# Microservices

## Understanding Microservices

While the term microservices has probably been used in various forms for many years, the association it now has with a particular way of building software came from a meeting attended by a handful of software architects. This group saw some commonality in the way a particular set of companies was building software and gave it a name.

James Lewis is a Principal Consultant at ThoughtWorks and member of the Technology Advisory Board. James' interest in building applications out of small collaborating services stems from a background in integrating enterprise systems at scale. He's built a number of systems using microservices and has been an active participant in the growing community for a couple of years. He identifies three concepts that are principal to the style:

***Microservices are ideal for big systems***

The common theme among the problems that people were facing was related to size. This is significant because it highlights a particular characteristic of the microservices style—it is designed to solve problems for systems that are big. But size is a relative measure, and it is difficult to quantify the difference between small, normal, and big. You could of course come up with some way of deciding what constitutes big versus small, perhaps using averages or heuristic measurements, but that would miss the point. What the architects at this gathering were concerned with was not a question of the size of the system. Instead, they were grappling with a situation in which the system was *too* big. What they identified is that systems that grow in size beyond the boundaries we initially define pose particular problems when it comes to changing them. In other words, new problems arise due to their *scale*.

***Microservice architecture is goal-oriented***

Something else we can derive from James' recollection of the day is the focus on a *goal* rather than just a solution. Microservice architecture isn't about identifying a specific collection of practices, rather it's an acknowledgment that software professionals are trying to solve a similar goal using a particular approach. There may be a set of common characteristics that arise from this style of software development, but the focus is meant to be on solving the initial problem of systems that are too big.

***Microservices are focused on replaceability***

The revelation that microservices are really about replaceability is the most enlightening aspect of the story. This idea that driving toward replacement of components rather than maintaining existing components get to the very heart of what makes the microservices approach special.

## Adopting Microservices

If you are responsible for implementing technology at your company, the microservices proposition should sound enticing. Chances are you face increasing pressure to improve the changeability of the software you write in order to align better with a business team that wants to be more innovative. It isn't easy to make a system more amenable to change, but the microservice focus on building replaceable components offers some hope.

In particular, after learning more about microservices methods, potential adopters frequently identify the following issues:

- They have already built a microservice architecture, but they didn't know it had a name.
- The management, coordination, and control of a microservices system would be too difficult.
- The microservices style doesn't account for their unique context, environment, and requirements.

## What are microservices?

Microservices are small, autonomous services that work together.

—Sam Newman, Thoughtworks

Loosely coupled service-oriented architecture with bounded contexts.

—Adrian Cockcroft, Battery Ventures

"A *microservice* is an independently deployable component of bounded scope that supports interoperability through message-based communication. *Microservice architecture* is a style of engineering highly automated, evolvable software systems made up of capability-aligned microservices."

We cannot say there is a formal definition of the microservices architectural style. If you are considering adopting a microservice architecture for your organization, consider how effective the existing architecture is in terms of changeability and more specifically replaceability. Are their opportunities to improve? Could you go beyond modularity, Agile practices, or DevOps to gain value? We think you'll stand a better chance at providing value to your business team if you are open to making changes that will get you closer to those goals.

## "How could this work here?"

Microservice applications share some important characteristics:

- Small in size
- Messaging enabled
- Bounded by contexts
- Autonomously developed
- Independently deployable
- Decentralized
- Built and released with automated processes

That's a big scope! So big that some people believe that microservices describe a software development utopia—a set of principles so idealistic that they simply can't be realized in the real world. But this type of claim is countered with the growing list of companies who are sharing their microservice success stories with the world. You've probably heard some of those stories already—Netflix, SoundCloud, and Spotify have all gone public about their microservices experiences.

## "How would we deal with all the parts? Who is in charge?"

Two microservices characteristics that you might find especially concerning are decentralization and autonomy. **Decentralization** means that the bulk of the work done within your system will no longer be managed and controlled by a central body. Embracing team autonomy means trusting your development teams to make their own decisions about the software they produce. The key benefit to both of these approaches is that software changes become both easier and faster—less centralization results in fewer bottlenecks and less resistance to change, while more autonomy means decisions can be made much quicker.

But if your organization hasn't worked this way in the past, how confident are you that it could do so in the future? For example, your company probably does its best to prevent the damage that any single person's decisions can have on the organization as a whole. In large companies, the desire to limit negative impact is almost always implemented with centralized controls—security teams, enterprise architecture teams, and the enterprise service bus are all manifestations of this concept. So, how do you reconcile the ideals of a microservice architecture within a risk-averse culture? How do we *govern* the work done by microservices teams?

Similarly, how do you manage the output of all these teams? Who decides which services should be created? How will services communicate efficiently? How will you understand what is happening?

We've found that decentralization and control are not opposing forces. In other words, the idea that there is a trade-off between a decentralized system and a governed system is a myth. But this doesn't mean that you gain the benefits of decentralization and autonomy for free. When you build software in this way, the cost of controlling and managing output increases significantly. In a microservice architecture, the services tend to get simpler, but the architecture tends to get more complex. That complexity is often managed with tooling, automation, and process.

# The Microservices Way

More specifically, the real value of microservices is realized when we focus on two key aspects—*speed* and *safety*. Every single decision you make about your software development ends up as a trade-off that impacts these two ideals. Finding an effective balance between them at *scale* is what we call the *microservices way*.

## The Speed of Change

The desire for speed is a desire for immediate change and ultimately a desire for adaptability. On one hand, we could build software that is capable of changing itself— this might require a massive technological leap and incredibly complex system. But the solution that is more realistic for our present state of technological advancement is to shorten the time it takes for changes to move from individual workers to a production environment.

