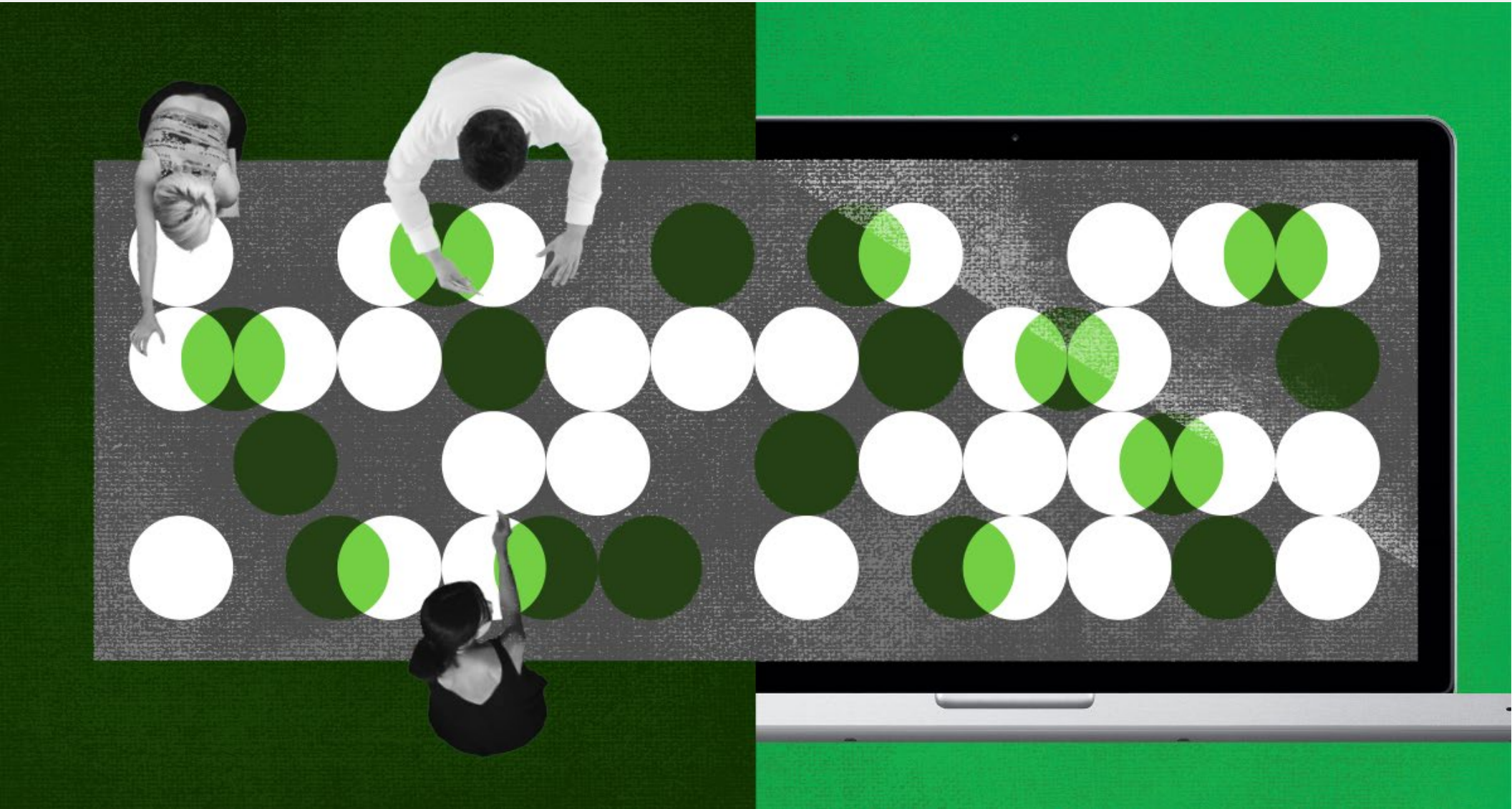
circleci

A Brief History of DevOps

by Alek Sharma



Introduction: History in Progress

Software engineers spend most of their waking hours wading through the mud of their predecessors. Only a few are lucky enough to see green fields before conflict transforms the terrain; the rest are shipped to the front (end). There, they languish in trenches as shells of outages explode around them. Progress is usually glacial, though ground can be covered through heroic sprints.

But veterans do emerge, scarred and battle-hardened. They revel in relating their most daring exploits and bug fixes to new recruits. And just as individuals have learned individual lessons about writing code, our industry has learned **collective** lessons about software development at scale. It's not always easy to see these larger trends when you're on the ground — buried in bugs and focusing fire on features. DevOps is one of these larger trends. It's the unification of two traditionally disparate worlds into one cohesive cycle. But it's not some recent invention or fad; it's the result of years of iteration, as engineers have cracked risk into progressively smaller chunks. Not every engineer experienced this process first-hand. And for those who missed the march towards continuous development, there may be gaps in understanding **why** or **how** we got here.

