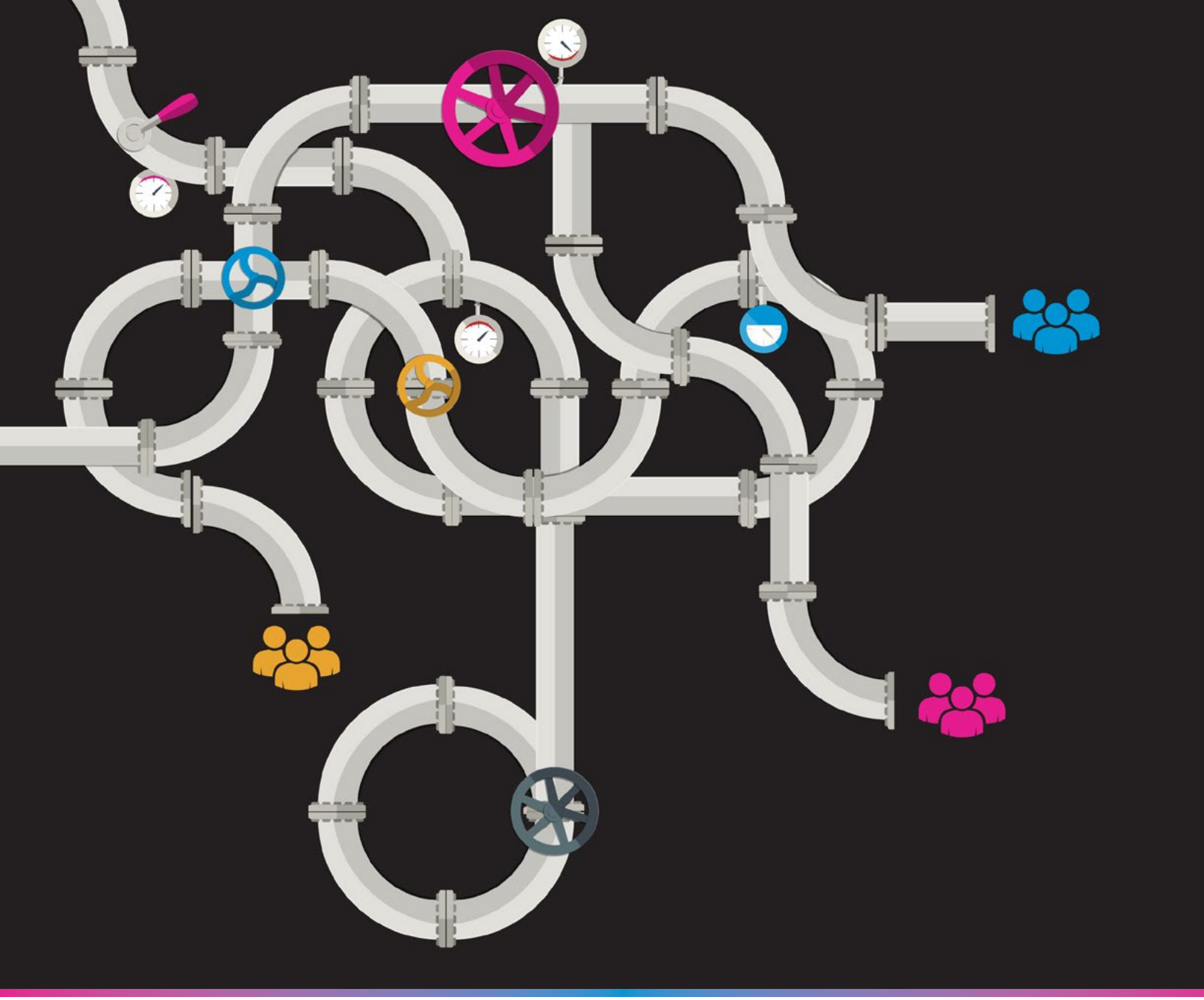


DevOps From the Trenches

Lessons learned from the DevOps community

sumologic

Table of Contents

Introduction 1

DevOps is a Strategy 3

Above all else, DevOps is a strategy - a strategic framework for transforming both development and operations into something new and not yet entirely defined.

Do Containers Become the DevOps Pipeline? 5

Containerization strips the virtual environment down to its essentials, providing a lightweight, fully-tailored container for the app. But what does that mean?

Change Management in a Change-Dominated World 8

DevOps isn't just about change–it's about continuous, automated change. Change management attempts to impose orderly processes on disorder.

Automated Testing in a DevOps World 11

The objective of automated testing is to simplify as much of the testing effort as possible with a minimum set of scripts.

Who Controls Docker Containers? 15

When it comes to Docker containers, disagreements can arise as to which department actually owns them: Development or Ops.

Introduction

Transparency Across DevOps Culture, Process and Toolchain

BY MICHAEL FLOYD

With DevOps currently at the peak of the hype curve according Gartner's Hype Cycle for Enterprise Architecture, 2015, there's little question that DevOps has become a highly charged, and some would argue, over-marketed term, giving rise to confusion and misunderstandings of what DevOps really is. Organizations in turn risk time, resources and money attempting to solve the wrong problem.

As you'll hear in the book, DevOps is a strategy for developing what is becoming a universally recognized process for managing continuous delivery, but it is not that process. Continuous Delivery (CD) is the process whereby software is developed iteratively and delivered in stages along a deployment pipeline. The benefits of CD for larger IT organizations are clear and widely embraced.

As you'll learn, DevOps is the culture, process and tools used in the CD cycle. From an agile perspective it facilitates teams to work harmoniously together with the goals of software quality and continuous improvement. The process, as just stated is continuous delivery, but the goals are similar: To deliver quality software more often while teams increase their velocity. That leaves tools.

The DevOps Toolchain

Tools used along the CD pipeline range from compilers, debuggers, and build tools to configuration management, monitoring and deployment tools. Depending on whether the platform is on-premise or in the cloud, tools can be categorized into team collaboration, application development, system level, and API-level tools. These tools include:

+ **Integrated Development Environments** - This can be Visual Studio on local workstation or a hosted development environment like salesforce.com's Force.com DE environment.

+ **Source control** - Repositories like Artifactory and Github provide an accessible, controlled means for storing scripts, code and other key assets.

+ **Continuous integration servers** - CI servers like Jenkins merge code and automate the build, test, and commit stages, providing early feedback.

+ **Configuration management** - Tools like Puppet and Chef help maintain state and consistency between staging and production environments.

+ **Issue tracking** - These tools can help improve responsiveness and visibility.

+ **Monitoring tools** - Provide clear, shared responsibility for relevant parts of service health.

+ **Deployment tools -** For building out environments and regularly updating systems.

+ **Collaboration and Planning -** Tools like Kanban provide project management while offering transparency into the process.

Troubleshooting Code in a DevOps World

While increasing code coverage for unit tests and automated testing have greatly improved the quality of software, clean code doesn't mean software always behaves as expected. A faulty algorithm or failure to account for unforeseen conditions can cause software to behave unpredictably. Within the CD pipeline, troubleshooting can be difficult, and in cases like debugging in a production environment it may not even be possible.

