Software Engineering Practices for Scrum (Expansion of the SGEP)

Dave Farley

2026-01-18T09:00:00Z

Collected Resources for Scrum Guide Expansion Pack

This document is a collection of independent works. Each section retains its original license or copyright status, as indicated. Please refer to each section for specific usage rights and requirements.

License/Copyright: CC BY-NC-ND 4.0

Note: This section is included in its original, unaltered form with permission under the terms of the CC BY-NC-ND 4.0 license. No changes have been made.

Scrum is a framework for developing and sustaining complex products. It provides the structure within which Scrum Teams can deliver value iteratively and incrementally. Scrum does not specify how Scrum Teams should build products or which techniques they must use. However, Scrum requires Scrum Teams to execute this iterative, incremental work effectively.

If the way work is organized does not support step-by-step learning, experimentation, and adaptation, many of Scrum's advantages disappear. That is why Scrum Teams and organizations must deliberately organize their work to encourage exploration, feedback, and continuous adjustment to succeed with Scrum. The quality and sustainability of outcomes depend on the engineering practices that Scrum Teams apply.

This document outlines essential engineering practices that enhance Scrum's effectiveness in software engineering for digital products.

Purpose

The purpose of engineering practices within Scrum is to ensure that each Increment is in a usable state that enables learning and insight. What "high quality" means can change depending on the stage of development and the level of uncertainty in the product's environment. Engineering practices should therefore support releasing when it makes sense, learning quickly from real use, and helping the people doing the work sustain the ability to deliver reliable outcomes again and again over time.

Without a disciplined engineering approach, Scrum devolves into short cycles of poorly integrated work, which accumulate technical debt and reduce agility.

The Case for "Engineering"

In simple terms, supported by user access and their feedback, modern engineering is the practical side of empirical process control. It is about making things work in the real world. In many fields, "engineering" basically means "the stuff that works reliably."

The purpose of engineering is not to find the one perfect solution. Instead, it helps rule out bad ideas and narrow the options to better ones. These options can then be compared to determine the best fit.

Engineering uses principles and practices that help people avoid common mistakes and weak solutions. Examples of this way of thinking include building in safety margins and designing systems that fail safely rather than causing damage.

Engineering cannot guarantee success. However, it produces far better results than not using engineering at all. When this mindset is applied continuously, it helps Scrum Teams learn, improve, and gradually move toward better outcomes, which is exactly what makes inspect & adapt possible.

So what are the software equivalent of "engineering" principles that can guide people towards those better outcomes?

Principles

Perhaps the most foundational principle in empirical process control, science, and engineering is the idea that one begins by assuming that one's guesses are probably wrong.

Science is a satisfactory philosophy of ignorance: *Because we have the doubt, we then propose looking in new directions for new ideas.*
– Richard Feynman

This is also fundamentally the philosophy behind agile development. One is permitted the freedom to make mistakes and organize the work so mistakes are detected as soon as possible. This idea encourages people to make progress in small, safe, verifiable steps, so they can check for mistakes after each step and correct any problems, whatever their nature. This idea is at the core of any sound engineering approach to software development and shapes how people think about and approach the work as a whole.

It is important, however, to avoid becoming overly focused on the process alone. The ultimate goal of software development is to create value or utility for users. To succeed as an engineering discipline, the organization of work must reliably guide people toward outcomes that are more likely to deliver that value or utility.

So while customer value is the real goal, engineering practices in the context of Scrum share these common principles that will enable people to achieve the real goals of solving customer problems and discovering new opportunities:

1. **Enable Adaptiveness**: *enable rapid, safe change without compromising quality.*
2. **Accelerate Feedback**: *reduce the time to discover errors or validate value.*
3. **Increase Transparency**: *provide a clear picture of product quality and operational state.*
4. **Support Sustainability**: *maintain a consistent (but sustainable) pace without compromising product quality.*

Enabling Adaptiveness through Technical Practices

These principles are deeply interlinked, and at their heart lie the fundamentals that make Scrum work: the ability to inspect & adapt to changing circumstances.

If the goal, from a technical perspective, is to support this ability to inspect & adapt, then clearly one should be able to detect and highlight (inspect) problems easily and quickly, and be able to change (adapt) the code to meet new circumstances, whatever they may be. To do this, people need to efficiently identify problems and, when they do, make changes to resolve them safely and easily.

An adaptive approach to [Modern Software Engineering][https://www.amazon.com/Modern-Software-Engineering-Discipline-Development/dp/0137314914/](1) is then built on two foundational assumptions:

1. that the discipline of software development is fundamentally a process of exploration and discovery, and
2. that the best way to facilitate such a process is by **Optimizing for Learning** and sustaining the ability to learn by **Optimizing to Manage Complexity**.

Core Practices

Optimizing for Learning One needs to be good at learning at multiple levels. Guesses about the nature of the products we create will likely be wrong, and there will be misunderstandings about what users want or need. Even if not, once stakeholders (including, but not limited to, users) see the system, their wants and needs will change.

One needs to learn continuously from users; solutions may not work as expected, product ideas may not work, so some ideas and theories will need to be established and tested to see whether they address users' needs and goals. Learning, unless deliberately dismissed, only counts when it feeds back into deliberate action, such as adaptations to Product Backlog ordering, architectural decisions, or operational changes. Learning without consequential action is theater, and Scrum already suffers from that misinterpretation. Learning with adaptation can be wasteful.

The result of all this uncertainty is that one needs to be flexible and adaptive. There are five core practices at the heart of an adaptive approach to engineering that enable the kind of learning and exploration of the problems stakeholders (including, but not limited to, users) face and the solutions adopted to address them needed to build and adapt systems that better meet the needs of users and other stakeholders.

