Solution Path for Applying Microservices Architecture Principles

Published 17 December 2021 - ID G00757391 - 36 min read

By Kevin Matheny

Initiatives: Application Architecture and Platforms for Technical Professionals

Microservices architecture is popular, but the principles behind it are poorly understood. This research guides application technical professionals in when and how to apply the principles of microservices to deliver advanced agility and flexibility.

Overview

Key Findings

- Teams that naively adopt microservices without understanding the inherent trade-offs introduce needless complexity that results in brittle, hard-to-manage applications.
- Mastery of agile development practices and robust DevOps capabilities are prerequisites for the successful adoption and implementation of microservices architecture (MSA).
- MSA is not all-or-nothing. However, applying MSA principles creates a trade-off between the agility of an architecture and its complexity and cost to operate. The more complex and flexible your architecture is, the more expensive it will be to operate.
- MSA is best suited to applications that need to change rapidly to meet business needs. Systems that are stable, predictable and less complex are poor fits for MSA.

Recommendations

As an application technical professional responsible for application architecture and platforms, you should:

- Choose target applications and systems for MSA carefully, ensuring that your stakeholders and users are ready for the changes that will be required to gain value from an investment in MSA.

- Master agile development and DevOps practices before applying MSA principles. Your organization will benefit from this regardless of whether you ultimately adopt microservices.
- Evolve your architecture incrementally with short-term business value in mind, using agile methods to insulate against failure and avoid a “big bang” MSA implementation.
- Support teams delivering microservices with a self-service microservices platform managed by a Platform Ops team.

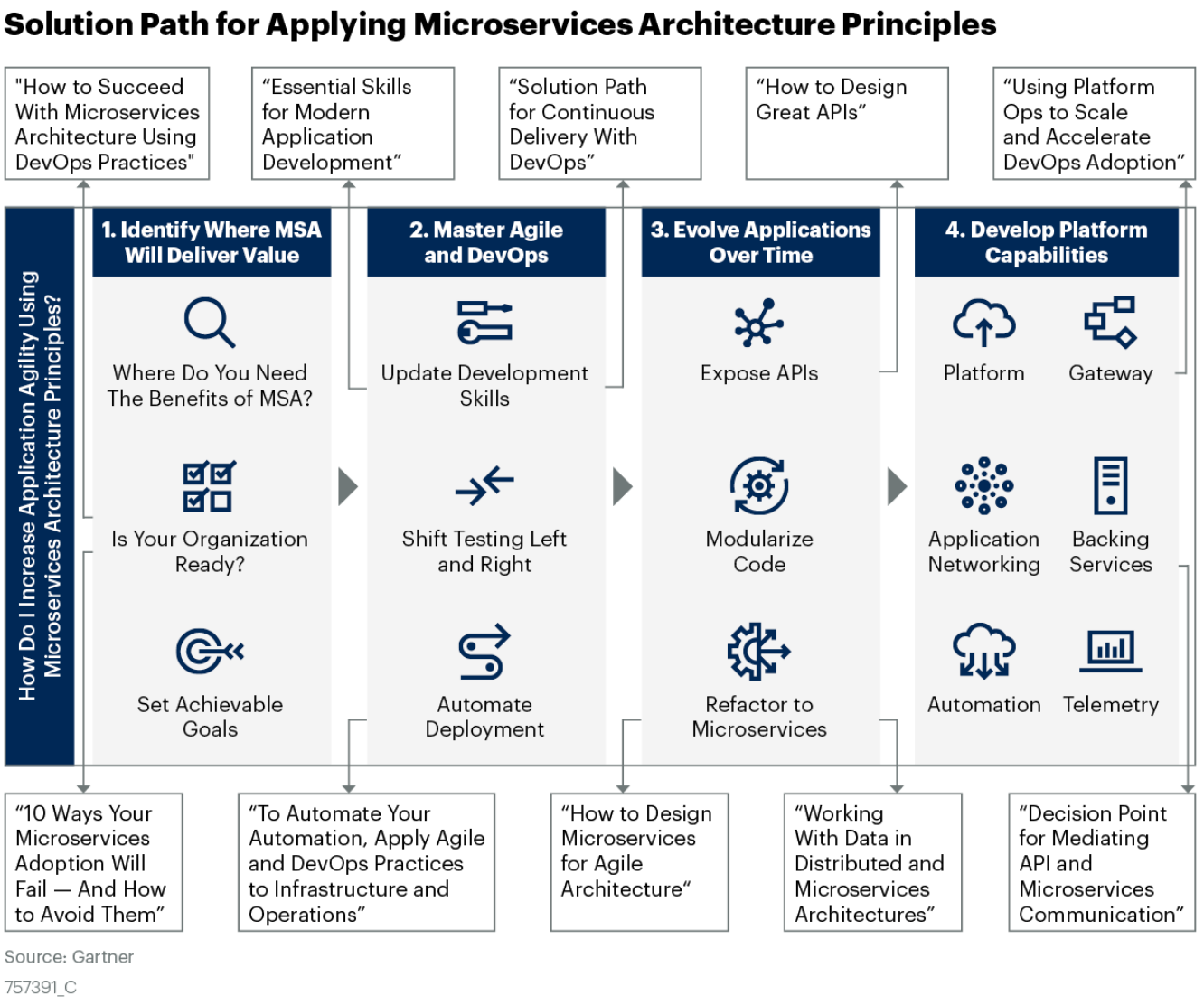
Problem Statement

How do I use microservices architecture principles to deliver applications that need increased agility and deployment flexibility?

Solution Path Diagram

Figure 1 shows the Solution Path for applying microservices architecture principles.

Figure 1. Solution Path for Applying Microservices Architecture Principles



Source: Gartner
757391_C

Solution Path

To successfully apply MSA principles, application technical professionals must target the right applications, master agile and DevOps and use an iterative approach to evolving application architecture and developing platform capabilities. To inform this investment in agility, flexibility and scalability, they must also understand what microservices and MSA are:

- **Microservices** are application components that are tightly scoped, highly cohesive, strongly encapsulated, loosely coupled, independently deployable and independently scalable. They are deployed as services behind open, standard network-accessible interfaces with well-defined API contracts. These components own the data they present, and only present data that they own.
- **Microservices architecture (MSA)** is a design paradigm that applies SOA and domain-driven design principles to distributed applications. MSA has three core objectives: development agility, deployment flexibility and precise scalability.

> **MSA is an application architecture, not an enterprise architecture. Reuse is not an objective of MSA, nor is cost savings.**

Adopting MSA requires more than changes to application architecture. It requires significant changes to an organization's practices across development, operations and data management, as well as investments in complex infrastructure and tooling. Successful adoption of MSA requires:

- New skills and new ways of thinking for developers, architects, quality professionals and infrastructure and operations staff.
- Mastery of agile and DevOps practices, including organizational changes to enable a product-oriented — rather than project-oriented — model for development work.
- Fully automated testing, with a "shift left" that has developers creating tests and a "shift right" that embraces validation in production, observability and canary deployments.
- Automated, cloud-native infrastructure with infrastructure-as-code and immutable services.

For guidance on the prerequisites needed to successfully adopt MSA, see How to Succeed With Microservices Architecture Using DevOps Practices. For insight into the pitfalls of microservice adoption, see 10 Ways Your Microservices Adoption Will Fail — And How to Avoid Them.

The good news is that MSA is not an all-or-nothing approach. It is possible to use the principles of MSA to deliver benefit to your organization incrementally, and to move in stages toward an application architecture composed of loosely coupled, independently scalable components. Gartner's research has shown that organizations can increase agility and scalability by following an iterative, incremental approach with four key stages:

- Identify When MSA Will Deliver Value: Choose targets carefully, identifying cases where an application will benefit from investment in more frequent deployments or the ability to scale in a fine-grained way. Systems that are stable, predictable and low-complexity are poor fits for MSA.
- Master Agile and DevOps: Build the skills, processes and tools necessary to design, develop and deliver fine-grained, loosely coupled applications. Without these prerequisites, you will not be able to take advantage of improved architecture.
- Evolve Applications Over Time: Iteratively refactor applications to be more modular, more granular and less coupled. Do not make the mistake of jumping directly to microservices.
- Develop Platform Capabilities: Acquire and/or develop the capabilities that make up the complex "outer architecture" that MSA requires.

As shown in Figure 1, there is a logical sequence to the adoption of MSA principles, with successive stages building on the results of previous efforts.

