DZone
A DEVADA MEDIA PROPERTY

Containers

BROUGHT TO YOU IN PARTNERSHIP WITH

Lacework™

Welcome Letter

By Guillaume Hugot, Director of Engineering at DZone

Containerization is probably one of the highest-trending words in the tech world today. However, the principle is not new. From the early times of multitasking — which introduced process parallelization and shared memory space — until today, there has been an increasing demand for abstraction from the limitations of hardware. The concept of modern containerization originated 15 years ago, back in 2006 when Google began working on Linux control groups that isolated the resource usage of a collection of processes.

The successive adoption of virtual machines, followed by the emergence of containerization itself, has become widespread across companies, accelerated by out-of-the-box solutions offered by cloud service providers. From less than 20% of global organizations running containers for their applications in 2019, market analysts expect a rise to more than 70% of them using one or more containers by 2023, only four years later.

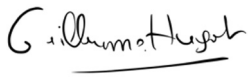
With this mainstream shift toward cloud-native development, more and more organizations are realizing substantial benefits as they modernize their architectures with containerized environments: increased agility, full control of the deployment chain by engineers, automatic scaling, simplified and standardized monitoring, and better integration with modern solutions. And the pervasive use of simple services dedicated to single tasks (i.e., microservices) also fueled a greater need for an advanced solution to support such services — one equivalent in terms of infrastructure: simple, independent, and easily replicable.

While this move promises to accelerate application development, it also introduces a new set of challenges that occur with a fundamentally altered software delivery pipeline — shared kernels lead to new attack vectors, hybrid and multi-container environments grow in complexity as they scale, and infrastructure provisioning must be managed differently.

Today, containerization is mostly implemented through the Docker/Kubernetes ecosystem. Many solutions designed to simplify container use are offered by major cloud players like Amazon ECS, Google GKE, Azure Container Instances, and Red Hat OpenShift.

In DZone's 2021 Containers Trend Report, we explore the current state of container adoption, uncover the common pain points of increasingly complex containerized architectures, and introduce modern solutions for building scalable, secure, stable, and performant containerized applications.

Sincerely,



Guillaume Hugot

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

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

ORIGINAL RESEARCH

Key Research Findings

An Analysis of Results from DZone's 2021 Containers Survey

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

In April 2021, DZone surveyed software developers, architects, and other IT professionals in order to understand how containers (sub-VM-level infrastructure abstractions) are used today.

Major research targets were:

1. Real-world benefits and challenges of using containers
2. Design and architecture of containerized apps
3. Creation and maintenance of containers and container images

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 March 19th and closed on April 6th. The survey recorded 496 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: Real-World Benefits and Challenges of Containers

Motivations:

1. Containers are a nice abstraction, but every abstraction comes at a cost. We wanted to know what those costs cash out to in practice.
2. Developers know that abstractions involve drawbacks, but not every fear is equally well founded or evenly distributed. We wanted to know which expected challenges of container adoption turned out better or worse than expected.
3. Ever since Docker took off, "container" has become a buzzword, which ups the signal:noise ratio, especially in such adoption-related discussion domains as benefits and challenges. We wanted to take an empirical look at the reality behind the advertising noise.

RELATIVE IMPORTANCE OF ASPECTS OF CONTAINERIZATION

The concept of "containerization" captures both how things work and what things are for. Further, while x may be true of containers, x may not figure prominently in implementers' mental model of containers. In order to understand how software professionals think about containerization, we wanted to distinguish the "how" from the "why," further subdivide each genus into multiple species, and draw a picture of how these aspects are weighted in developers' minds.

So we asked:

"Containerization" can mean many things. Please rank the following aspects of containerization in order of importance (top = most important).

Results (n=410):

Table 1

ASPECTS OF CONTAINERIZATION RANKED

Aspect	Score
High availability	2561
Process isolation	2470
Quick spin-up development environments	2368
Magical, effortless deployment	2178
Horizontal elasticity	2154
Memory isolation	1815
Filesystem isolation	1800
Network stack isolation	1706
Granular resource control	1684

Note: Many provided answer choices that overlap. Our goal was to understand respondents' mental model, which often includes redundancy (e.g., for use in multiple types of problem analysis), not paint a purely non-subjective map of the relative importance of containerization aspects.

Observations:

1. Granular, resource-isolation-type aspects of containers are considered less important than higher-level, architectural-type aspects.

   Process isolation ranked third most important, but process is a higher-level concept that need not map onto an OS-level resource management concept, which contrasts with the other four named types of isolation (memory, filesystem, network stack, and general granularity). The lowest ranked four aspects were all types of non-process isolation.

2. The most DevOps-y aspect of containerization — high availability — ranked as most important. But top rankings vary significantly by respondent role. Pure developers ranked process isolation as (slightly) most important (score 1054 vs. 1034 for high availability), while non-developer technical respondents (technical architects, SREs, sysadmins, and QA engineers) ranked high availability as most important by a higher margin (1000 vs. 937).

3. The percent of computational work done in containers impacts containerization aspect ranking slightly. Respondents whose organizations run ≥ 50% of computational work in containers ranked magical, effortless deployment higher than horizontal elasticity (4th vs. 5th place, respectively).

   While among respondents whose organization run < 50% of computational work in containers, the order of these two aspects is flipped. Since the "percent of computational work" numbers come entirely from respondents' estimates, this result may partly reflect a conjunction of software professionals' attitudes rather than a correlation between attitudes and actual workload.

4. Attitudes toward container immutability also somewhat impact responses. Respondents who consider immutability high importance ranked high availability above process isolation (1707 points vs. 1688), while respondents who consider immutability low importance ranked the two equally (1624 points).

   Relative rankings of horizontal elasticity vs. magical, effortless deployment were also flipped between the two groups,

with pro-immutable respondents ranking horizontal elasticity slightly higher. Part of the point in asking about the importance of container immutability was to distinguish "sysadmin" from "DevOps" stances.

We supposed that, *ceteris paribus*, someone who more strongly thinks containers ought to be immutable thinks in a more "DevOps" way than someone who believes in container immutability less strongly, on the analogical contrast between "immutable container as build" and "mutable container as configure."

This assumption seems intuitively consistent with the relative rankings of container aspect importance: High availability and horizontal elasticity intuitively seem more consistent with "DevOps" attitudes, while process isolation seems more consistent with "sysadmin" attitudes. Effortless deployment seems no more or less DevOps-related than sysadmins.

BENEFITS OF CONTAINERS: EXPECTED VS. OBSERVED

Lists of benefits are probably skewed by marketing pressures, and actual benefits probably don't distribute evenly. We wanted to know where containers aren't all they're cracked up to be, and who found bigger expected vs. observed differences for each benefit.

So we asked:

What benefits from containers have you expected vs. observed in practice? For example, if you expected much faster deployments but observed no increase in deployment speed after adopting containers, rate "Expected" five (or whatever) stars and rate "Observed" zero stars (click the "x" to clear).

Results (n=415):

Table 2

BENEFITS OF CONTAINERS: EXPECTED VS. OBSERVED

Benefit	Expected (avg)	Observed (avg)
Faster deployment	4.5	4
Easier dev environment setup	4.5	4
Consistent environments	4.5	4
High availability	4.5	4
Modularity	4.5	4
Build efficiency	4.5	4
Simplified version control	4	3.5
Portability	4.5	4
Lightweight footprint	4	4
Ease of maintenance	4.5	3.5
Scalability	4.5	4
Security	4	3.5
Ease of sharing	4.5	4

Observations:

1. Most purported benefits of containers were slightly lower in observation vs. expectation (by an average of 0.5 stars).

   The magnitude of this negative gap between expectations and observed reality is consistent with responses to similar questions across our research program, and it perhaps more expresses the mildly pessimistic metaphysic of the developer ("things are usually a bit worse than you might think in advance") than signifies anything in particular about container overhyping.

2. The most disappointing purported benefit of containers (i.e., the benefit with the greatest gap between expectations and observations) was ease of maintenance.

   So it may be harder to maintain containers than it looks. This ratio held across users of different container runtimes (`containerd`, `rkt`, etc.) and attitudes toward the importance of container immutability.

3. One purported benefit was observed by overall respondents as much as expected: lightweight footprint.

   However, among respondents who consider container immutability to be low importance (≥ 3 out of 5 importance rating), containers' lightweight footprint is less observed (3.5) than expected (4).

   This suggests that while containers overall do not bloat unexpectedly, not consciously considering container immutability important may result in unexpected container bloat.

   Presumably, this is because accretion of intra-container changes over time is more likely to leave cruft behind than clean container rebuilds, on the general principle that updates are more likely to result in more garbage than creates.

4. Quantity of containerized code impacts expected vs. observed container benefit. Container-heavy organizations, defined as organizations that have ≥ 50% of code (by LoC or other quantity metric) in containers, see no difference between expected vs. observed portability, while container-light organizations (< 50% of code in containers) see a portability observed benefit drop-off.

   We might imagine a snowball effect: If more code is containerized, then more can be re-used in other call chains (e.g., in a microservices architecture) with little or no architectural coupling. So where more code is containerized, the portability benefit experienced would be higher.

   Container-heavy organizations also experienced no gap between observed vs. expected security benefits of containers (in comparison to a half-star drop-off among container-light organizations).

   Here, the explanation is likely not a snowball. We conjecture (with low confidence) that container-heavy organizations are likely to have more mature SDLCs than container-light organizations. If that is the case, then perhaps security is baked in earlier thanks to overall SDLC maturity. This conjecture will be interrogated in our upcoming surveys on continuous delivery.

CHALLENGES OF CONTAINERS: EXPECTED VS. OBSERVED

Marketing buzz adds noise to discussion of both benefits and challenges, since products and services respond to both positive and negative motivations. We wanted to cut through the noise around challenges of adopting containers as well as purported benefits. So we asked:

What challenges from adopting containers have you expected vs. observed in practice? For example, if you expected refactoring for containerization to be half as difficult as it turned out to be, then rate "Expected" two stars and "Observed" four stars.

Results (n=406):

Table 3

CHALLENGES OF CONTAINERIZATION: EXPECTED VS. OBSERVED		
Challenge	**Expected (avg)**	**Observed (avg)**
Refactoring/rearchitecting legacy applications	4	3.5
Ensuring application and network security	4	3.5
Lack of developer experience with containers	4	4
Application performance monitoring	4	3.5
Limited toolsets	3.5	3.5
Storage scaling	3.5	3.5
Platform selection	3.5	3.5
Immature technologies	3.5	3.5
Unproven ROI of containers	3.5	3.5

Observations:

1. Most challenges were well understood before implementation: Six out of nine challenges showed no difference between expectation and observation.

2. No challenge exceeded expectations. In three cases, expectation was worse than reality. All three — refactoring/rearchitecting, ensuring security, and performance monitoring — pertain to the knock-on effects of containers on applications rather than the creation and maintenance of containers themselves.

   This suggests that containers are slightly more successfully black-box than software professionals fear. We conjecture that this is due to an unusually low hype:maturity ratio with respect to application design.

   Containers have been a mature Unix-family technology for decades. While the hype has grown recently only around the ecosystem, this is not the part of the technology that directly affects applications, i.e., resource isolation and infrastructure ephemerality/definition itself.

   But in many cases, a recent upsurge in hype has corresponded with low technology maturity level (e.g., IoT), which leads developers to expect maturity in inverse proportion to hype (e.g., in Gartner-hype cycle terms, the developer's "trough of disillusionment" is less precipitous than the CIO's).

3. Challenge expectation-observation gap varied slightly with the quantity of code in containers. Respondents at organizations with < 50% code (by LoC or some other metric) in containers encountered no difference between the expected and observed challenge of refactoring and/or rearchitecting legacy applications — i.e., their fears were somewhat more justified.

   We interpret this as a simple function of relevant experience or container-friendly starting point. As an organization containerizes more applications, experience getting applications to run in containers increases, and increase in experience produces less observed difficulty in doing the same thing going forward.

   Further, organizations whose applications are already well suited to containers (e.g., built with well-defined and well-understood context boundaries, not dependent on long-running, OS-level threads) are more likely to containerize more and find it easier to do so.

4. Free responses mentioned debugging and error handling more often than any other challenge mentioned in free response. In future research, we will examine challenges with debugging containers more explicitly.

Research Target Two: Design and Architecture of Containerized Apps

Motivations:

1. The abstract idea of infrastructure abstraction is simple. The simple idea becomes more interesting when its impact on complex things (e.g., application-level) is better understood. So we decided to explore this idea further.

2. Higher-level programming paradigms (object-oriented, functional, actor model) and many specific application domains — most notably request-response with high request frequency and low operation complexity — do not always map onto OS-level constructs naturally.

   Sometimes, the mismatch is handled at the application level. For example, the web server NGINX improves high-volume performance vs. Apache by avoiding web server thread: POSIX thread mapping because pthreads are designed for relatively complex and long-running operations (e.g., that require blocking and interrupt logic).