Years ago, most of us released software in the same way that NASA launches rockets. After deliberate effort and careful quality control, our software was burned into a permanent state and *delivered* to users on tapes, CDs, DVDs, and diskettes. Of course, the popularity of the Web changed the nature of software delivery and the mechanics of releases have become much cheaper and easier. Ease of access combined with improved automation has drastically reduced the cost of a software change. Most organizations have the platforms, tools, and infrastructure in place to implement thousands of application releases within a single day.

## The Safety of Change

Speed of change gets a lot of attention in stories about microservice architecture, but the unspoken, yet equally important counterpart is change safety. After all, "speed kills" and in most software shops nobody wants to be responsible for breaking production. Every change is potentially a breaking change and a system optimized purely for speed is only realistic if the cost of breaking the system is near zero. Most development environments are optimized for release speed, enabling the software developer to make multiple changes in as short a time as possible. On the other hand, most production environments are optimized for safety, restricting the rate of change to those releases that carry the minimum risk of damage.

### At Scale

On top of everything else, today's software architect needs to be able to "think big" when building applications. As we heard earlier in this chapter, the microservices style is rooted in the idea of solving the problems that arise when software gets too big. To build at scale means to build software that can continue to work when demand grows beyond our initial expectations. Systems that can work at scale don't break when under pressure; instead they incorporate built-in mechanisms to increase capacity in a safe way. This added dimension requires a special perspective to building software and is essential to the microservices way.

### In Harmony

Your life is filled with decisions that impact speed and safety. Not just in the software domain, but in most of your everyday life; how fast are you willing to drive a car to get where you need to be on time? How does that maximum speed change when there is someone else in the car with you? Is that number different if one of your passengers is a child? The need to balance these ideals is something you were probably taught at a young age and you are probably familiar with the well-worn proverb, "haste makes waste."

We've found that all of the characteristics that we associate with microservice architecture (i.e., replaceability, decentralization, context-bound, message-based communication, modularity, etc.) have been employed by practitioners in pursuit of providing speed and safety at scale. This is the reason a universal characteristic- driven definition of microservices is unimportant—the real lessons are found in the practices successful companies have employed in pursuit of this balancing act.

# The Microservices Value Proposition

## Microservice Architecture Benefits

Why are organizations adopting microservices? What are the motivations and challenges? How can the leaders of these organizations tell that taking on the challenges of managing a collection of small, loosely coupled, independently deployable services is actually paying off for the company? What is the measure of success? Surveying the early adopters of microservices, we find that the answers to these questions vary quite a bit.

> *We can scale our operation independently, maintain unparalleled system availability, and introduce new services quickly without the need for massive reconfiguration.*
>
> —Werner Vogels, Chief Technology Officer, Amazon Web Services

By focusing on scalability and component independence, Amazon has been able to increase their speed of delivery while also improving the safety—in the form of scalability and availability—of their environment.

UK e-retailer Gilt is another early adopter of microservice architecture. Their Senior Vice President of Engineering, Adrian Trenaman, listed these resulting benefits in an InfoQ article:

- Lessens dependencies between teams, resulting in faster code to production
- Allows lots of initiatives to run in parallel
- Supports multiple technologies/languages/frameworks
- Enables graceful degradation of service
- Promotes ease of innovation through *disposable code*—it is easy to fail and move on

*Some services require high availability, but are low volume, and it's the opposite for other services. A microservice approach allows us to tune for both of these situations, whereas in a monolith it's all or nothing.*

—Beier Cai, Director of Software Development, Hootsuite

*With microservices, we have reduced the time it takes to deploy a useful piece of code and also reduced the frequency of deploying code that hasn't changed. Ultimately we strive to be flexible in our interpretation of microservice architecture, using its strengths where we can, but realizing that the business does not care about how we achieve results, only that we move quickly with good quality and flexible design.*

—Clay Garrard, Senior Manager of Cloud Services, Disney

*The monolithic code base we had was so massive and so broad no one knew all of it. People had developed their own areas of expertise and custodianship around submodules of the application.*

—Phil Calcado, former Director of Engineering, SoundCloud

## Deriving Business Value

Successful companies do not focus on increasing software delivery speed for its own sake. They do it because they are compelled by the speed of their business. Similarly, the level of safety implemented in an organization's software system should be tied to specific business objectives. Conversely, the safety measures must not get in the way of the speed unnecessarily. Balance is required.

For each organization, that balance will be a function of its delivery speed, the safety of its systems, and the growth of the organization's functional scope and scale. Each organization will have its own balance. A media company that aims to reach the widest possible audience for its content may place a much higher value on delivery speed than a retail bank whose compliance requirements mandate specific measures around safety. Nonetheless, in an increasingly digital economy, more companies are recognizing that software development needs to become one of their core competencies.

In this new business environment, where disruptive competitors can cross industry boundaries or start up from scratch seemingly overnight, fast software delivery is essential to staying ahead of the competition and achieving sustainable growth. In fact, each of the microservice architecture benefits that drive delivery speed contribute real business value:

- ***Agility*** allows organizations to deliver new products, functions, and features more quickly and pivot more easily if needed.
- ***Composability*** reduces development time and provides a compound benefit through reusability over time.
- ***Comprehensibility*** of the software system simplifies development planning, increases accuracy, and allows new resources to come up to speed more quickly.
- ***Independent deployability*** of components gets new features into production more quickly and provides more flexible options for piloting and prototyping.
- ***Organizational alignment*** of services to teams reduces ramp-up time and encourages teams to build more complex products and features iteratively.
- ***Polyglotism*** permits the use of the right tools for the right task, thus accelerating technology introduction and increasing solution options.

The safety-aligned benefits discussed earlier also provide particular business value:

- **Greater *efficiency*** in the software system reduces infrastructure costs and reduces the risk of capacity-related service outages.
- ***Independent manageability*** contributes to improved efficiency, and also reduces the need for scheduled downtime.
- ***Replaceability*** of components reduces the technical debt that can lead to aging, unreliable environments.
- **Stronger *resilience*** and higher *availability* ensure a good customer experience.
- **Better *runtime scalability*** allows the software system to grow or shrink with the business.
- **Improved *testability*** allows the business to mitigate implementation risks.

