# Benchmarking Web API Quality – Revisited

David Bermbach[1] and Erik Wittern[2]

[1]*TU Berlin & Einstein Center Digital Future, Mobile Cloud Computing Research Group, Berlin, Germany*
[2]*IBM, Hybrid Cloud Integration, Hamburg, Germany*
*E-mail: db@mcc.tu-berlin.de; erik.wittern@ibm.com*

Received 10 May 2020; Accepted 03 July 2020;
Publication 24 October 2020

## Abstract

Modern applications increasingly interact with web APIs – reusable components, deployed and operated outside the application, and accessed over the network. Their existence, arguably, spurs application innovations, making it easy to integrate data or functionalities. While previous work has analyzed the ecosystem of web APIs and their design, little is known about web API quality at runtime. This gap is critical, as qualities including availability, latency, or provider security preferences can severely impact applications and user experience.

In this paper, we revisit a 3-month, geo-distributed benchmark of popular web APIs, originally performed in 2015. We repeat this benchmark in 2018 and compare results from these two benchmarks regarding availability and latency. We furthermore introduce new results from assessing provider security preferences, collected both in 2015 and 2018, and results from our attempts to reach out to API providers with the results from our 2015 experiments. Our extensive experiments show that web API qualities vary 1.) based on the geo-distribution of clients, 2.) during our individual experiments, and 3.) between the two experiments. Our findings provide evidence to foster the discussion around web API quality, and can act as a basis for the creation of tools and approaches to mitigate quality issues.

**Keywords:** Web APIs, Benchmarking, Quality of Service.

*Journal of Web Engineering, Vol. 19_5–6,* 603–646.
doi: 10.13052/jwe1540-9589.19563

## 1 Introduction

Today, mobile, web, or even desktop applications regularly rely on third-party data or functionalities, which they consume through web Application Programming Interfaces (web APIs). Web APIs provide these applications with otherwise inaccessible resources such as access to global social networks (e.g., web APIs provided by Twitter, Facebook, or LinkedIn), advanced machine-learning capabilities (e.g., web APIs provided by IBM Watson or Google Cloud AI), or complex transaction processing (e.g., web APIs by Stripe or PayPal for payment processing or the Flight Booking API). Today, application developers can build on

- ubiquitous technologies, e.g, the Hypertext Transfer Protocol (HTTP) or Asynchronous JavaScript and XML (AJAX),
- architectural styles, e.g., the Representational State Transfer (REST) [23],
- de-facto standards for describing web APIs, e.g., the OpenAPI Specification[1],
- research results, e.g., from service-oriented computing, cloud computing, or mash-ups, and finally
- a myriad of web API client libraries in any established programming language,

so that integrating web APIs with an application no longer poses a technological challenge. Therefore, we now see thousands of public APIs as well as applications using them [73].

In consequence, though, application developers now heavily rely on third-party entities beyond their control sphere for core functionality of their applications. This can have impacts on applications' user experience. For example, erroneous integration of core application capabilities, e.g., a payment service, via APIs can impede end user experience [3]. High request latency can lead to slow application response times, which have been found to disrupt users' flow of thought or eventually cause loss of attention [50]. A long-term experiment performed by Google showed that increasing response times for search results artificially from 100ms to 400ms did measurably decrease the average amount of searches performed by users [12]. User experience and, hence, application reputation is thus directly affected by actions and non-actions of the API providers. As another example, APIs may be discontinued or changed without notice, thus disabling applications. Previous work found that mobile applications silently fail and, in cases,

[1] https://www.openapis.org/

even crash when confronted with mutated (e.g., adapted or faulty) web API responses [20]. In sum, web APIs often present themselves as black-boxes with volatile *qualities* – i.e., availability, latency, security, or usage limitations – to clients.

Based on this observation, we published a study on web API quality [8] in 2016, in which we described our findings from an experiment of benchmarking 15 endpoints[2] of diverse web APIs for three months from geo-distributed clients. Core results were that availability and latency highly depend on the geo-origin of requests as well as the protocol used (HTTP vs. HTTPS). For this paper, we extend, repeat, and correct our previous study[3]. Namely, we repeated all experiments in 2018, reanalyze old and new availability and latency measurements, analyze yet unpublished measurements on provider security preferences, and present the results of trying to contact the providers of all benchmarked APIs with the goal of (i) finding explanations for the observations from our 2016 paper and (ii) to assess availability and quality of customer service. We thus make the following contributions:

1. We extend our previous measurement method to also collect information on Transport Layer Security (TLS) cipher suite preferences of providers.
2. We report findings on how TLS cipher suite preferences of providers evolved over time in both our 2015 and 2018 measurements.
3. We report detailed findings from analyzing and comparing our 2015 and 2018 latency and availability measurements.
4. We discuss the results of reaching out to the providers of all benchmarked APIs about our original findings from 2015.

It should be noted that our work takes the perspective of an application developer, i.e., we have little insight into what happens behind the scenes but instead report things observable in practice: While API implementations may change or APIs may be discontinued, this is not a limitation of our results but rather a finding in itself.

This paper is structured as follows: we present background on web APIs, possible failures when consuming them, and web API qualities in Section 2. We present the implementation of our benchmarking client and the design of our experiments – both with regard to benchmarked API endpoints and experiment setup – in Section 3. In the following sections, we compare

[2]We denote an endpoint to be the combination of a resource, identified by a URL, and an HTTP *method* as proposed in [64].

[3]Where applicable, this paper reuses parts of the text material and figures from our previous publication [8].

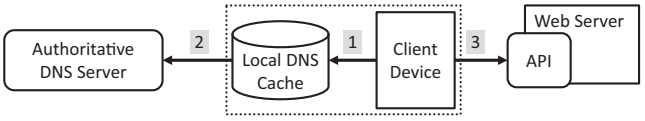


**Figure 1** Overview of the Steps Involved in Sending an HTTP Request.

findings from the 2015 and 2018 experiments, addressing availability in Section 4, performance in Section 5, and provider security preferences in Section 6. We present findings from reaching out to API providers with our 2015 results in Section 7. We discuss our approach, findings, and implications for application developers in Section 8. After presenting related work in Section 9, we conclude in Section 10.

## 2 Background

In this section, we give an overview of selected qualities in web APIs and discuss how they can be measured. For this purpose, we start with a description of individual steps in performing web API requests (Section 2.1) and potential root causes of failures (Section 2.2). Afterwards, we characterize the qualities which we have studied for this paper (Section 2.3).

### 2.1 Interaction with Web APIs

Web APIs expose *data*, e.g., a user profile or an image file, and *functionalities*, e.g., a payment process or the management of a virtual machine through a resource abstraction. This abstraction enables users to manipulate these resources without requiring insight into the underlying implementation.

Developers can access Web APIs through the *Hypertext Transfer Protocol* (HTTP), which again uses the *Transmission Control Protocol* (TCP) for error-free, complete, and ordered data transmission on the transport layer, and the *Internet Protocol* (IP) at the network layer. Figure 1 illustrates the steps involved in a typical HTTP request[4].

The resources exposed by an API are identified by *unified resource locators* (URLs), describing the scheme to be used for interaction, the server Internet address, and the specific resource identifier. The semantics of interactions with a resource depend upon the HTTP *method*, e.g., GET, POST, or

[4]For simplicity's sake, we do not include possible complications such as proxies, keep alive connections, caches, or gateways in this Figure.

DELETE. Before a client can send a request to the server that offers the web API, client and server need to establish a TCP connection. For this purpose, the client first sends a lookup request for the URL of the server to a *Domain Name Service* (DNS) server which returns the IP address and port number of the target host. If available, IP address and port may be returned from a local cache (step 1); otherwise, an external DNS authority is consulted (step 2). Afterwards, the client opens a socket connection to the server, i.e., it initiates TCP's three-way handshake, thus, establishing a TCP connection (step 3). Based on this connection, multiple HTTP requests with application data can be sent to the server.

If additional security is required, the client will typically use HTTPS which introduces the *Transport Layer Security* (TLS) protocol[5] between HTTP and TCP/IP. TLS has two main phases: a negotiation phase and a bulk data transfer phase. In the negotiation phase, the server authenticates itself through its X.509 certificate. Afterwards, the client sends its list of supported *cipher suites* (a combination of symmetric encryption algorithm and a message authentication code (MAC)) to the server which then selects a cipher suite supported by both client and server and responds accordingly. Using asymmetric encryption (e.g., RSA) and key exchange protocols (e.g., DHE), client and server also agree on a symmetric key and other TLS sessions parameters.

After this TLS handshake has been completed, the server signals a change to the bulk data transfer phase. During that phase, each HTTP request is broken down into data packets which are – based on the agreed session parameters – encrypted and signed before transmission over the network. Cipher suite and protocol version determine whether encrypt-then-MAC or the reverse order is used. The recipient can then reassemble the original request and verify its integrity based on the received MAC.

## 2.2 Sources of Failures

Considering the typical HTTP request flow described in Section 2.1, a number of possible breakpoints emerge at which a request may fail [13]. As we will see, while some of these are in control of a web API provider, others are not.

A **failed DNS lookup** is caused by attempting to look up a host for which no DNS entry exists or by a network partitioning which causes the lookup

[5]TLS has largely replaced its predecessor SSL which is typically supported only for compatibility with old clients.

request to an authoritative DNS server to time out. The first error source is rather unlikely for web API requests with the correct URL, as it would either imply the disappearance of the API's host altogether or indicate problems in the DNS system itself. Typically, a failed lookup results in a timeout error reported to the client. The second error source appears only in case that the network is not available and the DNS entry is not yet cached locally.

A **client connection error** appears if no TCP connection can be established between the client and the server hosting the web API. Reasons for this error are network partitioning or that the server is in a state where it cannot accept connections (for example, because it crashed).

In the case of HTTPS, a request can also fail if authentication of the server is not possible due to certificate issues or if there is no cipher suite supported by both client and server.

A **client error** appears if the request sent by the client cannot be processed by the server. One common reason for client errors is that the requested resource cannot be found on the server (resulting in a $404$ error code being returned). Furthermore, users may not be authorized to access the requested resource. The client may not have been aware of authentication mechanisms, e.g., basic authentication or OAuth, or may not own proper credentials. Furthermore, providers may deny authorization for specific clients if their usage of an API exceeds certain thresholds. A broad range of client errors are considered by HTTP and should result in the server sending 4xx status codes. While these errors are attested to the client, it is important to note that their appearance can be tightly related to actions of the web API provider. For example, many changes on the server, e.g., introducing authentication, removing or renaming resources, or changing request formatting, cause existing clients to malfunction, i.e., the client error is in fact caused by the web API provider.

A **server error** appears if the server fails to process an otherwise correct request. Reasons for server errors may include failed lookups for resources in databases or errors in the execution of functionalities. Server errors are, similar to client errors, considered by HTTP and should result in the server sending 5xx status codes.

## 2.3 Qualities

Systems have a number of properties. These can be functional, i.e., describe the abilities of said system, or non-functional, i.e., describe the quality of

said system. Quality describes how "good" or "bad" a system fulfills its tasks along several dimensions[6] – the qualities [9].

There is a plethora of qualities that we can see in web APIs. Examples range from availability and performance, to security, reliability, scalability, cost, or correctness (of results). All these qualities are inherently connected through complex direct and indirect tradeoff relationships [7]. In this paper, we focus on three qualities: availability, performance, and provider preferences regarding (transport layer) security.