They are:

1. **Work Iteratively** - *Work in small steps and evaluate the effectiveness of decisions and choices after each small step.*
2. **Optimize for Fast Feedback** - *Collect high-quality, accurate feedback on decisions and deliver it to the person best positioned to act on it.*
3. **Work Incrementally** - *Complex systems never spring fully formed from the mind of some creator; they are the products of an incremental accretion of understanding and features, built over time. One organizes to support that process of accretion, maintaining and enhancing the ability to change the system quickly and easily.*
4. **Adopt an Experimental Approach** - *Treat every change to process, product, or technology as an experiment. Work to control the variables to attempt to understand the results. Treat failure as an opportunity to learn.*
5. **Be Pragmatic - Empirical Learning** - *Engineering is not the same as pure science or mathematics; it is not seeking perfection; it aims to solve practical problems with high-quality solutions. One should learn from the reality of the system, as well as from the theories of how it does, or should, work.*

Optimizing to Manage Complexity Software development operates in inherently complex environments, built on deep stacks of abstractions and technologies, where even simple systems are only a few steps away from significant challenges. This complexity is not inherently negative; it also creates hidden opportunities, technical advances, and new ways users derive value—but when left unmanaged, it becomes a primary reason organisations struggle to sustain effective ways of working. Humans have evolved effective strategies for coping with complexity, such as compartmentalisation, and successful approaches to software development deliberately apply such strategies to steer complexity toward learning, value, and better outcomes.

In software, this is vital to avoid the disastrous end of so many failed systems - the 'Big Ball of Mud,' legacy systems that everyone is too afraid to change. The goal of effective software development is fundamentally different, and to support a sustainable approach, people need to retain the ability to modify the software. This is not about predicting the future of all possible uses, but about compartmentalizing the system so that one can change one part or facet without those changes forcing change elsewhere.

Of course, that also means that one should be able to detect when changes are unsafe, which is firmly in the realm of "optimizing for learning". Managing the system's complexity is about making it easy and safe to change. Here are five practices to inform every design decision and help people build software that is easier to change.

1. **Modularity** - *the degree to which a system's components may be separated and recombined, often with the benefit of flexibility and variety in use.*[2]
2. **Cohesion** - *the degree to which the elements inside a module belong together.* [3]
3. **Separation of Concerns** - *a design principle for separating a computer program into distinct sections such that each section addresses a separate concern.*[4]
4. **Abstraction** - *the process of removing physical, spatial, or temporal details or attributes in the study of objects or systems to focus attention on details of greater importance.* [5]

5. **Managed Coupling** - managing the degree of interdependence between software modules; a measure of how closely connected two routines or modules are; the strength of the relationships between modules.[6]

Unmanaged complexity is seen by many as the primary reason organizations abandon Scrum under delivery pressure. Make the failure mode explicit so leaders recognize themselves in it.

Putting the 'Core Practices' into Practice

It is easy to read these ten ideas as so obvious and so generally accepted that they have become a kind of "motherhood and apple pie" that everyone agrees with, but in doing so, something important is missed. Most of today's software development isn't practiced this way, but most great software development **is**. One would expect this kind of result from a genuine "engineering" approach. In other disciplines, **'Engineering' is the stuff that works!**.

Controlling the variables with Small Steps

One should focus on things that can be understood and controlled, and on how work might be organized to achieve desired outcomes more deliberately, right from the start. Guidelines are needed to help people do this before they have learned enough to fully understand solutions to issues not yet well understood. People need a way of working that allows them to continuously grow their understanding so they can adapt the work. Grow understanding and adapt solutions to match it, relying on knowledge and learning rather than wishful thinking about desired outcomes.

If engineers get this right, they can use this approach to explore the details of understood areas, moving them closer to desired outcomes with a lower risk of failure, while deferring other aspects of the system until they are ready to learn more about them.

Progress via Incremental Change

Treating learning as the core of discipline shifts the focus from relying on arbitrary predictions of the future, which are inherently wrong or at least imprecise, to a more deliberate, exploratory approach that defines how to make useful, dependable, incremental progress.

Some people trade off an illusion of precision in predicting the future, "We will deliver features x, y, and z by Q2 next year", for optimizing to work more efficiently to achieve great results more quickly.

Adopting a more Rational Approach

To adopt a stronger **engineering-led approach** means being grounded in rationality and reality. Let's be clear: **trying to fix time, budget, and scope is irrational**; it won't work, and it provides a very poor basis for planning.

When a specific delivery date matters, a generally reliable approach—where circumstances allow—is to keep the software continuously releasable. If advance communication is necessary, it can be helpful to remain deliberately flexible about the exact content of the release. At the time of writing, Apple exemplified this approach by avoiding detailed pre-announcements whenever possible.

To deliver a fixed set of features, engineers should again work to ensure the system is always releasable, but remain vague about when to release it. Engineers can announce new features once they are in production and ready for release. Once again, some of the most successful companies in the world work to that model.

These strategies are more effective at achieving realistic goals than crossing your fingers and hoping for luck this time while attempting to fix both time and scope.

The Engineering Constraints - Guide-rails to achieve better results

The idea behind this list of ten practices is not to serve as gentle reminders of ideas we have all heard before, but to recommend them as tools to actively steer decision-making. If engineers consistently prioritize these factors in everything they do, they **will achieve better overall results**.

This is not a model where people switch off their thinking and mechanically follow a set of steps. These are practices to adopt and apply intelligently. By default, people often choose options that maximize opportunities to learn, the ability to handle the complexity of the systems built, and the problems they are designed to address.