## Defining a Goal-Oriented, Layered Approach

In spite of the fact that microservice architecture was originally a reaction to the limitations of monolithic applications, there is a fair amount of guidance in the industry that says new applications should still be built as monoliths first. The thinking is that only through the creation and ownership of a monolith can the right service boundaries be identified. This path is certainly well trodden, given that early microservice adopters generally went through the process of unbundling their own monolithic applications. The "monolith first" approach also appears to follow Gall's Law, which states that, "A complex system that works is invariably found to have evolved from a simple system that worked." However, is a monolithic application architecture the only simple system starting point? Is it possible to start simple with a microservice architecture?

In fact, the complexity of a software system is driven by its scale. Scale comes in the form of functional scope, operational magnitude, and change frequency. The first companies to use microservice architecture made the switch from monolithic applications once they passed a certain scale threshold. With the benefit of hindsight, and with an analysis of the common goals and benefits of microservice architecture, we can map out a set of layered characteristics to consider when adopting microservice architecture.

## Modularized Microservice Architecture

At its most basic level, microservice architecture is about breaking up an application or system into smaller parts. A software system that is modularized arbitrarily will obviously have some limitations, but there is still a potential upside. Network- accessible modularization facilitates automation and provides a concrete means of abstraction. Beyond that, some of the microservice architecture benefits discussed earlier already apply at this base layer.

To help software delivery speed, modularized services are independently deployable. It is also possible to take a polyglot approach to tool and platform selection for individual services, regardless of what the service boundaries are. With respect to safety, services can be managed individually at this layer. Also, the abstracted service interfaces allow for more granular testing.

This is the most technologically focused microservice architecture layer. In order to address this layer and achieve its associated benefits, you must establish a foundation for your microservice architecture.

## Cohesive Microservice Architecture

The next layer to consider in your microservice architecture is the cohesion of services. In order to have a cohesive microservice architecture, it must already be modularized. Achieving service cohesion comes from defining the right service boundaries and analyzing the semantics of the system. The concept of domains is useful at this layer, whether they are business-oriented or defined by some other axis.

A cohesive microservice architecture can enable software speed by aligning the system's services with the supporting organization's structure. It can also yield composable services that are permitted to change at the pace the business dictates, rather than through unnecessary dependencies. Reducing the dependencies of a system featuring cohesive services also facilitates replaceability of services. Moreover, service cohesion lessens the need for highly orchestrated message exchanges between components, thereby creating a more efficient system.

It takes a synthesized view of business, technology, and organizational considerations to build a cohesive system.

## Systematized Microservice Architecture

The final and most advanced layer to consider in a microservice architecture is its system elements. After breaking the system into pieces through modularization, and addressing the services' contents through cohesion, it is time to examine the interrelationships between the services. This is where the greatest level of complexity in the system needs to be addressed, but also where the biggest and longest-lasting benefits can be realized.

There are two ways speed of delivery is impacted in a systematized microservice architecture. Although a single service may be understandable even in a modularized microservice architecture, the overall software system is only comprehensible when the connectivity between services is known. Also, agility is only possible when the impacts of changes on the whole system can be identified and assessed rapidly. This applies on the safety side as well, where runtime scalability is concerned. Lastly, although individual components may be isolated and made resilient in a modularized or cohesive microservice architecture, the system availability is not assured unless the interdependencies of the components are understood.

## Maturity Model for Microservice Architecture Goals and Benefits

These layered characteristics—modularized, cohesive, and systematized—help to define a maturity model that serves a number of purposes. First, it classifies the benefits according to phase and goal (speed or safety) as discussed previously. Secondly, it illustrates the relative impact and priority of benefits as scale and complexity increase. Lastly, it shows the activities needed to address each architectural phase. This maturity model is depicted in Figure 2-1.

Note that an organization's microservice architecture can be at different phases for different goals. Many companies have become systematized in their approach to safety—through automation and other operational considerations—without seeking the speed-aligned system-level benefits. The point of this model is not for every organization to achieve systematized actualization with their microservice architecture. Rather, the model is meant to clarify goals and benefits in order to help organizations focus their microservice strategies and prepare for what could come next.

