32 Principles and Practices of Successful Continuous Integration, Continuous Delivery, & DevOps

De·vel·op·ment (*dĭ-vĕl′əp-mənt*) is the application of systematic, disciplined, and quantifiable principles to the development, operation, and maintenance of new software-based products and services. It entailed project *planning*, requirements *synthesis*, architecture and design, coding and unit testing, integration, system, and acceptance testing, and operations and maintenance. Much of this involved *synthesizing* market, customer, and end-user needs ad nauseum, detailed architectures and designs, methodical coding practices, exhaustive test planning and execution, and dozens of software lifecycle documents. The term "*synthesis*" is applied because managers and analysts often derived, predicted, or manufactured bloated requirements leading to over scoping, uncertainty, risk, cost, duration, and poor quality. Traditional project failure rates were extremely high and the few that made it into production were either abandoned due to technological obsolescence or were extremely expensive to operate and maintain surviving on life support. Since operations and maintenance was 80% of total software life cycle costs spanning decades, expensive libraries were created to capture tacit developer knowledge and assumptions in explicit software documents, exceeding the cost of the computer programming itself. Thousands of hours were required to test bloated requirements leading to complex, open-looped systems with infinite states, latent defects, and unpredictable failures. Each development phase or activity was functionally organized and performed by a separate firm, business unit, department or team, spread across multiple states, countries, and time zones leading to unnecessarily slow handoffs. Manual lean and agile methods emerged to reduce the scope, complexity, risk, cost, duration, handoffs, and failure using medium-sized batches or product backlogs and vertical feature teams.

Op·er·a·tions (*ŏp′ə-rā′shənz*) is the philosophy, culture, people, process, tools, and discipline of managing an IT infrastructure that hosts new software-based products and services. At its most basic level, it involves IT platform, infrastructure, application, and incidence or help desk management. A granular view includes planning, architecture, design, contracting, facility, HVAC, data center, server, storage, network, equipment, cabling, end user device, telecommunication, deployment, provisioning, access, monitoring, auditing, performance, reliability, security, backup, recovery, inventory, asset, safety, and configuration management, etc. As trillions of dollars were spent on software projects over the last 70 years, organizations accumulated portfolios of software-based products and services requiring IT operations and maintenance services. Therefore, multi-million-dollar IT operations firms, business units, divisions, departments, and teams emerged. Functional IT departments were separate from functional software development organizations producing new software-based products and services. Software was developed by organizational functions like R&D, marketing, engineering, manufacturing, hardware, software departments, and IT operations itself. Although few large and complex software projects were ever completed, those that did were often handed over to IT operations for deployment, support, and maintenance. To ensure successful handoff, long linear waterfall lifecycles were applied, numerous documents were produced, and exhaustive testing was performed, which only served to reduce the number of finished systems handed off to IT operations in a brutal Darwinistic manner. Although IT operations was often undisciplined, software unreliable, and the IT infrastructure was unreliable too, the pre-World Wide Web era had relatively few end users and little risk or exposure.

De·vel·op·ment • Op·er·a·tions (*dĭ-vĕl′əp-mənt • ŏp′ə-rā′shənz*) is the synergy of IT development and operations people, processes, and tools to quickly deliver high-quality software-based products or services to the market, customer, or end-user. Traditional development and operations involve functional departments, roles, specialists, processes, tools, cultures, and standards. These include project managers, architects, designers, programmers, testers, security, sysadmins, operators, maintainers, and data center designers. These functions are performed by a different firm, vendor, business unit, department, project, or team. They are split across international and national boundaries, making difficult to move data across these functions. DevOps increases speed, quality, customer satisfaction, and value by streamlining handoffs. DevOps involves the use of vertical cross functional (collocated) feature teams. If a feature team is to develop a new product or service in a few days or weeks, then it needs one set of standards, guidelines, processes, tools, and data. They should automate manual labor, consolidate development and operations platforms, and standardize tools for version control, continuous integration, testing, delivery, deployment, and operations. As developers commit code check-ins, they are built, tested, staged, deployed, and operated in seconds or minutes. DevOps eliminates 80% of traditional software development costs and most of the IT operations costs using Cloud services. Amazon and Google perform hundreds of millions of tests, tens of thousands of deployments per day, and split-second rollbacks 24x7. The synergy of lean and agile thinking, evolutionary architecture, and DevOps is a powerful combination decimating lead and cycle times, enabling rapid business and market experimentation, optimal user experience, and maximum business value.

Attempts to create a software development process, language, and automation started with the invention of the ENIAC computer (1945), FORTRAN programming language (1954), General Motors IBM 701 operating system (1955), SAGE air defense system (1959), IBM operating systems (1960), and custom software industry (1965). The term "software crisis" was coined in 1968 along with its antidotes, "software engineering" and "software reuse!" The commercial shrink-wrapped software industry emerged (1969) and Hitachi created the first Software Factory (1969) followed by dozens of Japanese and American software factories (1975). Key elements were achieving economies of scale by collocating thousands of collaborating developers in one building to share knowledge, defining rigorous software development processes, automating every conceivable software activity, rigorous automated testing, and, of course software reuse. While some gains were made, processes were simply too long, rigorous, WIP or waste intensive, too much documentation was produced to capture tacit knowledge, and, of course, computers were just too darn slow. NTT made some important advances automating millions of (regression) tests for producing ultra-reliable multi-million line of code telecommunication switching systems (was this the first DevOps process?). With the advent of engineering workstations (1980), software factories were reincarnated in the form of computer aided software engineering (CASE) tools, and government agencies spent the modern equivalent of billions of dollars creating them. However, nothing changed much, as CASE focused on requirements synthesis, architecture, and design. The 1990s gave rise to rapid application development and visual programming, and agile methods emerged focusing on working software, unit testing, and continuous integration (early DevOps).

So, what exactly is the point of this trip down memory lane? Well, the more things changed, the more they stayed the same! Amidst the flurry of software development automation activity starting in 1945, an explosion of bone-crushing software lifecycles and maturity models also emerged from 1960-2000. International standards also emerged in the 1990s to document every conceivable business process and document related to the development of new products and services. Somehow, good software development was translated into creating processes and documents for every conceivable business activity, automating it, and increasing size, cost, complexity, uncertainty, risk, duration, queue size, lead time, cycle time, and failure. In 1967, Melvin Conway submitted a paper to the Harvard Business Review which was rejected entitled, “How Do Committees Invent?” which states, “Any organization that designs a system will produce a design whose structure is a copy of the organization's communication structure!” In plain English, enterprises create software development processes, activities, documents, tools, and software to automate every element of their bureaucracy without any effect except spending money. Therefore, software development standards and maturity models were created to promulgate Conway’s Law to the Nth degree! This was precisely why lean and agile methods emerged in the 1990s, to rapidly create small scoped working software to address market, customer, and end-user needs instead of automating bureaucracies. Although this was a large step forward, organizational bureaucracies still have yet to be streamlined, combined, automated, and focus placed on rapidly exploring tacit market, customer, and end-user needs vs. documenting errant assumptions at great cost, consequence, and ill-effect. Hence, DevOps was born, so let’s examine some its key principles.

DevOps Principles

1. **Lean • Think·ing** (*lēn • thĭng'kĭng*) Thin, slim, slender, trim, light, fast, flexible, agile; To apply just-in-time waste-free product and service delivery principles

   ✓ Value stream mapping
   ✓ Fast lead and cycle time
   ✓ Continuous improvement

   A major, underlying or foundational principle of DevOps is Lean Thinking. That is, delivery of software-based new products or services to end-users, customer, and the market with the shortest lead and cycle times possible. James Womack suggests focusing on value, value streams, flow, pull, and perfection. LeanKit suggests focusing on optimizing the whole, creating knowledge, eliminating waste, building quality in, delivering fast, deferring commitment, and respecting people. While, Don Reinertsen suggests taking an economic view, managing queues, exploiting variability, reducing batch size, applying work in process (WIP) constraints, controlling flow, getting fast feedback, and decentralizing control. What these lean thinking principles boil down to is the antithesis of traditional thinking, such as multi-decade long linear waterfall or staged lifecycles, voluminous process, document, and governance WIP (or waste), multi-million-dollar enterprise or system architectures, mountains of business requirements, and fixed-scope contracts, programs, and projects. Traditional thinking is predicated upon predicting highly uncertain market and product needs years in advance, spending years on over scoped projects with high-failure rates, and simply melting, fading, or wasting away without any success. Lean-agile thinking is predicated responding to market, customer, or end-user needs, fulfilling smaller and shorter-term batches, and delivering these with the shortest possible lead and cycle times and least amount of WIP (project inventory). When you apply lean thinking principles of short lead and cycle times, development WIP cost is lower and so is the market price, development uncertainty and risk are lower, delivery success is higher, markets, customers, and end-users get what they want and when they need it and they are happier if not delighted, and business, market, and relationship capital grow exponentially for both buyers and suppliers.

2. **Lean • Start·up • Think·ing** (*lēn • stärt'ŭp' • thĭng'kĭng*) Test, trial, explore, investigate, demonstrate, measure, validate; To apply hypothesis-driven experimentation to new product and service delivery

   ✓ Form hypothesis
   ✓ Small experiments
   ✓ Measure and repeat

   Lean Startup Thinking is a variation of lean thinking based on principles of market, customer, and end-user experimentation. The reason traditional thinking principles have such a high historical failure rate is that they predict (make up, synthesize, or fabricate) a large, costly, and lengthy program or project scope, requirements, architecture, and implementation leading to unmanageable uncertainty and risk. Researchers in the field of innovation and new product development quickly came to the conclusion that market, customer, and end-user requirements or needs exist in the form of tacit (unspoken) vs. explicit (documented) knowledge. Therefore, lean thinking, lean startup thinking, and principles of experimentation are predicated upon the notion that tacit (unspoken) market, customer, and end-user needs must be teased out gradually, incrementally, or in an evolutionary or even emergent style. In other words, markets, customers, and end-users don’t know what they want until they see it! Therefore, Lean startup thinking involves creating one or more extremely small and narrowly scoped hypotheses, designs, market probes, or models of a new product or services, getting market, customer, or end-user feedback as quickly as possible, and rinsing in and repeating this cycle in short two-week to 90 day windows until tacit knowledge can be codified as explicit knowledge, requirements emerge, and uncertainty is minimized. From there, lean thinking principles can continue to be applied to evolve the new product or service incrementally. With software-based products and services and cost-effective automated DevOps pipelines, many experimental cycles take place very quickly. Dot coms perform hundreds and thousands of micro-experiments per day, automatically collect quantitative and qualitative feedback on these experiments and refine their minimum viable products (MVPs) as quickly as possible to rapidly optimize business value for buyers and suppliers.

3. **Pull • Driv·en • One·piece • Work·flow** (*p͡ool • drĭv'ən • wŭn'pēs • wûrk'flō'*) Buy, order, select, request, demand, purchase; To apply customer or market made-to-order-based product and service delivery principles

- ✓ Pull-based system
- ✓ One-piece workflow
- ✓ Work in process limits

Pull-Driven One-Piece Workflow is also a subset of lean thinking that deserves special attention for a number of reasons. One, it stands in opposition to traditional thinking that pushes an inordinately large, complex, costly, and lengthy pre-fabricated, synthesized, or made-up product or service scope onto program and project teams to achieve full utilization and, of course, overallocation. Two, it places the onus upon the market, customer, or end-user to request a new product or service from a supplier's value stream, thus minimizing uncertainty and the practice, custom, or principle of making an over scoped request. And, three, it places a premium on WIP constraints, minimizing or eliminating full utilization and overallocation, decreasing lead and cycle times, and increasing the probability of successful delivery. Traditional thinking practitioners have arrived at the conclusion, assumption, and belief that full utilization increases productivity, reduces lead and cycle times, and optimizes operating cost. However, mathematically speaking full-utilization and overallocation has quite the opposite effect, it slows productivity, increases lead and cycle times to infinity, increases uncertainty, risk, and cost, and decreases the probability of successful product and service delivery. If markets, customers, and end-users must wait until infinity to receive a new product and service, they will immediately become dissatisfied, disillusioned, unhappy, angry, stressed out, and simply switch to a new supplier. Therefore, the key is to reduce batch size (scope), limit WIP (utilization), build-in extra operating capacity (slack), allow delivery teams to work on one tightly-scope market, customer, or end-user need, request, requirement, or work item type at a time. Doing so, actually decimates lead and cycle time, increases quality and delivery speed, reduces operating cost and market price, and increases morale, and ultimately the satisfaction, happiness, and delight of end-users.