The good news is that virtually all of the tools across the CD pipeline emit machine data, which are typically captured in log files. At the applications level, logs capture the stream of events of your app's running processes including events generated by the application server and libraries. System logs capture events such as restarting a crashed process, and API logs can provide useful information when deploying new code.

Sumo Logic provides developers and others on DevOps teams with visibility into critical issues across the DevOps tool chain using event logs. Gone are the days of setting breakpoints in a debugger and stepping through code. Once a collector is configured for your app, logs are fed through the Sumo Logic service. Sumo Logic delivers out-of-the box dashboards, reports, saved searches, and field extraction for popular data sources. When an app is installed in Sumo Logic, these pre-set searches and dashboards are customized with source configurations and populated in a folder selected by the user.

Through Sumo Logic's LogReduce pattern-matching algorithm selects patterns as well as outliers that help developers quickly spot patterns in event logs, CI server logs and troubleshoot their applications in real time. So they can spend less time troubleshooting and more time developing code.

Sumo Logic address five common use cases:

+ **Increase availability and performance.** Sumo Logic enables issues to be identified before they impact the application and customer. Precise, proactive analytics quickly uncover hidden root causes across all layers of the application and infrastructure stack.

+ **Provide real time insights.** With Sumo Logic DevOps teams can easily extract machine data insights to provide greater intelligence around their customers, products, and application usage. These insights provide a more accurate and complete analysis for business users.

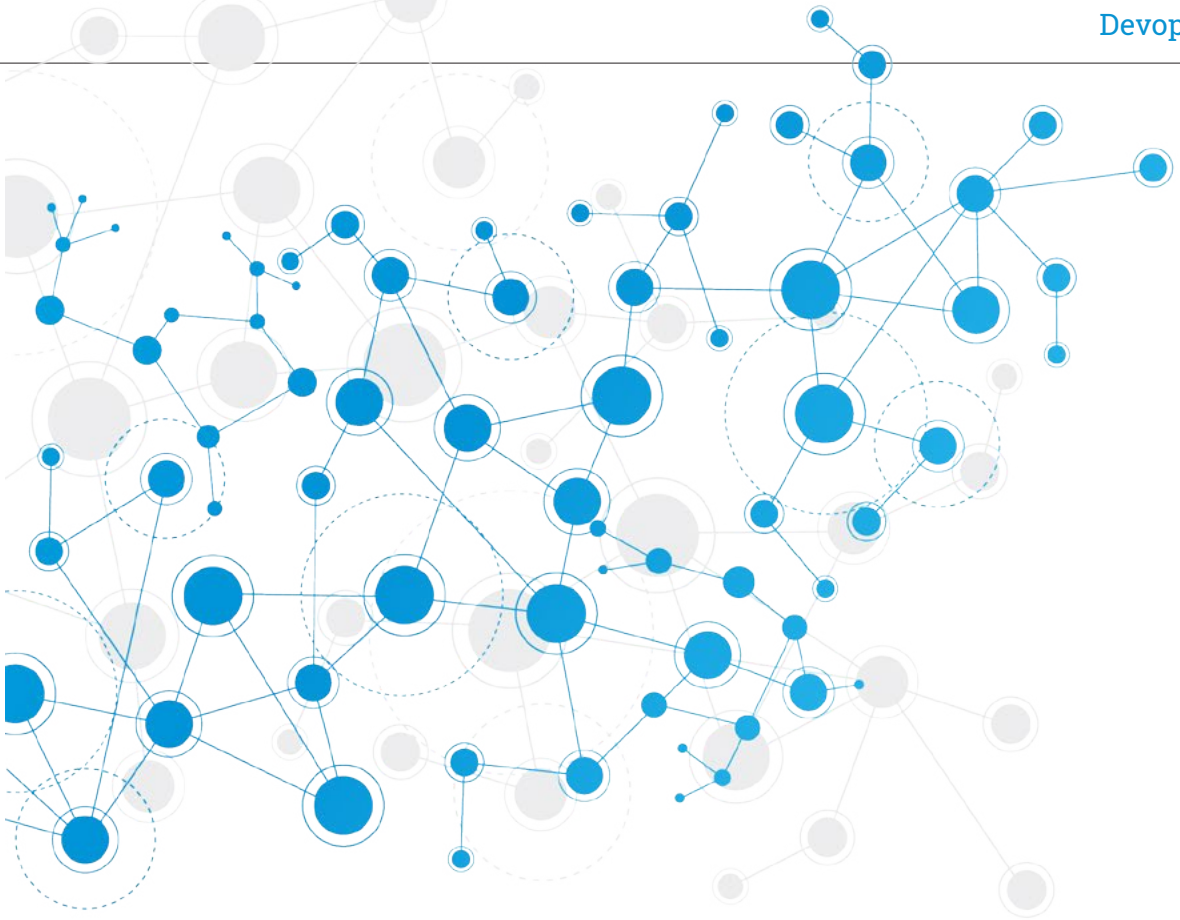
+ **Accelerate cloud deployment.** Sumo Logic enables DevOps to automate and speed the development and deployment process for cloud-based applications. Companies can rapidly detect, identify and resolve application issues.

+ **Decrease app time to market.** With Sumo Logic, DevOps teams can implement a consistent release process resulting in on-time releases. They can easily identify application issues and configuration changes across development, test and deployment environments.

+ **Enforce compliance.** Sumo Logic delivers a simple, proactive and automated process to audit and investigate operational, security and regulatory compliance incidents. All data is centralized, secured, and easily analyzed in real-time through a single, highly scalable solution.

Summary

The following articles in this eBook first appeared on devops.sumologic.com, and are written by members of the Sumo Logic DevOps community. This eBook looks not only at DevOps, but technologies like containerization that are changing how DevOps teams configure environments for deployment. The opinions are theirs: We hope you'll find useful insights to problems DevOps teams are challenged with today.



DevOps is a Strategy

BY MICHAEL CHURCHMAN

What is DevOps, anyway? There are dozens of definitions, of course, but sometimes you need to go beyond definitions and look at how people really use a word. What do people in the industry really mean when they talk about DevOps? Is it simply a way of moving developers into operations, of making them full participants in the operations team and in management of infrastructure?

If you look at DevOps that way, you'll have plenty of company, and you won't really be wrong. Functionally, operations and the demands of infrastructure are the main drivers of DevOps. DevOps brings development into the operations tent and applies development practices to infrastructure, but it doesn't move operations out of the tent.

But DevOps is far more than just a functional framework. If that's all it were, both its supporters and its critics would treat it as a package of technical and temporary fixes for slow and inefficient development/release cycles, and not much else. While technical remedies are part of DevOps, and improving the speed and efficiency of the the development/release cycle is one of its major goals, a definition based strictly on those factors hardly seems adequate. Just a glance at the online discussion of DevOps would be enough to convince even a casual observer that there is far more to the subject.