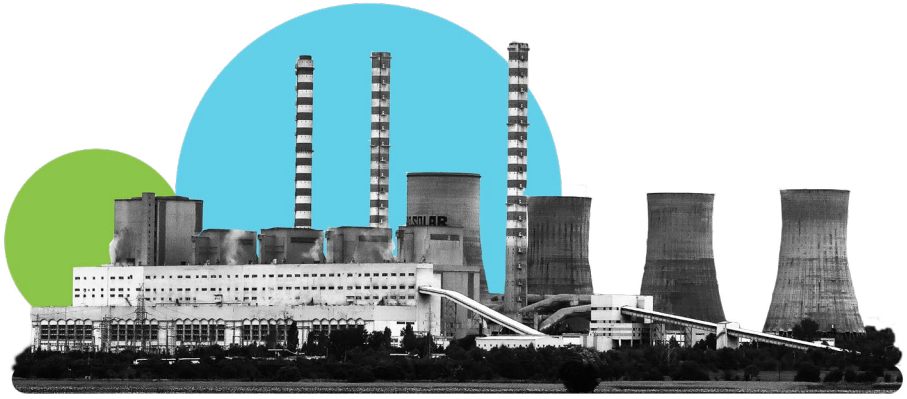
George Santayana wrote that "those who cannot remember the past are condemned to repeat it." He was definitely not thinking about software when he wrote this, but he's dead now, which means he can be quoted out of context. Oh, the joys of public domain!

This ebook will be about the history of software development methodologies — especially where they intersect with traditional best practices. Think of it as *The Silmarillion* of Silicon Valley, except shorter and with more pictures. Before plunging into this rushing river of time, please note that the presented chronology is both theoretically complete and practically in progress. In other words, even though a term or process might have been coined, it always takes more time for Best Practices to trickle down to Real Products. And trickling down is where we'll start.

Part I: Waterfall

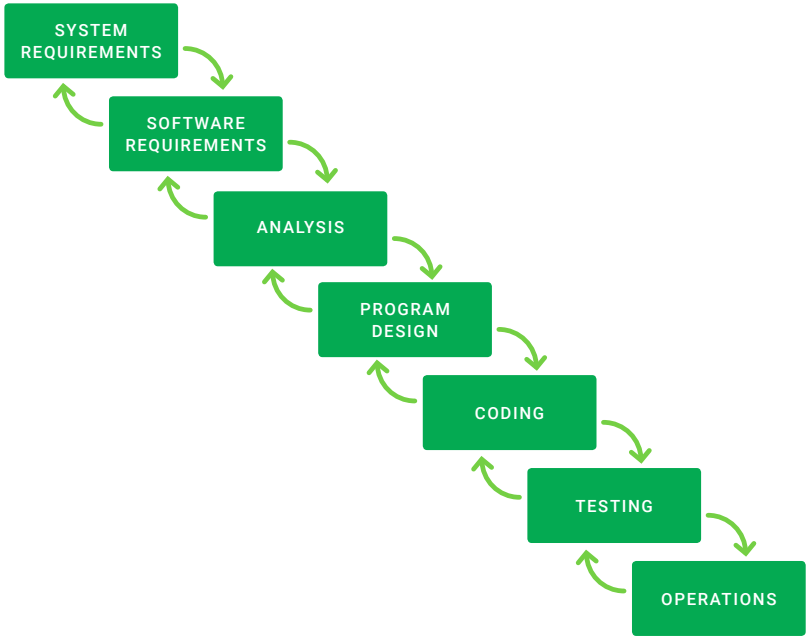
WATERFALL

A sequential software development practice which emphasizes planning and hinges on predictability.



When software development was still young, engineers modeled their work on the hardware world. This is an environment where making mistakes is expensive, so it made sense to iron out kinks before producing a thousand units of a thing. If a toy company, for example, decided that a stuffed moose should **also** provide witty political commentary, that discovery would need to happen **before** production started. Otherwise, they'd make thousands of silent moose units, and everybody knows those don't sell.

In 1970, Dr. Winston Royce wrote an article called Managing the Development of Large Software Systems. This article outlined a process (shown below) that bore an eerie resemblance to the accepted methodology of the hardware world.



The process itself comprises several phases, each of which must be completed in its entirety before moving to the next phase — with the exception of minor iteration between adjacent phases. Royce never used the word “waterfall” anywhere in his paper; in fact, the original genius who attached this word to the process has been lost to history.

But everyone readily accepted it because it made sense: as waterfalls flow downward, so too do the phases of software development. How brilliantly expressive! Never let it be said that software engineers do not have passionate, poetic cores.

The intent behind the waterfall model was to create a methodical, unsurprising production cycle. Its advocates loudly proclaimed its strengths: it generates oodles of documentation before coding even begins; it delivers a Certain Something defined In Advance; it’s always easy to know how much work remains. In theory, these all sound wonderful, but waterfall development suffers when put into practice.

Behold nature in all its Relatively Linear Sequential glory!

Under Pressure

While Royce believed in the waterfall model conceptually, he thought actual implementation was "risky and invited failure". This is because of the method's (in)famous inflexibility. Its success lies in getting each phase **exactly right** the first time. There is very little room to take lessons from the Testing Phase back to the Design Phase; this would undermine the very goals of waterfall development: predictability, certainty, consistency — all those gluey goodies that keep things slow and steady.

