DZone

DZONE TREND REPORT

AUGUST 2022

Microservices and Containerization

The Intersection of Cloud Architectures and Design Principles

BROUGHT TO YOU IN PARTNERSHIP WITH

vFunction

Welcome Letter

By Guillaume Hugot, Director of Engineering at DZone

Scalability has always been one of the greatest challenges in software development. The computer science industry has grown at an unprecedentedly rapid rate in its short 50 years. Software products have become more complex over time, having interacted with more heterogenous and massive amounts of data and having been developed by teams with various expertise at the same time. Different approaches toward this complexity have been explored.

Microservices and containerization are two trending words that have gained traction over the past several years. They are complementary attempts to address the challenges of complexity with the same idea: reducing everything into the smallest and simplest possible pieces — microservices for software architecture and containers for its deployment.

Despite the terms' recent gains in popularity, microservices architecture is nothing new and has been part of the ecosystem for a couple of decades.

The first research in this direction — for making code less brittle and easier to scale — was made in 1999 at HP Labs, but this solution started to become popular only six years later, when the term « Micro-Web-Services » was introduced by Peter Rodgers.

Then, in almost the same year, in 2006, when Google began working on Linux control groups that isolated the resource usage of a collection of processes, there was the first step in creating virtualization, which would eventually become containerization.

Where microservices architecture promises that every class could become a service, containerization offers to deploy them independently with the exact needed requirements.

Today, we are in the first decade when the coupled microservices architecture/containerization infrastructure is mature and widely used in the industry, and its adoption continues to grow every year. Industry analysts predict that the global microservices architecture market size will increase at a compound annual growth rate, reaching more than $2 billion in a matter of years.

As a reader of this year's "Microservices and Containerization" Trend Report, you are probably familiar with the benefits of this architecture — maybe you have even used it in one of your projects already. Experts from the DZone community will share their diverse opinions, analyses, and perhaps some unexpected insights that, we hope, will allow you to gain a better understanding of microservices and containers standards in 2022.

Sincerely,

Guillaume Hugot

Guillaume Hugot, Director of Engineering at DZone
@guillaumehugot on LinkedIn

Guillaume Hugot is a 15-year experienced engineer specialized in web technologies and the media industry, and he is part of DZone as the head of engineering. At DZone, Guillaume conducts project developments and ensures we always deliver the best experience to our site visitors and our contributors.

ORIGINAL RESEARCH

Key Research Findings

An Analysis of Results from DZone's 2022 Microservices Survey

By John Esposito, PhD, Technical Architect at 6st Technologies

From May–June 2022, DZone surveyed software developers, architects, and other IT professionals in order to understand how microservices are being developed and deployed.

Major research targets were:

1. Expected vs. actual benefits and pains of adopting microservices
2. Effect of microservices on various software engineering *desiderata*
3. Relation of microservices implementation experience to general software design principles

Methods: We created and distributed a survey to a global audience of software professionals. Question formats included multiple choice, free response, and ranking. Survey links were distributed via email to an opt-in subscriber list, popups on DZone.com, the DZone Core Slack Workspace, and LinkedIn. The survey was open June 14–29, 2022 and recorded 346 responses.

In this report, we review some of our key research findings. Many secondary findings of interest are not included here.

Research Target One: Expected vs. Actual Benefits and Pains of Adopting Microservices

Motivations:

1. The buzz around any buzzword exerts non-technical social pressure toward doing the thing signified by the buzzword. Moreover, in engineering, deciding whether to do a thing always involves tradeoffs. Deciding to do a new thing involves imagination; deciding to reject an old thing involves arguments to keep it. Microservices still generate buzz, so software professionals will feel pressure to consider using them. We wanted to know how people who have implemented at least one microservice feel before and after implementation.

2. The hard problems posed by distributed systems — and therefore by microservices — are now increasingly handled by mature support mechanisms (Kubernetes, API gateways, programmable infrastructure, etc.). We imagine that the hard problems posed by microservices will be increasingly abstracted away, and we wanted to begin studying whether this turns out to be the case over time in practice.

3. Nothing ignites an architecture-graph-theoretic flame war more than "fat nodes or fat edges?" We wanted to get a current picture of software professionals' experience with the latest edge- or system-thickening architectural fashion (à la service-oriented architecture [SOA] a decade ago).

EXPECTED VS. ACTUAL BENEFITS OF MICROSERVICES

First the carrot: What attractions do microservices offer, and how much do they deliver on these promises? Or more negatively: We wanted to know which supposed benefits of microservices were overhyped. So we asked:

Rate the following benefits of microservices by how much benefit you expected microservices to provide (left column) vs. how much benefit microservices actually provided (right column):

Observations (on results, n=299):

1. With one exception (discussed below), the reported actual benefits of microservices were equal to or slightly less than the expected benefits:
    - Actual benefits were **equal to** expected benefits of reusability, fault isolation, and runtime independence.
    - Actual benefits were **less than** expected benefits of bounded context facilitation, cloud-native functionality, decentralized governance, fault tolerance, flexibility, resource isolation, loose coupling, CI/CD facilitation, single

responsibility principle (SRP) facilitation, data isolation, technology freedom, small/manageable codebase per microservice, development independence, scalability, and updateability.

We would be surprised if results were too much otherwise. Any buzzword is likely to be worse than the buzz, and even experienced engineers' (experience-crumbled) hopes regularly exceed grim reality. That the differences were consistently small suggests rather an impressive perceived success rate of microservices across a wide range of possible benefits. Moreover, it is worth noting that respondents' non-disappointment with respect to reusability, fault isolation, and runtime independence cluster around (in the object-oriented metaphor) the "phospholipid bilayer" of the microserviced system's "cell membranes": While fault isolation expectations did not exceed actuals, loose coupling (the "membrane channels") did.

2. Microservices consistently benefited security slightly more than expected. The two explicitly security-related purported benefits of microservices available as predefined survey answers were both reported to have higher actual vs. perceived benefit levels. This was not the case for any other domain.

   We hypothesize two reasons for this. First, access to each microservice is less likely to grant the intruder free access to the rest of the system because subsystem boundaries within monoliths are less likely to be "hard" than boundaries between microservices. Second, because microservices coordinate over narrow (usually HTTP) channels, the points of interaction (i.e., the API contracts) are more likely to receive sustained design thought in a microservices vs. monolith architecture.

   We imagine these two security-related attributes mutually reinforce: A system of microservices has more bulkheads, and the bulkheads themselves are built with more attention and expertise (vs. a monolith).

3. The largest bucket of free-response benefits reported (12 of 34) relate to release/DevOps. In future surveys, we will include more granular DevOps-related purported benefits of microservices (or whatever the latest flavor of service-orientated) architecture than the four we included.

EXPECTED VS. ACTUAL PAINS OF MICROSERVICES

Next, the stick: What repellents appear on microservices' surface, and how unpleasant were these purported pains in practice? Or more positively: We wanted to know which fears about microservices were overblown. So we asked:

Rate the following pains of microservices by how much pain you expected microservices to cause (left column) vs. how much pain microservices actually caused (right column):

Observations (on results, n=282):

1. Many fewer pains caused by microservices differed between expectations and reality than benefits promised by microservices differed between expectations and reality:

   - Actual pains were **equal to** expected pains of heavyweight service contracts, API versioning, complexity, decentralized access control, uncoordinated CI/CD, endpoint proliferation, hidden complexity, data consistency, performance overhead, query complexity, performance testing, distributed storage heterogeneity, cascading failures, design complexity, service coordination, logging, debugging, and source repository/package complexity.
   - Actual pains were **less than** expected pains of communication heterogeneity, integration testing, distributed transactions, and source repository/package complexity.
   - Actual pains were **greater than** expected pains of operational complexity.

   From this greater overlap between actual and expected pains of microservices vs. overlap between actual and expected benefits of microservices, we conclude that the pains of microservices are better understood by software professionals, and that the positive buzz may still have space to do some good.

2. The simpler problems appear to be better understood, while the deeper/fuzzier problems appear to inspire some unwarranted fear. Any developer who has tried to implement an e-commerce system over the web — let alone wrestled with CAP at the level of a database management system — is likely to react strongly to greater distributedness of a proposed architecture, and (in our experience) masters of source/package management are many fewer than skilled developers.

   We imagine that fear of source/package and integration testing complexity will decrease as source and test management tools mature, but we do not expect that trepidation over distributed transactions will ever be cleanly abstracted away (just ask the sub-cerebral complexities of spinal motor control).

Research Target Two: Effect of Microservices on Various Software Engineering *Desiderata*

Motivation: To any engineering question, "Is X a good idea?" the answer is always some variant of "It depends." And as any sufficiently senior engineer knows the correct follow-up question is not "Depends on what?" but rather, "What are you trying to accomplish?" So we wanted to know how microservices affect some things that software engineers want — specifically, feature velocity, performance, software quality, and technical debt.

1. Feature velocity, because:
    - Microservices promise to decouple different teams' development timelines.
    - Microservices, in principle, require harder boundaries between subsystems, which reduces coupling and thereby up-front complexity, slowing feature development.
2. Performance, because:
    - Network communication is usually slower and less reliable than intra-process/runtime/application-container communication (a threat to performance posed by microservices).
    - Performance bottlenecks are theoretically harder to isolate in more monolithic architecture (an opportunity for microservices to improve performance).
3. Software quality, because:
    - Service-orientation seems to encourage higher-level, less procedural, more modular design thinking.
    - Well-defined interfaces between services should make meaningful integration tests easier to write.
    - Delayed refactoring (facilitated by heavy decoupling) may result in lower software quality over time.
4. Technical debt, because:
    - The small size of each microservice makes technical debt more likely to decrease patently (because any given part of a system need not drag down other parts, like an individualist Californian) but increase latently (because the independent growth of each microservice may result in emergent fractioning of the system, like Dutch mathematics).

PERCEIVED EFFECT OF MICROSERVICES ON FEATURE VELOCITY

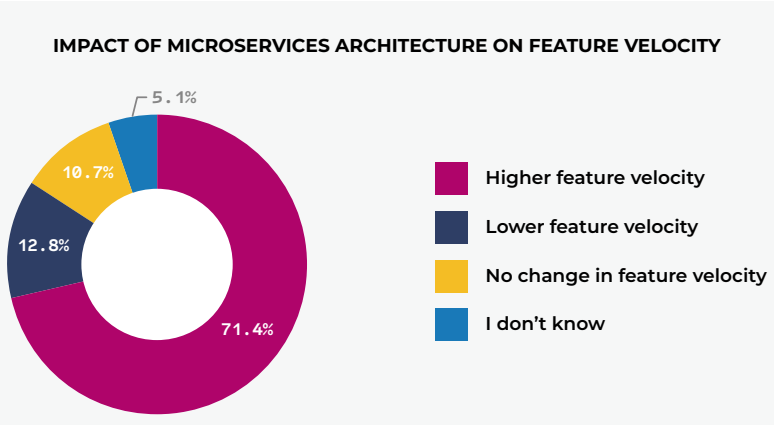
Scenario one: *I'm building a business intelligence system. And you're saying I need to wait for the database folks to upgrade MySQL? Ugh, well, push back that BI development another six months.* ***Scenario two:*** *I'm building a business intelligence system. And you're saying I can just send you some arbitrary GraphQL query and not care what the database folks are doing? Sweet, expect your BI on schedule.*

The first, they say, is your brain on monoliths. The second, the story goes, is your brain on microservices. That is, decoupling means that microservices should, in theory, result in increased feature velocity. We wanted to find out if this is what actually happens. So we asked:

In your experience, adopting a microservices architecture has resulted in: {Higher feature velocity, Lower feature velocity, No change in feature velocity, I don't know}

Results (n=336):

Figure 1



Observations:

1. Respondents overwhelmingly (71.4%) reported that microservices adoption does result in higher feature velocity. This is a powerful vindication of the release cadence promise of microservices.

   Of course, one might eyebrow-cock a quiz: *But what if those features were released prematurely?* One disadvantage of radical decoupling is that nobody takes a system-wide view, which makes side effects more likely. This is possible, but the not-quite-as-but-still-impressively-strong perceived positive effect of microservices on software quality enervates such an objection.

2. Moreover, when microservices increase feature velocity, post-deployment incidents tend to decrease. Only 15.9% of respondents who reported that microservices increase feature velocity reported incidents or rollbacks almost every deployment vs. 25.6% of respondents who reported that microservices decrease feature velocity.

   This vindicates microservices and continuous delivery together: Microservices facilitate a technique (rapid releases) that (as shown in other research published here and elsewhere) tend to decrease incidents without incidentally taking away that technique's power.

3. Affirmation of software ↔ organizational isomorphism (Conway's Law) correlates with increase in feature velocity as a result of adopting microservices. Specifically, 39.7% of respondents who reported that microservices result in higher feature velocity agree with Conway's Law, and 26.2% agree strongly, while 34.9% of respondents who reported that microservices result in lower feature velocity agree, and 23.3% agree strongly.

   The overlap suggests, loosely and in proportion to respondents' omniscience, that the impact of microservices on feature velocity is related to organizational interactions of microservices.

4. Experience personally designing microservices also correlates with increase in feature velocity as a result of adopting microservices. 77.3% of respondents who reported having designed microservices personally also reported higher feature velocity as a result of microservices adoption vs. only 51.7% of respondents who haven't personally designed microservices.

   Since deeper technical experience generally implies greater understanding, we are more inclined to accept the higher feature velocity correlation claimed by the personally experienced microservices developers.

5. The impact of microservices on feature velocity can perhaps be bottlenecked by reliance on an external database to maintain application state — a common "we-cannot-really-afford-to-trust-consensus" pattern that we have observed in enterprise software development. That is, 37.1% of respondents who reported no change in feature velocity as a result of microservices adoption reported that they also rely on an external database to maintain application state a little too often vs. 19.6% of respondents who reported higher feature velocity (and lower for other velocity changes). Respondents who reported higher feature velocity from microservices are also significantly more likely to report reliance on an external database for application state just the right amount (40% vs. only 8.6% of respondents who reported no change in feature velocity).

   We hypothesize that a fuller, if more technically challenging, commitment to distributed design might multiply the feature-velocity-increasing benefits of microservices.

PERCEIVED EFFECT OF MICROSERVICES ON PERFORMANCE ENGINEERING

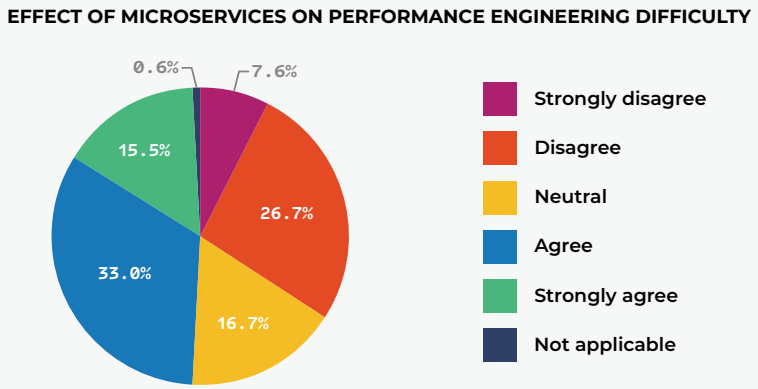
The variability of how distributedness effects performance was too high for us to survey with much confidence: Fewer request/response cycles may slow performance (noting that the handshake thrash decreases quite a lot after HTTP/1.1), but looser coupling allows more leeway for each microservice to optimize independently. So while we were interested in performance of microservices, we chose to focus on the difficulties that microservices pose for performance engineering rather than actual performance.

In a monolith, one can look at one metric of time spent querying and compare that with a second metric of time spent doing arithmetic. However, in a set of interacting microservices, each service has its own time spent querying and time spent doing arithmetic. And the harder boundaries between each distinct service make simple performance-analytical abstraction over all services (e.g., summing database query time over all services – something that is tricky enough even over a cluster of identical containers) slouch toward the worst kind of iffiness. So *prima facie*, we might guess that microservices performance is harder to tune. To test our hypothesis against the broader software development world's experience, we asked:

Agree/disagree: Microservices make performance engineering, tuning, and monitoring more difficult. {Strongly agree, Agree, Neutral, Disagree, Strongly disagree, Not applicable}

Results (n=330):

Figure 2



Observations:

1. Just over half of respondents (56.1%) think that microservices make performance engineering, tuning, and monitoring more difficult; only about a quarter (26.7%) disagree. The general opinion of software professionals is clearly on the "fear performance tuning microservices" side.

2. This risk-skewed picture is reinforced by segmenting respondents into those who have personally designed microservices and those who have not. 33.8% of respondents who reported having personally designed microservices disagree or strongly disagree that microservices make performance engineering difficult vs. 38.6% of respondents who have not.

   The difference is small but significant: Experience with microservices tends to strengthen the impression that optimizing microservices performance is hard.

3. Again, affirmation of software ↔ organizational isomorphism correlates with affirmation of the difficulty to performance tuning posed by microservices. Of those who agree with Conway's Law, 35.7% agree that microservices impact performance engineering and 20.8% strongly agree vs. only 20.6% agree and 2.9% strongly agree, respectively, of those who disagree with Conway's law.

   Because Conway's Law has only very high-level technical teeth, we suppose that the orthogonality of performance tuning to microservices nodes — one does not have a dedicated "performance microservice" except perhaps for monitoring — suggests some organizational impedance mismatch (apart from the technical reasons for performance worries outlined in the introduction to this question above). Perhaps the domain-specific specialization permitted by microservices does not map well to the specialization of performance engineering.

PERCEIVED EFFECT OF MICROSERVICES ON SOFTWARE QUALITY

Different systems have varying takes on how boundaries affect proximity to some ideal:

> *Surely modularity makes software better. –The Cell That Invented Organelles*
>
> *Separation of concerns is a luxury, and in the long run, luxury is bad design. – Also Every Biological System*
>
> *Everything that does not absolutely need to be called by some other class absolutely must be declared private. Also, have you ever heard of our lord and savior, Dependency Injection? –Some Annoying PR Reviewer*
>
> *Private, schmivate; here's an underscore. –The JavaScript Language*
>
> *How are you doing? –Some Total Stranger on Zoom*

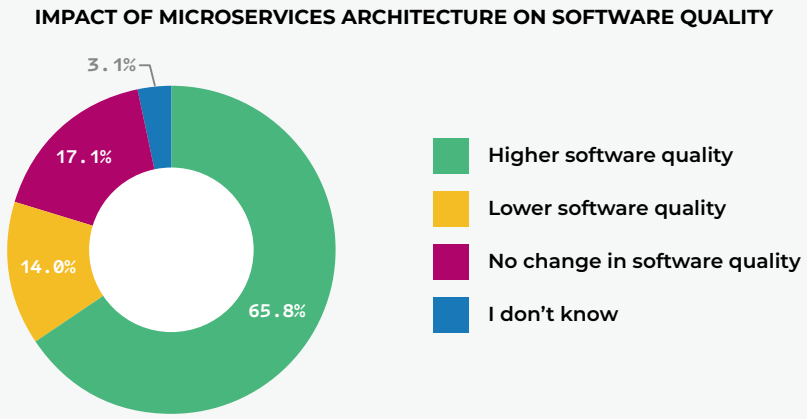
We suppose that microservices might have ambivalent effects on software quality. On the one hand, separation of concerns (to our OO-indoctrinated brains anyway) seems like generally a good thing. Again, clean API design results in better integration tests — which should, theoretically, make more accessible the sweet middle ground between trivial "does this *concat(a,b)* method in fact return *ab*" overly unit-y unit tests and cosmic "if I click this button does every document in this multinational corporation get TF-IDFed after 52 hours" overly integration-y integration tests. On the other hand, in our experience, high-level architectural paradigms do less to improve software quality than a good craftsperson's commitment to excellence and attention to detail.

We wanted to see how software professionals at large perceive the relation between microservices and software quality, so we asked:

In your experience, adopting a microservices architecture has resulted in: {Higher software quality, Lower software quality, No change in software quality, I don't know}

Results (n=322):

Figure 3



Observations:

1. Nearly two thirds of respondents (65.8%) reported that adopting microservices resulted in higher software quality. This is quite an endorsement. In a field as complex and volatile as software engineering, any paradigm shift is likely to have diffuse effects on something as broad as "quality." In future surveys, we will attempt to dive deeper into causes.

2. Experience designing microservices correlates positively with the relation of microservices adoption and software quality: 69.4% of respondents who have personally designed microservices reported that microservices adoption resulted in higher quality vs. 53.7% of respondents who have not personally designed microservices. Similarly, 24.1% of respondents who have not personally designed microservices reported that microservices adoption resulted in lower software quality vs. 11% of respondents who have personally designed microservices.