### 2.3.1 Availability

Generally, availability describes the likelihood of a system – here, a web API – being able to respond to requests. Providing a concise definition of availability, though, is non-trivial: Does an API have to send correct responses or does it suffice if it is still able to tell about current problems? For this paper, we distinguish – based on our previous work [8] – three different kinds of availability to consider these questions:

*Pingability* describes whether there is anything "alive" at the web API provider's site. This may be a load balancer or even a fault endpoint. For a single machine deployment, pingability describes whether said machine is reachable at an operating system level. Pingability is fulfilled if, at the web API's URL, some entity responds to basic low level requests, e.g., ping requests (using the ICMP protocol).

*Accessibility* describes whether the resource represented by the web API is still accessible but not necessarily able to fulfill its task. For a single machine deployment, accessibility describes whether the web server component is reachable but does not require the hosted application logic to be accessible. A web API is accessible if it responds to HTTP requests using one of the predefined HTTP status codes.

*Successability* describes whether the web API is fully functional. For a single host deployment, it requires the application logic to be working[7]. Hence, we define successability to be fulfilled if a web API responds to requests using 2xx or 3xx status codes.

### 2.3.2 Performance

Performance has two dimensions: *latency* and *throughput*. Latency describes the amount of time between the start of a request at the client and the end

[6] It depends on the respective quality what "good" or "bad" implies.

[7] Please note that successability does not say anything about correctness of results.

of receiving a response, also at the client. Throughput, on the other hand, describes the number of requests a web API is handling at a given point in time. Typically, throughput measurements try to determine the maximum throughput, i.e., the maximum number of requests that a web API is able to handle without timeouts [9, 35, 36, 39].

Usually, these two dimensions are interconnected: If the load increases towards maximum throughput, then latency will increase. If this is not the case, then the system behind the web API is typically referred to as elastically scalable [36, 39].

### 2.3.3 Security

Security is typically characterized along several dimensions – here, we will focus on the two arguably most relevant for TLS: *confidentiality* and *integrity*. Basically, confidentiality describes whether unauthorized entities are able to access the content of API requests, and integrity asserts that transmitted data packets cannot be manipulated without the manipulation being noticed, cf., e.g., [48, 53].

It is hard to quantify how secure a web API interaction is with regards to confidentiality and integrity as this would require knowledge on all possible attack vectors. However, it is possible to interpret the selected *cipher suites* of actual API requests and the general preference order of the API provider and distinguish between weaker and stronger cipher suites. This is particularly important since the server, i.e., in this case the API provider, selects the cipher suite from the client's list of supported cipher suites.

Cipher suites list cryptographic algorithms for use in TLS (and previously SSL). A cipher suite generally names the algorithms to use for 1) key exchange, 2) authentication, 3) bulk encryption (this algorithm is also called the "cipher"), and 4) message authentication code (MAC; which is used to ensure the integrity of a message). Servers typically support a ranked list of cipher suites, from which one suite to use is agreed-on with the client during the TLS handshake procedure. Using the preference list of provider cipher suites, we can rate the security preferences of the provider.

## 3 Experiment Design

In this section, we describe our experiment design. We start by laying out the goals of our experiments (Section 3.1) before describing the qualities we want to measure (Section 3.2). We state the set of API endpoints in focus, and describe required changes that we needed to make between our two

experiment runs (Section 3.3). We then outline our measurement approach (Section 3.4) before finally describing the implementation and deployment of our measurement client (Section 3.5).

### 3.1 Goals

In our experiments, we measure qualities of web APIs *as perceived by applications*. Our goal is, for one, to be descriptive, i.e., we want to systematically assess the qualities that applications can expect from APIs. Furthermore, the goal of our measurements from an application perspective is to provide a basis for mitigation mechanisms that clients (i.e., applications) can use, as we summarize in Section 8 based on our original paper [8].

As such, we treat the API itself as a blackbox in our experiments, and measure qualities on the client level. This means that measured qualities may not only be affected by things in control of the API provider, e.g., the design or provisioning of API servers, but also by intermediaries, e.g., the network connection between client and API. As such, the results of our measurements cannot be used to compliment or blame individual API providers. Rather, they reveal real risks associated with varying or insufficient qualities that applications face when using APIs.

We initially measured web API qualities in 2015, and published a dedicated paper discussing our findings [8]. For this work, we repeated the same experiments once more in 2018 with the intention of comparing results and possibly revealing changes in qualities over time.

### 3.2 Qualities in Scope

Our experiments cover all three API qualities described in Section 2.3, namely, availability, performance, and provider security preferences. We selected these qualities as they all have possibly significant implications for applications. The availability and performance of an application can be directly impacted by these qualities in used APIs. Both, availability [46, 78] and performance [12, 50] have been shown to affect user satisfaction, user retention, and ultimately revenues. Implications of poor security choices, while harder to quantify, obviously impact the success of applications.

In addition to these technical qualities, we are interested in seeing how providers react to the results of our measurements. By trying to reach out to providers, we hope to get explanations for our results and to provide feedback from our side. Furthermore, we aim to explore how easy it is to reach providers, i.e., how easy it is for application developers to get help.

**Table 1** Benchmarked API Endpoints and Supported Protocols in 2015 [8].

| API Name | ICMP/HTTP/HTTPS | Request Meaning |
|---|---|---|
| Apple iTunes | X / X / X | Get links to resources on artists |
| Amazon S3 | - / X / X | Get file list for the 1000 genomes public data set |
| BBC | - / X / - | Get the playlist for BBC Radio 1 |
| Consumer Finance | X / X / X | Get consumer complaints on financial products |
| Flickr | X / X / X | Get list of recent photo and video uploads |
| Google Books | X / - / X | Get book metadata by ISBN |
| Google Maps | X / X / X | Query location information by address |
| MusicBrainz | X / X / X | Get information about artists and their music |
| OpenWeather Map | X / X / - | Get weather data by address |
| Postcodes.io | X / X / X | Get location information based on UK zip codes |
| Police.uk.co | - / X / X | Get street level crime data from the UK |
| Spotify | X / X / X | Get information on a given artist |
| Twitter | X / X / - | Get the number of mentions for a given URL |
| Wikipedia | X / X / X | Get a Wikipedia article |
| Yahoo | X / X / X | Get weather data by address |

We deliberately refrain from measuring throughput (and, thus, also scalability since we do not have any insight into the API provider's implementation) as we do not intend to unreasonably strain APIs, and do not want to violate terms of service explicitly prohibiting excessive request rates.

### 3.3 Selected Web API Endpoints

For our 2015 experiments, we originally selected 15 unauthenticated API endpoints (11 accessible via HTTP or HTTPS, 3 accessible only via HTTP, and one accessible via HTTPS only) [8].[8] Aiming to compare experiment results, we relied on the same endpoints in our 2018 experiments. The selected endpoints stem from a broad variety of different providers with regards to company size, country of origin, local or global target users, public or private sector. We specifically included some of the most well-known providers, e.g., Google, Apple, Amazon, and Twitter. Table 1 gives an overview of these web API endpoints, and of the protocols they supported.

Already during our 2015 experiments, two endpoints became permanently unavailable [8].

[8]We rely on unauthenticated endpoints only as authentication mechanisms may require mandatory manual setup steps, may affect measurments by introducing additional roundtrips, or may necessitate the use of software development kits (SDKs), again, with possible runtime implications.

**Table 2** Changes in API Endpoints between 2015 and 2018

| API | HTTP Change | HTTPS Change | Resp. Size |
|---|---|---|---|
| 1 | - | - | -3.1% |
| 2 | no longer available | - | n.a. |
| 3 | - | - | +12.48% |
| 4 | new resource | new resource | -99.92% |
| 5 | - | - | +254.56% |
| 6 | - | - | -13.52% |
| 7 | - | - | +25.57% |
| 8 | - | - | -45.68% |
| 9 | - | - | +42.85% |
| 10 | - | - | +16.03% |
| 11 | - | - | +50.03% |
| 12 | no longer available | - | n.a. |
| 13 | no longer available | no longer available | n.a. |
| 14 | no longer available | - | n.a. |
| 15 | no longer available | - | +38.53% |

Since then there have been additional changes as some resources, their endpoints, or even entire APIs that we originally targeted were no longer accessible in 2018. Table 2 lists all changes to the HTTP and HTTPS endpoints for the anonymized APIs. Notably, some HTTP endpoints available in 2015 have been deprecated since. In one case, for API 4, the originally targeted resource seized to exist, and we had to replace it (with an, unfortunately, much smaller one). Table 2 also indicates changes in response size between 2015 and 2018. Response sizes evolved significantly, even for cases where the API, endpoint, or even targeted resource remained constant. For some APIs, we cannot report response size changes as we were no longer able to complete requests, whether with HTTP or HTTPS.

As our measurements include factors beyond the control of API providers, and as we do not intend to discredit individual API providers, we anonymize results. For the remainder of this paper we refer to the API endpoints in focus as API-1 to API-15. There is no correlation between these identifiers and the order of API endpoints in Table 1. However, we use the same mapping as in our 2016 paper [8] for comparability. We will reveal the mapping information upon request if we are convinced that the information will not be used to discredit individual providers.

### 3.4 Measurement Approach

The qualities availability, performance, and provider security preferences are not constant, but evolve over time. As such, we opt for long-lasting

experiments, where we repeatedly measure these qualities. This approach allows us a) to increase the confidence in results as (temporary) variance can be ruled out, and b) to obtain insights into the evolution of qualities. Specifically, both in 2015 and in 2018, we executed experiments for three months each (from August 20th to November 20th 2015, and from January 30th to April 30th 2018).

We measure availability of an API in terms of pingability, accessability, and successability as described in Section 2.3. I.e., during the duration of our experiments, we repeatedly pinged the API endpoints in question and sent both HTTP and HTTPS requests to them (given their support of these protocols). We store the results of ping attempts, whether HTTP(S) requests resulted in a response from the server, and, if so, what HTTP status code was returned. To measure latency, we also measure the time between sending an HTTP(S) request from the client and receiving a response. We perform latency and availability measurements periodically every five minutes.

We measure the provider's security preferences by requesting the ordered list of supported cipher suites. We request an updated list of preferred cipher suites every twelve hours.

The above qualities of a web API can depend on the geographic region that a client is in. For example, latency depends on the geographic distance between client and API server. Or, availability and cipher suite preferences may differ as dedicated servers may respond to requests in different regions. As such, we measure qualities in a geo-distributed way, allowing us to compare results from different parts of the world.

Finally, we try to reach out to providers about our benchmarking results. We try to find contact details for all providers using Google search but also via contacts of contacts on LinkedIn. To each contact we found, we tried to send the same text including our original paper and deviated only where necessary due to the respective communication channel (e.g., on Twitter). We describe the results of this reaching out process – including both observations on contact methods but also on outcomes of attempted contact – in Section 7.

### 3.5 Benchmark Implementation

We implemented a benchmarking client consisting of three parts: First, a Java-based custom benchmarking client, that uses the standard library's HttpURLConnection class to execute HTTP and HTTPS requests to the API endpoints. We scheduled the function to run every five minutes using standard Java thread scheduling. Results are appended to a local file. Second, a bash

script that uses the standard Linux ping implementation to repeatedly ping the API endpoints in scope. We invoked the script every five minutes using the Java ProcessBuilder class, parsed the results, and also appended them to a local file. Third, a bash script that uses the cipherscan[9] open-source tool to request the API servers' preferred cipher suites. We scheduled this script to run every twelve hours by invoking it via the ProcessBuilder class and appended results to a local file.

