Rollbar

eBook

Maintaining Software Quality with Microservices

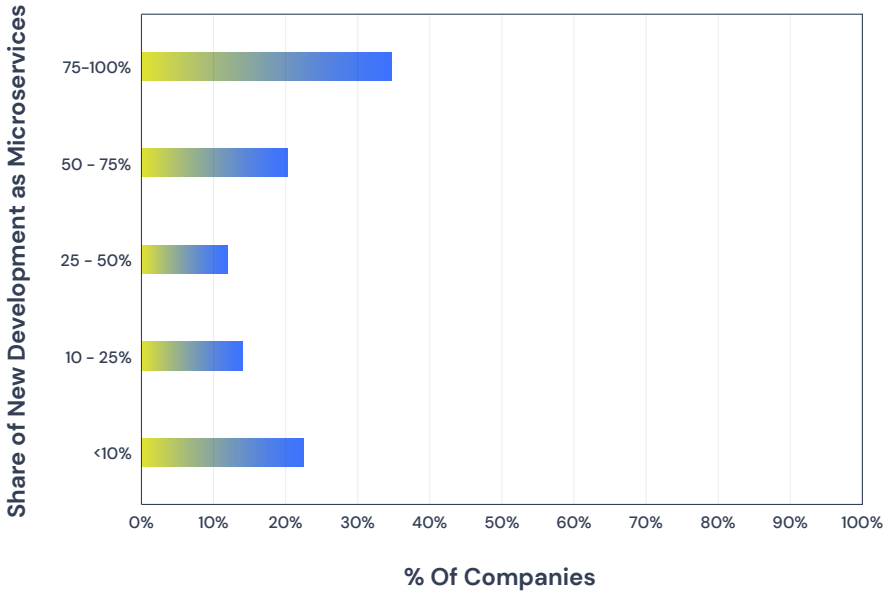
Table of Contents

Introduction

Microservices are here to stay

As observed in the State of Microservices Maturity report published by O'Reilly in 2018, microservices have moved "from fad to trend," with a majority of survey respondents using microservices for over half their new development. Other reports show similar adoption.

This rise in the popularity of microservices didn't happen in isolation. The transition to cloud infrastructure, the availability of new automation tools to continuously integrate and deploy software (CI/CD), and the growth of DevOps culture have all contributed to the trend. As a result, the time to bring new features and applications to market has dramatically decreased.



