New Product Development Using Lean and Agile

If you are embarking on your first time using Agile as an engineer, or your first time working with a development team as an Entrepreneur, learning about this ahead of time is essential. There are quite a few concepts to immerse yourself in prior to actually doing the work. Building the product you've defined is obviously a critical step, and solid execution really matters here.

As you work through the Agile and Lean processes with your venture team, you will validate your target customer, their underserved needs, your value proposition, your MVP feature set, and your UX. As a result, you will feel confident about the blueprint you've developed. Validating product-market fit with prototypes is incredibly valuable, and once done, it’s time to turn your blueprint into an actual working product that customers can use.

There are many risks that could impede you while trying to build your blueprint. You may run into issues with technical feasibility, where what you've designed is impossible or too challenging to build, either in general, or with the resources you have available. Your product may be feasible but have such a large scope relative to your resources that it will just take too long to build. Good market opportunities only exist for so long before competition moves them to the upper right quadrant of the importance's vs. satisfaction framework.

An important part of product-market fit is having the right product at the right time (recall the product strategy discussion in the section on Define Your Value Proposition.) Even if you have an appropriate scope, poor execution can result in your actual product falling quite short of the promise of your prototype. You clearly want to minimize these types of risks—and the product development process you use can have a big impact on that. Agile Development is currently the best practice in product development to help teams deliver great products more quickly with less risk.

AGILE DEVELOPMENT

Just as you took an iterative approach to arrive at this point, you want to do the same in building your product. “Agile development” is the broad term used to describe a variety of iterative and incremental product development methodologies. Before the adoption of Agile development, most software products were built using the “waterfall” approach—one that proceeds sequentially through a series of steps. The team first defines all of the requirements, and then designs the product. They then implement the product, followed by testing to verify it works as intended. The key characteristic of waterfall is that the team does not progress to the next step until the previous step is 100 percent complete. In other words, no design happens until *all* of the requirements are defined, and no coding happens until the *entire* product is designed. Waterfall is also referred to as a “big design up front” (BDUF) approach.

In contrast, teams using Agile methodologies break the product down into smaller pieces that undergo shorter cycles of requirements definition, design, and coding. There are several benefits of Agile:

1. First, because you are planning in smaller increments, you can react to changes in the market or other new information more quickly.
2. Second, your product reaches customers earlier—which means that you start hearing feedback from customers on your actual product sooner, which helps guide your subsequent product development efforts.
3. Third, teams can reduce their margin of error in estimating scope by working in smaller batch sizes.

The Lean concept of small batch sizes (covered in the section on Specify Your MVP Feature Set and User Stories) described how it works, here we discuss why they are so beneficial in software development (or any development under conditions of high uncertainty).

When developers estimate the amount of time it will take them to implement new functionality, there is a degree of uncertainty in their estimated values. This uncertainty results in estimation errors where the actual duration differs from the estimated duration. A good way to compare the actual duration and the estimated duration is to take the ratio of the former to the latter. If a project took twice as long as expected, the ratio would be 2×; if it took half as long as expected, the ratio would be 0.5×.

Developers are not as likely to finish tasks early as they are to finish them late. Most of the time, software development tasks take *longer* than estimated. Industry research indicates that these late projects can typically run from 50% to 100% longer than estimated. And while it's true that some tasks do get completed early, the magnitude of positive surprises tends to be much smaller than the magnitude of negative surprises. Why is that? To help explain the asymmetric nature of software estimation errors, recall the quote from epistemologist and former Secretary of Defense Donald Rumsfeld:

There are known knowns. There are things we know we know. We also know there are known unknowns. That is to say, we know there are some things we do not know. But there are also unknown unknowns. The ones we don't know we don't know.

When developers are asked to estimate the effort for a task, they take into account the "known knowns." Skilled estimators will also account for the known unknowns in their estimates. It's true that some estimation error can come from an inaccurate understanding of the known knowns or the known unknowns. But the biggest wild cards in estimate after estimate are often the unknown unknowns, and they are what make the distribution of estimation errors asymmetric.

Let's say I estimate that task A will take me five minutes and task B will take me five months. Both tasks could have unknown unknowns. But the uncertainty is nonlinear with increasing scope, as the top curve of the cone of uncertainty suggests. The chances that the five-minute task will spiral out of control are pretty low. The five-month task is over 30,000 times larger in scope, which is a lot more room for unknown unknowns to hide.

When developers go through the thinking and investigation required to break a large task into smaller ones, they reduce the unknown unknowns by converting them into known unknowns. You can't completely escape unknown unknowns, but by using smaller batch sizes, you can rein them in to be more manageable and ship product more predictably. In contrast, waterfall projects, which are typically large in scope, are notorious for taking much longer than original estimates.
Aside from these delays, some Agile does have zealots who like to bash waterfall because they strongly object to the notion of a process having sequential steps that rely on prior steps. They act like Agile makes it okay to just jump in and start coding things. However, many industry insiders disagree with this and think that

perspective goes too far. Even in Agile, you should design before you code; you're just doing so in much smaller increments.

It's worth pointing out that waterfall *is* a better approach for some projects. For example, we wouldn't want to send humans into space with a minimally viable space ship. Nor would we want to release medical surgical robot software haphazardly. In these dangerous scenarios, checking requirements and reviewing designs multiple times before starting construction is a given. The risk of failure is just too high in these situations; that is, people would experience catastrophe. Also, unlike the code for a website—which can be quickly changed at will—it's much harder to make changes to a space ship or submarine after it's built. When the risk of failure or the cost of making changes is too high, it's better to spend more time gaining a higher level of confidence before starting implementation. In cases like these the testing phase is critical and lengthy. As the software begins coming together, it undergoes rigorous testing that continues until no more software bugs are being found. Note that we are not saying "there are no more bugs" instead we are saying that "no more bugs are being found" despite rigorous and thorough testing.

Agile development's core principles were laid out in the Agile Manifesto, which was written in 2001 (you can view the manifesto and the principles at http://agilemanifesto.org.) Agile encourages early and continuous delivery of working software with a mindset focused on creating value for customers. A key part of Agile is defining your product in a customer-centric way with user stories. As the section on Specify Your MVP Feature Set and User Stories discusses, a user story is a brief description of the benefit that the particular functionality should provide, including whom the benefit is for, and why the user wants the benefit.

Well-written user stories usually follow the template:
As a [type of user],
I want to [do something],
so that I can [desired benefit].