In fact, the specific tools and practices which are currently associated with DevOps are largely incidental, and not intrinsic to it. Some scripting languages may be superior to others, but in a pinch, most (and maybe any) of them will be adequate. Virtualization and containerization matter, but the methods used to wrap an application and insulate it from its environment may not really be that important in the long run. The only thing that you can really count on is that today's tools are likely to be obsolete in a few years.

So what is DevOps, if it isn't a clearly defined set of tools or practices? Is it anything at all, besides a grab-bag collection of contemporary software management products and catchphrases?

Yes, it is; above all else, DevOps is a strategy. More accurately, it is a strategic framework for transforming both development and operations into something new and not yet entirely defined. Basic to this transformation is the fundamental redefinition not only of development and operations, but of their underlying context. What are we developing, and what are we operating, and for what base of users? If today we are producing a stream of services aimed at a mass-market user base, can we afford to define what we are doing in terms more appropriate to an era when software consisted of clearly-defined packages running on corporate networks, and when waterfall development was an adequate methodology?

In many ways, the basic problem that gave rise to DevOps is the mere fact that software development has a history. Like any other human institution or endeavor, its history has become a kind of cage, trapping it in the definitions, categories, and practices of the past. So much that we take for granted — from the names and formats of low-level code commands to the institutional environments in which development and operations are embedded — are artifacts of that history, and not fundamental to the practice of producing, deploying, and maintaining software.

If a group of intelligent people with access to current hardware and basic programming tools, but with absolutely no knowledge of software history or of past or current development and operations practices, were to devise a set of tools and practices for creating, deploying, and maintaining web-based services, what would they come up with? Would they define what they were doing or producing in terms and categories that bore any relation at all to those which exist now?

> Above all else, DevOps is a strategy - a strategic framework for transforming both development and operations into something new and not yet entirely defined.

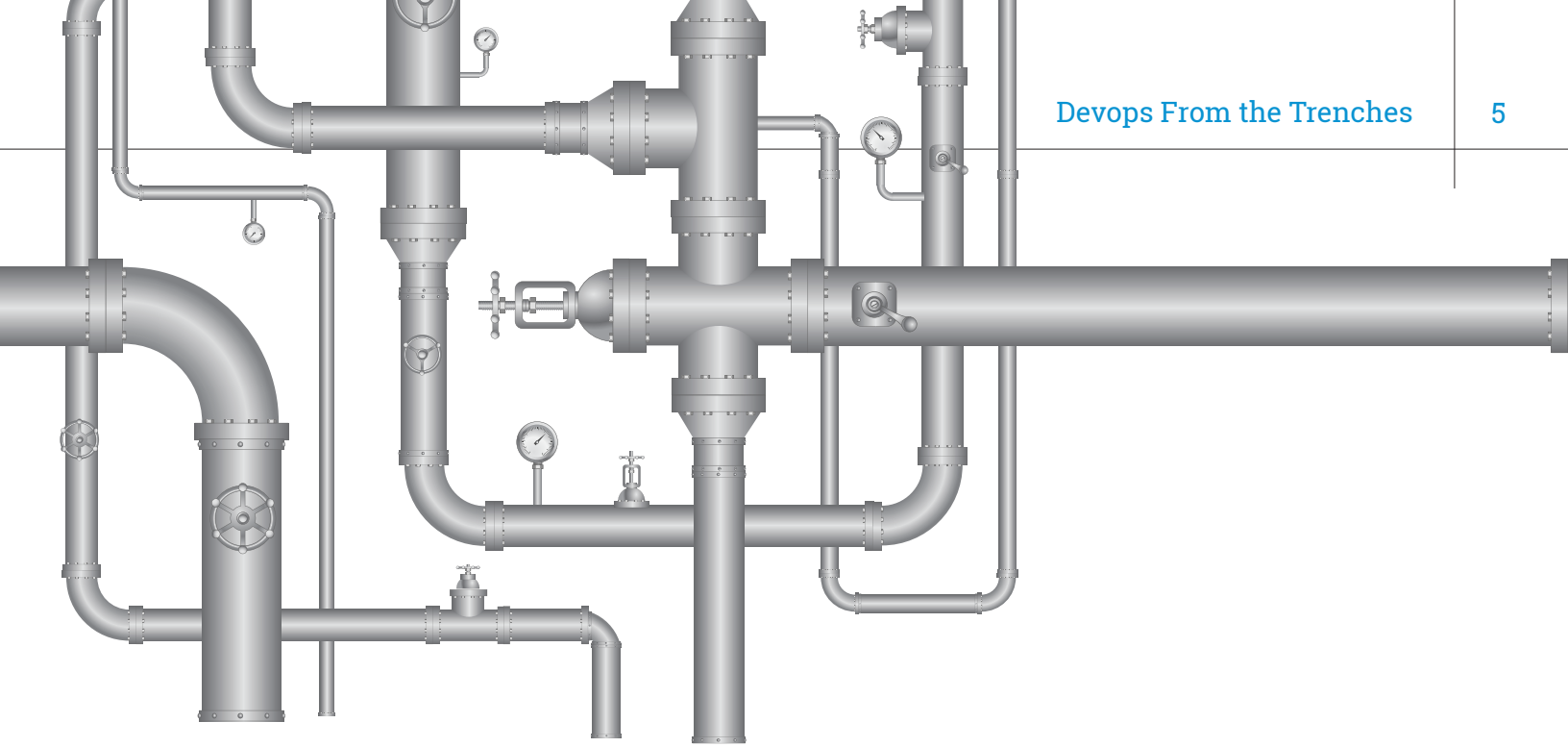
Devops Strategy

DevOps can be seen as a strategic framework for finding answers to these questions without actually resorting to Ren and Stimpy's History Eraser button. It starts by redefining the basic task at hand as "providing a stream of web-based services to the general public," as opposed to the more traditional "developing computer/network-based applications for a set of end-users, with both the applications and end-users defined by a set of functional requirements." Theoretically, everything else in DevOps could flow from that redefinition. In practice, DevOps is a strategy for discovering and implementing the practices that naturally arise from the "stream of web-based services" approach.

As a strategy, its focus is necessarily going to be on people even more than it is on tools or methods, because it is people who actually carry out strategies; a hammer is a good tool, but the hand that holds the hammer is what determines where and when it is used. The core of the DevOps people strategy must be to form a single team in which members take on a variety of roles, depending on what needs to be done. Developers who write infrastructure code should ideally not feel as if they are crossing some kind of boundary or temporarily working in "the other silo", but rather as if they are working as part of an integrated team. IT staff should see deployment as part of the continuous stream that includes development and testing.

And as a strategy, DevOps' use of tools and methods is necessarily going to be focused on two goals: supporting continuous release and the associated processes, and making it possible for team members to do their jobs in a manner that is as efficient and painless as possible.