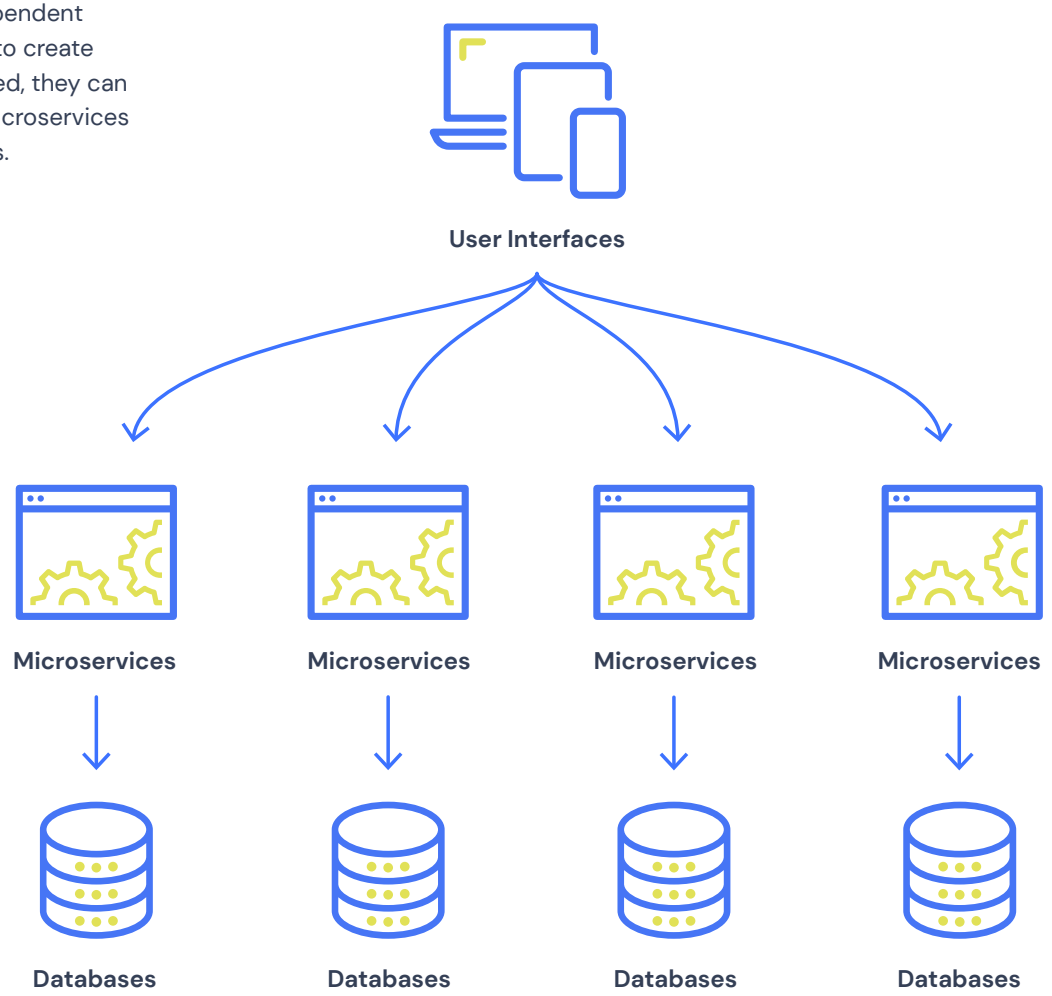
So it's not surprising to find enterprises with legacy applications moving their systems to a microservices architecture. There are many examples of companies successfully making the change, but these examples shouldn't suggest that the move is easy. Microservices are complex, and maintaining quality during the move is especially difficult. A great deal of success depends on whether or not the enterprise is prepared to make the necessary changes, both organizational and technical, to maintain quality.

In this eBook, we'll take a look at the strategies for moving to a microservices-based architecture, and how to maintain quality during that transition. We'll start with a background of microservices and why an enterprise might make the move, then we'll identify the challenges in making a successful transition, and then finally we'll talk about best practices for achieving software quality during this process.

1 Background and Motivation

What are Microservices?

Microservices are a style of architecting software into small, independent services that are autonomous, decoupled, and yet work together to create a variety of larger applications. Because the services are decoupled, they can easily be deployed and upgraded independently of each other. Microservices are typically designed as individual, reusable business capabilities.



Motivation for Microservices

To identify why microservices have become so popular, it's easiest to compare them to their foil — the monolithic application. A monolithic application has all of its services — UI, business logic, data access logic, etc. — combined into a single, large, "monolithic" system.

It's a well-known challenge to modify and deploy these large monolithic systems smoothly and quickly.

- **Their codebases do not have distinct and independent interfaces between their components or modules.**
- **When errors occur, it can take significant time to isolate the specific problem. While debugging in microservices presents its own challenges, understanding where in the code the error is occurring can be more difficult with monoliths.**
- **Upgrading one part of the system requires deploying the entire monolith.**
- **Fixing one component or module can impact others in unexpected ways, which requires expensive, additional testing of the entire application to unravel.**

Moreover, a monolithic system must scale as a single unit regardless of which service or function needs more resources. As a result, resources end up being allocated unnecessarily for the whole application when the needs for those resources may arise infrequently. This poor resource utilization undermines any potential cost savings from deploying monolithic applications to the cloud.

