DZone

DZONE TREND REPORT

MARCH 2022

DevOps

CI/CD and Application Release Orchestration

BROUGHT TO YOU IN PARTNERSHIP WITH

Coralogix

Welcome Letter

By Jacob Doiron, Senior Software Engineer at DZone

Picture this: A hot, new startup is pursuing a SaaS model that's yet to exist, and they are flying by the seat of their pants. Employees are juggling multiple hats as the company struggles to find enough hours in the day to hire new employees, each with dedicated roles. Johnny, the startup's most senior engineer, starts to lay the foundation for a CI/CD pipeline but then quickly halts in his tracks. He can surely figure out the environment setup himself with the assistance of some online documentation, but perhaps he should consider if finding an off-the-shelf solution would make more sense.

Enter managed services.

With a managed service, the startup moves a lot of the complexity out of the business. Instead of needing to think through all the security implications involved in a self-hosted pipeline, employees will simply have access to a service that just works and is accessible over the internet — all out of the box. The managed service will also take care of applying security updates and ensuring systems are fully operational 24/7. But wait, can the startup trust this service provider? What's the managed service's uptime SLA? What control will the startup relinquish by giving the managed service control of their pipeline?

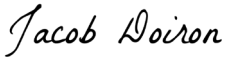
Maybe Johnny would be better off hosting the CI/CD pipeline internally, which does come with some advantages: more granular control, lower cost, and the ability to customize the software based on the startup's needs. It also comes with some disadvantages: much more complex setup and configuration, the need to continuously maintain the system, and required internal knowledge of CI/CD software and security requirements.

What is better for the startup will largely depend on several factors and will likely be different for every company. The truth is there is no universally right solution for all CI/CD implementations, and it can be challenging trying to identify the best course of action for each particular use case. The awesome part, in either case, is what the pipeline ***enables***.

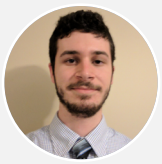
We joined forces with multiple experts in the field to give a clear picture of how CI/CD stands in the industry today. In this report, you will learn about software delivery practices and the impact of continuous delivery on software quality based on our research analyses, as well as discover a comparison of managed vs. self-hosted CI/CD, the role of automated testing in pipelines, threat mitigation strategies, infrastructure provisioning, and much, much more!

Welcome to DZone's 2022 "DevOps: CI/CD and Application Release Orchestration" Trend Report — we hope you enjoy reading this piece as much as we did creating it!

Kind regards,



Jacob Doiron



Jacob Doiron, Senior Software Engineer at DZone
@jdoiron94 on LinkedIn

Jacob was introduced to software development after finding curiosity in third-party game clients and exploring the extent to which gameplay could be automated. When he is not focused on addressing JIRA tickets, he can be found playing games, reading a fantasy novel, or playing with his rescue dog, Shadow.

ORIGINAL RESEARCH

Key Research Findings

An Analysis of Results from DZone's 2022 CI/CD Survey

John Esposito, PhD, Technical Architect at 6st Technologies

From January-February 2022, 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. Software delivery practices and metrics
2. Relation of software delivery and design
3. 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, popups on DZone.com, LinkedIn, and the DZone Core Slack Workspace. The survey opened on January 24, 2022 and closed on February 6, 2022, recording 1,187 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: Software Delivery Practices, Metrics, and Tooling

Motivation:

1. It is a truism that nobody knows how to develop software. It is perhaps less evidently true that nobody knows how to deliver software, but for many software professionals, software delivery is something learned on the job, in battle, through sweat and blood and tears. We suppose that the difficulties of delivery may be a function less of the intrinsic muddiness of physics-free engineering than of the contingent fact that nobody teaches you DevOps in Algorithms and Data Structures 101, Compiler Design 211, or Programming Language Theory 412. So we wanted to help software professionals learn from others' experiences.

2. The (relatively recent) firm establishment of DevOps as a professional subdiscipline risks a new Cantor set/Zeno's paradox specialization problem: Will the intersection of the development and operations silo become its own, third silo? We want to pop this bubble before its wall grows too thick, so we set out to study "build phase" and "release phase" together.

3. "What is 'done'?" is hard enough to answer before production release. "What is 'done' well?" is trickier to ask coherently (let alone answer) still. Yet rapid releases are supposed to facilitate iterative development by providing feedback on software performance after the software is called "done." We wanted to know how software professionals cut through the fuzziness of "done" and "done well" — what metrics they use to decide whether corrective work is needed and under what circumstances — to improve software as delivered.

REASONS FOR ADOPTING CONTINUOUS DELIVERY

Continuous delivery makes many promises, each of which may enjoy a varying degree of salience to different software professionals. We wanted to know what benefits software professionals overall expect from continuous delivery and how these reasons for adopting continuous delivery vary by job description.

So we asked:

What do you believe are the top reasons for adopting continuous delivery? Please rank in order of most important (top) to least important (bottom).

Results (n=445), ranked by overall score and segmented by respondent job description:

Table 1

REASONS FOR ADOPTING CONTINUOUS DELIVERY BY JOB ROLE				
		Respondent Role		
Rank	Overall	Developers and architects	DevOps leads and SREs	Sysadmins
1	Increased speed of feature delivery	Increased speed of feature delivery	Increased speed of feature delivery	Increased speed of feature delivery
2	Shortened development cycles	Shortened development cycles	Shortened development cycles	Shortened development cycles
3	Increased release frequency	Improved developer/ team flow/productivity	Improved developer/ team flow/productivity	Improved developer/ team flow/productivity
4	Improved developer/ team flow/productivity	Increased release frequency	Increased release frequency	Increased release frequency
5	Reduced complexity of development cycle	Reduced complexity of development cycle	Reduced complexity of development cycle	Reduced overhead costs
6	Reduced deployment error rate	Reduced deployment error rate	Reduction in the number of bugs post deployment	Reduced complexity of development cycle
7	Reduction in the number of bugs post deployment	Reduction in the number of bugs post deployment	Reduced deployment error rate	Reduced time to complete QA feedback loops
8	Reduced time to complete QA feedback loops	Reduced time to complete QA feedback loops	Reduced time to complete QA feedback loops	Reduced maintenance costs
9	Reduced overhead costs	Reduced overhead costs	Reduced overhead costs	Reduced mean time to discovery
10	Reduced maintenance costs	Reduced maintenance costs	Reduced mean time to discovery	Reduction in the number of bugs post deployment
11	Reduced mean time to discovery	Reduced mean time to discovery	Reduced maintenance costs	Reduced error budget
12	Reduced error budget	Reduced error budget	Reduced error budget	Reduced deployment error rate

Scores are computed by weighted rank: If, for a given response, answer A is ranked highest out of N answer options, A's score is incremented by N, while if answer B is ranked second highest out of N answer options, B's score is incremented by N-1.

Observations:

1. The four overall top-ranked reasons for adopting continuous delivery did not vary by stakeholder role. We take this to mean that continuous delivery is more or less equally appealing to all software professionals with respect to these four *desiderata*.

2. Variations by stakeholder role appear below these top four reasons in ways that seem to map what we might guess from stakeholder role. Some notable differences are color-coded in the table above:
    - Sysadmins ranked both maintenance and overhead costs higher than developers/architects or DevOps professionals.
    - Developers/architects and DevOps professionals ranked post-release bug counts and error rate reductions higher than sysadmins.
    - DevOps professionals ranked reduced mean time to discovery (MTTD) higher than developers/architects and lower than sysadmins.
3. Overall, we are struck by the high degree of similarity between developers'/architects' and DevOps professionals' reasons for adopting continuous delivery. This suggests that DevOps leads and SREs are more "in developer headspace" than they are "in sysadmin headspace" — as the historical origins of DevOps and the sequence of abbreviations in the portmanteau both suggest.

RANKING OF METRICS FOR CONTINUOUS DELIVERY

Reasons to adopt CD are adoption-antecedent; metrics for CD are adoption-sequent. Just as we wanted to know why software professionals adopt continuous delivery, we wanted to know how software professionals measure continuous delivery, again segmented by professional specialization. So we asked:

What do you believe are the most important metrics for continuous delivery? Please rank in order of most important (top) to least important (bottom).

Results (n=459), ranked by overall score and segmented by respondent job description:

Table 2

METRICS FOR CONTINUOUS DELIVERY BY JOB ROLE

		Respondent Role		
Rank	**Overall**	**Developers and architects**	**DevOps leads and SREs**	**Sysadmins**
1	Deployment frequency	Deployment frequency	Deployment frequency	Deployment frequency
2	Production downtime during deployment	Production downtime during deployment	Production downtime during deployment	Change failure rate
3	Lead time	Lead time	Lead time	Absolute number of bugs
4	Change failure rate	Change failure rate	Change failure rate	Production downtime during deployment
5	Mean time to recovery	Mean time to recovery	Mean time to recovery	Lead time
6	Regression test duration	Regression test duration	Regression test duration	Mean time to discovery
7	Mean time to discovery	Absolute number of bugs	Error rate	Mean time to recovery
8	Absolute number of bugs	Mean time to discovery	Mean time to discovery	Regression test duration
9	Error rate	Error rate	Absolute number of bugs	Error rate
10	Error budget	Error budget	Error budget	Error budget

Observations:

1. The top-ranked CD metric, deployment frequency, maps well to the top-ranked reason for adopting continuous delivery, increased speed of feature delivery.

2. Developers/architects and DevOps professionals ranked all metrics in the same order except for error rate and absolute number of bugs. The swapping of these latter metrics maps well to the areas each role is responsible for: Developers/architects fail in proportion to released bugs, while DevOps professionals fail in proportion to deployment errors. Note: In retrospect, we realize that "error rate" may not unambiguously indicate "deployment error rate." In future surveys, we will phrase this option more explicitly.

3. Sysadmins' ranking of CD metrics does not map to job responsibilities in the same way. If it did, we would expect sysadmins to rank production downtime during deployment higher as a CD metric than the other two roles because sysadmins are responsible for production uptime. We would also expect absolute number of bugs to be ranked much lower because sysadmins do not create bugs.

    In fact, among our respondents, the opposite is true. Because our sysadmins response count is very low (n=7), we cannot form any conclusions with high confidence. But based on this inversion of expectations, with suitable small-n caveats, we conjecture that sysadmins think of CD more defensively than developers/architects and DevOps professionals. That is, sysadmins ranked the CD metrics that measure things *others* are responsible for higher, while developers/architects ranked the CD metrics that *they themselves* are responsible for higher.

    We imagine this reversal may be accounted for by the conjunction of greater immediate pressure placed on sysadmins after release (an asymmetry) and agreement between sysadmins and other software professionals that CD is more about development than about ops (a symmetry). This provides empirical support for the suggestion — which we might separately hypothesize from the history of CD — that sysadmins feel less ownership over CD than developers/architects or DevOps professionals.

PREREQUISITES FOR AUTOMATED PRODUCTION DEPLOYMENTS

Continuous delivery requires automation; safe continuous delivery requires the right kind of gating either during the release pipeline or before code is committed to the build source. We wanted to know what software professionals are treating as sufficient prerequisites for automated production deployments, so we asked:

Check all things that must be true (i.e., join by Boolean AND) in order to automate production deployments.

Results (n=521):

Table 3

PREREQUISITES FOR PRODUCTION DEPLOYMENTS

Prerequisite	%	n=
Unit test coverage is 100%	25.2%	132
Unit test coverage is >75%	57.0%	298
Unit test coverage is >50%	14.7%	77
Code review process is robust	75.3%	394
All whitelisted user paths are covered by automated UI tests	41.5%	217
Most (a fuzzy definition of "most") user paths are covered by automated UI tests	37.3%	195
Realistic load test times are performed on every build	45.1%	236
Production and pre-production environments are provisioned by the same code	72.1%	377
Crucial features are called behind feature flags and can be turned off without a new deployment	40.3%	211
Rollback deployments have been used successfully in production many times before	49.3%	258
There is an auditable approval process to promote changes from pre-production to production	47.8%	250

Observations:

1. Robust code review is the most common automated production deployment prerequisite (75.3% of respondents). Strictly speaking, of course, code review is not part of the release process — where release begins after source has been committed) — but its high performance in this survey indicates that, in the judgment of software professionals, the importance of robust code review for CD should not be overlooked.

2. Infrastructure as Code (IaC), phrased as "production and pre-production environments are provisioned by the same code," is the second most common automated production deployment prerequisite (72.1% of respondents) and easily the highest-scoring release-specific prerequisite. We are surprised not by how high this number is, but by how low: Over a quarter of respondents do not consider IaC a prerequisite for production deployments.

3. The most preferred unit test coverage is 75% (57% of respondents), followed by 100% (25.2%). These numbers are slightly up from the same audience last year, only 21.7% of whom require 100% unit test coverage. As in last year's survey, however, 100% unit test coverage does not correlate with decreased incidents and rollbacks:

    - 11.5% of respondents who require 100% unit test coverage reported incidents or rollbacks almost every deployment vs. 4% of respondents who require 75% unit test coverage.
    - 45.8% of respondents who require 100% unit test coverage reported incidents or rollbacks occasionally vs. 39.7% of respondents who require 75% unit test coverage.

Again, we conjecture that this difference may be due to overconfidence in unit test coverage and/or excessive focus on a "perfect" numerical target (100%) when mapping of asserts to business cases, quality of the data mocks, and design of higher-integration tests are more important for test quality than pure unit-level coverage.

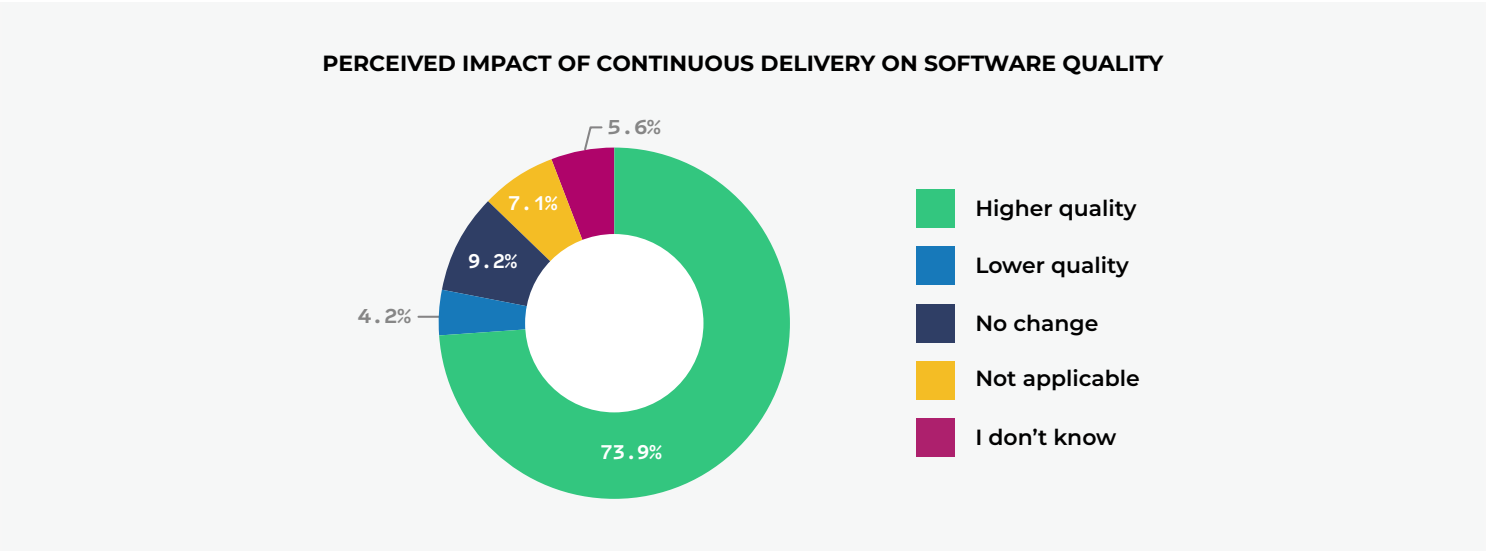
IMPACT OF CONTINUOUS DELIVERY ON PERCEIVED SOFTWARE QUALITY

In addition to specific metrics used to evaluate CD itself, we wanted to know what software professionals think about the impact of CD on software quality. So we asked:

Overall, adopting continuous delivery has made my applications: {Higher quality, Lower quality, No change, Not applicable, I don't know}

Results (n=522):

Figure 1



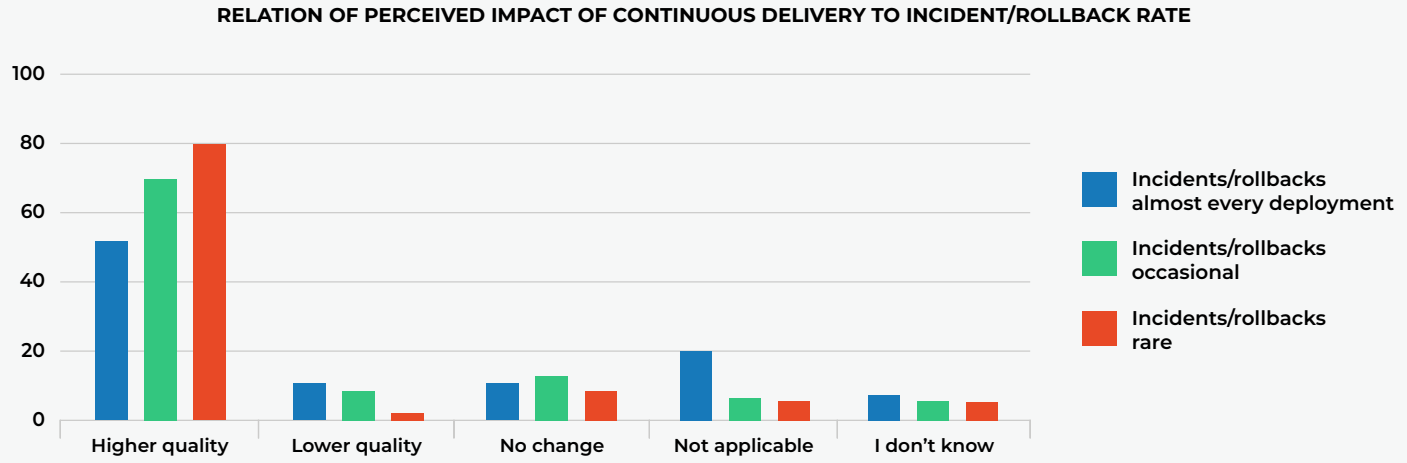
Observations:

1. A large majority of respondents (73.9%) judged that CD makes their applications higher quality. This is good news for CD: It is not only making releases more pleasant, but actually resulting in better software.

2. DevOps professionals are slightly more likely to judge that CD makes their applications higher quality (83.7%) vs. developers/architects (75.7%), a difference mostly accounted for by greater neutrality on the part of developers/architects — 9% reporting no change vs. 4.1% of DevOps professionals reporting no change.