Step 1: Identify When MSA Will Deliver Value

The first step on the path toward MSA is to identify places where it will add value. MSA enables greater development agility, increased deployment flexibility and more precise scalability, but not all applications need these benefits. MSA also requires organizational changes, and it increases the complexity and operational overhead of an application. Your organization needs to be ready for these changes and costs.

Where Do You Need the Benefits of MSA?

The principal reason to increase the granularity of your applications should be to increase their agility in response to demonstrated business need for that increase. There are five common drivers for refactoring an application into separate pieces:

- **Continuous change demand** — When the business owners of an application need that application to change frequently.
- **Disparate change cadence** — When separate parts of an application need to change at different rates.
- **Application scaling** — When separate parts of an application need to scale at different times and/or in different amounts.

- **Team size** — When application development teams have grown beyond a manageable "two-pizza" size, or multiple teams are working in a single codebase, splitting the application can reduce collisions and errors.
- **Architecture and technology changes** — When you want to take advantage of new technology, such as cloud-native platforms or modern programming languages.

To identify the right applications, look for cases where one or more of these drivers apply. The more of these drivers apply, the more likely the application is to benefit from MSA.

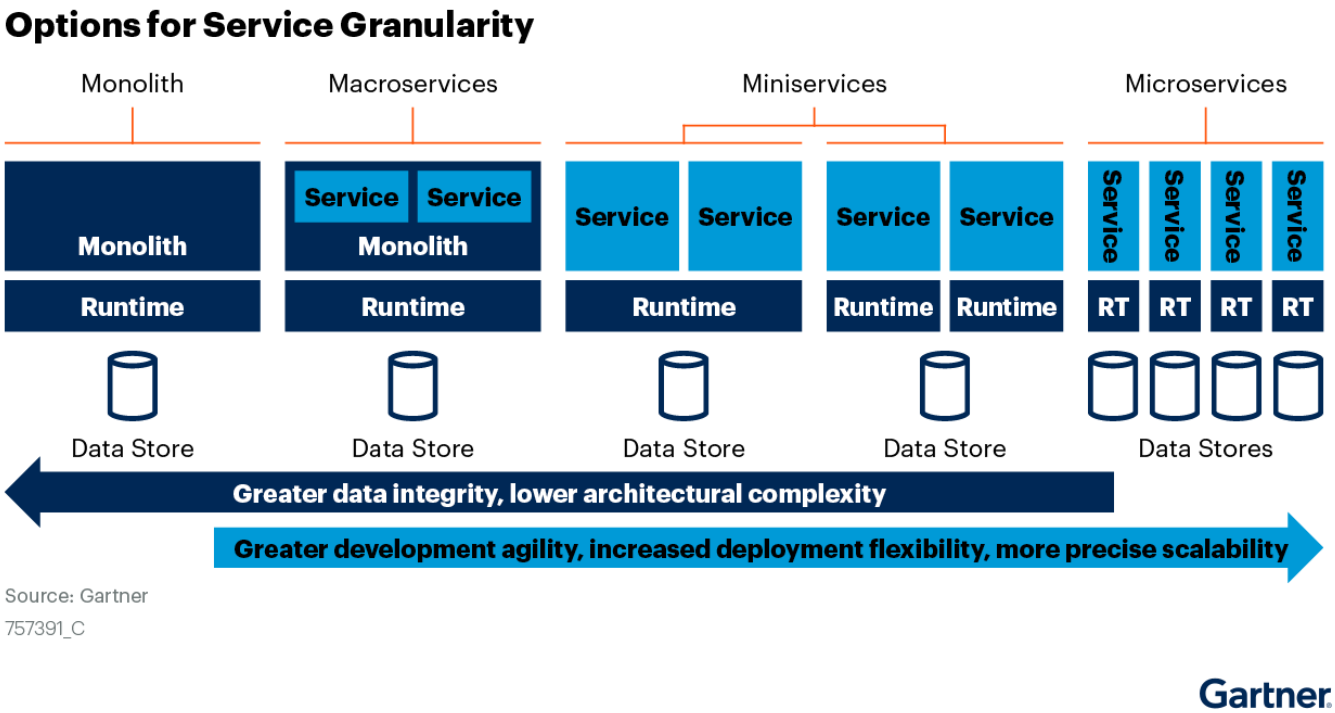
How Far Along the MSA Path Will You Need to Go?

Microservices are not the only alternative to monolithic architectures. A real-world architecture will have services at different levels of granularity. In addition to microservices, Gartner has identified two other "services" patterns:

- **Miniservices** are similar to microservices but have a larger scope and relaxed architectural constraints. Like a microservice, a miniservice is a physically encapsulated, loosely coupled, independently deployable and scalable application component. Miniservices have a broader scope than microservices, often incorporating all the functionality of a given domain. Miniservices can also share backing data stores.
- **Macroservices** logically encapsulate functionality within a monolithic application, typically exposing a request/response API. They are not physically encapsulated or independently deployable, and only rarely will they have exclusive rights to update their data source.

As shown in Figure 2, these represent points between monolith and microservice on a spectrum of options. As you move further to the right, you are gaining development agility, deployment flexibility and precision of scalability. However, you are giving up data integrity, and the complexity of the application architecture is increasing.

Figure 2. Options for Service Granularity



It is tempting to pick a target architecture and jump directly to it. It even seems like a good idea — you'll skip the "extra work" of intermediary changes. However, the likelihood of identifying the right set of services in your first iteration is low. You are better off making incremental changes, as discussed in Evolve Applications Over Time below.

Is Your Organization Ready?

Applying MSA principles, whether by decomposing your existing monoliths or by creating new independently deployable miniservices or microservices, will require you to address multiple complex problem domains, including:

- Provisioning resources for these services cost-effectively.
- Organizing people for the development and operation of these new independent services.
- Deploying from multiple independent development pipelines.
- Monitoring the servers, containers and services.
- Providing service discovery capabilities.
- Managing configuration for services.
- Tracing and troubleshooting issues across a distributed, loosely coupled architecture.

Although you do not need to (and should not) solve all of these problems before beginning your microservices journey, you will eventually encounter and need to solve all of them.

For more information on microservices options, requirements and trade-offs, see How to Succeed With Microservices Architecture Using DevOps Practices. For guidance on the leadership challenges of microservices adoption, see Leading Teams to Success With Microservices Architecture. For an overview of the pitfalls of microservices adoption, see 10 Ways Your Microservices Adoption Will Fail — And How to Avoid Them and Client Question Video: What Are the 6 Key Steps to Success With Microservices Architecture?

Set Achievable, Incremental Goals

As you proceed toward MSA, you should set goals for your work that are both achievable and incremental. Your goals should follow the SMART approach — that is, they should be specific, measurable, actionable, relevant and time-bound — and they should have the following characteristics: [1]

- **Deliverable in the short term** — Choose timelines for delivery that are framed in weeks, rather than months. This will keep the scope of individual changes small and reduce the risk of external factors disrupting the work.
- **Measured against business objectives** — Link your success criteria to business outcomes (such as faster processing of loan applications or increased customer satisfaction with a mobile app) rather than project milestones (such as release dates). This will reduce the risk of microservices adoption becoming a "technology for technology's sake" effort.
- **Scoped to deliver incremental improvement** — Focus on making meaningful changes to existing systems or delivering small new systems. Smaller efforts have a higher likelihood of success than those that attempt to deliver dramatic change or create entire new applications from scratch.
- **Aimed at "safe to fail" objectives** — Avoid "big bets." Adopting microservices requires a broad spectrum of changes, which increases the risk that any given effort will fail to reach its objectives. Regardless of the degree of success enjoyed by other organizations, until MSA is a known quantity within your organization, avoid using this approach for mission-critical efforts.

If setting near-term goals for development efforts or defining clear metrics is challenging for your organization, then it is unlikely to benefit from the kind of agility that microservices offers. You should instead aim for refactoring efforts within existing monolithic software. For additional guidance, see Refactor Monolithic Software to Maximize Architectural and Delivery Agility.

Step 2: Master Agile and DevOps

For MSA principles to deliver value to your organization, you need a software development process that can make changes quickly. The primary value of a more fine-grained service architecture is that it enables you to make changes to your software that can be quickly tested and deployed without the need for coordinated changes.

