LIQUID SOFTWARE

How to Achieve Trusted Continuous Updates in the DevOps World

Fred Simon, Yoav Landman, Baruch Sadogursky

Title: Liquid Software

ISBN: 13: 978-1981855728

Subtitle: How to Achieve Trusted Continuous Updates in the DevOps World

Authors: Fred Simon, Yoav Landman, Baruch Sadogursky

To contact the authors or the copyright holder,
please send an email to LSbook@jfrog.com

Liquid Software is dedicated to everyone who has
ever wished machines would work for us,
instead of the other way around.

Acknowledgements

This book has been a journey for us. A journey that explored the present and looked into what we believe the software industry will look like in the not too distant future. A journey that has forced us to plumb the depths of our experience as we brainstormed together to shape our thoughts into a coherent vision.

Among those who joined us on the journey, we would particularly like to thank: Shlomi Ben Haim – for his total support of our vision for *Liquid Software*; Kit Merker – for his thorough review of the texts and thought provoking, insightful comments; Rami Honig and Shani Levy – for paying attention to every detail, while tirelessly driving this work to completion; and Jody Ben-David and Steve Spencer of J-R Research – for their invaluable contributions to getting our thoughts and vision down on paper.

THE END

We'd like to tell you that we're software soothsayers, capable of predicting with pinpoint accuracy where the industry as a whole will be ten years from now. We'd love to say that the **liquid software revolution of continuous updates with zero downtime** will lead to ponies, rainbows, and Happy Ever Afters for everyone. Of course, we can't do that.

We do know, however, that breakthroughs *can and do* occur. The kind that radically change our perceptions of the possible. The kind that fundamentally alter what we manufacture and consume. We're confident that the adoption of continuous updates will be *that* transformative. It will accelerate with the rise of cloud computing and the Internet of Things, as those and other technologies will demand it. The new normal that is still evolving includes: anywhere, anytime, always running, fully interconnected, transparent, cross-platform computing. People want every software-driven thing to seamlessly integrate with all other software-driven things.

Software already runs practically everything that keeps modern society functioning. There is, and will continue to be, demand for more software, and for software that is ever more responsive and versatile. As software becomes more complex, more mistakes will be made. Updates will need to occur with greater regularity, whether they are new functionalities or patches. The only practical way to accommodate these rising and accelerating demands is to make software more liquid.

Liquefaction also makes sense in terms of user psychology and preference. Our greatest digital achievements happen when average users don't see or concern themselves with the inner mechanics of their software-powered devices. All the engineering ingenuity and prowess stays behind the curtain

in service of easy, intuitive operations. Every time we eliminate a confusing or irritating technical procedure, while delivering more and better functionality, everyone's a winner.

Consider this tremendously powerful argument in favor of liquid software: *Spectre*. Publicly disclosed in January 2018, Spectre is a vulnerability that affects microprocessors that perform branch prediction. It fools computer and device applications into accessing arbitrary sectors of their memory space. This gives attackers the ability to read that memory and potentially obtain sensitive data. It's extremely pernicious, and its impacts are far-reaching. All current CPU architectures are vulnerable! Darkest of all, there's no protection against it. Spectre-based exploits are only discoverable *after* they've been applied and the damage has been done. That's the kind of five-alarm fire that demands rapid response. Continuous updates are currently the quickest and best way to solve the problem –securely, and without incurring downtime.

Barriers to the acceptance and implementation of liquid software are multi-faceted – the most significant being developers' unfamiliarity with continuous update methodologies and the DevOps practices that we believe are fundamental to ensuring success. Even those who do have some knowledge of these matters have concerns. Everyone in the industry would like to provide updates with greater speed, flexibility, and transparency. So, conceptually, liquid software scores big points. The issue is how to achieve these goals and deliver software that's secure and able to maintain high levels of uninterrupted productivity. Do we keep nursing along legacies, or do we become the pioneers of innovation?

Blazing new trails is in the nature of things. These are next generation software ideas. And new generations are often prepared to work in fundamentally different ways than those that have preceded them. They dream of things that never

were. Their motto is: If it ain't broke, break it! That's how new paradigms are born.

Let the revolution begin!

Fred Simon
Yoav Landman
Baruch Sadogursky

May 2018

CHAPTER 1:
THE ROAD TO DISRUPTION

"Learning and innovation go hand in hand.
The arrogance of success is to think that what you did
yesterday will be sufficient for tomorrow."
– William G. Pollard, physicist

Not Your Father's Software Release

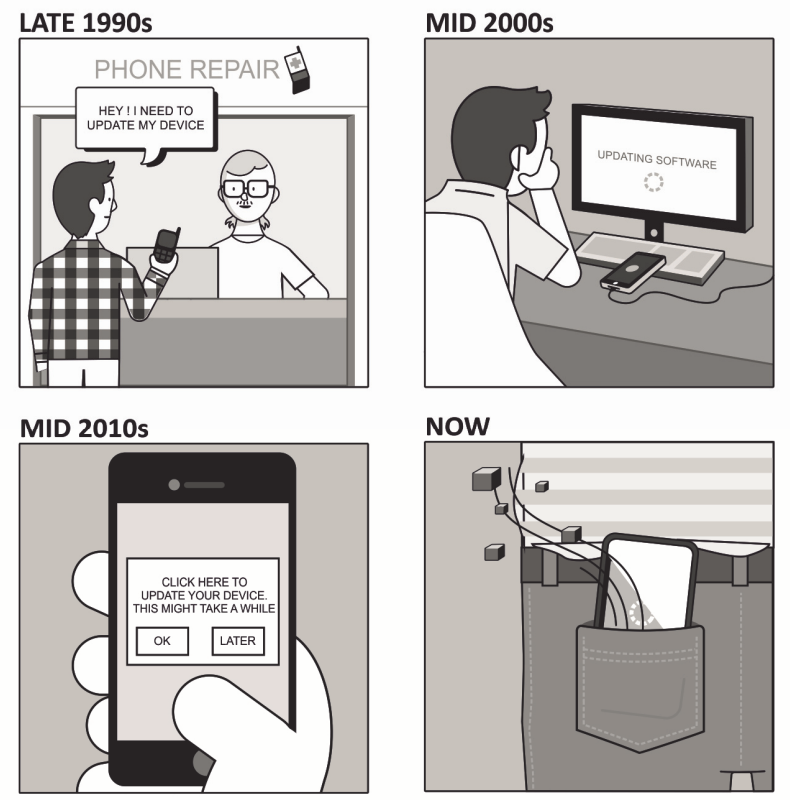
Changes in the ways software, and software updates, are conceived, developed, and deployed – and in the nature of how software operates – are changing the way R&D works. Continuous improvement is the goal. To achieve this, the era of big releases is coming to an end.

