DZone

TREND REPORT

Kubernetes and the Enterprise

BROUGHT TO YOU IN PARTNERSHIP WITH

REPLICATED

Welcome Letter

By Jesse Davis, Chief Technologist at Devada

It's crazy to think that 20 years ago, we hired Build Engineers whose whole job was to take our code, pull it from source control, and compile it. Then they would copy those onto physical distribution media (CDs or Floppy Disks) and start actually "shipping" software (in packages). We have come such a long way!

Software systems have matured to the point where we don't even have physical media anymore. Our application bits live in the cloud and are deployed automatically by autonomous, carefully orchestrated robots. Armed with their knowledge of the intricacies of our code, they proceed to compile and assemble it, place it into an appropriate container and deploy it — all without needing us pesky humans involved.

As I sit here writing this letter for our new Kubernetes Trend Report, I can't help but look back on how far we've come... and why it's so important that we have tools like K8s in our toolbox. That Build Engineer from the days of yore would quickly become overwhelmed with the complexity of today's environments.

Simple client-server applications have given way to cloud native applications that run across multiple operating environments — including dedicated on-prem servers, virtualized private clouds, and public clouds such as AWS and Azure. That's a lot of platform-specific expertise for just one person, but tools like Kubernetes free us from worrying about the underlying infrastructure so we can concentrate on delivering great applications.

Without K8s, we would have to find new ways to deal with application balancing and distribution, zero-downtime container updates, and a whole host of other on-the-fly tasks that humans can't (and don't) do well. Applications running critical workloads 24/7 require that there's no lunch break, and that everything is humming along and secure with millisecond response times when something goes wrong.

We've asked some of the amazing K8s experts to share their knowledge with us for this report. In these pages, you'll get help choosing the right flavor of K8s for your project, learn the latest open-source tools, get a security lesson with best practices from NSA and CISA, and understand the current state of the Kubernetes world with insights from our own research team.

Welcome to our 2021 Kubernetes and the Enterprise Trend Report — we hope you enjoy reading it as much as we enjoyed preparing it for you!

Cheers,

[illegible]

Jesse Davis

Jesse Davis, Chief Technologist at Devada

As Chief Technologist @ Devada, Jesse helps customers build the world's largest and most engaging developer communities. Jesse has been building enterprise software product and engineering teams for over 20 years, and he is a respected executive, author, speaker, and coach. Jesse helped develop the first data access for Java in the 90s and served as an expert and innovator on industry data standards including JDBC, ODBC, and ANSI SQL. When not doing computer-y things, you can find him crafting in his woodworking shop with his wife and kids.

ORIGINAL RESEARCH

Key Research Findings

An Analysis of Results from DZone's 2021 Kubernetes Survey

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

In September 2021, DZone surveyed software developers, architects, and other IT professionals in order to understand how sub-VM clusters are designed and orchestrated, with special (but not exclusive) focus on Kubernetes. Our survey represents a continuation and expansion of our 2020 research on the same topic.

Major research targets:

1. The state of resource isolation, container orchestration, and infrastructure abstraction
2. Kubernetes promises vs. real-world performance
3. Classification of applications that run on Kubernetes clusters

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 September 8th and closed on September 22nd. The survey recorded 499 responses*.

**Note: This excludes 331 responses we threw out as spam based on our usual data cleansing criteria (e.g., response time averaging 1s per question, all multiple-choice questions responded to with the first answer choice or in excessively simple response patterns, bogus email addresses) — a much higher percentage (nearly 40%!) than we normally discard using the same criteria (usually ~7%). We're not sure what to make of this. Our guess is that Kubernetes and related technologies are "hot" enough that an unusually large number of bots are trying to manipulate research on Kubernetes-related topics.*

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: The State of Resource Isolation and Container Orchestration

Motivations (remain unchanged from our 2020 report):

1. Software development and runtime ecosystems are now complex and tangled enough that OS-level resource management is often insufficient for avoiding conflicts in build and runtime environments.
2. As more applications run on the web, where state management is not built into the application protocol, application state management becomes increasingly difficult to manage through explicit application-level code but is easier to automate at a lower level.
3. As software architectures increasingly take advantage of the "metal-indifference" of cloud computing while depending on multi-platform runtimes (JVM, Node.js, high-performance interpreted languages like Python) and complex graphs of dependencies — often with their own rapid development cycles — a dimension for horizontal scaling that allows more granular control over runtime environment (rather than the application source) than VM-level (needed if OS-level WORA runtimes weren't used) becomes increasingly attractive.
4. Further, as Agile development methodologies encourage radically service-oriented designs that comprise independently built and managed systems with well-defined system boundaries ("microservices") and strongly opaque internals, maintenance of a single OS-level environment that serves many services' heterogeneous needs becomes increasingly difficult — sometimes impossible.
5. Finally, as container use increases, the need for high-level container orchestration also increases.

HOW RESOURCES ARE ISOLATED IN LINUX SYSTEMS

Modern container management developed organically from manually configurable resource isolation capabilities offered by multi-user operating systems. We wanted to know how users were isolating resources on Linux systems without necessarily using higher-level abstractions like Docker or rkt. We hypothesized that fewer respondents would report using granular, manual resource isolation techniques this year versus last year, based on the following assumptions:

1. Higher-level containerization/orchestration tooling continues to mature.
2. Distributed architectures in which individual runtimes perform narrowly defined tasks require more automated, source-controlled resource management.
3. DevOps professionals/SREs (who are more likely to use or develop higher-level resource management tools) are absorbing more work that was previously handled by pure Ops professionals (who are more likely to use OS-level constructs).

So we asked:

What methods do you use to isolate resources in Linux systems?

Results across all responses, comparing 2020 and 2021 data, are shown in Table 1.

Observations:

1. Usage of each lower-level resource isolation method decreased year over year (YOY), verifying our hypothesis.

   Drops in LXD and LXCFS usage were insignificant, but drops in LXC (38.1% to 32.8%) and chroot (31.2% to 23.1%) were significant. The fact that the most sysadmin-like of these methods experienced the largest drop is also consistent with our assumption that pure ops work is increasingly being done by IT professionals using higher-level approaches.

2. Relative ranking of each resource isolation method did not change YOY. The gap between the most popular (LXC) and the second most popular (chroot) increased — again consistent with our hypothesis.

Table 1

LINUX RESOURCE ISOLATION METHODS

	% of total respondents using	
Method used	**2020**	**2021**
LXC	38.1%	32.8%
chroot "jail"	31.2%	23.1%
LXD	22.4%	21.6%
LXCFS	18.8%	17.5%

Next, we segmented responses by development vs. production environments, hypothesizing that we would see a bigger YOY drop in lower-level resource isolation method use in development environments. We reasoned development environments are more likely to be set up using less complex but less scalable techniques because they are less likely to receive as much upfront investment in set-up time, and hence, are more likely to benefit from increasing ease of use of higher-level tools.

Results, comparing 2020 and 2021 data:

Table 2

LINUX RESOURCE ISOLATION METHODS BY ENVIRONMENT TYPE

	Development environments		Production environments	
Method used	**2020**	**2021**	**2020**	**2021**
chroot “jail”	57.6%	61.6%	42.4%	38.4%
LXC	53.3%	56.3%	46.7%	43.7%
LXD	54.9%	62.4%	45.1%	37.6%
LXCFS	51.8%	55.3%	48.2%	44.7%

Observation:
Our hypothesis was incorrect across all resource isolation methods surveyed. In all cases, usage of lower-level resource isolation techniques *increased* in *development* environments YOY while *decreasing* in *production* environments. Perhaps this indicates that, while higher-level tooling further obviates the need for more sysadmin-like work in production, developers are growing more uncomfortable with increasingly high levels of infrastructure abstraction — and are reverting to more granular control over development runtimes. We base this conjecture also on YOY changes in respondents' subjective evaluation of the state of infrastructure abstraction (discussed below, but spoiler: Sanguinity is lightly flagging).

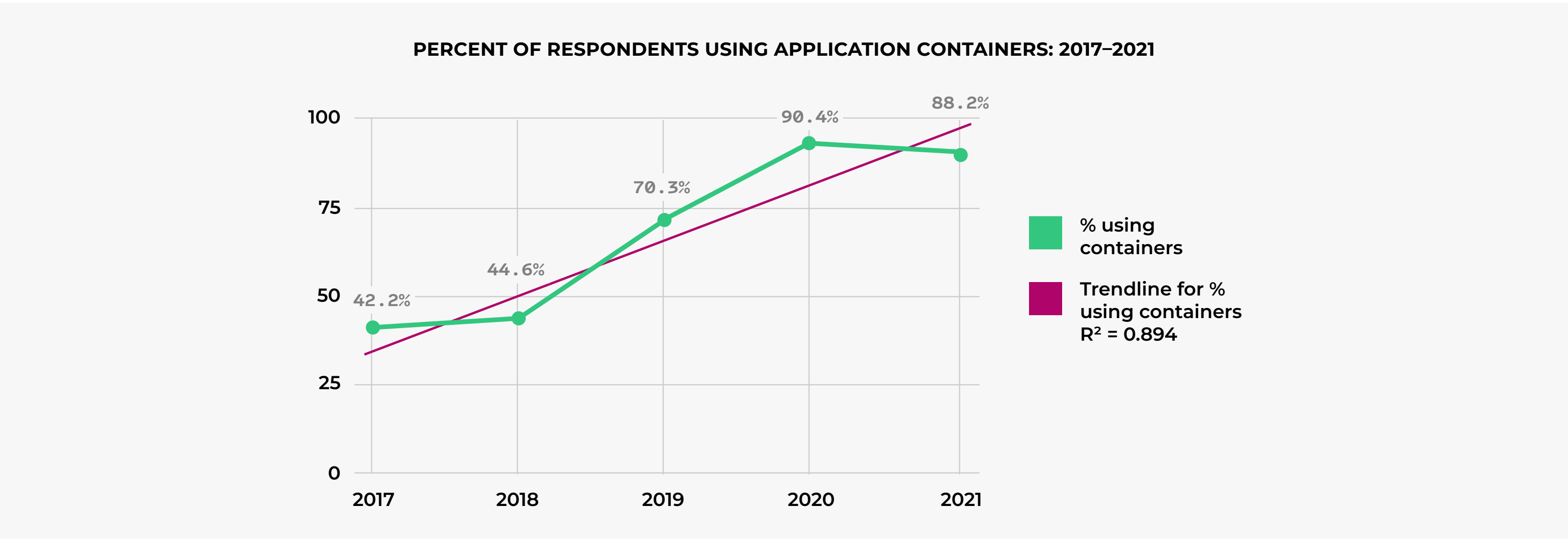
HOW OFTEN APPLICATION CONTAINERS ARE USED

Given the YOY decrease observed in lower-level resource isolation methods, we expected that application container usage would have increased proportionally since 2020. To answer this question, we asked:

Do you use application containers in either development or production environments?

Results since 2017:

Figure 1



Observations:

1. Contrary to our expectations, application container usage did not increase year over year.

   In 2021, 88.2% of respondents reported using application containers vs. 90.4% in 2020 — not a significant difference but also not an increase that would account for the observed decrease in lower-level resource isolation techniques.

2. This result may indicate that containerization has reached a saturation point. However, note that 3.3% of 2021 respondents reported that they don't know whether they use application containers vs. 1.7% in 2020.

   This further shrinks the (already negligible) decrease observed — though it does not vitiate the difference between expected increase and observed stagnation — and perhaps is a whiff of a suggestion that leaks in the abstraction are being plugged. Presumably, ideal infrastructure would allow users not to care whether bare metal, VM, or containers, with or without additional orchestration layers, are running their code.

3. Another possible clue: A slightly smaller percentage of respondents who use application containers reported that their organization uses microservices in 2021 (91.9%) vs. in 2020 (95.3%).

   We conjecture that this drop-off may indicate increasing usage of "serverless" abstractions and the resulting performance optimizations that are needed — "micro-VM" serverless-supporting technologies save spin-up overhead by allocating VM resources using kernel-level approaches vs. creating new containers, which, while lighter than whole VM instances, are heavier than kernel-level virtualization technologies like Firecracker and QEMU. In future surveys, we intend to investigate serverless approaches from both client and server (*har*) perspectives.

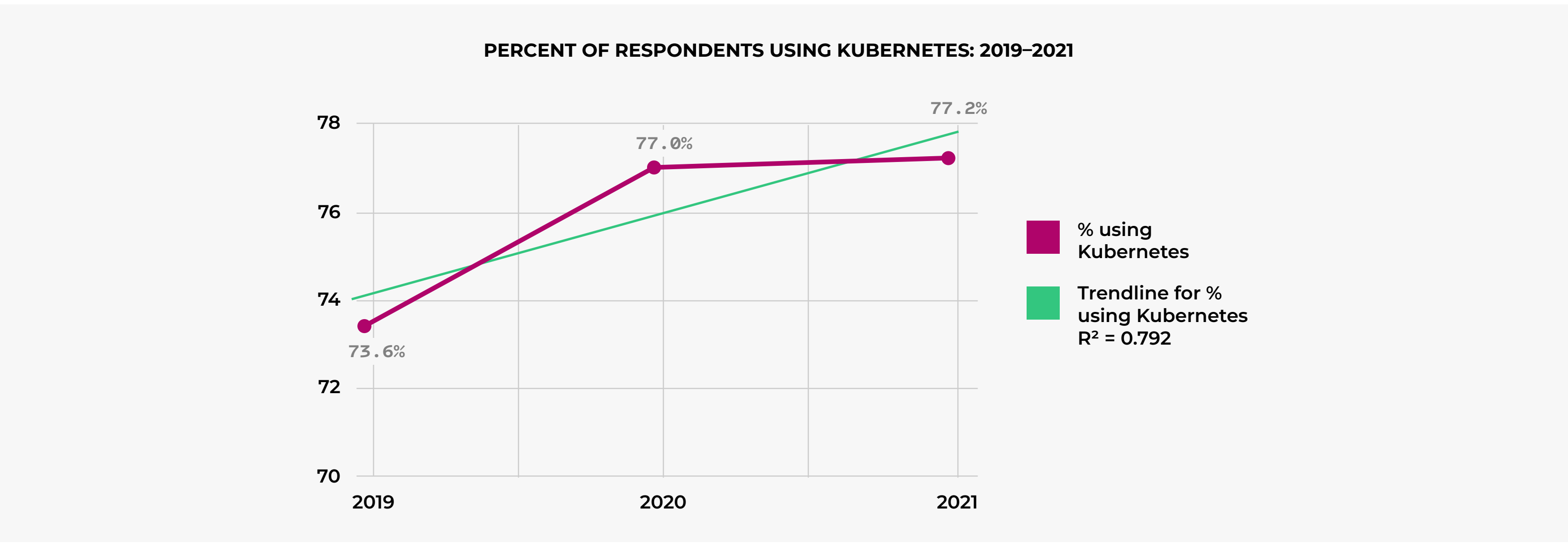
4. Complicating the matter further, the 2020-2021 coronavirus pandemic makes any observations about trends in human activity relatively and rationally easy to ignore.

