How to Modernize Your Application to Adopt Cloud-Native Architecture

Published: 30 March 2020 **ID:** G00451153

Analyst(s): Traverse Clayton

For existing applications to gain the agility, scalability, resilience and cost benefits of cloud-native computing, application technical professionals modernizing existing applications must find and address critical hot spots in the architecture.

Key Findings

- Application modernization and migrations to the cloud fall short of stakeholder expectations because of the organization's inability to identify change requirements and size the effort required for the changes, as well as overly complex migrations and misalignment of the migration outcomes with business goals.
- Lift-and-shift migrations of precloud applications never achieve cloud-native goals. Successful migrations start with individually assessing applications to determine the degree of remediation required.
- Applications based on service-oriented architecture (SOA) best practices and distributed computing patterns require less remediation for cloud-native adoption.
- Architectural challenges in cloud computing include adapting to elastic scalability and coping with increased latency between components. Your architecture will have "hot spots" that limit its ability to deal with these challenges and must be identified, assessed and remediated before adopting cloud-native architecture.

Recommendations

Technical professionals responsible for migrating applications to a cloud-native architecture must:

- Assess the architecture of each application to identify hot spots that need to change and to what degree in order to meet cloud migration and modernization goals.
- Prioritize the assessment and resolution of your application's hot spots alongside other backlog items. This should be no different than adding new use cases, fixing bugs or repaying technical debt.

- Identify and fix critical hot spots, including single points of failure and improper state management to mitigate the risks of your application failing in a cloud-native environment.
- Resolve hot spots where your application interfaces with external dependencies that are not being migrated along with the application.

Table of Contents

List of Figures

Problem Statement

This document was revised on 31 March 2020. The document you are viewing is the corrected version. For more information, see the Corrections page on gartner.com.

> How do I modernize my application to support my organization's business goals when migrating it to a cloud platform?

Enabling your existing "precloud" applications to take full advantage of cloud computing is fraught with both technical obstacles and difficulty in meeting stakeholders' expectations. "Greenfield" applications adopting cloud-native architecture suffer much less from the burden of these challenges, but these new applications only make up a small percentage of an organization's application portfolio. Existing applications represent the majority of your application portfolio. Modernizing these applications to take advantage of cloud computing requires setting and meeting stakeholder expectations early and often. To address these challenges, start by assessing applications to:

- Identify what needs to be changed by looking for key hot spots in the architecture
- Analyze to what degree the hot spots need to be changed
- Evaluate the costs, resources, effort and time required to make the changes

A typical legacy application that has been rehosted on infrastructure as a service (IaaS; i.e., migrated using a "lift and shift" approach) cannot take full advantage of cloud characteristics. Organizations that follow the rehost approach often find out afterward that the migration team took shortcuts and came up with workarounds just to get it running. These shortcuts and workarounds arise due to underestimated time constraints placed on teams and the lack of a comprehensive assessment of the application. One example is viewing cloud as a vessel for cost savings, when the reality is that per-CPU-hour costs for cloud are often higher than they are in a colocation or data center. Due to stakeholders' misconceptions, their high expectations of cloud migration are often not met using the rehost strategy. This leads to a pessimistic view of what cloud computing has to offer.

For your team, organization or company to meet or exceed your stakeholders' expectations for their investment in cloud computing, you must assess how your application's architecture limits adopting cloud-native architecture — and address those limitations.

The Gartner Approach

This research guides teams to make incremental changes to their applications to adopt cloud-native architecture principles. For some applications with simple requirements, required changes are minimal. For instance, if the goal is to reduce operating expenditure (opex), a less-mature, but still appropriate, variation called a “cloud-ready” application architecture is required. A cloud-ready application can run safely in the cloud and embody some cloud characteristics, but it does not fully exploit all that the cloud has to offer.

The guidance framework assumes your deployment targets are cloud-native platforms. Cloud-native platforms prioritize flexibility and elasticity over single-instance availability. A cloud-native platform supports:

- Using cloud characteristics in the application runtime to improve elasticity, availability and scalability of applications
- Composing applications and services and agile and distributed development to provide a consistent experience for development and operations teams
- Deploying code frequently through continuous integration/continuous development (CI/CD) to reduce risk and improve the safety of releases
- Developing and testing on identical configurations to production through immutable infrastructure and containers to increase predictability and repeatability of releases
- Automating configuration changes and system updates to large fleets of applications to improve operational efficiency, security and business continuity of applications

Cloud-native platforms can be categorized into platform as a service (PaaS), function PaaS (fPaaS), containers as a service (CaaS) and self-hosted platforms on IaaS. These platforms have varying degrees of support to address the challenges of moving an application to the cloud. Examples of cloud-native platforms include Cloud Foundry, Red Hat OpenShift, Amazon Web Services (AWS) Lambda, Amazon Elastic Kubernetes Service (Amazon EKS), Microsoft Azure Kubernetes Services (AKS), Google Kubernetes Engine (GKE), AWS Elastic Beanstalk, Microsoft Azure App Service, and Google App Engine. Cloud-native application platforms provide application development services and features beyond just raw compute, storage and infrastructure services such as Amazon Elastic Compute Cloud (Amazon EC2) or Azure Virtual Machines (Azure VM) scale sets. Cloud-native application platforms take advantage of these infrastructure services and provide abstractions to support the aforementioned attributes of cloud-native architecture.

Adopting cloud computing requires careful planning and a deep understanding of your existing applications’ impediments to cloud-native architecture. Gartner clients often cite poor planning and inappropriate migration strategies as reasons for failed cloud adoption. Without planning and structure, decisions will be made in-flight just to get the application running. This will lead to misalignment with stated project goals.

A Note on Cold Spots

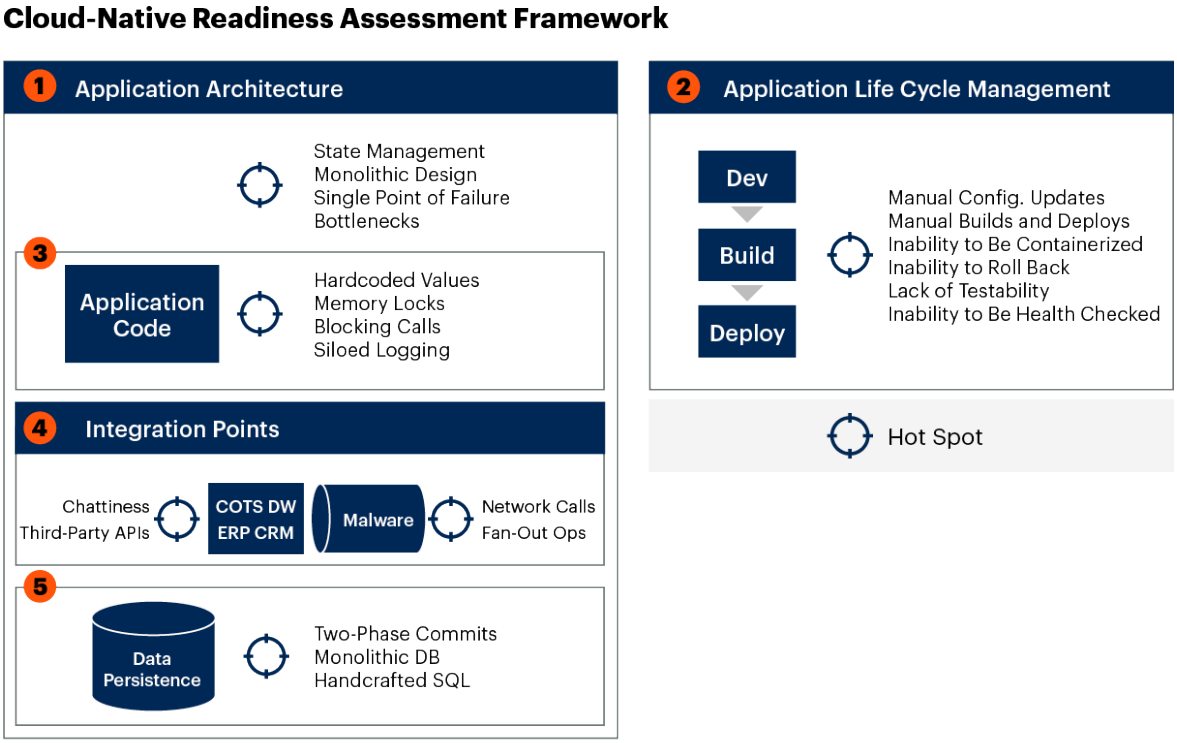
Some areas of your application will require little to no change. Change will vary from application to application. For example, if you are following SOA best practices and applying distributed computing principles in your applications today, then you could be well-positioned for running in a cloud environment. The objective of this framework is to identify the common hot spots that can be troublesome when running in a cloud environment, not to identify those areas that are commonly not an issue (i.e., cold spots).

The Guidance Framework

The guidance framework (see Figure 1) helps identify, assess and fix the common hot spots within your application that need to be addressed prior to running in a cloud-native context. The hot spots outlined in this framework are aligned with common architecture anti-patterns. When prioritizing budget, this framework should be applied prior to making an investment decision as to whether you will rehost, revise, rearchitect or rebuild your application.

> In order for existing applications to take advantage of the promises of cloud computing, take a systematic approach to assess your architecture, identify what needs to change and change it.

Figure 1. Cloud-Native Readiness Assessment Framework



Source: Gartner

451153_C

The framework provides a structure for your assessment, but there is no strict order to follow. An application cannot be viewed through a single lens, but must be considered holistically. Each domain within the framework is closely related, and the impact of rearchitecting one part of the application potentially impacts another part of the application.

This note builds on other research, including:

- "Guidance Framework for Modernizing Microsoft .NET Applications"
- "A Guidance Framework for Architecting Highly Available Cloud-Native Applications"
- "Decision Point for Choosing a Cloud Migration Strategy for Applications"
- "Solution Path for Applying Microservices Architecture Principles to Application Architecture"
- "From Fragile to Agile Software Architecture"

Prework

Cloud migrations typically encompass a mix of applications, some of which are due for an overhaul or an update and can benefit from what the cloud has to offer. These initiatives have varying business goals from one organization to the next. The decisions, process and people behind the initiatives are specific to your organization. However, Gartner has identified and outlined recurring technical goals. Your prework prior to assessing your application's cloud readiness consists of:

- Ensuring clarity of goals to guide downstream architecture choices
- Measuring the "migrateability" of an application
- Preparing application scorecard

Ensure Clarity of Goals to Guide Downstream Architecture Choices

It is critical that your cloud migration efforts be supported by specific business goals and that stakeholders up, down and across the organization are aligned with those goals. Before moving forward, assess and drive alignment of your organization's and stakeholders' goals for migrating the application to the cloud.