Agile also promotes strong cross-functional communication and collaboration, with business people and developers working together daily, ideally face-to-face. Instead of encouraging adherence to a rigid plan, Agile emphasizes flexibility to quickly respond to change. Teams can accomplish this by completing small batch sizes of work in short iterative cycles with feedback and learning, as opposed to trying to specify the entire set of detailed requirements upfront. Finally, Agile

is about continuously improving your product development process via feedback and experimentation.

There are several different varieties of Agile development, including Extreme Programming (XP for short) and Lean Software Development. Here is an overview of two of the most commonly used Agile methodologies: Scrum and Kanban.

SCRUM

Scrum is the most popular Agile framework. It's relatively easy to adopt because there is ample prescriptive guidance available on how to practice Scrum. A key aspect of Scrum is that the team works in *time-boxed* increments—that is, limited to a specific timeframe. This period of work, called a *sprint* or *iteration*, is a fixed length of time. Two-week sprints are very common, but you also see companies using one-week, three-week, and four-week sprints.

All work that the team completes comes from the *product backlog* of user stories. A backlog is a rank-ordered to do list. User stories are written and placed on the product backlog by the *Product Owner*, one of the three roles specified in Scrum. The Product Owner, or PO for short, is responsible for using input from customers and stakeholders to create the prioritized backlog of user stories. The product manager on the team usually fills the Product Owner role. Some companies have a dedicated PO in addition to the product manager, and the two people coordinate closely. In smaller startups that don't have a dedicated product manager, one of the founders usually wears this hat.

The second role is “development team member.” The Scrum guidelines say that the team should be multidisciplinary with all the skills required to complete the work. Scrum teams usually include several developers, whose job is to estimate the size of stories and build the product. Three other important team roles are UX designers, visual designers, and quality assurance (QA) testers. The traditional Scrum guidelines don't differentiate among team members, but it's fine to acknowledge distinct roles within the team. The designers bring the user stories to life by designing the user experience, which they convey through design deliverables. Well-written user stories include acceptance criteria, which are used to confirm when a story is completed and working as intended. QA testers help check to see if acceptance criteria are met and ensure the quality of the product. The ideal size of a Scrum team is five to nine people. You may have heard of “the two pizza rule”: if two pizzas aren't enough to feed your team, then it's too big.

With this size, you should have enough people to accomplish a meaningful amount of work per sprint. Yet the team is small enough to feel like a cohesive unit and avoid the communication challenges that usually occur with larger groups.

The third role is Scrum Master, whose job is to help the team with the Scrum process and improve its productivity over time. Larger companies may have a Scrum Master that works with one or more Scrum teams, but a dev lead or dev manager often fills this role. Although it's not consistent with the Scrum guidelines, sometimes the role isn't explicitly filled by a single person—it's either ignored or the responsibilities of the role are distributed among the team.

The team carries out certain activities to prepare for the next sprint before it starts. The Product Owner will groom the backlog to make sure that stories being considered for the next sprint are well written and understood by the team. The PO usually does this with the dev lead or dev manager in a backlog grooming meeting (also called a backlog refinement meeting).

See Figure 12.1 for a visual depiction of the flow of work, meetings, and deliverables in Scrum.

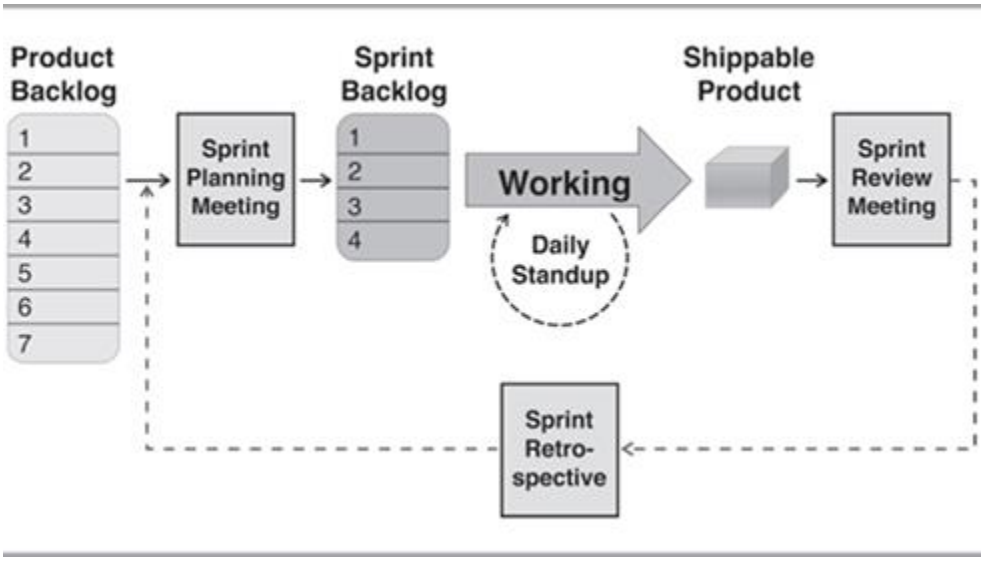

FIGURE 12.1 Scrum Framework

At the start of each sprint, the team holds a *sprint planning meeting* where they decide which stories they plan to accomplish in the iteration and move those stories from the product backlog to the *sprint backlog*. Part of this process requires that the team estimate the scope of each story using *story points*, which are a relative measure of effort. Estimating points can often be more of an art than a science. You can find a variety of point systems in use. Some systems let the team assign any number of points to a story: 1, 2, 3, 4, 5, 6, and so forth. A common approach

is to use the Fibonacci series for points, where the only valid values are 1, 2, 3, 5, 8, 13, and so forth. The benefit of this approach is that it forces distinct differences in estimated values. Another popular point system that forces even larger differences in estimated values is the "powers of two" scale: 1, 2, 4, 8, 16, and so forth. *T-shirt sizing*, another popular technique, uses sizes such as small, medium, large, and extra large to estimate the scope of stories.

Stories with points at the high end of your scoring range have large scope and uncertainty and should be broken down into smaller stories, as discussed in Specify Your MVP Feature Set and User Stories, stories that are too big to complete in one iteration are called *epics*, which *must* be broken down before they can be accepted into a sprint, Many Agile tracking tools enable the use of epics to organize related stories and manage them across multiple iterations.

If story points seem a bit abstract to you, it's because they are—at least at first. The goal is to determine a team's capacity for work by tracking how many story points they complete each iteration—which is called *velocity*. Once a team has calculated their average velocity, they can use that number of story points to plan their sprints. While story points start out a bit abstract, they provide a measuring stick for determining empirical values. In order to calculate velocity, story point estimates need to have a numerical value; so in the case of T-shirt sizing, the team would have to map each size to a relative number of points.

