What is Dual Track Development: Does Agility Result in Fast, Ugly, & Poor-Quality Code?

Abstract

Bottomline Upfront. Dual track development is the art and science of reducing bloated product disciplines into bite-sized just-in-time chunks directly integrated into PBIs, Sprints, and DevSecOps pipelines.

Motivation. *Global firms prefer to apply lightweight lean-agile frameworks for discovering and delivering innovatively new products and services with the shortest possible lead and cycle times. New product and service MVPs must be delivered to production endpoints (websites) to global markets in just a few weeks, months, and quarters to successfully thwart global competition by being first to market with innovative product and service features. Traditional linear waterfall staged product and service initiatives typically took decades and billions of dollars to incorporate long-lead time product disciplinary practices such as user experience (UX), testing, quality, performance, security, maintenance, operations, etc. As such, global organizations are befuddled about how to successfully incorporate product disciplines in short cycles.*

Problem. *The main challenge is that new product and service disciplines have matured, blossomed, and exploded into veritable portfolios of principles, practices, metrics, and tools. Just some of these disciplines include UX, testing, quality, performance, security, maintenance, operations, and much more. Typical disciplines include performance, interoperability, reliability, availability, security, usability, operability, extensibility, scalability, testability, auditability, observability, etc. Yet another website identifies over 90 attributes of high-quality products and services. Each of these disciplines are replete with 1,000+ page guides for applying long-lead time principles, practices, and tools with enticing but diminishing returns. In other words, textbook approaches are orthogonal and counterproductive to modern lean-agile thinking.*

Approach. *Our approach is to first illustrate the basic operational principles, practices, and goals of simple, lightweight lean-agile frameworks, where lean-agile frameworks initially skipped traditional product and service disciplines, how traditional product disciplines tried to shoehorn themselves into lean-agile frameworks, and then how lean-agile frameworks adapted, absorbed, and subsumed traditional disciplines. In doing so, second-generation lean-agile frameworks achieve a good balance of bite-sized just-in-time product disciplines with the need to deliver new products and services in the shortest possible lead and cycle times to measure customer feedback and rinse-and-repeat fewer times for optimal performance.*

Results. *The result was recognizing traditional product disciplines, like linear staged waterfall frameworks, were simply over-bloated with an unnecessarily large number of practices. As such, lean-agile practitioners cherry picked the best product discipline practices, shortened them into bite-sized chunks, and integrated them directly into PBIs and user stories as definition of done, acceptance criteria, tasks, automated DevSecOps-driven tests, and occasional non-developmental discovery PBIs, epics, features, and stories.*

Conclusion. *Our conclusion is a multifaceted examination of dual track development principles and practices consisting of a progression of maturity stages, phases, plateaus, or steppingstones to modern lean-agile principles, practices, tools, and performance measurement. First is realizing out-of-the-box first generation lean-agile frameworks were devoid of product discipline practices. Second is recognizing Scrummerfalling approaches where sprints were simple waterfalls of analysis, design, coding, and testing, with separate personnel, PBIs, and user stories for each major product discipline. Third is recognizing traditional product disciplines could exist in separate, parallel, and sometimes concurrent tracks, teams, or lanes offset by an earlier sprint or a release to feed implementation teams with high value epics, features, user stories, and PBIs. Fourth is recognizing product discipline practices could be cherry picked and downsized into bite-sized*

just-in-time chunks and directly integrated into implementation sprints in the form of small but separate PBIs, definitions of done, acceptance criteria, or tasks. Fifth is recognizing product discipline needs, requirements, and criteria could be codified in the form of automated tests and run, evaluated, and ensured in the form of CI, CD, and DevSecOps pipelines without the need for dozens of separate product development teams and professionals while achieving a balance of robust product MVPs and the shortest possible lead and cycle times for optimal customer performance.

