AGILE

It’s an exciting time to be agile! For the first time, our industry has found a real, sustainable way to solve problems that generations of software development teams have been struggling with. Here are just a few of the solutions that agile promises:

- Agile projects come in on time, which is great for teams that have struggled with projects delivered very late and far over budget.
- Agile projects deliver high-quality software, which is a big change for teams that have been bogged down by buggy, inefficient software.
- Code built by agile teams is well constructed and highly maintainable, which should be a relief to teams accustomed to maintaining convoluted and tangled spaghetti code.
- Agile teams make their users happy, which is a huge change from software that fails to deliver value to the users.
- Best of all, developers on an effective agile team find themselves working normal hours, and are able to spend their nights and weekends with their friends and families—possibly for the first time in their careers.

  Agile is popular because many teams that have “gone agile” report great results: they build better software, work together better, satisfy their users, and do it all while hav- ing a much more relaxed and enjoyable working environment. Some agile teams seem to have finally made headway in fixing problems that have vexed software teams for decades. So how do great teams use agile to build better software? More specifically, how can *you* use agile to get results like this?

What Is Agile?

Agile is a **set of methods and methodologies** that help your team to think more effectively, work more efficiently, and make better decisions.

These methods and methodologies address all of the areas of traditional software engineering, including project management, software design and architecture, and process improvement. Each of those methods and methodologies consists of **practi- ces** that are streamlined and optimized to make them as easy as possible to adopt.

Agile is *also* a **mindset**, because the right mindset can make a big difference in how effectively a team uses the practices. This mindset helps people on a team share infor- mation with one another, so that they can make important project decisions together —instead of having a manager who makes all of those decisions alone. An agile mindset is about opening up planning, design, and process improvement to the entire team. An agile team uses practices in a way where everyone shares the same informa- tion, and each person on the team has a say in how the practices are applied.

The reality of agile for many teams that have not had as much success is quite differ- ent from its promise, and the key to that difference is often the mindset the team brings to each project. The majority of companies that build software have experi- mented with agile, and while many of them have found success, some teams have got- ten less-than-stellar results. They've achieved some improvement in how they run their projects—enough to make the effort to adopt agile worth it—but they haven't seen the substantial changes that they feel agile promised them. This is what that mindset shift is all about; "going agile" means helping the team find an effective mindset.

But what does "mindset shift" really mean? If you're on a software team, then what you do every day is plan, design, build, and ship software. What does "mindset" have to do with that? As it turns out, the practices you use to do your everyday work depend a lot on the attitude that you and your teammates have toward them.

Here's an example. One of the most common agile practices that teams adopt is called the **daily standup**, a meeting during which team members talk about what they're working on and their challenges. The meeting is kept short by making everyone stand for the duration. Many teams have had a lot of success adding a daily standup to their projects.

So imagine that a project manager is just learning about agile, and wants to add the daily standup to his project. To his surprise, not everyone on his team is as excited about this new practice as he is. One of his developers is angry that he's even suggest- ing that they add a new meeting, and seems to feel insulted by the idea of attending a meeting every day where he's asked prying questions about his day-to-day work.

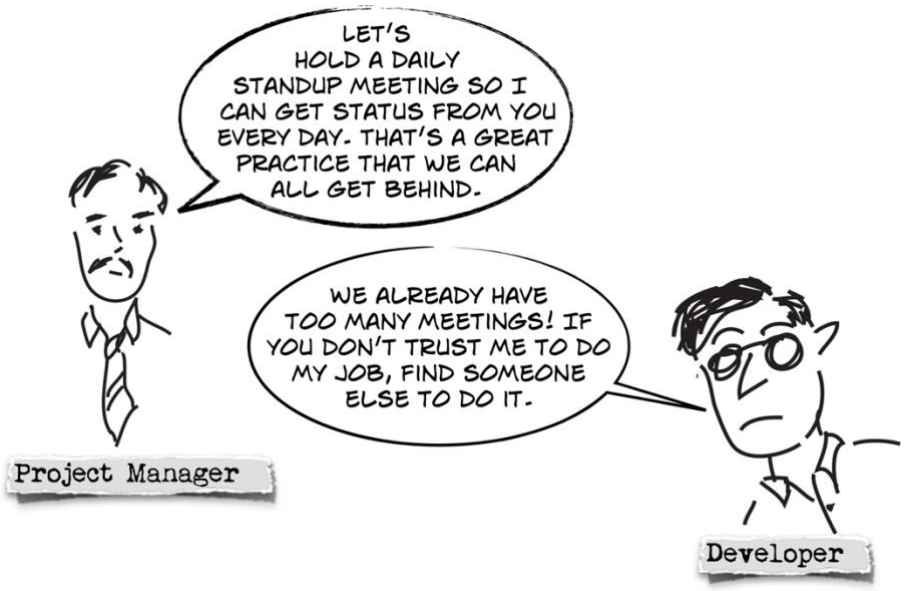


Figure: A project manager who wants to have the team start holding a daily standup is surprised that not everyone is immediately on board with it.

So what's going on here? Is the developer being irrational? Is the project manager being too demanding? Why is this simple, well-accepted practice causing a conflict?

Both the project manager and the developer have different—and valid—points of view. One of this project manager's biggest challenges is that he puts a lot of effort into planning the project, but when the team runs into problems building the soft- ware they deviate from the plan. He has to work very hard to stay on top of everyone on the team, so that he can make adjustments to the plan and help them deal with their problems.

The developer, on the other hand, feels like he's interrupted several times a day for meetings, which makes it very hard for him to get his work done. He already knows what he needs to do to get his own code built, and he doesn't need another person nagging him about plans and changes. He just wants to be left alone to code, and the last thing he wants is yet another meeting.

Both people seem to have a valid reason for their attitude toward the daily standup meeting. What effect will this have on the project?

Now imagine that the project manager is able to get everyone—even this reluctant developer—to start attending a daily standup meeting. What will that meeting look like? The project manager will be thinking mainly about how people are deviating from his plan, so he'll concentrate on getting status from each person. The developer, on the other hand, wants the meeting to end as quickly as possible, so he'll tune out everyone else while he's waiting to give his update, then say as little as possible when it's his turn, and hope that the whole thing ends quickly.

Let's be clear: this is how many daily standups are run, and while it's not optimal, a daily standup that's run this way *will still produce results*. The project manager will find out about problems with his plan, and the developer will benefit in the long run because those problems that *do* affect him can be taken care of sooner rather than

later—and the whole practice generally saves the team more time and effort than it costs. That makes it worth doing.

