DZone®

TREND REPORT

Application Performance Management

BROUGHT TO YOU IN PARTNERSHIP WITH

Coralogix

Welcome Letter

By Tyler Sedlar, Software Engineer at DZone

Application performance management (APM) is one of the most crucial toolsets one can utilize when releasing applications to the public. Broadly, APM enables teams to create a successful application that drives the end-user experience in a positive direction. In particular, profiling an application for CPU and memory performance enhancements is incredibly important to any application — enterprise or not.

Application performance management consists of two different components: passive and active monitoring. Passive monitoring relies on incoming web traffic for analytics and metrics. Active monitoring, on the other hand, relies on simulations and behavioral scripts.

Both components are vital to improve the user experience behind the scenes (via the back end) and up front on the user-facing application prior to multiple users encountering off-putting issues like slow performance or downtime.

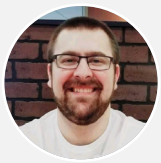
Passive monitoring can be used to analyze web traffic and collect engagement metrics on targeted pages or subdomains. This APM approach can also be used to detect Denial-of-Service (DDoS) attacks. Further, it helps measure endpoint latency, giving teams the ability to know which popular endpoints should receive engineering attention in order to improve performance. Active monitoring, sometimes referenced as synthetic monitoring, consists of simulation scripts tailored to performing actions on a live site while recording metrics such as availability and response times. Programmatically, Selenium is often the tool of choice for active monitoring. It gives webmasters the ability to proactively identify if a web page is experiencing slow speeds or even downtime, then mitigate any problems before they affect end users. Many common metrics are collected in this way, including time to first byte, speed index, time to interactive, and page complete, just to name a few.

DZone's 2021 Application Performance Management Trend Report explores the industry's current state through our research on application design and architecture for performance and self-healing, practices to ensure reliability, and how organizations are measuring performance.

Featured contributors also share their insights into areas including performance of distributed cloud-based architectures, OpenTelemetry, and criteria for choosing APM tools. Read on to learn more about the importance of integrating APM into your application's lifecycle for a better end-user experience.

Sincerely,

Tyler Sedlar



Tyler Sedlar, Software Engineer at DZone

@tsedlar on DZone | @tsedlar on LinkedIn

Tyler is a Software Engineer at DZone. He was introduced to software development by creating automation scripts for MMORPGs with Java and has made software development his career since. He also enjoys programming as a hobby as well — noting his favorite languages as Kotlin and Node.js. He otherwise enjoys being around family, playing board games, and playing video games such as LoL, WoW, and OSRS.

ORIGINAL RESEARCH

Key Research Findings

An Analysis of Results from DZone's 2021 Application Performance Survey

John Esposito, PhD, Technical Architect at 6st Technologies

In November 2021, DZone surveyed software developers, architects, and other IT professionals in order to understand how applications are being designed, released, tuned, and monitored for performance.

Major research targets were:

1. Designing and architecting applications for performance and self-healing
2. Organizational practices aimed at ensuring performance and reliability
3. Measuring application performance

Methods:

We created a survey and distributed it to a global audience of software professionals. Question formats included multiple choice, free response, and ranking. Survey links were distributed via email to an opt-in subscriber list and popups on DZone.com. The survey was opened on October 28 and closed on November 12. The survey recorded 321 responses.

In this report, we review some of our key research findings. Many secondary findings of interest are not included here. Additional findings will be published piecemeal on DZone.com.

Research Target One: Designing and Architecting Applications for Performance and Self-Healing

Motivations:

1. Undergraduate computer science curricula rigorously treat algorithmic complexity (and hence, algorithm performance on an abstract Turing machine), but often treat less rigorously other more engineering-like, less mathematical aspects of software performance. However, the combination of modern hardware, tooling (including compiler and interpreter) maturity, commodity compute infrastructure, and ubiquity of web applications has tended to centralize the importance of algorithmic complexity in a few codebases run on a proportionally small number of machines — more than was the case when undergraduate algorithm/data structure pedagogies were developed.

   This has left a large portion (in fact, we conjecture, the large majority) of performance optimization to 'higher' levels that undergraduate degrees — especially those with a heavier focus on *computer science* as distinguished from *software engineering* — are less precisely aimed at treating. We wanted to see how developers are handling performance at these supra-control-flow levels.

2. Tony Hoare's catchy (and obviously useful) dictum, "premature optimization is the root of all evil," too often serves as a blanket excuse to consider performance too late — sometimes due to engineering incompetence, sometimes due to managerial cost-cutting.

   Of course, any human (and perhaps especially engineering) endeavor must balance costs and benefits of up-front planning time, real-world feedback from a running system, likelihood of future changes that obsolete fragile early tuning, runtime costs amortized over time, etc. But this simply means that one may pay too much attention to performance both too early and too late. We wanted to see how developers are making these trade-offs under real-world pressures.

3. Modern commodity infrastructure enables sloppy programming, and like everything, it's subject to the how-much-is-gold-plating-worth trade-off. But our own experience in enterprise consulting suggests the amount of entropy (or less abstractly, heat) code creates — that might easily have been written differently — must be incalculable.

   The aesthetic side of any programmer must recoil in horror at this, even when the trade-off (based on time, expense, source complexity, readability, etc.) seems desirable. We wanted to get a better sense of the triumphs, tactics, roadblocks, and annoyances encountered in the war against entropy that software professionals fight every day.

SOFTWARE PERFORMANCE PATTERNS

Many performance-related software design choices depend too precisely on specific data structures, access patterns, and runtime infrastructure to be treated meaningfully at the level addressed by current research. However, we wanted to understand how thinking about performance affects software design as accurately as our high-level survey would allow, so we devised a list of common "patterns" that describe high-level approaches to designing for performance that may be taken across many variations of data structures, algorithms, hardware architectures, application architectures, etc. We asked:

Which of the following software performance patterns have you implemented?

Definitions:

- ***Express Train*** *– for some tasks, create an alternate path that does only the minimal required work (e.g., for data fetches requiring maximum performance, create multiple DAOs — some enriched, some impoverished)*
- ***Hard Sequence*** *– enforce sequential completion of high-priority tasks, even if multiple threads are available (e.g., chain Ajax calls enabling optional interactions only after minimal page load, even if later calls do not physically depend on earlier returns)*
- ***Batching*** *– chunk similar tasks together to avoid spin-up/spin-down overhead (e.g., create a service that monitors a queue of vectorizable tasks and groups them into one vectorized process once some threshold count is reached)*
- ***Interface Matching*** *– for tasks commonly accessed through a particular interface, create a coarse-grained object that combines anything required for tasks defined by the interface (e.g., for an e-commerce cart, create a CartDecorated object that handles all cart-related calculations and initializes with all data required for these calculations)*
- ***Copy-Merge*** *– for tasks using the same physical resource, distribute multiple physical copies of the resource and reconcile in a separate process if needed (e.g., database sharding)*
- ***Calendar Control*** *– when workload timing can be predicted, block predicted work times from schedulers so that simultaneous demand does not exceed available resources*

Results (n=321):

Figure 1

IMPLEMENTATION OF SOFTWARE PERFORMANCE PATTERNS

	Express train	Hard sequence	Batching	Interface matching	Copy-merge	Calendar control
Often	24.4%	19.3%	38.0%	28.1%	24.1%	22.4%
Sometimes	43.7%	39.6%	37.1%	38.8%	33.7%	32.3%
Rarely	18.7%	25.0%	18.1%	22.4%	25.1%	28.8%
Never	13.3%	16.1%	6.9%	10.7%	17.2%	16.6%
n=	316	316	321	317	303	313

Observations:

1. Batching is the performance pattern most commonly applied often (38%) or at all (93.1%).

   This is perhaps the most broadly applicable and least complex pattern listed, and it is similar enough to basic principles such as data locality (e.g., when batching permits reused values to be held as in-memory static variables rather than requiring multiple I/O calls) that we might expect batching from even the most junior developers. This turns out to be only partly true: Both senior (>5 years' experience) and junior (≤5 years' experience) software professionals used batching more overall (93.1% and 89.2%, respectively) than any other performance pattern, but junior respondents report using the interface matching pattern more often (34.7%) than batching (31.1%).

   The difference is small, but from this, we weakly conjecture that less-experienced developers' minds may fly a bit more in "application space" than "infrastructure space" since the pattern we called interface matching involves grouping work under constructs meaningful in the application domain.

2. Interface matching is the second most commonly applied performance pattern, both often (28.1%) and overall (89.3%).

   To interpret this, we note that the use of the interface concept and the example provided in our definition suggest a class/inheritance-based, object-oriented approach to encapsulation, which may or may not be aimed intentionally (or effectively) at performance. Segmentation of responses by "primary programming language used at work" suggests this may influence responses: Java-primary developers were significantly more likely than JavaScript-primary developers to use interface matching often (28.8% vs. 18.2%).

   This may indicate that Java or another interface-happy OO language is more naturally suited to this particular performance pattern. It may also indicate some answer contamination by respondents who might have checked this answer option simply if they use interfaces or decorators, whether or not they do with specific intent to optimize performance. Note: Differences in "often" responses between Java-primary and JavaScript-primary respondents were significantly smaller for all other answer options — most within two or three percentage points, and one around six percentage points.

3. Express train and copy-merge are effectively tied for third most commonly implemented performance pattern. No significant differences obtained between senior and junior respondents' use of express train. Copy-merge, however, was significantly more likely to be used by senior developers: 25.4% (senior) vs. 18.2% (junior) use this pattern often, and 84.6% vs. 74.2% use this pattern ever.

   We tentatively explain this difference by the varying levels of experience we suppose are required to implement these patterns effectively: Any sufficiently clever person (of any experience level) understands that, for instance, a database query can be accelerated by selecting fewer fields — an insight that does not involve a wide-ranging mental model of the system. But experience (as distinguished from cleverness) helps build confidence in the more global mental modeling required to make scheduling decisions.

APPROACHES TO SELF-HEALING

We suppose that self-healing is especially important in modern software development for three primary reasons:

1. Even simple programs written in modern high-level languages can quickly explode in complexity beyond the human ability to understand execution clearly and precisely (or else formal verification would be trivial for most programs, which is far from the case). This means that reliably doing computational work requires robustness against unimagined execution sequences.

2. Core multiplication, branch prediction (and related machine-level optimizations) and horizontal worker scaling — often over the Internet — have borne the brunt of the modern extension of Moore's Law. This means that communication channels (represented as graphs) have high cardinality, grow rapidly, may not be deterministic, and are likely to encounter unpredictable partitioning.

   Further, many popular modern languages implement garbage collection, whose details are (by design) should not enter the application programmer's awareness. That is, a lot of low-level work is supposed to be handled invisibly to the programmer — which means that modern programs must be able to respond increasingly well to situations that were not considered the application's own formal design.

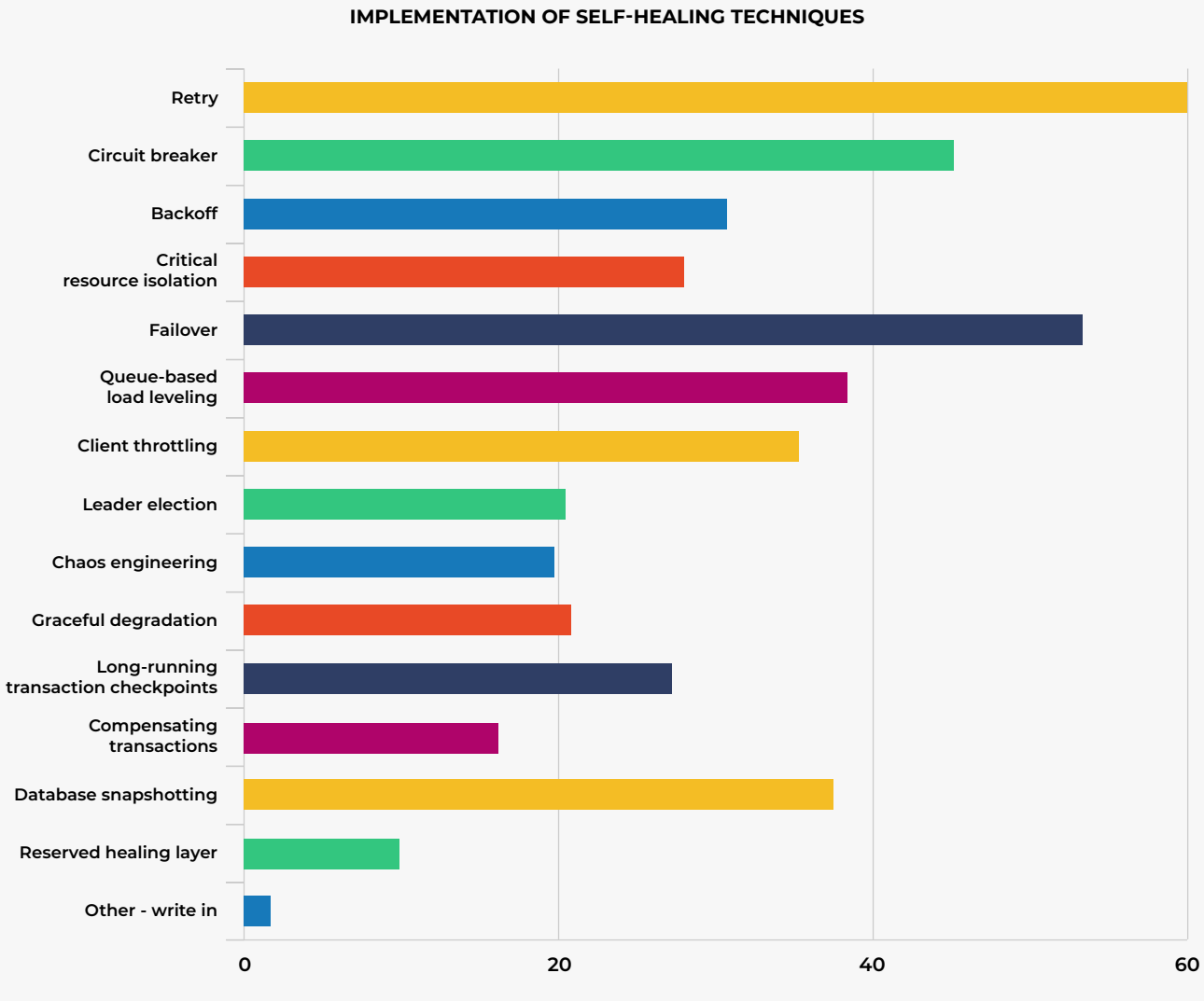
3. Mobile and IoT devices, increasingly dominant execution platforms, are constantly losing radio connections to the Internet — for physical reasons even more so, as demands for high bandwidth require higher-frequency (and hence, more obstruction/weather-susceptible) radio signals. So network partitioning worsens and becomes harder to hand-wave away.

We wanted to know what techniques software professionals are using for automatic recovery from application failure, asking:

Which of the following approaches to self-healing have you implemented? {Retry | Circuit breaker | Backoff | Critical resource isolation | Failover | Queue-based load leveling | Client throttling | Leader election | Chaos engineering | Graceful degradation | Long-running transaction checkpoints | Compensating transactions | Database snapshotting}

Results:

Figure 2



Observations:

1. The top three self-healing techniques — retry, failover, and database snapshotting — are old enough to have obvious pre-computer, and probably even pre-engineering, analogues: All living things retry, anyone planning a dinner party makes failover plans, and redundancy is as old as the scribe.

2. Two of the next three self-healing techniques — circuit breaker, queue-based load leveling, and client throttling — are more organizationally sophisticated in the sense that they require representation in more complex data structures and execution paths.

Circuit breakers involve thinking holistically and defensively about systems nonlinearly (i.e., involving cascades), and queue-based load leveling is simply a service-oriented sort of buffer. Client throttling is as simple as telling a caffeinated interlocutor to slow down, so in principle, it is simple. In practice, of course, client throttling can become complex because it requires rate coordination between provider and consumer, which itself is quite tricky, especially when the involved systems' elasticities are heterogeneous.