4. **Col·lab·o·ra·tive • Teams** (*kə-lăb'ə-rə-tĭv • tēmz*) Cooperate, synergize, join forces, teamwork, brainstorm; To co-design, co-create, or compose new products and services together

- ✓ Small team size
- ✓ Self-selected groups
- ✓ Informal and temporary

Small, self-selected Collaborative Teams is also a key principle to successful DevOps. For software-intensive products and services, especially fixing bugs, making enhancements, or creating microservices, team size should be as few as two people, as many as three to four, and certainly no more than five to seven people. Furthermore, these teams should be self-selected, self-organizing, and work well together. Not only should the teams be as small as possible, but they should be allowed to choose their own work from the product backlog to the greatest extent possible. This doesn't mean they have the freedom to choose from the bottom or cherry pick the backlog, but that individuals should examine the backlog, focus on the highest priority user stories, and self-organize to begin solutioning one of these items. Work should never be assigned to people by directors, managers, coordinators, nor supervisors. To do so, would be to defeat the purpose of self-selection, self-organization, and free-will. It may be necessary to rotate teams once in a while to ensure that free riders or unproductive team members do not emerge. Furthermore, rotating teams helps build T-skills, cross train, encourage knowledge sharing and on-the-job learning, and distribute the skills and risk of new product and service development as much as possible. It's okay to have specialists on your team, including planners, analysts, architects, designers, coders, testers, network engineers, operating system and middle ware specialists, design thinking and user experience experts, security engineers, documenters, and other functional analysts such as DevOps pipeline designers. However, in so much as possible, one possible operating goal and objective would be to build full stack or full lifecycle engineers, and ultimately feature teams. While it may not be possible for everyone to have every skill, a noble goal would be for each person to have at least three functional specialties.

5. **E·mer·gent • De·sign** (*ĭ-mûr'jənt • dĭ-zīn'*) Rise, grow, appear, spring, surface, blossom, originate; To extemporaneously improvise or contrive the structure of a new product or service

- ✓ Just-in-time
- ✓ Situational solution
- ✓ Extemporaneous structure

Emergent Design is also a key principle to successful DevOps. As teams self-select high priority user stories and self-organize to complete them, they should have the freedom to improvise and implement the smallest possible situational design necessary to address the market, customer, or end-user need. Ultimately, designs should be extremely simple, small, minimalistic, organizationally independent, just-enough, just-in-time, waste free, object-oriented, fast, and high quality. Acceptance test criteria should be established, and just enough design and code created to satisfy these criteria in the shortest possible time. This is not to say that domain experts, user experience specialists, architecture and design specialists, security, performance, and quality engineers, and other programming experts, patterns, styles, and reusable objects should not be consulted, applied, and utilized. However, the ultimate goal is the smallest possible waste free design to satisfy the test criteria for that user story. Implementation speed, code quality, and system performance are paramount, along with minimizing the attack surface. The larger and more complex a design becomes, the larger the attack surface, possibility of dead, unreached, or untested code, and the greater the possibility of failure or compromise in the field. Designs should evolve, aggregate, and amalgamate over time, however, they should not become unwieldy, since the final form, fit, and function is a quantum or atomic level microservice. A microservice is not a 100 million line of code operating system that should take 1,000 people, a billion dollars, and decade to develop. It is conceivable that a microservice may have a complex operating system or environment encapsulated within its container, however, a microservice only links in the modules it needs vs. the entire operating system, however, the container walls protect it from leakage, spillage, and contamination of the larger ecosystem.

6. **Ev·o·lu·tio·nar·y • Ar·chi·tec·ture** (*ĕv'ə-lo͞o'shə-nĕr'ē • är'kĭ-tĕk'chər*) Xyz; To gradually conceive, fashion, or invent an innovatively new product or service

   ✓ Incremental change
   ✓ Trial and error based
   ✓ Elastic scope and size

   Evolutionary Architecture is also a key element of DevOps. Its cousin, emergent design focuses on a single user story at a time, that is, a software unit, module, fix, enhancement, or small collection of highly-interdependent modules. Evolutionary architecture focuses on the greater software-based product or service sub-structure, skeletal framework, or foundation. Evolutionary architecture is also based on principles such as extremely simple, small, minimalistic, organizationally independent, just-enough, just-in-time, waste free, object-oriented, fast, and high quality. However, rather than at the user story, software unit, or module-level, evolutionary architecture applies these lean principles to the architectural and design foundation for the entire product or service itself. Like emergent design, evolutionary architecture does not exclude consulting, applying, or utilizing domain experts, user experience specialists, architecture and design specialists, security, performance, and quality engineers, and other programming experts, patterns, styles, and reusable objects. However, like emergent design, evolutionary architecture only considers lean, minimalistic, and just-enough architectural foundation for a small microservice, web app, or market, customer, and end-user facing software-based solution. The capital costs of fixing, enhancing, refining, refactoring, merging, or simply replacing a software-based evolutionary architecture are minimal. This is a hard pill to swallow for traditional thinking architects who are bent on creating a fictitious multi-billion-dollar, decade long big design up front (BDUF). Traditional architects believe that a multi-million-dollar architecture heads off downstream costs, however, the evolutionary architect believes in getting the smallest possible solution to market quickly, obtaining measurable feedback, and rinsing and repeating (refactoring) the malleable software architecture at the lowest possible cost now to tease out value.

7. **Do·main • Driv·en • De·sign** (*dō-mān' • drĭv'ən • dĭ-zīn'*) Area, field, arena, realm, sphere, sector, territory; To create new products or services for a highly-specialized problem space

   ✓ Highly specialized
   ✓ Problem set oriented
   ✓ Limited generalizations

   Domain Driven Design is an approach to new products or services to segment, divide, and allocate monolithic enterprise architectures to smaller, specialized market, customer, end-user, enterprise, or business focus areas, concerns, and functional areas. Traditional enterprise and system architectures attempt to address the entire problem space at one time, such as a global ecosystem, system of system, market, enterprise, government agency, business, firm, or business unit. This is an attempt to achieve economies of scale and engineer individual components to be reused across the entire architectural landscape forever. However, scope, scale, size, complexity, uncertainty, risk, cost, and duration of monolithic architectures costs millions and billions of dollars in the public sector. That's just the architecture phase, which doesn't include the design, implementation, evaluation, deployment, operation, and maintenance costs. Domain driven design seeks to minimize the risk of monolithic enterprise architectures by segmenting them into smaller areas of concern, business subsystems, or functions. Once the monolith is segmented, then a highly cohesive domain architecture can be created, upon which domain-specific designs, code, tests, DevOps pipelines, and operational platforms are created. Design patterns, reusable objects, tests, tooling, and other highly specialized assumptions, operations, and outcomes can be made a little more easily with slightly lower risk. While domain driven designs were an evolutionary leap forward over monolithic enterprise architectures costing hundreds of millions or billions of dollars over decades by hundreds of architects, domain independent microservices are the latest evolution of this attempt to disaggregate and disintermediate monolithic enterprise scale BDUFs in to smaller, lower cost and risk, and quickly implementable market, customer, and end user software based products and services with DevOps.

8. **Se·cu·ri·ty • by • De·sign** (*sĭ-kyo͝or'ĭ-tē • bī • dĭ-zīn'*) Safe, protect, shield, defend, guard, shelter, preserve; To architect products and services with minimal vulnerabilities and maximum protection

   ✓ Small attack surface
   ✓ Multi-layer authentication
   ✓ Limited single point of failures

   Security by Design is yet another major principle of DevOps. Many outcomes of DevOps pipelines are market, customer, and end-user microservices deployed to billions of global hosts, platforms, and devices hundreds of times per day. Microservices contain or operate on highly sensitive security and privacy data. These include access control information such as passwords, usernames, and authentication data, as well as personally identifiable information such as names, addresses, phone numbers, email addresses, social security numbers, birthdates, credit card numbers, healthcare data, and other unique data. Therefore, the security architectures and designs of software-based products and services is now a major concern more than ever, is front and center among software developers, and is clearly in the spotlight. It's important to have security engineers on software teams or available for consultation; identify, develop, or derive appropriate security and privacy requirements; and select or develop architecture and design patterns to maximize security and privacy. Vulnerability tests must also be created and run at all levels of the architecture and design as well as DevOps pipeline (toolchain). The implementation technologies and hosting platforms must also exhibit appropriate levels of privacy and security. Attack surfaces must be modeled, design, and minimized, they must be evaluated and certified, code must adhere to them, and tests must be constantly run to probe the attack surface and security perimeter. There should be no single point of failure, multiple authentication and encryption applied, and data loss and exposure minimized and segmented. That is, the loss of one code segment or microservice must

not result in the loss of the entire data set. Containerization, automated static and dynamic analysis and testing, automated security monitoring, and automatic mitigation of security breeches should be designed and implemented using DevOps.

9. **Con·tai·ne·rized • Mi·cro·ser·vic·es** (*kən-tā′nə-rīzd′ • mī′krō-sûr′vĭs′əs*) Tiny, little, minute, compact, miniature, miniscule; To create small, portable and fully-encapsulated software-based products and services

   ✓ Small bare-metal solutions
   ✓ Fully encapsulated applications
   ✓ Self-contained operating environment

   Containerized Microservices is a major innovation for optimizing DevOps. Microservices are an architectural pattern for composing software-based new products and services as an ecosystem of small, loosely coupled, and independent services. Microservice families or ecosystems tend to be highly cohesive within a market, customer, end-user or business area, and dovetail very nicely off of Domain Driven Design principles extremely well. Each microservice may be developed by an independent team very quickly, independently tested and validated, and quickly deployed through a fully automated DevOps pipeline to the production host, platform, or device. They are often fully encapsulated in containers, replete with a self-contained operating system microkernel, middleware stack, and application programming interfaces (APIs) for interacting with other microservices within the domain. Containers create boundaries that are impenetrable to external attack and internal spillage as well. If a microservice fails or works itself into a failure state, it can be shut down internally or remotely, and replaced or restarted with a clean state microservice without affecting the overall operation and resiliency of the application ecosystem to which it belongs. The container is basically a fully self-contained virtual computer, but only contains the necessary operating system, middleware, and application components or slices necessary to perform its individual functions, thus minimizing its size and footprint, optimizing its performance, and minimizing its known and unknown common vulnerabilities and exposures (CVEs). Because each container encapsulates the entire micro-feature slice of end-user application functionality, containerized microservices run on bare-metal hosts, platforms, and end-user devices, thus minimizing the cost, risk, size, footprint, uncertainty, and vulnerability of operating a heavyweight production system.