Continuous improvement is not only about continuous development and deployment of software. It is an adjustment to how the marketplace operates. For the past several decades, software has been sold as a commodity, or a good. A customer would pay a price to own a license for a piece of software or a software package. Revenue generated would be immediately transactional, with customers paying directly to acquire it. The marketplace is shifting, and it will continue to shift away from this model toward one in which software consumption is fluid and revenues are generated not as one-time payments, but as a constant stream, as users access a software *service*. This is particularly significant for newer software vendors. If large upfront sums of cash are no longer secured through big major releases, it becomes more difficult to set aside necessary sums for personnel-heavy and capital-intensive research and development (R&D) and quality assurance (QA). The push, then, is toward continuous improvement that can coexist alongside development.

This concept is not to be confused with continuous deployment, which is usually associated with installing new versions to runtimes in data centers and production systems that are strictly under a given company's control. In that environment, it is usually taken for granted that each company will have a firm grasp and understanding of the runtimes to which deployments are being pushed. **Continuous updating that produces continual improvement** is the rational expansion of this approach. This means establishing the reliable and secure manner by which

companies handle runtimes that they can push updates to, or pull updates from. It's a paradigm shift that's already underway.

This radical, yet highly logical response to the challenges of our increasingly software-driven world is the **liquid software revolution**.



The Evolution of Mobile Updates

Liquid Software?

In the traditional software scenario, an update is developed, delivered, and installed as an individual stand-alone item. It arrives as a neat little (or big) file which thousands – or even millions – of users open, and voila –

there's the update. Continuous, fluid (liquid) delivery and deployment of updates, on the other hand, is like the constant, unending flow of a stream or river. It includes the monitoring of this flow, and unceasing interactivity with the software that is being continuously updated.

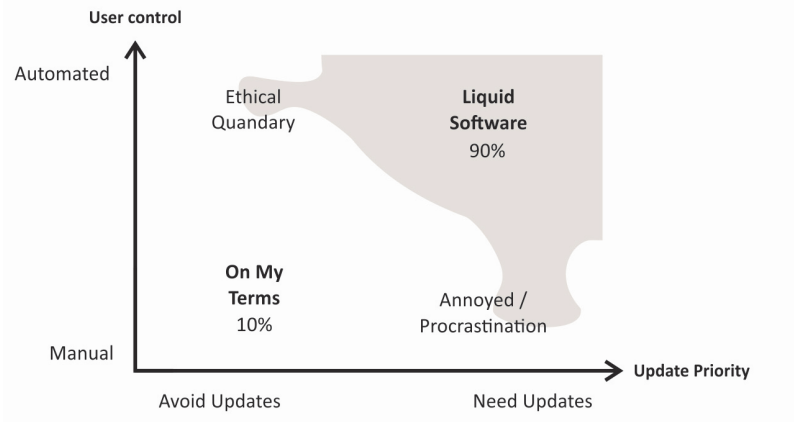
Liquid software is able to continuously update itself because it's simultaneously impacting and communicating data from something that is already running and in use by end-users. How those customers use and interact with their software will evidence new demands, unforeseen limitations, and other issues, which establish the basis for software to be improved by way of adjustments or the addition of new features.

While of course a discrete system undergirds it, liquid software is made up of tiny pieces, like drops of water make up the ocean. And there are so many of these pieces that it becomes no longer possible for any single person to distinguish individual components. The liquid software revolution now taking place is the transition from transferring packages to transferring micro-deltas of software. This order of magnitude advancement in software DevOps is being spurred on by a world that needs software (and software companies) to be ever more responsive to market demands, and disruptive of old (non-informed) ways of doing things. Development, testing, distribution, and implementation processes are getting faster and faster, with smaller and smaller bits being delivered to more and more environments.

We have reached the stage where the creation, bundling, transmission, and installation of big packages are impediments to business growth and productivity. As well, the ability of government, NGOs, and other service providers to assist more people, and to operate more efficiently, is restrained. This is keeping all of us from using and updating everyday device software in what could be an almost completely transparent manner. Our present experiences

make us desire a world in which continuous-and-seamless is the norm. While most users are unaware of it, our collective expectations are fueling the liquid software revolution.

This is not to suggest that we should plunge headlong into this future, ignoring the challenges that liquid updates can present. We can't ignore the way we humans absorb changes. There's no sense or purpose in pushing modifications that disrupt the present expectations of users. As ingenious and cutting edge as we can be on the technological side of things, we need to be every bit as clever in the ways we handle user experience, change management, feature promotion, training and support services, etc. Such concerns won't apply to every release, but they'll certainly pertain to those that are most likely to be disruptive. These, in particular, should be thought of as design constraints that give us pause to consider whether the adoption costs associated with an improvement can really be justified. Optimally, new features should be designed to be intuitive, requiring zero training; they should feel natural, as if they've always been here, just waiting for users to discover them. This approach will improve performance and connectivity, and ultimately lead to software that's faster, more secure, and easier to navigate.



Driving Forces Behind Liquid Software

The Source (Code) of Liquidity

The drive toward liquidity is fueled by user needs and desires as well as by the software developer community. Much of this drive has come about because more developers are becoming involved with open source projects. As these people communicate and create code together, the nature of the processes in which they are engaged represent a stark example of how software can be developed more quickly and in a profoundly more efficient and inventive manner than can be accomplished in a tech firm's office. These software craftspeople – through their easy access to better tools at home – have established the framework for how software can and should be created, built, and distributed in the future.

And let's be crystal clear. We're not waxing poetic about the creativity of DevOps professionals, or positing academic notions about efficiency. The liquid software revolution is as much about return on investment as it is about radically transforming processes. Companies create software to assist customers to achieve goals, and if they get the job done right, their bottom lines will reflect that success. Increasingly, doing it right is seen as the adoption of small, efficient methods pioneered in the open source community. Large software firms are realizing that the path to better software and greater profit is through the establishment of a continual feedback process between their enterprise customers (and end-users in general) and the software they are creating. Building the pipelines (the liquid software infrastructure) and the management systems for ongoing, continuous operations can dramatically shorten the time span between perceiving user needs that should be accommodated (or detecting bugs that require patching), and making software improvements. This overall improvement in quality, and the consistent delivery of continuous improvements, will be rewarded. Software companies that join the liquid software revolution

will see their costs decline and their productivity, marketplace respect, and profits increase.