The benchmarking client is parametrized with a list of API endpoints to target. It then starts measuring qualities using the three implementation parts. The client selects different starting points per protocol and API to avoid interference. We further implemented an HTML dashboard that allows to check that the client runs without issues and to see intermediary results. We make the benchmarking client publicly available as open-source.[10]

As in 2015, we deployed the benchmarking client on Elastic Compute Cloud (EC2) instances in the Amazon Web Services (AWS) cloud in the following seven regions: US East (Virginia), US West (Oregon), EU (Ireland), Asia (Singapore), Asia (Sydney), Asia (Tokyo), and South America (Sao Paulo).

After completing the benchmark runs, we copied created log files onto local workstations for data preprocessing and analysis.[11]

## 4 Availability Findings

In this section, we give an overview of our findings regarding availability of web APIs. Reported results are based on our experiments from both 2015 and 2018.

**Finding #1: Pingability.** Overall, all APIs show a very high pingability of around 99% in both 2015 and 2018. However, all APIs have very different pingability when comparing regions. This effect is particularly pronounced in the 2018 measurements for API-6: During the first month of our measurements, daily pingability went down to about 85% for three out of seven regions while the remaining regions still had 99% pingability. Figure 2 shows the geographical distribution of these regions. Beyond API-6, this was observable for all other APIs as well though less extreme as for these APIs

[9] https://github.com/jvehent/cipherscan

[10] https://github.com/dbermbach/web-api-bench

[11] The data collected in our experiments is available at https://github.com/ErikWittern/web-api-benchmarking-data

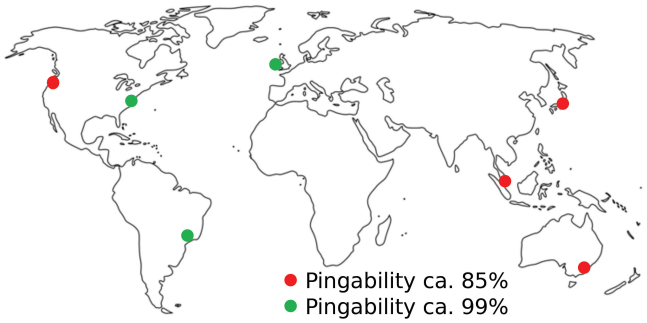


**Figure 2** Pingability of API-6 in February 2018 by Region.

**Table 3** Distribution of Lost Ping Packages Across Regions

| API | Best | Worst | Average | Worst Region |
|---|---|---|---|---|
| API-1 | 6 | 211 | 66 | Sao Paulo |
| API-3 | 4 | 374 | 82 | Sao Paulo |
| API-4 | 7 | 74 | 32 | Oregon |
| API-6 | 17 | 5852 | 2893 | Oregon |
| API-7 | 6 | 235 | 71 | Virginia |
| API-8 | 1 | 42 | 13 | Sao Paulo |
| API-9 | 20 | 476 | 155 | Sao Paulo |
| API-10 | 0 | 175 | 73 | Sao Paulo |
| API-15 | 0 | 31 | 13 | Sao Paulo |

only the number of nines varied. Table 3 shows the distribution of lost ping packages across regions; overall, clients in South America appear to have a higher probability for pingability problems than elsewhere. Noticeably, API-6 which had the strongest variance and problems encountered their worst problem in Oregon and had very good pingability in Sao Paulo. All this, however, should be taken with a grain of salt as our experiment sent close to one million ping packages per API in total.

**Finding #2: Pingability vs. Availability.** In our experiments, we also found that pingability is usually a good proxy for API availability. This, however, is not *always* the case. Figure 3 shows the pingability and accessibility of API-6 in our 2018 benchmark. As can be seen in the left part, API-6 had severe pingability problems which were not related to the overall availability of the HTTP(S) endpoints. In this period of time, using pingability as a proxy for accessibility would severely underestimate the API accessibility. In contrast, the right half of the chart shows the expected correlation of pingability and accessibility.

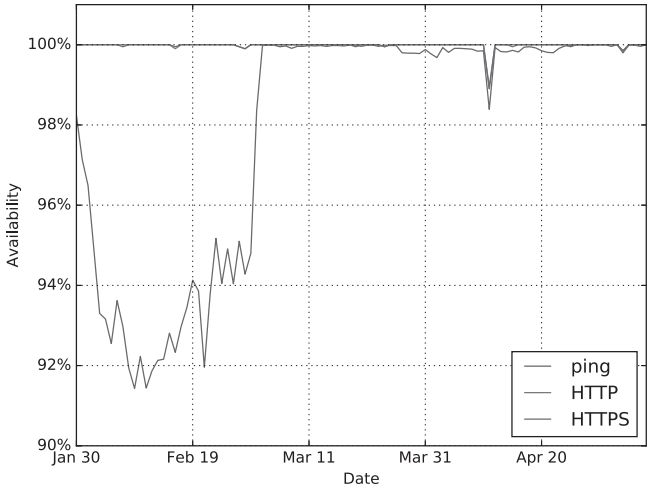


**Figure 3** Comparison of Daily Pingability and Accessibility of API-6 in 2018

**Finding #3: HTTP Status Codes.** Based on semantics of the HTTP protocol, applications are led to believe that they can expect 4xx HTTP status codes when they have done something wrong and 5xx when the fault lies with the server, or a response (with status codes 2xx or 3xx) otherwise. This could not be further from the truth. Across all our experiments, we found that up to nearly 90% of all unavailabilities resulted in no response at all, i.e., a timeout or a loss of connection. See table 4 which shows how frequently client-observable outcomes were a 4xx/5xx status code or some other sort of failure. Note, that during our 2015 benchmark, we saw two endpoints permanently going offline (the providers still exist at the time of writing this paper but deprecated the respective API endpoints): One permanently switched from 2xx to 4xx codes, the other did this for 48 hours only. Afterwards, requests failed without returning any status codes. In table 4, we show the full 2015 datasets as well as the dataset without these two APIs as both together account for about 94% of all failed requests in 2015. Overall, we can only recommend that applications not solely rely on receiving HTTP status codes, but also explicitly handle timeouts or similar failures.

**Finding #4: Longevity of API Endpoints.** As already described in finding #3, we saw two endpoints going offline during our 2015 experiments. Beyond this, four out of fourteen HTTP endpoints and one out of twelve HTTPS endpoints from our original paper were no longer available when we started our 2018 experiments (cf. Table 2). At the time of finishing this

**Table 4** Distribution of Client-Observable Results (HTTP Status Code or None) for Failed Requests

| Dataset | Protocol | 4xx | 5xx | None |
|---|---|---|---|---|
| 2015 (all) | HTTP | 43% | 5% | 53% |
| | HTTPS | 3% | 8% | 89% |
| 2015 (w/o offline endpoints) | HTTP | 0% | 74% | 26% |
| | HTTPS | 0% | 89% | 11% |
| 2018 | HTTP | 0% | 87% | 13% |
| | HTTPS | 0% | 80% | 20% |

**Table 5** Observed Success Ratios of Protocol or Region Change in Case of Failures Across all APIs

| Dataset | Strategy | Min | Max | Average |
|---|---|---|---|---|
| 2015 (all) | REGION_CHANGE | 7% | 100% | 86% |
| | HTTP→HTTPS | 0% | 100% | 88% |
| | HTTPS→HTTP | 7% | 100% | 91% |
| 2015 (w/o offline endpoints) | REGION_CHANGE | 89% | 100% | 98% |
| | HTTP→HTTPS | 78% | 100% | 97% |
| | HTTPS→HTTP | 94% | 100% | 99% |
| 2018 | REGION_CHANGE | 93% | 100% | 99% |
| | HTTP→HTTPS | 93% | 100% | 99% |
| | HTTPS→HTTP | 94% | 100% | 99% |

paper in May 2020, five out of fifteen API endpoints appear to be unavailable (either permanently or temporarily); two have changed their authentication requirements. We find this observation both surprising and troubling, especially since – contrary to what one might expect – there is no apparent relationship between size and popularity of an API and its longevity. All this together indicates that application developers should not necessarily rely on the longevity of the API endpoints used. Instead, they should closely monitor announcements on the provider's websites and have contingency plans for using alternative API endpoints.

**Finding #5: Correlation of Unavailability Across Regions and Protocols.** Based on our detailed result logs, we also explored to which degree requests to the same API endpoint experience the same availability behavior when sent (i) from different geographical regions or (ii) via different protocols (HTTP vs. HTTPS). For this analysis, we synchronized the detailed results logs as much as possible – requests did not start at the exact same time. This means that the actual numbers in the following results should be taken with a grain of salt as they are likely to underestimate the correlation: As requests may in fact be up to five minutes apart, it is certainly possible that there was a

short global outage for a period of less than five minutes. Nevertheless, we also saw longer lasting outages for requests from, e.g., a single region, while other regions remained available. So, while the actual numbers are likely to be overoptimistic, the results still indicate that the effect could be leveraged for *some* of the requests. In future work, we plan to explore to which degree this is possible in practice.

*How to read the results presented in Table 5:* The Table reports all aggregated results for different datasets (first column, for 2015 again with and without the two APIs that went completely offline) and strategies. For the strategies, REGION_CHANGE succeeds when one is able to successfully send a request from at least one other region once an API is not available from a specific region. HTTP→HTTPS and HTTPS→HTTP succeed when one is able to successfully send a request via the respective other protocol from the same machine once the first request failed. For each API, we calculated for which percentage of the failed requests the respective strategy would have been successful. The numbers in the table show these success ratios – for each dataset we show the API with the smallest success ratio ("Min"), with the highest success ratio ("Max"), and the average across all success ratios.

Yahoo! is a good example to explain how regional differences can exist. In their PNUTS paper [15], they describe how Yahoo! rolls out services globally using the underlying PNUTS system for geo-replication across data centers. User requests are then routed to the respectively closest data center and mastership for data records is adapted based on user location. If one data center has availability problems, then all read requests can be served elsewhere. The ability to serve update requests depends on whether the respective master record is stored within the affected data center.

**Finding #6: General Availability Over the Years.** Overall, availability improved from 2015 to 2018 for all three protocols. Only API-6 got slightly worse (still operating at 99.98% overall accessibility) and had some problems with ping as described in finding #2. Ignoring the two offline endpoints from 2015, availability improved from about 99.4% and 99.38% (HTTP and HTTPS respectively) to 99.94% and 99.93%. See table 6 which gives an overview of overall accessibility rates across all APIs in 2015 and 2018.

**Conclusion.** Overall, we see the main threat to developers in the longevity of APIs. When we started our experiment in 2015, we did not expect that so many API endpoints would be discontinued three and five years later. We were surprised when endpoints vanished without warning. Also, the correlation of availability across regions appears to be rather low – this makes it a

**Table 6** Overall Accessibility for HTTP and HTTPS Requests in 2015 and 2018

| Dataset | Protocol | Min | Max | Average |
|---|---|---|---|---|
| 2015 (all) | HTTP | 32.99% | 99.992% | 91.43% |
| | HTTPS | 34.17% | 99.990% | 93.95% |
| 2015 (w/o offline endpoints) | HTTP | 94.84% | 99.992% | 99.40% |
| | HTTPS | 94.79% | 99.990% | 99.38% |
| 2018 | HTTP | 99.69% | 99.998% | 99.94% |
| | HTTPS | 99.71% | 99.993% | 99.93% |

little bit challenging for an API provider to monitor API availability. Finally, availability appears to have improved from 2015 to 2018 but developers should still be careful in their API selection as not all APIs offer the same availability levels.