10. **Fea·ture • Flags** (*fē′chər • flăgz*) Switch, control, button, lever, toggle, cutoff; To build switches into software-based products and services for enabling deployment of new features

    ✓ Built-in switches and toggles
    ✓ Multiple feature release audiences
    ✓ Early user and market testing of features

    The use of Feature Flags is also another major innovation for optimizing DevOps. Software-based new products and services are now built with a series of capabilities, features, and functions that range in maturity from the very old and stable, moderately new, and extremely innovative, but not quite ready for primetime mass market release. Furthermore, developers may want to deploy highly experimental MVPs as part of a production release. Using a series of switches, developers turn off old, recent, new, or experimental features and functions at will for market, customer, and end-user evaluation, feedback, and hypothesis testing. Feature flags go by many names, such as toggles, switches, gates, flippers, explosive bolts, kill switches, etc. Once again, feature flags may used for adding a new feature and to software applications, fixing or enhancing defective or underperforming application features, disabling, hiding, or discontinuing a feature, or opening or closing APIs. Opening and closing feature flags support albatross launches, beta testing, canary launches, champagne brunches, ring deployment, test in production, A/B testing, and even access control, among many other uses. They may be used to allow lead users to evaluate innovative features, expose feature variations to multiple groups to vote on them and then expose the best ones to everyone, or progressive rollouts to ensure smooth scaling, operation, and use. They may also be use for performance and load testing, or only turning on these features for developers and testers to evaluate code before general release at all. Basically, it allows developers to maintain a single, simple, and non-complex code base with branching and merging, and release a complex aggregate of features to the general public without multiple deployments. It can also be used for access control to grant privileged users rights to run specific application features and even supports security and privacy rules.

11. **Blue • Green • De·ploy·ments** (*bl$\overline{oo}$ • grēn • dĭ-ploi′mənts*) Copy, spare, backup, standby, alternate, mirror image; To deploy a new product or service by pointing a router to an alternate computer host

    ✓ Multiple deployment hosts
    ✓ Release to hidden backup host
    ✓ Switch to new platform when ready

    The use of Blue/Green Deployment is also a major innovation for optimizing DevOps, especially among commercial, mass market off-premises cloud providers, hosts, and platforms. Basically, dot coms such as Amazon, Google, Microsoft, IBM, and others support instant virtual duplication of operational target clouds, platforms, and servers for hosting new products and services. Firms simply switch market, customer, or end-user traffic between virtual hot-spares in fractions of second at will by pointing network routers and switches to duplicate virtual hosts. This is a way of quickly deploying new software applications to millions and sometimes billions of users in fractions of a second with little or no risk, network bandwidth, or performance bottleneck. That is, the cloud provider builds up, tests, and certifies one virtual host and makes its applications available to the mass market. Then, firms may manually or automatically spin up virtual alternative hosts with new components such as network services, operating systems, middle ware, applications, and new features. When tested and certified, cloud providers simply switch network traffic over to the new host. If there are any issues with the new virtual platform, end users may simply be switched back to the prior operational virtual platform. In doing so, cloud providers can perform a series of roll forwards and roll backwards at will. They can even have failover or overload hosts. If hosts in one region of the country experience a

widespread failure, degradation of service, or other operational problem, international traffic can be switched over to a new operational host at will. Edge computing may even be locally hosted in globally distributed regions to reduce network latencies, end-user processing speeds, and overall load balancing, by forward deploying duplicate cloud hosts and platforms. Having multiple virtual hosts allows DevOps pipelines to build up new platforms without degrading operational performance.

12. **Prod·uct • O·ri·en·ta·tion** (*prŏd′əkt • ôr′ē-ĕn-tā′shən*) Wares, goods, yield, output, release, merchandise; To develop new products and services for the commercial mass market

- ✓ Long lived value streams
- ✓ Small just-in-time experiments
- ✓ Market facing products or services

Product Orientation is a great application of modern DevOps principles. In traditional thinking, the goal is to scope large multi-year, multi-million-dollar projects with unmeasurable cost, risk, and uncertainty. Furthermore, traditional project managers have us apply linear waterfall phases to architect, design, and implement the code it all at one time with full utilization and overallocation working overtime until it is done. Finally, traditional project managers then ask everyone to stay up 24 hours a day every 6 to 18 months to test and fix the code all day and night to ensure final solution is ready for general release. To add insult to injury, they'd have modern DevOps engineers automate testing to support this antiquated paradigm. After several years, if everyone hasn't quit or died, customers didn't already buy another product, or our technology stack isn't obsolete, we have about an 80% to 90% chance of releasing something our customers don't want, need, or like. This paradigm is called "project orientation," feature factories, and focusing on outputs and optimizing economies of scale, cost, and efficiency, like Henry Ford's Model T factories in the 1920s! With "product orientation," we assume we're not exactly sure what our customers want, we'll form a hypothesis, create a small MVP, solution and test it quickly, and deploy it to some lead users, and rinse and repeat until we've converged on an optimal solution. Furthermore, we'll apply lean principles, make it pull-based, limit WIP and batch sizes, and use principles of one-piece workflow and self-selected teams. Our lead and cycle times will be minutes, hours, and days, vs. years and decades, and the focus will be measurable customer outcomes vs. project outputs like documents, process assessments, performance measurements, and features. We may even create product roadmaps and journey maps instead of project plans and schedules to deliver profitable labor-saving products and services for customers.

13. **De·sign • Think·ing** (*dĭ-zīn′ • thĭng′kĭng*) Form, style, shape, model, figure, sketch, pattern; To create new products or services from the viewpoint, needs, and challenges of end users or customers

- ✓ Problem space analysis
- ✓ User centered perspectives
- ✓ Human factor based solutions

Design Thinking is a principle ideal for applying DevOps practices. Design thinking is a problem-solving approach for creating complex, innovatively new products and services from a holistic end-user perspective vs. a business or technical perspective. Design thinking is a five-step process consisting of empathizing with users, defining user needs, ideating innovative solutions, prototyping those solutions, testing assumptions, and rinsing and repeating until a satisfactory end-user solution is devised. Real lead users are consulted to identify their pain points and high-level journey maps or roadmaps are constructed to visualize their daily life cycle of interacting with your business, products, or services. Then innovative solutions are sought to maximize ease, usability, speed, and convenience, while minimizing pain, inconvenience, wasted time, frustration, despondence, abandonment, and switching to competitors. Small experimental MVPs are rapidly specified, solutioned, and prototyped, and these are quickly provided to lead users for evaluation and collection of valuable feedback. This process is repeated many times until problems have been improved, end user delight is achieved, retention, trust, and profitability are yielded, and referrals and market growth increase. When a customer pain point is mitigated, differentiating your product or service from competitors, the next pain point can be addressed. Journey maps and roadmaps are continually revisited, revised, and improved because the assumptions upon which they're based are radically altered by the learning process. Market, customer, and end-user needs exist as tacit vs. explicit knowledge, so journey or roadmaps are a poor facsimiles, models, simulations, and representations of real problems, needs, and pain points. Journey maps and roadmaps are not static IMSs, business requirements, enterprise or system architectures, or chains of static long-term features in product backlogs.

14. **Test • Driven • De·vel·op·ment** (*tĕst • drĭv′ən • dĭ-vĕl′əp-mənt*) Trial, check, assess, analyze, examine, appraise, evaluate; To develop tests from a technical perspective that new products and services must satisfy

- ✓ Establish unit test criteria
- ✓ Develop and run unit test first
- ✓ Write enough code to pass unit test

Test Driven Development is a manifestation of early DevOps principles. Test driven development embodies the notion that all source code comprising a software-based product or service must be tested before it is delivered. It was a common malady throughout the 1980s and 1990s for programmers to write and deliver code with little to no testing. Throughout the 1950s, 1960s, and 1970s, computer time was expensive, so early computer programmers developed the habit of thoroughly testing software source code and then later invested more time in upfront code reviews, walkthroughs, and inspections. These led to early theories that code errors were 10 to 10,000 less expensive to be caught and mitigated in test and inspection than after they made their way into market, customer, and end user operation. However, untested code and fielded defects became pervasive as computers became more powerful, compiler technology advanced, interactive coding became possible, and the pre-Cambrian software explosion of the 1980s and 1990s ensued. Therefore, early agilists decided to address this issue by declaring that all software source code must be tested by the developers themselves without fail before delivery to quality

assurance (QA) or customers. That is, the criteria and automated tests must be developed and run before coding to address technical and development requirements. This became known as test driven development, which was a radical change among programmers who were used to coding as quickly as possible as long as the compiler didn't complain too much. Therefore, test driven development became a five step process of writing automated unit tests first, running tests and watching them fail, writing just-enough waste-free code to make unit tests pass, rerunning tests until they pass, refactoring or simplifying software source code units to work out any excess waste, and rinsing and repeating as much as time allows.

15. **Be·hav·ior • Driven • De·vel·op·ment** (*bĭ-hāv'yər • drĭv'ən • dĭ-vĕl'əp-mənt*) Step, action, activity, operation, scenario, sequence, procedure; To develop tests from end user or customer perspective that new products and services must satisfy

- ✓ Establish end user criteria
- ✓ Develop and run behavior tests
- ✓ Write enough code to pass behavior tests

Behavior Driven Development was the next evolution early DevOps principles. Test Driven Development is squarely focused on what is known as "development testing." That is, these tests are based on technical requirements, architecture, and design of individual software units. While this enables developers to assess the technical attributes of their software units before delivery, this leaves large gaps in the functional and non-functional "behavioral" requirements from a market, customer, and end-user perspective. Therefore, "behavioral testing" is done haphazardly, left for QA testers, or skipped altogether. You can imagine the surprise of QA testers or end-users if key behaviors are absent, inadequate, or defective! Therefore, behavior driven development evolved from 2003 to 2006 to address this gap, need, or deficiency in the testing practices of software-based new products and services. It pretty much follows the same steps as test driven development, of writing automated tests first, running tests and watching them fail, writing just-enough waste-free code to make tests pass, rerunning tests until they pass, refactoring or simplifying software source code to work out any excess waste, and rinsing and repeating as much as time allows. However, rather than focusing on technical developer or unit testing of fine-grained code attributes, these tests focus on the market, customer, and end-user behaviors expected of the final new product or service. This entails product owners communicating with the end-users, business owners, and developers, recording the end-user behaviors in English like statements, and generating behavioral tests instead of technical ones. Although behavior driven development is an evolutionary step up beyond basic test-driven development, it did not substitute for it. That is, both paradigms are required to be used in unison to ensure software-based new products and services satisfy their end user and technical requirements.

16. **Ac·cep·tance • Test • Driven • De·vel·op·ment** (*ăk-sĕp'təns • tĕst • drĭv'ən • dĭ-vĕl'əp-mənt*) Take, agree, receive, collect, certify, ratify, acknowledge; To develop tests from a customer or acquirer perspective that new products and services must satisfy

- ✓ Establish acceptance criteria
- ✓ Develop and run acceptance tests
- ✓ Write enough code to pass acceptance tests

Acceptance Test Driven Development is another early DevOps principle. It is a refinement of test-driven development and had earlier beginnings than behavior driven development, but did not gain wider use until much later. Test driven development focused on getting developers to consistently test their code in an era when testing was done poorly, haphazardly, or not at all. It focused on evaluating the technical attributes of software units or modules, often ignoring the larger market, customer, or end user perspective. Behavior driven development emerged to address this gap, that is, focus testing on the end-user's perspective of the outside in or black box view of the final software-intensive application. However, both of these paradigms still left a few gaps in the testing cycle, often ignoring enterprise, business, contractual, legal, security, accreditation, performance, deployment, and other critical attributes and parameters of the final product or service. Therefore, acceptance test driven development emerged to herd all of these cats and dogs into a comprehensive set of testing criterion and automated tests. It often involves all of the usual suspects, such as end-users, product owners, and developers, but broadens the collaboration to a larger audience of business and technical functional specialists. It also follows the same steps as test driven development, of writing automated tests first, running tests and watching them fail, writing just-enough waste-free code to make tests pass, rerunning tests until they pass, refactoring or simplifying software source code to work out any excess waste, and rinsing and repeating as much as time allows. However, it does not substitute for test driven development nor behavior driven development. Some people feel it expands upon and subsumes behavior driven development. In the end, all three testing paradigms need to be performed to ensure products and services satisfy their product or service requirements.

