DZone
A DEVADA MEDIA PROPERTY

TREND REPORT

CI/CD

Automation for Reliable Software Delivery

BROUGHT TO YOU IN PARTNERSHIP WITH

SAUCELABS

Welcome Letter

By Kellet Atkinson, Director of Product at DZone

When embarking on the journey toward CI/CD implementation, few software engineers question the benefits of fully automated builds, tests, and releases. The promise of successful continuous integration and continuous delivery is reliable, frequent shipping of high-quality software, which in turn equates to fewer development trade-offs, faster value delivery, and a better return on development time invested. In other words, CI/CD is the hallmark of productive and mature software processes.

Despite the clear benefits, the journey to a well-established CI/CD pipeline has historically been a painful one defined by a miscellany of tools; hand-coded, custom scripts; and keen expertise. Depending on the complexity of the technology stack, engineering and workplace culture, and training of individuals responsible, full CI/CD has remained aspirational for most organizations.

A cursory glance at the current state of software development today would suggest that CI/CD is likely more difficult to achieve than ever. Increasingly distributed workforces means that developers may check in code at anytime from anywhere; cloud-native architectures introduce, potentially, thousands of different environments to be maintained; and an expanding selection of tools and frameworks leaves an infinite combination that must be stitched together.

The good news is that while achieving CI/CD is still a somewhat lofty goal, just as the landscape of development itself has changed, so too has the ecosystem of tools for building your software delivery pipeline. Where the historical approach to CI/CD involved building a custom-fit solution comprised of unopinionated tools, the newest era of tools lowers the barrier to CI/CD.

Opinionated tools offer ready-made solutions for common technology stacks (like Spinnaker for Kubernetes) where declarative pipelines remove shared engineering burdens, like endless scripting, and end-to-end DevOps platforms eliminate the reliance on numerous disparate tools.

Presumably, as a reader of this report, you already recognize the benefits of CI/CD, and you have even spent time examining the current landscape of tools and best practices for building a modern delivery pipeline. Our hope is that this report allows you to gain a better understanding of the ever-growing CI/CD landscape, learn from your peers, and ultimately, impart the feeling that you are not alone in the continuous journey to improving the way you both integrate and deliver high-quality software.

Sincerely,

Kellet Atkinson

Kellet Atkinson, Director of Product at DZone

@kellet on DZone | @kelletatkinson on LinkedIn

Kellet Atkinson is a graduate of UNC Chapel Hill, with an MBA from NC State University. He is currently the Director of Product at DZone.com, where he has spent the last eight years dedicated to growing DZone's global community of software professionals. Outside of DZone, Kellet is the guitarist for a local Raleigh, NC band and enjoys playing ice hockey.

ORIGINAL RESEARCH

Key Research Findings

An Analysis of Results from DZone's 2021 CI/CD: BUILD Survey

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

In May 2021, DZone surveyed software developers, architects, site reliability engineers, platform engineers, and other IT professionals in order to understand how the way software is built relates to the way software is delivered.

Major research targets were:

1. Relation of software delivery and design
2. Relation of software delivery techniques and effects

Methods:

We created a survey and distributed it to a global audience of software professionals. Question formats included multiple choice, free response, and ranking. Survey links were distributed via email to an opt-in subscriber list and popups on DZone.com. The survey was opened on May 6th and closed on May 13. The survey recorded 695 responses.

In this report, we review some of our key research findings. Many secondary findings of interest are not included here. Additional findings will be published piecemeal on DZone.com.

Research Target One: Relation of Software Delivery and Design

Motivations:

1. Engineering achieves more than trial-and-error because in some measure, engineering activity is always more than JIT. The Agile movement has rightly responded to the vastness of software possibility space (say, DSPACE) and the fuzziness of many real-world software requirements (which, say, pushes DSPACE toward NSPACE).

   But in practice, and exactly counter to its original intent, Agile is too often used as an excuse for no design. Similarly, the "fail fast, fix fast" ethos that continuous delivery enables — and perhaps that even distributed version control enables — can too easily lead to duct-tape redeployments that burden a clean initial design with a debt of extra-design patches. We wanted to understand how to prevent CD from regressing software engineering into trial-and-error.

2. Many principles and patterns of software design are not aimed at runtime efficiency (i.e., algorithm tuning) but rather at robustness, readability, and maintainability.

   These two objectives are often theoretically in tension. But in practice, decades of writing software in increasingly high-level languages that run on fewer hardware architectures has resulted in compilers and interpreters that apply the aggregated cleverness of thousands of developers to the translation of human-intelligible expression sets into machine-intelligible, runtime-efficient instruction sets. A similar tension may arise between good design and useful feedback on that design.

   Agile methodologies are more appropriate when the problem domain is poorly understood (i.e., when a precise specification is unlikely to match reality very well), and where quicker production feedback loops will actually result in better design. As modern compilers have aggregated millions of man-hours of developer hand-tuning, so too modern deployment/release tools and techniques have applied countless aggregated man-hours of deployment suffering to any given deployment/release.

We wanted to understand how best to use modern CI/CD tools and techniques, not only to release fast but also to release at a rate such that software design is improved.

3. Anecdotally, minimally mature software release pipelines depend entirely on manual steps. Moderately mature software release pipelines introduce automation but often without immediately pinpointing where humans ought best to apply their intelligence. Maximally mature software release pipelines have figured out where humans best fit in the loop.

   We wanted to understand how software professionals decide where to require manual intervention (e.g., code review) or enforce human processes (e.g., source branching strategies) all the way from writing code to production release.

USE OF TDD, BDD, DDD, AND OOAD

Overheard at developer happy hour:

> **TDD zealot:** *"Always write tests first. Tests are the only true specification."*
>
> **Non-TDD-zealot:** *"But I don't know the requirements yet. We need to prototype first and iterate."*
>
> **TDD zealot:** *"All the more reason to write tests first. Without them, you won't even know when your prototype has evolved internal contradictions."*
>
> *{...six months later, a week before go-live...}*
>
> **Non-TDD-zealot:** *"Dang it, the architect is telling us we can't release without increasing our test coverage by at least another 20%. Time to write some tautological asserts."*
>
> **TDD zealot:** *"..."*
>
> *{...eight months later, everything breaks because of something someone stuck in last-minute and never wrote a test case for...}*
>
> *{the team adopts TDD}*
>
> *{the team rewrites all tests fifty times as requirements change}*
>
> *{the team curses and abandons TDD}*
>
> *{GOTO 0}*

No rule-of-thumb software development methodology is likely to be optimal in all situations. But a given methodology may correlate with other attributes besides optimality. For example, one might reasonably conjecture that domain-driven design exerts some pressure toward object-oriented (OO) design because both paradigms heavily take advantage of bounded contexts. Or again, it might seem plausible that behavior-driven design should encourage test-driven development because the path from Cucumber to unit tests is comparatively straightforward.

We wanted to see what correlations with higher-level methodologies actually hold in practice, so we asked:

How often do you take the following approaches to software development and design? {Test-driven Development (TDD), Behavior-Driven Development (BDD), Domain-Driven Design (DDD)}

Results (n=592):

(See Figure 1 on the next page)

Figure 1

APPROACHES TO SOFTWARE DEVELOPMENT AND DESIGN BY FREQUENCY OF USE

	Test-driven development (TDD)	Behavior-driven development (BDD)	Domain-driven design (DDD)	Object-oriented analysis & design (OOAD)
Never	8.7%	16.5%	10.3%	6.9%
Rarely	14.5%	20.4%	15.6%	8.7%
Sometimes	26.5%	31.8%	31.4%	17.4%
Often	30.1%	23.1%	28.9%	35.3%
Always	20.1%	8.1%	13.9%	31.7%
n=	667	641	634	634

Observations:

1. Only a small minority of respondents overall (15.6%) rarely or never use object-oriented analysis and design. On its own, this result is unsurprising and intuitive, though it may be nice to put a number to the intuition.
2. Somewhat more interesting is that this result did not appear to hold robustly across software types. In particular, respondents who are currently developing high-risk software, defined as "bugs and failures can mean significant financial loss or loss of life," are significantly less likely to use object-oriented analysis and design:

Figure 2

SOFTWARE TYPE VS. USE OF OBJECT-ORIENTED ANALYSIS AND DESIGN

	Currently developing web apps	Currently developing enterprise apps	Currently developing high-risk apps
Never	6.2%	5.9%	6.9%
Rarely	8.8%	6.6%	13.8%
Sometimes	17.2%	20.3%	24.1%
Often	35.7%	33.9%	29.9%
Always	32%	33.2%	25.3%
n=	487	271	87

Since OO is not intrinsically unsafe, this result is difficult to interpret, and we can only speculate as to why at this point. Some possible explanations:

- OO design is not aimed at tight control over concrete execution paths.
- The OO class concept that allows class-level variables can make state hard to follow.
- Inheritance hierarchies can be hard to understand and can introduce unexpected ambiguities (and correspondingly opaque compiler decisions).
- Runtime polymorphism rapidly explodes path branches.
- Dynamic binding makes the code under consideration depend on externalities unpredictably.
- Formal verification methods are not ideally suited to OO code.

These are all guesses, however, and apply more to some OO languages than others. More research is needed to understand why developers of high-risk software are less likely to use object-oriented analysis and design (OOAD).

3. Nearly one third (30.1%) of respondents use test-driven development (TDD) "often," and one fifth (20.1%) use TDD "always." Both of these numbers seem impressively high, but especially the latter.

   These high numbers are robust across application types, with 19.6% of respondents currently developing web or enterprise apps using TDD "always" and 20.9% of developers of high-risk apps using TDD "often." Respondents developing high-risk apps are also more likely to use TDD "often" (34.1% vs. 30.4% for web apps and 31.6% for enterprise apps).

4. Interestingly, more experienced respondents (defined as > 5 years of experience as a software professional) are less likely to use TDD than less experienced respondents (defined as ≤ 5 years as a software professional):

   Table 1

   TDD USE BY EXPERIENCE LEVEL

Frequency	Senior	Junior
Always	18.5%	23.1%
Often	29.8%	31.5%
Rarely	16.0%	9.3%

   This cannot be explained by the claim that TDD is a new concept: Test-first development was already discussed within the extreme programming movement in the early 2000s. Our own experience, parodied in the dialogue that opens this section, might suggest that junior developers are more likely to try to do things "the right way," while senior developers are more likely to have tried to do things "the right way" in the past before giving up.

   On this interpretation, the "correctness-stickiness" (meaning the likelihood that an enthusiastic adopter will not abandon the approach due to experience) of TDD is less robust than the correctness-stickiness of OO analysis and design. Senior respondents are significantly more likely to use OOAD "always" (32.6%) than junior respondents (25.7%), though are less likely to use OOAD "often" (33.7% senior vs. 39% junior). This latter always/often discrepancy (not observed in the case of TDD) may reflect OOP's greater tendency to produce mental habits:

   Your brain on TDD differs far less from your brain not on TDD more so than your brain on OO differs from your brain not on OO.

5. Similarly, junior respondents were significantly more likely to report using behavior-driven development (BDD) "often" (28.2% junior vs. 22.7% senior) or "always" (12.6% junior vs. 6.5% senior).

   In conjunction with the parallel TDD usage difference by experience level, this may suggest that requirements capture in highly open problem domains is growing more mature. TDD and BDD are less likely to be useful for developing a

highly specified system in a relatively closed domain, but (we assume) as software "eats the world," newer software is more likely to address open problem domains.

RELATION OF TDD, BDD, DDD, AND OOAD TO INCIDENTS/ROLLBACKS

All of these methods, TDD especially, promise to get things right before release so that incidents or rollbacks after release are less likely. We wanted to see if these promises are fulfilled in practice. So we asked:

How often do your deploys result in incidents or rollbacks? {Almost every deployment, Occasionally, We rarely have to back due to deployment}

Observations:

**On results (n=641), correlated with methodologies*

1. Respondents who reported incidents/rollbacks on "almost every deployment" are significantly more likely to use TDD "always" (26.2%) than those who reported incidents/rollbacks "occasionally" (19.2%) or "rarely" (19.6%).

   Somewhat countervailing results appear among respondents who use TDD "often": They are slightly more likely to report incidents/rollbacks "rarely" (32.2%) vs. "occasionally" (26.7%) and "almost every deployment" (31%). These mixed results are consistent with mixed academic research on TDD's effect on software quality in literature beginning in the early 2000s.

   We might guess that perhaps those who use TDD "always" may be methodology zealots who are more likely to pay excessive attention to merely writing tests first and proportionally less attention to test quality and software design. But this remains a guess. Our present data do not alter the mixedness of reported TDD results.

2. Impact of object-oriented analysis and design on incident/rollback rate is similarly ambivalent.

   Respondents whose deployments "almost always" result in incidents/rollbacks are most likely to use OOAD "always" (34.2% vs. 28.6% for "occasionally" and 32.5% for "rarely"), but respondents who use OOAD "often" are most likely to report incidents "rarely" (35.9% vs. 35.5% for "almost every deployment" and 31.6% for "rarely").

   In this case, however, since OO is a programming paradigm as well as a design methodology — and on the premise that design intrinsically involves trade-offs while methodology may or may not involve trade-offs — we might take these numbers to mean, very roughly, that absolute adherence to OO design principles causes more trouble than frequent but flexible adherence does. For more on the relation between specific SOLID OO design principles and incidents/rollbacks, see analysis below.

3. Behavior-driven development (BDD) performs the worst with respect to incident/rollback rate:

   - 10.3% of those who use BDD "always" reported incidents/rollbacks "almost every deployment" vs. 4.7% "occasionally" and 9.3% "rarely."
   - 30.8% of those who use BDD "often" reported incidents/rollbacks "almost every deployment" vs. 27.2% "occasionally" and 18.5% "rarely."
   - 41% of those who use BDD "sometimes" reported incidents/rollbacks "almost every deployment" vs. 30.2% "occasionally" and 33.1% "rarely."

   This result is less ambivalent and proportionally more surprising, and we see no plausible reason why BDD would result in more incidents/rollbacks — apart from a vague "BDD requirements look more formal than they are, which results in test design complacency."

   However, the number of respondents who use BDD is sufficiently small that these segmented numbers are not very robust (e.g., only four respondents both use BDD "always" and experience rollbacks/incidents "almost every deployment"). So we should consider these numbers more as suggestions for future research opportunities than as meaningful on their own.

4. *Caveat lector:* The percent of respondents who reported that deploys "occasionally" or "rarely" result in incidents or rollback is reversed in this survey vs. another recent survey we ran.

This suggests that respondents to the present survey, which was distributed slightly differently, enjoy a more mature development and release cycle than respondents to our other survey. The absolute numbers, however, are not interesting, and none of our conclusions rely on them. We are currently interested only in correlations with other responses — this does not depend on absolute count (except where a problematically small-n-segment results, which we always note).

RELATION OF SOLID PRINCIPLE VIOLATION TO INCIDENT/ROLLBACK RATE AND PERCEIVED TECHNICAL DEBT

SOLID object-oriented design principles are mainly aimed at maintainability and robustness to change. Therefore, we might expect that violating SOLID principles would result in both objective harm to delivery, which we measure in terms of incident/rollback rate, and subjective increase in perceived technical debt. In our survey, we distinguished intentional vs. unintentional violations of SOLID principles. We reserve discussion of differences between intentional vs. unintentional SOLID violations for a future publication more focused on software design. In this report, we focus instead on the unintentional violations, since those were not separately/exceptionally judged beneficial before release.

In our survey, we defined the SOLID principles as follows:

- **Single-responsibility principle** – Every class should have only one reason to change.
- **Open-closed principle** – Every class should be open for extension but closed for modification.
- **Liskov substitution principle** – Every subclass should be substitutable for its base class.
- **Interface segregation principle** – No interface should require a client to worry about things it does not want from the implementation.
- **Dependency inversion principle** – No abstraction should depend on concrete implementation details.