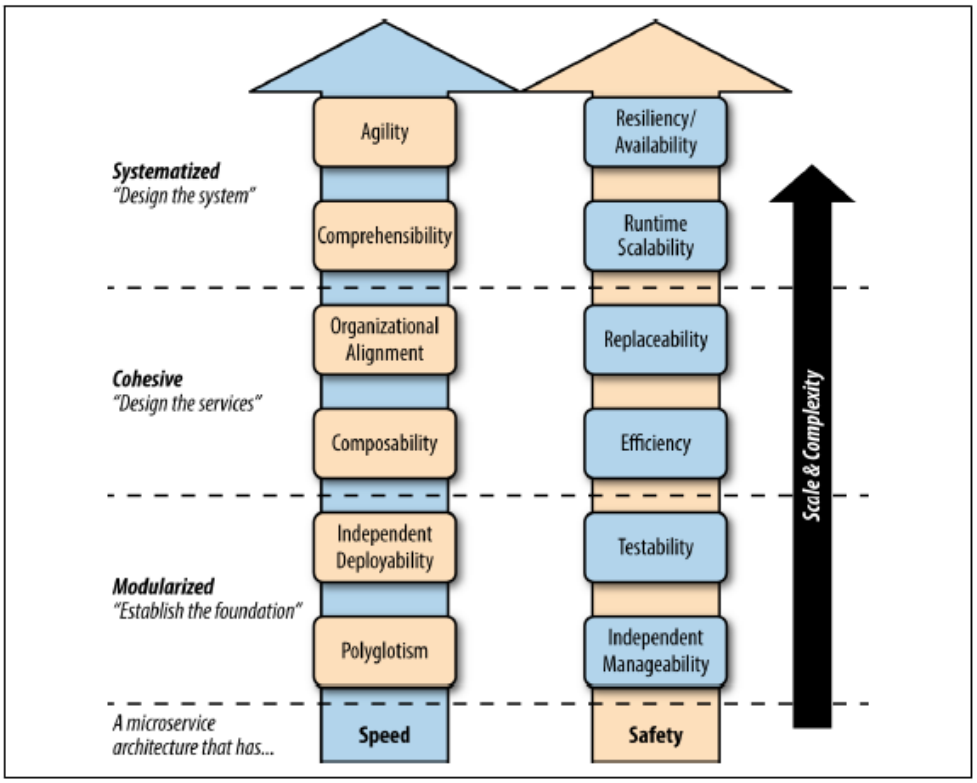


*Figure 2-1. A maturity model for microservice architecture goals and benefits*

## Applying the Goal-Oriented, Layered Approach

Now we have a good understanding of how a microservice architecture can bring value to an organization, and a model for understanding what characteristics can bring what goals and benefits at what stage of adoption. But what about your organization? What are your business goals? What problems do you need to solve? It is a common misstep to start down the microservices path for its own sake without thinking about the specific benefits you are targeting. In other cases, some organizations aim for a high-level goal and then only implement one aspect of microservices while ignoring its founding conditions. For example, an organization with a high- level divide between development and operations—an organizational red flag-might execute a containerization strategy on their existing applications and then wonder why they didn't speed up their software development sufficiently. A broad perspective is needed.

To begin with, define the high-level business objectives you want to accomplish, and then weigh these against the dual goals of speed and safety. Within that context, consider the distinct benefits you are targeting. You can then use the maturity model to determine complexity of the goal, and identify the best approach to achieve it.

# Designing Microservice Systems

## The Systems Approach to Microservices

A microservices system encompasses all of the things about your organization that are related to the application it produces. This means that the structure of your organization, the people who work there, the way they work, and the outputs they produce are all important system factors. Equally important are runtime architectural elements such as service coordination, error handling, and operational practices. In addition to the wide breadth of subject matter that you need to consider, there is the additional challenge that all of these elements are interconnected—a change to one part of the system can have an unforeseen impact on another part. For example, a change to the size of an implementation team can have a profound impact on the work that the implementation team produces.

The models mathematicians develop to study complex systems allow them to more accurately understand and predict the behavior of a system. But this is a field in its infancy and the models they produce tend to be very complicated. We don't expect you to understand the mathematics of complexity, nor do we think it will be particularly helpful in creating better microservice applications. But we do believe that a model-based approach can help all of us conceptualize our system of study and will make it easier for us talk about the parts of the system.

With that in mind, Figure 3-1 depicts a microservice design model comprised of five parts: Service, Solution, Process and Tools, Organization, and Culture.

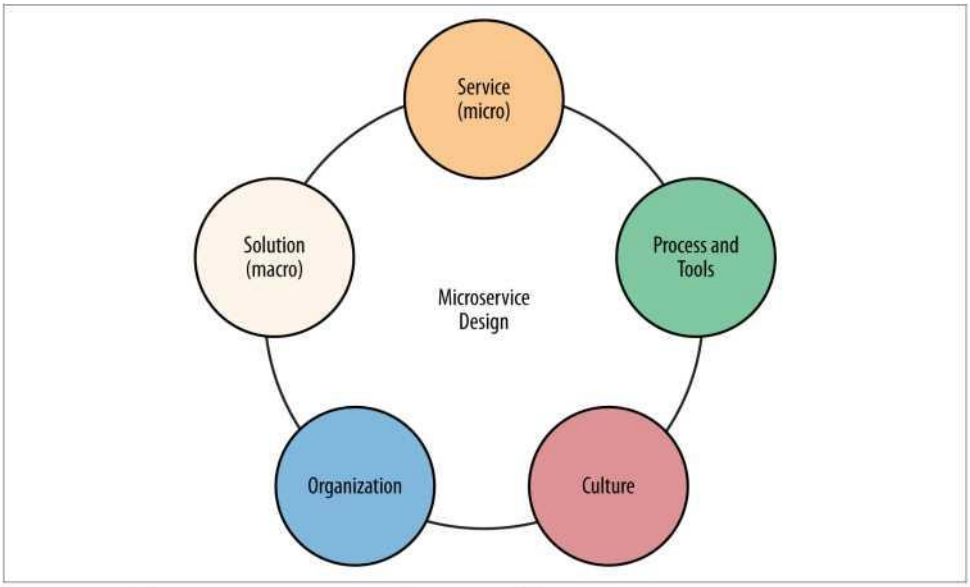


*Figure 3-1. The microservice system design model*

**Service**

Implementing well-designed microservices and APIs are essential to a microservice system. In a microservice system, the services form the atomic building blocks from which the entire organism is built. If you can get the design, scope, and granularity of your service just right you'll be able to induce complex behavior from a set of components that are deceptively simple.

**Solution**

A solution architecture is distinct from the individual service design elements because it represents a macro view of our solution. When designing a particular microservice your decisions are bounded by the need to produce a single output—the service itself. Conversely, when designing a solution architecture your decisions are

bounded by the need to coordinate all the inputs and outputs of multiple services. This macro-level view of the system allows the designer to induce more desirable system behavior. For example, a solution architecture that provides discovery, safety, and routing features can reduce the complexity of individual services.

**Process and Tools**

Your microservice system is not just a byproduct of the service components that handle messages at runtime. The system behavior is also a result of the processes and tools that workers in the system use to do their job. In the microservice's system, this usually includes tooling and processes related to software development, code deployment, maintenance, and product management.

Choosing the right processes and tools is an important factor in producing good microservice system behavior. For example, adopting standardized processes like DevOps and Agile or tools like Docker containers can increase the changeability of your system.