See Figure 12.2 for an example of how a team tracks their velocity over multiple iterations. The horizontal axis shows the iterations, numbered sequentially over time. The vertical axis shows the number of story points completed. Over these 12 iterations, the team's velocity has been variable (between 22 and 40 story points), which is normal. Despite this variability, the trend line shows that the team has been steadily improving their velocity over time.

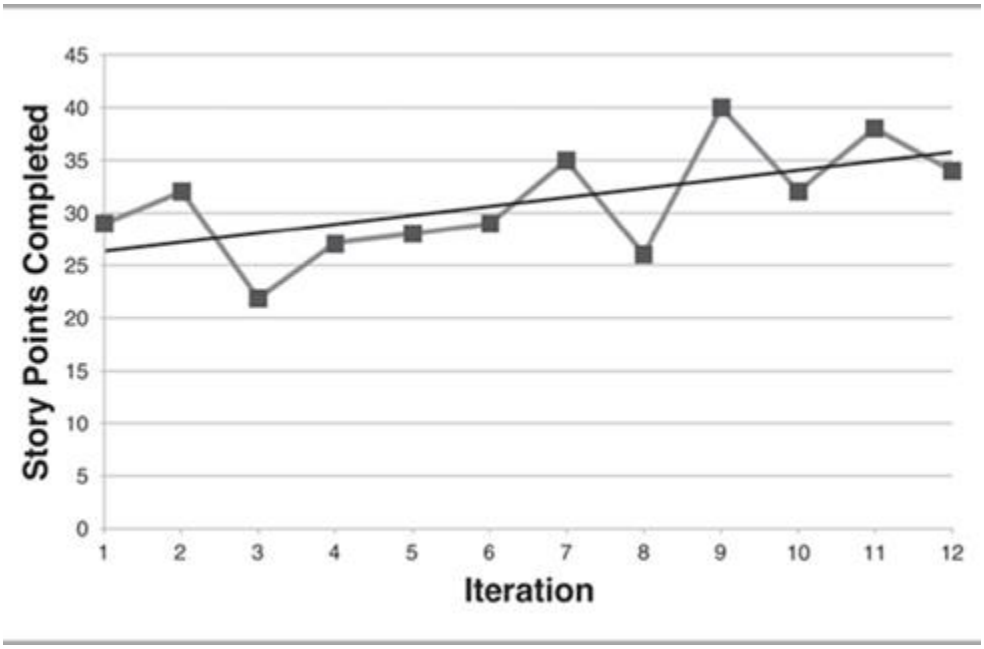

FIGURE 12.2 Team Velocity

Scrum teams use several techniques to reduce their story estimation error and achieve a more stable velocity. Teams will often discuss and estimate story points together, versus having only one team member size a given story. Some teams develop a reference set of user stories of different known sizes. Comparing stories to the reference stories helps them more accurately estimate scope.

Planning Poker is a popular technique for generating quick but reliable estimates as a group. Each team member receives a set of cards, with each card corresponding to one of the possible point values (e.g., 1, 2, 3, 5, and 8). After the team finishes discussing a story, each member privately selects the card with his or her points estimate, and then everyone reveals his or her card simultaneously. If the team has relative consensus, that gives higher confidence that the estimate is accurate. If there are material discrepancies in the estimates, they discuss the story further to try to reach a consensus estimate. Teams will often break each story down into the set of coding tasks required to implement it. This helps ensure that they thoughtfully consider the work required for a story and that they don't overlook anything. Plus, it's usually easier to estimate the effort of each of these smaller tasks compared to the whole story. Some teams estimate the size of tasks with points, while others prefer to use hours of effort. Some teams identify tasks but don't bother estimating them, keeping their estimates at the story level.

When sprint planning is complete, the team should be clear on the set of stories they plan to accomplish in the sprint. They should choose the highest priority stories from the product backlog, and the total number of points for those stories should match the team's expected velocity for the iteration. In teams where the skill set of developers vary, it's also a good idea to ensure each story has been assigned to a specific developer to ensure the team is properly load balanced for the sprint.
The team holds a daily Scrum meeting during the sprint, which is also called a *standup* because many teams stand up during the meeting to help keep it short. This meeting is usually held first thing in the morning so the team can discuss their plans for the day, and is generally time-boxed to 15 minutes. Team members each briefly describe what they did the previous day, what they plan to do today, and anything that is impeding their progress.

The team implements user stories starting at the top of the sprint backlog, collaborating as necessary. Many Scrum tools are available to help teams manage and track their work—some popular ones include JIRA Agile, Rally, VersionOne, and Pivotal Tracker. These facilitate product and sprint backlog management and sprint planning. Team members use them to track the state of each use story, changing states from "to be worked on," to "in development," to "code complete," to "done," for example.
Teams use a *burndown chart*—which shows how much work remains to be completed for the iteration—to track progress. The chart can display the remaining work in either points or hours, depending on the units your team uses for tracking. Figure 12.3 shows an example of a daily burndown chart, with the days of the sprint on the horizontal axis and the remaining story points for the sprint on the vertical axis. It starts on "day zero" of the sprint with the number of points to be completed, 45 in this case. This chart shows 10 working days, which corresponds to a two-week sprint (only weekdays are shown). Ideally, the team ends up with zero remaining story points at the end of the sprint.