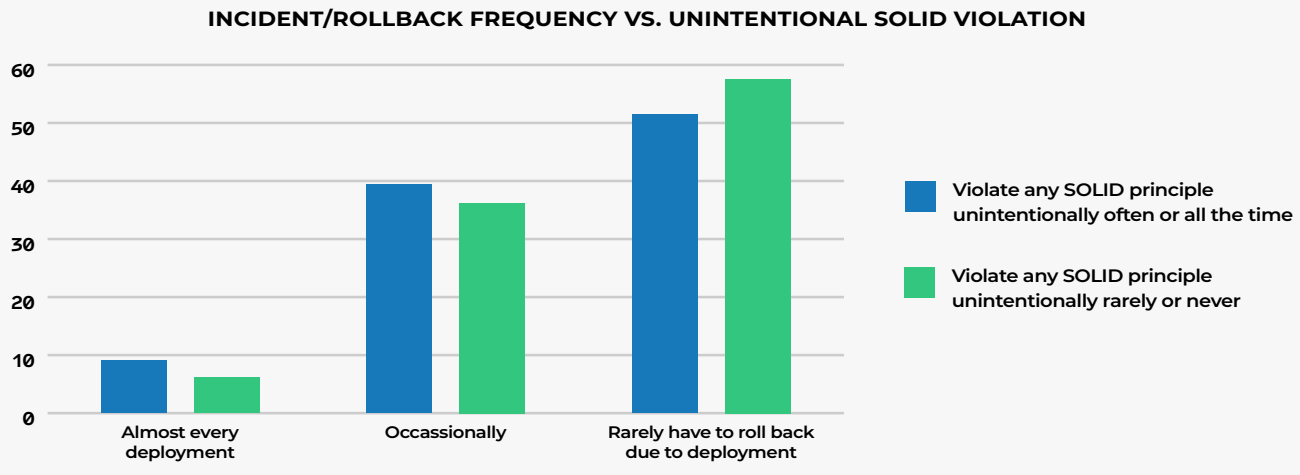
In future research, we plan to consider the relation of each SOLID principle to objective and subjective measures of excellence separately. In this report, we consider the relation of the violation of any SOLID principle *en bloc* to responses to two questions:

1. **Objective** – *How often do your deploys result in incidents or rollbacks?*
2. **Subjective** – *Overall, software built at my organization carries: {too much technical debt, too little technical debt, the optimal amount of technical debt, I have no opinion}*

IMPACT OF UNINTENTIONAL SOLID PRINCIPLE VIOLATION ON INCIDENTS/ROLLBACKS

Results (n=434):

Figure 3



Observation:

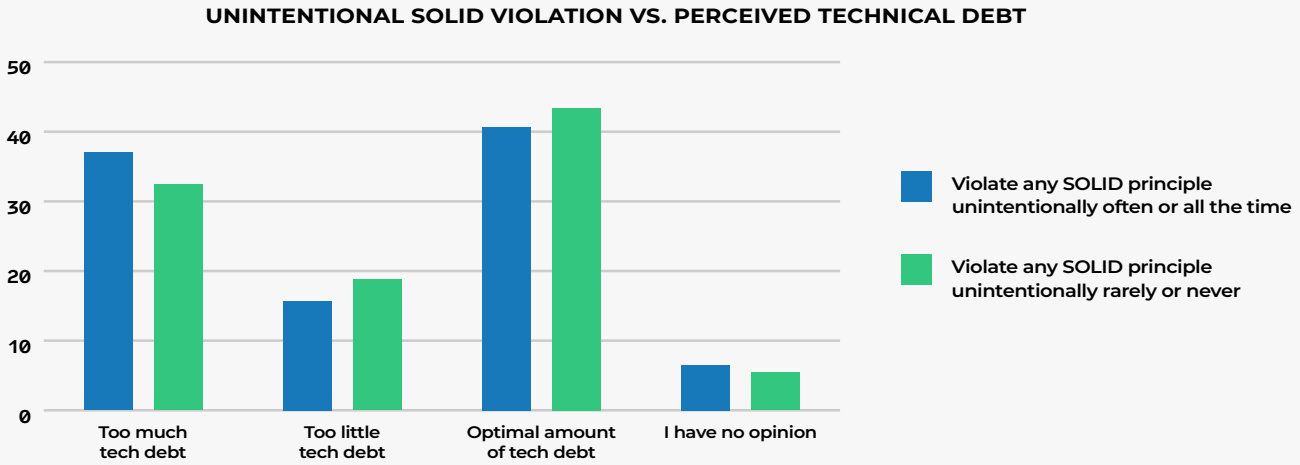
Violating SOLID principles correlates positively with incident/rollback frequency. 9.1% of respondents who violate any SOLID principle "often" or "all the time" reported incidents/rollbacks after "almost every deployment" vs. 6.2% of those who violate any SOLID principle "rarely" or "never." Similarly, 57.6% of respondents who violate any SOLID principle "rarely" or "never" reported that deployments result in incidents/rollbacks "rarely" vs. 51.5% of those who violate any SOLID principle "often" or "all the time."

Note: This correlation does vary across each SOLID principle individually; we plan to discuss this in a future publication.

IMPACT OF UNINTENTIONAL SOLID VIOLATION ON PERCEIVED TECHNICAL DEBT

Results (n=447):

Figure 4



Observations:

1. Again, those who "rarely" or "never" violate any SOLID principle fare better than those who do so "often" or "all the time." They are somewhat less likely to report that their organization carries "too much" technical debt (32.4% vs. 37%) and somewhat more likely to report that their organization carries "optimal" technical debt (43.4% vs. 40.7%).
2. The "rarely" or "never" SOLID violators also appear to have more confidence than the "often" or "always" violators. More "rarely" or "never" violators reported that their organization carries "too little" technical debt (18.8%) vs. 15.7% of "often" or "always" SOLID violators.

MICROSERVICES AND CONTINUOUS DELIVERY

In principle, microservices should facilitate continuous delivery: Well-defined contracts among independently growing service-oriented nodes allow each node to release independently of others. We wanted to see if the CD increase that microservice architecture promises holds true in practice, so we asked:

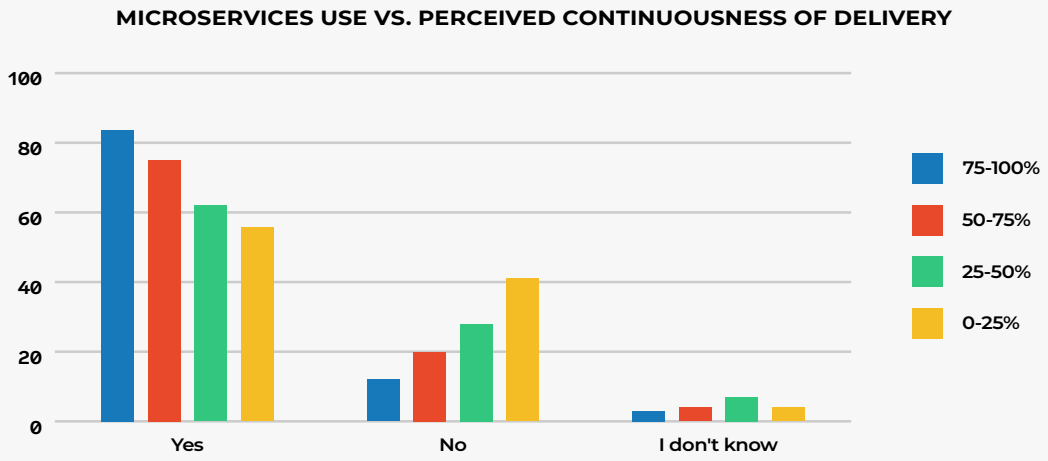
Does your organization use microservices?

Results, correlated with rated continuousness of delivery (n=584):

**For discussion of this rating, see discussion under "Perceived Continuousness of Delivery" below.*

(See Figure 5 on the next page)

Figure 5



Observation:

The promised CD-microservices correlation is apparently fulfilled. Each step in increasing continuousness of delivery results in increased adoption of microservices. 83.7% (n=210) of respondents who rated their organization's continuousness at 75-100% reported using microservices vs. only 55.7% of respondents who rated their organization's continuousness at 0-25%.

We also wanted to check whether the promise was fulfilled in the other direction — to see how microservice architecture impacts feature velocity. We asked directly:

In my experience, adopting a microservice architecture has resulted in: {higher feature velocity, lower feature velocity, no change in feature velocity, I don't know}

Results (n=468):

Table 2

MICROSERVICE ARCHITECTURE IMPACT ON FEATURE VELOCITY		
Perceived impact	**Percent**	**n**
Higher feature velocity	71.2%	333
Lower feature velocity	12.6%	59
No change in feature velocity	10.7%	50
I don't know	5.6%	26

Observation:

Nearly three quarters of respondents (71.2%, n=333) reported that adopting a microservice architecture results in "higher feature velocity." In respondents' judgment, microservices strongly deliver on their promise to increase feature velocity.

Research Target Two: Relation of Software Delivery Techniques and Effects

Motivations:

1. In any engineering discipline, increasing development and release speed involves trade-offs.

   In physical engineering disciplines, development/release/opening speed is automatically subject to physical constraints (e.g., specific heat, concrete setting time, martensite formation rate), while software engineering faces far fewer (and in some theoretical sense, no) physical constraints. It is, therefore, less obvious in the case of software vs.

physical engineering disciplines when the product is being deployed at the wrong rate.

2. Harmful effects of too fast or too slow software deployment are not always easy to measure.

   For example, technical debt often results in releasing prematurely, but technical debt measurement is not a straightforwardly solved problem. Again, code quality may be impacted positively or negatively by excessively fast (premature release) or excessively slow (insufficient advantage taken of the production crucible) release, but code quality measurement, too, is not a straightforwardly solved problem.

RELEASE FREQUENCY AND TECHNICAL DEBT

CD improves software only when it is not treated as an end in itself. High release frequency does not improve software without complementary feedback loops. In CD pipelines focused on release frequency or feature velocity in isolation, features may be released quickly — but with issues — before new features are released without fixing the issues revealed after release. So CD should not, but sometimes does, increase technical debt. We wanted to see how much this is true in practice.

We asked:

Overall, my organization releases new software features: {too quickly, too slowly, at the optimal rate, I have no opinion}

And:

Overall, software built at my organization carries: {too much technical debt, too little technical debt, the optimal amount of technical debt, I have no opinion}

Results (n=521):

Table 3

CORRELATION OF FEATURE VELOCITY WITH TECHNICAL DEBT		
New feature release frequency	**Perceived technical debt**	
	Too much	**Optimal amount**
Too quickly	24.9%	13.4%
Too slowly	45.1%	15.1%
At the optimal rate	24.9%	66.5%
I have no opinion	5.1%	4.9%

Observations:

1. Perceived feature velocity does not directly correlate with perceived technical debt.

   Organizations that release features "too quickly" and "too slowly" both carry "too much" technical debt by a significant margin. This means that developers evaluate feature velocity and technical debt consistently, which suggests that *developers' judgment of feature velocity and technical debt are worth paying attention to.*

2. Rather, very strongly, optimal feature velocity correlates with optimal technical debt.

Respondents whose organizations carry "optimal" technical debt are two and a half times as likely to judge that features are released at the "optimal" rate (66.5%) vs. those whose organizations carry "too much" technical debt (24.9%).

CODE REVIEW AND TECHNICAL DEBT

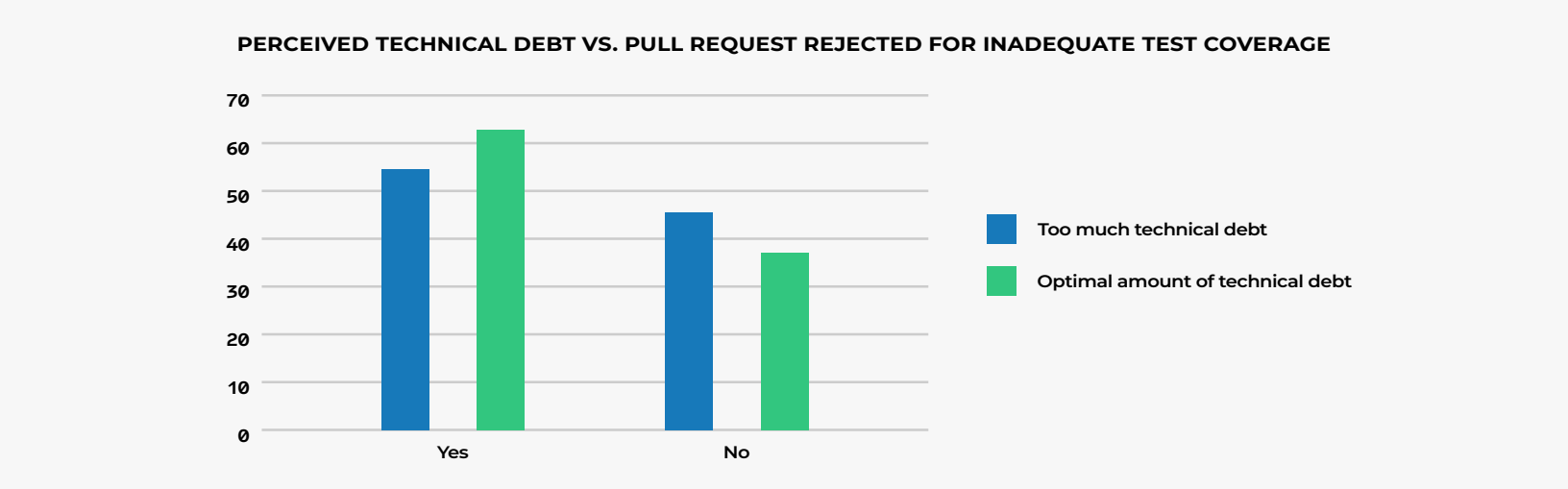
Code review may be done well or poorly. It is hard to tell when code review is done well, but it is easy to tell when code review is done lackadaisically. Moreover, we imagine that, especially when CD is desired, the temptation to skip code review — or treat it as a rubber-stamping formality — is greater than the temptation to block release via excessively strict code review. We

wanted to know at least whether very nonrestrictive code review leads to more technical debt. Using a very simple metric of restrictiveness and insistence on adequate test coverage, we asked:

Have you ever had a pull request rejected because you did not write adequate automated tests (e.g., because coverage measured by LoC was too low, or because test code did not test enough meaningful scenarios)?

Results, correlated with reported technical debt (n=487):

Figure 6



Observation:

Rejection of pull requests due to inadequate automated tests correlates negatively with perceived technical debt. 62.9% of respondents who reported that their organization carries the "optimal" amount of technical debt reported having pull requests rejected due to inadequate test coverage vs. 54.5% of respondents whose organization carries "too much" technical debt. With respect to the simple heuristic of gating code merge over automated test coverage, it appears that overly lax code review does correlate with greater technical debt.

RELEASE FREQUENCY AND PERCEIVED CODE QUALITY UNDER TIME CONSTRAINTS

In addition to specific techniques used to ensure code quality (such as code review), one might suspect that there is some correlation between release frequency and code quality. The direction of this correlation, however, is not obvious *prima facie*. On the one hand, it seems that faster releases should lead to higher quality code — on the Agile assumption that no model tests code as well as production. On the other hand, it seems that faster releases should lead to lower quality code — on the engineering assumption that pressure toward speed for its own sake tends to decrease planning and weaken design.

However, this correlation seems very difficult to measure directly (and earlier research attempting to do so remains mired in controversial, seemingly arbitrary objective measures of "code quality").

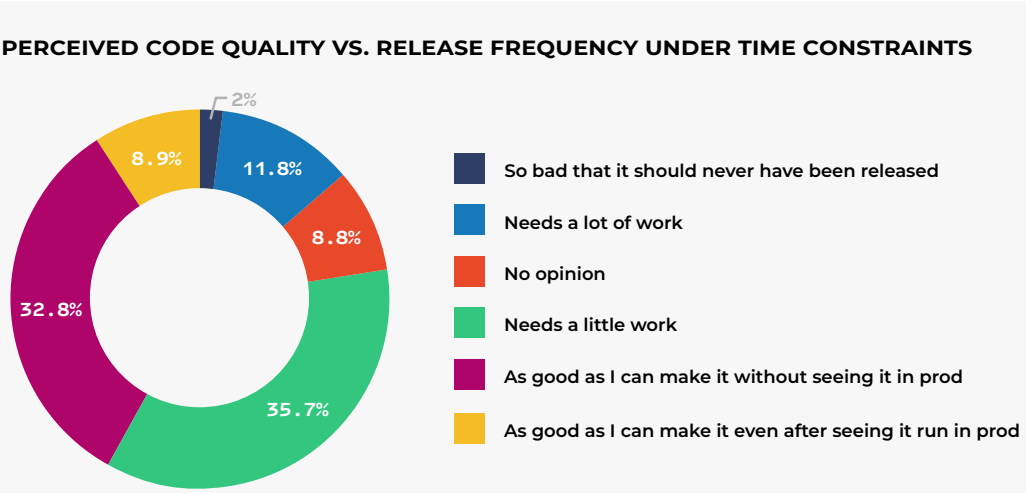
Every developer knows that code quality metrics are a decent place to start looking for opportunities to improve quality but do not measure code quality accurately. So we used software professionals' judgment of code quality as a better (and easier to measure) proxy for actual code quality in order to see how CD pressures positively or negatively affect (perceived) code quality.