But what would happen *if the developer and the project manager had a different mind- set*? What if each person on the team approached the daily standup with an entirely different attitude?

For example, what would happen if the project manager felt like everyone on the team worked together to plan the project? Then the project manager would genuinely listen to each team member, not just to find out how they've deviated from *his* plan, but to understand how the plan *that everyone on the team worked together to create* might need to change. Instead of dictating a plan, handing it to the team, and meas- uring how well the team is following it, he's now working with the team to figure out the best way to approach the project—and the daily standup becomes a way to work together to make sure everyone is doing the most effective thing possible at any given time. As the facts of the project change each day, the team uses the daily standup to work together to make the most effective decisions they can. And because the team meets every day, changes that they discover in the meeting can be put into effect immediately so they don't waste a lot of time and effort going down the wrong path.

And what if the developer felt like this meeting wasn't just about giving status, but about understanding how the project is going, and coming together every day to find ways that everyone can work better? Then the daily standup *becomes important to him*. A good developer almost always has opinions not just about his own code, but about the whole direction of the project. The daily standup becomes his way to make sure that the project is run in a sensible, efficient way—and he knows that in the long run that will make his job of coding more rewarding, because the rest of the project is being run well. And he knows that when he brings up a problem with the plan during the meeting, everyone will listen, and the project will run better because of it.

Figure. When every person on the team feels like they have an equal hand in planning and running the project, the daily standup becomes more valuable—and much more effective.

In other words, if people on the team are in the mindset that the daily standup is sim- ply a status meeting that must be endured, it's still worth doing, but it's only slightly more effective than a traditional status meeting. But if everyone on the team shares the mindset that the daily standup is their way of making sure that everyone is on track, they're all working toward the same goal, and they all have a say in how the project is run, it becomes much more effective—and also more satisfying. The devel- oper has the attitude that this meeting helps him and his team in the long run. The project manager has the attitude that when each person on the team can contribute to the plan, the project gets better results. And when everyone shares these attitudes, the daily standup helps them all work faster, communicate more directly, and get things done more easily.

This is just one small example of how the mindset and attitude of the team have a big effect on how well they'll adopt agile practices. An important goal of this book is to help you understand how your own team's mindset affects your projects and your own agile adoption. By exploring Scrum, XP, Lean, and Kanban, you will learn both sides of the agile coin—principles and practices—and how they come together to help you build better software.

Understanding Agile Values

Agile, as a movement, is different from any approach to software development that came before it, because it started with ideas, values, and principles that embody a mindset. It's through the lens of these ideas that you can begin to become more agile as a practitioner, and more valuable as a member of your project team.

The agile movement is revolutionizing the world of software development. Teams that adopt agile have consistently reported improvements—sometimes huge leaps— in their ability to build great software. Teams that successfully adopt agile build better, higher quality software products, and they do it faster than before.

Our industry is at a turning point with agile. Agile has gone from being the underdog to becoming an institution. For the first few years of agile, people adopting it strug- gled to convince theCir companies and teammates that it worked, and that it was worth doing. Now, there is little question that agile development is a highly effective way to build software. In fact, in 2008, an important survey[1] found that more than half of all software teams surveyed were using agile methodologies, practices, or prin- ciples—and agile has only grown since then. And agile teams are increasingly going beyond the problem of how to be agile themselves, and are starting to figure out how to spread agile development throughout their companies.

But it wasn't always like this. Traditionally, companies have used a **waterfall process** when running their software projects, in which the team defines the requirements up front, plans the project in its entirety, designs the software, and then builds the code and tests the product. Plenty of software—great software, but also lousy software— has been built this way over the years. But as the decades went on, different teams in different companies kept running into the same kinds of problems...and some people started to suspect that a major source of project failure might be the waterfall process itself.

The story of agile started when a small group of innovative people got together to try to find a different way of thinking about these problems. The first thing they did was come up with a set of four core values that are common to successful teams and their projects, which they called the *Manifesto for Agile Software Development*:

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan

In this, you'll learn about these values—where they came from, what they mean, and how they apply to your project. You'll follow a waterfall-weary team through their first attempt at implementing agile before they really understand how those values apply to them. As you read their story, see if you can spot how a better understanding of these values could have helped them avoid their problems.

A Team Lead, Architect, and Project Manager Walk into a Bar...

Dan is a lead developer and architect at a company that makes coin-op games and kiosks. He's worked on projects ranging from arcade pinball machines to ATMs. For

the last few years, he's been working with a team lead, Bruce. They bonded on a release of the company's biggest product, a Vegas slot machine called the "Slot-o- matic Weekend Warrior."

Joanna was hired a few months ago as a project manager to head up a project build- ing software for a new line of streaming audio jukeboxes, which the company wants to introduce and sell to bars and restaurants. She was a real prize—they poached her from a competitor that already has a successful jukebox on the market. She's been get- ting along really well with Dan and Bruce, and she's excited about getting started on a new project with them.

Dan and Bruce are much less excited about the new project than Joanna. They all went out for drinks after work one day, and Bruce and Dan started explaining why

the team came up with their own name for the slot machine project: the "Slog-o-matic Weekend Killer."

She wasn't happy to learn that in this company, failing projects are the rule, not the exception. The last three projects were declared a success by the company's managers, but they only got out the door because Dan and Bruce worked incredibly long hours. Worse, they held their noses and took shortcuts in the code that are causing support nightmares today—like when they hastily patched up a prototype for one feature and pushed it out into production, and it later turned out to have serious performance problems because pieces of it were never built to scale up.

As they talked, Joanna recognized the pattern, and knew what was causing the prob- lem: the company follows a particularly ineffective *waterfall process*. A team following a waterfall process tries to write down as early as possible a complete description of the software that will be built. Once all of the users, managers, and executives agree on exactly what the software must do (the *requirements*), they can hand a document that contains those requirements (the *specification*) to a development team, and that team will go off and build exactly what's written. Then a testing team comes in after- ward, and verifies that the software matches the document. Many agile practitioners refer to this as "big requirements up front" (sometimes abbreviated BRUF).

But Joanna also knew from her own projects over the years that theory often differed from practice, and that while some teams have a very effective waterfall process, many of them struggle with it. She was starting to think that this team might be one of those that struggled.