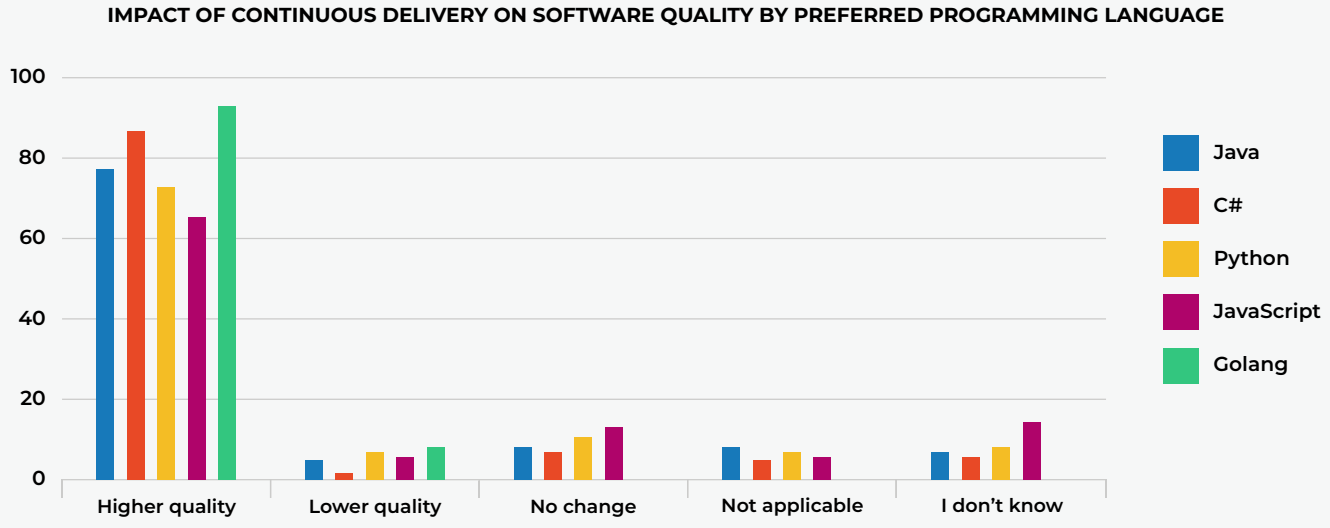
3. Overall subjective evaluations of CD's impact on software quality, as measured by this question, are consistent with reported incident/rollback rates:

Figure 2



4. Respondents whose favorite programming language is JavaScript were least likely to report that CD results in higher quality software overall, while respondents whose favorite programming language is Golang were most likely to report that CD results in higher quality software overall:

Figure 3



We conjecture (albeit without much confidence) that this may be because JavaScript-preferring developers' code is more likely to interact with human users. Hence, it requires more automated UI/UX testing for a satisfactory CD pipeline, and UI/UX test automation may be less reliable than non-UI/UX automated testing — if only because defining desired UX outcomes is more complex than defining, say, desired price calculation outcomes.

Research Target Two: Relation of Software Delivery and Design

Motivations:

1. Engineering involves trade-offs; therefore, engineering involves regrets. That is, when one *desideratum* is neglected for the sake of another (i.e., the trade-off is made), any situation that benefits from the neglected *desideratum* is a trade-off victim. And no engineer worth their craftsperson pride fully ignores these missed opportunities simply because the trade-off produced better results in aggregate.

   Trade-offs are design decisions made with an eye toward imagined aggregate results, so the excellence of the trade-off as a design decision is known only hypothetically before release. We wanted to understand how some basic software design decisions — and the trade-offs they entail — relate to what happens when the imaginary gives way to the actual.

2. Rollbacks come from unforeseen side effects. Much of software design is intended to avoid side effects, and a sign of good code is its transparency to the effects of change. (Some simplifying license might even suggest that the essence of formal languages is the identification of intent and expression.) But the chaotic fires of production may subject code to far more entropic conditions — unintended side effect incubators — than even the best-thought-through test suites. We wanted to see how software design principles aimed at avoiding side effects actually perform in the post-release wild.

3. At age barely-zero, we learn that gratification is, sadly, sometimes delayed. At age four, we're told that delaying gratification is a moral excellence that we would surely embrace if only our souls were not so defective. In school, for a decade or two, we power through boring lectures and more boring homework supposing (or nodding to our teachers who suppose) that our present effort is building up untold treasures of Great Skill in The Magical Future.

   As adults, so conditioned, we somehow attempt to affirm simultaneously (a) The Global Utilitarian Ethic of the Timeless Rational™ from our everything-inchoate childhood, (b) the Move Fast and Break Things dictum of The Silicon Valley Disruptor©, and eventually — if we're truly lucky — (c) the grizzled Nobody Knows Anything, So Just Build It wisdom of Agile®. Okay but...should we write unit tests before writing our substantive code? Is it okay to release without a creative human trying to break it? Is my meat brain clever enough to know what this monster I've fashioned will actually do with a million screaming `GETs` per second? It turns out none of these now-or-later work distribution principles really tell us.

   So we learn whether to TDD or Waterfall, which pre-release manual tests to skip, and which methods to re-parameterize or to refactor into multiple-dispatch style from the raw, atheoretical experience of the build/release-delay/observe-terror/relax cycle. We wanted to understand how software professionals learn how to distribute their work along the release cycle based on real-world development and delivery experience.

RELATION OF PREFERRED PROGRAMMING PARADIGM TO FREQUENCY OF INCIDENTS/ROLLBACKS

Higher-level programming paradigms (e.g., functional and object-oriented programming) promise better source legibility, which presumably leads to less slippage between intent and effect, and further, to fewer incidents/rollbacks. We wanted to see if this turned out to be the case, so we correlated frequency of incident/rollback data with preferred programming paradigm — object-oriented, functional, imperative, procedural, or actor-model. Because developers may use multiple paradigms over the course of their career, we made the simplifying assumption that, for the most part, developers are more likely to work with their preferred programming paradigm than not. And while this is certainly false in many cases, we suppose it is true of developers in aggregate, on the argument that demand for programming jobs is high enough to push developers at least somewhat toward being able to use their preferred paradigm.

We did not observe any significant difference in incident/rollback frequency by top programming paradigm preference.

RELATION OF PREFERENCE FOR DEPENDENCY INJECTION TO FREQUENCY OF INCIDENTS/ROLLBACKS

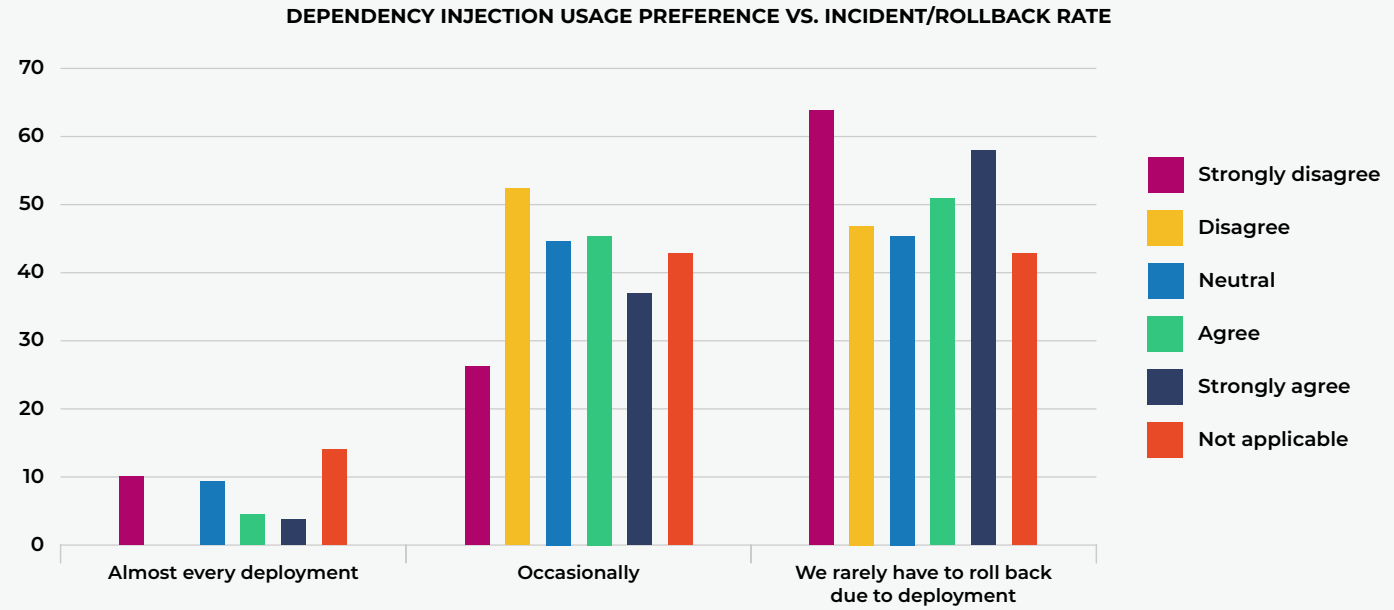
In theory, dependency injection increases modularity by decreasing coupling and simplifies testing by making mocking cleaner. But dependency injection can also increase complexity, which increases overall brittleness of the codebase. So the overall effect of dependency injection on software quality and performance in production is, as usual, a complex sum of costs and benefits. We wanted to understand the impact of dependency injection at a very high level, abstracting from the complexity of the sum.

So we asked:

Agree/disagree: Dependency injection should be used whenever possible.

Results vs. incident/rollback frequency:

Figure 4



Observation:
Respondents who strongly disagreed that dependency injection should be used whenever possible are most likely to report rare incidents/rollbacks after deployment (63.3%), but respondents who strongly agreed are second most likely (58.4%). We take this to suggest both that dependency injection may not result in smoother deployments and — because the smoothest releasers were also the most opinionated — that the presence of someone with strong opinions about dependency injection on a software team may encourage less frequent incidents/rollbacks after release. Presumably, this is because people who think carefully about coupling and complexity are more likely to have strong opinions about dependency injection than those who are neutral.

PERCEIVED IMPACT OF MICROSERVICE ARCHITECTURE ON FEATURE VELOCITY

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 feature velocity increase promised by microservice architecture holds true in practice, so we asked:

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=522), shown in Table 4:

Observations:

1. The feature-velocity promise of microservices is somewhat supported by survey respondents. A significant majority (64%) judged that microservices resulted in higher feature velocity, and only a small minority (12.1%) judged that microservices resulted in lower feature velocity.

Table 4

MICROSERVICES IMPACT ON FEATURE VELOCITY

	%	n=
Higher feature velocity	64.0%	334
Lower feature velocity	12.1%	63
No change in feature velocity	9.2%	48
I don't know	14.8%	77

2. These numbers are slightly less optimistic toward microservices than last year, when 71.0% of respondents reported that microservice adoption resulted in higher feature velocity. However, most of the difference is accounted for by an increase in agnostic ("I don't know") answers — 14.8% this year vs. 5.6% last year.

   We take this to mean that the impact of microservices on feature velocity is becoming less clear over time, which may suggest some kind of optimal shelf-life for a microservice architecture — an intriguing possibility we intend to examine in future surveys on software architecture.

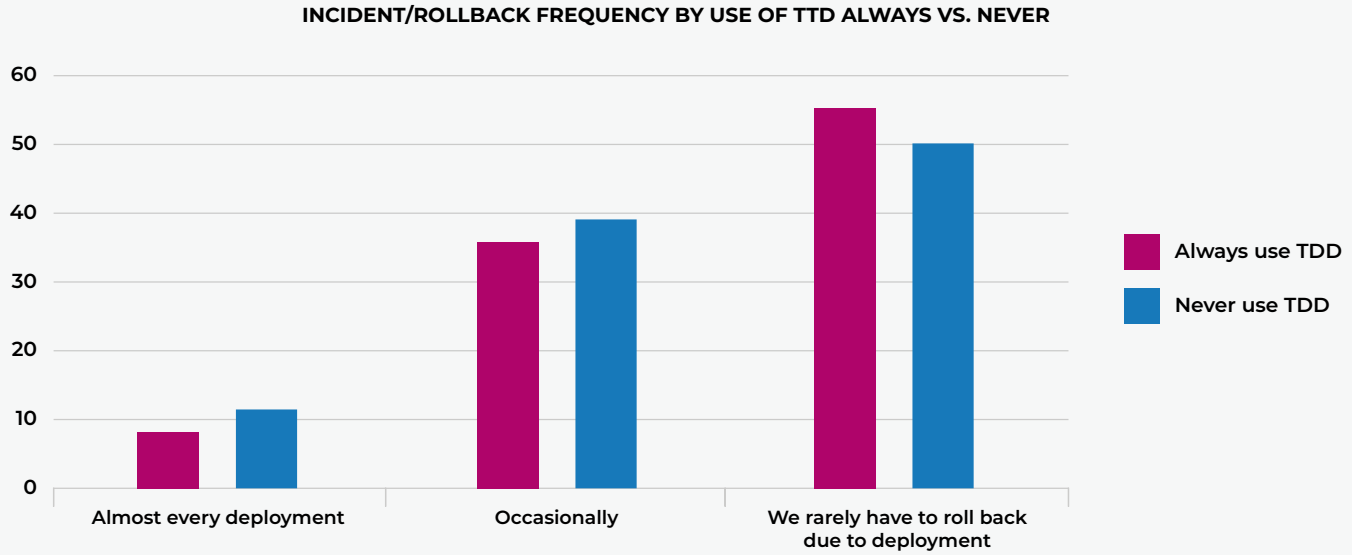
RELATION OF TEST-DRIVEN DEVELOPMENT USAGE TO INCIDENT/ROLLBACK FREQUENCY

Plausible: Yes, test-driven development (TDD) enables CD because TDD results in better tests, which means more reliable code. Also plausible: No, TDD does not enable CD because TDD results in many test rewrites, which slows velocity. We wanted to see which of these stories describes what actually happens, 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 vs. incident/rollback frequency:

Figure 5



Observations:

1. Respondents who always use TDD are more likely to report rare incidents/rollbacks (55.6%) than respondents who never use TDD (50%), and "almost every" and "occasional" numbers are consistent with this difference. This suggests that TDD may indeed facilitate CD and make releases smoother.

2. Respondents who 'always' use TDD are also more likely than respondents who 'never' use TDD to judge their current rate of deployment 'just right', more likely to judge it 'too fast', and far less likely to judge it 'too slow' (see Table 5).

   We take these results — especially the far higher rate of "too slow" responses among "never TDD" respondents, 48.5% vs. 20.8% of "always TDD" respondents — as tentative refutation of the "TDD slows you down" objection.

Table 5

DEPLOYMENT RATE BY USE OF TDD

Deployment rate	TDD always	TDD never
Too slow	20.8%	48.5%
Too fast	19.4%	7.6%
Just right	59.7%	43.9%

3. TDD use also strongly correlates with perceived optimal technical debt:

Table 6

TECHNICAL DEBT BY USE OF TDD		
	TDD always	TDD never
Too much technical debt	33.3%	51.5%
Too little technical debt	6.9%	10.6%
The optimal amount of technical debt	51.4%	25.8%
I have no opinion	8.3%	12.1%

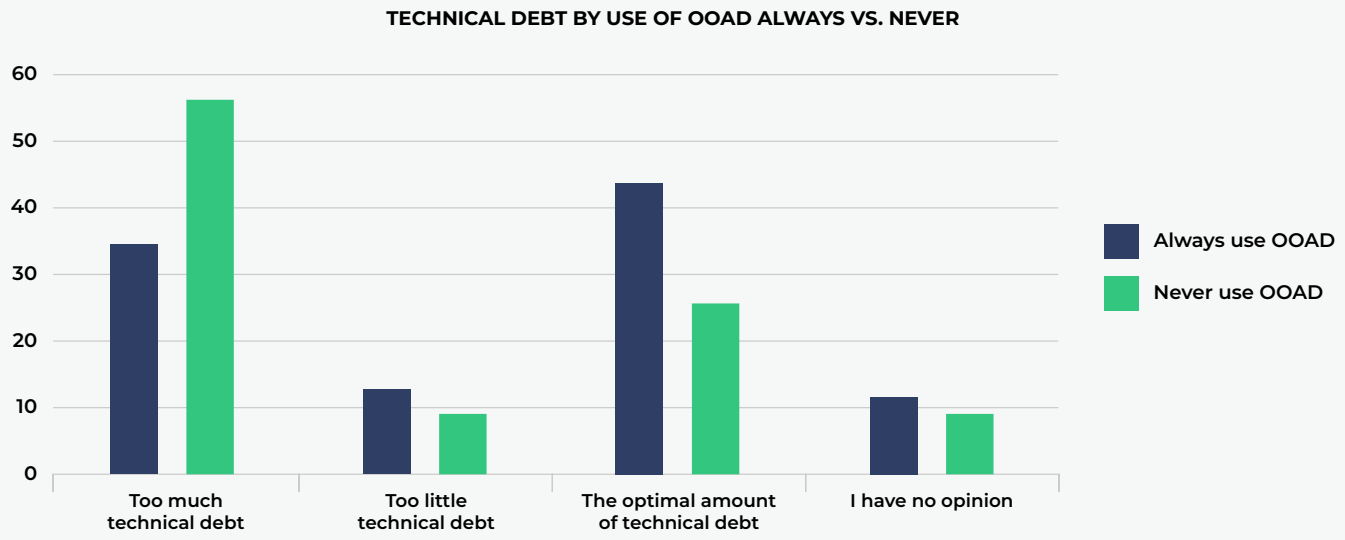
RELATION OF OBJECT-ORIENTED ANALYSIS AND DESIGN TO PERCEIVED TECHNICAL DEBT

The old message-board religious war: Do the well-membraned cells really take care of themselves? Or is the idea of state so unmathematical that object-oriented systems inevitably become too hard to reason about? Recognizing that technical debt is difficult to measure in the long term, we wanted to know whether the use of object-oriented analysis and design (OOAD) relates to technical debt as judged by software professionals. 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:

Figure 6



Observations:

Respondents who never use OOAD are much more likely to report that their organization carries too much technical debt (56.4%) vs. respondents who always use OOAD (33.8%) and are much less likely to report optimal technical debt (25.5% vs. 43.1%). Although technical debt is, of course, affected by many things besides programming paradigm, we take these large differences to suggest that explicit attention to object-oriented design may prevent technical debt from growing too large.

Research Target Three: Relation of Software Delivery Techniques and Effects

Motivations

1. They say that continuous delivery results in better software. But does it? Well, we have no magic universal metric of software quality, so we asked software developers what they think.

2. Pre-merge code reviews lie somewhere between writing code and releasing it. Decisions made by the reviewer may affect the next pending release, but prudent reviewers, if given enough review time, consider possible effects of the reviewed code long after the next pending release.

   This means that code review involves its own trade-offs between the same short-term (get this code in!) and long-term (preserve the integrity of the application!) considerations that the original developer must also keep in mind — a "left-side" pipeline activity — while also gating the application at the source level as a manual QA tester gates the application at the functional level — a "right-side" pipeline activity. We wanted to understand how developers' approach to these 'farther-right' pipeline activities relates to success or failure on release.

3. Of course, you want the machine you built (offspring you begot) to do the work it was intended to do, to run smoothly under various unforeseen conditions, and to grow easily in the future. But as monthly deployment time approaches, the chunk of your mind occupied by such hopes and dreams shrinks before the ballooning fear of a disastrous, weekend-ruining release. Or so the continuous delivery evangelists tell us. They promise that a sufficiently mature pipeline will drive these nightmares away. But will they? We wanted to know whether the anxiety-lysing promise of CD is borne out in software professionals' real-world experience.