Strategies are not end-products, and they are not mature, stable processes. They are by nature interim solutions on the way to the next plateau of semi-stability (because in the software industry, there is no true stability). DevOps is no different; it is a strategy for developing what will become the mature and universally recognized process for managing continuous delivery, but it is not that process, because we have not yet arrived at the plateau where continuous delivery will be as ubiquitous, as stable (and as boring) as waterfall once was. DevOps is the journey, and not the destination.



Do Containers Become the DevOps Pipeline?

BY CHRIS RILEY

The DevOps pipeline is a wonderful thing, isn't it? It streamlines the develop-and-release process to the point where you can (at least in theory, and often enough in practice) pour fresh code in one end, and get a bright, shiny new release out of the other. What more could you want?

That's a trick question, of course. In the contemporary world of software development, nothing is the be-all, end-all, definitive way of getting anything done. Any process, any methodology, no matter how much it offers, is really an interim step; a stopover on the way to something more useful or more relevant to the needs of the moment (or of the moment-after-next). And that includes the pipeline.

Consider for a moment what the software release pipeline is, and what it isn't. It is an automated, script-directed implementation of the post-coding phases of the code-test-release cycle. It uses a set of scriptable tools to do the work, so that the process does not require hands-on human intervention or supervision. It is not any of the specific tools, scripting systems, methodologies, architectures, or management philosophies which may be involved in specific versions of the pipeline.

Elements of the present-day release pipeline have been around for some time. Fully automated integrate-and-release systems were not uncommon (even at smaller software companies) by the mid-90s. The details may have been different (DOS batch files, output to floppy disks), and many of the current tools such as those for automated testing and virtualization were not yet available, but the release scripts and the tasks that they managed were at times complex and sophisticated. Within the limited scope that was then available, such scripts functioned as segments of a still-incomplete pipeline.

Today's pipeline has expanded to engulf the build and test phases, and it incorporates functions which at times have a profound effect on the nature of the release process itself, such as virtualization. Virtualization and containerization fundamentally alter the relationship between software (and along with it, the release process) and the environment. By wrapping the application in a mini-environment that is completely tailored to its requirements, virtualization eliminates the need to tailor the software to the environment. It also removes the need to provide supporting files to act as buffers or bridges between it and the environment. As long as the virtualized system itself is sufficiently portable, multi-platform release shrinks from being a major (and complex) issue to a near-triviality.

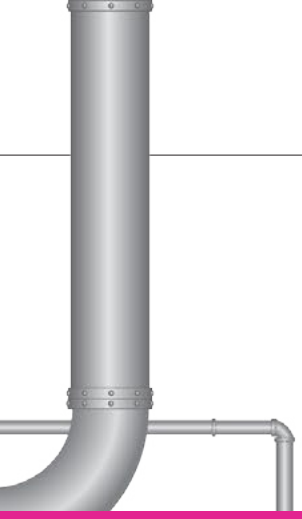
> "Virtualization and containerization fundamentally alter the relationship between software (and along with it, the release process) and the environment.

Containerization carries its own (and equally clear) logic. If virtualization makes the pipeline work more smoothly, then stripping the virtual environment down to only those essentials required by the software will streamline the process even more, by providing a lightweight, fully-tailored container for the application. It is virtualization without the extra baggage and added weight.

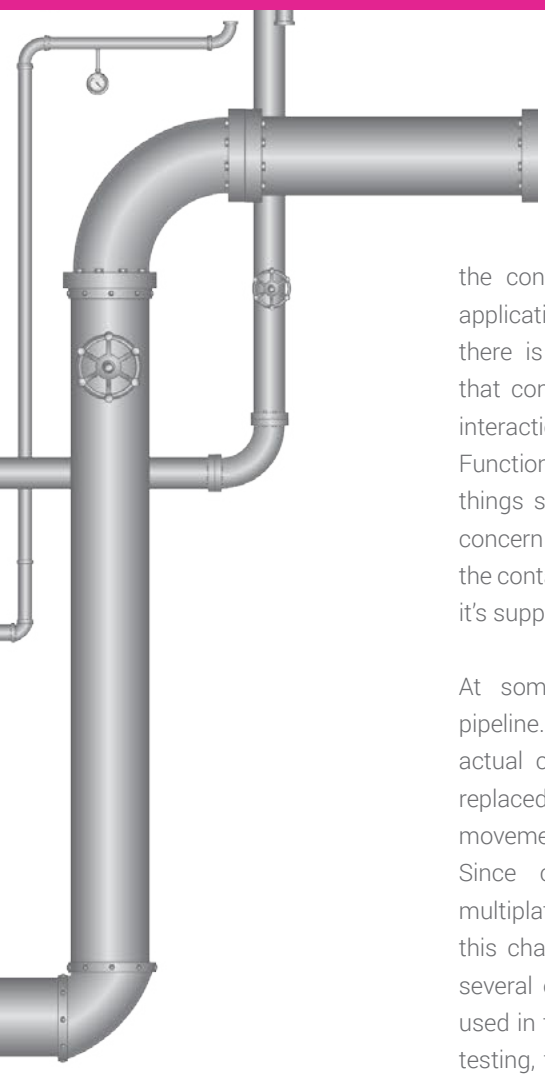
Once you start stripping nonessentials out of the virtual environment, the question naturally arises — what does a container really need to be? Is it better to look at it as a stripped-down virtual box, or something more like a form-fitting functional skin? The more that a container resembles a skin, the more that it can be regarded as a standardized layer insulating the application from (and integrating it into) the environment. At some point, it will simply become the software's outer skin, rather than something wrapped around it or added to it.

Containerization in the Pipeline

What does this mean for the pipeline? For one thing, containerization itself tends to push development toward the microservices model; with a fully containerized pipeline, individual components and services become modularized to the point where they can be viewed as discrete components for purposes such as debugging or analyzing a program's architecture. The model shifts all the way over from the "software as tangle of interlocking code" end of the spectrum to the "discrete modules with easily identifiable points of contact" end. Integration-and-test becomes largely a matter of testing the relationship and interactions of containerized modules.



> In fact, if all of the code moving through the pipeline is containerized, then management of the pipeline naturally becomes management of the containers.