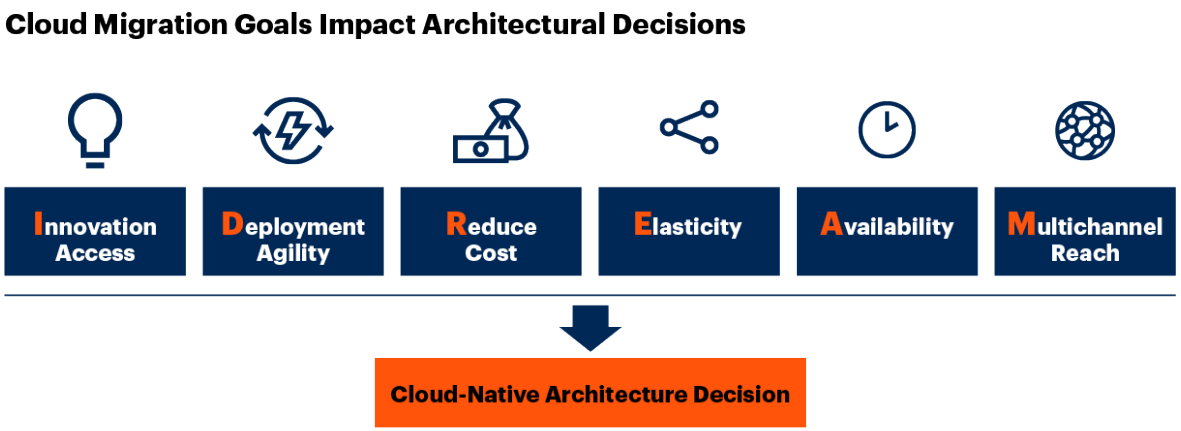
Without this alignment between strategy and execution, the team delivering on those goals will not be positioned to deliver a successful project. In order for everyone to be successful in the delivery of cloud-native capabilities, all constituents need to have a common frame of reference for making project and architectural decisions.

The goals of your stakeholders will largely dictate the architectural decisions you make downstream. For instance, not all application migrations to cloud will require rebuilding. You may not need to change the way your database is partitioned for scalability because that is not one of the goals of the cloud migration. Ask the following questions to determine what — and to what depth — you need to assess your application:

- Why are the stakeholders asking to move this application to the cloud?
- What are their desired outcomes?
- How much are stakeholders willing to spend?
- What if your assessment says that the rearchitecture effort will blow out the budget allotment?
- What are the priorities of stakeholders that will increase the criticality of addressing specific hot spots over others?

Figure 2 outlines the common goals that are strongly aligned with migrating to cloud and how these goals directly impact your cloud-native architectural decisions.

Figure 2. Cloud Migration Goals Impact Architectural Decisions



Source: Gartner

451153_C

For more details on cloud strategy and technical and business alignment, see:

- "How to Develop a Business Case for the Adoption of Public Cloud IaaS"
- "Designing a Cloud Strategy Document"
- "Build the Right Justification for Moving to the Cloud"
- "Applying a 'Cloud-First' Checklist to Ensure Successful Sourcing and Business-IT Alignment"

Measure the Migrateability of an Application

The migrateability of the domains represented within the guidance framework can be established by assessing each of the hot spots. The more easily changeable your application is, the less work you will need to do in order to adopt cloud-native architecture. The less changeable your application is, the larger the investment in time, effort and resources required to adopt cloud-native architecture will be.

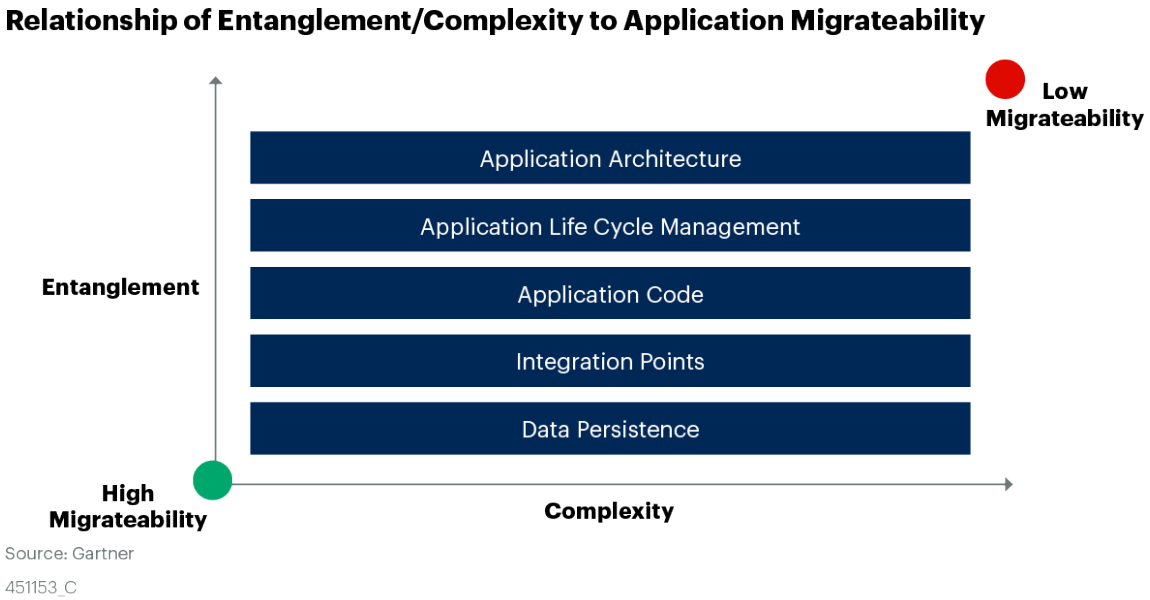
The two major inhibitors that lower the migrateability of your application are *entanglement* and *complexity*:

- **Entanglement:** Entanglement can be viewed as the number of interdependencies both within and outside of your application. For example, from a code perspective, you would examine how blocks of code interact with each other as well as the call graph — including how the methods, classes and functions are composed together. A script or block of code that is reliant on other blocks of code is considered entangled.
- **Complexity:** Compounding the issue of entanglement is the complexity of your architecture and code. Application components lead to complexity, particularly if they are tightly coupled and

heavily dependent upon specifics of the underlying software and hardware. This complexity limits your options as far as deployment, runtime and hosting are concerned when targeting cloud-native platforms. It is important to implement abstractions and encapsulation of those underlying dependencies and keep application components together that are similar and change together. In this sense, complexity means having dependencies that are hard to fulfill. Application-level dependencies — for example, dependencies between components of the application — can also play a part (see Figure 3).

> "Complexity is the single major difficulty in the successful development of large-scale software systems."[1]

Figure 3. Relationship of Entanglement/Complexity to Application Migrateability



Ignoring your technical debt throughout the life cycle of your application commonly leads to a phenomenon call the "Big Ball of Mud."[2] Common pitfalls that lead to a big ball of mud include:

- Tight coupling between application components
- Shared state
- Tangled interdependencies
- Lack of governance and naming conventions
- Various workarounds and hot fixes that solve the problem of the day but never land back into the main branch of code

- Ripping and replacing commonly used libraries (e.g., logging)

A big ball of mud is highly entangled and complex and equates to your application being difficult to change. Reducing the entanglement and complexity of your application requires work across all domains of your application, not just the code level.

A Note on Technical Debt

As a general practice, you should view your application as a living, breathing thing that needs to be groomed and cared for in order to keep your technical debt low and improve the migrateability of your application. The level of technical debt that an application carries tends to worsen over time. Eventually, it may be more cost-effective to replace the application rather than trying to tackle the technical debt that weighs it down. An application that meets this criterion of carrying technical debt will almost always need to be rebuilt or replaced. See “A Primer on Technical Debt” and “Protect Your Legacy and Technical Debt” for more information.

Prepare Application Scorecard

A self-scoring assessment spreadsheet accompanies this note. You can access it via the Download icon on the left rail of gartner.com. The spreadsheet follows the framework and uses the same structure of domains and hot spots. Review the spreadsheet in advance, and make modifications you believe are necessary to reflect the goals, priorities, culture and nuances of your organization. This includes adding new hot spots, removing hot spots or potentially adding new domains (e.g., security or operations).

Use the framework in conjunction with your self-scoring assessment spreadsheet to review the hot spots in your application and assign values to the hot spots:

1. **Low migrateability:** Hard to remediate, high level of effort required.
2. **Medium migrateability:** Moderate to remediate, moderate level of effort required.
3. **High migrateability:** Easy to remediate, minimal level of effort required.

Assess entanglement and complexity factors when reviewing hot spots to determine how changeable the hot spot is. Ask these questions:

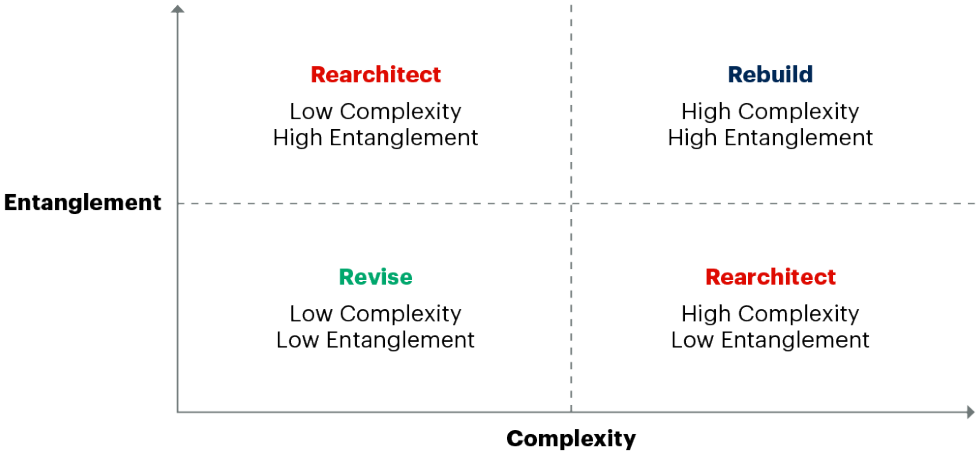
- How critical is the hot spot?
- How badly will it impact future-state cloud goals?
- How much effort is required to resolve it?
- How many interdependencies are there for a given hot spot?
- What is the complexity level of the hot spot being remediated?

The assessment of your application will reveal the depth of entanglement and complexity within the application and will allow you to determine the overall migrateability of each of these domains within your application. Depending on these varying degrees of migrateability, you may choose different strategies for different portions of your code. Migrateability is linked directly to the business goals. If the application is changeable enough to meet those goals, then refactoring the code and migrating the application to a cloud-native platform can be sufficient. If the migrateability is low across the board, then you may have no choice but to rebuild the entire application. However, you also have the option to rearchitect and rebuild individual domains respectively if, for instance, your migrateability is high from an application code standpoint and low from a data persistence standpoint.

Figure 4 provides a decision framework for determining your strategy for the different domains within your application.

Figure 4. Complexity/Entanglement Relationship of Revise, Rearchitect, Rebuild Decisions

Complexity/Entanglement Relationship of Architecture Revise, Rearchitect, Rebuild Decisions



Source: Gartner
451153_C

Domain 1: Address the Application Architecture

This domain covers/encompasses the larger, holistic matters you need to deal with at the architecture level of an application. Your architecture may need to fundamentally change to take advantage of cloud-native platforms, deal with the challenges of running in a distributed environment and employ defensive and safety measures to handle node failures, unreliable networks, availability, latency and eventual consistency trade-offs. In effect, you are attempting to build a reliable application using an abundance of unreliable components that fail — often in ways dissimilar from application component failure on a single machine.