Maximizing feedback creates more opportunities to learn. If all feedback indicates "all is well", engineers can be more confident in the changes, make progress with greater certainty, faster, and with less stress.

Optimizing working practices, technology choices, and designs to deliver fast, high-quality feedback means engineers will necessarily need to make progress in smaller steps.

Smaller steps are safer, easier to test, easier to revert if something goes wrong, and make it easier to identify the cause of any failure and understand how to correct it.

People will continue to optimize their work to gather clear, definitive feedback multiple times per day. At a minimum, engineers should **assess the software's releasability at least once per day**.

The ability to achieve this forces people to adopt other good behaviors and practices.

If engineers need fast, definitive feedback, they must verify that each small change is safe and sound, so they adopt the discipline of continuously verifying the releasability of the systems. But also, if they are going to make progress in this series of many small steps, and do that effectively at the lowest long-term costs (efficiently), they also need to be able to take those small steps easily and with minimum overhead.

Engineers can't afford lengthy, messy bureaucracy; they need validation to be fast, reliable, and comprehensive. This leads people to adopt high levels of automation and to rely on it to determine the correctness and releasability of their systems.

Automation can be complicated, but when done well, it reduces complexity. Many organizations have attempted to automate builds, tests, and deployments, only to struggle to do so effectively. Taking an engineering perspective helps people to do a better job of this. If engineers want the automated tests to be fast and effective, they need to actively **control variables** using version control, not just for source code but for every change to production.

We adopt techniques such as **Infrastructure as Code** and policies such as **All Change to Production is Made via Version Control.**

Suppose engineers want small changes to be safe and good. In that case, they need it to be easy to determine safety and goodness, and for that, they need **Easily Testable Code** that has a significant impact on design choices, because designs that are easy to test also demand code that effectively **Handles Complexity**.

By intentionally aiming for code and designs that are easier to test, engineers significantly improve their ability to handle complexity consistently and sustainably. Writing tests before writing the code is an important aspect of this approach, but it does not fully capture its purpose. Test-Driven Development (TDD)—often more accurately described as Test-Driven Design—uses tests as a design tool. By applying early pressure to design decisions, TDD encourages simpler structures, clearer responsibilities, and better separation of concerns, thereby improving testability and increasing overall design quality.

The dependencies and interactions among the **ten** principles are more complex than this description suggests; it represents only one possible route through them, one chain of rational reasoning. Starting with Fast Feedback and Small Steps is a particularly effective way to frame the discussion, but the same case for a rational engineering approach could equally be made by beginning with Modularity, Working Experimentally, or Managing Coupling, and arriving at the same conclusions.

All of these practices are very closely related to one another. This is a collection of deeply related fundamentals for discipline that, if used as a ***"North Star for decision making"***, will lead people to better, more effective outcomes. In this case, see the North Star as lighting the path rather than the destination.

Think for a moment about these two collections of ideas, imagine two software products, similar to a product that you are working on now. Now imagine a version of that product where there is:

- **No Iteration** - *getting everything right the first time*
- **No Feedback** - *making perfect predictions of what the users will need and what will work to deliver that to them*
- **No Incrementalism** - *building everything in one step, and crossing our fingers that everything will work in the end. I imagine that there will be a fairly lengthy 'integration period' to try everything out*
- **No Experimentation** - *All of the assumptions, guesses, technology choices, and design decisions were perfect, untried, and correct the first time at the beginning*

of the effort.
- **No Emergent Learning** - *the guesses of what the users wanted, how hackers would attack the system, and how the world changes in the future were all perfect too, so there is never a need to change anything. It's about 'keeping promises.'*

Now, imagine a second product we have described here.

1. **Iterative:** Engineers working in small iterative steps, always ensuring that there is a working, releasable product, even after minor changes.
2. **Continuous Learning:** We continuously gather feedback and learn from it, and
3. **Incremental - always meeting the Definition of Output Done:** build up the systems incrementally, small, verified, change, by small, verified, change.
4. **Controlled Experimentation**: This allows people to actively test their ideas; as they become more accustomed to it, they gain greater control and begin treating every change as a small, controlled experiment. One crucial practical engineering outcome of these small, controlled experiments is the creation of dependable, automated tests, a test harness.
5. **Emergent Learning:** Engineers become so used to working in small steps that they can handle any change - even changes that come from what the system "teaches" us in production and from user feedback. And that leads to an important lesson: it's better to change your plan than to stick to a promise that turned out to be a bad idea.

Which product would you prefer to work on? Which do you think is most likely to be a success? Which do you think is likely to deliver value most quickly and efficiently?

Now imagine another two products, in the first, engineers ignore all the advice about how to manage complexity, so:

- **No Modularity** - *All the code is in one big, messy function.*
- **No Cohesion** - *global variables everywhere and a mish-mash of code doing different jobs jumbled together.*
- **No Separation of Concerns** - *Code to calculate tax, mixed in with code to paint buttons blue, alongside code to store data in the cloud. Change one, and you inevitably change them all.*
- **No Abstraction** - *No obvious organizing principles that help you to answer the question 'where should I implement this change?'*
- **Coupling is Unmanaged** - *Which usually means 'coupling is high', so engineers can't change code in one place without forcing change elsewhere, maybe everywhere else.*

This description is so common that we have a name for it: we call systems that look like this **a Big Ball of Mud**. Now imagine the opposite: the system is:

1. **Modular** - *A system divided up into small, discrete parts, each of which has a clear demarcation at its boundary, an "inside" and an "outside", so that it can keep secrets from interactions from "outside".*
2. **Cohesive** - *All of the components "inside" are related to the job it performs, and everything it needs to do its job is there.*

3. Has a strong **Separation of Concerns** - *Each of these pieces is focused on doing one thing and doing it well.*
4. Using **Abstraction** to simplify conversations between parts - *Interactions from the "outside" via these clearly defined lines of demarcation that represent contracts with the outside world hide, or at least obscure, the internal workings of these parts.*
5. **Manages Coupling** with a general preference for looser coupling. This means that a change in one place or module is less likely to be disconnected from changes in another, keeping these parts *more decoupled. When a particular solution requires tighter coupling, engineers can proceed with care and greater control.*

Which of these two options would be preferable to work on, and which would have the best chance of success, both now and in the long term? It is reasonable to assume that the second product would be preferred, even without any knowledge of the technology it uses, the nature of the problems it addresses, or other specific details.