17. **Con·tin·u·ous • In·te·gra·tion** (*kən-tĭn'yo͞o-əs • ĭn'tĭ-grā'shən*) Join, knit, fuse, mesh, unite, merge, blend, combine, aggregate; To automatically incorporate new software modules into products and services in frequent short intervals

- ✓ Automatic integration server
- ✓ Automatically build every check-in
- ✓ Automatically integrate and initiate tests

Continuous Integration is an essential DevOps principle. It refers to developers who frequently integrate their software units into the main code branch to identify integration defects multiple times per day. Developers create code in an evolutionary and emergent style and it is in a high state of final testing readiness all of the time. User stories are implemented, integrated, and tested in priority order one at a time. Developers implement software units one at a time from the "beginning" of the project, which prevents bottlenecks associated with late big bang integration from traditional linear waterfall projects. If you save up integration and testing until the end of a project or release cycle, then testing takes hundreds, thousands, or tens of thousands of hours. Big bang integration and testing result in so many defects, it takes hundreds of hours to fix them, project timelines are extended, developers look bad, and customers, project managers, end-users are certainly dissatisfied. The practice of

frequently integrating code emerged in the late 1980s, gained some momentum with the object-oriented movement, became a regular practice at Microsoft as part of its daily operational build and smoke testing, and finally became a global practice with the introduction of Extreme Programming in the 1998 to 1999 timeframe. Continuous integration is roughly composed of eight broad steps, including monitoring version control, checking in software modules, automatic check in detection, automatic build and integration, automatic test initiation, automatic test reporting, immediately fix problems, generate deployment images and documentation, and release images for final testing if necessary. However, continuous integration didn't become a ubiquitous practice until early adopters started using it in the mid-2000s, which evolved into DevOps today. Continuous integration is at the heart of DevOps, while traditionalists still apply manual waterfall-based, late big bang, coding, integration, and testing.

18. **Con·tin·u·ous • Test·ing** (*kən-tĭn'yo͞o-əs • ˈte-stiŋ*) Trial, check, assess, analyze, examine, appraise, evaluate; To automatically test new software modules as they are added to products and services in frequent short intervals

- ✓ Run development tests upon check-in
- ✓ Run integration and system tests on check-in
- ✓ Run regression, operation, and acceptance on check-in

Continuous Testing is a new DevOps principle. It is the practice of identifying, detecting, mitigating, and preventing defects at the point of injection across the entire delivery pipeline. Extreme Programming brought at least two major practices to bear on the testing story in the 1998 to 1999 timeframe, automated unit testing and continuous integration. Continuous integration focused on building and integrating software units in an evolutionary style to ensure a high state of system readiness at all time. This of course, prevented late big bang integration failures and project bottlenecks. It supported iterative and incremental development, evolutionary and emergent architecture and design, and lean thinking. A forgotten element of continuous integration was the notion that automated unit, integration, system, and acceptance tests should be written by multiple stakeholders, codified as automated tests, and run by the continuous integration server. Like test driven development, it also implied continuous testing and final deployment. However, developers bypassed testing and skipped straight to build, integration, and deployment. Testing was ignored by developers and fell upon QA testers, comprising up to 80% of the team, and dumping this job upon them in late big bang two to three-week integration and testing marathons (24x7). Continuous testing brings automated testing back into DevOps, including unit, module, component, integration, system, behavior, usability, acceptance, performance, security, etc. It's hard to imagine DevOps without automated testing, but people somehow found a way. A holistic definition of continuous testing extends this paradigm to all points of the software development life cycle from planning, management, requirements specification, architecture and design, documentation, deployment, support, monitoring, security, etc. That is, at any point of delivery, the ecosystem should identify and prevent the injection of defects.

19. **Con·tin·u·ous • De·liv·er·y** (*kən-tĭn'yo͞o-əs • dĭ-lĭv'ə-rē*) Leg, step, phase, point, level, podium, platform; To automatically package a final certified version of new software-based products and service in frequent short intervals

- ✓ Deploy images to delivery host on check-in
- ✓ Run final performance and certification testing
- ✓ Generate final deployment image and information

Continuous Delivery is the next evolution of DevOps principles beyond continuous integration. Continuous integration enables basic evolutionary architecture and design principles. It does so by continuously integrating and initiating basic automated tests to smoke out the rudimentary issues of each software unit. There is nothing preventing continuous integration from initiating automated tests like system, behavior, acceptance, usability, performance, security, and deployment tests. A few early adopters used continuous integration as continuous delivery and deployment pipelines. However, continuous integration was squarely focused on automating the integration of software units into the main branch or trunk multiple times per day. Early adopters performed dozens and sometimes hundreds of integrations per day, and combined with sophisticated automated tests, decimated their bug counts and reliance on manual QA testing and testers. However, there remained a bridge, gap, or chasm between developers using continuous integration and IT operators who were responsible for final behavior, acceptance, load, security, and supportability testing. Furthermore, IT operations were responsible for designing computer platforms, hosts, and devices to satisfy stringent reliability, availability, and uptime requirements. In other words, even high-quality software images produced by developers using continuous integration were not to be trusted and needed to be independently verified, validated, and tested, manually if necessary. Thus, continuous delivery sprang forth to bridge this gap, to subsume many of the manual IT operations tests into automated continuous integration. Perhaps, independent continuous delivery servers were stood up to run automated behavior, acceptance, load, security, and supportability tests. This was the first major attempt to combine development and operations into what became DevOps as we know it today.

20. **Con·tin·u·ous • De·ploy·ment** (*kən-tĭn'yo͞o-əs • dĭ-ploi'mənt*) Give, fulfil, produce, provide, supply, transmit, distribute, transport; To automatically distribute new software-based products and service in frequent short intervals

- ✓ Deploy images to operational hosts upon check-in
- ✓ Immediately release images for general user availability
- ✓ Notify end users and begin customer support for new releases

Continuous Deployment is a fairly new evolution of DevOps principles beyond continuous delivery. Continuous delivery is designed to prepare a new software-based new product or service release for deployment in the form of a validated software image or load. However, continuous delivery is also concerned with preparing the final target hosting environment, platform, technology stack, and associated operations support services as well. Continuous delivery servers will therefore repeat or independently run many final acceptance, load, performance, security, accreditation, and supportability tests on the target environment as a whole, not just the functional application software release. Continuous deployment is often oversimplified by

describing it as actually releasing the final software image to the market, customer, or end users if it passes all final continuous delivery tests, criteria, and other packaging requirements. However, continuous deployment is not just a blue-green deployment of a software image but includes much more. Continuous deployment includes a host of very serious IT cloud and data center operations functions. These include application packaging release versioning, database updates, server configuration management, hosting resource configuration, operations and maintenance scheduling and calendaring, roll back and roll forward, security access and auditing, continuous scanning and testing, release monitoring, infrastructure performance monitoring, automated incidence response monitoring, automatic failover and redundancy, and canary releases, A/B testing, blue/green deployment, etc. Therefore, continuous deployment is far more than merely deploying software loads that pass continuous delivery tests or even running or rerunning acceptance, load, performance, security, and other tests on the images. Continuous deployment is concerned with automating the build and operation of the entire operating environment.

21. **Au·to·mat·ed • Ver·sion • Con·trol** (*ô′tə-mā′təd • vûr′zhən • kən-trōl′*) File, log, entry, image, record, duplicate, reproduction; To automatically store copies of new software modules created for products and services in frequent short intervals

- ✓ Automatically store software source code
- ✓ Automatically check-in and check-out source code
- ✓ Automatically build, integrate, test, deliver, deploy check-ins

Automated Version Control is one of the most basic DevOps principles. Automated version control is a repository for storing software source code units associated with a new product or service. It's basically an automated database that provides versioning, history, central storage, reliable backup, and shared repository visible to all team members. Contrast this to the use of private source code files. That is, developers who host their software source code files on their own PCs, computer systems, or file systems. They may even store their code on a thumbnail, other removable storage device, or even a private network repository or shared file service. Version control systems on the other hand provide a central storage location for all software source code, editors automatically store saved code in these repositories, everyone sees and owns the code, and there is no alternate storage. Automated version control repositories may be instrumented with continuous delivery servers that monitor all code check ins or saves. That is, upon code check in or commit, continuous integration servers compile and build the code, link it into to the rest of the trunk, run automated static and dynamic analysis tests, report to everyone whether the new code breaks the build or passes, and track code metrics too. They provide services like undo, diff, conflict, resolve, locking, checkout, check in, commit, stage, tracking, branching, local storage, remote storage, cloning, forking, merging, pulling, pushing, pull requests. While most of the version control functions support development automation, version control for IT operations is a little more robust, challenging, and demanding. IT operations personnel need to version control acceptance, load, performance, and security tests, baselines, and validated images. More importantly, IT operations needs to version control the support environment tools, build scripts, and target platform for repeatability, consistency, and reliability.

22. **Au·to·mat·ed • Con·fig·u·ra·tion • Man·age·ment** (*ô′tə-mā′təd • kən-fĭg′yə-rā′shən • măn′ĭj-mənt*) Hold, keep, store, retain, deposit, database, inventory; To automatically store artifacts for new products and services in frequent short intervals

- ✓ Automatically store deployed images
- ✓ Store application containers and environment
- ✓ Includes all tests, support services, and information

Automated Configuration Management is an important DevOps principle. Automated configuration management is a tool to identify and tag, control and change, report upon, and audit the state of software development assets belonging to a new product or service. Assets may be requirements, architectures, designs, code, tests, test data, documents, build scripts, or configuration data about operational hardware and software components. In good old-fashioned terms, it's an automated technical and management process for applying appropriate resources, processes, and tools to establish and maintain consistency between the product requirements, the product, and associated product configuration information. Another classical definition includes a discipline for applying technical and administrative direction and surveillance to identify and document the functional and physical characteristics of a configuration item, control changes to those characteristics, record and report change processing and implementation status, and verify compliance with specified requirements. A slightly more modern definition suggests its an automated process responsible for ensuring that the assets required to deliver services are properly controlled, and that accurate and reliable information about those assets is available when and where it is needed. And, a neo-classical definition includes an automated process for establishing and maintaining consistent settings and functional attributes for a system, including tools for system administration tasks such as IT infrastructure automation. It basically comes down to a content management system for user stories, tests, code, documentation, system administration scripts, and infrastructure as code as well. It not only applies to the software development assets, but the target environment, computer, platform, host, or device, along with IT operations tools to monitor, test, support, and maintain operational assets.

23. **Au·to·mat·ed • Stat·ic • A·nal·y·sis** (*ô′tə-mā′təd • stăt′ĭk • ə-năl′ĭ-sĭs*) Study, audit, check, assess, survey, examine, appraise, diagnose; To automatically verify physical attributes of new software modules for products and services in frequent intervals

- ✓ Automatic source code quality checks
- ✓ Automatic source code security checks
- ✓ Automatic test initiated if source code passes

Automated Static Analysis is a powerful DevOps principle. It consists of computer assisted code inspections, reviews, syntax and structural checking, semantic and security analysis, and many other properties. During the 1950s and 1960s, computer time was expensive, so developers spent thousands of hours reviewing punch cards, computer tapes, and software source code prior to compilation, interpretation, and operation. This was cost effective, considering computers had limited multi-

tasking abilities, there were thousands of jobs and users, and computers were slow. Code reviews fell by the wayside when microprocessors were commercialized, computers became faster, multi-processing capabilities became sophisticated, and microcomputers, engineering workstations, and personal computers became cost effective and ubiquitous. Testing practices from the 1950s and 1960s also fell by the wayside with faster computers, sophisticated operating systems and compilers, and basic debugging and operational testing. This led to an onslaught of computer bugs and software failures from 1980 to 1990, when the World Wide Web became a global phenomenon and end users mushroomed into the hundreds of millions and billions. The painful effects of poor testing and code reviews reared its ugly head as it had in the 1950s and 1960s and developers clamored for rigorous testing and code reviews once again. As software mushroomed into the hundreds of millions of lines of code, any manual review required thousands of hours and so many stage gates that software life cycles elongated to years, decades, and even approached the century mark in the case of fighter jets. The answer is rather obvious, simply automate the code review process, extend this to all life cycle artifacts, and let the computer perform thousands of reviews in fractions of a second before committing them to compilation, testing, delivery, release, deployment, and operation.