We asked:

On average, how good (by your own definition of "good") is the code you work on when it is released to production compared with how good you could make it given unlimited time? {So bad that it should never have been released, Needs a lot of work, As good as I can make it without seeing it run in production, As good as I can make it even after seeing it run in production, No opinion}

Results (n=638):

Figure 7



Observations:

1. Very few respondents (2%, n=13) object so strongly to released code that they judge it should not have been released. This is a good sign for CD maturity: In the judgment of software professionals, people are not pushing truly hideous code to production.

2. Of the two by far most common responses, "needs a little work" (35.7%, n=228) is only slightly more common than "as good as I can make it without seeing it run in production" (32.8%, n=209).

   This is also a good sign for release maturity: Roughly a third of respondents believe that their code cannot get better without the inimitable feedback of the real world, and another third see such a state as just "a little work" away.

3. However, a significant percent of respondents (8.9%, n=57) reported that their code is "as good as I can make it even after seeing it in production." For these respondents, CD would presumably hold less benefit.

   Cross-checking against incident/rollback frequency does not confirm their judgment; however, 8.8% (n=5) of those who reported that their released code is "as good as I can make it even after seeing it run in production" reported incidents/rollbacks after "almost every deployment," vs. 4.3% (n=9) of those who reported that their released code is "as good as I can make it without seeing it run in production." Because the n of each of these subsegments is very small, however, this counter-conclusion should be taken as weak.

4. The distinction between "needs a little work" and "needs a lot of work" tracks incident/rollback rate better.

   Significantly more respondents who reported that their released code "needs a lot of work" also reported that they roll back after "almost every deployment" (16.4%, n=21) vs. those whose code "needs a little work" (4.9%, n=11). Many more "needs a little work" respondents reported that deploying their code "rarely" results in incidents or rollbacks (56%, n=126) vs. those whose code "needs a lot of work" (31.5%, n=23).

TEST COVERAGE AND PERCEIVED CODE QUALITY

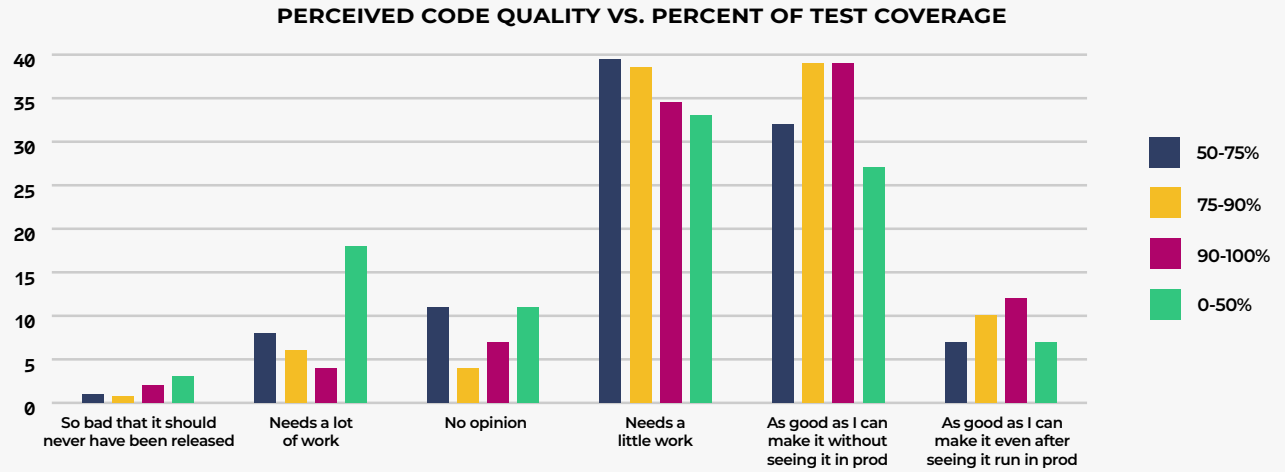
In theory, better test coverage implies better pre-production modeling of the real-world domain, which should result in code that needs less production feedback to achieve optimality. Since test quality is difficult to assess globally, we wanted to see how test coverage and perceived quality of released code relate with respect to a simple metric — percent of code covered by automated tests. We asked:

For software you work on, what percent of lines of code must be covered by unit tests before the code is allowed to move to production?

We accepted free response numbers, but for analytical purposes, we bucketed responses into ranges of code coverage percent: 0-50%, 50-75%, 75-90%, and 90+%.

Results (n=608):

Figure 8



Observation:

The relation between code coverage requirement and perceived code quality on release is complex. The most intuitive results:

1. Those who require very high coverage (90-100%) are most likely (12.6% vs. 10.3% for next-highest percentage — those who require 75-90% coverage) to judge their software as good as they can make it even after seeing it in production.

   That is, presumably, they judge their tests to have modeled production well. The fact that the second-highest code coverage requirement comes in second with respect to maximal release code quality judgment reinforces this interpretation.

2. Those who require very low test coverage (0-50%) are by far most likely (18.3% vs. 8.5% for next-highest percentage — those who require 50-75% coverage) to report that their code "needs a lot of work." Again, the fact that the second-lowest code coverage requirement comes in second with respect to lowest non-catastrophic ("should not have been released") code quality judgment reinforces this interpretation.

PERCEIVED CONTINUOUSNESS OF DELIVERY

In other recent research, we examined continuous delivery maturity with respect to specific, objective metrics (deployment frequency, lead time, MTTR, change failure rate, etc.). For this report, our research goal was to allow respondents to judge "continuousness of delivery" by their own standards, and then see how well this subjective rating correlated with indicators of CD success, code quality, etc. We wanted to treat a somewhat fuzzy concept (continuous delivery) in an appropriately fuzzy way.

We asked (response input was a slider):

How continuously does your organization deliver software (left = least continuous, right = most continuous)?

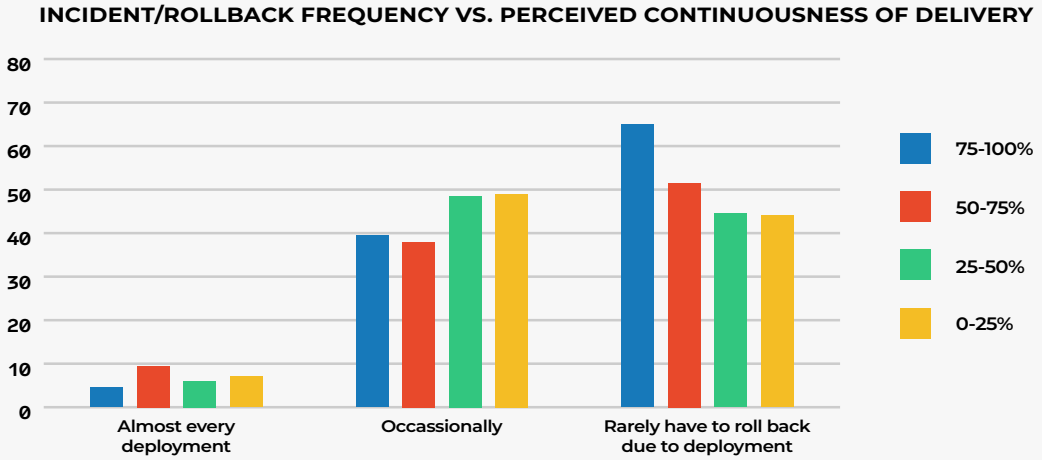
For example, if your organization uses git for source control and has a CD pipeline that deploys to production on git push to master, then the rightmost ("most continuous") term on the slider would be git push origin master => successful build+test => production deploy.

To simplify analysis, we grouped continuousness ratings into buckets of 0-25%, 25-50%, 50-75%, and 75-100% as segment of the maximum continuousness rating (accepted via slider input).

RELATION OF PERCEIVED CONTINUOUSNESS OF DELIVERY AND INCIDENT/ROLLBACK RATE

Results (n=584):

Figure 9



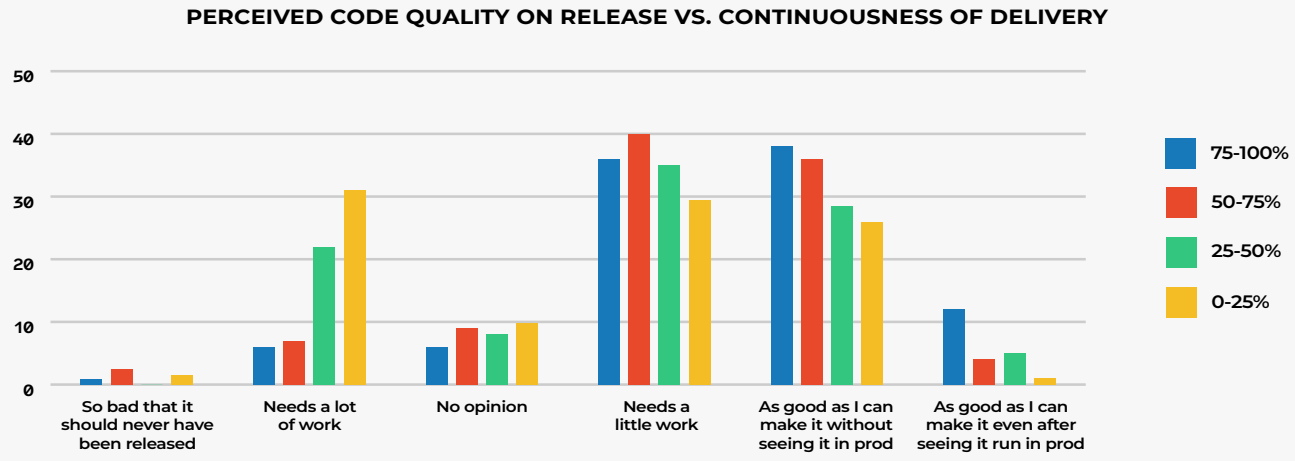
Observation:

Perceived continuousness of delivery correlates negatively with incident/rollback frequency. 65.7% (n=165) of respondents who rated their organization's continuousness between 75-100% of maximum reported that deployments "rarely" result in incidents or rollbacks vs. 52.2% of respondents who rate continuousness between 50-75% and 44.7% of those who rate continuousness between 25-50%.

RELATION OF PERCEIVED CONTINUOUSNESS OF DELIVERY AND CODE QUALITY ON RELEASE

Results (n=583):

Figure 10



Observations:

1. Rated continuousness of delivery correlates positively with perceived code quality on release.

   12.5% of respondents who rated their organization's continuousness of delivery between 75-100% reported that their code is released in a state that is "as good as I can make it even after seeing it in production" vs. 28.4% of those who reported continuousness ratings between 25-50%.

2. Similarly, rated continuousness above 50% correlates with significantly higher “as good as I can make it without seeing it in production” responses. And the released code quality judgment, “needs a lot of work,” is by far most common among those who rated their organization’s continuousness in the lowest quarter (31.3% vs. 22.1% among those who rated continuousness 25-50%).

Future Research

This report presents partial results of the second of our two 2021 surveys on continuous delivery. The first survey treats the release portion of the SDLC from a process point of view, while the survey discussed in this write-up focuses on relations between CD and software design and quality. The results of the second survey, only partly treated here, cover additional material that we intend to analyze elsewhere, including:

- Relation of programming paradigms to CD and deployment experience
- Differences between prevalence and effects of intentional vs. unintentional violations of SOLID OO design principles
- Count of manual interventions required to go from dev to production and relation to software architecture and testing strategy
- Relation of database normalization to deployment and incident/rollback frequency
- Complexity of DevOps pipeline in terms of tool count
- Overall effect of CD on application quality and security
- Prevalence of version control for various definables (source, build scripts, infrastructure configuration, etc.)
- Blue/green deployments
- Impact of CD on source branching strategy

John Esposito, PhD, Technical Architect at 6st Technologies
@subwayprophet on GitHub | @johnesposito on DZone

John Esposito works as technical architect at 6st Technologies, teaches undergrads whenever they will listen, and moonlights as research analyst at DZone.com. He wrote his first C in junior high and is finally starting to understand JavaScript NaN%. When he isn’t annoyed at code written by his past self, John hangs out with his wife and cats Gilgamesh and Behemoth, who look and act like their names.

A brief history of web and mobile app testing.

Start your free trial today

saucelabs.com/sign-up

SAUCELABS
Continuous Confidence

Email sales@saucelabs.com or call (855) 677-0011 to learn more.

SPONSOR OPINION

Optimizing CI/CD for Continuous Testing — Adapting Processes

By Marcus Merrell, Sr. Director of Technology Strategy at Sauce Labs

Moving to an effective and efficient CI/CD pipeline requires significant effort from organizations. It involves process and policy changes across every team. The payoff can be extraordinary: continual improvement as the organization moves to consistently deliver high-quality digital experiences to their customers at speed. However, as your release velocity begins to increase, how can you still ensure your apps are well-tested?

Adding continuous testing best practices from the outset can enable the transformation to CI/CD by allowing code to move through an accelerated pipeline without testing becoming a bottleneck. To achieve this requires an entirely new set of features — testability features — built into the architecture itself, along with other changes to the way software is built. Without these changes, organizations typically struggle to see the benefits that CI/CD promise. This article will focus on the processes to consider when implementing continuous testing in a CI/CD workflow.

ADAPTING PROCESSES

Moving to a pipeline that constantly delivers high-quality, well-tested software is far more a process change than a technical one. Process change can often be more challenging than technical change, as it requires non-technical roles such as stakeholders, product owners, and users to adjust their work habits and mindsets. Getting past the "I just want to do my job!" mindset to "How can I improve my job?" isn't always easy, but there are a number of ways to help ease this transition as team members adapt how they're doing their work in order to successfully move to a CI/CD model.

MOVING TO SMALLER BRANCHES

The branching strategy a team chooses will have a significant impact on the team's ability to work as smoothly as possible in a CI/CD environment. Small, short-lived branches (feature or smaller-sized) allow teams, or even small groups (pairs!) of team members, to do their own work isolated from others' churn in the codebase. Development and testing can be accomplished without interruption from other work in the database.

Normally, each branch will have its own build job in the pipeline. Each branch has a job to monitor that branch in source control, then pull, build, deploy, and test as required. This requires more set-up time up front; however, job/task templates make this much easier.

STORING TESTS IN SOURCE ALONGSIDE CODE

Keeping automated tests in sync with the application code they cover is critical to automated delivery pipelines. The entire team needs to have complete confidence in the checks guarding against regressions and confirming high-value business features. The smoothest approach to handling this is simply keeping automated test code in the same repository as the system code. Having tests right alongside the system makes it simple to pull the current branch from source, build, deploy, and test. There's no mess about determining versions from other repositories, passing of messy variables, etc.

CONCLUSION

While there are a number of technical considerations when building a well-tested CI/CD pipeline, no transformation is more difficult than process. This is because it is primarily a culture change and requires buy-in from all levels of the organization to take effect. By framing these process shifts as beneficial for everyone's daily work, it becomes easier to get the required buy-in across the organization and sets up your CI/CD initiative for success.

CONTRIBUTOR INSIGHTS

Building a CI/CD Pipeline for Cloud-Native Architectures

By Samir Behara, Platform Architect at EBSCO

Cloud-native architecture is a major shift in the software development process. It can be visualized as a combination of patterns like microservices, container orchestration, automated CI/CD, DevOps, and cloud infrastructure. Cloud-native applications need to adopt open standards so that you can avoid vendor lock-in, reduce costs, and innovate much faster with the support of the open-source community. Cloud native is a fundamental shift from the traditional development process. A cloud-native architecture enables developers to focus on the application code and business logic itself.

By adopting cloud-native technologies and practices, you can build loosely coupled applications and run them in dynamic cloud environments to meet modern-day systems' high scalability, speed, and availability needs. This article will:

- Explain the rise of cloud-native architecture and its design patterns.
- Showcase the practices of continuous integration and continuous delivery.
- Explain the key components of building a CI/CD pipeline for cloud-native applications.
- Overview tooling for cloud-native continuous deployment.
- Explain how to implement the GitOps pattern for cloud-native applications.