> **If you cannot develop software in small, easily testable increments, there is no point to having an architecture that could deploy those changes easily.**

Despite the name of this step, improving application agility requires more than a commitment to agile practices and DevOps. You must also adopt both “shift left” and “shift right” testing practices and embrace automation throughout the development and deployment process. Finally, you need to adopt an agile approach to application architecture.

If these practices are not already in place in your organization, you will need to develop them. Even monolithic applications benefit from shortened development cycles, greater focus on delivery of value, better test coverage and automation of the deployment pipeline.

See the following Gartner research for guidance:

- Solution Path for Agile Transformation
- Solution Path for Continuous Delivery With DevOps
- How to Test Services in an Agile Application Architecture
- To Automate Your Automation, Apply Agile and DevOps Practices to Infrastructure and Operations

Update Development Skills

Organizations that adopt MSA say the paradigm shift in their approach to development is one of the biggest challenges of adoption. Developing and operating microservices is different cognitive work from developing monolithic applications. Organizations should seek to hire for, encourage and develop these characteristics in their development teams:

- **Proficiency with modern development paradigms and concepts** — MSA encompasses service-oriented development, REST APIs, domain-driven design and event-driven architecture patterns. Developers who embrace an emergent, modular, agile approach to architecture, and who are inquisitive and receptive to new design philosophies and patterns, are likely to be successful with MSA.
- **Skills in more than one domain** — Competence across multiple domains — such as user interfaces, databases, networking, data science or operations — makes developers more effective team members. Not only are they able to fill more roles, but they are also able to understand different viewpoints and create communications channels.
- **Quality-mindedness** — Writing good unit tests and automating tests are key skills, and developers who treat these activities as valuable, first-class work instead of chores to be

avoided are highly valuable. Security, performance and stability testing may live outside the development team, but thinking about how to ensure these attributes is the responsibility of the development team as well.

- **Operations partnership** — Designing software that is easy to operate eases the burden of operations. This is especially useful in advanced DevOps, where teams that develop software are also responsible for the production operation of that software.

For more information on these skills, see the following Gartner research:

- Choosing Data-, Event- and Application-Centric Patterns for Integration and Composition
- Essential Patterns for Event-Driven and Streaming Architectures
- Essential Skills for Application Architecture
- Essential Skills for Modern Application Development
- Use Test-First Development Processes to Jump-Start Your SDLC

In addition to these behavioral characteristics, developing skills with agile development and DevOps practices is critical, not just for success with MSA, but for success with all forms of software development. Gartner recommends agile for all software development. Key aspects of agile methods and DevOps that increase the chances of success with MSA include:

- **Frequent delivery** — Agile teams deliver working code frequently, setting the stage for feedback from customers and enabling the incremental delivery of value. This also makes it easier to change direction as needed, because less work in progress must be abandoned.
- **Emphasis on quality** — Technical practices from extreme programming (XP) — such as user stories, pair programming, test-driven development and continuous integration — enable teams to deliver high-quality code that has fewer defects and greater alignment with customer needs.
- **Customer focus** — Product management as a discipline and product ownership as an explicit role in each development team create greater visibility into the desired outcomes for the end customer. These practices also provide development teams with the information they need to ensure that their software is meeting customer needs.
- **Frequent feedback** — Delivering working code into production on a regular cadence provides the opportunity for development teams to get feedback. This may be directly from end customers, from other development teams or from the production environment itself in the form of metrics and analytics.

For information on developing these skills and applying them to MSA, see:

- Create Awesome Software Using Agile Practices

- How to Succeed With Microservices Architecture Using DevOps Practices

Shift Left to Increase Quality

Testing is not just about validating that your software is free of known defects. It's also about ensuring that your software is reliable, secure and performant, and that it meets expectations. It is not possible to add quality to a system by inspecting it. The best and fastest road to quality is to make the quality of delivered code the responsibility of the development team, rather than a separate test or quality assurance (QA) team. This means embedding practices that increase quality into the day-to-day routine of development teams. This is a "shift left" approach, and it encompasses:

- **Architecting for testability** — Use modular design patterns, such as model view controller or layers, and minimize dependencies to decrease the scope of testing required to validate any given change. For information on best practices for developing modular, independent applications, see:
  - From Fragile to Agile Software Architecture
  - Refactor Monolithic Software to Maximize Architectural and Delivery Agility

- **Building in security and performance** — Create applications that are secure by design to decrease the risk of undetected security risks escaping into your production systems. Include static code analysis and incorporate guidance, such as the Open Web Application Security Project (OWASP) Top 10. [2] For additional information on implementing security in development, see:
  - A Guidance Framework for Establishing and Maturing an Application Security Program
  - Structuring Application Security Tools and Practices for DevOps and DevSecOps
  - Architect a Modern API Access Control Strategy
  - Advance Your Platform-as-a-Service Security

- **Continuous integration (CI)** — Automate build processes and leverage unit and integration tests to provide developers with quick feedback on whether their application can compile and run to reduce development cycle times. Require that developers merge code regularly — at least daily — to reduce the friction caused when code from one developer conflicts with code from another. For additional information on establishing and using CI, see Solution Path for Continuous Delivery With DevOps.

This does not mean that a separate QA team cannot add value by maintaining test frameworks and test data, or by taking responsibility for integration and regression testing. However, the

primary responsibility for quality lies with the development team, and their practices should reflect this.

Shift Right to Increase Agility

As the granularity of your services and the complexity of your architecture increase, the traditional approach of testing the application as a single, unified whole can become a significant impediment. Even a low-complexity miniservice architecture may involve a dozen or more independent applications and multiple data stores, each with its own dependencies.

In Testing Microservices, the Sane Way, Cindy Sridharan argues that continuing to use a monolithic approach to testing will result in creating a "distributed monolith." Imposing the need for coordinated testing on microservices infrastructure will rob the resulting application of many of its potential benefits because the organization will struggle to maintain a stable, consistent test environment.

As an alternative, Sridharan proposes that organizations focus on testing approaches that embrace emergent practices such as chaos engineering, blue-green deployments and canary releases to enable a significant portion of testing work to occur in production. [3,4,5]

This "shift right" in testing requires your organization to have a high degree of technical sophistication, including the ability to carefully segment user and other traffic, to detect anomalies and to respond quickly. It also requires a willingness to experiment on real users that your organization may not possess.

For guidance on this topic, see Sridharan's Testing in Production, the Safe Way and Testing in Production: The Hard Parts, as well as the following Gartner research:

- How to Safely Begin Chaos Engineering to Improve Reliability
- How to Test Services in an Agile Application Architecture

Automate Deployment

Code is not valuable until it reaches production and is being used. The process of getting it there should be stable, predictable and repeatable, and it should be built into the CI pipeline.

> **From a DevOps perspective, manual work in a repetitive task is both insulting and dangerous.**

Manual, repetitive tasks are insulting because they do not require problem solving or human intelligence, and they are dangerous because introducing humans into the process introduces the risk of human error into each iteration of the process.

As the granularity of your architecture and the frequency of deployment increase, having a reliable, automated deployment pipeline for each service will steadily become more important. If you do not have a CI pipeline in place — with each developer checking in code at least daily — do not attempt to build fine-grained services. Without a feedback loop to provide your developers with information about whether their applications are working or not, your systems will quickly become unstable and your pace of development will slow to a crawl.

There are three "levels" of CI:

- **Continuous integration** — In a CI pipeline, every check-in results in build and integration of the code, but not deployment to higher environments. For code to move toward production, it must be promoted by human intervention.
- **Continuous delivery** — Continuous delivery builds on CI, automating the deployment process to enable push-button delivery of code to production. Every check-in results in code being tested to prove that it is deployable, but code is not deployed automatically. Instead, code that has passed testing is deployed when humans trigger the deployment process. This is the approach used by organizations that deploy multiple times per day, with fully automated deployments that are initiated by human action.
- **Continuous deployment** — In a full continuous deployment pipeline, every check-in results in code, if it passes tests, being deployed to production without human intervention. This is the approach used by organizations at the forefront of development sophistication, such as Amazon, Netflix, Slack and Meta (formerly Facebook). In most cases, continuous deployment pipelines deploy to a small subset of production systems using the canary deployment pattern, with staged rollout to other systems after code is proven stable in production.

In all scenarios, the purpose of the pipeline is as much to provide feedback to developers as it is to promote code toward production. Fast, meaningful feedback about the quality of code, delivered frequently, enables developers to deliver applications more quickly and to reduce errors.