    Here, however, we must be extra careful: We imagine it might be psychologically more difficult for someone who has designed something to suppose that it resulted in lower software quality — a general and highly charged metric for anyone interested in excellence, as any good engineer is — than for that person to suppose that what they designed resulted in (for instance) lower feature velocity, which is not intrinsically tied to technical excellence.

PERCEIVED EFFECT OF MICROSERVICES ON TECHNICAL DEBT

As anyone who feels guilty over their mortgage knows, carrying debt is not necessarily low-quality finance. Similarly, carrying technical debt is not necessarily low-quality software development. However, as anyone with a spreadsheet and horrific student loan debt knows, compounding debt interest can get out of hand faster than you likely imagined. And as anyone in an enterprise development (or consulting) role knows, the same is true of technical debt.

We suppose microservices, in principle, are likely to have ambivalent effects on technical debt: lower up front (because one microservice's internal decisions affect other services' decisions minimally), but perhaps easier to spiral out of control in the long run (because the abstraction leakage from each microservice is invisible on its own but compounds in the aggregate).

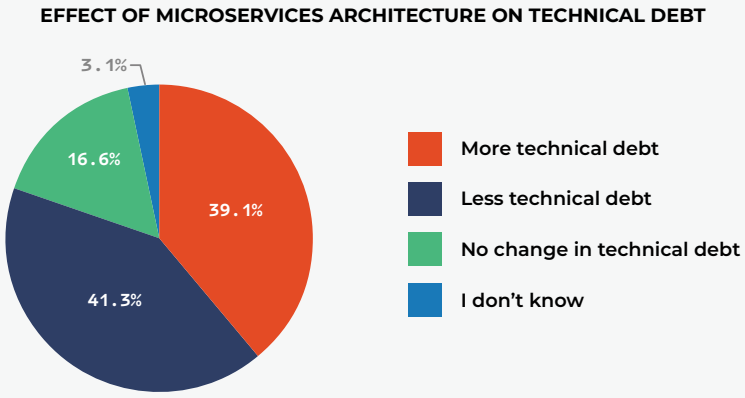
We wanted to know if this matches real-world experience with microservices, so we asked:

In your experience, adopting a microservices architecture has resulted in: {More technical debt, Less technical debt, No change in technical debt, I don't know}

Results (n=320):

SEE FIGURE 4 ON NEXT PAGE

Figure 4



Observations:

1. Our hypothesis about the ambivalence of microservices' effect on technical debt was confirmed: The difference between "more technical debt" and "less technical debt" responses was minimal (39.1% vs. 41.3%, respectively). Together, however, these responses make up a large majority (80.4%), so it seems highly likely that microservices have some effect on technical debt. In future surveys, we intend to ask separate questions about up-front vs. long-term technical debt.

2. Interestingly, no major differences appeared when we segmented answers to the technical debt question by whether respondents have mostly built microservices in greenfield environments or split existing systems into microservices. Mostly-brownfield respondents were slightly more likely to report an increase in technical debt as a result of microservices adoption, but the difference is only 3.9%.

   This lack of coupling encourages the hope that refactoring into microservices can be successful, while also denying that microservices adoption by itself magically washes away technical debt.

3. Junior respondents (≤ five years' experience as a software professional) were somewhat more likely to report that microservices resulted in less technical debt (48.2% vs. 40.6% of senior respondents), and the difference is balanced by a greater percent of senior respondents reporting no change in technical debt as a result of microservices adoption (19.3% vs. 8.9%).

   We are not sure how to interpret these results. It seems improbable that junior respondents are simply more likely to understand microservices better than senior respondents, many of whom in the latter group are likely to have built software when SOA was already cliche. Junior respondents were more sanguine about microservices' effect on other software engineering *desiderata*, also reporting higher software quality post-microservices. But we can also imagine that junior respondents might have a bias toward refactoring since as the size of the existing world codebase grows, the amount of refactoring required grows over time — possibly at geometric rates, but the relevant condition applies even if growth is linear. We intend to address this broader question in future research (on refactoring in general).

Research Target Three: Relation of Microservices Implementation Experience to General Software Design Principles

Motivations:

1. No abstraction is airtight; therefore, microservices should impact other aspects of software design. And accordingly, there should be some intelligible relation between experience with microservices and the use of many other approaches to software engineering.

2. Service orientation is a network-level expression of the general principle of "careful boundary definition" that is exemplified especially by object-oriented and domain-driven design paradigms. We wanted to see if microservices manifest some kind of "fractal" thinking about software design from class to microservices-aggregate level.

3. The "degrees of atomicity" problem is, in principle, greatly complicated by microservices. For instance, simple database snapshots-plus-rollbacks are enough to enforce transactional atomicity in a monolith — allowing the database engine to handle the physical difficulties of juggling any "undo" queue — but not in a microservice, which has no in-built integrity-defining algebra. We wanted to understand how software professionals approach this problem.

USE OF WORKFLOW AND ORCHESTRATION ENGINES

If processes are like services, then what are operating systems like? The relevant solution categories: workflow and orchestration engines. Perhaps a given world is best modeled as a swarm of Erlang processes, or perhaps it is best modeled as an infantry phalanx with a well-trained officer corps. Workflow and orchestration engines span the control spectrum from the former to the latter. We wanted to know where software professionals' work falls on that spectrum, so we asked:

Does your organization use a workflow or orchestration engine for state handling, monitoring, and reporting? {Yes, No, I don't know}

Results (n=316):

Table 1

USE OF WORKFLOW OR ORCHESTRATION ENGINES

	Percent	n=
Yes	69.0%	218
No	24.7%	78
I don't know	6.3%	20

Observation: Broadly speaking, software professionals have not hand-waved the "operating system" role to "emergent properties of the microservices system": 69% of respondents' organizations use a workflow or orchestration engine for state handling, monitoring, and reporting.

MICROSERVICES AND OBJECT-ORIENTED ANALYSIS AND DESIGN

As we have noted several times above, object-oriented design and microservices architectures share the "private first" concept. We expect that some relation between use of object-orientation (OO) and microservices adoption is likely to obtain.

We asked:

How often do you take the following approaches to software development and design? {Test-driven development (TDD), Behavior-driven development (BDD), Domain-driven design (DDD), Object-oriented analysis and design (OOAD)}

Results (n=332):

Figure 5

USE FREQUENCY OF APPROACHES TO SOFTWARE DEVELOPMENT AND DESIGN

	Never	Rarely	Sometimes	Often	Always	n=
Test-driven development (TDD)	6.6%	11.7%	38.3%	29.8%	13.6%	332
Behavior-driven development (BDD)	11.6%	20.1%	32.5%	28.3%	7.6%	329
Domain-driven design (DDD)	5.8%	13.9%	31.8%	33.6%	14.8%	330
Object-oriented analysis and design (OOAD)	5.3%	10.9%	25.2%	34.2%	24.5%	332

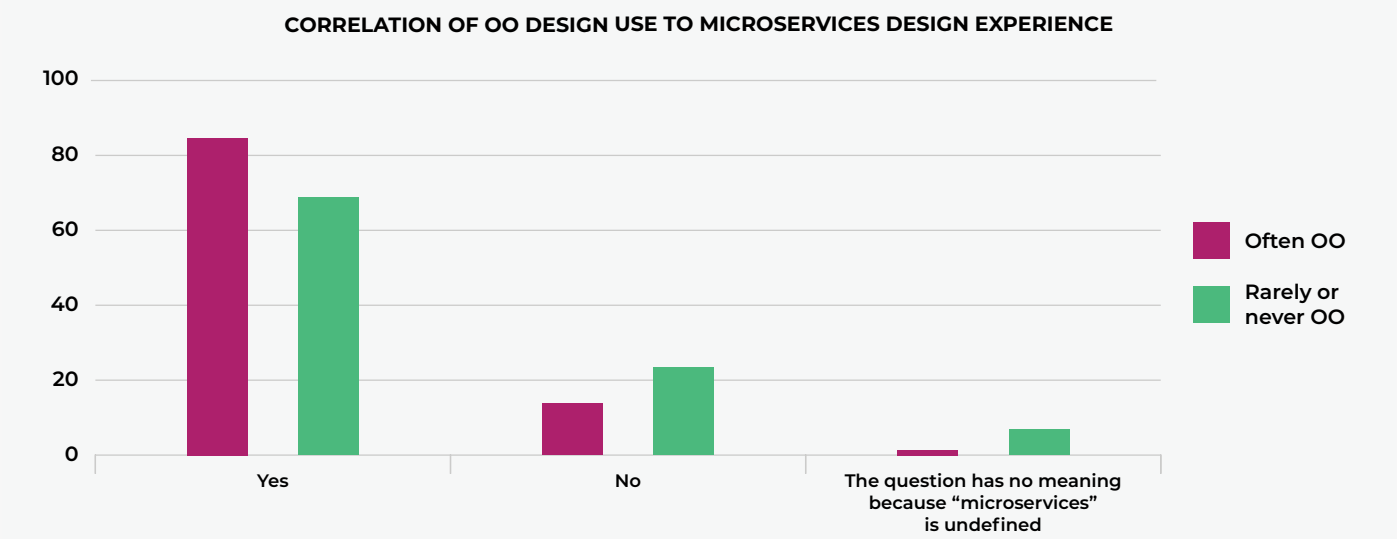
We used these results as a baseline for segmentation against answers to questions regarding microservices-specific experiences. We record only the most interesting results here but welcome interest in other correlations at publications@dzone.com.

CORRELATION BETWEEN OO DESIGN AND PERSONAL INVOLVEMENT IN MICROSERVICES DESIGN

Respondents who use object-orientation often are significantly more likely to design microservices architectures than respondents who use object-orientation rarely or never:

SEE FIGURE 6 ON NEXT PAGE

Figure 6

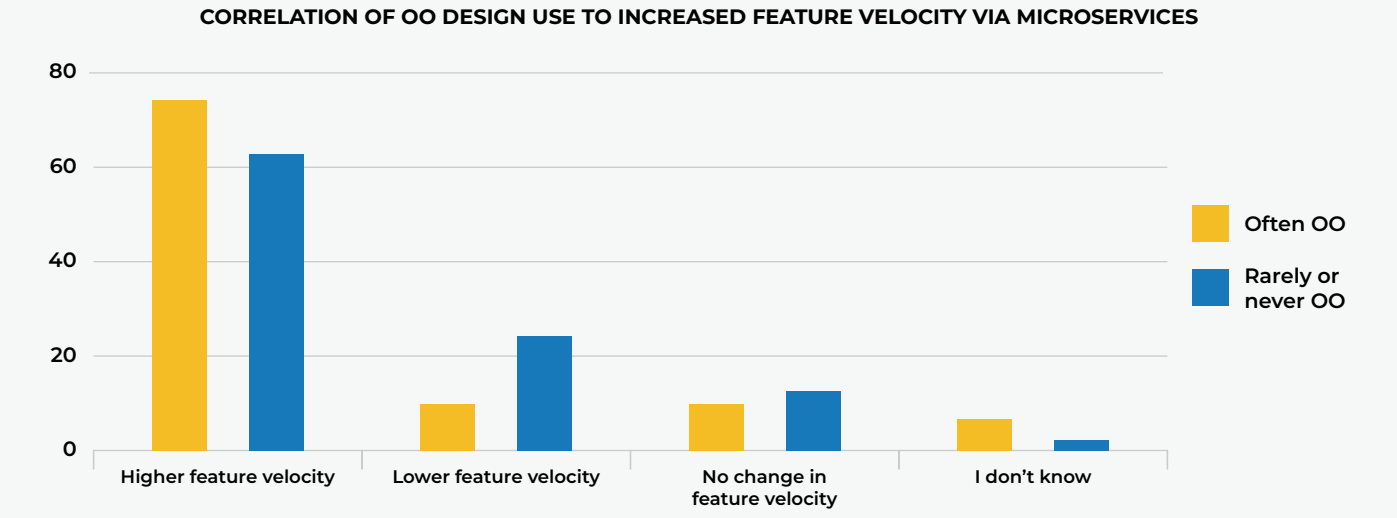


This result is as we anticipated, according to the OO-microservices homology: Significantly more respondents who often use OO have personally designed microservices (84.5% vs. 68.6% of those who rarely or never use OO).

CORRELATION BETWEEN OO DESIGN AND MICROSERVICES-CAUSED FEATURE VELOCITY INCREASE

Respondents who use object-orientation often are significantly more likely to report that adopting a microservices architecture resulted in higher feature velocity than respondents who use object-orientation rarely or never:

Figure 7



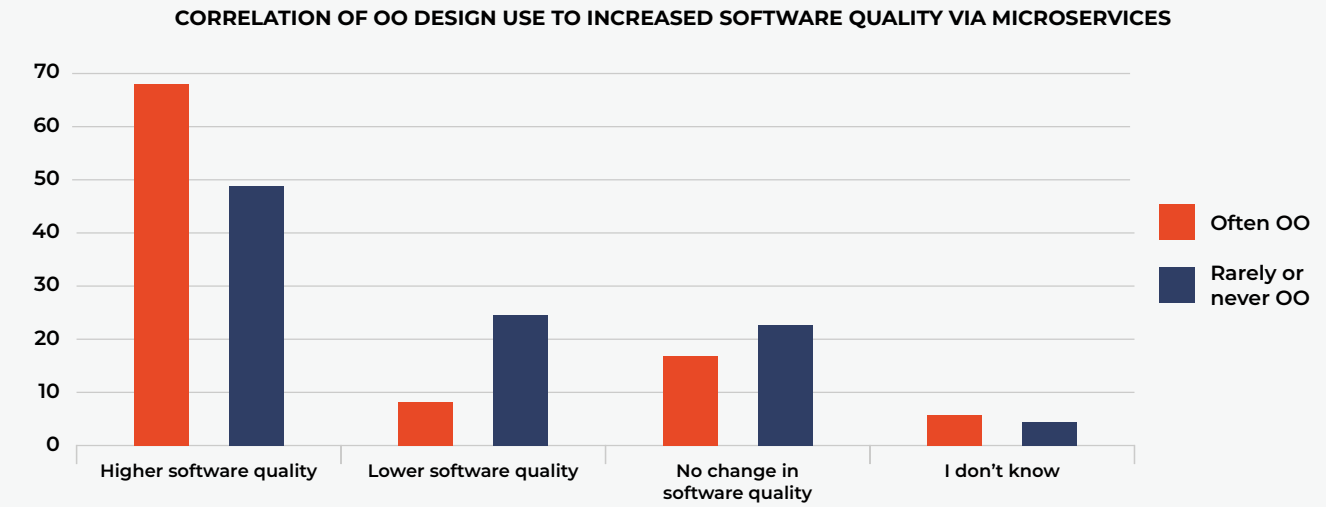
We take this to suggest that object-oriented thinkers are more likely to deliver features faster after adopting microservices because microservices are objects writ large, so technical facility with object decoupling is likely to transfer to technical facility with service decoupling.

CORRELATION BETWEEN OO DESIGN AND MICROSERVICES-CAUSED SOFTWARE QUALITY INCREASE

Again, respondents who use object-orientation often are significantly more likely to report that adopting a microservices architecture resulted in higher software quality than respondents who use object-orientation rarely or never:

SEE FIGURE 8 ON NEXT PAGE

Figure 8



The apparent impact of high OO experience on the software quality increase caused by microservices (68.5% vs. 49% of rarely or never OO respondents) is even stronger than other OO-microservices correlations. We take the extreme vagueness of software quality as an effect of OO-microservices homology at the paradigmatic level (while impact on feature velocity might be a more specific technical effect of, say, SOLID design "trickling up").

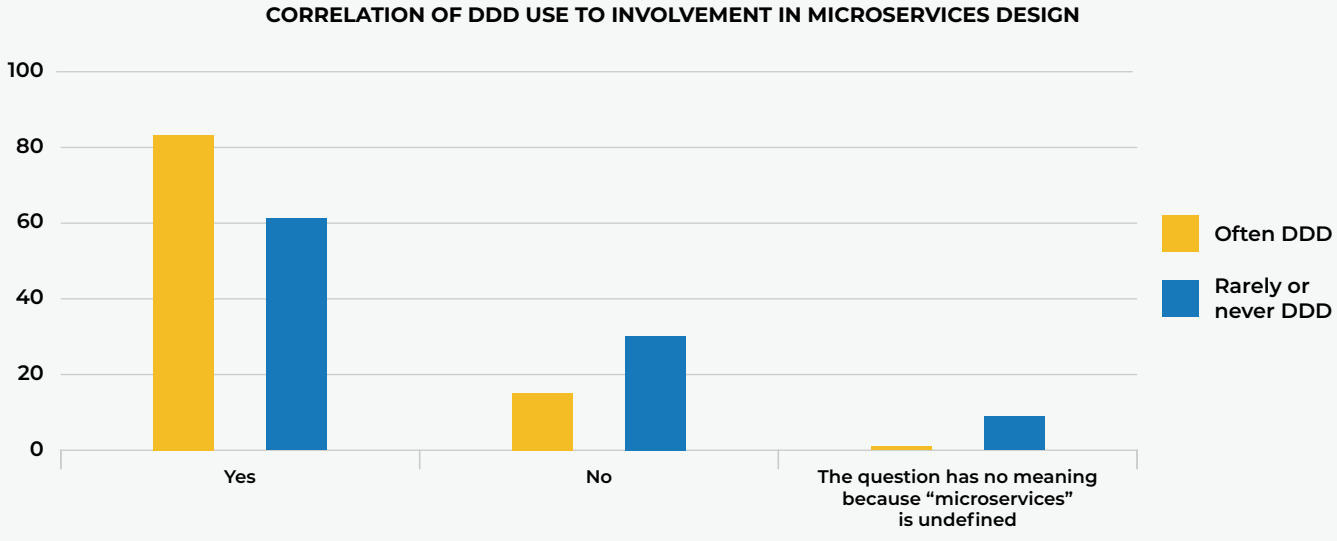
MICROSERVICES AND DOMAIN-DRIVEN DESIGN

Domain-driven design (DDD), in theory, tends to encourage both technical- and team-level aspects of microservices. As for OOAD, we wanted to know whether experience in DDD correlates with experience building microservices and with effects of microservices after adoption. In fact, we observed many of the same correlations.