KUBERNETES USAGE

Before observing year-over-year (YOY) stagnation in application container adoption, we supposed that Kubernetes usage had also increased since 2020. After noting the container flatline, we altered our Kubernetes adoption hypothesis: We expect that Kubernetes usage would not vary significantly versus last year.

Results since 2019:

Figure 2



Observations:

1. Our altered (not original) hypothesis was verified: Along with container usage, Kubernetes usage basically flatlined between 2020 and 2021 (neglecting the 0.2% increase).

   However, because Kubernetes is a much more complex technology than an application container, stagnation in absolute adoption number poorly reflects effort and ingenuity changes poured into Kubernetes over time. We discuss YOY changes in Kubernetes usage (within that unchanging 77% respondent percentage) below.

2. As with containers, the percent of respondents who reported that they don't know whether their organization is using Kubernetes increased in 2021 (6.4%) vs. 2020 (4.5%). We tentatively interpret this along the same lines as we interpreted a similar increase in container-usage ignorance: Perhaps the orchestration abstraction is growing more palpably solid.

3. The YOY change in relation of Kubernetes usage to microservice usage is more interesting than the YOY change in relation of container usage to microservice usage.

   While only 43.4% of 2020 respondents whose organizations did not use microservices did run Kubernetes clusters, 51% of 2021 respondents whose organizations do not use microservices do run Kubernetes clusters. That is, Kubernetes adoption does not appear to be tightly coupled with microservice adoption. Since we have only two years of segmentable data, we do not speculate on causes of what may not actually be a trend; however, this result is a little surprising, and we will inquire further in future surveys.

ATTITUDES TOWARD THE STATE OF INFRASTRUCTURE ABSTRACTION

We take the significant increase in usage of OS-level resource isolation methods (discussed above) to suggest an increasing suspicion of modern forest-with-invisible-trees infrastructure abstraction. In fact, we do have two years of data on precisely this question, having asked in 2020 and 2021:

(See question on next page)

Please select the option that best describes your attitude toward infrastructure abstraction in 2021. {Infrastructure abstraction in 2021 is excessive and getting out of hand | We're finally getting close to pure, non-leaky infrastructure abstraction | The cost in complexity of modern infrastructure abstraction is worth the benefits of infinite scaling and continuous delivery | No opinion}

Results, year over year:

Table 3

ATTITUDES TOWARD THE CURRENT STATE OF INFRASTRUCTURE ABSTRACTION

	2020	2021
Infrastructure abstraction in 2021 is excessive and getting out of hand	16.0%	18.4%
We're finally getting close to pure, non-leaky infrastructure abstraction	21.3%	18.8%
The cost in complexity of modern infrastructure abstraction is worth the benefits of infinite scaling and continuous delivery	47.9%	46.2%
No opinion	13.7%	15.5%

Observations:

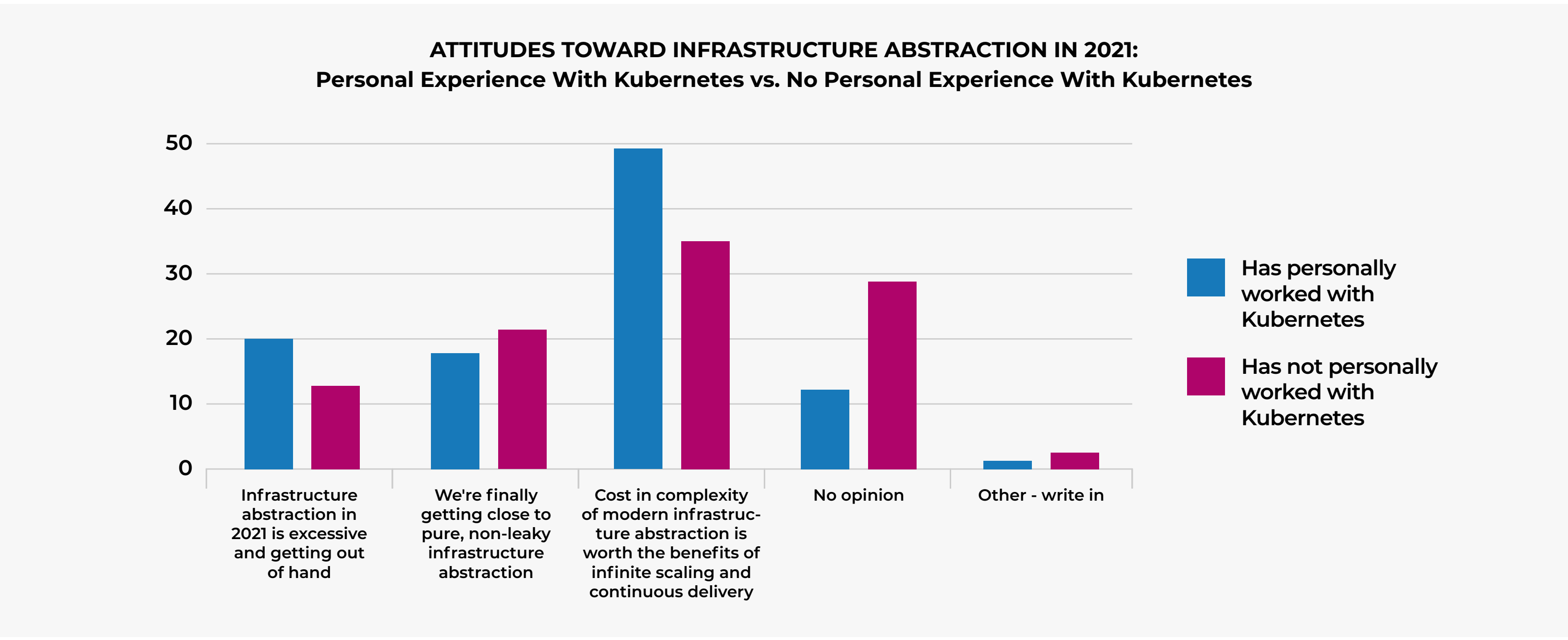
1. Year-over-year changes in attitudes toward modern infrastructure abstraction look weakly sad. A small increase in judgment that infrastructure abstractions are excessive (16% in 2020 vs. 18.4% in 2021) is matched by a decrease in trust in the airtightness of infrastructure abstractions (21.3% in 2020 vs. 18.8% in 2021).

2. Agnosticism toward modern infrastructure abstraction also increased from 13.7% in 2020 to 15.5% in 2021.

   This is consistent with the slightly growing ignorance of sub-application constructs discussed above (at virtualization and orchestration levels), but it also may suggest the same changes in infrastructure abstraction as more negative judgments, viewed through less critical eyes: A structure with a constant unintelligibility level may be rejected by an IT professional of one sort and shrugged at by an IT professional of another sort. But both answers can be accounted for by increasing infrastructure opacity and cannot reasonably be accounted for by decreasing infrastructure opacity — which we also mean to entail non-leaky abstraction.

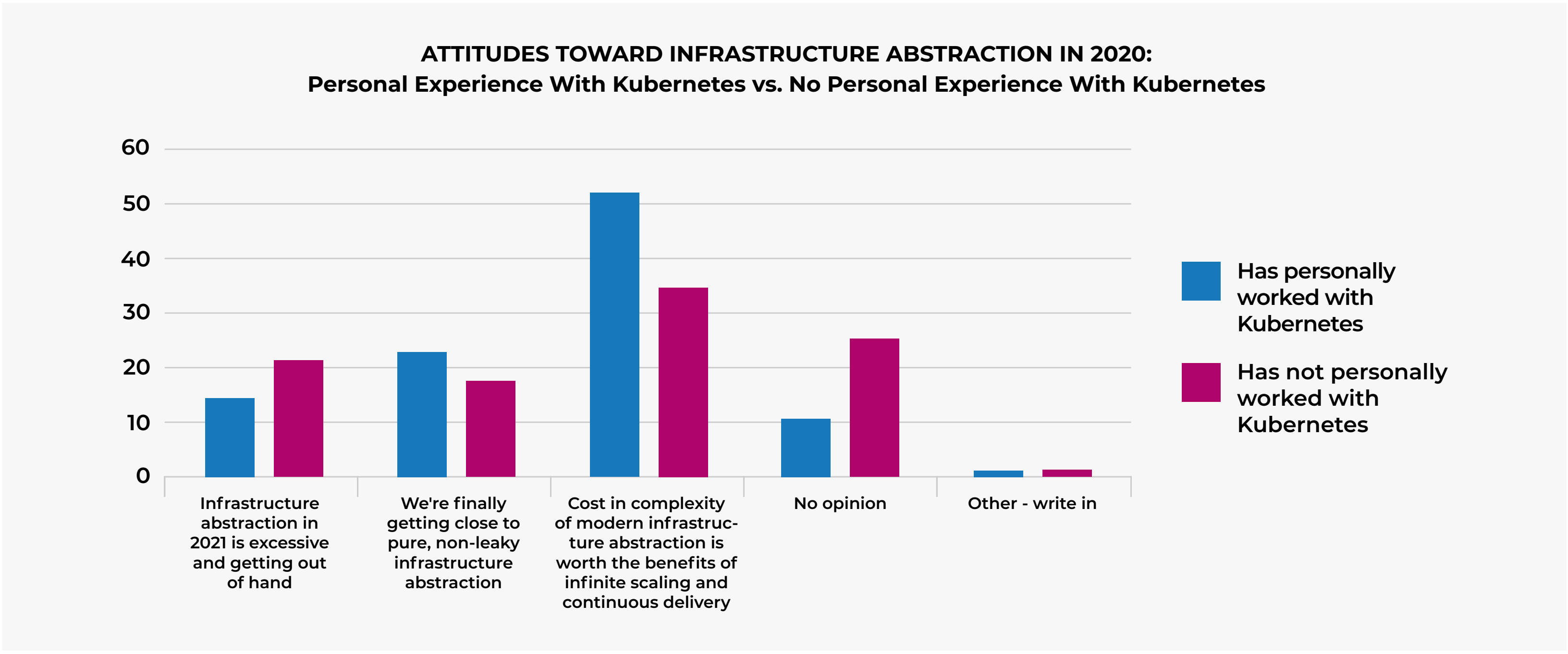
3. Personal experience with Kubernetes affects overall judgment of the state of infrastructure abstraction (from 2021 data):

Figure 3



It appears that working with Kubernetes correlates with both critical judgment of the state of infrastructure abstraction in itself and acceptance of this complexity as worth the cost. Changes in these correlations vs. in 2020, however, also paint a somewhat pessimistic picture of the relation between Kubernetes usage and attitudes toward modern infrastructure (from 2020 data):

Figure 4



A year ago, Kubernetes users were less likely to judge modern infrastructure as excessive and more likely to admire it for airtightness, while in 2021 the inverse is true on both counts.

Research Target Two: Kubernetes Promises vs. Real-World Performance

Motivations:

1. New approaches to old problems are likely to reinvent some wheels better, and some worse, than antecedent approaches. Since resource isolation, virtualization, cluster management, and other distributed computing problems are as old as time-sharing, we wanted to know how Kubernetes-using state compares with pre-Kubernetes state.

2. FAANG invention and usage *ipso facto* tips the tool evaluation scale toward adoption. But most organizations operate at way-sub-FAANG scale. We wanted to evaluate how users feel about using Kubernetes post-adoption when the Google/ Borg sheen has worn off.

3. Perhaps the greatest bugaboo of distributed systems — and a major place where distributed object systems like CORBA and DCOM have faltered — is application (as opposed to cluster) state. We wanted to see how well Kubernetes performs the magical infra- and workload-state matching task promised by a mature cluster management tool.

WHAT KUBERNETES IMPROVES

A well-designed tool is amusing but worthless unless its usage makes something better. We wanted to know what, in IT professionals' judgment, Kubernetes actually improves. Further, as users grow more familiar with a tool, we expect the improvement brought about by that tool to increase. We wanted to learn several related outcomes of using Kubernetes: what specific areas of software development it improves, and which ones it succeeds at, or falls short of, improving over time.

We asked:

What has Kubernetes improved at your organization?

Results from 2020 and 2021 are shown in Table 4:

Observations:

1. We see a mild un-honeymooning of Kubernetes here as well — though with considerable variance across improvement areas. All but one area of improvement decreased between 2020 and 2021.

2. Security, however, increased significantly: 31.3% of 2021 respondents reported that Kubernetes has improved security vs. only 24.9% of 2020 respondents.

   We do not know what in particular might account for this significant development. Anecdotally, we have heard increasing buzz about Kubernetes security over the past year, but of course buzz often indicates problems that need to be addressed as much as it indicates solutions. We welcome any readers' explanations at publications@dzone.com.

3. Reported design-level improvements (building microservices, application modularity, overall system design) decreased negligibly year over year. We take this to mean that Kubernetes is fundamentally well suited to modern application architectures.

Table 4

IMPROVEMENTS FROM USING KUBERNETES

	2020	2021
Deployment in general	66.0%	64.3%
Autoscaling	65.0%	62.8%
CI/CD	63.7%	61.1%
Building microservices	53.6%	52.3%
Reliability	46.0%	39.6%
Application modularity	44.3%	43.3%
Architectural refactoring	36.1%	32.2%
Overall system design	33.5%	32.2%
Cost	28.9%	28.2%
Security	24.9%	31.3%
Other	3.2%	2.6%

4. Some reported SDLC-linked improvements decreased by a more significant amount, notably architectural refactoring (36.1% in 2020 vs. 32.2% in 2021). Decreases in reported CI/CD improvement and deployment in general were negligible.

5. The biggest drop-off in Kubernetes-mediated improvement comes at runtime.

   The decrease in reported reliability was the largest by far (39.6% in 2021 vs. 46% in 2020), with the decrease in autoscaling improvement relatively distant but still a notable nonzero second (62.8% in 2021 vs. 65% in 2020). We might speculate what sorts of changes might have caused this decrease in reliability — changes in areas such as workload, Kubernetes cluster design, personnel training, managed services, or denial-of-service attack patterns — but we decline for now in anticipation of future research.

   Shifting correlations with workload and access pattern types might tempt explanations for reliability decreases, but although we have these data from the present survey (discussed below), we do not have any such data from previous years.