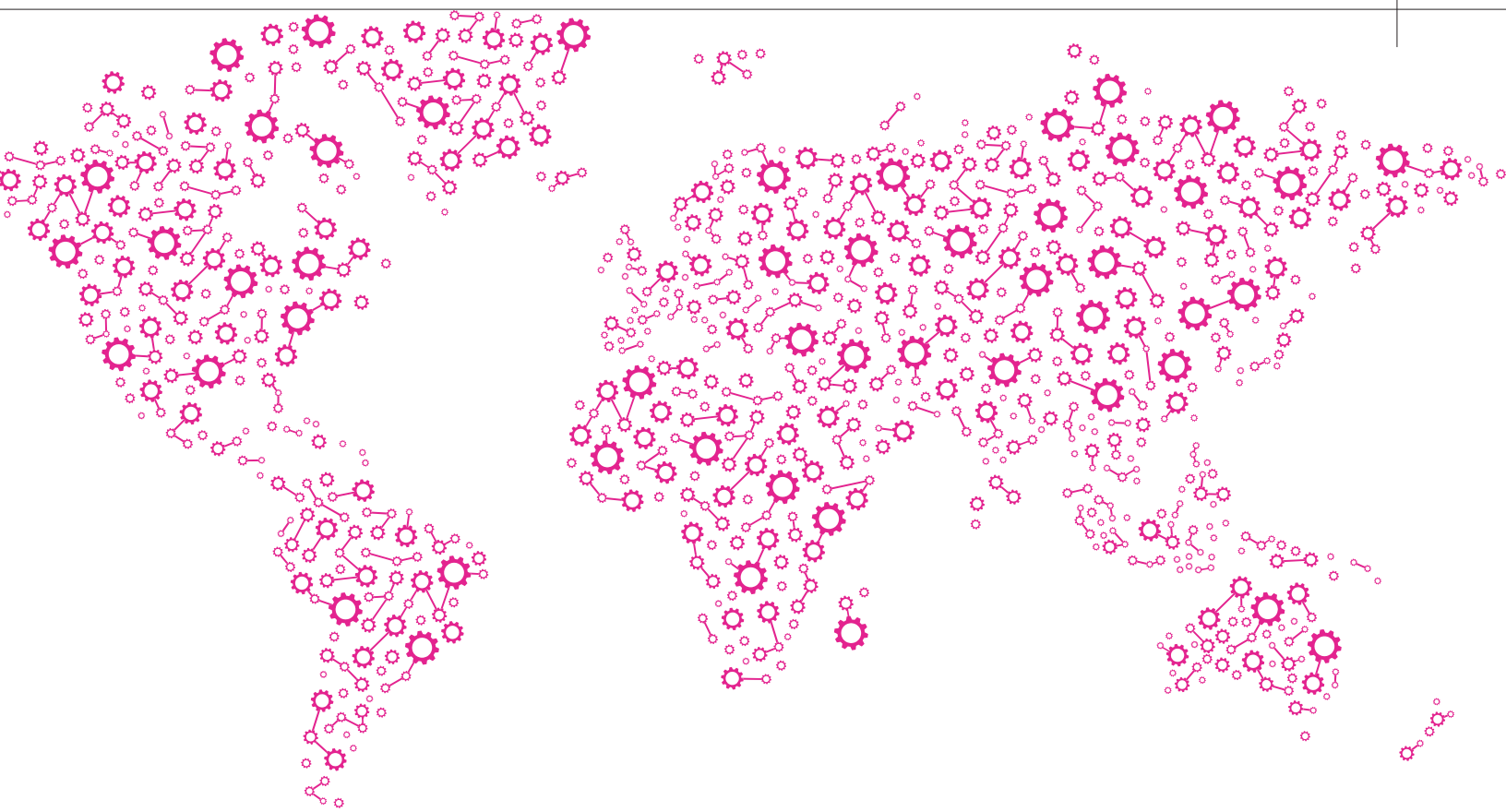


In fact, if all of the code moving through the pipeline is containerized, then management of the pipeline naturally becomes management of the containers. If the container mediates the application's interaction with its environment, there is very little point in having the scripts that control the pipeline directly address those interactions on the level of the application's code. Functional testing of the code can focus on things such as expected outputs, with minimal concern about environmental factors. If what's in the container does what it's supposed to do when it's supposed to do it, that's all you need to know.

At some point, two things happen to the pipeline. First, the scripts that constitute the actual control system for the pipeline can be replaced by a single script that controls the movement and interactions of the containers. Since containerization effectively eliminates multiplatform problems and special cases, this change alone may simplify the pipeline by several orders of magnitude. Second, the tools used in the pipeline (for integration or functional testing, for example) will become more focused on the containers, and on the applications as they operate from within the containers.

When this happens the containers become, if not the pipeline itself, then at least the driving factor that determines the nature of the pipeline. A pipeline of this sort will not need to take care of many of the issues handled by more traditional pipelines, since the containers themselves will handle them. To the degree that containers do take over functions previously handled by pipeline scripts (such as adaptation for specific platforms), they will then become the pipeline, while the pipeline becomes a means of orchestrating the containers.

None of this should really be surprising. The pipeline was originally developed to automatically manage release in a world without containerization, where the often tricky relationship between an application and the multiple platforms on which it had to run was a major concern. Containerization, in turn, was developed in response to the opportunities that were made possible by the pipeline. It's only natural that containers should then remake the pipeline in their own image, so that the distinction between the two begins to fade.



Change Management in a Change-Dominated World

BY CHRIS RILEY

DevOps isn't just about change — it's about continuous, automated change. It's about ongoing stakeholder input and shifting requirements; about rapid response and fluid priorities. In such a change-dominated world, how can the concept of change management mean anything?

But maybe that's the wrong question. Maybe a better question would be this: Can a change-dominated world even exist without some kind of built-in change management?

Change management is always an attempt to impose orderly processes on disorder. That, at least, doesn't change. What does change is the nature and the scope of the disorder, and the nature and the scope of the processes that must be imposed on it. This is what makes the DevOps world look so different, and appear to be so alien to any kind of recognizable change management.

Traditional change management, after all, seems inseparable from waterfall and other traditional development methodologies. You determine which changes will be part of a project, you schedule them, and there they are on a Gantt chart, each one following its predecessor in proper order. Your job is as much to keep out ad-hoc chaos as it is to manage the changes in the project.

And in many ways, Agile change management is a more fluid and responsive version of traditional change management, scaled down from project-level to iteration-level, with a shifting stack of priorities replacing the Gantt chart. Change management's role is to determine if and when there is a reason why a task should move higher or lower in the priority stack, but not to freeze priorities (as would have happened in the initial stages of a waterfall project). Agile change management is priority management as much as it is change management — but it still serves as a barrier against the disorder of ad-hoc decision-making.

In Agile, the actual processes involved in managing changes and priorities are still in human hands and are based on human decisions. DevOps moves many of those management processes out of human hands and places them under automated control. Is it still possible to manage changes or even maintain control over priorities in an environment where much of the on-the-ground decision-making is automated?

Consider what automation actually is in DevOps — it's the transfer of human management policies, decision-making, and functional processes to an automatically operating computer-based system. You move the responsibilities that can be implemented in an algorithm over to the automated system, leaving the DevOps team free to deal with the items that need actual, hands-on human attention.

This immediately suggests what naturally tends to happen with change management in DevOps. It splits into two forks, each of which is important to the overall DevOps effort. One fork consists of change management as implemented in the automates continuous release system, while the other fork consists of human-directed change management of the somewhat more traditional kind. Each of these requires first-rate change management expertise on an ongoing basis.

> DevOps moves many of those management processes out of human hands and places them under automated control.