   Containers facilitate ephemeral infrastructure at a higher level than application-level, so we wanted to understand how between-OS-and-application infrastructure layers impact application design.

3. When multiple things share a dependency, and the dependency mutates (e.g., is upgraded) and one of the dependent things mutates to match but the other does not, previously painless coupling becomes painful.

   Containers can be used to build cleaner separation between dependency trees (at the expense of storage but storage is cheap these days), which can facilitate more modular, and less coordinated development (e.g., microservices).

   We wanted to see if and how container usage actually does facilitate less centrally planned application and enterprise development.

4. Containers, in principle, encourage the development of applications as distributed systems such as loose coupling, supra-OS coordination tasks, and expected failures. We wanted to see how containers affect the distributed nature of applications in practice.

CONTAINER USAGE BY SDLC LOCATION

Intuition about environments: Production environment runtime capacity varies a lot, but environment configuration is comparatively stable. Developer environment runtime capacity changes comparatively little, but environment configuration — as developers add dependencies and play with new tools — varies often. Staging and build environments occupy respective intermediate places on the variable/stable spectrum.

But as they say in the real world: *deduction, schmeduction*. We wanted to know where people actually use containers.

So we asked:

Where do you use application containers?

Results (n=330):

Figure 1

LOCATION OF APPLICATION CONTAINER USAGE

	In production environments	In pre-production staging environments	In pre-production build-only enviornments	In development environments	Nowhere
I use containers...	303 (28.1%)	254 (23.5%)	166 (15.4%)	328 (30.4%)	28 (2.6%)
My company uses containers...	330 (30.4%)	262 (24.2%)	191 (17.6%)	278 (25.6%)	23 (2.1%)
Total Checks	633 (29.3%)	516 (23.9%)	357 (16.5%)	606 (28.0%)	51 (2.4%)

Observations:

1. Overall, containers are used more in production than other environments, but not by much. This is not surprising in itself but is interesting diachronically: In similar surveys we ran in 2017-2019 (2017, 2018, and 2019), more respondents reported using containers in development environments than in production environments, with the gap steadily decreasing year over year.

   The questions were formulated somewhat differently, and we do not have 2020 data, so absolute numbers are hard to compare. It may be the case that the "tipping point" in which containers became production-dominant came some time over the last year or two.

2. More synchronically interestingly: SDLC location of container usages varies significantly by the amount of computational work done in containers.

   30.4% (n=199) of respondents whose organizations run ≥ 50% of computational work in containers reported using containers in production, while only 24.5% (n=100) of respondents whose organizations run < 40% of computational work in containers reported using containers in production.

   Moreover, 29.2% (n=191) of respondents in the ≥ 50% group reported using containers in development environments vs. 32.4% (n=132) of respondents in the < 50% group.

   Note: A similar relationship obtains "I use" and "My company uses" respondents.

   Assuming constant "containerization maturity," this suggests that developers are not restricted from using containers to simplify development by their organizations' production container usage. It may also suggest that some significant portion of container-heavy organizations' computational work is relatively homogeneous, i.e., horizontal containerized node scaling.

TWELVE-FACTOR ATTRIBUTE DISTRIBUTION: APPLICATION CODE VS. CONTAINER DESIGN

The twelve-factor app concept was formalized 10 years ago, aggregated much older practices, and has recently come under scrutiny partly because cloud service proliferation has shifted some design-level decisions from the application to the infrastructure.

A paradigm once needed to break the application server monolith has become (somewhat) more a matter of habit and less a matter of conscious effort than a decade ago. Although how much this has penetrated into actual application design — as opposed to conversation about application design — will be treated in another survey.

We wanted to know how containers figure in this "sub-application-design" shift. So we asked:

Which of the following principles of twelve-factor apps have your applications followed recently? Distinguish "application code" from "container design" in your answers.

Results (n=331):

See Figure 2 on the following page.

Figure 2

PRINCIPLES OF TWELVE-FACTOR APPS: APPLICATION CODE VS. CONTAINER DESIGN

	In application code	In container design
One codebase, many deploys	331 (54.2%)	280 (45.8%)
Explicitly declared and isolated dependencies	246 (49.8%)	248 (50.2%)
Config stored in environment	250 (46.3%)	290 (53.7%)
Backing services as attached resources	199 (47.2%)	223 (52.8%)
Stateless processes	254 (52.6%)	229 (47.4%)
Port binding	188 (39.3%)	290 (60.7%)
Scale out by adding more processes	162 (35.6%)	293 (64.4%)
Disposability	167 (38.4%)	268 (61.6%)
Dev/prod parity	216 (43.5%)	281 (56.5%)
Logs as event streams	251 (51.2%)	239 (48.8%)
Admin tasks as one-off processes	171 (43.7%)	220 (56.3%)

Observations:

1. Containers most dominate application code as locus of twelve-factor/cloud-native software design with respect to scale-out (64.4% vs. 35.6%).

   This is consistent with the cheapness and commodification of modern compute resources: We might imagine that, when server resources were more precious, application-level control of scaleout might be required.

   Since, *ceteris paribus,* the application can provision more efficiently, it knows the problem domain more. But it might also reflect the increasing adaptability of modern container orchestration systems: If container-based scaleout were too coarse-grained, applications might still dominate scaleout logic, or be dominated less.

2. Application code and container design roughly share responsibility for explicitly declared and isolated dependencies (49.8% vs. 50.2% respectively).

   We might imagine some hyper-infrastructurized world in which application dependencies are successfully inferred by the environment, rather than declared explicitly in application packages, as in some CORBA fantasyland, or in more heavyweight Java EE implementations. But this does not appear to be the actual world.

3. Application code, however, is tasked slightly more with ensuring process statelessness than container design (52.6% vs. 47.4%).

We would like to know whether programming language affects this distribution — on the reasoning that a heavily functional-oriented language would tend to implement statelessness more naturally, and therefore with less deliberate effort than a heavily object-oriented or multi-paradigm language.

However, we do not have data broken out on languages in an in-container vs. not-in-container usage level, nor do we have enough overall respondents who write in purely functional languages.

4. Developers of high-risk applications — defined as: "bugs and failures can mean significant financial loss or loss of life" — are significantly less likely to rely on container design to store config in environment (51.5% vs. 55.7% and 55.1% for enterprise and SaaS developers, respectively), significantly more likely to rely on application code for scaleout (36.6% vs. 33.2% and 33.7%, respectively), and significantly more likely to rely on container design for admin tasks as one-off processes (62.3% vs. 58.4% and 56.5%, respectively).

   Of these three, a two-fold conjectural explanation for the second seems most obvious: (a) High-risk applications cannot afford coarse-grained resource allocation, and (b) high-risk application code is more likely to be more carefully engineered in general, and therefore more conducive to fine-tuning of resource allocation and deallocation.

IMPACT OF CONTAINERIZATION ON MICROSERVICES

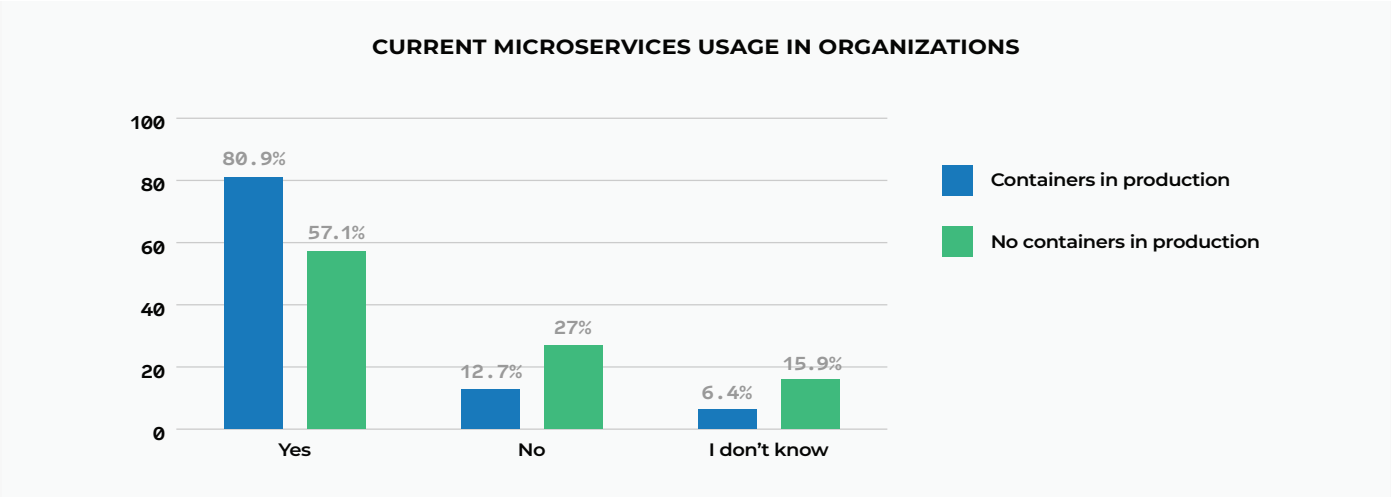
Containers facilitate smaller bounded contexts, which suggests that container and microservice usage might correlate positively — on the assumption that drawing more bounded contexts is generally a good thing.

We wanted to find out if this correlation obtains in reality, so we asked:

Does your organization currently run any microservices?

Results (n=440):

Figure 3



Observation:

1. Respondents who reported that their companies use containers in production are significantly more likely to report that their companies also use microservices (80.9% vs. 57.1%). This strongly supports the intuition that containerization and microservices correlate positively. The causal direction is unclear, however, and would need to be investigated separately.

The positive relation of microservices and containerization is also evident from responses to two more focused questions:

What percentage of these microservices by microservices count runs: {in containers | not in containers}?

And:

What percentage of these microservices by business logic complexity runs: {in containers | not in containers}?

Results (n=320):

Table 4

DISTRIBUTION OF CONTAINERIZATION WITHIN MICROSERVICE ARCHITECTURES		
Location	**By microservice count**	**By business logic complexity**
In containers	70.2%	67.6%
Not in containers	38.6%	41.8%

Observations:

1. Code run in containers dominates microservices both by count and by business logic complexity. This, again, verifies the containers<=>microservices positive correlation intuition, this time within the set of microservices themselves.

   Note: We solicited business complexity estimates as follows: Estimate "business logic complexity" as a best guess. For example, lines of code that format and parse messages — as opposed to decide what the payload should meaningfully contain — count less toward "business logic complexity" than lines of code that implement a pathfinding algorithm.

2. The small decrease in containers' dominance level between "by microservice count" (70.2%) and "by business logic complexity" (67.6%) may suggest that containers tend to perform simpler work than non-containers. Or perhaps, this suggests that microservices that implement complex business logic are less distant from monoliths than microservices that implement simple business logic, on the reasoning that business logic complexity tends toward weakened bounded contexts.

IMPACT OF CONTAINERIZATION ON SECURITY

In terms of first-order effects, it seems obvious that any resource isolation technology would make software more secure. But anyone who has desperately debugged a shockingly horrific production release knows that second-order effects might dominate.

For instance, if containers made patching difficult — and thereby exerted pressure toward out-of-date dependencies, or encouraged laziness in application code — this might convey the assumption that a precisely defined environment might automatically make the environment more safe. (But how would this defend against a SQL injection attack, for example?)

Although pen testing of container usage across the world of software is far beyond the scope of this analysis, we wanted to know at least software professionals' judgment of the impact of containerization on security.

