able to gain enough benefit to justify the skills and staff required to put together your own platform and run it.

However, for the vast majority of organizations, this approach is not practical. To develop, integrate and deploy a cloud operating environment comparable to a public cloud PaaS product or a PaaS framework, an organization would need to:

- Use multiple open-source software (OSS) offerings
- Integrate and test the OSS components
- Devote engineer time and effort to stay on top of all the nuances and updates involved in these "moving target" OSS environments

On the other hand, if you are totally committed to containers, you can take advantage of the cloud services that support container-level hosting and get application portability in the public cloud with minimal pain. You're better off focusing your time on business logic instead.

- **Dependencies:**
  - A VM (or physical hardware)
  - Use of container technology
- **Interfaces:** Interfaces specific to the frameworks you choose, although all provide APIs and command-line scripting options
- **Strengths:**
  - Total control over platform capabilities and behavior
  - A wide variety of OSS components that can do the job
- **Weaknesses:**
  - Incredible complexity and technical responsibility for the IT organization, which must maintain and monitor multiple "moving targets"
  - Ineffective without an all-in commitment to containers

## Step 4: Implement Application- or Service-Level Contextual Independence

Earlier, we defined contextually independent applications as those with:

- Few dependencies
- Easily satisfied dependencies
- Well-defined and easily understood interfaces

At a fundamental level, contextual independence requires a strong commitment to avoiding external dependencies — a concept that is easy to understand but often hard to follow. In practice, implementing application-level contextual independence means applying architectural design patterns that don't inherently bias your application toward external dependencies. The principles described here apply regardless of which pathway to portability you choose for your applications.

### Favor Modular and Composite Design

How many pebbles can you fit into a given jar? The answer, of course, depends on the size of the pebbles. By decreasing the footprint of software components, or VM images, you can fit more into the same volume. What is your deployment footprint? Is it a 2GB application server? Or is it a minimalistic Docker container image of a microservice bundle that includes a servlet engine, an HTTP server and your application's code within a Java Archive (JAR) file? Finer-grained component design also gives you the ability to run different types of workloads in a finite shared-resource set. In a shared virtual and dynamic environment, you can not only include more small pebbles, but also accommodate pebbles of different shapes and colors.

This notion of granularity impacts design and requires solution architects to think more about composite design than monolithic design. In composite design, you need more of the different types of pebbles to create an application. Well-designed services are simple, autonomous and single-purpose. More complex tasks get done by composing simple services. Modularity makes overall system design testable and comprehensible. A module implementer can focus on the implementation, optimization and verification of the module, with minimal concern about the rest of the design.

### Choose Rightsize Application and Service Deployment

The n-tier application architecture model has been around so long that modern enterprise application design has matured to an art form. Architects are good at creating cleanly separated components with object orientation (OO) patterns, and then carefully layering those components on top of each other with abstractions embodied in frameworks such as Spring. However, at deployment time, they stuff all the components and all their dependencies into one monolithic archive file for deployment into an application server. Small changes, even if isolated to specific areas of an application, require repackaging and redeploying the entire application (and, sometimes, restarting the virtual machine instance hosting the application). This is the stuff of maintenance and operations nightmares. This monolithic approach to deployment will not support the agility you need to move and change your applications dynamically around the different topology options that the cloud gives you.

Besides its obvious agility limitations, monolithic architecture makes individual pieces of the application hard to move — indeed, there are no individual pieces to move. Dependencies are baked into the very fabric of the application's architecture. They are almost impossible to accommodate in other environments without applying heavy effort toward either refactoring the system or replicating all of the underlying dependencies to which the application is coupled elsewhere. Replicating proprietary public cloud services may not even be possible, in which case, you're stuck. Moreover, because the whole application is a monolith, the whole application is stuck, not just the components requiring the external dependency.

Instead of a monolith, choose a rightsized application architecture that offers the best combination of simplicity and agility for your project. Choose the most appropriate implementation architecture to support your nonfunctional requirements on a service-by-service, application-by-application and project-by-project basis. Do not assume one size fits all (see "Decision Point for Application Services Implementation Architecture"). This could be a coarse-grained architecture or a finer-grained one, depending on needs. When in doubt, make sure to loosely couple the inner architecture of whatever you build, even if, as a deployable unit, it resembles the monolith of old.