## 5 Performance Findings

In this section, we describe findings of analyzing the latency of web API requests both in 2015 and in 2018, with a focus on the comparison of these findings. The majority of our findings are reflected in Figure 4, which uses box plots to summarize HTTPS latencies per API and region, comparing results from 2015 (shown in gray) with those of 2018 (shown in red)[12]. We focus on HTTPS to enable comparison with our previous work [8], and, again, as HTTPS is increasingly becoming the default transport protocol used in the web. Finally, HTTP results do not provide additional insights.

**Finding #1: Inter-Regional Differences.** Latency of requests to a single API generally varies significantly depending on the geographic location of the client. We reported this finding in previous work [8] and can confirm this for 2018 as well. This variance, which we call "geofactor", is for a specific API best described by first calculating the mean latency per region $lat_i^{avg}, i \in \{regions\}$. Then the geofactor is calculated as $max(lat_i^{avg})/min(lat_i^{avg})$.

In 2015, API-7 had the lowest geofactor at 1.65; in 2018, this honor went to API-8 at 1.47. In both years, API-5 had the highest geofactor at 28 (2015)

[12]Please, note that the semantics of the whiskers have changed in comparison to [8]. Whiskers now denote the 5th and 95th percentiles which we find more intuitive; originally, we used the default of the Pandas library which is defined as follows: from the top of the box, a space of $1.5 * (Q3 - Q1)$ (with Q1 and Q3 being the 25th and 75th percentiles) is measured. The largest data point within this space is marked with the end of the whiskers, likewise for the lower whisker.

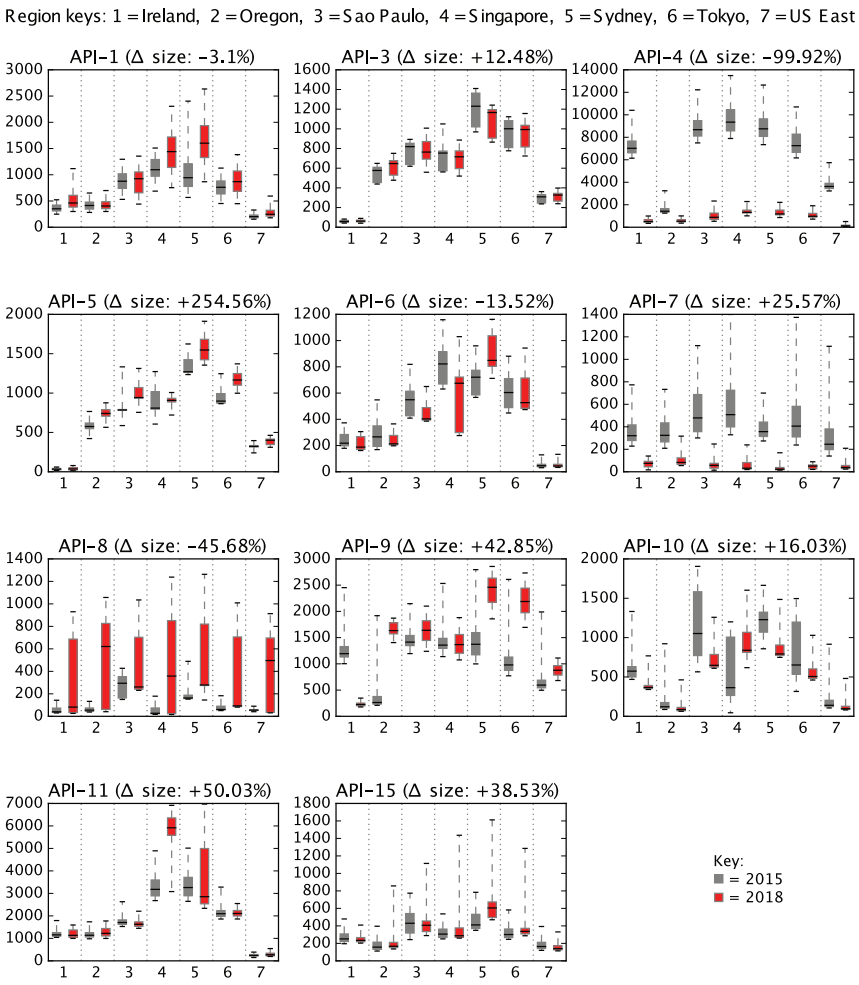


**Figure 4** Comparison of 2015 and 2018 HTTPS Request Latency Across Regions in Milliseconds; whiskers mark the 5th and 95th percentiles.

and 32 (2018) respectively. The average geofactor across all APIs increased from 9 in 2015 to 10.9 in 2018.

**Finding #2: Latency Differences within Regions over Time.** Even within individual regions, the measured response times of requests to a single API can vary a lot over time. Consider, for example, Figure 5, which shows the histogram of the HTTPS latency measured for API-8 in region Ireland in

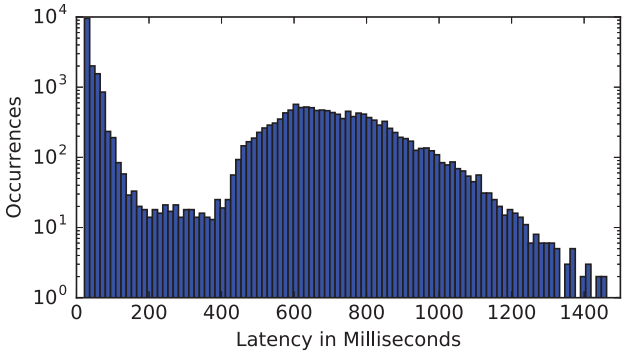


**Figure 5** Histogram of HTTPS Latency of API-8 in Ireland in 2018

2018 (note that the y-axis uses a log scale). As can be seen, a large number of requests finished in 30-150ms. However, another large share of requests requires between 400-1500ms to finish, indicating the vast latency variance in this case.

Furthermore, as reported in our previous work [8], there are significant outliers in the data reaching response times in the hundreds of seconds, which effectively render these APIs unavailable for those requests. In the 2018 data, 69 requests across seven APIs take over one minute to complete – just over half of these belong to API-1.

**Finding #3: Geo-Distribution of API Endpoints.** Measuring latency from different geographic locations allows, in some cases, to draw conclusions about the geographic distribution of API servers. Consider as an example Figures 6 and 7, which show the average daily latency for API-8 and API-9 respectively across regions. In case of API-8, latency values fall into two clusters: they are considerably worse in Sao Paulo and Sydney as compared to the rest of the world. Furthermore, the peaks of the regions' latency curves are not aligned, indicating that the API is provided by different servers in different geo locations. In contrast, in the case of API-9, the peaks of the latency curves of all regions are pretty much aligned, indicating that a single location is responsible for providing that API. Since the measured latency is consistently the lowest in Oregon, it can be assumed that this single API location is closer to Oregon than all other regions.

In the 2018 measurements, API-9 no longer exhibits the described behavior. Rather, latency peaks do not align in a clear way. In fact, in the 2018 measurements, no other API shows latency timeseries that align across all regions.

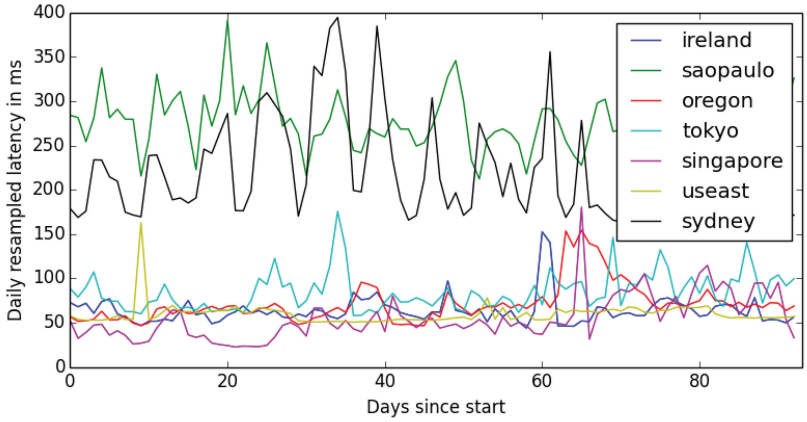


**Figure 6** Daily resampled HTTPS latency of API 8 across regions [8]

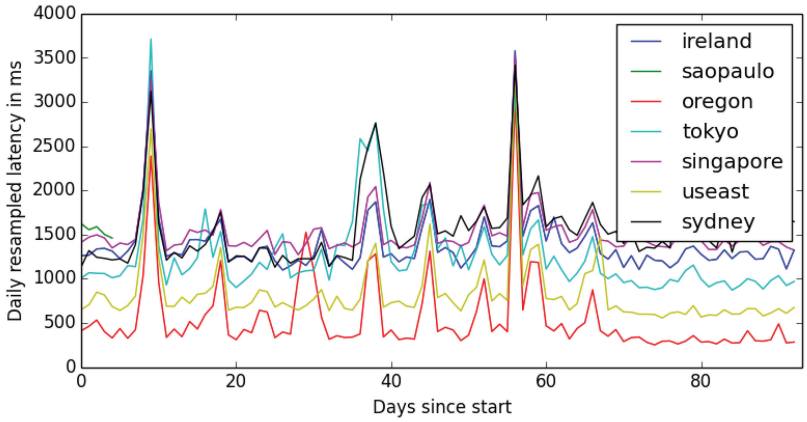


**Figure 7** Daily resampled HTTPS latency of API 9 across regions [8]

**Finding #4: Changes Between Years.** One way to depict the latency changes per API and region between 2015 and 2018 is to look at the 90% percentiles. Doing so allows us to delimit the distorting effect that outliers have. We find that, on average across regions, the latency 90% percentiles increase for 8 APIs, and decrease for 3 APIs. However, given also the changes in response sizes between the years, individual cases need to be considered: the changes for individual APIs are so different, that aggregated findings across sets of APIs are questionable. One notable example is API-4, where we had to replace the previously very large resource by a much smaller one, which is reflected in drastically reduced latency in 2018. Another example

is API-7 where, despite the resource size having increased by over $25\%$ between the years, latency has significantly decreased, across all regions. Possible causes for such improvements are the use of caching in the API backend, scaled-up servers to answer requests, or better network speeds. API-7 is also an interesting example when considering the changes of latency variance between the years. On average across regions, the standard deviation of latencies of API-7 decreased by $88.2\%$. On the other hand, consider API-8, where the standard deviation grew on average by $193.4\%$ across regions – an example that shows that increased network speed cannot be seen as the sole factor explaining latency changes between the years.

In summary, we find that multiple APIs denote stark changes in latencies and the variance of latency between 2015 and 2018. This finding underlines once more – in addition to the latency changes we observed in our individual 3-month experiments – the necessity for application developers (i) to continuously monitor the qualities of the external services they depend on, and eventually (ii) to deploy mechanisms to mitigate occurring problems.

**Conclusion.** Measuring latency of web APIs reveals severe implications for application developers. They cannot assume that latency remains constant over time (even on a short time scale, latency can increase by about 10x for a significant number of requests) and have to expect stark latency variance when accessing APIs across the globe (for a given API, the slowest region experiences on average 10x higher latency than the fastest one). If web APIs provide functionalities fundamental to an application, latency variation can have negative impacts on user attention [50] or even result in lost business [12]. This can be especially expected in cases where latency (temporarily) rises to multiple seconds. Application developers should consider mitigation strategies, as we discussed in previous work [8] and summarize in Section 8.