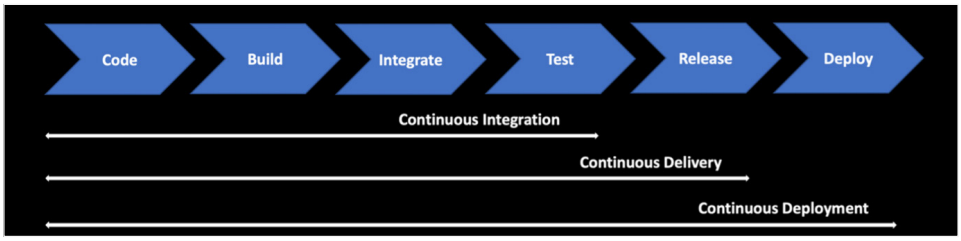
What Is Cloud-Native CI/CD?

To build cloud-native applications, you must adhere to design patterns and technologies like containers, microservices, domain-driven design, service meshes, dynamic environments, immutable infrastructure, automated pipelines, declarative APIs, and public/private clouds. Traditional monolithic architectures are difficult to scale, maintain, and deploy at a frequent cadence. Hence, a microservice architecture is a foundational practice for building cloud-native systems.

Containers are essential to running your services in a dynamic environment. You can package your application code, dependencies, and runtime and store them in a container registry like Docker. Once you containerize your service, it becomes portable and can be executed independently across any environment. To manage containers at scale, you need a container orchestrator. Kubernetes is the most popular open-source container orchestration platform — it provides functionalities like horizontal autoscaling, service discovery, immutable infrastructure, and self-healing systems. Using Kubernetes, you can offload many of these complexities to the container orchestration layer, which your application needn't handle.

In a cloud-native world, automating the development lifecycle and deployment process is necessary. Figure 1 shows the various stages of application builds using CI/CD.

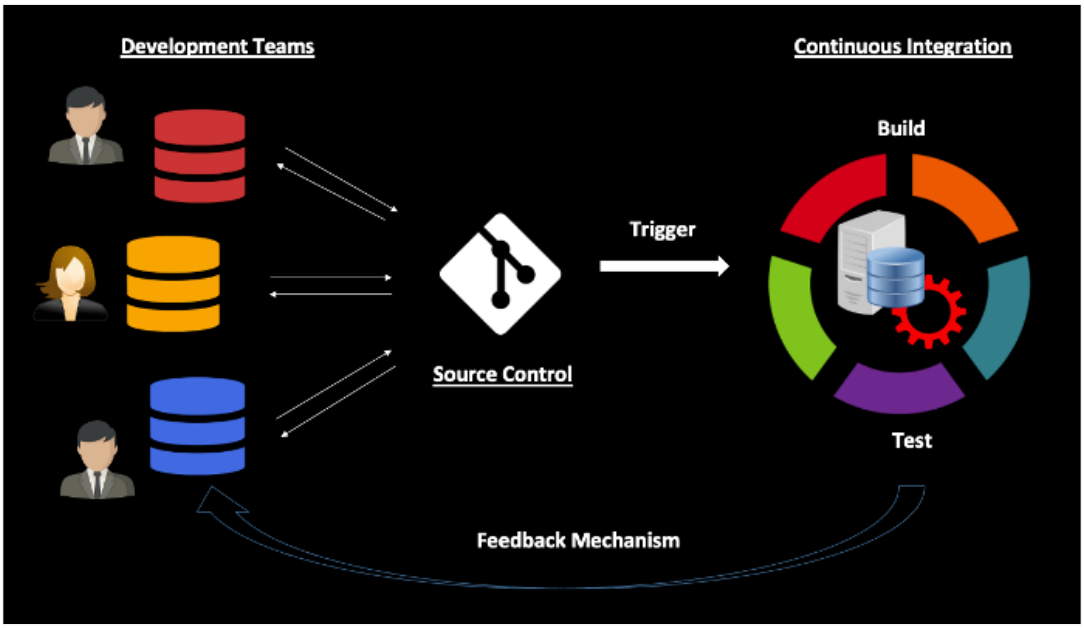
Figure 1: Cloud-native application lifecycle



Continuous Integration as a Development Model

Continuous integration is a development practice where you frequently commit smaller pieces of code into source control to catch issues at an earlier stage. Code commits trigger automated builds on a separate build server where unit/integration tests are executed. You can also perform static code analyses on your code changes to ensure compliance with quality standards. Code is not checked into source control if either failed tests or critical vulnerabilities are detected.

Figure 2: Continuous integration overview



SHIFTING LEFT WITH CLOUD-NATIVE CI/CD

In the SDLC, the "Shift Left" principle lets you identify issues earlier and fix them before deploying to production. Software quality is prioritized by building continuous testing and deployment, static code analysis, and automation into your pipeline at an early stage. This helps increase your delivery speed, reduce the failure rate, and shorten the feedback loop.

Treating your infrastructure as code, storing it in source control, and adopting the immutability principle can ensure that your deployment environments are standardized, which helps address configuration drift issues. It is much cheaper to resolve your development or test environment problems than to fix them in production.

Continuous Delivery — The Right Way

Continuous delivery is the next step after your code is built and tested on a CI server. The goal of continuous delivery is to automate your release process and have production-ready code, which deploys to the production environment after manual testing or approval from business stakeholders. With continuous deployment, if your code passes all stages in your release pipeline, it is automatically deployed to production without any manual intervention. These development practices bring several benefits like improving developer productivity, finding issues earlier in the process, and delivering business features faster.

Let us look at few deployment strategies through which you can successfully achieve continuous delivery:

BLUE-GREEN DEPLOYMENTS

Blue-green deployments allow you to automatically deploy code changes to production with minimal downtime and risk. Multiple production-ready environments are termed "Blue" and "Green." One environment receives the production traffic, and when there is a new release, code is deployed to the other environment for testing to verify the live readiness of the code. If successful, you can flip the switch and route all production traffic to the new environment. If any issues are found in production, you can quickly roll back by shifting traffic back to the old environment.

Figure 3: Blue-green deployments

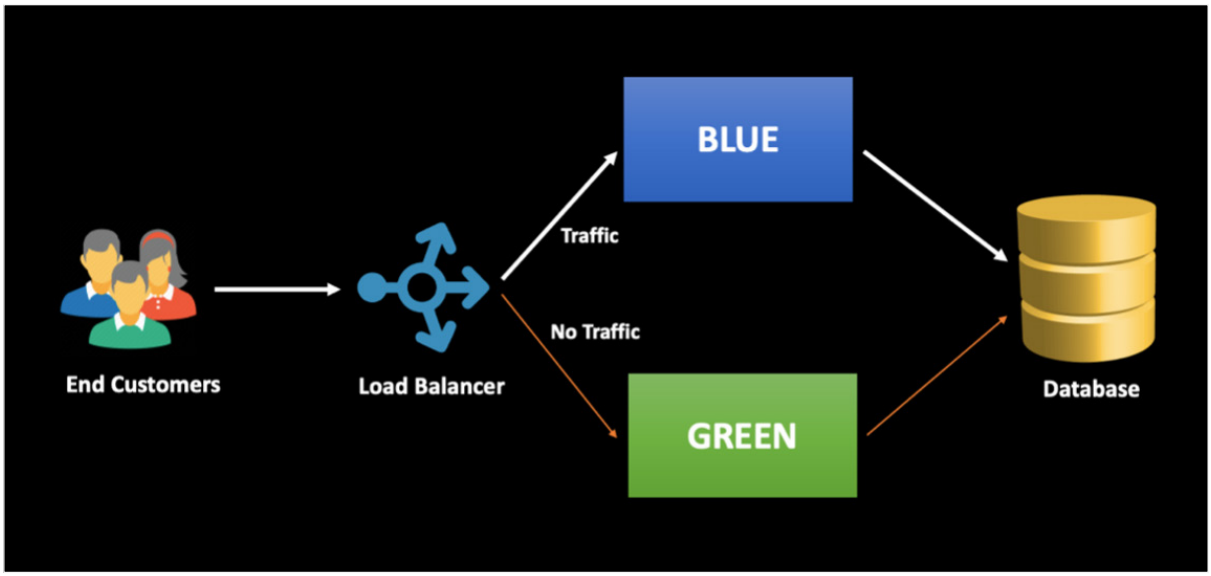


Image Source: "Continuous Delivery with Blue-Green Deployments"

FEATURE FLAGS

Feature flags are a popular deployment strategy for cloud-native applications that helps you remotely enable/disable functionality at runtime in production without making code changes or requiring application downtime. Implementing conditional logic in the code during development lets you release new features in production to a smaller set of end users. If you use feature flags to deploy your latest code changes to production, you can disable the flag if any issues occur after deployment rather than perform a deployment to roll back the released code.

Feature flags can be an alternative to maintaining multiple feature branches, helping reduce the need for constant branching and merging. When working on a large feature, you can continuously deploy incremental code changes with the feature flags turned off. Hence, you don't have to deal with long-lived branches or merge many code changes in one go. Allowing your end users to verify features progressively is beneficial since you can gather feedback quicker and avoid potential issues later in the process.

PREVIEW ENVIRONMENTS

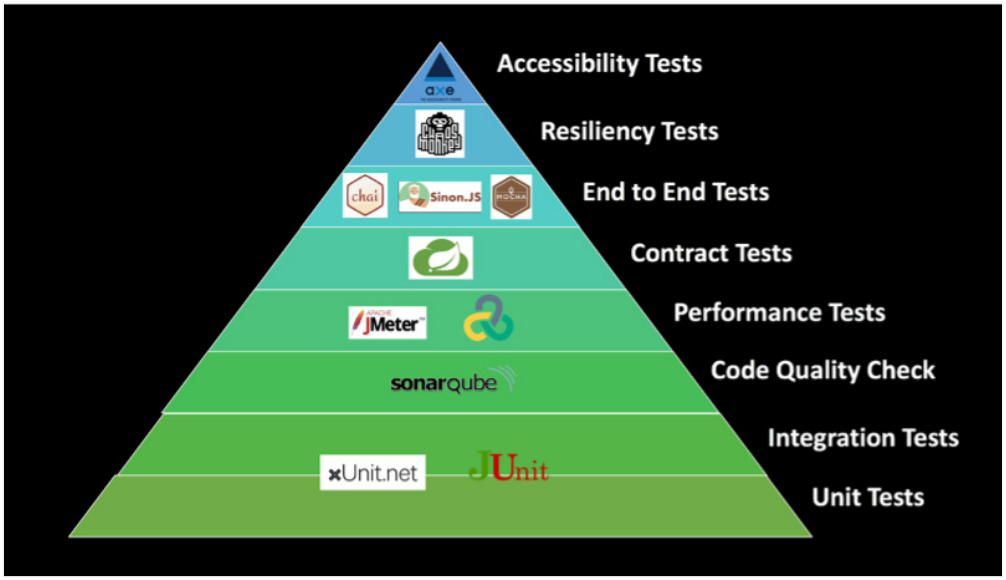
Preview environments provide faster feedback on code changes — with every pull request, you spin up an ephemeral environment that mimics production, which allows you to test code changes in isolation. Development teams can perform acceptance testing in this preview environment. If your application has a dependency on other services, they are also deployed to the preview environment. Once the pull request is closed, the environment is automatically torn down.

Preview environments work well for cloud-native applications because they eliminate the need for a dedicated test environment. They are a good fit for the shift-left principle because more focus is on testing code. Also, developers are more confident in merging code changes to the master branch after verifying them in the preview environment.

AUTOMATED TESTING

Integrating test automation into your CI/CD pipeline is a necessary step to shorten the feedback loop, enhance software quality, and deliver business value faster. Proper tooling and strategies are needed to ensure reliable test automation. In addition to unit and integration tests, you can use performance, end-to-end resiliency, and contract tests to increase confidence in your build process.

Figure 4: Building a testing pyramid

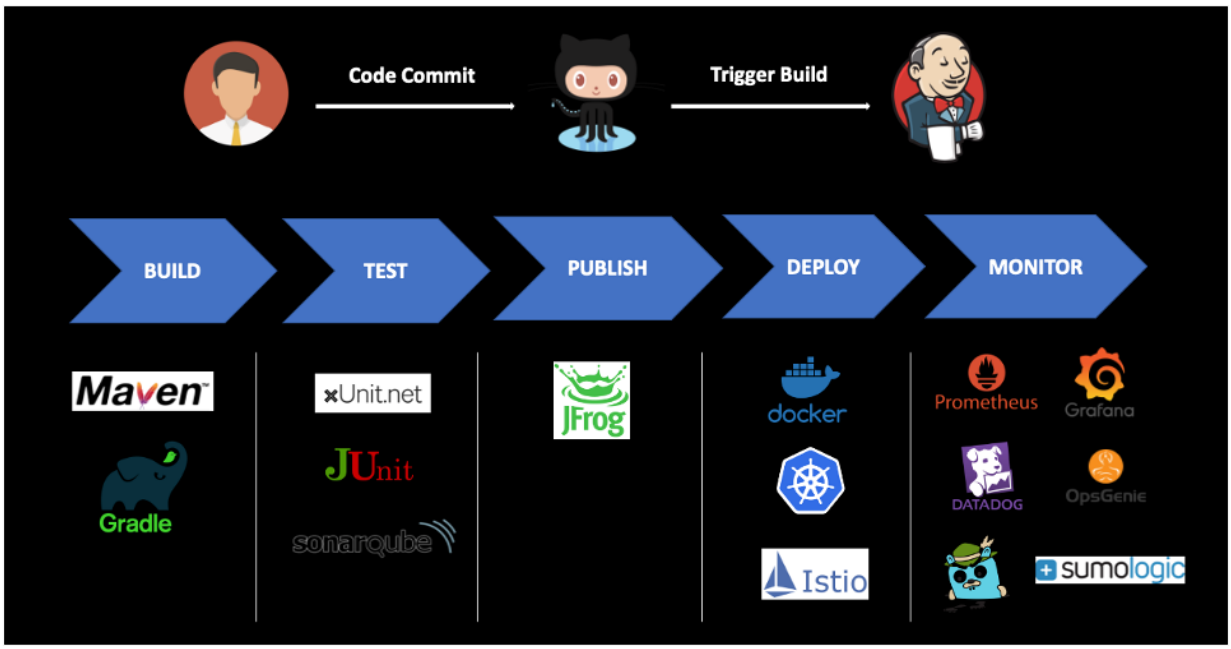


Setting Up a CI/CD Pipeline With a Cloud-Native Toolset

When building a CI/CD pipeline for cloud-native applications, focus on automating as many stages as possible and ensure that your application is independent of the execution environment, so it can deploy to different environments without modifying code. Also, implement a proper branching strategy for your services, where the master branch denotes the current version, and deployments are triggered from the master itself. You can create short-lived feature branches to work on business requirements and merge changes to the master branch after pull request approval. Any changes to the master are built and pushed to a container registry for storing Docker images. Note that the master branch should always remain in a releasable state.

Any code changes will trigger the CI process, which builds your code, executes all unit tests, performs static code analysis, and publishes binaries. Next, the CD process deploys code to a staging environment, where additional tests are executed. If all pipeline stages pass, the code is deployed to the live environment, and then various open-source observability tools can monitor service behavior.

Figure 5: Cloud-native CI/CD pipeline and tooling

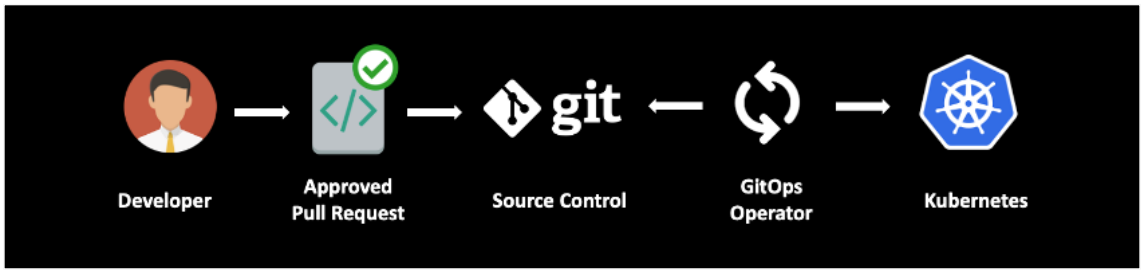


GitOps for Cloud Native

GitOps is defined as an operating model used to build cloud-native applications and is driven by the declarative configuration and infrastructure-as-code (IaC) principles. Git stores your infrastructure and application code and is considered the single source of truth — a declarative approach that has many benefits. GitOps can tremendously boost your deployment frequency and shorten the lead time to production. It has a minimal learning curve since developers are already well versed in using Git and the pull request process for application deployments, so including infrastructure code won't be a heavy lift.