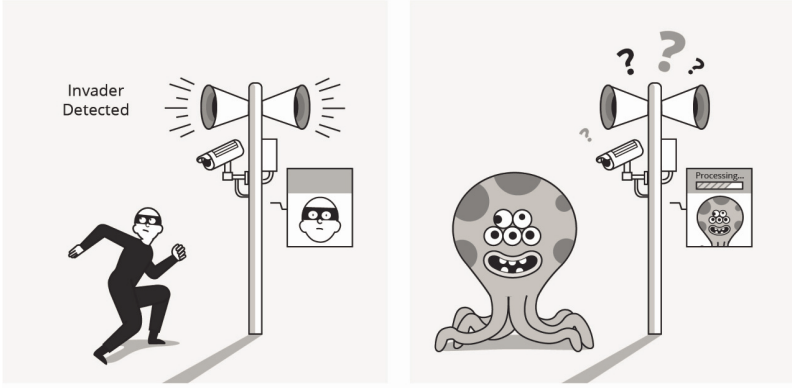
As we have seen with many innovations in many different industries, change is not always immediate, nor is it always rapidly embraced. Large software companies have entrenched cultures and ways of doing business that often have evolved over the course of many years. It's frequently not enough for an innovation to present itself as a great creative idea or even a new approach that's winning the hearts and minds of the geek class. It's the real, tangible results being produced by liquid software revolutionaries that are now motivating the industry as a whole to pay attention to what's happening, and in a desire not to be left behind, to learn what they need to do to transform their own operations.

An especially interesting aspect of the liquid software revolution is the cross-boundary nature of the developer community. Developers are moving from industry to industry, sharing their knowledge and their craft with their developer peers, whether those individuals are working in the automotive, manufacturing, or retail world. It's not about a specific, industrial goal, but rather about how developers – all developers – are creating software. We are now experiencing a wave of cross-industry adoption of new DevOps practices, tools, and technologies. It's reached the point where every industry – and even every company – knows it must ride this wave. Those already on board are benefiting from making the shift, and many others are acknowledging the need to move in this direction.

When Software Starts to Wilt

Software is everywhere. And *every* company is a software company. The trend toward the digitalization of anything and everything is growing rapidly. This trend shows every sign of continuing into the foreseeable future, and beyond. We see

this exponential growth particularly in the Internet of Things (IoT), where devices, home appliances, automobiles, and so much more, are already or are soon to become a part of our "smart" world of electronics. People are becoming accustomed to devices that respond to their behaviors, and that respond situationally to the environments in which they operate. In the coming new normal, the digital adjuncts of daily life will routinely function in the ways individuals and businesses want them to. In this new normal, users will instinctively wonder why a particular functionality is not performing as well as it might or is not available at all. This need for software to quickly adapt to immediate circumstances, and to practically intuit what will best serve situations yet to come, cannot be adequately addressed even at the current state of software evolution, where continuous *deployments* are quite common, but continuous *updates* are not.



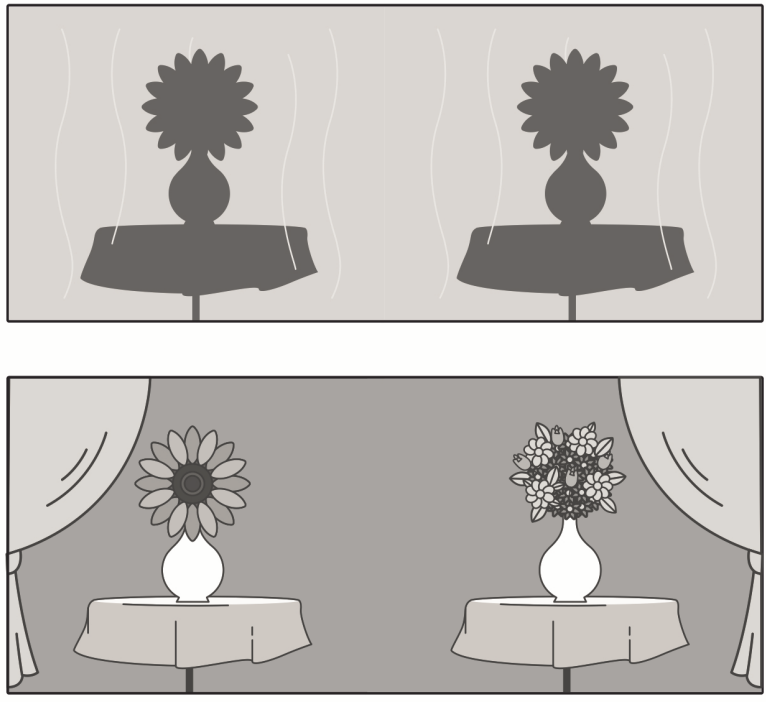
When Software Can't Adapt to Changing Environments

The continuous deployment of software versions is often seen today, particularly with mobile devices. Apps running on these devices are frequently being updated, which is an advancement that assures enhanced quality, as user needs and software issues are being handled with increased speed.

However, for the most part, the actual changes from one version release to the next are getting smaller and smaller. Furthermore, the software being updated is generally operating on a production runtime with existing data and live requests. The liquid software revolution is intensely concerned with the miniaturization of these updates – creating the mechanisms by which improvements, adjustments, and patches can instantly be incorporated into running systems.

We might conceptualize this in terms of a bouquet of flowers that we purchase to beautify a room. Once set in a vase of water, our bouquet has been installed and deployed. Through time and exposure to its environment, the bouquet will change. The normal and anticipated process of aging and decay will take place, making the bouquet less “functional”. The useful life of the bouquet might be prematurely cut short simply because one or two individual flowers within the presentation become less attractive.

In this bouquet runtime scenario, what if we could make the bouquet self-perpetuating? What if every time a flower in that vase began to lose its appeal, it would instantly be renewed? What if the vase’s water supply was continuously replenished? We, the users of the flower arrangement, would no longer have to tend to it. Everything would be done for us. Our purpose would be served, and we could go about our other business.



Software That Never Wilts

What might happen if this newfangled bouquet technology – let's call it *eBouquet* – was really a thing, right now? It is likely that we would see the rapid adoption of eBouquet and a decline in marketplace interest in the older bouquet technology, and its innovations would become the norm. This might well inspire the market to wonder what if, instead of individual flowers in an arrangement auto-rejuvenating themselves, those flowers could automatically be replaced by different flowers. Maybe the arrangement could be programmed to slowly morph into a new arrangement. The new one might be more appropriate for a particular occasion or time of year. We could well imagine many more conceptual variations.

This metaphor illustrates something inherent in the human condition. Sure, necessity is the mother of invention

– but some of the world's most interesting, compelling, and useful advances have come from creative minds giving people things they never knew they wanted. At minimum, those innovators tapped into a general (sometimes unarticulated) sense that if we can already do one thing well, surely we should be able to do something else that builds on how far we have come.

It is precisely the same with software. The further down the road we go in terms of what software can do for us, the more intense is our desire for it to do more. This is a perfectly normal impulse. How often and in how many different ways since the Industrial Revolution has some grand innovation inspired people to exclaim, "I can't imagine how we lived all this time without [insert new wonder of the world here]!"

We've arrived at a moment in time where our expectations and assumptions about what software should do demand that we move beyond the deployment and lifecycle of traditional "bouquets". Greater progress and convenience intensify the desire for more of both. Users increasingly believe that their devices should operate in certain ways and that particular functionalities should be rapidly available to them as a matter of course. It will be impossible to make those beliefs manifest without continuous updates.

Naturally, before we can know where we are heading, we have to understand where we have been.