Cloud computing requires an architecture that will work in an environment with ephemeral systems, lack of dedicated storage and the presumption of horizontal rather than vertical scalability. Cloud-native architectures are able to balance the benefits cloud has to offer along with the areas that are different and outsourced to the provider, thus limiting your control over them.

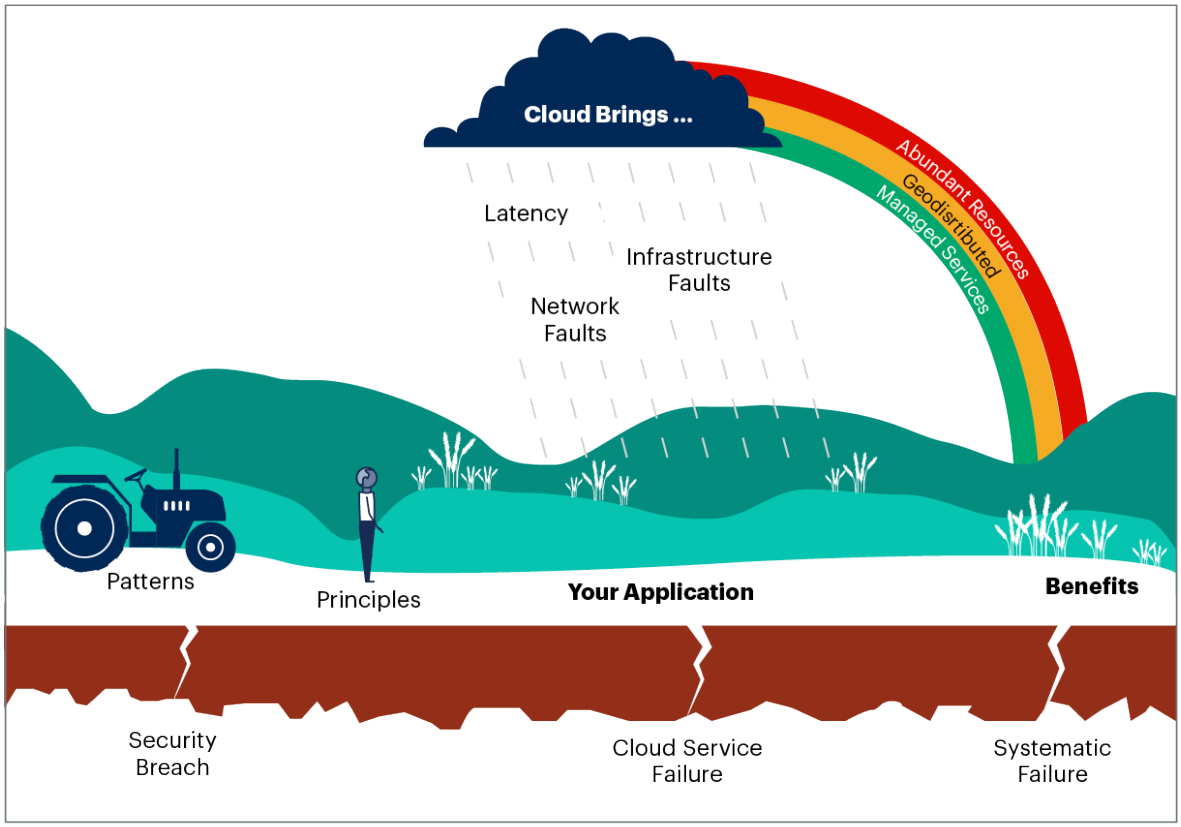
Examples of why your architecture needs to change include:

- **Elasticity:** Your organization's motivations for moving to the cloud are additional scalability and the expansion of the business to adopt new channels. This can lead to unpredictable load that varies from normal day-to-day operations.
- **Latency:** Utilizing native, managed services in your cloud provider will introduce latency into your application because your application components are now distributed across a network.
- **Failure:** Some cloud service failures, systematic failures and security breaches are entirely out of your control.

These examples represent some of the unintended consequences of adopting cloud computing. Building safety measures into your application to mitigate these risks and consequences is a priority in a cloud-native architecture. Abundant resources, geodistributed infrastructure and fully managed services introduce risks, and consequences become the foundation of the safety measures in your architecture. Figure 5 outlines the factors to consider in the dichotomy of cloud computing.

Figure 5. The Dichotomy of Cloud Computing

The Dichotomy of Cloud Computing



Source: Gartner

451153_C

Application architecture hot spots can impact development and operations teams and guide downstream decisions made by those teams, and they must be reconsidered from a cloud-native design standpoint.

These hot spots include:

- Unsupported languages and runtimes
- Direct code-to-runtime relationship
- Improper state management
- Monolithic design and deployment
- Points of failure
- Bottlenecks

Use of Unsupported Languages and Runtimes

What it is: Using languages or runtimes that your target cloud-native application platform does not support. These can vary across cloud providers. Some cloud providers are explicit about what their runtimes support. An example of this is the Google App Engine allow list for the Java 8 Runtime.[3]

Where to find it: Look for areas in your source code where teams are leveraging languages and runtimes that your cloud provider does not support. For instance, Red Hat OpenShift has no support for .NET Framework, but supports .NET Core.

How to resolve it: You need the closest support to minimize runtime/language migration effort, otherwise it's a rewrite. Potentially, this means rewriting code if you are not using a runtime and language supported by your chosen cloud-native platform. This also extends to the impact on your build, packaging and versioning processes and how those must be modified to handle a new language and runtime.

Direct Code-to-Runtime Relationship

What it is: Code that is tightly coupled to its infrastructure breaks several loose-coupling and separation-of-concerns principles that enable application migration to and around virtual infrastructure. Managed code that runs in a runtime such as Java on the Java Virtual Machine (JVM) or C# on the Common Language Runtime (CLR) avoids many code/container relationship issues. But unmanaged code, such as C or C++, that is written specifically for an OS version or C standard library, or GNU's Not UNIX (GNU) C Library version, can be more challenging to migrate.

Where to find it: Look for areas in your source code where teams are leveraging native code that is bound to a specific OS. Another example that is more subtle is if you are writing code for Java EE specifications that runs on one Java EE server but breaks on another because of assumptions about constraints like default thread pool sizes. This is particularly noticeable with event/messaging platforms where your code may assume that the messaging platform will guarantee message order. If you then change platforms, your new platform may not guarantee order.

How to resolve it: Use languages that the target cloud-native application platform supports, including constructs like containers. Avoid using unmanaged code that is tightly dependent on the underlying OS.

Improper State Management

What it is: Scaling your application in a cloud environment means that you cannot rely on complex server clustering systems to manage user, session or process state replication across application instances. Although applications all require dealing with state, cloud-native applications require you to reduce your reliance on state, externalize it from the application runtime and eliminate it from the equation when possible. As compute nodes become ephemeral, you can no longer rely on session affinity or static disk and volatile memory state bound to specific compute nodes, so you must rethink how you manage state. State begins to incur costs as well by taking up memory. This means that a given server can handle fewer transactions, which leads to an increase in the number of servers required to run your application at peak load.

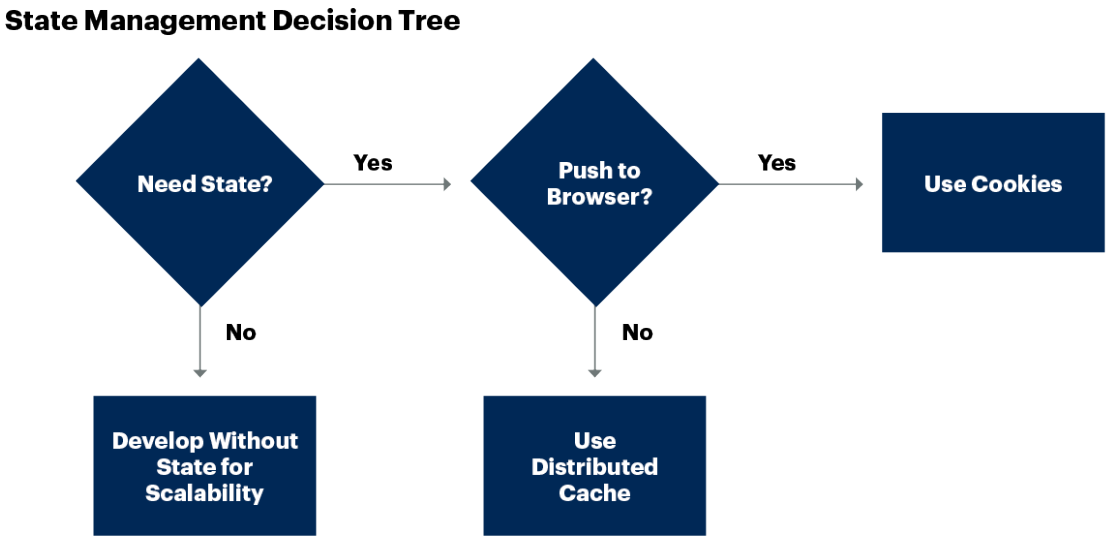
In a stateless service, if a server instance fails, requests can be routed to another server instance, processing can be restarted and, if necessary, the infrastructure can automatically create a new instance to replace it. Statelessness also means that any server instance in your resource pool can service any request. All this can and should be achieved without the consumer becoming aware. For more information on service orientation, see "MASA: How to Create an Agile Application Architecture With Apps, APIs and Services."

> State/server affinity is the enemy of scalability in a cloud-native environment.

Where to find it: Look for state both on the client and on the server. Session state is often stored in HTTP cookies and can be useful for decoupled state. However, a session cookie can be problematic. It can also be stored using a session management system that requires affinity of a user to a specific application instance or using state management systems that use clustering or replication mechanisms. Session state can also be stored in the local application cache, either in-memory or on disk. Another hot spot in your source code is direct reads and writes to disk and writing temporary files to local disk (as local disk is ephemeral in a cloud-native environment).

How to resolve it: Relentlessly try to reduce the need for session and application state in your application. When you do have sessions, keep in mind they are consuming resources, so kill them often by implementing an aggressive timeout. One option to consider for session management when the size of your payload is small enough is to store in the browser's cookies.[4] If that is not possible, then consider using a distributed object-caching mechanism. Also consider the impact of state loss. It may be that your users will accept some loss of state (e.g., if the system crashes and they have to fill the form in again). Common implementations of this include memcached and Redis or the equivalent service provided by your cloud platform.

Figure 6. State Management Decision Tree



Source: Adapted From Scalability Rules by Martin Abbott and Michael Fisher

451153_C

See “Assessing the Optimal Data Stores for Modern Architectures” for distributed caching guidance.

Monolithic Design and Deployment

What it is: The N-tier application architecture model has been around so long that modern enterprise application design has matured to a normal way of architecting, developing and deploying applications. We’re good at creating cleanly separated components with object orientation (OO) patterns and carefully layering them on top of each other with abstractions embodied in frameworks such as Spring. But at deployment time, we stuff all the components and all their dependencies into one monolithic artifact for deployment into an application server. Small changes, even if isolated to specific areas of an application, require repackaging, redeploying and regression testing the entire application (and sometimes a restart of the virtual machine instance hosting the application). This is a nightmare for delivering continuous operations.