In GitOps, your CI/CD pipeline is initiated by code commits, and once pull requests are approved, changes are automatically deployed. Using GitOps tools enables you to keep your environment and source control in sync, which can increase your confidence in troubleshooting and rolling back changes if required. Every change is driven from Git, creating a complete audit trail of all changes deployed to your environment.

Figure 6: The GitOps Pipeline



If you are using Kubernetes for container orchestration, the GitOps model is a good fit for cluster management and ideal for helping achieve continuous delivery. For example, using GitOps workflows, you can deploy Kubernetes at scale in your production environment. In a GitOps pipeline, you don't need to use `kubectl apply` for manual configurations to the Kubernetes cluster since the desired state of your system is stored in Git, and all application/cluster modifications are driven by changes to your Git repository. There are many GitOps tools available to help keep Kubernetes clusters synchronized with application code in Git.

Conclusion

Building an automated CI/CD pipeline for your cloud-native application is challenging. The shift from traditional on-premises workloads to cloud native brings in additional complexity like cost optimization, security, resource utilization, on-demand scaling, monitoring, logging, tracing, and many more.

Make sure to use the proper tooling and solutions for your architecture. I would highly recommend checking out the vendor-neutral and open-source projects under the Cloud Native Computing Foundation (CNCF) umbrella. You can use any graduated and incubating projects like Kubernetes, Prometheus, Envoy, Helm, Jaeger, Fluentd, etcd, OpenTracing, and CoreDNS to assist you in your migration strategy to cloud native. It is essential to have an open mindset and visualize the effort to automate your deployment process for your cloud-native application as a continuous journey.

Samir Behara, Platform Architect at EBSCO

@samirbehara on DZone | @samirbehara on LinkedIn | Author of https://samirbehara.com

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

CONTRIBUTOR INSIGHTS

Observability & AIOps

Using Analytics, AI, and ML in DevOps and DevSecOps

By Stephanie Phifer, CTO at EITR Technologies

As systems shift toward Everything as Code, DevOps, DevSecOps, and automation have played a major role in improving the resilience, reliability, and scalability of complex infrastructure. Visibility into these systems in terms of health, status, and security posture becomes both an arduous task and a critical gap in managing them. To identify this gap, we can introduce the concept of observability.

According to IBM, observability is the "extent to which you can understand the internal start or condition of a complex system based only on knowledge of its external outputs."[1] Regarding software architectures, this covers application performance, usage patterns, metrics, system uptime, etc. The amount of information produced by these systems can be extremely large and intimidating to tackle. So how do we increase the observability of our systems in an efficient and effective manner? The answer: Artificial Intelligence Operations — AIOps. If you haven't heard, it's not just for Terminators... yet. #illbeback

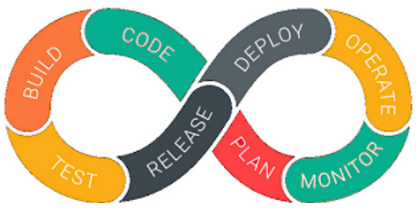
What Is AIOps?

Before we get to AIOps, let's talk about DevOps. DevOps is not just one thing. It's as much a cultural shift as it is a technological one. DevOps focuses on continuously integrating and delivering code. It also focuses on providing real-time monitoring of systems and processes. DevOps ultimately works in service of providing as near to 100% uptime as can be achieved by automating those processes. This trend transitioned rapidly into DevSecOps. Security is now considered and addressed throughout the development and operations lifecycles instead of as an afterthought. While automation largely enabled this massive cultural shift toward DevOps, the concept of observability takes it one step further into the realm of AIOps — artificial intelligence in IT operations.

The term was originally coined by Gartner in 2016 as the "combination of big data and machine learning (ML) to automate IT operations processes, including event correlation, anomaly detection and causality determination."[2] AIOps further enhances the system owner's awareness of IT operations through identification, reaction, prediction, and prevention. When it comes to establishing advanced computational environments, this is not just the latest trend in DevOps — this is the natural evolution of IT operations.

Where's My Data?

To gain the insights into our systems, we need data. The DevOps cycle is broken down into several phases that capture the design, development, testing, and deployment of software, which is depicted below. Throughout each of these phases, data will be generated and aggregated for monitoring, metrics, review, and analysis.



See Reference 3

In perhaps the most traditional case, this data will likely be reviewed following an abnormal event that then requires further investigation. These investigations can be exceptionally tedious and time-consuming without proper data management. Those tasked with investigation might find themselves awash in log files and reports. Those structured and unstructured datasets are where data analysis and data science techniques can begin to play an integral role in IT operations.

Data science is an interdisciplinary field. It uses scientific methods, processes, algorithms, and systems to extract knowledge and insights from structured and unstructured data. That knowledge can then be applied to provide actionable insights and make informed decisions across a broad range of application domains. This umbrella term covers data analytics, ML, and AI. These disciplines each have their own applications and strengths. When integrated with the DevSecOps cycle, they can achieve an increased observability and awareness of system activities and help focus on addressing questions such as:

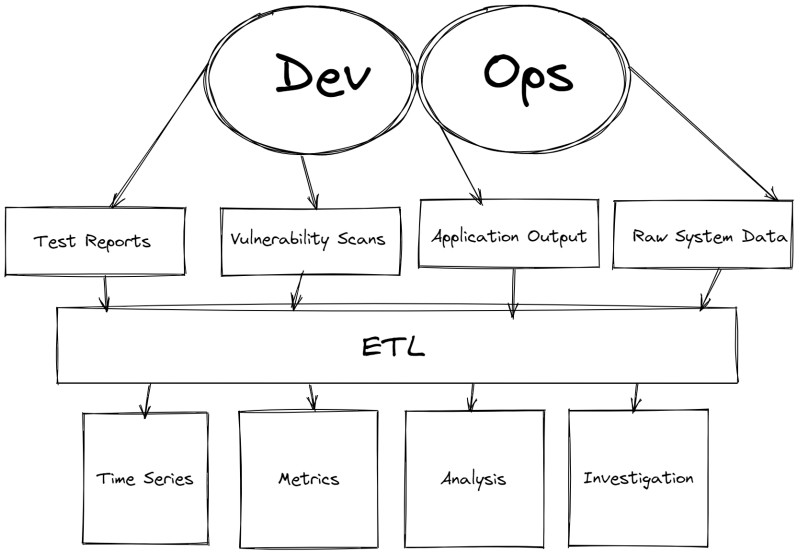
- **What has happened?**
  - Analyze and interpret system outputs
  - Identify events and issues, then notify/react
  - Generate metrics
- **What is happening?**
  - Calculate workflow, establish timelines, identify system patterns
  - Establish a baseline for system activity
  - Detect anomalies
- **What could happen?**
  - Identify potential gaps
  - Perform monitoring and systematic analyses
  - Identify, define, and detect threshold limits
  - Predict and prevent

The introduction of AIOps can greatly streamline operations, improve system visibility, and hasten the detection and response to problems. All of which increases the likelihood of greater uptime, better code quality, reduced security risks, and even deeper system knowledge. Wins all around! Let's dive into each discipline as an example of how they can impact IT operations.

Data Analytics and Operations

"Data analytics" is a very broadly used term these days. It mainly refers to concepts in data science focused on the examination and interpretation of one-to-many datasets to both identify events and establish trends. It is one of the many tools AIOps can deploy to improve your DevOps. As you can probably surmise, the ultimate end goal of applying AIOps is to gain actionable insights into system activity, status, health, and security. Data analytics can help accomplish this by generating metrics, observing workflows, and performing trend analyses.

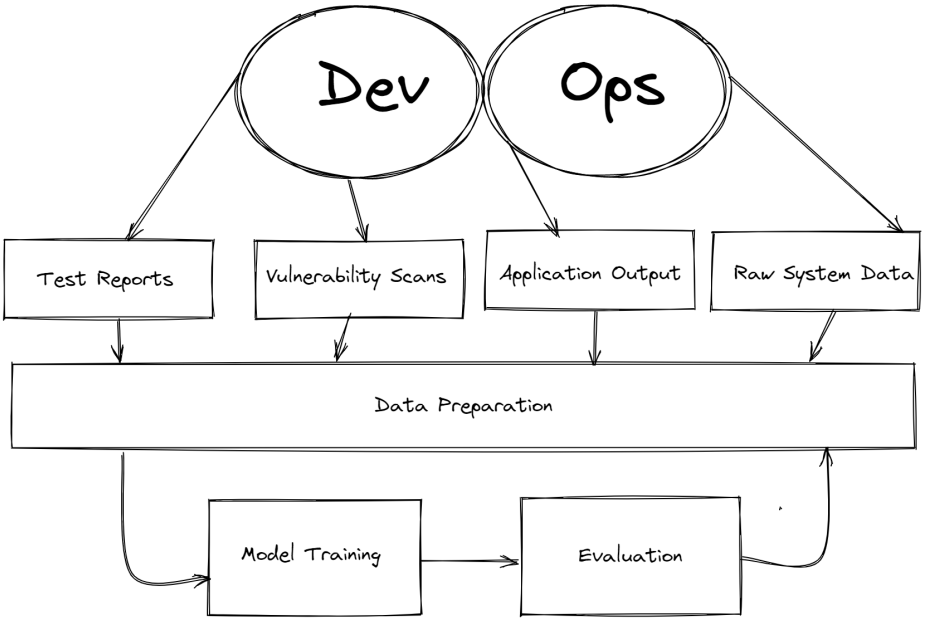
One of the initial opportunities to introduce data analytics into the DevOps cycle is through data aggregation and metrics calculation.



There are several advantages observed here:

- Raw data is flowing into the Extract, Transform, Load (ETL) layer. These analytics can do the tedious work of parsing that data into an understandable format for both humans and machines.
- Data aggregation enables the ability to analyze multiple datasets in a single view or analytic.
- Calculation of metrics can be used to generate baseline information used in the development of models for more advanced analytics and data science techniques.
- Time series analysis across all system components provides a richer and fuller view into the system status as a whole.
- Analytics can even be deployed in the investigation of datasets to identify potential issues.

Machine Learning and Operations



Machine learning is a subset of artificial intelligence. In machine learning, machines are trained to replicate a general pattern and adapt it to the situations they encounter. As you can see in the image above, data flowing from the DevOps phases now flows into a data preparation area instead of a simpler ETL stage. This includes data normalization, transformation, validation, and any other steps necessary to generate the desired dataset. This data is used to train models that will later be deployed for evaluation. That evaluation of the result enables the end user to iteratively refine the model, deploy once more, and verify its results. This feedback loop enables the model to more accurately process data and evaluate information as it enters data preparation and analysis stages of the lifecycle.

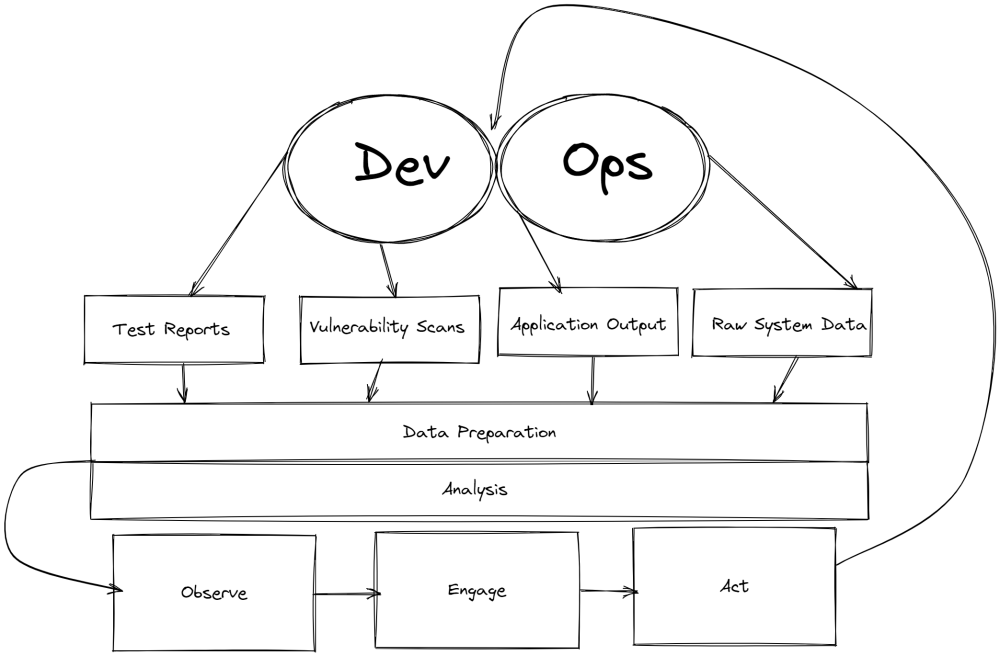
Machine learning can enable a DevOps system to track behaviors and detect anomalies by leveraging strategic models. Properly trained models can be used to do many things, such as:

- Monitor system thresholds
- Measure orchestration efficiency
- Identify system usage patterns
- Analyze threat potential
- Alert on and triage defined issues

As the accuracy of those models becomes more complete, the system's normal baseline can be established. As events outside of the baseline are observed, system owners can be notified and issues can be remediated.

AI and Operations

Now that we have at least an extremely high-level understanding of data analytics in machine learning, let's talk about AIOps more specifically. AIOps allows systems to ingest information and make decisions without human intervention, exposing the power of machine-based learning and adapting. In terms of DevOps, this is where we explore the ability to proactively prevent and take action to remediate issues automatically. AIOps couples the AI and ML with the power of automation, enabling DevOps to detect anomalies, eliminate noise, and discover patterns.



As displayed in the image above, in AIOps, three main phases are considered:

- **Observe:** The analysis of real-time and historical data to correlate and contextualize information across datasets. This stage can aggregate data points for a system and performance baseline, perform anomaly detection, and establish trend definitions.
- **Engage:** IT service management integration. This phase is used to assess risk and service insights for additional analysis, remediation, and tracking.
- **Act:** Automation of remediation through application release automation, scripts, runbooks, etc.

Together, these stages provide a feedback loop of information, enabling a pattern of continuous integration, delivery, and especially refinement. This design can have a dramatic impact across system components, such as:

- **Systems resilience and reliability:** ML, trained using historical and trend data, can predict and prevent unexpected outages and identify potential breaking points across system usage and releases.
- **Critical alarm visibility:** AI can be used to perform anomaly detection, identify patterns, and suppress events. ML can also be used to identify anomalies, but these techniques can produce a lot of noise (false positives and low risk, yet highly active events that are observed within normal baseline operations). AI can reduce that noise through historical and real-time event analyses to identify and suppress events that fall within the normal baseline while exposing critical alarms for remediation.
- **Cost reduction:** Data analytics can be used to identify infrastructure usage patterns to scale appropriately for expected and actual system utilization.

AIOps, if you can't tell by now, isn't just one thing. It's a myriad of techniques. An arsenal of strategies. A toolbox for anyone looking to supercharge the observability of their software. The trifecta of data analytics, machine learning, and AI, coupled with tried-and-true DevOps and DevSecOps practices, can enable humans to finally harness the learning power of machines and unleash it for the good of software developers everywhere.

REFERENCES

1. https://www.ibm.com/cloud/learn/observability#:~:text=In%20cloud%20computing%2C%20observability%20also,the%20application%20to%20meet%20customer
2. https://www.gartner.com/en/information-technology/glossary/aiops-artificial-intelligence-operations
3. https://dzone.com/articles/life-cycle-of-devops
4. https://www.bmc.com/blogs/why-event-noise-reduction-and-predictive-alerting-are-critical-for-aiops/
5. https://devops.com/how-devops-powered-by-ai-and-machine-learning-is-delivering-business-transformation/#:~:text=In%20DevSecOps%2C%20there%20has%20always,use%20the%20data%20at%20hand
6. https://assets.ctfassets.net/vtn4rfaw6n2j/3i4mfSI9mGaeB4FCfBjgjV/7a4c47abcd6b87dbb641bb208ada93e1/T87D27_-_Maria_Nichole_Schwenger_-_2020_DevOps-World-DevSecOps_-_An_Ideal_Use_Case_For_Applying_AI.pdf

Stephanie Phifer, CTO at EITR Technologies

@BellaNotte1986 on DZone | @stephanie-phifer-7 on LinkedIn | https://eitr.tech

Stephanie Phifer is CTO and founding partner of EITR Technologies. Her most recent focus has been Cloud-based Enterprise Big Data Systems and Analytics supported by robust Automation and Orchestration capabilities. When she is not being a complete nerd, she can be found jamming with her favorite local musicians or “momming” for her wonderful husband, two amazing daughters, and their pets.