DevOps Rules! (At Least It Should)

The sudden, gargantuan demand for IoT devices has revealed just how far away we are from being able to deliver seamless, responsive, flexible, almost intuitive continuous updates. At present, most IoT devices have very low update rates. While the firmware in smart watches and fitness trackers may be updated every few weeks, updates are fewer

and farther between for smart home HVAC systems, smart TVs, health monitoring equipment in hospitals, and NASA space vehicles. Additionally, there are substantial issues related to security and trust, with failures occurring regularly. Hackers seeking to establish new and improper gateways to the Internet are routinely attacking IoT devices. In some instances, hacks are shockingly simple to execute, including some that cannot be reversed through a software patch, requiring customers to send their devices away for dedicated, hands-on, professional care.

Confronted with this, some manufacturers have chosen to hide behind the expectations people have of hardware refresh cycles, which are far less demanding than software update cycles, never mind continuous updates. These companies know that problems exist and that the bad street buzz generated by these problems is costing them business. So, what do they do? They either make it so inconvenient to update device software that users don't bother, or they stop providing updates altogether. Consider a smart TV that offers an update as an app is being launched, but allows the user to skip the update and launch the app anyway. The message on the screen doesn't explain what's in the update, why it should be installed, or how it could be installed when the TV is not being used. In this scenario, the odds are high that the average user will close the update message and go back to what they were doing.

Of course, this is not a practical solution in the short term, nor is it a sensible one in the long term, because the pressure to update will remain and only get stronger by the day. There's a tremendous tug of war going on now between end-users desiring the immediate ability to connect IoT devices to one another and to other networked devices, and the fact that the updates these devices receive are still not secure, transparent, or reliable. Moreover, even if we could tag an update as secure, a security flaw might be discovered

following release, with no mitigation option possible other than a patch or a dreaded device recall. All of this leaves software manufacturers only one prudent option – to embrace the liquid software revolution.

The transformation we envision isn't the acceptance of continuous updates as an abstract notion. A company can affirm the wisdom of moving in this direction and still fail in the effort if it does not also restructure internal systems and practices. Currently, most IoT developers and the IoT community that is creating the software for these devices are *not* DevOps personnel with a DevOps mindset. They bear the heritage of hardware producers accustomed to investing lengthy periods of time in production. They believe that any post-production updates, such as patches, can be inherently risky and are to be avoided. For many, DevOps is still a new concept. Still, whether through training, conferences, trade publications, peer or market pressures, DevOps *must* be a part of every IoT firm's business plan.

This should not be a tremendous hurdle to overcome. It's just applying to IoT environments and systems the same processes and techniques that are already in place at big web and web server companies. Among other things, it's creating QA testing tools, build tools, validation tools, promotion tools, signed software pipelines – indeed everything that is discussed throughout the book you are now reading. Nevertheless, some will need a little added incentive to get them to where they need to be; where sturdy, bottom line business sense dictates that they ought to be.

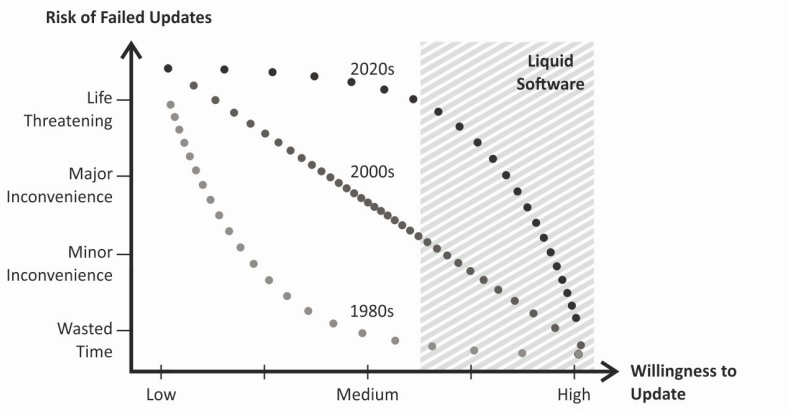
Consider this: In 1993, AT&T launched a series of sophisticated television ads, all of which posed provocative, "Have you ever…" questions, such as "Have you ever watched a movie you wanted to, the minute you wanted to?" "Have you ever kept an eye on your home when you're not at home?" and "Have you ever carried your medical history in your wallet?" The tag line for each of these commercials was,

"You will." And of course – every one of those prophecies have come to pass.

We'd like to be just as clear and direct: DevOps is not the future. It's here! It's now! And it's not going away. Yes, making the shift is an investment and is not to be taken lightly. But neither is it to be ignored. Every software firm, every CEO, every business development executive, and every operations manager owes it to their company, their board, their investors, their employees, and their clients to be curious about DevOps. They should devote some reasonable amount of time to the subject, and seek out those with the knowledge and experience to assist them in getting up to speed with what DevOps is all about. Those who do may well discover that DevOps is something they can't afford not to embrace. They might even be stunned to realize that the benefits can be staggering when theirs is the company providing users with products that have the built-in solutions and the in-house know-how to reliably, continually improve lives.

We Built It, But They Didn't Come

Need still more encouragement? Well, consider the fact that secure and highly accurate updates are already being delivered through automated systems all the time. It's being done to practically every commercial airplane on the ground for servicing, to sophisticated warehousing and logistics operations, and to many modern automotive systems. Even NASA's Mars rover, *Curiosity,* received a full system upgrade from a distance of almost 140 million miles.



Penetration of Continuous Updates through the Decades

Of course, the fact that we have the wherewithal to do the job right doesn't always mean the job is done right. Are there examples of mishaps and failures? Certainly. But those aren't reasons to slow down the progress of the liquid software revolution and the promise of continuous updates. Rather, we should be working in an organized and resolute fashion toward globalizing and standardizing. We should be communicating to the entire software industry that updates are not something to be considered after the fact. Updates should always be part of the code that is written right from the start. Our watchcry should be: "If it's not updateable, it's not software!"

We firmly believe that the liquid software revolution will succeed in making continuous updates the norm throughout the industry. Yet we do understand that we're currently travelling through a sort of middle passage. Much of the software we all currently use (web-based and mobile apps) is, in fact, being continuously updated. These are the updates we never really think about. We see them, but their inner workings are completely transparent to us. Almost all the websites we interact with are continuously being updated, and most mobile device apps receive automatic updates, but

the majority of users don't pay much attention to this. In other words, on a daily basis, users are already getting what they want and prefer – optimally functioning software with the latest feature and security updates, delivered without their having to be involved in the process. Yet for most people, the penny hasn't completely dropped. Most haven't quite reached the stage where they're wondering (or complaining) aloud about continuous update technology not being *everywhere* yet. Perhaps there would be more pressure from the market if the market was crystal clear in understanding that it could have what it wants at a significantly accelerated pace – if it would just make some noise about it. Perhaps we need to launch a high-profile advertising campaign with the tag line: "Ask your software provider if continuous updates are right for *you*!"