So waterfall development is, ironically, not very fluid. At scale, the model just doesn't move quickly enough to leverage the speed of software or adapt to the market's changing needs. And since the schedule reigns supreme, there's often massive procrastination and subsequent death marches to keep the train on time.

"One does not simply develop software like hardware." — Dr. Winston Royce

Like your grandparents, waterfall development is motivated by good intentions but is kind of out of the technological loop. When projects stick to their waterfall guns, there is often a Grand Piling Up of work. In the software world, this can mean months of design and planning before any coding even happens. By the time it's finished, the product might not be relevant or even wanted!

Agile Minds

Software is built on hardware.

And early software took pointers from hardware because it didn't know any better. So, software developers inherited a development strategy that just wasn't a great fit for writing code. And as the world accelerated, companies realized they needed to be more nimble.

Waterfall is often depicted as the clown in the court of software development methodologies. We've defined it to distance ourselves from the "backward traditionalists" who haven't managed to move onto modern practices. But the truth is that waterfall development was reality at one point; without it as a baseline, we wouldn't have been able to iterate ourselves into the future.

Every journey has to start somewhere! In the case of this ebook, that somewhere was an outdated practice — guaranteed to cause stress and dramatic tension.

Part II: Agile

AGILE

A software development practice designed to be flexible and iterative, encompassing adaptive planning, early feedback, and continual improvement.

In the last chapter, we discussed why the history of software development is important and how waterfall development fit into that narrative. Remember that waterfall development was actually rather rigid. It lacked the flexibility to adapt to change, a noticeable weakness in a world that is increasingly volatile.

In this chapter, we're going to explore how (and to what extent) engineers iterated on the waterfall model through **Agile Software Development**. Instead of trying to **control** change by locking down phases of development, Agile methodology is about **embracing** change. Risk is lessened not by devising the perfect plan, but by cutting projects into small chunks and adapting on the fly.

Agile Ancestors

While it would be really convenient if waterfall development led directly to agile development, the actual timeline isn't so clean. Even in Winston Royce's 1970 paper, there was already a general awareness of the need for iteration and lightweight processes. This awareness only grew as developers discovered weaknesses in prevailing workflows.

These workflows were heavy, inflexible things. Software was still treated as something to be manufactured, and the methods in vogue reflected this mindset. Behemoths like Microsoft focused on planning and eliminating surprises during development. Perfectionism was encouraged, manifesting in astronaut architecture and Big Design Up Front.

Generally, Agile methodology was a reaction to overzealous documentation and low thresholds for change. The software industry, advocates believed, was too focused on process and planning — to the detriment of the product and customer. While features languished in bureaucratic, corporate purgatories, everyone suffered: developers couldn't ship their creations, products fell behind the competition, and users were condemned to either use outdated technology or flee to a competitor.

All this pressure to improve led to a slew of workflow experiments in the 90s. The results of these experiments were the precursors to Agile methodology and shared common beliefs around speed, risk, and chaos.

Rapid Application Development

At IBM, James Martin created **Rapid Application Development**, which emphasized process and prototyping over the excessive planning of waterfall development. He consolidated his ideas into a 1991 book, creatively titled *Rapid Application Development* (RAD).

RAD had no trouble finding fans, as prototyping became increasingly viable across the industry. But its fans disagreed on the **proper** way to RAD, in part because of the intentionally vague instructions left by Dr. Martin. Businesses were cautious, hesitant to embrace a fad without a guarantee that they wouldn't be sacrificing quality for speed.

Dynamic Systems Development Method

In 1994, the **Dynamic Systems Development Method** emerged to unify the various versions of RAD. Unlike its predecessor, DSDM was spearheaded not by an individual, but by a consortium of motivated vendors and experts. And instead of just outlining a process, DSDM got its hands dirty by defining explicit techniques and project roles.

Two of those noteworthy techniques are **timeboxing** and **MoSCoW** prioritization. Timeboxing involves cutting a project into chunks that have resource caps — on both time and money. If either resource begins to run out, efforts are focused on the highest priorities, defined by MoSCoW. While the actual acronym is questionable, MoSCoW's intents are sound. Look at what those capital letters stand for:

- Must have
- Should have
- Could have
- Won't have

By grouping requirements into these four levels, engineers always know exactly which ones to drop when pressed for time or money. Which is inevitable.

Both timeboxing and MoSCoW are concerned with **time**, safeguarding against the software industry's traditional perfectionism. By implementing constraints on **how much** work can be done and establishing frameworks for the **value** of that work, DSDM kept people moving.

Scrum

But folks weren't done iterating! In 1995, Kevin Schwaber and Jeff Sutherland went to a research conference on Object-Oriented Programming, Systems, Languages & Applications. The acronym for this is OOPSLA, which sounds accidental but is actually very intentional.

There, Schwaber and Sutherland presented a jointly-authored paper on a new process called **Scrum**. Scrum was yet another framework for developing software and focused on **collaboration** and **flexibility**. The term comes from rugby, where both teams restart play by putting their heads down and focusing on getting the ball. It's not a subtle metaphor.

Diligent, athletic software developers with their heads down.