DEPLOYMENT FREQUENCY AND FREQUENCY OF INCIDENTS/ROLLBACKS

Move too fast, too many things break. Move too slow, too many things...still break — and don't exist. To know optimal deployment frequency requires feedback on released software quality. Because measurement of software quality is complex and purpose-dependent (e.g., a startup's fast-release minimum viable product may not need to be as modular as an enterprise back end), we wanted some sense of how to tune velocity based on two high-level measures:

1. Respondents' subjective opinions, measured as the difference between actual and desired deployment frequency
2. Reported incidents and rollbacks, correlated with actual deployment frequency

So we asked three questions:

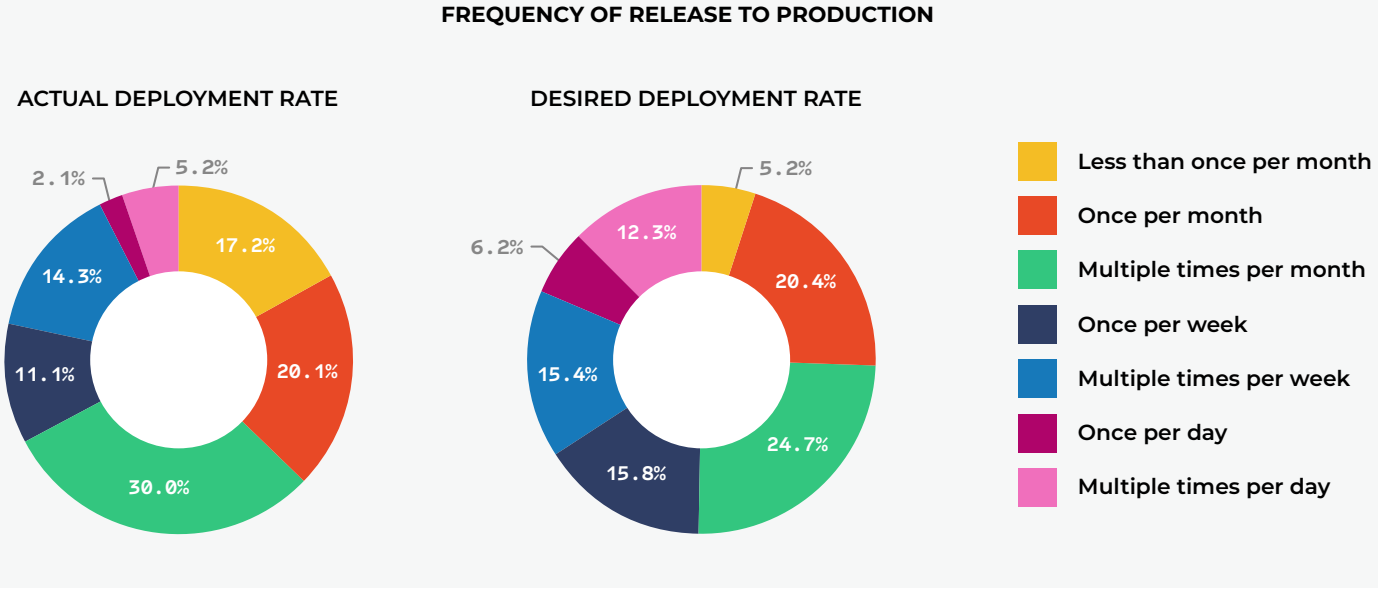
How often do you release to production?

How often would you like to release to production?

How often do your deploys result in incidents or rollbacks?

Results (n=523 and 519, respectively):

Figure 7



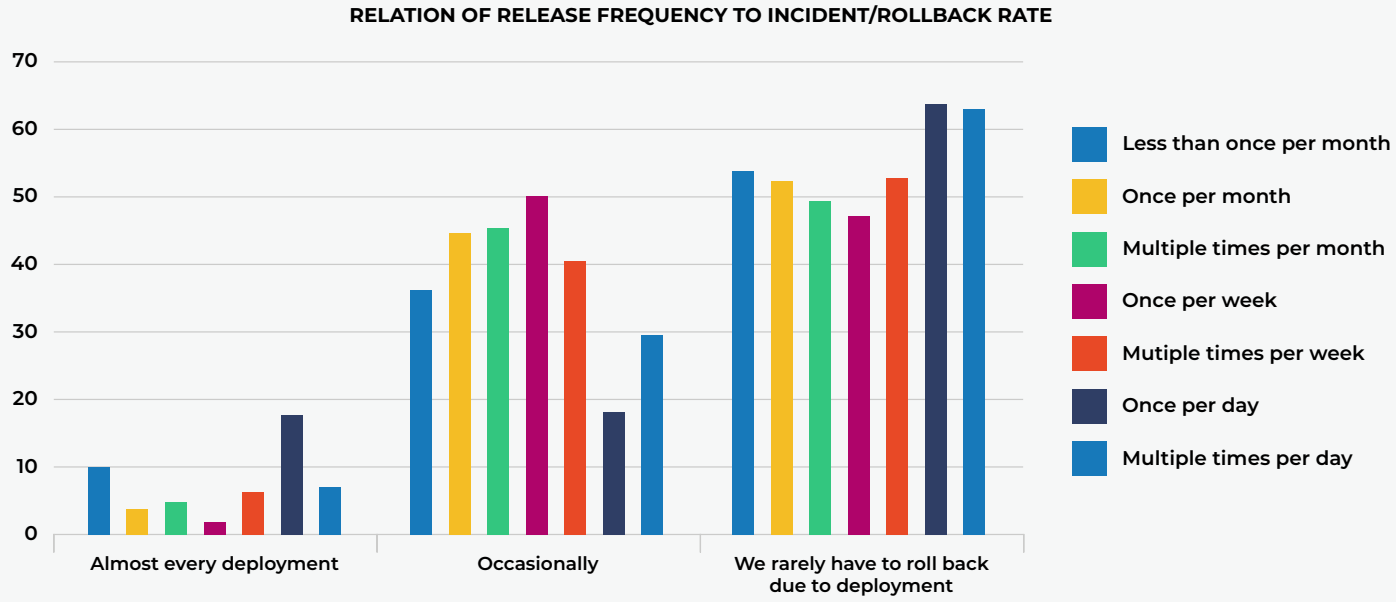
Observations:

1. As we observed last year (and to no one's surprise), respondents would like to release more frequently than they do overall. The biggest absolute difference between actual and desired release frequency obtains with respect to a once-per-week cadence (11.1% actual vs. 15.8% desired), a difference that seems to draw responses mostly from the once-per-month cadence (30% actual vs. 24.7% desired).

2. DevOps professionals are more than twice as likely as developers/architects to want to release multiple times per day (22% vs. 10.5%). Presumably, this is partly because, as DevOps professionals, they are especially committed to CD. We might imagine that if their experience suggested that releasing multiple times per day were causing trouble, then they would not prefer it, so we take this high number as evidence that multiple-times-per-day releases are generally desirable.

The relation of actual release frequency to frequency of incidents/rollbacks observed is complex:

Figure 8



Observations:

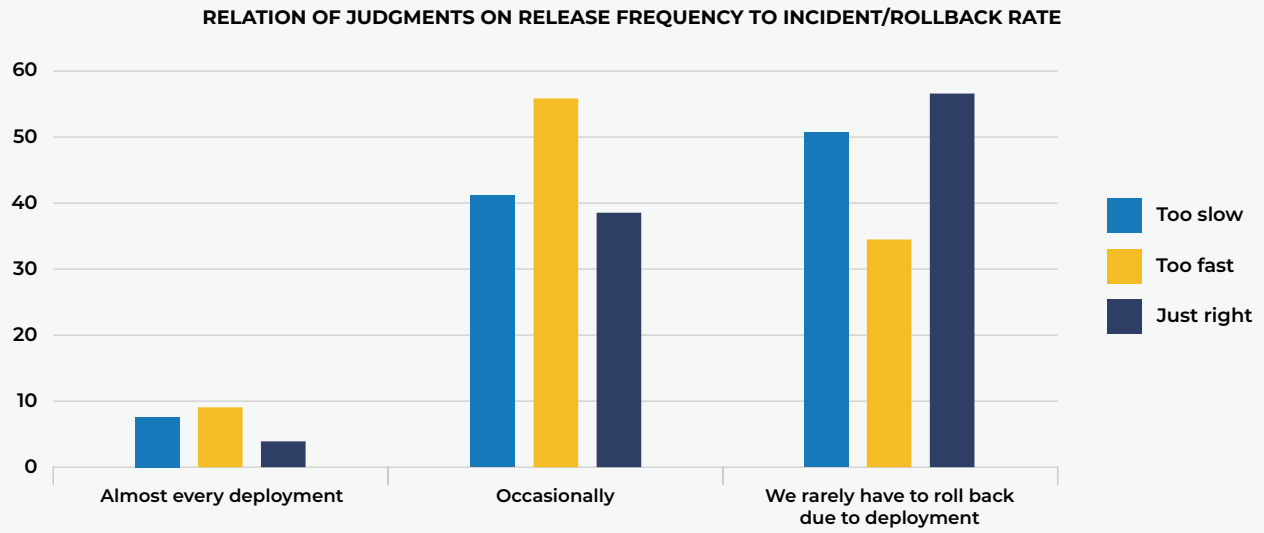
1. Those who release once per day or multiple times per day are by far the most likely to report that they rarely have to roll back due to deployment issues — 63.6% and 63%, respectively vs. 53.4% of the next nearest release frequency reporting rare rollbacks (less than once per month). So the maximally frequent releases correlate with the most reliable rollouts. The minimally frequent releases are a (distant) second, and the differences between sub-once-per-day rare rollback rates are smaller than the difference between the two most frequent and the rest.

2. Overall rates of incident/rollback seem encouragingly low with only 6% reporting incident/rollback almost every deployment, a slight improvement over our last survey on this topic (7.4% in 2021).

3. A word of caution, however: When we cross incident/rollback rate with subjectively evaluated current rate of deployment (an answer to the question, *Our current rate of deployment is {Too slow, Too fast, Just right}*), it appears that *too fast* is significantly less likely than *too slow* to correlate with rare rollbacks:

See Figure 9 on next page

Figure 9



In the subjective judgment of survey respondents, at least, *too slow* releases are safer than *too fast* releases.

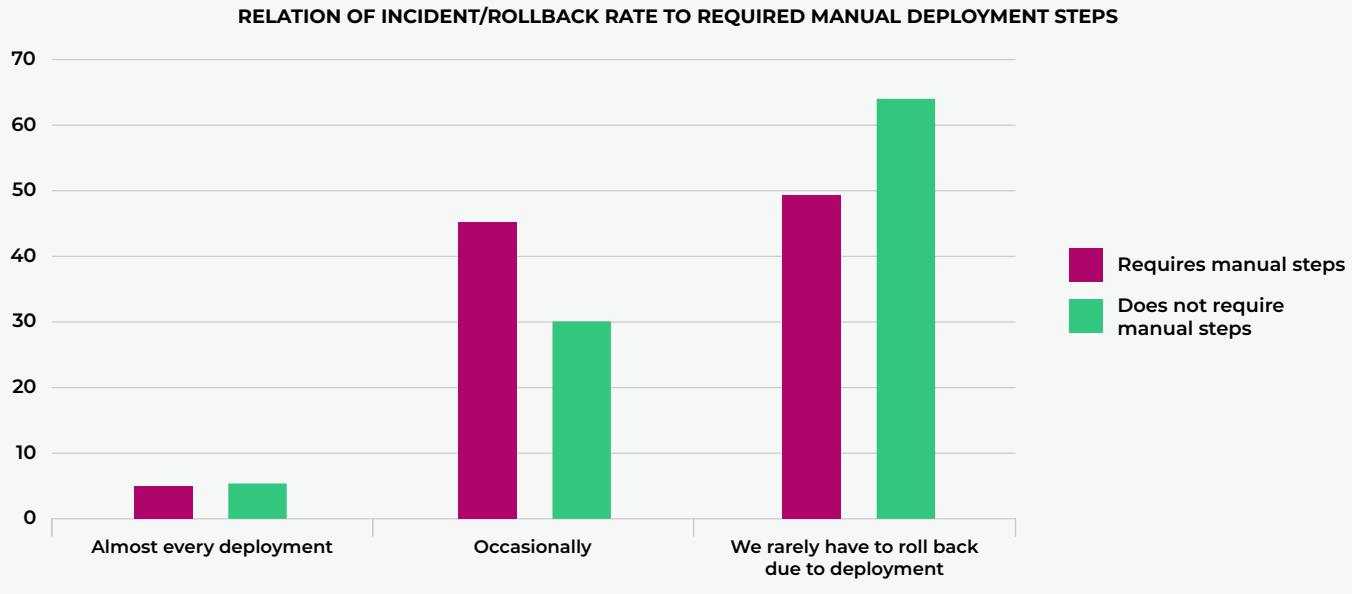
RELATION OF MANUAL INTERVENTIONS TO FREQUENCY OF INCIDENTS/ROLLBACKS

Software errs more repeatably than humans. Automated tests may be good or bad, but they can easily improve monotonically over time. Because human evaluations are both slower and less reliable, the usual CD story excludes manual intervention from release as much as possible. We wanted to see if manual intervention does in fact result in rougher releases, so we asked:

Do your deployments to production require any manual steps?

Results vs. incidents/rollbacks:

Figure 10



Observation:

Respondents whose production deploys do not require any manual steps are significantly more likely to report rare incidents/rollbacks (63.8%) vs. respondents whose production deploys do require manual steps (49%). We take this as empirical endorsement of the "avoid manual steps where possible" dictum — not only in order to achieve CD in some abstract sense, but also, concretely, to avoid having to rollback after a production release.

RELATION OF PULL REQUEST REJECTION ON FREQUENCY OF INCIDENTS/ROLLBACKS

Code reviews are a technical and social dance. Cue internal dialogue:

> "How far out should I cast my imagination when considering the full implications of this code for, what I take to be, the future of this application? Too short a time-horizon permits ungraspable spaghetti (counter-acronym: DRY); too long a time-horizon encourages useless, weighty abstraction (counter-acronym: YAGNI).
>
> Or again: Should I request modifications because this easy-reading code makes many more I/O calls than necessary (for the sake of clean design), or let it through because readability is more important than performance in this case — we can always upgrade the hard disk anyway? And how much are all these teaching opportunities? And how much do I care about the ego of the committer? (Anyone in a senior technical position knows that it is foolish to mock or ignore this latter consideration: Skilled people learn when egos are bruised, but not bruised too much, and only when removing the cause of the bruise is within reach.)"

Pull request (PR) reviews are the point where code review (build-side) most nearly enters the deployment pipeline (release-side), so we wanted to know, at a very high level, how treatment of pull requests relates to post-release incidents/rollbacks. This is an intractable question, though: Pull requests can be rejected for many reasons, the rejection may or may not result in better code, rejection for long-term considerations is not expected to affect imminent releases as much as future releases or future development time/pain, some rejections may be wise and others foolish, the psychological/learning effect is totally orthogonal to all these questions, etc.

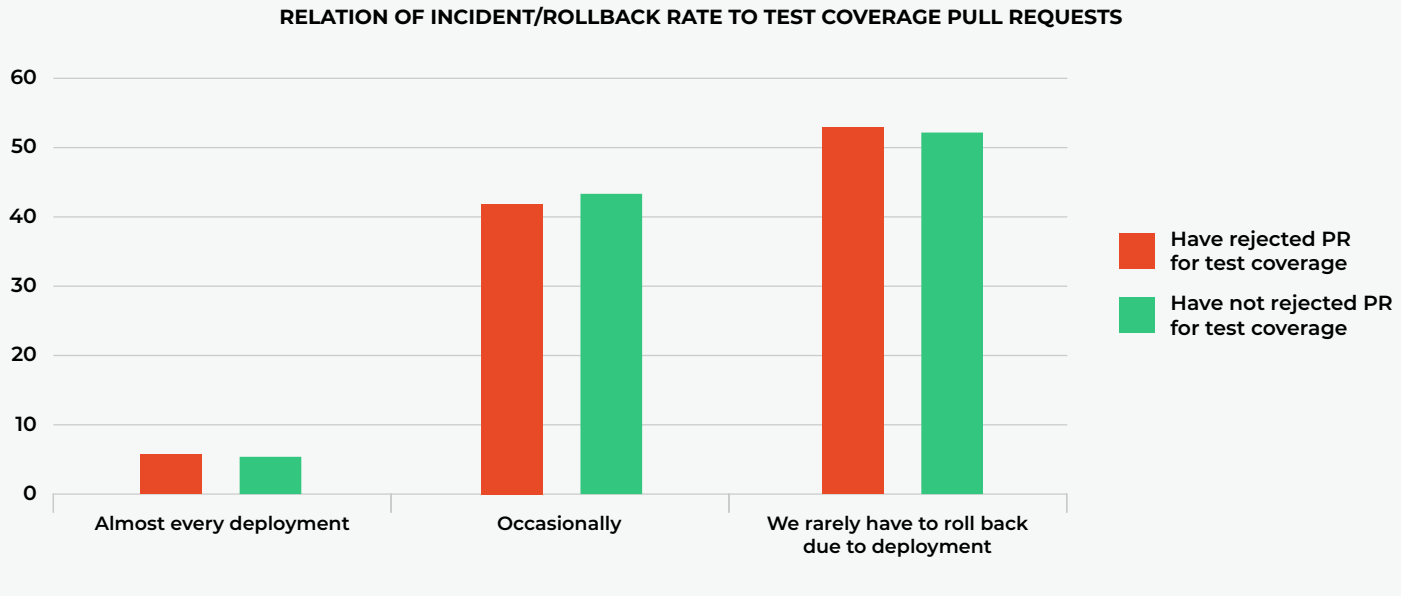
As a very rough first approximation, therefore, we asked a more focused question:

Have you ever personally rejected a pull request because the code in the pull request did not include adequate automated tests (e.g., because coverage by LoC was too low, because test code did not test enough meaningful scenarios)?

And we correlated that with incident/rollback frequency. The reasoning is that if test coverage is important to enforce during code review for the sake of smooth release, then at a very hazy level of approximation, people who have rejected PRs for insufficient test coverage are less likely to experience incidents/rollbacks.

Results:

Figure 11



Observation: No significant relation is evident between rejecting a PR for insufficient test coverage and incident/rollback rate. Because of the very high level of these questions, we do not take this as evidence for or against any relation between PR rejection for insufficient test coverage and release smoothness.

Future Research

In future research, we aim to focus more granularly on relations of software design, release pipeline design, and downstream effects both within a specific feature release and in relation to more downstream metrics than incident/rollback frequency or snapshotted technical debt (e.g., relation of release frequency to actual refactoring time or development slowdown).

Our survey included material not covered in this publication, including:

- Specific types of tests run before deployment
- Degree of environment automation across the release pipeline
- Effect of environment drift on release frequency and deployment anxiety
- Relation of manual intervention count to incident/rollback frequency
- Organizational barriers to adopting continuous delivery
- Source branching strategies
- Organizational focus on development vs. operations

Please contact publications@dzone.com if you would like to discuss any of our findings or supplementary data.

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.

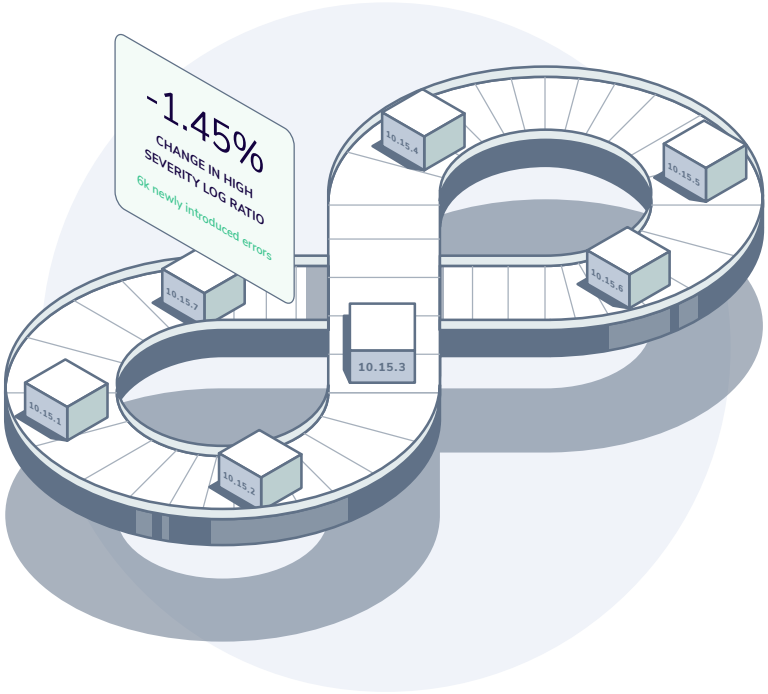
Coralogix

Ensure End-to-End Observability for your CI/CD Pipeline

Analyze and monitor all of your system data in a single, centralized platform. Connect any data, in any format, from any source to visualize and query together using any syntax for extensive event correlation and streamlined troubleshooting.