USAGE OF KUBERNETES FOR STATEFUL WORKLOADS

Highfalutinly: The need to speculate about the nature of before-and-after in Leslie Lamport's fundamental paper suggests the profoundness of the problem with distributed application state. More colloquially: given the option, we would probably all choose to leave state maintenance to a single data store (memory address, hard disk), or some approximation (e.g., an eventually consistent NoSQL DBMS), even before the 12-factor paradigm explicitly promoted stateful backing services.

In the Kubernetes world: Kubernetes' StatefulSet abstraction is designed precisely to simplify distributed state maintenance by imposing a managed before-and-after on top of a physically distributed cluster. This is no mean task, so we wanted to know how often Kubernetes is being used for stateful workloads and whether it's handling these workloads well.

So we asked:

Do any of your organization's Kubernetes clusters run any stateful workloads where state is maintained within the cluster?

We also asked this question in 2020, although we did not distribute the results publicly, but we can now compare results from 2020 and 2021:

Table 5

RUNNING STATEFUL WORKLOADS WHERE STATE IS MAINTAINED WITHIN A KUBERNETES CLUSTER

	2020	2021
Yes	45.7% (n=29)	49.6% (n=235)
No	40.1% (n=221)	33.8% (n=160)
I don't know	14.2% (n=71)	16.7% (n=79)

Observations:

1. Approximately half of respondents reported running stateful workloads where state is maintained in a Kubernetes cluster. To us — who have not used Kubernetes for stateful workloads but have often struggled with problems of distributed state managed more manually — this number is incredibly impressive.
2. The significant YOY increase (almost 4%) in stateful workloads on Kubernetes clusters is consistent with our general impression that Kubernetes is reviving the competition between older and (perhaps) more ambitious concepts of distributed applications and the cloud-native, http-first, radically stateless immutability facilitated by pre-Kubernetes container adoption.

To answer the why-not-just-use-Postgres-for-everything-stateful skeptic (constantly screaming from inside our brains), we also asked:

Are you glad that your Kubernetes cluster(s) maintain(s) state within the cluster(s)?

Results, for 2020 and 2021:

Table 6

APPROVAL OF KUBERNETES' STATEFUL WORKLOAD MANAGEMENT

	2020	2021
Yes, Kubernetes is doing a good job of maintaining state	73.8% (n=166)	77.0% (n=174)
No, I wish we were maintaining state outside the Kubernetes cluster	17.3% (n=39)	19.5% (n=44)
I have no opinion	8.9% (n=20)	3.5% (n=8)

Observation:
Not only are more applications letting Kubernetes maintain state in 2021 vs. 2020, but also, of those Kubernetes-stateful applications, more are doing a good job maintaining state in 2021 (77%) than in 2020 (73.8%).

Research Target Three: Classification of Applications Running on Kubernetes Clusters

Motivations:

1. Sometimes distributedness is imposed by choice of variable infrastructure constraints (e.g., low cost of commodity hardware); sometimes the nature of the workload exerts strong pressure toward running in a cluster (e.g., highly parallelizable algorithms). We wanted to tease out the relationship between the nature of the work performed and the distributed infrastructure on which the work is done.
2. Cluster management enjoys a long history in High-Performance Computing (HPC), but knowledge gained from HPC domains — often scientific, disproportionately emphasizing partial differential equations — does not always map neatly

onto the web-scale applications that Kubernetes was originally built for because of basic differences in workload type. We wanted to see what sorts of workloads are run, not on clusters in general, but on Kubernetes clusters in particular.

3. A closed distributed system is more predictable; an open distributed system (like the web) is less predictable. Broadly speaking, tuning involves trading performance against flexibility. These three observations together generate the kind of tension between "runs fast" and "runs on the web" that an autoscaling system like Kubernetes — adaptable to different traffic patterns — tries to resolve.

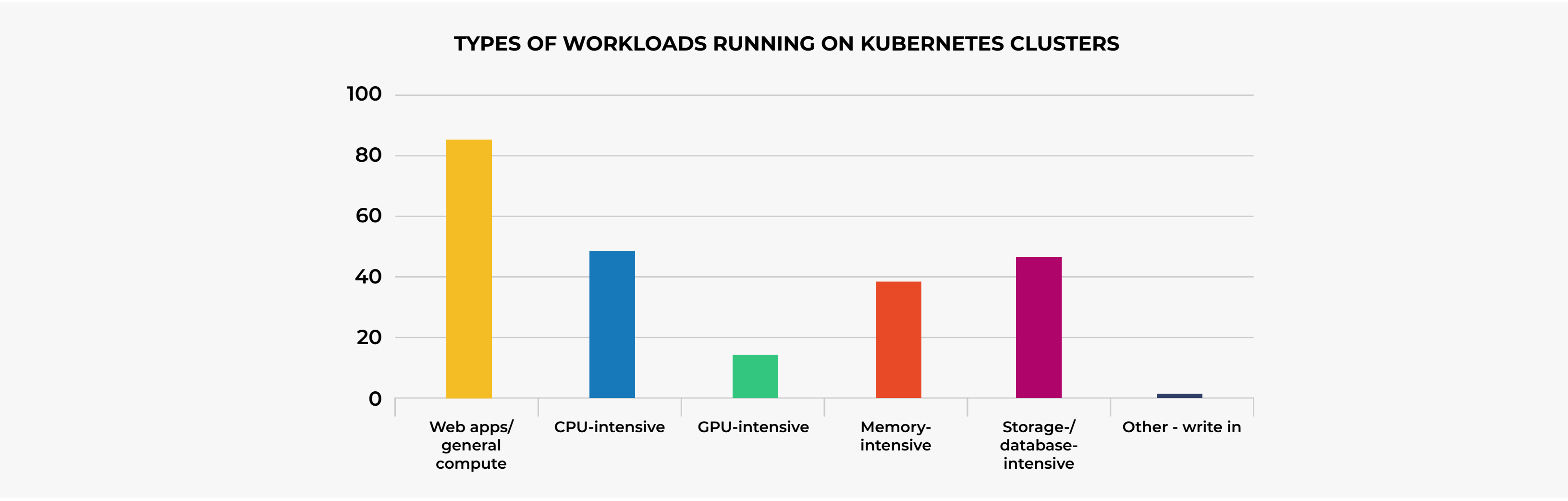
TYPES OF WORKLOADS THAT RUN ON KUBERNETES CLUSTERS

Dynamically managed compute clusters may be needed for many kinds of workloads, some of which Kubernetes might handle better than others. We wanted to know what kinds of workloads people are running on Kubernetes clusters and whether these workloads correlated with overall satisfaction with Kubernetes. So we asked:

What types of workloads does your organization run on Kubernetes clusters? {Web applications/general compute | CPU-intensive | GPU-intensive | Memory-intensive | Storage/database-intensive}

Results:

Figure 5



Observations:

1. 84.6% of respondents run web/general applications on Kubernetes cluster, by far the most common use case.

   This number tracks type of software currently being developed by survey respondents overall (84.6% web) and whether respondents run Kubernetes; hence, it does not indicate especially heavy use of Kubernetes for web applications. Our reason for lumping "web" and "general" answer choices together was to consider how Kubernetes handles comparatively ephemeral connections with strong connection-level transaction semantics. In future research, we intend to make this overlap explicit.

2. GPU-intensive workloads were least likely to run on Kubernetes clusters (15.4% of respondents), but while Kubernetes has supported multi-node jobs on Nvidia and AMD GPUs since 2017, Kubernetes documentation still lists this feature as experimental.

   In this light, 15% seems fairly substantial for an experimental feature. Our guess is that most of this GPU-intensive work on Kubernetes comprises (heavily parallelizable) deep learning jobs, but we will inquire further in future surveys.

3. Respondents whose organizations run GPU-intensive jobs on Kubernetes clusters were the most likely to judge that modern infrastructure abstraction is getting out of hand. And respondents whose organizations run storage/database jobs on Kubernetes clusters were least likely to judge that modern infrastructure abstraction is getting out of hand.

   This may reflect some impedance mismatch between GPU-intensive workloads and Kubernetes' design, which is itself somewhat suggested by the slow pace of development of GPU support on Kubernetes.

We imagine two reasons for this:

- Mature, robust compute parallelization models (CUDA, OpenCL) that handle matrix transformations (heavily used by deep learning) have existed for many years.
- Deep learning jobs are less likely to require the dynamic autoscaling capabilities that Kubernetes offers and are especially useful for web applications.

Perhaps for allied reasons, respondents whose organizations run web/general applications on Kubernetes clusters were most likely to judge that the cost in complexity of modern infrastructure abstraction is worth the benefits of infinite scaling and continuous delivery.

4. Respondents whose organizations run storage/database-intensive applications on Kubernetes clusters were somewhat more likely (84%) to have personally written code that runs in a Kubernetes cluster vs. respondents whose organizations run web/general compute applications in Kubernetes clusters (82.6%), CPU-intensive (80.7%), memory-intensive (81.6%), or GPU-intensive (80%) applications.

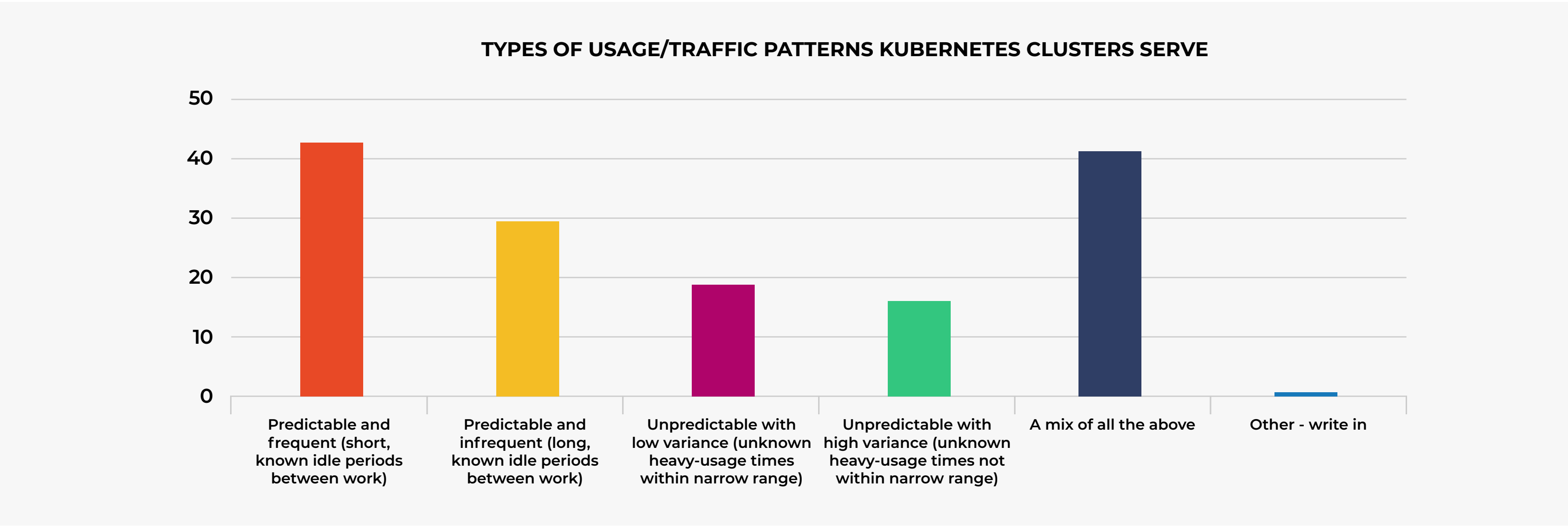
KINDS OF USAGE/TRAFFIC PATTERNS HANDLED BY KUBERNETES CLUSTERS

The distribution of compute work across the various parts of the von Neumann architecture does not capture one of Kubernetes' major value-adds (inherited from its Borg origins): the ability to handle high-variance usage/traffic patterns without wastefully overcommitting physical resources or underperforming under certain traffic conditions. We wanted to see what usage/traffic Kubernetes clusters are actually being applied to. We also wanted to probe our conjecture that sub-containerized jobs are picking up some work from containerized applications. So we asked:

What kinds of usage/traffic patterns do your organization's Kubernetes clusters serve? {Predictable and frequent (short, known idle periods between work) | Predictable and infrequent (long, known idle periods between work) | Unpredictable with low variance (unknown heavy-usage times within narrow range) | Unpredictable with high variance (unknown heavy-usage times not within narrow range) | A mix of all the above}

Results:

Figure 6



Observations:

1. Predictable and frequent (short, known idle periods between work) were the most common (42.3%) traffic patterns run on Kubernetes workloads, followed closely by a mix of all the above (40.9%). Short idle periods suggest that lighter-weight services are especially desirable — an observation consistent with our conjectures about possible reasons for apparently maturing use of containers and Kubernetes in practice.

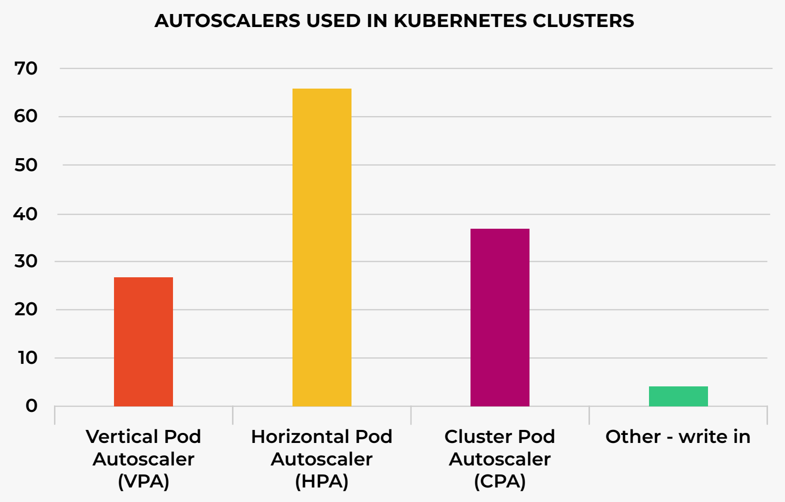
2. The high level of "a mix of all the above" responses (40.9%) suggests that Kubernetes' utility as a general-purpose distributed workload orchestrator is significantly exercised in practice.

3. Some relationship may exist between usage/traffic pattern of Kubernetes clusters and area of improvement seen from Kubernetes adoption.

For example, "predictable and frequent" traffic respondents were most likely to see improvements from Kubernetes in reliability, CI/CD, and autoscaling, while "a mix of all the above" respondents were most likely to see improvements in overall system design and cost.