In contrast, microservices offer the following advantages:

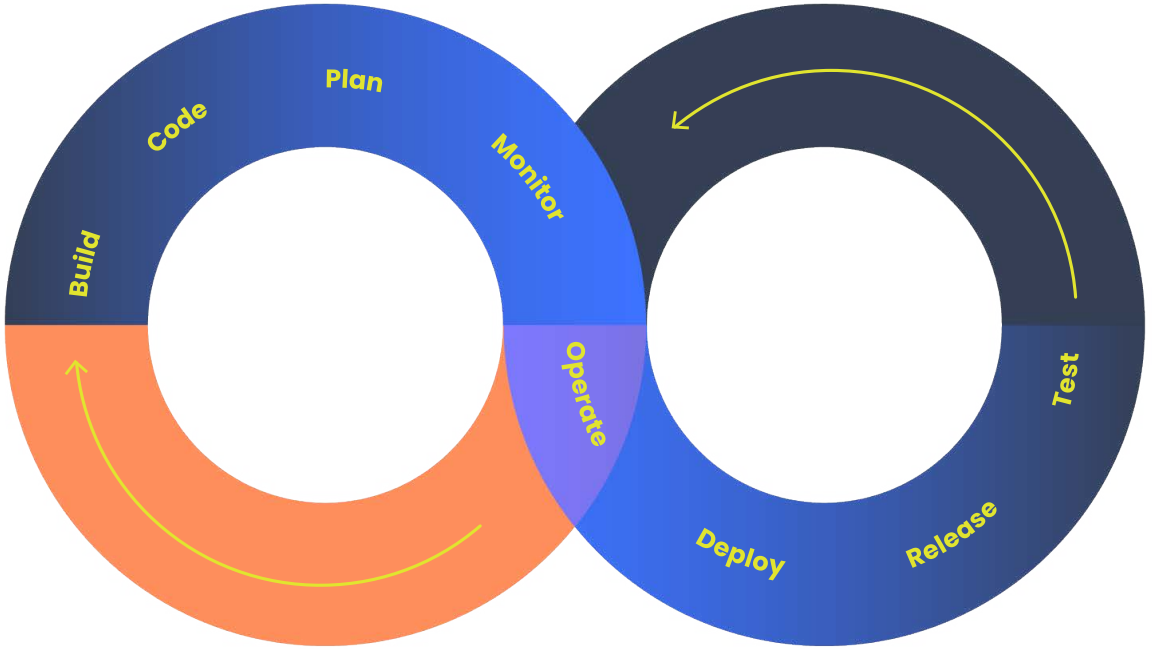
- **The code for a microservice is considerably smaller and can be tested and deployed independently of other services.**
- **Each microservice can also be scaled independently, allowing for targeted scaling.**
- **Code changes for a microservice are isolated to the service in question, making changes much less risky and costly.**
- **Problems are isolated to a specific microservice and don't cascade through the entire application. At worst, there is a partial loss of functionality rather than total application failure while the errant microservice is rolled back.**
- **Language, development framework, and database choices for a microservice are flexible and depend on the team assigned to that service, allowing them to choose the best technology and tools for their task.**

Some Complementary Trends

To fully understand the microservices environment, we also need to understand related trends that complement microservice-based architectures. Let's look at three of those trends — DevOps, CI/CD, and containers.

DevOps is the marriage of two previously decoupled activities: Dev for developing and testing the software and Ops for operations, or deploying the software. DevOps is a model for software development and deployment that empowers small groups of developers, testers, and operations personnel to work as a single unit, moving the code from creation to testing to deployment and support. In short, with DevOps, the same team is responsible for the microservice during its entire lifecycle.

This is complemented by our second trend — Continuous Integration/ Continuous Delivery (CI/CD). Automated tools such as Jenkins enable a CI/ CD strategy to frequently merge, build, and deploy code. This decreases time to market by creating shorter, incremental cycles of commit, build, test, and deploy. With CI/CD, it's now common to see deployment frequencies of many times per day.



The third trend is containers. With the move to cloud computing, two deployment techniques have become popular: virtual machines (VMs) and containers. Both isolate an enterprise's applications from others on the shared cloud platform, but they differ in significant ways.

VMs create individual silos that contain an entire software stack — including the OS — into which a customer can place anything, including monolithic applications. As demand on the application fluctuates, the number of VMs can spin up or down. However, adding one of these heavy-weight VMs takes time, so most teams buy more VMs than they need in anticipation of demand spikes. As a result, much of the purchased computing capacity remains unused, which is exactly what a cloud-based solution is designed to prevent.