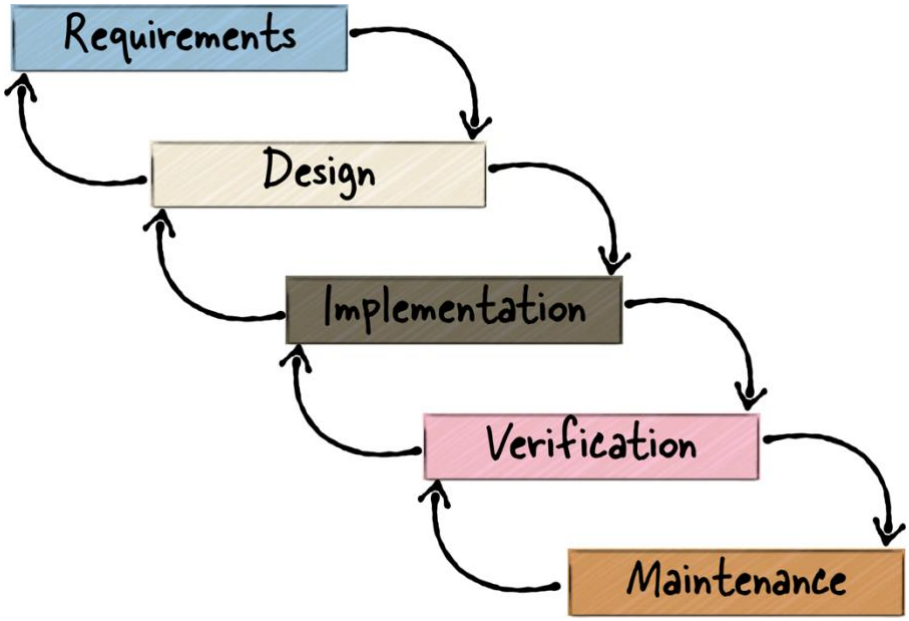


Figure. The waterfall model.

As they talked, Bruce and Dan confirmed a few things that reinforced her opinion. Just as Joanna suspected, there were a lot of specifications sitting in large binders and gathering dust on shelves all across the company. Somehow, everyone expected a group of users, managers, and executives to create a perfect requirements specifica- tion. In real life, the spec had a nasty habit of changing so that it would be inaccurate by the time the team got their hands on it, and would progress to being disastrously wrong by the time the team finished building the software. Bruce, Dan, and a lot of other people at the company knew that it was unreasonable to expect the perfect spec, but they still ran their projects as if it were possible.

As the evening went on, Bruce got more comfortable (and tipsy), and he brought up another problem that had been nagging him: that many teams he'd been on at their company had a lot of trouble actually building their software. Even if the users got the requirements right (which rarely happened), and even when the team understood those written requirements perfectly after reading them (which had yet to happen to this day), they often used inferior tools and had a lot of trouble with software design and architecture. This led to Bruce's teams repeatedly building software that had a lot of bugs, and was often an unmaintainable mess.

Both of these problems led to many failed projects, especially because they considered any project where they had to work many 90-hour weeks and deliver buggy code to be a failure. Joanna explained that the biggest cause of those failures was the *inability*

of the waterfall process the company followed to handle change. In a perfect world, the waterfall process would work just fine, because at the start of the project everyone would know exactly what they'd need at the end. That way, they could write it all down in a neat spec and hand it off to the team to build. But real-life projects never seemed to work out that way.

Dan and Bruce were now officially drunk, and deep into a marathon gripe session with Joanna. Dan told her that on almost every project that they'd worked on, the customers decided partway through that they needed the team to build something different than what they'd originally planned on. Then everyone had to go back to the beginning of the waterfall. According to the strict waterfall process the team was fol- lowing, they were supposed to write entirely new specifications, come up with a dif- ferent design, and build a whole new plan. But in reality, this almost never happened, and it was rare that they had time to throw out all of the code that had been written so far. Instead, they usually ended up hacking together a solution out of the existing code. This rework led to bugs, because taking software that was designed for one pur- pose and hastily modifying it to do something else often results in messy, tangled code—especially when the team is under pressure. Adapting it to the new use would have taken up precious project time, so they ended up with poor workarounds and brittle code.

What Dan, Bruce, and Joanna were starting to realize by the end of the night was that their project's problems were caused by *overly rigid documentation*, *poor communica- tion*, and *bugs*, which led to projects that could not keep up with normal project changes.

At the end of the evening, the bartender called taxis for all three of them. Just before they left, Dan said he was relieved to get a lot of that off of his chest. Joanna was happy to have a better picture of the project that she was joining...but a lot less opti- mistic. Will she be able to find a way, she wondered, to fix some of these problems?

The Solution

You probably know what a waterfall process feels like, even if you're just learning the term "waterfall" for the first time.[2] Joanna, Bruce, and Dan do, too. Before jumping into planning the jukebox project, they talked about how waterfall processes had caused problems for their teams in the past.

On their last project, Bruce and Dan worked with Tom, a customer account manager at the company, who spent a lot of time on the road helping customers at arcades, casinos, bars, and restaurants install and use their products. The three of them spent the first few weeks of the project building up a specification for the new slot machine. Tom was only in the office half the time, which gave Bruce and Dan time to start designing the software and planning the architecture. Once all three of them had agreed on the requirements, they called a big meeting with the CEO and senior man- agers of the company to review the requirements, make necessary changes, and get approval to start building.

At that point, Tom went back out on the road, leaving the work up to Bruce and Dan. They broke the project down into tasks, dividing them up among the team, and had everyone start building the software. When the team was almost done building the software, Tom gathered a group of business users, project managers, and executives into a big conference room to do a demo of the nearly complete Slot-o-matic Week- end Warrior software.

It didn't go nearly as smoothly as they'd expected.

At the demo, they had an awkward conversation where the CEO asked, "The software looks great, but wasn't it supposed have a video poker mode?" That was an unfortu- nate discovery. Apparently, the CEO was under the impression that they were work- ing on software that could be deployed to either the slot machine hardware or the video poker hardware. There had been a lot of discussion of this between the senior managers, the board, and the owners of their two biggest customers. Too bad nobody had bothered to tell the team.

The worst part about it was that if Dan had known about this earlier in the project, it wouldn't have been difficult to change direction. But now, they had to rip out

enormous amounts of code they'd already written, and replace it with code retrofitted from the video poker project. They spent weeks troubleshooting weird bugs that were caused by integration problems. Dan spent many late nights complaining to Bruce that this was 100% predictable, and that this almost always happens when code built

for one job is hastily repurposed to do something else. Now he's going to be stuck with a tangled mess of spaghetti code. It's going to be difficult to maintain this code- base, and the team is frustrated because it clearly didn't have to be this way.