AUTOSCALERS USED IN KUBERNETES CLUSTERS

Figure 7

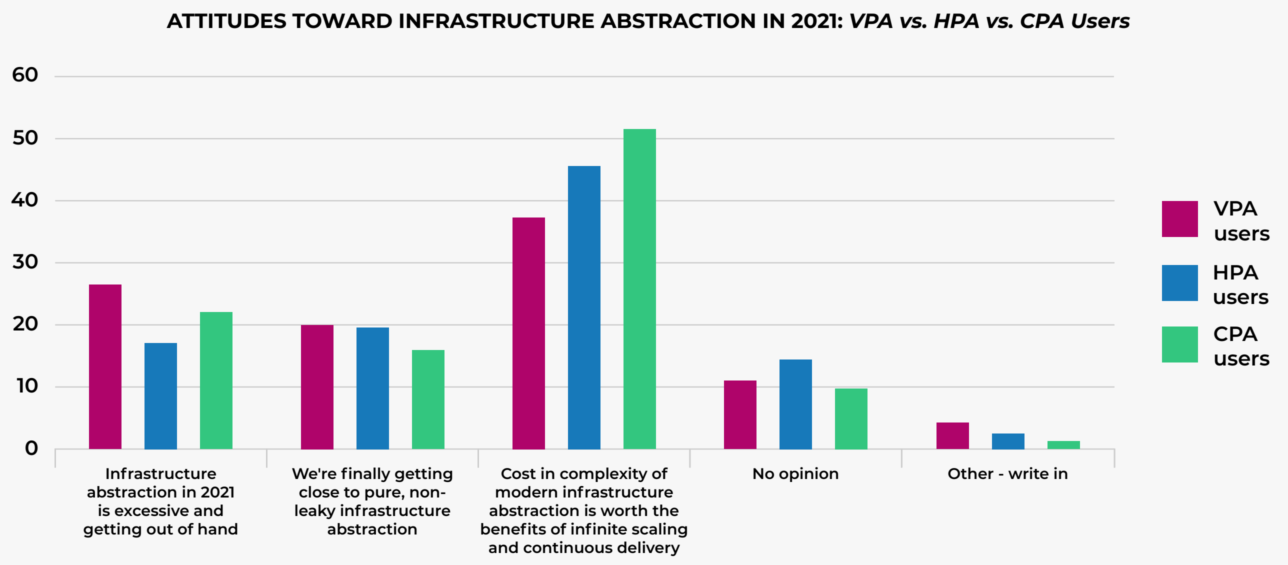


Observations:

1. The most common autoscaling strategy by far is horizontal. This is also perhaps the most opinionated, least requiring of accurate guessing during cluster configuration, which may suggest that Kubernetes is being used to pluck relatively low-hanging cluster management fruit.

2. Correlations between autoscaler type usage and attitudes toward infrastructure abstraction are interesting:

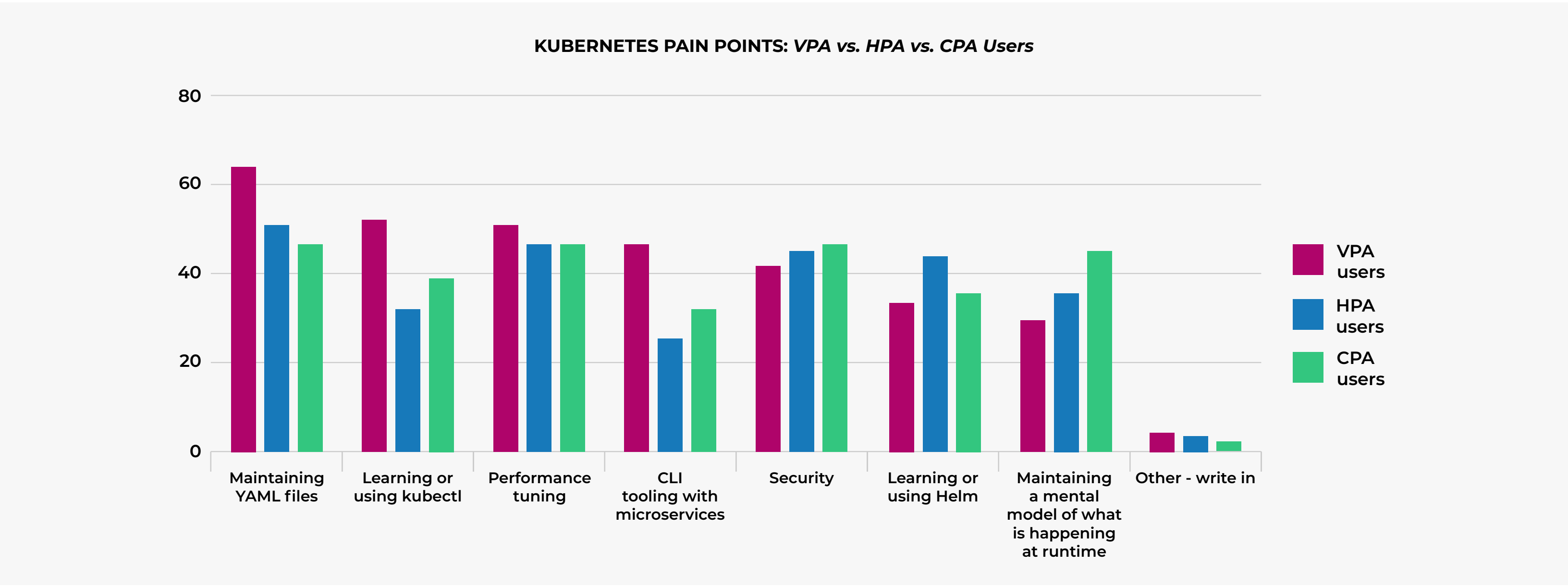
Figure 8



Respondents who use the type of autoscaler that most resembles simply adding memory or cores (vertical) are more likely than those who use other autoscaler types to judge modern infrastructure abstraction excessive and out of hand. And they are least likely to consider the benefits of modern infrastructure abstraction worth the cost — perhaps reflecting a link between vertical autoscaling and sysadmin-like or hardware-minded thinking.

3. Correlation between autoscaler type use and pain points encountered while working with Kubernetes seems intelligible:

Figure 9



Respondents who use horizontal autoscalers are by far least likely to report CLI tooling with microservices as a pain point — perhaps reflecting the maturity of using containers for microservices. Respondents who use cluster pod autoscalers are most likely to have trouble maintaining a mental mode of what is actually happening — perhaps reflecting that the Kubernetes (more nearly physical) *node* abstraction is less tightly linked with application demand than the (artificially atomic) pod abstraction.

Future Research

We will continue to ask our core questions around container and Kubernetes adoption in order to track year-over-year usage trends. Because the effects of the coronavirus pandemic are surely vast but hard to understand at a granular level, we are not confident that the ceased linear growth of container and Kubernetes usage observed between 2020 and 2021 is not at least partially an artifact of non-technical pressures. Data from next year may indicate trends more reliably.

Although this is our first Kubernetes survey to ask about workload types, traffic/usage patterns, and autoscaler selection, we have asked similar questions in other surveys — especially regarding application performance and web development. And in future surveys, we will coordinate these inquiries more closely on the assumption that the share of web and high-performance compute work performed by managed clusters of containers and serverless functions will continue to increase. We also intend to watch Kubernetes support for GPU compute closely, as the growth of real-time machine learning requires combining calculational advantages of parallel computing models (e.g., CUDA, OpenCL) built for GPU-based pipelines with autoscaling and additional dynamic resource management capabilities offered by Kubernetes and other cluster management systems.

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.

REPLICATED

Deliver Kubernetes Apps Anywhere

Take the friction out of shipping enterprise-grade modern applications securely

Software vendors are modernizing their apps with containerization and Kubernetes. They need to deliver those apps to enterprises whose own infrastructure modernization journeys may be slower, more complex, and not necessarily containerized themselves. Shipping modern software to these diverse, complex, customer-controlled environments is *hard* and keeps vendors from doing what they do best – developing great apps.

Replicated helps software vendors ship their Kubernetes apps with **flexibility**, **speed**, and **ease**, **for any customer** (running Kubernetes or not) on-prem, in air gap, or in the cloud. This is ***multi-prem***.

REPLICATED

The State of On-Prem: Modern Solutions for a Traditional Problem

On-prem software, and the parallel rise of Kubernetes

01. Introduction
02. Avoiding on-premises = leaving money on the table
03. Key findings and results at a glance
04. On-premises software continues to grow to meet customer demand
07. Container-based deployments are the norm
09. On-premises deployments are challenging and slow
11. The solution
12. Looking forward
13. About page

On-premises software continues to grow to meet customer needs

92% of companies indicate on-premises software sales are growing

Download Now

On-Prem Trends

Over 90% of respondents reported growth in on-prem sales over the past 5 years.

Revenue Growth

Over 50% of respondents' revenue can be attributed to on-prem software sales.

Kubernetes Everywhere

86% of respondents are using Kubernetes for their on-prem software.

replicated.com

CASE STUDY

Case Study: Kubos

Kubos Leverages Replicated to Provide Air Gap Installs to Their Most Security-Conscious Customers

Kubos is a satellite software company that provides cloud technologies to the satellite industry via its mission control platform, Major Tom.

CHALLENGE

Kubos' mission is to bring modern software development and practices to the space industry by providing a robust set of API gateways and seamless integrations. When Kubos began to generate interest from military and government contracts around the globe, they soon realized that the requirements and policies in place to operate Major Tom under strict security protocols requiring on-premises/physical access wouldn't allow for data transmission through the cloud. Previously a SaaS product, Kubos suddenly needed to package their software for deployment into an air gapped environment — a project that they were not prepared to build out internally.

SOLUTION

Kubos uses Replicated to ship apps configured to the needs of their customers' specific environments with embedded troubleshooting tools for environmental conformance and configuration validation. Kubos' customers can take advantage of Replicated's full breadth of pre-packaged `kubectl` plugins to manage the disconnected applications in a single admin console. The Replicated platform provides commercial and open-source tools to streamline the distribution and management of third-party applications to complex enterprise environments — even the most secure environments with no or limited inbound and outbound internet access.

RESULT

By leveraging Replicated, Kubos saves operational costs and increases productivity by allowing their engineering team to refocus on development of their core products. As a small team, Kubos can scale their software delivery at a pace that building delivery capabilities internally would not have allowed. Most importantly, Kubos finally accepted contracts with customers that needed a level of data security that their previous SaaS-only offering could not provide.

Replicated delivered to Kubos:

- **An increased total addressable market** with the ability to take on higher-value enterprise and government contracts.
- **Operationalized development and distribution of their software** with Replicated's vendor and admin toolsets.
- **Reduced app maintenance** with Replicated's Day 2 support toolset, allowing them to troubleshoot customer air gap problems remotely through test instances.

KUBOS

COMPANY
Kubos

COMPANY SIZE
<10 employees | $10M

INDUSTRY
Defense and Space

PRODUCTS USED
Replicated and Kubernetes

PRIMARY OUTCOME
Increased revenue opportunity and operational scale through leveraging Replicated, while keeping internal resources focused on core products

"Replicated solved our problem immediately. Not only can we now provide an air gap solution for our customers, but Replicated keeps our on-prem offering easy and consistent with the rest of our product."

— **Tyler Browder**,
CEO, Kubos

CREATED IN PARTNERSHIP WITH

REPLICATED

CONTRIBUTOR INSIGHTS

Kubernetes Security Guide

High-Level K8s Hardening Guide Based on Recommendations From CNCF, NSA, and CISA

By Yitaek Hwang, Sr. Software Engineer II at Axoni

As more organizations have begun to embrace cloud-native technologies, Kubernetes adoption has become the industry standard for container orchestration. This shift toward Kubernetes has largely automated and simplified the deployment, scaling, and management of containerized applications, providing numerous benefits over legacy management protocols for traditional monolithic systems. However, securely managing Kubernetes at scale comes with a unique set of challenges, including hardening the cluster, securing the supply chain, and detecting threats at runtime.

This introduction to Kubernetes security combines best practices from the Cloud Native Computing Foundation (CNCF), National Security Agency (NSA), and Cybersecurity and Infrastructure Security Agency (CISA) to help organizations mitigate risks and adopt a multi-layered security approach.

Cluster Setup and Hardening

Securing a Kubernetes environment starts with hardening the cluster. For users of a managed Kubernetes service (e.g., GKE, EKS, AKS), the respective cloud provider manages the security of the master node and implements various secure-by-default settings for the cluster. GKE Autopilot takes additional measures, implementing GKE hardening guidelines and GCP security best practices. But even for GKE standard or EKS/AKS users, there are guidelines maintained by the cloud providers to secure access to the Kubernetes API server, container access to cloud resources, and Kubernetes upgrades:

- GKE Hardening Guide
- EKS Best Practices Guide for Security
- AKS Cluster Security

For self-managed Kubernetes clusters (e.g., `kube-adm`, `kops`), kube-bench can be used to test whether the cluster meets the security guidelines laid out in the CIS Kubernetes Benchmark. Key recommendations include encrypting the secrets stored in `etcd` at rest, protecting control plane communication with TLS certificates, and turning on audit logging.

NETWORK AND RESOURCE POLICIES

By default, Kubernetes allows communication from any pod to another pod within the same cluster. While this is ideal for service discovery, it provides zero network separation, allowing bad actors or compromised systems unlimited access to all resources. This becomes extremely problematic for teams using namespaces as the primary means of multi-tenancy inside Kubernetes.

To control the traffic flow between pods, namespaces, and external endpoints, use a CNI plugin that supports the NetworkPolicy API (e.g., calico, flannel, or cloud-specific CNI) for network isolation. Following the zero-trust model, the best practice is to implement a default deny-all policy to block all ingress and egress traffic unless it is specifically allowed by another policy.

In addition to network policies, Kubernetes provides two resource-level policies: `LimitRange` and `ResourceQuotas`. `LimitRanges` can be used to constrain individual resource usage (e.g., max 2 CPUs per pod), whereas `ResourceQuota` controls the aggregate resource usage (e.g., a total of 20 CPU in the dev namespace).

RBAC AND SERVICE ACCOUNTS

With strong network and resource policies in place, the next step is to enforce RBAC authorization to restrict access. Kubernetes admins can enforce RBAC to users and groups to access the cluster, as well as to restrict services from accessing resources within and external to the cluster (e.g., cloud-hosted databases).

Exercise caution in using the default service account that is mounted to every pod upon creation. Depending on the permissions given to the default service account, the pod may be granted more permissions than required. If the service does not require any specific communication with Kubernetes service, set `automountServiceAccountToken` to false to prevent mounting.