The entanglement and complexity interwoven into a monolithic application creates headaches for teams that require more agility and reduced cycle time to deliver for the business. This monolithic approach to deployment typically will not support the agility you need to move and change those applications dynamically and in an agile fashion by leveraging the different topology options that the cloud gives you.

Keep in mind that the monolithic architecture is not always a hot spot. A well-structured monolith can still leverage a cloud platform, achieve CI/CD and scale well. Examples of this include Facebook, Etsy and Slack. However, without addressing this hot spot, you will likely inhibit the

ability to precisely scale your application by independently scaling up/down the capacity allocated to specific workloads.

Where to find it: The overall application design is not conducive to horizontal scaling. Teams that need to deploy new features or updates are impeded by the current release cycles. Applications with abnormally long build, packaging and release cycles; teams that cannot release without waiting for other teams; and applications with a small number of large application components are good candidates to look for when identifying monolithic applications.

How to resolve it: Apply application decomposition patterns including splitting, strangling and extending your existing application. See “From Fragile to Agile Software Architecture” and “How to Design Microservices for Agile Architecture” for details on refactoring a monolithic application.

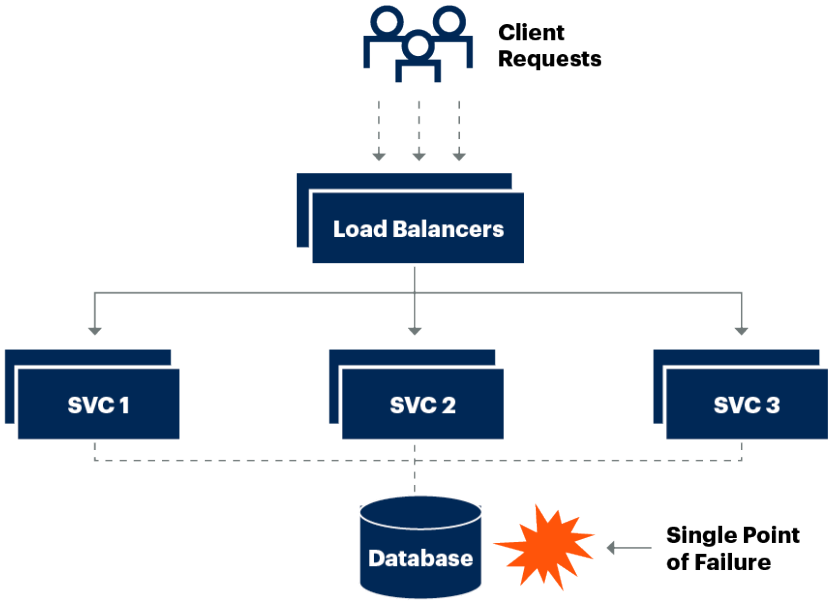
Single Points of Failure

What it is: A single point of failure (SPOF) exists when there is only one instance of an architectural component within a system and when failure of that instance will cause a failure of the whole system. The resilience and availability of an application is at risk if all traffic-servicing requests for the application are routed through any single-instance resource. If the hardware, software or network connections to that concentrator fail, the system cannot operate.

Where to find it: SPOFs can exist in your application or system at any level including front-end or back-end servers, network components, storage disks, middleware, databases, and third-party libraries and components. As an example, most traditional relational database management systems (RDBMSs) become SPOFs. This is due to the nature of the availability requirements that are the most costly and difficult to scale across multiple nodes in an active/passive or active/active mode. An SPOF crash can lead to failures that can be simple to fix, like an inbound network link, or something more severe that causes cascading failures that are difficult to repair and recover from. SPOFs appear in your architecture where you have a single application component that is not redundant (see Figure 7). See “A Guidance Framework for Architecting Highly Available Cloud-Native Applications” for details on highly available applications.

Figure 7. Database as Single Point of Failure Example

Database as Single Point of Failure Example



Source: Gartner
451153_C

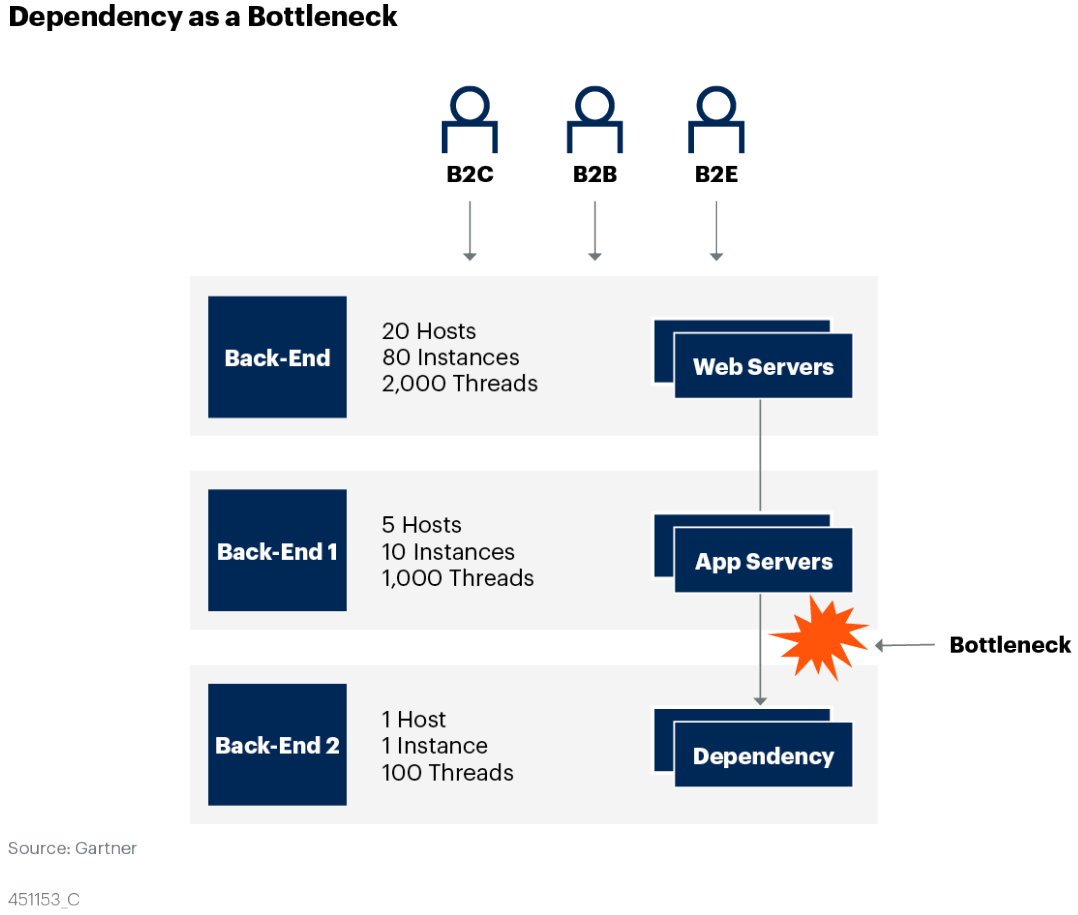
How to resolve it: Adding additional server instances through redundancy is the common way to fix an SPOF. Part of the challenge with this approach is that it introduces additional complexity. An example of this complexity is eventual consistency leading to race conditions for write operations that will violate data integrity rules. To mitigate this risk, you must fix your code to handle this condition or implement controls at the database level for locking records to prevent dirty writes.

Bottlenecks

What it is: One reason an application may not scale is that it has a bottleneck or, more likely, many bottlenecks. A bottleneck is any server, service or architectural component that requests get routed through and that has any latency. Examples of bottlenecks external to your application include databases, web services and APIs, as well as other enterprise applications integrated with the solution. All of these limit parallelism (and hence, scalability).

The ultimate bottleneck is the speed of light in your CPU core. The issue arises when people are not aware what the potential bottlenecks are. Cloud allows you to remove some bottlenecks, which may reveal other ones sitting just below them that now become the limiting factor (see Figure 8).

Figure 8. Dependency as a Bottleneck Example



Where to find it: The simplest cause of a bottleneck is code that depends on a shared or dedicated resource, such as a database. Internal bottlenecks include connection pools, explicit synchronization and resource locking, and singletons and mutable states shared across threads. All code depends on shared resources (CPU, memory, threads). The best way to find bottlenecks is to soak-test your application while monitoring all of its resources and observing which resources are getting near limits. Another way to search for bottlenecks is by using a tracing tool across your application components.

A common example of a bottleneck is a connection pool to a database. Database connections are costly, and using a connection pool to reuse these can improve your throughput and capacity. The bottleneck arises when the number of threads contending for resources is greater than the amount of resources in the connection pool. The connection pool will simply block the requesting thread indefinitely until a connection becomes available. This in turn leads to decreased throughput and a slowdown of your system.

How to resolve it: There are lots of patterns available to remove bottlenecks. The right one for a particular instance is dependent on performance and scalability requirements. One way to start solving this problem is to stop sharing resources, which is commonly known as the share-nothing model. To optimize shared resources, rightsize your connection or resource pools to handle the typical load your application experiences to ensure there are resources available when needed. Leverage timeouts and circuit breakers so that your threads do not hang indefinitely and can report back when resources are unavailable. Implement redundancy and data partitions to segment traffic out to processes or application components that can run in parallel. Implement queues and worker patterns to throttle the load on shared application resources.

Domain 2: Address the Application Life Cycle Management Process

Running your application in a cloud environment means picking up additional dependencies on backing services delivered by your cloud provider. After going through the assessment and determining that you will rearchitect your application, you will decompose it into more granular functionality to deliver APIs. Implementing agile methodologies will help reduce your cycle time for deploying new code. This means you will have more components, dependencies and application instances to develop, test, manage, monitor and troubleshoot. In almost all ways, it is easier to maintain a monolithic application over a distributed one with the exception being managing regression risk for small, isolated changes. In order to ease the burden of managing your cloud-native application, you must assess the following hot spots:

- Manual configuration updates
- Manual builds and deployments
- Inability to be containerized
- Inability to roll back releases
- Lack of testability
- Inability to be health checked

Agile, DevOps and application life cycle management are broad topics. For more details on how to be successful in these areas, see “Solution Path for Achieving Continuous Delivery With Agile and DevOps” and “Become an Agile Superhero: 8 Attributes for Success.”

Manual Configuration Updates

What it is: Configuration drift between development, testing, QA, staging, user acceptance testing (UAT) and production environments leads to a myriad of problems when it comes to the deployment and troubleshooting of your application. This problem is compounded in a cloud environment when you are deploying more granular services and increasing the number of external dependencies you have on your cloud provider’s services.

Where to find it: Your application will depend on configuration files that are scattered between different applications, different environments and different servers. These typically will not match or

follow consistent naming conventions or use consistent environment variables. Configuration values in scope here are anything that changes per deployment (e.g., development, testing and production environments; hostnames; and ports).