Containers such as Docker, on the other hand, maintain all the benefits of VMs, while greatly improving resource utilization under varying loads. Containers hold only the microservice code and any libraries needed to run it. Instead of including the OS, containers rely on the underlying platform, giving containers a footprint of just a few megabytes compared to the hundreds of megabytes required for VMs. With the small footprint and limited code, it's easy to scale individual microservices by quickly spinning up more container images. For these reasons, containers have become the deployment unit of choice for microservices. As one gauge of its popularity, Google claims to run everything on containers and spins up billions(!) of containers each week.

Working together, microservices supported by these three trends — DevOps, containers, and CI/CD automation — have greatly increased the agility and velocity of application lifecycles. A developer can now code, test, and deploy a containerized microservice to the cloud in hours, if not minutes.

2 Challenges with Microservices

Of course, despite all their advantages and popularity, microservices aren't a silver bullet for every situation. A premature or uninformed move to microservices can cause pitfalls and serious quality issues. The challenges with moving to (and maintaining quality with) microservices-based applications are those inherent to any distributed computing solution: performance, failures (especially partial failures which can be tricky to handle), data consistency, architectural choices, and more.

Let's look in detail at some of these challenges with microservices, how they can affect the quality of your systems, and ways to address each challenge.

Granularity

When you make the move from a monolith to microservices, one of your first decisions is how granular your new services should be. Deciding on the right level of granularity is an art, not a science. Breaking your systems into too many microservices leads to an overwhelming number of chatty interactions, resulting in latency and a slow system. On the other hand, divisions that are too coarse lead to a tight coupling between your services, which brings you right back to the problems of monoliths and the reasons you decided to switch in the first place. This art of finding the right size for your microservices is complex and debated, but in general, it's recommended that a microservice should be small enough that a team can build and deploy a user story in one day, yet large enough that it contains an entire business concept and its associated services.

Network Communication

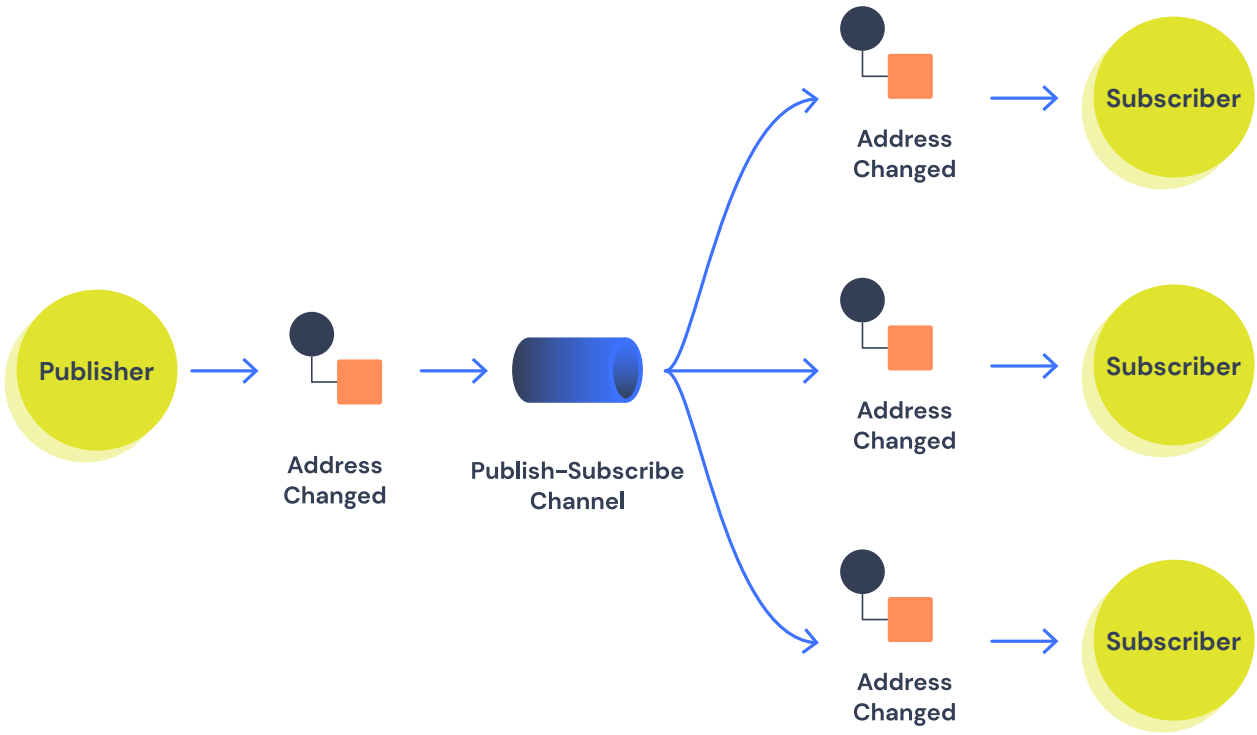
Unlike in-memory direct calls between parts of a monolithic application, network communications are essential for maintaining the loose coupling between microservices. To avoid quality issues caused by the unexpected effects of networking, developers need to be aware of the fallacies of distributed computing, such as misconceptions that the network is reliable, latency is zero, and bandwidth is infinite. One approach to reduce latency is to use the container deployments previously mentioned. Some tools, such as Kubernetes, can minimize communication overhead by placing microservices in the same container (a pod, in Kubernetes parlance).