Scrum is one of the first philosophies that explicitly emphasized **agility** and **empirical process** control. All of Scrum's values revolve around a single principle: uncertainty. Are you **certain** the customers will like this? Are you **certain** you understand everything about this project?

True certainty, argue the Scrumsters, is impossible to achieve — an asymptote of productivity. Where Waterfall attempted to define everything ahead of time, Scrum abandoned that fantasy in favor of smaller batch sizes. Hone the delivery process instead, focusing on adaptation and emerging requirements. This is the essence of Scrum and drives all of its tactics.

Those tactics include familiar routines like **sprints**, **daily standups**, and **retrospectives**. All of these are meant to drive work in focused bursts of productivity. The regular check-ins provide opportunities to course-correct and aim energy at emerging priorities.

One important tool of Scrum is the **backlog**. There are usually multiple backlogs at any given point: one for the product generally, and then a backlog per sprint. These backlogs give the team an efficient method for prioritizing work. These artifacts are so important to proper Scrum execution that whole industries have grown up around them. Pivotal Labs and Atlassian's JIRA are just two of the companies creating issue trackers.

Extreme Programming

The final entrant into this motley crew of methodologies is Extreme Programming (XP), coined by Kent Beck in March of 1996. Beck was working on a payroll project with Chrysler when he devised the basic tenets of XP. While the project was cancelled, the methodology survived. And what does the methodology teach? Gather all the best practices and ratchet them up to an **EXTREME** level.

For example, if some testing is a Good Thing, then **more** testing must be a **Better** Thing. This translates to test-driven development: where tests describing behavior are written before the code proper. This practice not only increases the total number of tests in an application, but also changes how the code itself is written.

Another example involves optimization of feedback loops. Advocates of XP argue that a ruthless focus on short feedback loops increases the odds that you're making the right thing. To that end, Extreme Programmers should write code for the present and receive feedback on that code as quickly as possible.

This commitment to feedback manifests in practices like **pair programming**, where developers write code in two-person units. By having to articulate what they're writing, engineers write better code and find bugs before they happen. Teams also commit to frequent releases, often manifesting in processes like **continuous integration**, which will be covered in the next chapter.

Variations on an Agile Theme

If you've noticed a theme here, you're not alone. While these philosophies technically emerged independently of one another, there was a general awareness of the **need for speed**. But along with that collective recognition, there was also a genuine lack of semantic unification: words like **evolutionary** and **adaptive** were being tossed around, but these had different nuances depending on which methodology you happened to be living in.

There were simply too many cooks in the kitchen, and some of these cooks hadn't even met each other! Some of these thought leaders had crossed paths at other conferences, but it took a motivated soul named Bob Martin to herd all the cats into one place. In September of 2000, Martin sent out an email to gauge interest, and the rest is… well, the rest is below.

The Manifesto

Like all great philosophies, Agile proper originated in Utah. There, seventeen developers — many of them representatives of the aforementioned methodologies — met to organize their thoughts on what Being Agile even **meant**. The actual *Agile Manifesto* isn't very long, consisting of twelve principles total. These principles adhered to **four values** that summarized how Agile advocates were to make decisions:

Individuals and Interactions	→	Processes and Tools
Working Software	→	Comprehensive documentation
Customer Collaboration	→	Contract Negotiation
Responding to Change	→	Following a Plan

The Agile Manifesto consolidated several existing methodologies under one proverbial roof: from Extreme Programming, they lifted customer satisfaction and collaboration; from Scrum, they incorporated the introspection of standups and retrospectives; and from Pragmatic Programming, they took the… pragmatism.

It's important to note that the Agile manifesto was not itself a new methodology, but rather a collection of existing methodologies. The "Agile Alliance" brought all the best parts together, hoping that the whole would be better than its various components. And it has: Version One publishes an annual "State of Agile" report measuring the methodology's **consistently positive impact on the industry**.

To many engineers, however, "Agile" has become a buzzword — a term brandished by companies desperately seeking to prove their relevance to a shallow pool of talent. The "Agile Alliance" itself can seem somewhat pompous: why should a group of self-proclaimed leaders at a ski resort be able to decide what the industry should or shouldn't become?

The **idea of Agile** can sometimes distract from the Agile Manifesto itself. Ultimately, Agile methodology is nothing more than a collection of engineering principles, created to help software teams make practical decisions more quickly. It is **not** a well-defined process or magical recipe for success, and companies who say that they "are Agile" are often the furthest from living these principles.

Agile's Ultimate Impact

But even if Agile is just a collection of engineering principles, does that make it unhelpful or impractical? Not necessarily. Agile is a **movement**. And like all movements, it's easy to become emotionally invested without understanding everything about it. It doesn't help that "agile" is also just a normal word that means something simpler than an Entire Movement.

Agile Methodology is a reaction to — and revision of — a way of doing things that developers didn't think was working. It irked enough of them that they had to retreat to a safe space to collect their thoughts on the matter. What came out of that was a list of principles that weren't themselves processes, but **could be used** to generate processes.

And that's the larger theme here: the mission of this gathering was to help people change the **way they thought** about developing software. Instead of a product to be manufactured, code should be an art to be crafted. That subtle difference implies a host of alternate strategies, such as timeboxing, pair programming, and test-driven development.

