



TREND REPORT

MARCH 2023

Software Integration

The Intersection of APIs, Microservices,
and Cloud-Based Systems

BROUGHT TO YOU IN PARTNERSHIP WITH





Table of Contents

HIGHLIGHTS AND INTRODUCTION

03 Welcome Letter

Ricky Esposito, Salesforce Developer/Admin
at DZone

04 About DZone Publications

DZONE RESEARCH

05 Key Research Findings

AN ANALYSIS OF RESULTS FROM DZONE'S 2023
SOFTWARE INTEGRATION SURVEY

G. Ryan Spain, Freelance Software Engineer, Former
Engineer & Editor at DZone

FROM THE COMMUNITY

16 The Path From APIs to Containers

HOW MICROSERVICES FUELED THE JOURNEY

Saurabh Dashora, Architect at ProgressiveCoder

21 Full Lifecycle API Management Is Dead

BUILD APIs FOLLOWING YOUR SOFTWARE
DEVELOPMENT LIFECYCLE WITH AN INTERNAL
DEVELOPER PLATFORM

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

24 REST vs. Messaging for Microservices

CHOOSING THE RIGHT COMMUNICATION
STYLE FOR YOUR MICROSERVICES

Swathi Prasad, Software Architect at Syneco
Trading GmbH

29 Assessment of Scalability Constraints (and Solutions)

PRACTICAL ADVICE FOR OVERCOMING
SCALABILITY CHALLENGES

Shai Almog, CEO at Codename One

33 Application Architecture Design Principles A COORDINATED, CROSS-CUTTING APPROACH

Ray Elenteny, Solution Architect at SOLTECH, Inc.

37 Demystifying Multi-Cloud Integration COMPREHENSIVE STRATEGIES AND PATTERNS FOR INTEGRATING CLOUD SYSTEMS

Boris Zaikin, Senior Software & Cloud Architect at
Nordcloud GmbH, an IBM company

ADDITIONAL RESOURCES

42 Diving Deeper Into Software Integration

43 Solutions Directory

Welcome Letter

By Ricky Esposito, Salesforce Developer/Admin at DZone

Out of all the business-y buzzwords of the rickety 21st century, "integration" seems to be one of the most pervasive.

To the non-technical, it conjures vague notions of confusing screens called up by software packages of uncertain value; meanwhile, other non-technical people are badgering you to purchase the whole lot without another doubt: "What are you waiting for??" To the technical, it promises a lengthy horizon of careful configuration and inevitable anxiety. Perhaps add some custom code to meet the unique needs that we alone require?

Integrating different systems to work together in an efficient and user-friendly way is a continuing quest. Favoring a collection of smaller interconnected pieces over a single-entity behemoth with a thousand capabilities is certainly not a new idea, but it seems to increasingly reflect global trends in different areas of existence (preventing this collection from actually functioning like the same single-entity behemoth that it was supposed to stray from is another matter entirely).

Microservices particularly seems to be one of the current all-rages — both in the cloud and, in parallel fashion, here on the surface. Many people seem to simply want smaller, friendlier pieces to work with in their everyday tasks. Some try to buy from smaller businesses for quality *and* to help establish a stable atmosphere that will allow these companies to function.

Perhaps there is hope for those who are tired of massive corporations that promise to serve as a "one-stop shop" but really don't do any particular thing very well.

And now for the analogy: **the Irish breakfast.**

For those who may not be familiar, this meal traditionally includes eggs, bacon, sausage, black or white pudding, beans, tomatoes, mushrooms, toast, and some potato element, all served with coffee, tea, and sometimes even a Guinness (the whole thing being open to additional tweaks). Independently flavorful, loosely coupled, scalable, quick to work with — each component answers to the microservices criteria surprisingly well.

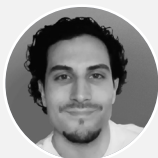
And although each piece may be individually delicious, when integrated with the rest of the picture, the finished plate offers thrills beyond the sum of the parts. Crucially, modifications can easily be made for particular tastes or customized use cases. Don't like the grilled tomato? Leave it raw! Like to operate in the cloud? Have the Guinness!

An ability to separate things and handle them independently is not only useful to keep machines and our own brains working, but also necessary for the long-term survival of both. As one of our expert authors wisely notes, the end goal is architecture that endures. How do we stay on course toward this lofty horizon? The way may be rough and steep, but perhaps the writings in our 2023 *Software Integration* Trend Report will prove useful... 🎲

Best,



Ricky Esposito



Ricky Esposito, Salesforce Developer/Admin at DZone

@ricky-espo-artist on LinkedIn

Ricky Esposito works with various software integrations to manage DZone's Salesforce platform. Off screen, he enjoys tap dancing, drawing, and playing the piano.

DZone Publications

Meet the DZone Publications team! Publishing Trend Reports and Refcards year-round, this team can often be found reviewing contributor pieces, working with authors and sponsors, and coordinating with designers. Part of their everyday includes collaborating across DZone's Content and Community team to deliver valuable, high-quality content to global DZone-ians.

DZone Mission Statement

At DZone, we foster a collaborative environment that empowers developers and tech professionals to share knowledge, build skills, and solve problems through content, code, and community.

We thoughtfully — and with intention — challenge the status quo and value diverse perspectives so that, as one, we can inspire positive change through technology.

Meet the Team



Caitlin Candelmo
Director, Content Products at DZone

[@CCandelmo](#) on DZone
[@caitlincandelmo](#) on LinkedIn

Caitlin works with her team to develop and execute a vision for DZone's content strategy as it pertains to DZone publications, content, and community. For publications, Caitlin oversees the creation and publication of all DZone Trend Reports and Refcards. She helps with topic selection and outline creation to ensure that the publications released are highly curated and appeal to our developer audience. Outside of DZone, Caitlin enjoys running, DIYing, living near the beach, and exploring new restaurants near her home.



Lindsay Smith
Senior Publications Manager at DZone

[@DZone_LindsayS](#) on DZone
[@lindsaynicolesmith](#) on LinkedIn

Lindsay oversees the Publication lifecycles end to end, delivering impactful content to DZone's global developer audience. Assessing Publications strategies across Trend Report and Refcard topics, contributor content, and sponsored materials — she works with both DZone authors and Sponsors. In her free time, Lindsay enjoys reading, biking, and walking her dog, Scout.



Lucy Marcum
Publications Coordinator at DZone

[@LucyMarcum](#) on DZone
[@lucy-marcum](#) on LinkedIn

As a Publications Coordinator, Lucy spends much of her time working with authors, from sourcing new contributors to setting them up to write for DZone. She also edits publications and creates different components of Trend Reports. Outside of work, Lucy spends her time reading, writing, running, and trying to keep her cat, Olive, out of trouble.



Melissa Habit
Senior Publications Manager at DZone

[@dzone_melissah](#) on DZone
[@melissahabit](#) on LinkedIn

Melissa leads the publication lifecycles of Trend Reports and Refcards — from overseeing workflows, research, and design to collaborating with authors on content creation and reviews. Focused on overall Publications operations and branding, she works cross-functionally to help foster an engaging learning experience for DZone readers. At home, Melissa passes the days reading, knitting, and adoring her cats, Bean and Whitney.



Lauren Forbes
Content Strategy Manager at DZone

[@laurenf](#) on DZone
[@laurenforbes26](#) on LinkedIn

Lauren identifies and implements areas of improvement when it comes to authorship, article quality, content coverage, and sponsored content. She also oversees our team of contract editors, which includes recruiting, training, managing, and fostering an efficient and collaborative work environment. When not working, Lauren enjoys playing with her cats, Stella and Louie, reading, and playing video games.



Jason Cockerham
Community Engagement Manager at DZone

[@Jason Cockerham](#) on DZone
[@jason-cockerham](#) on LinkedIn

Jason heads the DZone community, driving growth and engagement through new initiatives and building and nurturing relationships with existing members and industry subject matter experts. He also works closely with the content team to help identify new trends and hot topics in software development. When not at work, he's usually playing video games, spending time with his family, or tinkering in his garage.



Key Research Findings

An Analysis of Results from DZone's 2023 Software Integration Survey

By G. Ryan Spain, Freelance Software Engineer, Former Engineer & Editor at DZone

In February 2023, DZone surveyed software developers, architects, and other IT professionals in order to understand the state of software integration.

Major research targets were:

1. Integration practices and systems
2. Integration architectures and paradigms
3. Integration development experience

Methods: We created a survey and distributed it 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 various DZone social media channels. The survey was opened on February 11th and ended on March 1st; it recorded 117 complete and partial responses.

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

Research Target One: Integration Practices and Systems

Motivations:

1. To start, we wanted to know what respondents and their organizations were integrating. Software integration is used to meet a wide variety of different business requirements and to solve a plethora of application problems and pain points; it can cover adding CRM or payment functionality to web apps, supplying specific data to clients' software, or even just getting analytics from one internal enterprise system to another. So we tried to get an idea of what types of systems were being integrated, and how much respondents worked on inbound, outbound, and internal API integrations.
2. Thoughtful design can curb any potential integration difficulties — a topic we will look at more closely later in these findings — before they arise. There are many factors to keep in mind when trying to avoid complications, from the method by which data is transferred and the location it is stored to the complexity of the integrated systems themselves. We asked respondents which factors were most important to them when approaching integration problems.
3. One crucial aspect of API integration development is ensuring that the API is accessible to any authorized clients, ideally with minimal complexity exposed to the front end. With that in mind, we asked respondents their preferred method of simplifying API development and maintenance.

TYPES OF INTEGRATED SYSTEMS

From integrating organizational sales data with business intelligence applications to integrating personal LinkedIn notifications with a smart home assistant, it seems that everything can be integrated these days. We wanted to see how software professionals were using integration and APIs to build their applications, so we asked:

What types of systems does your organization integrate? Select all that apply.

and

Select the following types of APIs that you currently work on:

Results (n=110 and n=104, respectively):

SEE FIGURE 1 AND 2 ON NEXT PAGE

Figure 1

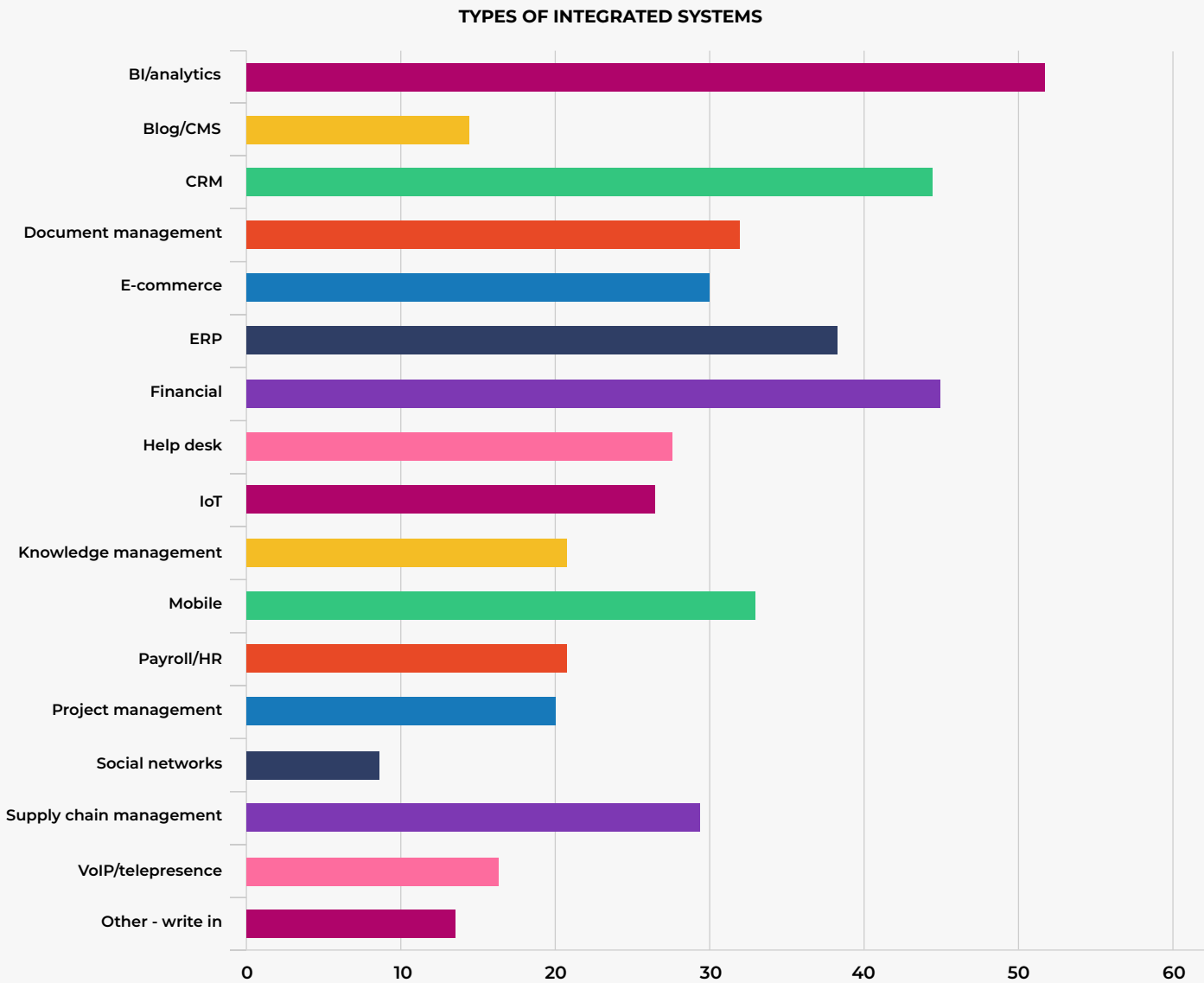
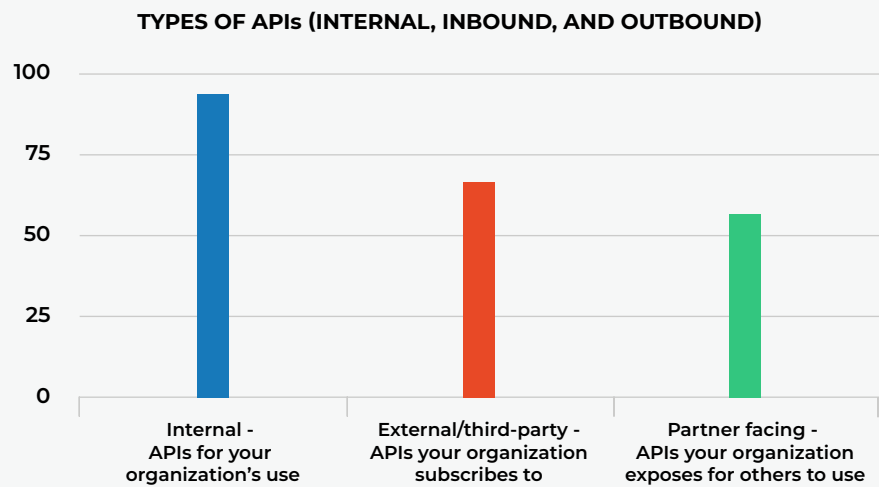


Figure 2



Observations:

- 1. Generally, there was a moderately even spread of responses among the types of integrated systems we offered in the survey question. The 16 provided selections received an average of 29% of responses, with a 10% standard deviation. This helps to illustrate how varied integration requirements can be as well as how commonly applications utilize integrated services. 94% of all respondents said their organization integrated at least one type of system; more than half of respondents (56%) said their organization integrated at least four.
- 2. "BI/analytics" was the most common type of integrated system (52%), followed by "CRM" and "Financial" integrations (both 45%). "ERP" (38%), "Mobile" (33%), "Document management" (32%), and "E-commerce" (30%) all broke 30%. "Social networks" had the lowest response rate with only 8%. Notable write-in responses included healthcare, version control, and authentication/authorization integrations.
- 3. The last time we collected survey data on types of integrated systems was for our 2019 *API Management* Trend Report. Interestingly, in the three-plus years since that survey, the change in response rates was relatively minor. The most significant increases from 2019's results were "Supply chain management" (+14%), "CRM" (+8%), "Help Desk" (+7%), "ERP" (+6%), and "VoIP/telepresence" (also +6%). "Social networks," on the other hand, fell 7% since 2019. All other given responses were within a 5% delta between 2019 and 2023.
- 4. Almost all respondents (91%) said that they work on internal APIs just for intra-company use. 68% of respondents said they work with external/third-party APIs that their company subscribes to, while 56% said they work with partner-facing APIs their organization exposes to external clients.