Interaction Exchange Patterns

Creating a decoupled microservices architecture requires a thorough understanding of how those microservices exchange messages. Typically, services interact either through a request-response or publish-subscribe pattern. Choosing the correct pattern is important, as the wrong pattern can cause issues throughout your system that can be difficult to track down.

With request-response (which closely mimics function calls in monolithic software), the requesting service sends a specific message to a known recipient, then idles while it waits for an answer. Request-response interactions imply a runtime coupling between services. While a popular architectural choice —and relatively easy to implement —request-response can lead to chatty interactions which increase latency and can cause services to hang while they wait for a response. An issue like this can be difficult to debug.

For more complex systems, a better solution might be event-driven architectures that use the publish-subscribe pattern (or "pub/sub"), where a service "subscribes" to system events by registering itself with a message broker. When events happen on other services, those services "publish" the event information to the broker, who in turn notifies all interested subscribers. This pattern achieves maximum decoupling through use of the broker.



Pub/Sub pattern courtesy of Enterprise Integration Patterns

System Latency

As we've seen above, system latency is a major concern with microservice architectures. In addition to the causes mentioned above, there are other factors that can increase system latency, such as the number of microservices involved in a request, the load on each of these services, the network bandwidth between these services, and the network placement of these services.

By closing monitoring system latency with APM tools such as AppOptics or New Relic, DevOps teams can evaluate which microservices need to be deployed together locally, the appropriate choice of network protocols, and tweak the APIs.

Partial Failures

In a distributed microservices architecture, there is always the risk of a difficult-to-identify partial failure that brings down the entire system. For example, a microservice might become isolated by a network failure, or an excessive load could cause a response time-out. To guard against these failures, it's essential that microservices (and the applications that rely on them) are designed to cleanly handle and recover from partial failures. Some techniques include providing fallbacks, such as cached responses, and limiting the number of outstanding requests allowed to queue. At the least, the application should be designed to continue operating with a missing piece rather than collapse completely.

Additionally, with cloud-based containerized microservices, it only takes seconds to automatically remove and replace failing containers or to spin up new container instances to meet demand spikes. So by simply monitoring the service and its demand, you can help prevent most failures. Similarly, geographical diversity of the deployments, which is common in most popular cloud platforms, can mitigate national or regional outages.

Consistency

The possibility of partial failures also makes it especially difficult to maintain data consistency. Monolithic applications use a single data store to ensure data consistency during and after updates. Microservices, on the other hand, typically each use their own databases, allowing each team to use its preferred database stack. Because of this, data consistency is much more difficult to implement in distributed systems.

Monolithic software uses Atomicity, Consistency, Isolation, and Durability (ACID) transactions to ensure that data remains consistent during and after updates. For microservices, however, distributed transactions are very difficult. One popular approach for data consistency is to strive for eventual consistency. You can also mitigate risks by compensating operations that reverse mistaken actions, such as in the Saga pattern. This mimics the way businesses handle inconsistencies, all while maintaining operational continuity.