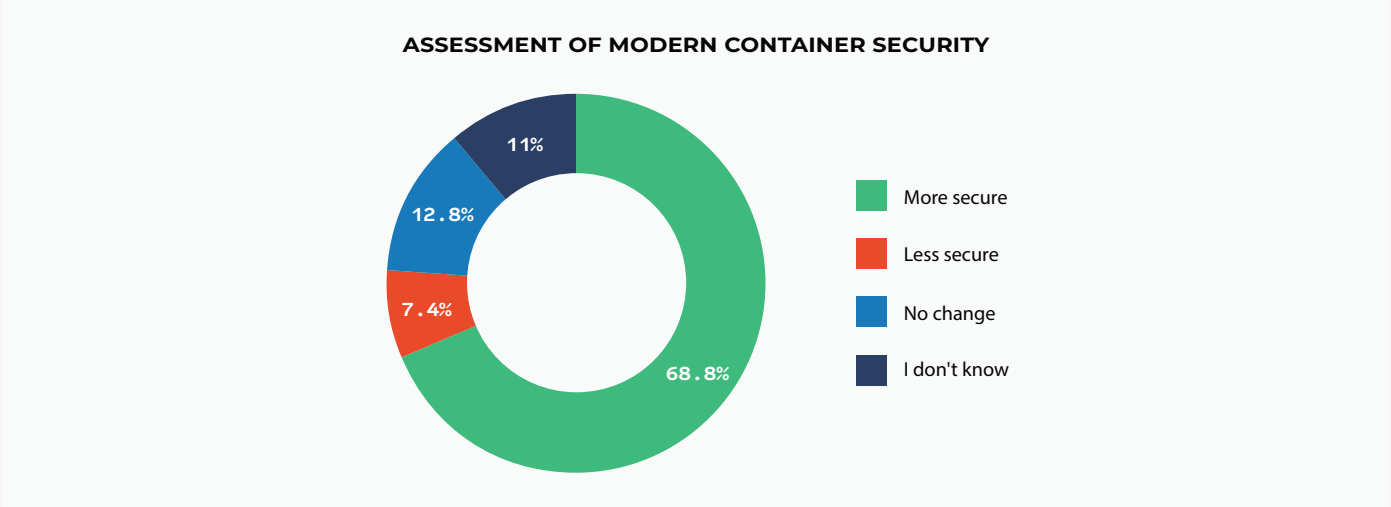
So we asked:

Overall, containerization has made my applications... {more secure | less secure | no change | I don't know}

Results (n=446):

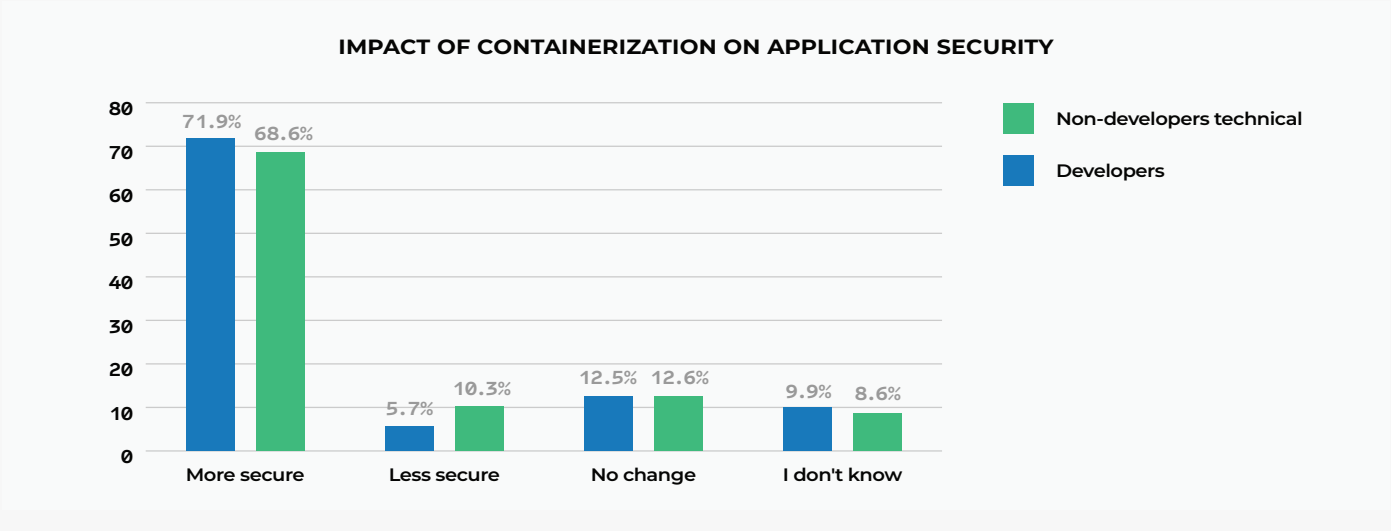
See Figure 4 on the following page.

Figure 4



Obviously, respondents' overall impression that containers benefit security is strong. However, although our research does not extend beyond subjective impressions, we wanted to triangulate impressions across job descriptions at least to help get a better sense of how trustworthy these intuitions are. So we segmented responses to this question between developers and non-developer technical respondents (for definition, see above):

Figure 5



The general sense that containers make software more secure holds true in both groups. But non-developer technical respondents were significantly more likely than developers to judge that containers have made their applications less secure (10.3% vs. 5.7%).

On the assumption that developers are typically farther from security than other technical software professionals (an assumption that we think holds true even factoring in the presumably small number of developers who specialize in securing code), this discrepancy may suggest that developers should rely a little less on containers for application security.

We do not have any data from self-reported security specialists, however; their judgment here would, of course, be the most meaningful, and we intend to solicit it in future research.

Research Target Three: Container and Container Image Creation and Maintenance

Motivations:

1. As every CIO complains or is complained to: Infrastructure is costly. As every developer has thought at least once: Maintaining the stuff my code runs on is just overhead, right?

   As containers play an increasingly large role in application build and rollout, the relation between developers and sysadmins grows more complex: "Toss over the wall and they will make it run" becomes "include instructions for how to catch the thing on the other side of the wall" or "include the design of the wall in the definition of the thing tossed." We wanted to know how and where these handoffs and interminglings occur.

2. The duplication that permits loose coupling — e.g., every container has its own JVM — requires additional maintenance work. Further, the ephemerality of container runtimes and granularity of container image definitions facilitate a create-only approach to container deployment.

   But it can seem overkill to recreate from scratch something that is running just fine simply in order to patch one dependency. In long-running containers, it can be stressful to restart something that may or may not maintain a more externally entangled internal state than the distributed application design requires: One doesn't really know as well as one does in the case of truly ephemeral containers.

IMPORTANCE AND DIFFICULTY OF CONTAINER DESIGN PRINCIPLES

Like the code they house, and everything else, containers can be designed well or poorly. Some principles of good container design have been proposed, and it seems especially important in the developer-empowering, post-Docker world to validate these proposed principles against the set of all pedal-metal strikes. So we asked:

Rate the following principles of container design with respect to importance and difficulty of implementation:

Note: The publication that inspired this question is a list of "principles of container-based application design." But most of the principles are about how to design containers and select contents optimally, so we phrased the question a little differently.

Results (n=430):

Table 5

RANKING OF CONTAINER DESIGN PRINCIPLES

Principle	Importance	Difficulty
One concern per container	4.5	3
Provide APIs for process health, readiness, and/or liveness	4.5	3
Log errors and non-error outputs	4.5	3
Handle sigterm/sigkill and similar events gracefully	4	3.5
Immutability (both spinup and runtime)	4	3
Process disposability, including statelessness/out-of-process state, and/or quick startup/shutdown	4	3.5
Build-time self-containment (all dependencies included in container)	4	3
Runtime self-containment (run with explicitly defined SLAs)	4	3.5

Observations:

1. The most important rated principles of container design follow the "cell" paradigm more familiar from object-oriented programming: "One concern per container," "provide APIs to report interaction-relevant internal state," and "log errors and other outputs" are ways to split up work cleanly and manage private and public attributes.

This may be a useful way to think about containers vs. virtual machines: These "most important" principles are shared between software modules and containers naturally and fit together naturally, as they would not be shared with nor fit together well when applied to virtual machines.

2. The most difficult principles to maintain, by contrast, resemble the kind of work operation systems do — handling sigterm/sigkill vel sim, maintaining process disposability, and limiting runtime with explicitly defined SLAs (i.e., playing the "good participant" role in zero-sum resource management).

3. These two observations together may also imply that most software professionals think like OO programmers, not OS designers, so the OO-related stuff is easy and the OS-related stuff is hard.

    It would be interesting to learn how OS designers judge these principles of container design, but because so few people have designed non-toy operating systems, such research would be better suited to long-form interviews than surveys.

CONTAINER PATCH/UPGRADE PREFERENCES

Containers encourage duplication, like every other decoupler in a dependency-heavy domain, like modern software. In general, the build-time benefit of decoupling must be balanced against the maintenance-time cost of keeping all the copies up to date.

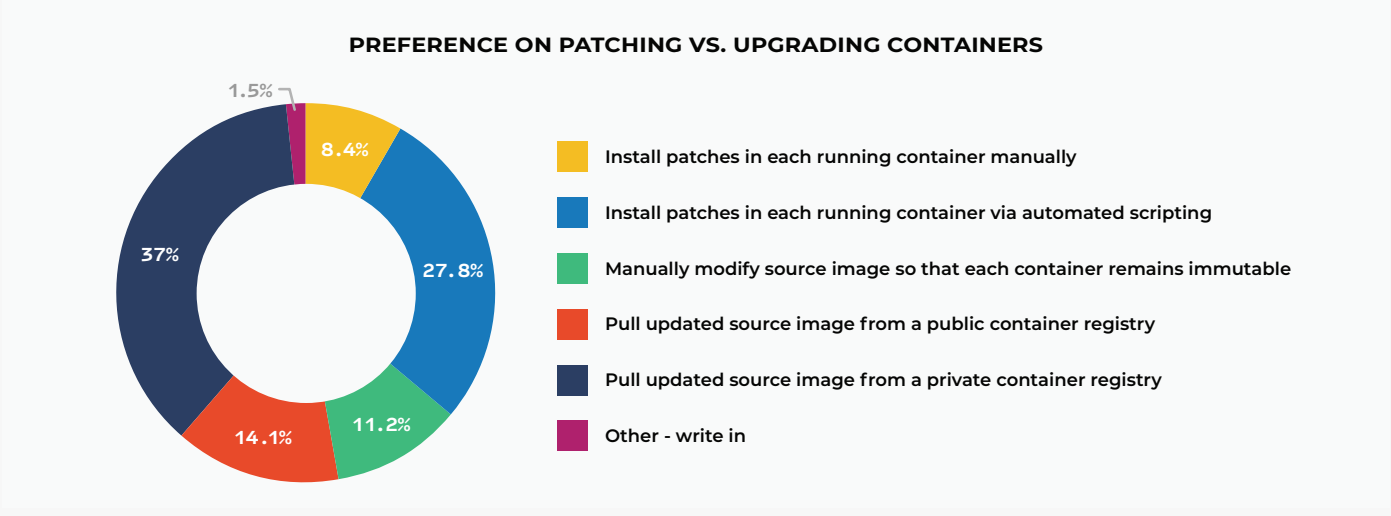
In theory, of course, the potential ephemerality of containers simplifies the duplication task by eliding the distinction between create and update. But if the physical world has state, then in many real-world contexts, create/destroy-only software models are mismatched to their non-software domain — and pure runtime ephemerality must be imposed artificially.

So we wanted to know how software professionals keep containers up to date in practice. To learn this, we asked:

How do you prefer to patch/upgrade most containers?

Results (n=454):

Figure 6



Observations:

1. By far, the most common response was also the most container-esque, in both the "lockdown" and "shipping" senses of "container": 37% of respondents (n=168) prefer to pull an updated source image from a private container registry.

    That is, they both treat containers as relatively immutable and also want the containers to be defined by some central node. This means that software professionals continue to appreciate the strengths of modern, post-Docker containers in registries.

2. The second most common response, however, runs far — though not maximally extremely — in the opposite direction: 27.8% of respondents (n=126) prefer to install patches in each running container via automated scripting.

   The most extreme opposite answer received only 8.4% (n=38) responses: Install patches in each running container manually.

   Presumably, these containers are not effortlessly ephemeral enough to merit create-only treatment. We did not ask how long these containers typically run, but we assume that respondents patch only long-running containers, where "long-running" is defined not necessarily by cesium photon absorption rate but by accumulated state complexity.

   Lambda-style architects might frown on this relatively high number, but further research would be needed to understand the reasoning behind the choice to install patches in running containers, presumably in most cases due to interesting tradeoffs that architectural astronauts might easily miss.