Coralogix enables full system visibility and can correlate real-time insights with specific releases without the restrictive costs of other solutions.

- → **Reduce MTTR**
- → **Improve Build Quality**
- → **Optimize Costs**



Scale Effortlessly and Continuously Optimize Costs

Coralogix scales effortlessly alongside your system growth and can easily ingest data from new sources. Using breakthrough data prioritization, Coralogix is able to reduce your total cost of ownership and ensure that data costs don't increase exponentially as your systems grow.

Investigate & Correlate Issues Across Releases

View all of your raw logs from your archive and our hot index, plus all ingested and generated metrics data, in the same place. Leverage advanced clustering and data aggregation features to improve feedback loops and streamline troubleshooting.

Improve Build Quality with Automated Version Benchmarks

Generate version benchmarks when new code is released, and integrate them directly to any deployment pipeline to enable teams to reduce issue resolution time, decrease maintenance costs, and improve customer satisfaction.

"On the first day, and without any customization whatsoever, we already received new insights that we were not able to see before. A week in, and our Ops teams across the board were already able to get so much more out of our logs."

Dekel Shavit | VP of Operations & CISO at BioCatch

CASE STUDY

Case Study: Armis

How Armis Improved Their CI/CD Process and Build Quality

CHALLENGE

Armis is working with a massive data pipeline, processing an average of 10B events and ingesting about 3.7TB of log data every day. The R&D team needed a centralized observability platform to help monitor their data, improve their build pipelines, and manage their hybrid cloud infrastructures at scale.

When it came to the CI/CD process, build times averaged around two hours, and the teams didn't have good visibility into which tests were failing or causing delays in the build.

As new features and services were added, more event data was created. The only way for teams to control the amount of data ingested was to block it at the client-side, which required deployment cycles and caused coverage gaps.

SOLUTION

With data from their release pipeline being sent to Coralogix, Armis accelerated time to market while improving stability and quality. Overall, the integration process took only a few hours, with the Coralogix support team available 24x7 to assist.

Now, the Armis team has a dashboard with complete insights into the slowest tests, produced builds, P95 and P99 response times, failed tests, successful tests, time spent on each build, and more. This way, they can easily pinpoint problems in their build systems without any downtime.

Using Coralogix, Armis can zoom into every detail to further analyze the current status of their applications. As a result, developer productivity has increased due to efficiency, centralization, and ease of use.

RESULTS

Immediately upon integration with the Coralogix platform, Armis was able to identify and resolve numerous issues and improve its build system overall. As a result, it has improved efficiency to save both time and money, specifically:

- Median build time was reduced from two hours to five minutes
- Cached artifacts increased from five percent to 90 percent
- The team can run 20 steps in parallel, compared to the original three

COMPANY

Armis

COMPANY SIZE

450+ employees

INDUSTRY

Device Security

PRODUCTS USED

Coralogix, AWS, Jenkins+Bazel

PRIMARY OUTCOME

Armis implemented centralized observability to improve their CI/CD process and build quality.

"We ship all of our logs to Coralogix — test logs, pipeline logs, container building logs, Bazel logs. Those logs are processed and turned into metrics. We then monitor them in a dashboard which allows us to pinpoint exact problems."

— **Roi Amitay**,
Head of DevOps, Armis

CREATED IN PARTNERSHIP WITH

CONTRIBUTOR INSIGHTS

How to Enable CI/CD to Boost the Potential of DevOps

Achieve the Need for Speed in Software Delivery

By Pavan Belagatti, Developer Advocate at Harness

DevOps is a hot topic that is quickly becoming the way of software development. It aims to promote development speed and reduce costs while increasing productivity and efficiency in your organization. DevOps is powered by automating your entire development, delivery, and operations processes. With continuous integration (CI) and continuous delivery (CD), you can do more with less, so it is beneficial to start implementing these concepts into your company as early as possible.

What Is DevOps?

DevOps is a cultural phenomenon used by companies that like to release quality software fast. It is done by automating the entire development and delivery process with the help of techniques and tools. However, it is not just a set of tools but a broader movement that focuses on how to improve the flow of software by streamlining the processes and mindset.

The idea behind DevOps is to have an end-to-end automated pipeline from Dev (development) to Ops (operations). This way, the software moves quickly and can be tested as it moves through different phases of delivery and deployment. Continuous integration and continuous delivery become an essential part of the DevOps initiative and carry a lot of importance.

DEVOPS VS. TRADITIONAL SOFTWARE DEVELOPMENT

Over many years with traditional software development methodologies, development and operations teams have always been treated as separate entities. They each focused on their own efforts, which often resulted in a lack of communication between the two groups. The DevOps movement solves this particular problem by enriching team collaboration.

There are four major differences between DevOps and traditional software development:

1. DevOps can be viewed as a natural extension of the Agile movement, focusing on how to break down communication barriers between development and operations.
2. In the traditional software development lifecycle model, projects move through linear and sequential phases without rapid feedback loops or ongoing iterations, whereas the DevOps approach is more iterative.
3. The traditional software development approach takes a lot of time to deliver software projects because everybody works on a big chunk of software without proper planning. In contrast, DevOps work is divided into small batches, with each batch delivered quickly and then iterated on rapidly.
4. The traditional software development approach follows sequential steps that are hard to bypass, like gathering requirements, planning, writing code, testing, deploying to production, etc. But it doesn't work that way in the DevOps world — testers test the code alongside developers so that things don't have to be redone when problems arise. This makes for a more streamlined and efficient software development lifecycle.

Introduction to CI/CD

Continuous integration and continuous delivery (CI/CD), as an iterative process, requires developers to have a working build of the application on which they will release new features. It is a system that allows teams to integrate changes quickly without sacrificing quality or safety.

CI/CD is composed of three core tenets: continuous integration, continuous testing, and continuous delivery.

CI/CD works by using the principles of automation. When code is committed to the repository, it triggers a pipeline of build tasks that executes the following steps:

1. Check out the latest version of code from the repository
2. Perform compilation and unit tests
3. Generate artifacts such as documentation or reports on the build status (whether everything passes)
4. Start deployment on a staging environment (i.e., an identical copy of production)

WHY CI/CD?

Continuous integration and continuous delivery can be a blessing to both developers and customers. They allow you to test your code, find bugs, and fix them quickly before your customer has even had the chance to notice. In addition, once you have a working build on which you will release new features, CI/CD allows you to deploy it automatically to a staging environment that is identical to your production environment. This way, you don't have to rebuild the entire application every time it needs an update.

CI/CD is beneficial for developers because it enables them to work more efficiently and productively. CI/CD tools are of value to the developer because they can automate tasks like testing and deploying, which saves them time. For example, when a new build is ready, they can use CI/CD to automatically deploy that build to a staging environment or their customer's production environment. This way, tests are already in place before the update is live, so any new bugs will be caught before they make it into the hands of your customers.

CI AND CD FOR AUTOMATION

DevOps usually revolves around these simple pillars:

- CI (continuous integration)
- CD (continuous delivery)
- Continuous testing
- Containerization
- Continuous monitoring

As discussed above, CI/CD helps you automate releasing software from development to production by breaking down the process into stages. CI is the automated testing of code changes before they are released to production. CD is the automated release of code from development to production environments. Continuous monitoring ensures everything is always running smoothly. Containerization forms an integral part of the DevOps process, as it helps package the software and move it along the pipeline stages, making it easy for developers. It became popular with platforms like Docker that help companies package and ship their applications quickly. Most of the CI/CD tools today work with containerization in mind.

HOW TO IMPLEMENT CI/CD IN YOUR ORGANIZATION

BUILDING THE RIGHT MINDSET

If you aren’t a developer, learning to think like one is just as important as the tools in CI/CD. To be successful with CI/CD, you need to know how to automate and modify your current processes. You also need to be able to work closely with developers. When developing software, an individual or team will have to go through different stages of development, and it’s important to understand them to communicate effectively with the development team. These stages include research and analysis, architecture, design, coding, testing, release, and so on.

CHOOSING THE RIGHT TOOLS

There are various tools available for CI/CD, but before you decide which one to use, it's important to understand what your goals are. For example, what parts of your application will you want to integrate? Do you need deployments? And how many people will be working on the code? Once you have an idea of what features you'll want to implement in CI, it becomes easier

to choose which tool is best for your project. In addition to tools, hiring the right set of DevOps people can also help you quickly organize things and make decisions.

ADDING CI/CD INTO YOUR DEVELOPMENT PROCESS

It's not enough to just set up CI/CD and forget about it. You also need to make sure you're using it correctly. For example, there are a number of best practices that can help you increase your efficiency and reduce operational costs:

- **Using pipelines** – Developers should create a pipeline that goes from code check-in, through tests, and into production. This will allow developers and testers to collaborate more effectively on the process, as they both know what stage the code is in.
- **Automating release management** – The best way to do this is by setting up a series of automated release gates for each environment. This will help ensure that all bugs get caught before being released into your production environment.
- **Monitoring every step of the way** – You also want to keep an eye on how long it takes for jobs to run in CI/CD environments so you can optimize the entire process. The longer it takes for something to run, the more expensive it becomes — and the less time your team actually has available for developing new features or fixing bugs.

CAN YOU USE CI AND CD SEPARATELY?

CI is necessary for any software development project, but CD adds an additional layer to CI in the DevOps automation framework. We often see this confusion about whether to use CI, CD, or both. There are many benefits to using CI/CD together when developing and deploying your software, but sometimes it can be overkill for certain projects. You may just need CI. For example, if you're working on a small web application with a few developers, there's no point in spending the time and money required to set up CD. But for big organizations and startups working on big projects, both CI and CD help developers focus better on their jobs.

Conclusion

CI takes the first step toward a successful DevOps approach. CD goes a little further to change software development by deploying software multiple times a day with confidence. Automation is key to any successful CI/CD strategy, but it doesn't stop there; security is becoming a high priority to keep the software development pipeline clean and to minimize the attack surface. Successful implementation of CI/CD with the right culture, mindset, and people can help you win in this highly complicated DevOps landscape.

Pavan Belagatti, Developer Advocate at Harness
@pavanshippable on DZone | @Pavan_Belagatti on Twitter
@pavan-belagatti-growthmarketer on LinkedIn

Pavan is a global DevOps influencer, tech storyteller, and guest author at various publications. He has written hundreds of articles on cloud-native technologies. Pavan is on a mission to help developers deploy software with ease. He is currently working at Harness, the industry's first continuous-delivery-as-a-service platform as a developer advocate. In his free time, he likes singing and watching Bollywood movies :).

CONTRIBUTOR INSIGHTS

CI/CD for Cloud-Native Applications

Building Your CI/CD Process With DevOps and IaC Tools

By Boris Zaikin, Software & Cloud Architect at Nordcloud GmbH

Continuous integration (CI) and continuous delivery (CD) are crucial parts of developing and maintaining any cloud-native application. From my experience, proper adoption of tools and processes makes a CI/CD pipeline simple, secure, and extendable. Cloud native (or cloud based) simply means that an application utilizes cloud services. For example, a cloud-native app can be a web application deployed via Docker containers and uses Azure Container Registry deployed to Azure Kubernetes Services or uses Amazon EC2, AWS Lambda, or Amazon S3 services.

In this article, we will:

- Define continuous integration and continuous delivery
- Review the steps in a CI/CD pipeline
- Explore DevOps and IaC tools used to build a CI/CD process

An Overview of CI/CD

The continuous integration process is when software engineers combine all parts of the code to validate before releasing tested applications to dev, test, or production stages. CI includes the following steps:

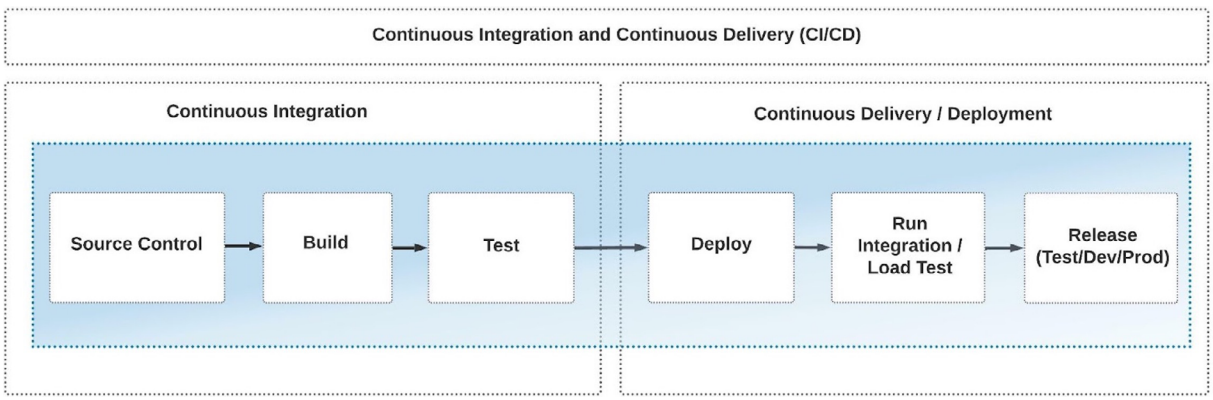
1. **Source control** – Pulls the latest source code of the application from source control.
2. **Build** – Compiles, builds, and validates the code — or creates bundles and "linting" in terms of JavaScript/Python code.
3. **Test** – Runs unit tests and validates coding styles.

Following CI is the continuous delivery process, which includes the following steps:

1. **Deploy** – Places prepared code into the test (or stage) environment.
2. **Testing** – Runs integration and/or load tests. This step is optional as an application can be small and not have a huge load.
3. **Release** – Deploys an application to the development, test, and production stages.

In my opinion, CI and CD are two parts of one process. However, in the cloud-native world, you can implement CD without CI. You can see the whole CI/CD process in the diagram below:

Figure 1



The Importance of CI/CD in Cloud-Native Application Development

Building a successful cloud-native CI/CD process relies on the Infrastructure-as-Code (IaC) toolset. Many cloud-native applications have integrated CI/CD processes that include steps to build and deploy the app and provision and manage its cloud resources.

HOW IAC SUPPORTS CI/CD

Infrastructure as Code is an approach where you can describe and manage the configuration of your application's infrastructure. Many DevOps platforms support the IaC approach, integrating it directly into the venue — for example, Azure DevOps, GitHub, GitLab, and Bitbucket support YAML pipelines. With the YAML pipeline, you can build CI/CD processes for your application with infrastructure deployment. Below, you can see DevOps platforms that can easily be integrated with IaC tools:

- Azure Resource Manager, Bicep, and Farmer
- Terraform
- Tekton
- Pulumi
- AWS CloudFormation

Building a Successful Cloud-Native CI/CD Process

In many cases, the CI/CD process for cloud-native applications includes documenting and deploying infrastructure using an IaC approach. The IaC approach allows you to:

- Prepare infrastructure for your application before it is deployed
- Add new cloud resources and configuration
- Manage existing configurations and solve problems like "environment drift"

Environment drift problems appear when teams have to support multiple environments manually. Drift can lead to an inconsistent environment setting that causes application outages. A successful CI/CD process with IaC relies on what tool and platform you use.

Let's have a look into the combination of the most popular DevOps and IaC tools.

AZURE DEVOPS

Azure DevOps is a widely used tool to organize and build your cloud-native CI/CD process, especially for Azure Cloud. It supports UI and YAML-based approaches to building pipelines. I prefer using this tool when building an automated CI/CD process for Azure Cloud. Let's look at a simple YAML pipeline that creates a virtual machine (VM) in Azure DevTest Labs:

```
. . . . . .
jobs:
- deployment: deploy
  displayName: Deploy
  pool:
    vmImage: 'ubuntu-latest'
  environment: ${{ parameters.environment }}

  strategy:
    runOnce:
      deploy:
        steps:
        - checkout: self
        - task: AzurePowerShell@4
          displayName: 'Check vm-name variable exists'
          continueOnError: true
```

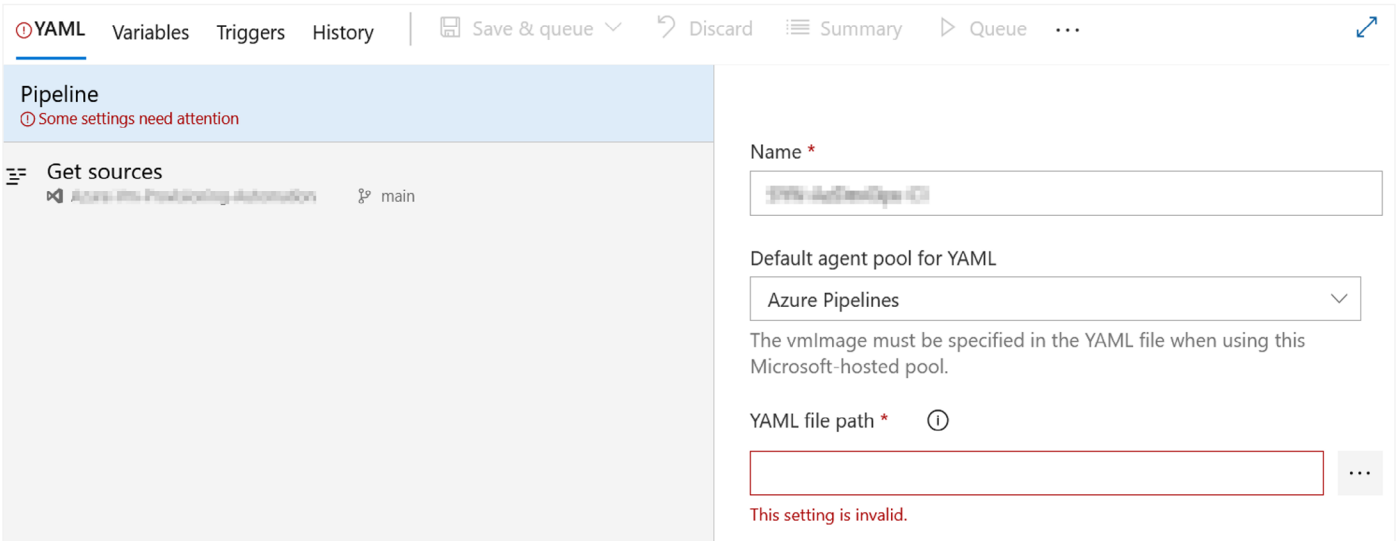
Code continues on next page

```
          inputs:
            azureSubscription: ${{ parameters.azSubscription }}
            scriptType: "inlineScript"
            azurePowerShellVersion: LatestVersion
            inline: |
              $vm_name="$(vm-name)"
              echo $(vm-name) - $vm_name
              #if ([string]::IsNullOrWhitespace($vm_name))
              {
                  throw "vm-name is not set"
              }

        - task: AzureResourceManagerTemplateDeployment@3
          displayName: 'New deploy VM to DevTestLab'
          inputs:
            deploymentScope: 'Resource Group'
            azureResourceManagerConnection: ${{ parameters.azSubscription }}
            subscriptionId: ${{ parameters.idSubscription }}
            deploymentMode: 'Incremental'
            resourceGroupName: ${{ parameters.resourceGroup }}
            location: '$(location)'
            templateLocation: 'Linked artifact'
            csmFile: templates/vm.json
            csmParametersFile: templates/vm.parameters.json
            overrideParameters: '-labName  "${{parameters.devTestLabsName}}"
                                 -vmName ${{parameters.vmName}} -password ${{parameters.password}}
                                 -userName ${{parameters.user}}
                                 -storageType
. . . . . .
```

To simplify the pipeline listing, I've shortened the example above. As you can see, the pipeline code can also be generic; therefore, you can reuse it in multiple projects. The pipeline's first two steps are in-line PowerShell scripts that validate and print required variables. Then this pipeline can be integrated easily into Azure DevOps, as shown in the image below:

Figure 2



The last step creates the VM in Azure DevTest Labs using Azure Resource Manager (ARM) templates and is represented via the JSON format. This step is quite simple: It sends (overrides) variables into the ARM scripts, which Azure uses to create a VM in the DevTest Labs.