**Organization**

How we work is often a product of who we work with and how we communicate. From a microservice system perspective, organizational design includes the structure, direction of authority, granularity, and composition of teams. Many of the companies that have had success with microservice architecture point to their organizational design as a key ingredient. But organizational design is incredibly context-sensitive and you may find yourself in a terrible situation if you try to model your 500+ employee enterprise structure after a 10-person startup (and vice versa).

A good microservice system designer understands the implications of changing these organizational properties and knows that good service design is a byproduct of good organizational design.

**Culture**

Of all the microservice system domains, culture is perhaps the most intangible yet may also be the most important. We can broadly define culture as a set of values, beliefs, or ideals that are shared by all of the workers within an organization. Your organization's culture is important because it shapes all of the atomic decisions that people within the system will make. This large scope of influence is what makes it such a powerful tool in your system design endeavor.

Much like organizational design, culture is a context-sensitive feature of your system. What works in Japan may not work in the United States and what works in a large insurance firm may not work at an ecommerce company.

As important as it is, the culture of an organization is incredibly difficult to measure. Formal methods of surveying and modeling exist, but many business and technology leaders evaluate the culture of their teams in a more instinctual way. You can get a sense of the culture of your organization through your daily interactions with team members, team products, and the customers they cater to.

## Embracing Change

Time is an essential element of a microservice system and failing to account for it is a grave mistake. All of the decisions you make about the organization, culture, processes, services, and solutions should be rooted in the notion that change is inevitable. You cannot afford to be purely deterministic in your system design; instead, you should design adaptability into the system as a feature.

There is good reason for taking this perspective: first, trying to determine what the end state of your organization and solution design should look like is a near impossible task. Second, it is unlikely that the context

in which you made your design decisions will stay the same. Changes in requirements, markets, and technology all have a way of making today's good decisions obsolete very quickly.

A good microservice designer understands the need for adaptability and endeavors to continually improve the system instead of working to simply produce a solution.

## The Holistic System

When put together all of these design elements form the microservices system. They are interconnected and a change to one element can have a meaningful and sometimes unpredictable impact on other elements. The system changes over time and is unpredictable. It produces behavior that is greater than the behavior of its individual components. It adapts to changing contexts, environments, and stimuli.

In short, the microservices system is complex and teasing desirable behaviors and outcomes from that system isn't an easy task. But some organizations have had enormous success in doing so.

## Standardization and Coordination

Almost all of us work in organizations that operate within constraints. These constraints arise because the wrong type of system behavior can be harmful to the organization, even resulting in the organization failing as a result of particularly bad behavior. For example, a banking technology system that makes it easy to steal someone else's money or a tax system that fails to protect its users' private information are unacceptable.

With the cost of unwanted system behavior so high, it's no wonder that so many architects and designers do their best to control system behavior. In practice, the system designer decides that there is some behavior or expectation that must be universally applied to the actors within the system. Policies, governance, and audits are all introduced as a way of policing the behavior of the system and ensuring that the actors conform. In other words, some parts of the system are standardized.

But true control of this type of complex system is an illusion. You have as much chance of guaranteeing that your banking system will be perfectly secure as a farmer does of guaranteeing that his crops will always grow. No matter how many rules, checks, and governance methods you apply you are always at the mercy of actors in a system that can make poor decisions.

However, control of the system comes at a steep price. Standardization is the enemy of adaptability and if you standardize too many parts of your system you risk creating something that is costly and difficult to change.

**Standardizing process**

By standardizing the way that people work and the tools they use, you can influence the behavior in a more predictable way. For example, standardizing a deployment process that reduces the time for component deployment may improve the overall changeability of the system as the cost of new deployments decreases.

Standardizing how we work has broad-reaching implications on the type of work we can produce, the kind of people we hire, and the culture of an organization. The Agile methodology is a great example of process standardization. Agile institutionalizes the concept that change should be introduced in small measurable increments that allow the organization to handle change easier. One observable system impact for Agile teams is that the output they produce begins to change. Software releases become smaller and measurability becomes a feature of the product they output. There are also usually follow-on effects to culture and organizational design.

In addition to process standardization, most companies employ some form of tool standardization as well.

In fact, many large organizations have departments whose sole purpose is to define the types of tools their workers are allowed to utilize. For example, some firms forbid the use of open source software and limit their teams to the use of centrally approved software, procured by a specialist team.

We can define a team as a group of workers who take a set of inputs and transform them into one or more outputs. Output standardization is way of setting a universal standard for what that output should look like. For example, in an assembly line the output of the line workers is standardized—everyone on the line must produce exactly the same result. Any deviation from the standard output is considered a failure.

In the microservices context, output standardization often means developing some standards for the APIs that expose the services. For example, you might decide that all the organization's services should have an HTTP interface or that all services should be capable of subscribing to and emitting events. Some organizations even standardize how the interfaces should be designed in an effort to improve the usability, changeability, and overall experience of using the service.

**Standardizing people**

You can also decide to standardize the types of people that do the work within your organization. For example, you could introduce a minimum skill requirement for anyone who wants to work on a microservice team. In fact, many of the companies that have shared microservice stories point to the skill level of their people as a primary characteristic of their success.

Standardizing skills or talent can be an effective way of introducing more autonomy into your microservices system. When the people who are implementing the services are more skilled they have a better chance of making decisions that will create the system behavior you want.

All organizations have some level of minimum skill and experience level for their workers, but organizations that prioritize skill standardization often set very high specialist requirements in order to reap system benefits. If only the best and brightest are good enough to work within your system, be prepared to pay a high cost to maintain that standard.

Standardizing helps you exert influence over your system, but you don't have to choose just one of these standards to utilize. But keep in mind that while they aren't mutually exclusive, the introduction of different modes of standardization can create unintended consequences in other parts of the system.