CORRELATION BETWEEN DDD AND PERSONAL INVOLVEMENT IN MICROSERVICES DESIGN

Respondents who use domain-driven design often are significantly more likely to design microservices architectures than respondents who use domain-driven design rarely or never:

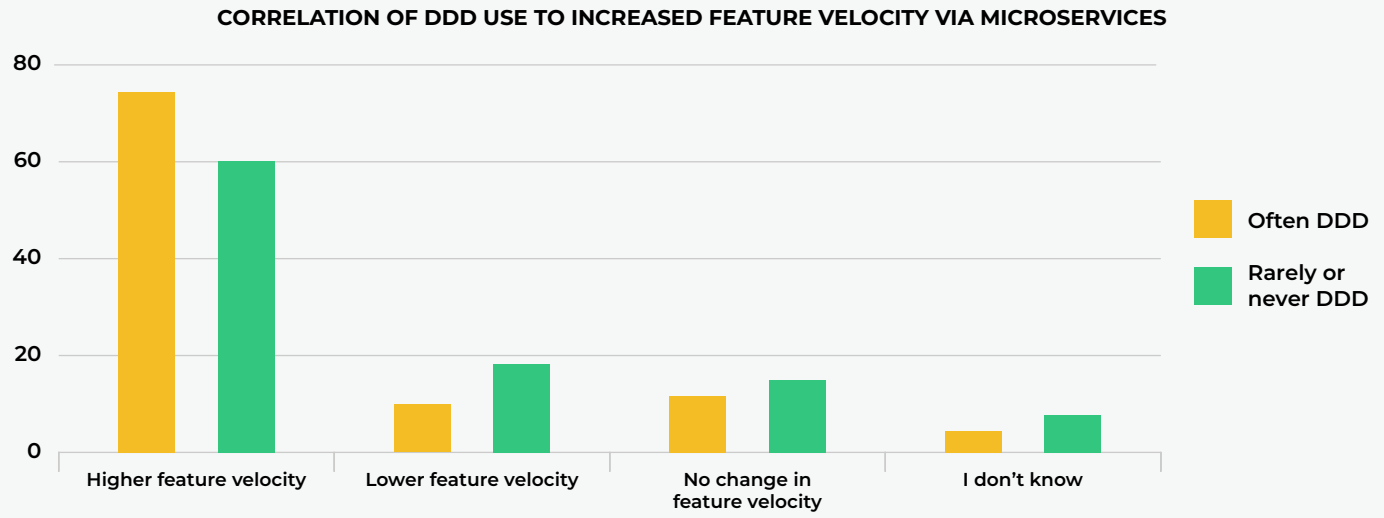
Figure 9



CORRELATION BETWEEN DDD AND MICROSERVICES-CAUSED FEATURE VELOCITY INCREASE

Respondents who use domain-driven design often are significantly more likely to report that adopting a microservices architecture resulted in higher feature velocity than respondents who use domain-driven design rarely or never:

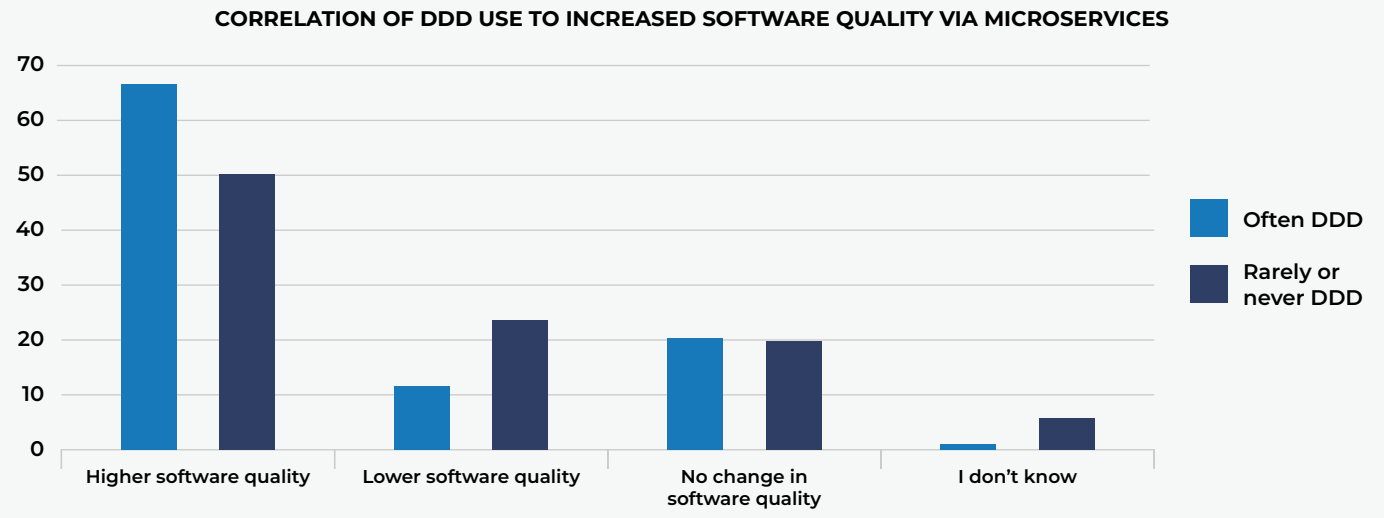
Figure 10



CORRELATION BETWEEN DDD AND MICROSERVICES-CAUSED SOFTWARE QUALITY INCREASE

Again, respondents who use domain-driven design often are significantly more likely to report that adopting a microservices architecture resulted in higher software quality than respondents who use object-orientation rarely or never:

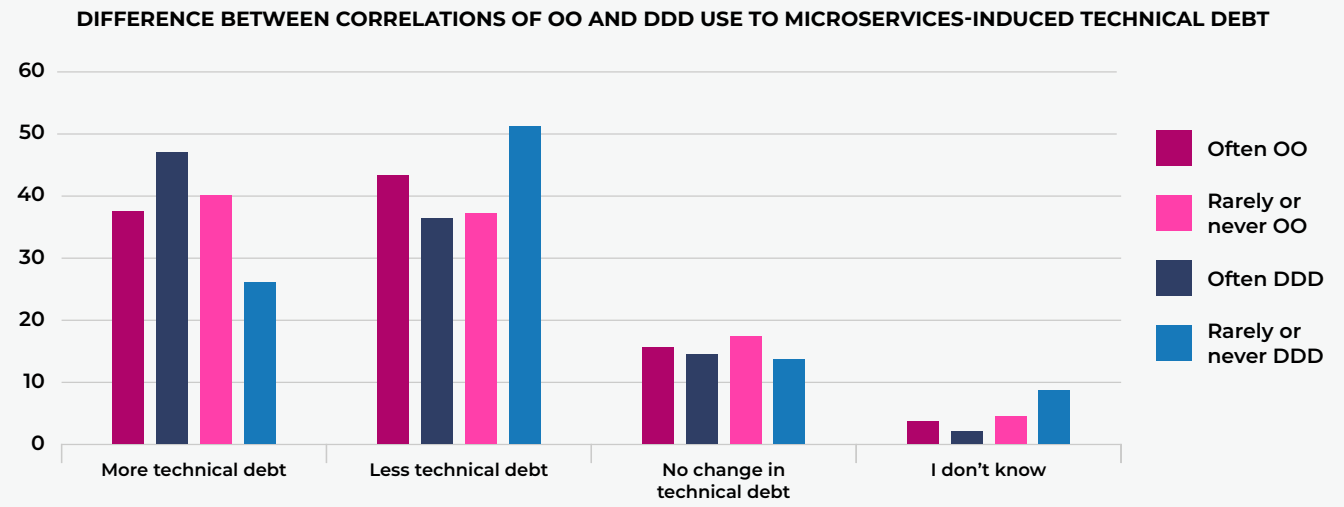
Figure 11



In one case, however, the relation of OO vs. DDD to microservices adoption diverged:

SEE FIGURE 12 ON NEXT PAGE

Figure 12



As a result of adopting microservices, DDD-experienced respondents were significantly more likely to report more technical debt than OO-experienced respondents. And DDD-inexperienced respondents were significantly more likely to report less technical technical debt than OO-inexperienced respondents. We imagine the reason being that many of microservices' benefits were already available to DDD experts, and that as a result, the distributed complexity of microservices relatively overwhelms the comparatively smaller benefit squeezed by microservices out of the DDD-pre-squeezed system.

Further Research

Although many of our previous surveys have touched on microservices, this was our first survey since 2018 to focus on microservices in particular. In this survey, we focused on higher-level correlations; in future research, we aim to extend our analysis to a lower, intra-service level. Our survey included material not published in this report, much of which is of interest at that lower intra-service level in relation to higher-level effects of microservices adoption, including use of distributed design patterns, twelve-factor app principles, and SOLID OO design principles; container design principles; organizational attitudes toward Docker in particular; and implementation of consensus protocols.

Please contact publications@dzone.com if you would like to discuss any of our findings or supplementary data.

John Esposito, PhD, Technical Architect at 6st Technologies

@subwayprophet on GitHub | @johnesposito on DZone

John works as technical architect, teaches undergrads whenever they will listen, and moonlights as a research analyst at DZone. He wrote his first C in junior high and is finally starting to understand JavaScript NaN%. When he isn't annoyed at code written by his past self, John hangs out with his wife and cats, Gilgamesh and Behemoth, who look and act like their names.

vFunction

About vFunction Platform

vFunction's application modernization platform is designed for IT organizations—from the CIOs who answer to the business to the architects and developers who manage applications—to rapidly and incrementally modernize their legacy application portfolios with little risk. Partnering with Microsoft, AWS, HPE, and leading Systems Integrators, vFunction's award-winning, patented technology is the first and only AI for App Modernization platform, empowering IT teams to decompose monolithic applications into microservices.

vFunction Assessment Hub

vFunction Assessment Hub analyzes the technical debt of a company's monolithic applications, accurately identifies the source of that debt, and measures its negative impact on innovation. The AI-powered solution measures app complexity based on code modularity and dependency entanglements, measures the risk of changes impacting stability based on the depth and length of the dependency chains, and then aggregates these to assess the overall technical debt level. It then benchmarks debt, risk, and complexity against the organization's own estate, while identifying aging frameworks that could pose future security and licensing risks. vFunction Assessment Hub integrates seamlessly with the vFunction Modernization Hub which can directly lead to refactoring, re-architecting, and rewriting applications with the full vFunction Platform. It is a free for one app up to one year, available at vFunction.com/Trial.

vFunction Modernization Hub

vFunction Modernization Hub is an AI-driven modernization solution that automatically transforms complex monolithic applications into microservices, restoring engineering velocity, increasing application scalability, and unlocking the value of the cloud. Utilizing both deep domain-driven observability via a passive JVM agent and sophisticated static analysis, vFunction Modernization Hub analyzes architectural flows, classes, usage, memory, and resources to detect and unearth critical business domain functions buried within a monolith. Whether your application is on-premise or you have already lifted and shifted to the cloud, the world's most innovative organizations are applying vFunction on their complex "megaliths" (large monoliths) to untangle complex, hidden, and dense dependencies for business critical applications that often total over 10 million lines of code and consist of 1000's of classes. See more at vFunction.com/Demo.

Request Demo

PARTNER CASE STUDY

Case Study: Fortune 100 Bank*

With over $2 trillion in assets, this Fortune 100 financial services provider is one of the largest asset holders in the world.

Challenge

The company processes around $1 billion per day, mainly through legacy systems built with Java EE 6 and Oracle's WebLogic web server. Their largest and most complex application is a 20-year-old monolith with over 10,000 classes and 8 million lines of code (LoC).

With a mandate to become cloud-ready, they needed to ensure their future application architecture was architected to work in a complementary way to take advantage of cloud-native services. This meant refactoring, which is extremely difficult and time consuming for humans, but made much easier with automation, AI, and data science.

Solution

The client turned to vFunction to automate the analysis of the company's legacy Java applications, assessing the complexity of selected apps to determine readiness for modernization. This included deep tracking of call stacks, memory, and object behaviors from actual user activity, events, and tests. This analysis uses patented methods of static analysis, dynamic analysis, and dead code detection.

This enabled them to better preview refactorability, stack rank applications, estimate schedules, and manage the modernization process to accelerate cloud-native migrations. Fortune 100 Bank used vFunction to accelerate Java monolith decomposition.

Results

- **25x reduction in time to market** – Within just a few weeks of installing vFunction, the company was able to unlock and take action on never-before-seen insights about their largest Java monolith — after years of efforts.
- **3x reduction in cost of modernization** – AI-driven insights and actionable recommendations reduced the cost of modernization by 3x compared to manual decomposition and refactoring efforts.
- **ROI on the horizon for internal cloud platform** – The combination of vFunction and the company's cloud ecosystem built on AWS has enabled them to move forward with their strategy of refactoring and migrating hundreds of legacy Java applications to the AWS cloud.

**Undisclosed client name*

COMPANY
Fortune 100 Bank*

COMPANY SIZE
6,000+ employees

INDUSTRY
Financial services

PRODUCTS USED
vFunction Assessment Hub, vFunction Modernization Hub

PRIMARY OUTCOME
The customer was able to refactor and migrate hundreds of legacy Java applications to the AWS cloud faster and less expensively than ever imagined.

"In a matter of weeks, vFunction gave us visibility and insights into some of our most complex applications — saving us significant time and reducing costs.

After years of ultimately unsuccessful efforts trying to refactor and migrate our legacy application portfolio manually, we now have a repeatable, AI-driven model to refactor and modernize for the AWS cloud."

— **Senior Architect**, Fortune 100 Bank

CREATED IN PARTNERSHIP WITH

vFunction

CONTRIBUTOR INSIGHTS

Multi-Cloud Strategies Using Microservices Architecture

A Complete Guide to Building Microservices in Azure, AWS, and Google Cloud

By Boris Zaikin, Senior Software Cloud Architect at IBM/Nordcloud GmbH

In 2005, Dr. Peter Rodgers addressed micro web services during a presentation at a Web Services Edge conference, where the first generation of micro web services was based on service-oriented architecture (SOA). SOA is a "self-contained" software module that performs as a task, and this allows the service to communicate based on SOAP (Simple Object Access Protocol). The main SOAP idea is, "Do the simplest thing possible."

Nowadays, there is no option to avoid talking about microservices during architectural calls, especially if you want to design cloud or multi-cloud, modular, scalable, and multi-user applications. In this article, I will explain microservices and how to design applications based on the multi-cloud scenario. I will walk you through the microservice design pattern and wrap this information into an architectural example.

Microservices and Multi-Cloud Environments

Before we jump into microservices architecture, patterns, and how you can build microservices-based applications that support multi-cloud deployment, let's define the terms. A microservice is an architectural pattern that allows applications to consist of loosely coupled modules. However, microservices designs should follow these rules:

- Each module or microservice has its own data and therefore should have an independent database.
- Each microservice should be developed by its own team. This doesn't mean that microservices-based applications can't be developed by one team, however, having one team per microservice shows how independent microservices can be.
- Microservices should have an independent deployment process.
- Microservices should give better control over resources and compute power so you can scale each service independently and according to the service needs.
- Microservices can be written in different languages, but they should communicate with a single protocol like REST.

Multi-cloud (i.e., hybrid cloud) means there are two different approaches to spreading an app through multiple cloud providers. For example, we can build core applications in AWS, and there are some parts that we can deploy to Azure and Google Cloud. Another multi-cloud example is when an app that was designed for one cloud can be migrated to another with minor changes.

MICROSERVICES VS. MONOLITH

Before we jump into examples, let's look at a brief list of the pros and cons of microservices and monolithic architecture:

Table 1: Microservices

Pros	Cons
• **Scalability** – As we have all separate services, we can scale each service independently. • **Isolation** – A large project may be unaffected when one service goes down. • **Flexibility** – You can use different languages and technology as all services are independent. We can separate the whole project into microservices where each microservice will be developed and supported by a separate team. • **DevOps independence** – All microservices are independent; therefore, we can implement an independent deployment process for each microservice.	• **Complexity** – Microservices architecture can be a good choice for huge, enterprise-level companies and platforms. Take Netflix, for example. There you can separate domains and subdomains to different services and teams. However, for a small-level company, separating may add redundant complexity. Moreover, it can be impossible to separate small projects into microservices. • **Testing** – Testing microservices applications can be a difficult task as you may need to have other services running. • **Debugging** – Debugging microservices can be a painful task; it may include running several microservices. Also, we need to constantly investigate logs. This issue can be partially resolved by integrating a monitoring platform.

When I start designing a solution, I always use monolithic architecture as the starting point. Using this approach, we can achieve the following:

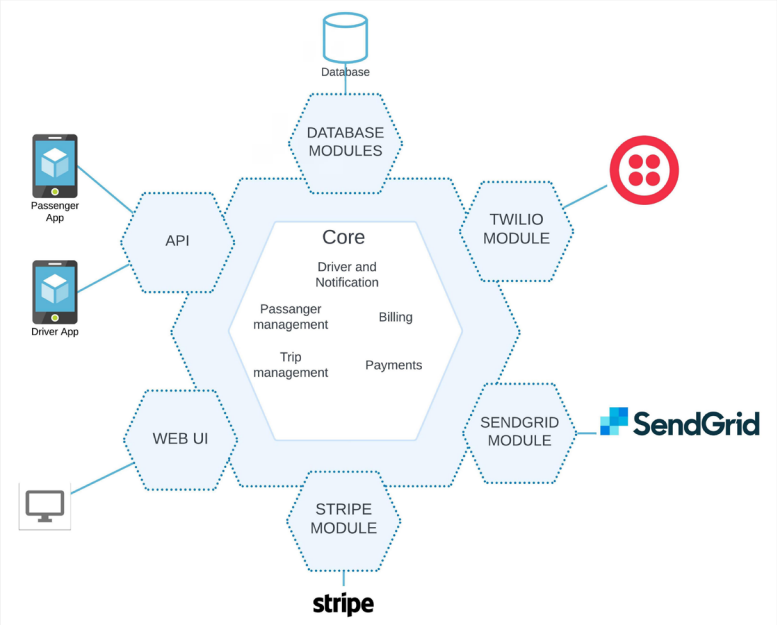
- During design and implementation, we can already see if our application can be moved to microservices.
- We can identify monolithic applications that can be moved to microservices using a step-by-step approach.

Table 2: Monolith

Pros	Cons
• **Simplicity** – Monolithic architecture is relatively simple and can be used as a base architecture or the first microservice step. • **Simple DevOps** – The DevOps process can be simple as we may need to automate one simple application.	• **Vendor lock-in** – With monolithic architecture, we may be locked with one vendor/cloud provider. All modules in monolithic architecture are closely tied to each other. It's hard to spread them across different vendors. • **Inflexible DevOps** – The DevOps process for one enterprise-level, monolithic app can take a lot of time as we need to wait until all modules are built and tested. • **Stick with one programming language/technology** – Monolithic architecture is not too flexible — you need to stick with one technology/programming language. Otherwise, you must rewrite the whole application to change the core technology.

In Figure 1, you can see an example of a typical modular monolithic architecture of a traveling system. It allows passengers to find drivers, book, and join the trip.

Figure 1: Travel booking application



The system consists of several modules:

- UI/front end
- API
- SQL adapter
- Stripe adapter to process payments
- SendGrid to manage emails
- Adapter for Twillio to be able to communicate over the phone (calls, SMS, video)

This monolithic architecture can be migrated to the microservices one. Each service can operate independently and can keep the state in its own database.

The solution explained above is quite simple and can be transformed into microservices without a lot of issues. However, when it comes to the enterprise level, this migration is far more complex, as is seen at large companies such as Netflix.

Cloud Components and Services for Building Microservices Architecture

In this section, we will get into a few cloud services that we can use to build microservices architecture so we know what service or cloud component to choose during microservices architecture design and implementation.

CONTAINER ENGINE

Container engines are essential to building microservices as they allow for separation, orchestration, and management of the microservices within various cloud providers. Docker is a widely used container engine that allows one to wrap each microservice into a container and put it into a cloud-based container orchestration system like Kubernetes (AKS, EKS) or directly spin up the application. Containerd is the same as Docker but has a lightweight and more secure architecture.

ORCHESTRATOR

Kubernetes is a popular open-source system for orchestrating, automating deployment, and scaling the containers apps. It contains and automates the deployment. Azure, AWS, and Google Cloud have their own managed orchestration services that already include load balancing, auto scaling, workload management, monitoring, and a service mesh.

Azure Service Fabric is a distributed platform to orchestrate microservices. For several years, the main goal was to provide the best support for .NET Core/Framework- and Windows-based microservices; we can see it in the selection service flowchart. Microsoft claims that Service Fabric supports other languages and platforms like Linux as well.

MESSAGE BUS

A queue is a service that is based on the FIFO (first in, first out) principle. All message bus systems are based on this service. For example, Azure has queue storage that contains simple business logic. If you have a simple architecture that needs centralized message storage, you can rely on the queue. AWS and Google Cloud also have the Simple Queue Service and GC Pub/Sub, respectively. These allow you to send, store, and receive messages between microservices and software components.

Service Bus/Message Bus is based on the same approach as a queue. However, a Message Bus has more features on top — it contains a dead-letter queue, scheduled delivery, message deferral, transactions, duplicate detection, and many other features. For example, Azure Service Bus and AWS Managed Kafka service are highly available message brokers for enterprise-level applications that can deal with thousands of messages.

SERVERLESS

Serverless allows us to build microservices architecture purely with an event-driven approach. It's a single cloud service unit that is available on demand with an intended focus on building the microservices straight away in the cloud without thinking about what container engine, orchestrator, or cloud service you should use. AWS and Azure have Lambda and Azure Functions, respectively. Google Cloud Functions follows the same principle.

Essential Microservices Design Patterns

Now we understand microservices architecture and can start developing the application using microservices. We also understand the benefits of using microservices and how to refactor applications to support this architecture. The best point to start with building the app is to know microservices design patterns.