And this wasn't just a problem for Dan and Bruce. The project manager for that project was so unhappy that he left the company. He had trusted the team's estimates and status, which were completely destroyed by the video poker change. The team had no idea that they would have to deal with the unexpected hardware change, but that didn't make the project manager's life any easier. Even though the facts on the ground had changed, the deadline was set in stone. By the time the project ended, the plan was out of date and basically useless, but the project manager was being held accountable for it anyway. After being raked over the coals by the senior managers in the company, all he could say in his defense is that his team didn't give good numbers to start with. Pretty soon, he left the company for another job—and Joanna was hired shortly after that.

Tom may have been the most frustrated of the group, because he was the one who had to face the customers when they ran into problems. Their biggest customer for this product was the Little Rock, a casino in Las Vegas that wanted to customize all of their slot machines with themes to match their reproductions of cities in Arkansas. The customer had requested this new feature so they could change games between shifts without having to move kiosks around. Their engineers kept running into bugs left in the software by Bruce's team, which meant that Tom and Dan had to spend weeks on the phone with the engineers coming up with patches and workarounds. The Little Rock didn't cancel any contracts, but their next big order was with a com- petitor, and it wasn't a secret that the CEO and managers blamed the Slot-o-matic Weekend Warrior project for the loss of business.

In the end, everyone knew that the project went wrong. And each person had a pretty good idea of how it was somebody else's fault. But nobody seemed to have any idea how to fix these types of recurring problems. And the software that they'd delivered was still a mess.

SCRUM – INTRODUCTION

Scrum is a framework within which people can address complex adaptive problems, while productively and creatively delivering products of the highest possible value.

Scrum itself is a simple framework for effective team collaboration on complex products. Scrum co-creators Ken Schwaber and Jeff Sutherland have written The Scrum Guide to explain Scrum clearly and succinctly. This Guide contains the definition of Scrum. This definition consists of Scrum's roles, events, artifacts, and the rules that bind them together.

Scrum is:

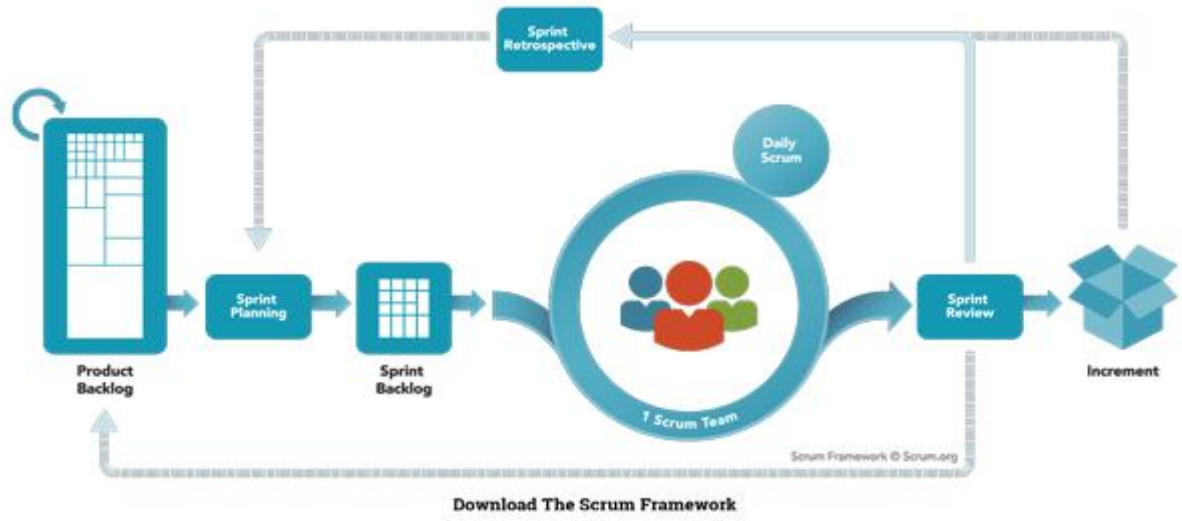
Lightweight

Simple to understand

Difficult to master

The Scrum Framework

Scrum is simple. It is the opposite of a big collection of interwoven mandatory components. Scrum is not a *methodology*. Scrum implements the scientific *method* of empiricism. Scrum replaces a programmed algorithmic approach with a heuristic one, with respect for people and self-organization to deal with unpredictability and solving complex problems. The below graphic represents Scrum in Action as described by Ken Schwaber and Jeff Sutherland in their book *Software in 30 Days* taking us from planning through software delivery.



Download The Scrum Framework

The Scrum Values

Although always considered to be a part of Scrum and often written about, in July 2016, the Scrum Values were added to The Scrum Guide. These values include Courage, Focus, Commitment, Respect, and Openness. Read the Scrum Guide to learn more about these values, how they apply to Scrum and download this poster.

The Roles of the Scrum Team

The Scrum Team consists of a Product Owner, the Development Team, and a Scrum Master. Scrum Teams are self-organizing and cross-functional. Self-

organizing teams choose how best to accomplish their work, rather than being directed by others outside the team. Cross-functional teams have all competencies needed to accomplish the work without depending on others not part of the team. The team model in Scrum is designed to optimize flexibility, creativity, and productivity.

Where Does Your Current Role Fit?

Watch this webinar on The Truth About Job Titles in Scrum to learn more about how roles have evolved and where you may fit.

Scrum defines three roles, the Product Owner, Scrum Master and Development Team Member. But what happens if you have a different job title? It doesn't mean that you are out of luck or out of a job, in most cases it means the exact opposite with your job expanding to deliver more value in the Scrum Team. So, where do you fit in Scrum?

The Scrum Events

Prescribed events are used in Scrum to create regularity and to minimize the need for meetings not defined in Scrum. All events are time-boxed. Once a Sprint begins, its duration is fixed and cannot be shortened or lengthened. The remaining events may end whenever the purpose of the event is achieved, ensuring an appropriate amount of time is spent without allowing waste in the process. The Scrum Events are:

Sprint

Sprint Planning

Daily Scrum

Sprint Review

Sprint Retrospective

Scrum Artifacts

Scrum’s artifacts represent work or value to provide transparency and opportunities for inspection and adaptation. Artifacts defined by Scrum are specifically designed to maximize transparency of key information so that everybody has the same understanding of the artifact. The Scrum Artifacts are:

Product Backlog

Sprint Backlog

Increment

DevOps

DevOps and it's resulting technical, architectural, and cultural practices rep- resent a convergence of many philosophical and management movements. While many organizations have developed these principles independently, understanding that DevOps resulted from a broad stroke of movements, a phenomenon described by John Willis (one of the co-authors of this book) as the "convergence of DevOps," shows an amazing progression of thinking and improbable connections. There are decades of lessons learned from manu- facturing, high reliability organization, high-trust management models, and others that have brought us to the DevOps practices we know today.