APPROACHING INTEGRATION PROBLEMS

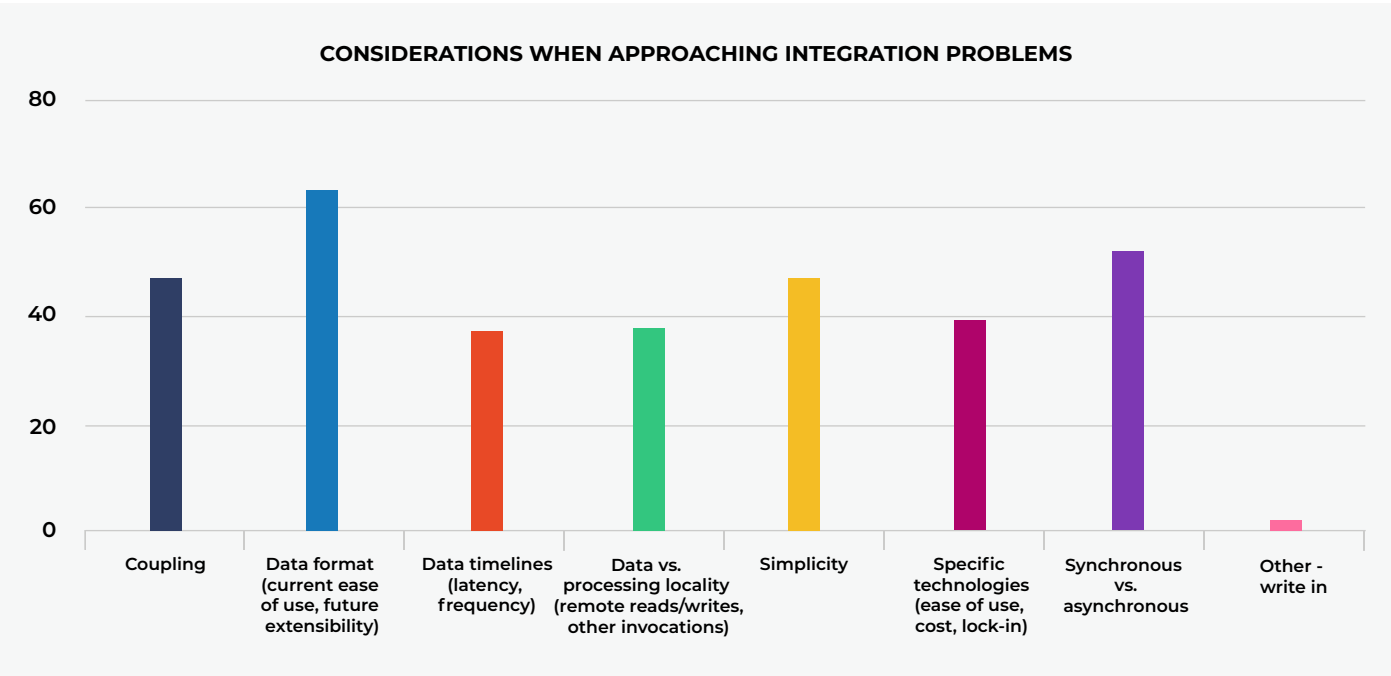
Software development is, at its root, an exercise in problem solving. Sometimes problems must be solved *ex post facto*: New software is generally created to solve an existing problem; squashing bugs and refactoring legacy code are just fixing mistakes, eliminating vulnerabilities, or strengthening fragile systems that are already there. Many of these problems, however, can be solved before they even exist — careful consideration and design can create more resilient, robust software.

We wanted to know the factors that mattered most to respondents in preventing integration problems, so we asked:

In your personal opinion, what are the most important factors to consider when approaching integration problems? Select all that apply.

Results (n=108):

Figure 3



Observations:

1. Much like we saw with the results from our question about which types of systems respondents' organizations were integrating, we again found minimal fluctuation in response rates regarding factors to consider when approaching integration problems. On average, the seven selections given received 46% of responses, with a standard deviation of 9%. This could be an indication that many of these factors may be more or less situationally specific than others, rather than more or less important.
2. Still, some options were focused on more than others. "Data format (current ease of use, future extensibility)" was the top response (62%), followed by "Synchronous vs. asynchronous" (52%), "Coupling" (47%), and "Simplicity" (46%). These options are more universally applicable to API design and integration in general: Data format and sync vs. async are both necessary decisions to make when integrating with any system, even if those decisions aren't necessarily top of mind, while loose coupling and minimizing complexity are best practices in all kinds of systems — and integrated systems are no exception.
3. Once again, the most recent previous data we have for this question is from our survey conducted in 2019, and in that time, there have been a few significant shifts. "Specific technologies (ease of use, cost, lock-in)" responses increased by 7% since 2019, which could imply a rise in adoption of integration technologies and platforms. "Synchronous vs. asynchronous" responses decreased by 9%, potentially indicating an increased familiarity with the use cases for each, causing the decision between sync and async communication to be less a "pressing consideration" and more second nature.

"Data vs. processing locality (remote reads/writes and other invocations)" increased by 6% since 2019. This could possibly be due to concerns about performance issues and cloud sprawl side effects; however, "Data timeliness (latency, frequency)" decreased by 10%, which might imply less general concern over high-performance integrations. It does seem more fitting, though, that these two linked options received almost the same response rates this year.

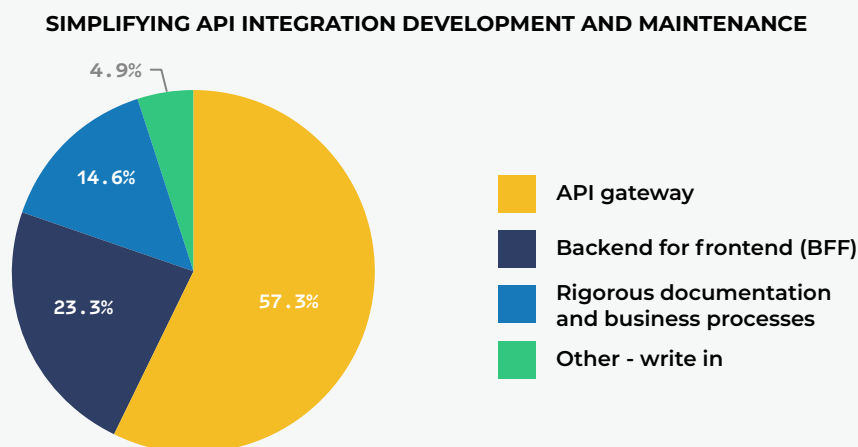
SIMPLIFYING API DEVELOPMENT AND MAINTENANCE

Simplifying API integration (i.e., reducing unnecessary complexity for the client) is an important part of creating accessible APIs — and what good is an API if it can't be accessed? We asked respondents to choose one of three potential best practices/design patterns to explain how they simplified their APIs:

How do you simplify API integration development and maintenance?

Results (n=103):

Figure 4



Observations:

1. Most respondents (57%) said they primarily use an API gateway to simplify integration development and maintenance, while 23% said they use the backend for frontend (BFF) pattern, and 15% said they opt for "Rigorous documentation and business processes" instead.
2. From our 2022 integration survey results, there was no significant change in the percentage of respondents choosing "API gateway" (58% in 2022 vs. 57% in 2023). However, respondents favoring BFF rose 5% since 2022 (from 18% to 23%), while respondents in the camp of documentation and business processes fell by 6% (from 21% to 15%). While this change

is minor, it does hint that docs and processes alone are not enough to keep API complexity in check, and that well-executed, formal design patterns are necessary for simplifying API integration. And BFF's modest popularity increase points to the growing need to factor all kinds of clients into API design.

3. Given that BFF is often considered a variant of the API gateway pattern, it is possible that limiting respondents to one selection in this question downplays the popularity of BFF. In future research, we would like to look more at how these two patterns are linked, as well as examine additional patterns/practices that may help simplify API integration development and maintenance.

Research Target Two: Integration Architectures and Paradigms

Motivations:

1. As practices and approaches to working around integration problems and simplifying API access change, so too do the architectures favored for implementing those APIs and integrations. To find out more about potential changes in the landscape of architecture paradigms, we asked respondents how their organizations were architecting their APIs.
2. There are numerous reasons why organizations might shift which architecture paradigms they favor, and we wanted to know more about those reasons. We asked respondents if they had worked on transitioning either SOAP to REST or REST to GraphQL and followed up by asking for their reasons behind the adoption of the new paradigm.
3. Beyond these architectural paradigms, we also wanted to know what kinds of architecture patterns and styles were being used, asking respondents if their organization utilized cloud-based architectures, event-driven architectures, microservices architectures, and more.

PARADIGM PREVALENCE

In our 2022 [Enterprise Application Integration](#) Trend Report "Key Research Findings," we described a condensed history of integration architectures in the form of a miniature epic (we won't repeat that tale here, but if you have not read it, we recommend checking it out on pages 11-12). Our research goal was to examine how often different API architecture paradigms were being used in respondents' organizations. Now, we want to revisit the popularity of these paradigms once again to see how often these different approaches are taken and how it compares to the results that we received last year. We asked:

What percent of each of the following API architectures exist in your organization?

Results (n=102):

Table 1

API ARCHITECTURE PREVALENCE			
API Architecture	Avg. % of Integrations	Sum	n=
SOAP	18.4%	1,285	70
REST	67.1%	6,779	101
GraphQL	13.4%	815	61
gRPC	8.9%	454	51
Other	17.2%	758	44

Observations:

1. Unsurprisingly, REST was by far the most popular API architecture paradigm — on average, respondents said that two-thirds of their organizations' architectures were REST-based (67%). Almost all respondents (98%) said that at least some REST existed in their org, 79% of respondents said at least 50% of their orgs' API architectures used REST, and 47% estimated this value to be 75% or greater. 15% of respondents said that 100% of the API architectures at their organization used REST.
2. SOAP and GraphQL architectures were reported much less commonly. On average, respondents estimated their organizations' API architectures consisted of 18% SOAP and 13% GraphQL. 54% of respondents said at least some SOAP was present in their organization, and 45% said at least some GraphQL was present. This represents a significant decrease in reported GraphQL usage from the results we saw in our 2022 Integration survey, where the average response was 23%; SOAP's average, on the other hand, was relatively unchanged, down 2% from 20% in 2022.

3. Respondents who said their organization runs any microservices (67%), on average, estimated that their organization uses GraphQL much more than respondents saying their organization did not run any microservices; the former's average GraphQL usage was 16%, compared to only a 6% average for the latter.

REASONS FOR ADOPTION

While overall paradigm usage levels may not have changed in the ways we expected, there are still organizations abandoning more antiquated paradigms in favor of more modern solutions, and we wanted to know what advantages the newer solutions offered that influenced adoption. We asked respondents if they had worked on these paradigm transitions, and if so, what their reasons were for adopting the new solution:

Have you ever worked on transitioning integrations from SOAP to REST?

Why did you adopt REST? Select all that apply.

and

Have you ever worked on transitioning integrations from REST to GraphQL?

Why did you adopt GraphQL? Select all that apply.

Results (n=101):

Figure 5

WORK ON TRANSITIONING INTEGRATIONS: SOAP TO REST AND REST TO GRAPHQL

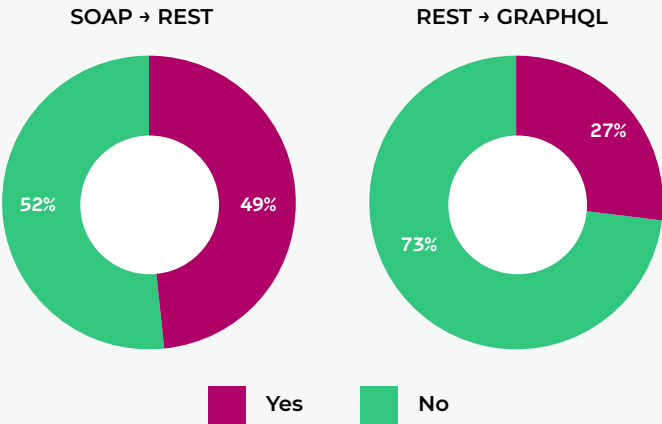


Table 2

REASONS FOR ADOPTING REST		
	%	n=
Specific commitment to HTTP verbs	54.2%	26
Easier generation of documentation	45.8%	22
Faster client development	64.6%	31
Faster app performance	37.5%	18
Wanted to support multiple clients with one API	54.2%	26
To unify customer experience across different clients/business segments	39.6%	19
General desire to stay current	41.7%	20
Other - write in	18.8%	9

Table 3

REASONS FOR ADOPTING GRAPHQL		
	%	n=
Faster client development	45.8%	11
Faster app performance	58.3%	14
Wanted to support multiple clients with one API	45.8%	11
To unify customer experiences across different clients/business segments	25.0%	6
General desire to stay current	33.3%	8
Other - write in	16.7%	4

Observations:

- 1. About half of respondents (49%) said they have worked on transitioning from SOAP to REST, while only about one-quarter (27%) said they have worked on transitioning from REST to GraphQL. This aligns with the paradigm prevalence statistics we found in the previous section, and even seems a little high for GraphQL adoption considering the low average usage response. Given that from another survey question we saw that 33% of respondents said their organization uses GraphQL in production, this may indicate that GraphQL is being adopted for more specific purposes within an organization, while REST is replacing SOAP more generally.
- 2. Most respondents who said they have worked on transitioning integrations from SOAP to REST said they did so for "Faster client development" (65%), for a "Specific commitment to HTTP verbs" (54%), and because they "Wanted to support multiple clients with one API" (54%).

Compared to last year's survey, 14% fewer respondents said they adopted REST for "Easier generation of documentation" (46% in 2023 vs. 60% in 2022), 7% fewer respondents adopted REST for "Faster client development" (65% in 2023 vs. 72% in 2022), and 7% fewer respondents adopted REST for a "Specific commitment to HTTP verbs" (54% in 2023 vs. 61% in 2022). This year, 15% more respondents said they adopted REST because of a "General desire to stay current" (42% in 2023 vs. 27% in 2022).

- 3. For respondents who have worked on transitioning from REST to GraphQL, the primary motivations were "Faster app performance" (58%), "Faster client development" and "[Wanting] to support multiple clients with one API" (46%), and a "General desire to stay current" (33%).

As opposed to reasons for switching to REST, changes in reasons for switching to GraphQL from last year were a bit less drastic. 8% fewer respondents said they adopted GraphQL for "Faster client development" (46% in 2023 vs. 54% in 2022), and 7% fewer adopted GraphQL "To unify customer experiences across different clients/business segments" (25% in 2023 vs. 32% in 2022). This year, 6% more respondents adopted GraphQL because of a "General desire to stay current" (33% in 2023 vs. 27% in 2022).

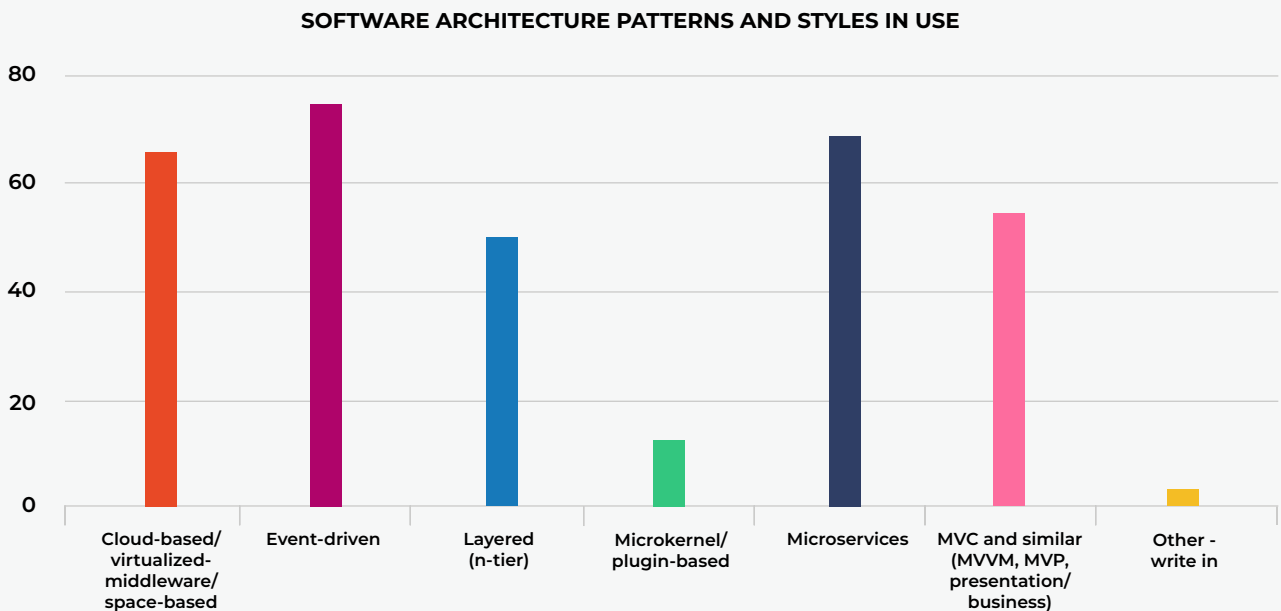
ARCHITECTURE PATTERNS AND STYLES

In addition to API architecture paradigms, we also wanted to get a sense of what kind of architectural patterns organizations were using in the systems underlying (or interconnected with) those integrations. We asked respondents:

What software architecture patterns and styles are in use at your organization? Select all that apply.

Results (n=100):

Figure 6



Observations:

- 1. Out of the six options supplied, respondents, on average, said their organization uses about three of them (2.86), and there were several popular architecture options. "Event-driven" was the most used, with three-quarters of respondents selecting this option (75%). Also favored were "Microservices" (68%), "Cloud-based/virtualized-middleware/space-based" (66%), "MVC and similar (MVVM, MVP, presentation/business)" (57%), and "Layered (n-tier)" (50%), all of which having at least half of respondents saying that these are used in their organization. "Microkernel/plugin-based" was by far the least popular pattern (14%).
- 2. Once again, the latest data we have regarding these software architecture patterns is from our 2019 API Management survey, and a lot has changed since then. Several of these patterns have seen significant increases in that time: "Event-driven" architecture increased 26% from 48% in 2019, "Cloud-based/virtualized-middleware/space-based" increased 10% from 56%, "Layered (n-tier)" increased 9% from 41%, and even "Microkernel/plugin-based" increased 8% from 5% in 2019. The only patterns that did not see significant change were "Microservices" (68% in 2023 vs. 66% in 2019) and "MVC and similar (MVVM, MVP, presentation/business)" (57% in 2023 and 2019).
- 3. Company size played a factor in the prevalence of several architecture patterns. Larger organizations (those with ≥100 employees) were 24% more likely to use event-driven architectures (76%) than smaller ones (52%), 23% more likely to use microservices architectures (75% vs. 52%), and 17% more likely to use cloud-based architectures (73% vs. 56%). There was no significant variation in the uses of layered, microkernel, or MVC architectures between larger and smaller companies.