Waterfall was slow and rigid. It couldn't adapt to change, so engineers tried to control it. But that just wasn't responsive enough to deliver quality software customers actually wanted. Agile methodology was a response to this weakness, and its advocates realized that they could never completely eliminate risk or uncertainty from a project. Instead, they focused on **containing** risk through rapid development and constant validation that their work was heading the right way.

And this idea of constant validation will be the seed for the next chapters in the history of DevOps: continuous integration and delivery.

Part III: Automated Testing and Continuous Integration

AUTOMATED TESTING

The process of automatically and continuously testing software to prevent regressions.

CONTINUOUS INTEGRATION

A process by which developers merge frequent, small commits to the main codebase, reducing the potential conflicts of large, infrequent integrations.

Automated Testing

We've already seen how the Agile movement was inspired by moving away from Heavy Documentation and towards Light Chunks of Work. Everyone was experiencing a collective revelation: risk is unavoidable, so you might as well minimize that risk by cutting it into small chunks.

As people grew more comfortable with failure, they created places where failure was not only acceptable, it was **encouraged**. The rise of testing frameworks like JUnit and Cucumber reflected this new attitude toward risk. Tools don't precede processes — we create them to optimize existing processes. This meant that engineers were spending enough time writing tests to warrant the creation of these frameworks.

Extreme Programming was the primary influence behind this Zest for Tests. Kent Beck and Ron Jeffries had championed test-driven development with a convincing argument: "If bugs are going to happen (and they will), engineers should fix at least **some** of them while they still remember the code."

Writing tests **before** the code forced engineers to think about how they wanted software to work. In many ways, this practice served the same purpose as the exhaustive documentation of the Waterfall years: tests specified how things **should** be. But **automated** tests went one step further by **enforcing** these requirements on each code change, loudly complaining when something wasn't right.

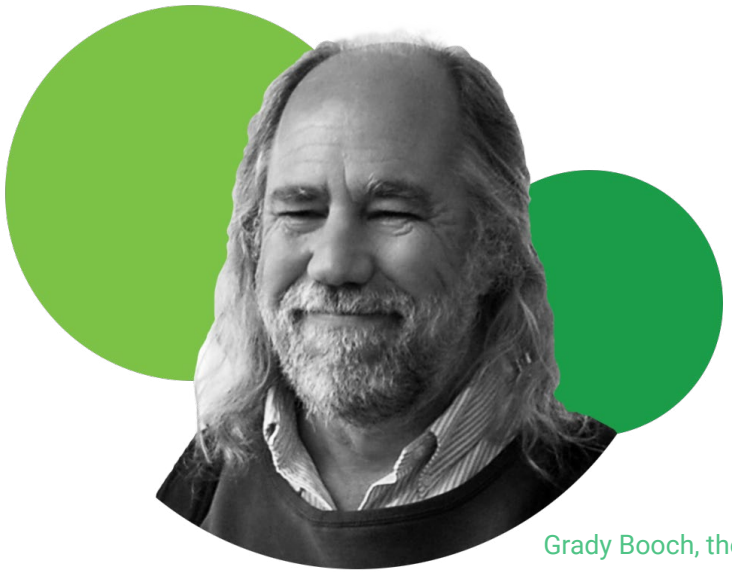
Engineers discovered that, by running tests more frequently, they increased the odds of catching bugs and preventing catastrophes. That meant checking **as you go** instead of **at the end**. This might seem obvious today but, at the time, it was a fundamentally different way of developing software.

And it's from this bubbling, primordial soup of testing that continuous integration was born.

On the Origin of Continuous Integration

The DevOps Timeline isn't as clean as we'd like it to be.

Rather than a discrete progression from the Agile Era to the Age of Continuous Integration, the two concepts are actually contemporaries. The phrase **continuous integration** (CI for short) is actually older than many of Agile's ancestors. It was coined in 1994 by Grady Booch, a man of considerable academic and mutton chops.



Grady Booch, the first continuous integrator

Booch is most (in)famous for developing the Unified Modeling Language (UML) with two colleagues: Ivar Jacobson and James Rumbaugh. UML is a great example of the overzealous design phase from which the Agile Alliance tried to distance itself, but beyond that, has no place in the rest of the story of DevOps.

The phrase itself appears in Booch's book, *Object-Oriented Analysis and Design with Applications*:

> The needs of the micro process dictate that many more internal releases to the development team will be accomplished, with only a few executable releases turned over to external parties. These internal releases represent a sort of **continuous integration** of the system, and exist to force closure of the micro process.

While the bolding is for emphasis, the rest is directly from Booch's mind. He also writes that testing should be "a continuous activity during the development process", so from early on, testing and continuous integration were closely intertwined.

Pay close attention to Booch's choice of words: he describes the "needs of the micro process", and this is a recognition of the **pace of software development**. He's not discussing normal-sized or macro processes — no, these are **micro** processes because the slices of delivery were being cut ever more thinly.

Spot the Difference

If you've read our discussion of feedback loops, these concepts should be ringing all sorts of bells. The Agile movement itself could be described as a grand reduction of feedback loops in software. Continuous integration — while more process-oriented than Agile's various tactics — is a natural extension of this reduction.