DevOps is the outcome of applying the most trusted principles from the domain of physical manufacturing and leadership to the IT value stream. DevOps relies on bodies of knowledge from Lean, Theory of Constraints, the Toyota Production System, resilience engineering, learning organiza- tions, safety culture, human factors, and many others. Other valuable contexts that DevOps draws from include high-trust management cultures, servant leadership, and organizational change management. The result is world-class quality, reliability, stability, and security at ever lower cost and effort; and accelerated flow and reliability throughout the technology value stream, including Product Management, Development, QA, IT Op- erations, and Infosec.

While the foundation of DevOps can be seen as being derived from Lean, the Theory of Constraints, and the Toyota Kata movement, many also view DevOps as the logical continuation of the Agile software journey that began in 2001.

THE LEAN MOVEMENT

Techniques such as Value Stream Mapping, Kanban Boards, and Total Pro- ductive Maintenance were codified for the Toyota Production System in the 1980s. In 1997, the Lean Enterprise Institute started researching applications of Lean to other value streams, such as the service industry and healthcare.

Two of Lean's major tenets include the deeply held belief that *manufacturing lead time* required to convert raw materials into finished goods was the best predictor of quality, customer satisfaction, and employee happiness, and that one of the best predictors of short lead times was small batch sizes of work.

Lean principles focus on how to create value for the customer through systems thinking by creating constancy of purpose, embracing scientific thinking, creating flow and pull

(versus push), assuring quality at the source, leading with humility, and respecting every individual.

THE AGILE MANIFESTO

The Agile Manifesto was created in 2001 by seventeen of the leading thinkers in software development. They wanted to create a lightweight set of values and principles against heavyweight software development processes such as waterfall development, and methodologies such as the Rational Unified Process.

One key principle was to "deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale," emphasizing the desire for small batch sizes, incremental re- leases instead of large, waterfall releases. Other principles emphasized the need for small, self-motivated teams, working in a high-trust management model.

Agile is credited for dramatically increasing the productivity of many devel- opment organizations. And interestingly, many of the key moments in DevOps history also occurred within the Agile community or at Agile conferences, as described below.

AGILE INFRASTRUCTURE AND VELOCITY MOVEMENT

At the 2008 Agile conference in Toronto, Canada, Patrick Debois and Andrew Schafer held a "birds of a feather" session on applying Agile principles to infrastructure as opposed to application code. Although they were the only people who showed up, they rapidly gained a following of like-minded thinkers, including co-author John Willis.

Later, at the 2009 Velocity conference, John Allspaw and Paul Hammond gave the seminal "10 Deploys per Day: Dev and Ops Cooperation at Flickr" presen- tation, where they described how they created shared goals between Dev and Ops and used continuous integration practices to make deployment part of everyone's daily work. According to first hand accounts, everyone attending the presentation immediately knew they were in the presence of something profound and of historic significance.

Patrick Debois was not there, but was so excited by Allspaw and Hammond's idea that he created the first DevOpsDays in Ghent, Belgium, (where he lived) in 2009. There the term "DevOps" was coined.

THE CONTINUOUS DELIVERY MOVEMENT

Building upon the development discipline of continuous build, test, and integration, Jez Humble and David Farley extended the concept to *continuous delivery*, which defined the role of a "deployment pipeline" to ensure that code and infrastructure are always in a deployable state, and that all code checked in to trunk can be safely deployed into production. This idea was first presented at the 2006 Agile conference, and was also independently developed in 2009 by Tim Fitz in a blog post on his website titled "Continuous Deployment."[†]

TOYOTA KATA

In 2009, Mike Rother wrote *Toyota Kata: Managing People for Improvement, Adaptiveness and Superior Results*, which framed his twenty-year journey to understand and codify the Toyota Production System. He had been one of the graduate students who flew with GM executives to visit Toyota plants and helped develop the Lean toolkit, but he was puzzled when none of the com- panies adopting these practices replicated the level of performance observed at the Toyota plants.

He concluded that the Lean community missed the most important practice of all, which he called the *improvement kata*. He explains that every organization has work routines, and the improvement kata requires creating structure for the daily, habitual practice of improvement work, because daily practice is what improves outcomes. The constant cycle of establishing desired future states, setting weekly target outcomes, and the continual improvement of daily work is what guided improvement at Toyota.

The above describes the history of DevOps and relevant movements that it draws upon. Throughout the rest of Part I, we look at value streams, how Lean principles can be applied to the technology value stream, and the Three Ways of Flow, Feedback, and Continual Learning and Experimentation.

Agile, Continuous Delivery, and the Three Ways

an introduction to the underpinning theory of Lean Manu- facturing is presented, as well as the Three Ways, the principles from which all of the observed DevOps behaviors can be derived.

Our focus here is primarily on theory and principles, describing many decades of lessons learned from manufacturing, high-reliability organizations, high- trust management models, and others, from which DevOps practices have been derived.

THE MANUFACTURING VALUE STREAM

One of the fundamental concepts in Lean is the *value stream*. We will define it first in the context of manufacturing and then extrapolate how it applies to DevOps and the technology value stream.

Karen Martin and Mike Osterling define value stream in their book *Value Stream Mapping: How to Visualize Work and Align Leadership for Organiza- tional Transformation* as "the sequence of activities an organization under- takes to deliver upon a customer request," or "the sequence of activities re- quired to design, produce, and deliver a good or service to a customer, including the dual flows of information and material."

In manufacturing operations, the value stream is often easy to see and observe: it starts when a customer order is received and the raw materials are released onto the plant floor. To enable fast and predictable lead times in any value stream, there is usually a relentless focus on creating a smooth and even flow of work, using techniques such as small batch sizes, reducing work in process (WIP), preventing rework to ensure we don't pass defects to downstream work centers, and constantly optimizing our system toward our global goals.

THE TECHNOLOGY VALUE STREAM

The same principles and patterns that enable the fast flow of work in physical processes are equally applicable to technology work (and, for that matter, for all knowledge work). In DevOps, we typically define our technology value stream as the process required to convert a business hypothesis into a technology- enabled service that delivers value to the customer.

The input to our process is the formulation of a business objective, concept, idea, or hypothesis, and starts when we accept the work in Development, adding it to our committed backlog of work.