3. Java-primary developers were significantly more likely to implement circuit breakers than JavaScript-primary developers (50.7% vs. 36.4%), perhaps because Java applications are more likely to be server-side and, therefore, more likely to locate the architectural complexity that benefits from circuit breakers.
4. Java-primary developers were also significantly more likely to implement backoffs (33.6% vs. 18.2%), perhaps because web servers, most likely to be called by JavaScript, are designed to accept promiscuous connection attempts (i.e., do not expect backoff from clients).
5. The most architecturally explicit self-healing technique — a reserved healing layer — was rare but not negligibly so (10.3% of respondents). This number was higher than we expected and will be interesting to monitor in the future.

IMPACT OF MICROSERVICES ON PERFORMANCE ENGINEERING

We imagine that microservice architectures impact designing for performance in three ways:

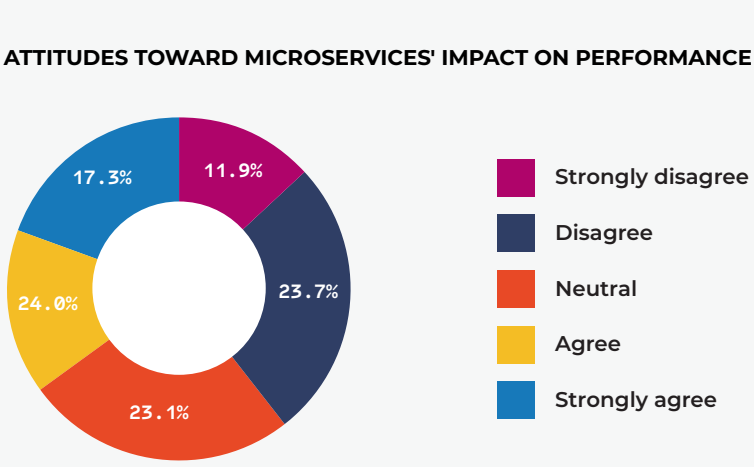
1. More complex, more distributed, and more heterogeneous architecture makes precise understanding of low-level details more difficult; low-level details often make a huge difference in performance.
2. Many mature tools (profilers, loggers) and techniques (core dump analysis, resource checkpoints) were originally designed for single virtual machines. Microservices make these tools less useful, and the analogous tooling ecosystem for more distributed systems is not yet as mature.
3. The release independence that microservices offer as a development time benefit can make runtime performance a gambling-esque headache, as a change used to improve performance in one microservice may harm performance in another without anyone noticing until the change is released. For instance, if an identity microservice that all other work needs to call suddenly slows down, the other services will all slow down even though none of their code has changed.

We wanted to see how software professionals think about microservices' impact on performance at a high level and how attitudes toward microservices relate to approaches taken to performance optimization, so we asked:

Microservices make performance engineering, tuning, and monitoring more difficult: {Strongly disagree | Disagree | Neutral | Agree | Strongly agree}

Results (n=312):

Figure 3



Observations:

1. Almost a fifth (17.3%) of respondents strongly agreed, and 41.3% agreed or strongly agreed, that microservices make performance engineering, tuning, and monitoring more difficult.

   We did not ask *how much* more difficult — presumably the *how much* enters the trade-off calculus that weighs performance and other factors when deciding on an overall approach to architecture — but these high numbers should remind software architects not to forget about performance when considering microservices.

2. Senior respondents were almost twice as likely to respond that they "strongly agree" than junior respondents (19.3% vs. 10.6%).

   At this high level, it is impossible to know which group is more correct, but if we apply a general principle of "more trust in more experience," then we might take these responses as a strong indicator that special attention should be paid to performance when implementing microservices.

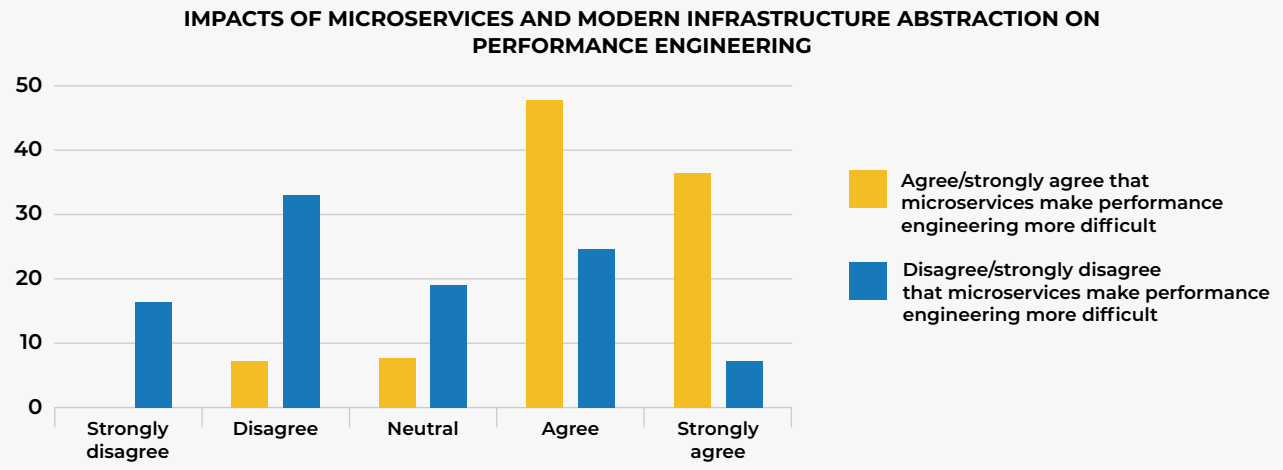
3. However, a comparable number of respondents strongly disagreed (11.9%) or strongly agreed (35.6%) that microservices make performance engineering, tuning, and monitoring more difficult.

   Complementing the senior vs. junior divergence with respect to agreement or strong agreement, more junior respondents disagreed (27.3% vs. 22.3%), but slightly more senior respondents strongly disagreed (11.6% vs. 9.1%). This perhaps mitigates the "beware performance w.r.t. microservices" inference we might draw from the senior agreement/strong agreement.

4. Attitudes toward microservices' impact on performance engineering map onto attitudes toward infrastructure abstraction's impact on performance engineering.

   This suggests that perhaps some of respondents' judgment of microservices' impact on performance engineering stems from the general impact of complexity and abstraction on performance engineering, rather than exclusively suggesting something specific about microservices:

Figure 4



IMPACT OF MODERN DEPLOYMENT ABSTRACTIONS ON PERFORMANCE ENGINEERING

Microservices can be understood as part of an overall distributed application architecture, but lower-level system architecture also impacts performance independently of application architecture. So, just as we wanted to know how software professionals feel about microservices' impact on performance engineering, we also wanted to know (independently) attitudes toward the impact of modern deployment/runtime abstractions on performance engineering.

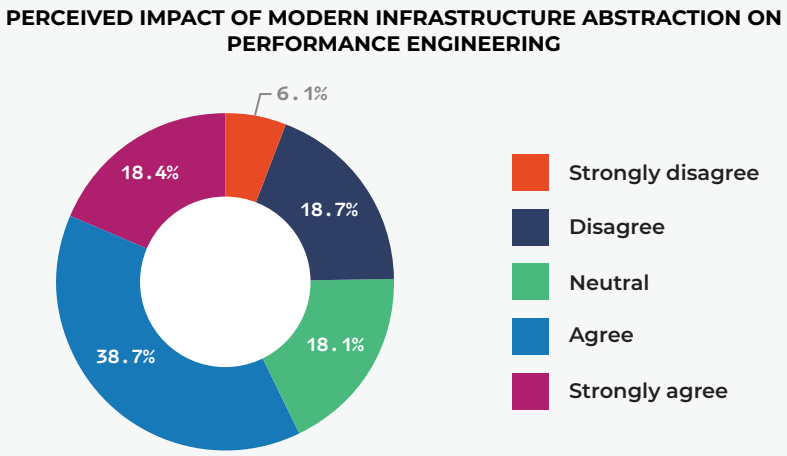
(Continued on next page)

We asked:

The high degrees of abstraction, elasticity, and virtualization layers common in modern software deployments make performance engineering, tuning, and monitoring more difficult: {Strongly disagree | Disagree | Neutral | Agree | Strongly agree}

Results (n=310):

Figure 5



Observations:

1. Results for this question are less equivocal than for the analogous question w.r.t. microservices. The majority (57.1%) of respondents agreed or strongly agreed (18.4%) that the high degrees of abstraction, elasticity, and virtualization layers common in modern software deployments make performance engineering, tuning, and monitoring more difficult.

2. Almost a fifth of respondents (18.7%) disagreed, but only 6.1% strongly disagreed. So respondents' ambivalence toward the impact of microservices on performance engineering is absent here.

3. In other surveys, we asked about general attitudes toward infrastructure abstraction. Between 2020 and 2021, the percent of respondents who answered:

   - "Infrastructure abstraction in {YEAR} is excessive and getting out of control" grew slightly (16% to 18.4%).
   - "We're finally getting close to pure, non-leaky infrastructure abstraction" fell by a similar proportion (21.3% to 18.8%).
   - "No opinion" grew comparably (13.7% to 15.5%).

   Together, with the results of the present question, we speculate that the difficulty of performance engineering on these abstractions contributes to growing suspicion of these levels of abstraction. However, since we have only one data point regarding performance, we cannot yet determine a parallel trend. We intend to ask this performance-related infrastructure abstraction question in future surveys, as we continue to ask about general infrastructure abstraction each year.

ROOT CAUSES OF WEB PERFORMANCE DEGRADATION

Taking web apps as the most common example of a complex distributed system over unknown lower layers, we wanted to know what makes web apps slow. So we asked:

How often have you encountered the following root causes of web performance degradation? {High CPU load | CPU thrashing | Memory exhausted (paging) | I/O bottleneck | Network bottleneck | Too many disk I/O operations due to bad code or configuration | Slow disk I/O due to bad code or configuration | Excessive algorithmic complexity due to bad code | Misuse of language features | Deadlocks or thread starvation | Load balancing lag | Geographic location lag | Selective/rolling deployment lag | Log rotation batch | Database reorganization | Garbage collection | Network backup}

Results (n=311):

Figure 6

FREQUENCY OF WEB PERFORMANCE DEGRADATION ROOT CAUSES

	Often	Sometimes	Rarely	Never
High CPU load	38.3%	43.7%	15.4%	2.6%
CPU thrashing	15.7%	38.7%	35.3%	10.3%
Memory exhausted (paging)	28.9%	37.8%	28.3%	4.9%
I/O bottleneck	25.1%	38.3%	28.7%	7.9%
Network bottleneck	20.5%	41.0%	30.3%	8.1%
Too many disk I/O operations due to bad code or configuration	17.0%	41.5%	30.7%	10.8%
Slow disk I/O due to bad code or configuration	12.2%	39.3%	36.6%	11.9%
Excessive algorithmic complexity due to bad code	22.2%	39.0%	30.8%	8.1%
Misuse of language features	14.4%	33.4%	34.1%	18.1%
Deadlocks or threat starvation	13.5%	33.6%	40.8%	12.2%
Load balancing lag	14.0%	31.2%	41.2%	13.6%
Geographic location lag	11.0%	26.4%	33.8%	28.8%
Selective/rolling deployment lag	10.1%	25.9%	40.7%	23.2%
Log rotation batch	9.9%	25.1%	37.3%	27.7%
Database reorganization	14.6%	34.1%	34.1%	17.2%
Garbage collection	13.2%	41.1%	31.8%	13.9%
Network backup	6.7%	28.9%	39.3%	25.2%

Observations:

1. High CPU load was the most common overall (97.4%) and by far the most likely cause of web performance degradation to be encountered often (38.3% vs. 28.9% for its nearest competitor, memory exhaustion/paging).

   In future surveys, we will distinguish explicitly between client-side and server-side CPU load, but given these numbers, we interpret most responses as referring mainly to server-side CPU load. This is interesting because we expected I/O bottlenecks to rank highest — in fact, they come in as third most common to have been encountered often (25.1%).

   The fact that CPUs are much more likely than I/O to cause web application performance degradation suggests very rapid adoption of high-performance non-volatile memory (e.g., NVMe) and the ability of I/O systems to take advantage of modern persistence hardware, which may in turn, follow from high adoption of cloud/managed server hardware (given the cost of upgrading self-maintained datacenter hardware).

2. Further, it seems that high CPU load is responsible for web application performance degradation due to genuinely large amounts of processing work. CPU thrashing, which might indicate some misuse of resources (e.g., threading at the wrong level, poor memory alloc/dealloc), was reported as far less likely to degrade web performance often (only 15.7% vs. 38.3% for high CPU load) or at all (89.7% vs. 97.4% for high CPU load).

3. Geographic location lag was the least likely factor to cause web performance degradation ever (71.2%) and among the least likely to cause web performance degradation often (11%). The Internet works; BGP is doing a good job.

4. Excessive algorithmic complexity had a fatter response curve than we expected: Almost a quarter (22%) of respondents attributed web performance degradation to algorithmic complexity often, while 39% reported sometimes and 30.8% reported rarely.

   If algorithmic complexity is responsible for a quarter of web performance degradation incidents, then we might suppose that static code analysis and/or more low-level code reviews might significantly improve web performance. However, since our data does not indicate how *severe* the performance degradation from each cause in each instance was, we cannot draw any conclusions about the overall impact of algorithmic improvements on web performance. And since time complexity can easily vary by many orders of magnitude across algorithms, we expect that calculations based on *mean* effect of big-O fails will not be very useful.

   In future research, we may attempt to combine our survey data with empirical data from open-source static code analysis and commit messages. But we are not yet sure how to factor in the potentially massive variance in performance degradation effect caused by poor algorithm design without careful manual analysis (thanks, Entscheidungsproblem).

HIGH-LEVEL CAUSES OF POOR SOFTWARE PERFORMANCE IN GENERAL

In addition to (web) platform-specific root causes, we wanted to know, at a higher level, where blame for poor performance lies. This is, of course, an objective question, but given there are many complex causes, we decided to piggyback on software professionals' judgment to figure out what makes software slow. We asked:

Overall, I blame the following factors for poor performance of the software I have worked on — rank from "blame most" (top) to "blame least" (bottom): {Bad code that others wrote | Bad code that I wrote | Database misconfiguration | Network issues | Insufficient memory | Slow I/O | Slow CPU | Slow disk read/write | Slow GPU}

Results (n=252):

Observations:

1. Bad code is by far the most (recognized as) to blame for poor performance. This holds true across all code-producing or code-adjacent respondents — developers, developer team leads, and architects.

2. Architects blame bad code even more than developers. The score gap between:

   - "Bad code that I wrote" and the next-highest-scored cause, "database misconfiguration," is larger for architects than for developers.
   - "Bad code that others wrote" and "bad code that I wrote" is larger for architects than for developers.

   Whether this is because architects write better code, or simply write less code, cannot be determined from this data; in future surveys, we may split "others" into "other developers" and "other architects."

Table 1

HIGH-LEVEL CAUSES OF POOR SOFTWARE PERFORMANCE

Performance Factors	Rank	Score	n=
Bad code that others wrote	1	2,085	252
Bad code that I wrote	2	1,709	225
Database misconfiguration	3	1,524	233
Network issues	4	1,419	203
Insufficient memory	5	1,391	214
Slow I/O	6	1,301	192
Slow CPU	7	1,263	191
Slow disk read/write	8	1,050	187
Slow GPU	9	658	139
Other	10	274	101

3. Ranking of blame is remarkably similar across all code-adjacent respondents with only one notable exception: slow CPU.

   Developers ranked slow CPU as the sixth highest cause, while architects ranked slow CPU as eighth highest. We conjecture that this is accounted for mainly by developers working "closer to the CPU" than architects, but this inference is weakened by the (presumed) likelihood that technical architects may be assigned their role due to high level of programming skill.