When done well, continuous integration acts as a kind of unending course correction — a formalized application of the belief that **uncertainty is the one certainty**. While it has much in common with its Agile cousins, continuous integration differs in scale. The Agile movement provides a bundle of tactics that makes an organization more responsive; continuous integration is a strategic shift that fundamentally changes an organization's culture.

But these distinctions quickly become blurry. When Kent Beck and Ron Jeffries created Extreme Programming in 1996, they officially adopted continuous integration as one of its twelve core practices. They even took it further, advocating **multiple integrations per day**.

So: Agile and continuous integration were contemporaries, one didn't necessarily lead to the other, and continuous integration is not **just** another Agile tactic — even though it is an application of Agile principles. All of this ambiguity is an unfortunate side effect of discussing recent events: we haven't really had time to agree on a single narrative yet!

But there was one thing everyone **could** agree on: CI was a Good Thing — good enough to build tooling around to make it repeatable and scalable.

ThoughtWorks and CruiseControl

In 2000, one of the founding members of the Agile Alliance became an internal advocate for continuous integration at ThoughtWorks. His name is Martin Fowler, and he's rather famous. Fowler points out that continuous integration isn't conceptually tied to specific software; developers can **practice** CI without a tool dedicated to it. But — as with any home-rolled solution — personalization comes at the cost of time, money, and the ability to **build a network of users**.

So practicing CI can be made easier with a server built specifically for daily builds and continuous testing. And in 2001, another ThoughtWorker named Matthew Foemmel built this and dubbed it CruiseControl. The tool is open-source and still around, though there are fewer maintainers these days.

CruiseControl was important. It helped bring a best practice into reality, and its public nature encouraged adoption across the industry. It didn't have a ton of responsibility: watch for changes in an application's codebase and, if necessary, create a new build and run it through a suite of tests to verify that everything actually still works.

Hudson and Jenkins

While CruiseControl was a relatively simple tool, it represented a commitment to both software and chronological integrity. Soon, other contenders were sprouting up around the concept like daffodils. Hudson, another CI tool written in Java, was released in 2005 but didn't really start to overtake CruiseControl until 2008.

Then! As all good stories are wont to do, there was a Dramatic Dispute. Hudson's author, Kohsuke Kawaguchi, had been working at Sun Microsystems, which Oracle purchased in 2010. Like CruiseControl, Hudson was free and open-source, so when Oracle announced its plans to trademark and commercialize it, the Hudson community fled the scene in 2011 under the new name of Jenkins, which continues strong to this day.

(Best) Practice Makes Perfect

There are many places to read about the fallout of that split, so we won't cover them here. Instead, let's take a step back and discuss how these various tools (and disagreements) represent the industry's rapid acceptance of a best practice.

In just a decade, developers formulated, proposed, implemented, and commercialized a suggested process. The fact that Oracle was willing to put up a fight indicates that the idea — and its implementation — had value. And that value has percolated across the software world until it's become so well-ingrained that the question has shifted from "**Are** you continuously integrating?" to "**When** and **how often** are you continuously integrating?"

For a while, Facebook's motto was "Move Fast and Break Things". As the software industry has matured, engineers have realized that you can't actually move that quickly if everything is breaking. That's why Facebook changed its motto to "Move Fast With Stable Infra".

Continuous integration is the first step on that path. Catching problems before they get out of hand demonstrates a shift in the treatment of risk. It's the logical next step on the agile escalator, and the reason it's an escalator in this metaphor is because the **journey never ends**.

Part IV: Continuous Delivery and Deployment

CONTINUOUS DELIVERY

Maintaining a code base in a state of perpetual readiness to be released, with high confidence, at any time.

CONTINUOUS DEPLOYMENT

The automatic deployment of that code base whenever all automated tests are passed.

We have arrived at the final chapter (for now) in our epic trek through the history of DevOps! We've talked about how the Agile movement gave rise to the more defensive practices of automated testing and continuous integration. Now, we'll be discussing the progression of that theme with **continuous delivery and continuous deployment**.

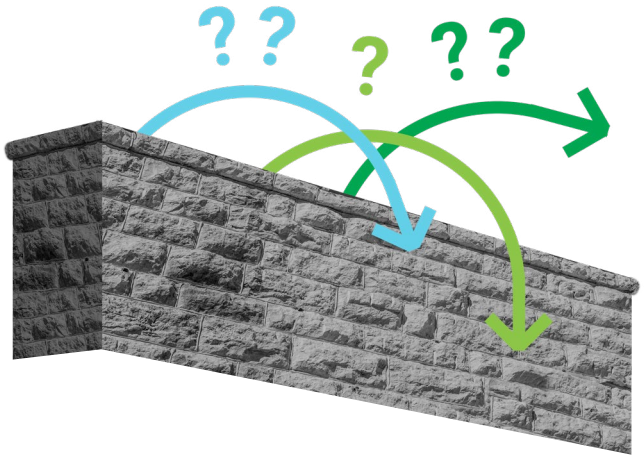
Admittedly, the introduction of two more "continuous" terms is a bit confusing, especially because the lines between one term and the next are so blurry; it doesn't help that they are recent developments and could be renamed **at any moment**.