System Hardening

Now that the cluster is secure, the next step is to minimize the attack surface on the systems. This applies to the OS running on the nodes as well as the kernel on the containers. Instead of general-purpose Linux nodes, opt for a specialized OS optimized for running containers, such as AWS Bottlerocket or GKE COS.

Next, take advantage of Linux kernel security features, such as SELinux, AppArmor (beta since 1.4), and/or seccomp (stable since 1.19). AppArmor defines the permissions for a Linux user or group to confine programs to a limited set of resources. Once an AppArmor profile is defined, pods with AppArmor annotations will enforce those rules.

```
apiVersion: v1
kind: Pod
metadata:
  name: apparmor
  annotations:
    container.apparmor.security.beta.kubernetes.io/hello: localhost/k8s-apparmor-example-deny-write
spec:
  containers:
  - name: hello
    image: busybox
    command: [ "sh", "-c", "echo 'Hello AppArmor!' && sleep 1h" ]
```

Seccomp, on the other hand, restricts a container's syscalls. As long as seccomp profiles are available on the underlying Kubernetes node, seccomp profiles can be defined under the `securityContext` section:

```
apiVersion: v1
kind: Pod
metadata:
  name: audit-pod
  labels:
    app: audit-pod
spec:
  securityContext:
    seccompProfile:
      type: Localhost
      localhostProfile: profiles/audit.json
  containers:
  - name: test-container
    image: hashicorp/http-echo:0.2.3
    args:
    - "-text=just made some syscalls!"
```

Even if seccomp profiles are not available, users can still restrict the container from various privilege escalation attacks. Under security contexts, Kubernetes allows configuring whether the container can run as privileged, root, or escalate privileges to root. Users can also restrict `hostPID`, `hostIPC`, `hostNetwork`, and `hostPaths`. All of these settings can be enforced via the Pod Security Policy (which was deprecated in v1.21) or with other open-source tools, such as K-Rail, Kyverno, and OPA/Gatekeeper.

Finally, if additional security assurance is required, a custom `RuntimeClass` can be configured to take advantage of hardware virtualization (e.g., gVisor, kata). Define the `RuntimeClass` at the node level and specify it under the pod definition:

(See code block on next page)

```
apiVersion: node.k8s.io/v1 # RuntimeClass is defined in the node.k8s.io API group
kind: RuntimeClass
metadata:
  name: myclass # The name the RuntimeClass will be referenced by
  # RuntimeClass is a non-namespaced resource
handler: myconfiguration # The name of the corresponding CRI configuration
---
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  runtimeClassName: myclass
```

Supply Chain Security

Even if the cluster and system are secure, to ensure end-to-end security of the entire application, the supply chain must also be taken into consideration. For applications developed in house, follow the best practices for creating containers. Namely, use a minimal base image to reduce the attack surface, pin package versions, and use multi-stage builds to create small images. Also, define a non-root user the container must run with, or build rootless containers with `podman` to restrict root access.

Next, scan all images for vulnerabilities using open-source tools (e.g., Trivy, Clair, Anchore) or commercial tools (e.g., Xray from Artifactory or container scanning on cloud provider build process). Some tools also allow signing images and verifying the signatures to ensure that the containers were not tampered with during the build and upload process. Finally, define whitelisted registries that Kubernetes can pull images from using `ImagePolicyWebhook` or any policy enforcement tool mentioned above.

Monitoring, Logging, and Runtime Security

At this point, we have a secure cluster with a locked-down supply chain that produces clean, verified images with limited permissions. However, environments are dynamic, and security teams must be able to respond to incidents in running environments. First and foremost, ensure immutability of containers at runtime by setting `readOnlyRootFilesystem` to true and storing tmp log files to an `emptyDir`.

On top of the typical application monitoring (e.g., Prometheus/Grafana) or logging (e.g., EFK), analyze syscall processes and Kubernetes API logs using Falco or Sysdig. Both tools can parse Linux system calls from the kernel at runtime and trigger alerts when a rule is violated. Example rules include alerting when privilege escalation occurs, when read/write events are detected on well-known directories, or when a shell is invoked. Finally, integrate Kubernetes API audit logs with existing log aggregation and alerting tools to monitor all activities in the cluster. These include API request history, performance metrics, deployments, resource consumption, OS calls, and network traffic.

Conclusion

Due to the complex nature of cloud-native systems, a multi-layered approach is required to secure a Kubernetes environment. Kubernetes recommends the 4Cs of cloud-native security: cloud, cluster, container, and code. Start by hardening the cluster and following best practices for cloud security. Then, lock down the container, reduce the attack surface, limit access, and ensure immutability at runtime. Next, secure the supply chain and analyze the code and container for vulnerabilities. Finally, monitor all activity at runtime to build defense into every layer of your software running inside Kubernetes.

Yitaek Hwang, Sr. Software Engineer II at Axoni
@yitaek on DZone | @yitaekhwang on LinkedIn | www.yitaekhwang.com

Yitaek Hwang is a Senior Software Engineer at Axoni focused on building infrastructure and developer tools for high-performance teams. He often writes about cloud, DevOps/SRE, and crypto topics.

CONTRIBUTOR INSIGHTS

An Overview of Popular Open-Source Kubernetes Tools

Exploring Cloud-Native Development, Monitoring, and Deployment

By Samir Behara, Platform Architect at EBSCO

Kubernetes is the industry-standard technology used by enterprises to deliver microservices-based container orchestration platforms. The Kubernetes ecosystem is growing rapidly and has a vibrant community that has built several free open-source tools and extensions to make it easier to run your workloads. This article will explain some of the more popular open-source Kubernetes tools used for development, monitoring, and deployment purposes that can improve your cloud-native experience.

Kubernetes CLI Tool: kubectl

`Kubectl` is the command-line tool that enables you to run commands against your Kubernetes cluster. You can refer to the `kubectl` installation document to install the tool on any operating system, including Windows, Linux, or macOS. Once installed, you can inspect and manage the cluster by running various command-line operations to create new cluster resources, view the details, and delete resources as required. Please refer to the `kubectl` cheat sheet here that contains the most used Kubernetes commands.

Minikube: Running Kubernetes Locally

If you want to get started with Kubernetes and run a cluster locally on your machine for application development or training, Minikube is the easiest way. Minikube runs a single-node Kubernetes cluster and lets you interact with the cluster by writing `kubectl` commands. Installing Minikube is straightforward, and it greatly simplifies the developer experience. You can also monitor your Kubernetes cluster via the browser and visually interpret the cluster state by accessing the Kubernetes dashboard from Minikube. Table 1 lists some common Minikube commands:

Table 1

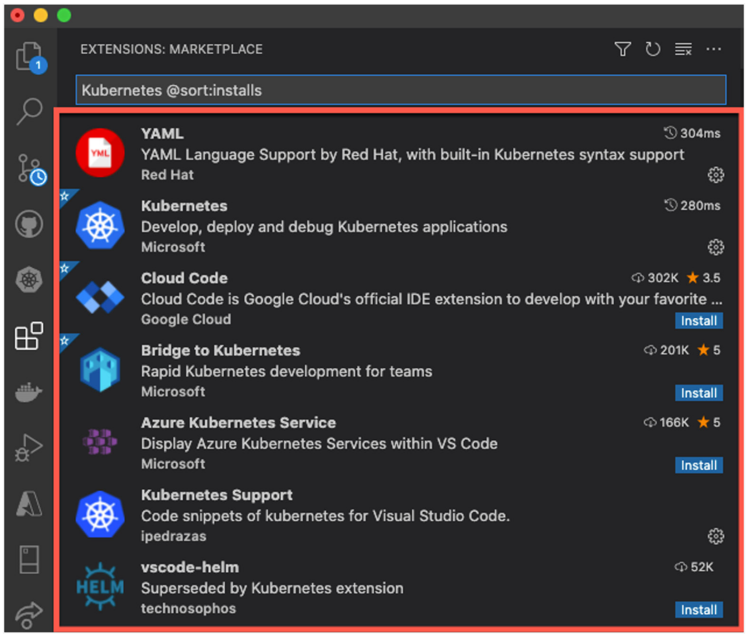
Command	Definition
`minikube status`	Check if Minikube is running
`minikube start`	Start the Kubernetes cluster
`minikube stop`	Stop the Kubernetes cluster
`minikube dashboard`	Open the Kubernetes dashboard in the browser
`minikube delete`	Delete the Minikube VM

Kubernetes Extensions for Visual Studio Code

VS Code is one of the most popular open-source IDEs and provides numerous features to improve the cloud-native development experience. VS Code also offers many extensions that assist in developing apps in Kubernetes environments. The VS Code extensions for Kubernetes simplify microservice development and help interact with Kubernetes clusters, irrespective of where they get deployed — AWS, Azure, GCP, on-premises, etc.

The Kubernetes extension lets you connect to the clusters and drill down into workloads, services, nodes, and pods. You also can modify the manifest files for resources, install Helm charts, and apply them to the cluster right from the IDE. The YAML plugin provides autocomplete, error detection, formatting, document outlining, and syntax validation functionalities in Kubernetes YAML definitions. The Kubernetes Support plugin includes code snippets to create resources like pods, services, and deployments quickly. Developers can leverage similar extensions available in the marketplace that can increase their productivity by automating repetitive actions.

Figure 1: Popular Kubernetes extensions in VS Code



EXTENSIONS: MARKETPLACE

Kubernetes @sort:installs

YAML 304ms
YAML Language Support by Red Hat, with built-in Kubernetes syntax support
Red Hat

Kubernetes 280ms
Develop, deploy and debug Kubernetes applications
Microsoft

Cloud Code 302K ★ 3.5
Cloud Code is Google Cloud's official IDE extension to develop with your favorite ...
Google Cloud
Install

Bridge to Kubernetes 201K ★ 5
Rapid Kubernetes development for teams
Microsoft
Install

Azure Kubernetes Service 166K ★ 5
Display Azure Kubernetes Services within VS Code
Microsoft
Install

Kubernetes Support
Code snippets of kubernetes for Visual Studio Code.
ipedrazas

vscode-helm 52K
Superseded by Kubernetes extension
technosophos
Install

Helm: K8s Package Manager for Repeatable Deployments

Helm is an open-source package manager that reduces the complexity of deploying cloud-native applications on Kubernetes and makes releases repeatable and reliable. Helm Charts are pre-configured Kubernetes resources, applications, or bundles of YAML configuration files that can be templatized and reused across different environments. Developers can use Helm to configure, package, and deploy applications to Kubernetes clusters for a complex business process. There is no need to hardcode configurations when deploying your applications to multiple environments. Many Helm Charts are publicly available in Helm repositories and are a great way to promote the reusability of charts. You can install several Helm Charts on your Kubernetes cluster — like Prometheus, Istio, Jaeger, Fluentd, Jenkins, ArgoCD, and more.

Figure 2: Introduction to Helm

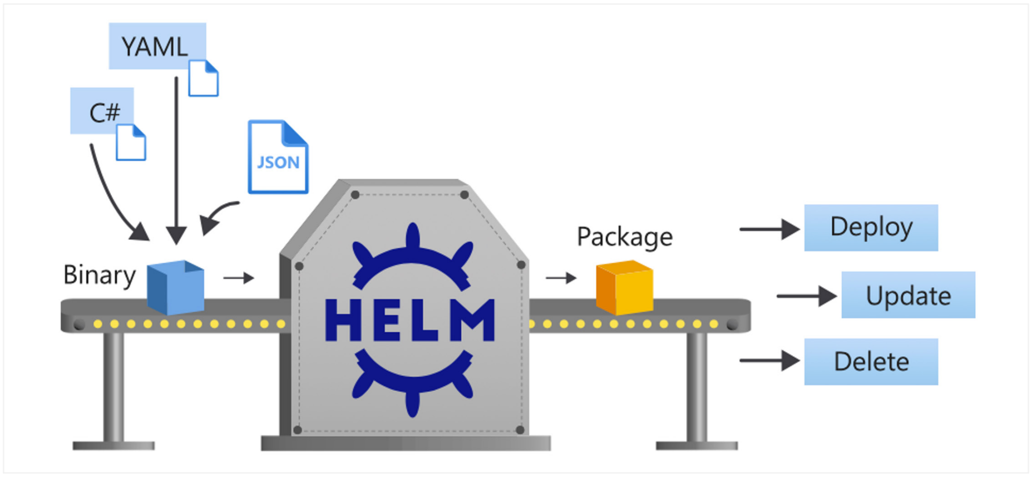


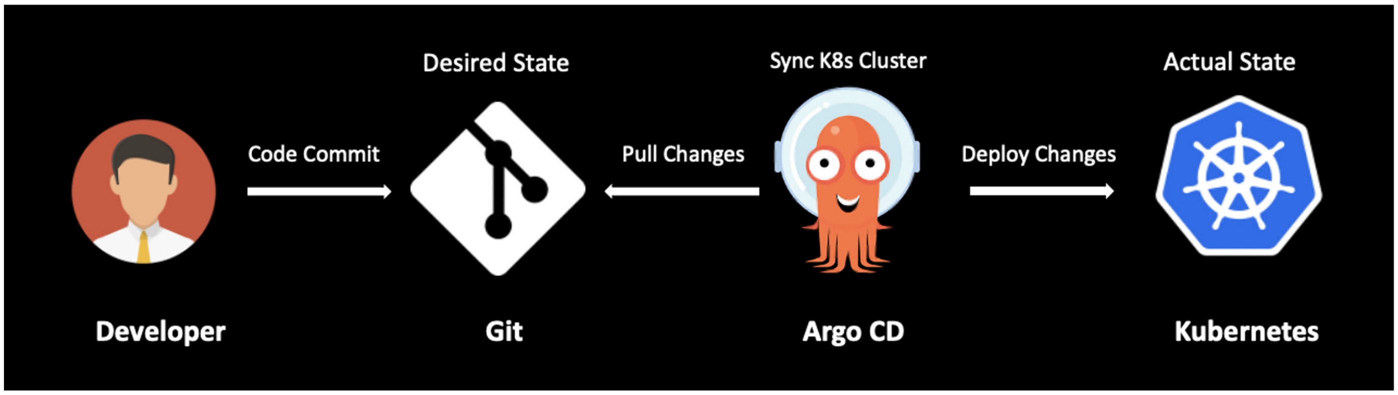
Image source: "Application and package management using Helm — Introduction," Microsoft Docs

Argo CD: Declarative GitOps Continuous Delivery Tool

Argo CD is a declarative GitOps tool to deploy your applications seamlessly to Kubernetes. Argo CD allows you to automatically synchronize the Kubernetes cluster state with the desired state stored in Git. Behind the scenes, Argo CD runs as a controller within the Kubernetes cluster, watches for changes to the Git repository, compares it against the resources deployed in the cluster, and synchronizes state. The GitOps process increases the productivity of the developers since they don't have to deal with the deployment of their codebase into the cluster and make manual changes to the underlying infrastructure.