Research Target Two: Organizational Practices Aimed at Performance and Reliability

Motivations:

1. Pain caused by poor performance is not intrinsically assigned to specific roles. This is different from, for example, code readability: Developers suffer more from lack of code readability (or at least more immediately) than PMs or architects who wrangle code less. Contrast performance:
    - Some developers take much more pride in performance than others; performance is a "harder" metric than code readability, for instance (and craftsperson-ego is better serviced by 'hard' metrics).
    - Architects may or may not be impacted, at high variance of degree of impact, by performance considerations.
    - Functional requirements, acceptance criteria for individual user stories, or whatever task-definitional equivalent may or may not contain performance constraints.

    Assignment of responsibility for performance, therefore, seems subject to a nontrivial degree of choice — and hence, a useful target for survey-based inquiry. We wanted to understand how these semi-arbitrary assignments are made.

2. Whether optimization is, in reality, premature does not generally determine when performance optimizations are considered — in no small part because (as Hoare is noting) the on-the-ground reality is hard to know at build time. In practice, management decisions — including service-level objectives driven by business requirements, product, or marketing — greatly determine when and how much application performance is considered. We wanted to know where performance enters the SDLC.

3. Some organizations have teams dedicated to responding to performance degradation in production, or even teams dedicated to performance engineering at all stages of the SDLC. In contrast, other organizations may throw a dashboard at a non-specialist developer, SRE, or sysadmin; others lack organizationally defined systems for addressing performance. We wanted to understand how these approaches distribute across organizations.

APPROACHES TO IDENTIFYING PERFORMANCE PROBLEMS

Because performance is a quantitative attribute of a design (e.g., two designs may produce the same outputs but at different rates), many performance problems are not immediately obvious from reviewing acceptance criteria or running unit tests. We wanted to know how software professionals in fact identify performance problems, so we asked:

How often do you identify a performance problem using the following approaches? Rank from most often (top) to least often (bottom). {Log monitoring | Customer/end-user reporting/feedback | Server monitoring | Infrastructure monitoring | Crash reporting | Audits | Intelligent alerting | End-user monitoring | Transaction tracing | Deployment tracking | Manual instrumentation | Synthetic transactions | Browser-synthetic monitoring | Not sure}

Results (n=250):

Table 2

APPROACHES TO IDENTIFYING PERFORMANCE ISSUES			
Method	**Rank**	**Score**	**n=**
Log monitoring	1	2,766	244
Customer/end-user reporting/feedback	2	2,686	246
Server monitoring	3	2,677	250
Infrastructure monitoring	4	2,390	229
Crash reporting	5	1,745	204
Audits	6	1,632	189
Intelligent alerting	7	1,591	190

Table 2 (cont'd)

APPROACHES TO IDENTIFYING PERFORMANCE ISSUES			
Method	**Rank**	**Score**	**n=**
End-user monitoring	8	1,553	191
Transaction tracing	9	1,497	195
Deployment tracking	10	1,497	184
Manual instrumentation	11	1,418	173
Synthetic transactions	12	1,280	166
Browser-synthetic monitoring	13	980	156
Not sure	14	211	73

Observations:

1. Log monitoring — the simplest, most immediate, most manual way to identify performance problems — is the highest-scoring method of identifying performance problems. Further, log monitoring (and its kin, server monitoring) are rarely ranked low (i.e., the rank distribution is skewed right), meaning that most respondents use log monitoring often.

   Because logs facilitate performance tuning at any stage of the SDLC, the popularity of log monitoring suggests some continuity between how software professionals approach performance in any environment.

2. However, rankings differed significantly between senior and junior respondents. Among junior respondents, customer/end-user reporting/feedback is, by a slight margin, the highest-scoring performance issue identification method. Next is log monitoring and server monitoring, while among senior respondents, log monitoring ranks highest, followed by server monitoring and customer reporting/feedback — though somewhat more distantly.

   We take this difference as a reflection of the tendency to assign support tickets to junior developers, which we observe anecdotally but ubiquitously.

3. More specialized and sophisticated performance issue identification methods scored considerably lower than simpler and more generic methods. Worth noting, however, is another difference between senior and junior respondents: While senior respondents ranked manual instrumentation as the eleventh most common method, junior respondents ranked manual instrumentation as seventh.

   This may reflect difference in experience with new tools, which is partly a function of the learning overhead. Junior developers must learn a lot about many things very quickly, so they may not have the capacity to learn a more general-purpose tool than they need to track a specific performance metric. Senior developers are under less intrinsic pressure to learn as much as quickly, so they may have more time to learn a general-purpose tool, in addition to the greater likelihood that they have encountered these tools before.

4. Unsurprisingly, Java-primary developers ranked log monitoring higher (No. 2) than JavaScript-primary developers (No. 5). The difference may be compensated for by higher usage of audits reported by JavaScript-primary respondents (No. 4) vs. Java-primary respondents (No. 7) — an active intervention that makes up for the lack of centralized log availability common in (at least client-side) JavaScript applications.

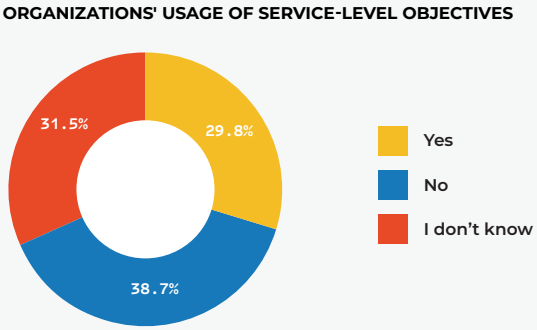
SERVICE-LEVEL OBJECTIVES

If the performance goal is something vague like "better performance," then effort spent on performance optimization is a matter of personality and leisure time — i.e., not forced from the outside. But if a performance goal ties to the business requirements, software professionals must explicitly consider performance before release. We wanted to see how often organizations set service-level objectives (SLOs) and what these SLOs are, so we asked:

Does your organization have any service-level objectives (SLOs)?

Results (n=302):

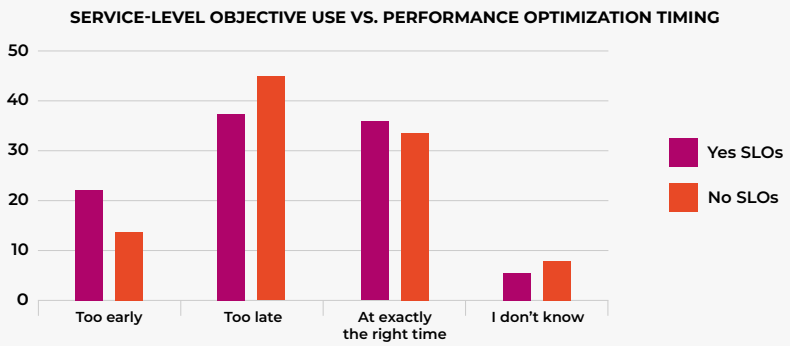
Figure 7



Observations:

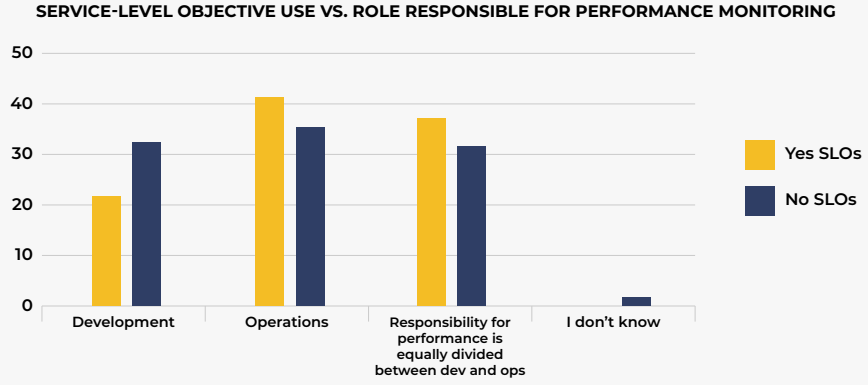
1. Almost a third of respondents (31.5%) did not know whether their organization has any SLOs. This, at the very least, means that SLOs do not contribute to respondents' decision-making.
2. The largest portion of respondents reported positive knowledge that their organization does not have any SLOs (38.7%). Insofar as SLOs improve performance (and whether they do or not is another question, discussed below), this high number suggests that the industry has room for improvement with respect to setting performance-related goals.
3. SLOs appear to impact perception of performance optimization timing. Respondents at organizations that set SLOs are more likely to report that their organization optimizes performance too early and less likely to report that their organization optimized performance too late, while the inverse is true of respondents at organizations that set no SLOs:

Figure 8



4. Similarly, the presence/absence of SLOs impacts where responsibility for monitoring application performance is located within the organization:

Figure 9



Research Target Three: Measuring Application Performance

Motivations:

1. As the bean-counters *er* expensive business consultants *er* scientific statisticians say: A goal unmeasured is a goal in name only. So we wanted to know how often software performance is measured.
2. As the cynical engineers *er* put-upon workers *er* anti-Taylorist management theorists say: A goal measured is a goal sought irrespective of actual desirability of the thing for which the metric is a proxy. So we wanted to know how the use of specific metrics related to assignment of blame for performance issues.
3. Because measurement itself is a nontrivial task, performance measurement may be sought or avoided proportionally to the pain or pleasure of measuring. Performance monitoring tools exist on spectra of features, breadth of target systems, hosting requirements, complexity, support, etc. We wanted to know how often such tools along these spectra are used.

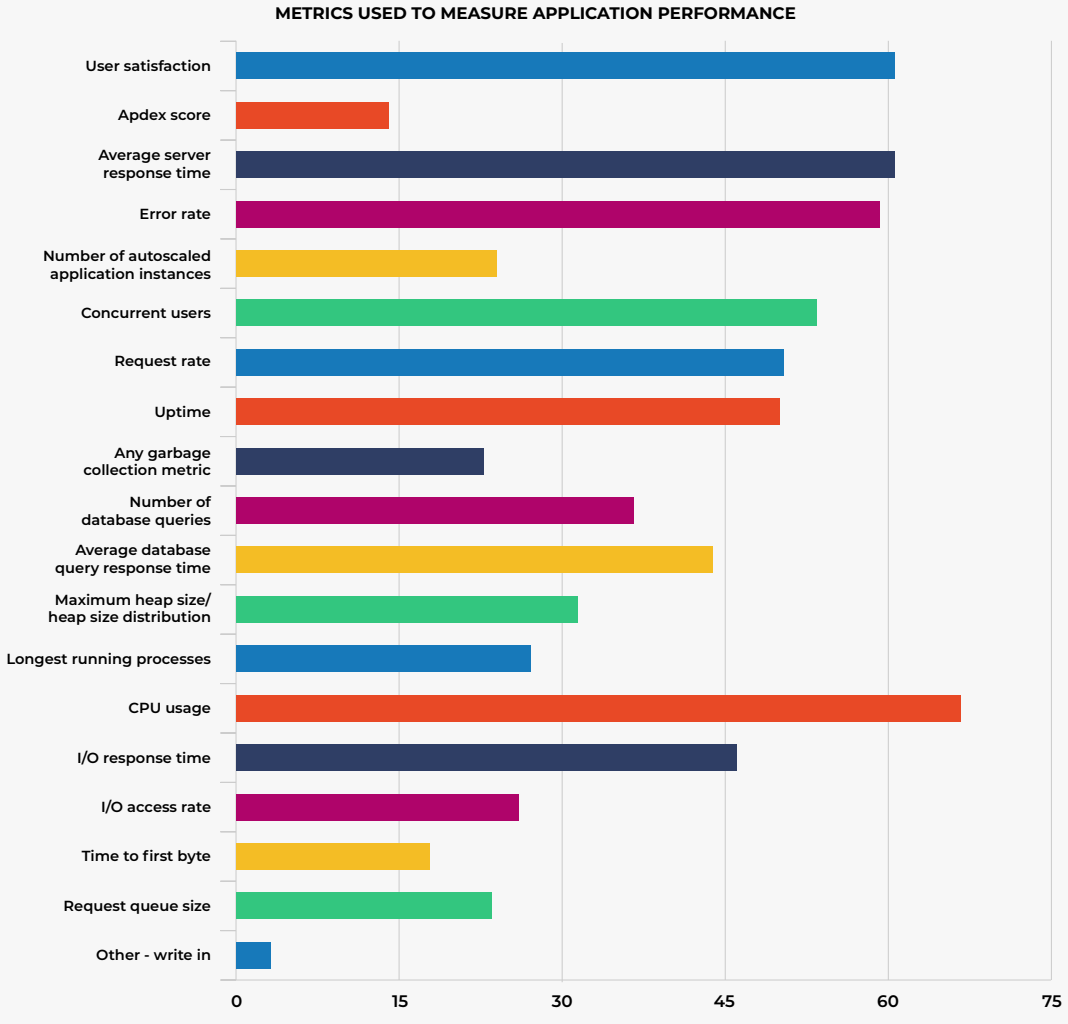
METRICS USED TO MEASURE APPLICATION PERFORMANCE

We wanted to get a baseline empirical sense of what metrics are used to measure application performance across a wide range of measurable objects and metric atomicity, so we asked:

Which of the following metrics does your organization use to measure application performance? {User satisfaction | Apdex score | Average server response time | Error rate | Number of autoscaled application instances | Concurrent users | Request rate | Uptime | Any garbage collection metric | Number of database queries | Average database query response time | Maximum heap size/heap size distribution | Longest running processes | CPU usage | I/O response time | I/O access rate | Time to first byte | Request queue size | Other (write in)}

Results:

Figure 10



Observations:

1. Technical metrics are more common than non-technical metrics overall. CPU usage is the most commonly reported metric (65.7%), average server response time (61.8%) and user satisfaction (61.4%) effectively tied for second place, and error rate (60.5%) third.

2. As with the elevated identification rate of high CPU usage as a root cause of web performance problems, we were mildly surprised about the prevalence of attention paid to CPU usage as a performance metric. Perhaps some or all of the following might account for this result:

    - It is a relatively intelligible metric (or thought to be — modern predictive-pipelined multicore architectures are harder to reason about w.r.t. utilization metrics).
    - It is a common cost center in commodity compute clouds.
    - It is commonly noted, but not weighted heavily, in performance analyses (our survey did not ask for weighting of each metric).
    - CPU-feeding subsystems (main memory, memory and I/O buses, non-volatile storage) really have caught up with processor speeds.

3. Performance metrics reported by developer and architects correlate rather strongly with a few notable exceptions: Architects are significantly more likely than developers to report tracking average server response time, concurrent users, uptime, CPU usage, I/O response time, and time to first byte.

    These specific metrics are what we might expect from their job descriptions, with the possible exception of CPU usage (where the difference is comparatively small), for reasons noted above: that developers are more likely to live "close to the CPU" than architects.

4. Developer team leads are more likely to report tracking database query response time, maximum heap size/heap size distribution, and I/O response time than developers and architects — with the exception of architects for I/O response time, where response rates are nearly identical.

    These metric-tracking specializations suggest the sort of developer team leads who function as deep technical experts worth consulting on tricky low-level matters, rather than the sort who function as technically inclined project managers. In future surveys, we intend to distinguish kinds of work done by developer team leads more precisely.