24. **Au·to·mat·ed • Test·ing** (*ô′tə-mā′təd • ˈte-stiŋ*) Unit, component, integration, system, acceptance, security, performance; To automatically test new software modules at all levels from smallest to final as they are added to products and services

- ✓ Automatic unit, module, and component tests
- ✓ Automatic integration, system, and regression tests
- ✓ Automatic performance, load, security, and deployment tests.

Automated Testing is an important DevOps principle. Software testing is a process of running, analyzing, and evaluating software to determine whether it satisfies its requirements, where deviations are known as bugs, defects, failures, or non-conformances. That is, does it meet its behavioral requirements, structural requirements, respond to inputs or stimuli correctly, satisfy its performance, usability, and security requirements, meet installation and operation requirements, various quality attributes, or acceptance criteria including legal, contractual, and other needs. Software testing is an operational examination performed by executing the resulting image produced by a compiler, build, integration, and installation process. Automated testing is the process of designing and executing computer readable test scripts, procedures, functions, and even test data much like the software source code itself and either compiling these test procedures or interpreting them to allow a computer to run the tests, detect conformances and non-conformances, and generate test reports and measurements. Creating and running automated test scripts allows for greater consistency, repeatability, completeness, speed, and overall test quality. Manual testing is a laborious, inconsistent, and qualitative with respect to depth, certainty, and quality. Hundreds, thousands, and millions of tests can be written, reused, run quickly and cheaply, smoke out any issues, validate images, builds, and integrations, and help ensure software is ready for deployment, use, and general availability. Automated tools are used for unit, module, component, integration, system, acceptance, regression, behavior, performance, load, capacity, security, installation, and operational testing, and evaluating any test level, attribute, feature, or need. Once again, automated testing not only supports development work but IT operations as well to evaluate, operate, and maintain virtual target platforms.

25. **Au·to·mat·ed • Re·port·ing** (*ô′tə-mā′təd • rĭ-pôr′tiŋ*) Note, record, account, message, statement, dispatch, communication; To To automatically generate status messages about new software modules as they are added to products and services

- ✓ Automatic test reports at all levels
- ✓ Automatic test distribution to everyone
- ✓ Automatic test visualizations and projections

Automated Reporting is an important DevOps principle. Test reports summarize the scope, strategy, and goals of software tests, the results of testing in terms of what requirements and test objectives were satisfied and which ones were not, and recommendations on whether to proceed or not. They also describe the operating environment and conditions at the time of the test to establish the context for the results. Defect profiles may be embedded within test reports identifying the class, type, and kind of defects, number and severity, and a graphical or statistical data to communicate this data. Automated reporting accompanies testing tools, which generate reports to summarize test results, are configurable by software testers, and may be tailored to communicate important details to stakeholders. Automated reports are generated for all test levels, including unit, module, component, integration, system, acceptance, regression, behavior, performance, load, capacity, security, installation, and operational testing. Automated reports may be generated one by one, when used in conjunction with exploratory testing, experimental MVPs, evolutionary and emergent design, lean thinking, and one-piece workflow. In this fashion, defects can be headed off early, they are not propagated into the overall code branch or trunk, prevention measures can be instituted, and the overall quality of the product and service improves dramatically. More importantly, late big bang integration testing is prevented, resulting in two-week, 24x7 release testing marathons, bottlenecks, frozen queues, unmanageable technical debt, delays, increased costs, and unpredictable project, product, and service performance. From an IT operations perspective, automated reporting is used to evaluate the health of building up and deploying virtual target systems as well as reporting upon the overall health and status of the deployed system, which is used for maximizing uptime.

26. **Au·to·mat·ed • Doc·u·men·ta·tion** (*ô′tə-mā′təd • dŏk′yə-mĕn-tā′shən*) Form, record, description, specification, design, tests, instructions; To automatically generate information about new software modules as they are added to products and services

- ✓ Automatic life cycle documentation
- ✓ Automated version and revision history
- ✓ Automated end-user and maintenance data

Automated Documentation is a key DevOps principle. Traditional documents consist of contracts, project plans, requirements, architectures, designs, tests, user and maintenance manuals, etc. Standards evolved to include dozens of documents for

each software project, and were the source of complexity, cost, and schedule risk. The theory was maintenance is 80% of the total software life cycle cost, so it's necessary to document tacit development knowledge for maintainers. Documents grew to have greater value than the software, a successful project, or customer satisfaction. It was a great way for traditionalists to get paid up front for high risk projects that would never complete! Large libraries were built to house documents for each project that made floors collapse! Studies showed maintainers only needed the software source code, so code comments became important. Software firms learned their most valuable asset was software source code and user and IT operations assistance were included in the source code. They simply gave source code to a temporary contract programmer who modified it and returned the new source code. This was a source of distrust among traditionalists during the age of agile methods, since working software was more important than expensive documents. Automated DevOps documentation is generated by the build, integration, test, staging, packaging, deployment, and IT operations tools in the form of valuable scripts. Firms perform thousands of deployments per day using these scripts, which are stored in configuration management tools for orchestration. Development and operations tools perform logging and IT operations tools do logging, monitoring, mitigation, rollbacks, rollforwards, blue/green testing, etc. in real-time. All of this is done thousands of times per day in an cost-effective automated way for bug fixing, platform hardening, security control, performance tuning, maintenance, enhancement, and upgrades.

27. **Au·to·mat·ed • Mon·i·tor·ing** (*ô′tə-mā′təd • mŏn′ĭ-tə′rĭŋ*) Check, follow, watch, survey, observe, oversee, supervise; To automatically keep an eye on new products and services as they are being used, interacted with, and operate in the field

- ✓ Automatic application monitoring
- ✓ Automatic application operations logging
- ✓ Automatic application vulnerability monitoring

Automated Monitoring is a central DevOps principle. IT monitoring is used to surveil, scan, and report data about the IT infrastructure to ensure uptime, security, and resource availability. Alerts, reports, and dashboards contain graphs, data, metrics, and statistics to warn IT operators when thresholds are tripped, such as saturated CPUs, storage devices, network components, bandwidth, memory, traffic, etc. These include operating system, middleware, status, and end-user demand, usage, and loading. This also includes monitoring application, storage, and web servers, as well as network devices such as firewalls, routers, switches, hubs, bridges, gateways, interface cards, modems, channel and data service units, and wireless access points. Automated monitoring consists of four broad practices, including application, infrastructure, log, and incident monitoring. The difference between IT and DevOps monitoring, is the latter is automated to the greatest extent possible. Commercial clouds provide pre-packaged automated monitoring services. Automated monitoring is undergoing a "shift-left" into "synthetic monitoring" to simulate end-user and operational usage to smoke out problems in the pre-deployment stages. The most advanced automated monitoring innovators are shifting even further left into the continuous delivery, continuous integration, and even development testing environment, as well as DevOps pipeline components themselves. Development, test, and staging servers mimic the deployment packaging, configuration, and load, and end-user, application, platform, and network traffic tests are run as part of development, integration, and acceptance testing. Virtual deployment platforms can be quickly and inexpensively spun up for development testing and real-time operational data streams can be fed into virtual development test servers to evaluate exploratory code against live behaviors without deploying development code first.

28. **Au·to·mat·ed • Roll·backs** (*ô′tə-mā′təd • rōl′băk′z*) Cancel, repeal, revoke, rescind, retract, replace, exchange, transpose; To revert new software-based products and services to a previously acceptable state if they become inoperable

- ✓ Automatic rollbacks upon system failure
- ✓ Automatic system restores to known state
- ✓ Automatic roll forward to future state as well

Automated Rollbacks are a bleeding edge DevOps principle. Rolling back returns a platform, server, application, hardware, or software to a previous working state or version if failures are encountered. This was a problem for mainframes, server farms, and web hosts in the 1990s. Any system upgrade, replacement, or fix required taking the entire system down for hours, days, weeks, or even months. As computers evolved, hot-swapping hardware and software became easier. Upgrading or downgrading cloud computing high-performance clusters involved taking down large mission critical enterprise servers to upgrade microcode, memory, microprocessors, and network components. This increased performance without incurring replacement costs. However, it was never a pleasant experience to take down a mission critical enterprise server, especially if they were 24x7 multi-tenant systems. Classical releases were done manually and haphazardly, so rollbacks were hit or miss! Automated rollbacks address physical and virtual rollbacks in DevOps. Automated DevOps pipelines and deployment scripting allows IT operations to repeat deployments with ease. This requires an investment in DevOps pipelines and automated deployment scripts. Some DevOps pipelines still require manual intervention, so it's best to have a well-defined deployment playbook. With cloud technology, rollbacks are even easier, because automated deployment scripts can spin up a prior virtual version at any time. The only cost is time and possible loss of customer data between deployment and rollback. Rollbacks are easier with blue/green deployments, because enterprises can maintain mirrored virtual deployment servers and simply switch routers between old and new versions. Furthermore, virtual hot backups can be spun up on an as needed basis. Containers make rollbacks even easier because fresh containers can be automatically reconstituted without complete redeployment.

29. **Au·to·mat·ed • Pipe·lines** (*ô′tə-mā′təd • pīp′līn′z*) Duct, tube, path, canal, course, channel, tunnel, conduit, passage; To automate the supply chain steps from concept to operation for deliverying new software-based products and services

- ✓ Automated concept to operations toolchain
- ✓ Automated build, test, delivery, and deployment
- ✓ Automated monitoring, security, and mitigation actions

Automated Pipelines are a critically important DevOps principle. DevOps is the synergy, combination, and intersection of IT development and operations people, processes, and tools to deliver high-quality products or services to the market, customer, or end-user in the shortest lead and cycle time. Traditional development and operations involve a loose series of functional departments, roles, specialists, processes, tools, cultures, and standards. These include project managers, architects, designers, programmers, testers, security, sysadmins, data center designers, operators, and maintainers. Each of these functions may be performed by a different company, vendor, business unit, department, project function, or team. They may even be split across international and national boundaries, making difficult to move data across these functions, because each has its own culture, standards, guidelines, processes, tool, and data. DevOps increases speed, quality, customer satisfaction, and business value by streamlining these functions. DevOps involves the use of vertical cross functional (collocated) feature teams. If a feature team is to develop a new product or service in a few days or weeks, then it needs one set of standards, guidelines, processes, tools, and data. They should strive automate as much manual labor as possible, consolidate the development and deployment platforms, and standardize tools for version control, continuous integration, testing, delivery, deployment, and operations. As developers commit code check-ins, they are built, tested, staged, deployed, and operated in seconds or minutes. (Automated) DevOps pipelines eliminate 80% of traditional software development costs and most of the IT operations costs, especially with cloud services implemented as infrastructure as code. Firms such as Amazon and Google perform hundreds of millions of tests, tens of thousands of deployments per day, and split-second virtual rollbacks 24x7.