### Explore Microservice Architecture

Microservices have quickly gained traction in advanced application architecture discussions for their unique blend of cohesiveness and scalability. MSA is a pattern for building and delivering service-oriented applications with three primary objectives: agility of delivery, flexibility of deployment and precision of scalability. A supplemental benefit is the ability to choose, and update, implementation technology on a per-service basis. The principles that make microservices independently deployable and manageable are the same ones that make them more portable than traditional applications and services. Because a microservice bundles all of its capabilities into a lightweight unit, it is a good architecture to satisfy the constraints of contextual independence. Even if you're just learning about MSA now, you should strongly consider experimenting with it if you anticipate that portability will be a high priority for future cloud application development in your organization.

## Commit to Frequent Refactoring

Refactoring is the process of improving your existing code without changing its function. Refactoring improves the nonfunctional attributes of your codebase, such as the overall code quality to improve readability. It helps you manage the technical debt that increases maintenance costs and hinders your ability to understand the software system. Architectural refactoring emphasizes continuously improving the architecture as system needs evolve (see "From Fragile to Agile Software Architecture").

Refactoring is a skill you should practice regularly, no matter what kind of application you're building. However, refactoring is particularly important when you're implementing a portable or multicloud architecture.

Refactoring for portability offers up additional benefits beyond simply paying down technical debt, including:

- Discovering inadvertent or implicit dependencies that have been introduced into the codebase, ensuring that you have the opportunity to mitigate those dependencies before you need to move the application
- Identifying opportunities to simplify the inner architecture of your solution by making components more independently deployable and portable

You should dedicate a portion of each project phase to paying down technical debt through refactoring. If you're using agile practices, commit to spending a day per iteration purely on refactoring or to executing a refactoring sprint every few iterations. Either way, refactoring will bring benefits to both the quality and the portability of your solution.

### Run Your Own Backing Services

All modern applications require backing services to provide the capabilities that organizations need but aren't willing to build from scratch. Backing services are commonly referred to as application middleware. Examples include technologies like:

- Database platforms, including both relational database management systems (RDBMSs) and NoSQL databases
- IMDGs
- Message-oriented middleware

Without such backing services, even the simplest enterprise application can't do much work.

Cloud service providers are only too happy to oblige with hosted PaaS capabilities that eliminate the burden of operating what can be complex middleware. AWS SimpleDB, Microsoft Azure Service Bus and Google Cloud Functions are all fine examples of PaaS capabilities that reduce your operational effort and speed time to market. When possible, cloud service providers prefer to use OSS as the basis of these solutions. For example, both AWS and Microsoft use the open-source Redis IMDG as the basis of their caching services. What's not to love about this relatively portable approach to PaaS-style backing services?

Indeed, basing service offerings on OSS is a commendable vision, and the cloud providers have made their service offerings inherently more portable by doing so. However, that's only part of the story. If you rely on the cloud service provider to operate your backing services for you, you're not building the skills and internal capabilities needed to run that part of the infrastructure yourself. In other words, you can certainly pick up Redis and configure your own caching cluster, but without experienced team members that know how to run it, you won't be successful. The same is true for other types of middleware products: To get good at running them at scale, you simply have to run them yourself.

> When you value portability highly, you should strongly consider running your own backing services so that you maintain the skill set required to operate them independently of the cloud service provider's platform capabilities.

### Prove Contextual Independence Works via Continuous Delivery

Famous management consultant and researcher Peter Drucker once said, "What gets measured gets managed."[10]

> The best way to measure your ability to implement, deploy and operate a contextually independent application is to do it — repeatedly.

Therefore, continuous delivery must be a precursor to any serious effort at contextual independence.

By providing objective evidence, continuous delivery gives you peace of mind that you can deploy your application whenever you need to, wherever you need to and for whatever reason. Engineering a continuous delivery toolchain and practice that allows you to reconfigure your deployment targets is a must. How else will your application adjust to a new home, should it require one?

For those who haven't begun to travel it, the road to continuous delivery is a long one lined with pitfalls. Begin by adopting agile practices. Be honest with yourself about just how much you need to improve before you can depend on automation to be a help, rather than a hindrance, to your efforts at contextual independence. Then, recognize that there is no "end" to the journey — the process of continuous measurement and improvement *is* the destination. Start traveling toward yours with the "Solution Path for Achieving Continuous Delivery With Agile and DevOps."