AWS CLOUDFORMATION

AWS CloudFormation is an IaC tool from the AWS Cloud stack and is intended to provision resources like ES2, DNS, S3 buckets, and many others. CloudFormation templates are represented in JSON and YAML formats; therefore, they can be an excellent choice to build reliable, cloud-native CI/CD processes. Also, many tools like Azure DevOps, GitHub, Bitbucket, and AWS CodePipelines have integration options for CloudFormation.

Below is an example of what an AWS CloudFormation template may look like, which I've shortened to fit in this article:

```
{
  "AWSTemplateFormatVersion" : "2010-09-09",
  "Parameters" : {
     "AccessControl" : {
     "Description" : " The IP address range that can be used to access the CloudFormer tool. NOTE:
We highly recommend that you specify a customized address range to lock down the tool.",
     "Type": "String",
     "MinLength": "9",
     }
  },
     "Mappings" : {
       "RegionMap" : {
        "us-east-1"        : { "AMI" : "ami-21341f48" },
         }
  }
  "Resources" : {
     "CFNRole": {
     "Type": "AWS::IAM::Role",
     "Properties": {
     "AssumeRolePolicyDocument": {
            "Statement": [{
            "Effect": "Allow",
            "Principal": { "Service": [ "ec2.amazonaws.com" ] },
            "Action": [ "sts:AssumeRole" ]
            }]
     },
     "Path": "/"
     }
     },
  }
}
```

CloudFormation templates contain sections including:

- `Parameters` – You can specify input parameters to run templates from the CLI or pass data from AWS CodePipeline (or any other CI/CD tool).
- `Mappings` – You can match the key to a specific value (or set of values) based on a region.
- `Resources` – You can declare the resources included, answering the "what will be provisioned" question, and you can adjust the resource according to your requirement using a parameter.

AWS CloudFormation templates look similar to Azure ARM templates as they do the same work but for different cloud providers.

GOOGLE CLOUD DEPLOYMENT MANAGER

Google Cloud (GC) offers Cloud Deployment Manager, the all-in-one tool that includes templates that describe resources for provisioning and templates to build CI/CD pipelines. The templates support:

- Python 3.x
- Jinja 2.10.x
- YAML

Let's explore a deployment of the VM using Jinja templates — it looks common to YAML:

```
resources:
- type: compute.v1.instance
  name: {{env["project"]}}-deployment-vm
  properties:
      zone: {{properties["zone"]}}
      machineType:https://www.googleapis.com/compute/v1/projects/{{env["project"]}}/zones/
{{properties["zone"]}}/machineTypes/f1-micro
      disks:
      - deviceName: boot
      type: PERSISTENT
      boot: true
      autoDelete: true
      initializeParams:
      sourceImage: https://www.googleapis.com/compute/v1/projects/debian-cloud/global/images/family/
debian-9
      networkInterfaces:
      - network: https://www.googleapis.com/compute/v1/projects/{{env["project"]}}/global/networks/
default
      accessConfigs:
      - name: External NAT
      type: ONE_TO_ONE_NAT
```

The example above is similar to ARM and CloudFormation templates as it describes resources to deploy. In my opinion, the YAML/Jinja 2.10.x format works better than the JSON-based structure because:

- YAML increases readily and can read much faster than JSON
- Teams can find and fix errors in YAML and Jinja faster than in JSON
- YAML pipelines (with small adaptations) can be reused on many other platforms

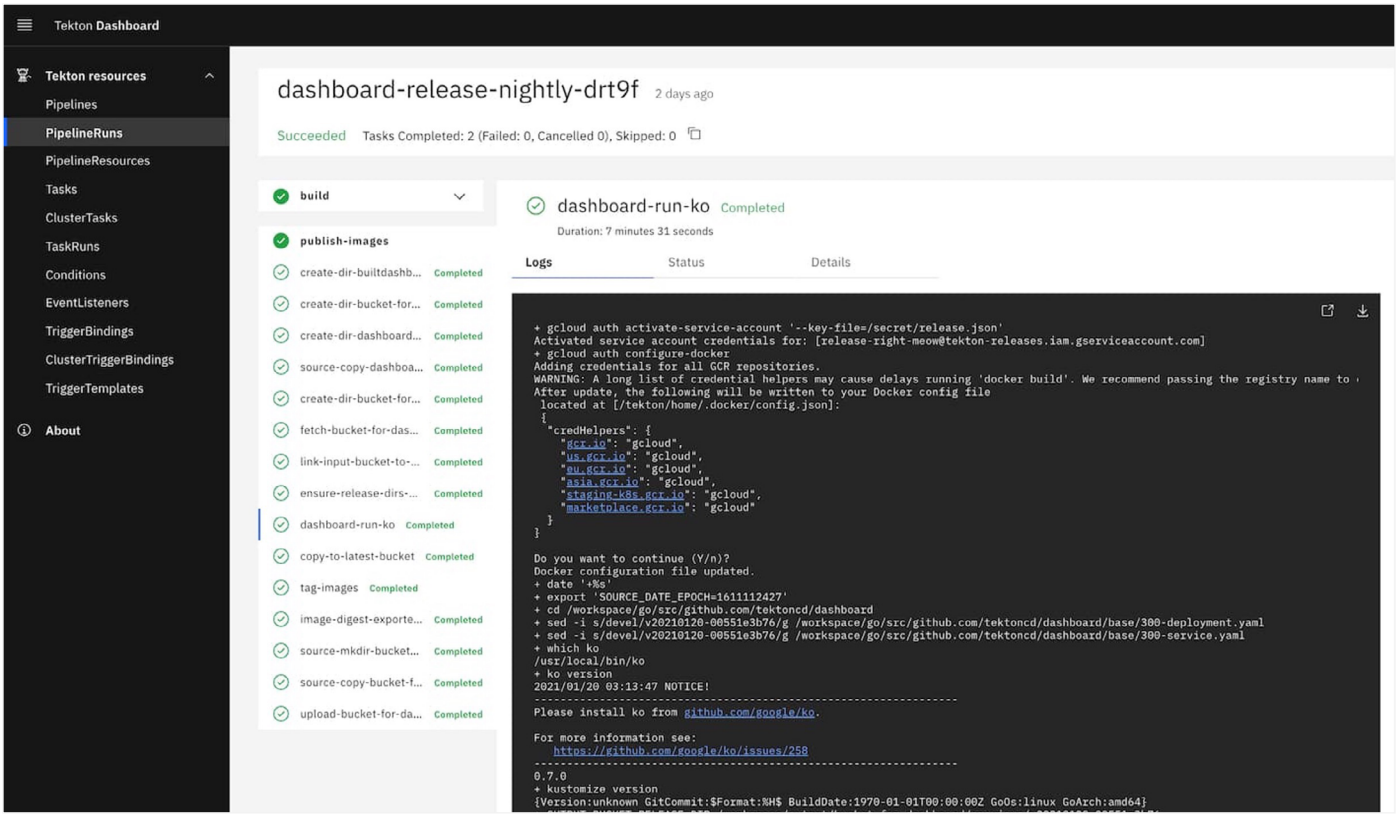
You can find an extended version of this example in the GitHub gist.

TEKTON AND KUBERNETES

Tekton, supported by the CD Foundation (part of the Linux Foundation), is positioned as an open-source CI/CD framework for cloud-native applications based on Kubernetes. The Tekton framework's components include:

- **Tekton Pipelines** – Most essential and are intended to build CI/CD pipelines based on Kubernetes Custom Resources.
- **Tekton Triggers** – Provide logic to run pipelines based on an event-driven approach.
- **Tekton CLI** – Built on top of the Kubernetes CLI and allows you to run pipelines, check statuses, and manage other options.
- **Tekton Dashboard** and **Hub** – Use web-based graphical interfaces to run pipelines and observe pipeline execution logs, resource details, and resources for the entire cluster (see dashboard in Figure 3 on next page).

Figure 3



I like the idea behind Tekton Hub — that you can share your pipelines and other reusable components. Let's look at a Tekton Pipeline example:

```
apiVersion: tekton.dev/v1beta1
kind: Pipeline
metadata:
  name: say-things
spec:
  tasks:
    - name: first-task
      params:
        - name: pause-duration
          value: "2"
        - name: say-what
          value: "Hello, this is the first task"
      taskRef:
        name: say-something
    - name: second-task
      params:
        - name: say-what
          value: "And this is the second task"
      taskRef:
        name: say-something
```

The pipeline code above is written in the native Kubernetes format/manifests and represents a set of tasks. Therefore, you can build native cross-cloud CI/CD processes. In the tasks, you can add steps to operate with Kubernetes resources, build images, print information, and many other actions. You can find the complete tutorial for Tekton Pipelines here.

TERRAFORM, AZURE BICEP, AND FARMER

Terraform is a leading platform for building reliable CI/CD processes based on the IaC approach. I will not go in depth on Terraform as it requires a separate article (or even book). Terraform uses a specific language that simplifies building CI/CD templates. Also, it allows you to reuse code parts, adding dynamic flavor to the CI/CD process. Therefore, you can build templates that are much better than ARM/JSON templates. Let's see a basic example of Terraform template code:

```
terraform {
  required_providers {
  }
  backend "remote" {
    organization = "YOUR_ORGANIZATION_NAME"
    workspaces {
      name = "YOUR_WORKSPACE_NAME"
    }
  }
}
```

The basic template above contains the resources, providers, and workspace definition. The same approach follows the Azure Bicep and Farmer tools. These Terraform analogs can drastically improve and shorten your code. Let's look at the Bicep example below:

```
param location string = resourceGroup().location
param storageAccountName string = 'toylaunch${uniqueString(resourceGroup().id)}'

resource storageAccount 'Microsoft.Storage/storageAccounts@2021-06-01' = {
  name: storageAccountName
  location: location
  sku: {
    name: 'Standard_LRS'
  }
  kind: 'StorageV2'
  properties: {
    accessTier: 'Hot'
  }
}
```

The Bicep code above deploys storage accounts in the US region. You can see that the JSON ARM example acts the same. In my opinion, Terraform and Azure Bicep are shorter and much easier to understand than ARM templates. The Farmer tool can also show impressive results in readability and type-safe code:

```
 // Create a storage account with a container
let myStorageAccount = storageAccount {
    name "myTestStorage"
    add_public_container "myContainer"
}

// Create a web app with application insights that's connected to the storage account.
let myWebApp = webApp {
    name "myTestWebApp"
    setting "storageKey" myStorageAccount.Key
}

// Create an ARM template
let deployment = arm {
    location Location.NorthEurope
    add_resources [
```

Code continues on next page

```
        myStorageAccount
        myWebApp
    ]
}

// Deploy it to Azure!
deployment
|> Writer.quickDeploy "myResourceGroup" Deploy.NoParameters
```

Above, you can see how to deploy your web application to Azure in 20 lines of easily readable and extensible code.

PULUMI

The Pulumi framework follows a different approach to building and organizing your CI/CD processes: It allows you to deploy your app and infrastructure using your favorite programming language. Pulumi supports Python, C#, TypeScript, Go, and many others. Let's look at the example below:

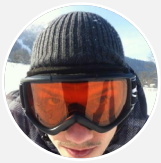
```
public MyStack()
{
    var app = new WebApplication("hello", new()
    {
        DockerImage = "strm/helloworld-http"
    });

    this.Endpoint = app.Endpoint;
}
```

This part of the code deploys the web app to the Azure cloud and uses Docker containers to spin up the web app. You can find examples of how to spin up the web app in an AKS cluster as well as from the Docker container here.

Conclusion

Building a cloud-native CI/CD pipeline for your application can be a never-ending story if you don't know the principles, tools, and frameworks best suited for doing so. It is easy to get lost in various tools, providers, and buzzwords, so this article aimed to explain what CI/CD cloud-native applications are and walk you through the widely used tools and principles of building reliable CI/CD pipelines. Having this guide helps you to feel comfortable while designing CI/CD processes and choosing the right tools for your cloud-native application.



Boris Zaikin, Software & Cloud Architect at Nordcloud GmbH

@borisza on DZone | @boris-zaikin on LinkedIn | boriszaikin.com

I'm a Certified Software and Cloud Architect who has solid experience designing and developing complex solutions based on the Azure, Google, and AWS clouds. I have expertise in building distributed systems and frameworks based on Kubernetes and Azure Service Fabric. My areas of interest include enterprise cloud solutions, edge computing, high load applications, multitenant distributed systems, and IoT solutions.

CONTRIBUTOR INSIGHTS

Infrastructure Provisioning for Cloud-Native Applications

By Samir Behara, Platform Architect at EBSCO

Enterprises are embracing cloud-native technologies to migrate their monolithic services to a microservices architecture. Containers, microservices, container orchestration, automated deployments, and real-time monitoring enable you to take advantage of cloud-native capabilities. However, the infrastructure required to run cloud-native applications differs from traditional ones.

This article will describe Infrastructure as Code, its benefits, and the popular IaC tools. You will also learn how to model infrastructure as part of the CI/CD pipeline and incorporate it into the standard development lifecycle.

What Is Infrastructure as Code?

IaC helps to manage and provision infrastructure resources through code and automation. In a traditional on-premises environment, operators log into the server and execute a series of commands via the command line or console to perform changes. However, these manual configurations are prone to drifts and create inconsistent environments. It's also a time-consuming process to deploy similar changes across your infrastructure. There is no quick way to verify the correctness of these manual changes. If there are issues, it isn't easy to recreate them.

In comes IaC to simplify the management and provisioning of your infrastructure resources. With IaC, you don't make manual changes to servers, instances, containers, or environments. Both the creation and modification of the resources are automated. IaC is the practice of using code to create, describe, and manage infrastructure resources.

Infrastructure as Code Benefits

Let's now take a close look at how IaC can benefit your organization.

INCREASED PRODUCTIVITY THROUGH AUTOMATION

Making infrastructure changes and deploying them is a repetitive process, and it is time-consuming if your developers/operations team needs to do these manually in a recurring interval. With IaC, you can focus on coding and be more productive by automating the infrastructure deployments. As a result, IaC enables faster time to market for your business features.

REPEATABLE DEPLOYMENTS WITH HIGH PREDICTABILITY

Having your infrastructure defined in source control lets you automate the deployment process and enable developers to follow the software development lifecycle for infrastructure changes. It also empowers developers to perform successful deployments backed by efficient practices like peer review, static code analysis, and automated testing.

IMPROVED CONSISTENCY WITH MINIMAL CONFIGURATION DRIFT

Performing infrastructure changes manually can lead to inconsistency between the servers and is generally the primary reason for configuration drifts. Applying changes manually is error-prone, and so IaC saves the day by standardizing the infrastructure modification process. The deployment process is faster, repeatable, and more consistent with automation and version control.

DOCUMENTED PROCESS FOR DEPLOYING CHANGES

It would be best not to completely rely upon your sysadmin or operations team to deploy infrastructure changes. This increased dependency on selected operators to make server changes can create blocker scenarios when they are not available. In the long run, it is not a scalable solution. Having the infrastructure code in source control provides visibility to everyone in your organization about the current state. Developers can deploy infrastructure changes if all the stages in the CI/CD pipeline pass. If there are any issues, it's easy to troubleshoot by looking at the change history in the source control.

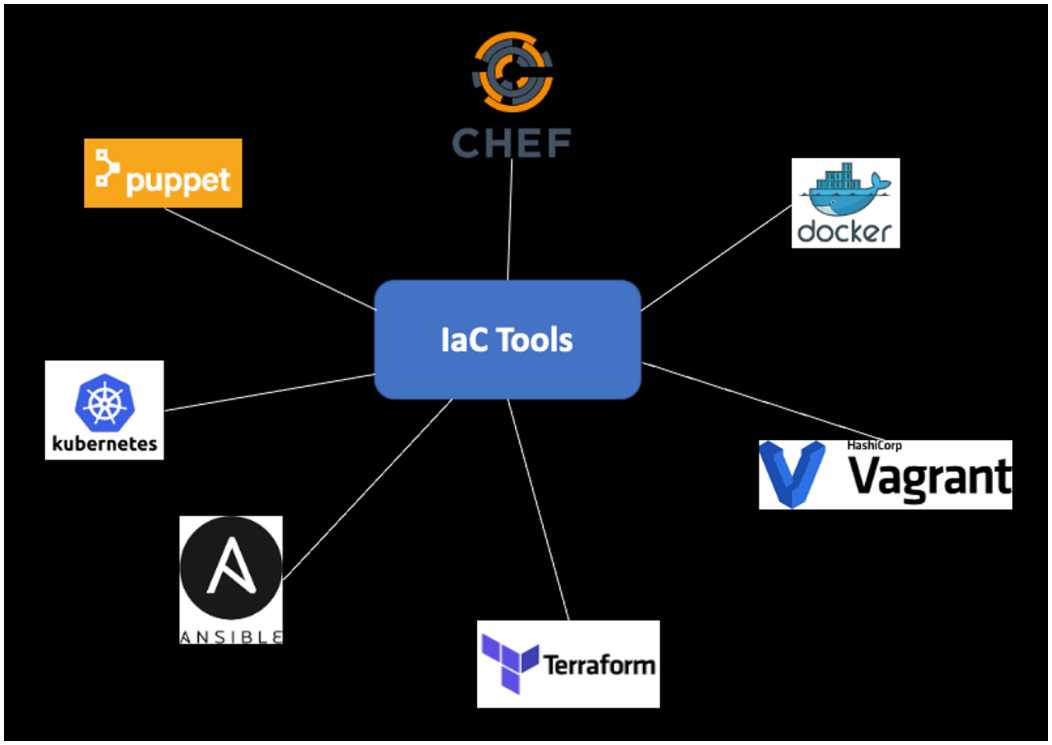
Figure 1: Infrastructure as Code in action

Infrastructure as Code Tools

Some open-source IaC tools can be categorized into various groups:

- **Configuration management tools** – Chef, Puppet, and Ansible are popular tools that allow you to install, update, and manage resources on existing infrastructure.
- **Server templating tools** – Tools like Docker and Vagrant allow you to create an image from a configuration file, which is then used to provision infrastructure in a repeatable manner. They promote the idea of an immutable infrastructure, which we will explain later in this article.
- **Container orchestration tools** – Tools like Kubernetes and Docker Swarm allow you to orchestrate container workloads in a dynamic cloud-native landscape.
- **Provisioning tools** – Tools like Terraform, Azure Resource Manager, Google Cloud Deployment Manager, and AWS CloudFormation allow you to provision servers and other resources in the respective cloud environment.