But we may find that constraining the types of APIs our people are allowed to produce limits the types of tools they can use to create them. It might be the case that the development tool we want everyone to use doesn't support the interface description language we have already chosen. In other words, the decision to standardize the team's output has had unintended consequences on the team's work process. This happens because standardization is an attempt to remove uncertainty from our system, but comes at the cost of reducing innovation and changeability.

The benefit of standardization is a reduction in the set of all possible outcomes. It gives us a way to shape the system by setting constraints and boundaries for the actions that people within the system can take. But this benefit comes at a cost. Standardization also constrains the autonomy of individual decision-makers.

The challenge for designers is to introduce just enough standardization to achieve the best emergent system outcome, while also employing standards and constraints that complement each other.

# A Microservices Design Process

**Figure 3-2** illustrates a framework for a design process that you can use in your own microservice system designs. In practice, it is likely that you'll need to customize the process to fit within your own unique constraints and context.

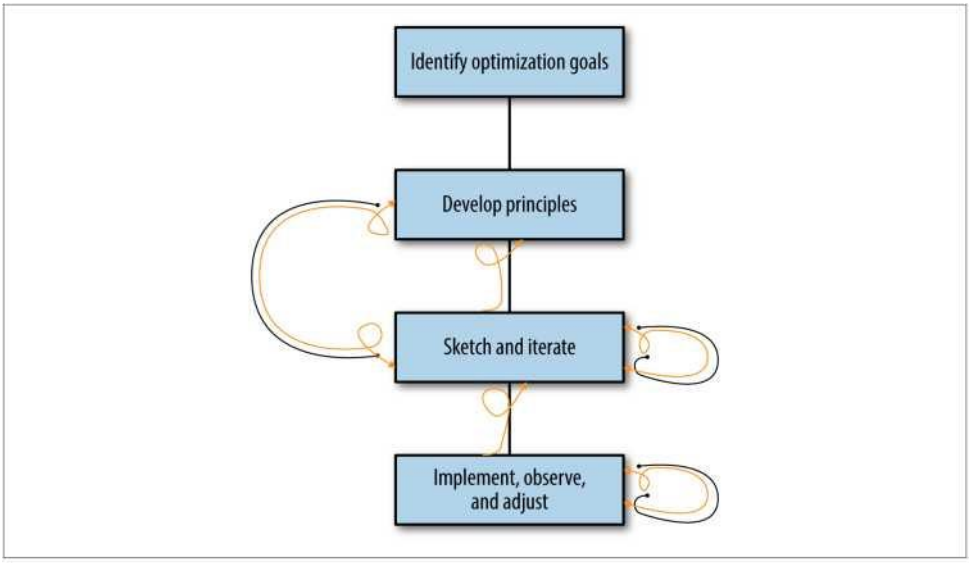


*Figure 3-2. Microservice system design process*

### 1-Set Optimization Goals

The behavior of your microservice system is "correct" when it helps you achieve your goals. There isn't a set of optimization goals that perfectly apply to all organizations, so one of your first tasks will be to identify the goals that make sense for your particular situation. The choice you make here is important—every decision in the design process after this is a trade-off made in favor of the optimization goal.

Note that optimization doesn't mean that other system qualities are undesirable. In fact, it is extremely likely that you will initially list many desirable outcomes for the system you create. But as you go through the system design process you will find that it is difficult to pull your system into many directions at the same time. A smaller set of optimization goals is easier to design for.

For example, a financial information system might be optimized for reliability and security above all other factors. That doesn't mean that changeability, usability, and other system qualities are unimportant—it simply means that the designers will always make decisions that favor security and reliability above all other things.

### 2-Development Principles

Underpinning a system optimization goal is a set of principles. Principles outline the general policies, constraints, and ideals that should be applied universally to the actors within the system to guide decision-making and behavior. The best designed principles are simply stated, easy to understand, and have a profound impact on the system they act upon.

### 3-Sketch the System Design

If you find yourself building the application in a *greenfield* environment with no existing organization or solution architecture in place, it is important that you establish a good starting point for your system design. You won't be able to create the perfect system on your first try and you aren't likely to have the time or information to do that anyway. Instead, a good approach is to sketch the important parts of your system design for the purposes of evaluation and iteration.

How you do this is entirely up to you. There is a wealth of modeling and communication tools available to conceptualize organizational and solution architectures; choose the ones that work well for you. But the value of this step in the design process is to serialize some of the abstract concepts from your head into a tangible form that can be evaluated. The goal of a sketching exercise is to continually improve the design until you are comfortable moving forward.

The goal is to sketch out the core parts of your system, including organizational structure (how big are the teams? what is the direction of authority? who is on the team?), the solution architecture (how are services organized? what infrastructure must be in place?), the service design (what outputs? how big?), and the processes and tools (how do services get deployed? what tools are necessary?). You should evaluate these decisions against the goals and principles you've outlined earlier. Will your system foster those goals? Do the principles make sense? Do the principles need to change? Does the system design need to change?

Sketching is powerful when the risk of starting over is small. Good sketches are easy to make and easy to destroy, so avoid modeling your system in a way that requires a heavy investment of time or effort. The more effort it takes to sketch your system the less likely you are to throw it away. At this early stage of system design, change should be cheap.

**4-Implement, Observe, and Adjust**

Bad designers make assumptions about how a system works, apply changes in the hope that it will produce desired behavior, and call it a day. Good designers make small system changes, assess the impact of those changes, and continually prod the system behavior toward a desired outcome. But a good design process is predicated on your ability to get feedback from the system you are designing. This is actually much more difficult than it sounds—the impact of a change to one small part of the system may result in a ripple of changes that impact other parts of your system with low visibility.

The perfect microservice system provides perfect information about all aspects of the system across all the domains of culture, organization, solution architecture, services, and process. Of course, this is unrealistic. It is more realistic to gain *essential* visibility into our system by identifying a few key measurements that give us the most valuable information about system behavior. In organizational design, this type of metric is known as a key performance indicator (KPI). The challenge for the microservice designer is to identify the right ones.