From there, Development teams that follow a typical Agile or iterative process will likely transform that idea into user stories and some sort of feature specification, which is then implemented in code into the application or service being built. The code is then checked

in to the version control repository, where each change is integrated and tested with the rest of the software system.

Because value is created only when our services are running in production, we must ensure that we are not only delivering fast flow, but that our deploy- ments can also be performed without causing chaos and disruptions such as service outages, service impairments, or security or compliance failures.

FOCUS ON DEPLOYMENT LEAD TIME

Our attention will be on deployment lead time, a subset of the value stream described above. This value stream begins when any engineer† in our value stream (which includes Development, QA, IT Operations, and Infosec) checks a change into version control and ends when that change is successfully running in production, providing value to the customer and generating useful feedback and telemetry.

The first phase of work that includes Design and Development is akin to Lean Product Development and is highly variable and highly uncertain, often re- quiring high degrees of creativity and work that may never be performed again, resulting in high variability of process times. In contrast, the second phase of work, which includes Testing and Operations, is akin to Lean Man- ufacturing. It requires creativity and expertise, and strives to be predictable and mechanistic, with the goal of achieving work outputs with minimized variability (e.g., short and predictable lead times, near zero defects).

Instead of large batches of work being processed sequentially through the design/development value stream and then through the test/operations value stream (such as when we have a large batch waterfall process or long-lived feature branches), our goal is to have testing and operations happening simultaneously with design/development, enabling fast flow and high quality. This method succeeds when we work in small batches and build quality into every part of value stream.

Defining Lead Time vs. Processing Time

In the Lean community, lead time is one of two measures commonly used to measure performance in value streams, with the other being processing time

Whereas the lead time clock starts when the request is made and ends when it is fulfilled, the process time clock starts only when we begin work on the customer request—specifically, it omits the time that the work is in queue, waiting to be processed

The Common Scenario: Deployment Lead Times Requiring Months

In business as usual, we often find ourselves in situations where our deploy- ment lead times require months. This is especially common in large, complex organizations that are working with tightly-coupled, monolithic applications, often with scarce integration test environments, long test and production environment lead times, high reliance on manual testing, and multiple required approval processes.

When we have long deployment lead times, heroics are required at almost every stage of the value stream. We may discover that nothing works at the end of the project when we merge all the development team's changes together, resulting in code that no longer builds correctly or passes any of our tests. Fixing each problem requires days or weeks of investigation to determine who broke the code and how it can be fixed, and still results in poor customer outcomes.

Our DevOps Ideal: Deployment Lead Times of Minutes

In the DevOps ideal, developers receive fast, constant feedback on their work, which enables them to quickly and independently implement, integrate, and validate their code, and have the code deployed into the production environment (either by deploying the code themselves or by others).

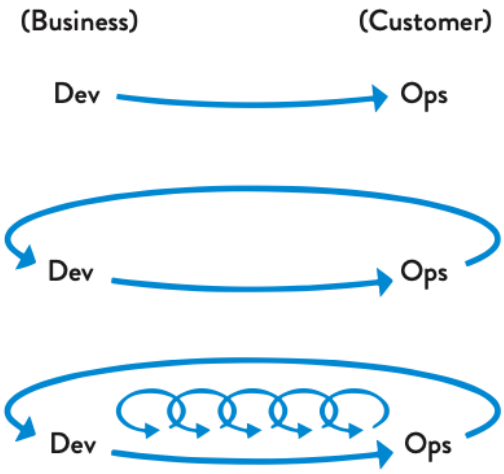
We achieve this by continually checking small code changes into our version control repository, performing automated and exploratory testing against it, and deploying it into production. This enables us to have a high degree of confidence that our changes will operate as designed in production and that any problems can be quickly detected and corrected.

This is most easily achieved when we have architecture that is modular, well encapsulated, and loosely-coupled so that small teams are able to work with high degrees of autonomy, with failures being small and contained, and without causing global disruptions.

THE THREE WAYS: THE PRINCIPLES UNDERPINNING DEVOPS

The Phoenix Project presents the Three Ways as the set of underpinning prin- ciples from which all the observed DevOps behaviors and patterns are derived

The First Way enables fast left-to-right flow of work from Development to Operations to the customer. In order to maximize flow, we need to make work visible, reduce our batch sizes and intervals of work, build in quality by pre- venting defects from being passed to downstream work centers, and constantly optimize for the global goals.



By speeding up flow through the technology value stream, we reduce the lead time required to fulfill internal or customer requests, especially the time re- quired to deploy code into the production environment. By doing this, we increase the quality of work as well as our throughput, and boost our ability to out-experiment the competition.

The resulting practices include continuous build, integration, test, and de- ployment processes; creating environments on demand; limiting work in process (WIP); and building systems and organizations that are safe to change.

The Second Way enables the fast and constant flow of feedback from right to left at all stages of our value stream. It requires that we amplify feedback to prevent problems from happening again, or enable faster detection and recovery. By doing this, we create quality at the source and generate or embed knowledge where it is needed—this allows us to create ever-safer systems of work where problems are found and fixed long before a catastrophic failure occurs.

By seeing problems as they occur and swarming them until effective counter- measures are in place, we continually shorten and amplify our feedback loops, a core tenet of virtually all modern process improvement methodologies. This maximizes the opportunities for our organization to learn and improve.

The Third Way enables the creation of a generative, high-trust culture that supports a dynamic, disciplined, and scientific approach to experimentation

and risk-taking, facilitating the creation of organizational learning, both from our successes and failures. Furthermore, by continually shortening and amplifying our feedback loops, we create ever-safer systems of work and are better able to take risks and perform experiments that help us learn faster than our competition and win in the marketplace.

As part of the Third Way, we also design our system of work so that we can multiply the effects of new knowledge, transforming local discoveries into global improvements. Regardless of where someone performs work, they do so with the cumulative and collective experience of everyone in the organization.

What is cloud computing?

Simply put, cloud computing is the delivery of computing services—including servers, storage, databases, networking, software, analytics, and intelligence—over the Internet ("the cloud") to offer faster innovation, flexible resources, and economies of scale. You typically pay only for cloud services you use, helping lower your operating costs, run your infrastructure more efficiently and scale as your business needs change.

Top benefits of cloud computing

Cost

Cloud computing eliminates the capital expense of buying hardware and software and setting up and running on-site datacenters—the racks of servers, the round-the-clock electricity for power and cooling, the IT experts for managing the infrastructure. It adds up fast.

Speed

Most cloud computing services are provided self service and on demand, so even vast amounts of computing resources can be provisioned in minutes, typically with just a few mouse clicks, giving businesses a lot of flexibility and taking the pressure off capacity planning.