Imagine how much value could be created if you combined the approaches from both of the second product examples: Agile, grounded in sound software engineering practices.

These ten ideas are generically better than the alternatives. If engineers consistently prioritize these ten things and maximize them across every aspect of their work, they will often achieve better results than alternatives.

Apply these ten ideas to how you…

- Organize Scrum Teams and collaborate with others
- Decompose problems or opportunities
- Plan product development
- Architect systems
- Build systems
- Verify releasability
- Sign-off
- Deploy and deliver software products to your users
- Do rework to attempt to attain desired outcomes
- ***or anything else…***

For all of the above, always **Optimize for Learning.** And, always **Optimize to Handle Complexity.**

If whatever engineers are doing does NOT build 'Better Software Faster' (ideally in a direction of travel), it doesn't count as 'Engineering' yet!

"Engineering" should work to improve the results; otherwise, the idea makes no real sense.

If what "engineers" are doing is not yet "engineering", then they should change something that will improve the ability to learn, enhance the handling of complexity, or both. If engineers do any of those things, they will be closer to a workable solution. Keep changing things so you continually improve at building **Better Software Faster** than before (ideally in a coherent direction of travel); that is what “Inspect & Adapt” is really for! Use both of the [DORA metrics][https://dora.dev/guides/dora-metrics-four-keys/](7), **Stability & Throughput,** to [measure progress][https://leanpub.com/measuringcontinuousdelivery](8).

Behaviours that Help to Drive the Change

Continuous Quality

- [Test-Driven Development][https://courses.cd.training/courses/tdd-tutorial](9) and other [test-first][https://nkdagility.com/resources/test-first-development/](10) approaches.
- [Automated unit][https://en.wikipedia.org/wiki/Unit_testing](11), [Integration][https://en.wikipedia.org/wiki/Integration_testing](12), and [Acceptance Tests][https://courses.cd.training/courses/acceptance-testing-webinar](13).
- A Definition of Output Done that requires tested and working software capable of addressing the Definition of Outcome Done.

These behaviors when combined together support learning and the ability to manage complexity. Test-Driven Development, give fast iterative feedback on work, but also supplies a forcing-function that guides design in better directions, encouraging the creation of more modular, cohesive, better abstracted, more loosely-coupled systems with better separation of concerns, because systems like that are more easily testable.

Working to establish, and vitally, maintain, **high-quality and desired outcomes is fundamental to any successful approach**. Delivering “rubbish” faster does not promote adaptiveness. If engineers can reliably and repeatedly deliver a high-quality product, they can confidently proceed in small, safe steps.

For this, engineers need fast, timely feedback on their work, and for that, they need fast, repeatable test results so that small, safe (sometimes parallel) steps help build confidence in their work and provide evidence of success.

Good automated testing is often central to a sound engineering approach, providing a form of “measurement” that determines the correctness of systems, and, in turn, leads to [better-designed systems][https://youtu.be/ln4WnxX-wrw](14).

Testable systems are **better systems**:

- More modular so that engineers can focus testing on smaller and simpler pieces.
- More cohesive so that engineers can better control the variables within the scope of a test, and so get more determinate results.
- Have better separation of concerns so engineers can test each system behavior in isolation, making tests simpler and more focused on the problem at hand.
- Good tests test behavioural outcomes rather than internal implementation details. So, engineers have clear lines of abstraction to separate system components, allowing them to evaluate what they do without over-testing implementation details

in place. This means engineers can change the implementation without invalidating the test.
- Good tests act as specifications of the intent, rather than "after the fact" assertions that the code works. This often works best when engineers write the test before the code, which helps ensure tests are easy to write. If the test is difficult to write, or many tests are needed for a single piece of code, it indicates a poor design. Product Developers can change the external design of the code and systems to make them more testable, thereby increasing abstraction and reducing coupling, almost as a side effect. It's often helpful to treat TDD as test-driven design.

In summary, testable systems can be tested early and throughout the development process, not only after implementation is complete. Waiting for a system to be deemed complete is missing the most significant value of testing in software development, which is to use it as a form of measurement of the correctness of our working. In this, more engineering-focused approach, automated testing is the equivalent of a carpenter using a ruler to decide where to cut the wood. Our testing steers our design decisions, and this is how we achieve better results.

Continuous Integration and Delivery

- Integrating into the main [branch][https://en.wikipedia.org/wiki/Version_control](15) frequently, at least once a day.
- Automated build and [deployment pipelines][https://youtu.be/m1oMj29P–Y](16).
- Test environments that closely mirror production.

Continuous Integration Continuous Integration (CI) is not about tools; it is about a [better way of working][https://youtu.be/NcU0oEk6z8Y](17). If you think about two or more copies of the same code being worked on and modified in parallel, this represents a messy state to be in. Which one is "correct"? Maybe even more important, what is the current **truth of the system**?

There is no way to definitively answer either question until engineers merge the two copies and resolve any issues they identify. Only then can they tell that the changes work together. This point of merging changes is how engineers establish a definitive **shared truth for the system**. CI focuses on increasing the frequency with which changes are merged to make such a valuation and gain insight into the correctness, or otherwise, of the system.