Gathering information about your system by identifying KPIs is useful, but being able to utilize those metrics to predict future behavior is incredibly valuable. One of the challenges that all system designers face is the uncertainty about the future. With perfect information about how our system might need to change we could build boundaries in exactly the right places and make perfect decisions about the size of our services and teams.

So, in order to design a microservice system that is dynamic you'll need to identify the right KPIs, be able to interpret the data, and make small, cheap changes to the system that can guide you back on the right course. This is only possible if the right organization, culture, processes, and system architecture are in place to make it cheap and easy to do so.

Rather than trying to predict the future, a good microservices designer examines the current state and makes small, measurable changes to the system. This is a bit like taking a wrong turn on a long road trip—if you don't know that you've made a mistake you might not find out you're going the wrong way until it is too late to turn back. But if you have a navigator with you, they may inform you right away and you can take corrective action.

# Establishing a Foundation

## Goals and Principles

Regardless of the software architecture style you employ, it is important to have some overall *goals* and *principles* to help inform your design choices and guide the implementation efforts. This is especially true in companies where a higher degree of autonomy is provided to developer teams. The more autonomy you allow, the more guidance and context you need to provide to those teams.

### Goals for the Microservices Way

It is a good idea to have a set of high-level goals to use as a guide when making decisions about *what* to do and *how* to go about doing it. We've already introduced our ultimate goal in building applications in the microservices way: *finding the right harmony of speed and safety at scale.* This overarching goal gives you a destination to aim for and given enough time, iterations, and persistence, will allow you to build a system that hits the right notes for your own organization.

There is of course a glaring problem with this strategy—it might take a very long time for you to find that perfect harmony of speed and safety at scale if you are starting from scratch. But thanks to the efforts of generations of technologists we have access to proven methods for boosting both speed and safety. So, you don't need to reinvent established software development practices. Instead, you can experiment with the parameters of those practices. Here are the four goals to consider:

1. **Reduce Cost:** Will this reduce overall cost of designing, implementing, and maintaining IT services?
2. **Increase Release Speed:** Will this increase the speed at which my team can get from idea to deployment of services?
3. **Improve Resilience:** Will this improve the resilience of our service network?
4. **Enable Visibility:** Does this help me better *see* what is going on in my service network?

### Reduce cost

The ability to reduce the cost of designing, implementing, and deploying services allows you more flexibility when deciding whether to create a service at all. For example, if the work of creating a new service component includes three months of design and review, six months of coding and testing, and two more weeks to get into production, that's a very high cost—one that you would likely think very carefully about before starting. However, if creating a new service component takes only a matter of a few weeks, you might be more likely to build the component and see if it can help solve an important problem. Reducing costs can increase your agility because it makes it more likely that you'll experiment with new ideas.

In the operations world, reducing costs was achieved by virtualizing hardware. By making the cost of a "server" almost trivial, it makes it more likely that you can spin up a bunch of servers in order to experiment with load testing, how a component will behave when interacting with others, and so on. For microservices, this means coming up with ways to reduce the cost of coding and connecting services together.

### Increase release speed

Increasing the speed of the "from design to deploy" cycle is another common goal. A more useful way to view this goal is that you want to *shorten* the time between idea and deployment. Sometimes, you don't need to "go faster," you just need to take a shortcut. When you can get from idea to running example quickly, you have the chance to get feedback early, to learn from mistakes, and iterate on the design more often before a final

production release. Like the goal of reducing costs, the ability to increase speed can also lower the risk for attempting new product ideas or even things as simple as new, more efficient data-handling routines.

One place where you can increase speed is in the deployment process. By automating important elements of the deployment cycle, you can speed up the whole process of getting services into production. Some of the companies we talked with for this book spend a great deal of time building a highly effective deployment pipeline for their organization.

### Improve resilience

No matter the speed or cost of solutions, it is also important to build systems that can "stand up" to unexpected failures. In other words, systems that don't crash, even when errors occur. When you have an overall system approach (not just focused on a single component or solution) you can aim for creating resilient systems. This goal is often much more reasonable than trying to create a single component that is totally free of bugs or errors. In fact, creating a component that will have zero bugs is often impossible and sometimes simply not worth the time and money it takes to try.

One of the ways DevOps practices has focused on improving resilience is through the use of automated testing. By making testing part of the build process, the tests are constantly run against checked-in code, which increases the chances of finding errors in the code. This covers the code, but not the errors that could occur at runtime.

### Enable visibility

Another key goal should be to enable runtime visibility. In other words, improve the ability of stakeholders to see and understand what is going on in the system. There is a good set of tools for enabling visibility during the coding process. We often get reports on the coding backlog, how many builds were created, the number of bugs in the system versus bug completed, and so on. But we also need visibility into the runtime system.

## Platforms

Along with a set of general goals and concrete principles, you'll need tangible tools to make them real—a platform with which to make your microservice environment a reality. From a microservice architecture perspective, good platforms increase the harmonic balance between speed and safety of change at scale. We typically think about speed and safety as opposing properties that require a trade-off to be made but the right tooling and automation give you an opportunity to cheat the trade-off.

With a platform we pass from the conceptual world to the actual world. The good news is that there are many examples of companies establishing—and even sharing— their microservice platforms. The challenge is that it seems every company is doing this their own way, which presents some choices to anyone who wants to build their own microservice environment. Do you just select one of the existing OSS platforms? Do you try to purchase one? Build one from scratch?

It would be a mistake to just select one of the popular company's platforms and adopt it without careful consideration. Does this company provide the same types of services that mine does? Does this company optimize for the same things that mine will? Do we have similar staffing and training environments? Are our target customers similar (priorities, skills, desired outcomes, etc.)?

Instead of focusing on a single existing company's platform, we'll look at a general model for microservice platforms.