For additional guidance on creating and using automation and CI pipelines, see:

- To Automate Your Automation, Apply Agile and DevOps Practices to Infrastructure and Operations
- How to Architect Continuous Delivery Pipelines for Cloud-Native Applications
- Keys to DevOps Success

Develop Architecture Skills to Succeed With Microservices

With MSA, the job of the architect is even more important. In addition to architecting the platform, they are also responsible for deciding on the overall structure of the system and the inner architecture of the services. Because the services are no longer part of a single application, the application becomes an emergent property of the set of services. To create a coherent

application from independently developed services, someone needs to think and design at a high level to coordinate the work of the development teams creating services.

See “Part A: Coordinate the Product Architecture” in MASA: How to Create an Agile Application Architecture With Apps, APIs and Services for guidance on the responsibilities of architects in creating composite applications.

Step 3: Evolve Applications Over Time

For the best results, you should apply MSA principles gradually, increasing the granularity and agility of your applications and architecture in an iterative and incremental fashion. This may not result in the creation of “true” microservices. The stopping point of your journey is the point at which making the application more fine-grained will not deliver enough value to justify the costs of both development and ongoing operations.

Each application’s path toward MSA is different, but most follow one of three paths:

- **Refactoring** an existing application to services to increase agility, flexibility and scalability. This is a common practice when moving applications to the cloud.
- **Replacing** an existing application by creating a new version, built from the ground up as services. The replacement may be driven by a desire to use a more modern technology stack for the application, accompanied by a decision to move to modern architecture, such as MSA.
- **Creating** an entirely new “greenfield” application that leverages MSA from the start. This includes cases where an existing application is being extended with new functionality using MSA.

These paths are shown in Figures 3, 4 and 5.

Figure 3. Refactoring an Existing Application to Adopt Service-Based Architecture

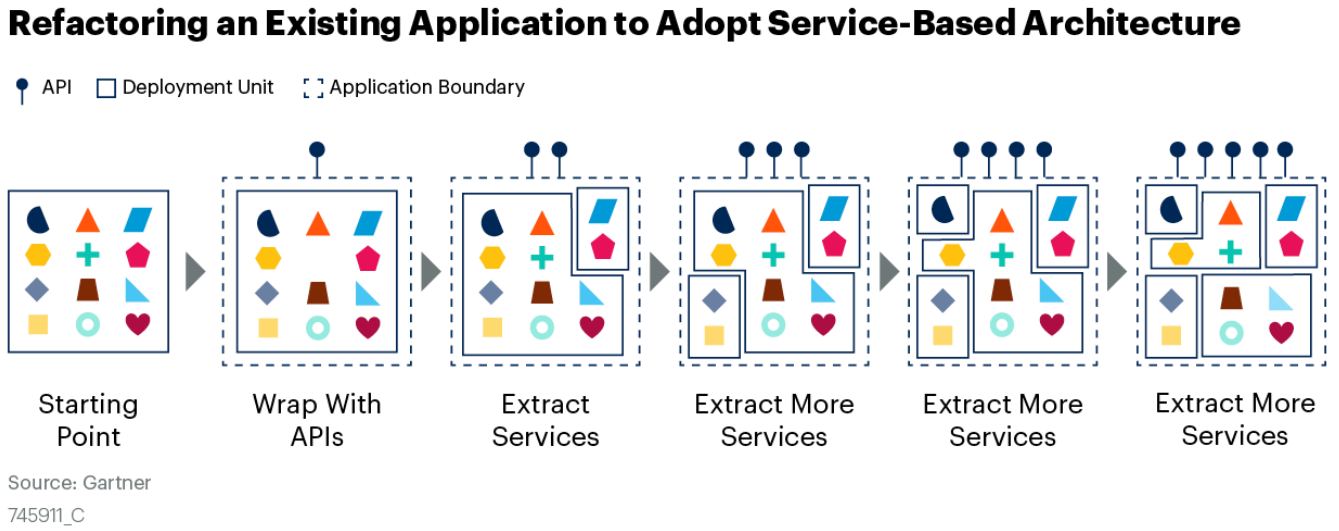
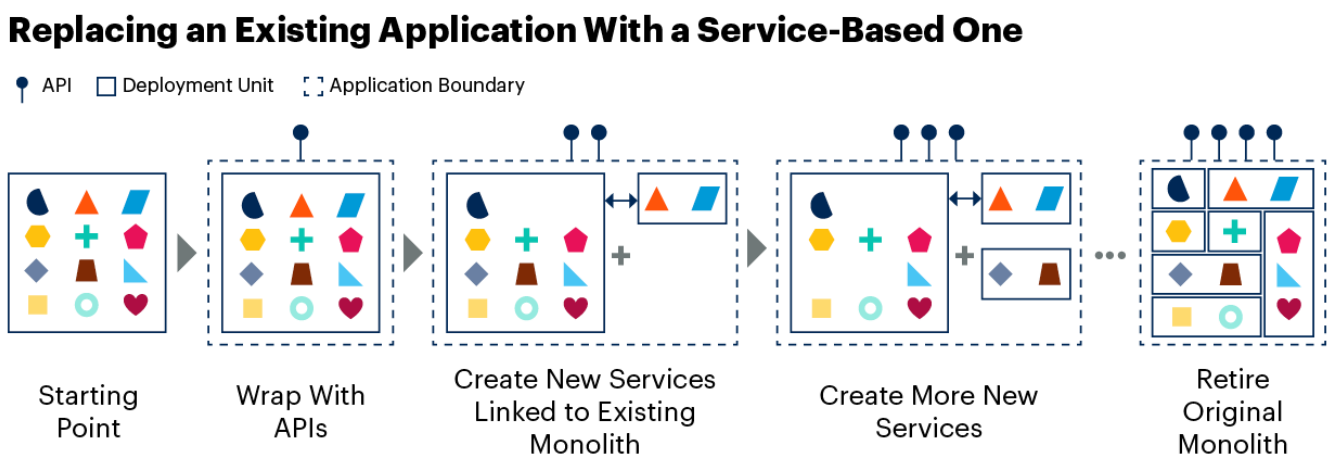


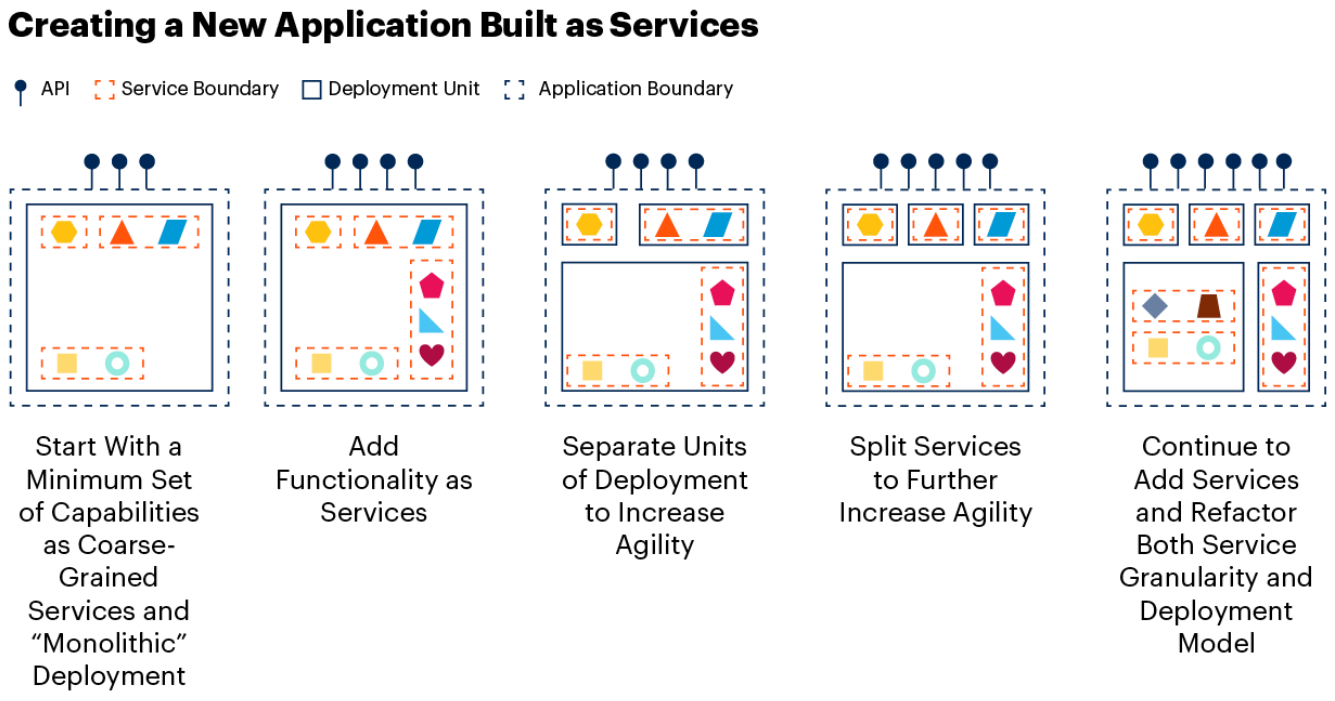
Figure 4. Replacing an Existing Application With a Service-Based One



