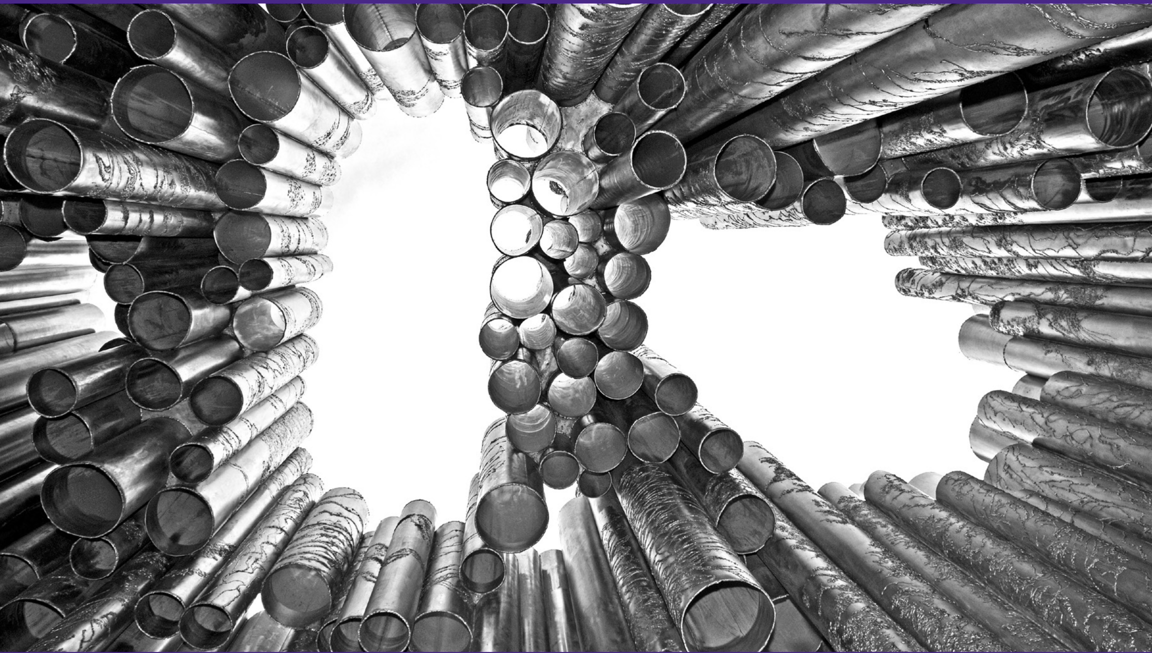
O'REILLY®

Microservices in Production

Standard Principles and Requirements



Susan J. Fowler

O'REILLY®

Additional Resources

4 Easy Ways to Learn More and Stay Current

Programming Newsletter
Get programming related news and content delivered weekly to your inbox.
oreilly.com/programming/newsletter

Free Webcast Series
Learn about popular programming topics from experts live, online.
webcasts.oreilly.com

O'Reilly Radar
Read more insight and analysis about emerging technologies.
radar.oreilly.com

Conferences
Immerse yourself in learning at an upcoming O'Reilly conference.
conferences.oreilly.com

©2015 O'Reilly Media, Inc. The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. #15305

Microservices in Production

Standard Principles and Requirements

Susan J. Fowler

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Microservices in Production

by Susan J. Fowler

Copyright © 2017 O'Reilly Media. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editors: Nan Barber and Brian Foster
Production Editor: Colleen Lobner
Copyeditor: Octal Publishing, Inc.
Interior Designer: David Futato
Cover Designer: Randy Comer
Illustrator: Rebecca Demarest

October 2016: First Edition

Revision History for the First Edition

2016-09-07: First Release

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Microservices in Production*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

While the publisher and the author have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-491-97297-7

[LSI]

Table of Contents

1. Microservices in Production. 1
Introduction 1
The Challenges of Microservice Standardization 1
Availability: The Goal of Standardization 3
Production-Readiness Standards 4
Stability 5
Reliability 6
Scalability 7
Fault Tolerance and Catastrophe Preparedness 8
Performance 10
Monitoring 11
Documentation 13
Implementing Production-Readiness 15

CHAPTER 1

Microservices in Production

Introduction

Although the adoption of microservice architecture brings considerable freedom to developers, ensuring availability requires holding microservices to high architectural, operational, and organizational standards. This report covers the challenges of microservice standardization in production, introduces availability as the goal of standardization, presents the eight production-readiness standards, and includes strategies for implementing production-readiness standardization across an engineering organization.