30. **Dev·Ops • Ex·pe·ri·ence** (*dĕv'ŏps • ĭk-spîr'ē-əns*) Step, means, action, policy, process, operation, procedure; To make process of developing, delivering, and operating new software-based products and services easy, fast, and fun

- ✓ Design thinking for DevOps pipeline
- ✓ Empathetic view of all DevOps personnel
- ✓ Design DevOps for simplicity, ease, and speed

DevOps Experience is an emerging DevOps principle. Experience design is a principle, philosophy, and process of improving customer, market, and end-user satisfaction, engagement, and delight with new products or services. This is achieved by improving the form, function, performance, safety, security, ease, usability, accessibility, desirability, meaning, relevance, and other relevant attributes. This involves applying design thinking principles, such as interacting and empathizing with users, taking their perspective, understanding their challenges, defining user needs, ideating innovative solutions, prototyping those solutions, testing assumptions, and rinsing and repeating until a satisfactory end-user solution is devised. This improves market, customer, and end-user satisfaction, delight, and trust, which increases business value for buyers and suppliers. Likewise, DevOps experience is a principle, philosophy, process, and practice of improving developer and operator satisfaction, morale, engagement, trust, and delight with the entire pipeline of development, delivery, deployment, and operations processes, practices, and tools. People belonging to both buyer and supplier stakeholder groups must be deeply satisfied in order to create innovatively new products and services with the shortest possible lead and cycle times. Don't forget lowering risk, uncertainty, cost, complexity, duration, defects, complications, and project failures. That is, we must consider the form, function, performance, safety, security, ease, usability, accessibility, desirability, meaning, relevance, and other relevant attributes of the entire DevOps pipeline, including people, processes, and tools. DevOps experience principles include the front-end ALM system, version control, build, integration, testing, staging, deployment, and operations lifecycle. This is how Amazon and Google run hundreds of millions of tests to make tens of thousands of deployments per day.

31. **In·fra·struc·ture • as • Code** (*ĭn'frə-strŭk'chər • ăz • kōd*) Stand, stage, podium, platform, grounding, foundation, substructure; To virtualize network, computer, operating system, middleware, and interfaces of new software-based products and services

- ✓ Automatic scripting of platform generation
- ✓ Version control, test, and deployment of scripting
- ✓ Automatically generate Platform-as-a-Service (PaaS)

Infrastructure as Code is also an powerful DevOps principle. Development, testing, and operations teams were functionally divided, segregated, and organized, along with their policies, standards, guidelines, processes, practices, tools, and platforms. That is, developers, testers, and operators created and tested their outputs on unique platforms. Functional silos led to poor communication, collaboration, and compatibility, bringing lead and cycle times to a complete halt in many cases. Continuous integration better prepared new code for testing by early builds, integrations, and unit tests, continuous delivery tested new code by better system, behavior, and acceptance testing, and continuous deployment tested code by better performance, load, security, and installation testing. These teams automated many of these functions and even scripted them for repeatability, consistency, and completeness of results. DevOps teams collaborated to share tools, scripts, platforms, and tests to ensure that that any one team targeted their output to identical operating platforms instead of unique ones leading to impediments, bottlenecks, and deployment failures. The best teams consolidated, married, and linked their tools and tests together and targeted their outputs to the same platform for blue-green deployments for A/B testing. Since the latter process involved automating the entire process including check in, build, test, staging, deployment, and operations, a single end-to-end automated DevOps script could be created eliminating 80% of testing, staging, deployment, and operations costs. Cloud services enable teams to virtually constitute target platforms as code scripts to provision platform, infrastructure, and software as a service. Designing, combining, testing, storing, and executing automated DevOps and virtual provisioning scripts into a single chain results in infrastructure as code. Deployments occur in seconds as developers check-in code into version control.

32. **Ar·ti·fi·cial • In·tel·li·gence • Op·er·a·tions** (*är'tə-fĭsh'əl • ĭn-tĕl'ə-jəns • ŏp'ə-rā'shənz*) Mind, sense, reason, awareness, knowledge, judgement, understanding; To automate delivery of software-based products and services using expert systems

- ✓ Expert systems for DevOps pipeline

- ✓ Expert system-based test, delivery, and deployment
- ✓ Expert system-based monitoring, security, and mitigation

Artificial Intelligence Operations is an advanced DevOps principle. IT operations is the philosophy, culture, people, process, tools, and discipline of managing an IT infrastructure that hosts new products and services. IT operations involves IT platform, infrastructure, application, and incidence or help desk management. IT operations also involves planning, architecture, design, contracting, facility, HVAC, data center, server, storage, network, equipment, cabling, end user device, telecommunication, deployment, provisioning, access, monitoring, auditing, performance, reliability, security, backup, recovery, inventory, asset, safety, and configuration management, etc. Artificial intelligence operations uses highly-automated, self-managing expert systems tools and technologies to manage many of the traditional IT operations functions mentioned above. Artificial intelligence operations uses big data, analytic, and machine learning processes, tools, and technologies to automate real-time deployment, monitoring, incidence, and recovery management. It's used to perform routine administration, identify and differentiate between nominal and serious incidents and issues, display dashboards and reports, facilitate collaboration between operators and end-users, support decision making responses, protect the infrastructure from intrusion, and automatically correct problems. Artificial intelligence operations is even "shifting left" into the planning and development stages, to provide assistive, evaluation, and corrective services during lean-agile planning and management, development, and testing, staging, and initial deployment. That is, it's used across the end-to-end DevOps pipeline, including application lifecycle management, version control, build, integration, test, staging, deployment, monitoring, reporting, and response ecosystem to decimate lead and cycle times, improve quality and reliability, customer satisfaction, and business value.

Summary

So, what exactly have we learned by this short treatise on continuous integration, continuous delivery, and DevOps principles? Well, we've learned that the more things change, the more they seem to stay the same! Software development and IT operations are still treated as two large functionally divided silos that do not collaborate very well slowing creation of new software-based products and services by the majority of worldwide organizations. We've also learned that DevOps is a philosophy, culture, and discipline of combining the people, processes, and tools of developers and operators to decimate lead and cycle times. We've learned that early lean and agile methods from the 1990s, while reducing large batch sizes with inordinate size, complexity, cost, risk, duration, and failure a bit, were still manually intensive carbon copies of large linear manufacturing era waterfall lifecycles. Although lean and agile methods have entered the early majority of technology adoption, many of them still jam too much scope into product backlogs, functionally divide their stages, make poor handoffs, and give lip service to collaboration with IT operations. Surprisingly, we've learned there are many closet case traditionalists taking a wait-n-see attitude toward lean, agile, DevOps, and Cloud Computing, hoping they'll fail, and 1970s document driven lifecycles will return! More importantly, we've learned that IT operations is still locked in the Stone Age too, but early adopters are flocking to public clouds, reducing IT operating costs by more than 80% instantly. Customers only know what they want when they see it, so IT operators are seeing the benefits of Cloud Computing sooner than developers struggling with DevOps principles. Although, software developers were the drivers of lean, agile, and DevOps principles, IT operators may be keys to pushing software developers to the tipping point of adopting DevOps!

32 Principles of Successful Continuous Integration, Continuous Delivery, & DevOps

1. **Lean Thinking**—Value stream mapping; Fast lead and cycle time; Continuous improvement
2. **Lean Startup Thinking**—Form hypothesis; Small experiments; Measure and repeat
3. **Pull Driven Onepiece Workflow**—Pull-based system; One-piece workflow; Work in process limits
4. **Collaborative Teams**—Small team size; Self-selected groups; Informal and temporary
5. **Emergent Design**—Just-in-time; Situational solution; Extemporaneous structure
6. **Evolutionary Architecture**—Incremental change; Trial and error based; Elastic scope and size
7. **Domain Driven Design**—Highly specialized; Problem set oriented; Limited generalizations
8. **Security by Design—**Small attack surface; Multi-layer authentication; Limited single point of failures
9. **Containerized Microservices**—Small bare-metal solutions; Fully encapsulated applications; Self-contained operating sys.
10. **Feature Flags**—Built-in switches and toggles; Multiple feature release audiences; Early user and market testing of features
11. **Blue Green Deployments**—Multiple deployment hosts; Release to hidden backup host; Switch to new platform when ready
12. **Product Orientation**—Long lived value streams; Small just-in-time experiments; Market facing products or services
13. **Design Thinking**—Problem space analysis; User centered perspectives; Human factor-based solutions
14. **Test Driven Development**—Establish unit test criteria; Develop and run unit test first; Write enough code to pass unit test
15. **Behavior Driven Development**—Establish end user criteria; Develop and run behavior tests; Write enough code to pass
16. **Acceptance Test Driven Development**—Establish acceptance criteria; Develop and run tests; Write enough code to pass
17. **Continuous Integration**—Automatic integration server; Automatically build check-ins; Automatically integrate & initiate tests
18. **Continuous Testing**—Run development unit, integration, system, regression, operation, and acceptance on check-in
19. **Continuous Delivery**—Deploy images to host; Run final performance and certification testing; Generate final deployments
20. **Continuous Deployment**—Deploy images to hosts; Immediately release to users; Begin customer support for new releases
21. **Automated Version Control**—Automatically store, build, integrate, test, integrate, test, deliver, and deploy source code
22. **Automated Configuration Management**—Automatically store images, containers, environment, and support information
23. **Automated Static Analysis**—Automatic software source code quality and software security and vulnerability checks
24. **Automated Testing**—Automatic unit, integration, system, regression, performance, load, security, and deployment tests
25. **Automated Reporting**—Automatic test reports, distribution, visualizations, and projections at all levels to everyone
26. **Automated Documentation**—Automatic lifecycle documents, version and revision history, and user and maintenance data
27. **Automated Monitoring**—Automatic application monitoring, operations logging, intrusion detection, and vulnerability profile

28. **Automated Rollbacks**—Automatic rollbacks, restores to known state, and roll forward to future state upon system failure
29. **Automated Pipelines**—Automated concept, design, development, testing, monitoring, security, and mitigation actions
30. **DevOps Experience**—Design thinking, empathetic view of DevOps personnel, and simple, easy, and fast DevOps pipeline
31. **Infrastructure as Code**—Automatic scripting, version control, and generation of Platform-as-a-Service (PaaS)
32. **Artificial Intelligence Operations**—Expert systems for planning, development, test, delivery, staging, and operation

We've also learned that people have streamlined software development processes since the inception of computers, programming languages, and operating systems. Software factories emerged to achieve economies of scale by combining thousands of programmers to collaborate and share knowledge, define end-to-end processes, and automate them, but computers were too primitive. Shockingly, we've learned computers were so slow and expensive, people put more effort into the activities preceding programming, testing, deployment, and operations to prevent problems before touching a computer. These came in the form of billion-dollar enterprise architectures, project plans, requirements, architectures, designs, test plans, and painful code inspections. Mathematicians chipped in with reliability theories from the 1950s, illustrating that upfront investments in pre-computer activities reduced downstream errors, cycle times, and frightening maintenance costs, a myth that would survive for 50 years. We also learned that maintenance engineers put the fear of god into software developers, forcing them to create mountains of documents to convert tacit knowledge into explicit data for downstream IT operators. Software factories errantly focused on automating these upstream activities with primitive computers from the 1970s to no avail. It wasn't until the 1990s that lean mathematics emerged to undo the damage caused by these paleolithic doomsday theories, showing that upfront investments created so much WIP and waste, they elongated queues, caused deadlocks, stopped productivity, increased cost and duration, and, of course led to frighteningly high project failure rates. That is, waterfall lifecycles actually caused failure instead of reducing it. Lean and agile methods emerged in the 1990s to thwart waterfall methods exacerbating the software crisis, which developers continue to resist.