Future Research

Because the scope of this survey was extremely wide, we only scratched the surface in each of our three research areas. We did collect additional data, which we intend to analyze and publish soon, pertaining to:

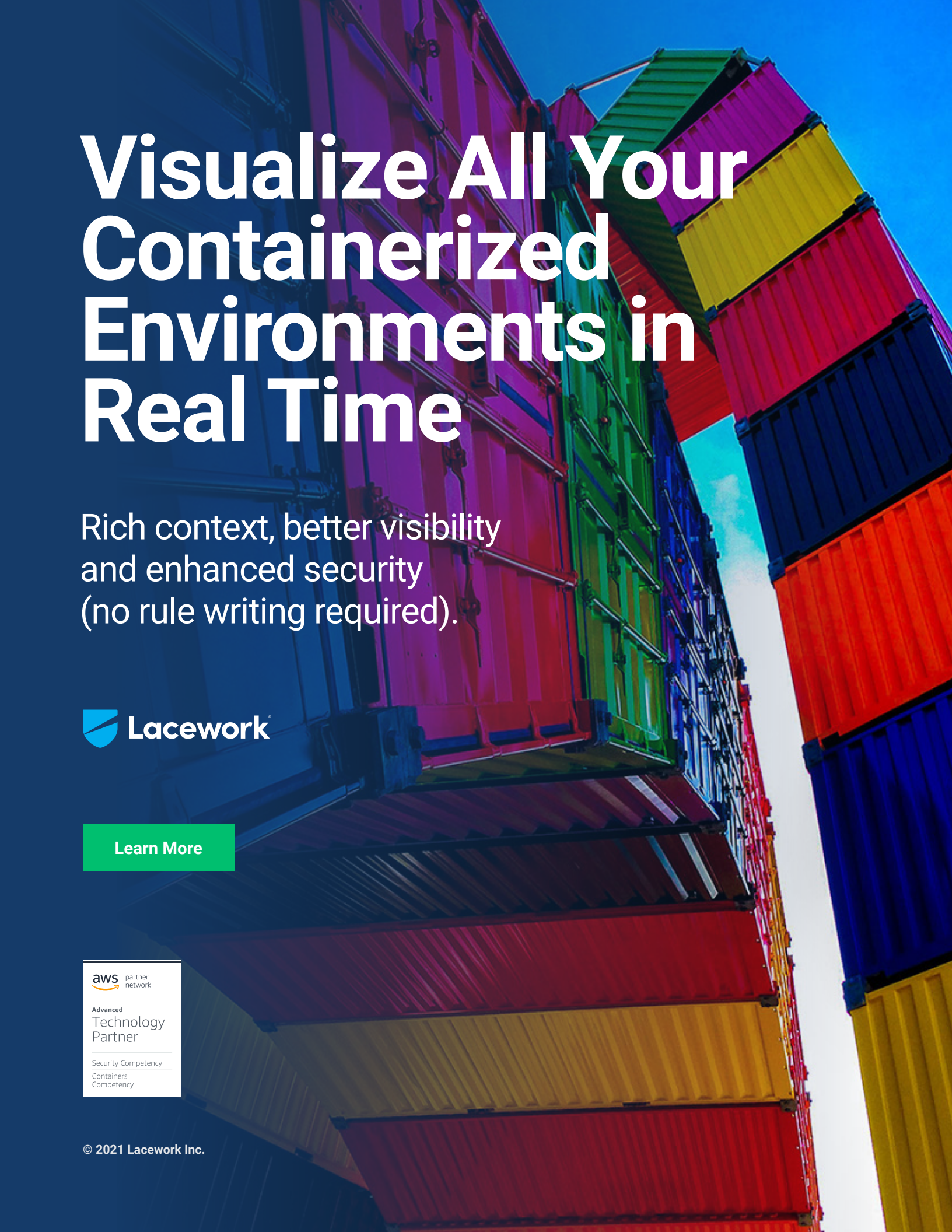
- Automated test design for containerized applications
- Environmental parity across SDLC stages
- Correlations between containerization levels by the amounts of code and computational work
- Reliance on registries vs. designer control of image definitions (e.g., use of "latest" tags in Dockerfiles)
- Usage of specific container runtimes and runtime interfaces
- Secrets management in containerized applications

Some container-related questions were reserved for forthcoming research on CI/CD, with respect to containers' deployment enablement abilities, and Kubernetes, with respect to container orchestration.

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.


Visualize All Your Containerized Environments in Real Time
Rich context, better visibility and enhanced security (no rule writing required).
Lacework
Learn More
aws partner network
Advanced Technology Partner
Security Competency
Containers Competency
© 2021 Lacework Inc.

CASE STUDY

Case Study: Poka

CHALLENGE

As Poka grew, customers wanted to see more evidence that their proprietary operational knowledge would be protected. Also, as Poka added more ECS containerized instances, they struggled to keep up with a growing number of security alerts. Tools they adopted to automate aspects of monitoring and log analysis failed to provide the visibility they needed. “We were missing depth,” says Maxime Leblanc, Poka’s Information Security Specialist. “All those tools are host-based. We were receiving logs from the machines, but not the containers. We weren’t getting the application context we needed.”

SOLUTION

Early on, Poka had adopted Threat Stack to help automate alerts, but they spent too much time analyzing alerts to see which ones posed real threats, and they had to continually adjust rules to whitelist certain types of alerts. Then they discovered Lacework, which checked all of the boxes. It provided full visibility into the inner workings of AWS ECS, included a dashboard to demonstrate compliance to customers, and promised to ease alert fatigue through AI-learning. Poka found that implementing Lacework went very fast. “It was very easy and only took one day,” says Leblanc. He credits the short learning curve to Lacework’s industry-standard user interface.

RESULTS

Having Lacework enabled Poka to accelerate alert analysis and resolution, as well as consolidate their tech stack, by eliminating other tools they were using. This saved them money and simplified their security management workflow. “It’s much easier to centralize our entire alerting system,” says Leblanc. It also reduced alert fatigue and streamlined the alert whitelisting process. “Lacework learns the rules and changes as they come. I can spend my time fixing real issues instead of fixing the whitelisting.”

Poka

COMPANY

Poka

INDUSTRY

Information Technology

PRODUCTS USED

Lacework Cloud Security Platform

PRIMARY OUTCOME

Better application context for alerts, faster alert resolution, reduced alert fatigue, and tool consolidation for improved cost efficiencies.

“We were receiving logs from the machines, but not the containers. We weren’t getting the application context we needed.”

— **Maxime LeBlanc**, Information Security Specialist (SecOps) at Poka

CONTRIBUTOR INSIGHTS

Securing Containerized Workloads

By Boris Zaikin, Software and Cloud Architect at Nordcloud GmbH

The container itself is a small operating system (OS) that can be susceptible to attacks using malicious code. In this article, we explore what is needed to secure your containerized applications; container security tools, rules, and policies; and how to apply common security principles for preventing attacks. I consider Docker and Kubernetes as two industry-leading options for container engines and orchestration.

Docker Container Security Principles and Rules

Below are key security rules and policies for your docker containers and images.

EMPLOY THE PRINCIPLE OF LEAST PRIVILEGE

The principle of least privilege, in relation to containers, can apply in several ways, such as avoiding execution of containers using admin users. Rather, create users who have admin access and can only operate with this particular container.

You can set a specific user identifier while building a Docker image:

```
docker run -u 1010 some_docker_image
```

You can also make groups and add users to them, as shown in the example below:

```
FROM alpine
RUN groupadd -r myuser && useradd -r -g myuser myuser
. . . . .
USER myuser
```

Find more info about container user access in the Docker docs.

USE VERIFIED DOCKER IMAGES AND DOCKER CONTENT TRUST

First of all, you should use official, user-verified and signed images. To find and check images, you can use docker trust inspect. For example: `$ docker trust inspect --pretty google/apigee-mart-server:1.3.6.` Docker Content Trust (DCT) enforces the communication process between the client and registry. The DCT is based on digital signatures that exchange with both parts: client and registry. The DCT verifies images and publishers during runtime — a process based on generation Docker Content Trust Keys, which has several keys used by DST during the interaction between the client and registry.

IMPLEMENT RESOURCE LIMITS

By default, Docker containers do not have resource constraints, so you should set up memory and CPU limits for your containers. For example, you can set up a memory limit to prevent your container from consuming all memory:

```
$ docker run -m 210m nginx
```

We also should limit our CPU resources. For example, we can limit **`nginx`** to use max 0.5 of our CPU:

```
$ docker run --cpus=0.5 nginx
```

This principle is a way to prevent DoS attacks from malicious containers that can consume all CPU and memory. It may also cause a system outage. There is also an option to set up resource limits on a Kubernetes level:

```
apiVersion: v1
kind: LimitRange
metadata:
  name: backend-limit-range
  namespace: backend
spec:
  limits:
  - default:
      memory: 110Mi
      cpu: 500m
    defaultRequest:
      memory: 20Mi
      cpu: 100m
    type: Container
```

The code above sets a `cpu` and `memory` limit to all containers in the back-end namespace. Also, it sets a resource amount that containers are guaranteed to receive.

SECURE DOCKER NETWORKS

Understanding Docker's networking principles, as well as the default Docker network drivers (e.g., `bridge`, `host`, and `overlay`), is essential to securing your containers. By default, one container network stack doesn't have access to another container. However, if you configure the `bridge` or `host` driver for one container to accept traffic from any other containers or external networks, you can create a potential security back door for an attack. To mitigate this issue, you can disable inter-container communication using the docker daemon flag, `--icc=false`.

MONITOR CONTAINER SECURITY

Security monitoring is essential for the detection of malicious code and attacks on your containers, and a proper monitoring tool enables you to identify these issues. The tool should allow you to build a real-time dashboard, as well as set up alerts to send you messages via email, SMS, or even your preferable chat platform. To find vulnerabilities in Docker container local images, you can use docker scan command, `$ docker scan IMAGE_NAME`. CAdvisor is also a pretty powerful tool to monitor your container. Moreover, you can also run it in the Kubernetes cluster.

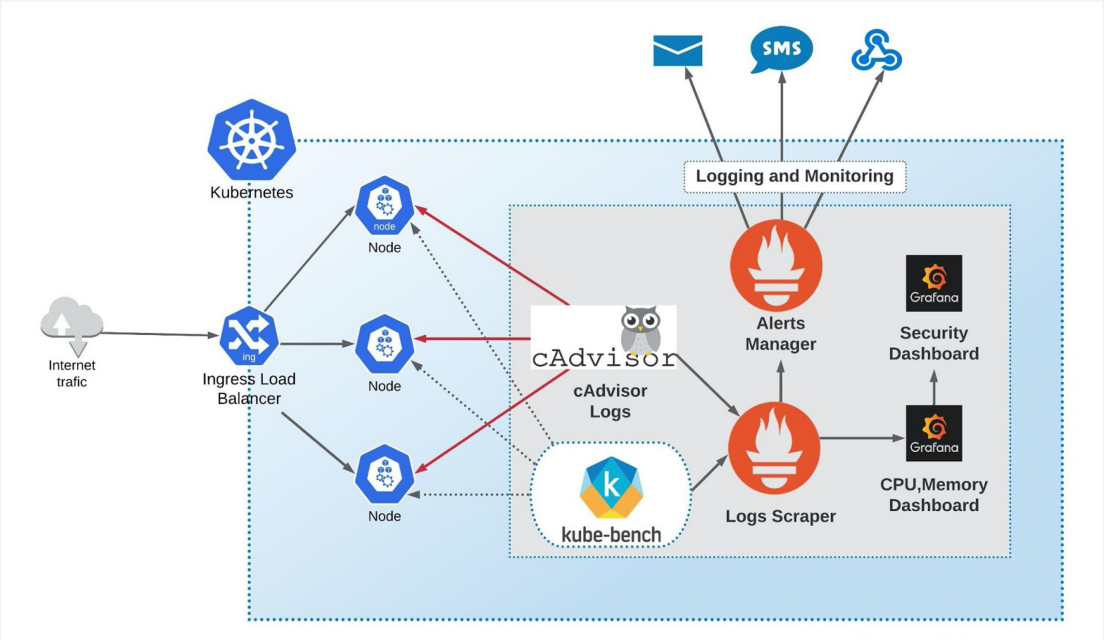
CAdvisor and `docker scan` are simple solutions that apply only to one particular container. For more complex scenarios (e.g., when you run 50+ containers in Kubernetes), you need comprehensive monitoring tools. In Table 1, I've listed some solutions that, in my opinion, are pretty popular. However, there are many more available in the market worth mentioning like Sysdig, Sematext, and Dynatrace.

Table 1

Prometheus	A logging component that "scrapes" container information and puts it into a data source (can be SQL or NoSQL data storage). It also allows users to create rule-based alerts.
Grafana	A framework for building complex UI dashboards that can be easily configured to retrieve data from Prometheus.
Datadog	An all-in-one monitoring tool that contains logging component subsystems and sidecars, plus a complex, interactive UI framework.
Azure Log Analytics	A handy option if your container solution is under Azure Cloud Services, as it's supported out of the box.

I like to "cook" by combining Prometheus + CAdvisor + Grafana. Prometheus is a powerful, open-source option for monitoring CPU, GPU, memory, images, and other metrics, and CAdvisor is quite good at detecting vulnerabilities. Grafana is good at building and configuring dashboards and alerts, and it imports all components together. Also, I involve the security tool kube-bench, which covers vulnerability scanning only but brings an additional layer to your cluster security monitoring.

Figure 1: Kubernetes Security Monitoring



AVOID PUTTING SENSITIVE DATA IN DOCKER IMAGES

All sensitive data should be moved outside of the container. You can manage secrets and other sensitive data using Docker secrets, which allow you to store secrets outside of the image. If you run Docker containers in Kubernetes, you can use Secrets to store your passwords, certificates, or any other sensitive data. Also, use cloud-specific storage for sensitive data — for example, Azure Key Vault or AWS Secret Manager.