The Challenges of Microservice Standardization

The architecture of a monolithic application is usually determined at the beginning of the application's lifecycle. For many applications, the architecture is determined at the time a company begins. As the company grows and the application scales, developers who are adding new features often find themselves constrained and limited by the choices made when the application was first designed. They are constrained by choice of language, by the libraries they are able to use, by the development tools they can work with, and by the need for extensive regression testing to ensure that every new feature they add does not disturb or compromise the entirety of the application. Any refactoring that happens to the standalone, monolithic application is still essentially constrained by initial architec-

tural decisions: initial conditions exclusively determine the future of the application.

The adoption of microservice architecture brings a considerable amount of freedom to developers. They are no longer tied to the architectural decisions of the past; they can design their service however they wish; and they have free reign regarding decisions of language, of database, of development tools, and the like. The message accompanying microservice architecture guidance is usually understood and heard by developers as follows: build an application that does one thing—one thing only—and does that one thing *extraordinarily well*; do whatever you need to do, and build it however you want. Just make sure it gets the job done.

Even though this romantic idealization of microservice development is true in principle, not all microservices are created equal—nor should they be. Each microservice is part of a microservice ecosystem, and complex dependency chains are necessary. When you have one hundred, one thousand, or even ten thousand microservices, each of them are playing a small part in a very large system. The services must interact seamlessly with one another, and, most importantly, no service or set of services should compromise the integrity of the overall system or product of which they are a part. If the overall system or product is to be any good, it must be held to certain standards, and, consequently, each of its parts must abide by these standards as well.

It's relatively simple to determine standards and give requirements to a microservice team if we focus on the needs of that specific team and the role its service is to play. We can say, "your microservice must do *x*, *y*, and *z*, and to do *x*, *y*, and *z* well, you need to make sure you meet this set *S* of requirements." Of course, you must also give each team a set of requirements that is relevant to its service, and its service alone. This simply isn't scalable and ignores the fact that a microservice is but a very small piece of an absurdly large puzzle. We must define standards and requirements for our microservices, and they must be general enough to apply to every single microservice yet specific enough to be quantifiable and produce measurable results. This is where the concept of *production-readiness* comes in.

Availability: The Goal of Standardization

Within microservice ecosystems, service-level agreements (SLAs) regarding the availability of a service are the most commonly used methods of measuring a service's success: if a service is highly available (that is, has very little downtime), we can say with reasonable confidence (and a few caveats) that the service is doing its job.

Calculating and measuring availability is easy. You need to calculate only three measurable quantities: *uptime* (the length of time that the microservice worked correctly), *downtime* (the length of time that the microservice was *not* working correctly), and the total time a service was operational (the sum of uptime and downtime). Availability is then the uptime divided by the total time a service was operational (uptime + downtime).

As useful as it is, availability is not in and of itself a principle of microservice standardization, it is the goal. It can't be a principle of standardization because it gives no guidance as to how to build the microservice; telling a developer to make her microservice more available without telling her how to do so is useless. Availability alone comes with no concrete, applicable steps, but, as we will see in the following sections, there are concrete, applicable steps we can take toward reaching the goal of building an available microservice.

Calculating Availability

Availability is measured in so-called "nines" notation, which corresponds to the%age of time that a service is available. For example, a service that is available 99% of the time is said to have "2-nines availability."

This notation is useful because it gives us a specific amount of downtime that a service is allowed to have. If your service is required to have 4-nines availability, it is allowed 52.56 minutes of downtime per year, which is 4.38 minutes of downtime per month, 1.01 minutes of downtime per week, and 8.66 seconds of downtime per day.