The most surprising thing we've learned is that lean and agile methods were created in the 1990s to put the spotlight back on working software which frightened traditionalists to death! Screaming fast and dirt cheap WinTel personal computers from the 1990s reawakened software developers to the notion of automating software development with a vengeance. Early practices like code sharing, version control, unit testing, test driven development, continuous integration, and automated software testing also came under the spotlight in the 1990s, but were slow to catch on for the next 25 years. Continuous delivery and DevOps practices emerged from 2005 to 2010 to form closer collaboration between project managers, developers, testers, and IT operators and combine and automate their practices into a single ecosystem called a DevOps pipeline. At first traditionalists complained loudly about these practices, but then moved into the dark recesses as the 21st century marched forward hoping integrated master schedules, maturity models, waterfall lifecycles, and floor crushing software document libraries would be vindicated. Mature software developers stood on the edge between the old and brave new DevOps world focusing on upstream processes, Amazon conquered the world and Stone Age IT operators saw the light of automated virtual provisioning. Oftentimes, software developers were the best and the brightest, while weaker technical people became IT operators. But, the tables turned, the meek inherited the Earth, and it is the IT operators that are starting to drive the Lean, Agile, DevOps, and Cloud Computing revolution! So, while traditionalists moved into the shadows and DevOps adoption among software developers remained surprisingly low, IT operators virtually migrated their infrastructure, application, and operations to the cloud returning 80% of IT costs back to the business!

Further Reading

- AbouZaid, A. (2016). *Continuous delivery and maturity model*: *Devops*. Retrieved September 18, 2019, from http://tech.aabouzaid.com/2016/01/continuous-delivery-and-maturity-model.html
- Armstrong, S. (2016). *Devops for networking*: *Boost your organization's growth by incorporating networking in the devops culture*. Birmingham, UK: Packt Publishing.
- ASPE, Inc. (2017). *Devops maturity model quiz*: *Deployments edition*. Retrieved September 18, 2019, from http://techtowntraining.com/resources/tools-resources/devops-maturity-model-quiz
- Barker, K., & Humphries, C. (2008). *Foundations of rspec*: *Behavior driven development with ruby and rails*. New York, NY: Apress.
- Bass, L., & Klein, J. (2019). *Deployment and operations for software engineers*. Middletown, DE: Independent.
- Bass, L., Weber, I., & Zhu, L. (2015). *Devops*: *A software architect's perspective*. Old Tappan, NJ: Pearson Education.
- Beck, K. (2003). *Test-driven development*: *By example*. Boston, MA: Addison-Wesley.
- Bell, L., Brunton-Spall, M., Smith, R., & Bird, J. (2017). *Agile application security*: *Enabling security in a continuous delivery pipeline*. Sebastopol, CA: O'Reilly Media.
- Belmont, J. M. (2018). *Hands-on continuous integration and delivery*: *Build and release quality software at scale with jenkins, travis ci, and circleci*. Birmingham, UK: Packt Publishing.
- Bird, J. (2016). *Devopssec*: *Delivering secure software through continuous delivery*. Sebastopol, CA: O'Reilly Media.
- Bonofous, P. (2018). *The art of rollback part 1*: *Is rollback really possible*? Retrieved September 24, 2019, from https://www.ca.com/en/blog-automation/the-art-of-rollback.html
- Bonofous, P. (2018). *The art of rollback part 2*: *Defining object and component types, and which rollback strategy to select*. Retrieved September 24, 2019, https://www.ca.com/en/blog-automation/the-art-of-rollback-part-2.html
- Bonofous, P. (2018). *The art of rollback part 3*: *Rollback strategies through the ages*. Retrieved September 24, 2019, from https://www.ca.com/en/blog-automation/the-art-of-rollback-part-3.html
- Branca, J., Maguire, C., & Hoeflin, C. (2019). *The total economic impact of broadcom continuous testing*: *Cost savings and business benefits enabled by broadcom*. Cambridge, MA: Forrester Research.
- Bucena, I. (2017). *Devops maturity model*. Retrieved September 18, 2019, from http://devopsadoptmeth.wordpress.com/method-description/devops-maturity-model

- CA Technologies. (2018). *The definitive guide to continuous testing*: *How to succeed in your continuous testing journey*. San Jose, CA: Broadcom.
- Crispin, L., & Gregory, J. (2009). *Agile testing*: *A practical guide for testers and agile teams*. Boston, MA: Addison-Wesley.
- Crispin, L., & Gregory, J. (2015). *More agile testing*: *Learning journeys for the whole team*. Boston, MA: Addison-Wesley.
- Cusumano, M. A. (1991). *Japan's software factories*: *A challenge to U.S. management*. New York, NY: Oxford University Press.
- Davis, J., & Daniels, R. (2016). *Effective devops*: *Building a culture of collaboration, affinity, and tooling at scale*. Sebastopol, CA: O'Reilly Media.
- Duffy, M. (2015). *Devops automation cookbook*: *Over 120 recipes covering key automation techniques*. Birmingham, UK: Packt Publishing.
- Duvall, P., Matyas, S., & Glover, A. (2006). *Continuous integration*. Boston, MA: Addison-Wesley.
- Farcic, V. (2016). *The devops 2.0 toolkit*: *Automating the continuous deployment pipelines with containerized microservices*. Victoria, CA: LeanPub.
- Farcic, V. (2017). *The devops 2.1 toolkit*: *Docker swarm*. Victoria, CA: LeanPub.
- Farcic, V. (2017). *The devops 2.2 toolkit*: *Self-sufficient docker clusters*. Victoria, CA: LeanPub.
- Farcic, V. (2018). *The devops 2.3 toolkit*: *Kubernetes*. Victoria, CA: LeanPub.
- Farcic, V. (2018). *The devops 2.4 toolkit*: *Continuous deployment to kubernetes*. Victoria, CA: LeanPub.
- Farcic, V. (2018). *The devops 2.5 toolkit*: *Monitoring, logging, and auto-scaling kubernetes*. Victoria, CA: LeanPub.
- Farcic, V. (2019). *The devops 2.6 toolkit*: *Jenkins x*. Victoria, CA: LeanPub.
- Farcic, V. (2019). *Devops paradox*: *The truth about devops by the people on the front line*. Birmingham, UK: Packt Publishing.
- Farooqui, S. M. (2018). *Enterprise devops framework*: *Transforming it operations*. New York, NY: Apress Media.
- Fleming, S. (2018). *Devops with kubernetes*: *Non-programmer's handbook*. Charleston, SC: CreateSpace.
- Forsgren, N., Humble, J., & Kim, G. (2017). *Forecasting the value of devops transformations*: *Measuring roi of devops*. Portland, OR: DevOps Research.
- Forsgren, N., Humble, J., & Kim, G. (2018) *Accelerate*: *The science of lean software and devops*. Portland, OR: IT Revolution.
- Freeman, E. (2019). Devops For dummies. Hoboken, NJ: John Wiley & Sons.
- Gartner, M. (2013). *Atdd by example*: *A practical guide to acceptance test-driven development*. Upper Saddle River, NJ: Pearson Education.
- Gruver, G. (2016). *Start and scaling devops in the enterprise*. Pennsauken, NJ: BookBaby.
- Gruver, G., & Mouser, T. (2015). *Leading the transformation*: *Applying agile and devops at scale*. Portland, OR: IT Revolution.
- Gruver, G., Young, M., & Fulghum, P. (2013). *A practical approach to large-scale agile development*. Upper Saddle River, NJ: Pearson Education.
- Harrison, D., & Lively, K. (2019). *Achieving devops*: *A novel about delivering the best of agile, devops, and microservices*. New York, NY: Apress Media.
- Hiring, M. (2018). *Devops for the modern enterprise*: *Winning practices to transform legacy it organizations*. Portland, OR: IT Revolution.
- Howard, M., & LeBlanc, D. (2003). *Writing secure code*: *Practical strategies and techniques for secure application coding in a networked world*. Redmond, WA: Microsoft Press.
- Howard, M., & Lipner, S. (2006). *The security development lifecycle*: *A process for developing demonstrably more secure software*. Redmond, WA: Microsoft Press.
- Howard, M., LeBlanc, D., & Viega, J. (2005). *19 deadly sins of software security*: *Programming flaws and how to fix them*. Emeryville, CA: McGraw-Hill.
- Hsu, H. T. (2018). *Hands-on security in devops*: *Ensure continuous security, deployment, and delivery with devsecops*. Birmingham, UK: Packt Publishing.
- Hsu, H. T. (2019). *Practical security automation and testing*: *Tools and techniques for automated security scanning and testing in devsecops*. Birmingham, UK: Packt Publishing.
- Humble, J., & Farley, D. (2011). *Continuous delivery*. Boston, MA: Pearson Education.
- Huttermann, M. (2012). *Devops for developers*: *Integrate development and operations the agile way*. New York, NY: Apress.
- Kan, S. H. (2002). *Metrics and models in software quality engineering*. Boston, MA: Addison-Wesley.
- Kandha, U. (2019). *The devops assessment playbook*: *A comprehensive assessment primer to assess technology organizations for devops*. New Delhi, India. Adhyyan Books.
- Katz, D. (2018). *Continuous delivery for mobile with fastlane*: *Automating mobile application development and deployment for ios and android*. Birmingham, UK: Packt Publishing.
- Kim, G., Debois, P., Willis, J., & Humble, J. (2016). *The devops handbook*: *How to create world-class agility, reliability, and security in technology organizations*. Portland, OR: IT Revolution Press.
- Kinsbruner, E. (2018). *Continuous testing for devops professionals*: *A practical guide from industry experts*. Charleston, SC: CreateSpace.
- Liebel, O., (2019). *Scalable container infrastructures with docker, kubernetes, and openshift*: *The compendium on container clusters for enterprise administrators and devops teams*. Bonn, Germany: Rheinwerk Verlag GmbH.
- Martynov, M., & Evstigneev, K. (2019). *Continuous delivery blueprint*: *Software change management for enterprises in the era of cloud, microservices, devops, and automation*. San Ramon, CA: Grid Dynamics.
- Miller, M., et al. (2018). *Epic failures in devsecops*: Volume 1. New York, NY: DevSecOps Days Press.
- Millstein, F. (2018). *Devops adoption*: *How to build a devops it environment and kickstart your digital transformation*. Charleston, SC: CreateSpace.