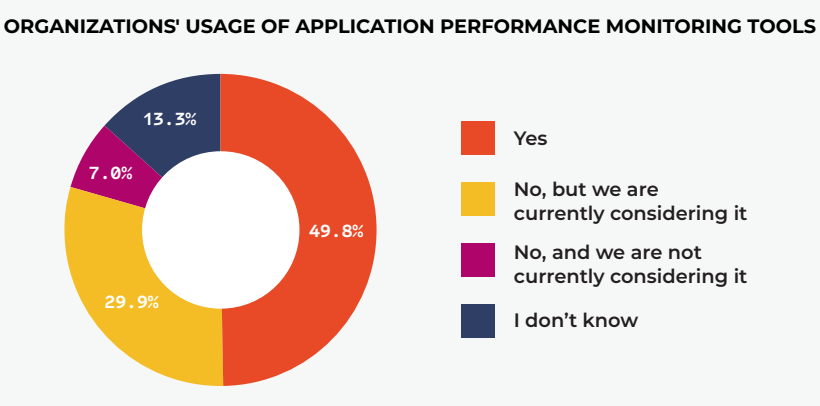
USAGE OF APPLICATION PERFORMANCE MONITORING TOOLS

As discussed above, log monitoring is the most common method for identifying performance problems, but does this mean grepping huge ASCII files manually? Or what? And how do simple methods like this interact with the more advanced methods that are less, but far from, negligibly popular? These questions boil down to: How often are specialized application performance monitoring tools used? To approach this question, we asked:

Is your organization currently using any application performance monitoring tool(s)? {Yes | No, but we are currently considering it | No, and we are not currently considering it | I don't know}

Results (n=301):

Figure 11



Observations:

1. Half of respondents (49.8%) reported using application performance monitoring tools. This is impressive: It seems that application performance monitoring (APM) has reached a significant level of market penetration — perhaps higher than the high ranking of manual issue identification methods discussed above might suggest. Note, of course, that APM tools normally include modules for tasks like log monitoring and crash reporting, so APM users are also typically monitoring logs, etc.

2. Easily the second largest single response bucket is "no, but we are currently considering it," further suggesting that the scuttlebutt on APM tools is positive enough to merit a high level of consideration among those who do not currently use any APM tool.

3. The percent of respondents who reported they are not considering using APM tools is extremely low (7%). This may indicate that manual performance monitoring is painful enough that few are uninterested in taking dedicated steps to handle monitoring in a more automated manner.

Further Research

Application performance is a vast topic that touches all aspects of software development and operations, requires expertise in the most mathematically oriented and most physically oriented aspects of computer science and engineering, often elicits philosophical and ethical debates during technical decision-making processes, and yet noticeably impacts (and annoys) end users (human or machine) densely *in perpetuum*.

We have barely begun to address this topic in our research, but we should note that the present survey included questions whose analyses we did not have space to present here, including:

- Percent of application performance problems solved without satisfactory root cause identification
- Percent of application performance problems solved later than preferred
- Location of performance monitoring in the SDLC
- Responsibility for monitoring application performance within organizations
- Use of AI for application performance monitoring

We intend to analyze this data in future publications. If you are interested in this data for research purposes, please contact publications@dzone.com and we may be able to share, depending on your research proposal.

John Esposito, PhD, Technical Architect at 6st Technologies
@subwayprophet on GitHub | @johnesposito on DZone

John Esposito works as technical architect at 6st Technologies, teaches undergrads whenever they will listen, and moonlights as research analyst at DZone.com. He wrote his first C in junior high and is finally starting to understand JavaScript NaN%. When he isn't annoyed at code written by his past self, John hangs out with his wife and cats Gilgamesh and Behemoth, who look and act like their names.

In-Stream Data Analytics for Modern Observability

An open, fully-managed SaaS platform that provides a single, aggregated view of system health.

Coralogix leverages advanced streaming technology to produce real-time insights and trend analysis for log, metric, and security data without relying on storage or indexing.

This unique approach to monitoring and observability enables organizations to overcome the challenges of exponential data growth in large-scale systems.

Using proprietary Streama© technology, Coralogix provides modern engineering teams with everything they need to monitor system health and proactively resolve issues without needing to index the data.

As data is ingested, millions of events are automatically narrowed down to common patterns for deeper insights and faster troubleshooting.

Machine learning algorithms continuously observe data patterns and flows between system components and trigger dynamic alerts so you know when a pattern deviates from the norm without static thresholds or the need for pre-configurations.

Connect any data, in any format, and view your insights anywhere including our purpose-built UI, Kibana, Grafana, SQL clients, Tableau, or using our CLI and full API support.

Key Benefits

PERFORMANCE

Our unique streaming architecture gives users the benefits of stateless speed and scale with the power and granularity of stateful correlation.

SCALE

Using advanced auto-scaling techniques, Coralogix seamlessly scales up and down to meet the demands of any environment at any scale.

COST OPTIMIZATION

By analyzing data and extracting insights without needing to store or index it, users benefit from complete monitoring, visualization, and alerting capabilities with storage savings of up to 70%.

We ship all of our logs to Coralogix and turn them into metrics. Then we can monitor them all in a dashboard and pinpoint exact problems.

Roi Amitay - Head of DevOps

ARMIS

SPONSOR OPINION

Unlikely Benefits of Cloud-Native Observability

Tali Soroker, Product Marketing Manager at Coralogix

Choosing new cloud-native approaches to building and deploying software allows for better agility and flexibility, although it does demand that we update our monitoring approach to ensure we don't lose visibility.

Implementing and managing observability for distributed, cloud-native applications at scale is more costly and less effective with traditional tooling. New containerized environments are running a more diverse set of technologies and configurations, and generate higher data volumes faster. The nature of cloud-native applications means not only more data, but more complex data flows coming from more varied data sources.

When we overcome observability challenges such as coverage, correlation, and cost in our distributed systems, we can look forward to unlocking benefits beyond keeping customers happy.

Improved Build Processes

Most companies moving to cloud-native approaches lack the visibility to identify and pinpoint problems not only within their applications but within their build tools and CI/CD processes.

Increased agility promised by the move to containerized environments requires better observability in order to be fully realized. As more features and services are being introduced, together with increased users, the volume of event data being generated grows exponentially.

Companies that are shipping comprehensive logs from across their systems and workflows are better able to pinpoint exact problems not only within their application, but within their build pipelines. One way to do this is to convert log data to metrics which can then be monitored in a centralized dashboard.

Better End-to-End Security

With cloud-native, services are more susceptible to attacks compared to their monolithic predecessors. It's critical to secure data centers and services holistically rather than focusing solely on endpoints. Detecting an attack also requires tools as dynamic as your infrastructure to ensure that every part is observable.

Cloud-native observability tools are designed to help companies monitor typical behavior within their applications and alert on any anomalous behavior. This can enable teams to identify where they may be vulnerable and to detect an attack.

Scale, Scale, Scale

Scalability is not an entirely unlikely benefit — given it's one of the major selling points of going cloud-native — but perhaps an underrated outcome of a winning observability strategy.

Scaling capacity as usage increases and decreases allows businesses to be very cost-efficient, paying only for the storage and compute that they use. It can also allow them to allocate private servers according to what is needed at that time, scaling back capacity for computing that is not time-critical.

Developers can use observability data to fine-tune capacity settings to further increase efficiency. For example, Armis Security has scaled 7X in the past 3 years using Coralogix as their observability platform. As their systems grow and generate more data, they maintain good coverage and are able to pinpoint issues and constantly make improvements.

CONTRIBUTOR INSIGHTS

OpenTelemetry Moves Past the Three Pillars

Why a Single Braid of Data Will Power the Future of Observability

By Ted Young, Director of Developer Education at Lightstep

Last summer, the OpenTelemetry project reached the incubation stage within the Cloud Native Computing Foundation. At the same time, OpenTelemetry passed another mile marker: over 1,000 contributing developers representing over 200 different organizations. This includes significant investments from three major cloud providers (Google, Microsoft, and AWS), numerous observability providers (Lightstep, Splunk, Honeycomb, Dynatrace, New Relic, Red Hat, Data Dog, etc.), and large end-user companies (Shopify, Postmates, Zillow, Uber, Mailchimp, etc.). It is the second largest project within the CNCF, after Kubernetes.

What is driving such widespread collaboration? And why has the industry moved so quickly to adopt OpenTelemetry as a major part of their observability toolchain? This article attempts to answer these questions, and to predict several important trends that will continue to gain momentum as OpenTelemetry stabilizes over the next year.

OpenTelemetry Overview

OpenTelemetry is a *telemetry* system. This means that OpenTelemetry is used to generate *metrics*, *logs*, and *traces*, and then transmits them to various storage and analysis tools. OpenTelemetry is not a complete observability system, meaning it does not perform any long-term storage, analysis, or alerting. OpenTelemetry also does not have a GUI. Instead, OpenTelemetry is designed to work with every major logging, tracing, and metrics product currently available. This makes OpenTelemetry vendor-neutral: It is designed to be the telemetry pipeline to any observability back end you may choose.

The goal of the OpenTelemetry project is to standardize the language that computer systems use to describe what they are doing. This is intended to replace the current babel of competing vendor-specific telemetry systems, a situation which has ceased to be desirable by either users or providers. At the same time, OpenTelemetry seeks to provide a data model that is more comprehensive than any previous system, and better suited for the needs of machine learning and other forms of large-scale statistical analysis.

OpenTelemetry Project Structure

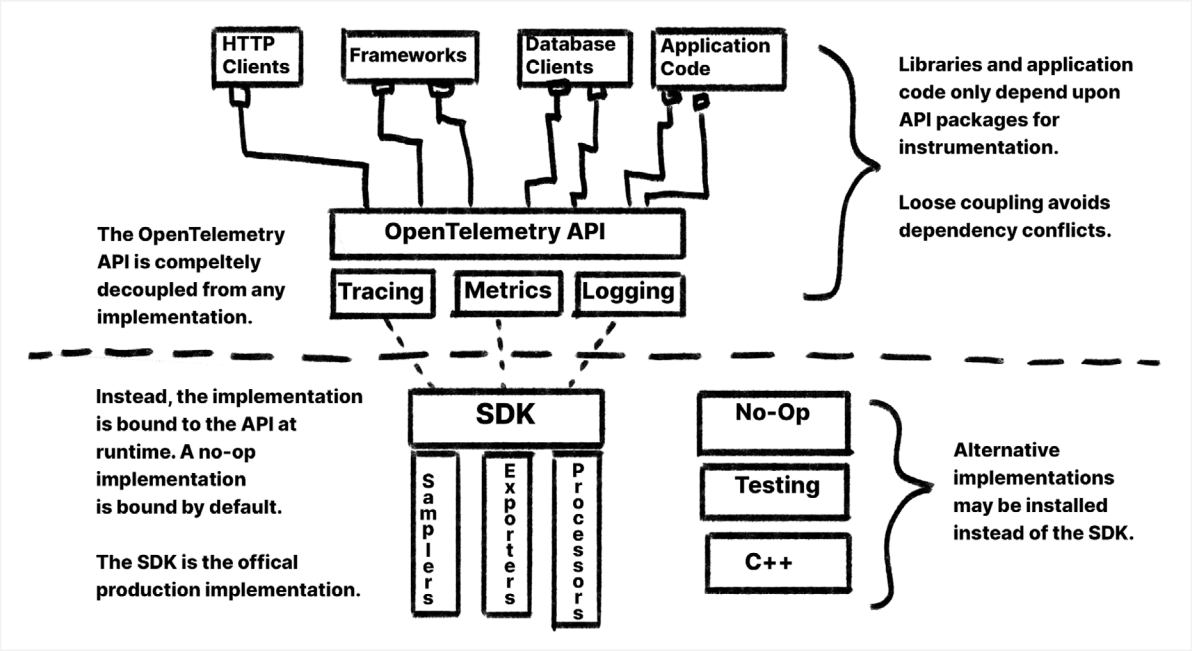
The OpenTelemetry architecture is divided into three major components: the *OpenTelemetry Protocol* (OTLP), the *OpenTelemetry Collector*, and the *OpenTelemetry language clients*. All these components are designed via a specification process, which is managed by the *OpenTelemetry Technical Committee*. The features described in the specification are then implemented in OTLP, the Collector, and various languages via *Special Interest Groups* (SIGs), each of which is led by a set of maintainers and approvers. This project structure is defined by the *OpenTelemetry Governance Committee*, who work to ensure these rules and processes support and grow a healthy community.

OPENTELEMETRY CLIENTS

In order to instrument applications, databases, and other services, OpenTelemetry provides clients in many languages. Java, C#, Python, Go, JavaScript, Ruby, Swift, Erlang, PHP, Rust, and C++ all have OpenTelemetry clients. OpenTelemetry is officially integrated into the .NET framework.

(See Figure 1 on next page)

Figure 1



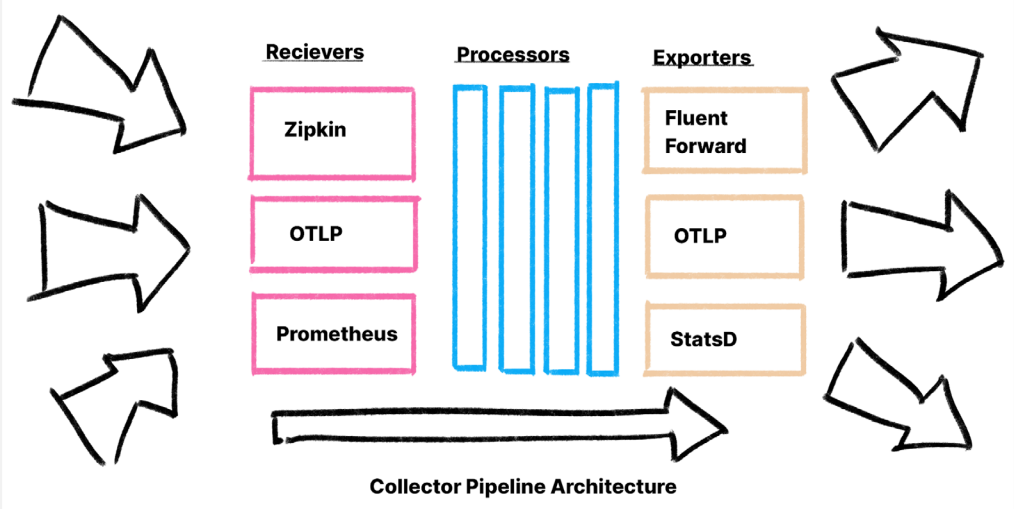
OpenTelemetry clients are separated into two core components: the *instrumentation API* and the *SDK*. This loose coupling creates an important separation of concerns between the developers who want to write instrumentation and the application owners who want to choose what to do with the data. The API packages only contain interfaces and constants and have very few dependencies. When adding instrumentation, libraries and application code only interact with the API. At runtime, any implementation may be bound to the API and begin handling the API calls. If no implementation is registered, a No-Op implementation is used by default.

The SDK is a production-ready framework that implements the API and includes a set of plugins for sampling, processing, and exporting data in a variety of formats, including OTLP. Although the SDK is the recommended implementation, using it is not a requirement. For extreme circumstances where the standard implementation will not work, an alternative implementation can be used. For example, the API could be bound to the C++ SDK, trading flexibility for performance.

THE OPENTELEMETRY COLLECTOR

The OpenTelemetry Collector is a stand-alone service for processing telemetry. It provides a flexible pipeline that can handle a variety of tasks: Routing, configuration, format translation, scrubbing, and sampling are just examples.

Figure 2



The Collector has a straightforward architecture. *Receivers* ingest data in a variety of formats, and over 45 data sources are currently supported. Receivers translate this data into OTLP, which is then handed to *processors*, which perform a variety of data manipulation tasks. Data can be scrubbed, normalized, and decorated with additional information from Kubernetes, host machines, and cloud providers. With processors, every common data processing task for metrics, logs, or traces can be accomplished in the same place.

After passing through processors, *exporters* convert the data into a variety of formats and send it on to the back end — or whatever the next service in the telemetry pipeline may be. Multiple exporters may be run at the same time, allowing data to be teed off into multiple observability back ends simultaneously.

OTLP

The OpenTelemetry Protocol (OTLP) is a novel data structure. It combines tracing, metrics, logs, and resource information into a single graph of data. Not only are all these streams of data sent together through the same pipe, but they are also interconnected. Logs are correlated with traces, allowing transaction logs to be easily indexed. Metrics are also correlated with traces. When a metric is emitted while a trace is active, trace exemplars are created, associating that metric with a sample of traces that represent different metric values. And all this data is associated with resource information describing host machines, Kubernetes, cloud providers, and other infrastructure components.