DOMAIN MICROSERVICES

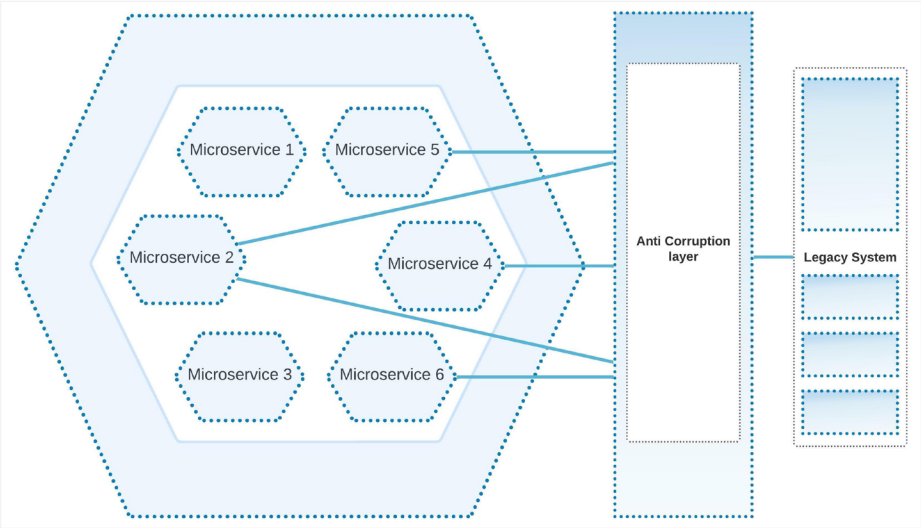
The microservices domain model (part of domain-driven design) is more than a pattern — it is a set of principles to design and scope a microservice. A microservices domain should be designed using following rules:

- A single microservice should be an independent business function. Therefore, the overall service should be scoped to a single business concept.
- Business logic should be encapsulated inside an API that is based on REST.

ANTI-CORRUPTION LAYER

A legacy system may have unmaintainable code and an overall poor design, but we still rely on the data that comes from this module. An anti-corruption layer provides the façade, or bridge, between new microservices architecture and legacy architecture. Therefore, it allows us to stay away from manipulating legacy code and focus on feature development.

Figure 2: Anticorruption layer

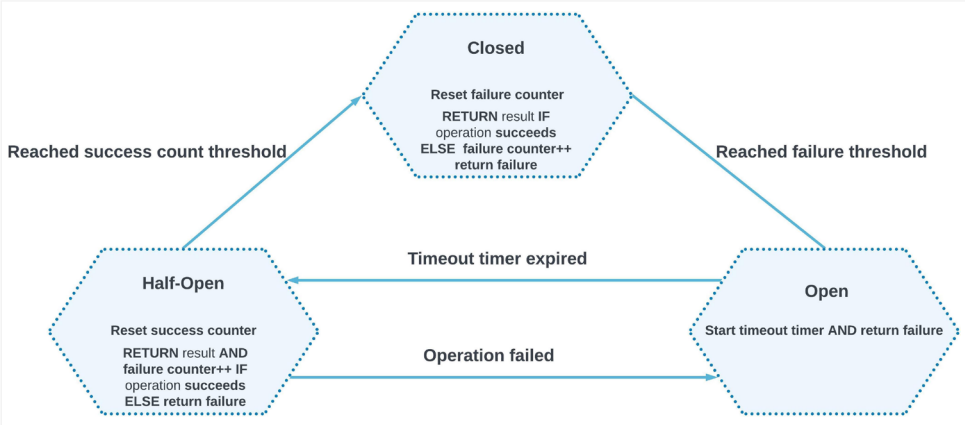


CIRCUIT BREAKER

The circuit breaker provides mechanisms to stop a cascade from failing and automatically recovering to a normal state. Imagine we have service A and service B that rely on data from service C. We introduce a bug in service C that affects services A and B. In a well-designed microservices architecture, each service should be independent of the other.

However, dependencies may occur during refactoring from monolith to microservices. In this case, you need to implement a circuit breaker to predict a cascade failure. Circuit breakers act as a state machine. They monitor numerous recent failures and decide what to do next. They can allow operation (**closed** state) or return the exception (**open** or **half-open** state).

Figure 3: Circuit breaker states



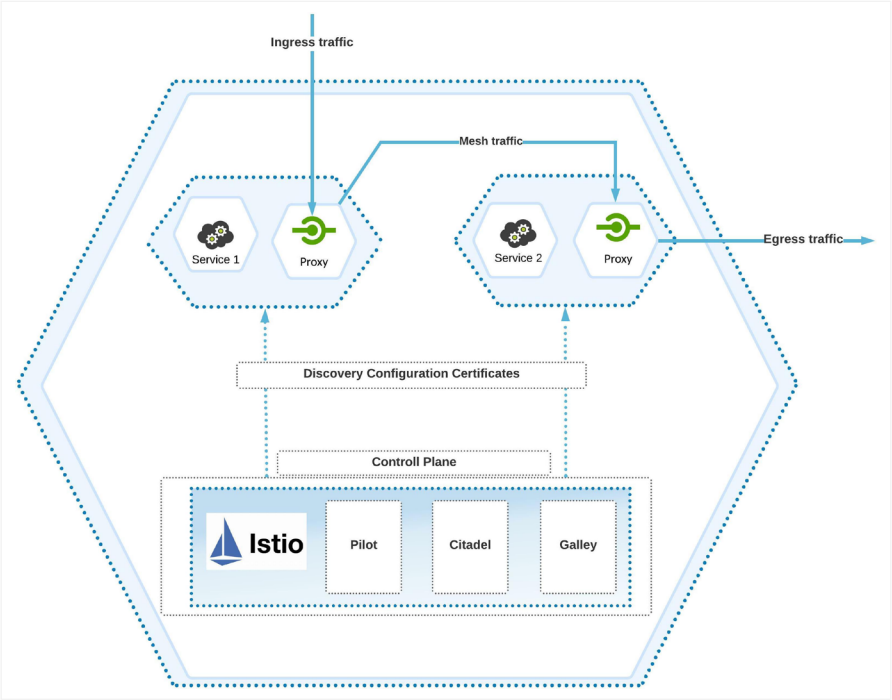
SERVICE MESH

A service mesh implements the communication layer for microservices architecture. It ensures the delivery of service requests, provides load balancing, encrypts data (with TLS), and provides the discovery of other services. A service mesh also:

- Provides circuit breaker logic
- Provides the process to manage traffic control, network resilience, security, and authentication
- Has a proxy that integrates with microservices using a sidecar pattern

It allows you to not only manage the service but also collect telemetry data and send it to the control plane. We implement a service mesh such as Istio, which is the most popular service mesh framework for managing microservices in Kubernetes.

Figure 4: Service mesh algorithm

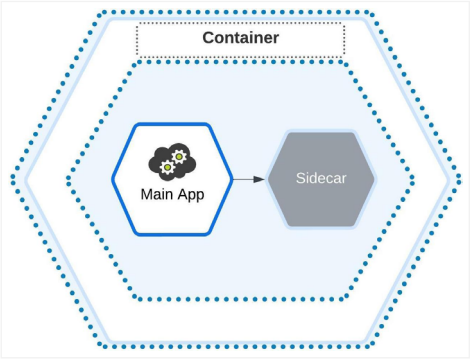


SIDECAR

Sidecar is a utility application that is deployed alongside the main service. Sidecar is helpful in:

- Collecting logs
- Managing configuration
- Controlling connection to the service

Figure 5: Sidecar



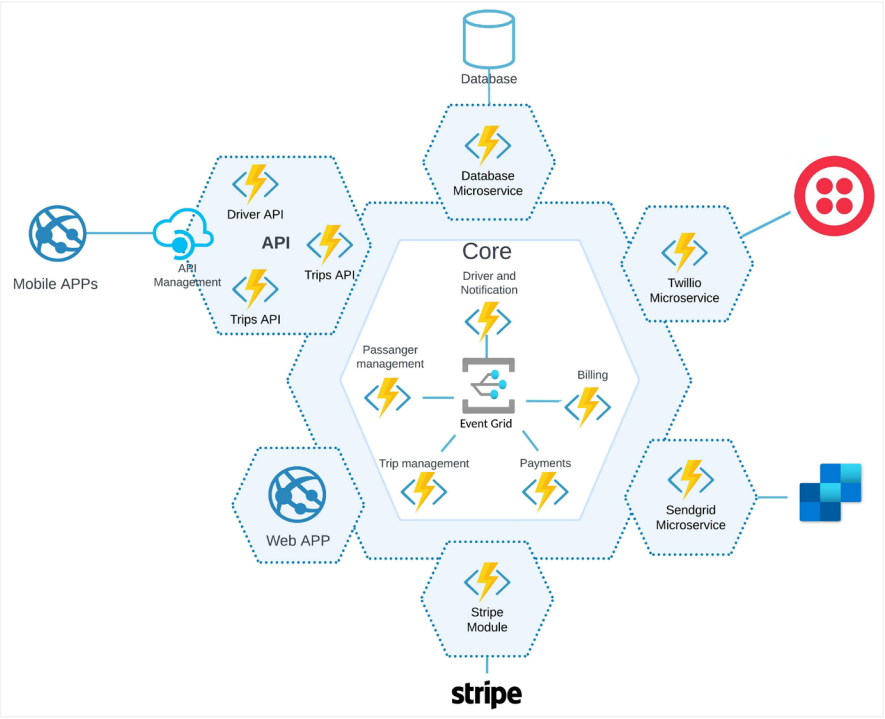
Microservices in Action

To demonstrate the power of microservices, we will migrate our monolithic travel application (see Figure 1) to the microservices architecture.

We will use the serverless approach and Azure cloud.

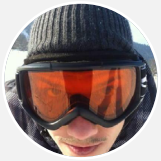
- **API Gateway** – Using API Management to expose the endpoints of the back-end services so the client application can consume them securely.
- **Entry points** – The public-facing APIs that the client application will be using, powered by Azure Functions responding to HTTP requests.
- **Async queue** – Messaging service to handle services intercommunication and pass along information and data between the different services, represented by Azure Event Grid.
- **Backend services** – The services that are directly operating with the data layer and other components of the solution, isolated from the rest, and easily replaceable if needed.

Figure 6: Travel booking microservices



Conclusion

Building highly available, scalable, and performant applications can be challenging. Microservices architecture provides us the option to build not only independent services but also create several teams to support each service and introduce the DevOps approach. Microservices and all popular cloud providers allow us to build multi-cloud microservices architecture. This saves money, as some services have different pricing strategies. But be sure to choose the most appropriate service that's suited for specific microservices domains. For example, we can use AKS with an integrated service mesh or a serverless approach based on AWS Lambdas. Multi-cloud allows us to apply cloud-native DevOps to deliver services independently.



Boris Zaikin, Software & Cloud Architect at Nordcloud GmbH

@borisza on DZone | @boris-zaikin on LinkedIn | boriszaikin.com

I'm a Certified Software and Cloud Architect who has solid experience designing and developing complex solutions based on the Azure, Google, and AWS clouds. I have expertise in building distributed systems and frameworks based on Kubernetes and Azure Service Fabric. My areas of interest include enterprise cloud solutions, edge computing, high load applications, multitenant distributed systems, and IoT solutions.

CONTRIBUTOR INSIGHTS

How a Service Mesh Simplifies Microservice Observability

By Noorain Panjwani, Senior Consultant at Xebia

Service meshes and observability are hot topics within the microservices community. In this Trend Report, we'll explore in detail how a service mesh, along with a good observability stack, can help us overcome some of the most pressing challenges that we face when working with microservices.

Common Microservices Challenges

Adopting microservices is hard. Debugging and making sure they keep running is even harder. Microservices introduce a significant amount of overhead in the form of Day 2 operations. Let's dive into a few aspects that make working with microservices difficult.

DEBUGGING ERRORS IN A DISTRIBUTED SYSTEM IS A NIGHTMARE

The distributed nature of microservices makes debugging errors difficult. In a monolith, just going through the logs and stack trace is enough to identify the root cause of a bug. Things aren't so straightforward when it comes to microservices. In the case of errors, digging through a microservice's logs may not directly point to the precise problem. Instead, it could simply mention an error present in the request and/or response received from a dependent microservice.

In other words, we'll have to follow the entire network trace to figure out which microservice is the root cause of the problem. This is an extremely time-consuming process.

IDENTIFYING BOTTLENECKS IN THE SYSTEM ISN'T EASY

In monoliths, identifying performance bottlenecks is as easy as profiling our application. Profiling is often enough to figure out which methods in your codebase are consuming the most time. This helps you focus your optimization efforts to a narrow section of code. Unfortunately, figuring out which microservice is causing the entire system to slow down is challenging.

When tested individually, each microservice may seem to be performant. But in a real-world scenario, the load on each service may differ drastically. There could be certain core microservices that a bunch of other microservices depend on. Such scenarios can be extremely difficult to replicate in an isolated testing environment.

MAINTAINING A MICROSERVICE DEPENDENCY TREE IS DIFFICULT

The entire point of microservices is the agility at which we can release new software. Having said that, it is important to note that there can be several downstream effects when releasing a microservice. Certain events that often break functionality are:

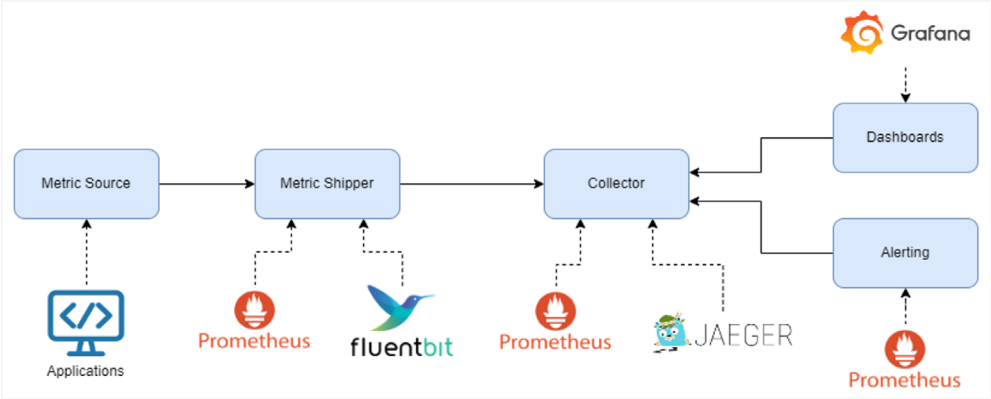
- Releasing a dependent microservice before its upstream dependency
- Removing a deprecated API that a legacy system still depends on
- Releasing a microservice that breaks API compatibility

These events become difficult to avoid when there is no clear dependency tree between microservices. A dependency tree makes it easier to inform the appropriate teams and plan releases better.

Observability: The Solution to the Microservices Problem

All the problems I've mentioned above can be resolved through observability. According to Jay Livens, observability is the practice to capture the system's current state based on the metrics and logs it generates. It's a system that helps us with monitoring the health of our application, generating alerts on failure conditions, and capturing enough information to debug issues whenever they happen.

Figure 1: Components of an observability stack with open-source examples



Any observability stack will have the following components:

- **Source of metrics/logs** – an agent or library we use to generate data
- **Metric/log shipper** – an agent to transport the data to a storage engine; often embedded within the metric source
- **Collector/store** – a stateful service responsible for storing the data generated
- **Dashboards** – fancy charts that make the data easy for us to interpret and digest
- **Alert manager** – the service responsible for triggering notifications

Luckily for us, there are some powerful open-source tools to help simplify the process of setting up an observability stack.

Introducing Service Meshes

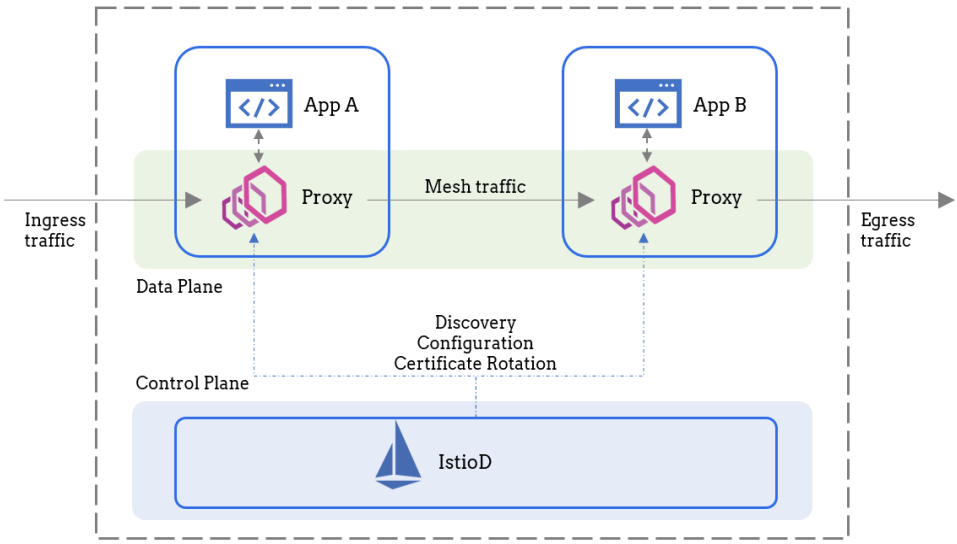
A major aspect of observability is capturing network telemetry, and having good network insights can help us solve a lot of the problems we spoke about initially. Normally, the task of generating this telemetry data is up to the developers to implement. This is an extremely tedious and error-prone process that doesn't really end at telemetry. Developers are also tasked with implementing security features and making communication resilient to failures.

Ideally, we want our developers to write application code and nothing else. The complications of microservices networking need to be pushed down to the underlying platform. A better way to achieve this decoupling would be to use a service mesh like Istio, Linkerd, or Consul Connect.

A service mesh is an architectural pattern to control and monitor microservices networking and communication.

Let's take the example of Istio to understand how a service mesh works.

Figure 2: Typical service mesh architecture



Source: Image adapted from Istio documentation

A service mesh has two major components: the data plane and the control plane. The **data plane** is responsible for managing all the network traffic our microservices will generate. To achieve this, a service mesh injects a sidecar proxy alongside each one of our microservices. This sidecar, which is usually Envoy, is responsible for transparently intercepting all traffic flowing through the service. The **control plane**, on the other hand, is merely responsible for configuring the proxies. No application traffic ever reaches the control plane.

The service mesh architecture, as illustrated in Figure 2, helps you abstract away all the complexities we spoke about earlier. The best part is that we can start using a service mesh without having to write a single line of code. A service mesh helps us with multiple aspects of managing a microservics-based architecture. Some of the notable advantages include:

- Obtaining a complete overview of how traffic flows
- Controlling the flow of network traffic
- Securing microservices communication

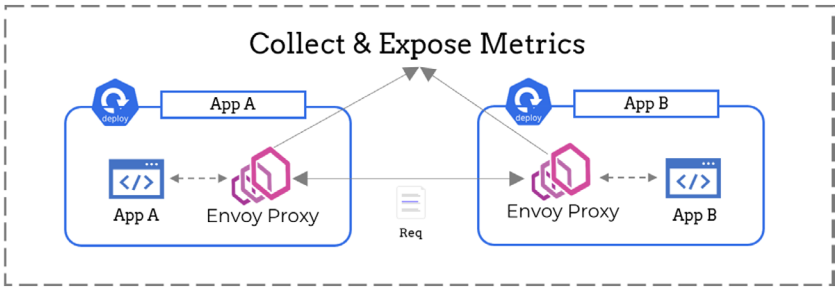
OBTAINING A COMPLETE OVERVIEW OF HOW TRAFFIC FLOWS

In Figure 3, App A is making a request to App B. Since the Envoy proxies sitting beside each app are intercepting the request, they have complete visibility over the traffic flowing through these two microservices. The proxies can inspect this traffic to gather information like the number of requests being made and the response status code of each request.

In other words, a service mesh can help us answer questions like:

- Which service is talking to which?
- What's the request throughput observed by each microservice?
- What's the error rate of each API?