Research Target Three: Integration Development Experience

Motivations:

- 1. As the complexity of software systems grows with time, integrating with other systems becomes more and more commonplace — and necessary. So the act of developing software becomes increasingly an act of connecting nodes as much as it is an act of creating nodes. We wanted to know how much time software professionals spent integrating, and whether they enjoyed this aspect of the job.
- 2. Diving further into the subsets of API management to which developers dedicate time, we wanted to know which primary aspects of API management software professionals were working on. We listed 28 different aspects and asked respondents to indicate whether they had contributed to each, as well as whether they had implemented each themselves.

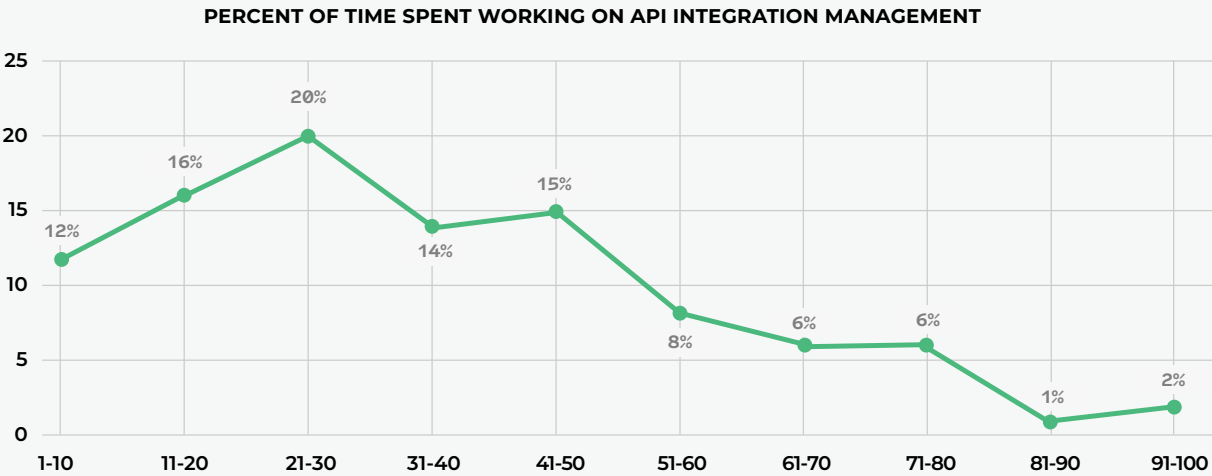
TIME AND ENJOYMENT

The process of developing software involves countless sub-processes: front ends, back ends, databases, version control systems, libraries, frameworks, project management tools... the list goes on. Generally speaking, developers have to spread their time between dozens (at least) of these sub-processes. Integration as a concept can be used to describe a subset of these. We wanted to know how much of their time, on average, software professionals spend working within that subset. We asked respondents the following:

What percent of your time at work is spent managing API integrations?

Results (n=100):

Figure 7

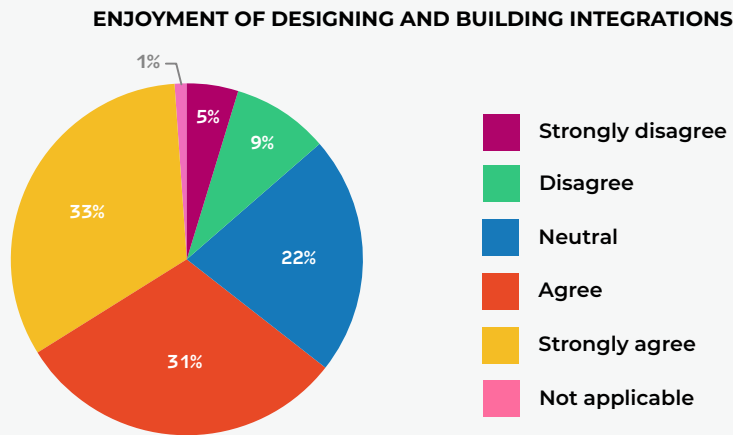


Additionally, given the time developers have to dedicate to integration, we wanted to get a sense of whether this was work that respondents found enjoyable, asking:

Agree/disagree: Your favorite part of software development is designing and building integrations.

Results (n=110):

Figure 8



Observations:

- 1. On average, respondents estimated they spend 39% of their time managing API integrations, with a standard deviation of 23% and a median of 37%. Last year, we saw respondents saying 38% of their time was spent managing API integrations, with a 24% standard deviation and 30% median. As such, the data this year has not significantly changed, on average, since last year, but it does appear to be more normalized.

88% of respondents said they spend more than 10% of their time managing APIs, more than half (51%) said they spend more than one-third of their time doing so, and almost a quarter of respondents (23%) said they spend more than half their time managing API integrations.

- 2. Respondents at organizations using microservices spend more of their time, on average, managing API integrations (estimating 43% of their time) than those at organizations not using microservices (only estimating 32% of their time). Likewise, respondents at larger organizations — orgs with ≥ 100 employees — also spent more time, on average, managing APIs, estimating 42% of their time vs. the 36% estimated by those at organizations with < 100 employees.

It should be noted that microservices use and company size were also correlated, with 73% of respondents at companies with ≥ 100 employees saying their organization uses microservices vs. 52% at organizations with < 100 employees.

- 3. Most respondents either "Strongly agree" (33%) or "Agree" (31%) that their favorite part of software development is designing and building integrations, 22% said they feel "Neutral," and a relatively small minority either "Disagree" (9%) or "Strongly disagree" (5%) that designing and building integrations is their favorite part.

This year, there were no significant differences in the responses about integration design and building favorability compared to the responses we received in last year's survey; all options were within 3% of last year's responses, well within the margin of error for the sample.

CONTRIBUTIONS TO API MANAGEMENT

As mentioned in the previous section, integration consists of a wide range of sub-processes. Just like "developing software" can describe a collection of any number of efforts, so too can "integration." To determine which of these sub-processes respondents were more likely to be working on, we offered a wide (though certainly not exhaustive) selection of API development aspects and asked respondents to answer if they had either contributed or implemented (or both) to each aspect:

Which of the following primary aspects of API management have you contributed to?

Results (n=104):

SEE TABLE 4 ON NEXT PAGE

Table 4

API MANAGEMENT CONTRIBUTIONS AND IMPLEMENTATIONS		
	% Contributed Ideas	% Implemented
API creation (e.g., resources, contracts)	74%	85%
Authentication	56%	65%
Authorization	58%	64%
API publication and deployment	59%	61%
Testing	53%	53%
Monitoring	54%	44%
Security	48%	42%
API discovery	53%	41%
Availability	43%	39%
Identity mediation (e.g., OAuth, SAML)	43%	39%
API proxy (e.g., setup, maintenance)	47%	38%
Version management	45%	38%
Encryption	40%	37%
Key and certificate management	35%	31%
Lifecycle management	45%	31%
Quality of service	49%	31%
Service routing	30%	31%
Service orchestration	37%	30%
Governance	37%	29%
Interface translation	30%	27%
Service-level monitoring	33%	27%
Service discovery	37%	25%
Scaling management	40%	24%
Service-level agreements (SLAs)	30%	24%
Service registry	33%	23%
Traffic management	32%	23%
Transformation	34%	23%
Threat protection	32%	20%

Observations:

1. On average, respondents contributed to about 11 (10.7) of the 28 API management aspects provided in this question, with a standard deviation of 9.09. Respondents implemented an average of about nine (9.3) of these aspects, with a standard deviation of 7.80. The median number of contributions and implementations was nine.
2. "API creation (e.g., resources, contracts)" was by far the most popular aspect of API management to be both contributed to and implemented; 74% of respondents said they had contributed to API creation, and 85% said they have implemented it themselves. This may be partially attributed to this particular option being broader and more generalized than most of the other response options, but it still serves to show how ubiquitous API creation is in the modern software development landscape.

3. Other popular API management aspects included "Authentication" (65% implemented, 56% contributed), "Authorization" (64% implemented, 58% contributed), "API publication and deployment" (61% implemented, 59% contributed), and "Testing" (53% implemented and contributed), each of which having been implemented by at least half of respondents.

Future Research

Our analysis here only touched the surface of the available data, and we will look to refine and expand our Software Integration survey as we produce further Trend Reports. Some of the topics we didn't get to in this report, but were incorporated in our survey, include:

- Integration difficulties
- Message schema enforcement
- GraphQL architectures
- Microservices benefits and pains
- API security and privacy sentiments

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



G. Ryan Spain, Freelance Software Engineer, Former Engineer & Editor at DZone

[@grspain](#) on DZone | [@grspain](#) on [GitHub](#) and [GitLab](#) | gryanspain.com

G. Ryan Spain lives on a beautiful two-acre farm in McCalla, Alabama with his lovely wife and adorable dog. He is a polyglot software engineer with an MFA in poetry, a die-hard Emacs fan and Linux user, a lover of The Legend of Zelda, a journeyman data scientist, and a home cooking enthusiast. When he isn't programming, he can often be found listening to The Adventure Zone with a glass of red wine or a cold beer.



The Path From APIs to Containers

How Microservices Fueled the Journey

By Saurabh Dashora, Architect at ProgressiveCoder

In recent years, the rise of microservices has drastically changed the way we build and deploy software. The most important aspect of this shift has been the move from traditional API architectures driven by monolithic applications to containerized microservices. This shift not only improved the **scalability** and **flexibility** of our systems, but it has also given rise to new ways of software development and deployment approaches.

In this article, we will explore the path from APIs to containers and examine how microservices have paved the way for enhanced API development and software integration.

The Two API Perspectives: Consumer and Provider

The inherent purpose of building an API is to exchange information. Therefore, APIs require two parties: consumers and providers of the information. However, both have completely different views.

For an **API consumer**, an API is nothing more than an interface definition and a URL. It does not matter to the consumer whether the URL is pointing to a mainframe system or a tiny IoT device hosted on the edge. Their main concern is ease of use, reliability, and security.

An **API provider**, on the other hand, is more focused on the scalability, maintainability, and monetization aspects of an API. They also need to be acutely aware of the infrastructure behind the API interface. This is the place where APIs actually live, and it can have a lot of impact on their overall behavior. For example, an API serving millions of consumers would have drastically different infrastructure requirements when compared to a single-consumer API. The success of an API offering often depends on how well it performs in a production-like environment with real users.

With the explosion of the internet and the rise of [always-online applications like Netflix, Amazon, Uber](#), and so on, API providers had to find ways to meet the increasing demand. They could not rely on large monolithic systems that were difficult to change and scale up as and when needed. This increased focus on scalability and maintainability, which led to the rise of microservices architecture.

The Rise of Microservices Architecture

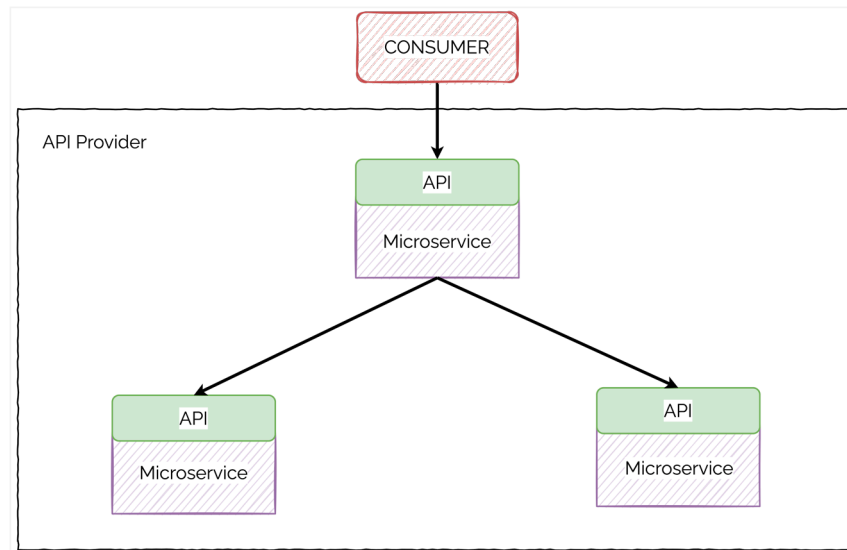
Microservices are not a completely new concept. They have been around for many years under various names, but the official term was actually [coined by a group of software architects](#) at a workshop near Venice in 2011/2012. The goal of microservices has always been to make a system flexible and maintainable. This is an extremely desirable target for API providers and led to the widespread adoption of microservices architecture styles across a wide variety of applications.

The adoption of microservices to build and deliver APIs addressed several challenges by providing important advantages:

- Since microservices are developed and deployed independently, they allow developers to work on different parts of the API in parallel. This reduces the time to market for new features.
- Microservices can be scaled up or down to meet the varying demands of specific API offerings. This helps to improve resource use and cost savings.
- There is a much better distribution of API ownership as different teams can focus on different sets of microservices.
- By breaking down an API into smaller and more manageable services, it becomes theoretically easier to manage outages and downtimes. This is because one service going down does not mean the entire application goes down.

The API consumers also benefit due to the microservices-based APIs. In general, consumer applications can model better interactions by integrating a bunch of smaller services rather than interfacing with a giant monolith.

Figure 1: APIs perspectives for consumer and provider



Since each microservice has a smaller scope when compared to a monolith, there is less impact on the client application in case of changes to the API endpoints. Moreover, testing for individual interactions becomes much easier.

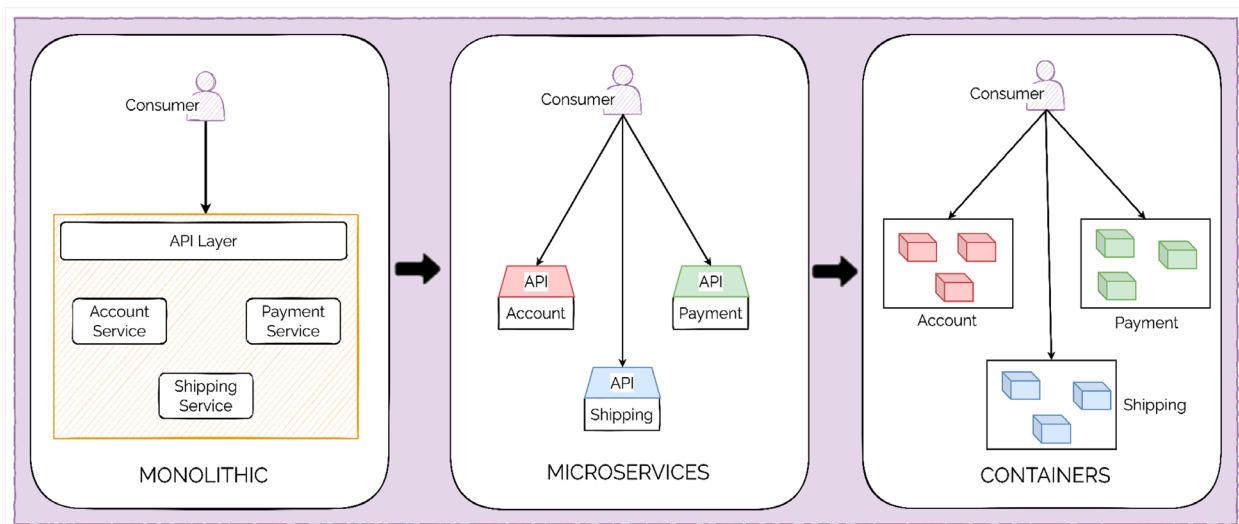
Ultimately, the **rise of microservices enhanced the API-development landscape**. Building an API was no longer a complicated affair. In fact, APIs became the de facto method of communication between different systems. Nonetheless, despite the huge number of benefits provided by microservices-based APIs, they also brought some initial challenges in terms of deployments and managing dependencies.

Streamlining Microservices Deployment With Containers

The twin challenges of deployment and managing dependencies in a microservices architecture led to the rise in container technologies. Over the years, containers have become increasingly popular, particularly in the context of microservices. With containers, we can easily package the software with its dependencies and configuration parameters in a container image and deploy it on a platform. This makes it trivial to manage and isolate dependencies in a microservices-based application.

Containers can be deployed in parallel, and each deployment is predictable since everything that is needed by an application is present within the container image. Also, containers make it easier to scale and load balance resources, further boosting the scalability of microservices and APIs. Figure 2 showcases the evolution from monolithic to containerized microservices:

Figure 2: Evolution of APIs from monolithic to containerized microservices



Due to the rapid advancement in cloud computing, container technologies and orchestration frameworks are now natively available on almost all cloud platforms. In a way, the growing need for **microservices and APIs boosted the use of containers** to deploy them in a scalable manner.

The Future of Microservices and APIs

Although APIs and microservices have been around for numerous years, they have yet to reach their full potential. Both [are going to evolve together in this decade](#), leading to some significant trends. One of the major trends is around **API governance**. [Proper API governance](#) is essential to make your APIs discoverable, reusable, secure, and consistent. In this regard, [OpenAPI](#), a language-agnostic interface to RESTful APIs, has more or less become the prominent and standard way of documenting APIs. It can be used by both humans and machines to discover and understand an API's capabilities without access to the source code.