## Step 5: Maintain Vigilance in Production

Application planners may do everything right to make an application portable upon its initial release. However, this doesn't mean that the application's portability will be guaranteed in the future, or that its multicloud capabilities will translate to every cloud service provider. You must remain vigilant to prevent proprietary services, interfaces or other dependencies from creeping in at a later date, because they will damage the contextual independence needed to maintain application portability over time. In addition, such postdeployment changes may impact the integrity of a multicloud application design. If capabilities are added to a cloud application service or instance hosted in one cloud provider, will the same capabilities be applied to the rest of the multicloud application? Or are different, inconsistent versions being created instead?

Applications spend the majority of their lives in maintenance mode, but during that time, they don't simply run without being altered. Rather, they tend to get upgraded, enhanced, integrated or otherwise modified over time. The people making those decisions may ignore, or be unaware of, the prior design commitment to contextual independence. Thus, they may add proprietary services or interfaces that lock the application into the cloud provider's environment.

> Vigilance and communication will be especially critical to maintaining long-term portability and multicloud integrity for cloud applications.

The likelihood of this scenario increases over time, as the people in the original team move on to other projects, or even to other organizations. Although the original team may have designed a very coherent approach to portability, the new people making enhancements or decisions may not value, or be aware of, that original portability approach. Moreover, the longer the application exists in the organization without ever moving to a different provider or platform, the more likely it is that complacency and apathy will set in. Over time, the application may require functionality to scale out faster. To meet this requirement, the future maintainers of the application may gradually start pulling in off-the-shelf services, such as elastic load balancers. Later, the need or desire to move the application may suddenly arise. However, the organization may discover that portability is impossible without major rework, because the changes that crept in deteriorated the contextual independence that was designed into the application at its birth.

A similar scenario unfolded countless times in past middleware scenarios. System planners and IT leaders were determined to avoid the use of lock-in-inducing features, such as stored procedures that are proprietary to one database vendor's version of SQL. However, later on, either out of ignorance or out of defiance, someone used those features anyway, leaving the enterprise locked into a nonportable implementation. With cloud applications, the risk of this scenario is likely even higher. Unlike today's cloud environments, the traditional IT environments of the past typically didn't pose opportunities for constant contact with the provider. In the public cloud, users may confront promotional messages about a new service or functionality every time they access a management portal or log into their consoles. Although they may be useful, these services or functions may not necessarily be portable.

## Risks and Pitfalls

### Pitfall 1: Overengineering Portability Without a Business Case

There are various drivers and levels of portability, and each has an increasing cost. Architecting the "perfectly portable" application is extraordinarily expensive and will rarely be necessary. Every reader of this document will be able to internalize that fact. Yet, it is easy to fall into the trap best articulated by the well-known, embellished fundamental theorem of software engineering, originally credited to David J. Wheeler:

> We can solve any problem by introducing an extra level of indirection (except for the problem of too many levels of indirection).[11]

Layers of indirection, in fact, do improve the portability of cloud applications and, therefore, are largely the focus of this document. Yet, organizations can inadvertently overinvest in them, creating cost and complexity not justified by the business value of their efforts. Multicloud toolkits provide a cautionary tale. Although they once appeared to be a promising component of a portable cloud application architecture, waning interest has turned them into an expensive proposition. The promised support for more services and providers has not materialized. Thus, the organizations that invested in multicloud toolkits might as well have built their own, perhaps simpler or more internally consistent, abstraction layers.

Mitigate this risk by clearly understanding and applying this document's guidance around the bimodal nature of cloud application architecture portability. Based on this research, you should already understand that one size doesn't fit all — systematic projects are more likely to demand investments in portability than opportunistic ones, for example. However, you can take this understanding further by fully analyzing the need for portability on a project-by-project basis. Don't overinvest in portability for solutions that you're never likely to move.

### Pitfall 2: Investing in Future Value That Never Materializes

Cloud application portability is like an insurance policy. Even if you've established a clear business case for portability — for example, a provider's survivability, or lack thereof, would dramatically impact your business — you may never get the chance to cash it in. Major cloud service providers may experience temporary outages at any time, but if you build your application according to the LIFESPAR principles, you may never be impacted by those outages (see "A Guidance Framework for Architecting Highly Available Cloud-Native Applications"). However, not engineering for portability leaves you exposed and uninsured.