Organizational Changes

Another major challenge when moving to microservices is your organizational structure and culture. Some organizations are not prepared for the cultural changes that need to occur.

For example, does your organization have experience with, and embrace, independent teams? Deploying a monolith is usually a coordinated effort spanning many large teams and departments, each responsible for well-defined silos such as UI, middleware, database, and testing. But the small, autonomous teams necessary for DevOps-based development are the exact opposite. Microservice teams own their service from inception to retirement — they build it, deploy it, and support it. This change in approach can be difficult.

Also, what is your overall organizational culture? Is your culture open to the changes required to adopt a microservice approach? How does your organization fit into the three organizational cultures — pathological, bureaucratic, and generative — as defined by sociologist Ron Westrum and popularized by Jez Humble? To determine your work culture: ask the question, "How are messengers handled by your organization?" If the "messenger is shot," then you have a pathological culture. If the "messenger is ignored," then you have a bureaucratic culture. Finally, if the "messenger is trained," then you have a generative culture. If your company is generative or moving towards a generative culture, then a microservices architecture might be a good fit for you.

Testing and Debugging Microservices

Finally, let's look at how the shift to microservices, DevOps, and CI/CD introduces serious challenges to your quality assurance processes. When a team deploys production software thousands of times a day (as some companies do) using potentially thousands of independent microservices, the old methods of testing and debugging will no longer work.

On one hand, microservice-based code that's built, tested, and deployed in smaller chunks and in more iterations means less code and more opportunity for tests, making it easier to troubleshoot and redeploy code. On the other hand, the choreography between microservices within the complete application creates much more complexity than before, and an inability to track down the root cause of defects.

Let's look at some of the traditional steps for testing and how they are affected in the new world of microservices.

Integration Testing

With microservices, testing at each stage of the delivery pipeline is still necessary. However, the focus of what needs testing is a little different.

The code for a typical monolithic application usually follows an organization-dependent structure — UI, middleware, backend, database. This is an example of Conway's Law, which says that software design typically resembles an organization's structure. As a result, most organizations have siloed tools and testing practices for each stage of the SDLC.

In contrast, microservice teams form around individual business capabilities and are composed of specialists — from developers to operations support — for the full application lifecycle. These teams independently choose the languages, frameworks, and tools they will use. The way one team approaches testing can be very different from the way another team does.

And since microservice testing must include the interactions with other microservices — both the network connections and the contracts between the services — testing must be coordinated among multiple teams. Testing the integration among thousands of microservices across disparate teams with disparate technologies requires new approaches. Some solutions include automated integration testing that can grow as the number of interconnected microservices grows, and hiring cross-functional quality assurance engineers that fully understand the different microservices and their interactions.

Debugging

Having multiple microservices composing an application also means that there are multiple points of failure. Therefore, when an error occurs, it can be difficult to pinpoint the precise step where that failure occurred. Failure investigation in a microservice environment requires monitoring multiple services and sifting through multiple log files and multiple network connections. Three strategies that can help are application monitoring, error monitoring, and better logging practices. Let's look at each one.

Application Performance Monitoring

Application performance monitoring (APM) tools are a way to periodically sample your systems for metrics on performance, resource consumption, availability, and error rate. APM is useful for:

- **Measuring the overall status of applications and infrastructure**
- **Tracking changes in performance or stability over time**
- **Identifying bottlenecks**

APM is also commonly used in transaction tracing, which measures the flow of requests throughout each component of the application. This shows you exactly how long each component takes to process the request, helping both development and operations find and address performance issues.

Error Baselines

APM can help you understand when and why your services are failing. However, recalling our earlier discussion of partial failures in a microservices environment, it's clear that monitoring everything to ensure 100% visibility is not feasible, nor desired. Being flooded with hundreds of events without a clear idea of what to measure is unproductive. The art of APM lies in finding the business impact of each possible failure, which leads to defining the appropriate metrics, and then determining which aspects to monitor. The general approach is to fix and redeploy those services that have the greatest business impact as quickly as possible.

Logging

Logging individual events from microservices can give you critical information on what has happened within your services, though this raw data does not provide easily actionable insights. A reasonable approach with microservices is to log individual events within a service, and then use an aggregation tool to see the bigger picture. In this way, microservice events can be logged locally and moved into a central aggregator for viewing, root cause analysis, and forensics.