OpenTelemetry Is Driving Three Trends

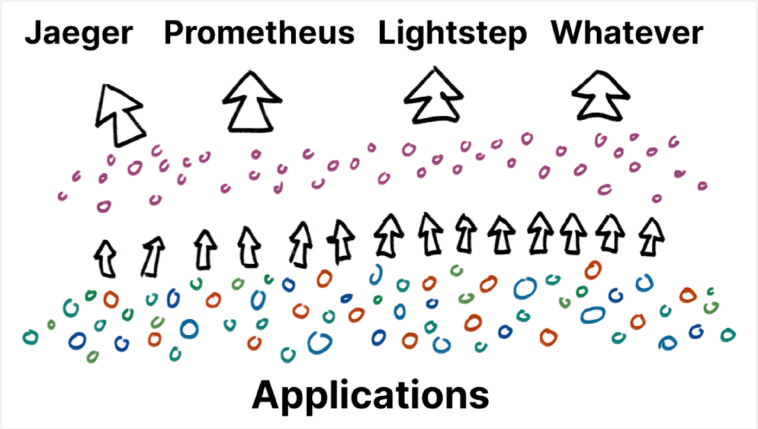
By design, industry adoption of OpenTelemetry is creating a number of benefits. In particular, there are three positive shifts that represent a significant change in the observability landscape.

SHARED TELEMETRY, COMPETING ANALYSIS

Traditionally, observability systems were vertically integrated. This is commonly referred to as the "Three Pillars" model of observability. For example, a new metrics product would develop a time-series database, and at the same time, create a protocol, a client for writing instrumentation, and an ecosystem of instrumentation plugins for binding to common web frameworks, HTTP clients, and databases.

This vertical integration created a form of vendor lock-in. Users could not switch to a different observability tool — or even try one out, really — without first ripping and replacing their entire telemetry pipeline. This was incredibly frustrating for users who want to be able to try out new tools and switch providers whenever they like. At the same time, every vendor had to invest heavily in providing (and maintaining) instrumentation for a rapidly expanding list of popular software libraries, a duplication of effort that took valuable resources away from providing the other thing that users want — new features.

Figure 3



OpenTelemetry rearranged this landscape. Now, everyone is working together to create a high-quality telemetry pipeline, which they share. It is possible to share this telemetry pipeline because, generally, we can all agree upon a standard definition of an HTTP request, a database call, and a message queue. Subject matter experts can participate in the standardization process, along with providers and end users. Through this broad combination of different voices, we create a shared language that meets everyone's needs.

At the same time, the OpenTelemetry project has a clear scope and attempts to define any sort of standard analysis tool or back end. This is because there is no standard definition for "good analysis." Instead, analysis is a rapidly growing field with new promising techniques appearing every year.

OpenTelemetry makes it easier than ever to develop a novel analysis tool. Now, you only have to write the back end. Just build a tool that can analyze OTLP, and the rest of the system is already built. At the same time, it is easier than ever for users to try these new tools. With just a small configuration change to your collector, you can begin teeing off your production data to any new analysis system that you would like to try.

NATIVE INSTRUMENTATION FOR OPEN-SOURCE SOFTWARE

Modern applications heavily leverage open-source software, and much of the telemetry we want comes from the shared libraries our systems are built out of. But for the authors of these open-source libraries, producing telemetry has always been onerous. Any instrumentation chosen was designed to work with a specific observability tool. And since the users of their libraries all use different observability tools, there was no right answer.

This is unfortunate because library authors *should* be the ones maintaining this instrumentation — they know better than anyone what information is important to report and how that information should be used. Instrumenting a library should be no different from instrumenting application code.

Instrumentation hooks, auto-instrumentation, monkey patching — all the things we do to add third-party instrumentation to libraries are bizarre and complicated. If we can provide library authors with a well-defined standard for describing common operations, plus the ability to add any additional information specific to their library, they can begin to think of observability as a practice much like testing: a way to communicate correctness to their users.

If library authors get to write their own instrumentation, it becomes much easier to also write playbooks that describe how to best make use of the data to tune and operate the workload that library is managing. Library author participation in this manner would be a huge step forward in our practice of operating and observing systems.

Library authors also need to be wary about taking on dependencies that may risk creating conflicts. The OpenTelemetry instrumentation API never breaks backwards compatibility and will never include other libraries, such as gRPC, which may potentially cause a dependency conflict. This commitment makes OpenTelemetry a safe choice to embed in OSS libraries, databases, and complex, long-lived applications.

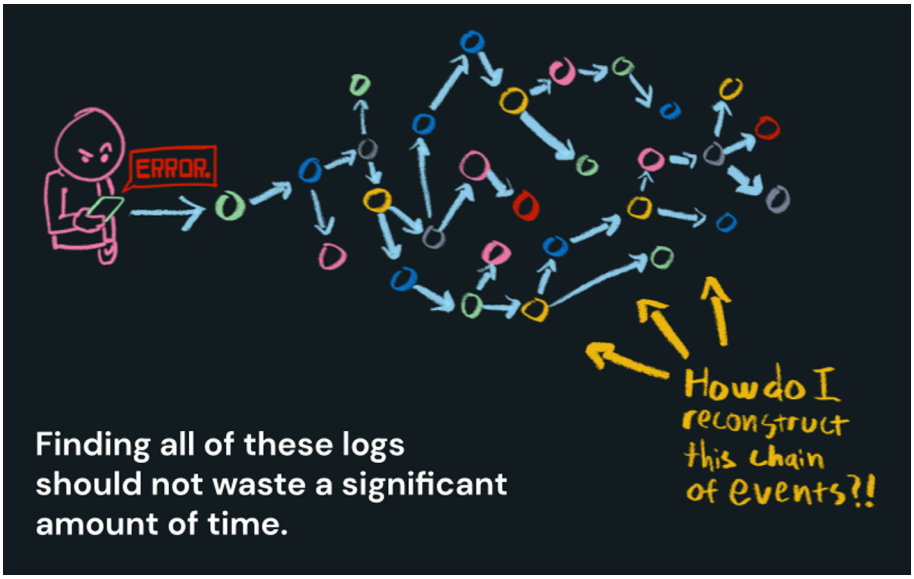
INTEGRATED LOGGING, METRICS, AND TRACING FOR AUTOMATED ANALYSIS

The biggest and most profound shift for operators will come in the form of the new, advanced analysis tools and integrated workflows that can be built from OpenTelemetry data. While the standardization and dedication to stability allow OpenTelemetry to go where no telemetry system has gone before, it is the unified structure of OpenTelemetry's data that makes it truly different from other systems.

Figure 4

A side effect of the vertically integrated "Three Pillars" approach was that every data source was siloed; the traces, logs, and metrics had no knowledge of each other. This has had a significant impact on how time consuming and difficult it is to observe our systems in production, but it is an impact that is difficult to realize given how prevalent it is. Much like how auto-formatting makes us realize how much time we were spending hitting the tab key and hunting for missing semicolons, integrated data makes us realize how much time we were spending doing that same integration by hand.

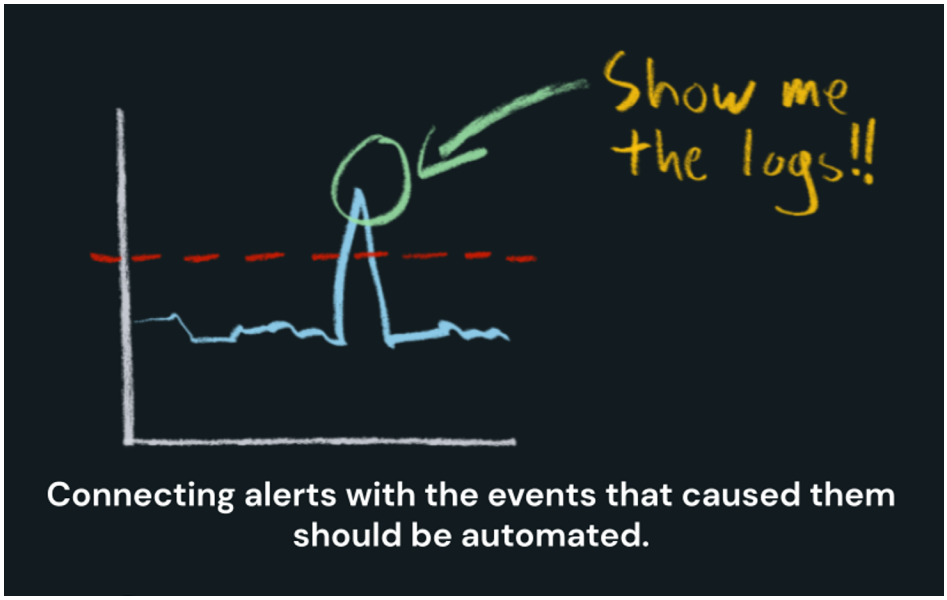
Figure 5



For example, one of our most common observational activities is looking at logs. When something goes wrong with a user request, we want to look at all of the logs related to that request from the browser, the front-end application, back-end applications, and any other services that may have been involved. But gathering these logs can be tedious and time consuming. The more servers you have, the harder it is to find the logs for any particular request. And finding the logs when they stretch across multiple servers can require some trickery.

However, there's a simple solution to this problem. If you are running a tracing system, then you have a trace ID available to index all of your logs. This makes finding all of the logs in a transaction quick and efficient — index your logs by trace ID, and now a single lookup finds them all!

Figure 6



But it isn't just tracing and logs that save time when they are integrated. As mentioned before, when describing OTLP, metrics are associated with traces via exemplars. Looking up logs is often the next step after looking at dashboards. But again, when our data is siloed, we have no way to automate this; instead, we have to guess and hunt around by hand until we find logs that are *possibly* relevant. But if our logs and metrics are directly connected by trace exemplars, looking at logs relevant to a dashboard or an alert is just a single click.

These time savings are compounded further when you consider what it is we are often looking for — correlations. For example, extreme latency may be highly correlated with traffic to a particular Kafka node — perhaps that node was misconfigured. A huge, overwhelming spike in traffic may be highly correlated with a single user. That situation requires a different response to a traffic spike caused by the sudden rush of new users.

Sometimes, it can get really nuanced. Imagine that an extremely problematic error only occurs when two services — at two particular versions — process a request with a negative value in a query parameter. Worse, if the error doesn't actually occur when those two services talk, it occurs later when a third service encounters a null in the data and the second service is handed back. Long, sleepless nights spin themselves from hunting down the source of these types of problems.

This is probably the biggest takeaway from this article: Machine learning systems are great at finding these correlations, *provided they are fed high-quality, structured data*. Using OpenTelemetry's highly integrated stream of telemetry, quickly identifying correlations becomes very feasible, even when those correlations are subtle or distantly related.

Significant time savings through automated analysis — this is the value that OpenTelemetry will ultimately unlock.

When Will OpenTelemetry Become Mainstream?

The current trend in observability products has been to expand beyond focusing on a single data type to becoming "observability platforms," which process multiple data sources. At the same time, many of these products have begun to either accept OTLP data or announce the retirement of their current clients and agents in favor of OpenTelemetry.

Continuing this trend into the future paints a fairly clear picture. In the first two quarters of 2022, all of the remaining OpenTelemetry core components are expected to be declared stable, making OTLP an attractive target for these products. By the end of next year, features that leverage this new integrated data structure will begin appearing in many observability products. At the same time, a number of databases and hosted services will begin offering OTLP data, leveraging OpenTelemetry to extend transaction-level observability beyond the clients embedded in their users' applications, deep into their storage systems and execution layers.

Access to these advanced capabilities will begin to motivate mainstream application developers to switch to OpenTelemetry. At this point, OpenTelemetry and modern observability practices will begin to cross the chasm from early adoption to best practice. This, in turn, will further motivate the development of features based on OTLP, and motivate more databases and managed services to integrate with OpenTelemetry.

This is an exciting shift in the world of observability, and I look forward to the advancement of time-saving analysis tools. In 2022, keep your eye on OpenTelemetry, and consider migrating as soon as it reaches a level of stability you are comfortable with. This will position your organization to take advantage of these new opportunities as they unfold.

Ted Young, Director of Developer Education at Lightstep

@tedsuo on DZone | @tedsuo on Twitter | @LightStepHQ on YouTube

Ted Young is one of the co-founders of the OpenTelemetry project. With 20 years of experience, he has built distributed systems in a variety of environments, from visual fx to container scheduling systems. He currently works as Director of Developer Education at Lightstep.

CONTRIBUTOR INSIGHTS

Making APM a Company-Wide Effort

Joana Carvalho, Performance Engineer at Postman

Today, more than ever, users are unwilling to wait and tolerate failure. Nearly 50 percent of users expect a load time of less than two seconds. Hyperconnectivity has become the new status quo, and with it comes higher pressure on the industry to provide the best service possible. This has also transformed the software application landscape into an intricate net of components — from APIs to CDNs — each of which can easily become the weak link when a problem occurs, leading to poor customer experiences and unhappy end users.

Teams of developers, product owners, test engineers, etc. must work more closely and seamlessly than ever before to solve these issues. This is where application performance management/monitoring (APM) can help manage expectations for performance, availability, and user experience. APM helps teams and companies understand these expectations by gathering software performance data and analyzing it to detect potential issues, alerting teams when baselines aren't met, providing visibility into root causes, and taking action to resolve issues faster (even automatically) so that the impact on users and businesses is minimal.

State-of-the-Art APM

When the first APM solutions were designed, the most common software architectures were different from what they are today — they were simpler and more predictable. Demand has driven applications to move from a monolithic architecture to a cloud-distributed one, which is often more complex and more challenging to manage and monitor without dedicated tools. This increased complexity has forced APM tools to seek new strategies and monitor a myriad of moving parts now present in the software stack.

In addition, over the last two years, social distancing due to the COVID-19 pandemic has forced a new shopping paradigm, and consumers — for safety and convenience — have turned to this new digital experience. In the US alone, in March 2020, 20-30 percent of the grocery business moved online, reaching a 9-12 percent penetration by the end of 2020 — and the forecast is to continue growing.

Figure 1

Countries like Germany and Switzerland, who are known for their preference to use physical currency, have massively turned to wired or contactless payments, either motivated by the restrictions from governments or to avoid unnecessary contact.

Not only has the pandemic changed the way we conduct ourselves in the world, but it also changed the way we interact with technology for our everyday needs. It is now imperative that software be more predictable and reliable than ever before, as it caters not only to our convenience but also to our safety.

APM will be crucial to fulfilling these requirements, giving DevOps teams insight into problem isolation and prioritization that consequently shortens MTTR (mean time to repair), hence, preserving service availability and experience. The increased demand by businesses to meet shorter time-to-market requirement roll-outs, the acceleration of cloud and containerized migrations, and the evolution of technology stacks all contribute to organizations' efforts to aggressively keep up with the new wave of users, which has increased the risk of service disruptions and delays.

The future is to combine observability with artificial intelligence to create self-healing applications. Together, with machine learning, real-time telemetry, and automation, it's possible to foresee application issues based on the system outputs and resolve them before they can have a negative impact. Further, machine learning will help determine motive, predict and detect anomalies, and reduce system noise.

Successful APM Adoption: A Shift in Culture

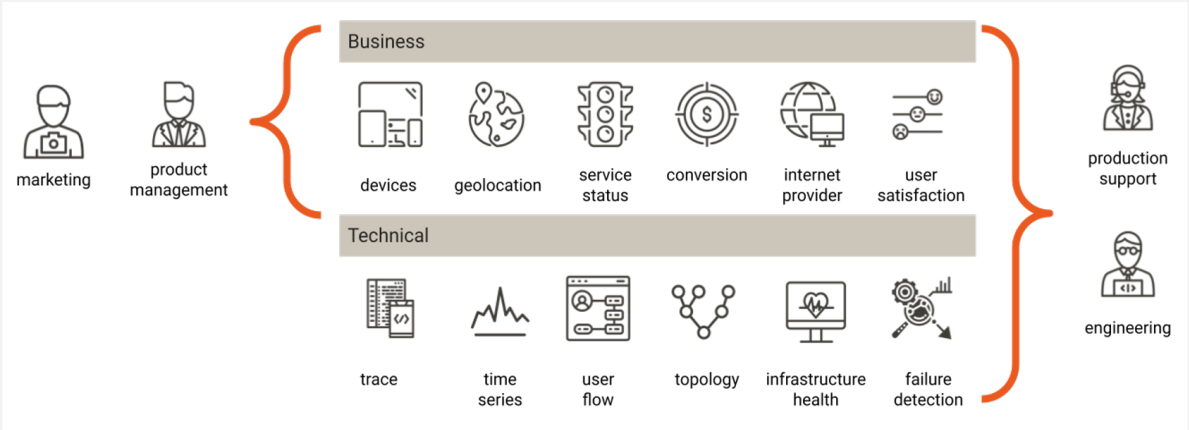
Monitoring and observability are a crucial part of the software development lifecycle. Not only do they help ensure user satisfaction, as well as prevent and detect anomalies and defects, but they also help inhibit the throw-over-the-wall mentality. This mentality is, in part, a result of observability and monitoring strategies often thought of as the responsibility of the Ops teams; therefore, they typically don't make it into the development cycle and become an afterthought.