Unfortunately, this simply isn't a risk that can be mitigated. You either invest in portability or you don't. Presuming you invest in portability under the right circumstances — mission-critical, systematic applications that deliver substantial business value — portability is just a cost of doing business. You need to clearly communicate this point to the leaders and other stakeholders in your organization. Your failure to allocate adequate resources to portability considerations, on the basis that the business didn't want to spend the money, won't amount to much of a defense when a provider disruption negatively impacts your business.

### Pitfall 3: Home-Brewed Solutions Cost Too Much to Implement and Maintain

Architects with great interest in near-total control over the elements of the outer architecture have begun experimenting with Kubernetes and other container orchestration tools. The near-infinite flexibility offered by a fully custom-made approach is seductive to both managers and architects.

> Managers see the opportunity to invest in an asset that will provide potentially differentiating value to the organization, while architects see the opportunity to invest in skills that will provide potentially differentiating value to their resumes.

If organizations can truly invest in the necessary skills and retain them, a home-brewed approach to outer architecture that depends on OSS and container orchestrators can indeed be a winning formula. Yet, the majority of enterprises report a preference to "buy before build" when possible. To pursue differentiating qualities that deliver superior business value, however, some organizations may opt for the do-it-yourself approach to outer architecture. Even if these organizations consider the various cautions in this research, they may still find themselves with spiraling budgets. They may devote countless person-hours to plumbing out the myriad open-source projects and to making the regular updates needed to keep everything secure, working and up to date.

Mitigate this problem by truly understanding the role of "buy versus build" in your organization. Choose a PaaS framework over the home-brewed, container-orchestrator-driven approach, which can create cost overruns and skills shortages. Technical professionals without prior portability and multicloud experience will have their hands full simply building the right cloud-native inner architecture to go with a relatively turnkey PaaS experience. Focus on business value first.

## Gartner Recommended Reading

*Some documents may not be available as part of your current Gartner subscription.*

"Bimodal IT: How to Be Digitally Agile Without Making a Mess"

"Extend IT's Reach With Citizen Developers"

"How to Develop a Pace-Layered Application Strategy"

"BBVA Implements a Sophisticated Multicloud Vision"

"How to Architect and Design Cloud-Native Applications"

"Hybrid Architectures for Cloud Computing"

"Five Developer Use Cases for Application Performance Monitoring Tools"

"Assessing Microservices for Cloud-Native Application Architecture and Delivery"

"What You Need to Know About Docker and Containers"

"Orchestrating Docker Container-Based Cloud and Microservices Applications"

"How to Succeed With Microservice Architecture Using DevOps Practices"

"Decision Point for Application Services Implementation Architecture"

"From Fragile to Agile Software Architecture"

"Solution Path for Achieving Continuous Delivery With Agile and DevOps"

"A Guidance Framework for Architecting Highly Available Cloud-Native Applications"

## Evidence

[1] "The Nirvanix Failure: Villains, Heroes and Lessons." Network Computing.

[2] "Murder in the Amazon Cloud." InfoWorld.

[3] "Early PaaS Provider CloudBees to Shut Its Service." CIO.

[4] "HP's Helion Public Cloud Will Soon Ride Into the Sunset." InfoWorld.

[5] "Verizon to Shut Down Its Public Cloud." ZDNet.

[6] "Moving On." Parse blog.

[7] See "2016 Strategic Roadmap for Data Center Infrastructure."

[8] "Dockerfile Reference." Docker.

[9] "OCI Image Manifest Specification." GitHub.

[10] "What Can't Be Measured." Harvard Business Review.

[11] B. Lampson. "Principles for Computer System Design." Microsoft. 1993.

**GARTNER HEADQUARTERS**

**Corporate Headquarters**
56 Top Gallant Road
Stamford, CT 06902-7700
USA
+1 203 964 0096

**Regional Headquarters**
AUSTRALIA
BRAZIL
JAPAN
UNITED KINGDOM

For a complete list of worldwide locations,
visit http://www.gartner.com/technology/about.jsp