How to resolve it: Use externalized application configuration that is injected into the application at deploy or runtime. Ensure that these configuration files are stored and versioned in your source control management (SCM) system of choice. Use configuration templating and automation tools to deploy your software and configure it consistently for each environment. This configuration needs to be parameterized and then populated at deployment time with the values associated with that environment, or the environment needs to make these values discoverable to the application at runtime. Utilize a distributed configuration system like Apache ZooKeeper, Red Hat Etcd, Netflix Eureka or HashiCorp Consul in conjunction with one of the infrastructure-as-code tools mentioned in the next section, or the equivalent your cloud provider has to offer (e.g., AWS CloudFormation, Azure Resource Manager [ARM], Google Cloud Deployment Manager).

Building DevOps practices into the development and testing part of the life cycle and managing this well in the lower environments helps this hot spot go away as well. See “Solution Path for Infrastructure Automation” for more information on this topic.

Manual Builds and Deployments

What it is: In addition to configuration drift, there is often a mismatch of topology between the various nonproduction and production environments. Application dependencies in one environment may not exist in another, or you may be running a high availability (HA) configuration of a service in one environment and not another. These additional, complex dependencies will limit the automation and adaptability of an application’s deployment.

Moreover, multiple interdependencies that need to work together will drive complexity. In this sense, complexity means having dependencies that are hard to fulfill.

Where to find it: Your application will often have hidden dependencies like the expectation of two instances in production that are shared in test. Additional examples of complex dependencies that constrain the solution include requiring a specific version of an underlying library, a proprietary protocol or a nonstandard device driver. Application-level dependencies — for example, dependencies between components of the application — can also play a part.

How to resolve it: Utilize template-driven deployments. All provisioning, instantiation and configuration for an application should be scripted and preferably encapsulated in an automatically unpacked and deployable template. Automating deployment also means encapsulated application dependencies for deployment as a single package, or a recipe. Much of the cloud automation innovation is concentrated in technology supporting “infrastructure as code.” These tools include HashiCorp Terraform, Spinnaker, Puppet, AWS CloudFormation and Chef.

Inability to Be Containerized

What it is: Unlike a VM managed by a hypervisor, a container shares its OS kernel with the other containers running on the host rather than being encapsulated by its own copy of a guest OS.

Shared OS virtualization offers resource isolation almost on par with a VM, but is much less resource intensive and has less overhead in terms of system call traps, input/output (I/O) access, and memory and disk space usage. An application that cannot adhere to certain constraints regarding storage, networking, security and size will not be able to be containerized. Beyond just the ability to run in a container, it's the ability to run in the context of a container orchestrator like Kubernetes. The ability for your application to run in a container orchestrator is largely enabled by addressing the hot spots identified in this research. For more information on container orchestration, see "Using Kubernetes to Orchestrate Container-Based Cloud and Microservices Applications" and "Solution Comparison for Delivering Cloud-Native Applications With Public Cloud Kubernetes Services."

Where to find it: Look anywhere you are currently using VMs to move or ship code between environments and in your development and testing environments where you are building, integrating and deploying software. The reason Docker and cloud are linked is because of Docker's runtime parity characteristic. The same container that runs on a developer's laptop and in production can also run on a VM on your private cloud or on any IaaS provider's platform. This includes Amazon EC2, a Linux VM running on Microsoft Azure, Google Cloud Platform and DigitalOcean.

How to resolve it: Containerization of a single application component is not difficult, but when you have complex application dependencies, it becomes more difficult. Examine your current development environments and take steps to modify your CI/CD pipeline to use containers. Docker has published a five-step best practice[5] for getting started with modernizing your applications to take advantage of containers that bear a resemblance to the hot spots described in this framework, including:

1. Identify languages used.
2. Look at dependencies.
3. Look at other services that the application needs to start up (typically can be found in configuration files).
4. Categorize operating system dependencies.
5. Does this application have a prebuilt container that you can use to get started or does the dependency have a prebuilt container?

For more information on this topic, see "Designing and Operating DevOps Workflows to Deploy Containerized Applications With Kubernetes" and "How to Architect Continuous Delivery Pipelines for Cloud-Native Applications."

Inability to Roll Back Releases

What it is: Code that is released into production and results in a one-way proposition when something goes wrong. The release from hell that requires "all hands on deck" is something that can happen when you have a complex application with lots of dependencies of which the deployment is not automated. Even if it is automated, there needs to be an automated way or at

minimum a set of manual steps that enable the team to pull back the release from production and revert to the previous working bits (i.e., last known good).

The inability to roll back leads to additional headaches for both development and operations teams who may have to:

- Do some last-minute troubleshooting
- Quickly create a hot fix
- Spend their nights and weekends at work to coddle the application until the next release

Where to find it: Assess your build and release processes to identify episodes where your team has been unable to roll back a release. This is common when a bug is found as part of the smoke testing of the application and you cannot move forward with the release. Going through several releases of your application will eventually expose this hot spot as well.

How to resolve it: Validate and ensure that the deployment is repeatable in nonproduction environments and is scripted and tested. Utilize canary releases, blue/green deployments and feature flagging as a way to turn features on and off and test those features for different populations of users. Avoid removing elements of your data schema directly and look for opportunities to simply add to the schema either through new columns or tables.

Lack of Testability

What it is: Testing an application in a highly distributed environment poses many challenges for both developers and QA teams. Some failure modes in production are difficult to reproduce in nonproduction environments. Integration testing alone does not tease out the various failure modes that can occur in production and is focused on validating that the application dependencies work together. Furthermore, these integration-testing scenarios can become a bottleneck for releasing software. Failure injection testing is when companies purposefully inject failures into staging and production environments to test the resiliency of an application.

Where to find it: Examine current integration-testing methodologies and processes. This integration testing is typically performed in an enterprisewide integration environment that can be clunky, difficult to manage, not version-controlled and not automated. This monolithic integration-testing environment is difficult to work with and does not account for various failure modes in production.

How to resolve it: Update your processes so that, whenever you find an error, you write the test that exposes it and then fix the problem. Start by building test harnesses beyond a simple stub or mock that emulate how the application integrates with its dependencies. Include the various failure modes that can and will occur in a production environment in your test harness, don't just validate the dependencies. These test harnesses operate over the network, which is the primary hot spot for failure in your application. The test harness should emulate how a network fails by dropping packets, refusing requests, sending malformed data, causing timeouts or causing the connection to hang. Start with testing failure injection in nonproduction environments like integration testing, and then slowly introduce failures in production for the areas that require the most resiliency.

Netflix has been the leader on this front by creating a "chaos service" called the Simian Army.[6] This service has various "monkeys" that inject failures in an environment at various levels of the application, including security, compute, networking and storage, going so far as to fail an entire Availability Zone in AWS.

Utilize a layered testing approach to test at the unit, feature, integration and performance levels and independently using mocking, stubs and injection. See "Solution Path for Testing Software Applications" for advice on implementing a layered testing strategy.

Domain 3: Address the Application Code

One of the more elusive areas to refactor within a custom application is with the code itself. If you are using a common runtime like Java, .NET or NodeJS, then the likelihood that your code "just runs" within a cloud platform is high. With that said, some insidious bugs can come to the fore when attempting to fully utilize cloud resources due to the distributed nature of cloud-native applications. These anti-patterns impede your ability to run processes in parallel, scale out horizontally, increase your throughput, diagnose and troubleshoot your code, and scale in an ephemeral compute environment.

The common hot spots that need to be assessed within your application code include:

- Hardcoded values
- Memory locks
- Blocking calls
- Siloed logging
- Nonrestartable application components

Cyclomatic Complexity

There are various ways to measure the complexity of your application code, but a common metric used in software development is cyclomatic complexity. The metric was developed by Thomas J. McCabe, Sr. in 1976.[7] The goal is to give the developer an understandable complexity calculation. The metric is computed and normalized in your code using a control flow graph. This can be used as a percentage to measure the complexity of the elements in your code including modules, functions, classes or methods.

Hardcoded Values

What it is: One common area of technical debt that is often overlooked or put on the back burner in applications is hardcoded values. Values specific to your application environment are commonly found deep within your source code. These values can include hardcoded Internet Protocol (IP)

addresses, port numbers, hostnames, file system locations, ID numbers, keys, usernames and passwords. An example of the havoc this can wreak on your application is when a hardcoded hostname or IP address in the source code creates inadvertent hardware affinity and impedes the code's ability to be hosted on any hardware platform.

Where to find it: Primarily, this will be found in your source code or scripts. It is quite common to see source code or scripts contain hardcoded names of local resources and hardcoded IP addresses of specific environments like test, quality assurance (QA), staging or UAT.

Another common place this is found is in RDBMS. For example, Oracle SOA Suite integration flows have the DNS addresses of the cluster nodes embedded in their stored state in the DB — if you migrate to the cloud, you need those DNS addresses to resolve because you can't change them.

How to resolve it: Externalize these values into configuration files or environment variables that can be referenced across different nodes in your application and possibly even across networks. One cloud design goal is to completely separate variable concerns (such as IP addresses, DNS names or file system volume names) from the other common concerns — the application configuration and code. That way, the code can be hosted on any hardware platform. To manage the sprawl of configuration files, utilize a distributed configuration system like Apache ZooKeeper, Red Hat Etcd, Netflix Eureka, HashiCorp Consul or the equivalent in your cloud platform.

Memory Locks

What it is: A common technique to protect the integrity of transaction data is to apply a memory-based lock in the code, thus wrapping the transaction to synchronize threads and dictate the transaction order. Using this locking mechanism in your code works well when synchronizing threads in an application component running on a single node, but it has unintended consequences in a multinode environment. For instance, when multiple clients are making requests and the code is using a memory-based lock, only one client at a time can execute the code. When this same code is running across multiple machines, there is no way to coordinate that lock across application instances, and the transaction will fail. The impact is that this style of code can no longer take advantage of parallel processing (see Figure 9).

Where to find it: Look for areas in your code where you are using memory-based locks. Examples include lock in Python, lock in C# and synchronized in Java.

Figure 9. Memory Lock Code Example

Memory Lock Code Example

```
lock.acquire() # will block if lock is already held
… access shared resource
lock.release()
```

Source: Gartner

451153_C

How to resolve it: Clustering technologies that utilize cross-server, shared memory and IP multicast can solve this challenge, but this is generally not an option in a dynamically scalable cloud environment. The better option is to treat your database (or other persistence mechanism) as the single source of truth for your data. Otherwise, your business requirement can accept the inconsistency or adopt an eventual consistency model.

Blocking Calls

What it is: Applications that use blocking calls turn into bottlenecks and lead to the classic, awful user experience of the "hang" or gradual slowing down of an application. There are plenty of sources of blocking calls including the use of third-party libraries and software development kits (SDKs), resource pools, connection pools and global cache objects. This hot spot can be difficult to test for in nonproduction environments.

Where to find it: This can occur wherever your code utilizes an external resource or dependency and where your code is utilizing multithreading and attempting to synchronize shared resources or data. In the following example, written in Python, the following code waits for the query to be completed (see Figure 10).

Figure 10. Blocking Code Example

Blocking Code Example

```
results = database.some_query()
for value in results
   # do something
end
# code not executed until after loop
```

Source: Gartner

451153_C

The code in Figure 11 does not wait for the query to be completed and continues to process using an iterator.