Figure 3: A service mesh can help collect metrics

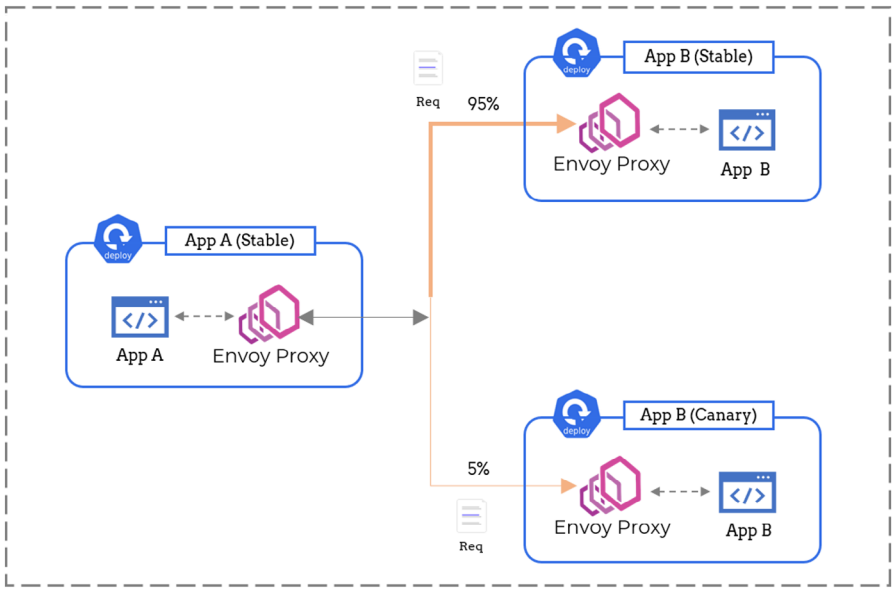


CONTROLLING NETWORK

A service mesh isn't just a silent spectator. It can actively take part in shaping all network traffic. The Envoy proxies used as sidecars are HTTP aware, and since all the requests are flowing through these proxies, they can be configured to achieve several useful features like:

- **Automatic retires** – ability to replay a request whenever a network error is encountered
- **Circuit breaking** – blacklisting an unhealthy replica of an upstream microservice that has stopped responding
- **Request rewriting** – ability to set headers or modify the request URL when certain conditions are met

Figure 4: A service mesh can shape network traffic



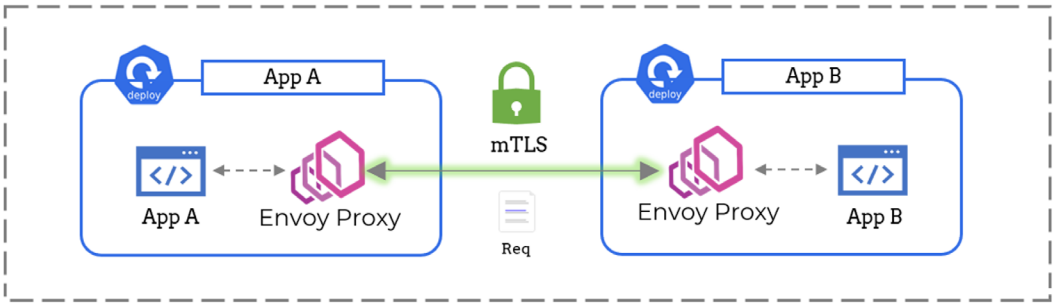
And it doesn't end here. The proxies can also split traffic based on a certain weight. For example, we can configure the proxies to send 95% of the incoming traffic to the stable version of the service while the rest can be redirected to the canary version. This can help us simplify the release management process by powering advanced practices like canary deployments.

SECURING MICROSERVICES COMMUNICATION

Another great advantage of using a service mesh is security. Our sidecar proxies can be configured to use mutual TLS. This ensures that all network traffic is automatically encrypted in transit. The task of managing and rotating the certificates required for mTLS is fully automated by the service mesh control plane.

A service mesh can assist in access control as well by selectively allowing which service is allowed to talk to which. All this can help us completely eradicate a whole breed of security vulnerabilities like man-in-the-middle attacks.

Figure 5: A service mesh can secure network traffic



How Does a Service Mesh Help With Observability?

We just saw how a service mesh can capture telemetry data. Let's dig a bit deeper to understand what kind of use cases this data can power.

DISTRIBUTED TRACING

We've discussed how difficult it is to debug microservices. One way to solve this debuggability problem is by means of distributed tracing — the process of capturing the lifecycle of a request. One graph alone can make it so much easier to figure out the root cause of the problem.

Most service meshes automatically collect and ship network traces to tools like Jaeger. All you need to do is forward a few HTTP headers in your application code. That's it!

TRAFFIC FLOW METRICS

A service mesh can help us collect three out of the four golden signals one must monitor to determine a service's health:

- **Request throughput** – the number of requests being serviced by each microservice
- **Response error rate** – the percentage of failed requests
- **Response latencies** – the time it takes for a microservice to respond; this is a histogram from which you can extract *n* percentiles of latency

There are many more metrics that a service mesh collects, but these are by far the most important ones. These metrics can be used to power several interesting use cases. Some of them include:

- Enabling scaling based on advanced parameters like request throughput
- Enabling advanced traffic control features like rate limiting and circuit breaking
- Performing automated canary deployments and A/B testing

NETWORK TOPOLOGY

A service mesh can help us automatically construct a network topology, which can be built by combining tracing data with traffic flow metrics. If you ask me, this is a real lifesaver. A network topology can help us visualize the entire microservice dependency tree in a single glance. Moreover, it can also highlight the network health of our cluster. This can be great for identifying bottlenecks in our application.

Conclusion

Observability is a wide umbrella covering several moving pieces. Luckily for us, tools like service meshes cover a lot of ground without requiring us to write a single line of code. In a nutshell, a service mesh helps us by:

- Generating distributed tracing data to simplify debugging
- Acting as a source for critical metrics like the golden signals for microservices monitoring
- Generating a network topology

As next steps, you can check out the following guides to dive deeper into the world of service meshes and observability.

1. Get started with Istio (video) – https://www.youtube.com/watch?v=Suu1Z9fbSKw
2. Get started with LinkerD (video) – https://www.youtube.com/watch?v=mDC3KA_6vfg
3. Get started with Kuma (video) – https://www.youtube.com/watch?v=_3y_4A9qdKU
4. "How to Set Up Monitoring Using Prometheus and Grafana" (article) – https://dzone.com/articles/how-to-setup-monitoring-using-prometheus-and-grafa

Noorain Panjwani, Senior Consultant at Xebia
@noorainp on DZone | @YourTechBud on Twitter

Noorain is a die-hard techie and a profound open source enthusiast. He is an AWS Certified Solutions Architect with 5+ years of experience in designing and developing cloud-native systems and architectures. He has demonstrated mastery in architecting solutions on top of AWS, GCP, and Kubernetes.

CONTRIBUTOR INSIGHTS

Popular Design Patterns for Microservices Architectures

By Rajesh Bhojwani, Development Architect

Applications have been built with monolithic architectures for decades; however, many are now moving to a microservices architecture. Microservices architectures gives us faster development speed, scalability, reliability, the flexibility to develop each component with the best tech stack suitable, and much more. Microservices architectures rely on independently deployable microservices. Each microservice has its own business logic and database consisting of a specific domain context. The testing, enhancing, and scaling of each service is independent of other microservices.

However, a microservices architecture is also prone to its own challenges and complexity. To solve the most common challenges and problems, some design patterns have evolved. In this article, we will look at a few of them.

Essential Design Patterns

There are more than eight must-have design patterns for smooth development in a typical microservices architecture. In this section, we will take a look at these patterns in more detail. We have divided them into two sections based on the type of applications being created — greenfield or brownfield.

DESIGN PATTERNS FOR GREENFIELD APPLICATIONS

When we build an application from scratch, we get the freedom to apply all the latest and modern design patterns required for a microservices architecture. Let's take a deep dive into a few of them.

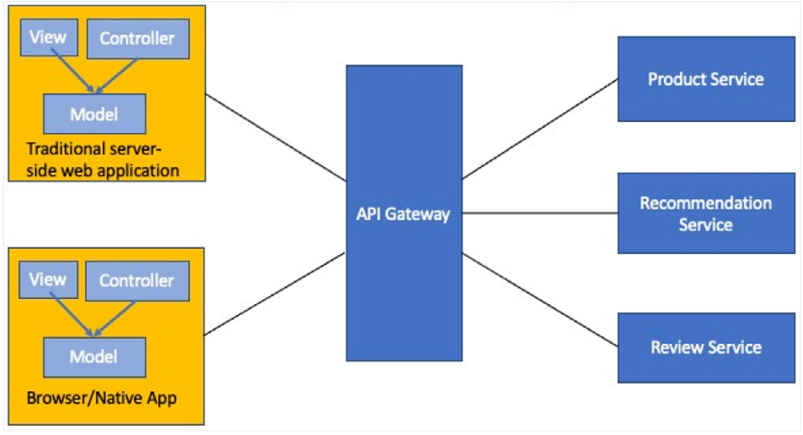
API GATEWAY PATTERN

Breaking down the whole business logic into multiple microservices brings various issues, leading to questions such as:

- How do you handle cross-cutting concerns such as authorization, rate limiting, load balancing, retry policies, service discovery, and so on?
- How do you avoid too many round trips and tight coupling due to direct client-to-microservice communication?
- Who will do the data filtering and mapping in case a subset of data is required by a client?
- Who will do the data aggregation in case a client requires calling multiple microservices to get data?

To address these concerns, an API gateway sits between the client applications and microservices. It brings features like reverse proxy, requests aggregation, gateway offloading, service discovery, and many more. It can expose a different API for each client.

Figure 1: API gateway example



Source: Diagram adapted from Microservices.io documentation

CLIENT-SIDE UI COMPOSITION PATTERN

In this pattern, microservices are developed by business-capabilities-oriented teams. Some UI screens may need data from multiple microservices. For example, a shopping site will include features such as a catalog, shopping cart, buying options, customer reviews, and so on. Each data item belongs to a specific microservice. Now each data item to be displayed is the responsibility of a different team. How can we solve this?

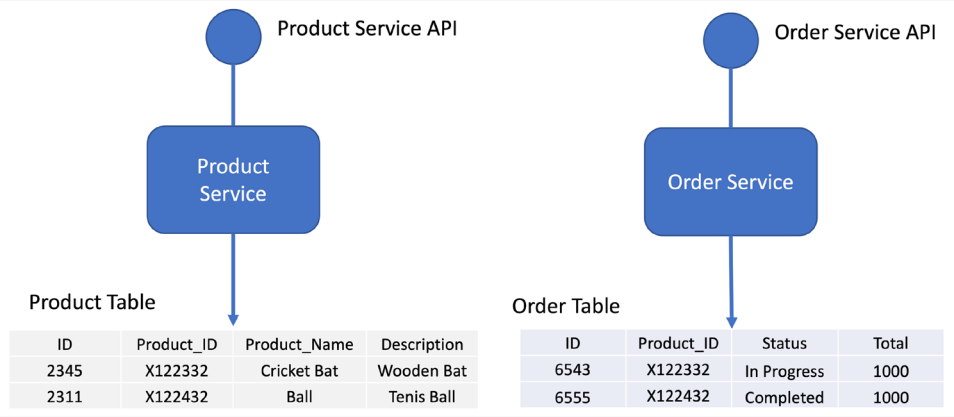
A UI team should create a page skeleton that builds screens by composing multiple UI components. Each team develops a client-side UI component that is service-specific. This skeleton is also known as a single-page application (SPA). Frameworks like AngularJS directives and ReactJS components supports this. This also allows users to refresh a specific region of the screen when any data changes, providing a better user experience.

DATABASE PER SERVICE PATTERN

Microservices need to be independent and loosely coupled. So what should the database architecture be for a microservice application?

- A typical business transaction may involve queries, joins, or data persistence actions from multiple services owned by different teams.
- In polyglot microservices architectures, where each microservice may have different data storage requirements, consider unstructured data (NoSQL database), structured data (relational database), and/or graph data (Neo4j).
- Databases need replications and sharding for scaling purposes.

Figure 2: Example of a database per service



Source: Diagram adapted from Microservices.io documentation

A microservice transaction must be limited to its own database. Any other service requiring that data must use a service API. If you are using a relational database, then a schema per service is the best option to make the data private to the microservice. To create a barrier, assign a different database user ID to each service. This ensures developers are not tempted to bypass a microservice's API and access the database directly.

This enables each microservice to use the type of database best suited for its requirements. For example, use Neo4j for social graph data and Elasticsearch for text search.

SAGA PATTERN

When we use one database per service, it creates a problem with implementing transactions that span multiple microservices. How do we bring data consistency in this case? Local ACID transactions don't help here. The solution is the saga pattern. A saga is a chain of local transactions where each transaction updates the database and publishes an event to trigger the next local transaction. The saga pattern mandates compensating transactions in case any local transaction fails.

There are two ways to implement saga:

- **Orchestration** – An orchestrator (object) coordinates with all the services to do local transactions, get updates, and execute the next event. In case of failure, it holds the responsibility to trigger the compensation events.
- **Choreography** – Each microservice is responsible for listening to and publishing events, and it enables the compensation events in case of failure.

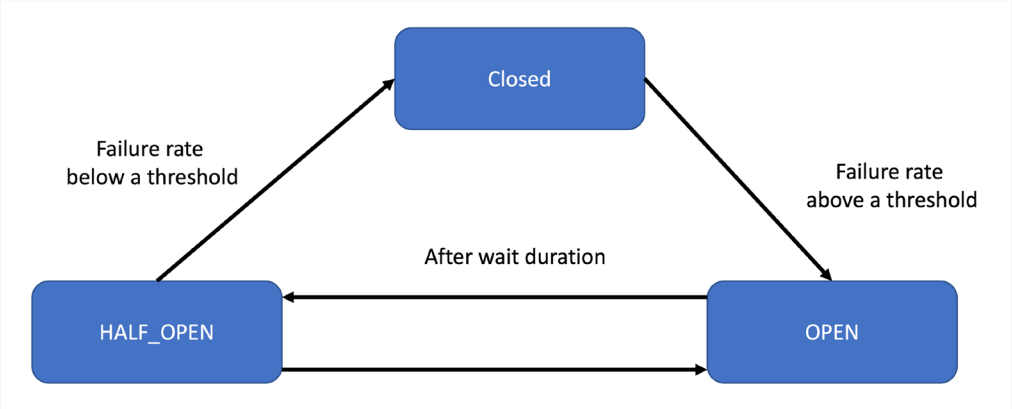
Orchestration is much simpler to implement than choreography. In orchestration, only one component needs to coordinate all the events, whereas in choreography, each microservice must listen and react to events.

CIRCUIT BREAKER PATTERN

In microservices architectures, a transaction involves calling multiple services. If a downstream microservice is down, it will keep calling to the service and exhaust network resources for all other services as well. It also impacts the user experience. How do we handle the cascading failures?

Electric circuit breaker functionality inspired the circuit breaker pattern — the solution to this issue. A proxy sits between a client and a microservice, which tracks the number of consecutive failures. If it crosses a threshold, it trips the connection and fails immediately. After a timeout period, the circuit breaker again allows a limited number of test requests to check if the circuit can be resumed. Otherwise, the timeout period resets.

Figure 3: Circuit breaker stages



Source: Diagram adapted from "Circuit Breaker Implementation in Resilience4j," Bohdan Storozhuk

Java's resilence4j framework provides this proxy service.

DECOMPOSITION BY BUSINESS CAPABILITY OR SUBDOMAIN PATTERN

In microservices architectures, a large complex application must be decomposed, cohesive, as well as loosely coupled. It should also be autonomous and small enough to be develop by a "pizza-sized" team (six to eight members). How do we decompose it?

There are two ways to decompose a greenfield application — by business capability or subdomain:

- A **business capability** is something that generates value. For example, in an airline company, business capabilities can be bookings, sales, payment, seat allocation, and so on.
- The **subdomain** concept comes from domain-driven design (DDD). A domain consists of multiple subdomains, such as product catalog, order management, delivery management, and so on.

DESIGN PATTERNS FOR BROWNFIELD APPLICATIONS

Because we have been building applications for decades, around 80 percent of companies run on existing applications that are known as brownfield applications. Migrating a brownfield application to microservices is the most challenging task. Let's have a look at certain design patterns that can help make migration easier.

STRANGLER FIG PATTERN

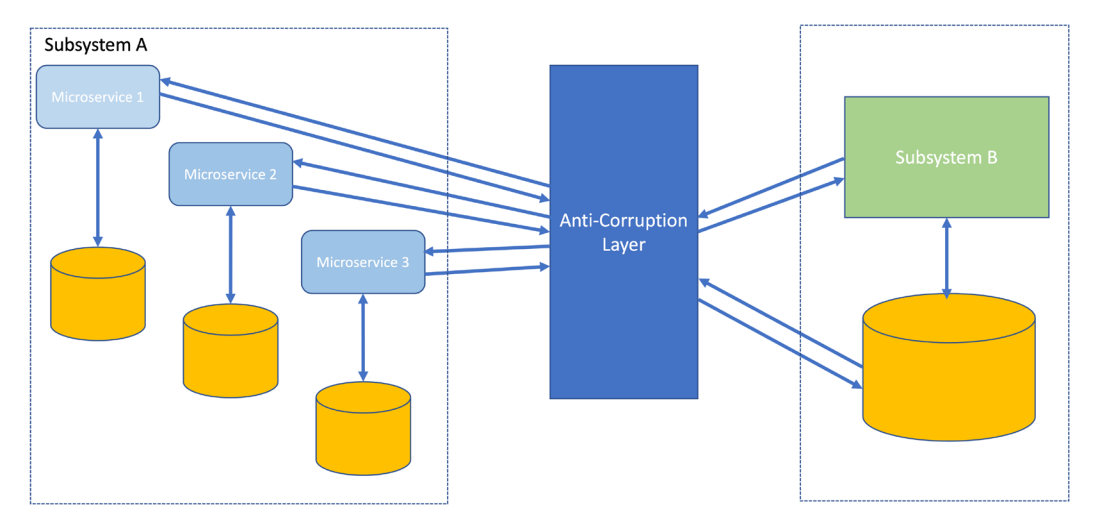
How do we migrate a monolithic application to a microservices architecture? The strangler pattern is based on a vine analogy that strangles the tree it wraps around. In this pattern, a small part of the monolithic application is converted to a microservice, and for the customer, nothing changes as the external API remains the same. Slowly, all the parts are refactored to microservices, and the new architecture "strangles," or replaces, the original monolithic architecture.

ANTI-CORRUPTION LAYER PATTERN

When a modern application needs to integrate with a legacy application, it is challenging to interoperate with outdated infrastructure protocols, APIs, and data models. Adhering to old patterns and semantics may corrupt the new system. How can we avoid that?

This requires a layer that translates communication between the two systems. An anti-corruption layer conforms to the data model of the legacy or modern system, depending on which system it is communicating with to get the data. It ensures the old system doesn't need to change, and the modern system doesn't compromise on its design and technology.

Figure 4: Anti-corruption layer example



Source: Diagram adapted from Microservices.io documentation

Conclusion

Microservices architectures provide a lot of flexibility for developers but also bring many challenges as the number of components to manage increases. In this article, we have talked about the most important design patterns that are essential to building and developing a microservices application.

Additional resources to help you get started:

- "Design Patterns for Microservices" – https://dzone.com/articles/design-patterns-for-microservices
- "Distributed Sagas for Microservices" – https://dzone.com/articles/distributed-sagas-for-microservices
- "API Gateway to the Rescue" – https://dzone.com/articles/gateway-pattern
- Microsoft Documentation, Strangler Fig Pattern – https://docs.microsoft.com/en-us/azure/architecture/patterns/strangler-fig

Rajesh Bhojwani, Development Architect
@rajesh.bhojwani on DZone | @rajesh.bhojwani on LinkedIn

Rajesh Bhojwani is a development architect with a rich 18+ years of experience. He is currently responsible for design, development, and implementation of cloud-native technologies around Spring Boot, Cloud Foundry, and AWS. He has experience not only as a developer and architect but has taken the role of consulting on-site coordinator. Rajesh understands customer issues very well. His hobby is educating and mentoring developers on cloud-native technology skills.

CONTRIBUTOR INSIGHTS

Simplify Your Microservices Architecture With a Data API

Replace Data Services With a Flexible Gateway for Reduced Development and Maintenance

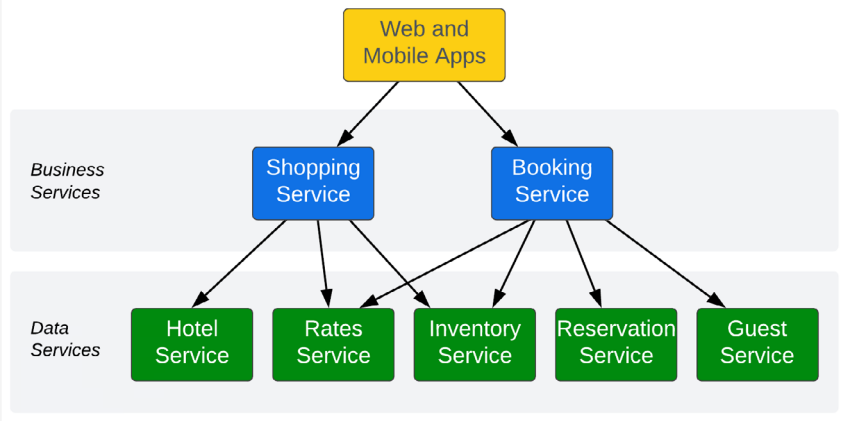
By Jeff Carpenter, Software Engineer at DataStax

Have you encountered challenges in how to manage data in a microservices architecture? In this article, we examine traditional approaches and introduce the data API gateway (also sometimes known as a "data gateway"), a new type of data infrastructure. We explore the features of a data API gateway, why you should implement it, and how to apply it to your architecture.