It isn't hard to see why an automated continuous release system that incorporates change management features would require the involvement of human change management experts during its initial design and implementation phases. Since the release system is supposed to incorporate human expertise, it naturally needs expert input at some point during its design. Input from experienced change managers (particularly those with a good understanding of the system being developed) can be extremely important during the early design phases of an automated continuous release system; you are in effect building their knowledge into the structure of the system.

But DevOps continuous release is by its very nature likely to be a continually changing process itself, which means that the automation software that directs it is going to be in a continual state of change. This continual flux will include the expertise that is embodied in the system, which means that its frequent revision and redesign will require input from human change management experts.

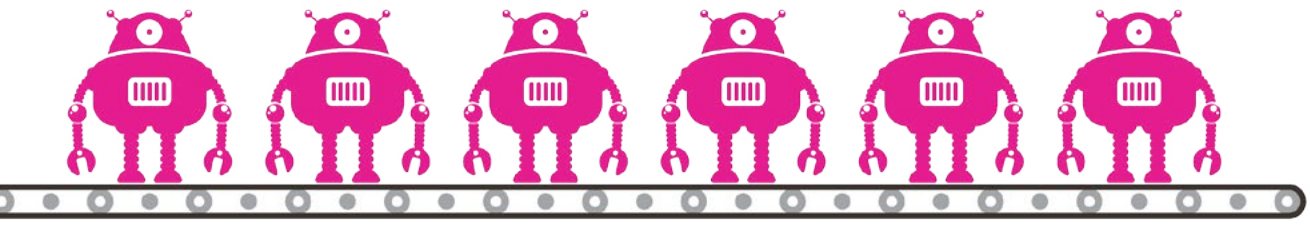
> But DevOps continuous release is by its very nature likely to be a continually changing process itself, which means that the automation software that directs it is going to be in a continual state of change.

And not all management duties can be automated. After human managers have been relieved of all of the responsibilities that can be automated, they are left with the ones that for one reason or another do not lend themselves well to automation — in essence, anything that can't be easily turned into an algorithm. This is likely to include at least some (and possibly many) of the kinds of decision that fall under the heading of change management. These unautomated responsibilities will require someone (or several people) to take the role of change manager.

And DevOps change management generally does not take its cue from waterfall in the first place. It is more likely to be a lineal descendant of Agile change management, with its emphasis on managing a flexible stack of priorities during the course of an iteration, and not a static list of requirements that must be included in the project. This kind of priority-balancing requires more human involvement than does waterfall's static list, which means that Agile-style change management is likely to result in a greater degree of unautomated change management than one would find with waterfall.

This shouldn't be surprising. As the more repetitive, time-consuming, and generally uninteresting tasks in any system are automated, it leaves greater time for complex and demanding tasks involving analysis and decision-making. This in turn makes it easier to implement methodologies which might not be practical in a less automated environment. In other words, human-based change management will now focus on managing shifting priorities and stakeholder demands, not because it has to, but because it can.

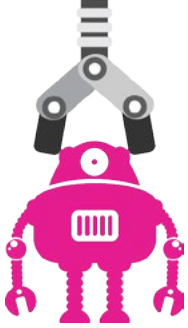
So what place does change management have in a change-dominated world? It transforms itself from being a relatively static discipline imposed on an inherently slow process (waterfall development) to an intrinsic (and dynamic) part of the change-driven environment itself. DevOps change management manages change from within the machinery of the system itself, while at the same time allowing greater latitude for human guidance of the flow of change in response to the shifting requirements imposed by that change-driven environment. To manage change in a change-dominated world, one becomes the change.



Automated Testing in a DevOps World

BY GREG SYPOLT

The objective of automated testing is to simplify as much of the testing effort as possible with a minimum set of scripts. Automated testing tools are capable of executing repeatable tests, reporting outcomes, and comparing results with faster feedback to the team. Automated tests perform precisely the same operation each time they are executed, thereby eliminating human errors – and can be run repeatedly, at any time of day. Below I've outline five steps how to get up and running (the right way) with automation.