Continuous Code Improvement

Error monitoring focuses on detecting and reporting problems in real time. As opposed to APM, which performs periodic sampling and tracing, error monitoring receives errors as they occur along with important diagnostic and contextual data. By implementing an error monitoring tool such as Rollbar, you can not only be alerted when failures in your systems occur, but also see contextual information, including the code behind the failure, user information such as browser and activities, affected environments, aggregated log and error information, and root cause of the error.

Error monitoring tools can also be used to set baselines for errors as you make the move to microservices. As dependencies get more complex, error monitoring can give you a holistic view of errors across all applications with historic trends on errors and occurrences. With a clear baseline, any new errors are easier to identify and fix.

3 Meeting the Quality Challenge

Now that we've seen the many challenges associated with moving to microservices, and talked about some of the ways you can address those challenges, let's look at some overall strategies to help you maintain quality while moving to microservices.

Organizational Aspects

Once you commit to using microservices, DevOps, and CI/CD pipelines, the question becomes how to effect this change. It's not feasible to expect an organization built around monolithic applications to adopt microservices architectures in one giant move. The transition should be gradual and include incremental changes.

Here are several best practices to help prepare your organization for such a transition.

Establish Goals

Your first microservice-based projects should have clear, measurable goals. Having a clearly defined plan, support system, and dedicated resources to ensure each team is successful will help make the transition as smooth as possible. These must be internalized and agreed to by everyone, from management down. The primacy effect of exposure to new experiences — which says the first information presented is more easily remembered — should be considered. A positive initial exposure and acceptance will have long-term benefits.

Create Cross-Functional Teams

The first microservices projects are also a good time to create appropriate cross-functional DevOps-centric teams that can take responsibility for the full lifecycle of their microservice. Such teams would ideally include business owners (who can identify requirements), developers, QA and testing experts, and operations.

To align with the microservice delivery model, where a microservice team takes responsibility for the entire lifecycle from concept to production, each team should be able to operate autonomously, without the need for additional interfaces or permissions.

These initial projects are also a good opportunity for the DevOps team to establish new workflows, work out the automation, and experiment with different ways to build and deploy microservices. If your organizational culture is a generative culture, then this experimentation and failure will be welcomed as a learning opportunity.

Get the Size Right

With your first microservice projects, you should also start small, choosing one or two small, but useful, projects that can be carried out as an adjunct to any ongoing monolith development. Avoid organization-wide changes without proving the success of a few small microservice projects.

Define Metrics

Businesses have long adopted Key Performance Indicators (KPIs), such as revenue, time to market, conversions, and so on, to measure the success of their plans and activities. Peter Drucker, inventor of modern business management and credited with the concept, once said "If you can't measure it, you can't improve it." Without such measurements, it is difficult to quantify — and eventually justify — the advantages of moving to microservices in future projects.

Microservice-based solutions should adopt a similar mindset and monitor KPIs such as software quality, time to market, developer productivity, and others. These can be computed by gathering smaller and more measurable quantities such as the time to deployment of a microservice or the frequency of updates to a microservice.

Handle Legacy

It's best to choose initial projects that don't depend on the monolith's code or release cycle. The ideal choice will be functionality that doesn't directly impact customer-facing features, but still provides enough exposure to realistic situations to prove a successful transition to a microservice. A good trial microservice might be adding a new feature that isn't currently in the monolith, can be deployed separately, but can still be used by the monolith. Some suggestions include:

- Start with edge functionality that allows you to move a common, but ancillary, service, such as authentication or communications (emails, texts, etc.). This will give your team practice in moving to the new architecture, but will limit the changes to a single point of failure.
- Start with functionality that uses only isolated data. Because moving to microservices can lead to complicated scenarios involving distributed database transactions (as mentioned above), it is often easiest to start with functionality that uses data in isolation in order to remove initial complexity.
- Start with relatively unimportant pieces of functionality that can fail with minimal impact, but that will allow your team to establish the baseline of your architecture (APIs, containers, etc.). That way, you can observe any issues with your new architecture (such as latencies, since microservices may introduce latencies that didn't exist in monoliths), and generally become comfortable with the idea of microservices.