## 6 Security Findings

In this section, we describe findings from our security experiments. To compare APIs, we have to translate the provider preference list of cipher suites into a numerical score. For this, we define in the following a *cipher suite security score* which describes the security level for a single cipher suite and a *server security score* which aggregates the scores of all cipher suites in the respective provider's preference list.

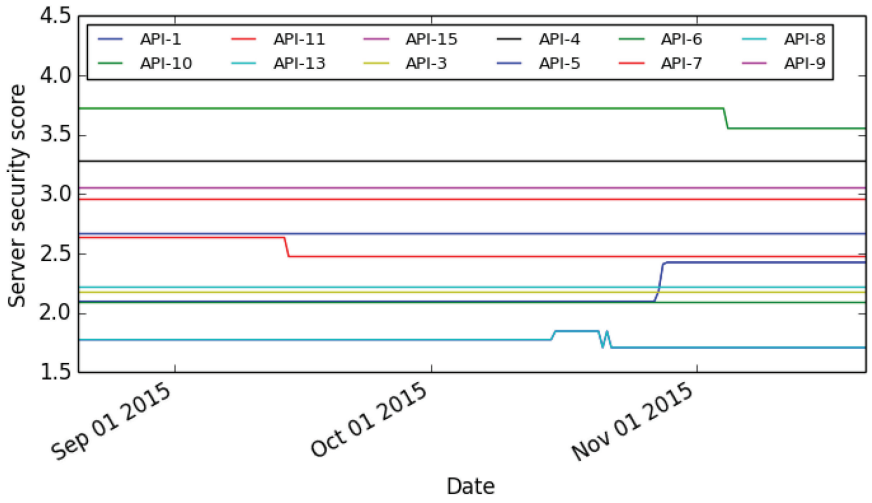


**Figure 8** Server Security Score Evolution in Region US East in 2015

We define the (heuristic) cipher suite security score $S^{CS}$ per cipher suite as follows:

$$S^{CS} = baseScore + keyLengthModifier$$

where $baseScore$ is 1 if modern algorithms are used, 0 if no perfect forward secrecy is guaranteed, and $-1$ if a generally known-to-be-weak algorithm (e.g., RC4, DES) is used. The $keyLengthModifier$ is 0.1 if the selected cipher has a strength of at least 256 bit.

The idea of these scores is not to make absolute statements about the security provided by a suite, but rather to be able to compare servers supporting different sets of cipher suites. For that purpose, based on individual cipher suite security scores, we devise the server security score $S^S$ of a server to be:

$$S^S = \sum_{i \in CS} \frac{S_i^{SC}}{r_i}$$

where $CS$ is the ranked list of cipher suites supported by the server (i.e., the ordered list of server preferences), $S_i^{SC}$ is the cipher suite security score of a suite $i$, and $r_i$ is the rank of the same suite $i$. In other words, the server security score is higher when modern cipher suites are near the top of the ranked list and lower when weak cipher suites are included, especially near the top.

As a toy example, consider a server whose cipher suite preference list is 1.) `ECDHE-RSA-AES256-SHA384`, 2.) `ECDHE-ECDSA-AES128-SHA`, and 3. `RC4-SHA`), which have cipher suite security scores, respectively, of 1.1, 1, and $-1$. The resulting server security score would be $(1.1/1)+(1/2)+(-1/3) = 1.267$. Preference list entries starting from the twelfth entry can only affect the second decimal place and beyond. For a preference list with about 30 entries, we can expect the server security score to be in the interval [-4;4].

**Finding #1: Server Security Scores Differ Across APIs.** The different APIs feature very different server security scores. Figure 8 shows scores of APIs in 2015 in region US East. As can be seen, API-8 has the lowest score, with an average of 1.76. On the other hand, API-6 has the highest score, with an average of 3.69. Similarly, in 2018, average server security scores range from 1.58 (API-3) to 3.15 (API-4).

**Finding #2: Lasting Score Changes in 2015.** To characterize whether and how security scores of an API change during our experiments, we define a *lasting change* to exist if the security score changes by at least 1% between measurements, and if the new score is maintained for at least ten subsequent measurements (i.e., five days). This definition of a lasting change is informed by exploring plots of the evolution of security scores, which denote some outlying scores and some changes that persist for longer periods (i.e., days/weeks).

We find in the 2015 experiments that out of the 12 APIs, 5 (or 41.7%) denote lasting changes in their security scores (cf. also Figure 8). Three of these APIs (API-1, API-6, API-7) denote a single change, while the other two APIs (API-8, API-15) denote two lasting changes. In fact, the security scores of of these two APIs nearly perfectly overlap. Information obtained prior to anonymization of our data indicates that these two APIs are deployed on the same managed cloud infrastructure. Figure 9 shows, as an example, the 2015 evolution of the server security score of API-15, which has a (positive) lasting change around Oct 20, and another (negative) lasting change around Oct 26. Notably, in the 2018 data, we did not find a single lasting change.

**Finding #3: Changes between 2015 and 2018.** During our experiments, the server security scores featured lasting changes for only a minority of APIs in 2015, and for no APIs in 2018. However, the server security scores of *all* APIs changed between our two experiments. The median value of the changes (no matter if positive or negative) is 18%, with a smallest change of 1.7% (a negative change; API-7) and a maximum change of 27.42% (also a negative

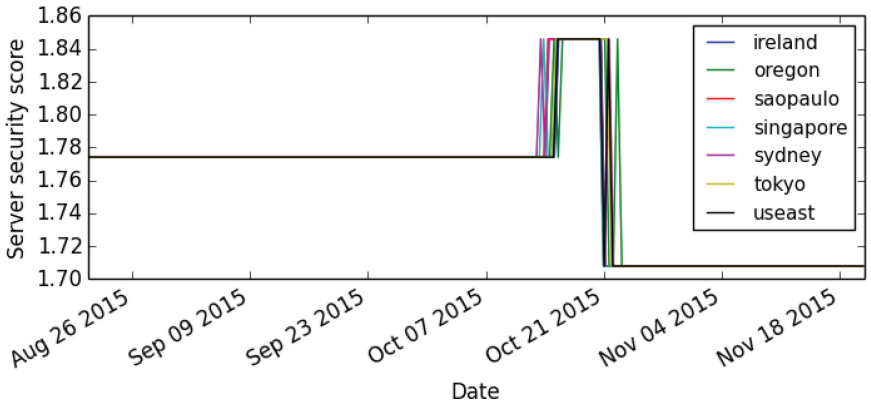


**Figure 9** Server Security Score Evolution of API-15 in 2015

change; API-3). In six cases, the score in 2018 is lower than in 2015, in the remaining five cases it is the other way round.

**Finding #4: Server Security Scores are the Same Across Regions.** We find that the security scores of APIs were, largely, the same across regions in both 2015 and 2018. I.e., we find no case where security scores were different for long periods of time, or where a lasting change (as defined in Finding #2) occurred in one region but not in others regions (within at least a few days). The largest time interval between lasting changes appeared for API-15, where the score lastingly decreased from 1.85 to 1.77 in region Ireland midday EST of Oct 25, 2015 while the same change occurred in Oregon only midday EST on Oct 27 (i.e., two days later). See again Figure 9.

**Finding #5: Discontinuation of Cipher Suites.** We observe six cipher suites that are only present in the 2015 data. Table 7 shows these suites, their cipher suite security score, and the APIs that support them in the 2015 data. Notably, the cipher suite security scores range from -1 to 1.1, i.e., these suites include the whole spectrum of scores (and not only, for example, weak cipher suites).

**Finding #6: Reduced Use of Insecure Cipher Suites.** Our cipher suite security score assigns a minimum value of -1 to suites that have known vulnerabilities. For example, the RC4 algorithm is known to be vulnerable and was prohibited from use in TLS by the Internet Engineering Task Force in 2015[13]. When counting the frequency in which cipher suites with a score of -1 appear across APIs and across regions, we find 31,316 occurrences in 2015

[13]For details, cf. https://www.rfc-editor.org/info/rfc7465

**Table 7** Cipher Suites Appearing only in 2015 Experiments

| Suite | $S^{CS}$ | APIs 2015 with Support |
|---|---|---|
| DHE-RSA-CAMELLIA256-SHA | 1.1 | 4, 6, 9 |
| DHE-RSA-CAMELLIA128-SHA | 1.0 | 4, 6, 9 |
| CAMELLIA256-SHA | 0.1 | 4, 9, 13 |
| CAMELLIA128-SHA | 0.0 | 4 7 |
| ECDHE-RSA-RC4-SHA | -1 | 1, 8, 10, 15 |
| RC4-MD5 | -1 | 1, 8, 10, 15 |

and only 6775 occurrences in 2018. While this improvement is not reflected in the evolution of server security scores between the years (cf. Finding #3), it does indicate that over time the "worst offenders" are removed.

**Conclusion.** Quantifying security is a notoriously hard if not impossible challenge. We defined cipher suite/server security scores not to make absolute statements about security of APIs, but to be able to compare API provider security preferences and assess the changes over time. The server security scores of an API are closely related across regions, with only short temporal discrepancy. Also, we only found lasting changes of server security scores in the 2015 experiments. However, considering the full picture of measurements between 2015 and 2018, we find that all APIs feature changes in security scores, and that these changes are not necessarily leading to higher security. We find, though, that the use of insecure cipher suites has dropped from 2015 to 2018.

## 7 Findings from Reaching Out

In this section, we give an overview of the results of reaching out to all API providers whose APIs we benchmarked in our original paper [8].

In our assessment, we decided that email or an official contact form is the best way of contacting a provider as it provides (i) a dedicated contact channel (other than a forum that might or might not be monitored) and (ii) a way of communicating potentially sensitive information of the API user. We believe that Twitter and online forums are the second best options as on these channels contact attempts are more likely to be overlooked and also, e.g., in the case of Twitter, require users to entrust potentially sensitive data to a third party. Finally, we ranked all other contact methods worse than this (but better than no contact method). Following this reasoning, we first tried to find an official contact form or support email address for the API. If that was not successful we tried Twitter or online forums. If that also did not work we

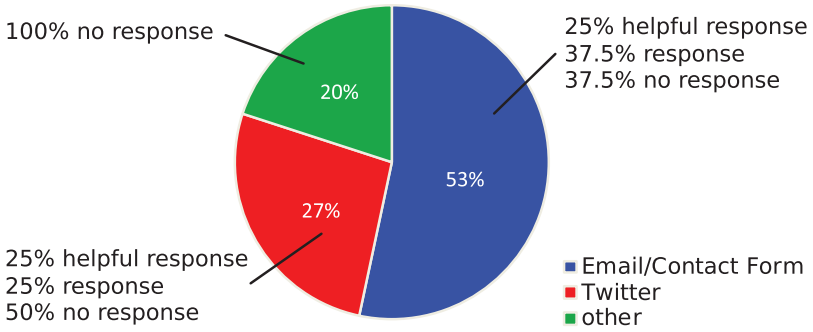


**Figure 10** Availability of Contact Methods and Results of Reaching Out to Providers

tried any other contact method that we could find via Google search, LinkedIn contacts of contacts, and other options.

In our contact request, we pointed to our original paper [8] and its findings. We asked the recipient to possibly comment on results, ask questions, or provide us with feedback. We also made it explicit that we intended to cover their response or reaction in this paper unless they would specifically ask us not do this in their response message (we received no such message).