CONTRIBUTOR INSIGHTS

Repository-Based CI/CD Pipelines

How GitHub Actions and GitLab CI/CD Integrate With Application Code

By Justin Albano, Software Engineer at IBM

Configuration-based continuous integration (CI) and continuous delivery (CD) has been available for many years through tools such as Travis CI and CircleCI, but in the past few years, both GitHub and GitLab have created their own tools that allow us to create configuration-based CI/CD pipelines directly within our repository.

These tools incorporate many of the benefits of external CI/CD tools — such as creating secure pipelines, cloud-specific pipelines, and executing various test cases, like Test-Drive Design (TDD) tests — but integrate them directly into our repository. In this article, we will look at the concept of repository-based CI/CD pipelines, as well as two of the most popular tools in this ecosystem: GitHub Actions and GitLab CI/CD.

Repository-Based CI/CD

Repository-based CI/CD is identical to existing third-party tools, but they allow us to configure our pipelines using a configuration file that resides with the rest of our code. In practice, this means fewer accounts, less external integration, and less friction (since our CI/CD pipeline is a first-class citizen in our repository and not an add-on).

For example, prior to repository-based CI/CD, we would create an account with a third-party tool — such as CircleCI — and then integrate our CircleCI account with GitHub and provide the necessary permissions. Then, we would then create our configuration file in the location (in our repository) specified by CircleCI. Lastly, upon check-in, CircleCI would hook into our GitHub repository and execute our pipeline. The reports and resulting artifacts would then be stored on CircleCI.

This requires us to manage two separate tools: (1) GitHub for our application code and (2) CircleCI for our pipeline and build artifacts. With the introduction of tools like GitHub Actions and GitLab CI/CD, we can manage our application code, CI/CD pipelines, and artifacts all in one location.

GitHub Actions

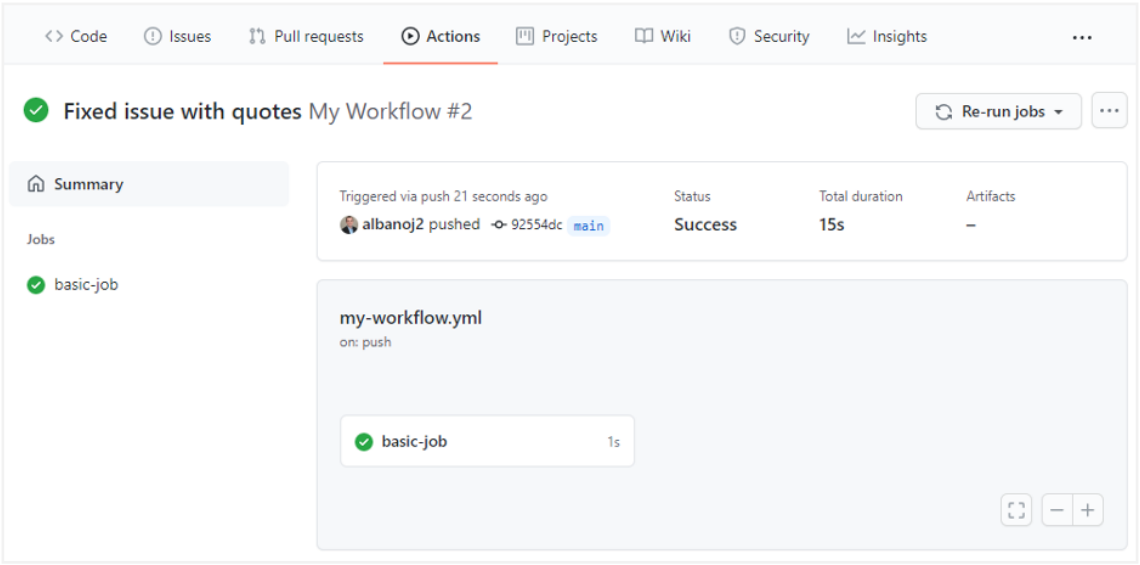
To create a CI/CD pipeline in a GitHub repository, we create a Yet Another Markup Language (YAML) file in the `.github/workflows/` directory of our repository. Each `.yml` file we create in this directory creates a new workflow. In practice, each workflow can be thought of as a pipeline that will be completed during a build. For example, we can create a workflow, `.github/workflows/my-workflow.yml`, with the following configuration:

```
name: My Workflow
on: [push]
jobs:
  basic-job:
    runs-on: ubuntu-latest
    steps:
      - run: echo "My first workflow"
```

This workflow defines the following:

- `name`: The name of the workflow displayed on GitHub
- `on`: The event that triggers the workflow (in this case, a push to the repository)
- `jobs`: The jobs that will run in the workflow
- `basic-job`: A single job we defined (where the key, `basic-job`, is the job name)
- `runs-on`: The Docker image the job should run on
- `steps`: The steps in the job

Once we push a new commit to our repository, we will see our pipeline executed and the results of the execution displayed under the **Actions** tab of our repository:



This simple example merely prints a line to the console and only scratches the surface of what GitHub Actions can do. GitHub also provides prepackaged steps called Actions that allow us to execute complex steps in a single line. For example, instead of writing the code to checkout our repository, we can use the Checkout Action. For more information on GitHub Actions, see the GitHub Actions official documentation and Quickstart.

GitLab CI/CD

GitLab CI/CD uses a similar format to GitHub Actions, but instead divides a pipeline into stages and steps. Each stage, such as build or test, can contain one or more steps, and each step contained in a stage is executed in parallel. To create a new pipeline, we create a `.gitlab-ci.yml` file in the root of our repository. For example, we can create the following `.gitlab-ci.yml` file:

```
build-job:
  stage: build
  script:
    - echo "Building something..."

test1:
  stage: test

  script:
    - echo "Testing the first thing."
```

Code continues on next page

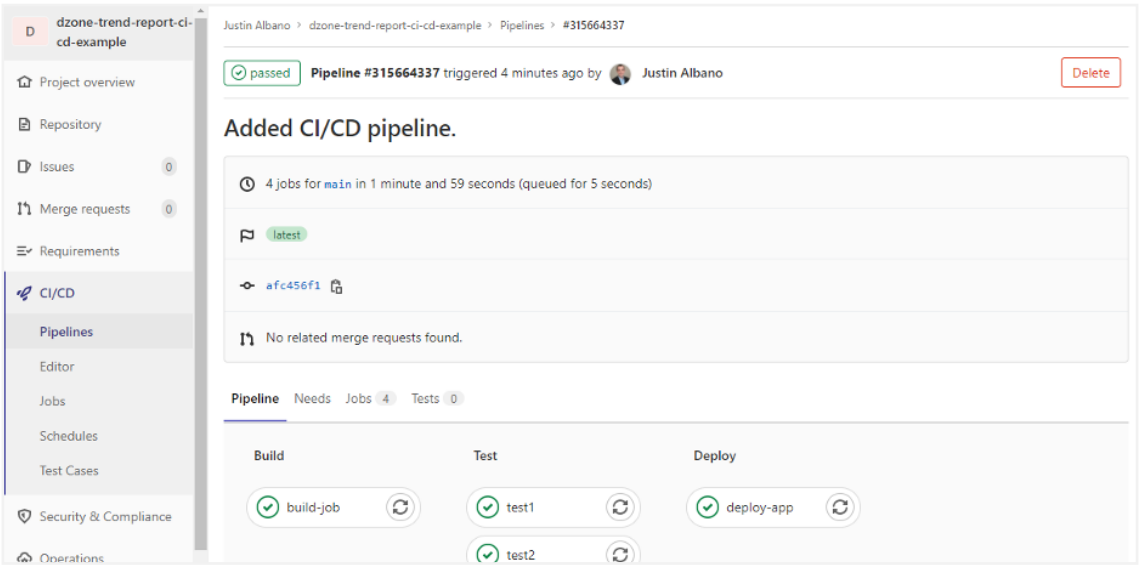
```
test2:
  stage: test
  script:
    - echo "Testing the second thing"

deploy-app:
  stage: deploy
  script:
    - echo "Deploying the application somewhere"
```

Using this configuration, we define the following:

- `build-job`: A job that executes during the `build` stage
- `test1`: A job that executes during the `test` stage
- `test2`: A job that executes during the `test` stage
- `deploy-app`: A job that executes during the `deploy` stage

When we commit a change, we can see the following pipeline execution on GitLab under **CI/CD > Pipelines**:



Similar to GitHub, our example is merely the peak of the iceberg of what GitLab CI/CD is capable of. For more information, see the GitLab CI/CD official documentation and Quickstart.

Disadvantages

While repository-based CI/CD pipelines are convenient and have numerous advantages over their third-party counterparts, there are still some disadvantages to be aware of:

1. **Interoperability:** There is no standardized language for defining pipeline configurations and, therefore, a pipeline defined on GitHub will not work in GitLab (and vice-versa). When creating a repository-based pipeline, it is important to follow the specific steps for each repository.

2. **Limited flexibility:** Repository-based pipelines are still maturing, and there are times when a capability is not yet supported. Although not as simple as built-in features, both GitHub Actions and GitLab CI/CD support the execution Bash commands, which allows nearly unlimited flexibility when defining our pipelines. Additionally, as pipelines are defined for larger projects, it may become necessary to have external build machines run the pipeline instead of utilizing GitHub or GitLab servers (see GitHub Actions Runners and GitLab CI/CD Runners).

Conclusion

CI/CD pipeline tools have been around for more than a decade, but in the last few years, we have seen two of the most popular repositories — GitHub and GitLab — integrate CI/CD directly into their tool suites. While these tools continue to mature, they have reached a point that makes them viable additions to most projects and an important asset to keep in our development toolbox.

Justin Albano, Software Engineer at IBM

@albanoj2 on DZone

Justin Albano is a Software Engineer at IBM responsible for building software-storage and backup/recovery solutions for some of the largest worldwide companies, focusing on Spring-based REST API and MongoDB development. When not working or writing, he can be found practicing Brazilian Jiu-Jitsu, playing or watching hockey, drawing, or reading.

CONTRIBUTOR INSIGHTS

Modern CI/CD Security Landscape

An Assessment of Common Challenges and Solutions

By Sudip Sengupta, Technical Writer at Javelynn

Introduction

Businesses that rely on IT-enabled innovation have long been in a marathon of consistently discarding underperforming legacy technologies and adopting emerging ones. Shifting to adopt a DevOps culture is one such choice that helps organizations eliminate operational silos between development and operations teams.

By adopting DevOps practices such as continuous integration and continuous delivery, organizations can deliver releases faster by forming a culture of efficient collaboration and automation.

THE RISE OF CI/CD ADOPTION

Given the increasing shift toward cloud deployments, future trends are expected to highlight a wider adoption of CI/CD processes. Though the adoption of CI/CD continues to be on an upward trend, a Mabl survey discovered that of the surveyed organizations, 53% use continuous integration while 38% use continuous delivery. Additionally, at least 25% of surveyed companies were in the process of adopting CI/CD.

While this endorses the industry's maturity and increasing quest to adopt best practices, it would be equally interesting to know how the industry faces CI/CD's most common challenges. A successfully implemented pipeline not only delivers application code faster but also ensures that post-implementation disruptions are countered diligently.

Inefficient governance, poorly configured pipelines, siloed team structure, and external security threats are most often potential disruptors to an efficient application workflow. Though such challenges may cause a considerable impact on an application workflow, mitigating security threats is often considered the biggest challenge. In this article, we assess the CI/CD security landscape and best practices to proactively counter security pitfalls.

Modern CI/CD Security Landscape

To automate every aspect of a workflow, the CI/CD pipeline accesses every resource within its technology stack. As a result, CI/CD pipelines are more vulnerable to attack vectors that have the potential to impact every stage of the development lifecycle. CI/CD pipelines, in general, are prone to malicious attacks as a result of the following security pitfalls:

TRUST RELATIONSHIPS

Interaction and communication between source code repositories and application servers in a CI/CD pipeline are dependent on a *trust relationship*, which hackers target to exploit for making malicious changes in the code — and commit to the main branch. Such unauthorized changes can cause extensive damages to the organization's current operations as well as future releases.

WORKFLOW CONFIGURATION ERRORS

Another source of vulnerability in a CI/CD pipeline occurs due to setting misconfigurations in a given workflow. Such misconfigurations not only include poor settings in the existing workflow but also relate to miscalculating the existing security landscape of the entire industry.

VULNERABILITIES IN MICROSERVICES

Embracing a microservices-based framework using containers and orchestration tools provides immense benefits such as operational efficiency, high availability, and enhanced scalability. However, the inclusion of such tools widens the *attack surface area* by opening more susceptible points. More so, due to the open-source nature of these tools, security vulnerabilities are difficult to contain and often give rise to an elaborate and planned attack.

AUTHORIZATION VULNERABILITIES

One of the most widely exploited components of a technology stack is *access credentials*. Misusing inadequately secured *access secrets*, attackers commonly abuse privileges to gain access into systems and, consequently, launch deeper strikes to affect the entire stack.

POORLY WRITTEN CODE

Poorly written code is a common enabler that introduces susceptible points into an application workflow. This aspect highlights not only poorly written code but also an improperly administered testing phase. Inadequately scanned code modules that integrate into the application codebase not only impact the software's performance but also throw open susceptible entry points for attack vectors.

Best Practices for Securing CI/CD Pipelines

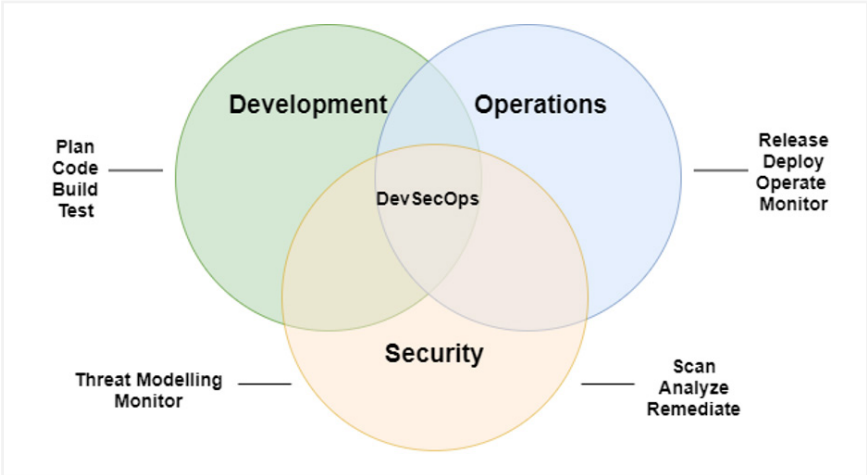
While securing an application workflow requires a holistic approach to administering security across all layers of the tech stack, here are the essential best practices to focus on when securing your CI/CD pipeline:

SHIFTING LEFT FOR SECURITY

Instead of treating security in the later stages of the SDLC, security measures should be integrated from the initial phases of the application development process. This forms a robust framework that continuously tests for vulnerabilities from the early phases of a development cycle. Doing so not only allows teams to identify and fix code vulnerabilities before they are deployed but also introduces a cost-effective model by reducing the time to delivery.

EMBRACING DEVSECOPS

DevSecOps is a security-oriented approach that integrates security with the development and operations stages of the SDLC. A DevSecOps model closely relates to "shifting left," as it involves automatic integration of the application and infrastructure security across every stage of the development lifecycle. With DevSecOps, an organization achieves an iterative, yet streamlined, workflow of faster application builds and hardened security.



ACCESS AND AUTHORIZATION

Implement clearly defined access control measures to determine *who* has access to *what* resources of a pipeline. Additionally, teams should consider grouping secrets at the access level while ensuring they are not shared internally.

As a recommended best practice, secrets should not be hardcoded into CI/CD configuration-related files to avoid a full-blown abuse by attack vectors. Besides these, organizations can also leverage third-party platforms that enforce rigorous security measures by implementing zero-trust, short-lived secrets that guard sensitive components of the tech stack.