The definition of CI is:

Continuous Integration is a software development practice where each member of a team merges their changes into a codebase together with their colleagues' changes at least daily. Each of these integrations is verified by an automated build (including test) to detect integration errors as quickly as possible. Teams find that this approach reduces the risk of delivery delays, reduces the effort of integration, and enables practices that foster a healthy codebase for rapid enhancement with new features [35].
– Martin Fowler

In short, everyone working in a shared codebase commits their changes to a shared version of the truth at least once per day, and that snapshot is evaluated for correctness.

After every commit, engineers definitively establish a new snapshot of the **Truth** - the "Current State of the System".

This serves as a synchronisation point, often eliminating ambiguity. Changes to this "Current State" are made sequentially and under strict version control, so they always represent a definitive, accurate, reproducible record of the system.

To qualify as "Continuous" Integration, everyone should **Commit Changes to CI at Least Once per Day**. This is the minimum frequency of "commit" required to count as "Continuous".

Without this daily sharing, engineers risk losing visibility into the system's current state and, as a result, face many problems. CI is the closest engineers can get to a definitive, clear view of the system's state; anything else is, by definition, less certain and therefore more risky. The DORA data show that teams that merge their changes at least daily produce significantly higher-quality systems, more quickly, than teams that don't. CI is the route to **Better Software Faster**.

The DORA group at Google initiated the most scientifically credible research on software development practices over many years, comprising many tens of thousands of surveys. They use a peer-reviewed approach to sociology and strong statistical analysis of the data, and have built an empirically grounded model that highlights patterns and relationships between software development practices and better outcomes, based on measures of:

- **Stability** - The quality of the systems that engineers build, and
- **Throughput** - The rate at which engineers can deliver changes of that quality.

Read about the science behind this study, and some of its more important findings in the book "Accelerate" by Nichole Forsgren, Jez Humble, and Gene Kim [36]

The problem with CI is that it demands some trade-offs. If engineers are **required** to commit changes at least daily to get sufficient feedback to enable people to work quickly, safely, and efficiently. This changes how engineers think about the nature of a **"commit"**. Even the word "commit" is fraught with misunderstanding. In CI, to "commit a change" means that "we have committed to the idea that this change is intended to work"; if engineers extend this to the definition of Continuous Delivery (CD explained below), they are "committing to this change being ready for production".

Unfortunately, in the terminology of the most popular version control system for engineers, "Git", "commit" has a very specific, but different meaning; it means, "we are keeping this change, for now, but may decide not to share it with others later". For the avoidance of doubt, when engineers say "commit", in git-speak, they mean **commit**, **merge** & **push** to a shared, definitive branch, the one place where CI operates.

This means engineers submit changes at least once per day that could end up in production.

Many developers are used to working on features and only "committing" those features once they believe them to be finished (aka "Feature Branching"). CI/CD assumes the software is ready for production after every successful commit.

So that means either, that each feature functionality can be finished in less than a single day, or that we work in ways that mean we are comfortable making changes that may be released at any time, and are merged into the "Trunk" where CI runs to establish the current "Truth of the system", even if they don't yet add up to a usable feature as they are hidden behind a Feature Flag.

Keeping feature branches alive for a long time (days, even weeks) is a problem in Continuous Integration because the code drifts away from the main branch and becomes harder and riskier to merge back (often referred to as merge hell).

There are a variety of ways to achieve this CI way of working, including:

- [Dark Launching][https://martinfowler.com/bliki/DarkLaunching.html](19) - We hide partially completed features by not providing a user-visible route until they are ready for use.
- [Branch by Abstraction][https://martinfowler.com/bliki/BranchByAbstraction.html](20) - works by having people refactor existing code to isolate the code we intend to change behind an abstraction, and develop a replacement for the old code using the new abstraction.
- [Feature Flags][https://martinfowler.com/bliki/FeatureFlag.html](21) - *Software switches that allow people to select which version of a feature users will interact with.*

Fundamentally, the objective of working this way is to separate the acts of deploying change into production from the decision to release new features to users. In CD terminology…

The decision to deploy is separated from the decision to release.

In a Scrum Team, the Developers work to ensure that their system is always ready for deployment, while the decision to release might be influenced by other, non-technical considerations from the Product Owner. Deployment means making changes technically available in the production environment, while release means making those changes visible or accessible to users.

This separation can feel challenging and represents a meaningful compromise, but it becomes necessary when enabling Continuous Integration for more complex features. It allows Scrum Teams to integrate and deploy work safely and frequently without forcing unfinished or experimental functionality onto users. Evidence from the DORA research consistently shows that Continuous Integration leads to faster delivery and higher-quality software than alternative approaches, making this trade-off well worth considering.

Continuous Delivery [Continuous Delivery][https://courses.cd.training/courses/cd-fundamentals](22) is CI's big brother, if CI is about continuously validating that the changes are working together, CD “ups the ante”:

Continuous Delivery: Keeping the software in a releasable state at all times.

The idea is to establish the system as releasable: it is designed, developed, packaged, and tested to production quality, so it can be released to production without further work. We then maintain it in that state for the rest of its life.

To do that, we need to version-control and automate everything we can: functional testing, unit testing, performance testing, security testing, configuration management, deployment, data migration, regulatory compliance, etc.

This is all described in much more detail in my book [“Continuous Delivery”][https://amzn.to/2WxRYmx](23)

The goal is to automate every check that determines whether the system is releasable - build, tests, security checks, quality rules, and deployment steps.