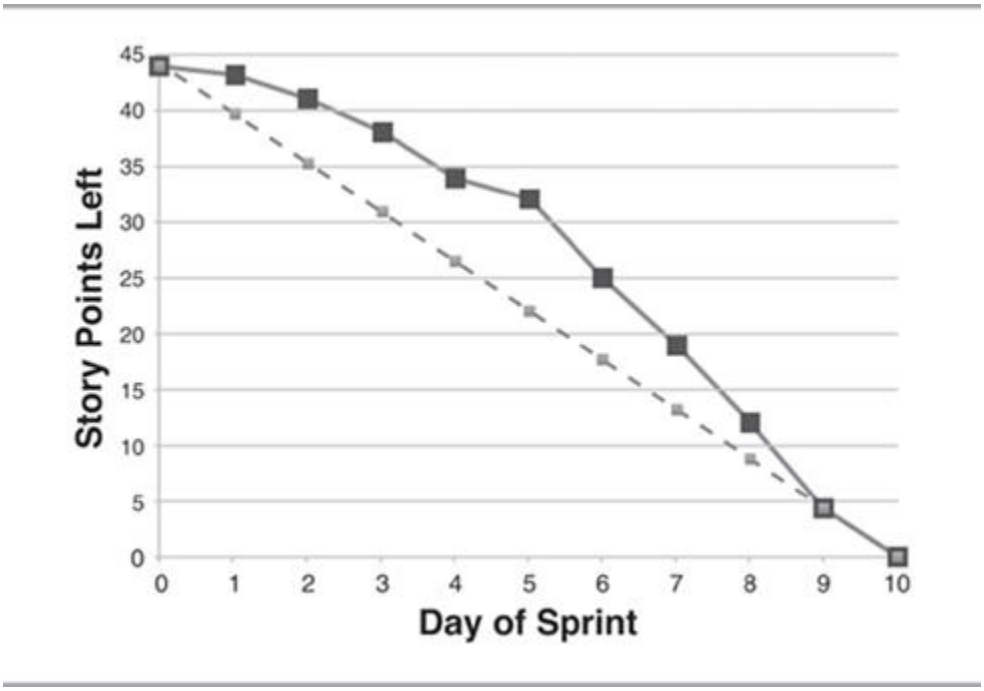

FIGURE 12.3 Burndown Chart

QA testing is conducted during the sprint. To achieve a higher velocity, team members should test stories as developers complete them. If the story meets its acceptance criteria, then it is accepted; otherwise, it is rejected and kicked back to development. The team should also reserve some time at the end of the sprint to test the entire product after development is complete and to fix any bugs they find.

The goal for the end of each sprint is to complete an "increment" of work that adds functionality to the product. The Scrum guidelines direct each team to define what "done" means for them. For many teams, "done" means a product that could be shipped, called a "shippable product" or a "potentially releasable product." Many teams release new product with the same frequency as their iterations, launching the output of their sprint to customers shortly after the sprint ends. Others have a separate release process with a longer cycle where the work from multiple sprints is released together at one time. Regardless of your deployment process, the goal is to ensure the product is in a shippable state at the end of the sprint. At the end of each sprint, the team holds a *sprint review meeting* (also called a sprint demo meeting) where they show what they have built. This helps ensure the product works as expected and lets everyone see the team's progress. Ideally, customers or stakeholders attend the demo to provide feedback to be considered for future sprints.

As with other Agile methodologies, Scrum also focuses on improving the team's process over time. To that end, teams hold *retrospectives* to specifically reflect on how the last sprint went. At these meetings, the team discusses what worked well, what didn't, and what improvements they want to make for the following sprint. Some teams hold retrospectives after each sprint; others do so after two or three sprints.

The basics of Scrum have been covered here—if you want to learn more, see the latest version of the Scrum guidelines at http://scrumguides.org.

KANBAN

Another popular flavor of Agile development is Kanban, a process adapted from the system Toyota developed to improve how they build cars. The Toyota Production System focused on just-in-time production and eliminating waste. Manufacturing workers used paper Kanban cards to physically signal when additional work should be pulled into the system. These cards have been adapted in software development as “virtual” cards that each represent a work item but don't actually generate a pull signal. Instead, it's up to the team members to proactively pull the next work item forward.

A core principle of Kanban is to visualize work. Each card is a user story or a development task that supports a user story. The cards are arranged on a Kanban board, which consists of a set of columns, one for each different state of work. The columns are arranged left-to-right in the order in which work flows. See Figure 12.4 for an example of a Kanban board. This Kanban board has the following set of columns from left to right: “backlog,” “ready,” “in development,” “development done,” “in testing,” “testing done,” and “deployed”—defined as follows:

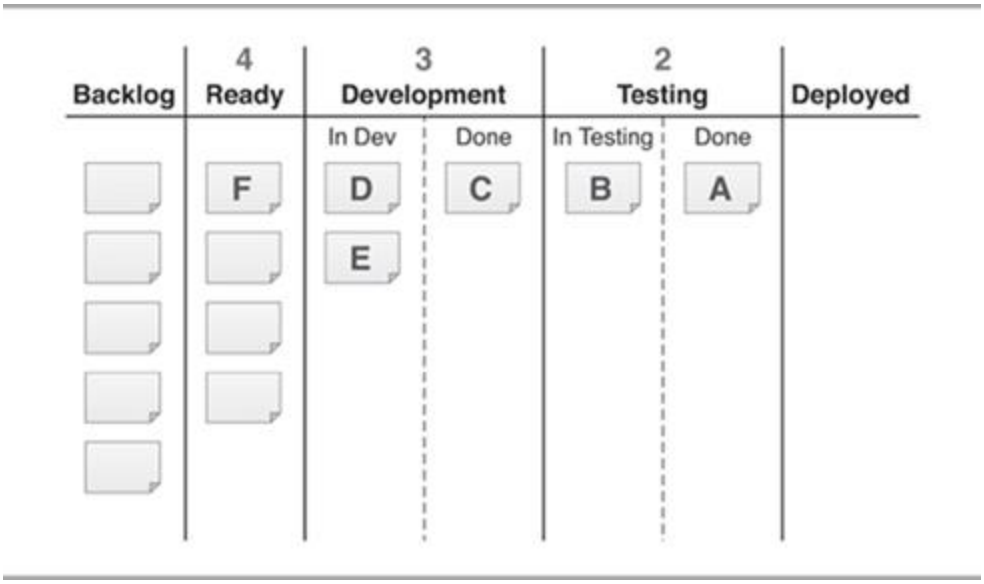

FIGURE 12.4 Kanban Board

- **Backlog**: Items to be potentially worked on, sorted in priority order
- **Ready**: Items that have been selected from the backlog and are ready for development
- **In development**: Items that a developer has started working on
- **Development done**: Items that the developer has finished working on but which have not been tested yet.
- **In testing**: Items in the process of being tested.
- **Testing done**: Items that have successfully passed testing but have not yet been deployed.
- **Deployed**: Items that have been launched.

Some columns represent work being done (e.g., in dev, in testing) while others represent items waiting to be worked on (e.g., ready, development done). The latter type of columns are queues of work. When a team member frees up capacity after finishing work on one item, they pull the top item from the appropriate queue and start working on it.

As a work item progresses through each stage, its card is moved from one column to the next. It's easy to visualize the state of what the team is working on at any point in time by just looking at the board. It's also easy to see where the bottlenecks are by looking at which columns are accumulating the most cards.

You may have noticed that instead of having just a single state for "testing," Figure 12.4 has two: one state for items being tested ("in testing") and a second state for "testing done" items. "Development" similarly uses two states. This helps create a clearer picture of the status of the team's work and helps make bottlenecks easier to identify.