IMPLEMENTING INFRASTRUCTURE AS CODE

Infrastructure as Code (IaC) is the management of large-scale, distributed infrastructure systems through machine-readable files. This approach allows the creation, deployment, and testing of infrastructure components across cloud platforms using any language. To harden the security of CI/CD pipelines, IaC allows teams to automatically provision security, audits, and compliance policies efficiently.

These can be further divided into:

SECURITY AS CODE

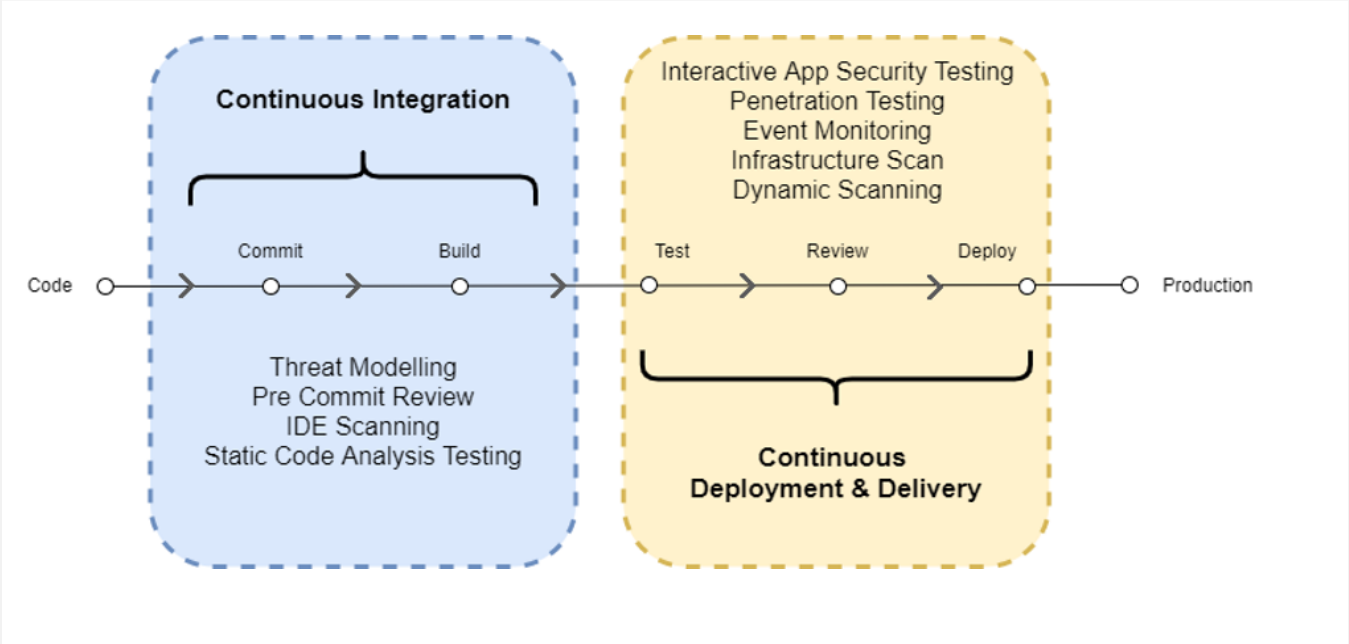
Security as Code (SaC) enforces security by examining how infrastructure changes are currently made in the SDLC and identifying vulnerable points. This benefits teams by allowing them to **automate feedback loops**, **set up monitoring**, **enforce vulnerability scanning**, and **implement security policy checks** through efficient, machine-readable code.

POLICY AS CODE

Similar to SaC, Policy as Code (PaC) is the process of using code to automate governance and policy enforcement across all layers of the infrastructure. Using PaC, regulatory compliances such as SOC2, HIPAA, and GDPR can be administered as part of a workflow module that outlines guidance on **data security**, **infrastructure integrity**, and **access control**.

SECURITY TESTING

While enforcing governance and validation checks are critical success factors to enable a robust framework for a secured CI/CD pipeline, organizations must plan to test code using security-focused tools and processes. These include:



Administering security in a typical CI/CD workflow

SAST

Static Application Security Testing (SAST) is a methodology to detect vulnerability sources within libraries, source code, and byte codes. Often referred to as the **white-box testing** approach, SAST is mainly used to spot code flaws in pre-deployment stages. Some popular SAST-based tools include Fortify Static Code Analyzer, Veracode, and Coverity Scan.

DAST

Dynamic Application Security Testing (DAST) involves examining an application for vulnerabilities while the application is running in production. By simulating an environment in the same way that a potential attacker could attack the system, DAST provides an efficient approach to identifying vulnerabilities of the entire tech stack. Some popular DAST-based tools are NetSparker, Tenable.io, and HCL App Scan.

MONITORING

With real-time monitoring of events, organizations can prevent large-scale cyber attacks by employing a *continuous monitoring* approach. While source codes are already tested and have met compliance checks, efficient monitoring enables a **reactive threat intelligence mechanism** that prevents newly introduced vulnerabilities from attacking a workflow.

Final Thoughts

One key aspect of a DevOps model is that it enables *continuous automation* across multiple layers and teams. While CI/CD pipelines continue to be key components of a DevOps-focused, cloud-native ecosystem, they come with their own sets of challenges. With respect to security, CI/CD pipelines are primarily vulnerable due to their interconnected nature. Such a framework allows vectors to replicate an attack at a much larger scale fairly quickly. To deal with this, it is of utmost importance that organizations adopt security best practices that counter potential vulnerabilities across all layers of an application workflow and its underlying infrastructure.

Embracing DevSecOps, implementing rigorous security testing, and monitoring events are some of the best practices that organizations can adopt to get the most out of a DevOps culture — a model that is efficient and secure at the same time.

Sudip Sengupta, Technical Writer at Javelynn

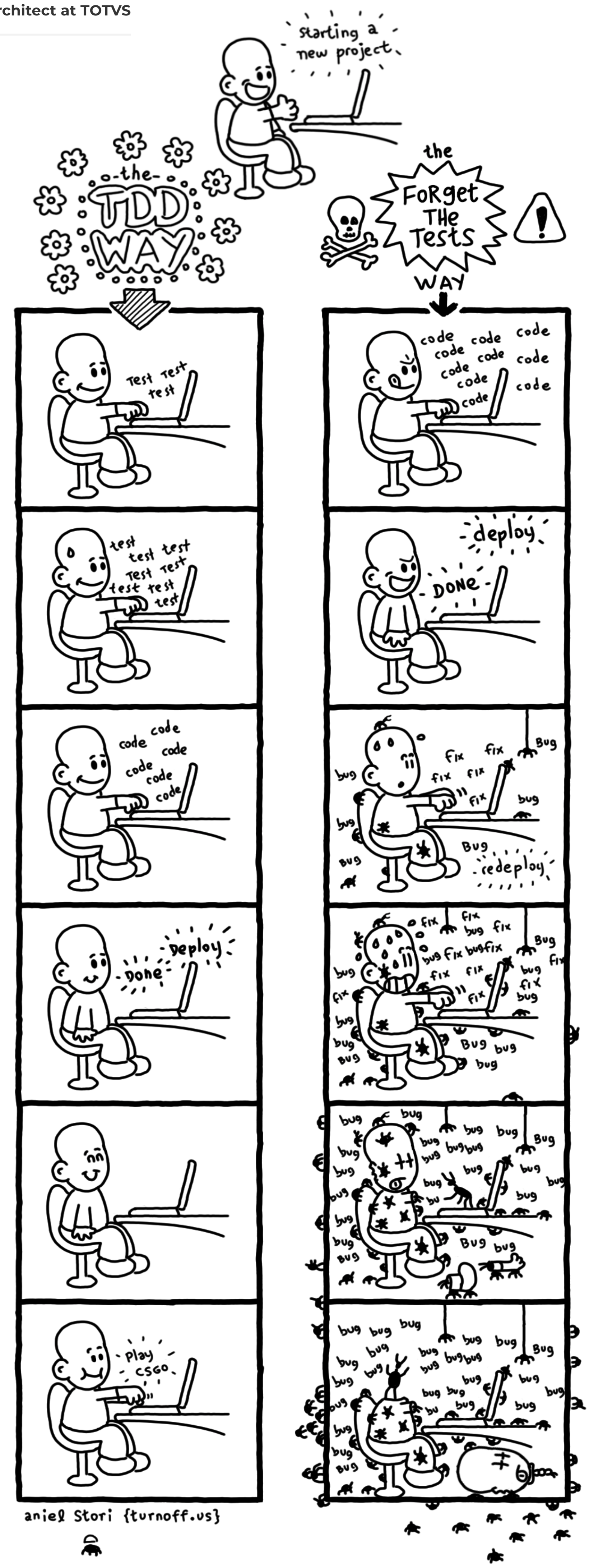
@ssengupta3 on DZone | @ssengupta3 on LinkedIn | www.javelynn.com

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

TDD vs. FTT:

Follow the Parallel Lives of a Programmer Using TDD vs. Jumping Straight to Deployment

By Daniel Stori, Software Architect at TOTVS



CONTRIBUTOR INSIGHTS

Infrastructure as Code: Where and How to Automate?

By Alireza Chegini, Senior DevOps Engineer at Coding as Creating

Introduction

Today, most companies are running their infrastructure on a public cloud, which means any infrastructure resource can be created via the UI or running a script in a matter of minutes. This emerging technology has changed the way companies handle hardware responsibilities. On the one hand, it becomes so much easier to create infrastructure, and software teams can do that without being dependent on other companies.

But on the other hand, infrastructure in the cloud still needs to be created, maintained, and supported by the organization. This creates extra work for teams, which can lead to high complexity if not handled properly. In this article, I am going to cover infrastructure as code and infrastructure automation explanations and recommendations.

Reasons to Automate Infrastructure

Whenever we used to talk about the software development lifecycle, the scope was only focused on the software level. So implementation was only used for applications, and hardware was something separate. As the majority of hardware was physical, IT administrators took care of it. But then servers became virtual, and physical hardware could be used for different projects and purposes. This was the moment when IT administrators started to use software to manage infrastructure provisioning. Thus, public clouds emerged, and the idea of creating infrastructure with just a few clicks was born.

Today, infrastructure isn't something that system engineers do — developers and other professionals handle it without in-depth knowledge. It becomes so much easier for developers to spin up a virtual server or create a cloud resource to test a piece of code. They can then create infrastructure either from the UI or running a script.

Scalability, maintainability, and high availability are the things that architects consider upfront during the design process so that there are no surprises. Developers are able to test scalability and high availability on the development environment so that they can simulate the production environment and adjust the system configurations accordingly based on requirements and test results.

Having said that, now infrastructure is part of the software development pipeline, which means it is created as a part of each software deployment. In the past, infrastructure was created at the end of the project, when everything was ready to deploy in production. Today, teams think of infrastructure from the beginning because they need infrastructure to test their code. Therefore, whenever you want to deploy an application, API, or service, there must be infrastructure in place *first*. Then, you can deploy your application and start testing it.

With infrastructure automation, teams can maintain infrastructure as code in a repository, and with that, anyone can create and improve infrastructure along with the application being developed. It becomes easy to roll back the infrastructure to the previous state — and losing part of your infrastructure is not as scary anymore now that you can replace it with an alternative in a matter of seconds.

Automation Tooling Challenge

When a team agrees on moving toward infrastructure as code and infrastructure automation, it is then time to think about *how* to implement IaC. Implementing IaC can often depend on your cloud provider. Specific tools are beyond the scope of this article, but it is important to consider the following criteria when considering infrastructure automation tooling.

First, we should always focus on picking a tool to help us automate infrastructure as fast as possible. This means that the tool is easy to understand and learn for all team members, and everyone should feel comfortable working with and maintaining it. Additionally, a tool provider should offer enough support in case you become stuck somewhere or have issues with the implementation. These days, there are various tools that can be used to automate infrastructure, including Terraform and Ansible.

As an example, you can use Microsoft Azure's Azure Resource Manager (ARM) or Amazon Web Services' CloudFormation to build templates to create your infrastructure. But keep in mind: Every provider has its own tool. Based on your application and cloud provider, there are different options to consider. Consider the criteria for each tool and select the one that can work best for your team.

Here are some important considerations:

- Should it be an open-source or paid tool?
- Is it easy to learn for all team members?
- Should it be cloud agnostic or not?
- Does it need to meet specific security compliance regulations, like GDPR?
- Is there enough documentation and support for this tool in case there is an issue?

Infrastructure Automation: Where to Start?

When we talk about automation, we mean that we do not want to get stuck with undocumented or manual tasks that only a few people know about or can do. Automation helps establish a process in which we can create infrastructure with all required configurations in the quickest way possible.

By automating infrastructure, teams should be able to create that infrastructure whenever needed. We should be able to rerun processes as many times as we want, and the results should always be the same. This makes us confident that we have a reliable process for infrastructure creation. If a disaster strikes, we should be ready to create our infrastructure and recover our data.

When thinking about infrastructure automation, we need to have a good understanding of the infrastructure we need to build for our application. Let's put this into a few simple steps:

1. **First try manually:** In this step, the aim is to try out the infrastructure creation, see how features are configured, and figure out what needs to be configured to reach your desired outcome.
2. **Document all manual steps:** As soon as you have finished the infrastructure and it is as expected, document all steps so that the process is clear each and every time you or any team member need to create that infrastructure. This also provides full details on the infrastructure configurations that will be helpful for automation.
3. **Test the document:** This may sound weird, but it is very important. You should be able to follow the document closely to create the infrastructure. Since this document is going to be used for automation later on, it is crucial to have accurate information regarding the steps and configurations.
4. **Automate all steps:** Once you have documented all steps, automation shouldn't be difficult. In this step, your goal is to figure out how you can turn the manual installation into an automated one. This can be done either using a script or an automation tool.
5. **Orchestrate the whole process:** In this step, you will create a pipeline and add all automated steps in a sequential order. If there are steps that can be executed in parallel, configure that in the pipeline. This will allow you to have everything in one place.

6. **Test and tune:** This is the final step where you test the pipeline several times to make sure you get the same results for each execution. Now, review all steps and improve where possible.
7. **Congratulations!** You now have an automated infrastructure. Remember: The mission is not complete since this is a continuous task, and you want to ensure that your infrastructure stays up to date.

When *Not* to Automate

Automation is great in concept, and these days, we are obsessed with automating everything. However, this does not mean automation is the right choice for every situation. Sometimes, it comes with a heavy price. Instead of making things easier and more efficient, it can cost more time and money than it is worth — and does not necessarily bring a return on investment.

Some might say this is not for discussion because we need to automate everything. While that is a valid point, do not forget that automation is not always a high priority for project managers. Sometimes, there is not enough universal acceptance or budget for automation tasks. If you are working on any project where automation cost is a concern, it is crucial to maintain balance to keep automation tasks alongside high priority tasks. See which infrastructure resources are required for a project and how often we need to create them.

For example, having a clear, documented process should be enough for every team member to read and do their job. Instead, you can use the time you have to focus on your actual platform-related infrastructure and automate those components. With this approach, you might be able to prioritize automation from high priority to low priority, which means you automate everything gradually. So instead, you can focus on identifying which parts need to be automated earlier and which parts can be done later.

One easy way to recognize this is to see how often you do infrastructure operations manually. You might look at your backlog and identify which tasks are repeating every week or even every day. These tasks should be automated as a higher priority than the ones you do per project or that happen less often.

The Bottom Line

Today, we are living in an era in which you not only code your application, but also the infrastructure on which to run it. It is amazing to have everything as code. So, within your CI/CD pipeline, you can create the infrastructure from scratch, then deploy your application there.

Infrastructure as code is a newer member of the software development lifecycle that is gradually being recognized by companies. It is a shared responsibility among all members of a team, not just DevOps engineers. That is why infrastructure as code has become a high-priority task in software projects, pushing software engineers to learn more about infrastructure alongside DevOps engineers.

Helping team members use their knowledge and experience to automate infrastructure, just like with software integration and deployment, IaC is one of the more interesting outcomes of implementing a DevOps culture. There is no separation between software, hardware, or even team members. One team is responsible for designing, building, and supporting the platform.

Alireza Chegini, Senior DevOps Engineer at Coding as Creating

@allirreza on DZone | @alirezachegini on LinkedIn | https://codingascreating.com

Alireza has more than 20 years of experience in software development. He started this journey as a software developer and continues working as a DevOps Engineer. In the last couple of years, he has been helping companies move away from traditional development workflows and embrace a DevOps culture. These days, Alireza is coaching organizations as an Azure Specialist in their migration to the public cloud.