Global scale

The benefits of cloud computing services include the ability to scale elastically. In cloud speak, that means delivering the right amount of IT resources—for example, more or less computing power, storage, bandwidth—right when it is needed and from the right geographic location.

Productivity

On-site datacenters typically require a lot of "racking and stacking"—hardware setup, software patching, and other time-consuming IT management chores. Cloud computing removes the need for many of these tasks, so IT teams can spend time on achieving more important business goals.

Performance

The biggest cloud computing services run on a worldwide network of secure datacenters, which are regularly upgraded to the latest generation of fast and efficient computing hardware. This offers several benefits over a single corporate datacenter, including reduced network latency for applications and greater economies of scale.

Reliability

Cloud computing makes data backup, disaster recovery and business continuity easier and less expensive because data can be mirrored at multiple redundant sites on the cloud provider's network.

Security

Many cloud providers offer a broad set of policies, technologies and controls that strengthen your security posture overall, helping protect your data, apps and infrastructure from potential threats.

Types of cloud computing

Not all clouds are the same and not one type of cloud computing is right for everyone. Several different models, types and services have evolved to help offer the right solution for your needs.

First, you need to determine the type of cloud deployment or cloud computing architecture, that your cloud services will be implemented on. There are three different ways to deploy cloud services: on a public cloud, private cloud or hybrid cloud.

Public cloud

Public clouds are owned and operated by a third-party cloud service providers, which deliver their computing resources like servers and storage over the Internet. Microsoft Azure is an example of a public cloud. With a public cloud, all hardware, software and other supporting infrastructure is owned and managed by the cloud provider. You access these services and manage your account using a web browser.

Private cloud

A private cloud refers to cloud computing resources used exclusively by a single business or organisation. A private cloud can be physically located on the company's on-site datacenter. Some companies also pay third-party service

providers to host their private cloud. A private cloud is one in which the services and infrastructure are maintained on a private network.

Hybrid cloud

Hybrid clouds combine public and private clouds, bound together by technology that allows data and applications to be shared between them. By allowing data and applications to move between private and public clouds, a hybrid cloud gives your business greater flexibility, more deployment options and helps optimise your existing infrastructure, security and compliance.

Types of cloud services: IaaS, PaaS, serverless and SaaS

Most cloud computing services fall into four broad categories: infrastructure as a service (IaaS), platform as a service (PaaS), serverless and software as a service (SaaS). These are sometimes called the cloud computing stack because they build on top of one another. Knowing what they are and how they are different makes it easier to accomplish your business goals.

Infrastructure as a service (IaaS)

The most basic category of cloud computing services. With IaaS, you rent IT infrastructure—servers and virtual machines (VMs), storage, networks, operating systems—from a cloud provider on a pay-as-you-go basis.

Platform as a service (PaaS)

Platform as a service refers to cloud computing services that supply an on-demand environment for developing, testing, delivering and managing software applications. PaaS is designed to make it easier for developers to quickly create web or mobile apps, without worrying about setting up or managing the underlying

infrastructure of servers, storage, network and databases needed for development.

Serverless computing

Overlapping with PaaS, serverless computing focuses on building app functionality without spending time continually managing the servers and infrastructure required to do so. The cloud provider handles the setup, capacity planning and server management for you. Serverless architectures are highly scalable and event-driven, only using resources when a specific function or trigger occurs.

Software as a service (SaaS)

Software as a service is a method for delivering software applications over the Internet, on demand and typically on a subscription basis. With SaaS, cloud providers host and manage the software application and underlying infrastructure and handle any maintenance, like software upgrades and security patching. Users connect to the application over the Internet, usually with a web browser on their phone, tablet or PC.

Uses of cloud computing

You are probably using cloud computing right now, even if you don't realise it. If you use an online service to send email, edit documents, watch movies or TV, listen to music, play games or store pictures and other files, it is likely that cloud computing is making it all possible behind the scenes. The first cloud computing services are barely a decade old, but already a variety of organisations—from tiny startups to global corporations, government agencies to non-profits—are embracing the technology for all sorts of reasons.

Create cloud-native applications

Quickly build, deploy and scale applications—web, mobile and API. Take advantage of cloud-native technologies and approaches, such as

containers, Kubernetes, microservices architecture, API-driven communication and DevOps.

Test and build applications

Reduce application development cost and time by using cloud infrastructures that can easily be scaled up or down, store, back up and recover data

Protect your data more cost-efficiently—and at massive scale—by transferring your data over the Internet to an offsite cloud storage system that is accessible from any location and any device.

Analyse data

Unify your data across teams, divisions and locations in the cloud. Then use cloud services, such as machine learning and artificial intelligence, to uncover insights for more informed decisions.

Stream audio and video

Connect with your audience anywhere, anytime, on any device with high-definition video and audio with global distribution.

Embed intelligence

Use intelligent models to help engage customers and provide valuable insights from the data captured.

Deliver software on demand

Also known as software as a service (SaaS), on-demand software lets you offer the latest software versions and updates around to customers—anytime they need, anywhere they are.

Benefits of version control

Version control systems are a category of software tools that help a software team manage changes to source code over time. Version control software keeps track of every modification to the code in a special kind of database. If a mistake is made, developers can turn back the clock and compare earlier versions of the code to help fix the mistake while minimizing disruption to all team members.

For almost all software projects, the source code is like the crown jewels - a precious asset whose value must be protected. For most software teams, the source code is a repository of the invaluable knowledge and understanding about the problem domain that the developers have collected and refined through careful effort. Version control protects source code from both catastrophe and the casual degradation of human error and unintended consequences.

Software developers working in teams are continually writing new source code and changing existing source code. The code for a project, app or software component is typically organized in a folder structure or "file tree". One developer on the team may be working on a new feature while another developer fixes an unrelated bug by changing code, each developer may make their changes in several parts of the file tree.

Version control helps teams solve these kinds of problems, tracking every individual change by each contributor and helping prevent

concurrent work from conflicting. Changes made in one part of the software can be incompatible with those made by another developer working at the same time. This problem should be discovered and solved in an orderly manner without blocking the work of the rest of the team. Further, in all software development, any change can introduce new bugs on its own and new software can't be trusted until it's tested. So testing and development proceed together until a new version is ready.

Good version control software supports a developer's preferred workflow without imposing one particular way of working. Ideally it also works on any platform, rather than dictate what operating system or tool chain developers must use. Great version control systems facilitate a smooth and continuous flow of changes to the code rather than the frustrating and clumsy mechanism of file locking - giving the green light to one developer at the expense of blocking the progress of others.