**Shared Capabilities**

It's common in large enterprises to create a shared set of services for everyone to use. These are typically centered around the common infrastructure for the organization. For example, anything that deals with hardware (actual or virtual) falls into this category. Common database technologies (MySQL, Cassandra, etc.) and other software-implemented infrastructure is another example of shared services.

Shared capabilities are platform services that all teams use. These are standardized things like container technology, policy enforcement, service orchestration/interop, and data storage services. Even in large organizations it makes sense to narrow the choices for these elements in order to limit complexity and gain cost efficiencies. Essentially, these are all services that are provided to every team in the organization.

While shared capabilities offer potential cost savings they are ultimately rooted in the microservices goal of change safety. Organizations that highly value safety of changes are more likely to deploy centralized shared capabilities that can offer consistent, predictable results. On the other hand, organizations that desire speed at all costs are likely to avoid shared components as much as possible as it has the potential to inhibit the speed at which decentralized change can be introduced. In these speedcentric companies, capability reuse is less important than speed of delivery.

The following is a quick rundown of what shared services platforms usually provide:

***Hardware services***

All organizations deal with the work of deploying OS- and protocol-level software infrastructure. In some companies there is a team of people who are charged with accepting shipments of hardware (e.g., 1-U servers), populating those machines with a baseline OS and common software for monitoring, health checks, etc., and then placing that completed unit into a rack in the "server room" ready for use by application teams.

Another approach is to virtualize the OS and baseline software package as a virtual machine (VM). VMs like Amazon's EC2 and VMWare's hypervisors are examples of this technology. VMs make it possible to automate most of the work of populating a "new machine" and placing it into production.

***Code management, testing, and deployment***

Once you have running servers as targets, you can deploy application code to them. That's where code management (e.g., source control and review), testing, and (eventually) deployment come in. There are quite a few options for all these services and some of them are tied to the developer environment, especially testing.

***Data stores***

There are many data storage platforms available today, from classic SQL-based systems to JSON document stores on through graph-style databases such as Riak and Neo4J. It is usually not effective for large organizations to support all possible storage technologies. Even today, some organizations struggle with providing proper support for the many storage implementations they have onsite. It makes sense for your organization to focus on a select few storage platforms and make those available to all your developer teams.

***Service orchestration***

The technology behind service orchestration or service interoperability is another one that is commonly shared across all teams. There is a wide range of options here. Many of the flagship microservice companies (e.g., Netflix and Amazon) wrote their own orchestration platforms.

***Security and identity***

Platform-level security is another shared service. This often happens at the perimeter via gateways and proxies. Again, some companies have written their own frameworks for this; Netflix's Security Monkey is an example. There are also a number of security products available.

***Architectural policy***

Finally, along with shared security, sometimes additional policy services are shared. These are services that are used to enforce company-specific patterns or models—often at runtime through a kind of inspection or even invasive testing.

One example of policy enforcement at runtime is Netflix's "Simian Army"—a set of services designed to purposely cause problems on the network (simulate missing packets, unresponsive services, and so on) to test the resiliency of the system.

**Local Capabilities**

Local capabilities are the ones that are selected and maintained at the team or group level. One of the primary goals of the local capabilities set is to help teams become more self-sufficient. This allows them to work at their own pace and reduces the number of blocking factors a team will encounter while they work to accomplish their goals. Also, it is common to allow teams to make their own determination on which developer tools, frameworks, support libraries, config utilities, etc., are best for their assigned job. Sometimes these tools are selected from a curated set of "approved" products. Sometimes these tools are created in-house (even by the same team). Often they are open source, community projects.

***tooling***

A key local capability is the power to automate the process of rolling out, monitoring, and managing VMs and deployment packages. Netflix created Asgard and Aminator for this. A popular open source tool for this is Jenkins.

***Runtime configuration***

A pattern found in many organizations using microservices is the ability roll out new features in a series of controlled stages. This allows teams to assess a new release's impact on the rest of the system (are we running slower?, is there an unexpected bug in the release?, etc.). Twitter's Decider configuration tool is used by a number of companies for this including Pinterest, Gilt, and Twitter. This tool lets teams use configuration files to route traffic from the "current" set of services to the "newly deployed" set of services in a controlled way.

***Service discovery***

There are a handful of popular service discovery tools including Apache Zookeeper, CoreOS' etcd, and HashiCorp's Consul. These tools make it possible to build and release services that, upon install, register themselves with a central source, and then allow other services to "discover" the exact address/location of each other at runtime. This ability to abstract the exact location of services allows various teams to make changes to the location of their own service deployments without fear of breaking some other team's existing running code.

***Request routing***

Once you have machines and deployments up and running and discovering services, the actual process of handling requests begins. All systems use some kind of request-routing technology to convert external calls

(usually over HTTP, Web- Sockets, etc.) into internal code execution (e.g., a function somewhere in the codebase). The simplest form of request routing is just exposing HTTP endpoints from a web server like Apache, Microsoft IIS, NodeJS, and others.

***System observability***

A big challenge in rapidly changing, distributed environments is getting a view of the running instances—seeing their failure/success rates, spotting bottlenecks in the system, etc. There are quite a few tools for this.

## Culture

Along with establishing goals and principles and arming your organization with the right tools for managing platform, code, and runtime environments, there is another critical foundation element to consider—your company culture. Culture is important because it not only sets the tone for the way people behave inside an organization, but it also affects the output of the group. The code your team produces is the *result* of the culture.

We'll look at three aspects of culture that you should consider as a foundation for your microservice efforts:

***Communication***

Research shows that the way your teams communicate (both to each other and to other teams) has a direct measurable effect on the quality of your software.

***Team alignment***

The size of your teams also has an effect on output. More people on the team means essentially more overhead.

***Fostering innovation***

Innovation can be disruptive to an organization but it is essential to growth and long-term success.