In Kanban, the quantity of active work is managed by constraining the amount of "work in progress" or WIP. The team decides on the maximum number of cards each column can contain, which is called a *WIP limit*. Team members *pull* work items forward sequentially through each state of work. However, they can only move a work item to the next column if that column has spare capacity. This rule helps smooth out the work and achieve a steady flow. Teams should fine-tune their WIP limits over time to optimize their workflow. The WIP limit is displayed above each column. As shown in Figure 12.4, teams often use a single WIP limit to constrain the total number of cards across the two related "in progress" and "done" states (versus having separate WIP limits for each of the two columns). For example, the total number of "development" cards cannot exceed 3. This helps encourage the flow of cards to the right out of the completed states.

Looking at the work item cards in Figure 12.4, when the developer working on card D finishes, he would move it from "in dev" to "done." However, he would not be able to pull Card F forward from "ready" because "development" is at its WIP limit of 3. Likewise, when QA finishes testing Card B, they would move it to "testing done" but could not pull Card C forward because "testing" is at its WIP limit of 2. For work to progress, one of the "testing done" cards needs to be deployed. Once it is, Card C can be pulled forward to "in testing," which frees up "development" so Card F can be pulled from "ready" to "in dev."

You can further organize your Kanban board with *swimlanes*—horizontal lines that separate cards into rows. There are a variety of ways to categorize cards with this technique. You can use swimlanes to prioritize cards (the higher the row, the higher the priority). You can give each epic or each user story its own row. Swimlanes can also show each person's workflow more clearly, by having a row for each team member. You can also track multiple related projects on one board by putting each project in its own row.

The focus in Kanban is on the flow of work. There is no time-boxed iteration as with Scrum. Work items move continuously from left to right on the Kanban board as work progresses. The scope of user stories isn't necessarily estimated, so the

Scrum concept of velocity (story points delivered per iteration) doesn't really apply. But you can measure the team's *throughput*, which is just the number of work items completed in a given timeframe, for example, 10 items per week. If you track your team's throughput over time, it should go up as they make process improvements and become more proficient.

Two commonly used metrics in Kanban are *cycle time*—the amount of time on average from when work starts on an item to when the item is delivered to the customer—and *lead time*, the amount of time on average from when a work item is created (e.g., requested by a customer) to when it is delivered. It's important to note that cycle time and lead time aren't necessarily correlated with effort. A work item could take only an hour to complete but have a much longer lead time if it sat around for a while without anyone working on it.

You can visualize the flow of work in a Kanban system with a *cumulative flow diagram* (Figure 12.5), a stacked area chart that shows how many cards were in each work state at the end of each day. For simplicity, Figure 12.5only uses three work states: “backlog,” “started,” and “done.” You can see the cycle time is the horizontal width of the “started” items, and the lead time is the combined horizontal width of the “backlog” and “started” items. The WIP is the vertical height of the “started” items.

The Kanban mindset focuses on continuous improvement—so your team should be regularly identifying and discussing ways to work better and faster. The idea is that your lead time and cycle time should go down over time as your team makes process improvements and becomes more proficient.

Many teams have a constantly changing backlog; items that were considered important at one point in time become less important as they add new items. Unlike Scrum, where the sprint backlog is locked down within an iteration, team members can change a Kanban backlog at any time. Cycle time may be the better metric on which to focus in such rapidly changing situations. You should still keep an eye on how long it takes to get backlog items ready for development to ensure that isn't decreasing team throughput.

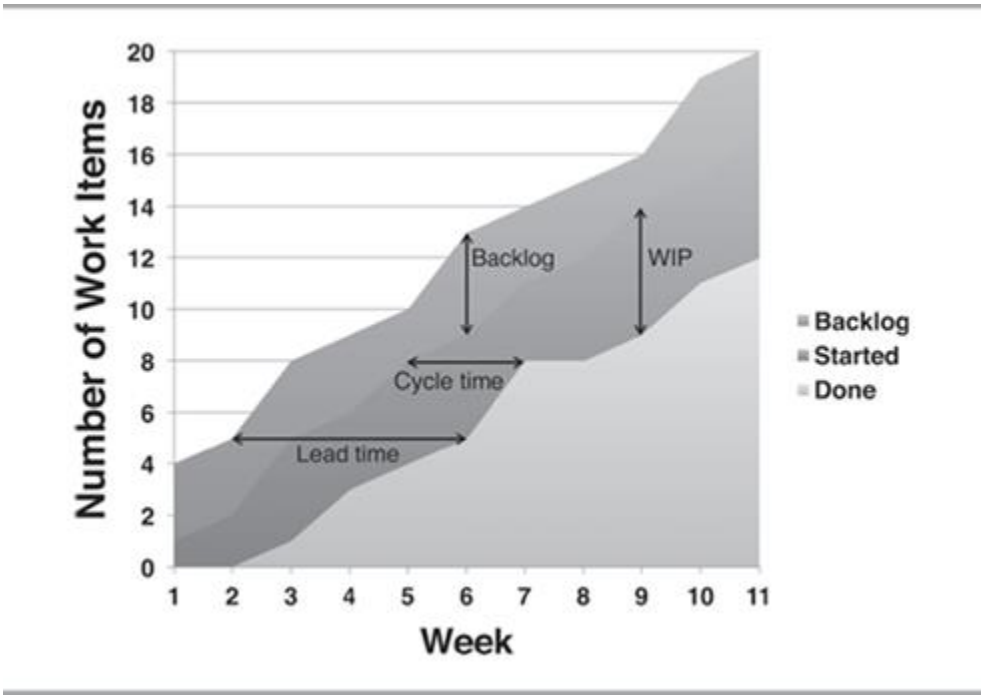

FIGURE 12.5 Cumulative Flow Diagram

If the scope of work items varies greatly, you can see a wide range in your cycle times, with smaller items having shorter cycle times and larger items having longer cycle times. Some Kanban teams use the T-shirt sizing approach mentioned before (small, medium, large, etc.) for work items to enable more precise cycle time values. In that case, you would have a distinct cycle time for each T-shirt size. Kanban does not have the level of process prescription that Scrum does; so no rituals are specified, but many teams practicing Kanban hold daily standups and periodic retrospectives.

KANBAN TOOLS

Many small product teams use a whiteboard for their Kanban board, drawing a column for each work state and then using a sticky note for each work item. This makes it easy to move the items around and for anyone in the workspace to look at the board and see the status of the team's work.

