"Continuous delivery" in software is more than a buzz phrase. Done right, continuous delivery of software is the holy grail of software development practice, customer retention, and it's the reason DevOps is such a hot concept today. To understand the importance of continuous delivery, you need to know something about the history of agility and where agile methods came from.

Agile software development history doesn't begin with the Agile Manifesto—its roots go back much earlier. This article covers a three-decades+ evolution in software development practice, including the origins of agile and how more recent knowledge is leading us to faster and faster deliver cycles.

**First came the crisis**

In the early 1990s, as PC computing began to proliferate in the enterprise, software development faced a crisis. At the time, it was widely referred to as "the application development crisis," or "application delivery lag." Industry experts estimated that the time between a validated business need and an actual application in production was about three years.

The problem was, businesses moved faster than that, even 25 years ago. Within the space of three years, requirements, systems, and even entire businesses were likely to change. That meant that many projects ended up being cancelled partway through, and many of those that were completed didn't meet all the business's current needs, even if the project's original objectives were met.

In certain industries, the lag was far greater than three years. In aerospace and defense, it could be 20 or more years before a complex system went into actual use. In an extreme but by no means unusual example, the Space Shuttle program, which operationally launched in 1982, used information and processing technologies from the 1960s. Highly complicated hardware and software systems were often designed, developed, and deployed in a time frame that spanned decades.

**Thought leaders were frustrated**

Jon Kern, an aerospace engineer in the 1990s, became increasingly frustrated with these long lead times and with the decisions made early in a project that couldn't be changed later. "We were looking for something that was more timely and responsive," he notes, joining a growing number of those who felt that there had to be a better way to build software. He was one of 17 software thought leaders who started meeting informally and talking about ways to develop software more simply, without the process and documentation overhead of waterfall and other popular software engineering techniques of the time.

Other industries were also undergoing transformation. It took the automotive industry six years or more to design a new car, and in the 1990s, that time was cut almost in half. AT&T had been broken up, and the so-called Baby Bells were drastically cutting the costs for phones and service.

For those products with a software development component, such as phone switches, autos, or aircraft, software was often an afterthought, mostly because software development didn't start until the hardware design was fixed in place. But building the software wasn't a priority for most product teams at the time.

**And agile was born**

These frustrations around seemingly unproductive software development activities, which were shared by like-minded professionals, led to the now-famous Snowbird meeting in Utah in early 2001. But that wasn't the first
time this particular group of software leaders had met. They had gathered the year before, at the Rogue River Lodge in Oregon in the spring of 2000.

This group included Kern, Extreme Programming pioneers Kent Beck and Ward Cunningham, Arie van Bennekum, Alistair Cockburn, and twelve others, all well known today in the agile community. Agile, as a practice, was not the ultimate goal; in fact, "agile" had yet to be used in formal conversation before that time. At that meeting, the terms "light" and "lightweight" were more common, although none of the participants were particularly satisfied with that description.

In particular, these thought leaders sought ways to quickly build working software and get it into the hands of end users. This fast delivery approach provided a couple of important benefits. First, it enabled users to get some of the business benefits of the new software faster. Second, it enabled the software team to get rapid feedback on the software's scope and direction.

Rapid feedback and willingness to change turned out to be the key features of the agile movement. If the software team isn't confident in understanding what the user needs, it delivers a first approximation and then listens to feedback. But little is set in stone at the beginning of the project.

A backlash against heavyweight processes
Agile is by no means critical of development methodologies developed in the 1970s and 1980s in response to the chaotic and unplanned approaches often used in the early days of software. In fact, 1970 to 1990 was largely when foundational theories and practices of software engineering came into being. The idea was to equate software engineering with physical engineering and borrow as much as possible from the design and building actual.

This approach manifested itself in what has become known as the waterfall methodology. This approach clearly defined major phases of the application development lifecycle, from requirements to deployment. It was termed "waterfall" because teams complete one step, fully, before moving on to the next. Requirements must be complete before moving on to functional design, functional design complete before detailed design, and so on through the sequence. And like water not flowing uphill, there are rarely provisions to return to an earlier stage of the process. Once you were finished with a stage, that stage was frozen in time.

This method brought a sense of organization and engineering practice to software development. But there is a key difference. Projects in civil or mechanical engineering rarely change over the course of a decade or more. If you need to design a bridge or a high-rise building today, it's very likely that the specifics won't require modification in a year or two.

In fact, waterfall as originally conceived was supposed to accommodate change and reconsideration of project decisions. There was accommodation for going back to the previous stage, and adjusting some of the decisions and expectations, and those changes could change aspects of the current stage. But in practice, schedules and budgets almost always made that impossible, forcing teams to stick with earlier decisions.

At the time, it was taken as gospel in most software development groups and university computer science departments that the more time you spent planning, the less time you would spend writing code, and the better that code would be. This only reinforced a process-heavy approach, which put more emphasis on planning and documentation than delivering working software.

We beg to disagree
Software projects rarely have the same kind of stability as traditional engineering projects. Business needs change, seemingly overnight, and certainly faster than the months or years formerly required to complete a software application. In retrospect, it seems obvious that software requires a different approach to engineering.
Of course, the other part of the problem is that software design is both a science and an art, with imperfections and associated human limitations. First, we simply don't know how to define software very well. Users can describe their business workflows, but they can't tell software designers what features will automate it and how those features should work. Our inability to precisely define what is needed before starting to build the product separates software engineering from most other engineering disciplines.