Another important trend is the growth in **API-powered capabilities** in the fields of NLP, image recognition, sentiment analysis, predictive analysis, chatbot APIs, and so on. With the increased sophistication of models, this trend is only going to grow stronger, and we will see many more applications of APIs in the coming years. The rise of tools like [ChatGPT](#) and [Google Bard](#) shows that we are only at the beginning of this journey.

A third trend is the increased use of **API-driven DevOps** for deploying microservices. With the rise of cloud computing and DevOps, managing infrastructure is an extremely important topic in most organizations. API-driven DevOps is a key enabler for Infrastructure as Code tools to provision infrastructure and deploy microservices. Under the covers, these tools rely on APIs exposed by the platforms.

Apart from major ones, there are also other important trends when it comes to the future of microservices and APIs:


- There is a growing role of API enablement on the edge networks to power millions of IoT devices.
- API security practices have become more important than ever in a world of unprecedented integrations and security threats.
- API ecosystems are expanding as more companies develop a suite of APIs that can be used in a variety of situations to build applications. Think of API suites like Google Maps API.
- There is an increased use of API gateways and service meshes to improve reliability, observability, and security of microservices-based systems.

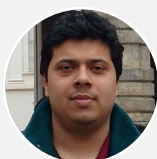
Conclusion

The transition from traditional APIs delivered via monolithic applications to microservices running on containers has opened up a world of possibilities for organizations. The change has enabled developers to build and deploy software faster and more reliably without compromising on the scalability aspects. They have made it possible to build extremely complex applications and operate them at an unprecedented scale.

Developers and architects working in this space should first focus on the key API trends such as governance and security. However, as these things become more reliable, they should explore cutting-edge areas such as API usage in the field of artificial intelligence and DevOps. This will keep them abreast with the latest innovations. Despite the maturity of the **API and microservices ecosystem**, there is a lot of growth potential in this area. With more advanced capabilities coming up every day and DevOps practices making it easier to manage the underlying infrastructure, the future of APIs and microservices looks bright.

References:

- ["A Brief History of Microservices"](#) by Keith D. Foote
- ["The Future of APIs: 7 Trends You Need to Know"](#) by Linus Håkansson
- ["Why Amazon, Netflix, and Uber Prefer Microservices over Monoliths"](#) by Nigel Pereira
- ["Google Announces ChatGPT Rival Bard, With Wider Availability in 'Coming Weeks'"](#) by James Vincent
- ["Best Practices in API Governance"](#) by Janet Wagner
- ["APIs Impact on DevOps: Exploring APIs Continuous Evolution,"](#) xMatters Blog 



Saurabh Dashora, Architect at Progressive Coder

[@saurabh.dashora](#) on DZone | [saurabh-dashora](#) on LinkedIn

I'm a full stack architect, a tech writer, and guest author in various publications. I have expertise building distributed systems across multiple business domains such as banking, autonomous driving, and retail. Throughout my career, I have worked at several large organizations. I also run a tech blog on cloud,

microservices, and web development where I have written hundreds of articles. Apart from work, I enjoy reading books and playing video games.



De-Risk Your Investment In Microservices-Based Composable Commerce

Why Composable?	How We Do It?
Agile delivery and faster time to market with modular MACH architecture	De-coupled MACH-based architecture with a focus on product experience manager
Build your preferred multi-vendor commerce solution	Create the solutions you need with opinionated and elegant APIs
Keep up with business team feature requirements	Accelerate frontend builds with pre-integrated NextJS framework
Innovate and make changes with speed	Simplified multi-vendor design and orchestration with no-code and low-code integrations
Reduce TCO and operational costs	Reduce cloudops resources with solution hosting, management and monitoring included

Start Free Trial

Case Study: Paro

Paro Implements Elastic Path to Launch Self-Service Portal

Paro is a growth platform bringing businesses and expert finance and accounting professionals together through AI technology and acute industry knowledge. Paro's community of professionals provides a range of services to clients, from bookkeeping and accounting to highly specialized corporate development and strategic advisory.

Challenge

Paro initially sought to launch a portal for their freelance experts to self-serve transactions surrounding placement and financial services. The initial launch served as a test toward addressing more complexities within their product offerings as their talent pool, and subsequently their menu of services, expands.

Given the limited resources of their engineering team and the fast-paced nature of the startup, Paro was looking for a solution with ease of use and a quick time to market.

Solution

Paro's e-commerce portal demanded:

- Flexibility
- Agility
- Ease of implementation
- Ability to scale at speed

Elastic Path's API-first, headless architecture played a critical role in providing the flexibility to innovate rapidly. Paro believes microservices are the future of e-commerce's fast-paced evolution. They pointed to Elastic Path Product Experience Manager (EP PXM), which combines re-imagined commerce PIM, product merchandising, and catalog composer capabilities in one central place.

These features empowered Paro's team to quickly create thousands of variations, support configurable product types, and power dynamic bundles such as service packages.

Results

This new capability will significantly impact Paro's ability to scale their products and services without additional lift to their existing teams. What lies ahead for Paro is a focus on conversion and the growth of services afforded by EP PXM.

COMPANY

Paro

COMPANY SIZE

150+ employees

INDUSTRY

Professional Services

PRODUCTS USED

Elastic Path Commerce Cloud

PRIMARY OUTCOME

Paro successfully launched their e-commerce portal in five months and leveraged Elastic Path technology partners — Algolia for search functionality and Stripe for payment processing.

"With Elastic Path, we could focus on scaling, future-proofing, and ultimately providing the experience we wanted for our customers."

— Kody Myers,
Director of Product, Paro

CREATED IN PARTNERSHIP WITH



Full Lifecycle API Management Is Dead

Build APIs Following Your Software Development Lifecycle With an Internal Developer Platform

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

As organizations look to enable integration, innovation, and digital experiences through their IT teams, they often build APIs and expose them by leveraging a full-lifecycle API management system. Historically, these API management systems provided tooling such as:

- Defining an API (e.g., Swagger, OpenAPI Spec, RAML)
- API testing
- API scaffolding and implementation
- Specifications for quota and usage policies/plans
- Documentation
- An API portal

These API management systems were often delivered as a fully integrated stack with a fancy UI, role-based access control, and push-button mechanisms to accomplish the lifecycle management functions.

While this all sounds very nice, there are some realities we face as organizations look to modernize their application and API delivery engines. An API management platform does not exist in a vacuum. DevOps philosophies have influenced organizational structures, automation, and self-service. Any API management system must fit within a modern development environment that is often multi-language, multi-platform, and multi-cloud. This infrastructure must also fit natively with Git-based deployment workflows (GitOps), including systems built for CI/CD.

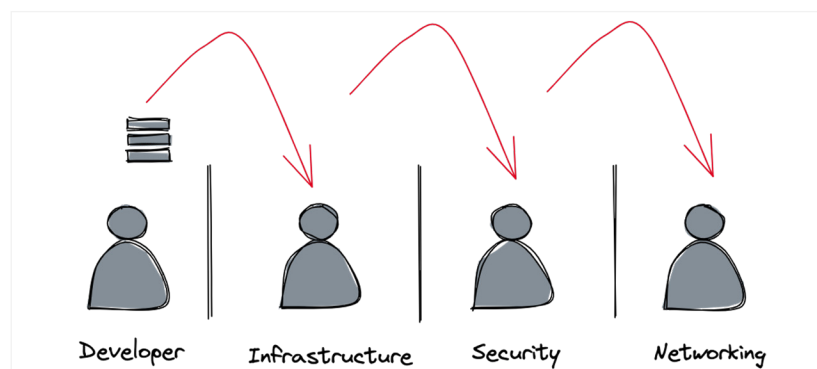
Avoid Yet Another Silo (YAS)

Although developer productivity can be difficult to measure, proxy metrics that can be useful include things like the following:

- Lead time to make code changes in production
- Number of deployments to production per week

Traditionally, developers write code, create services, build APIs, and then hand them off to operations to deploy and operate those services and APIs. The silos between development, infrastructure, security, and network teams often leads to complex synchronization points, handoffs, and a lot of waiting. This slows down code changes and deployments to production.

Figure 1: Siloed handoffs between teams cause a slowdown in delivery to production



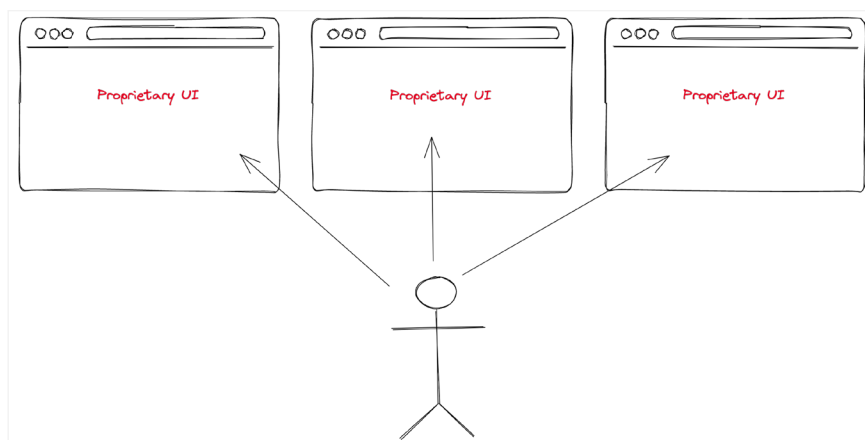
Large monolithic software systems can further this problem by forcing their own silos within each of the organizational silos. They have their own proprietary UIs, require specialized skills or privilege to operate, and are often owned by specific teams. If you need something from the large monolithic software system, you typically need to open a ticket to signal to the team who owns the system that they need to make a change.

In practice, traditional full lifecycle API management systems create silos by forcing users into an all-or-nothing set of tools for defining, implementing, testing, and exposing APIs even if these differ from what a development team wants to use. These systems are very difficult to automate and integrate with other parts of the software delivery systems, and they are usually guarded by some API management team that is responsible for configuring and deploying APIs. This centralization from both a technology and organizational standpoint creates bottlenecks that slow down delivery in a modern DevOps-minded organization.

Favor Automation Over Point-and-Click UIs

Most traditional full lifecycle API management systems do have some role-centric capabilities, like role-based UIs and tools for specific personas. One principle prevalent in modern DevOps implementations is around eliminating manual or repetitive tasks using automation. We cannot expect users to log into a system that runs tests, a totally different system to manage APIs, and yet another system to do a deployment.

Figure 2: We should reduce multiple, manual, point-and-click UIs in favor of automation



Ideally, we would automate a lot of these steps so a developer can go to a single self-service UI for anything related to software development and deployment. Any functionality we would like, including traditional API management and each of its "full lifecycle" functionalities, should be automatable. With a lot of the functionality in modern API management locked into proprietary UIs, automation is often very challenging and brittle, if accomplished at all.

The API Lifecycle Is The Software Development Lifecycle

The API lifecycle is often centered around design, implementation, testing, control, and consumption. Does this sound familiar? It should — because it's exactly what we do with any software we write. When developers create APIs, they use software to do so. The API lifecycle *is* the software development lifecycle. Trying to treat the lifecycle of APIs differently from the rest of our software development practices creates inconsistencies, fragmentation, and friction.

For example, when we create an API, we may need to develop it, test it, and will probably eventually need to notify users when we need to retire it. We need the same capabilities for internal services, libraries, and other system components. Although there may be some slight differences, should these be separate and different processes? Should these be completely different sets of tools? Trying to duplicate what is already necessary for the software development lifecycle with substandard and proprietary tools specific for API management causes adoption, governance, and bifurcation issues.

Use an Internal Developer Platform

As organizations attempt to improve developer productivity by shifting left and giving developers more responsibility and control over building and running their services and APIs, we've seen an emergence in platform teams responsible for building workflows and toolchains that enable self-service. These workflows get boiled down to "golden paths" that developers can easily follow and that automate a lot of the tasks around bootstrapping new projects, documenting their software, enforcing access/security policies, and controlling deployment rollouts.

This developer-focused self-service platform is known as an Internal Developer Platform (IDP) and aims to cover the operational necessities of the entire lifecycle of a service. Although many teams have built their own platforms, there are some good open-source frameworks that go a long way to building an IDP. For example, [Backstage](#) is a popular open-source project used to build IDPs.

Platform engineering teams typically have a lot of flexibility picking the best of breed tools for developers that support multiple types of languages and developer frameworks. Plus, these tools can be composed through automation and don't rely on proprietary vendor UIs. Platform engineering teams also typically build their platform around container technology that can be used across multiple clusters and stretch into on-premises deployments as well as the public cloud. These IDPs insulate from vendor lock-in whether that's a particular public cloud or vendor.

For example, here's a very common scenario that I've run into numerous times: An organization bought into a full-lifecycle API management vendor and finds itself in a situation where their modernization efforts are centered around containers and Kubernetes, GitOps, and CI/CD. They find the API management vendor may have strong tools around API design; however, runtime execution, the API portal, and analytics features are lagging, outdated, or cannot be automated with the rest of the container platform via GitOps. They often wish to use a different API gateway technology based on more modern open-source proxies like [Envoy Proxy](#) but are locked into a tightly integrated yet outdated gateway technology with their current vendor.

Instead, these organizations should opt to use newer proxy technologies, select more developer-friendly API testing tools, tie API analytics into their existing streaming and analytics efforts, and rely on tools like Backstage to tie all of this together. Doing so, they would reduce silos centered around vendor products, leverage best-of-breed tools, and automate these tools in a way that preserves governance and prescribed guard rails. These platforms can then support complex deployment strategies like multi-cluster, hybrid, and multi-cloud deployments.

Conclusion

Managing APIs will continue to be an important aspect of software development, but it doesn't happen in a vacuum. Large monolithic full lifecycle API management stacks are outdated, don't fit in with modern development practices, and cause silos when we are trying to break down silos. Choosing the best-of-breed tools for API development and policy management allows us to build a powerful software development platform (an IDP) that improves developer productivity, reduces lock-in, and allows organizations to deploy APIs and services across containers and cloud-infrastructure whether on-premises or any public cloud. 🧩



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.



REST vs. Messaging for Microservices

Choosing the Right Communication Style for Your Microservices

By Swathi Prasad, Software Architect at Syneco Trading GmbH

A microservices architecture is an established pattern for building a complex system that consists of loosely coupled modules. It is one of the most talked-about software architecture trends in the last few years. It seems to be a surprisingly simple idea to break a large, interdependent system into many small, lightweight modules that can make software management easier.

Here's the catch: After you have broken down your monolith application into small modules, how are you supposed to connect them together in a meaningful way? Unfortunately, there is no single right answer to this question, but as is so often the case, there are a few approaches that depend on the application and the individual use case.

Two common protocols used in microservices are HTTP request/response with resource APIs and lightweight asynchronous messaging when communicating updates across several microservices. Let's explore these protocols.

Types of Communication

Microservices can communicate through many different modes of communication, each one targeting a different use case. These types of communications can be primarily classified in two dimensions. The first dimension defines if the communication protocol is synchronous or asynchronous:

Table 1

SYNCHRONOUS vs. ASYNCHRONOUS COMMUNICATION		
	Synchronous	Asynchronous
Communication pattern	The client sends a request and waits for a response from the server.	Communication is not in sync, which means it does not happen in real time.
Protocols	HTTP/HTTPS	AMQP, MQTT
Coupling	The client code can only continue its task further when it receives the server response.	In the context of distributed messaging, coupling implies that request processing will occur at an arbitrary point in time.
Failure isolation	It requires the downstream server to be available or the request fails.	If the consumer fails, the sender can still send messages. The messages will be picked up when the consumer recovers.

The second dimension defines if the communication has a single receiver or multiple receivers:

Table 2

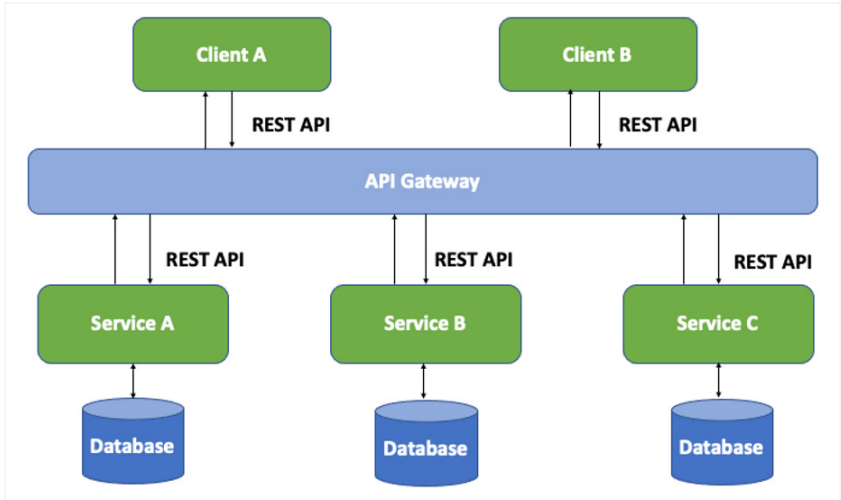
COMMUNICATION VIA SINGLE vs. MULTIPLE RECEIVERS		
	Single Receiver	Multiple Receivers
Communication pattern	It implies that there is point-to-point communication that delivers a message to exactly one consumer that is reading from the channel, and that the message is processed only once.	Communication from the sender is available to multiple receivers.
Example	It is well suited for sending asynchronous commands from one microservice to another.	The publish/subscribe mechanism is where a publisher publishes a message to a channel and the channel can be subscribed by multiple subscribers/receivers to receive the message asynchronously.