Here are the availability and downtime calculations for 99% availability to 99.999% availability:

99% availability: (2-nines):

- 3.65 days/year (of allowed downtime)

- 7.20 hours/month
- 1.68 hours/week
- 14.4 minutes/day

99.9% availability (3-nines):

- 8.76 hours/year
- 43.8 minutes/month
- 10.1 minutes/week
- 1.44 minutes/day

99.99% availability (4-nines):

- 52.56 minutes/year
- 4.38 minutes/month
- 1.01 minutes/week
- 8.66 seconds/day

99.999% availability (5-nines):

- 5.26 minutes/year
- 25.9 seconds/month
- 6.05 seconds/week
- 864.3 milliseconds/day

Production-Readiness Standards

The basic idea behind production-readiness is this: a production-ready application or service is one that can be trusted to serve production traffic. When we refer to an application or microservice as "production-ready," we confer a great deal of trust upon it: we trust it to behave reasonably; we trust it to perform reliably; and we trust it to get the job done and to do the job well. Production-readiness is the key to microservice standardization.

However, the idea of production-readiness as just stated isn't useful enough to serve as the exhaustive definition we need, and without further explication the concept is completely useless. We need to know exactly what requirements every service must meet in order to

be deemed production-ready and be trusted to serve production traffic in a reliable, appropriate way. The requirements must themselves be principles that are true for every microservice, for every application, and for every distributed system. Standardization without principle is meaningless.

It turns out that there is a set of eight principles that fits this criteria. Each of them is quantifiable, gives rise to a set of actionable requirements, and produces measurable results. They are: *stability*, *reliability*, *scalability*, *fault-tolerance*, *catastrophe-preparedness*, *performance*, *monitoring*, and *documentation*. The driving force behind each of these principles is that, together, they contribute to and drive the *availability* of a microservice.

Availability is, in some ways, an emergent property of a production-ready microservice. It emerges from building a scalable, reliable, fault-tolerant, performant, monitored, documented, and catastrophe-prepared microservice. Any one of these principles individually is not enough to ensure availability, but together they are. Building a microservice with these principles as the driving architectural and operational requirements guarantees a highly available system that can be trusted with production traffic.

Stability

With the introduction of microservice architecture, developers are given the freedom to develop and deploy at a very high velocity. They can add and deploy new features each day, they can quickly fix bugs and swap-out old technologies the newest ones, and they can rewrite outdated microservices and deprecate and decommission the old versions. With this increased velocity comes increased instability, and we can trace the majority of outages in microservice ecosystems to bad deployments containing buggy code. To ensure availability, we need to carefully guard against this instability that stems from increased developer velocity.

Stability allows us to reach availability by giving us ways to responsibly handle changes to microservices. A stable microservice is one for which development, deployment, the addition of new technologies, and the decommissioning and deprecation of services doesn't cause instability in the larger microservice ecosystem. We can determine stability requirements for each microservice to mitigate the negative side effects that might accompany each change.

To mitigate any problems that might arise from the development cycle, we can put stable development procedures into place. To counteract any instability introduced by deployment, we can ensure our microservices are deployed carefully with proper staging, canary, and production rollouts. To prevent the introduction of new technologies and the deprecation and decommissioning of old microservices from compromising the availability of other services, we can enforce stable introduction and deprecation procedures.

Stability Requirements

Following are the requirements of building a stable microservice:

- A stable development cycle
- A stable deployment process
- Stable introduction and deprecation procedures

Reliability

Stability alone isn't enough to ensure a microservice's availability—the service must also be *reliable*. A reliable microservice is one that can be trusted by its clients, by its dependencies, and by the ecosystem as a whole.

Although stability is related to mitigating the negative side effects accompanying change, and reliability is related to trust, the two are inextricably linked. Each stability requirement also carries a reliability requirement alongside it. For example, developers should not only seek to have stable deployment processes, they should ensure that each deployment is reliable from the point of view of one of their clients or dependencies.

The trust that reliability secures can be broken into several requirements, the same way we determined requirements for stability earlier. For example, we can make our deployment processes reliable by ensuring that our integration tests are comprehensive and our staging and canary deployment phases are successful.