Figure 11. Nonblocking Code Example

Nonblocking Code Example

```
database.some_query do [results]
   for value in results
      # do something
   end
# this code continues running while the database
performs the query
```

Source: Gartner

451153_C

How to resolve it: Take advantage of an event-driven or reactive programming approach by putting whatever code that needs to execute in response to a blocking event in a code block to be executed later. Also:

- Utilize proven concurrency patterns
- Thoroughly test any code that could be a blocking call
- Implement and use timeouts as a defensive programming mechanism whenever accessing resource and connection pools

Most languages have a nonblocking library that you can utilize including Netty, VertX.io, Reactor, Twisted for Python, NIO for Java and Interlace for C#. Node.js is inherently nonblocking.

Siloed Logging

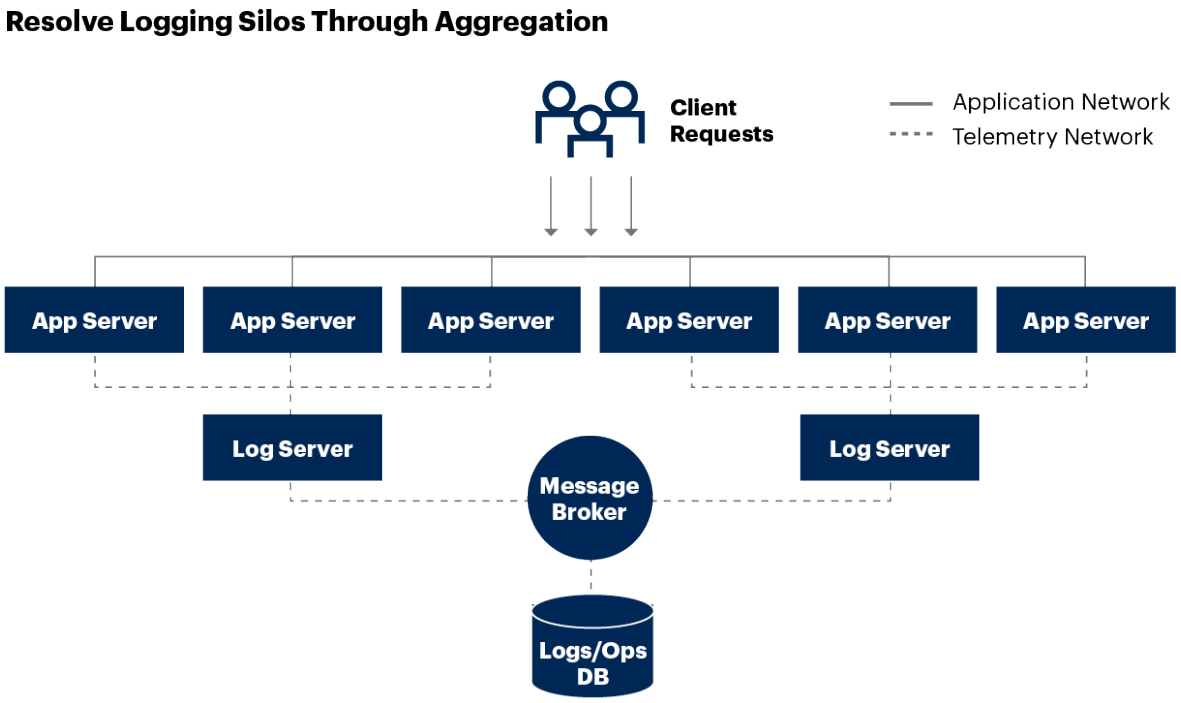
What it is: Log files are still the primary means to retrieve information about the health of an application and other metrics. Log files provide you with critical information for debugging and troubleshooting an application in production. For logs to be valuable, they must be persisted, and writing these to local disk in a cloud environment can lead to data loss. In the same vein, log files in a distributed environment are not valuable to anyone unless they can be pulled together and viewed as a whole. Log files can be a valuable tool to diagnose and prevent issues in your application if utilized correctly.

Where to find it: Logging mechanisms are heavily integrated directly within your source code. The first step is to see if your code is even logging and if not, turn it on and instrument your code so that it is producing log files. Next, look for hot spots in your source code tied to exception handling. This is generally where developers use their logging framework. Finally, make sure you understand where the log files are being written to (e.g., local disk or a logging server).

How to resolve it: Avoid logging directly to the local drive within the application instance due to the ephemeral disk. If you lose the instance, you will also lose your log data.

Ensure that logs are being aggregated, either through a monitoring tool scraping the logs off of the servers or utilizing a logging server outside of the application's network to stage the log files to a central location for analysis. This can help with implementation of distributed tracing, which is key for troubleshooting transactions that span disparate systems. Additionally, use a correlation ID in your log messages that lets you trace a thread of execution between multiple distributed services (see Figure 12).

Figure 12. Resolve Logging Silos Through Aggregation



Nonrestartable Application Components

What it is: Your application cannot rely on cloud infrastructure always being available and reliable. There are instances when your hosts will reboot due to operations maintenance or failure of the underlying node. For instance, AWS has performed patches in the past that have required EC2 hosts to be rebooted.[8] These reboots mean that your application needs to not only fail gracefully, but also to *restart* gracefully. This also occurs in an application platform such as AWS Elastic Beanstalk where you have different modes to update your environments including blue/green and immutable updates that will require your application to be restartable in a graceful manner.

Where to find it: Look at the startup sequence of your application, the dependencies required for startup, and the connections both to and from the application.

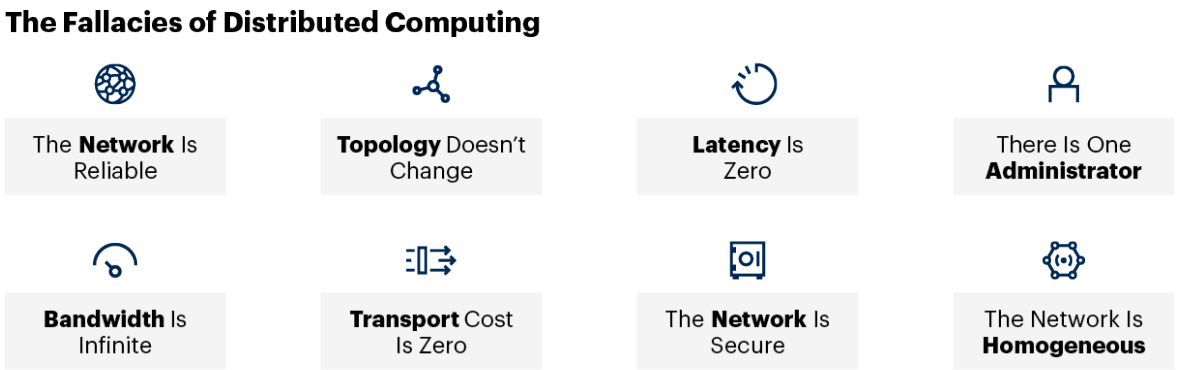
How to resolve it: Build a clean startup sequence into the application to ensure that components are started in the correct order. Part of the sequence will include a step to check for state that has been unprocessed (e.g., messages in a message queue) and complete those transactions. Ensure these steps are sequenced in a manner that the application does not begin accepting work until all are complete. For instance, your application should not begin accepting socket connections until certain conditions are met in order to begin the processing.

Domain 4: Address the Integration Points

Application code simply does not run in isolation, and this has been true for decades. The days of stand-alone systems are gone. For applications to be rich and relevant, they need to be connected to other applications, businesses, data sources, services, third parties, feeds, back-end systems and legacy systems. Running your application in a cloud-native environment only compounds this issue with additional integration points and the countless layers of abstraction between your application and the hardware, virtual resources and backing services it uses.

These integrations are the most common points of failure in an application, and they can severely degrade the functionality of your application, leading to cascading failures when you are not looking. These common issues that arise are summed up by the fallacies of distributed computing in Figure 13.

Figure 13. The Fallacies of Distributed Computing



Source: Gartner

451153_C

Integration points introduce latency, bottlenecks, complexity and factors out of your control as an application developer that you need to be aware of and defend against. As a developer, you need a clear understanding of where and how your application interacts with external components. It is important to implement the appropriate safety measures to ensure that your application does not fall victim to the fallacies of distributed computing.[9] These integration points are interacted through an unreliable network and include the following hot spots to look for:

- Chatty interactions

- Network calls
- Complex call graph
- Third-party libraries and APIs

Chatty Interactions

What it is: Designing services from finely grained objects follows the request/response model that is familiar for most developers. Chatty interfaces require excessive round trips to perform a single logical operation. One classic chatty anti-pattern is a service invocation populating a data object that requires the caller to assign multiple property values before executing a method. In a widely distributed system, however, this model is problematic because interactions take place across process and network boundaries with nontrivial latency and failure modes. This is especially true when hosting in the cloud because service interactions may be happening across the public internet. Chatty interactions are a direct consequence of the finer-grained component design pattern, so developers must perform a balancing act.

Where to find it: The client or browser is often a culprit for chatty interactions. webpages often consist of a lot of objects, including Cascading Style Sheets (CSS), JavaScript and HTML, that can increase the number of requests to the server. There are techniques that can be utilized like AJAX that aim to provide a more reactive user experience, but these can often initiate an overwhelming number of requests to the server from a single webpage. DNS look-ups can be another culprit on the client when making requests for various page resources. On the server side, the idea of location transparency in using remote procedure calls (RPCs) to remove the network complexity between service calls can lead to chatty interactions. Many fine-grained services can also lead to increased volume of communication between services.

How to resolve it: One technique is to build single-page apps where you move the UI code to the client and rely on asynchronous API interactions to deliver data to the client-side app. Another is to reduce your reliance on AJAX on the client if possible, reduce the number of objects on your webpage and reduce the number DNS look-ups your client-side code makes. Ensure you are rightsizing your services to be highly cohesive so that they rely less on external systems for data.

According to the Hypertext Transfer Protocol (HTTP) 1.1 Request for Comments (RFC) 2616, a browser should maintain a maximum of two persistent connections to a server or proxy.[10] On the server side, utilize chunkier interactions through API calls.

Network Calls

What it is: All interactions with external application components incur latency. In a cloud-native environment, you are trading off latency guarantees (in your traditional application environment) for increased efficiencies, resource utilization and potential cost savings. Latency cannot be guaranteed in this environment because resources are allocated dynamically, not statically. The delays incurred during resource allocation and scheduling provide you with cost savings but sacrifice the reduced latency inherent to statically allocated resources that are close to the machine. Applications with absolute-latency guarantees are not suitable candidates for public cloud deployment.

Where to find it: Each integration point within your application has the potential to introduce latency. Look for networked calls to external resources outside of your application to identify candidates for latency. Look for any application code utilizing RPC, APIs, web services or other socket protocols calling out to other systems. This is closely related to the bottleneck hot spot described earlier in the framework.