The most common type of communication between microservices is single-receiver communication with a synchronous protocol like HTTP/HTTPS when invoking a REST API. Microservices typically use messaging protocols for asynchronous communication between microservices. This asynchronous communication may involve a single receiver or multiple receivers depending on the application's needs.

Representational State Transfer

Representational state transfer (REST) is a popular architectural style for request and response communication, and it can serve as a good example for the synchronous communication type. This is based on the HTTP protocol, embracing verbs such as GET, POST, PUT, DELETE, etc. In this communication pattern, the caller waits for a response from the server.

Figure 1: REST API-based communication



REST is the most commonly used architectural style for communication between services, but heavy reliance on this type of communication has some negative consequences when it comes to a microservices architecture:

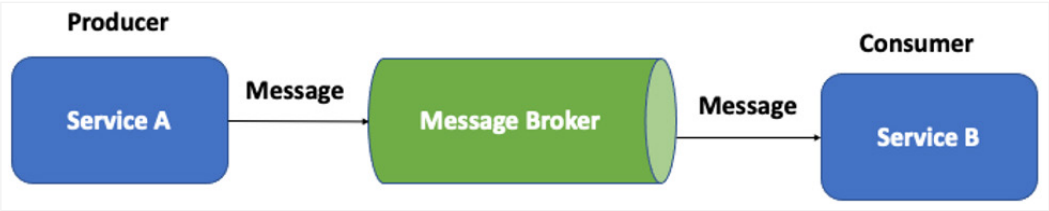
- 1. **Multiple round trips (latency)** – The client often needs to execute multiple trips to the server to fetch all the data the client requires. Each endpoint specifies a fixed amount of data, and in many cases, that data is only a subset of what a client needs to populate their page.
- 2. **Blocking** – When invoking a REST API, the client is blocked and is waiting for a server response. This may hurt application performance if the application thread is processing other concurrent requests.
- 3. **Tight coupling** – The client and server need to know about each other. It increases complexity over time and reduces portability.

Messaging

Messaging is widely used in a microservices architecture, which follows the asynchronous protocol. In this pattern, a service sends a message without waiting for a response, and one or more services process the message asynchronously. Asynchronous messaging provides many benefits but also brings challenges such as idempotency, message ordering, poison message handling, and complexity of message broker, which must be highly available.

It is important to note the difference between asynchronous I/O and the asynchronous protocol. **Asynchronous I/O** means that the calling thread is not blocked while the I/O operations are executed. This is an implementation detail in terms of the software design. The **asynchronous protocol** means the sender does not wait for a response.

Figure 2: Messaging-based communication



Asynchronous messaging has some advantages over synchronous messaging:

1. **Loose coupling** – The message producer does not need to know about the consumer(s).
2. **Multiple subscribers** – Using a publisher/subscriber (pub/sub) model, multiple consumers can subscribe to receive events.
3. **Resiliency or failure isolation** – If the consumer fails, the producer can still send messages. The messages will be picked up when the consumer recovers from failure. This is especially useful in a microservices architecture because each microservice has its own lifecycle.
4. **Non-blocking** – The producers and consumers can send and process messages at their own pace.

Though asynchronous messaging has many advantages, it comes with some tradeoffs:

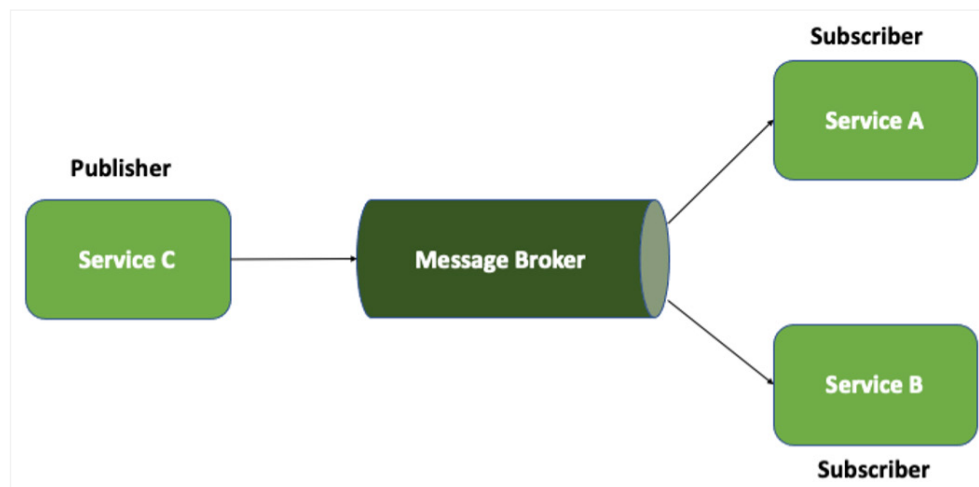
1. **Tight coupling with the messaging infrastructure** – Using a particular vendor/messaging infrastructure may cause tight coupling with that infrastructure. It may become difficult to switch to another vendor/messaging infrastructure later.
2. **Complexity** – Handling asynchronous messaging may not be as easy as designing a REST API. Duplicate messages must be handled by de-duplicating or making the operations idempotent. It is hard to implement request-response semantics using asynchronous messaging. To send a response, another queue and a way to correlate request and response messages are both needed. Debugging can also be difficult as it is hard to identify which request in Service A caused the wrong behavior in Service B.

Asynchronous messaging has matured into a number of messaging patterns. These patterns apply to scenarios when several parts of a distributed system must communicate with one another in a dependable and scalable way. Let's take a look at some of these patterns.

PUB/SUB PATTERN

The pub/sub pattern implies that a publisher sends a message to a channel on a message broker. One or more subscribers subscribe to the channel and receive messages from the channel in an asynchronous manner. This pattern is useful when a microservice needs to broadcast information to a significant number of consumers.

Figure 3: Pub/sub pattern



The pub/sub pattern has the following advantages:

- It **decouples publishers and subscribers** that need to communicate. Publishers and subscribers can be managed independently, and messages can be managed even if one or more subscribers are offline.
- It **increases scalability** and **improves responsiveness** of the publisher. The publisher can quickly publish a message to the input channel, then return to its core processing responsibilities. The messaging infrastructure is responsible for ensuring messages are delivered to interested subscribers.
- It provides **separation of concerns** for microservices. Each microservice can focus on its core responsibilities, while the message broker handles everything required to reliably route messages to multiple subscribers.

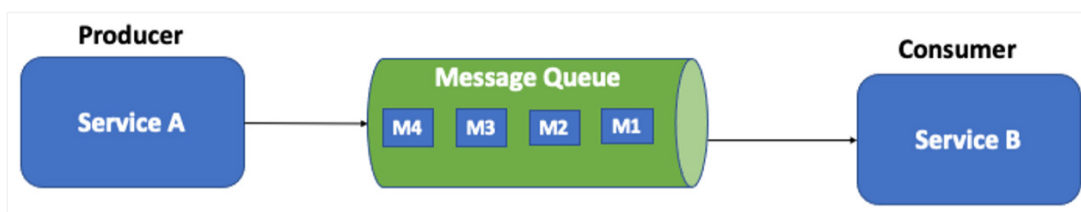
There are a few disadvantages of using this pattern:

- The pub/sub pattern introduces **high semantic coupling** in the messages passed by the publishers to the subscribers. Once the structure of the data is established, it is often difficult to change. To change the message structure, all subscribers must be altered to accept the changed format. This can be difficult or impossible if the subscribers are external.
- Another drawback of the pub/sub pattern is that it is **difficult to gauge the health** of subscribers. The publisher does not have knowledge of the health status of the systems listening to the messages.
- As a pub/sub system scales, the broker often **becomes a bottleneck** for message flow. Load surges can slow down the pub/sub system, and subscribers can get a spike in response time.

QUEUE-BASED PATTERN

In the queue-based pattern, a sender posts a message to a queue containing the data required by the receiver. The queue acts as a buffer, storing the message until it is retrieved by the receiver. The receiver retrieves messages from the queue and processes them at its own pace. This pattern is useful for any application that uses services that are subject to overloading.

Figure 4: Queue-based pattern



The queue-based pattern has the following advantages:

- It can help **maximize scalability** because both the number of queues and the number of services can be scaled to meet demand.
- It can help **maximize availability**. Delays arising in the producer or consumer won't have an immediate or direct impact on the services, which can continue to post messages to the queue even when the consumer isn't available or is under heavy load to process messages.

There are some disadvantages of using this pattern:

- When a consumer receives a message from the queue, the message is **no longer available** in the queue. If a consumer fails to process the message, the message is lost and may need a rollback in the consumer.
- Message queues do not come out of the box. We need to create, configure, and monitor them. It can cause **operational complexity** when systems are scaled up.

KEYS TO STREAMLINED MESSAGING INFRASTRUCTURE

Asynchronous communication is usually managed through a message broker. There are some factors to consider when choosing the right messaging infrastructure for asynchronous communication:

- **Scalability** – the ability to scale automatically when there is a load surge on the message broker
- **Data persistency** – the ability to recover messages in case of reboot/failure
- **Consumer capability** – whether the broker can manage one-to-one and/or one-to-many consumers
- **Monitoring** – whether monitoring capabilities are available
- **Push and pull queue** – the ability to handle push and pull delivery by message queues
- **Security** – proper authentication and authorization for messaging queues and topics
- **Automatic failover** – the ability to connect to a failover broker automatically when one broker fails without impacting publisher/consumer

Conclusion

More and more, microservices are becoming the de facto approach for designing scalable and resilient systems. There is no single approach for all communications between microservices. While RESTful APIs provide a request-response model to communicate between services, asynchronous messaging offers a more scalable producer-consumer relationship between

different services. And although microservices can communicate with each other via both messaging and REST APIs, messaging architectures are ideal for improving agility and moving quickly. They are commonly found in modern applications that use microservices or any application that has decoupled components.

When it comes to choosing a right style of communication for your microservices, be sure to match the needs of the consumer with one or more communication types to offer a robust interface for your services. 🧩



Swathi Prasad, Software Architect at Syneco Trading GmbH

[@swathiprasad88](#) on DZone | [@swathisprasad](#) on LinkedIn

Swathi Prasad has 13 years of experience in the IT industry. She is experienced in the full software development lifecycle. She also enjoys teaching backend development at a non-profit school in Munich. Outside of work, Swathi spends time reading, writing, and hiking.



Assessment of Scalability Constraints (and Solutions)

Practical Advice for Overcoming Scalability Challenges

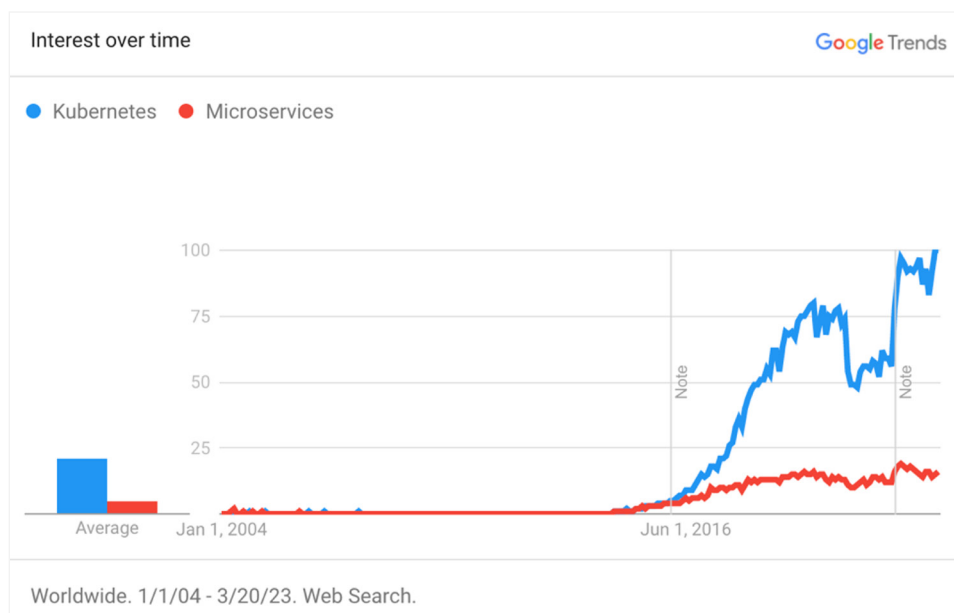
By Shai Almog, CEO at Codename One

Our approach to scalability has gone through a tectonic shift over the past decade. Technologies that were staples in every enterprise back end (e.g., IIOP) have vanished completely with a shift to approaches such as [eventual consistency](#). This shift introduced some complexities with the benefit of greater scalability. The rise of Kubernetes and serverless further cemented this approach: spinning a new container is cheap, turning scalability into a relatively simple problem. Orchestration changed our approach to scalability and facilitated the growth of microservices and observability, two key tools in modern scaling.

Horizontal to Vertical Scaling

The rise of Kubernetes correlates with the microservices trend as seen in Figure 1. Kubernetes heavily emphasizes horizontal scaling in which replications of servers provide scaling as opposed to vertical scaling in which we derive performance and throughput from a single host (many machines vs. few powerful machines).

Figure 1: Google Trends chart showing correlation between Kubernetes and microservices



Data source: Google Trends (<https://www.google.com/trends>)

In order to maximize horizontal scaling, companies focus on the idempotency and statelessness of their services. This is easier to accomplish with smaller isolated services, but the complexity shifts in two directions:

- **Ops** – Managing the complex relations between multiple disconnected services.
- **Dev** – Quality, uniformity, and consistency become an issue.

Complexity doesn't go away because of a switch to horizontal scaling. It shifts to a distinct form handled by a different team, such as network complexity instead of object graph complexity. The consensus of starting with a monolith isn't just about the ease of programming. Horizontal scaling is deceptively simple thanks to Kubernetes and serverless. However, this masks a level of complexity that is often harder to gauge for smaller projects. Scaling is a process, not a single operation; processes take time and require a team.

A good analogy is physical traffic: we often reach a slow junction and wonder why the city didn't build an overpass. The reason could be that this will ease the jam in the current junction, but it might create a much bigger traffic jam down the road. The same is true for scaling a system — all of our planning might make matters worse, meaning that a faster server can overload a node in another system. Scalability is not performance!

SCALABILITY vs. PERFORMANCE

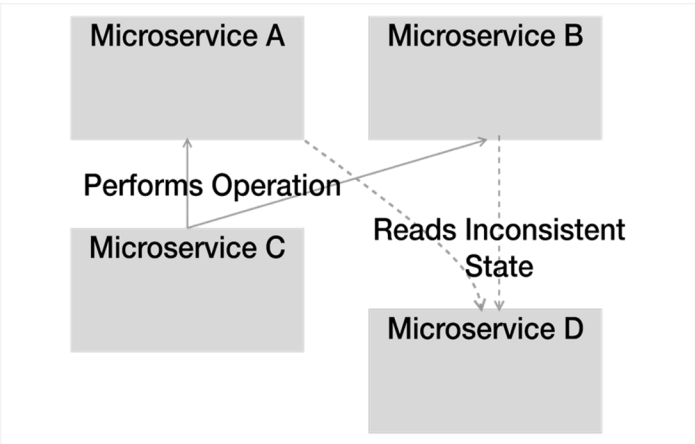
Scalability and performance can be closely related, in which case improving one can also improve the other. However, in other cases, there may be trade-offs between scalability and performance. For example, a system optimized for performance may be less scalable because it may require more resources to handle additional users or requests. Meanwhile, a system optimized for scalability may sacrifice some performance to ensure that it can handle a growing workload.

To strike a balance between scalability and performance, it's essential to understand the requirements of the system and the expected workload. For example, if we expect a system to have a few users, performance may be more critical than scalability. However, if we expect a rapidly growing user base, scalability may be more important than performance. We see this expressed perfectly with the trend towards horizontal scaling. Modern Kubernetes systems usually focus on many small VM images with a limited number of cores as opposed to powerful machines/VMs. A system focused on performance would deliver better performance using few high-performance machines.

CHALLENGES OF HORIZONTAL SCALE

Horizontal scaling brought with it a unique level of problems that birthed new fields in our industry: platform engineers and SREs are prime examples. The complexity of maintaining a system with thousands of concurrent server processes is fantastic. Such a scale makes it much harder to debug and isolate issues. The asynchronous nature of these systems exacerbates this problem. Eventual consistency creates situations we can't realistically replicate locally, as we see in Figure 2. When a change needs to occur on multiple microservices, they create an inconsistent state, which can lead to invalid states.

Figure 2: Inconsistent state may exist between wide sweeping changes



Typical solutions used for debugging dozens of instances don't apply when we have thousands of instances running concurrently. Failure is inevitable, and at these scales, it usually amounts to restarting an instance. On the surface, orchestration solved the problem, but the overhead and resulting edge cases make fixing such problems even harder.

Strategies for Success

We can answer such challenges with a combination of approaches and tools. There is no "one size fits all," and it is important to practice agility when dealing with scaling issues. We need to measure the impact of every decision and tool, then form decisions based on the results.

Observability serves a crucial role in measuring success. In the world of microservices, there's no way to measure the success of scaling without such tooling. Observability tools also serve as a benchmark to pinpoint scalability bottlenecks, as we will cover soon enough.

VERTICALLY INTEGRATED TEAMS

Over the years, developers tended to silo themselves based on expertise, and as a result, we formed teams to suit these processes. This is problematic. An engineer making a decision that might affect resource consumption or might impact such a tradeoff needs to be educated about the production environment.

When building a small system, we can afford to ignore such issues. Although as scale grows, we need to have a heterogeneous team that can advise on such matters. By assembling a full-stack team that is feature-driven and small, the team can handle all the different tasks required. However, this isn't a balanced team. Typically, a DevOps engineer will work with multiple teams simply because there are far more developers than DevOps.