Source: Gartner
757391_C

Gartner

Figure 5. Creating a New Application Built as Services



Source: Gartner
745911_C

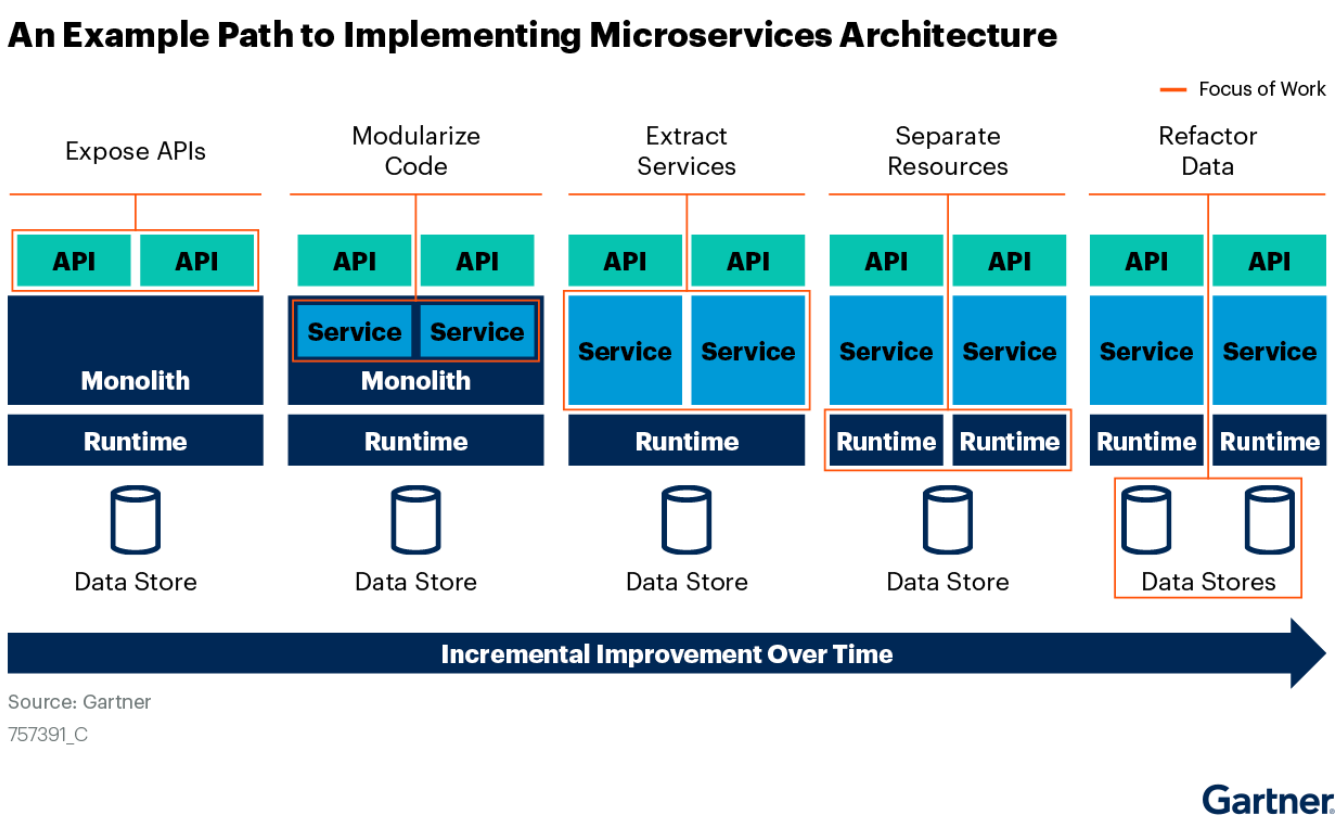
Gartner

Figure 6 shows an example of the steps involved in an effort to refactor an application from monolithic to microservices. The major steps of this process are:

- **Expose APIs** to gain control over interactions with your application.

- **Modularize the application's code** to increase agility, creating macroservices.
- **Extract services** from your monolith to create miniservices.
- **Separate resources** to enable greater flexibility in deployment and scaling.
- **Refactor data** to enable the creation of fine-grained microservices for extreme agility or scaling.

Figure 6. An Example Path to Implementing Microservices Architecture



In some cases, boundaries within an application will be well-understood and well-defined enough to support an immediate move to miniservices, or even microservices. In other cases, you may learn enough from remodularizing to develop macroservices that you are able to extract fine-grained microservices immediately.

In all cases, use caution when refactoring applications to make them more fine-grained. As you extract services, make sure that they are highly cohesive, so that they encapsulate needed changes within their boundary.

If your services are too fine-grained, you will need to make coordinated changes to multiple services, rather than changing a single service. This negates one of the primary benefits of MSA.

The reason to increase the granularity of your services is to decouple them from one another so that they can change independently. In addition, the more fine-grained your services are, the

greater demands they will place on your platform infrastructure and the teams operating it.

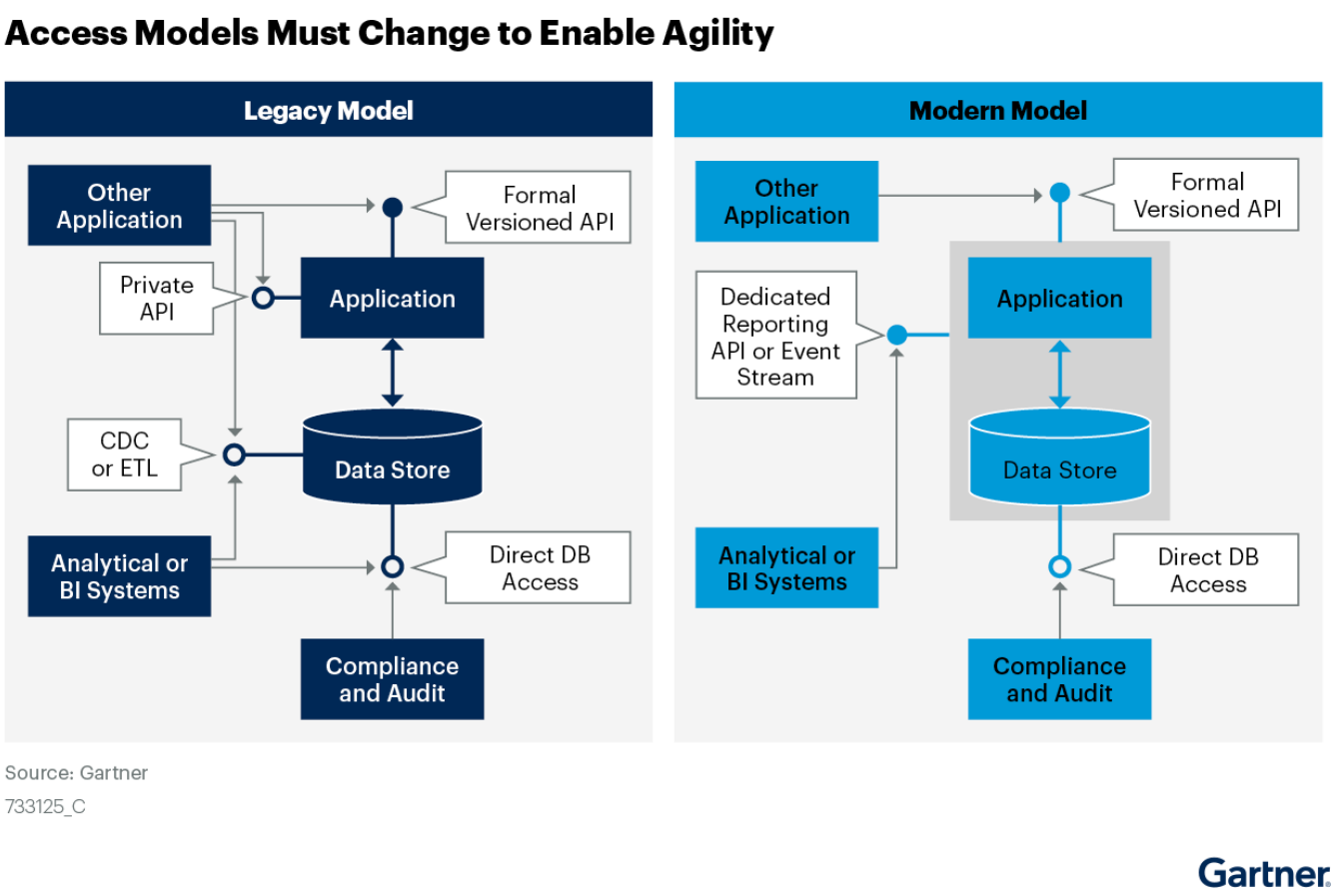
As you make changes, remember that not all applications move at the same pace, nor do they all have the same destination state. At any given time, your architecture may be composed of monolithic applications, macroservices exposed via API from within monoliths, independently operating miniservices and fine-grained microservices. Handling this increase in complexity is one of the key challenges of adopting microservices, and a key reason why improving skills and processes is a necessary step before increasing the granularity of your systems.