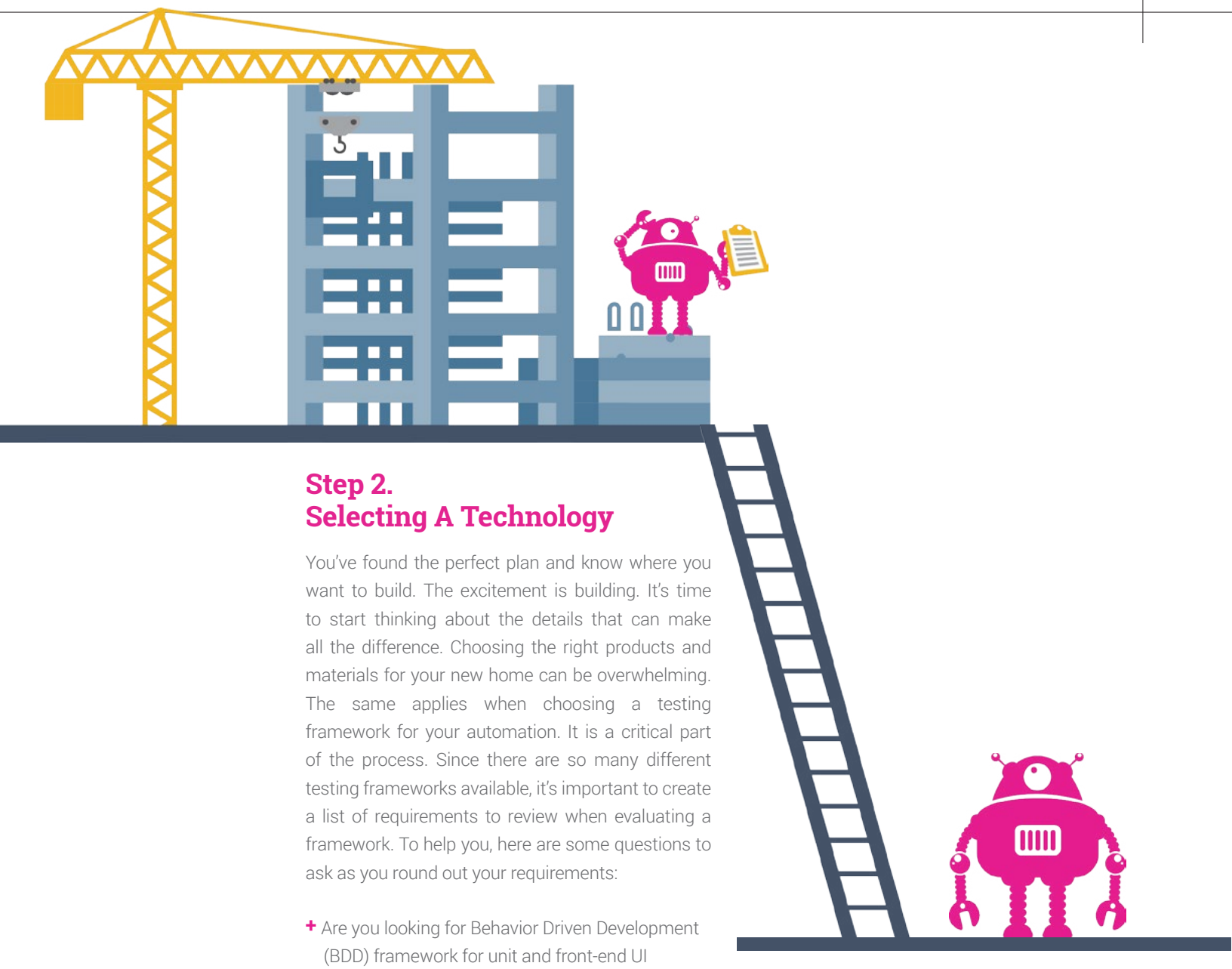
Step 1. Laying the Foundation

The foundation is the most important element of any building, be it a house or a high-rise. It may seem like a simple part of the overall construction process, but getting the foundation right is incredibly important. Mistakes made in the foundation will only get worse as you go up. It's known as compounding defects and it means that mistakes grow. Wait! How does this relate to automation? Proper planning will lay a solid automation foundation for project success; without it your project will be shaky and a maintenance nightmare.

To avoid these potential pitfalls and keep on task, you need a good road map – start by:

+ Planning your automation objective;
+ Designing the architecture;
+ Training the team;
+ Begin developing test scripts;
+ Releasing to the wild.

Throughout the onboarding process, it is important educate everyone. One of the common misconceptions of automation is that automation is a magic bullet – the initial setup and step creation will take time and effort. Automation requires effort to maintain. This needs to be factored into any planning and estimation. The principles of good programming are closely related to principles of good design and engineering. By enforcing standards, you can help developers become more efficient and to produce code which is easier to maintain with fewer defects. It's a great opportunity to start shaping your new testing portfolio by educating everyone – they need to understand what type of testing belongs at unit, integration, and API layers when having those conversations during the sprint planning phase.



Step 2.
Selecting A Technology

You've found the perfect plan and know where you want to build. The excitement is building. It's time to start thinking about the details that can make all the difference. Choosing the right products and materials for your new home can be overwhelming. The same applies when choosing a testing framework for your automation. It is a critical part of the process. Since there are so many different testing frameworks available, it's important to create a list of requirements to review when evaluating a framework. To help you, here are some questions to ask as you round out your requirements:

+ Are you looking for Behavior Driven Development (BDD) framework for unit and front-end UI testing?
+ Do you need to support browsers, mobile web, or mobile native apps?
+ Are you looking to run your selenium test locally or in the cloud?
+ Do you need cross-browser testing?
+ Do you need keyword driven framework for non-technical resources?
+ Does your team have sufficient programming knowledge for automation development?
+ Are you practicing continuous integration and need a tool that integrates seamlessly?

Step 3.
Configuration

Find the right pro who specializes in exactly the type of work you need done. You would never hire a plumber to do electrical work. Right?

This stage is critical and can be frustrating with a lack of experience. It would be ideal to find an automation expert to design your automation architecture, teach the team the fundamental skills how to write quality tests, and continuous mentoring. If hiring an expert is not an option, I strongly suggest finding an on-site training course for everyone planning to write tests.

Step 4. The Basics

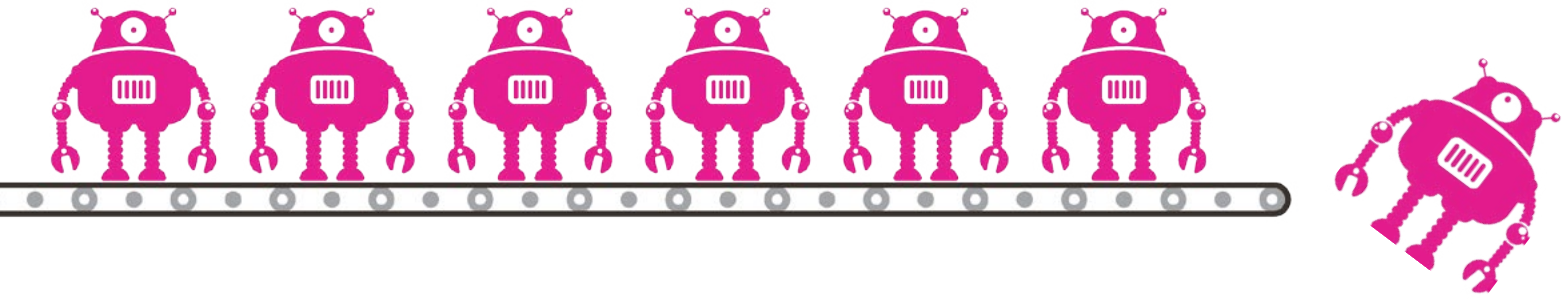
Every construction trade require basic knowledge of the service. I couldn't even imagine building interior framing wall without any basic knowledge. The framing basics of your wall include the sill plate on the bottom, the wall studs (which are the vertical beams), and the top sill plate (which is the beam running across the top). It sounds simple in theory, but to get a wall plumb and square takes basic knowledge and a lot of practice.

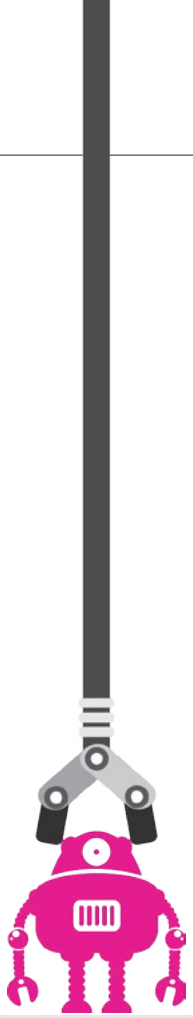
HTML code with well defined locators:

```
<div class="grid location-search-result js-store js-search-result" data-showclosed="true"
data-substituted="false" data-delivery="false" data-carryout="true" data-online="true"
data-orderable="Carryout Delivery" data-open="true" data-type="Carryout" data-
storeid="4348" data-services="Carryout">

    <a class="js-orderCarryoutNow js-carryoutAvailable btn btn--block" data-
    type="Carryout" data-ordertiming="current" href="#/section/Food/category/
    AllEntrees/">Order Carryout</a>

</div>
```

Write a script using Capybara testing framework:

```
expect(page).to have_css('.js-orderCarryoutNow')
find('.js-orderCarryoutNow').click
```





Step 5. Let's Code

FINALLY, the construction begins! Here are my tips for getting started with writing great automation scripts.

+ **Repeatable.** Automated scripts must be repeatable. You must be able to measure an expected outcome.

+ **Design Tests for Scalability.** The whole point of automation is to provide rapid feedback. Creating and executing large-scale automated tests need a thoughtful approach to elements in the process and a close eye on test design. Keep tests lean and independent.