- Morris, K. (2016). *Infrastructure as code*: *Managing servers in the cloud*. Sebastopol, CA: O'Reilly Media.
- Mukherjee, J. (2015). *Continuous delivery pipeline*: *Where does it choke*. Charleston, SC: CreateSpace.
- Munns, C. (2016). *Devops @ amazon*: Microservices, 2 pizza teams, & 50 million deploys a year. Canadian Executive Cloud & DevOps Summit, Vancouver, British Columbia, Canada.
- Ohara, D. (2012). *Continuous delivery and the world of devops*. San Francisco, CA: GigaOM Pro.
- Parks, J. (2016). *The solinea devops maturity model*. Retrieved September 18, 2019, from http://www.solinea.com/blog/solinea-devops-maturity-model
- Praqma, Inc. (2019). *Continuous delivery assessment*. Retrieved September 18, 2019, from http://www.praqma.com/services/assessments
- Pugh, K. (2011). *Lean-agile acceptance test-driven development*: *Better software through collaboration*. Boston, MA: Pearson Education.
- Ravichandran, A., Tayler, K., & Waterhouse, P. (2016). *Devops for digital leaders*: *Reignite business with a modern devops-enabled software factory*. New York, NY: Apress Media.
- Rehn, A., Palmborg, T., & Bostrom, P. (2013). *The continuous delivery maturity model*. Retrieved September 18, 2019, from http://www.infoq.com/articles/Continuous-Delivery-Maturity-Model
- Rico, D. F. (2014). *18 reasons why agile cost of quality (coq) is a fraction of traditional methods*. Retrieved June 25, 2014, from http://davidfrico.com/agile-vs-trad-coq.pdf
- Rico, D. F. (2016). *Business value, roi, and cost of quality (coq) for devops*. Retrieved June 1, 2016, from http://davidfrico.com/rico-devops-roi.pdf
- Rico, D. F. (2016). *The 10 attributes of successful teams, teamwork, and projects*. Retrieved September 26, 2016 from http://davidfrico.com/teamwork-attributes-2.pdf
- Rico, D. F. (2016). *The 12 attributes of successful collaboration between highly-creative people*. Retrieved February 29, 2016, from http://davidfrico.com/collaboration-attributes.pdf
- Rico, D. F. (2017). *U.S. dod vs. amazon*: *18 architectural principles to build fighter jets like amazon web services using devops*. Retrieved January 26, 2017, from http://davidfrico.com/dod-agile-principles.pdf
- Rico, D. F. (2018). *The agile mindset*: *18 attributes of successful business leaders, managers, and teams*. Retrieved January 1, 2018 from http://davidfrico.com/agile-mind-attributes.pdf
- Rico, D. F. (2019). *Business value of ci, cd, & devopssec*: *Scaling up to billion user global systems of systems using end-to-end automation and containerized docker ubuntu images*. Retrieved August 23, 2019, from http://davidfrico.com/rico19c.pdf
- Rico, D. F. (2019). *Evolutionary architecture*: *24 principles of emergent, organic, and highly-adaptive design*. Retrieved September 3, 2019, from http://davidfrico.com/evolutionary-architecture-principles.pdf
- Rico, D. F. (2019). *Lean leadership*: *22 attributes of successful business executives, directors, and managers*. Retrieved August 28, 2019, from http://davidfrico.com/lean-leadership-principles.pdf
- Riley, C. (2015). *Documenting devops*: *Agile, automation, and continuous documentation*. Boca Raton, FL: MediaOps, LLC.
- Rossel, S. (2017). *Continuous integration, delivery, and deployment*: *Reliable and faster software releases with automating builds, tests, and deployment*. Birmingham, UK: Packt Publishing.
- SANS Institute. (2017). *Security devops toolchain*: *Securing web application technologies* (*SWAT*). Retrieved September 24, 2019, from https://www.sans.org/security-resources/posters/secure-devops-toolchain-swat-checklist/60/download
- Sharma, S. (2017). *The devops adoption playbook*: *A guide to adopting devops in a multi-speed it enterprise*. Indianapolis, IN: John Wiley & Sons.
- Singleton, A. (2014). *Unblock*: *A guide to the new continuous agile*. Needham, MA: Assembla, Inc.
- Skelton, M., & Pais, M. (2019). *Team topologies*: *Organizing business and technology teams for fast flow*. Portland, OR: IT Revolution Press.
- Smart, J. F. (2014). *Bdd in action*: *Behavior-driven development for the whole software lifecycle*. Shelter Island, NY: Manning Publications.
- Soni, M. (2017). *Devops bootcamp*: *The fastest way to learn devops*. Birmingham, UK: Packt Publishing.
- St-Cyr, J. (2017). *Continuous improvement for sitecore devops*. Retrieved September 18, 2019, from http://jasonstcyr.com/tag/devops
- Swartout, P. (2014). *Continuous delivery and devops*: *A quickstart guide*. Birmingham, UK: Packt Publishing.
- TechWorld (2017). *The devops handbook*: *Transforming your organization through agile, scrum, and devops principles* (*An extensive guide*). Charleston, SC: CreateSpace.
- Vadapalli, S. (2017). *Hands-on devops*: *Explore the concept of continuous delivery and integrate it with data science concepts*. Birmingham, UK: Packt Publishing.
- Vadapalli, S. (2018). *Devops, continuous delivery, integration, and deployment with devops*: *Dive into the core devops strategies*. Birmingham, UK: Packt Publishing.
- Vehent, J. (2018). *Securing devops*: *Security in the cloud*. Shelter Island, NY: Manning Publications.
- Verona, J. (2018). *Practical devops*: *Implement devops in your organization by effectively building, deploying, testing, and monitoring code*. Birmingham, UK: Packt Publishing.
- Weeks, D. E. (2017). *Devops and continuous delivery reference architectures*. Fulton, MD: Sonatype.
- Weller, C. (2017). *Devops adoption*: *A must-have for any it leader facing digital transformation*. Charleston, SC: CreateSpace.
- Weller, C. (2017). *Devops handbook*: *Simple step by step instructions to devops*. Charleston, SC: CreateSpace.
- XebiaLabs. (2018). *Periodic table of devops tools*. Retrieved September 24, 2019, from https://xebialabs.com/periodic-table-of-devops-tools

Case Studies in Continuous Integration, Continuous Delivery, & DevOps

- Coursera. *Coursera is a large provider of 2,000 online classes to 150 universities in 29 countries and 25 million students. Coursera used domain driven design to decompose its monolithic code base into 60 individual applications. It then hosted these as containers and migrated these to AWS EC2 Container Registry, which was far better than the cost and time of mastering, maintaining, and performance tuning on-prem servers. Coursera then used AWS CodeBuild to build its applications for scaling and concurrency, which took only two weeks of effort. It now runs 60 jobs in parallel for the time of only one. Along with building its applications and uploading them to Amazon S3, AWS CodeBuild also creates Docker containers that include the assets and uploads them to the Amazon EC2 Container Registry. Coursera was able to deploy the highly customized build environment to AWS to manage its legacy dependencies from its monolithic application disintermediation. Building, deploying, and spinning up new containerized microservices with tightly coupled legacy interdependencies is a cinch now. Coursera increased its build speed by 10 times, while simultaneously reducing cost, complexity, effort, skill, and pain. The entire set of 60 applications decoupled from the legacy monolithic architecture take only 10 minutes to build. Coursera does 500 builds per day in the cloud with individual builds running in parallel. It's on-demand, so they never have any infrastructure resources running idle or unused. It's scalable, elastic, and highly flexible. Coursera also runs its continuous integration and continuous delivery in AWS and has decimated the QA testing costs, time, resources, and pain. Continuous delivery staging environments for behavior, user acceptance, performance, load, and security tests are spun up on-demand too. These tests can be run in parallel as many times as necessary and developers can quickly fix, build, and redeploy code to staging virtual servers quickly. The use of containerized microservices for its 50 to 60 applications makes builds, testing in stage, deployment to target servers, and operations fast, easy, inexpensive, consistent, and reliable. Rollbacks are as easy as terminating a containerized microservice, debugging and fixing it, building it in the cloud, deploying it back to cloud staging for testing, and reconstituting it in the virtual environment. Coursera only has to debug applications, not the build process, staging environment, or operational platform. Its customers always have the latest courseware without bugs, innovative options, great learning experiences, and highly interactive, high-performance 24x7 courseware availability. This is far better than their old single monolithic code base that took developers weeks to debug and build 24 hours a day. It's also less expensive, easier, faster, and less painful than hosting their own physical on-prem servers, DevOps pipeline, and trying to optimize its performance. AWS gives Coursera instant access to expertly designed, state-of-the-art platform, infrastructure, and application as a service technology.* (http://aws.amazon.com/solutions/case-studies/coursera-codebuild)
- MicroStrategy. *Founded in 1989, MicroStrategy is a $500 million firm with 2,500 employees that provides federated business intelligence, analytics, mobility automation, and zero-click hyperintelligence analytics based on personalized wireless biometrics. Because of the advent of big data, it was important to improve the speed of analytic processing that was not possible with home-grown physical on-prem data centers and servers. Therefore, MicroStrategy migrated its analytic processing platform to AWS and abandoned its expensive, lower-performing on-prem IT platform. Upgrading on-prem hardware and infrastructure was long, slow, painful, expensive, and unreliable, and the process to build their analytic platform took 24x7 releases over days and weeks. MicroStrategy's market went global, and large institutions now wanted to deploy its analytics to tens of thousands of users with each new enterprise it added to its portfolio. MicroStrategy simply did not have enough power, space, cooling, expertise, time, money, or consistency to scale its on-prem servers to the multi-billion end-user global Internet. MicroStrategy moved it entire platform to AWS and quickly elected to utilize the latest high-tech infrastructure as code for automated virtual provisioning. End-users can now launch highly available, high-reliability enterprise analytics instantly for tens of thousands of users. Behind the scenes, MicroStrategy uses Amazon Elastic Compute Cloud (EC2), Relational Database Service (RDS), security groups, and Elastic File Service (EFS). MicroStrategy uses AWS CodeDeploy to configure EC2 instances for each deployment. MicroStrategy also uses Amazon's state of the art big data services for the fastest possible massively parallel analytic processing speed of large data sets. They reduced the number of DevOps deployment, installation, and configuration clicks by over 133 times from 400 to only 3. Talk about fast, painless, easy, enjoyable, and inexpensive. MicroStrategy reduced deployment effort by 1,000 times from over 500 hours per deployment to only 30 minutes. That's about how much time it takes to onboard a new global client with tens of thousands of end users. Neither the buyer nor supplier have to concern themselves with the pain, time, expense, worry, stress, cost, risk, and uncertainty of standing up new custom data centers. MicroStrategy's customers in 60 countries can get one-click self-serve software upgrades in seconds on a 24x7 basis. MicroStrategy's clients can profit by instantly analyzing petabytes of market and enterprise data in seconds using its on-demand analytics. MicroStrategy can also do instant product demonstrations and training sessions by virtually provisioning deployment servers. Better yet, MicroStrategy can now focus on sales, marketing, and innovation instead of lagging IT operations. Using state-of-the-art automated DevOps principles, MicroStrategy transformed from a caterpillar into a butterfly overnight with respect to digital transformation without the expense of transitioning its stone-age development and IT department.* (http://aws.amazon.com/solutions/case-studies/microstrategy)
- Expedia. *Expedia is a top 10 online travel service with 24,000 employees and $10 billion in revenues. Expedia provides $100 billion in business and personal travel reservations for customers in over 70 countries. Expedia divested 80% of its on-prem data centers, while migrating its mission critical travel applications to AWS. This freed Expedia to focus on mergers, acquisitions, partnerships, sales, advertising, marketing, innovation, and new product and service development. Software developers innovate faster than ever without getting bogged down by testing, staging, deployment, and operations bottlenecks. Freed up IT operations resources were diverted to mitigate the causes of abandoned shopping carts, increasing revenues, profits, customers, and market share. Since 50% of its annual revenue comes from the international marketplace, the use of AWS allowed it to expand globally without having to incur the cost, expense, scalability, performance, and reliability problems associated with on-prem data centers. International latency times were reduced by 14 times from 700 milliseconds to only 50 milliseconds with 240 requests per second. Expedia uses AWS Elastic Map Reduce (EMR) and Simple Storage Services (S3) to store and process customer interaction big data sets in real time. Expedia deploys its entire travel service stack using Chef, AWS CloudFormation, and Virtual Private Cloud (VPC). Expedia scales new clusters based on elastic customer demand to manage its high-volume Global Deals Engine (GDE) and Expedia Suggest Service (ESS). They simply couldn't have done that with any degree of cost efficiency, speed, reliability, consistency, and availability using on prem data centers. Expedia runs its applications at capacity thresholds and triggers new clusters when exceeded, which burned out on-prem equipment. Expedia uses the AWS Identity and Access Management (IAM), Security Token Service (STS), and Management Console, easing the pain, overhead, and administration of multiple on-prem IAM services. Automated split-second server provisioning allows global developers to try out regional applications and tools, adopt them if appropriate, or rapidly innovate new ones. Expedia was able to innovate and deploy their new ESS MVP in only 90 days using AWS DevOps services, dramatically reducing shopping cart abandonment and increasing customer satisfaction, retention, revenues, profits, and market share. This would have taken years using their old on-prem data centers, due to administrative, IT, and performance bottlenecks. Expedia develops applications quickly, scales to demand fluctuation and big data better, and resolves issues much quicker. Infrastructure is no longer a multi-year, multi-million dollar administrative, cost, time, and performance bottleneck. Reliability, uptime, availability, and resiliency are maximized, and disaster recovery is painless. Because of this flexibility, architectures and designs dynamically evolve and emerge without investing in big up front, multi-year, multi-million-dollar enterprise architectures.* (http://aws.amazon.com/solutions/case-studies/expedia)