For more information on decomposition patterns and service and microservice design, see Designing Services and Microservices to Maximize Agility.

Expose APIs to Seize Control of Interactions

Enterprise applications inevitably have some kind of “backdoor integration” that makes changing them difficult — often some form of direct database access by other applications. To insulate other applications from the changes you will make to your application, you must move all interactions to programmatic interfaces with clear interface contracts. Figure 7 shows this shift from a legacy model with multiple entry points to a modern model where the application controls all interfaces.

Figure 7. Access Models Must Change to Enable Agility



To create these interfaces, REST APIs are the most common approach used by Gartner clients, but they are not the only option. Alternatives include gRPC, event-based APIs and GraphQL. To

create these interfaces:

- Start by identifying the desired uses for application functionality and the associated business use cases and target consumers.
- Using an iterative, consumer-feedback-driven process, create technical and legal contracts for APIs, including machine-readable API specifications.
- Choose REST over SOAP for newly created APIs due to its greater flexibility, ease of implementation and wider adoption.

For insights into the design and development process for APIs and event streams, see:

- How to Design Great APIs
- How to Deliver Sustainable APIs
- A Guidance Framework for Creating Usable REST API Specifications
- Assessing GraphQL for Modern API Delivery
- Choosing Event Brokers: The Foundation of Your Event-Driven Architecture
- Essential Patterns for Event-Driven and Streaming Architectures
- Choosing Data-, Event- and Application-Centric Patterns for Integration and Composition

Modularize Code to Reduce Testing Scope and Deployment Effort

Monolithic applications impede agility because their code is managed and deployed as a single unit. Even small coding changes require the entire codebase to be compiled, tested and packaged. In addition, the functionality and data locked inside these applications can be challenging or impossible to reuse outside the application.

It is possible to increase the delivery agility of a monolithic application without the need for the complex outer architecture that is required to support miniservices or microservices. To do this, you must modularize your application's code, reducing the scope of any given change. This not only increases the agility of your application, but also lays the groundwork for further decomposition to miniservices or microservices. To do this:

- Start by ensuring that you have a comprehensive set of tests to validate application functionality, so that you can make changes safely.
- Analyze the application's current structure to find functional or logical groupings with minimal coupling and few dependencies, and focus your efforts on those areas.
- Modularize the application in stages, validating continued functionality at each step, and use CI to ensure that your application is stable as you change its internal structure.

For additional guidance on refactoring applications to increase modularity, see:

- Refactor Monolithic Software to Maximize Architectural and Delivery Agility
- From Fragile to Agile Software Architecture
- Essential Skills for Application Architecture

Extract Functionality as Miniservices to Increase Agility

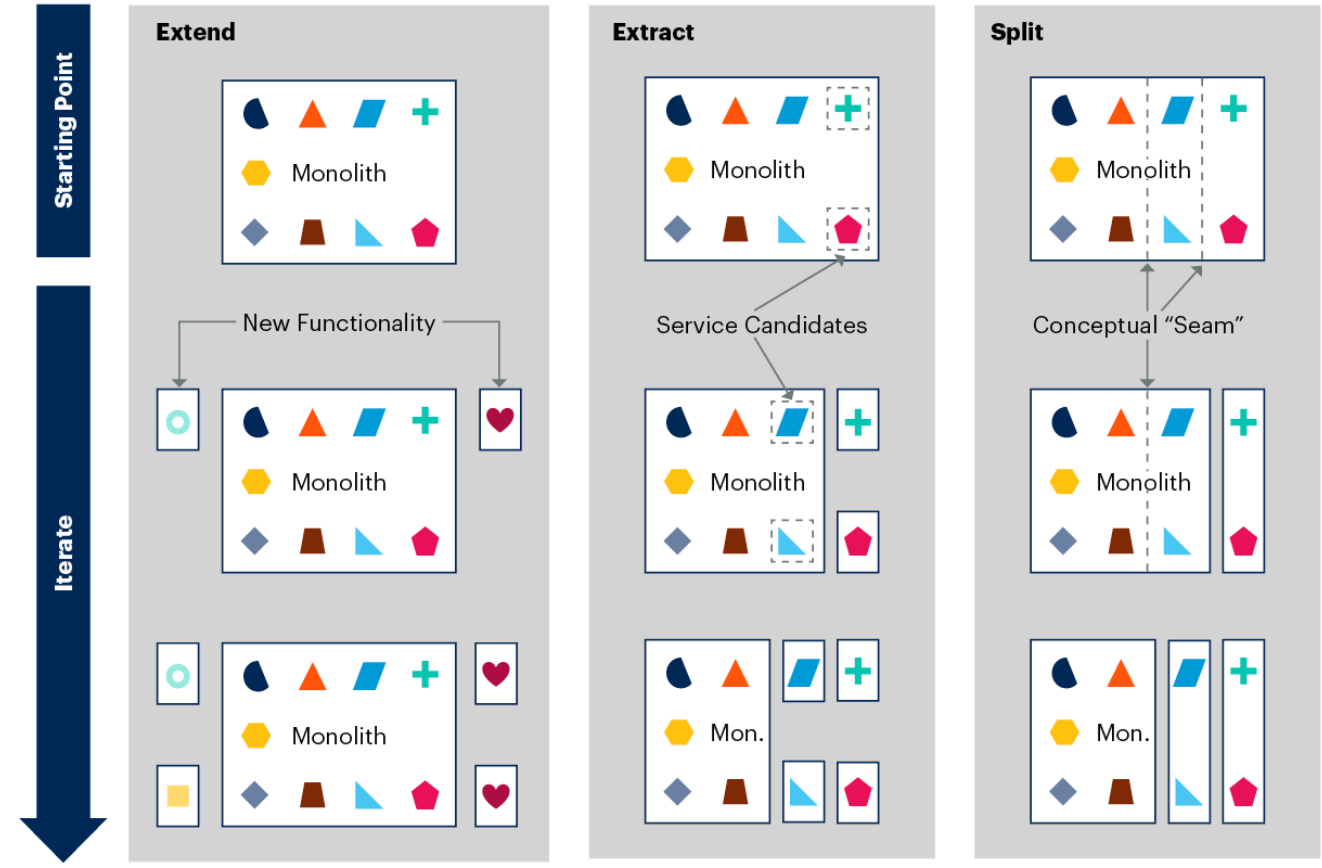
Miniservices are coarse-grained, independently deployable, independently scalable application components. Because they are independently deployable and scalable, they have similar agility and scalability benefits to microservices. Because they are coarse-grained and may share data between services, they are easier to develop and operate than microservices.

Designing Services and Microservices to Maximize Agility describes three approaches for refactoring monolithic applications to extract services. These approaches, shown in Figure 8, are:

- **Extend** the monolith, leaving it intact but adding new functionality as independent services.
- **Extract** services from the monolith, pulling functionality out into independent services.
- **Split** the monolith, breaking it into smaller pieces that are still coarsely grained.