+ **Easy to Understand.** Scripts should be readable. Ideally, your scripts also serve as a useful form of design and requirement documentation.

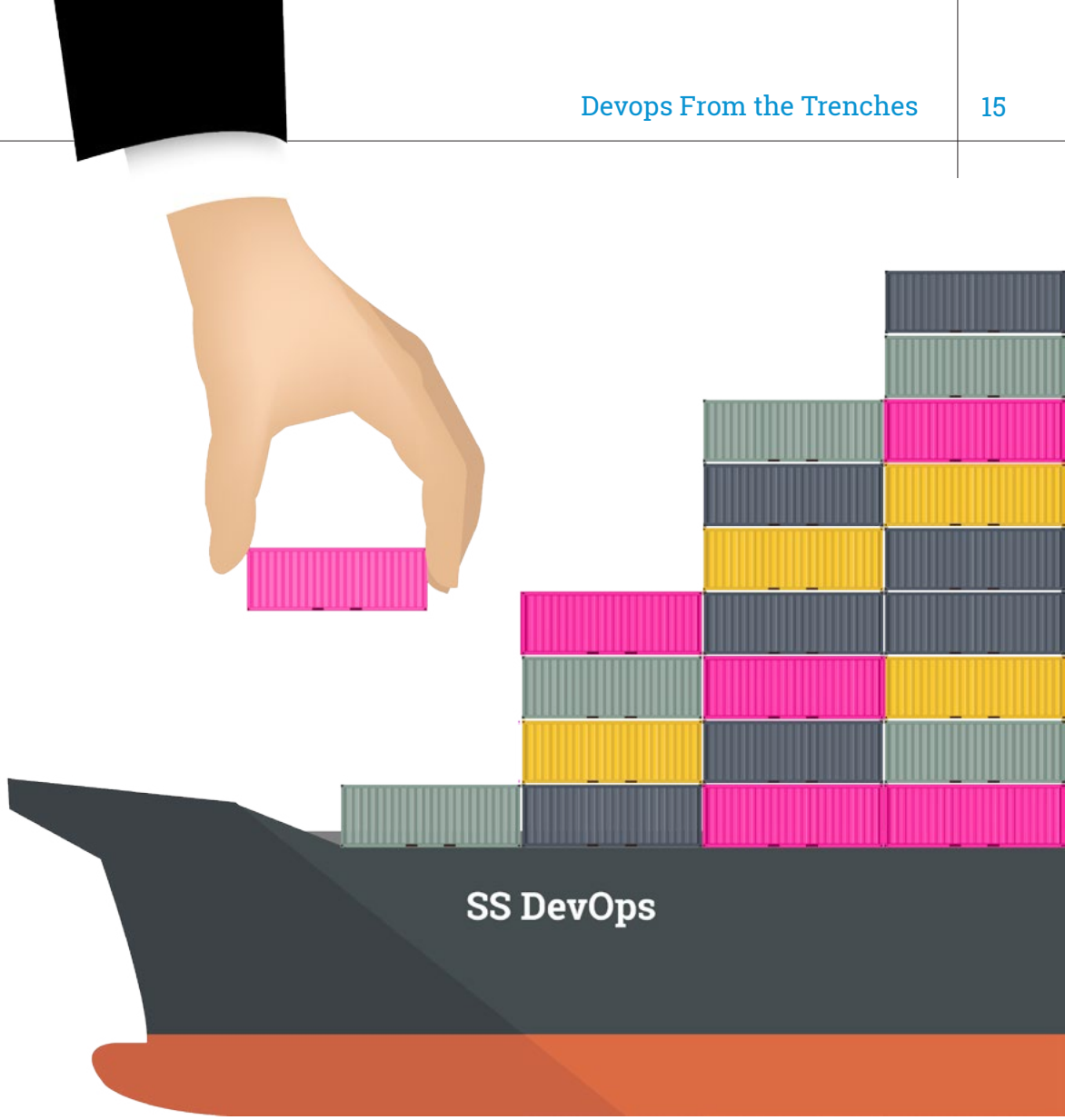
+ **Speed.** Your automated tests should run quickly. The end goal is to make the overall development process faster by establishing rapid feedback cycles to detect problems.

Write a script using Capybara testing framework:

```
describe "on the google homepage -", :google, :type => :feature  do

  before :each do
    visit 'https://google.com/'
  end

  it "the search input field is present" do
    expect(page).to have_css('input#lst-ib.gsfi')
  end
end
```

Takeaways

Your road to automation can be difficult, but it's worth it. To do it well you need the right attitude. Sometimes you will hit roadblocks. If something didn't work, you really learn to problem solve here. The training phase is so important and is an incredible opportunity to master your new craft – pay attention, have the right attitude, adapt, be engaged, and never stop learning...



Who Controls Docker Containers?

BY ZACHARY FLOWER

It's no secret that Development ("Dev") and Operations ("Ops") departments have a tendency to butt heads. The most common point of contention between these two departments is ownership. Traditionally Ops owns and manages everything that isn't direct development, such as systems administration, systems engineering, database administration, security, networking, and various other subdisciplines. On the flipside of the coin, Dev is responsible for product development and quality assurance. The conflict between the two departments happens in the overlap of duties, especially in the case of managing development resources. When it comes to Docker containers, there is often disagreement as to which department actually owns them because the same container can be used in both development and production environments.

If you were to ask me, I would say without hesitation that Dev owns Docker containers, but thanks to the obvious bias I have as a developer, that is probably an overly-simplistic viewpoint. In my personal experience, getting development-related resources from Ops can be tough. Now don't get me wrong, I'm under no impression that this is because of some Shakespearian blood-feud; Ops just has different priorities than Dev, and spinning up yet another test server just happens to land a little further down the list. When it comes to managing development resources, I think it is a no-brainer that they should fall under the Dev umbrella. Empowering Dev to manage their own resources reduces tension between the departments, manages time and priorities more appropriately, and keeps things running smoothly.

On the flipside, Docker containers that aren't used directly for development should fall under the purview of Ops. Database containers are good examples of this type of separation. While the MySQL container may be used by Dev, no development needs to be done directly on it, which makes the separation pretty clear. But what about more specialized containers that developers work directly on, like workers or even (in some instances) web servers? It doesn't really make sense for either department to have full control over these containers, as developers may need to make changes to the containers themselves (for the sake of development) that would normally fall under the Ops umbrella if there was clear separation between development and production environments.

The best solution I can think of to this particular problem would be joint custody of ambiguous containers. I think the reason this would work well is that it would require clear documentation and communication between Dev and Ops as to how these types of containers are maintained, which would in turn keep everybody happy and on the same page. A possible process that could work well would be for Ops to be responsible for provisioning base containers, with the understanding that the high-level configuration of these types of containers would be manageable by Dev. Because Ops typically handles releases, it would then be back on Ops to approve any changes made by Dev to these containers before deploying. This type of checks-and-balances system would provide a high level of transparency between the two departments, and also maintain a healthy partnership between them.