There are many digital tools for managing Kanban. Trello is a popular visual board application used to manage software development. In fact, many people use Trello to manage work outside of development. It's particularly popular with product managers and designers, who may maintain their own work boards that feed into

the development board. Many teams use JIRA Agile for Kanban, and other popular tools include SwiftKanban and LeanKit.

Although it's not a pure Kanban tool, another application worth checking out is Pivotal Tracker. Tracker uses a visual board of columns, one for each of the pre-defined work states. The tool supports an interesting blend of Kanban and Scrum (there is actually an Agile methodology called “scrumban” you should check out if that idea sounds appealing).

Pivotal Tracker lets you estimate story points and calculate velocity if you want; if you don't, it feels more like Kanban. If you do, it feels more like Scrum, except that the backlog for the current sprint is not fixed but rather determined dynamically. Stories are listed in priority order and automatically move in and out of the current iteration based on the estimate of story points that will be completed (using calculated velocity and the time remaining in the sprint). If you want Tracker to feel more like Scrum, you can use the “manual planning” mode (also called “commit” mode), which lets you lock down the set of stories in the sprint backlog.

PICKING THE RIGHT AGILE METHODOLOGY

You now have an overview of Scrum and Kanban—how should you decide which one to use for your team? While devotees of each methodology may view them as vastly different, the two methodologies share many common Agile principles. I've found that Agile frameworks are like shoes: you really have to try them on to figure out how well they fit. It's often wise to just pick the methodology that sounds best to you and try it out for a few months. Many teams start by trying out either Scrum or Kanban. If it's working well, then they stick with it. If not, they switch to the other methodology. After trying on both pairs of shoes, your team should be able to decide which one fits better.

That being said, here's some advice to increase your odds of starting off with the best fitting shoes: Kanban tends to work best with smaller development teams. The lower process overhead and the lack of a predetermined iteration length can enable faster delivery of product. But as a development organization grows to multiple teams, Kanban can start to become more challenging. The lack of a defined cadence to the work can contribute to this, since there is an increased amount of communication required to keep everyone on the same page. Teams that are strong at collaboration are able to scale Kanban to larger sizes. If your organization has

multiple development teams across which you need to coordinate work, then the predictable cadence of Scrum can be beneficial.

The idea of hard launch dates is tenuous with any Agile methodology. Most waterfall organizations are used to having a top-down roadmap that dictates what functionality they should launch each month or quarter, although those deadlines are often illusory due to delays. When these organizations transition to Agile, many still hold on to a waterfall mindset regarding their product backlog. Some developers like using the term “Agilefall” to describe companies undergoing the awkward transition from waterfall to Agile, with a foot in both camps. If your organization would have a hard time letting go of the security blanket of hard deadlines, then Scrum is probably a better fit than Kanban. At least with Scrum, you know you will have work done at the end of each iteration and can make high-level estimates for how many sprints a feature should take. Most Kanban teams don't spend time estimating effort or completion dates. By carefully tracking your cycle time and using simple statistical techniques, it's possible to create projections with Kanban that have relatively high confidence, but many teams don't achieve that level of tracking and precision.

Regardless of which flavor of Agile you choose, using a good tool to manage your work—and there are many available for each Agile methodology – will help considerably. Depending on the size and resources of your organization, you may want to have a few teams try different approaches, then teach other about the pros and cons of each. Following this a development organization can decide based on evidence which approach seems best. One mistake some teams make is to use a general-purpose tool instead of one that is optimized for your development methodology. Try out the tool(s) you think will work best. If you're not happy with it after a cycle or two, then try another one.

Before you adopt a new methodology, it's a good idea to assess each team member's level of knowledge with it. Even if several team members have worked with Scrum or Kanban at prior companies, chances are that there are meaningful differences with how they practiced it there. If you don't set new expectations, team members will likely assume you are following the practices with which they are familiar. There is significant value in everyone on the team hearing the same thing at the same time about how the product development process should work. This ensures that everyone has the same expectations, reduces misunderstandings, and should enhance productivity. Most organizations these days have adopted a learning organization approach, which in its simplest form is just the teams

meeting at regular points to discuss what is working well and what is working poorly. These discussions are documented and plans for action are disseminated. In this manner organizations continuously learn, and they get more efficient and effective.

SUCCEEDING WITH AGILE

Regardless of which Agile methodology you select, the additional advice below should help you succeed in building your product.

CROSS-FUNCTIONAL COLLABORATION

Agile depends on strong cross-functional collaboration. There should be free and frequent communication among product managers, designers, developers, QA, and any other team members, who should speak daily. It's essential to avoid creating silos where each function throws their work product “over the wall” to the next function in the workflow. A certain amount of face-to-face real-time communication is critical to maximize shared understanding and team velocity. High-performing teams also employ communication tools such as chat, a development-tracking tool (e.g., JIRA Agile), and knowledge collaboration tools (e.g., a wiki or Google Drive) to work together effectively.

Every function should be involved throughout the process, though it's natural for a particular function to be more involved than others and take the lead during a certain phase.

In a nutshell:

1. product managers write the user stories, then
2. designers create artifacts, then
3. developers code, and then testers test.

But product development is a team sport. Developers and testers should have some involvement early in the development process so that they understand the rationale behind product decisions, user stories, and UX designs. The team should encourage them to ask questions and make contributions at all stages. Similarly, product managers and designers should be in the loop during development and testing, especially since unforeseen questions or issues often crop up then. You can

tell the level of collaboration by how often team members refer to one another as “we” instead of “they.”

Effective collaboration helps the team achieve shared vision and avoid misunderstandings and allows the team to move faster. Each team member makes numerous decisions about the product every day. If the team has shared vision and understands the goals and rationale, members are more likely to independently make decisions that support that vision.

RUTHLESS PRIORITIZATION

You should maintain an up-to-date, prioritized backlog. It is important to be clear about the next set of user stories you plan to implement when resources permit. This allows you to act quickly. High-tech product teams usually operate in a dynamic environment where requirements and priorities change quickly. It's not enough to identify items as high, medium, or low priority. If a backlog has 15 high priority items, it won't be clear which of those items a developer should start on first when her time frees up. Priority levels are useful but not sufficient; you also need to rank order your backlog items within each level. Having your backlog rank ordered makes it clear which item should be done next. It also makes it much easier to determine where new requirements belong in the backlog when they come up.

The trick is to be both rigid and flexible when it comes to prioritizing your backlog. You must be clear on your rank order priorities at any point in time; but you must also be able to quickly incorporate new or changing requirements. It can be helpful to use the analogy of water and ice. Most of the time, your backlog is like ice; the rank order is frozen and fixed. But when new requirements come in or priorities change, you briefly melt the ice into liquid water so you can rearrange things. Once you're done reordering your backlog, you freeze it again. Following this approach means that your backlog will be up to date whenever anyone looks at it. Developers can reliably pull the item at the top of the stack and start working on it without having to confer with anyone.