Figure 8. Three Approaches to Extracting Services From Monolithic Applications

Three Strategies for Decomposing Monolithic Software



Source: Gartner
745911_C

Gartner

All of these are iterative, and you do not need to stick to a single approach. You might first split your monolith to reduce its size, then extract some services from it, and then extend the remaining monolith, having determined there is no value to be gained from further refactoring.

If you're unsure where to start, the "extract" pattern is the most likely to produce good results in the short term, because it decreases the scope of the monolith while also limiting the scope of the changes you are making.

Separate Runtime Resources to Increase Agility

The shared resources of a miniservices architecture, particularly the application runtime, can prevent teams from moving at the speed necessary to meet business goals. When two or more services share a runtime environment, a team that wants to change the technology stack of one service cannot do so without coordinating this change with other teams. Even less drastic changes can face resistance. A team that wants to introduce new or innovative functionality may present a danger to other teams that want their operating environment to remain stable. This leads to the need for coordinated testing and deployment, which greatly inhibits agility.

To address this, you must separate the application runtime environment that your services operate within. The considerations for this work include:

- Evaluating the behavior of your services to discover cases where separating their runtimes will cause issues.
- Discovering and removing all instances of shared state, such as HTTP session state.
- Discovering and removing all uses of shared cache.
- Modifying both the services that are leaving the shared environment and those that are remaining to remove direct calls that will no longer work when the services do not share an environment.

Note that as your services multiply in number, you will need more of the “outer architecture” capabilities that are described in Step 4.

For additional guidance on these topics, see:

- How to Modernize Your Application to Adopt Cloud-Native Architecture
- How to Succeed With Microservices Architecture Using DevOps Practices

Separate Data to Create Microservices

True microservices are tightly scoped, strongly encapsulated, loosely coupled, independently deployable and independently scalable. Your organization may never need to develop microservices. It is possible — even likely — that all of your needs will be met with a combination of macroservices and miniservices. In some cases, however, you will need either extreme agility, extreme scaling or both.

- **Extreme agility** — Large applications composed of many services can drive the need for extreme agility. For example, Meta pushes more than 1,000 changes per day into its codebase. [6] Having the ability to deploy frequently enables your organization to take risks safely. When deploying a valuable change or a fix to remediate an issue takes minutes instead of days, your organization can realize value more quickly.
- **Extreme scaling** — Applications with highly variable workloads that affect parts of the application differently can benefit from very fine-grained services. For example, an online retailer such as Best Buy can see its conversion rate (that is, the percentage of site visitors who are making purchases) increase dramatically during special events. Having the ability to scale up different capabilities at different rates (such as doubling front-end web servers but quadrupling order processing) can drive significant cost savings.

The final frontier for refactoring to microservices is the data that your services reference and modify. For most organizations, the hardest part of implementing microservices is separating

data and clearly enforcing transaction boundaries and data ownership. Managing data in an MSA is substantially harder than in a traditional monolithic architecture, because MSA prioritizes agility and scalability above data consistency.

> **Distributing your data across multiple services is challenging to implement and makes debugging, tracing and troubleshooting issues more difficult.**

The use of per-service data stores to enable independent changes to data models and data persistence technology means that data consistency and relationships are managed in service and interface design, not in the database.

To successfully separate data, you should use transaction boundaries to guide your refactoring. Mismatches between transaction boundaries and service boundaries create the need for coordinated transactions, which are difficult to implement.

As you implement microservices, you should investigate the following technologies and patterns:

- Event-driven architecture (EDA)
- Streaming architectures
- Event sourcing
- CQRS
- Sagas

For additional guidance on these topics, see:

- Working With Data in Distributed and Microservices Architectures
- Essential Patterns for Event-Driven and Streaming Architectures
- Choosing Event Brokers: The Foundation of Your Event-Driven Architecture

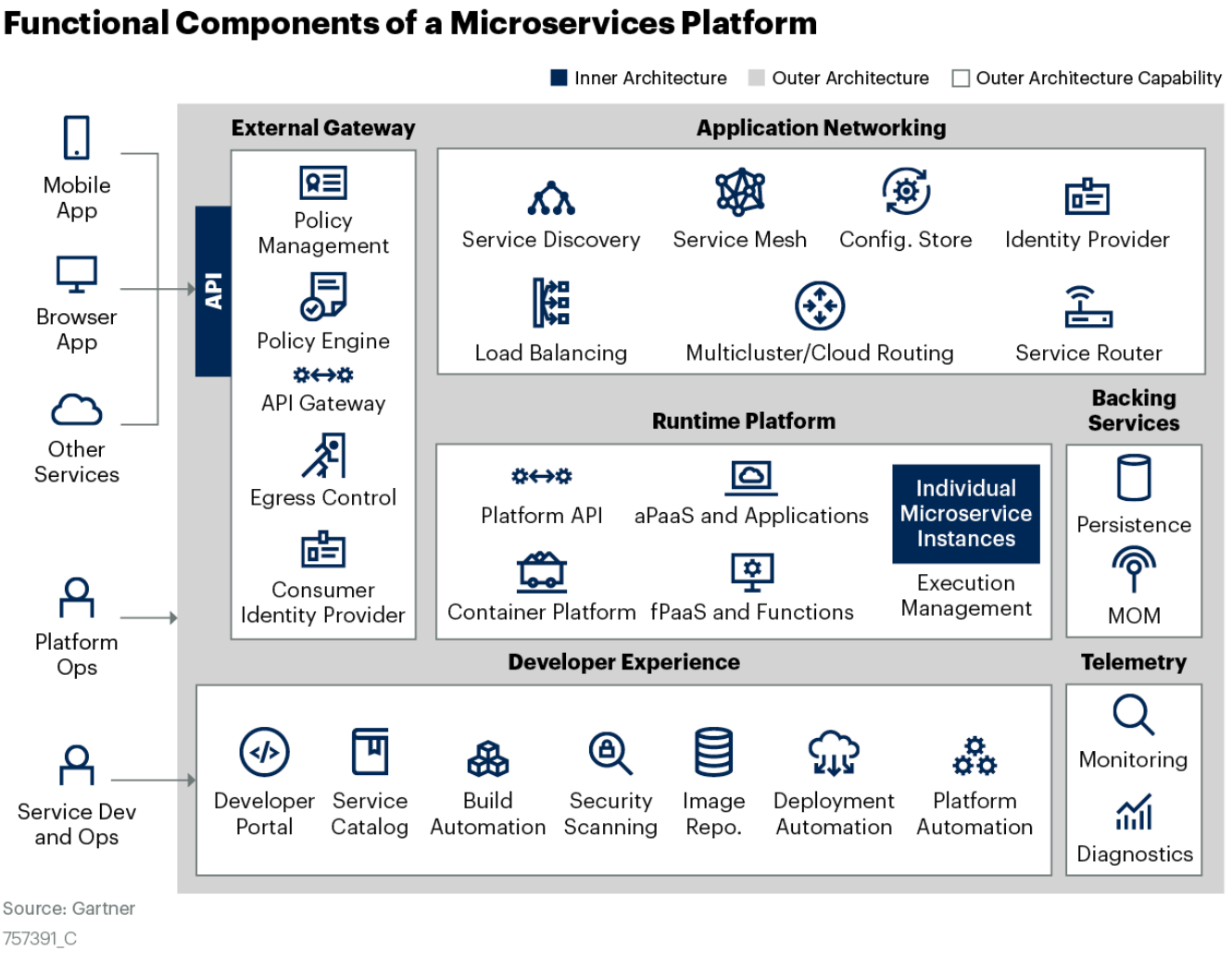
Step 4: Develop Platform Capabilities

Organizations that move toward more granular architectures find themselves dealing with a somewhat paradoxical situation: as the complexity of individual services decreases, that of the surrounding ecosystem increases. For this reason, it's useful to think about MSA from two perspectives:

- **The inner architecture** — This is the software architecture of the microservice itself, comprising the development and runtime stack of the service. In most cases, this is packaged in a container that includes the dependencies of the service to allow for easier deployment. The inner architecture may be heterogeneous to enable individual teams to optimize their development process for the specific problem they are solving. As you adopt MSA principles, your inner architecture should simplify because the microservice is becoming more focused.
- **The outer architecture** — This is the environment in which individual services run, providing the context in which they operate. Unlike the heterogeneous individual services, the outer architecture should be generally consistent because it must be supported across the entire organization. As you adopt MSA principles, your outer architecture will become more complex because the capabilities being moved out of the microservices will still be needed.