By building reliability into our microservices, we can protect their availability. We can cache data so that it will be readily available to client services, helping them protect their SLAs by making our own

services highly available. To protect our own SLA from any problems with the availability of our depencies, we can implement defensive caching.

The last reliability requirement is related to routing and discovery. Availability requires that the communication and routing between different services be reliable: health checks should be accurate, requests and responses should reach their destinations, and errors should be handled carefully and appropriately.

Reliability Requirements

Following are the requirements of building a reliable microservice:

- A reliable deployment process
- Planning, mitigating, and protecting against the failures of dependencies
- Reliable routing and discovery

Scalability

Microservice traffic is rarely static or constant, and the hallmark of a successful microservice (and of a successful microservice ecosystem) is a steady increase in traffic. We need to build microservices in preparation for this growth—they need to accommodate it easily, and they need to actively scale with it. A microservice that can't scale with growth experiences increased latency, poor availability, and, in extreme cases, a drastic increase in incidents and outages. *Scalability* is essential for availability, making it our third production-readiness standard.

A scalable microservice is one that can handle a large number of tasks or requests at the same time. To ensure a microservice is scalable, we need to know both its qualitative growth scale (e.g., whether it scales with page views or customer orders) and its quantitative growth scale (i.e., how many requests per second it can handle). As soon as we know the growth scale, we can plan for future capacity needs and identify resource bottlenecks and requirements.

The way a microservice handles traffic should also be scalable. It should be prepared for bursts of traffic, handle them carefully, and

prevent them from taking down the service entirely. Of course, this is easier said than done, but without scalable traffic handling, developers can (and will) find themselves looking at a broken microservice ecosystem.

Additional complexity is introduced by the rest of the microservice ecosystem. We need to prepare for the inevitable additional traffic and growth from a service's clients. Likewise, any dependencies of the service should be alerted when increases in traffic are expected. Cross-team communication and collaboration are essential for scalability: regularly communicating with clients and dependencies about a service's scalability requirements, status, and any bottlenecks ensures that any services relying on one another are prepared for growth and for potential pitfalls.

Last but not least, the way a microservice stores and handles data needs to be scalable. Building a scalable storage solution goes a long way toward ensuring the availability of a microservice, and is one of the most essential components of a truly production-ready system.

Scalability Requirements

Here are the requirements of building a scalable microservice:

- Well-defined quantitative and qualitative growth scales
- Identification of resource bottlenecks and requirements
- Careful, accurate capacity planning
- Scalable handling of traffic
- The scaling of dependencies
- Scalable data storage

Fault Tolerance and Catastrophe Preparedness

Even the simplest of microservices is a fairly complex system. As we know quite well, complex systems fail, they fail often, and any potential failure scenario can and will happen at some point in the microservice's lifetime. Microservices don't live in isolation, but within dependency chains as part of a larger, incredibly complex

microservice ecosystem. The complexity scales linearly with the number of microservice in the overall ecosystem, and ensuring the availability of not only an individual microservice, but the ecosystem as a whole, requires that we impose yet another production-readiness standard onto each microservice. Every microservice within the ecosystem must be *fault-tolerant* and *prepared for any catastrophe*.

A fault-tolerant, catastophe-prepared microservice is one that can withstand both internal and external failures. Internal failures are those that the microservice brings on itself: for example, code bugs that aren't caught by proper testing lead to bad deploys, causing outages that affect the entire ecosystem. External catastrophes such as datacenter outages and or poor configuration management across the ecosystem lead to outages that affect the availability of every microservice and the entire organization.

We can quite adequately (though not exhaustively) prepare for failure scenarios and potential catastrophes. Identifying failure and catastrophe scenarios is the first requirement of building a fault-tolerant, production-ready microservice. After these scenarios have been identified, the hard work of strategizing and planning for when they will occur begins. This must happen at every level of the microservice ecosystem, and any shared strategies should be communicated across the organization so that mitigation is standardized and predictable.