Vulnerability Scanning Tools in Docker Images

Vulnerability scanning tools are an essential part of detecting images that may have security holes. Moreover, properly selected tools can integrate into your CI/CD process. Below are some open-source vulnerability scanning tools available today:

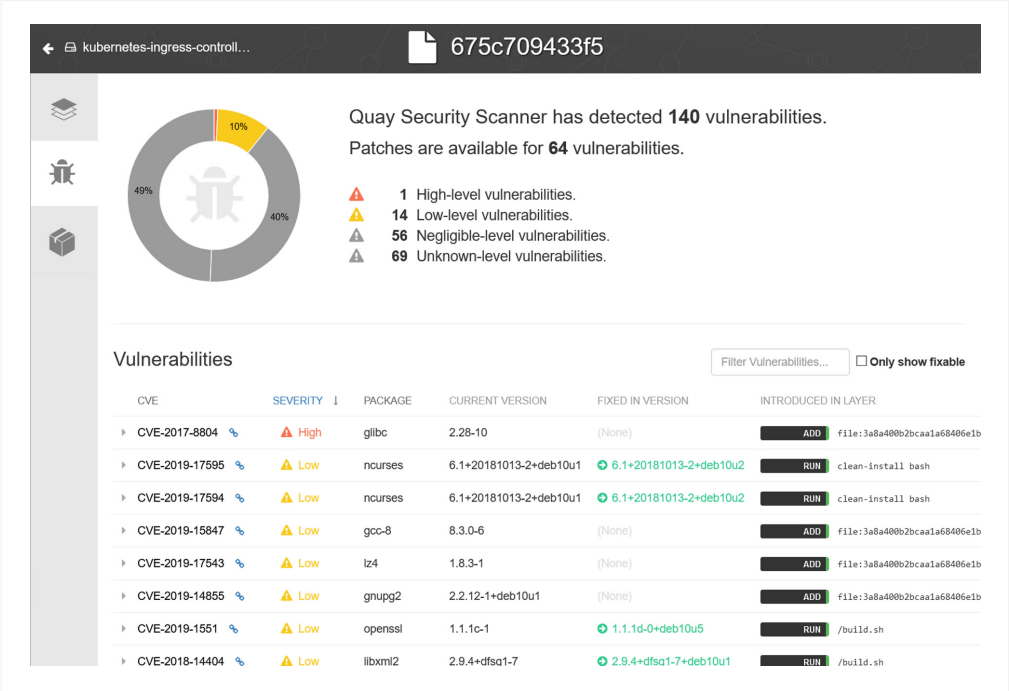
Table 2

Dagda	Uses a static analysis approach to find viruses, malware, and fake sub-images and trojans. It is based on the Red Hat Security Advisories (RHSA) library of existing vulnerabilities.
Anchore	Scans for vulnerabilities and viruses and checks image security signatures. It can sign images after a successful security check. It can run it in Kubernetes, Amazon ECS, and Docker Swarm.
Trivy	Can run in standalone or client/server modes and detects complex vulnerabilities with high accuracy for various OS, including Alpine Linux, RHEL/CentOS, Debian, and Ubuntu.
Clair	Used for static image analysis, supporting images based on the Open Container Initiative (OCI). Clair uses CVE databases to detect vulnerabilities. You can build services for scanning images based on Clair API.
Xray	Based on container artifact analysis. This option makes it easy to add to the CI/CD — the Xray analysis is based on JFrog's vulnerabilities database, which constantly updates.

Docker Registry Security

Protect your images by adding a security layer and using images from protected registries. Harbor has integrated vulnerability scanning based on security policies applying to docker artifacts. Quay, powered by Red Hat, scans images and offers a standalone image repository that you can install and use within your organization. Below you can see how Quay checks for vulnerabilities.

Figure 2: Quay Security Image Scanner



But what if you already use other registries like Azure Container Registry or Docker Hub? You can easily enable Azure Image Defender to scan images and build a detailed security report. Docker Hub's Vulnerability Scanning (available for pro or team plans) starts scanning images when you push it, a mechanism based on Snyk (open-source option).

Kubernetes Security

The topic of Kubernetes security is expansive and requires a separate book rather than an article. So below, I broadly describe a few fundamental rules, in my opinion:

- **Networking and network policies** – Understanding how the Kubernetes networking model works will help you set up proper network communication between pods and pretend to create open ports or direct access to the nodes. Network Policies can help you organize this communication.
- **Secure Ingress and Egress traffic to pods** – You can use NetworkPolicies to deny all egress and ingress traffic and then start opening. You can also use *service mesh* like Istio, which adds additional service layers, automates traffic, and helps monitor. However, you should be careful to use the service mesh as it may add further complexity.
- **Transport security layer** – Use TLS for communication between Kubernetes cluster services; you should enable TLS if it's disabled.
- **Restrict access to the kubelet** – Enable the authentication and authorization to use this tool. Only admins should have access to the kubelet.
- **Restrict access to Kubernetes Dashboard** – Use role-based access controls (RBAC) and follow the principle of least privilege to set up.

Note: The security principles mentioned in the docker container section also apply to Kubernetes clusters.

To detect security and misconfiguration issues in Kubernetes, you can use the following tools:

- Kube-bench (checks security based on CIS Kubernetes Benchmark)
- Kubeaudit
- Kubesect.io

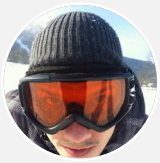
Docker and Kubernetes Attacks Examples

The *man-in-the-middle* (MITM) attack is widespread in Kubernetes and Docker. This attack includes additional malicious parts between a component that sends data and a component that receives this data. It can be a fake container, service, middleware, or even human. For example, CVE-2020-8554 is a vulnerability that allows attackers to gain access to the cluster's service by creating a `clusterIP`.

A *cryptojacking* attack allows an attacker to run malicious code in order to use a PC's CPU, GPU, and memory for mining cryptocurrencies. Example: CVE-2018-15664 gives access to the docker system with root permission.

Conclusion

The topic of security is quite essential and complex, especially in the Docker and Kubernetes world. This article contains a few crucial recommendations that everyone should keep an eye on. Also included are tools that can help you implement these security recommendations for your project based on Docker and Kubernetes.



Boris Zaikin, Software and Cloud Architect at Nordcloud GmbH

@borisza on DZone | @boris-zaikin on LinkedIn | @boriszn on GitHub

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, Azure Service Fabric. My areas of interest include Enterprise Cloud Solutions, Edge Computing, High Loaded Application, Multitenant Distributed Systems, and Internet-of-Things Solutions.

CONTRIBUTOR INSIGHTS

Adapting Your CI/CD Pipeline for Containers and Hybrid Virtualization

Traditional Pipeline to Modern CI/CD Pipeline Transformation

By Ajay Kanse, Lead DevOps Architect at Cognizant

Why "Change" Is Needed

As I write this article, I started thinking about my journey within the IT industry — some of the delivery models, processes, production incidents, and tools I encountered. I will relate it briefly to the phrase, "Change is the only constant."

If you worked in IT from 1999–2000 or prior, you might have managed your releases using packages or files stored on network share. In the past, developers coded locally, and there was no common place to store source code — development was a more sequential process with monolithic applications. As the IT industry evolved, source code version control tools were introduced. Initially, these tools were mainly used to store source code and then used for version control with client server and distributed models which made concurrent development and merging reality.

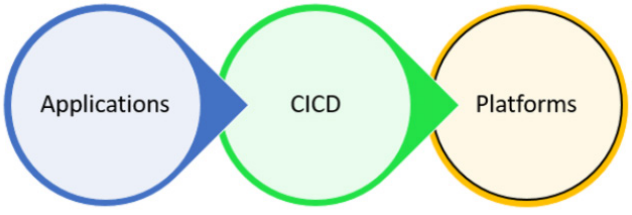
Under the Waterfall model, projects underwent multiple lifecycle phases with rigid gates, and there was very little room to go back and change the requirements. With monolithic deliverables come additional challenges of managing and maintaining long tail branches, developing test cases of the same volume, complexities with merging code at the end, and migrating packages through multiple environments. And making it to production was really painful. These challenges often came at the cost of organizations' time, effort, and money.

At the same time, continuous integration (CI) and Agile methodology started picking up to meet the shifting demands of customers and rapid changes in technologies. Many enterprises jumped on this bandwagon to improve their delivery models and processes, focusing more on DevOps to integrate various silos within the organization and on a common vision. The same situation is occurring with cloud and container adoption, which had made continuous integration and continuous delivery (CI/CD) even more crucial for software delivery.

One of the values in Agile is "Responding to change over following a plan." This is possible only if you have a process that is reliable, repeatable, scalable, secure, and sustainable, as well as provides immediate feedback and opportunity for improvement. And CI/CD accomplishes all of this.

A Brief Review of CI/CD and CD

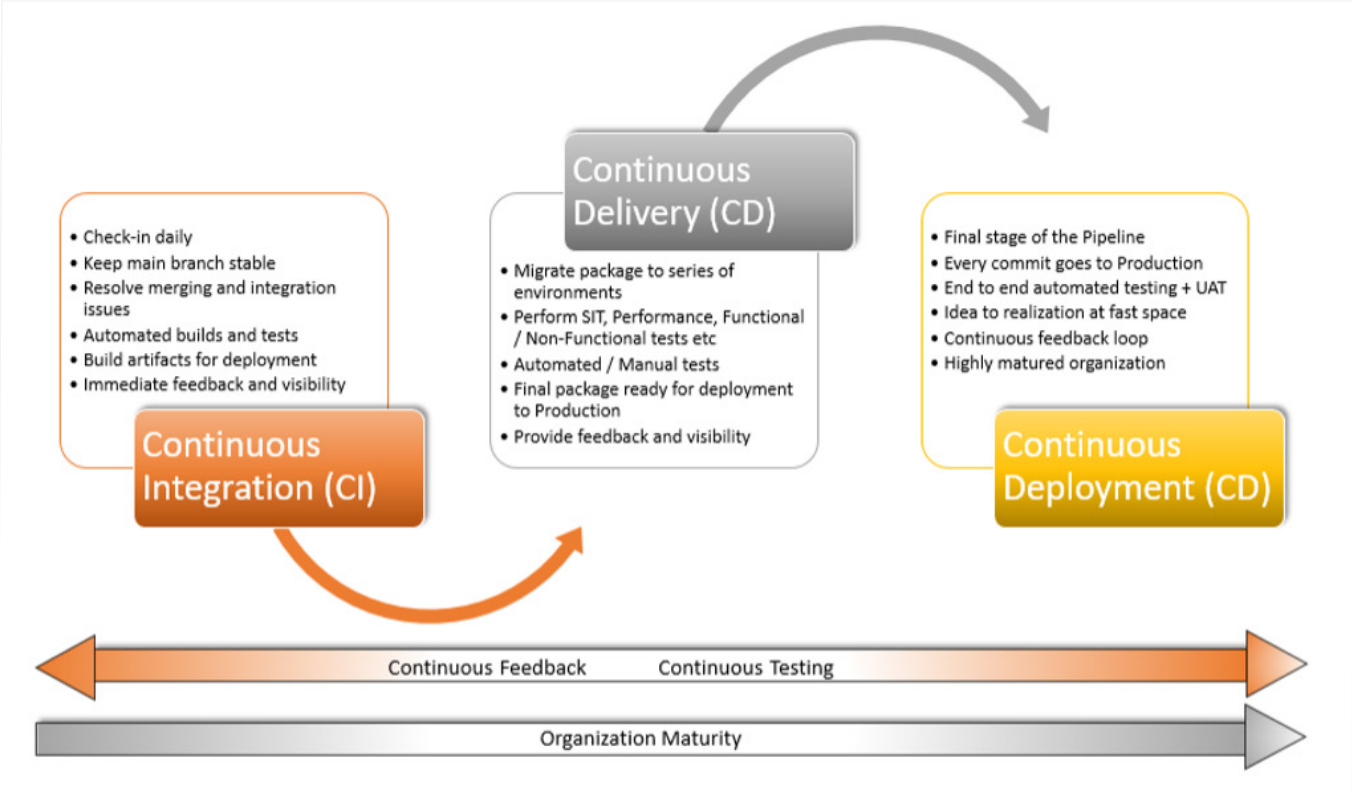
With the wide acceptance of DevOps practices, sometimes CI/CD is loosely considered DevOps; however, that's not the case. CI/CD is a critical and central piece of DevOps.



CI/CD is a bridge between applications and target platforms that aids to deliver apps in a faster, reliable, and consistent way.

Figure 1 illustrates high-level characteristics of continuous integration, continuous delivery, and continuous deployment. Note that continuous delivery keeps artifacts ready for deployment to production; however, continuous deployment takes it one step further to deploy every change to production. There are no more scheduled production deployments with continuous deployment.

Figure 1: CI/CD and CD characteristics



Where Do You Begin Transforming Your CI/CD Pipeline?

This is a very common question when you start thinking about how your existing pipeline can be adapted for containers. The recommendation would be to start small and use an incremental approach. Assuming you have an existing pipeline, which may be running in the cloud or on-prem VMs, below are some things to consider:

- **Collaboration with various stakeholders** – Support from management and decision makers is critical to achieving the goal and applicable for any changes that are required in the organization. Various departments within an organization should be on the same page with objectives, benefits, and industry trends for containers.
- **Learn container technology and the ecosystem** – Understanding container technology, usage, and the ecosystem (Docker) will help determine some basic requirements for setup, such as container runtime, images, registry requirements, and security concerns.
- **Identify less critical applications that can be used for proof of concept (POC)** – Select less business-critical applications to begin with — those that are simple enough to set up in the pipeline yet go through all stages of the pipeline (e.g., build, test, code quality checks).
- **Container tool selection and pipeline setup** – As you start getting more information about container technology, you'll come across multiple tools. Understand pros and cons of using these tools. Based on experience, I recommend looking into Docker or Podman. Consider security mandates from the organization while using these tools to make choices. As you set up pipelines using containers, continuously improve upon your implementations.