Second, the translation from requirements, imperfect as they are, to specifications, and from specifications to implementation, is rife with ambiguities. Some of that comes from the nature of the written word; if a statement can be misinterpreted, it almost certainly will be. But because teams are reading at a design level and translating it to an implementation level, mistakes and misunderstandings are bound to occur.

**Visionaries have different views of software development**

Some of the backlash was also driven by the largest software developer in the world: the US government. In particular, the Department of Defense (DoD) standards for software development (in particular, DOD-STD-2167) clearly favored the waterfall model until the late 1990s, when they were changed to explicitly support iterative processes.

Eventually, the DoD even designed its own programming language: Ada, named after Ada Lovelace, often referred to as the first computer programmer. An uncommonly large and highly structured language, it seemed to demand a heavyweight process, with a lot of documentation. It was perfect for the era of the highly documented and planned waterfall process.

Even as the waterfall model predominated in the 1980s and 1990s, industry and academic thought leaders were pushing back. An important initial response was James Martin with rapid application development, or RAD. The purpose of RAD was to reduce the preparation stage and to quickly get into development, so the business could begin to collaborate right away with the development team by seeing a working prototype in just a few days or weeks.

There was also a move toward so-called iterative software development. While iterative software development has its roots in at least the 1960s, the concept of incremental improvement had taken hold through the work of quality guru W. Edwards Deming and others even earlier. The concept is that you perform an activity, measure essential characteristics, make common-sense changes, and measure again for improvement.

Throughout the 1980s, software visionaries such as Gerald Weinberg, Fred Brooks, and Grady Booch wrote highly popular books and articles on iterative development techniques. Fred Brooks, well-known author of *The Mythical Man-Month*, wrote an article called "No Silver Bullet—Essence and Accidents of Software Engineering," in which he notes that there's no singular technique or process that will bring about significant improvements in software development productivity.

Probably the most recognized work on iterative development of the 1980s was Barry Boehm's "A Spiral Model of Software Development and Enhancement." The spiral model was a specific iterative technique whereby a project starts small and gradually grows as more features and capabilities are built into it. While major and highly visible projects often used a strict waterfall model, alternatives were lurking in the background.

**Practitioners want to iterate development**

At the same time, more specific iterative methodologies were being developed. For example, Jeff Sutherland and Ken Schwaber conceived the scrum process in the early 1990s. The term came from rugby and referred to a team working toward a common goal. They codified scrum in 1995 in order to present it at an object-oriented conference in Austin, Texas. They published it in the form of a paper titled "SCRUM Software Development Process."
Scrum was based on the concept that for the development of new, complex products, the best results occur when small and self-organizing teams are given objectives rather than specific assignments. The team had the freedom to determine the best way of meeting those objectives. Scrum also defined time-boxed iterative development cycles whose goal was to deliver working software. Today, most teams that claim to practice an agile methodology say they're using scrum.

At around the same time, Kent Beck was hired as a consultant on an experimental software development project at Chrysler. In time, he was named the project leader, and in an effort to succeed was determined to take best practices to an extreme level, giving the XP methodology its name. While the project was ultimately cancelled when Chrysler was acquired, Beck had published a book titled Extreme Programming Explained, and his name became synonymous with one of the leading alternative methodologies.

Perhaps various agile and iterative techniques would still be in the minority were it not for the Agile Manifesto, codified at that 2001 meeting in Snowbird. Despite the uncertain goals of this group, the Manifesto is the clearest and most succinct statement of purpose of an approach that was the antithesis of the waterfall model that was still prevalent at the time.

As a result, the software development community has latched onto the Agile Manifesto and its 12 principles as the definitive statement of the agile software development movement. Today, more and more teams self-identify as using an agile methodology. While many of those teams are likely using a hybrid model that includes elements of several agile methodologies as well as waterfall, that they identify so completely with the agile movement is a testament to both the strength of the statement and the power of the movement.

To agility, and beyond

And while agility got us to where we are, it's not the end of the story. There is life beyond agile, though agile was a necessary first step to see where software development might be able to venture. It may be foolish to demand a software change as quickly as our minds can see the need, but it isn't foolish to try speeding things up. Can agile concepts promote continuous, effective change in our software? Can we get to the point where a software "release," with all its improvements, is no longer an event to be planned for, but simply a daily, hourly, or minute-to-minute occurrence like breathing?

At the Better Software/Agile Development West conference in 2011, Kent Beck presented a keynote on how to accelerate software delivery. He gave this one-hour talk without ever once mentioning agile.

But there was no question as to what he was describing. He phrased it in terms of the trend toward releasing software to production more quickly, and discussed what it meant to be able to release new versions quarterly, monthly, weekly, and ultimately daily or even continuously. "If we don't know whether the feature is correct," he noted, "we deliver two versions of it, and let the users decide." This is the epitome of continuous delivery, or DevOps, as we might say today.

Agile processes are a necessary first step in that direction, but continuous delivery requires even more radical change. It means that developers try something based on their best knowledge at the time yet are fully prepared to remove or change it immediately based on user reaction.

And user reaction doesn't come in the form of words but rather in the form of actions. We monitor the application in production and collect data on user behavior. Real-time analysis of that data tells the team what to do next. And what happens next will take no more than a day or two.

https://techbeacon.com/agility-beyond-history%E2%80%94legacy%E2%80%94agile-development