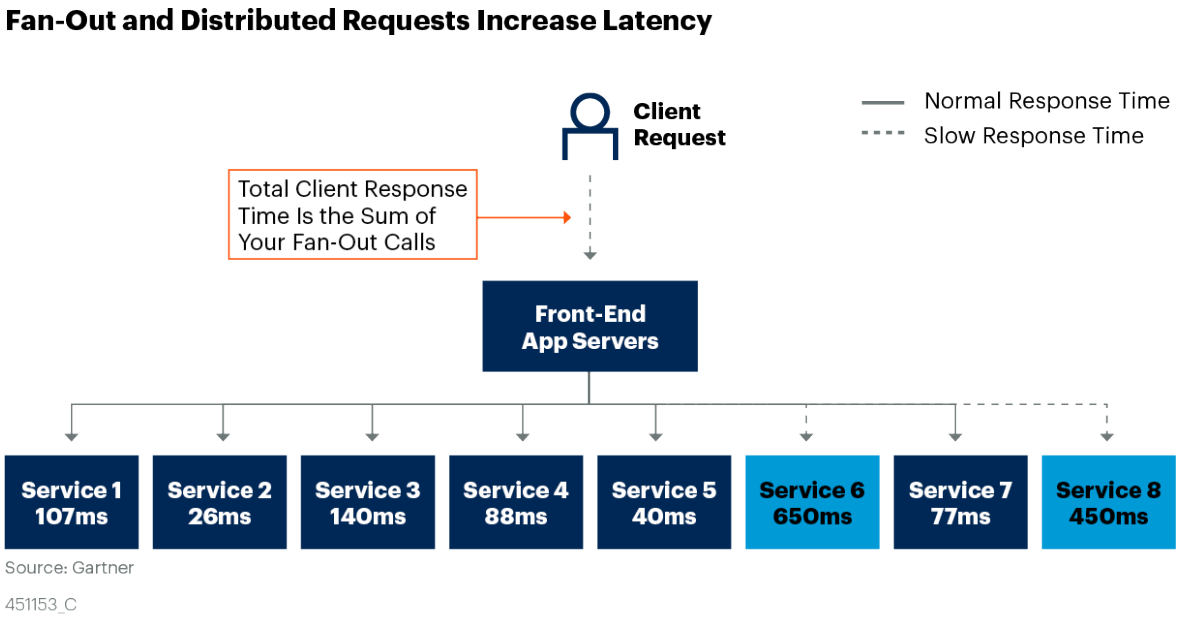
How to resolve it: This cannot be fully resolved in a cloud environment, so, where possible, minimize networked calls outside of your application without sacrificing other architectural constraints in your application (e.g., microservices). When this cannot be avoided, prefer chunky over chatty interactions. In addition, utilize event-driven models and asynchronous communication mechanisms between your application and others including a message queue, topics and worker patterns. There are instances where your application may not be a good candidate for a cloud-native environment. For instance, high-frequency algorithmic trading applications that require access to real-time data within a fixed time window would be better deployed where they are colocated with the data source (e.g., within stock exchanges).

Complex Call Graph

What it is: When a service delivers a message to a broad number of consumers that may, in turn, deliver that message to more consumers and so on, you have a network call graph that resembles a fan. Applications often implement these operations through direct messages to the recipients, which can lead to increased latency of transactions and tight coupling between your application and the consumers (see Figure 14).

Where to find it: Check architectural components that have a significant one-to-many relationship with a large quantity of recipients or consumers. Look for code that explicitly is calling a large number of recipients to notify them of a change that occurred within the application. For example, this is a fundamental issue Twitter deals with.[11]

Figure 14. Fan-Out and Distributed Requests Increase Latency



How to resolve it: Update your application code to work in an event-driven model for these types of one-to-many operations. This allows consumers to each "listen in" to the same broadcast of messages without affecting each other. In extreme scale cases, you may have to implement a cache based on specific criteria (e.g., partitions) of the data. Most cloud service providers (CSPs) and frameworks have a feature for doing this including: Azure Service Bus, Amazon Simple Notification Service (SNS), topic subscriptions in Java Message Service (JMS) and exchange bindings in Advanced Message Queuing Protocol (AMQP).

Third-Party Libraries and APIs

What it is: Third-party libraries can often be black boxes that you do not have control over. This can be an SDK, command line interface (CLI), API or other integration point that a vendor or commercial off-the-shelf (COTS) application has provided to you. The third-party library or API is not something you own, so you are not responsible for the testing, hardening, resiliency or performance of the library/API. This lack of control can lead to unpredictable behavior and increased latency in your application.

Where to find it: Look for third-party library and API references in your source code and configuration.

How to resolve it: Implement the proper safety measures to defend against errors that may occur from utilizing the API or library. This includes proper exception handling and logging. There are also times when you need to implement a timeout and retry mechanism when calls to the API/library fail.

Domain 5: Address Data Persistence

A typical legacy application relies on traditional RDBMS systems and relies on atomicity, consistency, isolation and durability (ACID) transactions for strong consistency of data. These traditional systems rely on the underlying infrastructure to be rock-solid and reliable and are not built for environments where the infrastructure is fluid and prone to failure. These systems are also built to scale up, not out, which impedes your ability to utilize the abundance of cloud resources and elastically to scale your application.

The need for multiple data storage options reflects contemporary application design, which requires polyglot persistence. This concept implies that developers should choose the data storage that best suits the data and their programming approach, rather than forcing the data to fit into a traditional SQL model. Using RDBMS for every storage option can lead to inflexibility in your design and significant costs in scaling the database. In the same vein, introducing polyglot persistence increases data management complexity, introduces new modes of data consistency and issues with aggregated reporting and analytics. Addressing the hot spots in your database architecture needs to be an iterative process. For details on this, see "Implement Agile Database Development to Achieve Continuous Delivery."

The database can be a common bottleneck for applications looking to scale for either reads or writes, and other modes of data persistence need to be considered in a distributed environment. Common hot spots to look for and avoid in the data persistence domain include:

- Use of multiphase commits and reliance on transaction isolation models
- Monolithic databases
- Use of select * statements and wildcards in SQL statements
- Developer-crafted SQL statements

Use of Multiphase Commits

What it is: ACID transactions are those that are atomic, consistent, isolated and durable. RDBMSs offer ACID transactions within the scope of a single database instance. ACID transactions ensure that, for example, records in two different tables within a database can be updated as a single unit of work. Multiphase commits, on the other hand, ensure that transactions are committed or rolled back across multiple database servers. Using multiphase commits, it is possible to update records that reside in two different database servers. However, applications that use multiphase commits may not be suitable for cloud platforms because of variable network latency, variable service provider availability and unguaranteed performance service levels. Architects should modify such applications to move transactional integrity semantics from protocol to application semantics (such as implementing idempotent operations). Naturally, this anti-pattern is a generalization — some scenarios exist where ACID transactions must be used. Examples of situations where developers cannot avoid global locking and atomic operations include creating globally unique keys (such as usernames) or transactions that are required to take a precise snapshot across possibly partitioned data (e.g., billing).

Where to find it: Look for distributed transaction managers at the application level — like Microsoft Distributed Transaction Coordinator (DTC)[12] or Java Transaction (XA protocol) — that interact either directly with databases or through an object-relational mapper (ORM) tool like Hibernate or LINQ.

How to resolve it: This scenario should be avoided by changing the application design. For applications that use RDBMSs, asynchronous database replication is often used effectively in cloud environments. For applications that use NoSQL databases, eventual consistency is another approach, and it is sometimes favored by cloud designers to govern visibility of updates across members of a distributed environment, such as a cloud. A common pattern in the microservices world is the use of SAGA and CQRS patterns. Applications must assume that reads of just-written data may be out of date. This condition requires a very different set of application assumptions than those available in an environment with ACID transactions.

Monolithic Database

What it is: A monolithic database refers to a database that has been created using a canonical data model that attempts to model all of the entities and processes for a given application or across applications at an organizational or company level. Such large databases attempt to solve every problem under the sun. This tight coupling of data schemas to each other can lead to resource contention, bottlenecks, single points of failure and the inability to scale the data under load in the database. Large databases are also difficult to maintain and can be slower. A larger database has challenges when attempting to distribute the data across multiple disks and servers. Common in monolithic databases are the use of stored procedures and triggers as a mean to deal with business and application logic. This is a clear anti-pattern that should be weeded out as well.

Where to find it: Look at large databases that back monolithic applications. Look for points of contention within your log files and monitoring systems to identify hot spots where monolithic databases are present. Another option for hunting these down is to work with database administrators (DBAs) for their insights as to which DBs smell monolithic.

How to resolve it: Utilize data partitioning by dividing a large dataset into smaller data schemas and ultimately smaller data stores. This can be done in a number of ways, including by index or by data domain. Partitioning by index is typically done horizontally (separating rows by key or data range — such as “northern region” and “southern region”), but can be done vertically in some systems (placing different columns on separate partitions). Partitioning a database horizontally is known as “sharding.” Shards can be distributed so that the system maintains performance as load increases, and extra nodes can be added to maintain scalability. Each node works on a subset of the data in parallel. For more detail on migrating databases to the cloud, see “The Paths Forward for Your On-Premises Databases.”

Use of Select * and Wildcards in SQL Statements

What it is: Utilizing wildcards and select * SQL statements means you are bringing all of the columns back from the table without a filter. This can lead to mass memory consumption and table scans on your database, which can hog resources from other processes and transactions. When you utilize wildcards in your queries, there is an opportunity to break your code and other processes

due to an underlying schema change and bring back unnecessary data. Performing these types of queries in the cloud exacerbates these issues because you are using database PaaS (dbPaaS), and resource usage can be charged based on usage and memory consumption, thus leading to increased costs and bottlenecks. You also don't have as much control over the tuning of the dbPaaS that you may have on-premises.

Where to find it: Look for hot spots in your application code including hardcoded SQL statements and ORM tools as well as in the database in saved queries, views, stored procedures and triggers.

How to resolve it: Be explicit about what you are selecting in your SQL statements. Your SQL statements should explicitly declare the columns that are important for your purpose. Avoid using wildcards in any of your SQL queries, especially against production databases.

Non-DBA-Crafted SQL Statements

What it is: SQL statements that have been created by a developer or a user other than the DBA directly. These are often executed against the database and have not been optimized by a DBA. These SQL statements often use nonindexed columns, join many tables together and otherwise misuse SQL to get a job done. The challenge with these SQL statements is that they are unpredictable and not optimized for the database you are accessing.

Where to find it: These statements are often issued directly to the database through code. The code often embeds a hardcoded SQL statement and calls the database directly. Another area to look for is using a data access layer or ORM tool for accessing the data. These frameworks often allow the developer to pass in SQL strings directly.

How to resolve it: Avoid using handcrafted SQL in your code or in stored procedures, triggers or views that have not been created, tested and optimized by a DBA. Ensure that you are working closely with your DBA team to determine the best query for what you are trying to accomplish, and have them write and optimize those SQL statements.