Figure 2: Pillars to adapt a CI/CD pipeline to containers

One use case for starting a POC would be to use Docker images as build nodes (runners, worker nodes) for building applications, as opposed to standalone build machines. External registries such as Docker Hub can be used as an image repository to source builder images. Based on the technology requirements, select appropriate images and versions for the build step. Based on the build requirements for app, you might need additional software in images and to create organization-specific builder images with all tools packaged.

The major benefit going forward is that you don't need many standalone build machines with specific configurations. Instead, immutable builder images will be used for build infrastructure. Then you can expand the POC with more technologies, standardize builder images, and tighten security. As you gain more expertise with containers, you can acquire additional benefits by moving the entire pipeline and other tools (e.g., artifact repository, image registry, scanning tools) onto a container platform such as Kubernetes.

In the sections below are two example CI/CD workflows that show maturation of adapting an existing pipeline to use for containers.

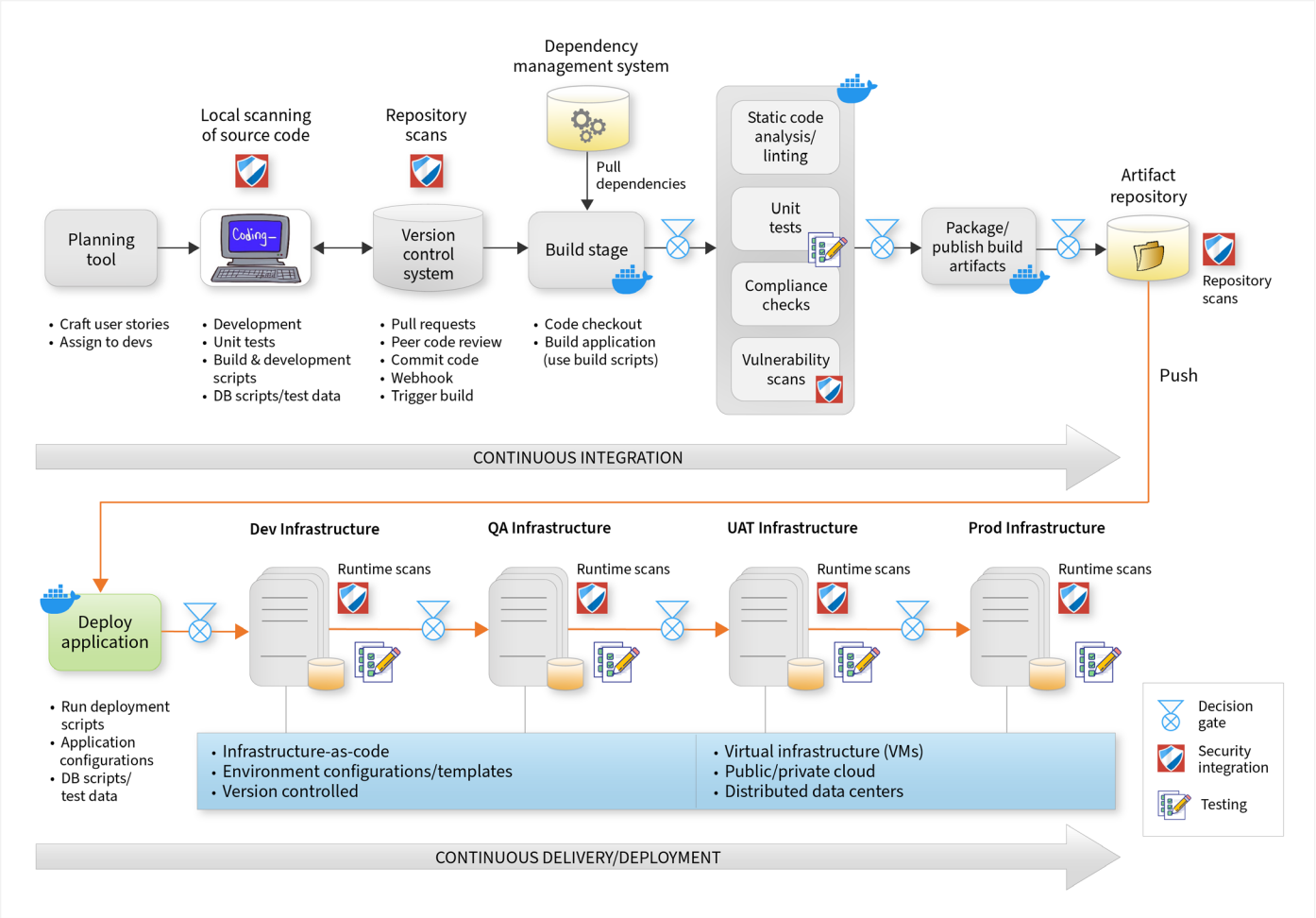
DevSecOps Pipeline for Hybrid Infrastructure: Cloud/On-Prem/Container Platforms

Figure 3 includes basic stages in a DevSecOps pipeline. The stages and sequence may vary based on an organization's requirements. It is important to note that CI tools and various stages are running inside containers. The entire CI/CD infrastructure can run on the Kubernetes platform to support non-containerized applications. With minimal or no changes to applications, it can take the benefit of immutable Docker images and flexible infrastructure with efficient use of resources in the pipeline.

The applications can be deployed to an on-prem data center, VM infrastructure, or the cloud. In order to deploy applications to a container platform, changes are required from the application side with respect to configurations and building images. An example workflow that is more suitable for containerized applications is shown in the GitOps Pipeline section below.

DevSecOps considers security and testing integral parts of the pipeline, and that they must be performed at various stages to minimize manual interruptions in the delivery pipeline. Testing and security are separate topics all together and are just touched upon in this article.

Figure 3: Example of a DevSecOps pipeline



In most cases, a single tool can manage CI and CD in both stages as long as it provides required controls for audit, security, and compliance. Some organizations use application release automation (ARA) tools to manage deployments — they provide benefits including:

- Tight integration with CI tools
- Centralized inventory and management of deployable packages
- Declarative and plugin-based deployment approaches
- Minimal scripting and platform-agnostic deployment steps
- Visibility and traceability of deployments across environments
- Ease of rollout and rollback processes
- Support for multiple target platforms, container platforms, and clouds

GitOps Pipeline With Integrated Security for Container Platforms

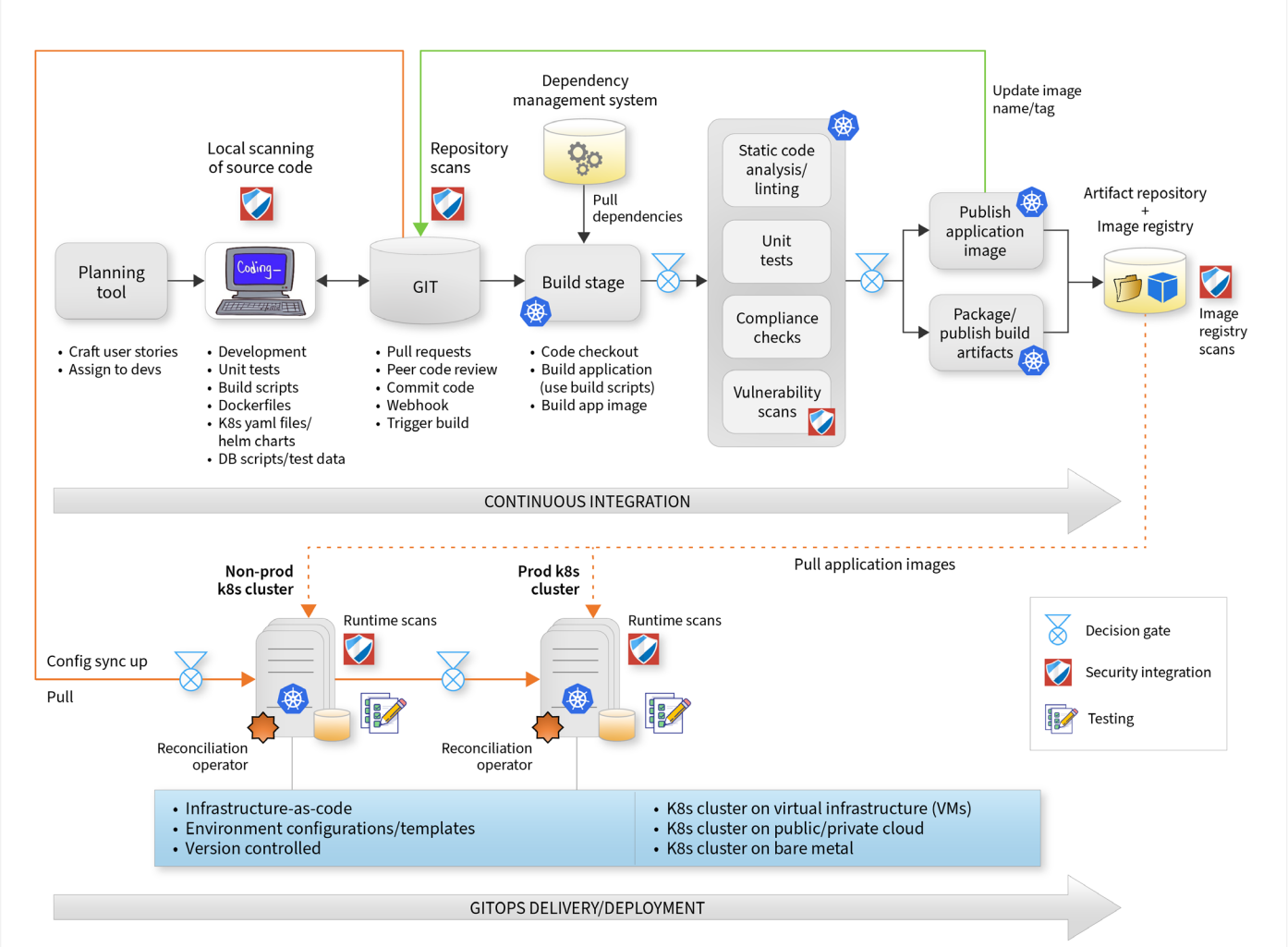
With the rise of containers and cloud-native transformation strategies, many organizations want to adopt containers as means to deploy applications. Containers help organizations realize ideas into production in a much faster, more consistent way irrespective of the cloud or hybrid platform they choose. More details around benefits can be found in the Kubernetes Documentation.

The GitOps model by Weaveworks introduces a container-native and cloud-agnostic workflow that uses Git as a source of truth, not just for application changes but also for infrastructure changes to meet the desired state of the Kubernetes cluster using its native capabilities.

In order to use the GitOps model, everything (application, infrastructure, network, security configurations, etc.) should be treated as code and checked into Git. Any changes in Git will be reconciled using Kubernetes controllers. Kubernetes is an ideal platform for reconciling and making sure the desired state of the system provided in Git matches the actual state. If they don't match, controllers take appropriate action to sync them. This approach also eliminates the need to have scripts for setting up environments. All changes are centrally managed in Git using Git-specific workflows, which helps ease tracking changes.

GitOps uses a pull model (i.e., pulls changed configurations from Git) as opposed to a push model (i.e., configurations are pushed to target platform) to trigger deployments. The pull model is more secure because it just needs read permissions from the version control system rather than giving the CI system permissions for the target platforms. While GitOps handles the CD phase of the pipeline, the CI phase still needs to be implemented and needs updates to Git, which are detailed in the workflow below. Additionally, security must be tightly integrated within the pipeline.

Figure 4: Example of a GitOps pipeline



In GitOps, no changes should be applied to infrastructure and/or applications outside of Git. Kubernetes requires the desired state to be defined in a declarative format. While there are no specific guidelines for how to manage configurations across multiple environments in Git, one of the recommended practices is to use a separate configuration repository for every application. This stores manifest (`yaml`) files for Kubernetes resources (e.g., deployments, services, ConfigMaps). Having separate config repos helps make updates to manifest files — mainly image name/tag and config changes — without making any commits to the application repo. Secrets should not be stored in Git. Explore various options for secrets management in this blog.

INTEGRATED SECURITY

Security should be inherent to the pipeline and aid developers to increase the confidence in code quality. To set up policies and controls that do not block the workflow:

- Integrate developer IDEs with security tools for immediate feedback instead of waiting for CI/CD to report security issues.
- Set up a version control system with continuous scanning (source code and dependencies).
- Perform build-time scanning on the source code and final artifacts (e.g., images, binaries).
- Enable static scanning at the artifact repository and/or image registry level.
- Enable runtime scanning on the target platform for containers scans, host scans, and other security vectors.