The Traditional Data Service Pattern

First, let's consider how data is managed in microservices architectures. A common pattern is a layer of data services that perform "CRUD" (create, read, update, delete) operations. For example, consider the notional hotel reservation application shown in Figure 1.

Figure 1: Data services in a sample microservice architecture



This microservices architecture includes a layer of data services which manage specific data types including hotels, rates, inventory, reservations, and guests, and a layer of business services which implement specific processes such as shopping and booking reservations. The business services provide a primary interface to web and mobile applications and delegate the storage and retrieval of data to the data services. The data services are responsible for performing CRUD operations on an underlying database.

While there are many ways of integrating and orchestrating interactions between these services, the basic pattern of separating services responsible for data and business logic has been around since the early days of service-oriented architecture (SOA).

IDENTIFYING, DESIGNING, AND IMPLEMENTING DATA SERVICES

The typical approach to developing these microservices has included steps like the following:

- **Identifying services** to manage specific data types in the domain using a technique such as domain-driven design. For more on the interaction between domain-driven design, service identification, and data modeling, see Chapter 7 of *Cassandra, The Definitive Guide*: 3rd Edition.

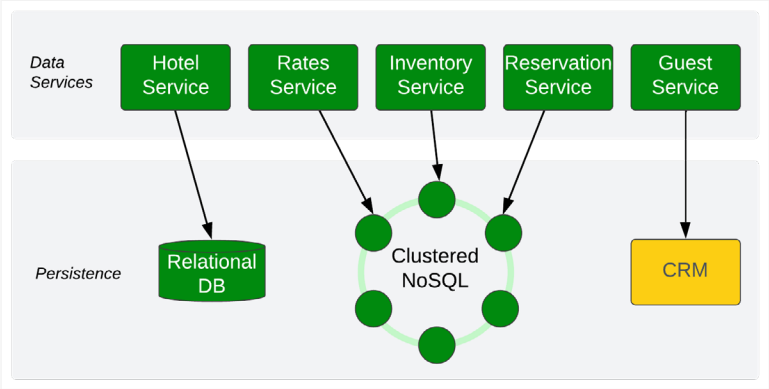
- **Designing services** including APIs and schema to manage assigned data types. Each individual service is the primary owner of a specific data type and is responsible for data storage, retrieval, and potential messaging or streaming. We'll expand on the implications of this for database selection below.
- **Implementing services** using a selected language and framework. In the Java world, frameworks like Spring Boot make it easy to build services with an embedded HTTP server that are then packaged into VMs or containers. Quarkus is a more recent framework which can build, test, and containerize services in a single CI workflow.

DATA SERVICES AND POLYGLOT PERSISTENCE

Many early SOA architectures included services that interacted with a single legacy relational database schema. An unfortunate consequence of this was a tendency to "integrate by database," where services might read and write freely to multiple tables. This lack of strong ownership often led to data integrity issues that were hard to debug.

As the move toward large-scale microservices architectures in the cloud began in the 2010s, large-scale innovators, including Netflix, advocated strongly for independent services managing their own data types. One consequence of this was that individual data services were free to select their own databases, a pattern known as *polyglot persistence*. An example of what this might look like in our hypothetical hotel application is shown in Figure 2.

Figure 2: Polyglot persistence approach for microservices architectures



In this architecture, data of a modest size that changes less frequently, like hotel descriptions, might be a natural choice for a document database or traditional relational database. Data with high volume or high read/write traffic such as rates, inventory, and reservations might use a clustered NoSQL-based solution in order to scale effectively. Other data services might be a front for a third-party API, such as guest information sourced from a customer relationship management (CRM) system.

Replacing Data Services With a Data API Gateway

In creating multiple data services, development teams often find that the implementations are highly similar, almost boilerplate code, due to their focused responsibility of executing simple CRUD operations on top of a database backend. Recognizing this duplication of effort, many organizations have begun to adopt data API gateways as an alternative to maintaining a layer of containerized data services.

A data API gateway is a piece of software infrastructure that provides access to data via APIs of various styles including REST, gRPC, and others. The gateway abstracts the details of storing and retrieving data using one or more persistent stores. This allows application developers to focus on writing business services that access data via easy-to-use APIs instead of having to learn the intricacies of a database query language.

Figure 3: Sample usage of a data API gateway

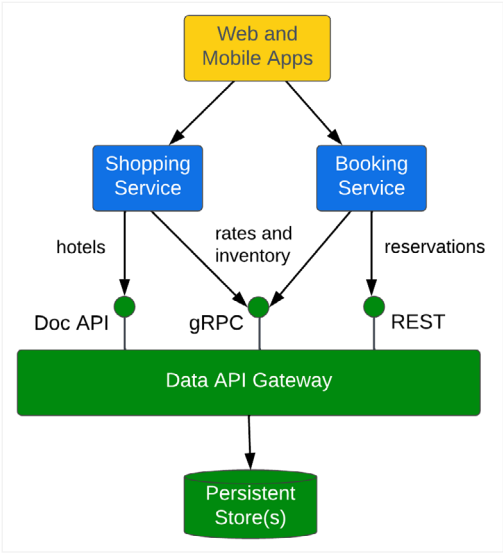


Figure 3 shows an example of how such a gateway could be applied to the hotel application example. The data API gateway takes on the responsibility of managing data persistence for hotels, rates, inventory, and other data types, eliminating the need for an entire layer of data services.

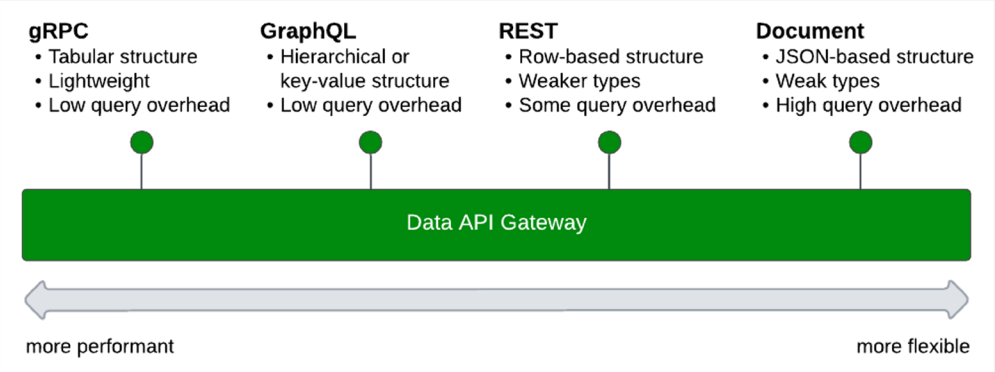
Data types can be added to the gateway by providing a schema or data model. Alternatively, document-style endpoints can provide a "schema-less" experience in which any valid JSON document can be stored, such as a hotel description, the structure of which could change frequently.

For this reason, adopting a data API gateway is quite similar to implementing the data services pattern. The design consists of identifying key data types and creating schema or JSON formats to describe them. These data types are then made available via APIs provided by the gateway.

COMPARING API STYLES

Data API gateways provide developers the freedom to access data types through the API that makes the most sense for their client services and applications. Figure 4 compares some of the most common API styles in terms of how they structure data and their performance characteristics.

Figure 4: Characteristics of APIs provided by a data API gateway



API styles like gRPC provide more structured data representations which can lead to more optimal performance. GraphQL and REST APIs provide more flexibility in how data is represented at the cost of additional latency. The maximum flexibility is provided by document-style APIs which can store and search JSON in whatever format the client chooses, at the cost of potentially lower performance for more complex queries.

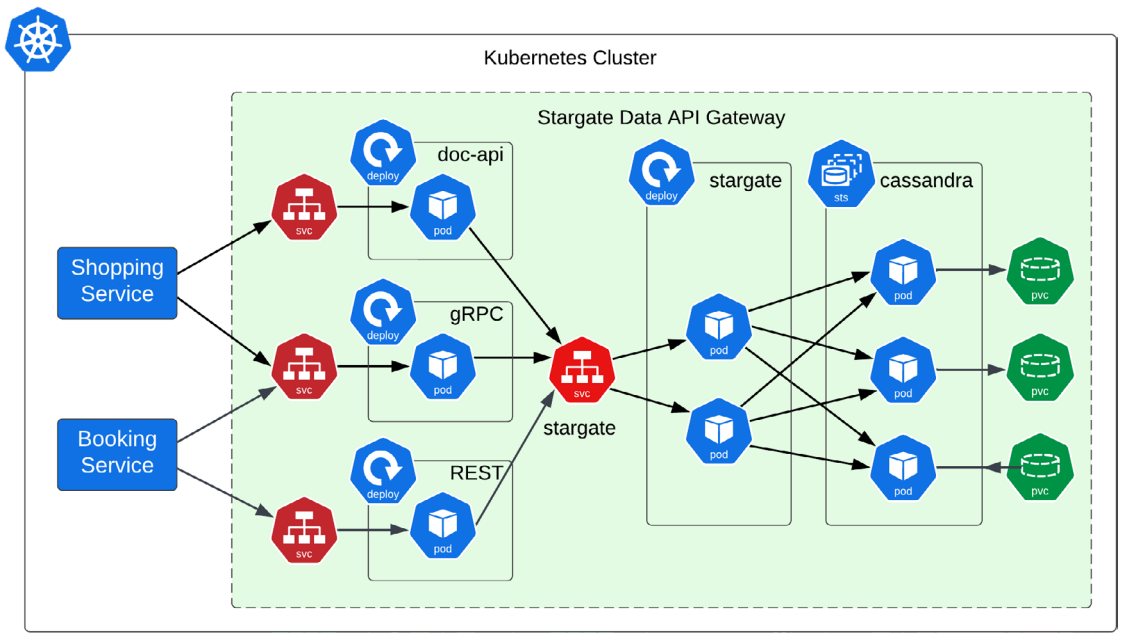
DATA API GATEWAY PROJECTS

Many organizations have built their own data API gateways, and a few of these are in various stages of being released as open-source projects. An example is Stargate, an open-source project that provides multiple API styles as a stateless proxy layer on top of Apache Cassandra. GraphQL frameworks such as the Apollo Supergraph Platform or Netlify's OneGraph might also be considered a specific tailoring of data API gateway pattern in the sense that they aggregate data from multiple persistence backends and APIs.

DEPLOYING A DATA API GATEWAY

Deploying a containerized gateway potentially includes multiple API services and backing data stores. Let's look at Stargate as an example of a data API gateway that is itself deployed as a microservices application. Figure 5 shows an example deployment of Stargate in Kubernetes.

Figure : Example deployment of Stargate in Kubernetes with a backing Cassandra cluster



A Cassandra cluster is deployed using a StatefulSet, which allows pods to be bound to PersistentVolumes for high availability of stored data. Stateless Stargate coordinator nodes and API services, such as document, gRPC, and REST, are managed in Kubernetes Deployments so that each microservice can scale independently. Kubernetes Services provide load balancing across multiple microservices instances.

Conclusion

The data API gateway is a new type of data infrastructure which can help eliminate layers of CRUD-style microservices that you have to develop and maintain. While there are multiple styles of gateway, they have a common set of features that benefit both developers and operators. Data API gateways enable developer productivity by providing a variety of API styles over a single supporting database. From an operations perspective, data API gateways and their supporting databases can be run in containers alongside other applications to simplify your overall deployment process. In summary, adopting a data API gateway is a great way to reduce development and maintenance cost for microservices architectures.

Jeff Carpenter, Software Engineer at DataStax

@jscarp on DZone | @jscarp on Twitter | @jscarp on Medium

Jeff has worked as a software engineer and architect in multiple industries and as a developer advocate helping engineers get up to speed on Apache Cassandra. He's involved in multiple open-source projects in the Cassandra and Kubernetes ecosystems and is co-writing the O'Reilly book, "Managing Cloud Native Data on Kubernetes," scheduled for publication in late 2022.

CONTRIBUTOR INSIGHTS

Approaches to Cloud-Native Application Security

Securing Microservices and Containers

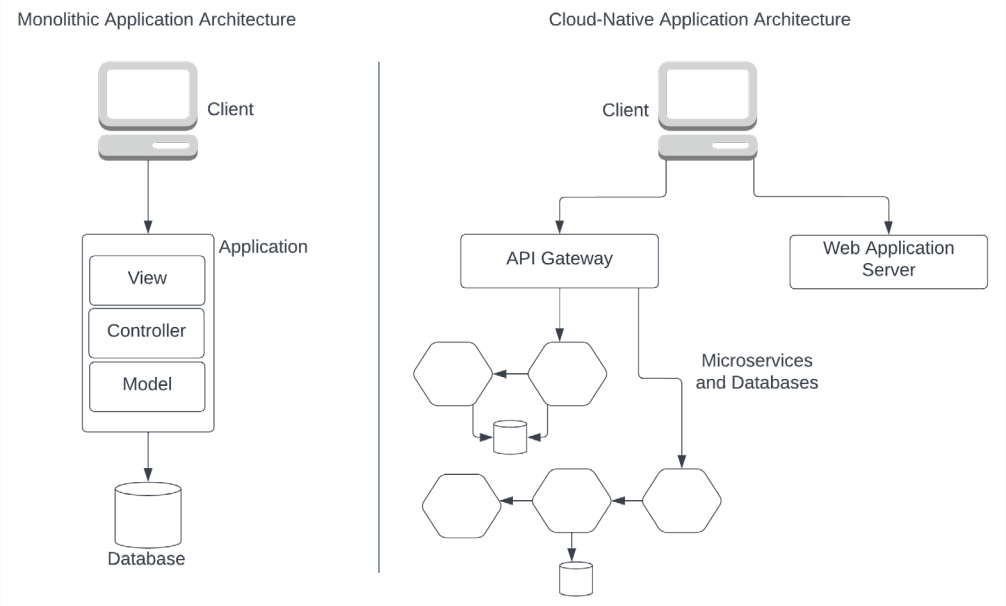
By Nuwan Dias, Deputy CTO at WSO2

Securing cloud-native applications requires proper understanding of the interfaces (boundaries) being exposed by your microservices to various consumers. Proper tools and mechanisms need to be applied on each boundary to achieve the right level of security. Properly securing the infrastructure on which your application runs is also very important. This includes securing container images, securely running container runtimes, and properly configuring and using the container orchestration system (Kubernetes).

Microservices Security Landscape

In a pre-microservices era, most applications followed the MVC architecture. Today, we call these *monolithic applications*. Compared to such applications, a cloud-native application is highly distributed, as shown in Figure 1.

Figure 1: Monolithic vs. cloud-native application



A monolithic application typically has one entry point. Beyond that, everything happens within a single process, except for database calls or similar interactions. Comparatively, the surface of exposure in a cloud-native application is much higher. As shown in Figure 1, a cloud-native application typically has multiple components (services) that communicate over the network. Each entry point into any given component needs to be appropriately secured.

Securing Application Boundaries

Let's elaborate further on application boundaries and understand the actual boundaries that we need to worry about.

IDENTIFYING COMMUNICATION BOUNDARIES IN A CLOUD-NATIVE APPLICATION

A typical cloud-native application's backend architecture would consist of several business domains. Each business domain encapsulates a set of microservices. Take a retail system, for example; order processing and inventory management can be two business domains that have their own collection of microservices.

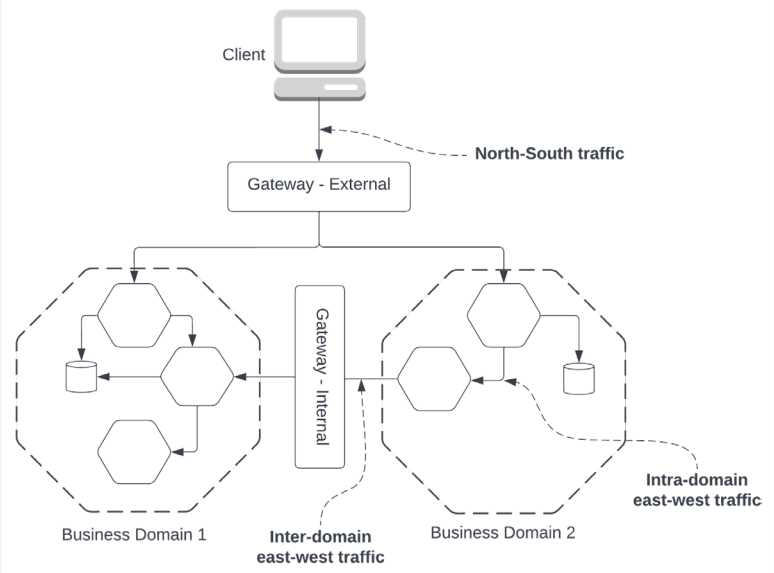
Microservices within a business domain would be able to communicate with each other securely with no boundaries. This is illustrated in Figure 2 as "Intra-domain east-west traffic." Secure communication within this domain can be implemented using mutual TLS. Mutual TLS can be implemented using a service mesh. A more lightweight approach could be to pass around an authorization token issued by one of the gateways. We will discuss this approach later in the article.

Microservices across business domains should not be able to communicate with each other freely unless they are exposed as APIs and explicitly allowed to communicate.

This is illustrated as "Inter-domain east-west traffic" in Figure 2. This concept of business domain boundaries is further explained in the paper "Cell-Based Architecture."

There is a clear boundary that separates all microservices from the client applications (web/mobile apps). This is illustrated as "North-South traffic" in Figure 2.

Figure 2: Cloud-native application architecture



APIS AND THE USE OF API GATEWAYS TO SECURE A CLOUD-NATIVE APPLICATION

Next, let's identify what we define as APIs and microservices. Any microservice (or collection) that needs to be exposed out of a given boundary needs to be defined as an API. An API typically has a specification such as OpenAPI, GraphQL, AsyncAPI, and so on. An API gateway is used to expose APIs across boundaries. An API gateway's main tasks are as follows:

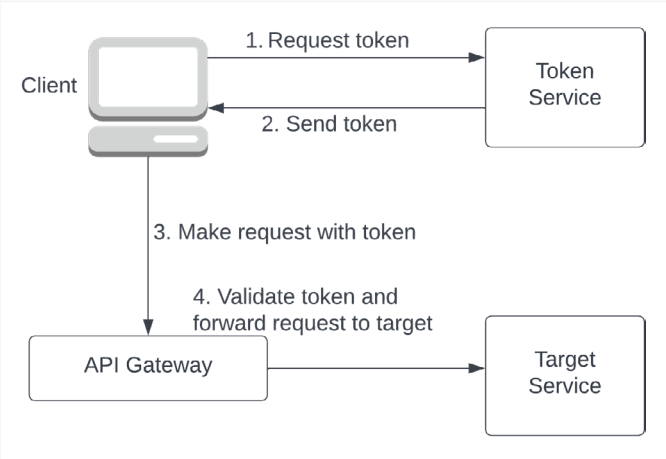
- Accept messages from calling clients.
- Ensure the client has possession of the right level of authentication/authorization.
- Forward the message to the correct target (microservice).

As shown in Figure 2, an API gateway protects the cloud-native application at the north-south channel as well as the inter-domain east-west channel.

THE ROLE OF OAUTH2.0 IN SECURING CLOUD-NATIVE APPLICATIONS

A client calling an API needs to obtain an OAuth2.0 access token from a token service before it can talk to the API. The API gateway validates the token and ensures it is issued by a trusted authority before it allows access to the target. This is shown in Figure 3. Although gaining access to APIs is common, types of tokens and how you get them differ based on the use case these tokens are used for.

Figure 3: Workflow for obtaining and using an OAuth2.0 access token



The OAuth2.0 specification has a concept called *Grant Types* that define the steps for getting access tokens. *Advanced API Security*, by Prabath Siriwardena, is a good book for understanding OAuth2.0 concepts and its use cases.

Authorization for Cloud-Native Applications

Possession of a valid access token is the primary requirement for a client to access an API. One of the key benefits of access tokens is that it not only allows you to call an API, but it can also specify the type of action you can perform with it.