We bundle those automated checks into a Deployment Pipeline. It’s not just a build script; it’s an end-to-end process that proves a change is safe to release.

A “real” Deployment Pipeline should:
• be the single source of truth for “can we release?”, and
• be the only path to production (no manual side routes).

So a deployment pipeline begins with CI and ends in production. It is best thought of as a falsification mechanism rather than a mechanism for proving correctness. This is another of those useful “engineering ideas”, in that however many tests we have, we can never be sure that we haven’t missed something, so **we can never prove the correctness of the system,** but if one test fails, we know definitively that the **system isn’t good enough,** so a key idea of CD is…

If one test fails, we reject the change!

In this automated world, test reliability is critical, so we need to do a good job of automated testing. Here are several resources that explore this topic in more detail:

“Continuous Delivery Pipelines”, - Book by Dave Farley[24]
“Where to Start with Automated Testing” - YouTube video[25]
“5 Reasons your Automated Tests Fail” - YouTube video[26]
“The Ultimate Guide to BDD” - YouTube video[27]
“TDD Tutorial” - Free Tutorial[28]
“Acceptance Testing” - Webinar recording[29]

Design and Architecture for Adaptability

- Continuous [refactoring][https://courses.cd.training/courses/refactoring-tutorial](30) to improve design.
- [Evolutionary architecture][https://youtu.be/ElMnHDSFaCw](31) (often called emergent architecture) is guided by feedback.

- Collaborative design techniques such as [pair or ensemble programming][https://youtu.be/fbxMV76e7_E](32).

If the goal is to maintain the software in an always-releasable state, we can only realistically achieve it by making changes easy. Part of that is being able to quickly spot when we make a mistake, so effective deployment pipelines and robust automated testing are essential components of this strategy.

Another equally important factor affecting our ability to achieve this is the system's testability. Systems designed from the outset to be easily and reliably testable are also fundamentally easier to change.

Making change easy is central to the CD approach. While there is no single "ideal architecture for CD," certain architectural approaches and choices can make it more difficult.

Fundamentally, this engineering-driven approach to progress through many small, safe, validated changes forces people to adopt a more evolutionary approach to software design & architecture.

Code Quality, Maintainability, and Fiscal Cost

- Collective ownership of code and responsibility for quality.
- Peer review through pairing, mobbing, or structured code review.
- Clean code practices and automated analysis tools.

The quality of a software system is defined by how easy it is to change! This reduces fiscal cost.

"Ease of Change" may seem like an odd definition of quality, but in reality, it underlies everything else we value and leverages the unique strengths of software as a medium. Ease of change is what makes software "soft".

Of course, when we think about "quality" in software, there are many other factors – often referred to as quality requirements, quality goals, or NFR (non-functional requirements) we might value, such as security, resilience, speed, usability, and more. But what does it take to achieve any of these if the software is not easy to change? The only alternative is to get whatever we are striving for, **perfectly right the first time.**

This is magical thinking, an illusion of unattainable perfection; engineering, of any form, is more pragmatic than that. Engineering as a discipline assumes that we can, and will, make mistakes, and good engineering helps people to fail safely or at least in a controlled manner to minimize the impact of mistakes so that we can recover from them more gracefully.

While we certainly want more from the system than ease of change, ease of change is the route to achieving all of those other things and is the most fundamental property of any high-quality system.

This engineering approach is fundamentally based on making systems easier and safer to change.

Operational Excellence

- Monitoring, observability, and fast feedback from running systems often referred to as telemetry.
- Techniques such as feature toggles, canary releases, or dark launches.
- Practices that build resilience, such as automated rollback or chaos testing.

Measuring Success through Observability To make change easier, we aim to establish and optimize feedback loops to help people continually build a shared understanding and lay the foundations of a learning culture and process.

The most important lessons that we need to learn if the aim is to build good software is what the users make of it. To do that, we need to find ways to close feedback loops all the way through production.

Excellence in this looks like treating every release, even the small ones, as an experiment. One of the complexities is that there are no standard measures available. The appropriate thing to measure changes not just feature by feature, but actually outcome by outcome.

Observability should influence product decision-making. Feedback and telemetry without decision authority is noise, and many organisations stop there.

Here are a few different kinds of "experiments" that may be interesting to perform for different kinds of features…

- "Up-time", does this change make the software more resilient?
- Security, is the software more secure?
- A/B testing: Which options do users prefer?
- Does this change recruit more users?
- Does this change generate more available money?
- …

…but there are also many, many, other measures that may matter, depending on the feature we are creating.

Site Reliability Engineering (SRE) defines two useful concepts to support this: Service Level Indicators (SLIs) and Service Level Objectives (SLOs). SLIs are metrics that indicate how well people perform with a particular service. We recommend considering how to measure success or failure, selecting the appropriate indicator for each feature, and enabling tracking of that indicator during feature development. This will provide people with strong observability, meaning greater insight into the system's operation in production at the level of the behaviours that really matter.

The next step to practicing SRE is to define SLOs, specific target measures based on the SLI that indicate some measure of success. So, for example, if the SLI is *"new users added,"* the SLO might be *"500 users,"* and if the feature didn't add 500 new users, it is not deemed successful. Now one can use evidence like this to make decision-making more rational, even adding SLOs to the acceptance criteria as outcome criteria for a feature, e.g., *"This feature is done when we recruit 500 new users."*

SLOs should be owned by the Product Owner and connected to Product Goals, not treated as engineering KPIs.

Read more about SRE in Vladyslav Ukis' book ["Establishing SRE Foundations"][https://amzn.to/3MbcT5C](34)

Continuous Releasability The working definition of Continuous Delivery is that the software is kept in an always releasable state. This is an important, generic measure of success and is more fundamental to the practice than merely frequent releases.