Figure 2: Infrastructure-as-Code tools



Mutable vs. Immutable Infrastructure

With immutable infrastructure, you create a new server with the revised configuration if you want to modify an existing server. In a cloud-native environment, where containers are spinning up and down every minute, immutability is a required feature. You should package your application and its dependencies into a container image, and as you want to modify the configuration, you deploy a new container version.

The pets vs. cattle analogy for infrastructure management is popular in the dynamic container space. You treat your infrastructure resources like cattle so that you can delete and build them from scratch when a configuration change is required. Pets are indispensable resources where you make configuration changes in place. Immutability helps simplify your operations, minimize drift, boost consistency between environments, and build a secure infrastructure.

Declarative vs. Procedural Approach

With a procedural approach, you write separate scripts to achieve the desired state. Over time, you tend to have many scripts that have been applied to your environment, and you can review the modifications via change history. The order of execution of the scripts matters, or else you end up with a different end state. Tools like Chef and Puppet follow the procedural style of infrastructure management. Please find below some of the frequently used Puppet commands:

Table 1

Command	Action
`puppet agent`	Retrieve configuration from a remote server and apply to localhost
`puppet apply`	Apply individual manifests to the local system
`puppet module`	Find, install, and manage modules in the puppet repository
`puppet describe`	Displays metadata about puppet resource types
`puppet config`	Review and modify settings in the puppet configuration file

With the declarative approach, you maintain the desired state in a configuration template. If you want to make any modifications, you update the same template to reflect the desired state and let the IaC tool generate the difference script and apply it to the environment. Tools like Terraform follow the declarative approach, where the code in source control always reflects the current state of the infrastructure. The declarative approach helps you to create reusable code since you are just focused on describing the desired state and offloading the complexity of syncing the current and desired state to the IaC tool.

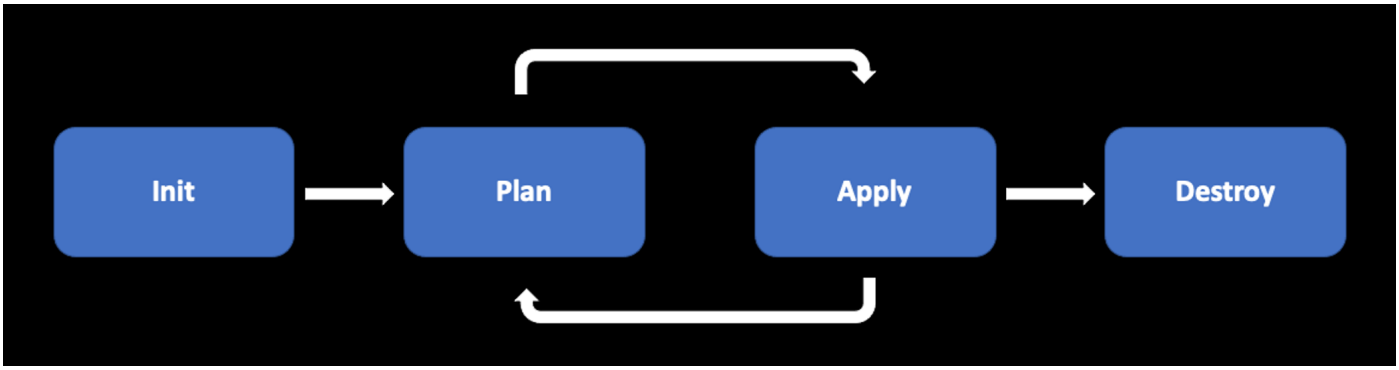
Using Terraform to Manage Infrastructure

Terraform is a cloud-agnostic, open-source tool for infrastructure provisioning. Created by HashiCorp in 2014, Terraform uses HashiCorp Configuration Language (HCL) to describe infrastructure code. It supports many providers and can help manage resources in individual cloud providers, such as AWS, Azure, and Google Cloud. Terraform is backed by a large and growing community. The primary function of Terraform is to help you create, modify, and destroy infrastructure resources. To provision resources using Terraform, you need to use the following commands:

Table 2

Command	Action
`terraform init`	Initializes the working directory that contains the configuration files
`terraform plan`	Compares the current state and desired state and creates an execution plan for making changes
`terraform apply`	Applies the changes that were proposed by the Terraform plan and ensures that the desired state is reached
`terraform destroy`	Cleans up the infrastructure resources that are managed by the configuration files

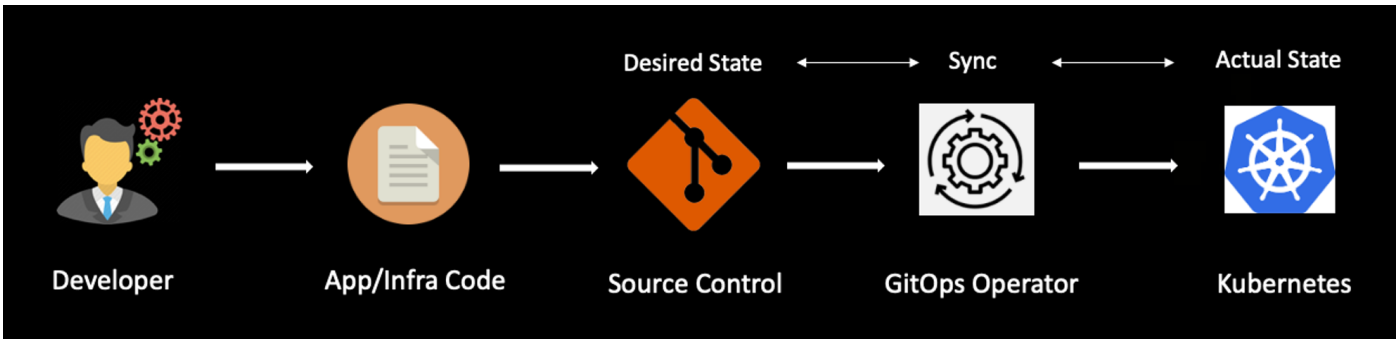
Figure 3: Terraform lifecycle



Infrastructure Automation With GitOps

GitOps workflows apply DevOps best practices for application development (version control, code review, automated deployments, etc.) to infrastructure deployments. GitOps is based on the declarative model, where configuration files get stored in Git, and approved changes get automatically deployed to the environment. As shown in Figure 4, the GitOps operator ensures that the desired state stored in Git is in sync with the actual state of the deployed infrastructure. Enterprises are rapidly adopting GitOps to manage their Kubernetes cloud-native environments at scale. Applying GitOps practices enables continuous deployments with proper auditing and high reliability.

Figure 4: GitOps pipeline



Conclusion

IT infrastructures are growing exponentially and embedding automation into every stage of the delivery pipeline ensures faster and more consistent deployments. Automating your IaC processes is as important as the automation of your application deployment. You can leverage multiple IaC tools together to automate your infrastructure management. Each of the tools mentioned in this article has its strengths and weaknesses, so having a sound understanding of those and selecting the right tool that fits your use case is critical. IaC is the future, and organizations are readily embracing it to increase the reliability of their infrastructure.

Samir Behara, Platform Architect at EBSCO

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

Samir Behara is a Platform Architect with EBSCO and builds software solutions using cutting edge cloud-native technologies. He is a Microsoft Data Platform MVP with over 16 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

3 Steps to Developing a Successful GitOps Model

By Marija Naumovska, Co-Founder & Technical Writer at Microtica

What Is GitOps and Why Is it Important for an Organization?

GitOps is a model to automate and manage infrastructure and applications. This is done by using the same DevOps best practices that many teams already use, such as version control, code review, and CI/CD pipelines. While implementing DevOps, we've found ways to automate the software development lifecycle, but when it comes to infrastructure setup and deployments, it's still mostly a manual process. With GitOps, teams can automate the infrastructure provisioning process. This is due to the ability to write your Infrastructure as Code (IaC), version the code in a Git repository, and apply continuous deployment principles to your cloud delivery.

Companies have been adopting GitOps because of its great potential to improve productivity and software quality. GitOps best serves organizations that develop cloud-native solutions based on containerization and microservices.

How Does GitOps Improve the Lives of Developers and Operations?

The increased infrastructure automation that comes with GitOps creates the opportunity to develop a more "self-service" approach for application developers. Rather than negotiating for cloud resources, skilled developers can use Infrastructure as Code to declare their cloud resource requirements. This becomes the desired state of the infrastructure, stored centrally and serving as the immutable reference point between the requirements as stated in the code and the actual state of the live environment.

The self-service approach is liberating for developers. It makes them more productive, allows them to focus on innovation, and gets their apps to market more quickly. Additionally, it avoids the mire that can be introduced when developers and operators need to negotiate resources.

On the other hand, there is a frequent misconception that the increased automation of operations means that Ops teams need fewer people and Ops's role in the pipeline is marginalized. Our view is the exact opposite; we believe that modern approaches such as GitOps and the Internal Developer Platform provide exciting opportunities for Ops (Platform Team) to enhance their skills and create more value for the organization. In a high-performing, cloud-native software development organization that embraces GitOps, you are likely to find a growing Platform team that is helping to make it all work.

The actual technology used by the Platform team may vary. In some cases, this could just be a closed PaaS solution. In others, it could be a combination of various tools to create a bespoke platform tailored to the organization's needs. This gives them the ability to exert more influence and control over the infrastructure resources and architecture and create "guardrails" that enforce a simple, efficient, and standardized approach to cloud-native application deployment.

GitOps helps improve the collaboration between developers and operation teams, increases their productivity, and increases deployment frequency. It enhances the developers' experience by enabling them to contribute with features without the need to know the underlying infrastructure. At the same time, it gives control to operations with code reviews and approvals. With these improvements, teams can release faster and more secure to maintain their position in the market.

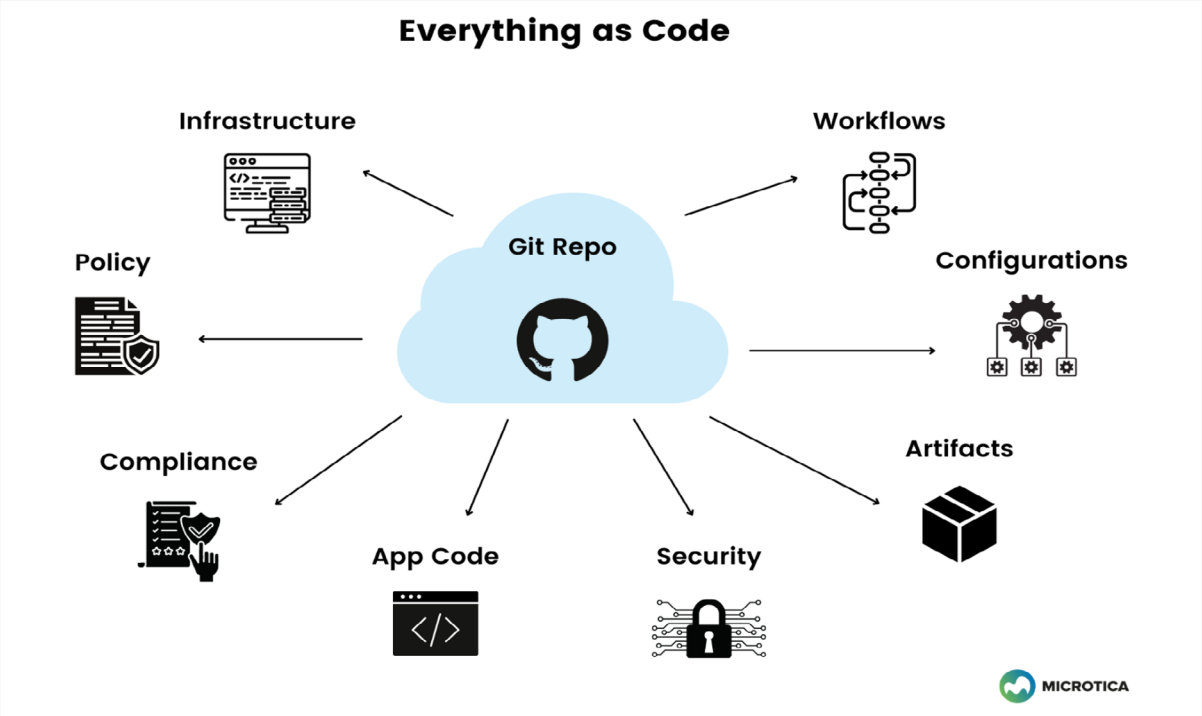
What Are the Three Must-Do Steps to Implement GitOps?

To experience the most prominent advantages of implementing a GitOps model in your company, like standardization and consistency in your overall workflow, here is what you need to consider.

EVERYTHING AS CODE

- Declare your IaC.
- Use a Git repository for your IaC development.
- Replicate the practices that are part of your application code lifecycle to your infrastructure code as well.
- Using technologies like Docker and Kubernetes, define your environment, versions, configurations, and dependencies as code, and ensure they get enforced on runtime.
- Gradually extend the GitOps model onto anything that can be defined as code, like security, policy, compliance, and all operations beyond infrastructure.

Figure 1: Everything as code



Declarative code improves readability and maintenance. CloudFormation, Terraform, Pulumi, and Crossplane are some possible declarative languages you can use to define the configuration of how you want your infrastructure to look.

When everything is defined as code, you can use a Git repository for your development and explore benefits, such as version control, collaboration, and audits.

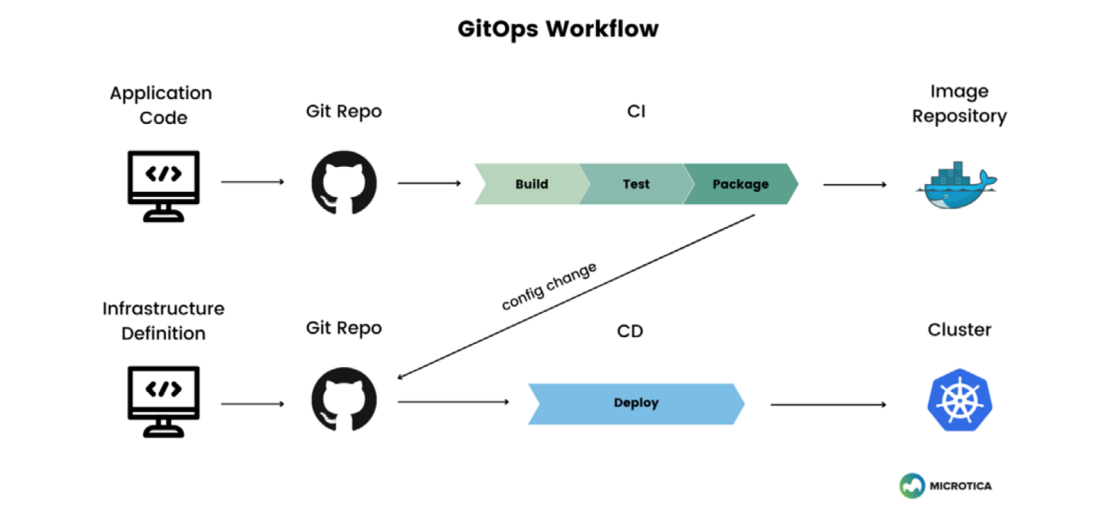
REVIEW PROCESS

A proper Git flow consists of:

- The main branch, which usually represents an environment, like dev, test, stage, prod, and the state running on that environment.
- When developers need to introduce changes to the code, they create a new branch from the main branch.
- When the changes are ready, the developers create a pull request that should be reviewed by operations to validate and approve. Security and compliance experts can also be involved in this stage to validate the environment's state properly.
- Once approved, the code can be merged into the main branch and delivered to test or production.

Using this workflow, you can track who made which change and ensure the environment has the correct version of the code.

Figure 2: GitOps workflow



If you already take advantage of the Git flow system by working with feature branches and pull requests, then you won't need to invest much in a new process for your GitOps workflow. Furthermore, as your infrastructure (and other operations) are defined as code, you'll be able to implement the same practices for code review.

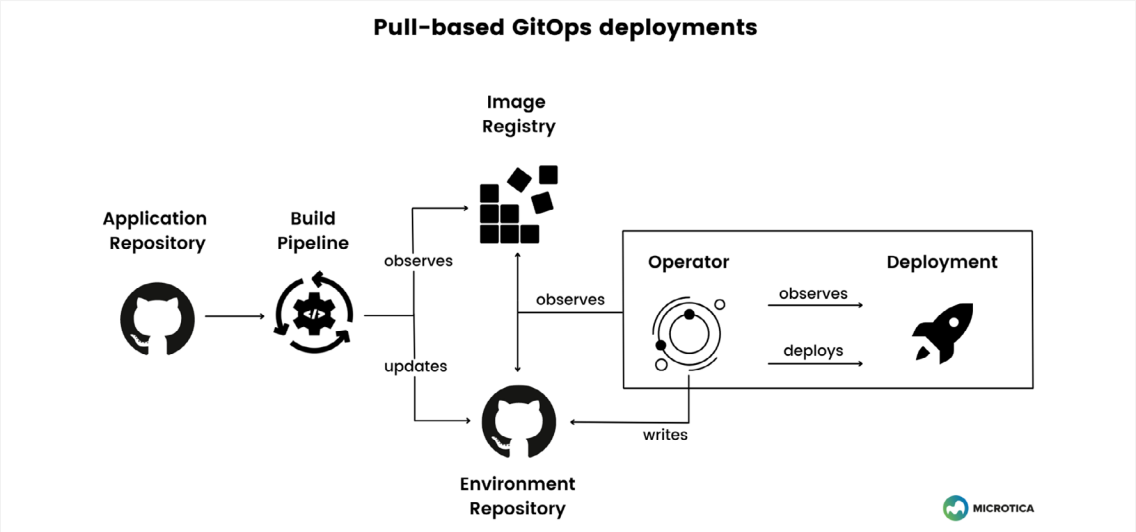
SEPARATE BUILD AND DEPLOY PROCESS (CI AND CD)

- A CI process is responsible for building and packaging application code into container images.
- The CD process executes the automation to bring the end state in line with the system's desired state, described in the repository code.

Ultimately, GitOps sees CI and CD as two separate processes — CI as a development process and CD as an operational process.

A GitOps approach commonly used to separate these processes is to introduce another Git repository as a mediator. This repo contains information about the environment, and with every commit there, the deployment process is triggered. There is a component, called the operator, residing between the pipeline and the orchestration tool. The operator constantly compares the target state in the environment repository with the actual state in the deployed infrastructure. The operator changes the infrastructure to fit the environment repository if it detects any changes. Also, it monitors the image registry to identify new versions of images to deploy. This way, the CI process never touches the underlying infrastructure (e.g., the Kubernetes cluster).

Figure 3: Pull-based GitOps deployments



Decoupling the build pipeline from the deployment pipeline is a powerful protection against misconfigurations and helps achieve higher security and compliance.

Conclusion

GitOps, as an operational model, uses DevOps practices known to many teams. Using GitOps, you can automate the infrastructure provisioning process and use Git as a single source of truth for your infrastructure. Therefore, to create a successful GitOps model, you need a declarative definition of the environment.

It would be best if you also had a pull request workflow in your team. To be able to collaborate on the infrastructure code and create operational changes, you should open a pull request. Senior DevOps engineers and security experts then review the pull request to validate the changes and merge them into the main branch if everything is okay.

For a full GitOps implementation, you need to have CI/CD automation for provisioning and configuration of the underlying environment and the deployment of the defined code.