Technical Aspects

Now let's look at a few technical best practices that can help with maintaining quality.

Use an Anti-Corruption Layer

As the name suggests, the anti-corruption layer is a design pattern implemented as an API between two different systems. This layer translates between their semantics and syntax so that neither is affected by their differences. Using an anti-corruption layer between your microservice and your monolith allows your microservice to safely call your legacy application while remaining isolated.

Refactor the Monolith

Refactoring your monolith to help untangle dependencies and optimize for microservices-style development can provide major benefits as you begin your transition. Of course, any legacy refactoring requires a good understanding of the replacement monolith. A legacy monolith may have accumulated more and more features while also remaining unexamined for years, and by this time it likely contains hundreds of modules, linked together in ways that aren't understood. This blog post provides more detailed technical best practices for refactoring your legacy software, such as streamlining your monolith's build and untangling dependencies, CI/CD techniques, optimizing for local development, supporting parallel development, adopting infrastructure as code, and several others.

Avoiding Undifferentiated Heavy Lifting

DevOps teams take on a lot of tasks that can often be done more easily (and perhaps better) by others who specialize in those tasks. Some examples include buying hardware, setting up and maintaining servers, managing network interconnections, and so on. These are examples of what Amazon's CTO, Werner Vogels, calls "undifferentiated heavy lifting." This describes work that could easily be outsourced to others (e.g., a cloud platform), allowing the DevOps teams to concentrate on providing business value. This sort of undifferentiated work also includes building something from scratch when commercial or open source software that fulfills the same purpose is readily available.

Observability

One of the challenges for DevOps teams is to understand and analyze the relationships between infrastructure, microservices, their source code, and software delivery processes. Truly understanding these relationships, and how these systems behave, is a key to getting to the root of complex errors and performance issues. This understanding into your systems is called observability.

In software practice, observability means understanding an application's behavior by looking at its outputs. Specifically, an observable system allows you to see, and determine the cause of, failures. Observability works by instrumenting the application with events, monitors, logs, and other data that can be analyzed to troubleshoot problems. Put another way, you don't query the system to find out its status, the data, and the status of the system; instead, it is given to you. Observability is especially important in microservice-based systems, where thousands of discrete services and moving parts make observability incredibly difficult.

Building in observability during development, testing, and staging also addresses issues that arise before production. It helps to identify performance issues, process inefficiencies, and user experience problems earlier in the development lifecycle. By using observability, teams can find issues earlier, build more efficient applications, improve performance, and optimize infrastructure decisions. This leads to a better experience for the end user and greater value for the business, which spends less time and money to deliver quality software.

Error Monitoring for Observability

One way to achieve this observability in a microservices architecture is through the previously mentioned error monitoring. As many microservices interact with one another to perform specific tasks, a single higher-order request can generate calls to multiple microservices. To understand the entire call chain, an error monitoring tool like Rollbar can assign a unique identifier — a correlation ID — to every message in the system.

This ID is propagated between every service and included in all log messages, enabling you to quickly search across calls and quickly debug the root cause of failures.

Rollbar also allows you to tie this correlation ID into your error and stack traces, and capture additional contextual information about the execution environment, such as the client device, payload, local variables, and more. Together, these can give you greater insight into your microservices, allowing you to pinpoint where errors occur, how many times, and why.

Conclusion

There are many benefits of moving to a microservices-based architecture. At its simplest, microservices enable agile development, faster time to market, and a faster pace of new feature releases. By using best practices in your organizational and technical strategies, you can successfully migrate your monoliths to microservices and still maintain a high level of quality in your systems.

About Rollbar

Rollbar is the leading continuous code improvement platform that proactively discovers, predicts, and remediates errors with real-time AI-assisted workflows. With Rollbar, developers continually improve their code and constantly innovate rather than spending time monitoring, investigating, and debugging. More than 5,000 businesses, including Twilio, Salesforce, Twitch, and Affirm, use Rollbar to deploy better software, faster while quickly recovering from critical errors as they happen. Learn more at **rollbar.com**