Whither Software Versioning

Throughout modern software history, localized installation and updating events have taken place in specific places, such as homes and offices, and on specific devices, such as PCs, laptops, servers, and mobile devices. New software and subsequent updates are assigned version numbers, which help to catalog the precise composition of any particular release of a given piece of software. These numbers are intended to highlight specific issues addressed, functionalities introduced, and patches applied. This paradigm is still very much with us today, although things are changing.

Change is evident across the landscape of mobile device apps. Many popular high-profile apps receive small updates as often as every few days. Significant change is also being driven by the fact that distributed software is becoming more common. What once was a component part of a software suite installed and running on a localized device may now be executed as a microservice that a user accesses and executes

in the cloud. In this environment, individual microservices can be updated discretely according to their own independent release cycles, with no need for their deployments to be bundled into a larger package of updates with a specific version number. Although each "micro-update" alters an aggregated macro version of a given piece of software, traditional version numbering is no longer an effective or meaningful way to reflect each and every minute change that takes place. This trend is dramatically on the rise with IoT introducing more new devices that are connected to the internet, the updating of which can only be efficiently managed through automatic updates requiring no human intervention.

The average user's awareness of software versioning is waning. Most software companies have embraced the fact that the vast majority of people care only about functionality and convenience. Whether for work or for leisure, users want to interact with the software in their lives only in ways that will help them to accomplish their goals. They want to *use* software, not *tend* to it. They certainly don't want to have to pay attention to its technical dimensions.

Making Informed Decisions?

Machines, not humans, need to do the logging and tracking of version numbers and software updates. They must be able to manage and adjust to a continuous liquid flow of very small packages that are continually updating software. Machines must monitor the impact such updates are having on the larger systems software operates within.

Versioning was created to assist professionals to better manage software updates. Machines, however, are more versatile at such management, as they have faster and better means of archiving and retrieving information. Machines can create versions of software packages, libraries, and applications. They can generate version numbers from many branches in parallel, and then, based on a machine-readable version, combine discrete packages into running software. And unlike humans, they have no need for text files detailing all the many versions of a piece of software that have been installed and updated on a particular platform or system.

Users and developers may not be expressing a desire for liquid software, because they are still unfamiliar with the

term. They may not articulate a desire for continuous updates, but are nevertheless eager for the benefits of automation. As it is now, most users do not upgrade major software packages, particularly computer operating systems. This is due in part to their fear of being involved in processes they believe are too technical or complicated. This fear persists even though software manufacturers have taken great care to make these experiences as simple and straightforward as possible. Most people are not confident that they possess the necessary knowledge or skill to handle such upgrades. Even those with some amount of savvy have learned (through rumor, if not experience) that it is often better to hold back on installing major upgrades, since initial releases have been introduced to the marketplace when they were less than ready for prime time.

This brings us back to what most people would prefer. In principle, most would very much like to have the latest and most improved versions of the software they use. However, they want by the best technicians. They want to be able to trust that what gets delivered to them has undergone appropriate testing and validation. They want to securely receive updates that will work properly and cause little or no disruption to their daily activities. Only machines are best equipped to satisfy all of these preferences.

Isn't Continuous Deployment Enough?

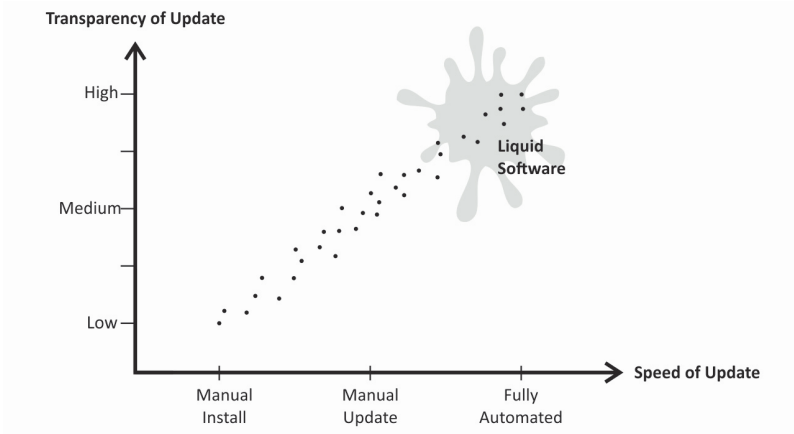
Right now, the answer to that question will depend on the end-user. With every passing day, however, the answer will increasingly be "no". We have already addressed the fact that software is everywhere and that IoT is exponentially reinforcing this. There are enterprises that require their software to be operational around the clock, and many average users want the same convenience. Only continuous updates can deliver on these demands and desires, as only

liquid software can provide continuous updates with zero downtime.

Another advantage afforded us by continuous updates is the opportunity to execute the odd-sounding, but very practical task of continuous downgrade. This option is highly relevant if, while a firm is running a critical operation, it suddenly detects something very wrong in a particular process. If an update had been delivered through continuous deployment, the company might be facing serious downtime and disruption of service. But the continuous downgrade procedure allows a rollback to be executed as seamlessly as an update. In a manner of thinking, it's not so much a downgrade as it is just another update, but this update is delivering a previous version of the software.

From Solid to Liquid

The demise of software versions – at least insofar as users are concerned – is already underway. And the degree to which it's happening parallels user confidence in the products and updates coming from software vendors. We see this with routers and self-updating IoT devices, and particularly with smartphones and tablets. The average user doesn't know (or care) what version of YouTube, WhatsApp, Amazon Echo, or Google Home is running. There *are* versions, but for all intents and purposes, they remain hidden. This information is pertinent to machines, not humans.



The Speed and Transparency of Going Liquid

So, we are all receiving and accepting app updates on almost a daily basis – most without the necessity for any intervention on our parts. We have confidence, at least, that software vendors are making sure that these app updates are not going to damage our devices or corrupt our personal data. The more we experience this (mobile devices and their apps serving our needs), the more we'll all be comfortable with updates taking place without our noticing.

Leaps of Faith

On our mobile devices, we typically allow app updates to be installed even when we know there will likely be no option to execute a rollback if something goes wrong. There's no user-side device testing of new version releases. They're placed directly into production with no acceptance tests needing to be run on the side. No one has a standby phone to use for the installation of new software updates.

Essentially, we're all taking an informed risk.

The risk most often encountered by running a new version of software is the loss of a functionality that we have enjoyed or relied on. Perhaps an app won't start at all because the device on which it's running has a new operating

system that doesn't match the new version of the software, or the software has some other bug. Even under these circumstances, we generally don't send complaint messages to software vendors about problematic app updates. Our experience has led us to expect that the vendor will deliver a fix as soon as possible. And we're willing to incur what is rarely more than a slight disruption, even if it's quite inconvenient while we wait for the solution to arrive.