This is logistically challenging, but the division of work makes more sense in this way. As a particular microservice fails, responsibilities are clear, and the team can respond swiftly.

FAIL-FAST

One of the biggest pitfalls to scalability is the fail-safe approach. Code might fail subtly and run in non-optimal form. A good example is code that tries to read a response from a website. In a case of failure, we might return cached data to facilitate a fail-safe strategy. However, since the delay happens, we still wait for the response. It seems like everything is working correctly with the cache, but the performance is still at the timeout boundaries.

This delays the processing. With asynchronous code, this is hard to notice and doesn't put an immediate toll on the system. Thus, such issues can go unnoticed. A request might succeed in the testing and staging environment, but it might always fall back to the fail-safe process in production.

Failing fast includes several advantages for these scenarios:

- It makes bugs easier to spot in the testing phase. Failure is relatively easy to test as opposed to durability.
- A failure will trigger fallback behavior faster and prevent a cascading effect.
- Problems are easier to fix as they are usually in the same isolated area as the failure.

API GATEWAY AND CACHING

Internal APIs can leverage an API gateway to provide smart load balancing, caching, and rate limiting. Typically, caching is the most universal performance tip one can give. But when it comes to scale, failing fast might be even more important. In typical cases of heavy load, the division of users is stark. By limiting the heaviest users, we can dramatically shift the load on the system. Distributed caching is one of the hardest problems in programming. Implementing a caching policy over microservices is impractical; we need to cache an individual service and use the API gateway to alleviate some of the overhead.

Level 2 caching is used to store database data in RAM and avoid DB access. This is often a major performance benefit that tips the scales, but sometimes it doesn't have an impact at all. Stack Overflow recently discovered that database caching had no impact on their architecture, and this was because higher-level caches filled in the gaps and grabbed all the cache hits at the web layer. By the time a call reached the database layer, it was clear this data wasn't in cache. Thus, they always missed the cache, and it had no impact. Only overhead.

This is where caching in the API gateway layer becomes immensely helpful. This is a system we can manage centrally and control, unlike the caching in an individual service that might get polluted.

OBSERVABILITY

What we can't see, we can't fix or improve. Without a proper observability stack, we are blind to scaling problems and to the appropriate fixes. When discussing observability, we often make the mistake of focusing on tools. Observability isn't about tools — **it's about questions and answers.**

When developing an observability stack, we need to understand the types of questions we will have for it and then provide two means to answer each question. It is important to have two means. Observability is often unreliable and misleading, so we need a way to verify its results. However, if we have more than two ways, it might mean we over-observe a system, which can have a serious impact on costs.

A typical exercise to verify an observability stack is to hypothesize common problems and then find two ways to solve them. For example, a performance problem in microservice X:

- Inspect the logs of the microservice for errors or latency — this might require adding a specific log for coverage.
- Inspect Prometheus metrics for the service.

Tracking a scalability issue within a microservices deployment is much easier when working with traces. They provide a context and a scale. When an edge service runs into an N+1 query bug, traces show that almost immediately when they're properly integrated throughout.

SEGREGATION

One of the most important scalability approaches is the separation of high-volume data. Modern business tools save tremendous amounts of meta-data for every operation. Most of this data isn't applicable for the day-to-day operations of the application. It is meta-data meant for business intelligence, monitoring, and accountability. We can stream this data to remove the immediate need to process it. We can store such data in a separate time-series database to alleviate the scaling challenges from the current database.

Conclusion

Scaling in the age of serverless and microservices is a very different process than it was a mere decade ago. Controlling costs has become far harder, especially with observability costs which in the case of logs often exceed 30 percent of the total cloud bill. The good news is that we have many new tools at our disposal — including API gateways, observability, and much more.

By leveraging these tools with a fail-fast strategy and tight observability, we can iteratively scale the deployment. This is key, as scaling is a process, not a single action. Tools can only go so far and often we can overuse them. In order to grow, we need to review and even eliminate unnecessary optimizations if they are not applicable. 🧩



Shai Almog, CEO at Codename One

[@sa74997](#) on DZone | [@shai-almog-81a42](#) on LinkedIn | [@debugagent](#) on YouTube and Twitter | [debugagent.com](#)

Shai is the author of *Practical Debugging at Scale: Cloud Native Debugging in Kubernetes and Production* (Apress). He is an entrepreneur, author, blogger, open-source hacker, speaker, Java rockstar, developer advocate, and more. As an ex-Sun/Oracle dev with 30+ years of experience, Shai has built JVMs, tools, mobile, startups/enterprise back ends, UIs, frameworks, observability tools, and more.



Application Architecture Design Principles

A Coordinated, Cross-Cutting Approach

By Ray Elenteney, Solution Architect at SOLTECH, Inc.

Designing an application architecture is never complete. Regularly, all decisions and components need to be reviewed, validated, and possibly updated. Stakeholders require that complex applications be delivered more quickly. It's a challenge for even the most senior technologists. A strategy is required, and it needs to be nimble. Strategy combines processes, which aid in keeping a team focused, and principles and patterns, which provide best practices for implementation. Regardless, it's a daunting task requiring organizational commitment.

Development, Design, and Architectural Processes

Applications developed without any process is chaos. A team that invents their own process and sticks to it is much better off than a team using no process. At the same time, holding a project hostage to a process can be just as detrimental. Best practices and patterns are developed over multiple years of teams looking for better ways to produce quality software in a timely manner. Processes are the codification of the best practices and patterns. By codifying best practices and patterns into processes, the processes can be scaled out to more organizations and teams.

For example, when an organization selects a development process, a senior leader may ascribe to a test-first development pattern. It becomes much easier for an organization to adopt a pattern by finding a process that outlines how the pattern is organizationally implemented. In the case of the test-first development pattern, test-driven development (TDD) may be selected as the development process.

Another technical leader in the same organization may choose to lead their team using domain-driven design (DDD), a pattern by which software design is communicated across technical teams as well as other stakeholders. Can these two design philosophies coexist? Yes. They can. Here, TDD defines *how* software is constructed while DDD defines the concepts that *describe* the software.

Software architecture works to remain neutral to specific development and design processes, and it is the specification on how an abstract pattern is implemented. The term, "abstract pattern," is used as most software architecture patterns can be applied across any development process and across any tech stack. For example, many architectures employ the use of inversion of control (or dependency injection). How Java, JavaScript, C#, etc. implement inversion of control is specific to the tech stack, but it accomplishes the same goal.

AVOIDING DOGMATIC ADHERENCE

Regardless of development, design, or architectural process, it's key that strict adherence to a given process does not become the end goal. Unfortunately, this happens more often than it should. Remember that the intent of a process is to codify best practices in a way that allows teams to scale using the same goals and objectives.

To that end, when implementing processes, here are some points to consider:

- There's no one size fits all.
- Allow culture to mold the process.
- Maturity takes time.
- Keep focused on what you're really doing — building quality software in a timely manner.

Cross-Cutting Concerns

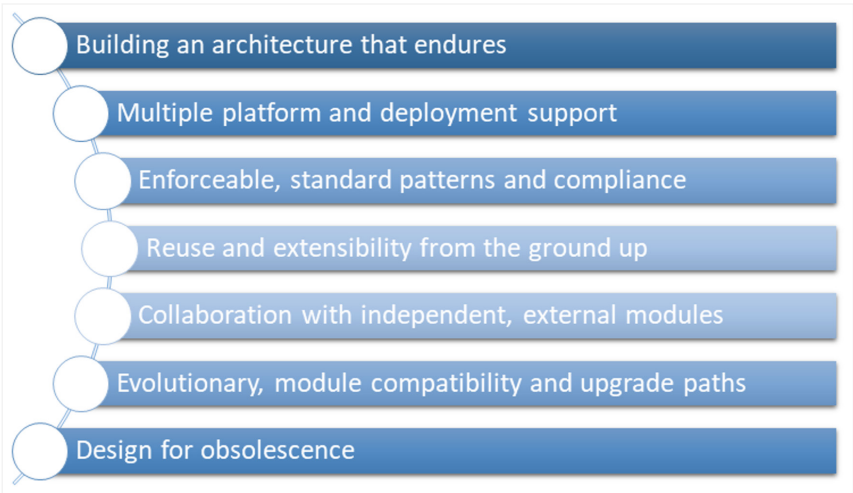
Software architecture can be designed, articulated, and implemented in several ways. Regardless of approach, most software architecture plans address two key points: simplicity and evolution. **Simplicity** is a relative term in that an architectural approach needs to be easily understood within the context of the business domain. Team members should look at an

architectural plan and say, "Of course, that's the obvious design." It may have taken several months to develop the plan, but a team responding in this manner is a sign that the plan is on the right track. **Evolution** is very important and can be the trickiest aspect of an architectural plan. It may sound difficult, but an architectural plan should be able to last ten-plus years. That may be challenging to comprehend, but with the right design principles and patterns in place, it's not as challenging as one might think.

At its core, good software architecture does its best to not paint itself into a corner. Figure 1 below contains no new revelations. However, each point is critical to a lasting software architecture:

- **Building architecture that endures.** This is the end goal. It entails using patterns that support the remaining points.
- **Multiple platform and deployment support.** The key here is that what exists today will very likely look different five years from now. An application needs to be readily able to adapt to changes in platform and deployment models, wherever the future takes it.
- **Enforceable, standard patterns and compliance.** Not that there's nothing new, but the software industry has decades of patterns to adopt and compliance initiatives to adhere to. Changes in both are gradual, so keeping an eye on the horizon is important.
- **Reuse and extensibility from the ground up.** Implementation patterns for reuse and extensibility will vary, but these points have been building blocks for many years.
- **Collaboration with independent, external modules.** The era of microservices helps enforce this principle. Watch for integrations that get convoluted. That is a red flag to the architecture.
- **Evolutionary, module compatibility and upgrade paths.** Everything in a software's architecture will evolve. Consider how compatibility and upgrades are managed.
- **Design for obsolescence.** Understand that many components within a software's architecture will eventually need to be totally replaced. At the beginning of each project or milestone, ask the question, "How much code are we getting rid of this release?" The effect of regular code pruning is no different than the effect of pruning plants.

Figure 1: Key architectural principles



Developing microservices is a combination of following these key architectural principles along with segmenting components into areas of responsibility. Microservices provide a unit of business functionality. Alone, they provide little value to a business. It's in the assembly of and integration with other microservices that business value is realized.

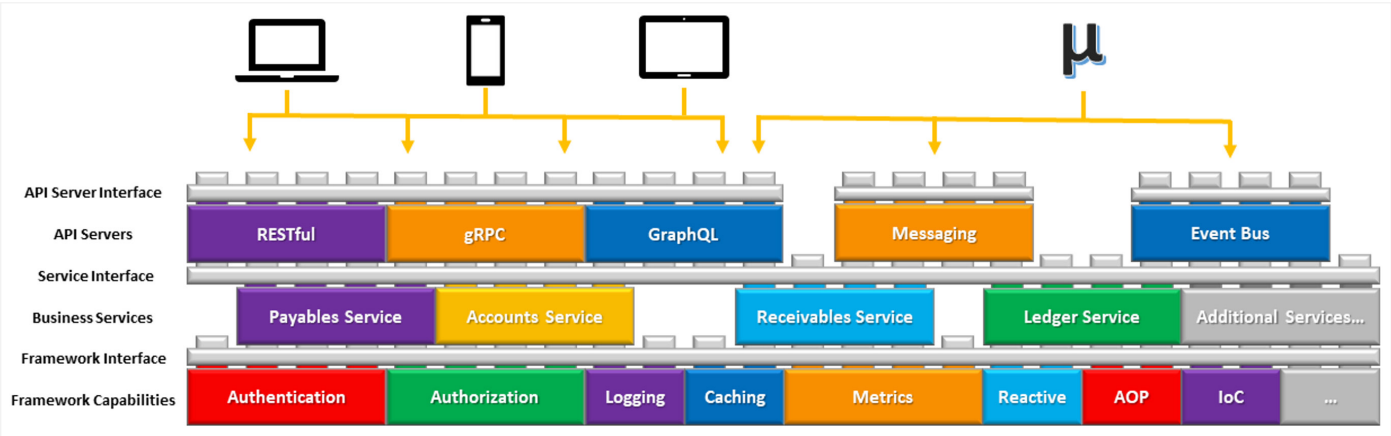
Good microservices assembly and integration implementations follow a multi-layered approach.

HORIZONTAL AND VERTICAL SLICES

Simply stated, slicing an application is about keeping things where they belong. In addition to adhering to relevant design patterns in a codebase, slicing an application applies the same patterns at the application level. Consider an application architecture as depicted by a Lego® brick structure in the figure below:

SEE FIGURE 2 ON NEXT PAGE

Figure 2: Microservices architecture



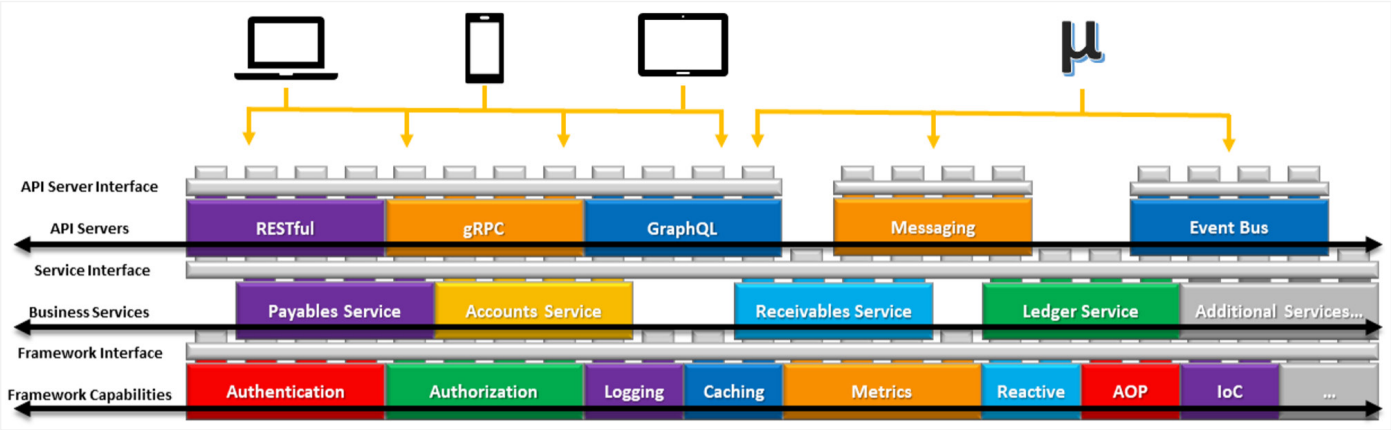
Each section of bricks is separated by that thin Lego® brick, indicating a strict separation of responsibility between each layer. Layers interact only through provided contracts/interfaces. Figure 2 depicts three layers with each having a distinct purpose. Whether it be integration with devices such as a laptop or tablet, or microservices integrating with other microservices, the point at which service requests are received remains logically the same. Here, there are several entry points ranging from web services and messaging services to an event bus.

HORIZONTAL SLICES

Horizontal slices of an application architecture are layers where, starting from the bottom, each layer provides services to the next layer. Typically, each layer of the stack refines the scope of underlying services to meet business use case logic. There can be no assumptions by services in lower layers on how above services interact with them. As mentioned, this is done with well-defined contracts.

In addition, services within a layer interact with one another through that layer's contracts. Maintaining strict adherence to contracts allows components at each layer to be replaced with new or enhanced versions with no disruption in interoperability.

Figure 3: Horizontal slices



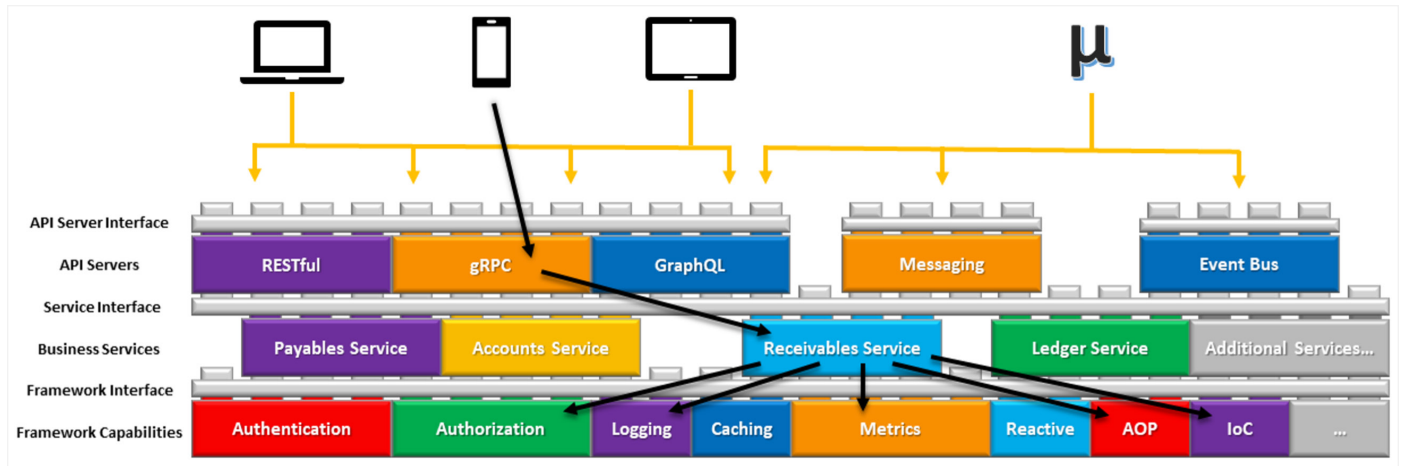
VERTICAL SLICES

Vertical slices are where everything comes together. A vertical slice is what delivers an application business objective. A vertical slice starts with an entry point that drills through the entire architecture. As depicted in Figure 4, business services can be exposed in multiple ways. Entry points are commonly exposed through some type of network protocol.