Standardization of failure mitigation and resolution at the organizational level means that incidents and outages of individual microservices, infrastructure components, or the ecosystem as a whole need to be wrapped into carefully executed, easily understandable procedures. Incident response procedures need to be handled in a coordinated, planned, and thoroughly communicated manner. If incidents and outages are handled in this way, and the structure of incident response is well defined, organizations can avoid lengthy downtimes and protect the availability of the microservices. If every developer knows exactly what they are supposed to do during an outage, knows how to mitigate and resolve problems quickly and appropriately, and knows how to escalate if an issue is beyond their capabilities or control, the time to mitigation and time to resolution drop drastically.

Making failures and catastrophes predictable means going one step further after identifying failure and catastrophe scenarios and planning for them. It means forcing the microservices, the infrastructure, and the ecosystem to fail in any and all known ways to test the availability of the entire system. We can accomplish this through various types of resiliency testing. Code testing (including unit tests, regression tests, and integration tests) is the first step in testing for resiliency. The second step is load testing—this is when we test microservices and infrastructure components for their ability to handle drastic changes in traffic. The last, most intense, and most relevant type of resiliency testing is chaos testing, in which failure scenarios are run (both scheduled and at randomly) on production services to ensure that microservices and infrastructure components are prepared for all known failure scenarios.

Fault-Tolerance and Catastrophe-Preparedness Requirements

Here are the requirements for building a fault-tolerant microservice that is prepared for any catastrophe:

- Potential catastrophes and failure scenarios are identified and planned for
- Single points of failure are identified and resolved
- Failure detection and remediation strategies are in place
- The microservice is tested for resiliency through code testing, load testing, and chaos testing
- Traffic is managed carefully in preparation for failure
- Incident and outages are handled appropriately and productively

Performance

In the context of the microservice ecosystem, scalability (which we covered in brief detail earlier), is related to how many requests a microservice can handle. Our next production-readiness principle, *performance*, refers to how well the microservice handles those requests. A performant microservice is one that handles requests

quickly, processes tasks efficiently, and properly utilizes resources (such as hardware and other infrastructure components).

A microservice that makes a large number of expensive network calls, for example, is not performant. Neither is a microservice that processes and handles tasks synchronously in cases for which asynchronous (non-blocking) task processing would increase the performance and availability of the service. Identifying and architecting away these performance problems is a strict production-readiness requirement.

Similarly, dedicating a large number of resources (like CPU) to a microservice that doesn't utilize them is inefficient. Inefficiency reduces performance—if it's not clear at the microservice level in every case, it's painful and costly at the ecosystem level. Underutilized hardware resources affect the bottom line, and hardware is not cheap. There's a fine line between underutilization and proper capacity planning, and so the two must be planned and understood together to avoid compromising the availability of any microservice and to keep the costs associated with underutilization reasonable.

Performance Requirements

Here are the requirements of building a performant microservice:

- Appropriate service-level agreements (SLAs) for availability
- Proper task handling and processing
- Efficient utilization of resources

Monitoring

Another principle necessary for guaranteeing microservice availability is proper microservice *monitoring*. Good monitoring has three components: (1) proper logging of all important and relevant information, (2) useful graphical displays (dashboards) that are easily understood by any developer in the company and that accurately reflect the health of the services, and (3) alerting on key metrics that is effective and actionable.

Logging belongs and begins in the codebase of each microservice. Determining precisely what information needs to be logged will dif-

fer for each service, but the goal of logging is quite simple: when faced with a bug—even one from many deployments in the past—you want and need your logging to be such that you can determine from the logs exactly what went wrong and where things fell apart. Versioning microservices (and their endpoints) is discouraged in microservice architecture, so you won't have a precise version to refer to in which to find any bugs or problems. Code is revised frequently, deployments happen multiple times per week, features are added constantly, and dependencies are constantly changing, but logs will remain the same, preserving the information needed to pinpoint any problems. Just ensure that your logs contain the information necessary to determine possible problems.

All key metrics (such as hardware utilization, database connections, responses and average response times, and the status of API endpoints) should be graphically displayed in real time on an easily accessible dashboard. Dashboards are an important component of building a well-monitored, production-ready microservice; they make it easy to determine the health of a microservice with one glance, and enable developers to detect strange patterns and anomalies that might not be extreme enough to trigger alerting thresholds. When used wisely, dashboards allow developers to determine whether a microservice is working correctly simply by looking at the dashboard, but developers should not need to watch the dashboard in order to detect incidents and outages.