Lastly, there should be a supporting organizational culture inside the company. In our experience, a GitOps approach has made it natural to get to a structure in which developers enjoy increased automation from self-service infrastructure resources and platform engineers enjoy taking on a more influential role in the organization. In that regard, it's a win-win approach that makes everyone more aligned and fulfilled.

Marija Naumovska, Co-Founder & Technical Writer at Microtica

@marulka on DZone | @mmarulka on Twitter | microtica.com

As a co-founder of Microtica, Marija helps developers get their applications deployed on the cloud in minutes. She's a Software Engineer with 8+ years of experience, who now works as a product person and technical writer full time. She writes about cloud, DevOps, GitOps, and Kubernetes topics.

CONTRIBUTOR INSIGHTS

Continuous Test Automation Using CI/CD

How CI/CD Has Revolutionized Automated Testing

By Justin Albano, Software Engineer at IBM

There have been a few breakthroughs throughout the short history of software development that have completely revolutionized the way we write and release code. From Object-Oriented Programming to web-based languages like JavaScript and TypeScript, these innovations have moved software engineering by leaps and bounds.

One of the more recent groundbreaking innovations is automated testing. Prior to automated testing, a large portion of the test cases for our software were executed manually. This painstaking process has many flaws, including:

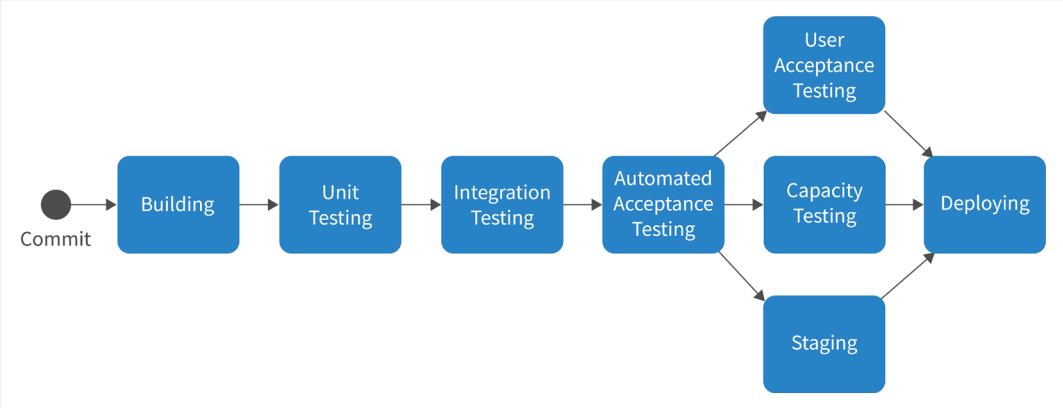
- Inconsistent execution of test cases
- Manualized setup of testing environments
- Tediousness and slowness
- Inconsistent format of test results

Automated testing — and the introduction of Continuous Integration (CI) and Continuous Delivery (CD) — has completely transformed the quality and the cadence with which we release our software. In this article, we will delve into the CI/CD pipeline and see how automated testing can be used to dramatically improve the quality and swiftness of our software releases. We will also look at some of the most popular and practical tools that we can use to create our CI/CD pipelines.

The CI/CD Pipeline

In order to release software, we must fulfill a set of business needs. In some cases, these business needs include a quick set of system tests and a suite of User Interface (UI) tests, while other releases may require more involved needs. Regardless of the complexity, these business needs can be conceptualized as a set of steps that are executed in serial and in parallel. In the CI/CD vernacular, each step is called a **stage**, and the set of ordered stages is called a **pipeline**. Below is an example pipeline:

Figure 1



The particular stages in the pipeline will vary based on the business needs of the project, but all pipelines will be executed when a **trigger** (such as a commit) is activated. Once the execution of the pipeline starts, each stage is executed one-by-one; when one stage successfully completes, the next stage is executed.

When a set of parallel stages is reached, such as the User Acceptance Testing, Capacity Testing, and Staging, stages in the example above, all of the stages are executed at the same time. The pipeline proceeds when all of the parallel stages are successfully completed. For example, execution of the Deploying stage will not start until User Acceptance Testing, Capacity Testing, and Staging successfully complete.

There is no requirement that all stages of a CI/CD pipeline must be automated, and in some cases, it can be difficult to introduce automated test cases into a CI/CD pipeline. For example:

- **Unclear business needs and specifications** – In most cases, the difficulty in defining automated tests stems from a lack of clarity about the business needs of our project (which defines our CI/CD pipeline) and the specifications of our software under test. Before we create stages in our CI/CD pipelines, we must understand *what* we need to test and *why* we are testing it.
- **UI tests** – UI tests can be difficult to automate due to the visual and fluctuating nature of UIs. We can overcome this by using a UI test framework, such as Selenium.
- **Inconsistent reporting** – Many CI/CD pipeline tools include a test summary that displays the number of executed and successful tests completed in a stage. This summary requires a consistent, well-known report to be generated by our automated tests. We can meet this requirement by using an automated testing tool whose reporting format is widely known, such as JUnit (or any of the xUnit frameworks) and Cucumber.

While there may be instances when manual testing is required, the greatest benefit of CI/CD pipelines is achieved when all tests, including UI tests, are automated.

Automated Testing in the CI/CD Pipeline

The major advantage to utilizing automated testing in a CI/CD pipeline is that a single commit can be tested against a gauntlet of tests — including unit, integration, system, performance, and acceptance tests — and then be deployed to a production system without having any human interaction. For example, even on a large-scale project, it is possible to have a single engineer make a commit that will automatically result in a feature being deployed to production in a few minutes or hours.

Conversely, an automated pipeline ensures that failed tests prohibit a feature from being deployed to production. For example, if a developer adds a new feature, and a unit or integration test fails, the execution of the pipeline immediately stops, and the feature is not deployed. The developer is then notified of the test failure and can track down the bug to the commit that triggered the failed execution of the pipeline.

In addition to the benefits reaped for deployment and release, there are a host of benefits that automated testing brings to the quality of the code itself:

- Documentation of its intended behavior
- Reduction in the number of regressions
- Decoupling into smaller, more independent components
- Reduction in test execution time
- Involvement of stakeholders in the generation of test specifications (i.e., acceptance tests)

Although it may not be possible for all tests in a CI/CD pipeline to be automated, in order to garner the greatest benefit from our pipelines, we should strive to maximize the number of automated stages and, if possible, completely automate our pipelines.

Popular CI/CD Tools

There are numerous tools and frameworks that can be used to create automated CI/CD pipelines. The list below is not comprehensive and represents only a small selection of the many excellent tools that can be used to facilitate CI/CD pipelines. Generally, these tools can be divided into two categories: native tools and third-party tools.

NATIVE TOOLS

Native tools are CI/CD tools that are integrated directly into our repositories. For these tools, we create a configuration file that resides alongside our source code, and when we make a commit, the repository consumes our configuration file and executes the stages that we define.

The two most popular native tools available today are:

1. GitHub Actions – An automated workflow tools that integrates directly with GitHub repositories. New pipelines, called workflows in the GitHub Action lexicon, can be constructed by creating a new Yet Another Markup Language (YAML) workflow file in the `.github/workflows/` directory of a GitHub repository. More information about GitHub Actions can be found in the official GitHub Action Documentation and the Getting Started with GitHub Actions Refcard.
2. GitLab CI/CD – Similar to GitHub Actions, GitLab CI/CD is integrated directly with GitLab repositories and allows developers to create new workflows by creating a `.gitlab-ci.yml` file in the root of GitLab repository. More information about GitLab CI/CD can be found in the official GitLab CI/CD documentation.

When a native tool is available, it is best to use it because it affords the greatest level of integration with a repository and the source code managed by the repository. For example, if our code is stored in a GitHub or GitLab repository, we should use GitHub Actions and GitLab CI/CD, respectively, by default unless we have a pressing need to use a third-party tool.

THIRD-PARTY TOOLS

Third-party tools are CI/CD tools that reside outside of our repository. For many of these tools, we create a hook in our repository that notifies that third-party tool when a commit has been made. The tools then check out our code from our repository and execute the configured pipeline. The two most popular third-party tools available today are:

1. Jenkins – An open-source automation server that allows developers to automate the build, test, and deployment of their projects. Jenkins is commonly used as a standalone service, deployed by a development team. Pipelines are either configured directly through the Jenkins UI or through the creation of a `Jenkinsfile` within a source code repository. More information about Jenkins can be found in the official Jenkins Handbook.
2. CircleCI – A hosted automation service that integrates with GitHub, GitHub Enterprise, and Bitbucket. The advantage of CircleCI is that a team does not have to deploy and maintain a CircleCI instance but, instead, can access CircleCI through the circleci.com. What it gains in convenience, though, it loses in its narrow repository support and lack of flexibility. More information about CircleCI can be found in the official CircleCI documentation.

While using a native tool should be our default option, there are some instances when a third-party tool may be a better choice, such as when:

- A native tool does not provide the functionality we need
- A third-party tool allows us to utilize more computing power (i.e., a native tool may only allow us to use the resources of a single machine or the resources associated with our repository to execute our pipeline)
- A standalone option is needed so that we can manage the CI/CD pipeline directly (i.e., we wish to manage a CI/CD server within a firewall or company subnet)

Conclusion

Test automation and the introduction of CI/CD into software development has irrevocably changed the way that we create, test, and release our software. While the CI/CD space continues to grow and advance, it is essential that we learn the fundamentals of automated testing in CI/CD and select tools that best enable the time-savings and quality-improvements that CI/CD offers.

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

Automated Tests: You Are Doing It Wrong

Over-Engineering: You Are Doing It Right

By Daniel Stori, Software Engineer at AWS

Daniel Stori, Software Engineer at AWS

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

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

CONTRIBUTOR INSIGHTS

Why a Site Reliability Engineer Is Important to Your CI/CD Pipeline

By Alireza Chegini, Senior DevOps Engineer & Azure Specialist at S-RM

Continuous integration and continuous deployment are the two major components of DevOps principles. Every organization that wants to move away from the traditional way of working has to learn, design, and implement a mature CI/CD pipeline. Having a mature CI/CD pipeline is a good start for site reliability engineering, but alone, it's not enough. The site reliability engineering (SRE) methodology brings a new perspective to the software development life cycle by aiming to achieve reliability at scale.

Drawing on my own experience of being an SRE for more than five years, I will touch on some of the key benefits I've experienced and why it's important for SREs to be involved in the CI/CD pipeline.

SRE Engineer vs. DevOps Engineer Approach Toward CI/CD

Although DevOps and the SRE approach have many things in common, they are still two different approaches that were created for different purposes. SRE was created after DevOps, when it became apparent that the DevOps way of working could not tackle all issues and satisfy all requirements. That's why we can see these different approaches toward the CI/CD pipeline, where the most important activities of the SDLC happen. I had a chance to work as both a DevOps engineer and an SRE engineer, and here are some differences that I observed:

Subject	DevOps Approach	SRE Approach
CI/CD pipeline	Aims at establishing a CI/CD pipeline where either there are no pipelines at all, or the one in place has not been properly implemented	Aims at modifying an existing pipeline and identifying bottlenecks and problems that impact lead time
Automation in CI/CD	Tries to automate everything in the CI/CD pipeline	Takes it one step further and automates incident management and production issues
Incident management and CI/CD	Has to align with different parties' engineers to apply any changes	Has more freedom to decide and execute operations in order to mitigate issues quickly
Normal in CI/CD	Having a good, working CI/CD pipeline	There is no normal. Reliability is never taken for granted and, it is assumed that unexpected incidents will occur. This results in constant improvement of the CI/CD pipeline, reducing incidents, and increasing team maturity in mitigating issues on time.

Improvement From the Ground Up

KPIs are the core of decision-making for SRE engineers, and they become a performance dashboard that developers can see to view the quality of their work across various metrics. This informative approach makes development more conscious and aware of application/service performance earlier than releasing in production. Therefore, developers have a chance to identify issues and inefficiencies much earlier than a usual development life cycle without SRE.

SRE engineers start from the measurements and look at the existing CI/CD KPIs, if there are any. Otherwise, they define the KPIs themselves to indicate the current status of pipelines. Then they can create a roadmap with measurable milestones to improve the pipeline. This approach helps engineers consider various performance factors right away when they are busy with the functionality design process.

As SRE engineers create KPIs, they need to work closely with developers to understand the system logic, architecture, and components relations. This collaboration creates a team synergy where all engineers not only learn from one another, they master various skills in a team and can replace each other whenever it's needed.

Traditionally, application functionalities are the most important part of the design architecture. That's why, sometimes, aspects like high availability and reliability are not taken into account at the beginning. The SRE approach considers reliability, availability, and resiliency from the design stage. This results in huge savings from development and operations up front, since it is very costly to redesign projects if these issues surface later in production.

SRE Incident Management and CI/CD

When it comes to production incidents, it is crucial to detect issues and restore the system to its normal state. SRE practices came as an enhancement to DevOps practices. One interesting SRE approach is that engineers can deploy new patches during an incident without impacting the other parts of the running system. Downtime is inevitable when there is an incident or a new deployment is in progress. SRE engineers constantly try to reduce downtime, however, and they use new techniques called zero downtime deployment. SRE engineers can decide on the required change or fix and immediately trigger the pipeline to release the change from dev to production.

SRE engineers do not follow a bureaucratic approach in which a certain number of parties have to be involved in the production environment's decisions, and there is no hierarchy in place. The SRE approach takes risk, but it creates an autonomous team that can decide and act fast on incidents.

However, this doesn't mean that the other parties are never involved or informed properly. Here are some examples of practical situations where communication should happen:

- Suppose there is something to be done in a high priority which disrupts the production applications and services. In that case, a client should be informed before, during, and after the operation to ensure everything is under control related to live operations, data loss, and so on.
- Suppose there is a rollback operation and an old version of an application or a service should be installed. In that case, the development team should be informed and involved to ensure there is no problem with other services after this rollback.
- Suppose any process, deployment step, or even any line of code which developers write should be changed by SRE developers in an emergency. In that case, the development team should be informed afterward to make sure they are aware of these changes and the reasons they were made.

SRE Proactively Trains Team Members on CI/CD

When we talk about quality, we should be able to turn the quality into numbers. Then, we can quantify the quality with some metrics. I remember when we created our first-ever production dashboard out of application performance. Most people did not get what we were doing. As we rolled it out, however, it was visible how much memory and disk space were being used by every production server. After a couple of weeks, non-operational people started to notify us about the application quality. They simply looked at the dashboard and spotted some warnings.

Since it was easy to understand, they were able to get the problem quickly — and even took initiative to make sure we were aware of it. This is an example of how we managed to create some basic metrics and define a baseline to check the production performance. Before having an extensive monitoring dashboard, you can still get better control over your platforms with basic monitoring metrics. Here are some common metrics you can create if you are still new to this area:

- If you have IaaS, you can start with monitoring your infrastructure availability— resources like your CPU, memory, and hard disk. These areas are the most common troublemakers, so you can identify issues before they become a disaster.
- If you have some web services running, you can start with monitoring your service availability by checking the endpoint. Additionally, you can monitor the HTTP errors to understand what is happening with requests.

The SRE approach is strongly against creating a silo. SRE engineers work closely with developers, testers, and anyone who impacts the software project. This collaboration creates a strong knowledge-sharing loop in which most team members can pick different tasks and responsibilities. Moreover, this approach creates new SRE engineers from developers and testers interested in understanding application design, implementation, and operations.

Common Misconceptions About SRE

When any methodology is used incorrectly, it might not be as useful or effective as when it's properly implemented. The same goes for SRE, a new version of DevOps. When an organization that is not mature enough in DevOps considers implementing SRE, wrong perceptions can lead to much confusion. After years of doing DevOps and SRE activities, I learned that you need to have a good understanding of DevOps to become a good SRE. The reason is that DevOps is the predecessor of SRE, and to identify why we are doing things in an SRE way, you need to know the history behind that.

Another common misunderstanding happens when companies see the SRE as just another expert in handling incidents and operations. Let's refer to the Google definition of SRE. We learn that SRE is considered a team made up of different experts who can build, run, and maintain application services autonomously. SRE goes one step further than DevOps and takes all responsibilities. This way, you have full control of your SDLC and have one team that communicates, decides, and implements things very quickly. Having a good understanding of the context of SRE is key to making sure you can implement it properly.

Bottom Line

The SRE approach is the latest advancement of the DevOps way of working. It offers best practices to keep all services and applications running reliably. SRE works smoothly with CI/CD pipelines; you can constantly see where you are and what can be improved in your SDLC. This keeps you on track at all times, and it helps you avoid taking any success for granted. SRE engineers are the frontrunners on these efforts — they bring this mindset to an organization. SRE engineers define their KPIs based on customer requirements and what makes the platforms reliable. These requirements can change every day, so SRE engineers help teams adapt to these changes while the production reliability stays intact.

Alireza Chegini, Senior DevOps Engineer at S-RM

@allirreza on DZone | @alirezachegini on LinkedIn | codingascreating.com

Alireza is a software engineer with more than 20 years of experience in software development. He started his career as a software developer, and in recent years he transitioned into DevOps practices. Currently, he is helping companies and organizations move away from traditional development workflows and embrace a DevOps culture. Additionally, Alireza is coaching organizations as Azure Specialists in their migration journey to the public cloud.

CONTRIBUTOR INSIGHTS

Managed vs. Self-Hosted CI/CD

By Eric Goebelbecker, DevRel at HitSubscribe

Continuous integration/continuous deployment (CI/CD) pipelines have matured from new forms of automation to mission-critical systems. DevOps teams rely on pipelines to deliver value to their customers by tightening developer feedback loops and standardizing processes. When a system becomes more valuable and important, it tends to increase in complexity. It must support more users, be more reliable, and perform, despite the increased load. Soon the CI/CD system built for one team has grown to support every business line in the firm.

Should your CI/CD system be self-hosted or a managed service? You may be asking yourself this as you review an existing CI/CD system or prepare to build a new one. Which approach will work best for you?

Managing Your Systems vs. Outsourcing to SaaS

When you outsource an internal system to a managed service provider, you're giving up control over the systems the application runs on. Sometimes this is an advantage. Having fewer systems can mean reduced headaches, less capital outlay, and the potential for a smaller headcount. But it also means relying on someone else to do the work. Let's compare four key systems support areas:

Table 1

SELF-HOSTED VS. MANAGED SERVICES: ADVANTAGES AND DISADVANTAGES		
	Self-Hosted	**Managed Services**
Cross-team coordination	Supporting CI/CD systems requires cross-team coordination for applying updates, system upgrades, and repairs.	Coordination is limited to what's required to share pipelines and artifacts between teams.
Scalability	You're responsible for monitoring systems, planning upgrades, and capital investment.	Scaling up means adding users or moving to a high service tier. This still requires more spending.
Software updates and maintenance	You're responsible for tracking and applying software updates.	The managed provider is responsible for software updates.
Hardware break/ fix support	You're responsible for keeping systems running and fixing them when they break.	The managed provider is responsible for maintaining systems.

Security

All of your IT infrastructures must be secure, but CI/CD holds your application code and configs as well as information about the users that can deploy them. It needs the highest level of security you have to offer.

SECURITY ADVANTAGES AND RISKS OF MANAGED SERVICES

When you outsource an application, you're outsourcing the security with it. You're not responsible for doing the work, but you're still accountable for the outcome. Even if the cloud provider is authenticating users against your directory services, you're trusting them with enforcing access to some of your most precious data. Should you?

"Cloud" is another word for "someone else's computer," and "managed services" means "on the Internet." Your staff will have convenient access from anywhere, regardless of whether your offices are open or accessible. This is a tremendous convenience that also has disaster recovery benefits. But it increases your attack surface and puts your fate in the hands of an external company.