However, there are cases where a network protocol doesn't suffice. In these cases, a business service may offer a native library supporting direct integration. Regardless of the use case, strict adherence to contracts must be maintained.

SEE FIGURE 4 ON NEXT PAGE

Figure 4: Vertical slices



Obvious, Yet Challenging

Microservices have become a predominant pattern by which large applications are assembled. Each microservice is concerned with a very specific set of functionalities. By their very nature, microservices dictate that well-defined contracts are in place, with which other microservices and systems can integrate. Microservices that are designed and implemented for cloud-native deployments can leverage cloud-native infrastructure to support several of the patterns discussed.

The patterns and diagrams presented here will look obvious to most. As mentioned, good architecture is "obvious." The challenge is adhering to it. Often, the biggest enemy to adherence is time. The pressure to meet delivery deadlines is real and where cracks in the contracts appear. Given the multiple factors in play, there are times when compromises need to be made. Make a note, create a ticket, add a comment, and leave a trail so that the compromise gets addressed as quickly as possible.

Well-designed application architecture married with good processes supports longevity, which from a business perspective provides an excellent return on investment. Greenfield opportunities are fewer than evolving existing applications. Regardless, bringing this all to bear can look intimidating. The key is to start somewhere. As a team, develop a plan and "make it so"! 🎯



Ray Elenteny, Solution Architect at SOLTECH, Inc.

@rbetae on DZone | @ray-elenteny on LinkedIn

With over 35 years of experience in the IT industry, Ray thoroughly enjoys sharing his experience by helping organizations deliver high-quality applications that drive business value. Ray has a passion for software engineering. Over the past 10 years or so, Ray has taken a keen interest in the cultural and technical dynamics of efficiently delivering applications.



Demystifying Multi-Cloud Integration

Comprehensive Strategies and Patterns for Integrating Cloud Systems

By Boris Zaikin, Senior Software & Cloud Architect at Nordcloud GmbH, an IBM company

Multi-cloud integration strategies may sound like buzzwords and marketing slang, but in this article, I will demystify them. I will also dive deeply into on-premises and legacy systems and how we can integrate with them. Before we jump into the topic, I would like to define what integration means in a cloud context.

Cloud integration is a process that allows organizations' applications, infrastructure, data, and components to properly work together within one or several cloud providers. It also includes connecting on-premises data centers to the cloud if migration can be done across the organization.

Cloud Integrations

An important part of cloud integration is understanding the strategies. Many medium- and enterprise-level companies choose multi-cloud and hybrid cloud approaches. Why is successful integration important for companies? Most companies building solutions have to exchange data with on-premises or out-of-support solutions.

Properly designed integration solutions will save a lot of time and money. We can see it in the example of a bank multi-cloud application at the end of the article.

HYBRID VS. MULTI-CLOUD

Below is a comparison table describing both strategies' pros and cons. Before we jump in, keep the differences between public and private clouds in mind. Remember that **public clouds** provide computing power, SaaS, and PaaS services for organizations that don't have (or where it is difficult to have) their data centers. A **private cloud** (on-premises) is an infrastructure the company maintains internally.

Table 1

HYBRID VS. MULTI-CLOUD PROS AND CONS		
	Hybrid Cloud	Multi-Cloud
Description	Hybrid clouds combine private clouds/on-prem data centers with a public cloud, an approach that companies usually take. For example, banks have secure on-prem environments that they won't move to the cloud. Meanwhile, they have other, less secure solutions that can be easily moved to a public cloud and have fewer connections to on-premises.	Multi-cloud combines several public clouds without using a private cloud. Usually, companies choose a multi-cloud strategy to avoid vendor lock-in.
Pros	<ul style="list-style-type: none">Flexibility to connect infrastructure that can't be moved to the public cloud.Increased security thanks to the on-prem component.Flexibility between using a legacy system and modern public cloud services.	<ul style="list-style-type: none">Flexible and scalable environments.You can choose the services in each cloud that work best for your company.Freedom to implement the solution across several clouds.
Cons	<ul style="list-style-type: none">It can be difficult to maintain legacy, on-prem environments.Additional cost for companies because they need to maintain their hardware.	<ul style="list-style-type: none">The cost of maintaining different services on several cloud providers can be prohibitive.Complexity in managing and separating different services.Securing network communication between clouds can be difficult.

Cloud Integration Patterns and Best Practices

Applying a good integration strategy also requires knowing some integration best practices and patterns.

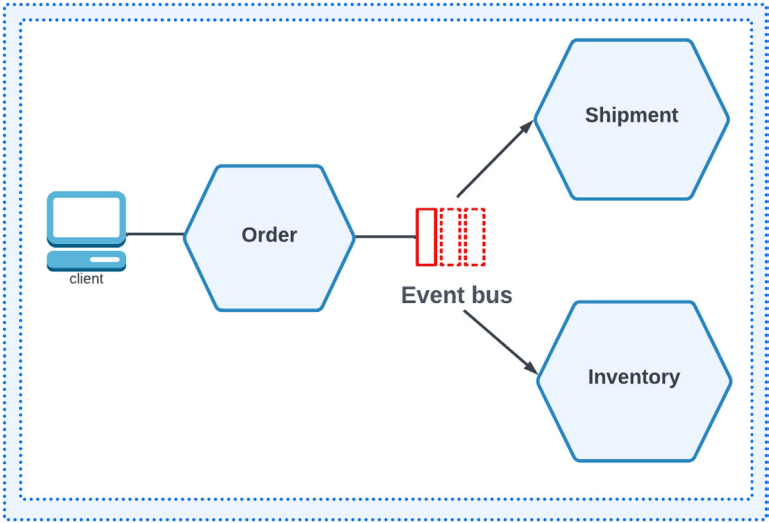
CLOUD INTEGRATION PATTERNS

Understanding the main set of integration patterns is key to using existing integration solutions or designing a new one from scratch. Also, having the knowledge of these patterns provides a massive benefit during the integration of cloud applications and enterprise, on-premises infrastructure.

ASYNCHRONOUS MESSAGING

Asynchronous messaging allows components and services to process data without waiting for each other. It also allows components to be decoupled from each other.

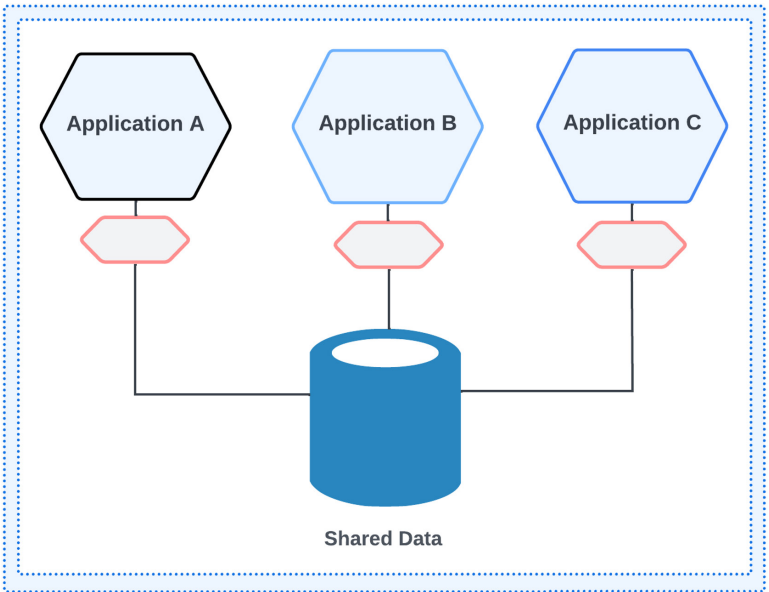
Figure 1



SHARED DATABASES

This pattern uses a **shared database** to communicate and exchange data between enterprise applications and services. As part of a shared database and communication bus, we can also use an enterprise service bus that can save and exchange data between several components.

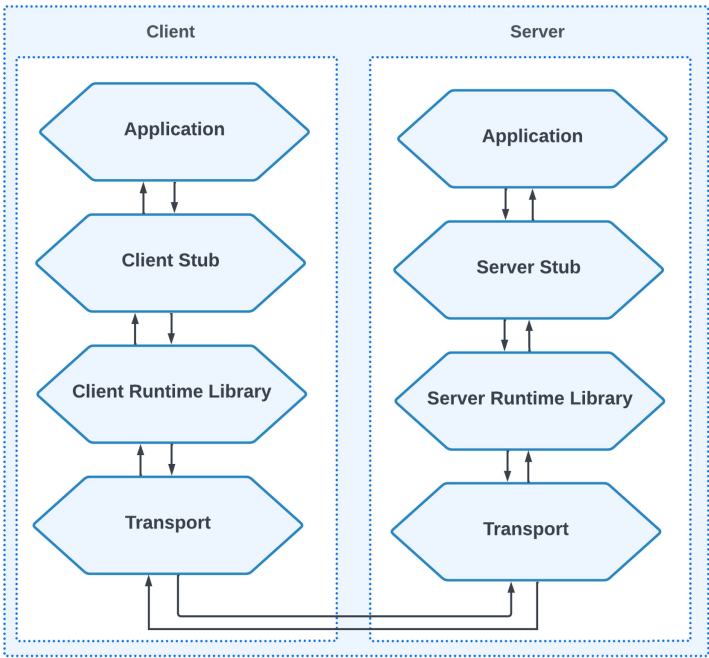
Figure 2



REMOTE PROCEDURE CALL

Remote procedure call (RPC) is an abstraction layer or protocol that allows one network component to communicate with another without knowing the whole network's complete functionality.

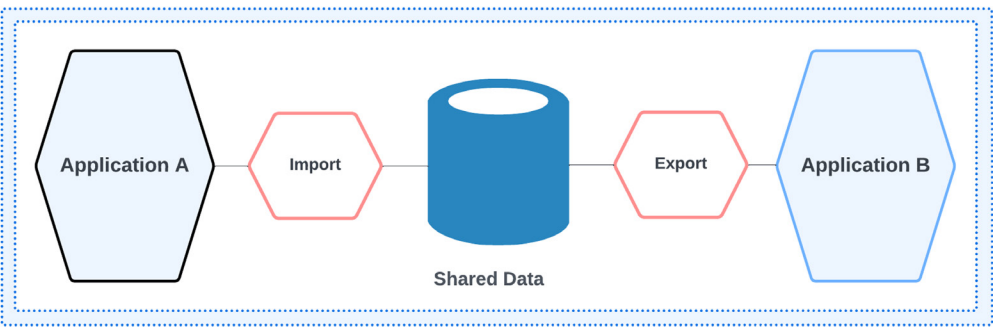
Figure 3



FILE TRANSFER

The **file transfer** pattern provides an interface to share files between cloud or application components. For example, file transfer is useful if an application produces CSV or XML reports — the integration service should adapt this file for other applications.

Figure 4



RECOMMENDED PRACTICES FOR CLOUD INTEGRATION

Here are three of the most important best practices for cloud integration:

1. **Use native SaaS tools that cloud providers offer.** This approach always provides the best integration options between applications and components. There are even "no-code" tools for non-technical people. We will get into native Azure, AWS, and Google Cloud Services in the next section.
2. **Use an Integration Platform as a Service (iPaaS).** Some services and components provide integration capabilities and are hosted as cloud services. For example, [triggermesh](#) and [cenit.io](#) are open-source integration platforms that allow building event-driven applications in Kubernetes, orchestrating data flow, and providing API management capabilities in cloud providers and on-premises.
3. **Use a Function Platform as Service (FPaaS).** These platforms provide huge customization levels of integration options, from which some organizations can benefit. This approach is intended for cloud solution architects and requires a knowledge of cloud architecture patterns and function-oriented software development skills. FPaaS tools include [AWS Lambda](#), [Azure Functions](#), [Google Cloud Functions](#), and [Apache OpenWhisk](#).

Common Integration Services

Knowing the general cloud integration best practices and patterns is crucial. However, knowing what exactly each cloud provider offers is also important. In this section, we will briefly touch upon common cloud integration services from providers such as AWS, Azure, and Google Cloud. Keep in mind: This section contains — but is not limited to — some of the most ubiquitous open-source integration services available. To learn more about the list below, common benefits, and drawbacks associated with each, check out [this platform breakdown](#) for more information.

AWS

AWS has several integration services that provide powerful features alongside simplicity. This list includes [SNS](#) (Simple Notification Service), [SQS](#) (Simple Queue Service), [SWF](#) (Simple Workflow Service), and [AWS step functions](#). To learn more, visit the [AWS Application Integration services](#) page.

GOOGLE CLOUD

Google Cloud has a vast integration ecosystem, also commonly referred to as [Integration Platform as a Service](#) (iPaaS). This provides a set of tools and services to manage and connect applications. The Google Cloud iPaaS contains the following core services: [Integration designer](#), [triggers](#), and [tasks](#). Learn more about each Google Cloud integration service [here](#).

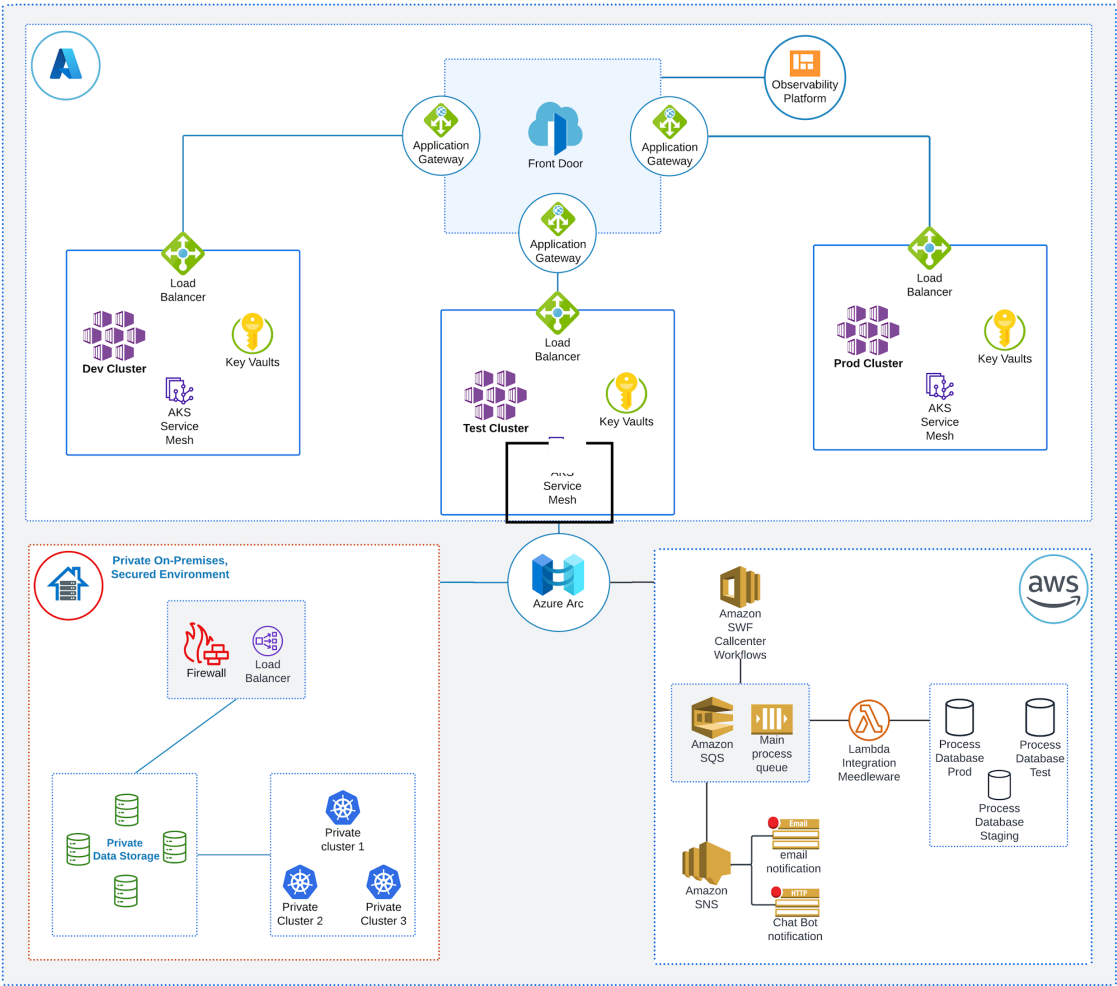
AZURE

Azure offers an Azure integration service set (also commonly referred to as Azure Integration Platform as a Service). This contains a variety of services set up to provide strong integration between applications. Some of the most powerful integration services Azure offers include [API Management](#), [Logic Apps](#), [Service Bus](#), [Event Grid](#), and [Azure Arc](#). If you are interested in reading more on the various Azure integration services, check out [this page](#) to learn more.

A Bank Multi-Cloud Application

As mentioned, banking applications require a massive security layer. Also, many banks contain their own highly secure data centers, and migrating all secured data to the cloud may not even be an option.