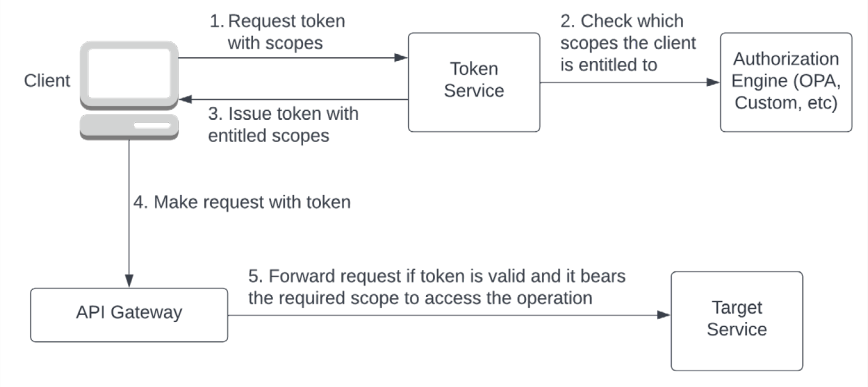
AUTHORIZATION USING OAUTH2.0 SCOPES

Imagine a product catalog API that has two operations: One for retrieving the product list (`GET /product-list`) and the other for modifying the product list (`PUT /product-list`). In a retail store application, all users should be able to retrieve the product list while only selected users should be able to modify the product list.

The standard way to model this on an API is to say the product list update operation requires a special "scope." Unless the token being used to access this API bears this scope, the request will not be permitted. The OpenAPI specification allows binding scopes to an operation.

Once the client knows that it requires a special scope to access an operation, it requests the token service to issue a token bearing the required scope. When verifying that the requesting user/client is authorized to obtain the requested scope(s), the token service binds the relevant scopes to the token. This workflow is shown in Figure 4.

Figure 4: Access an API with scopes



We can see that authentication and authorization are terminated at the API gateway. But there can be cases where the actual microservices need to know details of the user/client accessing the service for performing business logic. This requirement is done by the API gateway issuing a secondary JWT formatted token (not an access token) and forwarding it to the target service. This secondary token can be passed around the microservices within that domain and used for building mutual trust within that domain.

AUTHORIZATION WITH OPA

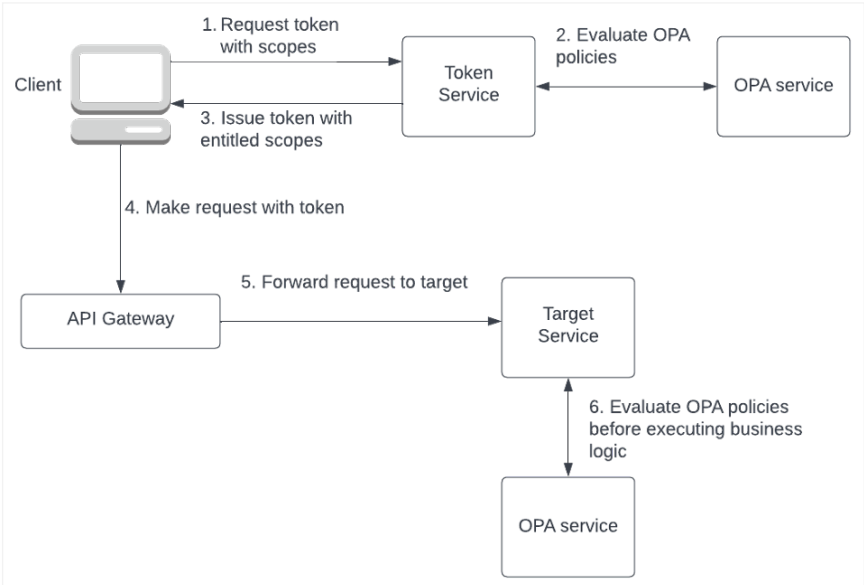
Beyond permissions, we may need to implement other authorization rules in cloud-native applications. Consider restricting access to a certain application functionality available between 8 a.m. to 6 p.m. on weekdays. While these can be implemented in the source code of the microservice, that is not a good practice.

These are organizational policies that can change. The best practice is to externalize such policies from code.

Open Policy Agent (OPA) is a lightweight general-purpose policy engine that has no dependency on the microservice. Authorization rules can be implemented in Rego and mounted to OPA.

Figure 5 illustrates the patterns in which OPA can be used for authorization rules.

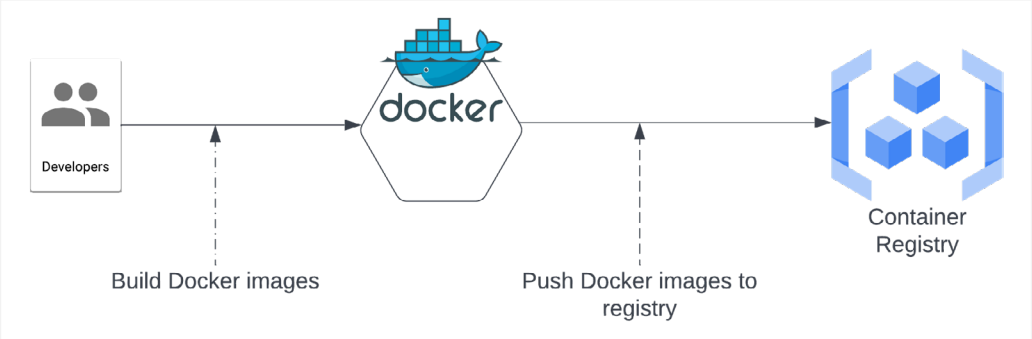
Figure 5: Using OPA for authorization



Securing Containers With Docker

Docker is the most popular tool for packaging and distributing microservices. A Docker container encapsulates a microservice and its dependencies and is stored in a container registry (private or public).

Figure 6: Docker build and push



EXTERNALIZING APPLICATION SECRETS

A microservice usually has dependencies on databases, third-party APIs, other microservices, and so on. To connect to these types of systems, a microservice may depend on sensitive information (secrets) such as certificates and passwords. In a monolithic application, these types of information are stored in a server configuration file.

Only privileged users get access to server configuration files. But in the microservices world, it is common for developers to store this information in property files along with the code of the microservice. When a developer builds such a container and pushes it to the container registry, this information becomes available for anyone who can access the container image!

To prevent this from happening, we need to externalize application secrets from code. Let's take a look at a sample Dockerfile in a Java program that does this:

```
FROM openjdk:17-jdk-alpine
ADD builds/sample-java-program.jar \
      sample-java-program.jar
ENV CONFIG_FILE=/opt/configs/service.properties
ENTRYPOINT ["java", "-jar", "sample-java-program.jar"]
```

The third line in this Dockerfile instructs Docker to create an environment variable named `CONFIG_FILE` and point it to the `/opt/configs/service.properties` location. Instead of having secrets hard coded in source code or the code read from a fixed file location, the microservice's code should be written so that it looks up the value of this environment variable to determine the configuration file location and load its contents to memory. With this, we have successfully avoided secrets within the code. If we build a Docker container with this file, it will not contain any sensitive information. Next, let's look at how we can externalize the values we need.

Before running a Docker image built from the above Dockerfile, we need to mount it to a location that has the actual configuration file with the proper values. This can be done with the following Docker `run` command:

```
:\> docker run -p 8090:8090 --mount type=bind, \ source="/hostmachine/configs/service.properties" \
target="/opt/configs/service.properties"
```

The `source` section contains a path of the filesystem on the container's host machine. The `target` section contains a path on the container filesystem. The `--mount` command instructs the Docker runtime to mount the source onto the target, meaning that the `service.properties` file can now be securely maintained on the host machine's filesystem and mounted to container runtimes before starting the containers. This way, we externalize sensitive information from the microservice itself on Docker.

DOCKER CONTENT TRUST

Modern software is made up of a lot of dependencies. A software supply chain is a collection of software dependencies all the way from application code through CI/CD and up to production. Software supply chain attacks have been quite frequent due to a malicious piece of software making its way into an application runtime through its dependency chain.

Cloud-native applications running on Docker depend on Docker images pulled down from one or more repositories. An unsuspecting developer could rely on a malicious Docker image that later compromises their application. To prevent this, Docker introduced a mechanism called Docker Content Trust (DCT), which allows image publishers to sign images using a cryptographic key and the users of Docker images to verify the images before use. Using DCT in your development and CI/CD flows will ensure that you only rely on trusted and verified Docker images in your cloud-native application.

Developers need to set an environment variable named `DOCKER_CONTENT_TRUST` and set its value to `1` to enforce DCT in all the environments on which Docker is used. For example: `:\> export DOCKER_CONTENT_TRUST=1`. Once this environment variable is set, it affects the following Docker commands: `push`, `build`, `create`, `pull`, and `run`. This means that if you try to issue a `docker run` command on an unverified image, your command will fail.

DOCKER PRIVILEGES

Any operating system has a super user known as root. All Docker containers by default run as the root user. This is not necessarily bad, thanks to the namespace partitions on the Linux Kernel. However, if you are using file mounts in your containers, an attacker gaining access to the container runtime can be very harmful. Another problem running containers with root access is that it grants an attacker access to the container runtime to install additional tools into the container. These tools can harm the application in various ways such as scanning for open ports and so on.

Docker provides a way to run containers as non-privileged users. The root user ID in Linux is 0. Docker allows us to run Docker containers by passing in a user ID and group ID. The following command would start the Docker container under user ID 900 and group ID 300. Since this is a non-root user, the actions it can perform on the container are limited.

```
Docker run --name sample-program --user 900:300 nuwan/sample-program:v1
```

Conclusion

Securing a cloud-native application properly is not trivial. API gateways and mutual trust are key to ensuring our communication channels are kept safe and we have a zero-trust architecture. OAuth2.0, scopes, and OPA (or similar) are fundamental to ensuring APIs are properly authenticated and authorized.

Going beyond this scope, we also need to be concerned about using proper security best practices on Kubernetes, properly handling secrets (passwords), securing event-driven APIs, and more. APIs, microservices, and containers are fundamental to cloud-native applications. Every developer needs to keep themselves up to date with the latest security advancements and best practices.

Nuwan Dias, Deputy CTO at WSO2

@nuwandias on DZone | @nuwandias on LinkedIn | @nuwandias on Twitter

Nuwan is an API enthusiast and is working on making cloud-native application developments simpler. He is the author of *Microservices Security in Action*. He speaks at conferences to share his knowledge on the topics of APIs, microservices, and security. He spends most of his spare time with friends and family. He is a big fan of Rugby.

CONTRIBUTOR INSIGHTS

Microservices Orchestration

Getting the Most Out of Your Services

By Christian Posta, VP, Global Field CTO at Solo.io

Does your organization use a microservices-style architecture to implement its business functionality? What approaches to microservices communication and orchestration do you use? Microservices have been a fairly dominant application architecture for the last few years and are usually coupled with the adoption of a cloud platform (e.g., containers, Kubernetes, FaaS, ephemeral cloud services). Communication patterns between these types of services vary quite a bit.

Microservices architectures stress independence and the ability to change frequently, but these services often need to share data and initiate complex interactions between themselves to accomplish their functionality. In this article, we'll take a look at patterns and strategies for microservices communication.

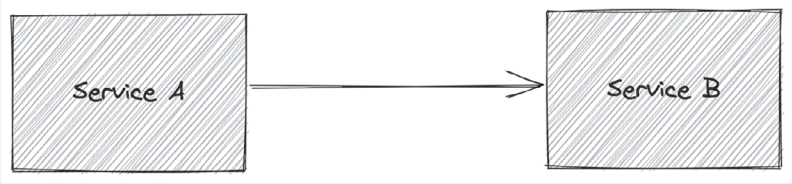
Problems in the Network

Communicating over the network introduces reliability concerns. Packets can be dropped, delayed, or duplicated, and all of this can contribute to misbehaving and unreliable service-to-service communication. In the most basic case — service A opening a connection to service B — we put a lot of trust in the application libraries and the network itself to open a connection and send a request to the target service (service B in this case).

But what happens if that connection takes too long to open? What if that connection times out and cannot be open? What if that connection succeeds but then later gets shut down after processing a request, but before a response?

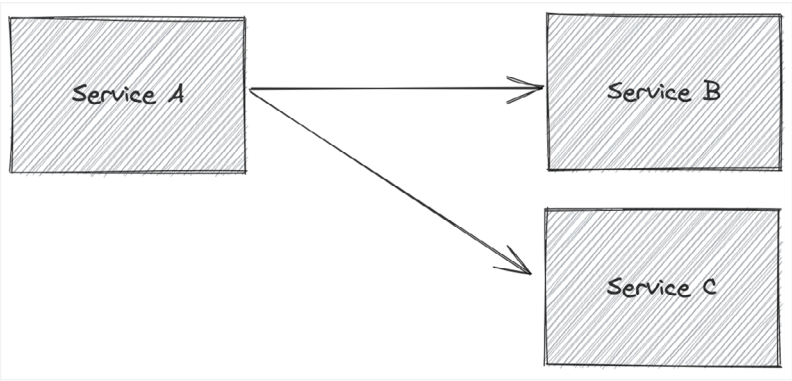
We need a way to quickly detect connection or request issues and decide what to do. Maybe if service A cannot communicate with service B, there is some reasonable fallback (e.g., return an error message, static response, respond with a cached value).

Figure 1: Simple example of service A calling service B



In a slightly more complicated case, service A might need to call service B, retrieve some values from the service B response, and use it to call service C. If the call to service B succeeds but the call to service C fails, the fallback option may be a bit more complicated.

Figure 2: More complicated example of calling multiple services



Maybe we can fallback to a predefined response, retry the request, pull from a cache based on some of the data from the service B response, or maybe call a different service?

Problems within the network that cause connection or request failures can, and do, happen intermittently and must be dealt with by the applications. These problems become more likely and more complicated with the more services calls orchestrated from a given service, as is seen in Figure 3 on the next page.

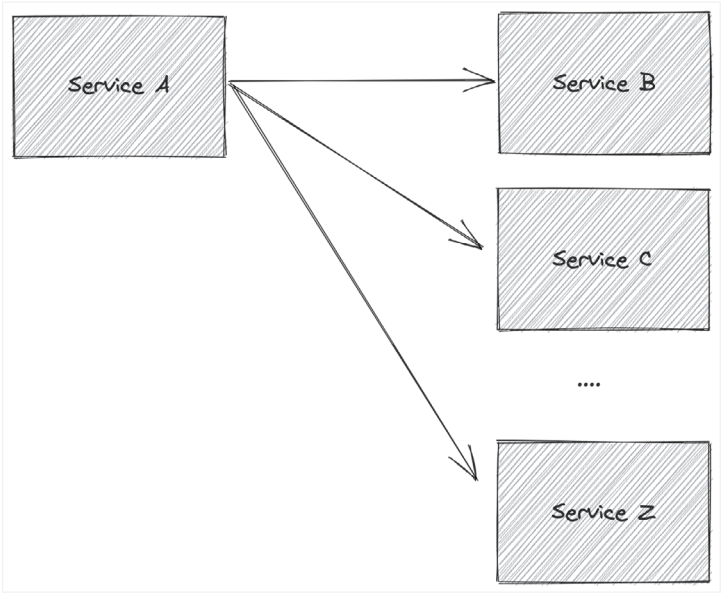
These problems become even more troubling when these calls between services aren't just "read" calls.

For example, if service A calls service B, which performs some kind of data mutation that must be coordinated with the next call to service C (e.g., service A tells service B that customer Joe's address is updated but must also tell service C to change the shipping because of the address change), then these failed calls are significant.

This can result in inconsistent data and an inconsistent state across services.

Network errors like this impact resilience, data consistency, and likely service-level objectives (SLOs) and service-level agreements (SLAs). We need a way to deal with these network issues while taking into account other issues that crop up when trying to account for failures.

Figure 3: Trying to orchestrate multiple service calls across read/write APIs



Helpful Network Resilience Patterns

Building APIs and services to be resilient to network unreliability is not always easy. Services (including the frameworks and libraries used to build a service) can fail due to the network in sometimes unpredictable ways. A few patterns that go a long way to building resilient service communication are presented here but are certainly not the only ones.

These three patterns can be used as needed or together to improve communication reliability (but each has its own drawbacks):

1. **Retry/backoff retry** – if a call fails, send the request again, possibly waiting a period of time before trying again
2. **Idempotent request handling** – the ability to handle a request multiple times with the same result (can involve de-duplication handling for write operations)
3. **Asynchronous request handling** – eliminating the temporal coupling between two services for request passing to succeed

Let's take a closer look at each of these patterns.

RETRIES WITH BACKOFF HANDLING

Network unreliability can strike at any time and if a request fails or a connection cannot be established, one of the easiest things to do is retry. Typically, we need some kind of bounded number of retries (e.g., "retry two times" vs. just retry indefinitely) and potentially a way to backoff the retries. With backoffs, we can stagger the time we spend between a call that fails and how long to retry.

One quick note about retries: We cannot just retry forever, and we cannot configure every service to retry the same number of times. Retries can contribute negatively to "retry storm" events where services are degrading and the calling services are retrying so much that it puts pressure on, and eventually takes down, a degraded service (or keeps it down as it tries to come back up). A starting point could be to use a small, fixed number of retries (say, two) higher up in a call chain and don't retry the deeper you get into a call chain.

IDEMPOTENT REQUEST HANDLING

Idempotent request handling is implemented on the service provider for services that make changes to data based on an incoming request. A simple example would be a counter service that keeps a running total count and increments the count based on requests that come in. For example, a request may come in with the value "5," and the counter service would increment the current count by 5. But what if the service processes the request (increments of 5), but somehow the response back to the client gets lost (network drops the packets, connection fails, etc.)?

The client may retry the request, but this would then increment the count by 5 again, and this may not be the desired state. What we want is the service to know that it's seen a particular request already and either disregard it or apply a "no-op." If a service is built to handle requests idempotently, a client can feel safe to retry the failed request with the service able to filter out those duplicate requests.

ASYNCHRONOUS REQUEST HANDLING

For the service interactions in the previous examples, we've assumed some kind of request/response interaction, but we can alleviate some of the pains of the network by relying on some kind of queue or log mechanism that persists a message in flight and delivers it to consumers. In this model, we remove the temporal aspect of both a sender and a receiver of a request being available at the same time.

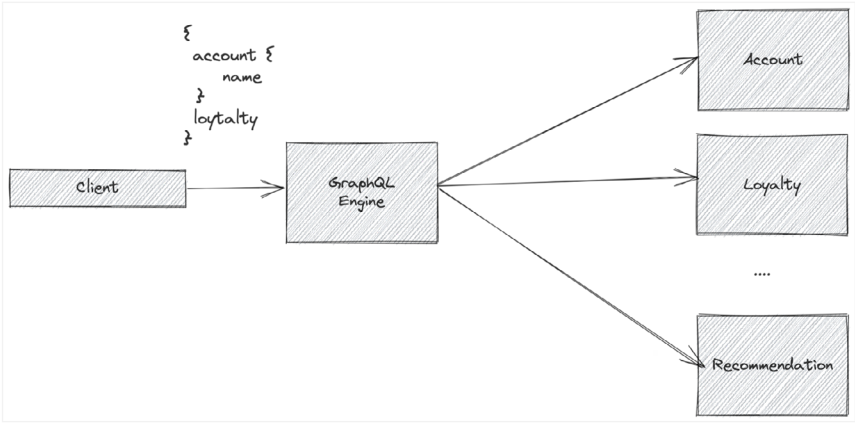
We can trust the message log or queue to persist and deliver the message at some point in the future. Retry and idempotent request handling are also applicable in the asynchronous scenario. If a message consumer can correctly apply changes that may occur in an "at-least once delivery" guarantee, then we don't need more complicated transaction coordination.

Essential Tools and Considerations for Service-to-Service Communication

To build resilience into service-to-service communication, teams may rely on additional platform infrastructure, for example, an asynchronous message log like Kafka or a microservices resilience framework like Istio service mesh. Handling tasks like retries, circuit breaking, and timeouts can be configured and enforced transparently to an application with a service mesh.

Since you can externally control and configure the behavior, these behaviors can be applied to any/all of your applications — regardless of the programming language they've been written in. Additionally, changes can be made quickly to these resilience policies without forcing code changes.

Figure 4: Orchestrating service calls across multiple services with a GraphQL engine



Another tool that helps with service orchestration in a microservices architecture is a GraphQL engine. GraphQL engines allow teams to fan out and orchestrate service calls across multiple services while taking care of authentication, authorization, caching, and other access mechanisms. GraphQL also allows teams to focus more on the data elements of a particular client or service call. GraphQL started primarily for presentation layer clients (web, mobile, etc.) but is being used more and more in service-to-service API calls as well.

GraphQL can also be combined with API Gateway technology or even service mesh technology, as described above. Combining these provides a common and consistent resilience policy layer — regardless of what protocols are being used to communicate between services (REST, gRPC, GraphQL, etc.).