The actual detection of failures is accomplished through alerting. All key metrics can be alerted on, including (at the very least) CPU and RAM utilization, number of file descriptors, number of database connections, the SLA of the service, requests and responses, the status of API endpoints, errors and exceptions, the health of the service's dependencies, information about any database(s), and the number of tasks being processed (if applicable).

Normal, warning, and critical thresholds can be set for each of these key metrics, and any deviation from the norm (i.e., hitting the warning or critical thresholds) should trigger an alert to the developers who are on-call for the service. The best thresholds are signal-providing: high enough to avoid noise, but low enough to catch any and all real problems.

Alerts need to be useful and actionable. A non-actionable alert is not a useful alert, and a waste of engineering hours. Every actionable

alert should be accompanied by a runbook. For example, if an alert is triggered on a high number of exceptions of a certain type, there needs to be a runbook containing mitigation strategies that any on-call developer can refer to while attempting to resolve the problem.

Monitoring Requirements

Following are the requirements of building a properly-monitored microservice:

- Proper logging and tracing throughout the stack
- Well-designed dashboards that are easy to understand and accurately reflect the health of the service
- Effective, actionable alerting accompanied by runbooks
- Implementing and maintaining an on-call rotation

Documentation

Microservice architecture carries the potential for increased technical debt—it's one of the key tradeoffs that come with adopting microservices. As a rule, technical debt tends to increase *with* developer velocity: the more quickly a service can be iterated on, changed, and deployed, the more frequently shortcuts and patches will be put into place. Organizational clarity and structure around the *documentation* and *understanding* of a microservice cut through this technical debt and shave off a lot of the confusion, lack of awareness, and lack of architectural comprehension that tend to accompany it.

Reducing technical debt isn't the only reason to make good documentation a production-readiness principle—doing so would make it somewhat of an afterthought (an important afterthought, but an afterthought nonetheless). No, just like each of the other production-readiness standards, documentation and its counterpart (understanding) directly and measurably influence the availability of a microservice.

To see why this is true, we can think about how teams of developers work together and share their knowledge of a microservice. You can do this yourself by getting one of your development teams together in a room, standing in front of a whiteboard, and asking them to

illustrate the architecture and all important details of the service. I promise you'll be surprised by the result of this exercise, and you'll most likely find that knowledge and understanding of the service is not cohesive or coherent across the group. One developer will know one thing about the application that nobody else does, whereas a second developer will have such a different understanding of the microservice that you will wonder if he is even contributing to the same codebase. When it's time for code changes to be reviewed, technologies to be swapped, or features to be added, the lack of alignment of knowledge and understanding will lead to the design and/or evolution of microservices that are not production-ready, containing serious flaws that undermine the service's ability to reliably serve production traffic.

This confusion and the problems that it creates can be successfully and rather easily avoided by requiring that every microservice follow a very strictly standardized set of documentation requirements. The best documentation contains all the essential knowledge (facts) about a microservice, including an architecture diagram, an onboarding and development guide, details about the request flow and any API endpoints, and an on-call runbook for each of the service's alerts.

You can accomplish understanding of a microservice in several ways. The first is by performing the aforementioned exercise: stick the development team in a conference room, and ask them to whiteboard the architecture of the service. Thanks to increased developer velocity, microservices change radically at different times throughout their lifecycle. By making these architecture reviews part of each team's process and scheduling them regularly, you can guarantee that knowledge and understanding about any changes in the microservice will be disseminated to the entire team.

To cover the second aspect of microservice understanding, we need to jump up by one level of abstraction and consider the production-readiness standards themselves. A great deal of microservice understanding is captured by determining whether a microservice is production-ready and where it stands with regard to the production-readiness standards and their individual requirements. You can achieve this in a myriad of ways, one of which is running audits of whether a microservice meets the requirements, and then creating a roadmap for the service detailing how to bring it to a production-ready state. Checking the requirements can also be

automated across the organization. We'll dive into other aspects of this in more detail in the next section on the implementation of production-readiness standards in an organization that has adopted microservice architecture.