Follow-Up

Determine Strategies for Cloud Migration

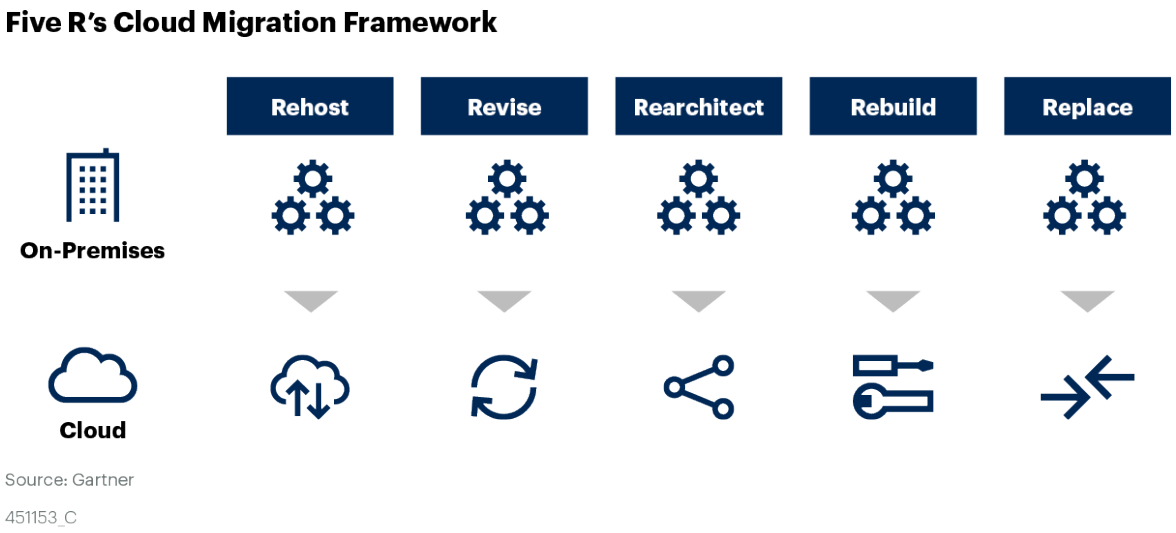
You will need to determine the most appropriate strategy for migrating applications to a cloud-native platform on an application-by-application basis:

- **Rehost:** Move the application to new infrastructure. This could include moving from bare-metal or dedicated VM infrastructure in a traditional infrastructure and operations (I&O) model to operating in IaaS to take advantage of cloud computing.
- **Revise:** Perhaps you are past rehosting and need to reconfigure your app for cloud services. For instance, you may choose to leverage a dbPaaS of MySQL over hosting it yourself on IaaS or introducing an external caching mechanism for session state.

- **Rearchitect:** Support modernization in your codebase and major architectural components. This requires major revisions to take advantage of cloud characteristics and your CSP's feature set.
- **Rebuild:** Rearchitect is a prerequisite to rebuild and entails moving to your CSP's application platform. For instance, you might move your application services into AWS Elastic Beanstalk and leverage a lightweight web container framework like Apache Tomcat or Eclipse Jetty.
- **Replace:** Discard your existing app in favor of a SaaS application. For instance, you are decommissioning your custom CRM in favor of Salesforce CRM or Microsoft Dynamics 365.

Most organizations are seeking to gain the greatest benefits with the least amount of engineering effort. Per Figure 15, the perception of most organizations is that they should start with either a rehost or a replace because these require less engineering effort. This is because making the application cloud-ready requires revise, rearchitect or rebuild engineering efforts. This framework is aimed at technical professionals interested in determining the areas of their applications that need to be reengineered through a revise, rearchitect or rebuild strategy.

Figure 15. Five R's Cloud Migration Framework



Gartner recommends that you revisit your cloud application migration after using this framework to reassess your strategy at a more granular level. See "Decision Point for Choosing a Cloud Migration Strategy for Applications" for more details on completing this follow-up step.

For specific patterns and practices to employ for resolving the issues found in this assessment, reference "How to Architect and Design Cloud-Native Applications."

The other important follow-up is to monitor the business goals. If they change, then more work may be required to help the cloud platform meet them.

A Note on Infrastructure, Operations, Security and Identity

Details on the areas that need to change from an I&O aspect are out of scope for this research note. Because many applications are tightly bound to the underlying infrastructure, hardware, networks and storage, this can be a large part of your rearchitecture and migration efforts. Most IT shops are conditioned to manage monolithic applications and find it is easier to maintain a monolithic application than a distributed one. Common hot spots for infrastructure, operations, security and identity include:

- Use of hardware load-balancers
- Too many firewalls and lack of firewall support
- Binding to specific networking architecture and the inability to operate with dynamic IPs
- Inability to monitor distributed environment
- Inability to troubleshoot distributed transactions
- Implementing security or load-balancing logic in your service components other than the platform

Several research notes can assist you in addressing these aspects of cloud readiness:

- "How to Prepare Your Network for Private and Hybrid Cloud Computing"
- "IAM Is Vital for Successful Application Migration to IaaS"
- "Solution Path for Security in the Public Cloud"
- "Foundations of a Production-Grade Public Cloud IaaS and PaaS Architecture"
- "Comparing Network Architectures for Interconnecting Data Centers and Clouds"
- "Adopting Cloud-Delivered Identity Governance and Administration"

Risks and Pitfalls

The engineering and organizational effort required to take advantage of cloud computing in your application portfolio is significant. Utilizing cloud computing is not a panacea for modernizing your applications. Your stakeholders' goals need to be strongly aligned with strategy and execution. The trade-offs for executing on a large cloud computing modernization effort are real and should be fully

understood and agreed on upfront. This will be a resource-intensive and lengthy program to execute. Following this guidance to the letter will impact your resources, time and budget.

Risk: Not Investing in Cloud-Native Operations

The failure to invest in how your application will function and operate in production leads to increased headaches down the road. Going into "Day 2" operations of your application with manual build and deployment scripts, lack of test automation, lack of instrumentation and lack of monitoring means you and your team will be working late nights and weekends. Failure to automate processes and the introduction of human error all equate to outages and potential loss of revenue for your business. Operating your application makes up the majority cost of your total cost of ownership (TCO), so you should avoid the perils of not investing in this early on.

How to Mitigate

Invest in organizational and cultural change management to follow DevOps principles. Align your operations team members with developers, and have them work together on developing new availability and operational policies. Work to adopt the "infrastructure as code" principles by building in developer workflows to environment configuration, builds and scripts. Automate the deployments of your application through the use of the CI/CD toolchain. Introduce failure injection testing early on into the application life cycle so that you are prepared for the worst-case scenarios. Google's Site Reliability Engineering (SRE) is a well-known example of operating applications at scale in a cloud environment.

Risk: Introducing Additional Complexity Through Distribution

Part of your cloud-native rearchitecture will inevitably include further distribution of your application by decomposing it into smaller services. Combined with introducing new dependencies in your application, you are now working in a highly distributed environment. You now have more application components to manage, which leads to increased complexity in deployment and operations.

How to Mitigate

Leverage the tools available to you in your cloud environment to manage this distribution:

- Tagging
- Templated configuration
- Version control
- Deployment automation
- Test automation
- Monitoring tools

Use an aPaaS or CaaS with developer workflows or opinionated frameworks as a way to "train" your team on cloud-native development. This will become a "forcing function" to employ cloud-native principles. See "Implementing a Tagging Strategy for Cloud IaaS and PaaS."

Pitfall: Positioning Cloud-Native Modernization as a "Project"

It is common in IT shops and project management offices (PMOs) to treat everything as a project or a program, but there is too much organization and cultural change involved in cloud-native development to do this. The challenge is that resources and budget are typically constrained and scoped at a project level. This constraint means that execution and lessons are only handled by a select few within the organization. This will leave the majority of operations and development team members "out in the cold."

How to Avoid

There must be an overall shift in mindset from the leadership level down to understand when and where cloud-native principles apply in the application portfolio. Leadership must empower individuals to continuously learn and improve on these lessons in the context of your organization. Make cloud-native modernization an inherent part of your application maintenance cycles. Heavy investment in organizational change processes is needed because cloud innovation outpaces existing organizational processes and culture and breaks existing processes.

Pitfall: Applying This Guidance to Every Application and Remediating Every Hot Spot

It is tempting to catch Maslow's hammer syndrome[13] and treat every application as if it needs to be cloud-native. This is simply not the case because a large proportion of your application portfolio does not require the scale and elasticity offered by cloud computing. Falling into this trap will lead your teams and resources down rabbit holes and become a black hole of cost for your organization. This typically happens when you do not perform the prework outlined in this guidance and are misaligned between strategy and execution.

How to Avoid

Fully understand your application portfolio on an application-by-application basis. Ask questions such as:

- "Will the user base of this application grow exponentially in the near term?"
- "Do I need to expand this application out to global markets?"
- "Are there innovation scenarios around Internet of Things (IoT) and machine learning that my application needs to take advantage of?"

When you can answer these questions and others related to cloud adoption goals with a definitive "yes," then you may have a candidate for a cloud-native application.

Related Guidance

This document guides architects through the process of assessing your application for its readiness to migrate to a cloud-native application architecture. As described in the Prework section, you must address your strategy, cloud goals, priorities and information sources before beginning to apply the steps described in the main body.

The document does not provide specifics on assessing your application for specific cloud platforms. The following documents provide details on AWS, Azure and GCP:

- "In-Depth Assessment of Microsoft Azure Application PaaS"
- "In-Depth Assessment of Amazon Web Services as an Application PaaS"
- "Solution Scorecard for Google Cloud Platform as a Public Cloud Application PaaS"

The following provides additional guidance on cloud-native architecture:

- "How to Architect and Design Cloud-Native Applications"
- "A Guidance Framework for Architecting Highly Available Cloud-Native Applications"
- "Comparing Leading Cloud-Native Application Platforms: Pivotal Cloud Foundry and Red Hat OpenShift"
- "How to Build Cloud-Native Applications Using Serverless PaaS"
- "How to Succeed With Microservices Architecture Using DevOps Practices"
- "How to Design Microservices for Agile Architecture"
- "Using Kubernetes to Orchestrate Container-Based Cloud and Microservices Applications"

Gartner Recommended Reading

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

"Comparing Leading Cloud-Native Application Platforms: Pivotal Cloud Foundry and Red Hat OpenShift"

"Using Kubernetes to Orchestrate Container-Based Cloud and Microservices Applications"

"How to Architect and Design Cloud-Native Applications"

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

Evidence

[1] B. Moseley and P. Marks. "Out of the Tar Pit." Software Practice Advancement 2006.

[2] B. Foote and J. Yoder. "Big Ball of Mud." Department of Computer Science, University of Illinois at Urbana-Champaign.

[3] "JRE Class White List for Java 7 Runtime," Google Cloud Platform.

[4] "HTTP State Management Mechanism," IETF Tools.

[5] "5 Steps to Take Before Moving Your Applications Into Docker," Nick Janetakis.

[6] "FIT: Failure Injection Testing," Netflix.

[7] "A Complexity Measure," IEEE Xplore.

[8] "EC2 Maintenance Update," Amazon Web Services.

[9] "Fallacies of Distributed Computing," Arnon Rotem-Gal-Oz.

[10] "Part of Hypertext Transfer Protocol — HTTP/1.1," World Wide Web Consortium (W3C).

[11] "Timelines at Scale," InfoQ.

[12] "Two-Phase Commit," Microsoft.

[13] "Maslow's Hammer," Psychology Today.

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