But fear not! We're well-equipped to deal with this kind of volatility and will tread carefully. At this point in our saga, "history" is hardly even the right word for what we're discussing. What we're really talking are **current events**, which is very exciting but also much squishier than more concrete subjects like waterfall development or Agile methodology. But enough throat-clearing; off we go!

Continuous Delivery

Remember when we explored the meaning of continuous integration. We talked a lot about the advantages of constantly merging bits of code — all in the name of avoiding enormous code conflicts at the end of a development cycle. People were excited to reap the rewards of this methodology and eagerly boarded the continuous integration train.

But the train didn't always cross the border between developers and operators who were pushing that code into production. Developers washed their hands of code they pushed, safe in the knowledge that it was "someone else's problem." Operators anticipated changes from the other side with dread, trembling at the horrors lobbed over the wall.



It didn't matter that developers were coming together to merge their code before releasing it; customers still couldn't actually use anything if it hadn't been released. In this way, there was still some very real distance between what the developers made and "the real world," and no one was minding the gap.

But in 2010, Jez Humble and David Farley released a book called Continuous Delivery, where they proposed an extension of continuous integration. They argued: "Software that's been successfully integrated into a mainline code stream still isn't software that's out in production doing its job." In other words, it doesn't matter how fast you assemble your product if it's just going to sit in a warehouse for months.

Continuous delivery is the practice of ensuring that software is always ready to be deployed. Part of that insurance is testing every change you've made (a.k.a. continuous integration). In addition, you've also made the effort to package up the actual artifacts that are going to be deployed — perhaps you've already deployed those artifacts to a staging environment.

So continuous delivery actually **requires** continuous integration. It relies on the same underlying principles: cutting work into small chunks and carefully titrating the flow of product to your users. But, to quote CircleCI's CTO, "software that isn't ready to deploy is dangerous"; you should always be prepared to send your code out into the world.

Developers + Operators = DevOps

DevOps is the mashup of development and operation. It's the crossover episode, the sodium chloride; the fusion of all that is good with both worlds. It's not a piece of software; it's not a process — it's not even really a **word**. It's a cultural movement that rests on an argument: code isn't really serving its purpose until it's been **delivered**. This is something that both developers and operators can agree on.

Another thing they can agree on is that the reason code **isn't** delivered is because of fear. "It's not ready," mutters a Reasonable Operator. "Why not?!" the developer shouts, a vein pulsing in their forehead. "Well," explains the Reasonable Operator, "it's a big change. We're going to need to make sure it doesn't screw anything up."

This operator is indeed quite reasonable. Developers love making big changes. But big changes mean big risks — these are the bane of operational stability. Developers also recognize the inherent problems of pushing too much code at once: it's harder to find bugs in huge haystacks of code.

Without continuous delivery, developers and operators are equally stressed by the same thing — the act of deploying. And the **purpose** of continuous delivery is to make the act of deploying so painless that no one notices. How many times did you blink while reading this page? That's right, you don't know because it doesn't hurt when you blink.

That is the whole point of continuous delivery. This reduction in friction is what brings developers and operators together. It's a healthier kind of **MAD** (mutually assured deploys), and one that tempers the once-raging tempers of both parties. DevOps is the symbolic merging of two archetypes that were once throttling each other's throats.

Continuous Deployment

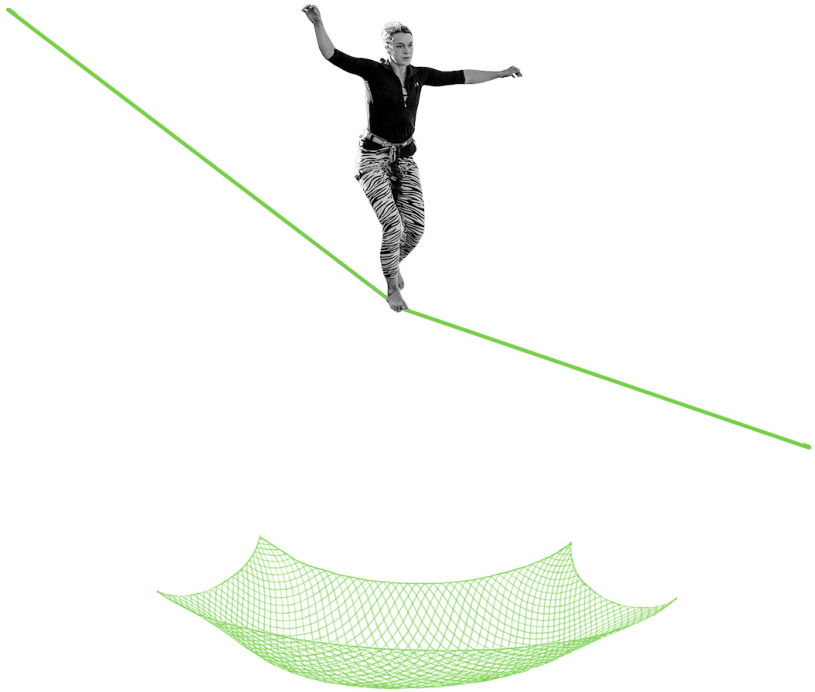
So, how far can we take the concept of being continuous? And what else can be automated?