Argo CD provides an interactive user dashboard that displays details about the services and resources deployed in the Kubernetes cluster. It helps you visualize the health of the applications deployed into the cluster and provides automated and manual sync to the desired state. Argo CD automatically detects configuration drift issues and marks the deployment as out of sync. You can manually trigger the sync process or use the auto-sync functionality in Argo CD to sync when it detects configuration changes.

Figure 3: Integrate Argo CD into your Kubernetes deployment workflow



Prometheus: Kubernetes Monitoring at Scale

Prometheus is the leading open-source, metrics-based Kubernetes monitoring tool. It has a powerful query language (PromQL) with a multidimensional data model that stores time-series data. Prometheus is a pull-based monitoring system that scrapes real-time metrics from the applications running in a Kubernetes cluster using exporters. These metrics get stored in local storage that can be queried using PromQL or viewed via Grafana dashboards. You can also configure the Alertmanager to handle the alerting strategy and send intelligent alerts to different notification systems like OpsGenie, Pagerduty, email, etc.

Figure 4: Prometheus architecture

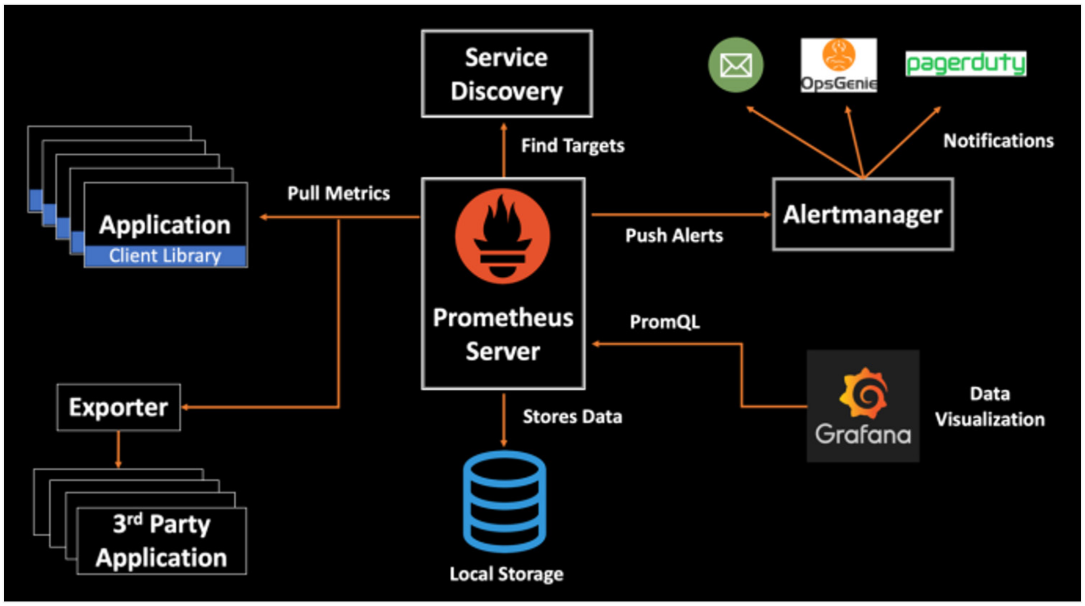


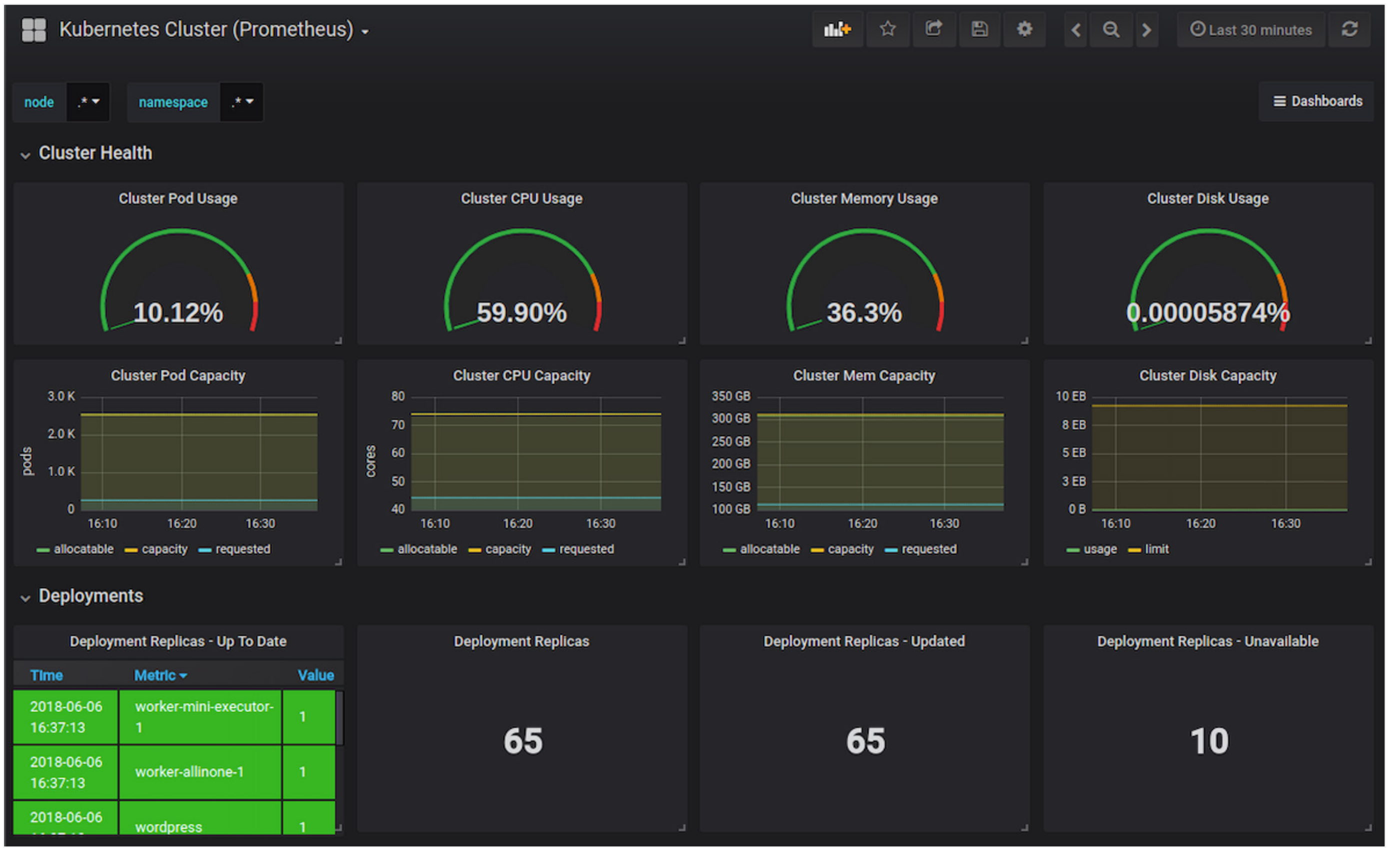
Image source: "Cloud Native Monitoring with Prometheus" samirbehara.com

Grafana: Observability and Dashboards

Grafana is the most popular data visualization and analytics tool that helps you understand complex time-series data. Grafana allows you to query, visualize, and alert on metrics stored in different data stores, such as Prometheus, Elasticsearch, InfluxDB, MySQL, PostgreSQL, AWS Cloudwatch, Azure Monitor, Graphite, etc. In addition, apart from creating custom monitoring dashboards, you can leverage the official and community-built open-source dashboards for monitoring Kubernetes clusters. From a Kubernetes monitoring perspective, you can have dashboards to display real-time CPU and memory utilization metrics for your cluster, nodes, and pods.

For troubleshooting issues, you should have metrics showing container restarts, throttled or unhealthy pods, resource requests, and limits in your dashboards.

Figure 5: Grafana monitoring dashboard for a Kubernetes cluster



Deployment Replicas - Up To Date

Time	Metric	Value
2018-06-06 16:37:13	worker-mini-executor-1	1
2018-06-06 16:37:13	worker-allinone-1	1
2018-06-06	wordpress	1

Image source: "Kubernetes Cluster (Prometheus)," Grafana Dashboards

Istio: Service Mesh

A service mesh design pattern simplifies the management of running a distributed microservices architecture. A service mesh is an infrastructure layer that handles service-to-service communication. It provides functionalities like traffic management, authentication, security, load balancing, service discovery, telemetry, fault injection, and circuit breaking without any application changes. The sidecar design pattern handles this complexity by deploying a sidecar proxy alongside your services. All traffic to your services goes through the proxy.

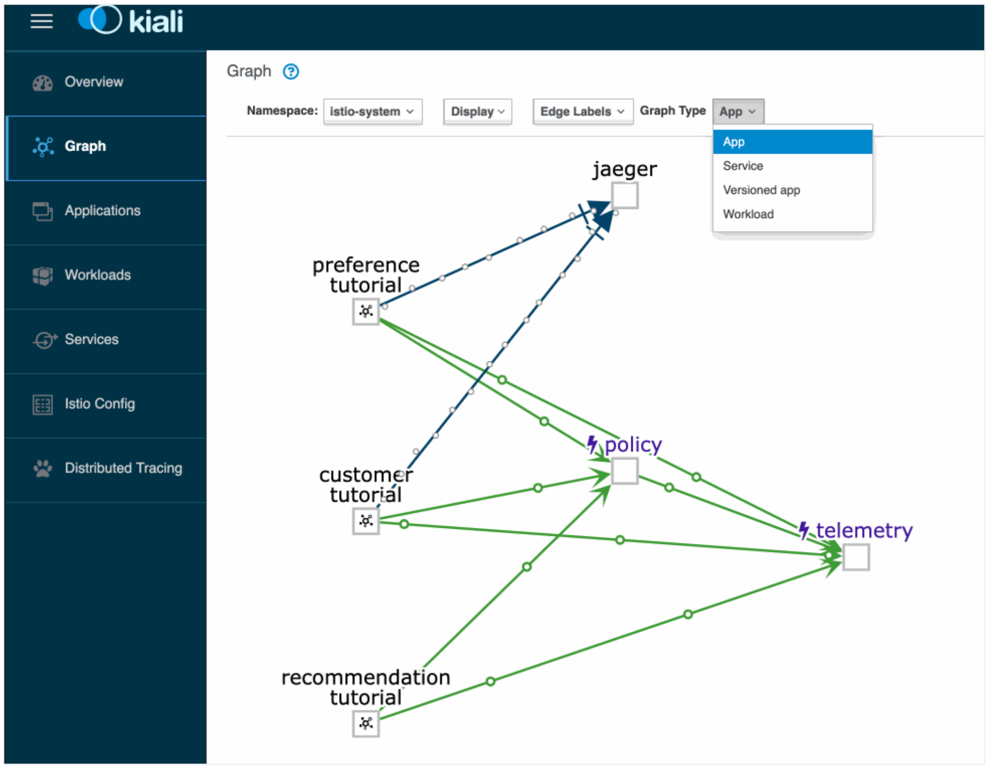
Envoy from Lyft is one of the most popular open-source proxies designed for cloud-native applications. Envoy runs alongside every service and provides the necessary features in a platform-agnostic manner. Istio is a popular service mesh framework that uses Lyft's Envoy as the sidecar proxy by default.

With Kubernetes deployments growing in size and complexity, it is becoming increasingly difficult to manage service functionalities. Istio comes to the rescue here by shifting this complexity from the application to the infrastructure layer. Istio, Envoy, and Kubernetes generally work together to operate a distributed microservice-based workload at scale.

Kiali: Service Mesh Observability

When you have microservices running in a Kubernetes environment and leveraging a service mesh, it becomes critical to have a strategy to monitor, visualize, and have complete visibility into your mesh. Kiali is a management console for Istio that provides in-depth observability into the service mesh topology. Kiali also displays the real-time traffic patterns within your mesh, shows the connectivity between microservices, and helps evaluate the health of the services running inside the cluster. Like Grafana, Kiali uses the metrics data stored in Prometheus and displays them in the console.

Figure 6: Kiali console showing a request flow between services



Conclusion

With the rise of distributed computing and container orchestration technologies, many Kubernetes tools are available to simplify your experience building and managing cloud-native applications. Selecting the right tool for your infrastructure that fits your architecture and business use case is critical. I encourage you to read through the open-source tools I have provided in this article and keep yourself up to date with the evolving community in the cloud-native space.

Samir Behara, Platform Architect at EBSCO

@samirbehara on DZone | @samirbehara on LinkedIn | Author of samirbehara.com

Samir Behara is a Platform Architect with EBSCO and builds software solutions using cutting edge technologies. He is a Microsoft Data Platform MVP with over 15 years of IT experience. Samir is a frequent speaker at technical conferences and is the Co-Chapter Lead of the Steel City SQL Server user group.

CONTRIBUTOR INSIGHTS

Advanced Kubernetes Deployment Strategies

Exploring Core Kubernetes Deployment Concepts, Use Cases, and Benefits

By Sudip Sengupta, Technical Writer at Javelynn

In the modern technology landscape, Kubernetes is a widely adopted platform that enables organizations to deploy and manage applications at scale. The container orchestration platform simplifies infrastructure provisioning for microservice-based applications, which empowers efficient workload management through modularity. Kubernetes supports various deployment resources to help implement CI/CD pipelines using updates and versioning. While Kubernetes offers **rolling updates** as the default deployment strategy, several use cases require a non-conventional approach to deploying or updating cluster services.

This article reviews concepts in Kubernetes deployment, as well as delves into various advanced Kubernetes deployment strategies, pros and cons, and use cases.

Kubernetes Deployment Concepts

Kubernetes uses deployment resources to update applications declaratively. With deployments, cluster administrators define an application's lifecycle and how related updates should be performed. Kubernetes deployments offer an automated way to achieve and maintain the desired state for cluster objects and applications. The Kubernetes back end manages the deployment process without manual intervention, offering a safe and repeatable way of performing application updates.

Kubernetes deployments allow cluster administrators to:

- Deploy a pod or replica set
- Update replica sets and pods
- Roll back to earlier versions
- Pause/continue deployments
- Scale deployments

The following section explores how Kubernetes simplifies the update process for containerized applications, and how it solves the challenges of continuous delivery.