Most companies only use observability and monitoring strategies in production environments, becoming challenging for the development and quality teams to fully get to know their application and take advantage of features like tracing, error prediction, etc. to mitigate defects during the development phase. This is why APM selection, implementation, and configuration should happen side by side with feature development early in the design process to guarantee the robustness of the solution.

Figure 2

The question to be answered is: Whose responsibility and ownership is it? As seen in the image below, the usages of APM solutions provide insight for several stakeholders.

Figure 3



All stakeholders should be involved in defining requirements, setting up expectations, and configuring the metrics and queries that will be responsible for creating the rules, budgets, and visualizations. It is not straightforward to only understand the metrics that should be examined more closely. Since all telemetry is stored and analyzed, and we are constantly bombarded with information, we must be frugal with the metrics chosen.

So what makes a good metric? Daniel Yankelovich summarizes some of the common errors that are made when measuring:

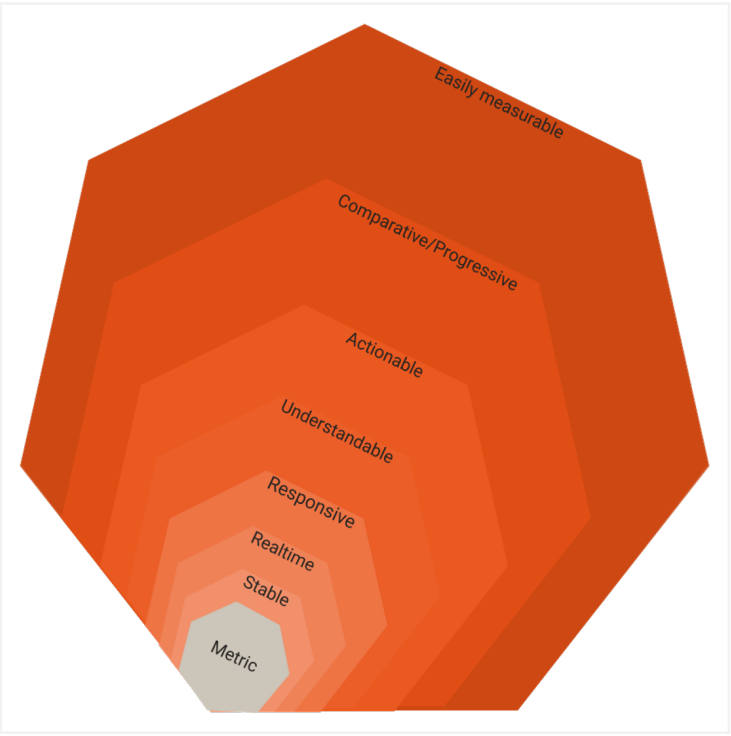
> "The first step is to measure whatever can be easily measured. This is OK as far as it goes. The second step is to disregard that which can't be easily measured or to give it an arbitrary quantitative value. This is artificial and misleading. The third step is to presume that what can't be measured easily really isn't important. This is blindness. The fourth step is to say that what can't be easily measured really doesn't exist. This is suicide."
>
> — Daniel Yankelovich, "Corporate Priorities: A continuing study of the new demands on business," 1972

CHARACTERISTICS OF A ROBUST METRIC

Metrics are used on a daily basis to support decisions and guide successful outcomes. For a metric to be effective and reliable, it should have the following characteristics:

- **Understandable** – When selecting a metric, anyone should be able to understand what each value means. If teams are not able to discuss a metric that they are tracking, it is meaningless.
- **Actionable** – The purpose of tracking a metric is to aid decisions being made. Tracking should always improve behaviors and motivate change. If a value goes over a threshold or a metric starts failing, it should be clear what the impact is and what must result from it.
- **Comparative/progressive** – The ability to compare a metric to other time periods or release versions, for example, can help understand progress, helping spot spikes or long-term trends. It is clearer reading "the throughput is 12 percent higher than last week" than "the throughput is 50 requests per second." The first conveys an improvement, and the second, without context, merely states the value.

Figure 4



- **Easily measurable, stable, and responsive** – If the effort spent to measure the metric is overwhelming and needs the implementation of new complex systems with the single purpose of measuring and computing that metric, it's probably not worth measuring in the first place. Basing decisions on a metric that is generated inside a black box is not ideal. The complexity of the solution needs to be balanced against the gain — or even lead to the selection of a more realistic metric. When implementing the metric, the future application should also be taken into account to accommodate the volume of data and ability to apply it throughout the platform.
- **Relevant** – It should align with teams and business objectives. All metrics should be tracked, if possible, but if creating custom metrics or collecting something that is not out of the box, vanity metrics are good for team morale, but they shouldn't be the totality.
- **Real-time** – The time to compute and collect a metric should not be so long that the time to display renders it obsolete.

These metrics, when carefully selected, can become the standards to communicate the status of the system with teams or stakeholders. They can also serve as a motivator when sharing good outcomes, for example, from technical or marketing initiatives.

Keep in mind: It is paramount to have the support at the executive level to guarantee the success of implementing APM systems as well as observability and monitoring techniques.

Embedding the power of application performance monitoring and AIOps, which are becoming an intertwined concept, teams can work together to implement them into their process and gain the ability to:

- Detect why applications are slow
- Enable end-to-end visibility in an automated way — with little intervention from teams
- Access a topological service map visualization that can help spot issues and fix them faster, while also showing all dependencies between application and infrastructure components
- Monitor and gather information for application usage from a user perspective — either proactively through synthetic monitoring or passively through RUM (real user monitoring)
- Identify and alert on deviation trends and performance issues to help build better contingency plans and predict future occurrences that might affect the user
- Profile user actions from front end to back end so that they can be traced to the code, database query, or third-party call

Conclusion

In the end, APM is about providing insights through a diagnostics view that exploits every element of the software so that the end-user experience can be understood and continuously improved. By making observability and monitoring first-class citizens in the development process, teams can focus more on the quality of the solutions and less on firefighting.

Joana Carvalho, Performance Engineer at Postman

@radra on DZone | @joanacarvalho1987 on LinkedIn | @radra on Twitter

Joana has been a performance engineer for the last 10 years, during which she did root cause analysis from user interaction to bare metal, performance tuning, and new technology evaluation. Her goal is to create solutions to empower the development teams to own performance investigation, visualization, and reporting so that they can, in a self-sufficient manner, own the quality of their services. Currently working at Postman, she mainly does performance profiling, evaluation, analysis, and tuning.

CONTRIBUTOR INSIGHTS

Application Self-Healing

Common Failures and How to Avoid Them

Alireza Chegini, Senior DevOps Engineer, Azure Specialist at Coding As Creating

Today, automation is one of the major goals in IT projects. Most platforms are running on a cloud infrastructure and fully automated at both the platform and infrastructure levels. Companies are moving forward with automation and extending it to disaster recovery. As a result, many applications are designed in such a way to avoid failures and recover automatically. This is often called "self-healing." In this article, we will familiarize readers with self-healing and review some common features when applications experience failures.

What Is a Self-Healing Application?

A self-healing application is an application that detects a failure and tries to restore the situation before it escalates into a larger issue. For users, self-healing applications reduce system downtime. For developers, self-healing allows them to spend more time on development instead of fixing issues. When something fails, a self-healing application keeps on running, replicating the application. In short, it tries to restore the application to its default state. The primary tasks of a self-healing feature are to detect and repair problems. A self-healing application can automatically detect failures and detect system errors and stop the system automatically. For the purposes of this article, when we say application, we mean the whole system to which the application or platform belongs.

Self-Healing Levels

Each application or platform is not just the developed code. The hardware in which you run your code on and the connecting third parties are also part of your application. It is quite common for applications to depend on several third parties. On one hand, it is easier to focus on your main application's functionality and use third parties for other services, but on the other hand, if a third party fails, that can lead to your application failing.

Then, it is on you to take action and fix the issue. Moreover, when running your application on the cloud, you are dealing with virtual infrastructure and responsible for your infrastructure disaster recovery strategy and setup. Although all cloud providers give an SLA time and promise to keep all services up and running at the highest level they can, it is up to you to ensure your application is always accessible no matter where the failure is.

Before thinking about how to create a self-healing mechanism, you need to identify the points of failure. To design a self-healing system, you need to have a holistic monitoring overview of your application. Make sure nothing is left out of your radar. Then, you can define the possible scenarios and act accordingly to keep your application up and running all the time. To get a better picture of what needs to be done on the monitoring side, let's break the monitoring into some sub-areas.

OBSERVABILITY LEVEL

Monitoring is one of the most important parts of any application. The monitoring solution gives an observation of application behavior during runtime. Additionally, we can check infrastructure performance, network transactions, and third-party availability. The monitoring setup is not only about the application itself, but covers everything related to the application, from infrastructure to application components and third parties.

SMART ALERTS

When setting up alerts, engineers have to specify warning and critical thresholds for each alert. However, people often begin to see notifications or emails about it as soon as alerts are set up. These notifications do not necessarily mean that there is a

failure — rather, they might say that the system has passed a threshold. After a while, most engineers tend to ignore these alert notifications if they seem unimportant. That can lead to real issues being missed among many false-positive alerts. The correct approach with monitoring is to fine-tune the alerts so that each one is something you should take seriously and act on. If you have alerts that are not important, they should either be tuned or removed.

LOG EVERYTHING

Logging is not necessarily part of monitoring, but what makes it important is the data you collect. With logging, you can record all events with the exact time and date. When something fails, logs are golden information you have that tells you what happened, when, and where. That's why it is vital to log everything to track all possible reasons behind failures. It is recommended to centralize the logging into your monitoring system.

In most monitoring tools, you can connect the logging system to the monitoring setup, and the monitoring system will process the logging data. Smart monitoring systems can identify the relationship between application components, hardware, and third parties. Therefore, they can create a summary out of monitoring and logging data at the time of failure, which helps find the root cause faster.

Common Failure Areas

These are some of the common areas in which we experience application failures. Let's look at each failure area separately and the associated solution for each.

LOSS OF NETWORK CONNECTIVITY

One of the most common failures is losing network connection. Connection loss can happen for the entire application or even inside an application between components, like a database connection drop. The best approach to self-heal these failures is to create a retry mechanism that increases the chance of recovery in a short period. A good monitoring tool comes in handy to help resolve these issues. You can easily trigger a retry operation from the monitoring alert. Therefore, you can record an incident, as well as resolve it, automatically.

LACK OF SCALABILITY

When the number of requests on an application is higher than it can handle, the application starts to fail or cannot handle the requests. The solution for this is to make the application scalable. Scaling is something that can be designed and handled in different stages of application development. The best place to think about scalability is at architecture time, when you design your application with all components. You can choose technologies that cover scalability in an automated manner. One example is using container-based architectures and tools like Kubernetes, which handle scalability at different levels.

LONG-RUNNING TRANSACTIONS

Failures happen for long-running transactions, and after each failure, the transaction should start from the beginning. To keep the resiliency of these transactions, you can create checkpoints that help understand at which stage the failure occurred. Then, the system can start the transaction and continue from where it left off.

INSTANCE FAILURE

If an instance of an application cannot be reached, the only solution is to have another instance failover. This should be considered at the design stage, and instances should be added or removed based on need. So, if the instance is a database, that can be replicated to other instances to failover. If the instance is an application, you can use a load balancer or any traffic distributor service and add instances behind it. Currently, all cloud providers are supporting this feature as high availability. So this has to be configured at the same time that infrastructure is created.

OVERWHELMED APIS

Sometimes, sudden spikes in traffic can lead to high pressure on APIs, enabling applications to process requests properly. This can be prevented by using a queue to take jobs asynchronously.

Bottom Line

First, have a good architecture to assure that you cover all possible scenarios related to workload, scalability, resiliency, and high application availability. This includes the application, infrastructure, and third parties. The next thing is to establish a good monitoring system that identifies anomalies. AIOps solutions are great options to cover all monitoring requirements. These solutions can process application behavior and find anomalies that are outside of the basic monitoring radar. Do not miss adding logging to your monitoring to record all events.

Last but not least is to test your application and infrastructure to make sure they are continuously in a good configuration for the current workload. You could consider conducting load tests from time to time to simulate a higher workload to measure this. This is a continuous activity that helps keep your application running and reliable.

Alireza Chegini, Senior DevOps Engineer, Azure Specialist at Coding As Creating

@allirreza on DZone | @alirezachegini on LinkedIn | codingascreating.com

Alireza has more than 20 years of experience in software development. He started this journey as a software developer and continues working as a DevOps Engineer. In last couple of years, he has been helping companies move away from traditional development workflows and embrace a DevOps culture. These days, Alireza is coaching organizations as an Azure Specialist in their migration to public cloud.

CONTRIBUTOR INSIGHTS

Developing Your Criteria for Choosing APM Tool Providers

Identifying APM Requirements, Researching Solutions, and Building a Checklist

By Wayne Yaddow, Independent Data Quality Consultant

Selecting an application performance management (APM) solution can be a big undertaking as many factors must be considered. Each available tool delivers different feature sets; some tools only provide a view into particular layers of your stack but not complete visibility.

Further, several tool options are designed to run only on specific operating environments. Careful consideration of each APM solution is essential to ensure that it meets budgetary requirements and specifications. A primary goal is to discover comprehensive toolsets that focus on analyzing and improving the performance of end-user experiences.

APM tools allow you to monitor and manage the performance and uptime of your applications. For example, an APM tool can enable IT teams to monitor applications' speeds and response times to maintain SLAs (service-level agreements). If applications are below a specific threshold, the team can be alerted immediately, and the issue can be diagnosed to avoid bottlenecks or delays.

KEY TAKEAWAYS

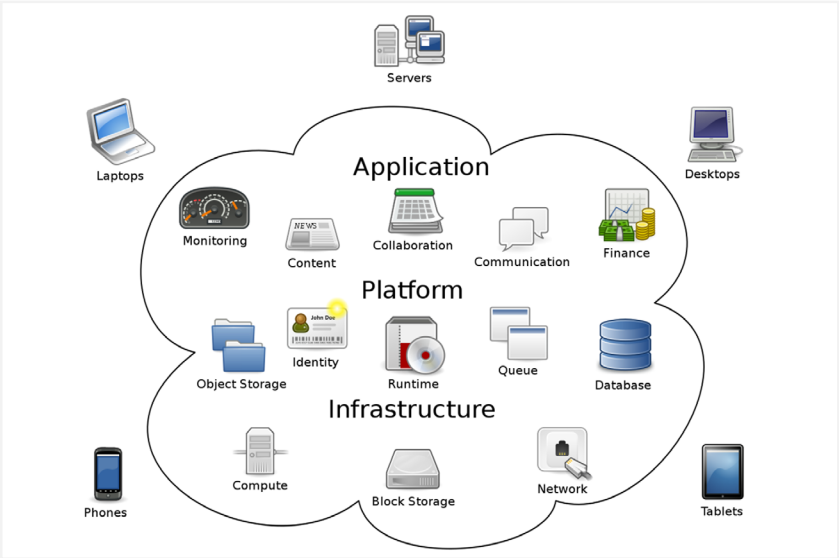
- There's increasing support by APMs to automatically discover and map application components (i.e., web servers, application servers, and microservices).
- A primary APM buyer's goal is to select a comprehensive APM tool to focus on analyzing and improving observability and end-user satisfaction.
- More than 20 major failure points often exist in today's systems; failure of any could result in a total outage or degraded performance. It may be time to consolidate APM tools.
- The right APM solution can literally revitalize your IT operations.