Documentation Requirements

Here are the requirements of building a well-documented microservice:

- Thorough, updated, and centralized documentation containing all of the relevant and essential information about the microservice
- Organizational understanding at the developer, team, and ecosystem levels

Implementing Production-Readiness

We now have a set of standards that apply to every microservice in any microservice ecosystem, each with its own set of specific requirements. Any microservice that satisfies these requirements can be trusted to serve production traffic and preserve a high level of availability.

Now that we have the production-readiness standards, the question that remains is *how* we can implement them in a specialized, real-world microservice ecosystem. Going from principle to practice, and applying theory to real-world applications always presents some significant level of difficulty. However, the power of these production-readiness standards and the requirements they impose lies in their remarkable applicability and strict granularity: they are both general enough to apply to any ecosystem yet specific enough to provide concrete strategies for implementation.

Standardization requires buy-in from all levels of the organization, and needs to be adopted and driven both from the top-down and from the bottom-up. At the executive and leadership (managerial and technical) levels, these principles need to be driven and supported as architectural requirements for the engineering organization. On the ground floor, within individual development teams, standardization needs to be embraced and implemented. Impor-

tantly, standardization needs to be seen and communicated not as a hindrance or gate to development and deployment, but as a guide for production-ready development and deployment.

Many developers might resist standardization. After all, they might argue, isn't the point of adopting microservice architecture to provide greater developer velocity, freedom, and productivity? The answer to these sorts of objections is not to deny that the adoption of microservice architecture brings freedom and velocity to development teams, but to agree and point out that that is *exactly* why production-readiness standards need to be in place. Developer velocity and productivity grind to a halt whenever an outage brings a service down, whenever a bad deploy compromises the availability of a microservice's clients and dependencies, whenever a failure that *could have been avoided with proper resiliency testing* brings the entire microservice ecosystem down. If we've learned anything in the past 50 years of software development, it's that standardization brings freedom and reduces entropy. As Brooks says in *The Mythical Man-Month*, perhaps the greatest collection of essays on the practice of software engineering, "form is liberating."

After the engineering organization has adopted and agreed to follow production-readiness standards, the next step is to evaluate and elaborate on each standard's requirements. The requirements presented here are very general and need the addition of context, organization-specific details, and implementation strategies. What remains to be done is to work through each production-readiness standard and its requirements, and then figure out how each requirement can be implemented across the engineering organization. For example, if the organization's microservice ecosystem has a self-service deployment tool, implementing a stable and reliable deployment process needs to be communicated in terms of the internal deployment tool and how it works. Rebuilding internal tools and/or adding features to them can also come out of this exercise.

The actual implementation of the requirements and determining whether a given microservice meets them can be done by the developers themselves, by team leads, by management, or by DevOps engineers, or by site reliability engineers. At both Uber and the several other companies I know that have adopted production-readiness standardization, the implementation and enforcement of the production-readiness standards is driven by the site reliability

engineering (SRE) organizations. SREs are, typically, responsible for the availability of the services, and so driving these standards across the microservice ecosystem fits in quite well with existing responsibilities. That isn't to say that the developers or development teams have no responsibility for ensuring their services are production ready, rather, SREs inform, drive, and enforce production-readiness within the microservice ecosystem, and the responsibility of implementation falls on both the SREs embedded within development teams and on the developers themselves.

Building and maintaining a production-ready microservice ecosystem is not an easy challenge to undertake, but the rewards are great and the impact is reflected in the increased availability of each microservice. Implementing production-readiness standards and their requirements provides measurable results and allows development teams work knowing that the services they depend on are trustworthy, stable, reliable, fault-tolerant, performant, monitored, documented, and prepared for any catastrophe.

About the Author

Susan J. Fowler is a site reliability engineer at Uber, where she runs a production-readiness initiative across all Uber microservices and embeds within business-critical teams to bring their services to a production-ready state. She worked on application platforms and infrastructure at several small startups before joining Uber, and before that, studied particle physics at Penn, where she searched for supersymmetry and designed hardware for the ATLAS and CMS detectors.