**Finding #1: Availability of Contact Methods.** We were able to find contact forms or official support email addresses for eight (out of fifteen) APIs, four had Twitter profiles. For the other APIs, we contacted one via an existing "contact of a contact" on LinkedIn, for one we found a self-proclaimed "senior developer" of the API on GitHub and his email address via Google search. Finally, we found one personal website on which the respective person claimed responsibility for that API (and provided some contact details).

**Finding #2: Quality of Responses (Email/Contact Form).** Of our eight contact requests via official support emails or contact forms, two resulted in interesting conversations where the API provider appeared interested in our results and provided helpful comments. Three of the eight requests resulted in either an automated email or an email that was a bit more personalized which indicated that our request had been forwarded to the respective development teams and that we would get a response soon (we did not). Another three requests were completely ignored.

**Finding #3: Quality of Responses (Twitter).** Of the four requests we sent via Twitter to the respective API support handles, one provider was very interested and provided helpful comments leading to an email-based discussion. Another provider asked us to send them a private message with further details

but we did not receive an answer to that. The last two providers ignored our requests completely.

**Finding #4: Quality of Responses for Other Contact Methods.** Neither of the three contact requests did result in any reaction of the API provider. See also Figure 10 which gives an overview of all responses.

**Finding #5: Quality of Responses (Summary).** All in all, only three out of fifteen providers offered helpful comments and were actually approachable. There was no correlation with the provider size, country, or industry.

**Finding #6: Insights Gained.** Aside from the meta-information on whether we received answers or not, we also learned some architectural details. For instance, one of the APIs provides their content strictly read-only. In their deployments, they bundle API code and a database instance. This makes it relatively easy to guarantee high availability and performance as the number of replicas is only restricted by the available budget. Other APIs have it much harder as their backends are subject to the well-known tradeoffs of PACELC [1] and other scalability restrictions.

**Conclusion.** Overall, developers should not rely on the ability to reach API providers in the case of problems. Based on our observations, we believe that it is a rare exception when developers can get support unless they are explicitly paying for it. We would recommend to primarily use APIs that are at least well documented and preferably widely used.

## 8 Discussion

In this work, we set out to repeat experiments originally performed in 2015 and published in previous work [8]. As such, the scope of our paper is naturally limited by the choices we made in 2015 and the developments on the provider side since then. In this section, we will discuss possible threats to validity and limitations of our approach as well as overall recommendations.

**Selection of APIs:** In 2015, we picked 15 public APIs from a variety of domains, countries, provider types, and popularity levels. While a larger number may have been preferably, this was not feasible due to the vast amount of data which we needed to analyze using rather time-consuming (manual) exploratory data analysis [9]. Nevertheless, we believe that our provider selection managed to achieve a large degree of diversity in the set of APIs benchmarked and, hence, covers a broad range of API characteristics that application developers are likely to encounter “in the wild”. As such,

our experiments also showcase a number of behaviors which application developers have to deal with in practice – by no means do we wish to insinuate that the list of behaviors observed is representative for the set of *all* APIs or that the list of behaviors is complete. In fact, the findings from our experiments should not necessarily be generalized to the overall API from which the endpoint stems. For example, providers may choose to remove an endpoint we benchmarked, while other endpoints of the same API remain available.

Another aspect is that we only used GET requests – due to caching, actual availability may be worse. Nonetheless, we deem our results valid examples of how web API qualities can impact applications, which ultimately rely on specific endpoints. For many APIs, GET may even be the standard way of accessing it, e.g., for a maps API. We, furthermore, limited our experiments to endpoints that do not require authentication. One might argue that these endpoints may be of less importance to their providers and may, thus, undergo less scrutiny than other endpoints. Nonetheless, these endpoints may well be used by applications and the here presented findings should, hence, be considered relevant to application developers. Still, our observations should not be generalized to endpoints with mandatory authentication in place.

Since our goal was to give a broad overview of behaviors and developments that are observable in practice, reporting anonymized results does not affect reproducibility of results. In fact, we will gladly provide the pseudonymization mapping on request, e.g., to reproduce our experiments or to correlate our results with external events. While we are aware that anonymized results render our findings less "sensational", we deemed it more important to avoid finger-pointing.

Overall, we believe that we achieved our goal of showing a range of behaviors that application developers can expect in practice as well as their evolution over time.

**Benchmarking vs. Monitoring:** Benchmarking is usually defined as creating stress on a system under test while observing its reaction [9]. Our experiments certainly did not create stress on the API endpoints as the terms of service tend to forbid anything that resembles a DDOS attack. Monitoring, however, is usually defined as passive observation which we definitely did not do in our experiments [9]. Overall, we believe that our measurement approach falls into the benchmarking category but is a rather unusual example of benchmarking.

**Application-Centric Benchmarking:** Benchmarking itself is a black box experiment in which a measurement process interacts with a system

under test. Depending on the scenario, additional information may be used in a second step to turn the experiment into a white box or grey box experiment. The latter category is particularly useful when taking the perspective of providers who aim to improve their offering.

Our goal, however, was to take an application perspective to understand the effects that application developers are confronted with in practice. As such, our analysis is naturally limited to identifying correlations and deriving possible explanations without the means to verify them through provider-internal knowledge. In contrast, this also allows us to find results such as the vanishing endpoints which would either not be observed or even disregarded in provider-centric benchmarking.

**Correction of Previous Results:** Due to a bug in our benchmarking client, presumably in the JVM runtime used, some measurement threads died and vanished without apparent reason in both 2015 and 2018. The unexpected death of measurement threads, inexplicably happening without any log entries, meant that data collected in both experiments has a number of missing data points. As the bug happened without any hint in our logs, which should have captured such an incident, we only discovered this as part of our reanalysis in 2019. Specifically, manual analysis of the raw collected data reveals that 26 (or $10.0\%$) of the 259 measurement threads (7 regions, 14 HTTP, 12 HTTPS, 11 ICMP endpoints) died during the 2015 experiments – dying threads appeared only for HTTP and HTTPS measurements. The death of threads produced no error logs, and subsequent data points were simply missing. Our original data analysis tool wrongly interpolated these missing data points to mean that the server was unreachable, leading to us over-reporting unavailabilities of APIs in Table 2 of our paper [8]. In Section 4 of this paper, we present corrected availability figures, where we no longer attribute missing data points to mean that the API was unavailable.

To avoid such a mishap, we re-wrote our analysis scripts for availability from scratch in Java, thus having an additional implementation (besides our updated Python script that used the pandas library). For both versions of the data analysis code, we wrote additional test cases and compared their outputs to improve their reliability.

**Implications for Application Engineering:** In our original paper, we discussed that API consumers are directly affected by provider decisions and have no control over the consumed service which was also visible in the results [8]. Namely, latency and availability varied depending on the geo-origin of requests and in general showed a high variance. Also, endpoints vanished during our experiment. As a consequence, we recommended to rely

on a variety of mechanisms such as caching, request queuing, monitoring, and API notification services[14]. We also discussed preliminary ideas for a geo-distributed middleware which acts as a proxy and performs protocol changes or tunnels requests through other regions when necessary and feasible.

Overall, none of these recommendations have become invalid. What is most disturbing, though, is the sheer number of API endpoints that does no longer exist in 2018 – and neither of these endpoints is for an API that might be simply obsolete such as a service that returns information on outdated technology. Based on this, we can only recommend to actively look for API alternatives (e.g., Google Maps *and* Bing Maps) or to build upon APIs that could be self-hosted if necessary (e.g., Open Street Map). In terms of reaching out, developers should not expect much customer support (this might be better for APIs with a paid plan) so that we would recommend to carefully check existing documentation and developer forums before committing to an API. Finally, security-wise, we have seen that many APIs still support obsolete cipher suites. Since the server selects the cipher suite, we would recommend to restrict the set of supported cipher suites on the client to modern ones only so that downgrades are no longer possible. This also asserts that users are less affected by varying cipher suite preferences of providers.

## 9 Related Work

In this section, we give an overview of related work starting with web API evolution before describing web API characteristics and benchmarking.

### 9.1 Web API Evolution

Previous work has studied how web APIs evolve, characterizing change patterns and resulting challenges for application developers [43], and assessing how developers react to these changes [69]. Focusing on implications for applications, Espinha et al. [20] have shown that mobile applications show diverse behavior in light of web API evolution. Nearly a third of the analyzed applications crash in light of the removal of fields from response messages, highlighting severe consequences that affect user experiences. In a similar vein, Aue et al. [3] report from a single payment API provider's point of view about the cause and scope of erroneous integrations – reflected by millions of

[14] e.g., https://www.apichangelog.com

error logs. These papers underline the important role web APIs play for many applications.

More recently, GraphQL [29, 71, 72] has evolved as a new paradigm for querying web APIs. Instead of providers defining a fixed sets of endpoints, consumers form queries within the bounds of an API's schema to retrieve or mutate precise subsets of data. This gives API providers more leeway in updating their APIs without breaking client applications as the queries can remain the same. Consequently, we now see wide adoption of GraphQL in industry, e.g., at Netflix [61], PayPal [63], or GitHub [65].

In contrast to these papers, we do not address web API evolution, i.e., how the functionality of a web API changes, but the rather focus on web API qualities, i.e., its changing non-functionalities. We argue that these two perspectives complement one another, and can have similar importance. Errors due to faulty integration or issues caused by unavailabilities, security issues, or high latency can equally threaten how users experience applications. Similar, both API providers and consuming developers are challenged to avoid such issues, relying for example on the mitigation strategies hinted at in Section 8.

Finally, as we could see in our experiments, many availability issues are due to API evolution. While our approach will detect such changes as soon as another benchmark run is started, Yang et al. [76] can detect web API behavior changes based on documentation updates – if, for instance, such a documentation is maintained in a versioned repository, it is easily possible to subscribe to content changes. Also, Bae et al. [4] can detect API misuse from the application source code and can thus help to avoid issues that result from applications that leverage an unintended loophole in an API which is later on fixed. Both approaches focus on functional behavior.

### 9.2 Web API Characteristics and Ecosystems

A wide array of related work assesses what web APIs exist, how they are used, and how they can be characterized. Early studies of API ecosystems focused on *ProgrammableWeb*, a community-maintained catalog of web APIs. Analyses have explored the evolution of this ecosystem [70, 77], how APIs in ProgrammableWeb are (reportedly) used in mashups [31], or utilized such usage relations to recommend web APIs to developers [42, 73]. Other studies assessed web API usage outside of ProgrammableWeb, and specifically in the context of mobile applications. For example, Oumaziz et al. perform static analysis on mobile applications to assess if and to what extent

they interact with web APIs [51]. Rapoport et al. combine static analysis with a dynamic execution of selected mobile applications, exploring how to best detect consumption of web APIs [56]. Finally, Wittern et al. studied how to detect the use of web APIs in JavaScript-based application code mined from GitHub using static analysis [75] and a recent study surveyed GraphQL schemas mined from GitHub [71].

These publications emphasize the importance of web APIs for application developers and an increasing recognition of related challenges and research opportunities [74].

Yet another set of studies assesses characteristics of existing web APIs. Rodriguez et al. extract web API requests from large volumes of HTTP traffic logs and assess their adherence to various design principles, including proper use of HTTP verbs or how URLs are structured [57]. Neumann et al. manually identify web APIs among popular web sites and assess their documentations for a similar set of design principles [49]. Zdun et al [79] study design choices in API design empirically, Ivanchikj et al. [33] describe how to derive interaction patterns from API logs, and [45, 62] model proven API design choices in the form of reusable patterns.

We again see our work to complement these approaches, which focus on design aspects of APIs, as this work focuses on runtime qualities.