Software teams that do not use any form of version control often run into problems like not knowing which changes that have been made are available to users or the creation of incompatible changes between two unrelated pieces of work that must then be painstakingly untangled and reworked. If you're a developer who has never used version control you may have added versions to your files, perhaps with suffixes like "final" or "latest" and then had to later deal with a new final version. Perhaps you've commented out code blocks because you want to disable certain functionality without deleting the code, fearing that there may be a use for it later. Version control is a way out of these problems.

Version control software is an essential part of the every-day of the modern software team's professional practices. Individual software developers who are accustomed to working with a capable version control system in their teams typically recognize the incredible value version control also gives them even on small solo projects. Once accustomed to the powerful benefits of version control systems, many developers wouldn't consider working without it even for non-software projects.

Benefits of version control systems

Developing software without using version control is risky, like not having backups. Version control can also enable developers to move faster and it allows software teams to preserve efficiency and agility as the team scales to include more developers.

Version Control Systems (VCS) have seen great improvements over the past few decades and some are better than others. VCS are sometimes known as SCM (Source Code Management) tools or RCS (Revision Control System). One of the most popular VCS tools in use today is called Git. Git is a *Distributed* VCS, a category known as DVCS, more on that later. Like many of the most popular VCS systems available today, Git is free and open source. Regardless of what they are called, or which system is used, the primary benefits you should expect from version control are as follows.

A complete long-term change history of every file. This means every change made by many individuals over the years. Changes include the creation and deletion of files as well as edits to their

contents. Different VCS tools differ on how well they handle renaming and moving of files. This history should also include the author, date and written notes on the purpose of each change. Having the complete history enables going back to previous versions to help in root cause analysis for bugs and it is crucial when needing to fix problems in older versions of software. If the software is being actively worked on, almost everything can be considered an "older version" of the software.

Branching and merging. Having team members work concurrently is a no-brainer, but even individuals working on their own can benefit from the ability to work on independent streams of changes. Creating a "branch" in VCS tools keeps multiple streams of work independent from each other while also providing the facility to merge that work back together, enabling developers to verify that the changes on each branch do not conflict. Many software teams adopt a practice of branching for each feature or perhaps branching for each release, or both. There are many different workflows that teams can choose from when they decide how to make use of branching and merging facilities in VCS.

Traceability. Being able to trace each change made to the software and connect it to project management and bug tracking software such as Jira, and being able to annotate each change with a message describing the purpose and intent of the change can help not only with root cause analysis and other forensics. Having the annotated history of the code at your fingertips when you are reading the code, trying to understand what it is doing and why it is so designed can enable developers to make correct and

harmonious changes that are in accord with the intended long-term design of the system. This can be especially important for working effectively with legacy code and is crucial in enabling developers to estimate future work with any accuracy.

While it is possible to develop software without using any version control, doing so subjects the project to a huge risk that no professional team would be advised to accept. So the question is not whether to use version control but which version control system to use.

What is Jenkins?

Jenkins is a self-contained, open source automation server which can be used to automate all sorts of tasks related to building, testing, and delivering or deploying software.

Jenkins can be installed through native system packages, Docker, or even run standalone by any machine with a Java Runtime Environment (JRE) installed.

Jenkins is a popular open source tool to perform continuous integration and build automation. Jenkins allows to execute a predefined list of steps, e.g. to compile Java source code and build a JAR from the resulting classes. The trigger for this execution can be time or event based.

For example, you can compile your Java based application every 20 minutes or after a new commit in the related Git repository.

Possible steps executed by Jenkins are for example:

- perform a software build using a build system like Apache Maven or Gradle
- execute a shell script
- archive a build result
- run software tests

Jenkins monitors the execution of the steps and allows to stop the process, if one of the steps fails. Jenkins can also send out notifications in case of a build success or failure.

- Jenkins can be extended by additional plug-ins. For example, you can install plug-ins to support building and testing Android applications.

1.2. Using continues integration

- *Continuous integration* is a process in which all development work is integrated as early as possible. The resulting artifacts are automatically created and tested. This process allows to identify errors in an early stage of the project.
- The Jenkins build server is a tool to provide this functionality.

Installing Jenkins

The procedures on this page are for new installations of Jenkins on a single/local machine.

Jenkins is typically run as a standalone application in its own process with the built-in Java servlet container/application server (Jetty).

Jenkins can also be run as a servlet in different Java servlet containers such as Apache Tomcat or GlassFish. However, instructions for setting up these types of installations are beyond the scope of this page.

Note: Although this page focuses on local installations of Jenkins, this content can also be used to help set up Jenkins in production environments.

Prerequisites

Minimum hardware requirements:

- 256 MB of RAM
- 1 GB of drive space (although 10 GB is a recommended minimum if running Jenkins as a Docker container)

Recommended hardware configuration for a small team:

- 1 GB+ of RAM
- 50 GB+ of drive space

Comprehensive hardware recommendations:

- Hardware: see the Hardware Recommendations page

Software requirements:

- Java: see the Java Requirements page
- Web browser: see the Web Browser Compatibility page
- For Windows operating system: Windows Support Policy

Installation platforms

This section describes how to install/run Jenkins on different platforms and operating systems.

Docker

Docker is a platform for running applications in an isolated environment called a "container" (or Docker container). Applications like Jenkins can be downloaded as read-only "images" (or Docker images), each of which is run in Docker as a container. A Docker container is in effect a "running instance" of a Docker image. From this perspective, an image is stored permanently more or less (i.e. insofar as image updates are published), whereas containers are stored temporarily. Read more about these concepts in the Docker documentation's Getting Started, Part 1: Orientation and setup page.

Docker's fundamental platform and container design means that a single Docker image (for any given application like Jenkins) can be run on any supported operating system (macOS, Linux and Windows) or cloud service (AWS and Azure) which is also running Docker.

Installing Docker

To install Docker on your operating system, follow "prerequisits" section.

As an alternative solution you can visit the Docker store website and click the **Docker Community Edition** box which is suitable for your operating system or cloud service. Follow the installation instructions on their website.

Jenkins can also run on Docker Enterprise Edition, which you can access through **Docker EE** on the Docker store website.

Downloading and running Jenkins in Docker

There are several Docker images of Jenkins available.

The recommended Docker image to use is the jenkinsci/blueocean image (from the Docker Hub repository). This image contains the current Long-Term Support (LTS) release of Jenkins (which is production-ready) bundled with all Blue Ocean plugins and features. This means that you do not need to install the Blue Ocean plugins separately.