The "outer architecture" is a shared self-service app platform and should be managed by a Platform Ops team, as described in Using Platform Ops to Scale and Accelerate DevOps Adoption. See also Guidance Framework for Implementing Cloud Platform Operations for additional guidance on Platform Ops. Figure 9 shows the key capabilities of a microservices platform.

Figure 9. Functional Components of a Microservices Platform



Source: Gartner
757391_C

The core capabilities of a microservices platform are:

- **Runtime platform** — Individual service instances need an environment in which to run. A public or private cloud platform manages the execution runtime of individual service instances and may provide some or all of the other capabilities of the outer architecture. For guidance on choosing a platform, see:
  - Selecting a Cloud Platform for DevOps Delivery With Microservice Architecture
  - Solution Criteria for Public Cloud Kubernetes Services
  - Solution Scorecard for Amazon Elastic Kubernetes Service
  - Solution Scorecard for Microsoft Azure Kubernetes Service

- **External gateway** — Whether you are operating macroservices, miniservices or microservices, you will almost certainly need to expose web APIs for consumption by business apps and applications. Managing access and enforcing traffic management and security policies require some form of external gateway. This may be implemented using a dedicated full-life-cycle API management solution or a distributed microgateway. For additional insight into this topic, see:
  - A Guidance Framework for Evaluating API Management Solutions
  - How to Successfully Implement API Management
  - Decision Point for Mediating API and Microservices Communication

- **Application networking** — Operating services in a loosely coupled fashion across a distributed environment creates the need for a set of capabilities that provide internal communications and dependency management between service instances, including internal gateways and lightweight mediation. The capabilities of a service mesh include orchestration, service routing, service discovery, load balancing and identity provisioning. For insights into the capabilities of service mesh, see:
  - Assessing Service Mesh for Use in Microservices Architectures
  - Choosing Data-, Event- and Application-Centric Patterns for Integration and Composition
  - Using Emerging Service Connectivity Technology to Optimize Microservice Application Networking
  - What a Microservices Process Orchestration Framework Is and When to Use One

- **Backing services** — Services that maintain state need to persist their data, and loosely coupled services benefit from using asynchronous, message-driven or event-driven communications. Persistence options include relational data stores, distributed NoSQL databases or in-memory data stores. For guidance on evaluating middleware solutions aimed at loosely coupled, event-driven architectures, see:
  - Decision Point for Mediating API and Microservices Communication
  - Choosing Event Brokers: The Foundation of Your Event-Driven Architecture

- **Developer experience** — As the complexity of your architecture increases, so too does the importance of removing manual effort and the associated risk of human error from building, testing and deploying software. In addition, automation of the platform itself can further reduce both labor and risk. For additional guidance on containers and platform automation, see:
  - How to Architect Continuous Delivery Pipelines for Cloud-Native Applications
  - Using Kubernetes to Orchestrate Container-Based Cloud and Microservices Applications
  - Solution Path for Implementing Containers and Kubernetes

- **Telemetry** — Keeping track of individual service instances in an increasingly fine-grained environment creates significant challenges. Key capabilities include monitoring, alerting, logging and diagnostics. Traditional monitoring tools that focus on the "server" are ill-suited to ephemeral, ever-changing MSA. Focusing on "observability" — the degree to which applications can be monitored, traced, assessed and analyzed — will help greatly when you are implementing your monitoring instrumentation, and even more when you are trying to figure out what went wrong. For additional insight into this subject, see:
  - Assessing OpenTelemetry's Impact on Application Performance Monitoring
  - Monitoring Containers and Kubernetes Workloads
  - Monitoring and Observability for Modern Services and Infrastructure

Build Capabilities as You Need Them

You do not need to build the entire platform at once, and it's often unwise to do so. Instead, use the same iterative approach that you are using to adopt MSA practices to guide your implementation of a microservices platform.

Avoid Delay by Making Low-Consequence Decisions Quickly

Making decisions about outer architecture may seem like a daunting task. You will be tempted to avoid the risk of bad decisions by deferring microservices adoption until key outer architecture decisions have been made. However, doing so creates delay, which can be more costly. To avoid

this delay, use the decision typing technique described in 6 Essential Techniques for Reducing Risk and Uncertainty in Architecture Decisions, which classifies decisions into Type 1 (difficult or costly to change later) and Type 2 (easy to reverse or change later).

Some decisions related to the outer architecture of MSA are Type 1 decisions, such as choosing an application platform. Others, such as the tech stack for a single service, are Type 2. Because the beginning of any effort is when you know the least about it, you should seek to make these Type 2 decisions quickly to minimize delay. If you determine that those decisions were incorrect, you can reverse or change them.

Implement the Platform in Stages

Because adopting MSA is not a short-term or one-time project, you will have the opportunity to revisit your initial decisions multiple times. Note that retaining the ability to change technology decisions at a later date requires that you follow good practices with regard to modularity of code and management of dependencies. In addition, following these principles will help limit your commitment to any given technology:

- **Limit the degree of integration —** Within any given application, use a new technology in as few places as possible until you are comfortable that you can make a longer-term commitment. The fewer places that you need to make changes if and when you swap out a given technology, the better.
- **Control scope of usage —** Across your organization, use a new technology in as few places as possible, for the reason noted in the previous point. In addition, limiting usage by other teams and applications reduces the chance of someone else in your organization creating strong coupling to a given technology.
- **Use abstraction to limit coupling —** Where possible, create abstraction layers to wrap the capabilities of a given technology and insulate your services.

Do Not Implement a Platform Until You Need One

A small but troubling subset of Gartner inquiry calls on microservices architecture come from operations teams who have acquired, installed and configured platform software and are seeking to convince applications teams to use it. This often reflects an attempt to follow this Solution Path from right to left — buying a platform, then changing architecture to adopt agile practices, then seeking to define business value for work already done. This does not work.

Instead, let your development processes surface your requirements for platform capabilities. Your enterprise architecture (EA) team should be closely involved in the adoption of MSA principles. Enterprise, solution and application architects should be working together to maintain an "architectural runway" of capabilities delivered just slightly in advance of need. [7]

For more information on emergent architecture and modular application design, see:

- From Fragile to Agile Software Architecture
- Best Practices for Unlocking Architectural Agility
- 10 Essential Skills of the Modern Software Architect
- Using Platform Ops to Scale and Accelerate DevOps Adoption

Evidence

[1] So You've Been Asked to Create an IT Service Portfolio and IT Service Catalog

[2] OWASP Top Ten, Open Web Application Security Project (OWASP).

[3] Principles of Chaos Engineering, Principles of Chaos Engineering.

[4] BlueGreenDeployment, martinfowler.com.

[5] CanaryRelease, martinfowler.com.

[6] Rapid Release at Massive Scale, Facebook.

[7] Architectural Runway, Scaled Agile Framework (SAFe).

Document Revision History

Solution Path for Applying Microservices Architecture Principles to Application Architecture - 17 December 2019

Solution Path for Using Microservices Architecture Principles to Deliver Applications - 9 March 2018

Recommended by the Author

How to Succeed With Microservices Architecture Using DevOps Practices

Designing Services and Microservices to Maximize Agility

Working With Data in Distributed and Microservices Architectures

10 Ways Your Microservices Adoption Will Fail — And How to Avoid Them

Client Question Video: What Are the 6 Key Steps to Success With Microservices Architecture?

Using Platform Ops to Scale and Accelerate DevOps Adoption

Decision Point for Mediating API and Microservices Communication

What a Microservices Process Orchestration Framework Is and When to Use One

Recommended For You

Essential Patterns for Event-Driven and Streaming Architectures

Working With Data in Distributed and Microservices Architectures

Designing Services and Microservices to Maximize Agility

Decision Point for Choosing a Mobile App Architecture

How to Identify and Resolve Application Performance Problems

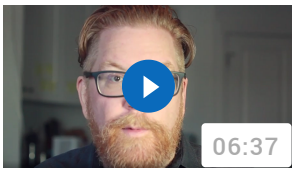
Recommended Multimedia

VIDEO

What Techniques Will Help Us Learn Application Architecture and Development Skills?

VIDEO

What Are the 6 Key Steps to Success With Microservices Architecture?



VIDEO

Mobile App Architecture Introduction

Supporting Initiatives

Application Architecture and Platforms for Technical Professionals

Track