Conclusion

Technology is changing fast, business models are transforming, and myths of the past have become reality. In order to meet the demands of customers and provide exceptional service, it is important that organizations strive to dissolve internal boundaries. CI/CD plays an important role to bridge the gaps between realizing ideas at the right speed and meeting customer expectations. Traditional CI/CD workflows have to be modernized to meet these changing needs.

Ajay Kanse, Lead DevOps Architect at Cognizant

@DevDZone33 on DZone | @ajay-kanse on LinkedIn

With a development background and performing various roles such as Configuration Manager, Project Manager, and now DevOps Architect, Ajay Kanse has witnessed organizations' transformations to Agile and DevOps culture. He has extensive experience with enabling CI/CD practices at the enterprise level and setting up ARA tools. He has passion toward Cloud, container technologies, and DevOps. His certifications include CKA, CKAD, AWS SA-A, CloudBees Jenkins Platform engineer, and IBM UCD professional.

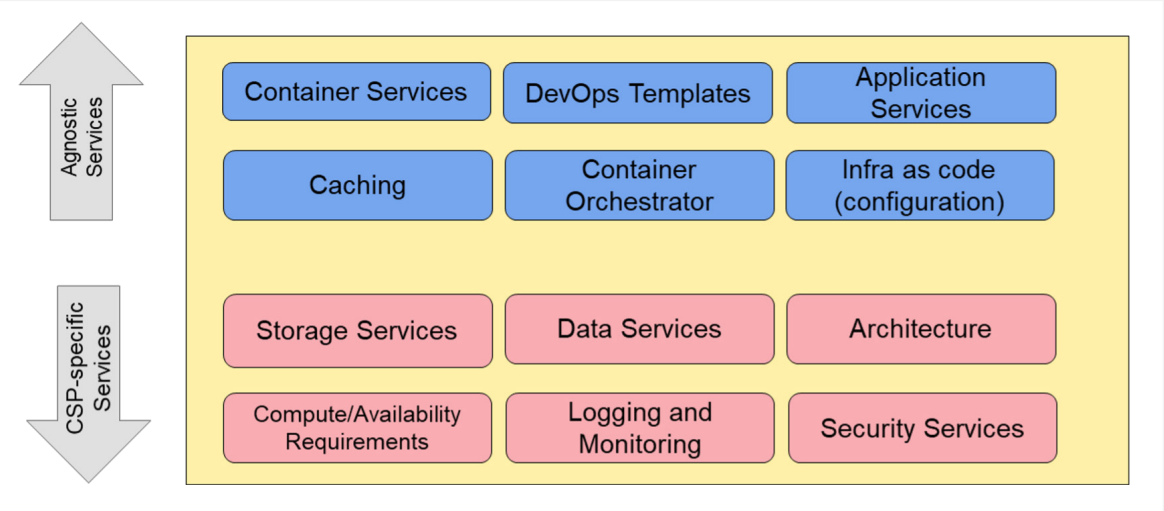
CONTRIBUTOR INSIGHTS

Creating Cloud-Agnostic Container Workloads

By Dr. Magesh Kasthuri, Distinguished Member of Technical Staff at Wipro Ltd.

Large enterprises rely on a cloud-first strategy as their key strategic approach for digital transformation, according to Gartner. During said transformations, the cloud architect is a strong influencer on the cloud roadmap, and cloud-agnostic architecture is important to provide a neutral cloud service provider (CSP) for the enterprise. In such an architecture, one has to separate CSP-specific services (which cannot be ported across CSPs) and standard, reusable services (which can be ported across CSPs), as shown in the figure below.

Figure 1: Cloud-agnostic components



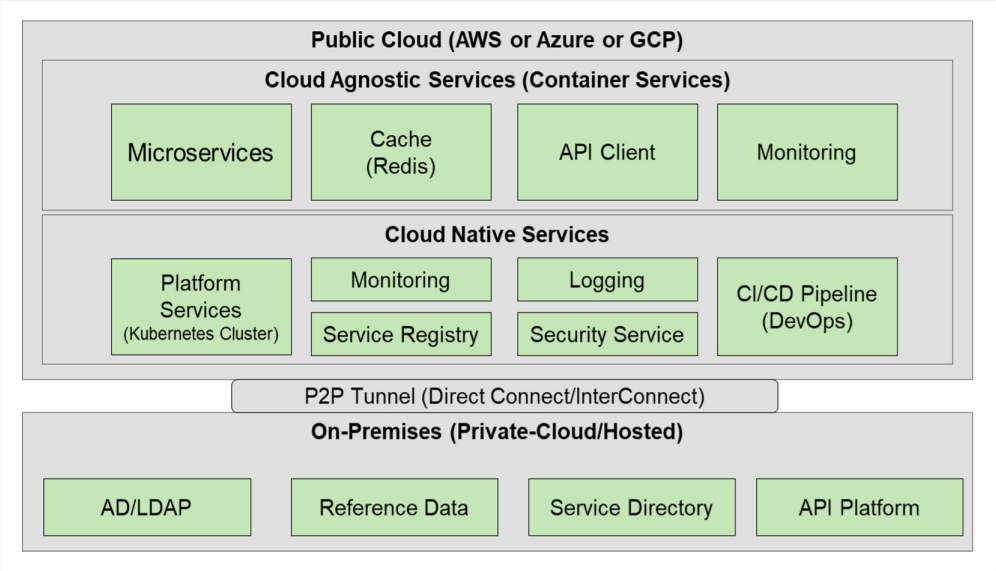
Keep in mind these best practices for designing a cloud-agnostic architecture:

- Use containers for microservices (Docker, Kubernetes) — e.g., Google Kubernetes Engine (GKE), Amazon Elastic Kubernetes Service (EKS), or Azure Kubernetes Service (AKS).
- Use a cloud-neutral solution like Terraform templates to configure container orchestration in GCP or EKS.
- Use automation as much as possible to create agnostic services like infrastructure provisioning using Terraform templates.
- Avoid vendor-specific services (vendor lock-in) like native storage, networking, and using containers.
- Use Spring Boot microservices to be loaded in cloud storage in Azure, GCP, or Amazon Web Services (AWS).
- Automate configuration (infrastructure-as-code [IaC]) to reduce CSP dependency, including networking, storage, and API configuration.
- Integrate cloud-native directory and authentication services (e.g., Cognito, Google Directory, Azure AD).
- Use templatized application setups for microservices, build, test, and deployment activities.
- Maintain the flexibility to integrate with API platforms using native API gateway services.

Application Architecture for a Multi-Cloud Pattern Solution

When developing a multi-cloud solution approach, we generally tend to choose containerized solutions, since they are agnostic by nature (if we use a generic Kubernetes/Dockerized approach). In general, multi-cloud is preferred by large enterprises for reasons like disaster recovery, cost optimization, and legacy app migration efficiency.

Figure 2: Example multi-cloud app architecture



When choosing a containerized approach, the application architecture for a multi-cloud pattern solution will have two major components. The first is a cloud-agnostic service, which is generally a third-party service like Kubernetes, Redis cache, and Apigee API management, so they can be ported to any cloud easily. The second is a cloud-native service, which is tightly coupled to a cloud platform, like the container registry (AKS, EKS, and GKE). These native services can be easily configured or integrated to work with cloud-agnostic services.

This kind of integration helps build a resilient and agile multi-cloud solution. Additionally, if your multi-cloud solution needs to support a hybrid solution, you can use P2P tunnelling like InterConnect in GCP, DirectConnect in AWS, or ExpressRoute in Azure to reduce network latency.

Characteristics of Cloud-Native Services to Design a Cloud-Agnostic Solution

The Cloud Native Computing Foundation (CNCF), an open-source software foundation, is dedicated to universal, sustainable cloud-native computing. They expect four characteristics for Cloud Native (CNF) to be agnostic: containerization, microservices, lifecycle management with DevOps, and CI/CD. Cloud-agnostic design helps develop cloud applications and infrastructure provisioning (using IaC) for any cloud or multi-cloud deployment. This multi-cloud-compatible design can be a multi-tenant public cloud or hybrid cloud solution with public and private cloud platforms.

Containerizing software and using native container orchestrator tools (Docker, Kubernetes) to build images and register and deploy containers ensure those containers are portable and work in multi-cloud environments. Most CSPs natively handle container images well with continuous runtime image protection to ensure application security and that infrastructure is shared during deployment, reducing the overall cost for app development and deployment. Cloud agnosticism helps avoid vendor lock-in and ensures software can be migrated to any CSP via the flexibility of IaC and security-as-code.

Cloud-agnostic solutions are:

- Easy to deploy to any cloud and offer clear guidance to cloud usage.
- Remain vendor neutral to cloud-native integration.
- Portable, allowing workloads to shift from on-prem to any cloud or between clouds.

You can also choose a parallel multi-cloud solution design so the same app can run on multiple cloud providers for high availability based on region or edge location. Cloud-agnostic services natively have a robust security design and enhanced risk management, and they are flexible, performant, and highly scalable. They also have uniform implementation of Cloud Service Expense Management (CSEM) across different CSPs without duplicating the effort in cloud cost management-related activities.

PRINCIPLES OF A CLOUD-AGNOSTIC APPROACH

In a multi-cloud scenario, cloud-agnostic architecture is essential to accommodate the different environments and reduce duplicate work across CSPs. Based on various industry standards, we can conclude that a cloud-agnostic approach contains seven common principles:

Table 1: Seven principles of a cloud-agnostic approach

System immutability	Use IaC so that everything in environment setup is immutable. Do not run any script outside this in any stages.
Automate everything	Automate builds, tests, validation, and deployment as much as possible to avoid human intervention and error.
Flexible in disposability	Everything in the setup should be temporary and short-lived. Regular updates to the environment avoid unnecessary service failures.
Externalize configuration	Make sure changeable configurations (e.g., passwords, file locations/paths, credentials) are stored externally from the image/binary to make life easier.
Constant telemetry	Enable logs, event streams, auditing, and alerts with dashboards so that any functional, operational, or security issues can be monitored up front.
Delegated governance	Define any SLAs and BLAs for managed services. Enable this with strict governance in the DevSecOps lifecycle.
Lifecycle management	Make the lifecycle management independent by decoupling activities like upgrading, scaling, and deploying services.

FnProject: A Container-Native Serverless Platform

In a multi-cloud architecture, creating functions-as-a-service (FaaS) is challenging since we can write common code across platforms, but setting up environments and resources for FaaS for a specific CSP is time-consuming. As an open-source container-native serverless platform, FnProject can be used to design a cloud-agnostic FaaS, which reduces the time and cost of configuring functions across different CSPs. FnProject users also don't need to worry about installation, configuration, or management, and FnProject uses a cost-effective pay-as-you-use model based on elastic resource utilization.

Figure 3: Fn architecture

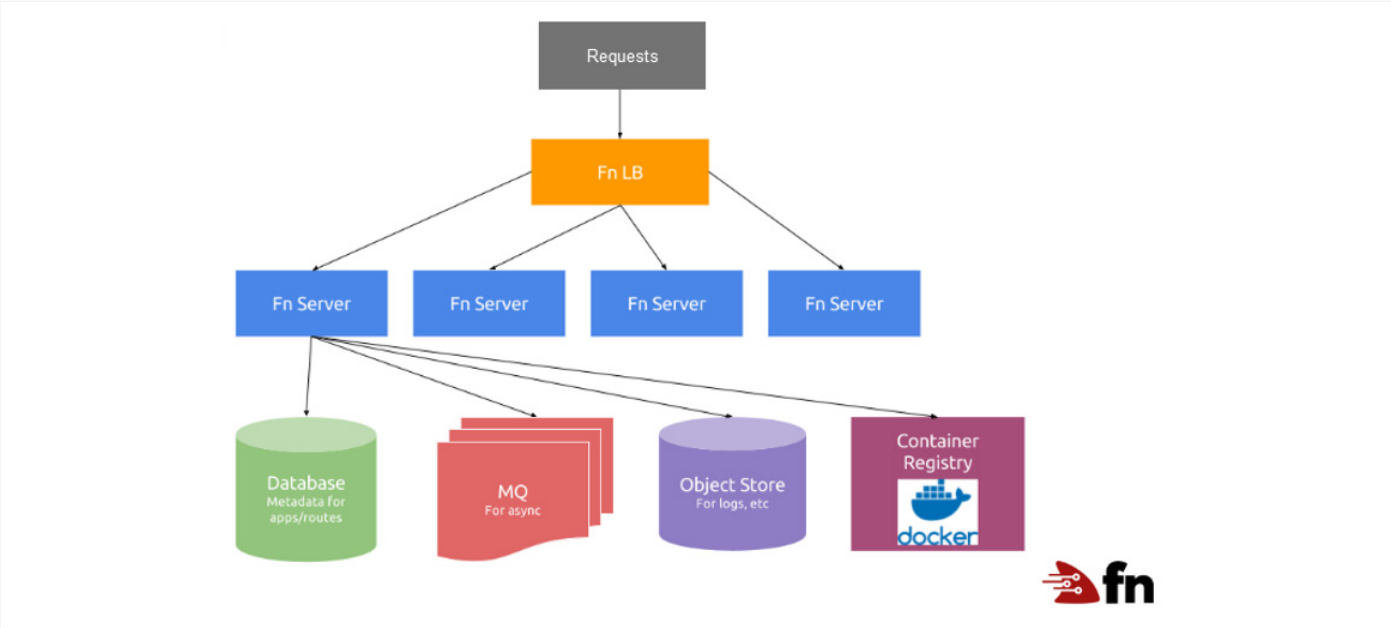


Image source: FnProject Documentation

Figure 3 shows how function code runs on FnServer instances while a load balancer handles user requests on top of it. Each function is considered its own Docker container, which is lightweight and uses a built-in data store for metadata handling and caching, as well as an object store for logs and configuration management.