Frequent releases are a very good idea and simplify many things, but they are also more contingent on the business context, the nature of the software being built, and other factors. The advice is to release as often as the external context allows.

Frequent release is a forcing function; one can't achieve it without doing a good job in all of the things described here. By increasing release frequency, one is forced to address development process deficits largely through greater and more effective automation.

The mistake, though, is to assume that "frequent release" is the real goal; it is not. The real goal is **Building Better Software Faster**. Frequent release is a technique to help people achieve that. The enabling, more fundamental, step to both of these ideas, the practical target that helps people to end up "building better software faster", is:

Work so the software is ALWAYS in a releasable state

Continuous Deployment **Continuous Deployment** - ***Push change into production on an automated basis after every successful change***

If one is providing public web services or other software-as-a-service, then automating the decision to release and practicing "Continuous Deployment" is what operational excellence looks like. In this model, if the **Deployment Pipeline** says "All looks good," the change is automatically pushed into production. Scrum Teams that work this way will update their production systems multiple times per day, sometimes every couple of seconds.

Releasing frequently is generally less risky than releasing infrequently. There are many reasons for this, but at its simplest, consider this. If each change is small, it is also inherently going to be simpler. That means there are fewer hiding places for mistakes. If each change is small, it can be tested more easily, and even if it is wrong, it can be more easily reverted too.

The other significant advantage is that, if each change is small, the difference between what was in production and what is released is also small, making the behavioural delta inherently lower risk. This is supported by data evaluating the impact of practices intended to make software safer, as described in the book "Accelerate, The Science of Lean Software and DevOps" by Nichole Forsgren, Jez Humble & Gene Kim. They found:

"That external approvals were negatively correlated with lead time, deployment frequency, and restore time, and had no correlation with change fail rate. In short, ap-

proval by an external body (such as a manager or CAB (Change Advisory Board)) simply doesn't work to increase the stability of production systems, measured by the time to restore service and change failure rate. However, it certainly slows things down. It is, in fact, worse than having no change approval process at all."

Most orgs that adopt this model of working in a sequence of frequent, small, verified changes use informal peer review, rather than the more formal, gatekeeper approach common in traditional development organizations.

Based on DORA's survey data, this is significantly a stronger predictor of high scores in **Stability & Throughput**, that is, scores that mean that these Scrum Teams produce **Better Software, Faster**. One of the more profound findings of the DORA research is that **There is NO Trade-Off between Speed & Quality.** All of this is true, even for Scrum Teams working on safety-critical systems. So, fast, high-quality feedback based on work divided into small, verifiable steps is the highest-quality approach and perhaps the most efficient found so far.

Conclusion

This document is an overdue integration with the Scrum Guide Expansion Pack, as engineering practices are expressed in the 'Definition of Output Done' and they:

- Enable each Sprint to produce a usable, potentially releasable Increment.
- Strengthen Scrum's pillars: Transparency (quality visible), Inspection (rapid feedback), and Adaptation (safe change).
- Reinforce the Scrum Values: Commitment to quality, Focus on working software, Openness about problems, Respect for users and Product Developers, Courage to improve practices.

These practices are well aligned with the core philosophies that underpin Scrum and strongly reinforce its effectiveness. Organizations that claim to 'do Scrum' without enabling these practices for engineering-oriented 'Scrum Teams' are running timeboxed project management under the guise of agile language, not the essence of Scrum.

Scrum Teams are responsible for choosing and applying engineering practices by defining and fulfilling an appropriate Definition of Done for their domain. It is inherent to Scrum and adaptive practices that improvement is ongoing. The engineering principles described here are the most evidenced version of what really works in practice in a very wide variety of different organizations, building different kinds of software from safety critical systems, through to high performance finance systems, global scale internet systems, banks, space craft, embedded systems, cars, military systems and in large scale enterprises, startups, legacy systems, and Scrum Teams building AAA Games.

Scrum Teams have successfully applied these techniques across all types of software and companies. These ideas are generic and work effectively wherever they are applied. If followed, they can help ensure success; while nothing guarantees results, one can be more confident that better outcomes will be achieved than if one does not do these things. After all, that's what "engineering" is really for!

References

[1] Farley, D. (2021) Modern Software Engineering: Doing What Works to Build Better Software Faster. Addison-Wesley Professional. Available at: https://www.amazon.com/Modern-Software-Engineering-Discipline-Development/dp/0137314914/ (Accessed: 22 November 2025).

[2] Wikipedia (2023) 'Modularity'. Wikipedia. Available at: https://en.wikipedia.org/wiki/Modularity (Accessed: 23 November 2025).

[3] Wikipedia (2023) 'Cohesion (computer science)'. Wikipedia. Available at: https://en.wikipedia.org/wiki/Cohesion_(computer_science) (Accessed: 23 November 2025).

[4] Wikipedia (2023) 'Separation of concerns'. Wikipedia. Available at: https://en.wikipedia.org/wiki/Separation_of_concerns (Accessed: 23 November 2025).

[5] Wikipedia (2023) 'Abstraction (computer science)'. Wikipedia. Available at: https://en.wikipedia.org/wiki/Abstraction_(computer_science) (Accessed: 23 November 2025).

[6] Wikipedia (2023) 'Coupling (computer programming)'. Wikipedia. Available at: https://en.wikipedia.org/wiki/Coupling_(computer_programming) (Accessed: 23 November 2025).

[7] Harvey, N. (2025) 'DORA's software delivery metrics: the four keys'. DORA. Available at: https://dora.dev/guides/dora-metrics-four-keys/ (Accessed: 22 November 2025).