## 9.3 Benchmarking

Benchmarking comprises a number of different areas. The oldest area is concerned with quantifying performance of (virtual) machines. These include many SPEC[15] benchmarks or collections such as the Phoronix Test Suite[16] but also more recent developments, e.g., Borhani et al. [11] who use WordPress as a realistic benchmark application, DocLite [67] which uses Dockerized light workloads to rank cloud-based virtual machines, or Cloud WorkBench [58, 59] which automates cloud VM benchmarking. Another well-studied area is the benchmarking of database systems which, in the last few years, has evolved from relational database benchmarks such as the well-established TPC[17] benchmarks towards modern NoSQL datastores. For instance, the de facto standard YCSB [16] and its extensions [18, 54] introduced database benchmarking based on CRUD interfaces, which are more compatible with modern NoSQL stores such as Apache Cassandra [41].

[15] https://www.spec.org/

[16] https://www.phoronix-test-suite.com/

[17] http://www.tpc.org

Other approaches, e.g., OLTPBench [19] or BenchFoundry [5, 6], aim to build comprehensive multi-quality benchmarking platforms that also include measurement approaches for qualities beyond performance, e.g., data consistency [2, 7, 68, 80] or elastic scalability [36, 39, 55]. Beyond these, there are a number of approaches studying performance impacts of TLS on NoSQL datastores [48, 52, 53], web services [34], and web servers [14]. Also, Ferme et al. [21] have studied the performance of workflow systems.

More recently, benchmarking has targeted Function-as-Service, e.g., [37, 38, 44], has been used in continuous integration and deployment pipelines [17, 22, 26, 30, 60], has been used to evaluate microservices [27] and their effects on today's data centers [25], and has been used to study capabilities of edge nodes [47].

Neither of these approaches is directly comparable to the work presented in this paper as these approaches all have in common that they *stress* the system under test while our approach is more *lightweight* and does not create significant load on web APIs as such a practice is explicitly forbidden in the terms of the service of most web APIs. In this regard, our approach is also comparable to monitoring approaches [40] but, in contrast to them, is not entirely passive. For an extended discussion of the broad area of benchmarking – this list is by no means exhaustive, we refer to our recent book on cloud service benchmarking [9] or foundational publications on the principles of benchmarking such as [10, 24, 28, 32, 66].

To our knowledge, this paper and our original paper [8] are the only publications that aim to quantify quality of service of web APIs through geo-distributed long-term experiments. Other measurement approaches such as artillery.io[18] can be used to run load tests against web APIs. This, however, is explicitly forbidden in the terms of service of all APIs that we have looked at (which includes a much longer list than those used for our experiments). As such, it is unlikely to provide insights into the behavior of web APIs "in the wild" or needs to be run by the provider. Finally, one might argue that approaches such as UptimeRobot[19] could replace our approach. This is not true since UptimeRobot and other monitoring tools (i) collect and expose only a fraction of the data that we collected and needed, (ii) invoke APIs only from a single location unless the request from that origin failed, and (iii) are neither open source nor extensible. Nevertheless, they are highly valuable tools in practice.

[18] https://artillery.io/

[19] https://uptimerobot.com/

## 10 Conclusion

Over the last few years, web APIs have found widespread adoption in mobile, web, and desktop applications. Hence, quality behavior of web APIs more and more affects user experience of such applications – from a developer perspective, this is highly problematic as they have absolutely no control over third-party APIs and their quality.

In 2016, we published a conference paper on benchmarking of web APIs [8]. For this paper, we extended our original benchmarking tool and repeated the three-month experiment. Of the original 15 APIs from 2015, only 11 remained in 2018. For all these, we reported detailed results and corresponding insights on availability, performance, and provider security preferences. We also analyzed to which degree the API providers can be contacted and actually respond to inquiries.

## References

[1] Daniel Abadi. Consistency tradeoffs in modern distributed database system design: Cap is only part of the story. *IEEE Computer*, 45(2):37–42, February 2012.

[2] Eric Anderson, Xiaozhou Li, Mehul A. Shah, Joseph Tucek, and Jay J. Wylie. What consistency does your key-value store actually provide? In *Proceedings of the 6th Workshop on Hot Topics in System Dependability (HOTDEP)*, HotDep'10, pages 1–16, Berkeley, CA, USA, 2010. USENIX Association.

[3] J. Aué, M. Aniche, M. Lobbezoo, and A. van Deursen. An Exploratory Study on Faults inWeb API Integration in a Large-Scale Payment Company. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, pages 13–22, May 2017.

[4] SungGyeong Bae, Hyunghun Cho, Inho Lim, and Sukyoung Ryu. Safewapi: web api misuse detector for web applications. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 507–517, 2014.

[5] D. Bermbach, J. Kuhlenkamp, A. Dey, A. Ramachandran, A. Fekete, and S. Tai. BenchFoundry: A Benchmarking Framework for Cloud Storage Services. In *Proceedings of the 15th International Conference on Service Oriented Computing (ICSOC 2017)*. Springer, 2017.

[6] D Bermbach, J Kuhlenkamp, A Dey, S Sakr, and R Nambiar. Towards an Extensible Middleware for Database Benchmarking. In *TPCTC 2014*, pages 82–96. Springer, 2014.

[7] David Bermbach. *Benchmarking Eventually Consistent Distributed Storage Systems*. PhD thesis, Karlsruhe Institute of Technology, 2014.

[8] David Bermbach and Erik Wittern. Benchmarking web api quality. In *Proceedings of the 16th International Conference on Web Engineering (ICWE 2016)*. Springer, 2016.

[9] David Bermbach, Erik Wittern, and Stefan Tai. *Cloud Service Benchmarking: Measuring Quality of Cloud Services from a Client Perspective*. Springer, 2017.

[10] Carsten Binnig, Donald Kossmann, Tim Kraska, and Simon Loesing. How is the Weather Tomorrow?: Towards a Benchmark for the Cloud. In *Proc. of DBTEST*, pages 1–6. ACM, 2009.

[11] Amir Hossein Borhani, Philipp Leitner, Bu-Sung Lee, Xiaorong Li, and Terence Hung. WPress: An Application-Driven Performance Benchmark for Cloud-Based Virtual Machines. In *Proc. of EDOC*, pages 101–109. IEEE, 2014.

[12] Jake Brutlag. Speed Matters for Google Web Search. Technical report, Google, Inc., 2009.

[13] Jürgen Cito, Devan Gotowka, Philipp Leitner, Ryan Pelette, Dritan Suljoti, and Schahram Dustdar. Identifying web performance degradations through synthetic and real-user monitoring. *J. Web Eng.*, 14(5&6):414–442, 2015.

[14] Cristian Coarfa, Peter Druschel, and Dan S Wallach. Performance Analysis of TLS Web Servers. *ACM Transactions on Computer Systems (TOCS)*, 24(1):39–69, 2006.

[15] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proceedings of the VLDB Endowment*, 1(2):1277–1288, August 2008.

[16] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proc. of SOCC*, pages 143–154. ACM, 2010.

[17] David Daly, William Brown, Henrik Ingo, Jim O'Leary, and David Bradford. The use of change point detection to identify software performance regressions in a continuous integration system. In *Proceedings of*

*the ACM/SPEC International Conference on Performance Engineering*, pages 67–75, 2020.

[18] Akon Dey, Alan Fekete, Raghunath Nambiar, and Uwe Röhm. Ycsb+ t: Benchmarking web-scale transactional databases. In *2014 IEEE 30th International Conference on Data Engineering Workshops*, pages 223–230. IEEE, 2014.

[19] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudre-Mauroux. Oltp-bench: An extensible testbed for benchmarking relational databases. *Proceedings of the VLDB Endowment*, 7(4):277–288, 2013.

[20] Tiago Espinha, Andy Zaidman, and Hans-Gerhard Gross. Web API Fragility: How Robust is Your Mobile Application? In *Proc. of MOBILESoft*, pages 12–21. IEEE, 2015.

[21] Vincenzo Ferme, Ana Ivanchikj, and Cesare Pautasso. A framework for benchmarking bpmn 2.0 workflow management systems. In *International conference on business process management*, pages 251–259. Springer, 2016.

[22] Vincenzo Ferme and Cesare Pautasso. A declarative approach for performance tests execution in continuous software development environments. In *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*, ICPE '18, page 261–272, New York, NY, USA, 2018. Association for Computing Machinery.

[23] Roy T Fielding and Richard N Taylor. *Architectural Styles and the Design of Network-based Software Architectures*, volume 7. University of California, Irvine Irvine, USA, 2000.

[24] Enno Folkerts, Alexander Alexandrov, Kai Sachs, Alexandru Iosup, Volker Markl, and Cafer Tosun. Benchmarking in the cloud: What it should, can, and cannot be. In *Proceedings of the 4th TPC Technology Conference on Performance Evaluation and Benchmarking (TPCTC 2012)*, pages 173–188. Springer, 2013.

[25] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 3–18, 2019.

[26] M. Grambow, F. Lehmann, and D. Bermbach. Continuous Benchmarking: Using System Benchmarking in Build Pipelines. In *Proceedings*

*of the 1st Workshop on Service Quality and Quantitative Evaluation in new Emerging Technologies*, 2019.

[27] M. Grambow, L. Meusel, E. Wittern, and D. Bermbach. Benchmarking Microservice Performance: A Pattern-based Approach. In *Proceedings of the 35th ACM Symposium on Applied Computing*, 2020.

[28] Jim Gray. *The Benchmark Handbook for Database and Transaction Systems*, chapter Database and Transaction Processing Handbook. Morgan Kaufmann, 2nd edition, 1993.

[29] Olaf Hartig and Jorge Pérez. Semantics and complexity of graphql. In *Proceedings of the 2018 World Wide Web Conference*, pages 1155–1164, 2018.

[30] A. Van Hoorn, J. Waller, and W. Hasselbring. Kieker: A Framework for Application Performance Monitoring and Dynamic Software Analysis. In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*, pages 247–248, 2012.

[31] Keman Huang, Yushun Fan, and Wei Tan. An Empirical Study of Programmable Web: A Network Analysis on a Service-Mashup System. In *2012 IEEE 19th International Conference on Web Services*, pages 552–559. IEEE, 2012.

[32] Karl Huppler. The art of building a good benchmark. In *Proceedings of the First TPC Technology Conference on Performance Evaluation and Benchmarking (TPCTC 2009)*, pages 18–30. Springer, 2009.

[33] Ana Ivanchikj, Ilija Gjorgjiev, and Cesare Pautasso. Restalk miner: mining restful conversations, pattern discovery and matching. In *International Conference on Service-Oriented Computing*, pages 470–475. Springer, 2018.

[34] Matjaz B Juric, Ivan Rozman, Bostjan Brumen, Matjaz Colnaric, and Marjan Hericko. Comparison of Performance of Web Services, WS-Security, RMI, and RMI–SSL. *Journal of Systems and Software*, 79(5):689–700, 2006.

[35] M. Klems, D. Bermbach, and R. Weinert. A Runtime Quality Measurement Framework for Cloud Database Service Systems. In *Proc. of QUATIC*, pages 38–46, 2012.

[36] Donald Kossmann, Tim Kraska, and Simon Loesing. An Evaluation of Alternative Architectures for Transaction Processing in the Cloud. In *Proc. of SIGMOD*, pages 579–590. ACM, 2010.

[37] J. Kuhlenkamp and S. Werner. Benchmarking FaaS Platforms: Call for Community Participation. In *Proceedings of the 4th International Workshop on Serverless Computing*. 2018.