Even in the wonderful world of Continuous Delivery, changes are still manually approved before launching into the choppy waters of Reality Ocean! But: if you've **truly** done your homework and created a robust testing framework, imagine the exhilaration of **automatically** pushing your code out to sea whenever the tests pass!

Nautical metaphors aside, we've been stressing that big changes require small changes. So if developers actually **committed** to committing small changes, there (theoretically) shouldn't be much harm in automatically deploying changes that pass all tests. In other words, if every deploy is already a tire fire, a smaller tire fire will be easier to extinguish.

Continuous deployment might seem extreme and dangerous. And in its purest form, it is — which is why it really only exists as an abstract concept. Very few companies can practice proper continuous deployment because it's hard to pull off. Teams usually need some amount of breathing room between development and deploys, especially for sensitive changes like database migrations.

Continuous deployment also encourages other kinds of continuous action, like analytics, error-reporting, and testing. Rollbar ties code changes to error frequency and key organizational metrics. PagerDuty shares the operations load



across the team and makes error resolution a little less hectic. LaunchDarkly champions feature flags to incrementally test and ship value.

Overall, there really aren't many technical differences separating continuous delivery from continuous deployment; it's more a matter of **culture** and **mindset**. Those in the deployment camp just happen to be a little more comfortable with regularly stepping off precipices — because they know that there's **another** precipice just below them.

If they fall, it's not going to be that far.

Conclusion: it's (Still) the Future

There's a reason you haven't seen very many dates in this last chapter. And it's because these changes are happening **right now**. Sure, there are leaders spearheading the movement and companies offering these sorts of services to make the continuous transition more continuous.

If you find all these continuous methodologies hard to tell apart, that's fine. At a high level, they can all happily live under the umbrella term of "DevOps". Together, they reflect a trend towards agility, speed, and a grand reduction in feedback loops between maker and consumer.

This "Brief History", as it turns out, is not so brief. And it's not done yet, which means we can't sew the events of these chapters into a nice quilt, lay it on a bed, and be done with it. No, this has been a recap of The Story Thus Far and, like most things in the software industry, it will be out of date in a few short months.

There are two ways to respond to this world of constant flux: you can stick your head in the sand and hope the changes disappear. Or you can lean in, abandon your perfectionism, and thrive. As the wall between developers and operators crumbles further, these groups will learn lessons from one another: developers will think about performant code, and operators will grease the wheels of deployment.

It's impossible to predict the future, so focus on understanding the present. This is the belief behind the DevOps movement, the monitoring services, the constant shipping. it's quickly becoming **the** way to effect meaningful change. This brief history is ongoing, and everyone has a chance to become an author.

> "I know that my apprehensions might never be allayed, and so I close, realizing that perhaps, the ending has not yet been written.
>
> Atrus (commenting on the DevOps movement)
> from the game Myst, 1993

Sources

https://aws.amazon.com/devops/what-is-devops/

http://www.base36.com/2012/12/agile-waterfall-methodologies-a-side-by-side-comparison/

https://www.techrepublic.com/article/understanding-the-pros-and-cons-of-the-waterfall-model-of-software-development/ https://en.wikipedia.org/wiki/Waterfall_model

http://www-scf.usc.edu/~csci201/lectures/Lecture11/royce1970.pdf

https://arstechnica.com/information-technology/2014/08/how-microsoft-dragged-its-development-practices-into-the-21st-century/

https://www.joelonsoftware.com/2008/05/01/architecture-astronauts-take-over/

https://en.wikipedia.org/wiki/Agile_software_development

http://agilemanifesto.org/

https://en.wikipedia.org/wiki/James_Martin_(author)

http://www.extremeprogramming.org/

https://www.scrum.org/resources/what-is-scrum

http://www.nceclusters.no/globalassets/filer/nce/diverse/the-pragmatic-programmer.pdf

http://www.agile247.pl/wp-content/uploads/2017/04/versionone-11th-annual-state-of-agile-report.pdf

https://www.agilealliance.org/

https://blog.dandyer.co.uk/2008/05/09/why-are-you-still-not-using-hudson/

https://www.infoworld.com/article/2624074/application-development/oracle-hands-hudson-to-eclipse--but-jenkins-fork-seems-permanent.html

https://mashable.com/2014/04/30/facebooks-new-mantra-move-fast-with-stability/#LWUcLYFRDPq4

http://www.cvauni.edu.vn/imgupload_dinhkem/file/pttkht/object-oriented-analysis-and-design-with-applications-2nd-edition.pdf

https://martinfowler.com/articles/continuousIntegration.html

https://shebanator.com/2007/08/21/a-brief-history-of-test-frameworks/

https://martinfowler.com/books/continuousDelivery.html

https://itrevolution.com/book/the-phoenix-project/

A Brief History of DevOps by Alek Sharma. Published by CircleCI, San Francisco CA.

www.circleci.com

Cover art, layout, and illustrations by Alex Moran.
Edited by Gillian BenAry.