Before Choosing a Vendor, Identify Your APM Needs

Consider the system diagram for a web application shown in Figure 1. In this configuration, there are nearly twenty IT components that are potential points of failure. Errors in any sub-systems could result in a complete outage or degraded performance of the application.

A frequent challenge is to procure a new or different APM solution and decide which best suits your environmental requirements. The solution that is easy to deploy and maintain while delivering on your requirements is the one you should seek.

Figure 1: Example of an IT environment with diverse performance management needs

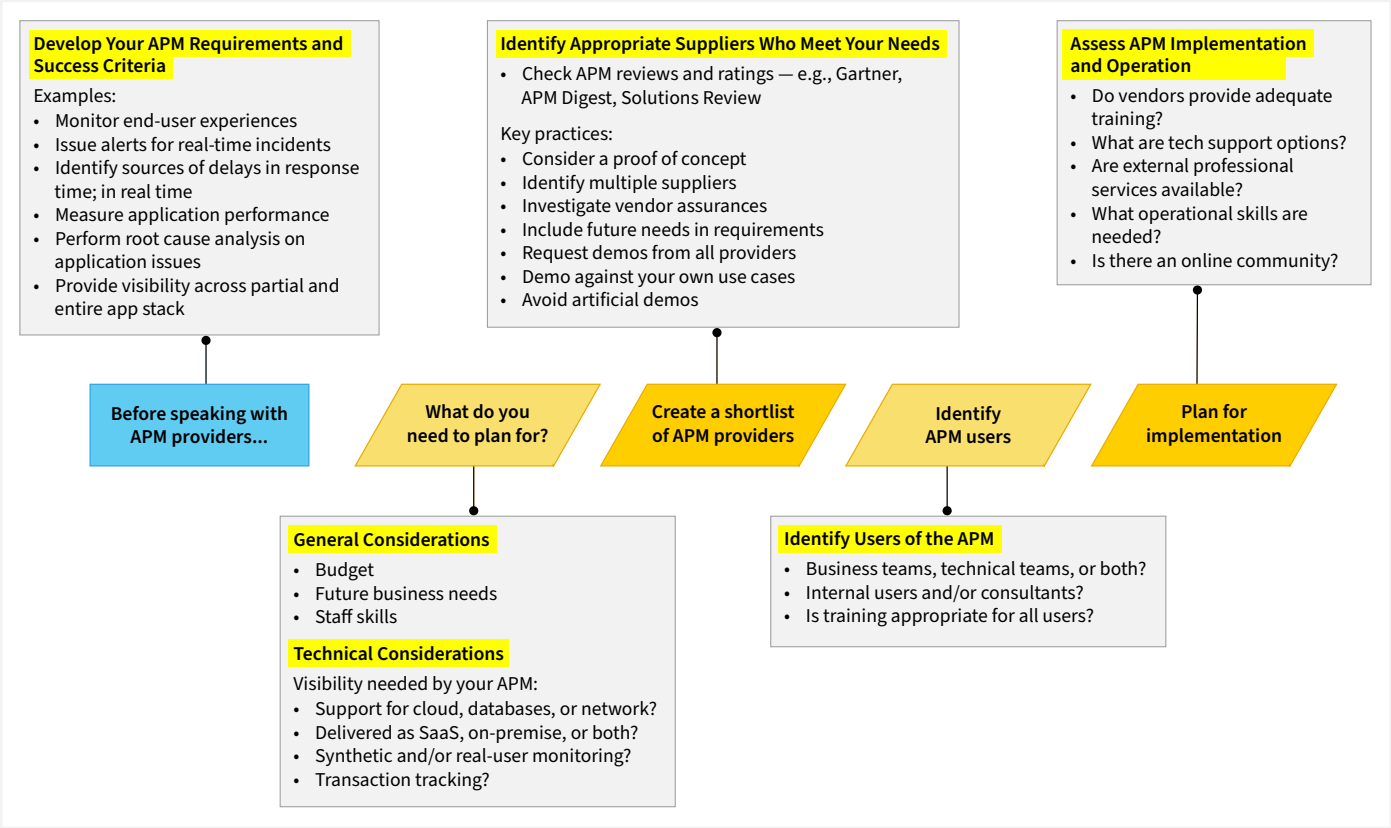


BEGIN BY CLEARLY DEFINING YOUR REQUIREMENTS

Reasons your organization needs to upgrade its approach to application performance management may seem simple. However, the current technologies and tools it relies on are inadequate. The reasons they are lacking are complex but exploring them will help you define the requirements for the new APM solution you'll choose. The chart in Figure 2 should help you get started with establishing your APM needs before reaching out to providers.

Conduct research, evaluate the sample criteria presented here, and add your own where needed. Carry out a rigorous proof of concept (POC) with the vendor's assistance and work with at least two or more vendors before deciding on a solution.

Figure 2: Quick guide to identifying your basic APM requirements



Tips to Help You Select an APM Solution

Each computing environment is composed of multiple platforms, operating systems, applications, and networks. It is important to list, review, and compare various APM tools to ensure that the one (or several) you choose meets your organization's requirements, budget, and staffing. Modern APM core functions include:

- Ability to integrate with other automation and service management tools
- Advanced alerting and reporting
- Features for error detection (e.g., root cause, multidimensional, stack trace for debugging)
- Analytics for business KPI and user paths (e.g., login to logout)
- Monitoring across application environments (e.g., mobile to mainframe)
- Automatic discovery and mapping of applications and their infrastructure components
- Integration with APM data from third-party sources
- Identifying problems and analyzing root causes
- Observability of applications' HTTP transactional performance
- Support for APM integrations and plug-ins
- Support for programming languages and frameworks (e.g., Python, Node.js, AngularJS, Java, Ruby)

Principal APM users include:

- Application owners and business users
- DevOps, ITOps, DataOps, and CloudOps teams
- Digital experience monitoring (DEM) and end users
- Site reliability engineers (SREs)

The APM requirements and selection process should quickly reveal features deemed essential for your organization. Users may access your apps from anywhere in the world through different browsers, devices, and connection speeds. Identify an APM tool or platform that provides real-time visibility into your websites, web services, infrastructure, and networks; simultaneously, the tool should include features that complement your monitoring objectives.

RESEARCH APM SOLUTIONS THROUGH INDUSTRY EXPERT REVIEWS

Descriptions, reviews, and ratings of leading APM features are detailed in "*2021 Gartner Magic Quadrant for APM Monitoring,*" Solutions Review's *Application Performance Monitoring Solutions Directory,* and the APM Digest website.

Many reputable resources are available to help guide you during the research process before adopting a solution. Look for experts and analysts in the industry who have researched, assessed, and collected reviews of leading providers and specific tools. In particular, groups such as Gartner and Solutions Review have resources dedicated to APM. Table 1 contains a summary of notable APM capabilities compiled by Solutions Review and offered by one or more of Gartner's top providers described as "Leaders" in the space.

Table 1

TRENDING CAPABILITIES OF LEADING APMS IN 2021

Capability	Description
Application analytics	• Allows users to analyze the content of critical processes across application hosts to identify performance bottlenecks • Takes your network's blind spots into account, offering a more authentic user experience
Applied intelligence	Allows users to detect, diagnose, and resolve problems before their IT staff or customers notice them, enabling them to identify and explain abnormalities for proactive and quick diagnosis and response
Automated anomaly detection	• Uses machine learning to automate anomaly detection and response, finding and fixing issues that affect application performance • Helps customers reduce MTTR with root cause diagnostic capabilities
Built-in integrations	Several APM solutions offer built-in (embedded) integrations in various categories, including orchestration, containers, service mesh, public cloud, messaging, DevOps toolchain, Internet of Things (IoT), and databases.
Code-level transaction monitoring	• Provides end-to-end observability of code-level transactions that affect the key performance indicators, including conversion and revenue • It helps users clearly understand the effect on business performance
Code-level visibility	Code-level visibility inspects methods, classes, and threads for requests
End-to-end distributed tracing	• Provides distributed tracing from the front end to the database • Tracks requests from RUM sessions to services, serverless functions, and databases, then connects API and browser test failures to back-end errors
Full-stack observability	Allows users to visualize, analyze, and optimize the entire software stack, including distributed services, applications, and serverless functions

(Table continues on next page)

TRENDING CAPABILITIES OF LEADING APMS IN 2021	
Capability	**Description**
Hybrid environment management	For businesses migrating to the cloud: • Performs real-time, automated performance insights before, during, and after the cloud transformation process • Delivers a consolidated view of application services and infrastructure
Integrated health monitoring	• Offers digital performance monitoring that keeps track of all application tiers, including server hosts and network health • Can collate application performance metrics with server resource metrics to provide an integrated view of network and application health
Intelligent observability	• Delivers intelligent observability for applications and networks with contextual information, artificial intelligence, and automation • Helps users understand the full context of observed data from user impact through entity interdependencies
Telemetry data platform	• Ingests and stores all of a user's operational data, including logs, in one place with live alerts and custom application support • Features several hundred agents and integrations, including OpenTelemetry

DEVELOP CHECKLISTS AND QUESTIONS FOR APM SUPPLIERS

Are you familiar with the features and behavior that your APM solution should offer? Setting business goals and objectives (e.g., increased revenue, reduced risk, and costs) is a critical step in selecting the best APM solution for your business. For application owners who utilize a due diligence list, consider the importance of the following information:

Table 2

CATEGORIES OF QUESTIONS AND CONSIDERATIONS FOR PRE-SELECTED APMS	
Capability	**Assessment Questions and Considerations**
APM data and reporting	• What is the complete selection of APM charts and graphs available? • What type and how many users will access and use the information — C-suite executives who want simplified visualizations to comprehend the data quickly, or are data experts the primary people on the team who should access and interact with the information outputs? • To process and visualize APM data, how many servers would be needed? • Do I need a supporting APM database on separate devices? • If the APM uses agents to collect data, how many and what kinds of agents are necessary to support my implementation? Is it all going to be on our network or the vendor's network? • Is the APM tool able to extract data from other existing tools and correlate the data? • Do you need to recover the data manually, or can it be sent to your development team's existing alert tools? • Does it include an API that allows other tools to extract data from it?
Budget and pricing	As you put your new tools (and new expertise) to use to solve more APM problems, you may be presented with opportunities to innovate with outside-of-the-box thinking. Ask yourself: • How much can you spend? • Does the vendor's pricing scheme work for you? • What is included and what options could be more costly?

(Table continues on next page)

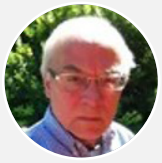
CATEGORIES OF QUESTIONS AND CONSIDERATIONS FOR PRE-SELECTED APMS

Capability	Assessment Questions and Considerations
Complexity	• Can your APM solution manage a few applications or hundreds? • Are all of these applications internal, or are they also in the cloud? • What application models are they adhering to (monolithic, service, and/or microservice)? • In what programming languages are applications written? • What is the status of your infrastructure modernization process? • Do you require SaaS solutions alone, on-site, or both? • What additional hardware, software, and/or applications are required to support the APM?
End-user	Some APM tools with digital experience monitoring are explicitly designed for application developers, which can be too restrictive for users not involved in development.
KPIs	Does the solution you are considering measure the KPIs you require? For example, is the granular data you need to be provided on areas such as diagnostics at the code level or tracking performance under specific parameters such as user location?
Integration	You likely have other tools in your organization to help manage and monitor the IT environment. You will want your new APM tool to slide right onto it without creating yet another place to search for data, so: • Learn about the tools that the APM product you are considering integrates with. • Compare APM tools under consideration with tools you already have or are considering implementing.
Scalability	• Should the APM solution only address your application and service management issues now, or should it be able to grow as your operations expand? • Does the APM tool tie you to proprietary hardware and/or software?
Support	• What options are available? Are they 24/7? • Does the vendor position itself as a partner or more as a product provider?

Conclusion

Implementing well-chosen APM solutions can renew your computing operations. The economic value your chosen APM solution will provide is remarkable and easily quantifiable. IT infrastructures have grown so complex that the traditional APM "silo" approach doesn't work anymore. Many organizations have a variety of surveillance products deployed. They can be reduced to one or a few in most cases, thereby lowering costs and increasing value.

If you have not assessed your existing application performance management process and tools recently, start looking right now to avoid pressure later. And if you don't have a current APM solution, it is time to consider setting up a proof-of-concept demo with prospective providers. It will cost nothing upfront, and the vendors will be eager to show you how their products can bring value to you and your team.



Wayne Yaddow, Independent Data Quality Analyst
@wyaddow on DZone | @wyaddow on LinkedIn | Website

Wayne Yaddow has 12 years of experience leading data migration/integration/ETL testing projects at organizations including J.P. Morgan Chase, Credit Suisse, Standard and Poor's, AIG, Oppenheimer Funds, and IBM. Wayne has written extensively on this topic and taught IIST (International Institute of Software Testing) courses on data warehouse, ETL, and data integration testing. He continues to lead ETL testing and coaching projects as a freelance consultant. You can contact Wayne at wyaddow@gmail.com.

CONTRIBUTOR INSIGHTS

Common Performance Management Mistakes

How to Avoid Pain Points Introduced by Cloud-Based Architectures

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

Performance in any cloud-distributed application is key to successful user experience. Thus, having a deep understanding of how to measure performance and what metric IO pattern to use is quite important. In this article, we will cover the following:

- Performance analysis checklist and strategies
- Performance issue detection and tools
- Performance anti-patterns and how to mitigate issues

Performance Monitoring in the Cloud Checklist

When improving the performance of your application or infrastructure architecture, use the following checklist:

- ☐ Introduce logging tools in your application and infrastructure components
- ☐ Find anti-patterns and bottlenecks of your code and infrastructure
- ☐ Set up monitoring tools to gather CPU, memory, and storage utilization
- ☐ Adjust monitoring tools to gather info about events between base components
- ☐ Do not log everything — identify important events to log

Most Common Application Performance Anti-Patterns

The most important item from the performance checklist is identifying anti-patterns. This will save valuable time once you already know the weak spots.

NOSY NEIGHBOR

Imagine you have a microservice that is deployed as a Docker container, and it is eating more CPU and memory than other containers. That can lead to outages since other services might not receive enough resources. For example, if you use Kubernetes, it may kill other containers to release some resources. You can easily fix this by setting up CPU and memory limits at the very beginning of the design and implementation phases.

NO CACHING

Some applications that tend to work under a high load do not contain any caching mechanisms. This may lead to fetching the same data and overusing the main database. You can fix this by introducing a caching layer in your application, and it can be based on a Redis cache or just a memory cache module. Of course, you don't need to use caching everywhere, since that may lead to data inconsistency.

Sometimes, you can improve your performance by simply adding output caching to your code. For example:

```
namespace MvcApplication1.Controllers
 {
      [HandleError]
      public class HomeController : Controller
```

(Code continues on next page)

```
    {
        [OutputCache(Duration=10, VaryByParam="none")]
        public ActionResult Index()
        {
            return View();
        }
    }
}
```

Above, I've added the output cache attribute to the MVC application. It will cache static content for 60 seconds.

BUSY DATABASE

This issue is often found in modern microservices architectures, when all services are in a Kubernetes cluster and deployed via containers, but they all use a single database instance. You can fix this problem by identifying the data scope for each microservice and splitting one database into several. You can also use the database pools mechanism. For example, Azure provides the Azure SQL elastic pool service.

RETRY STORM

Retrying storms and the issues they cause usually occur in microservice- or cloud-distributed applications; when some component or service is offline, other services try to reach it. This often results in a never-ending retry loop. It can be fixed, however, by using the circuit breaker pattern. The idea for circuit breaker comes from radio electronics. It can be implemented as a separate component, like an auto switch. When the circuit runs into an issue (like a short circuit), then the switch turns the circuit off.