SECURITY ADVANTAGES AND RISKS OF SELF-HOSTED

If you keep your CI/CD system in-house, you're responsible for its security. You know your system's requirements and your user community, which may be an advantage. But maybe a managed provider has more security knowledge and experience on their staff than you do. One thing you can do is keep your CI/CD system off the Internet. You can lock it down so it's only accessible from behind a firewall, or even go as far as isolating to internal networks and your VDI infrastructure. But that's no guarantee of safety, and you'll be giving up the convenience that an Internet-accessible managed service offers.

Regardless of where you locate your CI/CD, you still need to worry about supply-chain attacks. Many managed CI/CD providers offer vulnerability scanning and penetration testing solutions. If you keep your pipelines in-house, you're taking on responsibilities for that, too.

Control

When someone else manages your CI/CD pipeline, you have less control. Is it a worthwhile tradeoff? How much are you willing to relinquish? Will ceding control to a managed service hamper how you use your pipelines? What benefits do you receive in return for ceding some control?

ADVANTAGES TO CONTROLLING YOUR RESOURCES

When you're in control, you're responsible for defining all policies regarding how your CI/CD systems are used, run, and administered. For example, if your development teams want custom plugins for your CI/CD platform, the decision is yours. A managed provider may only allow approved plugins or have an onerous approval process that holds up progress. You also control your destiny regarding where you put your CI/CD servers and source code.

As we covered in the security section, managed services are accessible via the Internet:

- The managed CI/CD system needs to access your source control repositories. For some providers, your code needs to be in a managed repository like GitHub, GitLab, or Stash. Is this compatible with your intellectual property policies? Keeping your CI/CD in-house means you can keep your code there, too.
- You may be able to retain control over your code by opening access to your private repos instead of moving to a managed solution, but this opens up new risks.
- Your users will need to manage a new set of credentials for the managed service, or you'll need to expose your directory services to the provider.

PUTTING YOUR DESTINY IN SOMEONE ELSE'S HANDS

What happens when someone else controls your CI/CD systems?

- The managed providers control their pricing.
- The vendor is responsible for protecting your data and maintaining redundant systems and up-to-date backups.
- Connections between cloud CI/CD providers and cloud source control providers are secure and easy to manage.
- Most cloud vendors integrate easily with public OAuth providers like Google and GitHub, so it's easy to integrate cloud services.
- Your requirements will change, and so will the vendor. How much effort will it take to move your pipeline back in-house or to another vendor if it becomes necessary?
- Similarly, what happens when you outgrow the vendor?
- Does the vendor support all the integrations you need? Will they keep up with new products?

Cost

We've alluded to costs and potential cost-savings several times so far. Let's look at how you should evaluate managed vs. self-hosted CI/CD costs. Managed CI/CD systems are priced per user and per minute for CI/CD operations. You probably have a handle on how many users you'll have, but how can you estimate minutes? What happens when a process spins out of control? Accurately estimating month-to-month costs is difficult at best. Large enterprises may have some leverage to keep costs under control by negotiating flat pricing based on a minimum spend. Smaller companies may not.

Self-hosted CI/CD means you're responsible for licensing the software required to run your systems. While the major CI/CD platforms are open source, the more popular and useful enterprise editions have licenses for which you will be required to pay. Then there's also the cost of buying and maintaining hardware or cloud systems. Hardware requires a capital investment, colocation space (including power), and maintenance. Cloud systems have a monthly fee, and while there's no hardware to maintain, they must be monitored, updated, and fixed from time to time.

Conclusion

As cloud computing grows more prevalent, managed solutions for core functions like CI/CD become more attractive. In many cases, moving to managed services allows development teams to focus on their application domain, get more done, and perhaps even save some money. But choosing between managed or self-hosted CI/CD is difficult because there are many moving parts. Which option is best depends on your specific situation.

The wrong call can waste a great deal of time, effort, and money. Before you decide which path to take, it's critical that you consider all of the tradeoffs and advantages of both approaches.

Eric Goebelbecker, DevRel at HitSubscribe

@egoebelbecker on DZone and Twitter | @ericgoebelbecker on LinkedIn | ericgoebelbecker.com

For nearly 30 years, Eric worked a developer and systems engineer at a variety of Wall Street market data and trading firms. Now he spends his time writing about technology and science fiction, training dogs, and cycling.

CONTRIBUTOR INSIGHTS

Securing Your CI/CD Pipeline

Common CI/CD Security Challenges and Advanced Strategies to Mitigate Threats

By Sudip Sengupta, Technical Writer at Javelynn

Software firms have long relied on a DevOps approach to enhance agility and collaboration in software delivery. CI/CD pipelines automate processes in the software development lifecycle (SDLC) to enable seamless integration and delivery of new features. While CI/CD pipelines enhance software development through automation and agility, they involve integrating numerous tools and services, which can introduce security gaps. Identifying and remediating these security gaps is key to ensuring secure CI/CD practices. This article presents a general overview of what you need to know as you secure your CI/CD pipeline.

Introduction to CI/CD Security

While CI/CD pipelines enhance the efficiency of software development and delivery through automation, the core stages of the pipeline don't include security by default. CI/CD security is the set of practices aimed at identifying and fixing vulnerabilities without significantly slowing down processes in the pipeline. CI/CD security practices mainly involve injecting penetration tests and active security audits to help reduce bottlenecks caused by late handoffs to security and QA teams. Secure CI/CD pipelines enable software teams to automate security for multiple deployment environments and layers of the SDLC, enforcing agility.

COMMON SECURITY THREATS FOR A CI/CD PIPELINE

Each organization's CI/CD pipeline has unique characteristics based on the business case, workload, and tech stack used. As a result, the implementation of CI/CD security differs based on use case. There are, however, security risks that are common to almost all pipelines, which warrant similar identification and remediation approaches. These risks include:

UNAUTHORIZED ACCESS TO CODE REGISTRIES

CI/CD operations rely on shared repositories to enable collaboration, configuration management, updates, and version control. All source code and configuration files reside on the Git repository as a single source of truth. Public repositories are popular in modern CI/CD pipelines since they reduce development costs and time. These repositories pose a security threat, however, as developers publish source code from their private machines into public, shared folders. Attackers can search through open-source registries as a reconnaissance technique and leverage the data gained for targeted phishing, reverse engineering, and remote code execution attacks.

INSECURE CODE

The requirements of rapid development and delivery in CI/CD pipelines have led to the increasing use of open-source, third-party integrations. Some teams may import third-party integrations into the deployment environment without properly scanning the source code for security gaps. These integrations can introduce vulnerabilities into the CI/CD pipeline. Developers may also fail to follow best practices for code security, which increases the attack surface. Common code vulnerabilities include format string vulnerabilities, buffer overflows, improper error handling, and canonicalization issues, among others.

IMPROPER SECRETS MANAGEMENT

Secrets aid in managing access to data and resources within the CI/CD pipeline. These include passwords, tokens, API keys, and other authentication credentials used to validate users accessing sensitive systems in the pipeline. Exposed secrets can, therefore, grant an attacker control over part or all of the CI/CD processes. Secret management misconfigurations include hard-coded secrets, storing secrets in public cloud environments, and manual secrets management, among others.

SHIFTING SECURITY LEFT

In older pipelines, security was often a final step, contributing to deployment bottlenecks. Today, best practices require integrating security controls earlier in the SDLC, otherwise known as "shift security." Shifting left involves implementing security checks in every layer of the CI/CD pipeline, enabling more accurate threat detection in every step. The goal is to remove friction between DevOps and security teams, enhancing efficiency in software development and ensuring robust security practices.

KEY CONSIDERATIONS FOR ADOPTING A CI/CD SECURITY TOOL

Some factors to consider when selecting a tool to secure the CI/CD pipeline include:

- Scanning coverage
- Ownership costs and licensing terms
- Maintenance and configuration effort required
- Scalability
- Integration with existing development and security stack

Administering Security on a CI/CD Pipeline

With the changing threat landscape, administering security is one of the most crucial aspects of a CI/CD pipeline. The first step in securing DevOps workflows starts with assessing how you can apply the principles of DevSecOps to your CI/CD pipelines. One of the goals of a diligent assessment is to identify tools and strategies to administer robust security.

BEST PRACTICES TO SECURE A CI/CD PIPELINE

To fully realize the benefits of integrating security directly into the software lifecycle, teams should:

AVOID HARDCODING SECRETS IN CONFIG FILES AND CI/CD BUILD TOOLS

Secrets are often required at various stages of the SDLC. An easy way to provide these secrets is to reference them as environment variables in configuration files and manifests. Anyone who can access these templates and files can extract credential information from these files, potentially leading to a data breach. Software teams should use tools that encrypt, store, and enable the central management of secrets to keep credential data away from malicious users. To securely manage and distribute secrets, administrators should perform encryption at rest before storing them in the ETCD server.

First, encode the secrets in Base64 format as shown below:

```
$ username=$(echo -n "admin" | base64)
$ password=$(echo -n "a62fjbd37942dcs" | base64)
```

Define the secrets in a YAML file:

```
echo "apiVersion: v1
> kind: Secret
> metadata:
>   name: test-secret
> type: Opaque
> data:
>   username: $username
>   password: $password" >> secret.yaml
```

Next, once secrets are created, you can use those to be applied into a Kubernetes pod. This can be done by creating a `.yaml` file, `secret-env.yaml`, whose environment variables are populated with data from the secret. The file's specs would be similar to the following:

```
apiVersion: v1
kind: Pod
```

Code continues on next page

```
metadata:
  name: secret-env-pod
spec:
  containers:
    - name: mycontainer
      image: alpine:latest
      command: ["sleep", "9999"]
      env:
        - name: SECRET_USERNAME
          valueFrom:
            secretKeyRef:
              name: test-secret
              key: username
        - name: SECRET_PASSWORD
          valueFrom:
            secretKeyRef:
              name: test-secret
              key: password
  restartPolicy: Never
```

When populating environment variables, Kubernetes decodes Base64 values. These environment variables can be used in all Kubernetes API objects, eliminating the need to hardcode secret data.

ENFORCE ACCESS CONTROLS FOR CI/CD BUILD TOOLS

DevOps teams should implement authentication and authorization mechanisms to control the entities that can access specific processes and tools within the CI/CD pipeline. The teams should enforce the principle of least privileges to ensure resource access is only granted to those who absolutely need it. Data within the CI/CD pipeline should also be secured using tokens, access keys, and passwords to prevent the addition of malicious payloads into the pipeline.

ESTABLISH AUTHENTICATION MECHANISMS FOR SOURCE CONTROL

Version control repositories (most often in Git) are a must-have for CI/CD pipelines. They foster collaboration and enable continuous deployment of features. Since the Git repository contains the application's source code, Infrastructure-as-Code manifests, and intellectual property, a vulnerability in source control grants attackers access into the application's design and implementation logic. Access to Git repositories should be secured using multi-factor authentication since they are a high-value target for hackers. Teams can also prevent accidental branches and commits using the `.gitignore file`, and educate developers on Git best practices.

ENSURE CONFIGURATION PARITY ACROSS ALL ENVIRONMENTS IN THE PIPELINE

DevOps teams should ensure that all environments (development, testing, production, and so on) are configured similarly. With configuration parity, QA teams can accurately detect security issues during testing as these issues exist on all environment configurations. This parity can be achieved using virtualization and abstraction technologies like containers and Infrastructure-as-Code declarations.

CONFIGURE ROLLBACK CAPABILITIES

It is common for security and QA teams to uncover security issues after application updates or deployments. This requires the administrators to roll back (revert) the deployment to an earlier version. The deployment should be configured to achieve a graceful rollback that eliminates the security issue until the development team has worked on it. Rollbacks are best achieved by retaining artifacts of an older version until the new deployment is approved for production.

IMPLEMENT CONTINUOUS VULNERABILITY SCANNING AND MONITORING

It is important to monitor and test every resource in the CI/CD pipeline. Use a vulnerability scanning solution to test application code, environment configurations, and deployment scripts against a database of known vulnerabilities to eliminate potential attack vectors. These scanning and monitoring tools should be deployed across the entire SDLC to uncover vulnerabilities as soon as they occur to prevent exploits.

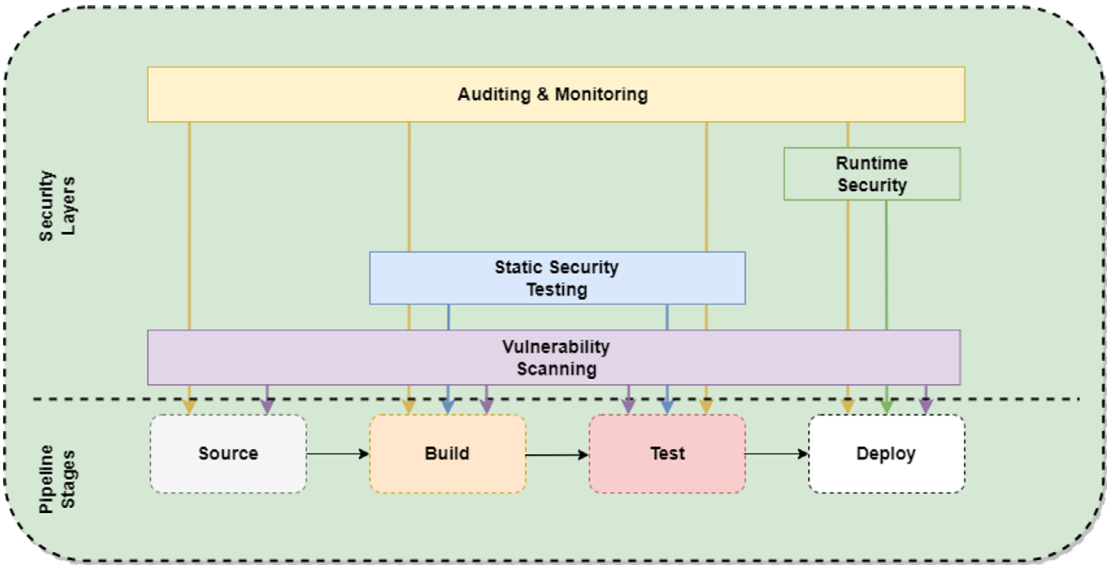
CLEAN UP REDUNDANT RESOURCES AND UTILITIES FREQUENTLY

CI/CD pipelines are typically built with immutable infrastructure, which run specific processes and then terminate. DevOps teams should ensure all temporary resources such as containers, services, and virtual machines are cleaned up after termination. Attackers could use their open ports to gain initial entry into the deployment environment, so it's important to properly manage these resources to reduce the security gap.

LAYERS OF CI/CD SECURITY

Administering CI/CD security requires a comprehensive, multi-layered approach to strengthen the defense across every point on the pipeline. These layers of security include:

Figure 1 : Security layers in a secure CI/CD pipeline



VULNERABILITY SCANNING

Vulnerability scanning involves using databases of known threats to identify and remediate security gaps in the entire CI/CD pipeline. Automated tests scan the application and deployment environment to identify and classify weaknesses in code, infrastructure, and third-party services.

STATIC SECURITY TESTING

These are software composition analysis techniques aimed at identifying potential vulnerabilities in code written by internal development teams. Security teams often use these tools to develop test cases to pinpoint insecure code vulnerabilities before deploying new application builds.

RUNTIME SECURITY

This layer relies on runtime application self-protection (RASP) tools to detect security threats for applications in live production environments. These tools scan configuration templates and continuously test the state of the deployment environment, then use a comparison to identify and respond to any runtime threats.

AUDITING AND MONITORING

Application and infrastructure logs continuously track and store application and deployment data. Auditing involves analyzing these logs to infer patterns that can be used to improve the application's security posture. Monitoring is the deployment of diagnostic tools that analyze metrics to gain an understanding of most system-related issues.

Constant auditing and monitoring help teams build context and predict baseline user behavior. Security teams can identify security threats by analyzing user actions that deviate from the established baseline.

Conclusion

As code that runs in a CI/CD pipeline can be executed by anyone with access to the pipeline's source code repository or container registry, DevOps workflows are known to introduce inherent security challenges. A recent survey projected that roughly 55 percent of organizations delay application rollouts due to security concerns. While a DevOps framework enables enhanced collaboration and automation, organizations must adopt a continuous security model that considers advanced strategies and tools to ensure comprehensive security across all layers of a CI/CD pipeline.

Sudip Sengupta, Technical Writer at Javelynn

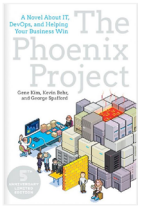
@ssengupta3 on DZone | @ssengupta3 on LinkedIn | www.javelynn.com

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.

ADDITIONAL RESOURCES

Diving Deeper Into DevOps and CI/CD

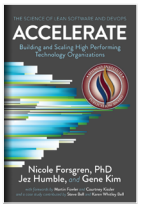
BOOKS



The Phoenix Project: A Novel About IT, DevOps, and Helping Your Business Win

By Gene Kim, Kevin Behr, and George Spafford

In this best-selling novel about DevOps, follow fictional character Bill Palmer on his journey from Director of IT operations to sudden Vice President of Operations. Learn lessons on tearing down silos between operations and development teams — and see why security should always be put first. Not only is this book a great tool for DevOps teams, but it can benefit anyone who wants their business to be successful.



Accelerate: The Science of Lean Software and DevOps – Building and Scaling High [...]

By Nicole Forsgren, Jez Humble, and Gene Kim

Does the performance of delivery teams actually drive business value? This book presents research and statistical methods that can help management at any level learn how to measure software delivery performance and explores how it can provide a competitive advantage to your company.

REFCARDS

Getting Started With GitHub Actions

Integrating CI and CD concepts into your repository enables you to further reap their benefits, and the most prominent option available is GitHub. In this Refcard, readers will learn the key concepts of GitHub Actions, as well as how to create automated workflows (or CI/CD pipelines) using text-based configurations that are stored directly within their GitHub repository.

Introduction to DevSecOps

DevSecOps enables you to reach higher security standards while observing DevOps principles. In this Refcard, you will explore how to get started with DevSecOps, covering key themes, pitfalls to avoid, crucial steps to begin your journey, and guidance for choosing security tools and technologies to build your DevSecOps pipeline.

TREND REPORTS

CI/CD: Automation for Reliable Software Delivery

In 2020, DevOps became more crucial than ever as many companies moved to distributed work and continued to accelerate their push toward cloud-native and hybrid infrastructures. This Trend Report examines how these changes have impacted development teams across the globe and dives deeper into the latest DevOps practices that are advancing the industry. You'll find observations from our original research and insights from DZone contributors on topics including IaC, AI and ML in DevOps, CI/CD security challenges, and more!

PODCASTS

Ship It! DevOps, Infra, Cloud Native

This podcast's goal is to get your ideas out into the world. Gerhard Lazu and friends explore topics that cover all things code, ops, and infrastructure.

The Pipeline: All Things CD & DevOps

The Pipeline, created and hosted by Jacqueline Salinas, covers a range of topics that are centered around CD and DevOps. Hear from industry experts, innovators, and thought leaders in order to gain more knowledge surrounding the CD and DevOps ecosystem.

The Humans of DevOps

What helps organizations achieve their goals, adapt and respond to challenges, and compete in a disruptive, digital landscape? People! This podcast showcases the human element of DevOps and includes episodes on building a healthy team culture, overcoming disconnected silos in remote teams, and much more!

DevOps and Docker Talk

Host Bret Fisher offers a catch-all podcast that covers Q&As from live shows, guest interviews, and chats with industry friends — all centered around cloud-native and DevOps topics like container tools, cloud management, and sysadmin.