KUBERNETES OBJECTS

While Kubernetes leverages a number of workload resource objects as persistent entities to manage the cluster state, the Kubernetes API uses the Deployment, ReplicaSet, StatefulSet, and DaemonSet resources for declarative updates to an application.

DEPLOYMENT

Deployment is a Kubernetes resource used to define and identify the application's desired state. A cluster administrator describes the desired state in the deployment's YAML file, which is used by the deployment controller to gradually change the actual state to the desired state. To ensure high availability, the deployment controller also constantly monitors and replaces failed cluster nodes and pods with healthy ones.

REPLICASET

A ReplicaSet is used to maintain a specific number of pods, ensuring high availability. The ReplicaSet's manifest file includes fields for:

- A selector to identify the pods that belong to the set
- The number of replicas, which shows how many pods should be in the set
- A pod template to show what data the new pods should create to meet the ReplicaSet's criteria

STATEFULSET

The StatefulSet object manages the deployment and scaling of pods in a stateful application. This resource manages the pods based on identical container specifications and then ensures appropriate ordering and uniqueness for a set of pods. The StatefulSet's persistent pod identifiers enable cluster administrators to connect their workloads to persistent storage volumes with guaranteed availability.

DAEMONSET

DaemonSets help to maintain application deployments by ensuring that a group of nodes runs a copy of a pod. A DaemonSet resource is mostly used to manage the deployment and lifecycle of various agents such as:

- Cluster storage agents on every node
- Log collection daemons
- Node monitoring daemons

Details on the list of various Kubernetes workload resources can also be found here.

UPDATING WITH DEPLOYMENTS

Kubernetes Deployments offer a predictable approach to starting and stopping pods. These resources make it easy to deploy, roll back changes, and manage the software release cycle iteratively and autonomously. Kubernetes offers various deployment strategies to enable smaller, more frequent updates as they offer benefits such as:

- Faster customer feedback for better feature optimization
- Reduced time to market
- Improved productivity in DevOps teams

By default, Kubernetes offers **rolling updates** as the standard deployment strategy, which involves replacing one pod at a time with a new version to avoid cluster downtime. Besides this, depending on the goal and type of features, Kubernetes also supports various advanced deployment strategies — these include *blue-green*, *canary*, and *A/B deployments*.

Let us take a closer look at what these strategies offer and how they differ from each other.

Advanced Strategies for Kubernetes Deployments

Kubernetes offers multiple ways to release application updates and features depending on the use case and workloads involved. In live production environments, it is crucial to use deployment configurations in conjunction with routing features so that updates impact specific versions. This enables release teams to test for the effectiveness of updated features in live environments before committing full versions. Kubernetes supports advanced deployment strategies so that developers can precisely control the flow of traffic toward particular versions.

BLUE-GREEN DEPLOYMENT

In the blue-green strategy, both the old and new instances of the application are deployed simultaneously. Users have access to the existing version (blue), while the new version (green) is available to site reliability engineering (SRE) and QA teams with an equal number of instances. Once QA teams have verified that the green version passes all the testing requirements, users are redirected to the new version. This is achieved by updating the `version` label in the `selector` field of the load balancing service.

Blue-green deployment is mostly applicable when developers want to avoid versioning issues.

USING THE BLUE-GREEN DEPLOYMENT STRATEGY

Let us assume the first version of the application is `v1.0.0` while the second version available is `v2.0.0`.

Below is the service pointing to the first version:

```
apiVersion: v1
kind: Service
metadata:
  name: darwin-service-a
spec:
  type: LoadBalancer
  selector:
    app: nginx
    version: v1.0.0
  ports:
       - name: http
         port: 80
         targetPort: 80
```

And here is the service pointing to the second version:

```
apiVersion: v1
kind: Service
metadata:
  name: darwin-service-b
spec:
  type: LoadBalancer
  selector:
    app: nginx
    version: v2.0.0
  ports:
       - name: http
         port: 80
         targetPort: http
```

Once the required tests are performed and the second version is approved, the first service's selector is changed to `v2.0.0`:

```
apiVersion: v1
kind: Service
metadata:
  name: darwin-service-a
spec:
  type: LoadBalancer
  selector:
    app: nginx
    version: v2.0.0
  ports:
       - name: http
         port: 80
         targetPort: http
```

If the application behaves as expected, `v1.0.0` is discarded.

CANARY DEPLOYMENT

In the canary strategy, a subset of users is routed to the pods that are hosting the new version. This subset is increased progressively while those connected to the old version are reduced. This strategy involves comparing the subsets of users connected to the two versions. If no bugs are detected, the new version is rolled out to the rest of the users.

USING THE CANARY DEPLOYMENT STRATEGY

The process for a native Kubernetes canary deployment involves the following:

1. Deploy the needed number of replicas to run version 1 by:

   Deploying the first application:

   ```
   $ kubectl apply -f darwin-v1.yaml
   ```

   Scaling it up to the needed number of replicas:

   ```
   $ kubectl scale --replicas=9 deploy darwin-v1
   ```

2. Deploy an instance of version 2:

   ```
   $ kubectl apply -f darwin-v2.yaml
   ```

3. Test if the second version was successfully deployed:

   ```
   $ service=$(minikube service darwin --url)
   $ while sleep 0.1; do curl "$service"; done
   ```

4. If the deployment is successful, scale up the number of instances of version 2:

   ```
   $ kubectl scale --replicas=10 deploy darwin-v2
   ```

5. Once all replicas are up, you can delete version 1 gracefully:

   ```
   $ kubectl delete deploy darwin-v1
   ```

A/B DEPLOYMENT

With A/B deployments, administrators can route a specific subset of users to a newer version with a few limitations and/or conditions. These deployments are mostly performed to assess the user base's response to certain features. A/B deployment is also referred as a "dark launch" since users are uninformed on the inclusion of newer features during testing.

USING THE A/B DEPLOYMENT STRATEGY

Here's how to perform A/B testing using the Istio service mesh, which helps roll out the versions using traffic weight:

1. Assuming Istio is already installed on the cluster, the first step is to deploy both versions of the application:

   ```
   $ kubectl apply -f darwin-v1.yaml -f darwin-v2.yaml
   ```

2. The versions can then be exposed via the Istio Gateway to match requests to the first service using the command:

   ```
   $ kubectl apply -f ./gateway.yaml -f ./virtualservice.yaml
   ```

3. The Istio `VirtualService` rule can then be applied based on weight using the command:

   ```
   $ kubectl apply -f ./virtualservice-weight.yaml
   ```

This splits traffic weight among versions on a 1:10 ratio. To shift the weight of traffic, the weight of each service is edited, after which the `VirtualService` rule is updated through the Kubernetes CLI.

When to Use Each Advanced Deployment Strategy

Since Kubernetes use cases vary based on availability requirements, budgetary constraints, available resources, and other considerations, there is no one-size-fits-all deployment strategy. Here are quick takeaways to consider when it comes to choosing the right deployment strategy:

Table 1

COMPARING KUBERNETES DEPLOYMENT STRATEGIES

Strategy	Takeaway	Key Pros	Key Cons
Blue-green	• Focuses on progressive delivery, which is crucial for testing functionality on an application's back end	• Enables instant rollouts and rollbacks • Allows administrators to change the entire cluster's state in one upgrade • Eliminates versioning issues	• Requires twice the number of resources and proper platform testing before production release
Canary	• Involves testing the new version while users still run instances of an older version • Is considered best for avoiding API versioning issues	• Is convenient to monitor performance through error-rate comparison • Enables rapid rollback • Includes user experience testing	• Is costly to fine-tune the distribution of traffic • Performs slower rollouts
A/B	• Involves availing both old and new application versions to users, then comparing their experiences • Is mostly used for front-end deployments and in instances where QA testing processes are insufficient	• Permits several versions to run in parallel • Enables performance monitoring	• Causes slow rollouts • Creates costly traffic balancing

Summary

Kubernetes objects are among the technology's core functionality, allowing rapid delivery of application updates and features. With deployment resources, Kubernetes administrators can establish an efficient versioning system to manage releases while ensuring minimal to zero application downtime. Deployments allow administrators to update pods, roll back to earlier versions, or scale up infrastructure to facilitate growing workloads.

The advanced Kubernetes deployment strategies covered here also enable administrators to route traffic and requests toward specific versions, allowing for live testing and error processing. These strategies are used to ensure newer features work as planned before the administrators and developers fully commit the changes. While deployment resources form the foundation of persisting application state, it is always recommended to diligently choose the right deployment strategy, prepare adequate rollback options, and consider the dynamic nature of the ecosystem that relies on multiple loosely coupled services.

Additional resources:

- Using kubectl to Create a Deployment
- Kubernetes Deployment Use Cases
- States of a Kubernetes Deployment Lifecycle

Sudip Sengupta, Technical Writer at Javelynn

@ssengupta3 on DZone | @ssengupta3 on LinkedIn | www.javelynn.com

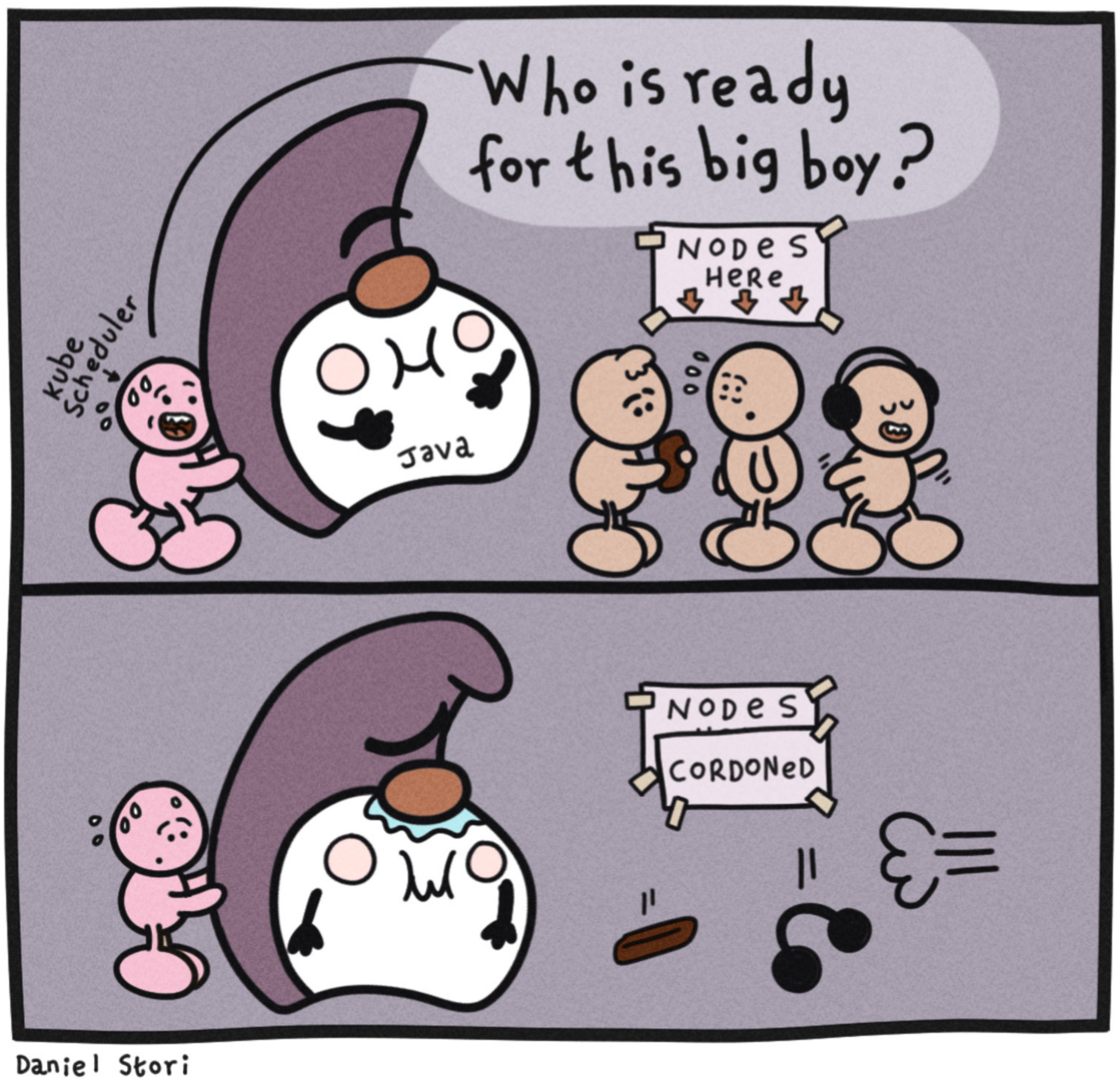
Sudip Sengupta is a TOGAF Certified Solutions Architect with more than 15 years of experience working for global majors such as CSC, Hewlett Packard Enterprise, and DXC Technology. Sudip now works as a full-time tech writer, focusing on Cloud, DevOps, SaaS, and cybersecurity. When not writing or reading, he's likely on the squash court or playing chess.

CONTRIBUTOR INSIGHTS

Scheduler Life

$ kubectl get fun

By Daniel Stori, Software Engineer at AWS



Daniel Stori, Software Engineer at AWS

@Daniel Stori on DZone | @dstori on LinkedIn | @turnoff_us on Twitter | turnoff.us

I started to code for fun on an Apple II at the end of the '80s, and professionally, in the middle of the '90s, so I have extensive experience in the field. I love to draw comics — much more than I have been able to create since my daughter was born. I've recently joined the AWS team to create a learning platform based on 3D games.