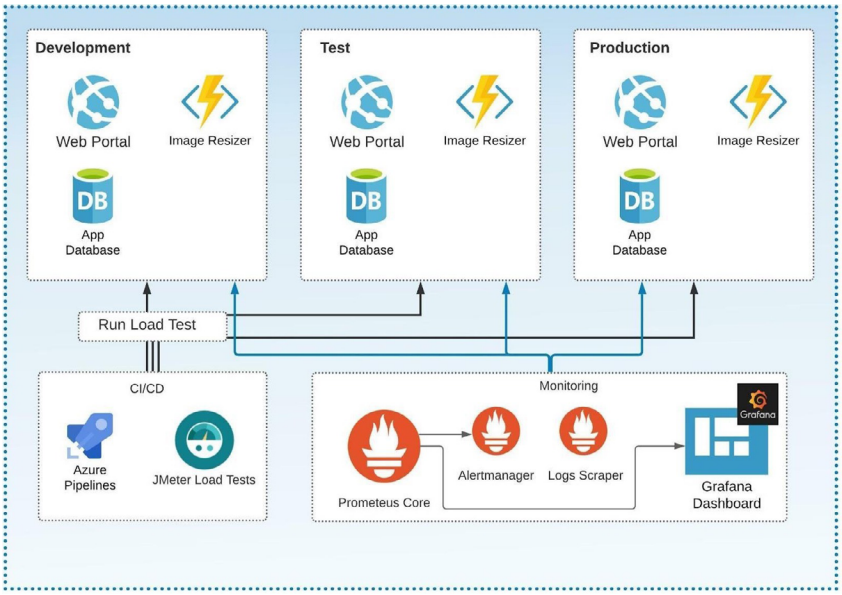
Example: Cloud Performance Monitoring Architecture That Result in an Outage

To check the performance of your architecture, you need to run load tests against your application. Below, you can see an architectural example that is based on Azure App Services and Azure Functions. The architecture contains the following components:

- Separated into four stages: Dev, Test, Staging, and Production.
- Login and monitoring are implemented with Prometheus and Grafana
- Loading tests are implemented with Azure DevOps and JMeter

Here, you can find an example of how to set up a JMeter load test in Azure DevOps.

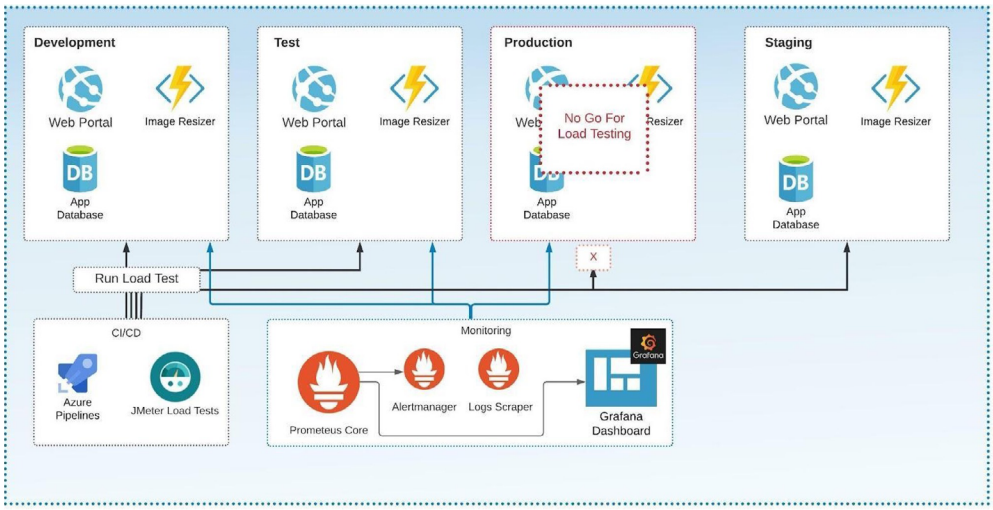
Figure 1



FIXING THE ISSUE

As you can see in the diagram, everything looks good at first glance. Where do you think there could be an issue? You can see the fixed architecture of the diagram in Figure 2.

Figure 2



First of all, you should never run load testing against the Production stage as it may (and often will) cause downtime when you run an excessive load test. Instead, you should run the test against the Test or Staging environments. You can also create the replica of your production environment explicitly for load test purposes. In this case, you should not use real production data, since that may result in sending emails to real customers! Next, let's look at an application that should be architected under a high load.

Example: High-Load Application Architecture on AKS

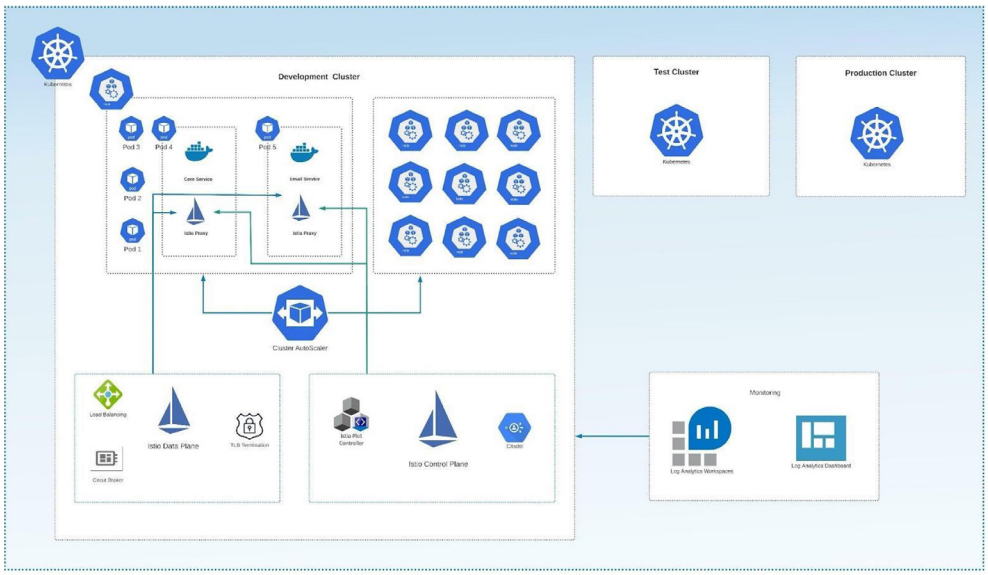
Imagine that we have a popular e-Commerce application that needs to survive Black Friday. The application architecture should be based around the following components:

- Azure Kubernetes Services (AKS) with Kubernetes Cluster Autoscaler are used as the main distributed environment and mechanism to scale compute power under load.
- Istio's service mesh is used to improve cluster observability, traffic management, and load balancing.
- Azure Log Analytics and Azure Portal Dashboards are used as a central logging system.

In the figure to the right, you can see that the AKS cluster contains nodes that are represented as virtual machines under the hood.

The Autoscaler is the main component here. It scales up the node count when the cluster is under high load. It also scales the node down to a standard size when the cluster is under normal load.

Figure 3



Istio provides the data plane and control plane, assisting with the following:

- Load balancing
- TLS termination
- Service discovery
- Health checks
- Identity and access management
- Configuration management
- Monitoring, logs, and traffic control

Please note that the architecture includes the stages Dev, Test, Staging, and Production, of course. The formula for the highly available Kubernetes cluster is having separate clusters per stage. However, for Dev and Test, you can use a single cluster separated by namespaces to reduce infrastructure costs.

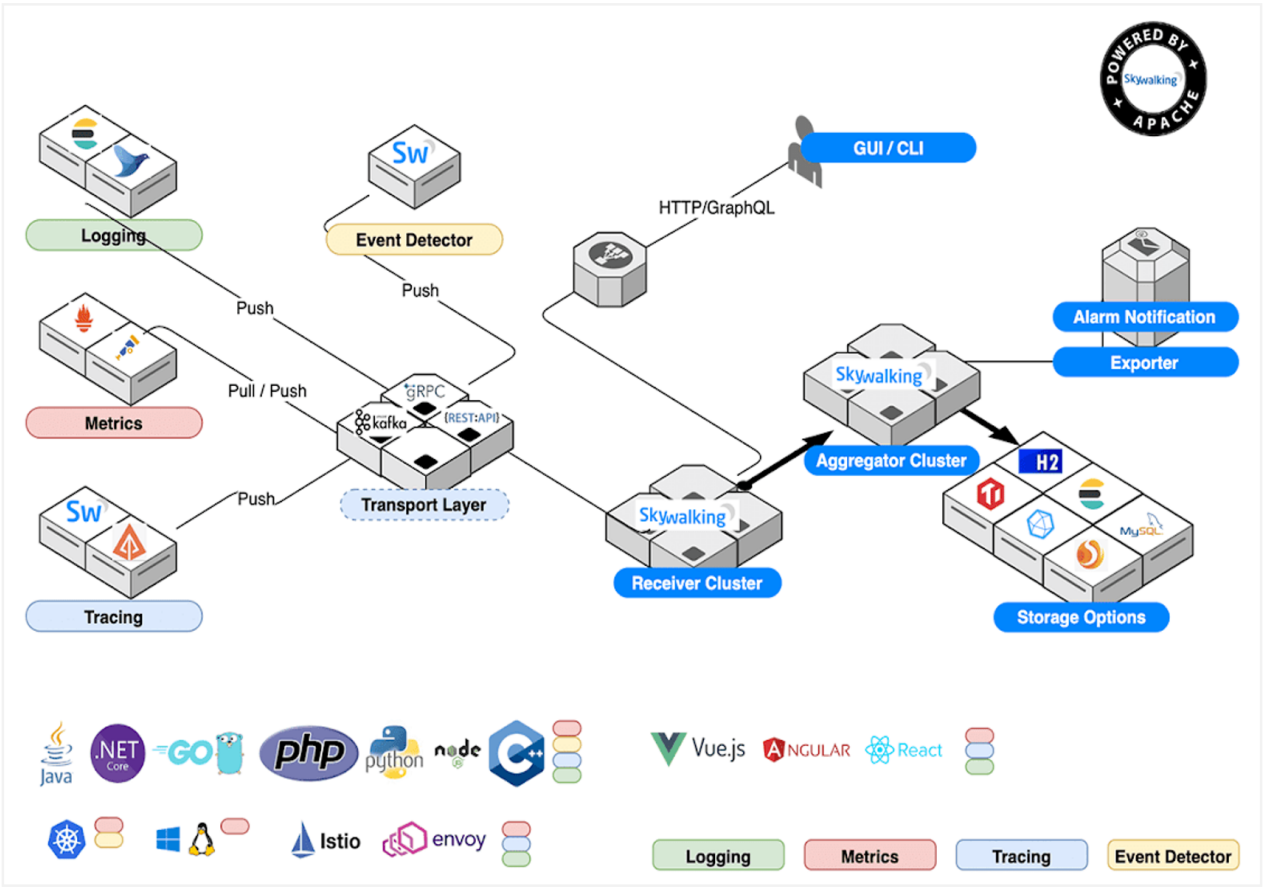
For additional logging reinforcement, we used Azure Log Analytics agents and the Portal to create a dashboard. Istio contains a lot of existing metrics, including those for performance and options to customize them. You can then integrate with a Grafana dashboard. Lastly, you can also set up a load test using Istio. Here is a good example.

Top Open-Source Application Monitoring Tools

Let's start off by listing some of the most popular open-source application performance tools:

- Apache Sky Walking is a powerful, distributed performance and log analysis platform. It can monitor applications written in .NET Core, Java, PHP, Node.js, Golang, LUA, C++, and Python. It supports cloud integration and contains features like performance optimization, slow service and endpoint detection, service topology map analysis, and much more. See the feature map in the image below:

Figure 4

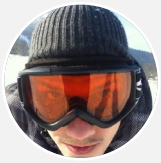


- Scouter is a powerful tool that can monitor Redis, Nginx, Kafka, MongoDB, Kubernetes, and other sources. It can monitor CPU, memory network and heap utilization, active users, active services, and more.
- GoappMonotr is a tool that provides performance monitoring for Golang applications.
- Pinpoint is a performance monitoring tool for Python, Java, and PHP applications. It can monitor CPU, memory, and storage utilization. You can integrate it into your project without changing a single line of code.
- Code Speed is a simple APM tool. It can be installed into your Python application to monitor and analyze the performance of your code.

There are various tools that have community licenses or trials. If you are using Azure, you can enable Azure AppInsights with low cost or no cost at all, depending on bandwidth.

Conclusion

In this article, we've dug into common performance mistakes and anti-patterns when working with distributed cloud-based architectures, introducing a checklist to consider when building applications that will face heavy loads. Also, we explored the open-source tools that can help with performance analysis, especially for projects on limited budgets. And, of course, we've covered a few examples of highly performant applications and an application that may have performance issues so that you, dear reader, can avoid these common performance mistakes in your cloud architectures.



Boris Zaikin, Software and Cloud Architect at Nordcloud/IBM GmbH

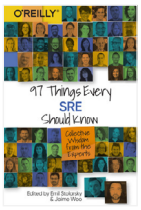
@borisza on DZone | @boris-zaikin on LinkedIn | boriszaikin.com

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

ADDITIONAL RESOURCES

Diving Deeper Into Application Performance

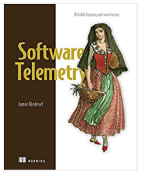
BOOKS



97 Things Every SRE Should Know

By Emil Stolarsky and Jaime Woo

In this thoughtfully curated collection, readers will discover diverse, wide-ranging perspectives from both newcomers and seasoned professionals in the SRE space. You'll not only get useful tips and trusted advice but also explore leading approaches to site reliability, the importance of SLOs, and methodologies for monitoring, observability, debugging, security, and more.



Software Telemetry: Reliable Logging and Monitoring

By Jamie Riedesel

Gain insight into the state of your applications and data sources with telemetry systems to better monitor and maintain your overall infrastructure and data. In this hands-on book, readers are guided through system instrumentation, centralized logging, distributed tracing, and other key telemetry techniques.

TREND REPORTS

CI/CD: Automation for Reliable Software Delivery

DevOps has become more crucial than ever as companies largely remain distributed workplaces and continue accelerating their push toward cloud-native and hybrid infrastructures. This Trend Report examines the impact on development teams globally and dives into the latest DevOps practices that are advancing CI/CD and release automation.

Application Performance Monitoring

DZone's 2020 APM Trend Report explored leading tools and processes, the integration of machine learning and automation, and emerging trends. Readers can compare findings from our research then with this year's report and revisit our exclusive talk with DevOps activist Andreas Grabner. Contributors also shared insights into APM's impact on team culture and the end-user experience, AIOps, and priorities for growth-minded organizations.

REFCARDS

Full-Stack Observability Essentials

Observability and telemetry join to correlate individual systems' strength with the overall business health, highlighting the state of complex systems, processes, and microservices of a tech stack and/or application — all purely from existing data streams. In this Refcard, explore the fundamentals of full-stack observability and adopting OpenTelemetry for increased flexibility.

End-to-End Testing Automation Essentials

End-to-end (E2E) tests are often neglected, due in large part to the effort required to implement them. Automation solves many of these problems by ensuring a consistent, production-like test coverage of the system. In this Refcard, learn the fundamentals of E2E test automation through test coverage, integration, and no-code options.

PODCASTS

AI+ITOPS Podcast

From APM Digest comes their audio resource on all things ITOps. Hear about issues IT teams are facing amidst their digital transformation, as well as guests' perspectives on machine learning, AIOps, application performance, and observability.

OpenObservability Talks

As its name suggests, this podcast serves to amplify the conversation on open-source technologies and advance observability efforts for DevOps. Listen to industry leaders and contributors to projects like OpenTelemetry and Jaeger discuss use cases, best practices, and visions for the space.

Performance Testing and SRE Podcast

Since 2019, TestGuild has brought us 80+ episodes that cover a wide range of performance-related topics. Tune in with Joe Colantonio to learn about chaos engineering, test automation, API load testing, monitoring, site reliability, and (much) more.