[8] Smith, S. (2016) Measuring Continuous Delivery. Leanpub. Available at: https://leanpub.com/measuringcontinuousdelivery (Accessed: 22 November 2025).

[9] Farley, D. (2025) 'TDD Tutorial'. CD.Training. Available at: https://courses.cd.training/courses/tdd-tutorial (Accessed: 22 November 2025).

[10] Naked Agility with Martin Hinshelwood (n.d.) 'Test First Development'. Available at: https://nkdagility.com/resources/test-first-development/ (Accessed: 23 November 2025).

[11] Wikipedia (2023) 'Unit testing'. Wikipedia. Available at: https://en.wikipedia.org/wiki/Unit_testing (Accessed: 23 November 2025).

[12] Wikipedia (2023) 'Integration testing'. Wikipedia. Available at: https://en.wikipedia.org/wiki/Integration_testing (Accessed: 23 November 2025).

[13] Farley, D. (2025) 'Acceptance Testing – Webinar'. CD.Training. Available at: https://courses.cd.training/courses/acceptance-testing-webinar (Accessed: 22 November 2025).

[14] Farley, D. (2022) 'TDD Is the Best Design Technique'. YouTube. Available at: https://youtu.be/In4WnXx-wrw (Accessed: 22 November 2025).

[15] Wikipedia (2023) ‘Version control’. Wikipedia. Available at: https://en.wikipedia .org/wiki/Version_control (Accessed: 23 November 2025).

[16] Farley, D. (2025) ‘3 Reasons Your CI/CD Pipeline Isn’t Working As It Should…’. YouTube. Available at: https://youtu.be/m1oMj29P-Y (Accessed: 22 November 2025).

[17] Farley, D. (2025) ‘The 10 Commandments of Continuous Integration (CI)’. YouTube. Available at: https://youtu.be/LO20eX6z8Y (Accessed: 22 November 2025).

[18] Farley, D. (2021) ‘Why CI Is Better Than Feature Branching’. YouTube. Available at: https://youtu.be/lXQEi1O5I0I (Accessed: 22 November 2025).

[19] Fowler, M. (2020) ‘Dark Launching’. Martin Fowler. Available at: https://martin fowler.com/bliki/DarkLaunching.html (Accessed: 22 November 2025).

[20] Fowler, M. (2014) ‘Branch by Abstraction’. Martin Fowler. Available at: https: //martinfowler.com/bliki/BranchByAbstraction.html (Accessed: 22 November 2025).

[21] Fowler, M. (2010) ‘Feature Flag’. Martin Fowler. Available at: https://martinfo wler.com/bliki/FeatureFlag.html (Accessed: 22 November 2025).

[22] CD Training (n.d.) Continuous Delivery Fundamentals. Available at: https://cour ses.cd.training/courses/cd-fundamentals (Accessed: 22 November 2025).

[23] Humble, J. and Farley, D. (2010) Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation. Addison-Wesley Professional. Available at: https://amzn.to/2WxRYmx (Accessed: 23 November 2025).

[24] Farley, D. (2020) Continuous Delivery Pipelines: How to Build Better Software Faster. Leanpub. Available at: https://leanpub.com/cd-pipelines (Accessed: 22 November 2025).

[25] Farley, D. (2023) ‘TDD or BDD When It Comes to Automated Testing?’. YouTube. Available at: https://youtu.be/Z9fGG1k6P40 (Accessed: 22 November 2025).

[26] Farley, D. (2022) ‘5 Reasons Your Automated Tests Fail’. YouTube. Available at: https://youtu.be/vHBzZHE4tJ0 (Accessed: 22 November 2025).

[27] Farley, D. (2022) ‘An Ultimate Guide to BDD’. YouTube, 14 December. Available at: https://youtu.be/gXh0iUt4TXA (Accessed: 22 November 2025).

[28] Continuous Delivery Ltd. (n.d.) ‘TDD Tutorial’. Available at: https://courses.cd.t raining/courses/tdd-tutorial (Accessed: 22 November 2025).

[29] Continuous Delivery Ltd. (n.d.) ‘Dave Farley on Acceptance Testing – Webinar’. CD Training. Available at: https://courses.cd.training/courses/acceptance-testing-webinar (Accessed: 22 November 2025).

[30] Continuous Delivery Ltd. (n.d.) ‘Dave Farley on How to Refactor Bad Legacy Code’. CD Training. Available at: https://courses.cd.training/courses/refactoring-tutorial (Accessed: 22 November 2025).

[31] Farley, D. (2022) ‘What Software Architecture Should Look Like?’. YouTube, 30 March. Available at: https://youtu.be/EIMHDSFaCw (Accessed: 22 November 2025).

[32] Farley, D. (2025) ‘The Pros & Cons of Pair Programming (With Examples)’. YouTube, 5 February. Available at: https://youtu.be/fbXMV76e7_E (Accessed: 22 November 2025).

[33] Farley, D. (2022) ‘Improving Observability and Testing in Production’. YouTube, 3 August. Available at: https://youtu.be/NmUAR7pSM (Accessed: 23 November 2025).

[34] Farley, D. (2021) Modern Software Engineering: Doing What Works to Build Better Software Faster. Addison-Wesley Professional. Available at: https://amzn.to/3Mbct5C (Accessed: 23 November 2025).

[35] Fowler, M. (2006) ‘Continuous Integration’. *Martin Fowler*. Available at: https://martinfowler.com/articles/continuousIntegration.html (Accessed: 18 January 2026).

[36] Forsgren, N., Humble, J. and Kim, G. (2018) *Accelerate: The Science of Lean Software and DevOps – Building and Scaling High Performing Technology Organizations*. Portland, OR: IT Revolution.