Enterprise Users: To Boldly Go…

It is possible that a user may end up with a completely malfunctioning device as a result of choosing to upgrade its operating system. For enterprise customers, however, the level of voluntary risk is, for the most part, very different. It is much more common for them to have staging servers that can test and assess the impact of upgrades. A compelling argument can be made that when the enterprise user eventually does receive a given software update, they can have a reasonable sense of security. And while most of the time this is justified, nothing's totally foolproof. Problems can definitely be revealed during staging that might not otherwise be detected. The critical factor is whether an enterprise user is well prepared to predict production problems that may occur during staging.

This places enterprise users in only a marginally better and safer position than the phone user who simply accepts any updates (whether automatic or manual) that are being fed by vendors. And while enterprise users may be able to validate some updates, they can't validate every single one. This means that, at best, new versions of enterprise software have only limited and inconsistent opportunities to be tested with real-world production loads. This places intensifying pressures on software vendors from both the consumer and enterprise segments of the market. Both groups are

demanding trustworthiness, and that means vendors must ratchet up their game when it comes to validating software.

Further risks are incurred with our growing need for speed, which is fueled by the mindset and expectations of individuals in relation to their mobile devices. People are already experiencing just how quickly they can receive bug fixes and new features. So, it's natural for these same people not to understand why smooth and speedy updating cannot occur in the enterprise environment. Large firms may have the facility to carry out client-side validations that can also be automated and included as part of business-to-business, liquid software flows. However, once again, the fundamental issue here is for vendors to establish continuous update infrastructures capable of validating each and every update through necessary and appropriate testing.

A critical mass of everyday experience is pushing all levels of DevOps toward liquid software. Regardless of the software being run, regardless of the environments in which it's being run, regardless of the devices on which it's operating, regardless of whether the end-user is an individual or a corporation, we all want fluid and continuous updates.

To rapidly produce new features, bug fixes, and updated versions – few of which are overtly tangible anymore – we need to expand software capacities at a pace that is not possible to achieve via traditional updates.

It's Right Here in Front of You

Elements of a continuous update architecture *do* exist presently in large web application companies, such as Netflix, Apple, Google, Facebook, Amazon and Twitter. In these firms, from *Docker* to data center to website, liquefaction is all internal and based on proprietary systems. However, software that these companies consume from suppliers

outside of those systems, as well as the software that these firms supply to external partners, is not fully liquid.

The main issue here is that DevOps is still dealing with a lot of large application packages that are not liquid. They are continuously deployed, with updates coming in the form of transfers of a lot of data and replicated services. To achieve full liquidity, continuous update systems must be able to execute continuous updates of libraries, and this will require having the concept of libraries nested within end-user devices. The way updates are currently delivered – duplicating full applications and data – is a huge waste of storage and network resources. So, once again, it's IoT that will benefit most from this focus on library updating and it's IoT that will accelerate the liquid software revolution. From advancements in IoT, the mobile device marketplace will catch on, with significant effort concentrated on continuous updates for cloud-distributed apps. Successes on these fronts will then spread across the entirety of the software development spectrum.

Flight Risk?

"Passengers, this is your captain speaking. We've reached a cruising altitude of 30,000 feet and in just a few minutes, we'll commence a software update of this airliner's major flight systems." Upon hearing such an announcement, most people might pause momentarily while absorbing the information, and then reactions could range from mild disquiet to panic.

So, let's start with the obvious question: Why the heck would anyone want to do this with an airplane, en route, carrying several hundred souls? Surely the risks involved outweigh any potential benefit? Well, before we address these questions, it should be noted that it's quite common today for airline companies to execute software updates for non-critical systems while their planes are on the ground

(e.g., in-flight entertainment services, Wi-Fi, corded phones, and mobile device connectivity).

Let's return, though, to our "scary" scenario and consider a bit of context. The U.S. Federal Aviation Administration (FAA) has been working with the airline industry for several years toward the implementation of a collision detection system update. The concept is to use high-level GPS instead of radar systems to track the precise location of all planes in the sky. The coordinates provided by the GPS system would allow for significantly improved management of flights, enabling more planes to take off and land, particularly those that service crowded urban hubs. However, this kind of technology opens up the possibility that further down the line a malicious GPS spoofing hack could be discovered in GPS processing software. The upload of such bad data could penetrate an airplane's software systems, giving a bad actor control of the aircraft. An in-flight software update could avert a potentially catastrophic event by removing the vulnerable GPS processing software. After an event like that, proponents of never updating software when a plane is flying will have a hard time arguing their case.

If You Love Control, Set It Free

As we've established, with the ongoing and exponential rise in software-driven, software-managed, and software-monitored, well…everything, we have an increasing need for speed. But speed is not enough for the liquid software revolution to succeed. Current technologies allow us to rapidly accomplish a huge amount within the continuous updates arena, but we also need to establish rock solid reliability and trust in update pipelines and the data that flows back and forth between them. This is what substantially distinguishes continuous updates from continuous deployment. We typically speak of continuous deployment in terms of pushing deployments to data center and production

system runtimes that are strictly under local control. As such, within many systems, we have a high level of control over the runtimes in which the software will be executed. Liquid software greatly expands the horizon because we are dealing with runtimes that can be pushed to or pulled from as part of a continuous update environment. This means we must continuously deal with runtimes that are outside of local control. When it becomes the norm for software developers and firms to confidently let go of this local control, we'll know that the liquid software era has truly arrived.

Just Sign on the Dotted Pipeline

For all that we can accomplish right now in terms of continuous updates, there are still challenges ahead. For example, API security standardization remains an issue to be overcome. In an optimal liquid software environment, the signing authority for certain types of software certification would be automated. We would be able to establish that a specific version of a specific package has been tested and validated by ABC, and passed; integration tested by XYZ, and passed; security tested by another entity, and passed. The same would occur down the line for validations of End-User License Agreements (EULAs), release and customer relationship notes, and so forth. This is yet another aspect of the engineering of completely trustworthy, worry-free pipelines. Customers wanting to obtain the latest client library and correct routing could then do so with total confidence and with full knowledge that the liquid software they receive has been properly certified and signed by all appropriate entities. The customer could then implement an automated filtering system that filters software inflow, such that they'll be receiving only that which they want. To accomplish this goal, we must build an infrastructure that establishes not only liquid communications, but also trust, between companies. This could come about through the use

of certified signatures that are associated with specific types of update clusters.

Such solutions could even extend to self-driving cars, trucks, and other autonomous vehicles. However, as we might imagine, there is enormous resistance to the notion of executing continuous updates in this realm, and this resistance will likely persist for some time. While it is true in a general sense that such updates would be no different than any others, reluctance to liquefy the software governing these systems is based almost entirely on the potentially lethal results should anything go wrong.

Nevertheless, there are some current examples – Tesla is notable – of automatic updating of vehicles' self-driving features. The only difference is the amount of quality, or responsibility for quality, the vendor assumes. Some of this has to do with legal obligations and liability issues with respect to precisely who is responsible for certifying and provisioning an update. Since Tesla owners are not aware of liquid updates occurring, they cannot be held wholly responsible for a malfunction or an accident that is the result of an update. Software consumers don't have the knowledge, skills, or facilities to ascertain whether one version or another is sound and ready for deployment, nor can the owner execute appropriate and necessary rollbacks. All of that must be the sole responsibility of the vendor.

We want to state clearly that despite the fact that transportation and other high-risk industries may be averse to continuous updates, it is our belief that firms that don't embrace the liquid software revolution by implementing methodologies and systems to guarantee the quality, security, and provenance of their software, could be destined for quick demise.

We Want Information, Information, Information…

Metadata is the information that allows us to make sensible decisions about whether a piece of software and all of its component pieces are good or not. It might be metadata about the origin of a particular component, the history and features of a particular version, or validation steps that software went through – all of this and much more is critical information for a well-designed, automated pipeline. Metadata enables us to determine whether our software should or should not be promoted to the next level in a continuous integration pipeline. There's internal metadata related to our own projects, internal metadata related to other projects that we use as libraries, and external metadata

related to all the components that we use. Because of the number of sources from which it can derive, our metadata can indeed be quite meta!

Ours is an age of binaries. With every software industry advancement, we've seen an exponential rise in the production of binaries. There are billions, if not trillions, of them now in play, with unknown numbers yet to come. Consequently, we emphasize in several sections of this book how and why copious amounts of highly targeted metadata are critical to the rise of the machines and the eventual success of the liquid software revolution. It's only through the robust use of metadata that we can make any sense of this vast and growing sea of artifacts. And for each artifact, we must answer questions that fall into three essential categories:

1. **Basic information**: What is it? Where did it come from? What we can do with it?
2. **Location**: What do we have, where? We have a multitude of environments – QA, pre-production, production, etc. With artifacts in every environment, we need to understand why they are where they are.
3. **Quality**: What validation did it go through and what were the results? Were Common Vulnerabilities and Exposures (CVEs) or any other bugs revealed? Was it tested internally? How did its performance test results compare to other versions of the same binary? In a continuous update system, artifacts will be generated from several different environments.

We need answers to these questions so we can make smart decisions about what should progress through our liquid software pipes at the processing phase. We must decide what should be deployed where vis-à-vis the environment from which an artifact was created.

It will be possible to address these issues only if we have enough metadata. But just how much is enough? To answer that question, let's examine security concerns related to metadata, such as CVE identifiers. And let's pretend we have an ultimate, magical solution. With a wave of our digital wand, our machine learning algorithms and big data analytics scan our source code and determine whether there are any security breaches that others might be able to exploit. Is this enough? No. Why not? Because modern software development includes software that is composed from different open source components. To be certain our code is secure, we cannot simply analyze all of our own source code, we must have confidence that third-party components are secure as well.

This is true for every aspect of metadata. For example, we can test our code for performance. We can use benchmarking tools and application performance management (APM) systems to discover if a component we've just written is performing well enough to go to production. Is that enough? Once again, the answer is no. We need to know about the performance of *all* the components we use – those coming from other teams in our organizations, as well as those coming from third-party sources.

The same type of thinking applies when it comes to licenses or any other aspect of the architecture of the software we produce and distribute. And we need to be mindful of how quickly things can and will shift in the world of software development. For instance, at one moment in time, we might incorporate a particular company's third-party libraries as a login component. Just because that company's product is good today doesn't mean it will be good tomorrow. Perhaps a new CVE about a fatal security flaw will be discovered; perhaps the provider won't be able to keep up with the pace of innovation; or perhaps some competitor will come out with a product that's better. And maybe we'll

decide to switch to another provider, only to discover later that our original provider has made improvements sufficient for us to switch back. If we're reliant on third-party solutions, we need to be paying attention, and making decisions accordingly.

"Your Data Security is Important to Us"

At present, if we want to check whether an artifact is vulnerable, we can rely on a number of good informational resources. One that is particularly helpful is the National Vulnerability Database (NVD), a service of the Information Technology Laboratory (ITL), operated by the U.S. National Institute of Standards and Technology (NIST). However, as of yet, there is no single repository of information that can address all of the questions we might have about all software components.

Regardless of which database we might use, we have a more fundamental question: How is a given artifact to be identified in any particular database? If there are no standards, how can we know if an inquiry we submit has been conclusively answered? A mistake here can impact the data security of millions of people. This might be what happened in 2017 to the consumer credit reporting agency, Equifax. A data breach occurred that exposed the sensitive private information (names, dates of birth, social security numbers, etc.) of over 145 million U.S. consumers. It also resulted in over 200,000 credit card numbers being illegally accessed.

So, what happened?

Equifax had built a container to store all of the personal identifying information (PII) of its customers. The company used a third-party resource to do this – *Apache Struts 2* – a free, open source, and (as it turned out) quite vulnerable Java library. The firm wasn't necessarily wrong to use Struts 2. After all, it had been embraced by the software industry for

almost a decade. A significant majority of websites written in Java had been using Struts, making it a de facto standard during that period.

By early 2017, however, Struts 2 was no longer the best available web framework in the marketplace. It had become a legacy resource and was on its way out, not least because it had racked up a history of security vulnerabilities. According to the Common Vulnerability Scoring System (CVSS), since the debut of Apache Struts 2 in 2007, fourteen of its vulnerabilities had rated hair-on-fire scores of 9 or above (on a scale of 10). Apache released a patch for the fifteenth such vulnerability – which had achieved the dubious distinction of a "perfect" CVSS 10-score – in March 2017. Disastrously, Equifax wasn't paying attention. Two months went by and the company still hadn't applied the patch. By May 2017, Equifax was so vulnerable that the breach it suffered was the cyberattack equivalent of punching a fist through a decorative Japanese room divider. Data flowed out of its systems over the course of several weeks, ultimately costing the company and its insurers hundreds of millions of dollars. In the aftermath of the fiasco, firms that had still been using Struts 2 abandoned it in droves.

Universalizing Metadata

Much has been written and said about what happened at Equifax. But is it a given that a simple dose of due diligence will always be the perfect path to avoiding catastrophe? Let's say that right now we want to determine the current vulnerability status of Struts 2. We could visit the National Vulnerabilities Database. How should we search for it there? Should we enter its name as *Struts 2*, *struts2*, or *Apache Struts 2*? Should we enter its SHA1 checksum, it's GAV coordinates (org.apache.struts:struts2-core), or should we use some other type of identifier? What if we don't find anything on the NVD? How should we carry out a search in another database?

Is there any way to perform a crosscheck? Maybe it's registered with some databases and not others.

So we have several big problems. First, when we want to perform a search, there's no place for "one-stop shopping." Second, regardless of where we go, the data we're seeking might not be available. Third, if the data is available, there's no uniform methodology for querying the components we're using. But with so much metadata being generated from so many different sources, we might begin to think that this is all getting to be a bit too complicated. It is. That's why, as part of the liquid software revolution, it will be wise to standardize the methodologies through which metadata is generated, transferred, and read.

We understand that many in the software industry react to the idea of standardization with trepidation, because past attempts at developing standards have not been very successful. This is not because there are serious disagreements about the benefits of establishing standards. And it certainly isn't because the industry lacks the talent or inventiveness to create useful standards. It's that in too many instances, even where consensus has existed that standardization would be helpful, when a dozen different entities created the needed "standard," instead of simplifying the situation, things only became more complex.

The Solution: A Metadata Scribe

In late 2017, an open source initiative called *Grafeas* (the Greek word for "scribe") was launched, its objective being to "define a uniform way for auditing and governing the modern software supply chain." Grafeas would like to gather metadata about everything through the implementation of an industry-accepted common model for sharing metadata about software artifacts and releases. The concept envisions acquiring and pooling metadata from both internal and

external sources, in order to construct a more complete picture about components in circulation and use.

In practical application, NVD can expose this data in Grafeas format, which renders it universally understandable, particularly to machine-driven platforms and systems. Metadata can be tagged with a mutually agreed upon identifier, such as a checksum of an artifact, which can then be compared with all known checksums and all known vulnerabilities. Thereafter, Grafeas can provide a response in the form of an adjacent file that can flow through continuous update pipelines. A given file might indicate that we have encountered instances of particular vulnerabilities. In an automated system, this would trigger a block, which would prevent the problematic artifact from being further promoted in a production or distribution pipeline.

While this central database of vulnerabilities is the most obvious example of what Grafeas can do, the possibilities are enormous. For example, when handling metadata that's internal to a company, we will typically see separate divisions and different teams sharing metadata. But let's say that one particular firm is using a component that is completely internal, and the outside world doesn't know it exists. Under normal circumstances, this component would never appear in any external database. However, through the use of a Grafeas-compatible source code security analyzer, we might be able to detect a pattern in that component's code that could present a security vulnerability. Again, Grafeas can provide a response in the form of a metadata file that streams this information inside an organization. Then, whenever an automatic pipeline needs to vet artifacts for security breaches, it will receive a uniformly formatted document, with the same exact metadata from the outside database and the inside source. This will help it to ban artifacts, as appropriate, whether they're based on Struts 2 or an internal

component that's just been revealed to contain some suspicious looking source code.

The Grafeas initiative is encouraging a dramatic increase in the production and use of metadata. As we have said several times, metadata is the key to continuous update success. Metadata should be registered for every action we take in development and should be consulted in every decision that needs to be made in production and promotion. We should produce metadata at every phase of continuous integration, starting from the build server. We should log information regarding how a particular artifact was built, who initiated the build, what the environment variables were, which versions of system dependencies were used, how long the build took, and so forth. Following this, we want metadata regarding QA. For example, as we perform unit tests, we should register whether our artifacts have passed or not, and whether the tests triggered any concerns or warnings. The same should hold true for recording integration test metadata, as we will want information to verify stability or alert us about instabilities. We should gather and register metadata equally about every test we perform. Since we cannot be certain about the future and what information we might need – let alone information that can help us avoid a calamity – the gathering of copious amounts of metadata should become routine across the industry.

A World in Which Grafeas Data is Everywhere

Grafeas is unique in that it acknowledges a reality in today's software industry – the existence of complicated use cases, where software includes components from internal *and* external sources. Grafeas is designed to be able to mix and match the metadata arising from both. Depending on the nature of what we're building, we might integrate information from various components that we're using. In

such situations, external metadata can become part of our internal metadata. Conversely, we might have internal metadata published in in-house databases. If we begin to work on an open source library, we might want to disclose our internal metadata to public metadata sources, so others could benefit from this information.

With increasing amounts of metadata, we will create opportunities to check and recheck our components before they are deployed to runtime servers. One example of this is an initiative called *Kritis* (Greek for "judge"), a rule engine for *Kubernetes* that operates on Grafeas metadata. This tool will allow us to write rules to direct the execution of a final pre-deployment check. If the check encounters any security vulnerabilities, the rule will direct the system not to move forward to deployment.

The Kritis project is an acknowledgement of another reality of modern software, which is that components that have already been tested as stable and secure today aren't guaranteed to remain in that state tomorrow. We might have a component that's been in production for two years, and then suddenly discover there are vulnerabilities in every layer of our dependencies. Obviously, this would require us to take action on artifacts that are already in production. We want the software industry to evolve to the point where liquid software is automatically pinging Grafeas-enabled metadata databases at regular intervals (say, every 24 hours or less) to make sure that something we've verified at one point in time is still secure a bit further down the line.

The Grafeas metadata description initiative seeks to establish universality in the way we register queries about components, and to standardize the format of responses returned from those queries. It's important to note, however, that any tools using this format must be adjusted individually by the companies that produce them. Grafeas doesn't

maintain any centralized service, nor does the project intend to acquire and centralize existing databases.

If Grafeas is successful, it might well become an integral and indispensable part of the liquid software revolution. This would certainly be the case if, eventually, continuous update pipelines can issue standardized Grafeas requests to the NVD and other vulnerability databases, and receive standardized Grafeas responses that can be parsed automatically. Under such a scenario, decision-making regarding whether we want our artifacts to proceed in a given pipeline or not would become decentralized, and therefore much simpler and more secure.

ABOUT THE AUTHORS

Fred Simon

Fred is an avid software visionary with over 25 years of hands-on open source coding experience. He is a co-founder and the Chief Architect of JFrog, the DevOps accelerator company. He was also the founder of AlphaCSP – a Java experts consulting firm that was acquired in 2005. As a community influencer, Fred has been part of the most challenging changes in the software industry and has led Fortune 500 companies in their transition to DevOps. In 2015, Fred envisioned a world in which software is "liquid" and revealed the driving force behind the DevOps Revolution: Continuous Software Updates.

Yoav Landman

Yoav is a devout engineer, the creator of Artifactory, and a co-founder and Chief Technology Officer of JFrog. With over 20 years of experience as a Software Architect of enterprise applications, he plays a significant role in the evolution of DevOps. In 2006, Yoav created Artifactory as an open source project paving the way for the software community to a new domain of managing binaries. Prior to JFrog, Yoav created many production solutions as a consultant in the fields of Continuous Integration and Distributed Systems. He is also an accredited speaker and a Java Rockstar.

Baruch Sadogursky

Baruch is a vibrant and passionate advocate in the software development community. He is known as a champion in vocalizing key technical problems and offering inventive solutions in the high-tech industry. Baruch has been a software professional, consultant, architect and speaker for almost 20 years. He has been JFrog's Developer Advocate since 2012. Prior to JFrog, Baruch was an Innovation Expert at BMC Software and a consultant and software architect at AlphaCSP. Baruch is a Cloud Native Computing Foundation Ambassador, an Oracle Developer Champion, a Java Rockstar and a leading DevOps evangelist.