ADEQUATELY DEFINE YOUR PRODUCT FOR DEVELOPERS

It's important to provide your developers with the information they need to build the desired product. A set of well-written user stories with accompanying wireframes or mockups usually does a good job of that. If the team already has a style guide in place and isn't introducing any new major UX components,

wireframes are usually adequate. If, however, visual design details need to be conveyed, then mockups should be used. For features that are purely back-end with no UX component, wireframes or mockups aren't required. The team should ensure that it isn't just the happy path—that is, the expected path of user behavior—that they're defining. Rather, they need to think through the different conditions and states that could apply. There is a balancing act here. On the one hand, you want to provide enough definition that developers can start building with confidence that you didn't fail to think through an important aspect. On the other hand, you don't want to experience analysis paralysis where you spend so much time fretting over every detail that implementation gets significantly delayed.

STAY AHEAD OF DEVELOPERS

Many teams have struggled with integrating UX design into their Agile development process. The guidelines for Scrum don't explicitly deal with how best to handle this. It doesn't work well if the designer is creating wireframes for a user story at the same time that the developer is trying to code it.

In order for Agile teams to achieve their highest velocity, developers need to be able to hit the ground running when they start on a new user story—which means that the team must finalize the user stories and design artifacts beforehand. Because you want to achieve a steady flow of work, designers need to be at least one or two sprints ahead of the next sprint. In other words, by the end of sprint N, they should have finalized the design artifacts for sprint N + 1 or N + 2. Of course, the designers need solid user stories on which to base their designs—so product managers need to be working one or two sprints ahead of the designers.

The goal is to make sure that you never starve developers for work and always have at least one sprint's worth of fully groomed backlog ready to go. This requires some balance, because you don't want to specify too many sprints in advance, as things could change. And while I've described the situation in terms of Scrum, it also applies to Kanban. Based on the designers' cycle time, PM should ensure there are enough cards in the "ready for design" queue. Likewise, based on the developer's cycle time, designers should ensure there are enough cards in the "ready for development" queue.

Neither the product managers nor the designers should be doing their work in a vacuum. The team needs to carve out a certain amount of time in the current sprint to review and discuss user stories and designs for future sprints.

BREAK STORIES DOWN

Being Agile requires working in small chunks. We discussed earlier that user stories should not be allowed to exceed some reasonable maximum size (i.e., number of story points). Beyond that, you should strive to break stories down into the smallest size possible. If you have a five-point story, try to find a way to break it into a three-point story and a two-point story. Better yet, try to break it into a couple of two-point stories and a one-point story. This may seem difficult at first, but like most things, you will get better with practice. If you're unable to break the story down any further, then the developers should try to break down the tasks required to implement the story. If they are having trouble doing that, start by enumerating the steps they plan to take to get the work done.

Smaller scope stories and tasks result in smaller estimation errors. Dividing user stories into smaller pieces usually requires that you think about them in more detail, which also reduces uncertainty and risk. You may realize when you break a story down that some elements of it are more important than others, which can help you refine your prioritization. The same advice applies for Kanban, even if you're not using story points. Try to break each larger scope card into several smaller scope cards.

This chapter has covered a lot of ground on how to use Agile methods to build your product. Another important part of the product development process is testing, where you check the quality of what you've built before you release it to customers. Testing is part of quality assurance, the broader discipline of how companies ensure their customers receive a high quality product.

QUALITY ASSURANCE

Software products are inherently complex. They rarely work as expected 100 percent of the time, so you need to have some plan for assuring your product's quality before you release it to customers. Not having a good handle on your product quality can cause headaches like irate customers, lost revenue, and a disruptive drain on your team's resources.

Finding defects as soon as possible is a Lean principle that helps reduce waste. A major bug that you don't detect until after you launch your product is much more costly than one found during development. First, it negatively impacts customers. Second, it is usually more time consuming for the team to figure out the root cause of production bugs and fix them because they are no longer actively working on

that code. Third, because the defect is live, the customer pain persists until the bug is fixed and you deploy the new code to customers.

QA testing should play a significant role, but there are also other ways to increase the software's quality. Coding standards help different developers on a team avoid arbitrary stylistic differences and achieve consistency in how they code, which helps eliminate inconsistencies that can result in quality issues. Coding standards also make it much easier for one developer to understand and modify another developer's code—thereby making the code easier to debug and maintain and improving developer productivity.

In a *code review*, one developer examines another's code—and can catch mistakes that the original developer missed. The reviewer also often has good ideas on how to improve the code. Code reviews allow defects to be found and fixed *before* testing, and are a great way for developers to learn from one another. Going one step further than code reviews is *pair programming*—a technique where two developers work on creating the code together at the same time. They sit next to each other in front of a single computer and keyboard looking at the same screen. The developer in the “driver” role controls the keyboard and writes the code. The second developer plays the “observer” role and reviews the code as his or her partner creates it. The two developers switch roles frequently. Working in pairs promotes learning and usually results in better product designs and higher quality. Pair programming is a central tenet of Extreme Programming, another well-known Agile methodology.

Getting back to QA testing, there are two main types: manual and automated testing. In manual testing, one or more people interact with the product to verify it works as expected. Manual testing is also called “black box” testing because the tester doesn't have to have any knowledge of how the product was built or the technology behind it. Many companies have dedicated, full-time QA testers. In companies that haven't staffed QA, the testing burden falls on the other team members (such as developers and product managers). In those situations, developers are often testing their own code. One benefit of having dedicated QA testers is that they are more likely to find unforeseen problems than a developer checking her own code because they approach testing with a fresh perspective. Additionally, the testing is usually more thorough with dedicated QA resources. First, because it's QA's primary job, they have more time to test. Second, good QA people approach the testing systematically, which results in checking more

conditions. Third, skilled QA people have a knack for being able to find ways to break software and are familiar with common issues that arise.

In automated testing, software is used to run tests on the product and compare the actual results with the predicted results. A person (usually the developer or the tester) has to initially define each automated test case, but once specified, they can be run whenever desired. Each time a set of tests is run, a report of which passed and which failed is generated. One benefit of automated testing is that it can save significant manual testing effort, especially for tests that are conducted repeatedly. However, there's a potential risk in that it is only as good as the set of test cases the team writes. If the team doesn't write test cases for certain functionality, then it won't get tested. By applying intelligence and creativity, a human tester hammering on a product will often test many conditions and combinations not explicitly called out in automated test cases. Such discoveries from manual testing should be used to add any missing automated test cases before the product is released. In addition, when the team makes functionality or user interface changes, they must revise the associated test cases accordingly.