Conclusion

Most teams expect a cloud infrastructure and microservices architecture to deliver big promises around service delivery and scale. We can set up CI/CD, container platforms, and a strong service architecture, but if we don't account for runtime microservices orchestration and the resilience challenges that come with it, then microservices are really just an overly complex deployment architecture with all of the drawbacks and none of the benefits. If you're going down a microservices journey (or already well down the path), make sure service communication, orchestration, security, and observability are at front of mind and consistently implemented across your services.

Christian Posta, VP, Global Field CTO at Solo.io

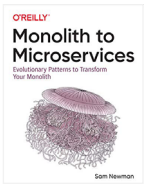
@christian.posta on DZone | @ceposta on LinkedIn | @christianposta on Twitter | blog.christianposta.com

Christian Posta is the author of *Istio in Action* and many other books on cloud-native architecture. He is well known as a speaker, blogger, and contributor to various open-source projects in the service mesh and cloud-native ecosystem. Christian has spent time at government and commercial enterprises and web-scale companies. He now helps organizations create and deploy large-scale, cloud-native, resilient, distributed architectures. He enjoys mentoring, training, and leading teams to be successful with distributed systems concepts, microservices, DevOps, and cloud-native app design.

ADDITIONAL RESOURCES

Diving Deeper Into Microservices and Containers

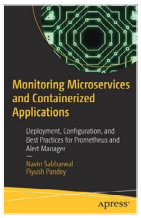
BOOKS



Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith

By Sam Newman

Thinking about transitioning to microservices? *Monolith to Microservices* is the book you need. Author Sam Newman provides guidance for migration and dives into patterns and examples to help you navigate microservices architectures.



Monitoring Microservices and Containerized Applications

By Navin Sabharwal

If you are still figuring out where to start with microservices and containerization, this book dives into methodologies and best practices to help developers and architects that work for enterprise customers.

TREND REPORTS

Kubernetes and the Enterprise

This report explores key developments in myriad technical areas related to the omnipresent container management platform, plus expert contributor articles highlighting key research findings like scaling a microservices architecture, cluster management, deployment strategies, and much more!

Containers

Explore the current state of container adoption, uncover common pain points of adopting containers in a legacy environment, and learn about modern solutions for building scalable, secure, stable, and performant containerized applications.

REFCARDS

Threat Detection for Containers

This Refcard presents a comprehensive examination of threat detection for containerized environments, spanning several focus areas such as common cloud security architectures and Kubernetes hardening guidelines.

Advanced Kubernetes

There are currently many learning resources to get started with the fundamentals of Kubernetes, but there is less information on how to manage Kubernetes infrastructure on an ongoing basis. This Refcard aims to deliver advanced-level quick, accessible information for operators using any Kubernetes product.

MULTIMEDIA

Microservices for Everyone

This podcast's title says it all. Host Tom Fanara discusses microservices and event-driven architectures so everyone can understand this concept, which also gives developers the language to explain microservices to others (including their non-technical colleagues).

Beyond the Hype

The Scott Logic team talks all things software development in this podcast, from microservices to Kubernetes to APIs and beyond. The team and their guests look beyond the buzz of new trends to break down these concepts and focus on their technical applications.

The Loosely Coupled Show

Focusing on software architecture and design, podcast hosts Derek Comartin and James Hickey, along with the occasional guest, share their thoughts, opinions, and ideas for this side of software development.

Building Leaner, Faster Microservices With Embedded NoSQL

Check out this webinar to learn more about building microservices with embedded databases, accessing data directly via database APIs instead of SQL, running Java microservices on Quarkus, and taking advantage of Java streams to augment database indexes.

A Microservices Movement: Tracing the Effects of Microservices on Java Development

What impact have microservices had on Java development? This webinar traces microservices' impact on technology and tool choices, team composition, and development strategy.

What Is Kubernetes? And How to Get it Right the First Time

While Kubernetes has become essential for the majority of container deployments, it can also cause security headaches. Many of its default security settings leave sensitive resources far too accessible — or even vulnerable to attack. This webinar covers pitfalls to avoid, how to identify risks, and ways to ensure safe, compliant workload deployments.

ADDITIONAL RESOURCES

Solutions Directory

This directory contains cloud platforms, container platforms, orchestration tools, service meshes, distributed tracing tools, and other products that aim to simplify building with microservices and containers. It provides free trial data and product category information gathered from vendor websites and project pages. Solutions are selected for inclusion based on several impartial criteria, including solution maturity, technical innovativeness, relevance, and data availability.

DZONE'S 2022 MICROSERVICES AND CONTAINERIZATION SOLUTIONS DIRECTORY

2022 PARTNERS

Company	Product	Product Type	Availability	Website
Camunda	Camunda Platform	Microservices orchestration	Free tier	camunda.com/platform
vFunction	vFunction Assessment Hub	Monolith to microservices assessment to transformation	Free tier	vfunction.com/products/assessment-hub
	vFunction Modernization Hub	AI-driven monolith to microservices transformation		vfunction.com/products/modernization-hub
Company	**Product**	**Product Type**	**Availability**	**Website**
Aiven	Aiven for Event Driven Architecture	Event-driven architecture	Trial period	aiven.io/solutions/aiven-for-event-driven-architecture
Akana	API Management Solution	API management	Trial period	akana.com/products/api-platform
Amazon Web Services	Amazon Elastic Container Registry	Container image registry	Free tier	aws.amazon.com/ecr
	Amazon Elastic Container Service	Containers-as-a-Service	By request	aws.amazon.com/ecs
	Amazon EC2	IaaS	Free tier	aws.amazon.com/ec2
	AWS Application Discovery Service	Migration project planner		aws.amazon.com/application-discovery
	Simple Queue Service	Message queuing service		aws.amazon.com/sqs
Apache Software Foundation	ActiveMQ	Message broker	Open source	activemq.apache.org
	Apache Mesos	Cluster management software		mesos.apache.org
	HTrace	Distributed tracing		incubator.apache.org/projects/htrace.html
	Kafka	Distributed event streaming platform		kafka.apache.org
	Zookeeper	Service discovery		zookeeper.apache.org
Aqua	Aqua	Cloud-native app protection platform	Free tier	aquasec.com
Aspen Mesh	Aspen Mesh	Containerized apps management system	Trial period	aspenmesh.io
Atos	Apprenda Cloud Platform	Cloud PaaS	By request	apprenda.com/platform
AxonIQ	Axon Framework	Build event-driven system	Open source	axoniq.io
	Axon Server	Message routing and event store functionality		

DZONE'S 2022 MICROSERVICES AND CONTAINERIZATION SOLUTIONS DIRECTORY

Company	Product	Product Type	Availability	Website
Axway	Axway AMPLIFY	API management services and platform	Trial period	axway.com/en/products/amplify-api-management-platform
Ballerina	Ballerina	Programming language	Open source	ballerina.io
Broadcom	Automic Automation	Service orchestration	By request	broadcom.com/products/software/automation/automic-automation
	Docker Monitoring	Container monitoring		broadcom.com/info/aiops/docker-monitoring
	Layer7	API management, API gateway		broadcom.com/products/software/api-management
Buoyant	Linkerd	Service mesh	Open source	linkerd.io
Canonical	LXD	Container management	Open source	linuxcontainers.org/lxd
Cisco	AppDynamics	Performance and monitoring tool	Trial period	appdynamics.com
Cloud Foundry	CF Container Runtime	Kubernetes cluster deployment and management	Open source	docs.cloudfoundry.org/cfcr
	Diego	App container management		github.com/cloudfoundry/diego-release
Cloud Native Computing Foundation	containerd	Container runtime system	Open source	containerd.io
	Envoy	Edge and service proxy		envoyproxy.io
	etcd	Distributed key-value store		etcd.io
	Kubernetes	Container orchestration		kubernetes.io
	NATS	Message-oriented middleware		nats.io
	OpenTracing	Distributed tracing APIs		opentracing.io
Cloudentity	Cloudentity	Identity and API security	By request	cloudentity.com
Commvault	Distributed Storage	Distributed storage solution	Trial period	commvault.com/distributed-storage
	Metallic® VM & Kubernetes Backup	Cloud-native BaaS solution		metallic.io/vm-kubernetes-backup
Couchbase	Autonomous Operator	Cloud-native database	Open source	couchbase.com/products/cloud/kubernetes
CyberArk	Conjur Enterprise	Secure secrets management for cloud-native and DevOps environments	By request	cyberark.com/products/secrets-manager-enterprise
	Conjur Open Source	Container security and authentication	Open source	conjur.org
D2iQ	D2iQ Kommander	Kubernetes cluster governance and control	By request	d2iq.com/products/kommander
	D2iQ Konvoy	Kubernetes distribution		d2iq.com/products/konvoy
	DKP Edge/IoT	Kubernetes platform for edge device management		d2iq.com/products/edge_iot
	DKP Enterprise	Multi-cluster Kubernetes solution		d2iq.com/products/enterprise
	DKP Essential	Single-cluster Kubernetes solution		d2iq.com/products/essential
Datadog	Datadog Container Monitoring	Container monitoring	Trial period	datadoghq.com/product/container-monitoring

DZONE'S 2022 MICROSERVICES AND CONTAINERIZATION SOLUTIONS DIRECTORY

Company	Product	Product Type	Availability	Website
Diamanti	Ultima Enterprise	Cloud-native storage and networking platform	By request	diamanti.com/products/ultima-enterprise
DigitalOcean	DigitalOcean Kubernetes	Managed Kubernetes service	By request	digitalocean.com/products/kubernetes
Docker	Docker	Container platform	Free tier	docker.com
	Docker Hub	Containers-as-a-Service	By request	
	Docker Swarm	Orchestrating distributed systems at scale	Open source	github.com/moby/swarmkit
Dynatrace	Dynatrace	Performance and monitoring tool	Trial period	dynatrace.com
Eclipse Foundation	Vert.x	Reactive app development platform	Open source	vertx.io
Elastic	Elasticsearch	Storage, search, and analytics	Trial period	elastic.co/elasticsearch
	Kibana	Elastic stack configuration and management	Open source	elastic.co/kibana
Epsagon	Epsagon	Microservices observability	Free tier	epsagon.com
Fluentd	Fluentd	Unified logging layer	Open source	fluentd.org
Google Cloud	Container Registry	Container image registry	Trial period	cloud.google.com/container-registry
	Kubernetes Engine	Containers-as-a-Service, managed Kubernetes platform		cloud.google.com/kubernetes-engine
	Apigee	API gateway, API management	Free tier	cloud.google.com/apigee#/products
Grails Foundation	Grails	Web app framework	Open source	grails.org
Greymatter	Greymatter	Enterprise microservices platform	By request	greymatter.io
gRPC	gRPC	RPC framework	Open source	grpc.io
HashiCorp	Consul	Identity-based networking	Open source	consul.io
IBM	Cloud Kubernetes Service	Containers-as-a-Service	Free tier	ibm.com/cloud/kubernetes-service
	IBM API Connect	API lifecycle management	Trial period	ibm.com/cloud/api-connect
	IBM App Connect	App integration and API development	Free tier	ibm.com/cloud/app-connect
	Instana	Microservices observability and automatic APM	Trial period	instana.com
Istio	Istio	Service mesh	Open source	istio.io
Jaeger	Jaeger	Distributed tracing	Open source	jaegertracing.io
JFrog	Artifactory	Artifact repository manager	Trial period	jfrog.com/artifactory
	Container Registry	Hybrid Docker and Helm registry	Free tier	jfrog.com/container-registry
JHipster	Jhipster	Web apps and microservices development platform	Open source	jhipster.tech
Kong	Kong Enterprise	API gateway, microservices management	By request	konghq.com/products/api-gateway-platform
	Kong Mesh	Service mesh		konghq.com/products/service-mesh-platform
Lacework	Polygraph® Data Platform	Data-driven cloud security platform	Trial period	lacework.com

DZONE'S 2022 MICROSERVICES AND CONTAINERIZATION SOLUTIONS DIRECTORY

Company	Product	Product Type	Availability	Website
Lightbend	Akka Platform	Reactive microservices frameworks	Open source	lightbend.com/akka-platform
	Lagom	Microservices development platform		lagomframework.com
Linux Foundation	The FLX Platform	Open-source project hosting	Open source	lfx.linuxfoundation.org
Micro Focus	Artix	ESB	Trial period	microfocus.com/en-us/products/artix/overview
Micrometer	Micrometer	JVM app monitoring	Open source	micrometer.io
MicroProfile	MicroProfile	Enterprise Java optimization for microservices architecture	Open source	microprofile.io
Microsoft Azure	API Management	API gateway, API management	Free tier	azure.microsoft.com/en-us/services/api-management
	Kubernetes Service	Containers-as-a-Service, Orchestration-as-a-Service	By request	azure.microsoft.com/en-us/services/kubernetes-service
	Service Bus	ESB	Free tier	azure.microsoft.com/en-us/services/service-bus
	Service Fabric	Microservices development platform		azure.microsoft.com/en-us/services/service-fabric
Mulesoft	Anypoint Platform	API and integration platform	Trial period	mulesoft.com
Netflix	Eureka	Service discovery	Open source	github.com/Netflix/eureka
	Ribbon	RPC library with software load balancers		github.com/Netflix/ribbon
	Zuul	Dynamic routing and service monitoring		github.com/Netflix/zuul
NeuVector	NeuVector	Cloud-native container security	By request	neuvector.com
New Relic	New Relic	Full-stack observability	Free tier	newrelic.com
NGINX	F5 NGINX Controller	Security and API management services	Trial period	nginx.com/products/nginx-controller
	F5 NGINX Ingress Controller	Kubernetes network traffic management		nginx.com/products/nginx-ingress-controller
	F5 NGINX Plus	API gateway		nginx.com/products/nginx
	F5 NGINX Service Mesh	Service mesh	Open source	nginx.com/products/nginx-service-mesh
Nirmata	DevSecOps Platform for Kubernetes	Kubernetes management and governance	By request	nirmata.com/nirmata-devsecops-platform
Nutanix	Nutanix Cloud Manager	Multi-cloud app monitoring	By request	nutanix.com/products/cloud-manager
	Nutanix Kubernetes Engine	Kubernetes management		nutanix.com/products/karbon
Okta	API Access Management	API management	Trial period	okta.com/products/api-access-management
Ondat	Ondat	Cloud-native container storage	Free tier	ondat.io
OpenLegacy	OpenLegacy Hub	Modernization platform	By request	openlegacy.com/ol-hub

DZONE'S 2022 MICROSERVICES AND CONTAINERIZATION SOLUTIONS DIRECTORY

Company	Product	Product Type	Availability	Website
OpsLevel	Service Catalog	SaaS microservices catalog	By request	opslevel.com/catalog
	Service Maturity	Microservices development and management		opslevel.com/checks
Oracle	Application Performance Monitoring	Performance and monitoring tool	Free tier	oracle.com/manageability/application-performance-monitoring
	Cloud Native	Cloud-native services and software		oracle.com/cloud/cloud-native
	Container Registry	Managed Docker registry service		oracle.com/cloud/cloud-native/container-registry
	Oracle SOA Suite	SOA governance, integration PaaS	By request	oracle.com/integration/soa
Palo Alto	Prisma Cloud	Container security	By request	paloaltonetworks.com/prisma/cloud
Particular Software	NServiceBus	ESB	Free tier	particular.net/nservicebus
Peregrine Connect	Neuron ESB	ESB	Trial period	peregrineconnect.com
Platform9	Platform9 Managed Kubernetes	Kubernetes-as-a-Service	Free tier	platform9.com/managed-kubernetes
Portainer	Portainer	Container management platform	Free tier	portainer.io
Portworx	Portworx Enterprise	Kubernetes storage platform	Free tier	portworx.com
Progress Chef	Chef Automate	Operational observability	Free tier	chef.io/products/chef-automate
	Chef Habitat	Cloud-native app automation	Open source	chef.io/products/chef-habitat
Prometheus	Prometheus	Event monitoring and alerting	Open source	prometheus.io
Rancher	Rancher	Kubernetes-as-a-Service	By request	rancher.com
Red Hat	Clair	Vulnerability static analysis for containers	Open source	redhat.com/en/topics/containers/what-is-clair
	Flannel	Container-defined networking		github.com/coreos/flannel
	Red Hat Fuse	Distributed, cloud-native integration platform	Trial period	redhat.com/en/technologies/jboss-middleware/fuse
	OpenShift	Kubernetes container platform		redhat.com/en/technologies/cloud-computing/openshift
	OpenShift Container Platform	PaaS, container platform		redhat.com/en/technologies/cloud-computing/openshift/container-platform
Redis Labs	Redis Enterprise Software	Self-managed data platform	Trial period	redis.com/redis-enterprise-software/overview
ReleaseHub	ReleaseHub	Environments-as-a-Service	By request	prod.releasehub.com
Salt Security	Salt Security	API Security	By request	salt.security
Seldon	Seldon Core	Platform for deploying ML models on Kubernetes	Open source	seldon.io
Sematext	Kubernetes Monitoring	Kubernetes monitoring and log management	Trial period	sematext.com/kubernetes
	Sematext Cloud	Container monitoring		sematext.com/container-monitoring

DZONE'S 2022 MICROSERVICES AND CONTAINERIZATION SOLUTIONS DIRECTORY

Company	Product	Product Type	Availability	Website
Sensu	Sensu	Container monitoring	Free tier	sensu.io
ServiceNow	LightStep	Microservices management	Free tier	lightstep.com
Solarwinds	Loggly	Cloud-based service for monitoring and log analysis	Trial period	loggly.com
Solo.io	Gloo Mesh	Service mesh and control plane	By request	solo.io/products/gloo-mesh
Splunk	Splunk Cloud Platform	PaaS	Trial period	splunk.com/en_us/products/splunk-cloud.html
Spring Boot	Spring Boot	Spring app development platform	Open source	spring.io/projects/spring-boot
	Spring Cloud Data Flow	Orchestration service		spring.io/projects/spring-cloud-dataflow
	Spring Cloud Sleuth	Distributed tracing		spring.io/projects/spring-cloud-sleuth
	Spring Cloud Stream	Event-driven microservices framework		spring.io/projects/spring-cloud-stream
	Spring Cloud Task	Short-lived microservices framework		spring.io/projects/spring-cloud-task
Sumo Logic	Sumo Logic	Cloud-native SaaS analytics	Trial period	sumologic.com
Sysdig	Sysdig Falco	Container security	Open source	sysdig.com/opensource/falco
	Sysdig Secure	Container, Kubernetes, and cloud security	Trial period	sysdig.com/products/secure
Temporal.io	Temporal	Workflow orchestration engine	Open source	temporal.io
TIBCO	Cloud™ API Management	API management	Trial period	tibco.com/products/api-management
Traefik Labs	Traefik Enterprise	Ingress, API management, and service mesh	Trial period	traefik.io/traefik-enterprise
Twilio	Serverless	Event-driven microservices	Free tier	twilio.com/serverless
Twitter	Finagle	RPC system	Open source	twitter.github.io/finagle
Tyk.io	Tyk	API management	Open source	tyk.io/api-lifecycle-management
VMWare	Photon	Container-optimized OS	Open source	vmware.github.io/photon
	Tanzu Application Service	Modern runtime for microservices	By request	tanzu.vmware.com/application-service
	Tanzu Build Service	Container development, management, and governance		tanzu.vmware.com/build-service
	Tanzu Kubernetes Grid	Streamline operations across multi-cloud infrastructure		tanzu.vmware.com/kubernetes-grid
	Tanzu Observability	Multi-cloud environment observability	Trial period	tanzu.vmware.com/observability
	Tanzu RabbitMQ	Message queuing service	By request	tanzu.vmware.com/rabbitmq
	Tanzu Service Mesh	Zero-trust microservices security		tanzu.vmware.com/service-mesh
	Tanzu Standard Edition	Multi-cloud Kubernetes operation and management	Open source	tanzu.vmware.com/tanzu/standard
	VMWare Application Catalog	Containers-as-a-Service		tanzu.vmware.com/application-catalog

DZONE'S 2022 MICROSERVICES AND CONTAINERIZATION SOLUTIONS DIRECTORY

Company	Product	Product Type	Availability	Website
Weaveworks	Weave GitOps Core	Continuous delivery for Kubernetes	Open source	weave.works/product/gitops-core
Windocks	Windocks	SQL Server containers	Free tier	windocks.com
WSO2	WSO2 API Manager	API management and governance	Open source	wso2.com/api-manager
Zipkin	Zipkin	Distributed tracing	Open source	zipkin.io
Zoho	Catalyst	Scalable serverless platform	Free tier	catalyst.zoho.com