CONTRIBUTOR INSIGHTS

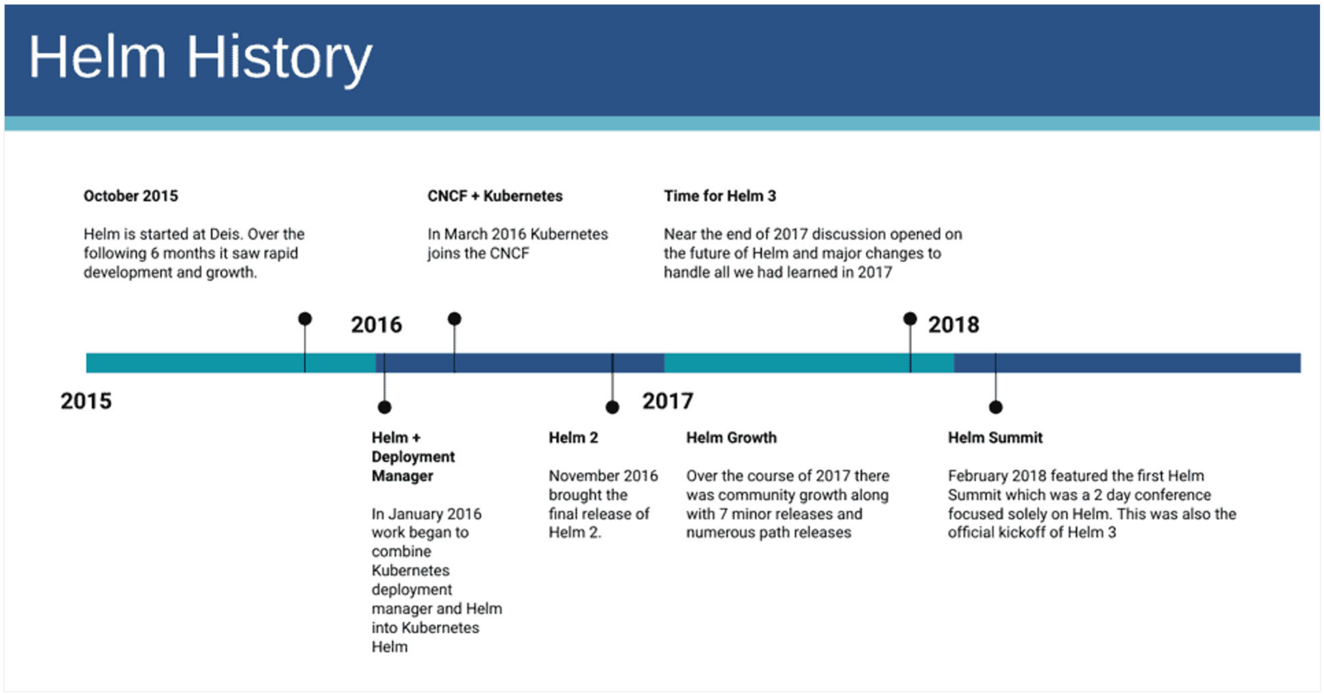
Kubernetes Package Management With Helm

By Ray Elenteny, Solution Architect at SOLTECH

Introducing Helm

Helm is a package manager for Kubernetes. Given a running Kubernetes cluster of any type, Helm is used to manage the deployment lifecycle of just about any type of Kubernetes resource, including the management of many Kubernetes runtime components. A very common analogy used in describing Helm is that Helm is to Kubernetes as apt is to Debian-based systems and yum or rpm is to Red Hat-based systems. Beyond package management, many aspects of configuration management are also built into Helm. Helm was initially developed by a company named Deis, which was acquired by Microsoft. Microsoft fully supported and accelerated the development of Helm, and it is now a part of the Cloud Native Computing Foundation (CNCF).[1]

Figure 1


Helm History

October 2015
Helm is started at Deis. Over the following 6 months it saw rapid development and growth.

CNCF + Kubernetes
In March 2016 Kubernetes joins the CNCF

Time for Helm 3
Near the end of 2017 discussion opened on the future of Helm and major changes to handle all we had learned in 2017

Helm + Deployment Manager
In January 2016 work began to combine Kubernetes deployment manager and Helm into Kubernetes Helm

Helm 2
November 2016 brought the final release of Helm 2.

Helm Growth
Over the course of 2017 there was community growth along with 7 minor releases and numerous path releases

Helm Summit
February 2018 featured the first Helm Summit which was a 2 day conference focused solely on Helm. This was also the official kickoff of Helm 3

Image source: "Helm history" — a slide from the Helm project presentation to CNCF TOC, May 2018

As part of the CNCF, Helm is actively developed and supported by numerous organizations, and it has since developed a large and active community. The following is a sample of Helm project contribution statistics as of October 2021:

Table 1

Contributors	Code Commits	Pull Requests	Contributions
15,449	17,079	16,897	152,442

Table source: "Overall Project Statistics Table," Helm DevStats Project of the CNCF

[1] https://www.cncf.io/reports/cncf-helm-project-journey-report/

Why a Package Manager for Kubernetes?

Like most technologies, "Hello World" examples introduce key concepts, and Kubernetes is no exception. Deploying anything beyond the most straightforward component to a Kubernetes cluster requires coordination across multiple components. For example, the process for managing application deployment outside a Kubernetes cluster is not complex. However, it is tedious since there are dependencies and dependency versions, configuration artifacts, pre- and post-deployment steps, validation, etc. Just as apt and yum manage this process for Linux, Helm handles it for Kubernetes.

Helm Features

- Kubernetes management of components' and applications' deployment lifecycle
- Template-based definition that supports portability across deployment environments (e.g., Development, QA, Production)
- Hooks mechanism to inject use-case-specific code at various points in the deployment lifecycle
- Deployment testing framework

HELM CONSTRUCTS

Using Helm is a matter of installing a single executable. The `helm` command provides more than 20 parameters used in building, deploying, deleting, rolling back, etc. the deployment of an application to a Kubernetes cluster.[2]

The Helm deployment artifact is a Helm Chart. Helm Charts consist of resources used to deploy a component or application to a Kubernetes cluster. The most common resources within a chart are YAML files, which follow standard Kubernetes resource descriptions. For those experienced in deploying to a Kubernetes cluster using commands such as `kubectl create` or `kubectl apply`, the YAML files within a Helm Chart will look familiar. Helm Charts will often contain additional resources, such as README files, a default parameters file, and additional files (such as certificates) required for deployment.

Developing a Helm Chart requires assembling files using a predefined directory structure. The Helm command, `helm create <chart name>`, creates a Helm Chart, which is the predefined directory structure and includes some sample files. The generated chart contains several YAML files. A Kubernetes deployment often requires multiple Kubernetes resource descriptions to be deployed, and in many cases, there's an order of precedence to which those deployments must occur. When deploying manually, the order must be known. This is not the case with Helm since Helm is aware of the order of precedence of Kubernetes resource descriptions.

A key feature to the Helm deployment process is Chart Hooks. During a Helm Chart's deployment lifecycle, Chart Hooks are a mechanism by which additional tasks are performed. Helm supports several points to introduce a Chart Hook:

Table 2

Chart Hook	Description
`pre-install`	Perform tasks prior to an application's deployment
`post-install`	Perform tasks after an application's deployment
`pre-delete`	Perform tasks prior to removing an application from the cluster
`post-delete`	Perform tasks after an application has been removed from the cluster
`pre-upgrade`	Perform tasks prior to executing a deployed application's upgrade process
`post-upgrade`	Perform tasks after a deployed application's upgrade process has completed
`pre-rollback`	Perform tasks prior to executing a deployed application rollback process
`post-rollback`	Perform tasks after a deployed application's rollback process has completed
`test`	Executes Helm tests as defined in the Helm Chart

Table source: "The Available Hooks — Chart Hooks," Helm Documentation

[2] https://helm.sh/docs/helm/helm/#see-also

Helm Charts can depend on other Helm Charts. For example, an application may contain dependencies on a collection of microservices where a microservice is defined by its own Helm Chart. When the application is deployed, Helm manages the dependencies. In keeping with the microservice pattern, each one can be updated independently of the rest so that it is still a cohesive part of an application's collective definition.

Planning a Helm Deployment

Helm plays a part in all aspects of application development and deployment, and that requires engineering and operations teams to work closely together to design solutions and answer deployment questions. With the teams coordinating, deployment decisions can be made iteratively, accommodating the variance in each deployment environment with the goal of having a single deployment package to support each environment.

Beyond the concept of hooks previously described, Helm enables teams to tackle this challenge of a single deployment package by providing a robust templating mechanism. Typically, YAML files in a Helm Chart do not look like a handwritten YAML Kubernetes resource description. Rather, YAML files in a Helm Chart are developed using Helm's template language:

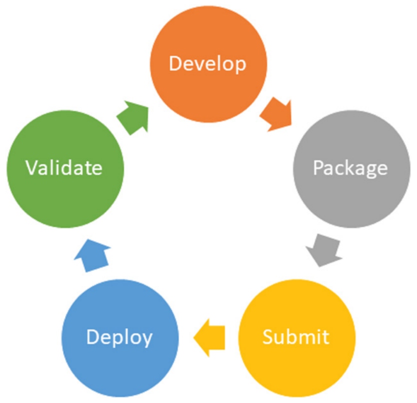
```
{{- if .Values.ingress.enabled -}}
{{- $fullName := include "helm-demo.fullname" . -}}
{{- $svcPort := .Values.service.port -}}
{{- if semverCompare ">=1.14-0" .Capabilities.KubeVersion.GitVersion -}}
apiVersion: networking.k8s.io/v1beta1
{{- else -}}
apiVersion: extensions/v1beta1
{{- end }}
kind: Ingress
metadata:
  name: {{ $fullName }}
  labels:
    {{- include "helm-demo.labels" . | nindent 4 }}
  {{- with .Values.ingress.annotations }}
  annotations:
    {{- toYaml . | nindent 4 }}
  {{- end }}
spec:
  {{- if .Values.ingress.tls }}
  tls:
    {{- range .Values.ingress.tls }}
    - hosts:
        {{- range .hosts }}
        - {{ . | quote }}
        {{- end }}
      secretName: {{ .secretName }}
    {{- end }}
  {{- end }}
  rules:
    {{- range .Values.ingress.hosts }}
    - host: {{ .host | quote }}
      http:
        paths:
          {{- range .paths }}
          - path: {{ .path }}
            backend:
              serviceName: {{ $fullName }}
              servicePort: {{ $svcPort }}
          {{- end }}
    {{- end }}
  {{- end }}
```

This sample ingress description, as generated by `helm create`, is templated, providing several variables by which an ingress resource is defined and configured, including whether the ingress resource should even be created. With templating, Helm offers a great deal of control over how Kubernetes resources are deployed. A well-planned templating pattern leads to producing a single deployment package that enables a Helm Chart to deploy successfully, ranging from a single-node Kubernetes cluster on a developer's workstation to production Kubernetes clusters.

Helm Charts and CI/CD

As part of an organization's continuous integration/continuous delivery pipeline, Helm plays the roles of enabler and component. As an enabler, it enhances a pipeline by becoming the mechanism that deploys applications or components across the spectrum of environments (engineering, QA, staging, certification, production, etc.). Automating Helm Chart deployments is straightforward within a CI/CD pipeline.

As an application component, just as application code is iteratively developed and deployed, so are Helm Charts. This means that the CI/CD pipeline is integral in validating the Helm Chart itself. In fact, a Helm Chart should be considered part of the application code and not treated as a peripheral aspect of an application's development process — even going so far as to include and manage the Helm Chart as part of the application's source code. Similar to how an application build might produce a versioned container image and push it to an image registry, `helm package` bundles a chart into a versioned archive. The resulting archive is submitted to a Helm Chart repository, from which it can be accessed for deployment.



The diagram above highlights stages in an application's software development lifecycle. Whichever pattern is used in managing a Helm Chart's source code, its participation in an application's CI/CD pipeline is as integral as the application itself.

The Maturing of the Kubernetes Toolset

Helm has been a part of the Kubernetes ecosystem hype curve, and, as the Kubernetes hype curve has started to flatten, Helm has matured as well. Is Helm revolutionary in its approach? Not really. Helm leverages years of knowledge gained with the package and configuration management tools that have come before it, bringing that experience to Kubernetes. At the same time, Helm's perspective in defining deployment packages via Helm Charts makes a direct impact on the efficiency of an organization's CI/CD pipeline, most notably around configuration patterns and deployment flexibility. A well-designed Helm Chart is an integral component of effective delivery.

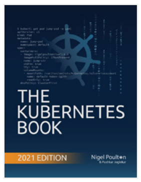
Ray Elenteny, Solution Architect at SOLTECH
@relenteny on DZone | @ray-elenteny on LinkedIn | @relenteny on GitHub

Ray is a Solution Architect at SOLTECH, an award-winning software development and IT staffing organization. With 35 years of experience, Ray thoroughly enjoys sharing his experience by helping organizations deliver high-quality applications that drive business value. Ray has a passion for software engineering, and over the past ten years, Ray has taken a keen interest in the cultural and technical dynamics of efficiently delivering applications.

ADDITIONAL RESOURCES

Diving Deeper Into Kubernetes

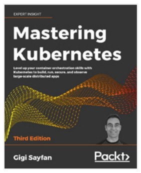
BOOKS



The Kubernetes Book

By Nigel Poulton

This book covers Kubernetes end to end — first explaining concepts in a learner-friendly manner, and then everything readers should know to become Kubernetes connoisseurs. Each subsequent edition contains the latest on Kubernetes, and in this updated version, you'll learn about its architecture, security practices, API, and how to deploy, scale, perform rolling updates, and more!



Mastering Kubernetes: Level up your container orchestration skills [...]

By Gigi Sayfan

Readers will dive into complex concepts and core practices of Kubernetes in this book aimed to help existing Kubernetes users master the advanced skills needed for designing and deploying large clusters on multiple cloud platforms. Concepts include running complex stateful microservices with features like horizontal pod autoscaling and resource quotas, plus using service meshes and serverless computing.

TREND REPORTS

Containers

Today's mainstream shift toward cloud native, including modernizing architecture with containers, promises to accelerate development. However, new obstacles arise from a fundamentally transformed software delivery pipeline in vital areas like security. In this Trend Report, you'll explore container adoption, common pain points of implementation in legacy environments, and modern practices for building scalable, secure, and performant containerized applications.

Kubernetes and the Enterprise

Look back on DZone's 2020 "Kubernetes and the Enterprise" report to see how the industry has evolved over the last year! The basis of our 2021 research began here, now enabling year-over-year insights into technical developments around Kubernetes. And as always, readers can examine their peers' perspectives in expert articles on topics like microservices architecture scalability, cluster management, and deployment strategies.

REFCARDS

Kubernetes Multi-Cluster Management and Governance

The performance nature of cloud-native apps requires that Kubernetes environments are highly distributed. Proper visibility and management across multiple Kubernetes clusters is *a must* to ensure consistent, secure operations throughout all environments. This Refcard explains multi-cluster management — its architecture, operations, isolation patterns, and useful tools — as well as Kubernetes governance and core practices for a successful governance strategy.

Advanced Kubernetes

While many learning resources on the fundamentals of Kubernetes exist, there are fewer that focus on more complex skills, concepts, and use cases. In this Refcard, readers learn how to manage Kubernetes infrastructure on an ongoing basis through quick, accessible information covering everything from kubectl commands and node management to cluster and node testing, troubleshooting practices, and debugging techniques.

PODCASTS

Kubernetes Podcast

Considering that Google produces it (and also created Kubernetes in 2014) you might call this podcast a classic. Enjoy news and interviews with prominent tech folks who work with K8s in weekly episodes hosted by Craig Box, a leader in the Cloud Native advocacy team at Google.

The Cloudcast

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

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.