The team must test two different aspects of the product when they build new functionality or make improvements to existing functionality. The first, called *validation testing or unit testing*, checks to see if the new or improved functionality works as expected—that it is consistent with the associated user stories and design artifacts. Sometimes, the product is implemented differently from how it was designed, often due to a mistake or a misunderstanding. The developer might also do this deliberately because it wasn't feasible to implement the product as specified, or he or she chose a lower-effort solution. Even in cases where the product is implemented exactly as specified, the team might then realize that they missed something or didn't get something right. Any of those issues should get detected during validation testing.

The second aspect of product testing is to ensure that none of the other existing functionality was inadvertently broken during the process of building the new or improved functionality. In other words, you add Feature D to your product and want to make sure that Features A, B, and C still work as they did before you added Feature D. This is called *system testing or regression testing*. In this context, the word “regression” means “going back to a worse state”—that is, introducing a bug in existing functionality that wasn't present before.

Many companies use a combination of manual and automated testing, which can be very powerful. Manual testing is valuable for testing new functionality for the first time (validation testing), because the team probably hasn't thought of all the relevant test cases. A manual tester can try out different combinations and conditions to help identify corner cases. As you build more functionality and your product grows over time, the burden of regression testing grows with it. While you can conduct manual regression testing when the scope of a product is small, it's usually not feasible to scale a QA team with your product. That's why automated testing is a great fit for regression testing. As the team adds new functionality, they just need to add new test cases and update previous test cases as necessary.

TEST-DRIVEN DEVELOPMENT

Many Agile product teams practice *test-driven development*, a technique where developers write automated tests *before* they write code. Before coding a desired new functionality or improvement, the developer thinks about how to test it and writes a new test case. The test case *should fail* when the developer first runs it—because the code has not been changed yet. If the initial test doesn't fail, it indicates that the developer did not write the test correctly. The developer writes code until he or she thinks he or she is done and then runs the test again. If the test doesn't pass, the developer keeps working until the test passes. After a successful test, the developer will often *refactor* the code to improve its structure, readability, and maintainability without altering its behavior (while still ensuring it still passes the test).

Test-driven development, also called TDD, has several advantages. First, it usually leads to higher *test coverage*, which is the percentage of your product's functionality that is covered by automated tests. As a result, you'll tend to miss fewer regression bugs—and enhance the team's confidence when they modify existing code (since automated testing lets them easily verify that they didn't break anything). TDD does require some overhead to maintain tests as the product changes over time. But if a team wants to scale their automated regression testing as the product grows, then they need to write new test cases as new functionality is developed—whether they decide to practice TDD or not.

CONTINUOUS INTEGRATION

Many product teams use *continuous integration* to iterate their product development more quickly. In order to explain continuous integration, it helps to understand how software developers manage their code. Development teams use a *version control system* to keep track of every single revision made to the code; this makes it easy to see and manage changes. Version control also simplifies the process of restoring the code base to any prior state, so unwanted changes can be reverted. As of the time of this writing, Git is arguably the most popular version control system for Agile development.

When developers make changes or additions, they start with the current, stable version of the code base, called the *mainline* or *trunk*. Version control lets developers start with separate copies of the trunk (called branches), that they can modify without affecting the trunk. When developers are done building new functionality, they commit their changes to the version control system. Before doing so, each developer should perform *unit testing* of his or her code by writing the relevant test cases and ensuring they all pass. A team of developers all work in parallel, each committing their changes. Before merging the new code with the trunk and releasing it, all the changes are combined or "integrated" to build the new version of the whole product. *Integration testing* is performed at this point to ensure that the new product works as intended.

Historically, integration has typically been a manual process. Continuous integration uses an automated build process to create a new version of the product based on the latest code commits. The new build is automatically tested, and the team is notified about which tests passed or failed. They fix any issues and once the new code passes all the tests, clear it for deployment. Different teams conduct continuous integration with different frequencies: some daily, some multiple times a day, and some after each individual code commit. Continuous integration helps teams identify and resolve product development issues sooner than they otherwise would, which improves the speed with which the team can iterate. This is consistent with the Lean principle of detecting defects as early as possible to minimize waste. Instead of letting issues unknowingly pile up into a big mess between less frequent integrations, continuous integration lets the team deal with each issue as it arises. Another benefit is that your code is always in a shippable state, giving you more flexibility to deliver your updated product whenever you choose. Your test coverage impacts how beneficial continuous integration is: the higher, the better.

CONTINUOUS DEPLOYMENT

Many teams that practice continuous integration also practice *continuous deployment*, where code that successfully passes all tests is automatically deployed. Some companies automatically deploy to a staging environment (an internal environment that customers can't access), while others deploy straight to production. This requires automating your deployment process. Advances in automating operational tasks are being driven by the emerging field of *DevOps*, which focuses on building and operating rapidly changing, resilient systems at scale. A key part of a successful continuous deployment system is having the ability to quickly revert to the previous version of the code if any problems are detected, which is called automated rollback. Metrics that track the health of the product are used to trigger an automated rollback.

Let's walk through an example. A developer commits new code that implements a new feature on a website. The committed change goes through continuous integration, passes all the tests, and is automatically deployed to production. Right after the new code is deployed, the page load times on the website increase to unacceptably high levels, resulting in very slow performance for customers. The high page load times trigger an automated rollback that reverts the version of the product that is live back to the previous version of the code.

In order to work well, continuous deployment requires a robust analytics system. Technical metrics that track server health and performance are required to make sure the system is working properly, as are metrics that track product usage. The system needs to be able to tell if a new deployment prevents users from logging in or using some other key functionality. You also need analytics that track the health of the business. For example, if you had an e-commerce site and the number of orders being placed by customers suddenly decreased sharply after deploying some new code, you'd want that to automatically trigger a rollback.

Many of the topics covered have entire books dedicated to them. These are considered best practices in the areas of Agile development, QA, and DevOps—have elevated the state of the art and made product teams much more effective. The common theme across these ideas is that they all help you build a great product more quickly with less risk.

Once you've launched your product, you can take advantage of the power of analytics. A robust analytics platform helps you understand how your business is doing and how customers are using your product. Analyzing your metrics over time and as you make changes gives you valuable insights that can help you drive

improvements. Subsequent sections cover how to use analytics to optimize your product and business.