CONTRIBUTOR INSIGHTS

Common Security Challenges in CI/CD Workflows

Rajeev Bera, Founder at aCompiler

Security is not a luxury anymore; it is a necessity. Now, we must treat security as a first-class DevOps citizen — and no longer ignore continuous security in continuous integration and deployment workflows.

What Is CI/CD Pipeline Security?

CI/CD pipelines automate your software delivery process and provide standardized feedback loops through automated channels that enable fast product iterations.

Pipeline security is nothing but a process for executing security rules and standards. CI/CD pipelines have access to different resources from development to the production environment. Code passes through the CI/CD structure and ensures security throughout the entire process to deliver quality code.

Because of potential data leaks, CI/CD pipelines are an engaging target for threat actors. So it becomes more and more important to keep your CI/CD pipelines secure.

In CI/CD pipelines, security can correlate to hundreds of different components. To make it more comprehensive, we can break CI/CD pipeline security into two distinct parts:

1. **Security of a CI/CD Pipeline**: This includes secure login, secure storage of build artifacts, authentication, etc. This type of security is pretty straightforward and easy to implement.
2. **Security in a CI/CD Pipeline**: This can be more challenging, and the Dev team needs special skills to add security into CI/CD workflows. For secure and continuous delivery, the first step is to conduct a risk evaluation. This exercise allows you to outline a relevant security posture.

Let's now introduce some common security challenges in CI/CD workflows and how to deal with them.

Common Security Challenges in CI/CD Workflows

INEFFICIENT IMPLEMENTATION

Lack of expertise can lead to the inefficient implementation of CI/CD pipelines. Development teams can automate the wrong processes, misconfigure CI, and even mix Continuous Delivery with Continuous Deployment.

Additionally, CI/CD tools are relatively new with limited expertise available in the market, prompting numerous challenges for CI/CD implementation.

RECOMMENDED SOLUTIONS

- Assign roles to individuals for each stage of the CI/CD pipeline.
- Participate in training and coaching to help implement CI/CD pipelines more effectively.

CONFIGURATION ERRORS

Configuration errors count for several security gaps. Along with basic misconfiguration, a key challenge is the lack of enabling different security features as they are released.

You should safely store your secrets and sensitive data similar to code signing keys. There should be an additional layer of isolation between your encrypted files and the repository.

RECOMMENDED SOLUTIONS

- Implement constraints on who can set or change pipeline configurations.
- Conduct a simple, regular check for overall configuration changes. The more frequent the check, the smaller the vulnerability window.

FALSE POSITIVE

There are many security tools available for CI/CD pipeline workflows. Most of the time, security tools suffer from false positives and/or false negatives. False negatives are more dangerous because they point to a false sense of security.

When you integrate a tool into your CI/CD pipeline without the necessary preparation and knowledge, false positives pose a significant challenge.

RECOMMENDED SOLUTIONS

- Identify any errors and mark as a false positive with the proper justification.
- Prevent the CI/CD pipeline from flagging the error again and again.

FLAWED AUTOMATED TESTING

In a CI/CD pipeline, flawed testing systems could be a nightmare for developers. For the high-quality build, CI/CD pipelines use testing in their automated system.

But if a test fails, there is a security risk for deploying faulty code. Also, for an inaccurate performance load test within your CI/CD pipeline, the final product might be filled with performance vulnerabilities.

RECOMMENDED SOLUTIONS

- Take every test flag seriously and only dismiss warnings if the test is fixed.
- Attempt misusing your pipeline by ensuring that your tests are entirely up to date for the type of application being deployed.

LIMITED ENVIRONMENTAL CHALLENGES

The testing team often contains limited resources to test the code. In such scenarios, testers tend to lean towards a shared testing environment.

For CI/CD pipelines, a shared testing environment is not always the best choice due to the need for different configurations for different tests. Poor configuration of a shared testing environment often leads to many failed tests and buggy deployments.

RECOMMENDED SOLUTIONS

- Create an on-demand test environment in the cloud. This is one of the easier and more effective options.
- Selenium automation testing could alter CI/CD difficulties that testing teams face, like limited testing environments or incompetent implementation.

MULTIPLE PIPELINES

When you scale your pipelines, pipeline best practices and management will either take you to various stages or bring you to a dead-end, and more deployment means more releases. With this comes added dependencies, which can be challenging.

In software terminology, compliance is the set of regulations believed to be industry best practices and processes. Developers need to ensure application compliance by taking responsibility for building it into the development process itself.

RECOMMENDED SOLUTIONS

- Introduce compliance into the ecosystem with developers and testers. This will help teams identify compliance flaws earlier in the process and fix bugs sooner. Additionally, it will decrease the codebase through standardized code being deployed each time.

COMMUNICATION

When you work on a CI/CD pipeline, there are different teams with varying responsibilities.

If the automated build output shows an error and the developer fails to communicate the details with other teams, there could be severe consequences. To avoid such scenarios, human communication plays an important role.

RECOMMENDED SOLUTIONS

- Avoid the main, human-prone obstacles that don't rely on automation — human communication, collaboration, and teamwork.
- Adopt transparency and strong communication skills as the primary keys for CI/CD pipeline workflow success.

Conclusion

If you use your time wisely and implement the right tool in your CI/CD pipeline, you will likely never need to worry about security. However, it is essential to understand that security tools are different. They are not like any other tool that you can use as plug-and-play. And to understand those complexities, these tools need some extra attention.

Speed and convenience are the main goals for CI/CD pipelines. Consequently, you should never ignore security. Keeping security in mind throughout development, you can avoid a lot of damage if and when a threat occurs.

Most CI/CD security challenges are due to poor security hygiene and inadequate implementation. Accurately evaluating CI/CD requirements and picking the right tools is essential. By precisely configuring CI/CD pipeline stages, teams will generate a successful and secure CI/CD implementation.

Rajeev Bera, Founder at aCompiler

@acompiler20 on DZone | @acompiler_com on Twitter | https://acompiler.com

Rajeev Bera is an enthusiastic IT Professional and founder of the aCompiler.com blog. The "aCompiler" is a place of next-level learning and training for your IT skills.

CONTRIBUTOR INSIGHTS

Automate or Not-o-Mate

The Who, What, When, Why, and How of Automation

By Judy Johnson, Software Engineer at EITR Technologies

If you ask a group of engineers how to "DevOps," many will say to automate everything. But DevOps is so much more than that. In this article, I discuss why automation is important, how it relates to DevOps, and when automation may or may not be the right tool for the job.

What Is Automation?

So what is automation? According to Wikipedia: "Automation or labor-saving technology is the technology by which a process or procedure is performed with minimal human assistance."

Automation is about using technology to perform or simplify processes that need to be completed, regardless of every day or on specific occasions. Or as I like to say, it's a way to get computers to do the work I don't want to do, and with minimal supervision — if this is the case, I'm all in.

Where Does Automation Fit Into DevOps?

According to Wikipedia, "DevOps is a set of practices that combines software development (Dev) and IT operations (Ops). It aims to shorten the systems development lifecycle and provide continuous delivery with high software quality. DevOps is complementary with Agile software development; several DevOps aspects came from Agile methodology."

And why do I like DevOps so much? As a "peace, love, and happiness" kind of person, I appreciate the emphasis on communication and cooperation. As a team lead, I appreciate the efficiency, consistency, and the fact that there will be fewer stovepipes and communication gaps as a team that works together and communicates well.

Here's how I explain DevOps: You have a process to automate — you'll perform it manually and make sure you have the procedure down, then you need to document it and make it easier to figure out the individual steps. Then the interesting thing here is that you may not know it, but you have already begun the DevOps process!

After this step, you'll go ahead and start to code or script your process. Once you are satisfied, you'll want someone to review your code, and you and/or teammates can now write the tests.

Once ready, the code can be released. It is then time to monitor, make sure everything is working properly, and watch for needed changes, whether it's bugs in the code, changes in the environment, or updates suggested by the customer. Consequently, we retrospect and continue to find ways to improve both the product and the process.

DevOps Automation Checklist

Step 1: Perform the function manually – Learn the ins and outs, make sure it is possible.
Step 2: Document – Ensure the steps are clear. At this stage, someone else could even take over. The cool thing here is that if something happens (e.g., you win the lottery and do not need to go back to work), a teammate would be able to take over where you left off!
Step 3: Code or script your process – The fun part! Make sure you document the code, as well.

Step 4: Create tests – You or a co-worker can create unit, acceptance, and integration/regression tests. This not only ensures the code works correctly, but also that goals are clear. Regression testing assures that changes can be safely integrated and tested.

Step 5: Have someone review code/tests – Again, someone else is familiar with the code, and a "second pair of eyes" should help ensure everything works correctly.

Step 6: Release! – Release of your module is a great milestone, but life happens, so...

Step 7: Monitor – Wait, we're not done yet! Ensure that everything is working smoothly. Watch for needed changes — whether it's bugs in the code, changes in the environment, or updates suggested by the customer.

Step 8: Retrospect – Document and discuss lessons learned for next time. "Rinse and repeat."

Here is the most important takeaway: Every step in this process brings improvement, so even if you do not get beyond the documentation step, you are already ahead of where you were before and headed toward DevOps Nirvana!

The Benefits of Automation

So why do we automate things? First, there is the magic triangle of time/cost/quality — all three of these can be improved by automation. Automation adds efficiency to your process, thus reducing the chance for human error regardless of the cause. Automating also allows for scaling; if you can automate a process once for one purpose, you can reuse the code, use the same code on multiple systems, and/or make quick changes on the fly without having to start the whole process over.

Automation allows for processes to be part of a larger automated process (i.e., CI/CD). The fact that code and documentation exist allows enterprise knowledge to be maintained and the training of new employees to be simplified. Those who are writing code, reviews, and tests are also learning from this process. And last but not least, time is being freed up to do more interesting and meaningful work.

How to Choose the Right Automation Tool

Now let's talk about some of the common tools. We'll start with the most common coding and scripting languages (Ruby, Perl, Java, and others). We've automated difficult processes and calculations for a long time using these languages. Taking it one step further, using text-based tools such as APIs (Application Processing Interface) and CLIs (Command Language Interface) allows you to take complicated GUIs and write them into commands or integrate them into your codebase to reduce some of the complications of navigation.

Continuous integration tools (such as Jenkins, Travis-CI, Circle-CI, and GitLab-CI) allow you to perform the steps of test, integration, and deployment based on a trigger such as a pull request or a timer. Provisioning tools allow you to automate a system's setup to ensure consistency among your servers. There are also tools like IFTTT (if this, then that) that you can use at home or at work to automate simple tasks like getting a text message when rain is predicted or adding a line to a spreadsheet when a PR is created on your GitHub repo.

But remember: **DevOps is tool-agnostic**. Your choice of tools may be different from what others select. It's important to balance your needs with your budget; perhaps create an "analysis of alternatives." Keep in mind not only the budget and project requirements, but the comfort level of your team and management — and look ahead at what needs may expand.

Is Automation Right for My Project?

Part of DevOps is knowing the appropriate development method, and believe it or not, there are times when it is not appropriate to automate processes. Here are some examples: Sometimes you have a task that you only do once, and although it may not be pleasant, it may be quicker to do it than to code it. For example, totaling some numbers for a one-time situation could involve copy-and-pasting them into an Excel spreadsheet rather than writing complicated code to read them from a file and perform the addition.

Sometimes manual intervention is needed — whether it is to add that second factor of authentication (where automation could also be a security issue) or to get an interim status, metric, or value. In many cases, human creativity is needed. This could involve making an interface human-understandable (automated tests can make sure the fields are set up, read in, and used properly, but a human really needs to ensure readability and ease-of-use within the actual interface) or running extra tests that involve some edge cases and human qualities that a machine may miss (accessibility, for example).

The human touch is also recommended in code reviews — yes, please use the code grammar checkers and test coverage tools. Having your code reviewed and reviewing others' code benefits everyone involved. Lastly, when starting a larger automation project, do not try to do everything at once. Prioritizing and easing into the automation process makes it simpler and increases the probability it can be done with no loss of functionality.

Here are some metrics that will indicate to your team, management, and the bean counters various improvements: cost and time savings; test coverage and speedup; customer satisfaction; fewer defects; faster time to release, as well as to recover from issues; and reduced risk.

Conclusion

So what can we do to ensure everyone is happy? Go through DevOps' proper stages — document, script, test, review, release, monitor, and retrospect. Confirm your team and management properly vet any tools used. Make sure changes are done gradually. Lastly, remember there are situations in which it is totally appropriate to use human creativity.

My favorite thing about DevOps is the feeling of community. You will likely reach a point where your product is fully automated, and you can move your creativity in another direction. Still, it is important to ensure that there is interaction, communication, knowledge-sharing, and documentation every step of the way. And feedback — you will definitely have lessons learned to apply to your next creative endeavor. To make an analogy to my second-favorite thing (after DevOps): Like baking, even a perfect recipe needs a little creativity, and often a slight tweak each time you make it. So automate away, but ensure that the human touch remains part of your process.

Judy Johnson, Software Engineer at EITR Technologies

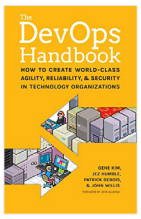
@jfjohnson on DZone | @miz_j on Twitter | @judyj on GitHub

Judy has worked as a software engineer for over thirty-five years. Working as a System Engineer, Project Manager, ScrumMaster, and a CD store clerk, she started programming in the 19XXes when her dad brought home a PDP-8 — she eventually progressed from paper-tape and punched cards to more modern computing systems. When not at work, Judy can be found baking for family, friends, and co-workers; attending hockey games and rock concerts; or trying to finish a good book. Judy loves to volunteer, especially in events that promote diversity in technology. Proof of her dedication to this cause is the fact that both of her awesome daughters are engineers!

ADDITIONAL RESOURCES

Diving Deeper Into CI/CD

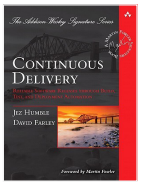
BOOKS



The DevOps Handbook

By Gene Kim, Jez Humble, Patrick DeBois, and John Willis

Following in the footsteps of The Phoenix Project, The DevOps Handbook shows leaders how to replicate these incredible outcomes by showing how to integrate product management, development, QA, and IT operations to elevate your company and win in the marketplace.



Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation

By Jez Humble and David Farley

This book explores the foundations of a fast, reliable, low-risk delivery "deployment pipeline," an automated process for managing all changes, from check-in to release. Through state-of-the-art techniques, including automated infrastructure management and data migration, authors assess key issues, identify best practices, and mitigate risks.

REFCARDS

End-to-End Test Automation Essentials

End-to-end (E2E) tests are often neglected, due in large part to the effort required to implement them. Automation solves many of these problems by ensuring a consistent, production-like test coverage of the system. In this Refcard, learn the fundamentals of E2E test automation through test coverage, integration, and no-code options.

The Essentials of GitOps

GitOps is an operational framework that leverages DevOps core practices used in application development, including version control, CI/CD, and collaboration, for infrastructure automation. This Refcard introduces the fundamentals of a mature GitOps model, the key benefits of GitOps, and the elements of a GitOps workflow for new feature deployment.

TREND REPORTS

Containers

In DZone's 2021 Containers Trend Report, we explore the current state of container adoption, uncover common pain points of adopting containers in a legacy environment, and explore modern solutions for building scalable, secure, stable, and performant containerized applications.

The State of CI/CD: Evaluating Pipeline Maturity

DZone surveyed developers about how DevOps teams are building up and taking advantage of their engineering maturity. Read our original research, articles from DZone's most reputable contributors, and our exclusive interview with Gene Kim, renowned author and DevOps thought leader.

The Rise of Continuous Testing

Continuous testing (CT) isn't just about automation or CI/CD pipelines. CT involves testing throughout the SDLC — automating the *appropriate* tests, implementing the proper policies, and ensuring teams have effective test automation frameworks in place. Read this Trend Report to learn more through original research and expert contributor articles.

PODCASTS

Arrested DevOps

Arrested DevOps is the podcast that helps you achieve understanding, develop good practices, and operate your team and organization for maximum DevOps awesomeness.

DevOps Radio

In a podcast that talks all things software delivery, your hosts explore a wide range of topics from Jenkins and CI/CD to feature flags, microservices, and more.