[38] J. Kuhlenkamp, S. Werner, M. C. Borges, K. El Tal, and S. Tai. An Evaluation of FaaS Platforms as a Foundation for Serverless Big Data Processing. In *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing*, 2019.

[39] Jörn Kuhlenkamp, Markus Klems, and Oliver Röss. Benchmarking Scalability and Elasticity of Distributed Database Systems. pages 1219–1230, 2014.

[40] Jörn Kuhlenkamp, Kevin Rudolph, and David Bermbach. AISLE: Assessment of Provisioned Service Levels in Public IaaS-based Database Systems. In *Proc. of ICSOC*, pages 154–168. Springer, 2015.

[41] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *SIGOPS Operating Systems Review*, 44(2):35–40, April 2010.

[42] Chune Li, Richong Zhang, Jinpeng Huai, and Hailong Sun. A Novel Approach for API Recommendation in Mashup Development. In *2014 IEEE International Conference on Web Services*, pages 289–296. IEEE, 2014.

[43] Jun Li, Yingfei Xiong, Xuanzhe Liu, and Lu Zhang. How Does Web Service API Evolution Affect Clients? In *2013 IEEE 20th International Conference on Web Services*, pages 300–307. IEEE, 2013.

[44] W. Lloyd, S. Ramesh, S. Chinthalapati, L. Ly, and S. Pallickara. Serverless computing: An Investigation of Factors Influencing Microservice Performance. In *Proceedings of the IEEE International Conference on Cloud Engineering*, pages 159–169, 2018.

[45] Daniel Lübke, Olaf Zimmermann, Cesare Pautasso, Uwe Zdun, and Mirko Stocker. Interface evolution patterns: balancing compatibility and extensibility across service life cycles. In Tiago Boldt Sousa, editor, *Proceedings of the 24th European Conference on Pattern Languages of Programs, EuroPLoP 2019, Irsee, Germany, July 3-7, 2019*, pages 15:1–15:24. ACM, 2019.

[46] Henry Martinez. How Much Does Downtime Really Cost? https://www.information-management.com/news/how-much-does-downtime-really-cost. Accessed: 2019-12-02.

[47] Jonathan McChesney, Nan Wang, Ashish Tanwer, Eyal de Lara, and Blesson Varghese. Defog: fog computing benchmarks. In *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*, pages 47–58, 2019.

[48] S Müller, D Bermbach, S Tai, and F Pallas. Benchmarking the Performance Impact of Transport Layer Security in Cloud Database Systems. In *Proc. of IC2E*, pages 27–36. IEEE, 2014.

[49] Andy Neumann, Nuno Laranjeiro, and Jorge Bernardino. An Analysis of Public REST Web Service APIs. *IEEE Transactions on Services Computing*, 2018.

[50] Jakob Nielsen. *Usability Engineering*. Elsevier, 1st edition, 1994.

[51] Mohamed A Oumaziz, Abdelkarim Belkhir, Tristan Vacher, Eric Beaudry, Xavier Blanc, Jean-Rémy Falleri, and Naouel Moha. Empirical Study on REST APIs Usage in Android Mobile Applications. In *International Conference on Service-Oriented Computing*, pages 614–622. Springer, 2017.

[52] F. Pallas, D. Bermbach, S. Müller, and S. Tai. Evidence-based security configurations for cloud datastores. In *Proceedings of the the 32nd ACM Symposium on Applied Computing*. ACM, 2017.

[53] Frank Pallas, Johannes Günther, and David Bermbach. Pick your choice in hbase: Security or performance. In *Proceedings of the IEEE International Conference on Big Data (Big Data 2016)*. IEEE, 2017.

[54] Swapnil Patil, Milo Polte, Kai Ren, Wittawat Tantisiriroj, Lin Xiao, Julio López, Garth Gibson, Adam Fuchs, and Billie Rinaldi. YCSB++: Benchmarking and Performance Debugging Advanced Features in Scalable Table Stores. In *Proc. of SOCC*, pages 1–14. ACM, 2011.

[55] T. Rabl, M. Sadoghi, H.-A. Jacobsen, S. Gómez-Villamor, V. Muntés-Mulero, and S. Mankovskii. Solving Big Data Challenges for Enterprise Application Performance Management. *Proceedings of the VLDB Endowment*, 5(12), 2012.

[56] Marianna Rapoport, Philippe Suter, Erik Wittern, Ondřej Lhóták, and Julian Dolby. Who you gonna call? Analyzing Web Requests in Android Applications. In *Proceedings of the 14th International Conference on Mining Software Repositories*, pages 80–90. IEEE Press, 2017.

[57] Carlos Rodríguez, Marcos Baez, Florian Daniel, Fabio Casati, Juan Carlos Trabucco, Luigi Canali, and Gianraffaele Percannella. Rest apis: a large-scale analysis of compliance with principles and best practices. In *International Conference on Web Engineering*, pages 21–39. Springer, 2016.

[58] J. Scheuner and P. Leitner. Performance Benchmarking of Infrastructure-as-a-Service (IaaS) Clouds with Cloud WorkBench. In *Companion of the 2019 ACM/SPEC International Conference on Performance Engineering*, pages 53–56, 2019.

[59] J. Scheuner, P. Leitner, J. Cito, and H. Gall. Cloud WorkBench – Infrastructure-as-Code Based Cloud Benchmarking. In *Proceedings of the IEEE 6th International Conference on Cloud Computing Technology and Science*, pages 246–253, 2014.

[60] Henning Schulz, Dušan Okanović, André van Hoorn, Vincenzo Ferme, and Cesare Pautasso. Behavior-driven load testing using contextual knowledge - approach and experiences. In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*, ICPE '19, page 265–272, New York, NY, USA, 2019. Association for Computing Machinery.

[61] Artem Shtatnov and Ravi Srinivas Ranganathan. Our learnings from adopting GraphQL. https://netflixtechblog.com/our-learnings-from-adopting-graphql-f099de39ae5f. Accessed: 2020-04-01.

[62] Mirko Stocker, Olaf Zimmermann, Uwe Zdun, Daniel Lübke, and Cesare Pautasso. Interface quality patterns: Communicating and improving the quality of microservices apis. In *Proceedings of the 23rd European Conference on Pattern Languages of Programs*, EuroPLoP '18, New York, NY, USA, 2018. Association for Computing Machinery.

[63] Mark Stuart. GraphQL: A success story for PayPal Checkout. https://medium.com/paypal-engineering/graphql-a-success-story-for-paypal-checkout-3482f724fb53. Accessed: 2020-04-01.

[64] Philippe Suter and Erik Wittern. Inferring Web API Descriptions from Usage Data. In *Proc. of the 3rd IEEE Workshop on Hot Topics in Web Systems and Technologies (HotWeb)*, pages 7–12, 2015.

[65] G. Torikian, B. Black, B. Swinnerton, C. Sommerville, D. Celis, and K. Daigler. The GitHub GraphQL API. https://github.blog/2016-09-14-the-github-graphql-api/. Accessed: 2020-04-01.

[66] Jóakim v. Kistowski, Jeremy A. Arnold, Karl Huppler, Klaus-Dieter Lange, John L. Henning, and Paul Cao. How to build a benchmark. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering (ICPE 2015)*, pages 333–336. ACM, 2015.

[67] B. Varghese, L. T. Subba, L. Thai, and A. Barker. Container-Based Cloud Virtual Machine Benchmarking. In *Proceedings of the IEEE International Conference on Cloud Engineering*, pages 192–201, 2016.

[68] Hiroshi Wada, Alan Fekete, Liang Zhao, Kevin Lee, and Anna Liu. Data Consistency Properties and the Trade-offs in Commercial Cloud Storages: the Consumers' Perspective. In *Proc. of CIDR*, pages 134–143, 2011.

[69] Shaohua Wang, Iman Keivanloo, and Ying Zou. How Do Developers React to RESTful API Evolution? In *Proc. of ICSOC*, pages 245–259. Springer, 2014.

[70] Michael Weiss and GR Gangadharan. Modeling the mashup ecosystem: Structure and growth. *R&d Management*, 40(1):40–49, 2010.

[71] Erik Wittern, Alan Cha, James C. Davis, Guillaume Baudart, and Louis Mandel. An empirical study of graphql schemas. In Sami Yangui, Ismael Bouassida Rodriguez, Khalil Drira, and Zahir Tari, editors, *Service-Oriented Computing*, pages 3–19, Cham, 2019. Springer International Publishing.

[72] Erik Wittern, Alan Cha, and Jim A. Laredo. Generating graphql-wrappers for rest(-like) apis. In Tommi Mikkonen, Ralf Klamma, and Juan Hernández, editors, *Web Engineering*, pages 65–83, Cham, 2018. Springer International Publishing.

[73] Erik Wittern, Jim Laredo, Maja Vukovic, Vinod Muthusamy, and Aleksander Slominski. A Graph-based Data Model for API Ecosystem Insights. In *Proc. of ICWS*, pages 41–48. IEEE, 2014.

[74] Erik Wittern, Annie Ying, Yunhui Zheng, Jim A. Laredo, Julian Dolby, Christopher C. Young, and Aleksander A. Slominski. Opportunities in Software Engineering Research for Web API Consumption. In *Proceedings of the 1st International Workshop on API Usage and Evolution*, WAPI '17, pages 7–10, Piscataway, NJ, USA, 2017. IEEE Press.

[75] Erik Wittern, Annie T. T. Ying, Yunhui Zheng, Julian Dolby, and Jim A. Laredo. Statically Checking Web API Requests in JavaScript. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, pages 244–254, Piscataway, NJ, USA, 2017. IEEE Press.

[76] Jinqiu Yang, Erik Wittern, Annie TT Ying, Julian Dolby, and Lin Tan. Towards extracting web api specifications from documentation. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, pages 454–464. IEEE, 2018.

[77] Shuli Yu and C Jason Woodard. Innovation in the Programmable Web: Characterizing the Mashup Ecosystem. In *International Conference on Service-Oriented Computing*, pages 136–147. Springer, 2008.

[78] Emmanuele Zambon, Sandro Etalle, Roel J Wieringa, and Pieter Hartel. Model-based qualitative risk assessment for availability of it infrastructures. *Software & Systems Modeling*, 10(4):553–580, 2011.

[79] Uwe Zdun, Mirko Stocker, Olaf Zimmermann, Cesare Pautasso, and Daniel Lübke. Guiding architectural decision making on quality aspects in microservice apis. In *International Conference on Service-Oriented Computing*, pages 73–89. Springer, 2018.

[80] Kamal Zellag and Bettina Kemme. How Consistent is Your Cloud Application? In *Proc. of SOCC*. ACM, 2012.

## Biographies

**David Bermbach** is an Assistant Professor at TU Berlin and is heading the Mobile Cloud Computing research group at the Einstein Center Digital Future in Berlin, Germany. In his research, he focuses on benchmarking as well as platforms and applications for cloud, edge, and fog computing. He holds a PhD in computer science and a diploma in business engineering, both from Karlsruhe Institute of Technology.

**Erik Wittern** is IBM's GraphQL Lead Architect, and works on bringing GraphQL support to IBM's API Management products. Prior to his current role, Erik spent five years as a Research Staff Member at the IBM T.J. Watson Research Center in New York. His research in the field of Software Engineering focuses on web APIs, their discovery and use, and the evolution of new API models like GraphQL. Erik holds a PhD in computer science from Karlsruhe Institute Of Technology.