FnProject uses Function Deployment Kits — integrated solutions with helper libraries to seamlessly deploy FnProject containers — and it currently supports Java, Python, Node.js, Go, and Ruby. For other languages like C# or Scala, you must use your own Docker container solution to create the image and deploy manually. A container-agnostic FaaS framework is useful for developing solutions with high agility and the flexibility to work within different developer communities.

Using Functions-as-a-Service for Agnostic Serverless Functions

For cloud development, microservices are a handy way to weave in business logic while isolating service integration. Most CSPs offer a service to help with this via a functions-as-a-service (FaaS or serverless) offering. FaaS is a managed service where developers don't have to worry about the infrastructure provisioning and deployment hurdles. With AWS Lambda, Google Functions, Azure Functions, or Logic Apps, developers can quickly develop and deploy FaaS.

From an architect and developer perspective, there are standard criteria for choosing the FaaS solution approach. Per Forrester New Wave™ research 2020, these criteria are listed as below:

- **Developer experience** – Provides easy development and deployment options for developers with native cloud support.
- **Programming model** – Since functions can be developed in any programming language, support for more programming models is important. For example, you can develop FaaS using Java, JavaScript, Python, or C# programming constructs.
- **Runtime execution environment** – A proper environment to configure build, deploy, and package FaaS along with stateful and stateless service workload support.
- **Observability** – Support for integration with native monitoring and management services to handle logs, metrics, and debugging.
- **API and event integration** – Support and flexibility to integrate with other applications, services, and workloads, as well as to handle request/response transformation during service invocation.
- **Security features** – Native support to integrate with security features like DDoS attack protection and IAM roles and policies.

- **Extensibility** – FaaS support for integration with any cloud infrastructure type (TPU, GPU, or FPGA) and the ability to extend technical code development/deployment for FaaS.
- **Vision** – A clear view of what CSP environment workload support is available and how it can support current and future developer needs.
- **Roadmap** – The release strategy, plan for extension services, preview services, and multi-regional support and feature expectations in future updates.
- **Market approach** – Support for multiple industries and different workloads, as well as a clear callout of features, limitations, and a go-to-market approach.

As per Forrester Wave™, AWS Lambda leads the FaaS taxonomy field with more agility and scalability in features, as compared to other leading cloud providers.

OpenFaaS is another popular open-source FaaS framework; it has a built-in function store that provides function templates. These templates can be reused in a community-driven model and allow developers to quickly and easily make functions or microservices from any tech stack, including GoLang, Java, Python, C#, Ruby, Express.js, Django, .NET Core, and even open binaries like JARs. They can be used to build a serverless app for any CSP within a few minutes. There is no need to worry about the infrastructure requirements, service monitoring/management, or OpenFaaS module deployments, which makes cloud transformation projects much more convenient.

Dr. Magesh Kasthuri, Distinguished Member of Technical Staff at Wipro Ltd.
@magesh678 on DZone | @magesh-kasthuri on LinkedIn

Dr. Magesh Kasthuri did his PhD in artificial intelligence and the genetic algorithm. He has over 20 years of IT experience and currently works as a Distinguished Member of Technical Staff at Wipro Ltd. He has published series of articles and whitepapers in various journals and magazines. He writes daily technical blogs with the hashtag #shorticle on LinkedIn, covering topics like AIML, Cloud, Blockchain, and big data. This article expresses the author's view and does not represent the views of his organization.

CONTRIBUTOR INSIGHTS

Containers With AWS Lambda

How AWS Used Containers to Make Lambda Even More Accessible

By James Sugrue, CTO at Over-C Technology Ltd

At the 2020 annual re:Invent conference, when Amazon Web Services announced container image support for AWS Lambda, it changed how we think about the possibilities of building serverless functions. Anyone who has built a function in AWS Lambda will enjoy the simplicity of the process, without being too concerned with runtime.

But there's another side to the coin — some developers **want** to have control over the container image that is used for their application. In this article, we'll explore the implications of this new way of building Lambdas, and how it compares to other serverless container-based options, namely AWS Fargate.

Unlocking AWS Lambda to a Wider Audience

With container support, developers can now package their application code using Docker and deploy that code to AWS Lambda.

This is a big change compared to the previous approach where the code for your Lambda would have been uploaded as a zip package in a particular structure. While tools like Chalice and the Serverless Framework have insulated developers from some of this, they also introduced new frameworks and approaches to the standard application development lifecycle.

Once you go beyond basic use cases, Lambda developers would need to use layers when dealing with custom runtimes, or in dealing with large dependencies. Each layer would have a maximum unzipped size of 250MB, and a function can only use five layers at a time; this gives you 1.25GB in total. The size limit for your container image is 10GB, a big step up from the previous limit of 250MB for Lambda functions. This makes Lambda suitable for a wider range of applications that have large dependencies, such as machine learning models.

AWS provide base images to get you started, covering all of the existing supported Lambda runtimes, as shown below:

Runtime	Tags	Repository Links
NodeJS 14.x	14	Amazon ECR DockerHub
NodeJS 12.x	12	
NodeJS 10.x	10	
Python 3.8	3, 3.8	Amazon ECR DockerHub
Python 3.7	3.7	
Python 3.6	3.6	
Python 2.7	2, 2.7	

Runtime	Tags	Repository Links
.NET 5.0	5.0	Amazon ECR DockerHub
.NET Core 3.1	core3.1	
.NET Core 2.1	core2.1	
Java 11 (Corretto)	11	Amazon ECR DockerHub
Java 8 (Corretto)	8.al2	
Java 8 (OpenJDK)	8	
Go 1.x	1	Amazon ECR DockerHub
Ruby 2.7	2, 2.7	Amazon ECR DockerHub
Ruby 2.5	2.5	

If you want to create a custom image, it must confirm to either Docker image manifest V2, schema 2 (used with Docker v1.10 and newer), or Open Container Initiative Specifications (v1.0.0 and up). The image must also implement the Lambda Runtime API. AWS also provide base images for both the Amazon Linux and Amazon Linux 2 operating systems. You can use these to then add your preferred runtime, dependencies, and code.

Something to keep in mind when using container images is that they are immutable. This means that the runtime/language version won't get automatically updated. While this is good for predictability, it comes at the cost of needing to maintain the image with any patches or security updates.

What Is the Difference Between Lambda and Fargate?

While having container support is a huge deal for Lambda development, there is still a big difference in the types of applications that run on Lambda compared to AWS Fargate.

AWS Fargate is a serverless compute engine for containers, making it easy to build applications while removing the need to provision and manage servers. Fargate can automatically scale up and down as required, but it is typically used for long-running applications and services. As your application is always running, no warm-up time required, and as such, it's a good solution for stateful applications.

AWS Lambda is perfect for event-driven applications where some integration will kick off an event that causes the Lambda to be invoked. At the time of writing, there are over 140 different services that can be used as event sources, the most popular of which are API Gateway, S3, DynamoDB, and simple cron-based events. The maximum execution time for a Lambda is 15 minutes. Unlike Fargate, your Lambda functions remain idle and need to start up from these events.

In summary, these are two different AWS services that happen to allow you to package your application into containers, but each caters to specific use cases. AWS Lambda is suitable for unpredictable and inconsistent workflows, including use cases such as document conversion, automated backups, and real-time data processing. AWS Fargate is suited to long-running services that need to be available for long periods of time, or for applications that require more than 15 minutes of execution.

LEVELLING UP WITH STEP FUNCTIONS

Once you have created a Lambda and integrated with another service to trigger it, you'll start to see the potential of building up event-based serverless applications. One way that AWS have made the orchestration of such applications easier is by using AWS Step Functions.

Through a visual interface, you can create event-driven workflows, creating data flows that go to and from different AWS

services and into your AWS Lambda functions. The official Step Functions page has a number of different patterns for step functions from error handling, branching, and chaining.

By taking this approach, you can start to decompose monolithic applications into smaller services and bite-sized steps while still maintaining a clear overview of flows through the application.

Who Is the Target Audience?

For developers who are used to creating Docker files for building applications, the shift to AWS Lambda involved another learning curve, with the peculiarities of the zip file format, limitations on size, and the need to use layers to handle dependencies. In one swoop, AWS have now bridged the divide between these development approaches, bringing Lambda to a whole new set of developers, and making serverless, as a whole, much easier to adopt.

Developers who need to work with large dependencies, such as those building machine learning or data analytic services, now have an easier way of managing the size and range of dependencies.

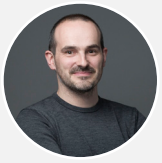
Additionally, those developers who want to use custom runtimes in their Lambdas now have the opportunity to create those by implementing the Lambda Runtime API.

Conclusion

The serverless developer in 2021 has more choice than ever before. Container support in AWS Lambda is something that can simplify dependency management and make it easier to switch between service types depending on your use case.

The following resources will help you to get started:

- AWS Tutorial on Creating Container Images for AWS Lambda
- Official AWS Lambda Documentation
- DZone Getting Started With Docker Refcard
- AWS Fargate Getting Started Guide
- AWS Step Functions Overview
- Introducing Container Image Support for AWS Lambda (re:Invent 2020 session)



James Sugrue, CTO at Over-C Technology Ltd

@jsugrue on DZone | @sugrue on Twitter | www.jamessugrue.ie

James is CTO at Over-C Technology, a provider of solutions for compliance and task management. He is an AWS Community Builder, with an expertise in serverless technologies and data analytics. James has worked in the software industry for 20 years, and builds products using Java, JavaScript, Swift, and Python. He is also founder of donatecode.com, which helps match developers with charitable causes in need of free technical expertise.

ADDITIONAL RESOURCES

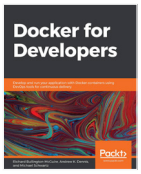
Diving Deeper Into Containers

BOOKS

Container Security

By Liz Rice

Security is an integral part of using containers to run applications in cloud-native environments. This book examines the common building blocks of underlying technologies to help developers and other software professionals better assess potential security risks and identify proper mitigation strategies.



Docker for Developers

By Richard Bullington-McGuire, Andrew K. Dennis & Michael Schwartz

As a growing number of teams migrate their projects to containers (using the de-facto standard for app containerization), it's all the more important to understand the fundamentals of building, deploying, and securing Docker environments effectively. This book explores best practices, use cases, and step-by-step guides for all things Docker.

REFCARDS

Getting Started With OpenShift

Open-source container orchestration platform OpenShift includes components of the Kubernetes container management project with added productivity and security features that are important to large-scale companies. This Refcard walks you through OpenShift setup — you'll get hands on by installing a local development cluster on your own machine.

Getting Started With Kubernetes

Containers weighing you down? Kubernetes can scale them. Organization is critical for running and maintaining successful containerized applications. This Refcard contains everything you need to know about Kubernetes, including key concepts, code examples and commands, how to successfully build your first container, and more!

TREND REPORTS

Kubernetes and the Enterprise

Want to know how the average Kubernetes user thinks? Wondering how modern infrastructure and app architectures interact? This report explores key advances in technical areas related to related to the pervasive container management platform. Included are contributor insights into scaling microservice architectures, cluster management, and deployment strategies.

Cloud Native

When every aspect of your application infrastructure has been adopted and implemented with the cloud in mind, you can expect results — faster, more efficient ways to run, develop, and deploy applications. This report covers findings from our research, an approach to adopting cloud-native app development, and the intersection of microservices and cloud native.

PODCASTS

Software Defined Talk

Keep up with cloud computing news and events (supplemented with some entertaining banter) in this go-to weekly podcast on Kubernetes, serverless, security, and other key areas. You'll also get the latest updates on all major cloud providers as well as the CNCF.

DevOps and Docker Talk

Host Bret Fisher offers a catch-all podcast that covers Q&As from live shows, guest interviews, and chats with industry friends — all centered around cloud-native and DevOps topics like container tools, cloud management, and sysadmin. Discover even more during his live YouTube shows.

The Cloudcast

Independent cloud computing podcast Cloudcast, co-hostsed by Aaron Delp and Brian Gracely, features interviews with technology and business leaders who are shaping the computing industry's future. Topics include serverless, DevOps, Kubernetes, and many more!