Figure 5: A banking multi-cloud integration application example



In this example, we selected Azure as the cloud for the main application. The application is based on a microservices architecture and is deployed to several Kubernetes clusters. Azure stores secrets, a configuration in a [Cosmos DB](#), and some files in [Blob Storage](#). Azure also provides an observability platform with a service mesh. All secured data is stored on the on-premises data center, and the AWS Cloud part contains a workflow for the call center.

Conclusion

In this article, we've reviewed top cloud integration patterns and services that start the integration process from scratch or that consider an existing environment. Designing integrations of software solutions in the cloud requires knowledge of best practices and patterns. Furthermore, it requires a deep understanding of the toolsets, services, and components each cloud and framework offer. For example, alongside Azure Arc, AWS offers services like [Systems Manager](#).

Before I start an integration project, I'm using the following algorithm:

- Keep in mind the [KISS principle](#)
- Have a look at existing integration patterns
- Check on what integration components and services other clouds provide

Therefore, multi-cloud integration means to make solutions and components of one cloud provider work with others using existing integration cloud components and patterns. 🧩



Boris Zaikin, Software & Cloud Architect at Nordcloud GmbH

[@borisza](#) on DZone | [@boris-zaikin](#) on LinkedIn | [boriszaikin.com](#)

I'm a certified senior 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.

Diving Deeper Into Software Integration



REFCARDS

API Integration Patterns

In this [Refcard](#), readers will explore the fundamental patterns for authentication, polling, querying, and more, helping you assess your integration needs and approach the design, build, and maintenance of your API integrations in the most effective ways for your business case.

Microservices and Workflow Engines: Getting Started With Agile Business Process Automation

This [Refcard](#) introduces a way to address business-process-related challenges using a microservices architecture and a workflow engine for orchestration. You'll learn key techniques in areas such as microservices design, communication, and state management, as well as first steps to take when getting started with business process automation.

Getting Started With OpenTelemetry: Observability and Monitoring for Modern Applications

As cloud migrations increase in number, OpenTelemetry aims to reduce data collection time through automation. It is an open-source collection of tools, APIs, SDKs, and specifications that standardizes how to model and collect telemetry data. This [Refcard](#) discusses core OpenTelemetry architecture components, key features, and how to set up for tracing and exporting telemetry data.

TREND REPORTS

Microservices and Containerization: The Intersection of Cloud Architectures and Design Principles

This [2022 Trend Report](#) dives into various cloud architecture practices, microservices orchestration techniques, security, and advice on design principles. The goal of this report is to explore the current state of microservices and containerized environments to help developers face the challenges of complex architectural patterns.

Enterprise Application Integration

This [2022 Trend Report](#) offers perspectives on cloud-based integrations vs. on-prem models, how organizational culture impacts successful API adoption, the different use cases for GraphQL vs. REST, and why the 2020s should now be considered the "Events decade." The goal of this report is to provide diverse perspectives on integration to help make the best choices for your organization.

MULTIMEDIA



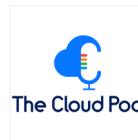
Coding Over Cocktails

The [Coding Over Cocktails](#) podcast gives you the tl;dr on application integration, digital transformation, data management, and more. Presenting easy-to-digest 30-minute episodes, host David Brown interviews industry experts to bring you insights into today's industry.



API Intersection: The Podcast

[Jason Harmon's](#) interviews with experienced API experts showcase need-to-know advice, information, and real-world experience. This podcast is perfect for those who need a little help solving a challenge — or for those who are simply interested in learning more about APIs.



The Cloud Pod

If you are looking for advice on your cloud-based system, check out [The Cloud Pod](#). This all tech, no marketing podcast will inform you about the latest cloud news and trends as cloud providers continue to evolve and add changes to their APIs.



APIs You Won't Hate

Featuring a podcast (with only a little nonsense), articles, books, and job postings, the [APIs You Won't Hate](#) website is a great go-to place for your API needs and wants. With an active community and plenty of resources, you'll find collaborative assistance and shared experiences to guide you on your coding journey.



OpenObservability Talks

Catch this [monthly podcast](#) on your favorite platform or stream it live, and tune in as hosts Jonah Kowall and Dotan Horovits discuss observability for DevOps. With a focus on open-source technologies, episode topics range from microservices based-systems to managing observability costs.



Solutions Directory

This directory contains tools for API, cloud, iPaaS, microservices, and more to help manage your integration processes. It provides pricing 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 2023 SOFTWARE INTEGRATION SOLUTIONS DIRECTORY

2023
PARTNER

Company	Product	Purpose	Availability	Website
Elastic Path	Elastic Path Commerce Cloud	SaaS microservices commerce	Trial period	elasticpath.com/products/elastic-path-commerce-cloud
Company	Product	Purpose	Availability	Website
Adeptia	Connect	Enterprise integration	By request	adeptia.com/products/adeptia-connect-enterprise-integration
AdroitLogic	UltraESB-X	Enterprise service bus	By request	adroitlogic.com/products/ultraesb
Aiven	Aiven for Apache Kafka	Fully managed distributed data streaming platform	Trial period	aiven.io/kafka
	Karapace	Rest API and schema registry for Apache Kafka	Open source	karapace.io
Akana	API Management Platform	API lifecycle management	Trial period	akana.com/products/api-platform
Amazon Web Services	Amazon MQ	Fully managed service for open-source message brokers	Trial period	aws.amazon.com/amazon-mq
	Amazon SNS	Fully managed pub/sub service		aws.amazon.com/sns
	Amazon SQS	Fully managed message queuing		aws.amazon.com/sqs
	Amazon SWF	Task coordination and workflow service		aws.amazon.com/swf
	AWS AppSync	App development with serverless GraphQL and pub/sub APIs		aws.amazon.com/appsync
	Step Functions	Visual workflows for distributed applications		aws.amazon.com/step-functions
Apache Software Foundation	ActiveMQ	Java-based message broker	Open source	activemq.apache.org
	Kafka	Distributed event streaming platform		kafka.apache.org
Apollo	Apollo	Enterprise integration	Free tier	apollo.io/product/product-overview
Aspen Mesh	Aspen Mesh	App intelligence platform	By request	aspenmesh.io
AxonIQ	Axon Framework	Event-driven microservices and DDD framework	Open source	developer.axoniq.io/axon-framework/overview
Axway	Amplify API Management Platform	API lifecycle management	By request	axway.com/en/products/amplify-api-management-platform
Ballerina	Ballerina	Programming language for the cloud	Open source	ballerina.io
Beamable	Beamable	Full-stack LiveOps platform for live games	Free tier	beamable.com
Boomi	AtomSphere Platform	iPaaS	Trial period	boomi.com/platform

DZONE'S 2023 SOFTWARE INTEGRATION SOLUTIONS DIRECTORY

Company	Product	Purpose	Availability	Website
Broadcom	Layer7 API Management	Continuous API management	By request	broadcom.com/products/software/api-management
Camunda	Camunda Platform	Microservices orchestration	Free tier	camunda.com/platform
Celigo	Integration Platform	iPaaS	Trial period	celigo.com/platform
Cleo	Cleo Integration Cloud	Ecosystem integration	By request	cleo.com
Cloud Foundry	Open Service Broker API	Back services for workloads running on cloud-native platforms	Open source	openservicebrokerapi.org
Cloud Native Computing Foundation	NATS	Connective technology for distributed systems	Open source	nats.io
DreamFactory	DreamFactory	API management	By request	dreamfactory.com
Dynatrace	Dynatrace	Software intelligence platform	Trial period	dynatrace.com/platform
Epsagon	Epsagon	Microservices observability	Free tier	epsagon.com
F5	NGINX App Protect	Modern WAF and denial of service for app and API protection	Trial period	nginx.com/products/nginx-app-protect
	NGINX Management Suite	Security and API management services		nginx.com/products/nginx-management-suite
	NGINX Plus	Reverse proxy, load balancer, API gateway, and more		nginx.com/products/nginx
	NGINX Service Mesh	Service mesh	Free	nginx.com/products/nginx-service-mesh
Fiorano Software	Hybrid Integration Platform	Workflow automation and integration	Trial period	fiorano.com/products/fiorano_hybrid_integration
Flowgear	Flowgear	iPaaS platform	Trial period	flowgear.net
Fujitsu Ltd	PRIMEFLEX Integrated Systems	Hybrid data architecture deployment and maintenance	By request	fujitsu.com/global/products/computing/integrated-systems
Google Cloud	Anthos Service Mesh	Istio-based, fully managed service mesh	Trial period	cloud.google.com/anthos/service-mesh
	API Gateway	API development, deployment, and management		cloud.google.com/api-gateway
	Apigee API Management	Native API management		cloud.google.com/apigee
	Apigee Integration	API-first integration to connect data and apps		cloud.google.com/apigee/integration
	Application Integration	iPaaS		cloud.google.com/application-integration/docs
	Cloud APIs	Workflow automation		cloud.google.com/apis
Hasura	Hasura	GraphQL and REST API creation from your database(s)	Open source	hasura.io
IBM	IBM API Connect	API lifecycle management	Trial period	ibm.com/cloud/api-connect
	IBM App Connect	App integration and API development		ibm.com/cloud/app-connect
IFTTT	IFTTT	Integration management	Free tier	ifttt.com
InfluxData	InfluxDB Cloud	Time series data platform	Free tier	influxdata.com/cloud
Instana	Instana	Automated real-time observability	Trial period	instana.com
Integrate.io	Integrate.io	No-code data pipeline platform	Trial period	integrate.io

DZONE'S 2023 SOFTWARE INTEGRATION SOLUTIONS DIRECTORY

Company	Product	Purpose	Availability	Website
Intersystems	Ensemble	Integration engine	By request	intersystems.com/ensemble
Istio	Istio	Service mesh	Open source	istio.io
Jaeger	Jaeger	End-to-end distributed tracing	Open source	jaegertracing.io
JHipster	JHipster	Web apps and microservices development platform	Open source	jhipster.tech
Jitterbit	Harmony	Low-code integration platform	By request	jitterbit.com/harmony
Kong	Insomnia	Collaborative API design platform	Free tier	insomnia.rest
	Kong Enterprise	API gateway, microservices management	By request	konghq.com/products/api-gateway-platform
Lightbend	Akka Platform	Reactive microservices frameworks	Free tier	lightbend.com/akka
	Kalix	High-performance microservices and APIs	Trial period	kalix.io
Lightstep	Lightstep	Cloud-native reliability platform	Free tier	lightstep.com
Linkerd	Linkerd	Service mesh	Open source	linkerd.io
Magic Software	Magic xpi	Integration platform	By request	magicsoftware.com/integration-platform/xpi
Microsoft Azure	API Management	Hybrid, multi-cloud management platform for APIs	Trial period	azure.microsoft.com/en-us/services/api-management
	Arc	Hybrid and multi-cloud management		azure.microsoft.com/en-us/products/azure-arc
	Cloud Services	Scalable cloud application and API management		azure.microsoft.com/en-us/products/cloud-services
	Logic Apps	Serverless workflow integration tool		azure.microsoft.com/en-us/products/logic-apps
	Service Bus	Cloud messaging as a service and hybrid integration		azure.microsoft.com/en-us/products/service-bus
	Service Fabric	Microservices development platform		azure.microsoft.com/en-us/services/service-fabric
Mulesoft	Anypoint Platform	Hybrid integration platform	Trial period	mulesoft.com/platform/enterprise-integration
Nutanix	Nutanix Cloud Infrastructure	Hyperconverged infrastructure, app, and data management	Trial period	nutanix.com/products/nutanix-cloud-infrastructure
Okta	Customer Identity Cloud	Consumer and SaaS app security	Trial period	okta.com/customer-identity
OpenText	Artix	Enterprise service bus	Trial period	microfocus.com/en-us/products/artix/overview
	Hybrid Integration Platform	Business integration environment	By request	opentext.com/products/hybrid-integration-platform
OpenLegacy	OpenLegacy Hub	Modernization platform	By request	openlegacy.com/ol-hub
OpenTelemetry	OpenTelemetry	Observability framework	Open source	opentelemetry.io
OpsLevel	OpsLevel	Microservices management	Trial period	opslevel.com
Oracle	Oracle Cloud Infrastructure	iPaaS, cloud infrastructure platform	Free tier	oracle.com/cloud
Pact	Pact	Web apps, API, and microservices integration testing	Open source	pact.io
Palo Alto Networks	Prisma Cloud	Cloud-native application protection platform	By request	paloaltonetworks.com/prisma/cloud

DZONE'S 2023 SOFTWARE INTEGRATION SOLUTIONS DIRECTORY

Company	Product	Purpose	Availability	Website
Particular Software	Particular Service Platform	.NET service	Free tier	particular.net/service-platform
Peregrine Connect	Management Suite	End-to-end integration	Trial period	peregrineconnect.com/products/management-suite
	Neuron ESB	App, API, and workflow integration		peregrineconnect.com/products/neuron-esb
Postman	Postman	API platform	Free tier	postman.com
Red Hat	3scale API Management	Self-managed API management	Trial period	redhat.com/en/technologies/jboss-middleware/3scale
	OpenShift API Management	Hosted and managed API management	Sandbox	redhat.com/en/technologies/cloud-computing/openshift/openshift-api-management
	Red Hat Fuse	Distributed, cloud-native integration platform	Open source	redhat.com/en/technologies/jboss-middleware/fuse
	Red Hat Integration	Integration and messaging technologies suite	By request	redhat.com/en/products/integration
Redis Labs	Redis Enterprise Cloud	Caching, front-end database, and real-time data platform	Trial period	redis.com/redis-enterprise-cloud/overview
	Redis Enterprise Software	Self-managed data platform		redis.com/redis-enterprise-software/overview
RoboMQ	Connect iPaaS	No-code for self-service API and data integration	Trial period	robomq.io/connect
	Hybrid Integration Platform	API and data integration	By request	robomq.io/hybrid-integration-platform
Runscope	Runscope	API monitoring	Trial period	runscope.com
Salt Security	Salt Security	API security	By request	salt.security
SAP	Business Technology Platform	Integration, data to value, and extensibility	By request	api.sap.com/products/SAPCloudPlatform/overview
SEEBURGER	Business Integration Suite	Hybrid integration platform	By request	seeburger.com/platform/business-integration-suite
Smartbear	AlertSite	Global, synthetic API monitoring	Trial period	smartbear.com/product/alertsite
	ReadyAPI	Low-code API testing platform		smartbear.com/product/ready-api
	SoapUI	API testing tool	Open source	soapui.org/tools/soapui
	SwaggerHub	API development platform	Trial period	swagger.io/tools/swaggerhub
Snaplogic	Intelligent Integration Platform	iPaaS	By request	snaplogic.com/products/intelligent-integration-platform
Software AG	webMethods	APIs, integration, and microservices management	By request	softwareag.com/en_corporate/platform/integration-apis.html
Solo.io	Gloo Gateway	API gateway and Kubernetes Ingress	Trial period	solo.io/products/gloo-gateway
	Gloo Mesh	Istio-based service mesh and control plane	Open source	solo.io/products/gloo-mesh
Talend	Data Fabric	Data management	Trial period	talend.com/products/data-fabric
Thriftly	Thriftly	API toolkit for Windows	Trial period	thriftly.io
TIBCO	Cloud™ API Management	API management	Trial period	tibco.com/products/api-management

DZONE'S 2023 SOFTWARE INTEGRATION SOLUTIONS DIRECTORY

Company	Product	Purpose	Availability	Website
Traefik Labs	Traefik Enterprise	Ingress, API management, and service mesh	Trial period	traefik.io/traefik-enterprise
	Traefik Mesh	Non-invasive service mesh	Open source	traefik.io/traefik-mesh
	Traefik Proxy	Cloud-native application proxy		traefik.io/traefik
Tyk	Tyk Cloud	API management	Trial period	tyk.io/api-lifecycle-management
	Tyk Gateway	API gateway	Open source	tyk.io/open-source
UiPath	UiPath Platform	Business automation	Trial period	uipath.com/product
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
VMWare	Spring Cloud Stream	Event-driven microservices framework	Open source	spring.io/projects/spring-cloud-stream
	Spring Cloud Task	Short-lived microservices framework		spring.io/projects/spring-cloud-task
	Tanzu Service Mesh	End-to-end connectivity and security for modern apps	By request	tanzu.vmware.com/service-mesh
Workato	Workato	Integration and workflow automation	By request	workato.com
WSO2	Choreo Internal Developer Platform	Full-lifecycle cloud-native application development platform	Free tier	wso2.com/choreo
	Enterprise Integrator	Hybrid integration platform	Open source	wso2.com/integration/micro-integrator
	WSO2 API Manager	API management, governance, and analysis	Trial period	wso2.com/api-manager
Zoho	Catalyst	Serverless development suite	Sandbox	catalyst.zoho.com



Software Design and Architecture

CLOUD ARCHITECTURE | CONTAINERS | INTEGRATION
MICROSERVICES | PERFORMANCE | SECURITY

Software design and architecture focus on the development decisions made to improve a system's overall structure and behavior in order to achieve essential qualities such as modifiability, availability, and security.

The Zones in this category are available to help developers stay up to date on the latest software design and architecture trends and techniques.

[VISIT THE ZONE](#)



TUTORIALS



INDUSTRY PRACTICES



CODE SNIPPETS



RESEARCH



600 Park Offices Drive, Suite 300
Research Triangle Park, NC 27709
888.678.0399 | 919.678.0300

At DZone, we foster a collaborative environment that empowers developers and tech professionals to share knowledge, build skills, and solve problems through content, code, and community. We thoughtfully — and with intention — challenge the status quo and value diverse perspectives so that, as one, we can inspire positive change through technology.

Copyright © 2023 DZone, Inc. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means of electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher.