XPRT.

Magazine N°11/2021

The shift to cloud-native: Accelerate your innovation

Upgrading user interface for the future

GitHub Actions running them securely

Creating an open source learning project

The reliability paradox: Why less can be more

Xpirit

Together we drive change.

DEVOPS BOOTCAMP

GLOBAL, LOCAL & VIRTUAL

ACCELERATE DEVOPS ADOPTION WITH THIS EXCLUSIVE DEVOPS EXPERIENCE

HAVE YOU EVER WANTED TO EXPERIENCE WHAT IT'S LIKE TO WORK IN A TEAM THAT PRACTICES REAL DEVOPS? DO YOU WANT TO RUN A DEVOPS BOOTCAMP?

Then this is the event for you! You learn how to build software with immediate feedback loops and push it to production, multiple times a day, without hesitation. You will be able to translate everything into your daily practices and initiate your DevOps transformation based on experience instead of text-book examples.

DO YOU WANT TO RUN A DEVOPS BOOTCAMP? CONTACT MAX FOR ALL OPTIONS.

Max Verhorst / +31 (0)6 13 46 80 02 /
mverhorst@xpirit.com

Colophon

XPRT. Magazine N°11/2021

Editorial Office
Xpirit Netherlands BV

This magazine was made by
Alex de Groot, Alex Thissen, Anne Meijer, Arjan van Bekkum, Bas van de Sande, Casper Dijkstra, Chris van Sluijsveld, Dennis Thie, Erick Segaar, Diederik Tiemstra, Bastiaan Weijers, Duncan Roosma, Max Verhorst, Erik Oppedijk Loek Duys, Geert van der Cruijsen, Jasper Gilhuis, Hindrik Bruinsma, Immanuel Kranendonk, Rob Bos, Jesse Houwing, Jesse Swart, Kees Verhaar, Maarten Blok, Marc Bruins, Maira Duijst - Camu, Manuel Riezebosch, Marc Duiker, Marcel de Vries, Sofie Wisse, Martijn van der Sijde, Michiel van Oudheusden, Natascha Former, Niels Nijveldt, Pascal Naber, Reinier van Maanen, René van Osnabrugge, Roy Cornelissen, Sander Aernouts, Suraj Sewbalak, Thijs Limmen

Contact
Xpirit Netherlands BV
Laapersveld 27
1213 VB Hilversum
The Netherlands
Call +3135 538 19 21
mverhorst@xpirit.com
www.xpirit.com

Layout and Design
Studio OOM
www.studio-oom.nl

Translations
TechText

If you prefer the digital version of this magazine, please scan the qr-code.

In this issue of **XPRT.** Magazine, our experts share their knowledge about the shift to cloud-native: Accelerate your innovation.

The shift to cloud-native: Accelerate your innovation

The trend in the industry is clear. We need to accelerate our innovation to stay on top of the curve. Cloud has become the default and now the challenge lies in how can we transform our organization so we can leverage all that is cloud. We need to move away from traditional operating models where we have the IT department dictating how we need to use IT. IT has become the business! This also means IT departments need to move away from a siloed demand supply organization and re-invent themselves to get a seat at the table in the business. They need to enable accelerated business innovation instead of being a constraint in speed of delivery. They need to become a high performance IT organization and reimagine how to empower everyone so they can use IT in self-service while staying secure, compliant and efficient.

Author Marcel de Vries (Chief Technical Officer)

In this episode of our magazine we have various articles that can help you paint a new picture of the future. How do you empower developers with secure and performant desktops so they can deliver the software we need so desperately while not compromising security? How you can automate your CI/CD with GitHub actions and make sure the supply chain that produces the software is secure by default? How can you embrace opensource to learn new technologies and how can you bring software that is mature and robust, but still a monolith to a cloud native environment? What User interface technologies can you embrace to create flexible and maintainable user interfaces to your customers? How can we embrace the cloud and employ new techniques to ensure our software is reliable and robust, while the cloud has a completely different reliability model than your on premise datacenter?

With the cloud being part of virtually every business strategy we come across we decided it is also time to spawn new businesses where we focus on the delivery of cloud native software delivery and providing customers with a cloud native managed services proposition. In this magazine you will also learn how we are embarking on those new journeys with our team of experts.

At Xpirit we had the pleasure and privilege to have already walked a journey with cloud native software development and managed services for the past seven years. We learned so much along this journey that we never dreamed possible. With our magazine we share many things we have learned throughout our journey and we hope they will help you become more successful in yours.

For us it is clear that the shift to cloud-native has started. We love to be there with you side by side with our experts and help you accelerate your innovation! </>

"Cloud native technologies empower organizations to build and run scalable applications in modern, dynamic environments such as public, private, and hybrid clouds. Containers, service meshes, microservices, immutable infrastructure, and declarative APIs exemplify this approach."

– Cloud Native Computing Foundation

Upgrading user interfaces for the future

Kongsberg is a company in the maritime industry – it is heavily regulated and in general, it does not spend too much time on 'how things look' - as long as the solution is functional. In the past, large legacy desktop systems have been built for Kongsberg's maritime simulation and training division. These systems use WPF (or older!) to show and control its state.

Author Albert Brand

Recently Kongsberg started to deliver a cloud-based training platform for maritime students, in which a view on the ship's bridge with all instruments is accessible from a browser. Together with the transformation to a web platform there was a great opportunity to rethink how to compose the user interface with reusable elements, how these user interfaces are connected to the simulation services, and how to achieve a maintainable system that may be compiled into something entirely different in a couple of years.

This article will discuss some details of this transformation to the web. If you are interested in the 'cloud side', make sure to read the article by Roy Cornelissen and Sander Aernouts in this magazine.

Rethinking the design

Kongsberg brought in a design agency to create a fresh new look for their entire simulation product suite called K-Sim. This covers:

- the simulated ship controls called **instruments**;
- the virtual ship's bridge where these instruments are shown to the user called **PanoramaWeb**;
- the portal to start a simulation, see assessment results, and buy licenses for specific instruments called **Connect**.

Initially there was a focus on the instrument design. These were crafted as a replica of the physical world (which is called skeuomorphic in experts terms). However, after several iterations it became clear that in some cases, a real life design is hard to manipulate using a display or touch screen.

Also, creating components from these designs was deemed to be pretty complicated (although we managed to deliver some!).

After a number of iterations, the agency took these learnings and they made the distinction between **replicas**, **abstractions** and **digital screens**. When a physical replica is too constraining, abstractions are used to present a design that is recognizable but does not exist in real life. For example, the heading repeater instrument has traits of a compass rose that add a relation to its functionality. The third distinct design type is digital screens. Today, some instruments on a ship already use a touch screen instead of a custom hardware panel. It makes sense to give a similar representation to a student.

Creating composable UI elements

I joined the K-Sim Connect team in April 2020 as a Xebia frontend architect. One of the goals was to coach the current team in building modern web frontends. Of course they also wanted me to help build some of the user interfaces, fast! That seemed like a job that suited my skills pretty well.

Some teams already created web versions of instruments (before they hired a design agency). The instruments were built using vanilla JavaScript with CSS and did only use some

really low-level libraries such as jQuery to help render the output. The build quality of the components was lacking in several areas: minimal tests, no proper separation of concerns, no reusable parts and of course the visual design was pretty old-school as well.

Together with one of the simulation software architects of Kongsberg I discussed several topics:
- we should create small components to compose larger ones;
- the components should use a modern web standard to expose and isolate itself, and allow for data ingestion and event publishing;
- we shouldn't build everything ourselves but use the best libraries out there to achieve it.

The architect was also thinking about a domain-specific language that expresses how the user interface is laid out in a platform-independent manner. He liked what he heard about *Web Components*[1] as it is the official set of web standards for creating components that encapsulate their presentation and behavior.

So we went forward and started to create a Web Component library based on the initial designs, with many composable elements such as buttons, areas and text elements. But what does composability mean in the context of a user interface? To give an example, let's say that you want to show a big button with a flashing text on it. One way of building such a component is by creating a new one from scratch with exactly that behavior. However, such a solution does not scale: you're probably copy-pasting parts of a similar button, and you need to repeat that process over and over again for new variants of the button. An improved way would be to add parameters to an existing button, such as "size" and "flashing". However, that would still not scale very well, as your component would keep on growing with all kinds of variations which get harder and harder to reason about, let alone write tests for all permutations.

A better way to solve this is by creating an extensible component, which allows for injecting other components that only bother about their own concerns. For instance, the flashing button could be created by the following structure:

```
<StyledButton>
  <Flashing colors="[red,white]">
    <SimpleText size="big">
      Emergency!
    </SimpleText>
  <Flashing>
<StyledButton>
```

And this is exactly what you can do with web components. It offers you custom elements that provide a 'slot' mechanism to pass in other elements, making your components composable from smaller parts.

Libraries? Yes please.

While implementing the first components it became clear quickly that the Web Component standard is a little bare-boned. This is actually often the case for web standards in general: the standard committee is pressed to agree on a generic solution, and they often choose low-level APIs. It is up to the web community to pick them up and use them as a foundation for modern libraries.

Many of the existing frameworks such as React, Vue.js and Angular offer a way to perform a special build that wraps components as custom elements. However, this comes at the cost of having to ship relatively large libraries, just to draw a single component. So we looked at alternative frameworks and libraries to create web components while adopting a modern approach, but without too much extra overhead.

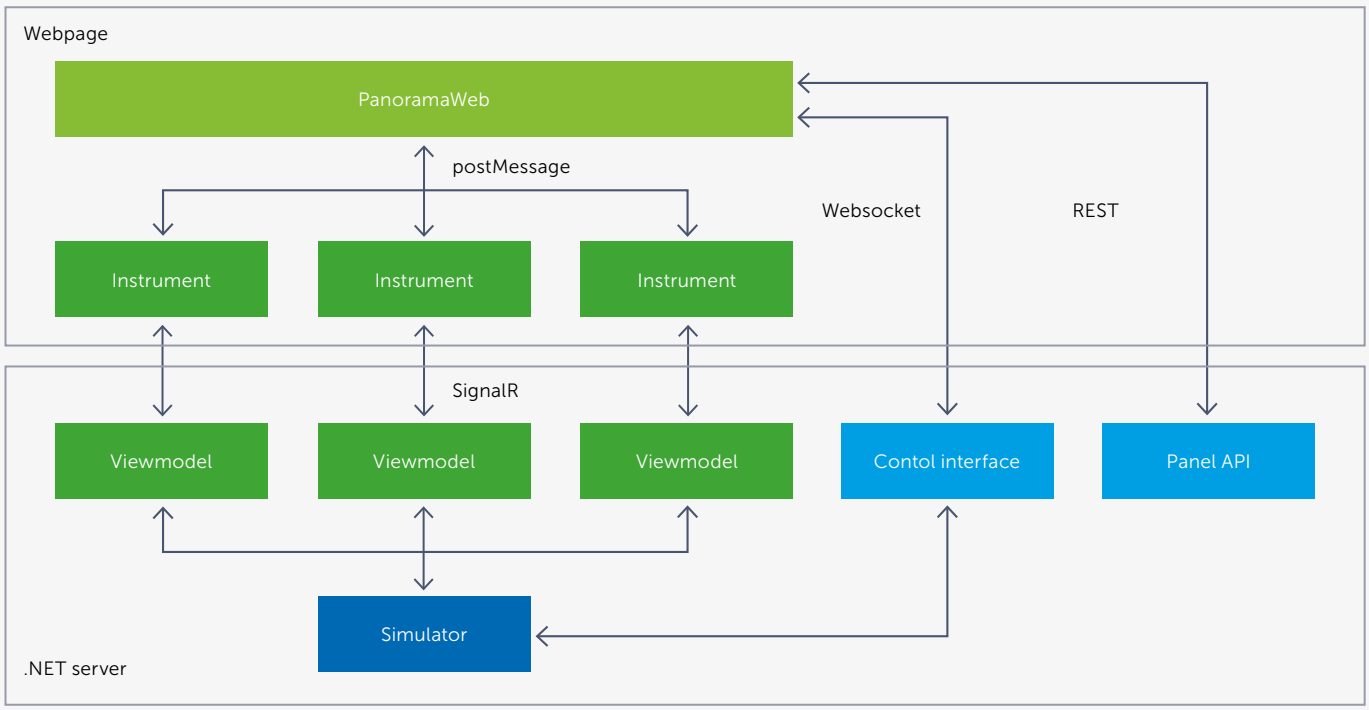
The choice quickly became clear: we wanted to follow the recommendations from *Open Web Components*[2], a collective of web components enthusiasts. These recommendations provide a powerful and battle-tested setup for creating and sharing web components. It recommends the *LitElement*[3] library for building web components, the successor of the Polymer project, which pushed the Web Component standard initially.

Presenting the ship's bridge in a browser

While building the shared component library, work was underway to build a new version of the PanoramaWeb web application to show the overview of instruments to the user in a modern way. As PanoramaWeb is a single page app that shows the 'chrome' around instruments, it was not necessary to build this as a web component. Instead, I opted to use Vue.js, as it an easy to pick up framework for building large component-oriented user interfaces.

PanoramaWeb initially retrieves the instruments it needs to show via a panel API. When the instruments are loaded, the app has some high-level control over the simulator. It can start and stop the loaded exercise and show the simulated time, which is presented in the top bar. This communication is done over a bidirectional stream of events that is exposed via a Websocket connection. In addition, each instrument connects to its own server-side view model instance using SignalR. And if that is not sufficient, each instrument can communicate with whatever service it wants, and with any protocol that is required for it. You can read about how the radar instrument uses a WebRTC stream for bringing the radar display to life in the article by Roy and Sander in this magazine.

[1] https://developer.mozilla.org/en-US/docs/Web/Web_Components
[2] https://open-wc.org
[3] https://lit-element.polymer-project.org



One of the challenges was to build a view to show and interact with the various instruments, while also being able to resize them and reorder them using drag and drop. As the instruments are built as separate web pages, it made most sense to use plain iframes to show the contents. iframes have a long history, as they have been one of the first browser features. They allow you to load and show two or more different pages of content in a single view, which means that they are a good candidate to 'stitch' multiple instruments together in a unified view.

Of course, there are other ways of combining multiple elements on a single page. You can choose to create large components that are then loaded on a single page. You could even use custom elements as a boundary for communicating between components. However, you need to make sure that these components have separate styles and dependencies, otherwise one component could influence another component in unexpected ways. And given the output of the in-house tool (which you'll read about shortly), I opted to go for iframes.

It took some sweat and tears, but PanoramaWeb started to shape up nicely after some time.

Dragging and dropping iframes that are holding those instruments did become a hassle at some point. iframes are quite limited; partly because of security concerns (you can load a page from a different domain so a browser needs to be very careful in sharing information between both), partly due to standardization reasons (it's just an element that shows a page in another page and that's it). And for some historical reason, if you move an iframe element to another parent element (which I implemented as a naïve first approach), the iframe contents will be reloaded. Even though this was not really a functional problem (the page is initially synced with its server view model), I really wanted to fix this bad user experience issue.

After investigating it became clear that if you want to ensure that iframes don't reload when being dropped in a different place, you should not move them at all in the DOM. Instead, I went for another strategy: when an instrument is visually dropped at a certain position, an InstrumentPlaceholder component is drawn. This component constantly determines its visual size and position on the screen (using the modern ResizeObserver and MutationObserver web APIs) and updates the internal state of PanoramaWeb. Thanks to Vue's built-in reactivity, it was a breeze to let the component that holds the actual iframe to pick up this change and position itself on the placeholder location. This allows for iframes to be placed anywhere in the component tree. Nice!

```
▼ <App>
  ▼ <AppInit>
    ▼ <TabViews>  fragment
        ▶ <InstrumentPanel>
      ▼ <GridContainer>  fragment
          <GridInit>  fragment
        ▼ <PageView>  fragment
          ▶ <TopBar>
          ▼ <Grid>
            ▼ <AspectRatioContainer>
                <InstrumentPlaceholder key='1'>
                <InstrumentPlaceholder key='2'>
                <InstrumentPlaceholder key='3'>
                <InstrumentPlaceholder key='4'>
                <InstrumentPlaceholder key='5'>
                <InstrumentPlaceholder key='6'>
              <SidePanelContainer>
            ▼ <InstrumentPanels>  fragment
              ▼ <InstrumentPanel>
                ▼ <Drag>
                    <InstrumentIframe>
              ▶ <InstrumentPanel>
              ▶ <InstrumentPanel>
              ▶ <InstrumentPanel>
              ▶ <InstrumentPanel>
              ▶ <InstrumentPanel>
              ▶ <InstrumentPanel>
      ▶ <TabView>
```

The tool that ties everything together

While I was having a go at the PanoramaWeb application, the software architect was happily working on a tool that soon would become the official Kongsberg-endorsed way of creating user interfaces for instruments. Mind you, Kongsberg already created hundreds of different simulated instruments, and maintainability is a big concern. Many of these instruments differ widely in style, technology stacks, architecture, layers, initialization and communication. Only giving developers guidelines on how to build user interfaces was not enough to streamline and standardize this process.

A domain-specific language called 'Blueprint' was designed and it allows you to specify how your user interface is built up using components, binding them to certain inputs from the view model (even with complex expressions), and listen to output of these components. The tool, which is written in .NET Core, can load libraries of components and compile a Blueprint file to an actual web page (including CSS and JS dependencies) that is ready to be served as part of the extension for the web server application.

In theory this tool could be used to output something completely different: a native desktop user interface, or a virtual or augmented reality variant. The possibilities are pretty much endless, however that is a chapter that still needs to be written.

We proposed numerous enhancements such as file includes with parameterization that found its way into the tool. At some point I even created a Visual Studio Code extension to syntax highlight the Blueprint file contents. My fellow teammates who wrote a lot of Blueprint code were very happy with that, as code readability is improved a lot this way. And of course, you get pretty bored looking at grey code all day long…

autopilot.blueprint

```
angle $(Heading)
towards-angle $(HeadingOrder)
allow-drag $(InstrumentPower) and $(InCommand)

# heading
  group
    offset 0, -20

   label-text
     text 'HEADING'
     font-size $(FontSize)

   group
     offset 7.5, 4

     readout-text
       text $(HeadingAsString)
       horizontal-align 'right'
       font-size $(FontSizeXL)
       status 'highlight'

     readout-text
       offset 1, -2
       text 'º'
       font-size 3.5
       status 'highlight'

# heading command
  group
    offset 0, 17.5

    include "autopilot-field.blueprint-part"
      $(Disabled) = not $(InCommand)
      $(Label) = 'HEADING COMMAND'
      $(FieldOffsetX) = -1.5
      $(Flashing) = $(HeadingOrderFlashing) and $(BlinkSync)
      $(EditableText) = $(HeadingOrderReadout)
      $(EnterPushed) = $(EnterHeadingOrder)

    label-text
      offset 0.5, 2.5
       text 'º'
       font-size 2.5

# mode selectors
  group
    offset -37, -17.5
```

In conclusion

We Xebians have been trained to aim for the sky and see problems as opportunities, not as roadblocks. However, other developers might not have that mindset. Learning a new library such as LitElement or a tool as Blueprint takes time, and you need to constantly remind yourself to take a step back, keep explaining when something is unclear, and in the end let others learn by doing, and stop 'holding their hand'.

Luckily, the approach that we kickstarted is being picked up, and more and more teams are now investing in learning and embracing that modern stack. There will always be growing pains, but teams are pretty happy so far.

So there you have it, a 'blueprint' of the future of Kongsberg user interfaces. I honestly believe that thanks to the chosen modern standards such as Web Components and the effort that is going into the Blueprint tool, Kongsberg does not have to invest in rebuilding their user interfaces every two years.

And the future looks bright as well. The adoption of the cloud e-learning environment is rising and demand for more teaching scenarios is clearly visible. Who knows which products will see the light of day and set a high bar for what you can do with an 'ordinary' browser and the cloud? </>

Albert Brand
Core Development lead from
Xebia Software Development

Never waste a good crisis

How COVID-19 drove innovation in maritime education

Rapid advances in new technology are changing the way seafarers learn. Advanced simulation is known to be one of the most effective applied training tools. They have been used in the training and education of seafarers for many years but often limited due to relatively high acquisition and operation costs. Democratization of simulation training is now happening, which will allow many more students to get access to high-quality simulation tools at an affordable price. Cloud technology and the increasing internet availability across the globe enable this transformation. The ongoing COVID-19 pandemic further accelerates it. We can expect improved quality of education, but this also could prove to be an essential tool in the digitalization transformation that the maritime industry faces.

Authors Gullik Jensen (Product Director for Digital Services at Kongsberg), Roy Cornelissen (Consultant @ Xpirit, working with Kongsberg since 2017) and Sander Aernouts (Consultant @ Xpirit, working with Kongsberg since 2017)

For close to five decades, Kongsberg has been a provider of simulators. Anticipating the digital shift, Kongsberg embraced the advancing technology and, in collaboration with Xpirit, pioneered the first simulator service based on its acclaimed engine room simulator platform. This service was made publicly available in March 2020, months earlier than its planned release date, motivated by the closing of maritime academies in the wake of the spread of the COVID-19 virus. By the end of the year, we had delivered a staggering thirty thousand simulations sessions to students globally.

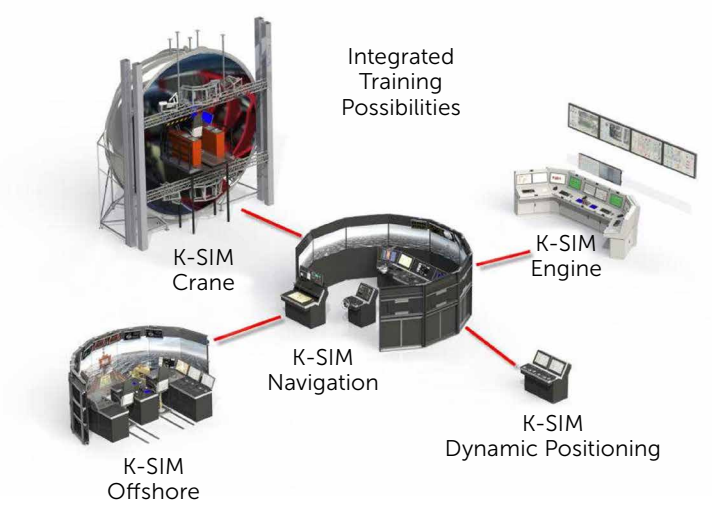
An ambitious plan

Motivated by this unconditional success, already in May 2020, Kongsberg accelerated its digitalization effort and started the cloudification of its navigation simulation platform, with the ambitious goal of offering the first public service, a RADAR simulation service, within the year. On the last day of November, we launched it. This story is about parts of the technology we created to deliver the first and probably the most advanced navigation simulator in the cloud.

We had learned a lot from bringing Kongsberg's Engine and Cargo simulator to the cloud, which we wrote about in our previous article in XPRT. Magazine #10. Could we also get their Navigation simulator to the cloud and have a working prototype in about eight weeks? Luckily, we could leverage all the work we had already done in the years before, but it wasn't a trivial task either!

First challenge: from (up to) 200 computers to 1 docker container

Kongsberg's simulator platform for navigation and offshore is called Spirit. It is a highly distributed system, with a simulator server at its core, simulating 'the world' and all hydrodynamics (motion of water and the forces acting on objects in the water). Spirit allows Kongsberg to build simulators ranging from a single desktop computer to full mission ship bridges consisting of hundreds of computers working together to drive instruments and provide real-time 3D visual imagery.



Source: https://www.kongsberg.com/digital/solutions/maritime-simulation/integrated-team-training/

Spirit has years of investment in its platform. It is entirely Windows-based, and most of its components have some form of GUI, even the server components. However, in a cloud environment there is not much use for a GUI. A cloud-native system should run headless on a server. Sure, you can install them on a VM and serve the UI over Remote Desktop, but that is an old-fashioned solution and a costly one.

We already had an entire platform and ecosystem for running simulators as containers in a Kubernetes cluster named K-Sim Connect. It handles everything for scheduling simulations, managing exercises and students in a SaaS offering. Spirit also had to land in this environment. We knew we were in for a challenge to containerize a system that wasn't designed with containerization in mind.

There are roughly two approaches for this:

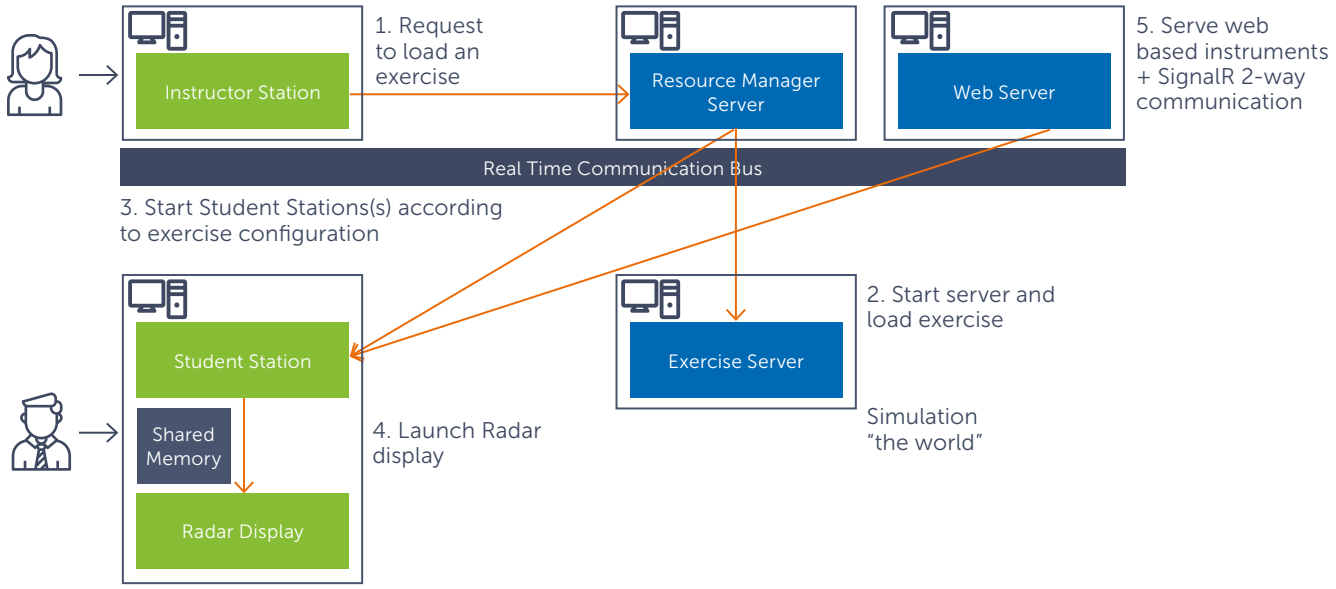
1. Rewrite from scratch as a headless system, using .NET Core/.NET 5 and Docker, and run it as Linux containers, or:
2. Adapt the existing system step by step and make it run in the cloud.

Many architects and developers would shout: "rewrite from scratch; this system is not cloud-native!". It would probably be the cheapest solution from an operational perspective in the long run, but it would have a very long time to market, tremendous development cost, not to mention disinvestment in an already successful and proven system. We had a very short time window to be successful, so while we were scaling up with our engine room simulators in production, we started working on bringing the Radar Navigation Trainer to the cloud.

Our existing platform had also proven that we could run Windows containers in the cloud just fine. Of course, Windows containers are big, and Windows nodes are more expensive to run, but it fully supports Windows-based software, including more "exotic" things like Win32 code and registry access. We knew we needed this for Spirit as well.

Our approach was somewhat trial-and-error at first because we needed to find out the obstacles we had to overcome. We took the Spirit installer and created a Docker file that installs it. Obstacle number one was that we needed to make the installer run headless. That was an easy fix in the InstallShield definition, by giving it a silent option.

Together with the Spirit architects, we looked at how we could make all the components involved in the simulation run headless.



This diagram depicts the critical components that participate in a Radar simulation. All server components (light blue boxes) had a GUI that displays their states and provides manual controls like stopping, starting and pausing. The first thing the Spirit team did for us was changing these components to run without any GUI in a Docker container.

The green components are full-blown GUI applications (mostly WPF). They all play an essential role in the system.

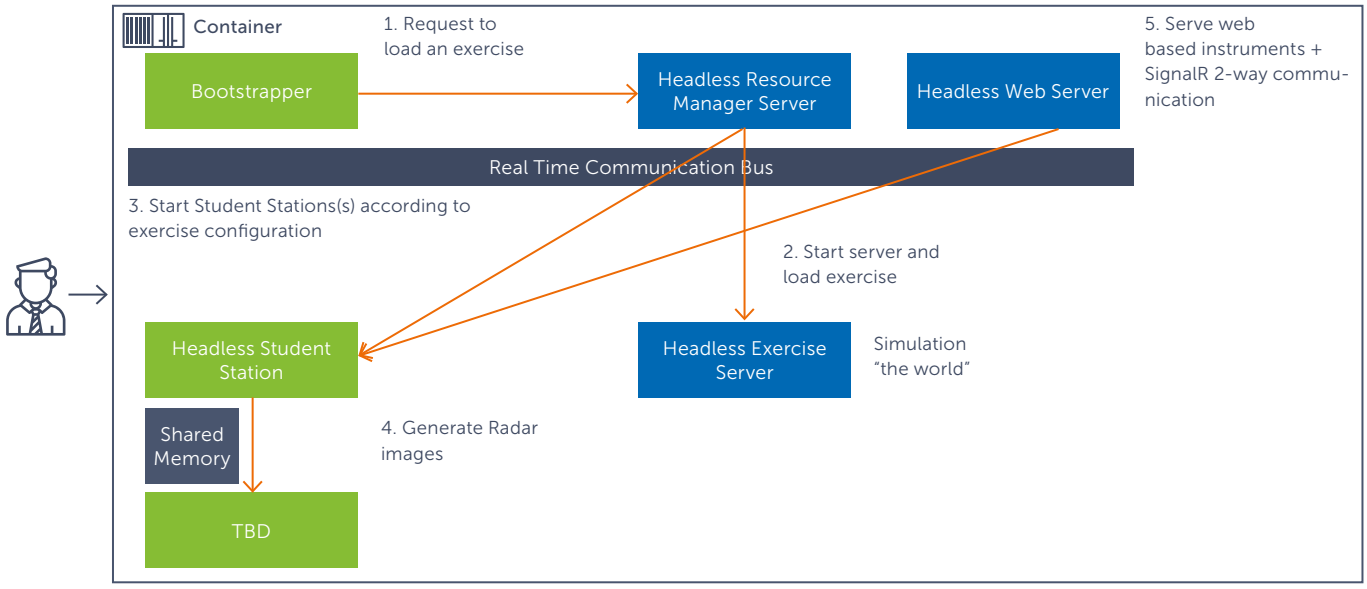
The Instructor Station is used to create exercises, add vessels, set up conditions like weather and the sailing area, assign students and start the exercise. Based on the configuration in the exercise and the simulator, the Resource Manager starts several other components.

We needed to automate the process of loading an exercise and starting the simulation without user interaction. Since we don't need all of the Instructor Station features in the cloud, only the ability to load an exercise, the Spirit team delivered a console application that did precisely that. This way, we could bootstrap the system via the command line.

The Student Station was trickier. It has the vital task of running the instruments that the students interact with. Instruments have a GUI but also hold logic to interact with the server components. The Radar instrument, in particular, has a part that generates sweeps based on the input data. A radar sweep is a full 360 degree turn of the radar beam, generating one picture. On each step in this turn, the radar generates a scan by shooting the beam in that direction.

This meant that we could reuse the existing components that generate radar sweeps. We just needed a new way to host them since they ran in the Student Station GUI application. Specifically, for the Docker container, we created a Headless Student Station. This is a .NET Console Application that loads the Spirit framework components that run the instrument logic but skips the presentation layer. One tricky part here was that the Student Station is a Windows application driven by the Windows message pump. Some components in the Student Station rely on having this message pump available. Also, the Radar's COM components require an STA thread (Single-Threaded Apartment) to run. We created a class that sets up an invisible Window that drives the message pump and sets up a Dispatcher that guarantees the Single-Threaded Apartment. It was a quick trick to make things work, but this is typically something you'd want to revisit later to make the application more container-friendly. However, it requires a more significant change in the architecture.

In the container, we launch this Headless Student Station instead of the regular one.



A separate executable called the Radar Display reads scans from shared memory and draws them on the screen. So, there were several things to address here:

- remove the GUI of the instruments while still running the logic;
- run Shared Memory in a docker container (could we do that?);
- replace the Radar Display application with something that could generate images without a GUI.

We had already learned that you could run quite a bit of "old" Windows mechanics in a Windows container (provided that you run a Windows Server Core image). COM, registry access, Win32, all of that works. We quickly verified that Shared Memory, which also is an old construct, worked as well.

You can run multiple processes in one container. This is what we do: all of the Spirit components run inside this single container, one container per student.

The Bootstrapper component that replaces the Instructor Station plays an important role here. It is the root process that determines the lifetime of the container. Furthermore, it communicates with the K-Sim Connect platform to track the status and progress of the simulation session. Recently, we added an automatic assessment of the student based on data from the simulator, which our web portal displays in real-time.

Second challenge: from WPF to a web-native UI

This brings us to the next elephant in the room: How to deal with the Student UI? Our first-generation Engine Room simulators still have a local GUI application. It works by virtue of a relatively simple client installation and a pure Client/

Server topology. We could bring the product to market fast, even though it's somewhat of a compromise to require a local client.

The Spirit platform is more complex, with its real-time communication bus. We knew that installing the Student Station on a client PC was not an option because of its large footprint, and we wanted to push the platform to be web-native anyway.

Over the past years, Kongsberg had invested in an extension framework for Spirit. An important driver for this was the ability to innovate on top of the platform without changing, testing, and releasing the entire platform itself every time. This multi-speed architecture of the extension framework significantly accelerated our efforts as well.

One of the tenets in the extension framework was that new instruments based on this framework would be served and rendered in a web UI. This is where the Spirit Web Server comes into play. It is an integral part of our solution. Normally these web components are hosted by the Student Station GUI application as individual panels with an embedded browser. The radar was going to be a web-based instrument as well, based on the extension framework. The Spirit Web Server would serve it, which we exposed in the Docker container, via a Kubernetes ingress. Each student gets his own (temporary) environment with a unique URL:
`<session id>.<cluster-region>.elearning.ksimconnect.com`

Kubernetes Ingress rules take care of the magic of routing traffic to the correct container.

The final piece of the puzzle was the replacement of the Student Station's "chrome", which handles the display and arrangement of the instrument panels. This application, named PanoramaWeb, was written specifically for our move to the web as a pure native web app, using Vue.js as its basis. Albert Brand's article in this magazine provides a more detailed background of the technology behind the web app. We will continue to extend PanoramaWeb and, as it matures, it will be the future Student Station.

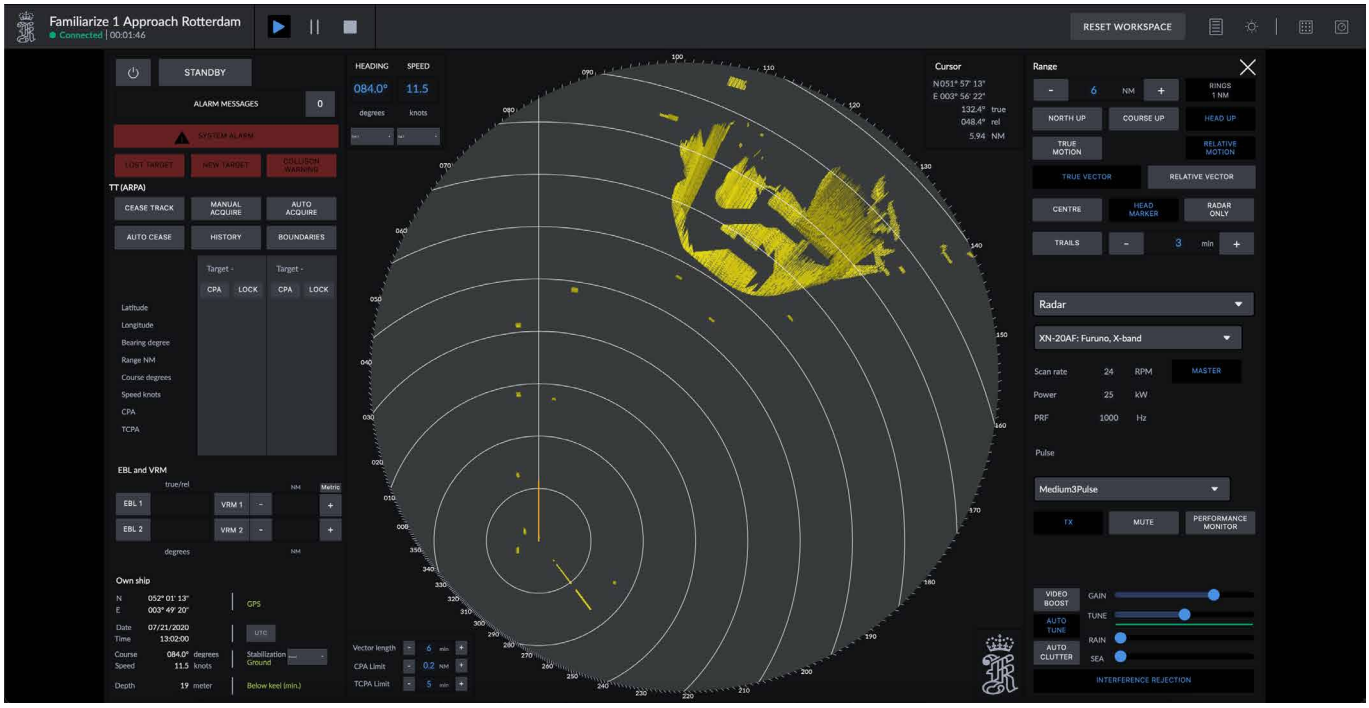
Now that we had a way to display instruments over the web, we could build the foundation of the Radar instrument. Buttons, status indicators and other user interaction like drawing range markers or bearing lines are all handled on the client side. The extension framework includes a SignalR connection with the server, which allows us to communicate state and updates between the browser and the container.

Replacing the radar display

An essential part of a radar instrument is the radar video, the well-known, often circular, view that displays the radar sweeps.

As the first diagram illustrates, the existing Radar Display is also a GUI application. It handles the drawing of the radar video, as well as all the user input. We had already dealt with the user input via the web panel. What was left was the radar video.

The data feed for the sweeps was already available in the shared memory block. The Radar Generator component constantly writes new values for each scan, much like a real radar would. We extracted the logic from the existing Radar Display GUI and created a new headless component to house that logic. It's called ScanConverter. Apart from PanoramaWeb, this is one of the few parts we rewrote for our cloud scenario. ScanConverter takes the data from Shared Memory and produces an image. We do this roughly 25 times per second, which is an acceptable frame rate.

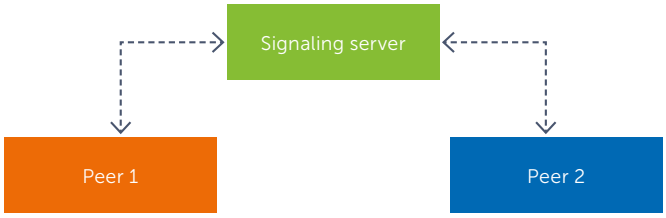


Third challenge: near-real-time communication on the web
Next, we needed a way to send these radar video frames to the browser.

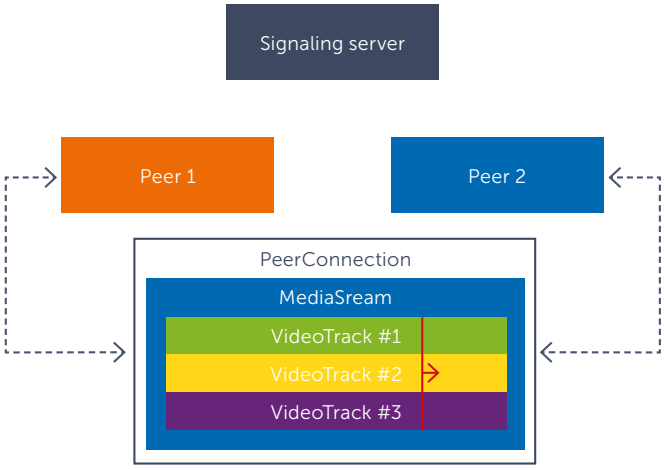
We started by looking at how streaming services such as YouTube or Netflix solved streaming video to clients at an incredible scale. But there is an important difference between streaming content such as videos and streaming a live radar video feed. When dealing with videos, users need to see them from start to finish without skipping parts of the video. Even when live streaming on YouTube, for example, the view does not have to be near real-time. For us it is more important that the user sees what is happening *right now* on the radar than that the user views the video from start to finish.

We looked at streaming technologies such as Dynamic Adaptive Streaming over HTTP (DASH or MPEG-DASH) or HTTP Live Streaming (HLS). Still, each of those prioritizes delivering a smooth (live) stream to a large number of users over providing a video stream as close to real-time as possible to a single user. We then looked at a different type of video streaming, focused on near real-time video conferencing: WebRTC. WebRTC is a protocol for real-time voice and video communication on the web. It focuses on peer-to-peer communication, and an important feature is that it is natively supported by browsers these days.

When using WebRTC, we need a so-called *signaling server*. The clients use this central server to discover each other when initially setting up the WebRTC connection. After the initial bootstrapping, the signaling server is no longer needed, and the WebRTC clients communicate directly with each other peer-to-peer. WebRTC has several mechanisms to enable such direct communication across different networks separated by the internet. When this fails, clients can use a TURN (Traversal Using Relays around NAT) relay as a fallback. With a TURN relay, clients no longer communicate peer-to-peer, but they use this central relay to communicate. The TURN relay is an essential component for us because we often need it in restricted environments such as corporate or school networks. These types of networks typically don't allow any of the mechanism that WebRTC uses to set up a peer-to-peer connection.



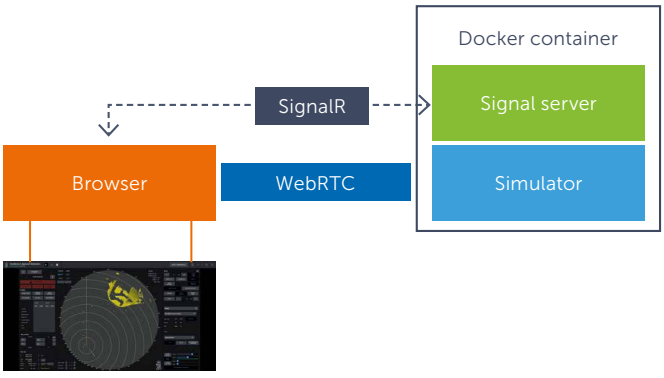
The peer connection in a WebRTC session can contain multiple video and audio streams that are synchronized. This is important in video conferencing because when you see people talk, you want to hear the sound that matches the movement of that person's mouth. In a simulation, we also want to synchronize multiple video streams such as a radar video, a 3D view and audio streams to make sure what the user sees and hears matches the current state of the simulated world.



WebRTC seemed to fit our needs perfectly, but our main challenge was to set up a WebRTC session between a Docker container and a web browser. While WebRTC is natively supported in browsers, using it on the side of the server in a C# .NET application was more complicated. We had to implement support for WebRTC into our C# application, just like the browser vendors did for their browsers. The source code for the WebRTC libraries is made public by Google, but its native C and C++ need to be integrated into your own application. After some digging, we found that Microsoft already had a project on GitHub that was aimed at supporting WebRTC in the HoloLens applications, and the library produced by this project allowed us to integrate WebRTC into our C# application with relative ease.

Since each simulator container is a self-contained application with its unique endpoint, we opted to put both the signaling server and the server-side peer in the same process in the Docker container: the Spirit Web Server.

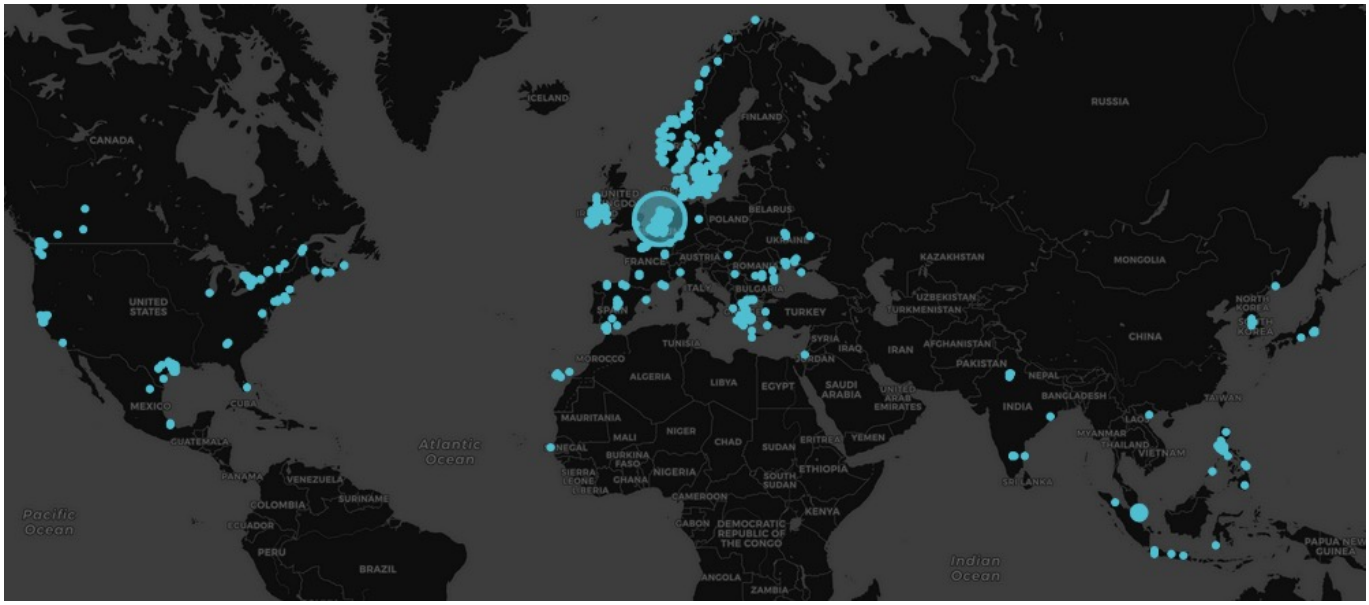
The browser connects to a SignalR hub (the signaling server) that is exposed on the Docker container and exchanges the required messages to set up a WebRTC connection with the simulator, running in the same container. Once the connection is established, the simulator starts streaming the radar video, generated by the ScanConverter component, over WebRTC to the browser.



The world after COVID-19

2020 started quiet for us, and we expected to steadily grow our customer base and start working on bringing the next simulator to the cloud. COVID-19 fast-tracked our plans and ambitions. Our product owner asked us "whether we were up for a challenge," and the team boldly accepted. And now, one year later, we have clusters running in multiple regions, with users across the globe using our simulators in the cloud.

market fast with targeted changes, but we realize we still have work to do. But by just doing it, we have learned a lot more about where and how to focus our efforts to optimize the Spirit platform for the cloud than starting with a complete redesign. Behind the scenes, teams within Kongsberg are now working hard on making the simulator leaner and more container-friendly. New features are already being developed "cloud-first".



But we are far from done, we merely started to unlock the navigation simulator's potential in the cloud, and we are already looking ahead to bring more and more features besides radar to the cloud.

As with the Engine Room simulator, this project shows that you don't need a complete rewrite of your system to capitalize on it in the cloud. We were able to bring it to

The launch of the RADAR service is one more important step in the democratization of maritime simulation. One hurdle at the time, we are shaping the future of maritime simulation and doing it to the benefit of the user and for the benefit of a safer and greener world. And on the way there we create some epic shit technology. </>

Roy Cornelissen
Distributed architecture, mobile development, creative

xpirit.com/roy

Sander Aernouts
Microsoft application lifecycle management (ALM)

xpirit.com/sander

Gullik Anthon Jensen
Lead digital transformation Maritime Simulation, Kongsberg Digital

Be Secure and Compliant with GitHub

How do we ensure security after we have deployed our application? This question comes up in many customer engagements. How do we make something secure and how can we ensure we are compliant? Unfortunately, many of these questions arise after the fact. After the application has been built, or even after it has been deployed, and this is exactly what makes it hard. Our answer to these questions is, you do not. You don't do this afterwards; you are secure and compliant by default.

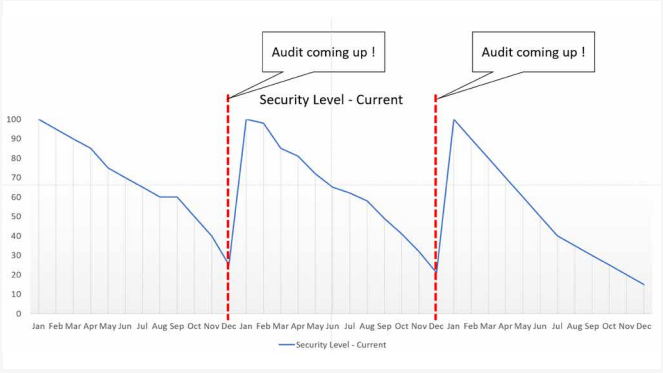
Authors René van Osnabrugge, Michiel van Oudheusden, Jesse Houwing and Arjan van Bekkum

Secure and Compliant by default

Nowadays, security is often implemented with a mindset of preventing breach. Make sure your perimeter is safe and prevent bad things from happening. This is often accompanied by a control framework of choice that targets three important areas - Confidentiality, Availability, and Integrity.

In many cases we receive an Excel list with hundreds of rules we need to implement to make our application "secure".

Following these rules makes us compliant but not necessarily secure, and in practice we can visualize the security score like in the graph below.



In this new world, where cyber threats are the new normal, you and your organization should assume that your software is or will be under attack, and people are going to use your software in ways you cannot anticipate. This is where "Rugged DevOps" or "SecDevOps" comes in. To be "rugged" means that you can deal with this unanticipated use and sudden attacks, that your software and infrastructure is resilient against abuse, that it does not contain vulnerabilities and that it is secure by design. Furthermore, your software as well your processes should be in such a state that you can deal with frequent changes. After all, it makes all your effort rather useless when your application becomes insecure after five releases because you have no time to maintain the periphery.

And that's why we should consider security in every phase of our development lifecycle and shift security as far to the left as possible.

Defining a secure and compliant delivery process

With the move to DevOps and Continuous Delivery, where deployments happen multiple times per day, it is even more important to be in control of the process. When the "security department" is outnumbered by the number of product teams and engineers, they have their hands tied. Without automation and the integration of security into the daily work of Engineers (Developers, IT-Operation, Test Engineers, etc.), this department can only do compliance checking. And as Gene Kim mentions in the DevOps Handbook: "Compliance checking is the opposite of security engineering" (source: The DevOps Handbook – Gene Kim – page 313).

In terms of compliance, it all boils down to being able to show that the code that has been produced is traceable (audit trail), reviewed (4-eyes) and that the artifact which has been published to production is unchanged from the code it originated from (integrity). But does all of this make the code secure? Probably it will, but certainly not all aspects are covered. If we keep in mind that we want to write and deploy secure software, we should enable teams to do just that.
We should make sure that code:
› is reviewed
› scanned for known vulnerabilities
› doesn't expose your passwords or keys
› checked against common errors
› uses approved standard libraries
› and is well tested.

Our process should:
› produce immutable artifacts
› test the application
› monitor for anomalies.

All of this is needed to develop secure and reliable software.

By focusing on security within the development and deployment process, the need for information shifts from the auditor to the teams themselves. To debug a problem in production, sufficient logging is needed. To ensure the same version is deployed to test and production, scripts need to be in place and, in order to get a notification when a problem occurs, sufficient monitoring needs to be implemented. When the need is within the team itself, security and Non Functional Requirements (NFR) become a different priority, and the result is that the teams become compliant automatically. By implementing the security and the necessary countermeasures, the required controls to be compliant will be fulfilled automatically. And the best part? It is verified continuously by an automated pipeline and evidence can be retrieved from the system at any time.

If we shift our focus from building software and making it "secure" to building "secure" software in a "secure" way, we create secure systems. And when we create secure systems, we can test and validate in each step of our process, and they are compliant systems as well. If you are secure, it is most likely that you are compliant as well.

It is vital to enable teams to integrate security into their processes and pipelines. This means at every stage of the so-called Application Lifecycle, which consists of the following phases:
› **Requirements**
How do you collect requirements? How do you make sure the requirements cover the security requirements and the Non-Functional Requirements (like availability, backups, privacy, etc.)?
› **Local Development**
What can engineers do within their local environment to develop, build, test and run more secure code?
› **Source Control**
Once code leaves the local machine and is checked in to Source Control, what can we do to make this more secure?
› **Build**
When building code that comes from a Source Control Repository, what do we need to check and validate, in the code as well as in the produced artifact? Furthermore, what can we do to ensure that the pipeline itself is secure?
› **Release**
When the artifact is released to a Non-Production Environment, what can we do in terms of security - of the artifact (integrity, are we sure it is the same code as in source control), the pipeline and the target environment?
› **Monitor**
What can we do to ensure that the infrastructure and application that has been deployed stays healthy, and how can we detect, respond, and recover from any unforeseen circumstance?

In the rest of this article, we will explore a number of GitHub features that can help us to take some steps into secure software development.

Moving your code to production

When we want to ship a new feature to production using GitHub, we can divide our attention to the following 5 phases:
› coding phase
› storing phase
› build phase
› deploy phase
› release phase

In the following paragraphs we will walk through each of these phases, explaining the various practices you can use, and we will show how GitHub can help you to implement some of these steps.

Coding phase

In the coding phase, code is being developed. In most cases this happens locally on the developer's machine. This is arguably the most important phase, because this is where security is ultimately put into the code. There are a number of techniques and tools that support the creation of secure code.

Static Code Analysis

Static Code Analysis analyzes the code base without running it. Some tools scan for textual patterns, more advanced tools parse the code and sometimes even build a model to analyze how data flows through your application.

Static Analysis tools then apply rules to detect issues in the code. Static Analysis can detect a multitude of known bad coding practices and often suggests more secure alternatives. Most Static Analysis tools are general purpose, but there's also a number of security specific analyzers.

In general, when these issues can be detected while the code is being written, the issue can be corrected immediately, and the developer is immediately confronted with an opportunity to learn.

Credential and secret scanning

While locally testing the application, a developer may need to connect to external systems, decrypt data, or store the credentials for its service account. The encryption keys - credentials and API keys - need to be stored securely, but they regularly end up in source or configuration files.

When such files leave the developer's workstation, they may fall into the hands of others, and they may be able to leverage these credentials to hack into your infrastructure.

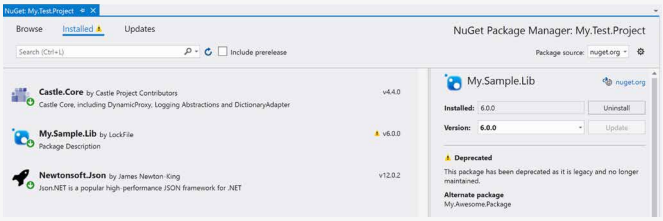
To prevent this from happening, a special breed of static analysis tool can analyze your local repository to prevent you from accidentally sharing your secrets to the world.

Curated dependencies

In today's modern applications we import more code from other developers and organizations than we write ourselves. We rely heavily on artifact repositories such as NPM, NuGet and Ruby Gems. Recent security research has shown that these public repositories offer interesting new ways to trick your teams from running code they didn't expect to run. Each time a new dependency is pulled in, it should be vetted to ensure it's secure and you don't want your build system to accidentally pull in new, unexpected dependencies.

› https://azure.microsoft.com/en-us/resources/3-ways-to-mitigate-risk-using-private-package-feeds/?WT.mc_id=DOP-MVP-5001511
› https://jessehouwing.net/99-percent-of-code-isnt-yours/
› https://medium.com/@alex.birsan/dependency-confusion-4a5d60fec610

Tools like npm audit and snyk will allow you to verify that a dependency has no known security vulnerabilities. Visual Studio has started highlighting problematic packages in recent updates:

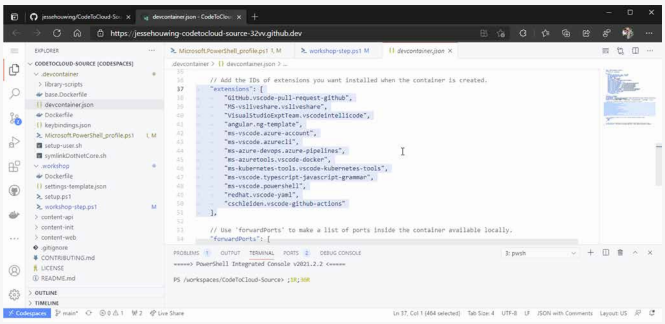


GitHub Codespaces

Setting up all these tools can be time-consuming and it's easy to make mistakes. It also creates a high barrier for people outside of your team to contribute to your projects, whether it is open source or inner source. It is even a high entry barrier for new people joining your team.

Advances in Visual Studio and Visual Studio Code now allow you to build easily extensible standard configurations for your development environments[1]. Visual Studio has basically been broken up into the backend, which manages, analyzes, and compiles your code and the front-end, which handles the user interactions.

GitHub CodeSpaces leverages this technology to run a full IDE from your code repository. Because GitHub CodeSpaces runs on a cloud VM outside your internal environment, it lowers the security risks. Anyone who needs to contribute to the repo is instantly transported into a ready-made environment that has all the aforementioned tools installed and configured to help them make their contribution secure.

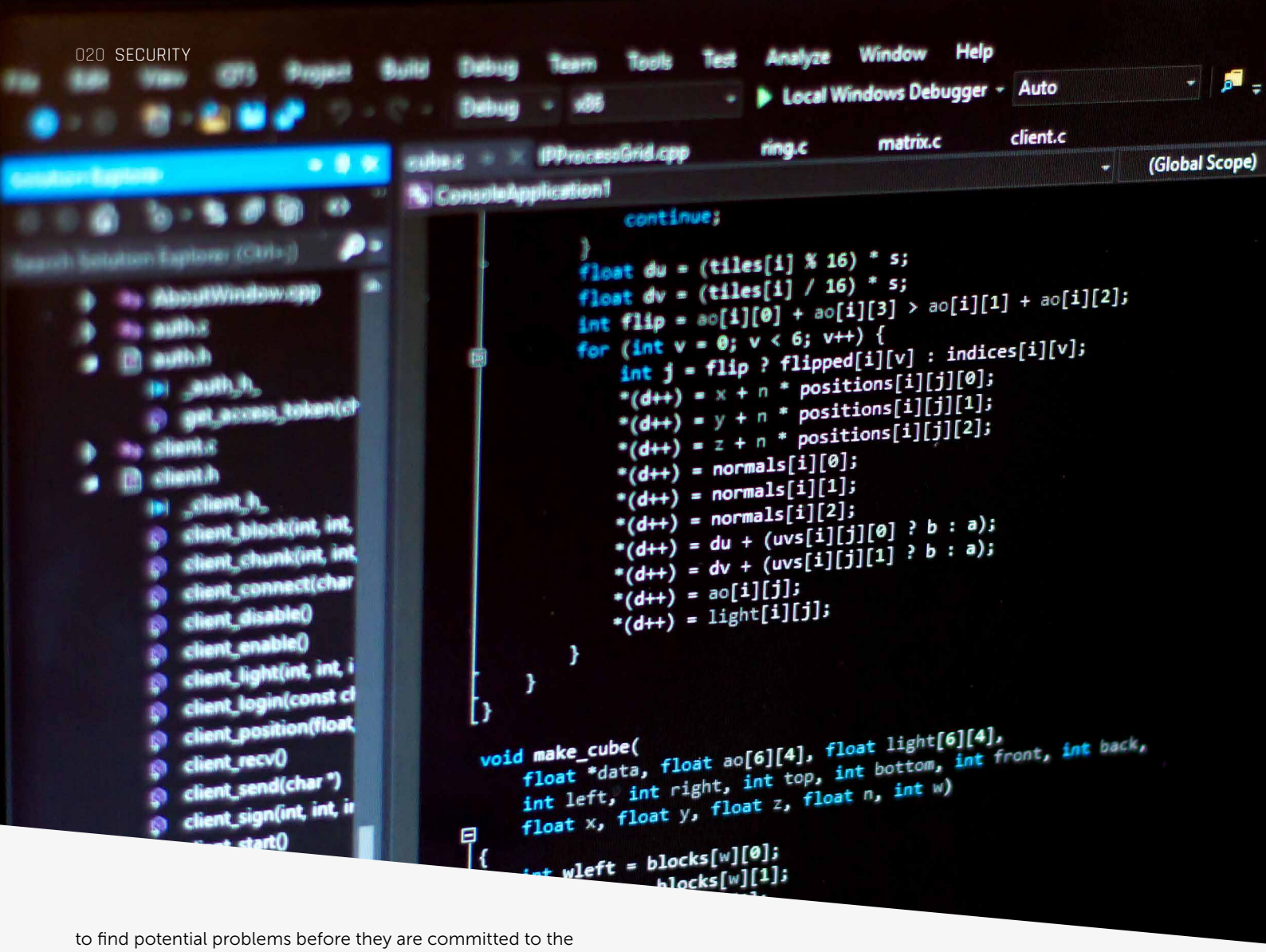


Because CodeSpaces runs Visual Studio on a remote container, you can even work from an iPad connected to a much more powerful remote container, only paying for the actual usage while the IDE is open. This even allows a casual contributor to propose changes while helping them do it the right way.

Visual Studio Live Share

With many developers being forced to work remotely, it has become a lot harder to just scoot over to your coder-buddy at the desk next to you to ask for quick feedback, pair or help you debug. Regularly reviewing your code with another person is one of the quickest ways to grow your own understanding and

[1] https://code.visualstudio.com/docs/remote/containers



to find potential problems before they are committed to the shared repository.

In the past we often used screen sharing and remote control to collaborate, but using this has its disadvantages. Especially when it comes to security and you give the other person full control over your system by giving them remote control.

Visual Studio Live Share can be compared to Google Docs for your code. It allows you to work in the same local repository with multiple people at the same time, even with multiple cursors simultaneously changing the same code file.

All participants can see the list of detected issues in the code as well as the status of all unit tests, and you can even debug code together. With Live Share you can essentially collaborate and review remotely without having to commit the code and pushing it to a remote repository.

It can even register who collaborated on the code when you decide to commit. By enabling the **liveshare.populate-GitCoAuthors**, the Source Control tab in VS Code will automatically generate the "Co-authored-by" trailer in the commit message, so hosts can attribute the collaborators they worked with during a pair programming session.

Who you can collaborate with and what they are allowed to do can be managed by GateKeeper[2].

[2] https://github.com/lostintangent/gatekeeper

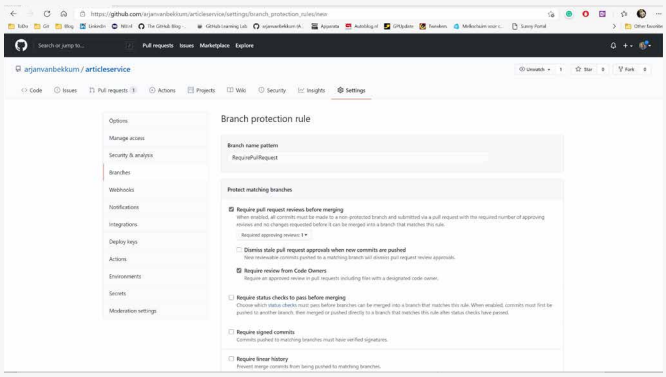
Storing phase

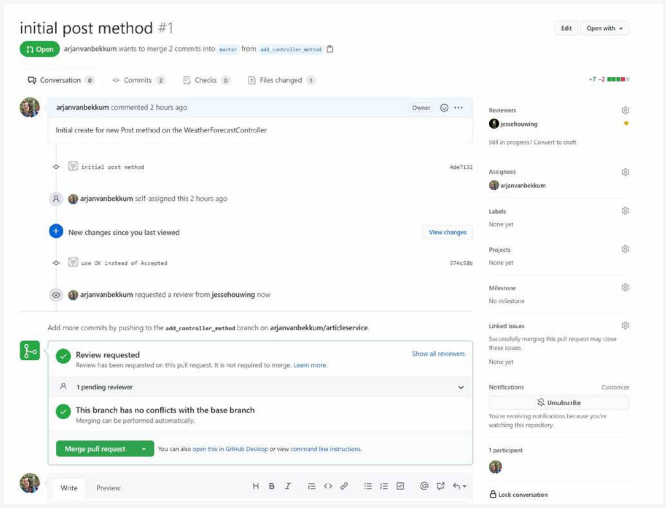
In the storing phase the engineer pushes code from his local machine to source control. When using GitHub, storing the source code consists of two phases. Committing the code to your local Git repository, and pushing the code to the Git repository that is also used by the rest of the development team.

To ensure a secure process, a number of things can be done.

Required code review

To ensure the 4 (6/8) eyes principle on every code change, generally the first occasion where you can do this is on code push to the Git repository. By defining a simple branching strategy where people create short-lived branches for their code changes and protect the main branch from direct check-ins with a branch policy, you can easily enforce that someone other than the author reviews and approves changes to the code base. With GitHub you can use the settings tab in your repository to set these policies on the branches. You can create different policies and apply them to different branches. To apply the policy to all branches, specify the “Branch name pattern”. Wildcards are allowed, so “*” will apply the policy to all branches.

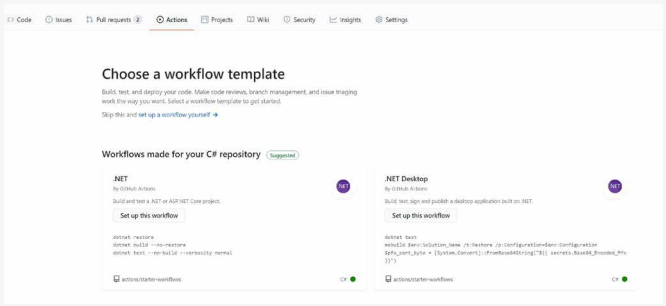




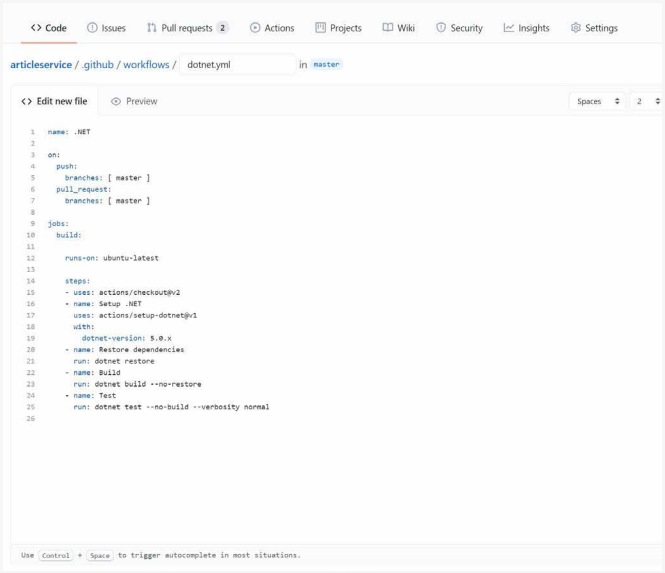
Validating code phase checks against the code base
Assuming that an engineer ran the checks described in the code phase on his workstation is fine, but it does involve risks. However, to make sure nothing slips through the cracks, you can set up a Continuous Integration (CI) build. Running a CI build after every push to the short-lived branch validates the changes made in the short-lived branch, combined with the code base where other engineers work. This practice gives even more assurance. This build should include (at least) the following code phase checks:

› compiling / syntax checking
› static code analysis
› unit tests
› credential and secret scanning.

When you use GitHub, you can use GitHub Actions to perform the actions. GitHub also includes automated security scanning for credentials on every check-in. When you check in a credential by accident, you will be informed about this by GitHub.



You can create a new action (or workflow) on the “Action” tab. Depending on your repository you can use a default workflow or you can create your own.



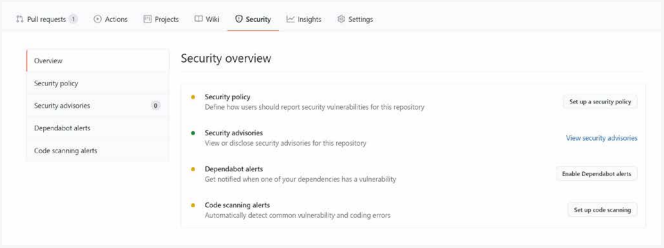
The workflow pipeline is created as code and added to your repository. Changing this workflow will result in a code change and thus it will be part of the branch policies on the repository.

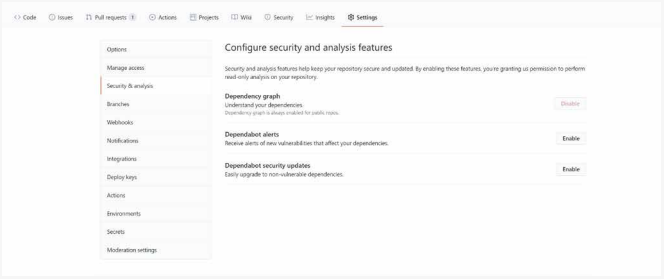
Vulnerability and dependency scanning
Scanning your own software is one thing, but in modern software development, over 70% of the software you deliver is not written by your own development team (check out version 9 of our magazine and read “99% of the code isn’t yours). With the rise of Open Source Software and Package Management Tooling, Artifact Repositories and Container Registries, the use of software that was written by others became mainstream.

Conceptually this is perfect. The less you have to do yourself, the better it is. It makes people more productive and in many cases, the people that wrote a specific Open Source Library are more knowledgeable on the subject than you yourself. However, using the software of others, open-source, or purchased from a vendor, is a potentially dangerous practice. The software that you use as part of your own software may contain hazardous vulnerabilities that can be exploited.

GitHub integrated security scanning for vulnerabilities in their repositories. When they find a vulnerability that is solved in a newer version, they file a Pull Request with the suggested fix. This is done by a tool called Dependabot (https://dependabot.com/).

You can enable Dependabot on your GitHub repository using the “Security” tab, click on “Enable Dependabot alerts”, and pick the setting you need.





An alternative for Dependabot is NuKeeper, which provided similar functionality (https://github.com/NuKeeperDotNet/NuKeeper).

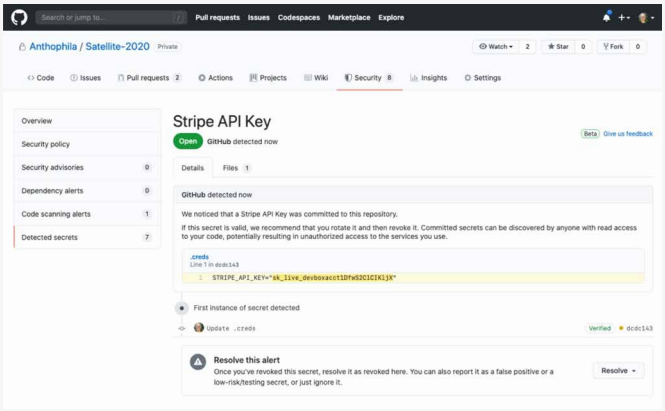
To learn more about integrating vulnerability scanning in your pipeline, you can follow the lab "Managing Open-source security and license with WhiteSource" on Azure DevOps Labs (https://www.azuredevopslabs.com/labs/vstsextend/whitesource/).

Credential and secret scanning

Of course, you are scanning your local repo for accidentally committed credentials, but sometimes your scanning tool will learn new patterns after the fact. GitHub Advanced security now has automated scans to detect leaked credentials on push and will keep monitoring your repository even afterwards.

If you are not scanning, be aware that many threat actors do. They look at a wide range of interesting repositories and offer GitHub-wide search patterns. It may take only five minutes for your shared AWS key to be detected and exploited to deploy miners or ransomware to your cloud environments. When undetected by you, it may cost $60k within a couple of days.[3]

When credentials are detected by GitHub, it will automatically revoke them to prevent others from exploiting the key. GitHub integrates with major cloud providers to provide this service.



Build phase

When code has been created on the local workstation and is stored safely in source control, the delivery process can really start. The software needs to be "packaged." Compare it with an assembly line where products roll off the belt and are packaged in a big box. This box is signed, sealed, and delivered to the warehouse where it can be picked up for further delivery. In essence, the build pipeline works in the same way.

During the coding and storing phase, we already ran several checks that quickly provided feedback about the quality, security, and stability of the code. In the build phase, we add some more checks and validation and, eventually, package the product:

› build activities from storing phases
› second stage - static code analysis
› vulnerability and dependency scanning
› license scanning
› securely storing the build artifact
› protecting the build history.

Set up a Continuous Integration pipeline on all your branches. When engineers push code to a branch in source control, the validation should start directly. On many occasions, the full build only runs after merging the changes to the main branch.

Securely storing the Build Artifact

One of the main purposes of a build pipeline is to produce:

› an artifact that can eventually be deployed on an environment;
› an artifact that creates an environment;
› a set of scripts that will set the required configuration.

In any case, it is essential that we make sure that the artifact is uniquely identifiable. This allows us to ensure that nobody tampered with an artifact before it landed on production and ensure that the code we produced is actually the code that runs. Storing the artifact that the build pipeline produces is, therefore, an essential task in a secure pipeline.

Within your build pipeline, you can produce two types of artifacts:

› packages or containers that will be consumed by other software and will not run by itself;
› software packages or containers that will be consumed by the end-user or run a process.

When we build software packages, like NuGet packages, NPM packages, PowerShell Modules or even containers, we should immediately think of artifact repositories. We publish our packages to a gallery or repository so that others can consume them. We can either make this publicly available (Open Source) or internally available (Inner Source). To be able to store the artifact, the artifact needs to adhere to a number of simple rules. For example, it needs to contain a unique version and a manifest that describes the package.
To publish the package, the publisher requires authentication.

[3] https://www.theregister.com/2017/11/14/dxc_github_aws_keys_leaked/

This combination, versioned package and secure connection will ensure the integrity of the package.

Strangely enough, when we deploy our website or application to our production servers, we treat it differently. We build our software in the pipeline and copy the files to production. Sometimes we store an artifact on a network share or disk before we release it. But storing it on a disk can allow others to modify the package. Also, our versioning is not always straightforward when it comes to our own files.

To ensure the integrity of our software, the build pipeline and storage location of the artifacts need to be secure as well. When using GitHub, you can upload the build artifacts on the server. There is no way somebody can modify the package on the server. By securing the pipeline and versioning the packages, you drastically reduce the risks of insecure software. In addition, uploading the artifacts GitHub also allows you to use GitHub Package Repository to store your inner source packages. You can even use the building GitHub Container Registry to store your Docker images directly from your workflow.

Deploy phase

The deployment phase is the phase where all the activities of previous phases come together. Code that has been checked by one or multiple teams has been transformed into packages or deployable artifacts. During the deploy phase, the release pipeline is the mechanism that is used to move things from a protected, private environment to a location where others can start using it.

Typically, a release pipeline is built up as follows:
› gather artifacts from one or more sources
› deploy infrastructure
› configure infrastructure
› validate infrastructure
› deploy application
› configure application
› validate application.

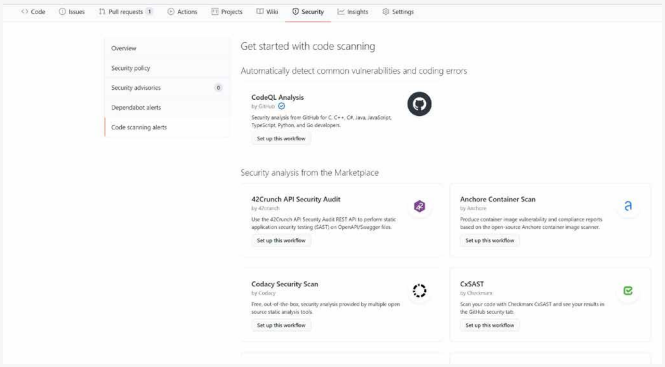
When we look at the activities mentioned above, there are a number of things we need to ensure when we talk about a secure pipeline.

Run dynamic security tests on infrastructure

Dynamic Application Security Testing (DAST) is a process of testing an application or software product in an operating state. This kind of testing is helpful for industry-standard compliance and general security protections for evolving projects. Good examples are scans for SQL injections, cross-site scripting etc. When an application is deployed multiple times a day, it is necessary to perform the security checks every time instead of checking it once (like in the old days). By using Automated Dynamics Security Testing tools, you can automate these attacks.

A great tool to run Dynamic Security Tests is OWASP ZAP. Find the OWASP ZAP task in either the GitHub[4] or Azure DevOps marketplace[5].

GitHub allows you to easily enable scanning on common vulnerabilities and coding errors. By using the security tab you can create this workflow, which will run on every branch you create. CodeQL is a semantic code analysis tool, and it allows you to query your code to find vulnerabilities.



Run tests that require a deployed application

Although the software has been tested in the build phase, preferably by running unit tests, you also need to run tests that require a deployed application, i.e.. integration tests or end-to-end tests.

GitHub offers GitHub action to integrate your own test runners and allows you to run this as part of the deployment process.

Monitor key metrics after deployment

When you have deployed your application, how do you ensure it is running correctly? Of course you need to check some fundamentals by running a smoke test, by checking whether the application responds. But it is also wise to start gathering metrics about the baselines of your application. What is the response time, what is the CPU load? When you know these baselines, you can check these metrics after a new deployment and validate whether they are still the same or at least did not deteriorate.

Set up secure endpoints to the target environment

Of course you need to check your own software for all kinds of security issues. But the pipeline itself and the connection to the target environment also needs to be secure. When you deploy a new version of an application, you probably need some sort of configuration in the application itself. You may also need some secrets like passwords or access tokens to deploy the application. Within Azure DevOps you can use Service Connections to create a secure endpoint. In GitHub you can store the publishing credentials in a GitHub Secret. This way you ensure that the pipeline is the only way to deploy an application. This simply uses a key-value pair where you can use the name of the secret in the action workflow as an environment variable.

[4] https://github.com/marketplace/actions/owasp-zap-full-scan
[5] https://marketplace.visualstudio.com/items?itemName=CSE-DevOps.zap-scanner

Release phase

In contrast to what many people and companies think, the release phase is not the same as the deploy phase. On many occasions, it is still the case that deployment is equal to releasing but by having this dependency, there is also an implicit security risk. When releasing is like deployment, this means that the moment you deploy the software, it becomes available to your end-users. Because you probably need to check a few things before allowing customers to start using the software, the only way to do this is to plan for downtime. A service window is usually the way to do this.

But restricting the release/deployment times to a strict release window also limits the possibility of delivering new features, or worse, security patches. We all know that waiting for an appropriate time to roll out a security fix may imply a much more significant risk.

Building your software and pipelines in such a way to allow the software to be released, without impacting the target environment, is not only the way forward for businesses to deliver new features quickly to their customers, but it dramatically reduces security risks because patching them is a matter of starting a new deployment. When you use feature toggles in your code, these can help in facilitating this. Feature toggles allow you to disable or enable functionality. If the toggle is "on", users are allowed to use the new functionality. If the toggle is "off", the functionality cannot be used. Feature toggles allow you to change the behavior of the application without changing the code.

Conclusion

When you develop an application you should do this securely by default. There are a lot of tools that can ease the life of developers and increase security. Just implementing the tools is not enough, you also need to understand why these tools are needed and support them.

GitHub supports a lot of security features out of the box. You need to secure the infrastructure, your software, but also your delivery pipeline. Focus on shifting the security left in your process. </>

René van Osnabrugge
ALM, DevOps, Continuous Delivery, Initiator and Inspirator

xpirit.com/rene

Jesse Houwing
Making software development fun, Trainer, Coach, Tinkerer

xpirit.com/jesse

Michiel van Oudheusden
Microsoft .NET consultant, developer, architect. With a focus on ALM, VSTS, DevOps, APIs, Azure, Containers and everything around it

xpirit.com/michiel

Arjan van Bekkum
Consultant

xpirit.com/arjan

GitHub Actions: running them securely

GitHub Actions[1] are a powerful way of creating a pipeline to act on events in GitHub. By creating a workflow file you run actions on code updates to build your application, automate triaging tasks from issues, and loads of other helpful uses.

Author Rob Bos

Tyranny of the default

Every demo on GitHub Actions shows how easy it is to get started: add a text file with some actions in it and you are good to go. Unfortunately, this is highly insecure! To understand why, you need to know what the attack vectors of your workflow are and how you can guard yourself against them.

Let's start with an introduction to GitHub Actions first.

By storing the dotnetcore.yml file in the right location, you have added a new workflow that can be triggered on events. There are a lot of events available, from the push event in this example(1), to comments on an issue and closing of a Pull Request.

main dotnetcore-webapp / .github / workflows / dotnetcore.yml

```yaml
name: .NET Core

on: [push]

jobs:
  build-and-deploy:
    environment: Production

    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v1
    - name: Setup .NET Core
      uses: actions/setup-dotnet@v1
      with:
        dotnet-version: 3.0.100

    # dotnet build
    - name: Build with dotnet
      run: |
        dotnet build --configuration Release ./dotnet-core-webapp/dotnetcore-webapp.csproj
```

Make your own Octocat: https://myoctocat.com/

[1] https://github.com/features/actions

In the jobs[2] section you can create one or more jobs that will run on a specific runner that executes the steps[3] in the sequential order within the file. In this example the repository is checked out(3) first, then a version of the .NET Core tooling is installed[4] and in the last step the .NET Core project is built using the tools[5].

Know your GitHub Actions

When using GitHub Actions it is important to understand what the actions you use are doing. You can use any action by leveraging the setup from GitHub: the action identifier is the organization or username that is hosting the action, and the name of the repository it is in.

In this example you can find both actions in the 'docker' organization in their own repositories. Adding the action path to https://github.com/ straight to the action repo.

```
-name: Login to DockerHub
 uses: docker/login-action
 with:
   username: ${{ secrets.DOCKERHUB_
   USERNAME }}
   password: ${{ secrets.DOCKERHUB_
   TOKEN }}

- name: Build and push
  uses: docker/build-push-action
    with:
      push: true
        tags: user/app:latest
```

Having a valid action.yml in the repository makes it useable for every workflow. Using the action like this will ensure that the workflows will always download the latest available version of the repository and execute the code that is in it. This is also the greatest downside of actions: the default is already insecure! Anyone can create an action like this and there is no process that will check the action you are using for quality or security issues. Even limiting the actions people can use in your organization, to only the actions listed on the marketplace is insecure: there is no process that checks whether your action is doing malicious things.

The source of every action is public, which also means that you can look at the action repository and verify what it is doing when it runs. You can check whether it is sending your environment variables over to their own API for example, or logging your OS information together with your IP-address.

What are the risks?

It is wonderful to able to use actions that someone else already spent time and effort to create, potentially saving you a lot of time. However, this also adds some risk to your repository, the application you are creating and the setup around it. To get some understanding of the risk we need to look at the results of an attack on your workflows.

A malicious actor can wreak havoc on your application or its environment in three different ways:

1. data theft
2. data integrity breaches
3. availability

Data theft

By working their way into your workflows, people could get access to the code in your repository, but potentially also to the environment your workflow is running in. That environment could be set up to have API keys available for accessing services you need to build or deploy your application, or have certificates installed for code signing. It could even have access to an account on your cloud platform that has administrative rights and could get access to data or delete infrastructure there. Limiting the access for the runner that executes your workflow to the bare minimum is key in preventing against data theft.

When you run your workflow on hosted runners[2], it is GitHub's responsibility to keep them up to date with the latest OS and tool updates. To make sure the attack surface on them is as small as possible, they will create a completely new environment for each run and clean up the environment after it is no longer used.

If you run the workflows on private runners[3], taking all these security measures is up to you. Keep in mind that you are taking that responsibility when you install a private runner. You need to secure the OS, limit access the account the workflow is running under to only the things it needs access to (so do not assign network admin permissions to it!). You also need to keep the tools on that machine up to date with all the security patches.

Data integrity breaches

If a malicious actor has a way to get into your workflow or execution environment, they can also inject malicious code into your application. Most workflows create an artifact to deploy into an environment and store the artifacts in the pipeline environment. A possibility is that the attacker injects something into the artifact and the deployment will then deploy the malicious code for you! The recent Solorigate[4] attack is a prime example of this type of attack. Adding one malicious assembly before the artifact was uploaded (and avoiding a lot of different detection methods) was the central point the attack was executing.

Other examples of data integrity breaches are poisoning your dependency cache: there are a lot of blogposts[5] available explaining that you need to verify the dependencies you are using with, for example SHA512 hashes of the commit[6] to make sure you are not unknowingly pulling in a newer version of the dependency when you build your application.

Something similar happens with typo squatting attacks[7]: can you spot the difference between using 'npm install crossenv' and 'npm install cross-env'? An easy mistake to make, but if the first one is a malicious copy of the package

[2] https://docs.github.com/en/actions/using-github-hosted-runners/about-github-hosted-runners
[3] https://docs.github.com/en/actions/hosting-your-own-runners
[4] http://xpir.it/Solorigate
[5] https://xpirit.com/99-of-code-isnt-yours/
[6] https://w3c.github.io/webappsec-subresource-integrity/
[7] https://snyk.io/blog/typosquatting-attacks/

you need, with some bonus code that executes at runtime, you might be compromised before you know it! These attacks are now getting even more sophisticated by finding out the names of internal packages you use and host a malicious version on the public repository site. Most package tools have a default to check the public hosted endpoints first. If the package is not found there, it will try the same on internal endpoints. Take a good look at those configurations you are using.

Availability

An attack vector that seems less likely is injecting something into your workflow that will cause the workflow to stop running. These days, most DevOps teams are very dependent on their pipelines to push code to production, and they have a hard time getting updates out if their pipelines are not working anymore. To limit what engineers have access to, everything is locked down and only a service account has access to production. What if your application is down, or worse: vulnerable to an attack? What if someone can trigger your workflow to be unable to execute, right at that moment? Does your DevOps team have a 'break glass' option[8] to fix the vulnerability without their pipelines?

Attack vectors

By pulling in the action from the internet you are executing its code in your environment: this can be a hosted runner on GitHub's infrastructure, or your own runner in your own cloud environment.

The code in the action can do multiple things: it can send out your data, code or environment setup (SSH Keys, locally stored certificates, etc.) to an endpoint of their own and exfiltrate data that way. They can also try to get access to your environment or your GitHub setup: either the code in the repository itself or even try to get administrative access to the complete repository. They could pull in extra dependencies in your code, add other actions to your workflow, or even misuse your action runs with Bitcoin miners for their own gain.

There are multiple ways to try and get in. Every now and again GitHub has 'Capture The Flag' (CTF) events where they invite the community to try out a repository and gain access. From those events they learn a lot about their setup and ways to break the security around the repository. A basic example of an attack vector is the use of sending in a Pull Request that alters the workflow files itself by adding in a malicious action. More sophisticated attacks examples are adding JavaScript in the issue comment that is being picked up by the workflow and not handled securely: the JavaScript is then executed by logging it to the output for example (helpful to see them in the logs) which in turn enables the attacker to break out of the action environment itself and run a process on the runner environment. With that setup someone can create a new Pull Request for the repository that added the next step of the attack by writing code back into the repository. From the CTF events we learn the new ways to get access, and GitHub can try to prevent those types of attack.

[8] https://docs.microsoft.com/en-us/azure/active-directory/roles/security-emergency-access?WT.mc_id=AZ-MVP-5003719

Securing the actions you run

There are several measures you can take to secure your actions. Just using the latest version of the action is not a good idea: new code could have nasty side-effects like introducing new vulnerabilities, as we have seen in the previous paragraphs. The action repository might even be taken over by a new maintainer with ill intent and still compromise your setup. That is why running the action (as displayed in every demo!) like this example is a bad idea:

```
- name: Login to DockerHub
  uses: docker/login-action
  with:
    username: ${{ secrets.DOCKERHUB_USERNAME }}
    password: ${{ secrets.DOCKERHUB_TOKEN }}

- name: Build and push
  uses: docker/build-push-action
  with:
    push: true
    tags: user/app:latest
```

Option 1: Version tags

You can add the version number of the action to the end of the configuration, but there is no way to verify if it is still the same code: the tag can be reused with new code changes in it, so adding this does not add real security to it.

```
uses docker/login-action@v1
```

Option 2: At least start here

Start by verifying the actions you are running by looking into the action's repository. Have a sanity check on the code in the repository and use the commit SHA from GitHub to add that at the end of your action configuration:

```
name: Login to DockerHub
  uses: docker/login-action@
  e2302b10ccc2c798f917336fe81ce41ea8dea0fd
  with:
    username: ${{ secrets.DOCKERHUB_USERNAME }}
    password: ${{ secrets.DOCKERHUB_TOKEN }}

- name: Build and push
  uses: docker/build-push-action@
  0ec1157bb54f3e4676c823ef3497b53135ed39de
    with:
      push: true
      tags: user/app:latest
```

The commit SHA is immutable: if the code in the repository changes, the SHA will be different. This is the only secure way to know for sure that the code you are executing is the code you have checked yourself and that you have approved the risks that come from using it.

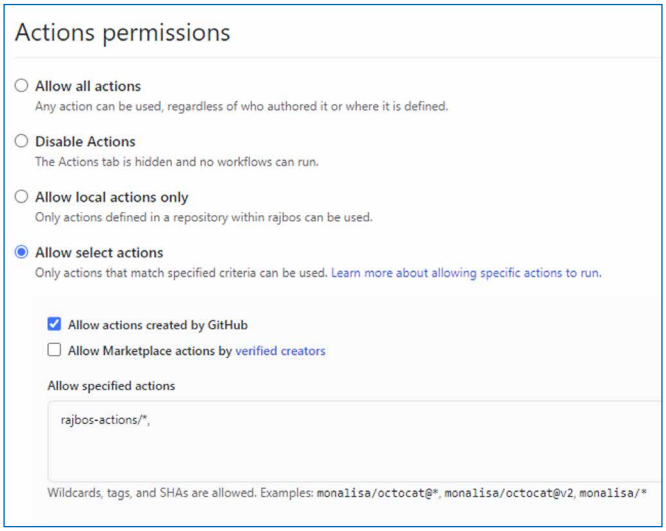
Staying up to date

Now that we are using the actions as securely as we can (by checking what it is **actually** doing and making sure no unseen changes can be added), the next question needs to be answered: how do we still get updates?

Since there is no update feed on the marketplace, or a blog that can be followed, I created a Twitter bot[9] that will regularly check for new or updated actions and will tweet them out. Checking the used action versions in your workflow files and updating them automatically can be done by using Dependabot[10]: it will scan your workflow files on a schedule and create a Pull Request for each updated action. This will give you a chance to manually verify the incoming changes and then accept the pull request.

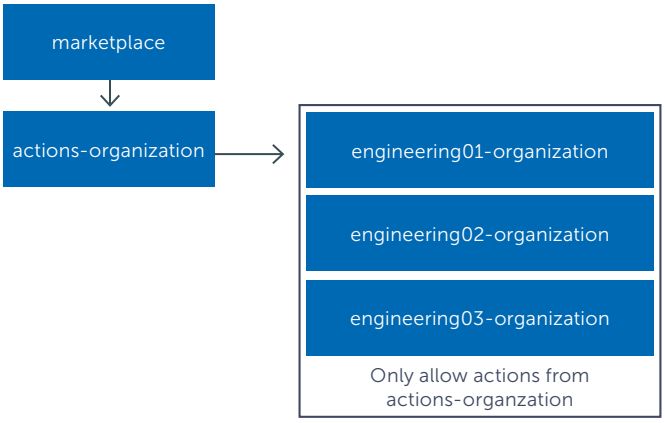
Option 3: Forking the action repository

The ultimate security setup I have found is forking the action repository to a specific organization for it. This way of working was suggested previously in documentation, but has not gained momentum.



Forking the repository gives you full control over the actions as well as their updates. It also provides a clear audit trail of the actions and secures you from actions being pulled by the maintainer. Additionally, you have a backup if the action gets deleted / renamed / moved to a different repository by the publisher. Remember the availability issues that can occur? This helps preventing that as well. You can now secure your other organizations (or separate repositories) to only allow actions being run from the forked repositories.

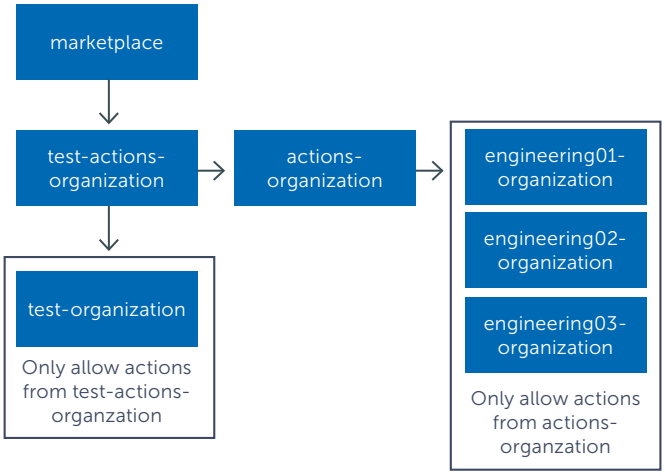
This is also an ideal strategy for enterprise organizations. You can create a specific actions-organization in which you fork all the actions you need. Then lock down the normal organization(s) everyone is using to only allow actions from the actions-organization. The setup would look like this:



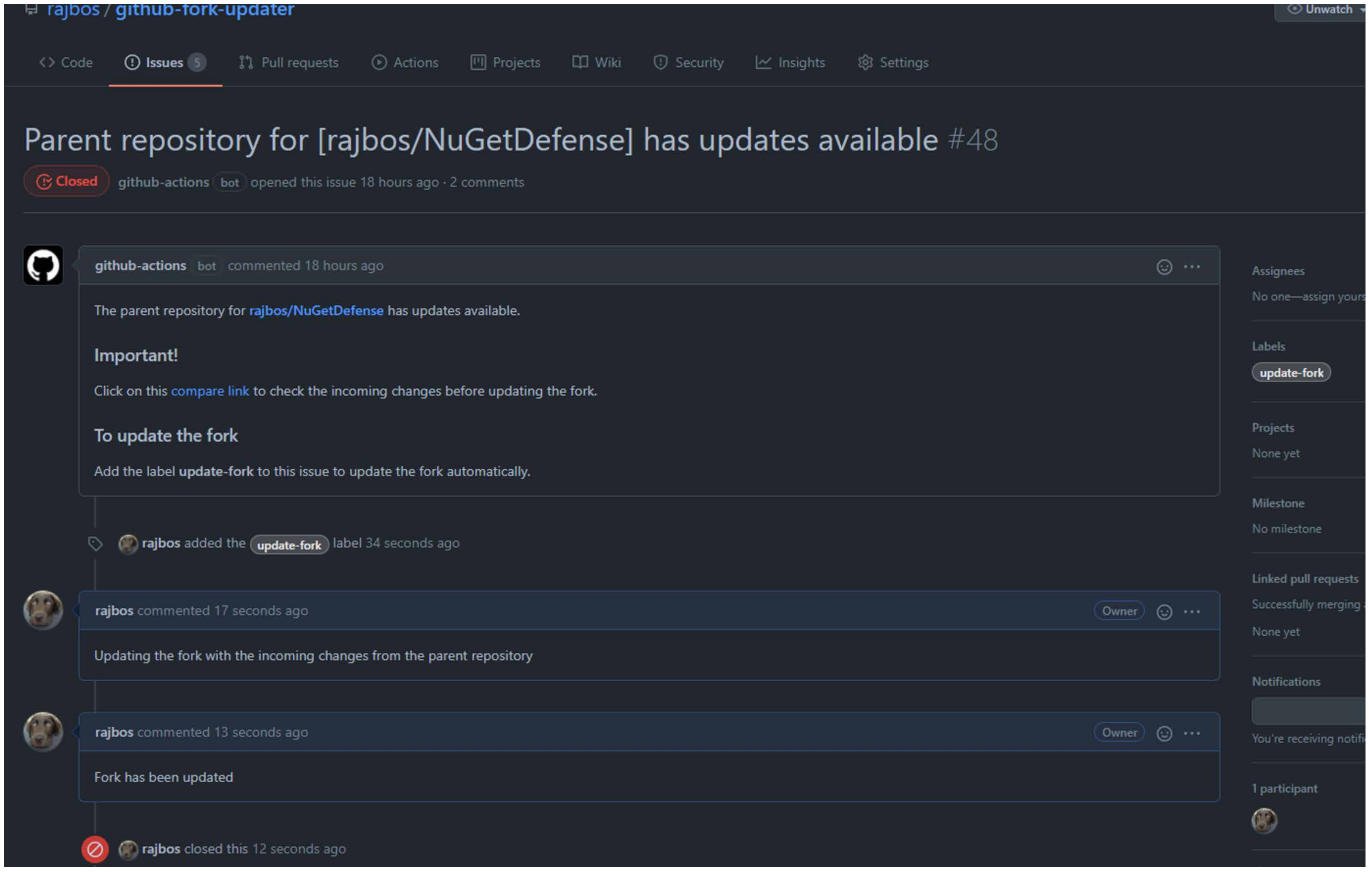
Enable your DevOps engineers!

Do not lock out you DevOps engineers: it is part of the DevOps way of working to let them take control over the tools they use. Add an organization in which people can pull in new actions to test with and validate their workflows, so they can still use new actions that you have not forked yet. They take ownership of the actions they want to use and fork the actions themselves!

That way they have full autonomy and will not be waiting for someone's approval before they can test new actions or updates.



[9] https://twitter.com/githubactions
[10] https://docs.github.com/en/github/administering-a-repository/keeping-your-actions-up-to-date-with-dependabot



Keeping your forks up to date

Now that you have secured your organization and made sure you are not blocking your DevOps engineers by empowering them to take control over the actions, you need a way to update your forks (all of them). To make this as easy and still secure as possible, I created the GitHub Fork Updater repository[11]: a specific repository that has everything in it you need. Fork it, add some configuration so that it can update all repositories in that organization, and you are good to go!
The update works as follows:

1. On a schedule, check all repositories in the organization of the fork using a workflow.
2. If there are updates, create an issue in the fork-updater repository.
3. With the default GitHub notification setup, your engineers will get notified of new issues.
4. They can check the issue and do the security check on the incoming changes using a special link in the issue.
5. By adding a label on the issue, they will indicate that they have validated the incoming changes and that they want to pull them into the forked repository.
6. A workflow is triggered on the labeling of the issue and the fork will be updated.
7. The issue is closed.

Summary

Using GitHub Actions from the marketplace is not secure by default: there are no real checks on the code they are executing, and it is up to you to verify whether the actions are safe to use.

Empower your DevOps engineers to take ownership of the actions by forking the repositories and doing the due diligence on them to make sure they will not send out your data to some unknown third party. This can be done by setting up a secured configuration with additional organizations in your GitHub account and forking all the actions you want to use there. Keeping your forks up to date can be automated as much as you can by leveraging the GitHub Fork Updater to stay on top of changes. Always verify the incoming changes! </>

Rob Bos
Consultant

xpirit.com/rob

[11] https://github.com/rajbos/github-fork-updater

Securing your Dev's Workstation

You don't want to be the developer who infects the company with malware or be the source of entry for an attacker. How do we stay secure and still have a happy CISO, complying with Security Rules and regulations? (And of course, having a fully working DevOps workstation).

Author Erik Oppedijk

Down the rabbit hole

Let's take a trip down the rabbit hole what typically might occur after a data breach/security incident. Management or a CISO might ask you to remove the Local Admin permissions from your laptop.

Without Local Admin, installing and updating some software is harder, so we need to rely on a support team to (quickly) package new software versions for the developers. Of course, this slows down as the support team isn't able to keep up with all packaged application updates.

Then "they" find out that the developers still run plenty of portable apps* (which don't require admin privileges to run).

This leads to a complete "Application Allowlisting"** scenario, where only a handful of approved applications is allowed to run. Of course, this slows down the developer productivity since no compiled executable can be made to run and any tool used must first be allowed by the security team.

A solution is devised that the developers should work from a VM or docker container. This in turn can be used by the developer for all kinds of things, including day-to-day tasks like reading his e-mail or installation of non-work-related software.

Finally, "they" find out that the VM/docker container with full access is used by the developers for all kinds of software. So, it is back to square one, "they" require the removal of Local Admin permissions from this VM/docker... and we start again at the top.

Take a step back: Looking at risk management

If we take a look at the developer population, we can divide them into several groups:

- Developers/Contributors to Code (Low Privileged Accounts)
- Project/Pipeline administrators (Medium/High Privileged Accounts)
- Production Access (High Privileged Accounts)

We want to utilize Privileged Identity Management (PIM) for the Medium/High Privileged accounts and for production access, so every time a user needs these permissions, an elevation is required. This would also be the group of people working with the most sensitive secrets and intellectual property/trade secrets.

This leaves the developer group with access to the source code, which, in a typical organization, does not contain any military secrets or extremely confidential source code. Our DevOps pipeline and four-eyes principle on check-in/merge already provides us with a nice first defense line.

As we saw in "Down the rabbit hole", blocking (or also called Application Allowlisting) isn't working very well for all developers, so we need to take another approach: Detection.

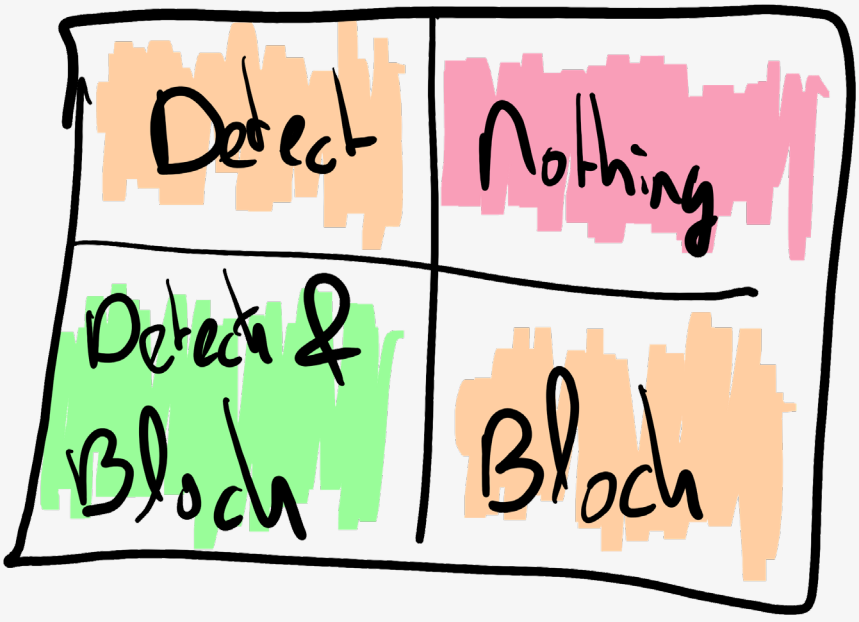
* Portable Apps are executables that allow to be run from read only or non admin locations, just from a user folder. Examples of this could be the user installation of Chrome, or tools like TeamViewer portable or 7-Zip portable.

** Application Allowlisting is only allowing certain applications to run, based on a hash value of the executable, any change (by an attacker or software update) will invalidate the hash and cause the application to be blocked.

With Detection, the developers can run all the things they want on their laptops, with an advanced "Endpoint Detection and Response (EDR)" tool available to detect malicious behavior. The EDR tool can spot suspicious processes/memory injections and detect connections to suspicious IP addresses. It works by looking at unusual behavior on the system.

With the EDR Tool in Detection mode, developer productivity is not harmed as it would be in the Block mode, which is used for regular users.

The biggest risk usually consists of unpatched software, which is often targeted by phishing attacks. Our best line of defense starts of course with educating the developers by informing them that they are the ones being attacked.



One of the ways is to patch all software on the developer's machine, coupled with the detection capability of an EDR, which should provide a nice line of defense.

Here come the auditors

The management/CISO go along with the proposal on how to secure your developer workstation, but quickly they start asking questions such as:

- Is the IT auditor pleased with this situation, how do the developers keep their machines up to date, and how does this fit in with the company policies which disallow everything, and more.
- How do we escape from this rabbit hole?
- If we take a look at the international ISO 27001 standard, there is a separate chapter (14) about software development in Annex A.[1]
- As with all standards, it is very important to read them and to understand the different terms including: **must**, **should**, **consider**, **depend**, **appropriately**, etc.

For instance, in A.14.2.6 Secure Development Environment, we should consider the sensitivity of the data, risk assessments, business/legal requirements. Back to our original assumption:

If the developer is not working with live production data, and is not working on extremely valuable code, then we don't need to take the same steps we take to protect our sensitive data/documents. According to the standard, we need to appropriately protect the environment, and not constrain it at all costs!

We need to classify code as "internal" and not as "top secret", because the secrets should not be accessible by all developers. With the concept of enterprise inner source (internal open source), almost all source code should not be sensitive. This allows you to focus on the real sensitive pieces, like the DevOps pipeline, or that single team that manages the code of your trade secrets.

Solutions

How do we solve our "problem"? Training and awareness should always be step number one, otherwise it is like rearranging the deck chairs on the Titanic. In addition, We need a combination of tooling and processes.

Tooling

There are several tooling options available to help remediate the problem:

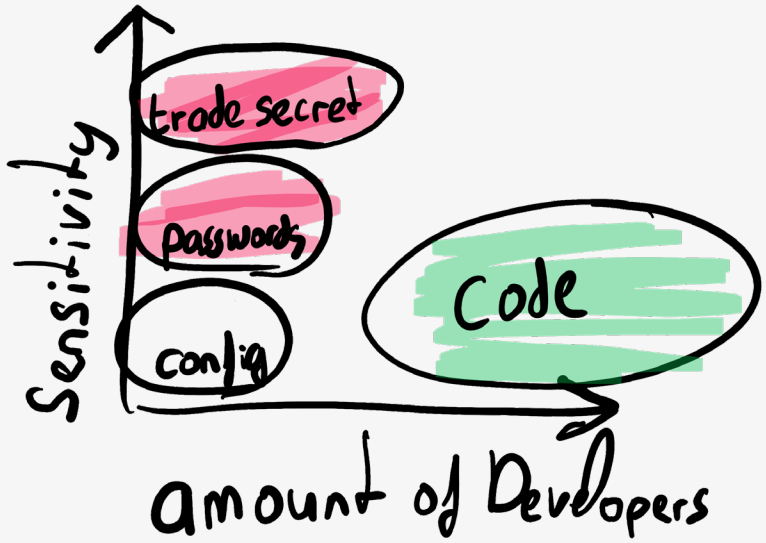
- Identity Protection (AAD P2 feature);
- Privileged Identity Management (AAD P2 feature);
- Defender for Endpoints (previously known as Defender ATP, not to be confused with Defender Antivirus which is a completely different product).

With Identity Protection we can specify conditional access rules based on risky behavior, e.g. a foreign logon location, change of browser, or a sudden location change during a session.

Privileged Identity Management is what we need when we need to elevate our permissions to perform a Medium or High Privileged Action, this is the Least Privileged concept: don't run as a High Privileged account by default.

The last one is Defender for Endpoints – this is an Endpoint Detection & Response (EDR) tool, which can detect suspicious behavior on the machine, like suspicious IP connections, running process modifications, but also vulnerable installed software.
EDR combines Alerting (and blocking/quarantine) together with Vulnerability Management capabilities to secure the endpoints.

[1] https://www.isms.online/iso-27001/annex-a-14-system-acquisition-development-and-maintenance/



Processes

The following processes are relevant:

- marking the developers as Priority investigation employees during SOC alerts.
- patching and automatically inform the developer on unpatched software and packages.
- The Security Operations Center with knowledge of development.

Whenever a security alert is processed by the Security Operations Center (SOC), the alerts for developers should receive investigation priority over regular users in the organization.
This ensures that suspicious behavior, like packages/scripts downloading extra content from the internet, is investigated with priority.

The best line of defense is keeping the machine up to date (See: background on patching). Tools like Defender for Endpoint can detect the vulnerable software, so that a workflow or preferably automation runs to directly inform the developer of the issues on his/her machine. This direct feedback loop is much better than having a monthly report being sent to the CISO on the state of all machines.

The last success factor is having a SOC team with development knowledge. How else can a series of suspicious activities followed by a flood of network connections be attributed to an attack, or just the test runner framework being used?

Best practices: background on patching and removing Local Admin

We all know that we need to patch our software, but only when we are not right in the middle of a refactoring session.

Combine this with running as a Local Admin and we have a potential disaster waiting to happen.

But what exactly is the impact of removing Local Admin, according to this research[2], of the 192 critical vulnerabilities on windows, 102 would be stopped by removing Local Admin permissions.

Applying system hardening (especially blocking process creation from Office or through WMI, very often used by malware) is another best practice for reducing the likelihood of spreading attacks. Hardening steps can be gathered from the Center for Internet Security (www.cisecurity.org) or if you've deployed that Endpoint Detection and Response (EDR) product, it will show you recommendations to beef up the security of your system.

But if we look at the total of critical vulnerabilities(192), only 2.5% is used in the wild to take over machines. This still leaves 5 critical items to fix, and they can be fixed by patching your machine. **There is no excuse for not patching your system.**

Quick patching is the best defense against almost all threats, so don't delay installing those patches for a long time.

Summary

Patching, patching, patching, just patch your machines, no excuses! Combine this with an alert system from the EDR where you as the developer directly receive the alert of out-of-date software and missing OS updates.

Don't run as admin by default on your DevOps workstation, run as a normal user, and make sure you can use that Local Admin account to temporarily elevate your permissions to Local Admin. (Just make sure that you/the developer cannot login with that account).

Apply hardening on the system so that for instance spawning processes from Office Application or WMI (well-known malware techniques) are blocked.
This is also known as Attack Surface Reduction.

Enable monitoring software to help you identity suspicious behavior, linked with direct feedback to the developer.
The Endpoint Detection and Response tools can also notify you of suspicious actions.

Establish priority for security warnings on developer machines and accounts in the SOC team, so alerts are investigated with high priority by a team of SOC analysts with developer knowledge.

But the most important of all is continuous training and awareness! </>

Erik Oppedijk
Cloud Architect, Public Speaker and Trainer

xpirit.com/erik

[2] https://www.theregister.com/2021/03/17/microsoft_vulns_admin_rights/

Creating an open source learning project

Azure Functions University is an educational project for learning about Azure Functions - the Functions as a Service offering in Azure. The content is aimed at people who do not have previous experience with serverless technology and want to learn by following exercises and writing code.

Author Marc Duiker

The idea behind the project

I started this project because I want to enable newcomers to serverless technology to get up and running with Azure Functions in a very low friction way. Learning new things can be challenging, and frequently, the official documentation alone is not enough to understand a new topic and put it into practice.

The dual-channel delivery, lessons on GitHub and videos on YouTube, is intentional because some people prefer watching (or listening) to videos, while others prefer reading.

How it started

The Azure Functions University project started in October 2020. I have had quite some content on both GitHub and YouTube for some years now, but most of that was intended for intermediate or experienced users of Azure Functions. Since there is a huge increase in people new to programming, I want to help out that group and make it easy for them to start with serverless technology.

I consider myself reasonably experienced with Azure Functions. On the one hand, that's good for the project, so I can share a lot of what I know. But on the other hand, this can be a pitfall because I'm likely to have assumptions on topics that people new to serverless don't have. To prevent too much bias from my side, I wanted someone relatively new to the technology to co-create the content and co-host the live streams. I was following Gwyneth Pena (US) on Twitter, and since I really like her personality and the style of her videos, I asked her to join. I was thrilled she said yes immediately.

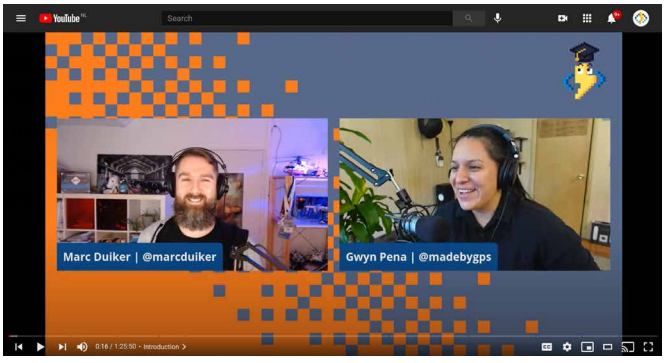


Figure 1. First Azure Functions University video about HTTP triggers

README.md

Azure Functions University

Welcome to Azure Functions University! 🎓

This repository contains everything you need to learn about Azure Functions <⚡> and complements the learning videos made by Marc and an amazing group of contributors!

You can watch the videos on the *Serverless on Azure* YouTube channel.

Figure 2. Azure Functions University GitHub repo

Curriculum

At this moment, the curriculum contains the following lessons:

› **HTTP;** How to do GET requests and use query string parameters and do POST requests where the data is read from the request body.
› **Blob;** How to use output and input bindings to read/write data from/to Blob storage using different binding types, using the BlobTrigger to start a function when a blob is written to storage.
› **Queue;** How to use output bindings with various binding types, using the QueueTrigger to start a function when a message is put in a queue.
› **Table;** How to use output and input bindings to read/write data from/to Table storage with various binding types.
› **Deployment;** How to deploy your Function App to Azure using VSCode, Azure CLI, and GitHUb Actions.
› **Configuration;** Why and how to use app settings in your Function App, using App Configuration service for easier management for app settings across multiple resources.
› **CosmosDB;** How to use the output and input bindings to read/write data from/to CosmosDB, using the CosmosDBTrigger to start a function when a new document is added to a collection, and using KeyVault to store the CosmosDB connection string.
› **Durable Functions I;** Why using Durable Functions is beneficial when dealing with multiple functions. This demonstrated by using the function chaining pattern to illustrate how orchestrations work.

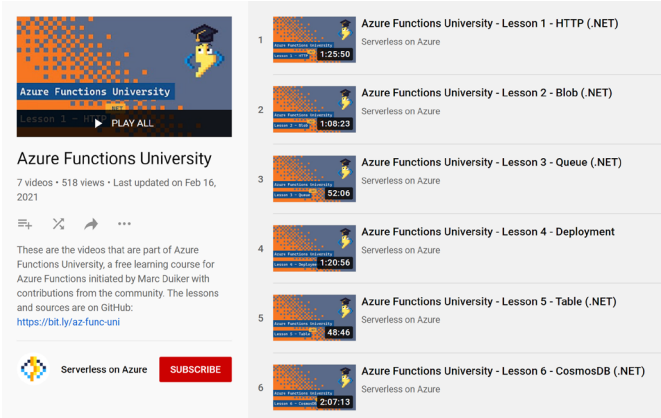


Figure 3. The Azure Functions University playlist on YouTube

I believe that consistency is key when creating educational content. Therefore each lesson follows the same structure:

› there are several exercises written in markdown, including code snippets;
› three types of call-outs are used: tips , observations and questions ;
› a complete Function App project is available as reference;
› at the end of each lesson, there's a homework assignment.

All coding exercises use VSCode as the code editor because this is a more beginner-friendly environment than Visual Studio 2019.

Although we started creating content for .NET functions, we're now also accepting contributions for other languages. TypeScript is the second language we have some lessons for now.

Challenges

Creating quality content is hard, and it is very time-consuming. For the first couple of lessons, I created most of the content myself, which was hard to combine with a full-time job. Since more people are helping now, it gets easier, although reviewing the pull requests is a considerable effort. I want to ensure the tone of the lessons remains constant and that inclusive language is used. I now realize what it feels like to be a maintainer of a small open source project.

The frequency between the lessons varies between two to four weeks. Ideally, I'd like to have a livestream every other week. However, planning is tricky since schedules and priorities shift, not only mine but also the contributors. This is voluntary work we all do in our free time, and sometimes other things are more important, and that's OK. Working on this project should be enjoyable, not stressful.

Keeping the lessons up-to-date is becoming a challenge right now. The current .NET content is targeted for .NET Core 3.1. Since Functions can now also be written in .NET 5, additional content needs to be created soon to reflect this. The .NET Core content will remain since .NET Core 3.1 has long-term support, and I expect the content will remain relevant for a while.

This brings us to another challenge, and that is the Azure Functions University GitHub repository. At the moment, there are eight lessons across two programming languages, .NET Core and TypeScript. Sub-folders are used for each language in order to keep everything tidy, but eventually, the source code needs to be split into separate repositories for each language/runtime. This will make the source code easier to manage, and VSCode will be less confused about which projects to run.

Marc Duiker
Consultant

xpirit.com/marc

What's next?

There's a lot of progress to be made. First, there is still a lot of new content to be written. Many topics have not been touched yet, e.g., security, SignalR, EventGrid. There are also content translations to the other languages that Azure Functions supports. Some people did show interest in helping out with Python and TypeScript, but it's still a long way to go until that's on the same level as the .NET lessons.

Secondly, I want to have better insight into how many people are using the GitHub repo and how they experience it. I'll be looking into GitHub classroom to see if I can get a better grip on the usage of the lessons. I prefer to have as little friction as possible, because additional sign-up boundaries might prevent people from using the material.

Will this project ever be finished? Not any time soon, I think. The Azure Functions team recently presented their roadmap for the next major releases. I expect plenty of opportunities to create new lessons and help more people to use serverless technology.

Help us!

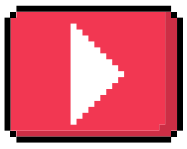
We're always looking for contributors who can help create content and co-host a live stream! Contributions can be new lessons, additions to existing lessons, or 'translations' to other programming languages (TypeScript, Python, PowerShell, Java).

Please have a look at the existing issues to see if you can contribute to those. If there is nothing to your liking, you can submit a new issue. You don't need to be an expert on the topic. We can work on the content together. </>

Links

YouTube playlist: https://bit.ly/az-func-uni-playlist
Azure Functions University GitHub repo:
http://bit.ly/az-func-uni
GitHub issue list: http://bit.ly/az-func-uni-issues

96
Stars on the GitHub repo



650
YouTube subscribers

>1850
Views of the first lesson

Introducing Xpirit Cloud-Native Software Development

At Xpirit, we believe that developing applications for the cloud is a new expertise that requires more than just a thorough understanding of cloud capabilities. You also need a specific way of working and a different mindset. It is essential to adopt the business perspective to see how organizations can achieve their goals by using the cloud as an enabler and an essential part of building software.

Authors Alex Thissen and Loek Duys

Looking at IT from a business perspective

Modern high-performing organizations make effective use of IT as part of doing and running their business. The created supporting software solutions require a fast time-to-market to be relevant for customers, companies and employees alike. Quick feedback from end-users and production systems enables an adaptive approach to evolve ideas and solutions to stay relevant. Looking at software solutions from the business perspective shows a couple of traits that define modern, competitive solutions:

- **Be cost efficient**
  A solution should have mostly operational costs and no significant capital expenditures, such as initial investment in hardware. The resulting operating model has low upfront investments and scales the costs less than proportional to the solution's use, growth, and success.
- **Differentiate on business essentials**
  You want to focus on the differentiating parts of the solutions. Common functionality and cross-cutting concerns should be ready-to-use building blocks. The custom-built parts should be essential to business to justify development.
- **Effective operations and maintenance**
  Automation makes software solutions effective in operation and easy to maintain. Achieving full automation eliminates any manual steps. It reduces the risk of human errors and speeds up development processes by avoiding the availability of people needed to perform actions.
- **Enable autonomous teams**
  Teams combining business and IT want to be in control of the solutions they create and take full responsibility for building and running it.
  These decisions and actions also relate to infrastructure, hosting, deploying and releasing software. Self-service provisioning gives teams the ability to create all aspects and parts of the software solution on-demand at any time.
- **Secure and compliant**
  Any solution must be secure and compliant by default. A solution architecture is designed with that in mind. The build and release process uses quality gates to automate security and compliance checks on every change of the solution.
- **Provide business agility**
  Becoming agile means drastically reducing the time from idea to production and being able to adapt as fast as possible to opportunities and changing circumstances.
  Again, automation helps to maintain a high-quality state of the system, allowing a release of functionality at any given moment.

Cloud-native applications as the new norm

Cloud-native applications are a perfect fit for software solutions in modern organizations. They have the mentioned characteristics by making optimal use of cloud capabilities. A cloud-native application deeply integrates with managed platform services in the cloud. It leverages these as building blocks for common functionality to focus on the differentiating, custom-built parts.

Additionally, teams can create automated pipelines for building and releasing cloud-native applications utilizing the high degree of automation in the cloud. Based on the cloud's pay-as-you-use model it becomes possible to take costs into consideration when architecting a cloud-native solution. The cloud also allows view the costs during operation, so teams can be in control of how much is spent on running and see the effects of scaling the applications.

The cloud allows on-demand provisioning of resources, a team can use this to automate the creation of environments. These environments can range from long lived in production to short lived during testing and even for training purposes. The cloud offers monitoring facilities to observe the application during operation and react to any incidents.

Cocreating business solutions

Creating business solutions using cloud-native applications should be a joint effort between business stakeholders, domain experts, the cloud engineers and developers.
The people with technical roles in the team should acquire the necessary insights into the domain. A thorough understanding of the business and domain logic is essential to build a successful application. The business stakeholders and domain experts need to transfer that knowledge by working inside the same team.

Cloud-native applications allows everyone in the team to focus on those differentiating, often complex parts of a solution, as the less relevant parts take less time to create. Also, as the entire team gathers more knowledge, it can quickly iterate to include new features, refactor for maintainability, improve performance and stability and fix any issues.

Agile practices with DevOps and SRE

Given how a team can utilize the capabilities of the cloud to create cloud-native applications, it can adopt new practices and methodologies in their way of working. The applications align well with teams that practice DevOps, Site Reliability Engineering (SRE) and other agile practices, such as Scrum. The applications facilitate a blurring of the line between development and operations. The teams can both create and operate the applications in full control and autonomy. SRE becomes a matter of using the cloud for global availability and replication, self-healing capabilities and applying resiliency patterns.

Identity and Access Management (IAM)

Dealing with security, accounts and social identities in cloud solutions can be challenging. It is a complex and specific set of features that involves practically every corner of your application landscape. Strangely enough, IAM is not part of an application but is required nevertheless to offer authentication and authorization functionality. It is a cross-cutting concern and essential to enable creating secure cloud-native solutions easily. Yet, this is often overlooked when starting a transition to the cloud.

A modern organization needs a proper cloud identity platform with IAM facilities to provide secure access to its applications and data. It will allow Single Sign-on (SSO) to web applications for its employees, customers and other users, who no longer need multiple accounts to login. Also, a cloud identity platform gives control, insights and monitoring capabilities for identity lifecycle management.

Application modernization

Usually, companies already have an existing landscape of applications, where not all applications meet current business requirements or standards for software development.
Such applications can be modernized to meet your ambition as a company. Application modernization means more than just lift-and-shift cloud migration. It is an ideal moment to choose an appropriate strategy for each application to be refactored, rehosted or rebuild on a new platform.

Our approach to determine the best strategy for modernization includes the following steps:

1. Identify current requirements, challenges and goals
2. Perform functional and technical decomposition of the current application(s)

3. Choose migration strategy per functional area and component
4. Define alignment of the new solution in future state architecture and application landscape
5. Design and develop new application parts as a cloud-native solution

Practice what you preach

From the very start Xpirit has been providing help to companies, teams and people who want to create modern solutions based on the Microsoft platform and development tooling with agile practices. The focus was mainly on consultancy and coaching around the software development process and using cloud technology, and only partly on building and implementing the solutions we advise on.

Early 2021 we decided that we should start offering additional services to our customers and help design and create cloud-native solutions and supporting capabilities. These services cover four areas:

1. **Develop mission-critical applications:** greenfield development of cloud-native solutions
2. **Application modernization:** migration of existing applications to become cloud-native
3. **Establish cloud identity platform**
4. **Training and workshops to learn practices, patterns and skills for cloud-native development**

Our services are geared towards helping customers solve business problems using modern, high-quality software designed and built using cloud-native technology. Our propositions are about people: skilled DevOps engineers that are experts in Microsoft Azure cloud technology, the .NET development platform and matching front-end technology. They can act as a team to build the solutions. Alternatively, they can augment existing teams in a leading role to provide cloud, DevOps and SRE knowledge to build the solution and train the team members while doing so. As a multi-disciplinary team, they create the new modernized, future-proof cloud-native solution from scratch or by modernizing existing ones. On top of this, we can help deliver a cloud identity platform using our experience and expertise to complement the creation of secure cloud-native solutions.

Let's fly to the cloud

Xpirit is venturing into the cloud even more by providing services to design and create your business solutions with cloud technology. We would love to make you part of that flight into the cloud. Whether you are a customer or a new team member, we are passionate about building the best cloud-native solutions together with you. Reach out and join us for a journey into the future. You can contact us at athissen@xpirit.com or lduys@xpirit.com. </>

Alex Thissen
Architecture and coding

xpirit.com/alex

Loek Duys
Cloud software architecture

xpirit.com/loek

Introducing Xpirit DevOps services

Xpirit is the authority on Microsoft consulting, ranging from DevOps and Cloud to management consultancy and cloud-native software development. However, our customers wanted more – more help from a great team of Microsoft experts in every part of the lifecycle, be it advice, building or maintaining.

Authors Marc Bruins and Suraj Sewbalak

To satisfy this need, we started with a new label Xpirit DevOps services which started on 1 January 2021. At Xpirit DevOps services, we believe that the existing managed services industry is about to be changed radically. Xpirit DevOps services is one of the companies that is leading this change.

For example, instead of focusing on maximized SLA percentiles, we focus on SLA percentiles thar are as low as we can afford. The lower the SLA, the more room there is for the teams to experiment, fail-fast, and innovate. This is in line with what the industry is accelerating towards, as we move towards a lean organization by embracing cloud, agile and DevOps. We believe that our managed services proposition is an enabler for your organization, not a blocker. Hence the name, Xpirit DevOps Services.

All the cloud experience from our consulting label allowed us to create a product that enables our customers safely inside the cloud. This is ideal when you are migrating, or when you want to restructure your cloud usage. Our Azure landing zone provides a safe place to land your workloads in the cloud. We offer a fully compliant and secure Azure in a box solution, with CI/CD pipelines, four-eyes approval, cloud native resources, monitoring options, etc. And it goes without saying that our Xpirit DevOps services also provide support. We have multiple, specialized solutions ranging from Government, Education, Data&AI and Business applications.

In short, we want your DevOps teams to have full control by embracing DevOps, cloud and SRE, together with all the experience from our customers that allow us to build suitable products. We are very happy to have already onboarded a number of customers and we are off to a good start.
We would love to have a talk with you if this article has peeked your interest. Let us know! </>

Marc Bruins
Architecture, Azure, mobile development

xpirit.com/marc

Suraj Sewbalak
COO Xpirit DevOps services

xpirit.com/suraj

The reliability paradox: Why less can be more

You've made the change from on-premise to the cloud, and your application is running like a charm. In true DevOps fashion you are focusing on building and running the app so you've taken certain precautions: retry mechanisms, fast failovers and smart alerting rules have been implemented. While the resilience of the system is improved, we should avoid the mindset that we are completely in control of the system's reliability.

Authors Geert van der Cruijsen and Casper Dijkstra

When we ask customers how reliable their application should be, expectations usually are around 100%. That would be desirable indeed, but is this really a target worth pursuing? Which price are we willing to pay for overly high availability targets? In order to answer this question, we should get some insight into the pros and cons of tightening and loosening reliability objectives. Are you focusing on the right things? Who decides how reliable your application should be? And is there a drawback to too much reliability?

What is reliable software?

Modern applications are based on multiple cloud components. These typically come with a Service Level Agreement (SLA) of three nines (99.9%) or three and a half nines (99.95%). Let's focus on the interplay of Azure App Service and an underlying SQL database as an example.

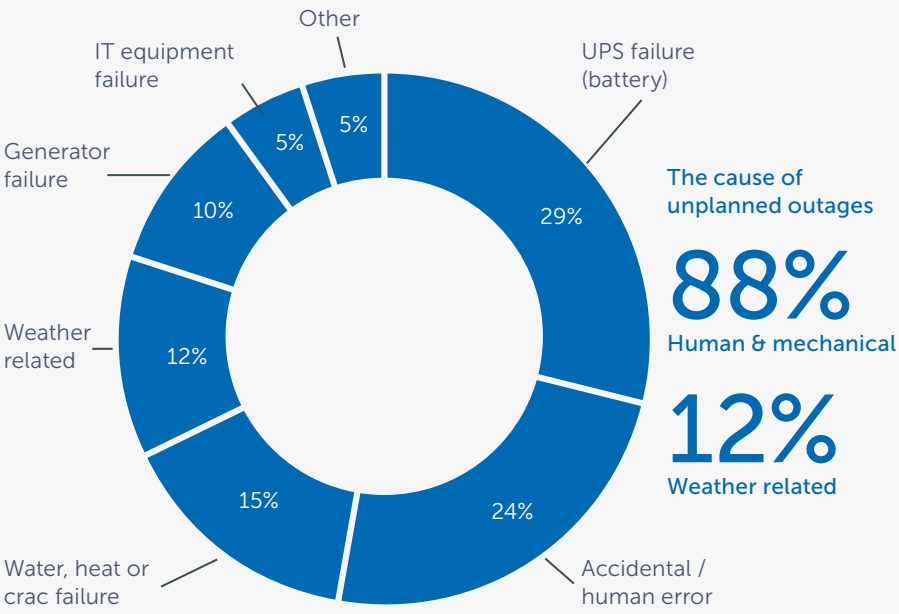
Both services have a guaranteed uptime of 99.95, so around 21 minutes of downtime are allowed per month. Our application needs both services to behave correctly in order to be fully reliable.

App Service: 99.95%

SQL Azure: 99.95%

Since these components are independent of each other, the App Service can be down on Monday from 06:00 to 06:20 and Azure SQL database can be down the ensuing day from 14:20 to 14:40. Because the services can have outages at different times, the compound SLA of multiple components is of course lower than their individual targets. Where they both satisfy their own reliability target, the overarching application may have a lower availability.

Then, all application components are communicating through the network of which we know that it is not always reliable. Starting to think of it, there are a lot of mechanical or human errors or natural disasters that may incur entire data centers outages. This means that our systems have an inherent risk of unavailability which has to be (and usually is) endorsed by developers, the business and its end-users. We should therefore expect that each modern application exhibits some degree of unreliability. But this does not have to stress us out. We will see that the impact of these (often short-lived) outages is smaller than commonly thought.



(https://www.365datacenters.com/portfolio-items/overcoming-causes-data-center-outages/)

There are architecture patterns that could be used to minimize user impact on certain issues.

Should we start paying significantly more on data redundancy offerings (like Geo-zone-redundant storages) to reduce our unreliability to an absolute minimum? We think that this should always be a business decision focusing on the business impact of certain failures.

Certain failures in our application can occur without the user being impacted, how important are these issues? We should not aim for perfection, but find the correlation between unreliability and user satisfaction. To find out, we should dive into the impact of failures – when does it actually matter?

Embracing the risk of failures

Aiming for higher reliability targets may seem like a reasonable (and ambitious) goal to pursue for product owners. We want to convey that setting higher targets is not always the right thing to do, and there may be high and concealed costs. If we want to improve the reliability of our system from 99.9% to 99.95%, and the application generates an annual revenue of €500.000, then a reasonable estimation of the additional revenue is only €250.

Moreover, there are many scenarios in which small unreliabilities do not bring about any noticeable consequences. When your LinkedIn feed is rendered incredibly slow, you probably press F5 and the problem is already over. There are many scenarios where neither economical nor user satisfaction factors run the risk of being drastically reduced. Let's focus on a warehouse example involving availability!

Example scenario: Warehouse solution

You're building software to handle all incoming orders that need to be collected in the warehouse by robots. This process is a key process within your business, so it should never be interrupted. If the robots stop working, trucks can't leave on time and customers won't be happy because their packages are late. So how reliable should things be? Our robots should never run out of work. This is a good business impact that we could measure. But what happens when communication to the robots fails?

Communication to the robots is super important, so our initial thoughts might be that we should do everything in our power to make this super reliable, but what if the robot can store up to 10 orders in advance? If each order takes about 30 seconds to complete, you have 5 full minutes before a robot runs out of work. So when looking at reliability, we should aim for a solution that focuses on achieving this business result instead of solely measuring which percentage of the messages to the robots were sent successfully.

It goes without saying that end-users care about reliability. However, we should form a realistic picture about which expectations customers have in mind about the application. When the effect of enhancing the reliability from three to four nines (99.9 to 99.99 percent) gets obfuscated by the unreliability of external factors (causing extra reliability to go *unnoticed*), then we can reasonably be reluctant to improve the reliability. Spending time on either rapid new feature development, lower latency or reducing accumulated technical debt would have been more fruitful for our end-users. The key things to monitor should be focused on user and business impact rather than technical errors.

Full reliability is overachieving, a single database failure is catastrophic and this uncertainty leads to imminent stress among your employees. There are ways to improve the reliability, but this should always be a conversation between the business owner and the engineering teams that build and manage the application.

Defining objectives that customers care about

Instead of focusing on overly high reliability targets, we should use our experience and common sense to contemplate which level of service we want to provide to our customers. The well-known service *level agreements* (SLAs) are backed up by *service level indicators* (SLIs) and *objectives* (SLOs).

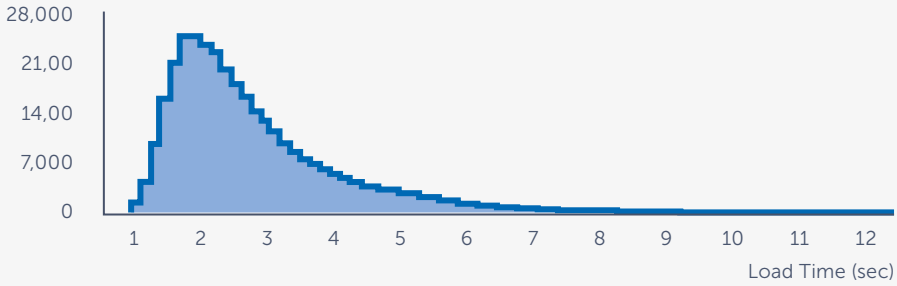
While any measurable quantity can be promoted to an indicator, we recommend choosing just a *few good probes*. These should encapsulate what users deem important in the application and it's usually a good idea to start working backwards from customer experience to SLIs rather than setting objectives based on

accumulated data. Google advocates that the most useful indicators of your system's health are: Latency, Traffic, Errors and Saturation, which they've coined *golden signals of monitoring*. What these indicators have in common is that all of them pertain to internal structures of your application.

The trick is to not lose ourselves in the anomalies of our internal system, but to get involved in the translation process to our end-users. For instance, the errors indicator does not always map directly on the user experience, but it's a fair bet that the following Service Level Indicators are strongly correlated with user satisfaction:

- *latency* (nobody wants to wait 2 seconds for each HTTP request);
- *availability* (2% downtime is simply too much);
- *throughput* (it shouldn't take too long to upload pictures);
- *correctness* (your shopping cart should show your selected items).

Now it's time to form objectives for these indicators to trace how much unreliability can reasonably be tolerated, and unsurprisingly, we'll look at the impact on customers. A frequently made mistake is to form objectives based on averages. This is problematic because distributions of our indicators are usually right-skewed, where the first 1% of the users have slightly better behavior, and the last 1% have incredibly slow responses of multiple seconds.



(Source: www.lognormal.com)

The risk of a far-reaching right tail is a valid concern, and in our experience it is useful to create objectives for high percentiles (e.g. 95%, 99% or even 99.9%) rather than for averages. This is based on the line of reasoning that every user has a good experience when the worst-case scenarios have reasonable experiences.

While the indicators are usually the same for different systems, objectives are where variation comes into play. Our objectives are reflected in the SLA and this sets expectations for the customers, and they may or may not select our service based on them.

Customers have different expectations about user-facing systems (webshops, social media) compared to archiving systems and, to name another, big data processing systems like Apache Spark.

SLI	SLO	SLA
Latency	95% of requests should be served within 100ms	within 150ms
	99% of requests should be served within 300ms	within 400ms
Availability	99.95% uptime per month	99.9% uptime per month

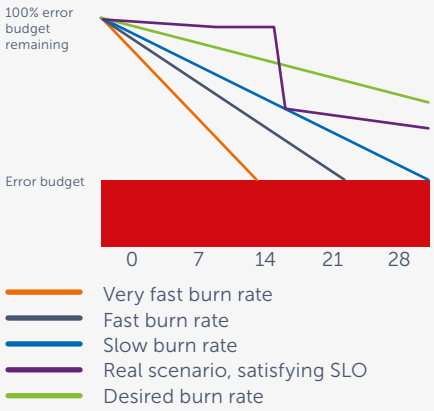
While end-users certainly care about these indicators, there is not always a trivial mapping function of these indicators to customer experience. What would provide more insight is to look at this through a functional angle, i.e., how do the application users experience the application? Spending excessive amounts of time on optimizing the reliability of the system means less time for rapid feature innovation, automation and experimentation.

A good example would be Outlook versus Youtube. The first application is used by businesses throughout the world for communication, and they expect the service to have a high availability. Youtube on the other hand is not used for critical purposes. While users may be somewhat annoyed by a lower uptime, this is probably outweighed by the positive experience of rapid bug fixes and new features. Google has also set lower reliability targets on Youtube versus Gmail for similar reasons.

At this point we should have a feeling for setting reasonable SLOs. What should we do next? The next step is to find a perfect balance between reliability and innovation!

Finding the sweet spot between reliability and innovation

Error budgets are based on the idea that a certain amount of unreliability is acceptable. For instance, Azure SQL strives for an availability target of 99.95%, which means that they are permitted to have a downtime of 21.6 minutes per month. These 21.6 minutes constitute their *monthly error budget*. Where traditionally outages would have been stressful events in need of immediate investigation, modern Site Reliability Engineering principles state that everything is under control as long as the error budget has not been burnt. Likewise, we can (and should!) form error budgets for microservices that we maintain.



This illustration shows the acceptable burn-rate (the blue line) and two unacceptably high burn rates (+25% and +50% slopes). On the other hand, the green line denotes a more positive trend: a burn-rate at which we would easily satisfy our objective. Now let's take a look at our real error budget burning - the purple line. During the first two weeks we're gradually spending a little bit of the error budget.

Then, due to a Daylight Saving bug on day 14, an excessive amount of error budget is spent, stopping at our *acceptable burn-down rate*. This means that we have to be more careful at this point, and we decide to reduce our release velocity. After a week (day 21) we notice that we're doing way better than the blue line, we can actually *embrace more* risk again.

Without an error budget, the business would have probably thought that we're not delivering enough value - the system was unreliable for quite a while! The change of philosophy with an error budget has noticeable advantages. Engineering teams can safely deploy and stay focused on what they were doing, and nobody is alerted when a fraction of the error budget is scorched. More intriguingly, we may even claim that most of the error budget should be used and this provides an excellent opportunity to experiment. Nowadays, it is even considered a best practice among Site Reliability Engineers to not aim for a significantly higher availability than our target, since this creates false expectations for the future.

The error budget reminds us that unreliability is not always undesirable. In fact, it even provides a *minimal amount of monthly downtime* (of course lower than the SLO target). If we haven't burnt any error budget, we simply haven't taken enough risks and customers will start to rely on their experience that the system is always reliable (which means they will be more bothered by future issues...)

Actionable metrics as a conversation between business & engineering

What we find a great benefit of Service Level Objectives and Error Budgets is that these create realistic expectations about the system that engineers and the product owner have agreed upon. Moreover, it removes a great deal of subjectivity out of any conversation on the application's health: we know exactly how much failure is permitted while keeping the end-users satisfied with the product

Aiming for overly high reliability targets has another issue: it is at odds with the desire for new features. Feature development is pretty dangerous from a reliability point of view:

- the complexity of the product increases with each new feature;
- in fact, each code change comes with implicit risks and changes the (assumed reliable) state of production.

While development teams are evaluated on their feature development velocity, tension can arise between business and engineering teams. An error budget is a very effective means to establish a balance between reliability and innovation. When the error budget is on track, we should not hesitate to develop and deploy. The system behaves as expected, and our end-users are certainly happy when new features become available quickly!

When we're on track for our objectives, we should feel encouraged to experiment. The risk of incurred unreliability is outweighed by the value

provided to users by our experiments (or to the development team by automating recurring tasks).

Let's take a look at new technologies. These often have the potential of adding a lot of value to the application, but they involve a risk for the reliability of the system. Let's give some concrete examples:

Potentially interesting experiment/improvement	Could provide value	Risk for reliability
More rapid feature development	– Users can get preview features faster	– Bugs are more easily introduced – Rollbacks
Migration from ARM to newer Infrastructure-as-Code frameworks	– Easier code changes – Better maintainability – Unit tests can be written for the infrastructure	– Not sure whether first deployments are successful – Downtime
Chaos engineering on production	– Expose vulnerabilities in the system – Rigorously test alerting scheme that is in place – Better understanding of strong and weak spots of our application reliability	– Chaos is invoked for a subset of the users – Reliability will certainly be lower than without chaos engineering tool

Being encouraged to experiment, we know when we are allowed to embrace risk for the greater good!

Getting started with reliability engineering in your application!

We have shown examples of SLOs raising expectations about the system's functioning and that loosening SLOs can have advantages. Error budgets help us to assess how much time we can spend on innovation versus improving reliability. Expecting full reliability comes at undesirable costs, of which we highlighted:

› **Increased stress among employees:**
- When the system is not behaving perfectly (even when nobody is using the application at that time).
- Resistance against feature development - what if something breaks on production?

› **Customer expectations:**
- Usually not as high as commonly thought.
- They tolerate occasional hiccups in the system.

› **The tradeoff between reliability and innovation:**
- The business and end-users not only care about reliability, but also about other aspects.
- Setting reasonable reliability targets allows us to make smart tradeoffs.
- Being on track for these targets means that we can embrace risks and experiment (to use new technologies, automate things, deploy faster et cetera), all of which may increase productivity and user satisfaction.

We hope that this article helps as a conversation starter for many organizations in order to make engineers and business work together in terms of thinking about reliability and how they can work together on making the right decisions for building an application that is as reliable as required. </>

Geert van der Cruijsen
Digital Kickstarter, Enabler for companies to embrace DevOps, Cloud & improve their engineering culture

xpirit.com/geert

Casper Dijkstra
Cloud Engineer

xpirit.com/casper

Xpirit embraces SPACE framework to measure developer productivity

Developed by Microsoft and GitHub and embraced by Xpirit, the new SPACE framework provides guidance to an industry challenge: measuring developer productivity. Why is a greater understanding of what affects software developers' productivity levels needed? Marcel de Vries, CTO of Xpirit, elaborates on the framework's usability and shares his view on productivity.

Author Marcel de Vries

Measuring output

Simply put, productivity indicates how efficiently you produce, usually measured by output. But is it that simple? In software development, this way of thinking has many pitfalls. For a long time, it hasn't been easy to measure productivity correctly. Does the number of tickets, lines of code, or deployments per day provide a good indication of how productive an individual is? Marcel doesn't think so: "If you look at how much code was written, that simply reflects how busy someone was, but it says nothing about the usefulness and quality of what was delivered. If someone solves a problem in ten lines instead of twenty, that can be considered as a more efficient solution, but is that really the case? And more importantly, you get what you measure. When people know they are measured by the number of deployments, they will deploy more often, but what do they deploy and does it solve the business need?"

> "Developer productivity is an elusive concept. You cannot think in terms of numbers. The only things you know, without diving in deep, are if the software solves the problem it's supposed to, and the time it took to get from idea to solution."
> – Marcel de Vries, CTO Xpirit

The importance of gaining insight

The reason that developer productivity is receiving more attention is twofold. On the one hand, the demand for software is greater than the supply. On the other hand, strict compliance and security requirements negatively impact productivity. To overcome these challenges, organizations require insight into the factors that affect productivity. Marcel adds: "The industry needs to change too. Instead of seeing software developers as an extra set of hands, I would like organizations to realize what we can contribute to the envisioning of business solutions. However, we also have a part in that. Developers often portray themselves as nerds who can't communicate. An image that doesn't fit the industry's current state and does our profession a disservice. We also need to change the stereotype. If you, as a developer, struggle to communicate, it's up to you to learn."

Taking a holistic approach

We cannot measure productivity with one single metric. The SPACE framework takes a much-needed holistic approach by using five factors: satisfaction, performance, activity, communication, and efficiency. Marcel elaborates: "Unlike DORA, which focuses on organizational indicators that show the success of DevOps, SPACE enables us to look at productivity from multiple dimensions that in relationship to each other can help us decode the actual productivity factors in your business context."

The first factor that SPACE addresses is satisfaction. Marcel elaborates: "Delivering software is a creative profession that requires a particular mindset. People must feel good about themselves, both in business and in life, to deliver. That's why, at Xpirit, we operate 'people first.' We pay attention to each others' wellbeing, learn from each other, voice our appreciation, and make sure we all feel safe in a group."

The second factor is performance. Instead of using this as a standalone metric, SPACE relates it to the other factors to produce a balanced outcome. Marcel: "The risk of measuring performance is that by measuring, you are already influencing productivity, which brings us back to the importance of making people feel safe. Additionally, you need to understand what you are measuring and if this is all-encompassing."

SPACE also focuses on communications and collaboration. According to Marcel, leadership has a significant role to play in this: "Leaders need to stimulate concepts like pair programming, mob programming and start create stable teams that get work done. We also need to move away from a traditional project-based approach where you create a temporary organization that the moment it becomes productive is destroyed, since the project is ended." On communication, Marcel laughingly says: "When software developers are asked to improve communication, our default is to build a new app or platform. That is unfortunately inherent to our profession, but of course not what we need."

Finding your flow

The pandemic helped us see the benefits of working online, such as more equality and less travel. But, it also introduced us to a new downside. The threshold to disturb someone while working lowered, making it more challenging to stay in your flow. Marcel believes we have to learn to switch off and go into focus mode. "It's ok not to answer your phone because you are busy writing code and stay in your flow! It is commonly known that task switching is the killer of productivity. To get back in your flow can take up to hours!"

When asked how to get into your flow, Marcel jokingly answers: "According to the Ballmer peak, a blood alcohol concentration between 0.129 and 0.138 % confers superhuman programming ability." He continues: "Being in the flow is different for everyone. From listening to music while coding, isolating yourself completely, or finding inspiration in an article. It's a unique state of mind in which your thoughts become code. You forget about eating, drinking, and time. All that matters is writing amazing code. If this only happens once every month, overall productivity might be considered low. Nonetheless, the quality of work you produce in your flow is unmatchable."

Xpirit and the SPACE framework

For a concluding reflection on productivity, Marcel cites his experience: "In our everyday work with customers, we come across many silos, which is not surprising, since operational excellence dictates dividing your business into departments. But, by doing that, you create delays in the process because you interrupt the flow. You can overcome that with Agile and DevOps. If you look at the measurements in the SPACE framework you can see that those ways of work can contribute significantly to higher productivity. The SPACE framework gives us an even better understanding of what factors increase or decrease productivity in your business. We use the framework to measure productivity, create dashboards and generate insights. We then observe your way of working, identify the indicators that influence productivity, and take an active, targeted approach to improve."

We'll elaborate on the SPACE framework in a series of blogposts, each covering one of the five dimensions during the coming weeks. Be sure to check the Xpirit website to stay tuned! </>

SPACE is the acronym for Satisfaction & well-being, Performance, Activity, Communication and collaboration, and Efficiency and flow. Each of these dimensions is key to understanding and measuring productivity, according to the researchers. For each of them, the framework suggests a number of distinct metrics that apply to different levels, including individual-, team- or group-, and system-level. Interestingly, SPACE does not advocate for using all of the metrics at once, rather to carefully select a reduced set of metrics that span across all three levels and capture different productivity dimensions. The full article outlining the SPACE framework has been published at https://queue.acm.org/.

Marcel de Vries
Chief Technical Officer

xpirit.com/marcel

www.xpirit.com

Together we drive change.

If you prefer the digital version of this magazine, please scan the qr-code.