Footnote. *The overriding conclusion is discovery is an ongoing process which cannot be skipped to achieve economies of scale with full utilization by forcing agile teams to continuously code while turning a blind eye to customer pain points and solutions. Dual track development is a pay-me-now-or-pay-me-later (or zero-sum-gain) paradigm as product managers must invest in product disciplines, slow coding productivity, and deliver higher quality MVPs sooner. In doing so, customer feedback is more meaningful, while avoiding fast, ugly, and poor-quality code. Finally, dual track development principles, practices, and goals can better be achieved by incorporating small automated multi-disciplinary tests into robust DevSecOps pipelines.*

Introduction

Agile frameworks emerged in the 1990s as a computer programming answer to the U.S. lean manufacturing movement. That is, agile frameworks like XP and Scrum were a lightweight answer to the monstrously heavy Toyota Production System (TPS). As such, agile frameworks were minimalistic one-piece-workflow Kanban-like systems. One starts with a simple backlog (queue) of independently scoped (modularized) PBIs or user stories, implements them one at a time as a team (pair programming), tests and integrates them, and moves on to the next one in single file. Agile frameworks opposed the century old practice of forming thousands of system requirements and implementing them using waterfall stages (analysis, design, code, test, etc.). The waterfall approach took decades and billions of dollars, requirements documents written in a vacuum didn't represent real customers, 90% of traditional projects completely failed, customers went unsatisfied, and investments wasted. Agile frameworks were meant to quickly gather customer needs, implement real (modularized MVP) solutions quickly, put them in customer hands, get feedback, and rinse and repeat until the customer was satisfied in rapid cycles. Each PBI or user story was a vertical slice vs. horizontal layer (and was a real, operational, and working product, MVP, or lightweight version of a product). In this system, small agile teams only implement exactly what the customer needs and then stop (instead of predicting thousands of customer needs or requirements decades into the future after the customers no longer need them). The challenge becomes, what happens to all of the product disciplines that must be represented by each PBI or user story and rapidly implemented (i.e., UX, testing, quality, performance, security, maintenance, operations, etc.). The goal of an agile team is to quickly develop, deliver, and evaluate a few small PBIs or user stories in days and weeks. Product disciplines felt their perspectives were ignored. So, alternatives to agile frameworks evolved to either ignore critical product disciplines to ship and evaluate new code solutions quickly or revert to implementing MVPs in layers vs. slices (i.e., linear user stories for requirements, design, code, and test disciplines). Agile sprints or iterations were called Scrummerfalling, Sprint Waterfalling, etc. Agile frameworks quickly gained a reputation as an approach to shipping bad code quickly that didn't satisfy various product disciplines or even the customers themselves (i.e., ship it now and fix it later).

(Phase I) **Early Approaches**

(Early) agile frameworks were not completely oblivious to the needs of product disciplines. Strategic frameworks like Scrum assumed developers would just do the right thing (i.e., responsibly incorporate the interests of various product disciplines, and incorporate customer needs too). However, they didn't specify

the mechanisms for capturing customer needs, product disciplines, or even production, operations, and maintenance of the final solution. Tactical frameworks like XP incorporated a few more mechanisms for representing key stakeholder needs. That is, in XP's case, customers were supposed to participate or be a part of the XP team, and the requirements of various product disciplines were to be captured in the form of automated static analysis, builds, integration, tests, and even deployment. As such, continuous integration (CI) practices co-emerged with XP. The first order of business was for customers to write the user stories, explain them if necessary, automated test chunks were supposed to be written in advance (TDD), version control systems were to be used to capture code (CM), the system was to be compiled and built upon code check-in (Operations), and automated test chunks were to be run and passed as well (acceptance, security, performance, etc.). XP's goal was to specify and capture the needs of the customer and product disciplines in automated test chunks and any code was required to pass the tests. Developers were incentivized to pass the tests, required to fix broken builds (failed tests) immediately, and each code check-in (completed user story) would be delivered to the production system (for customer evaluation). A little extra care would be necessary for security engineering and UX disciplines, but certainly possible using CI/CD practices (which evolved into DevSecOps pipelines). DevSecOps would later be adopted as the standard for Data Center (Cloud) operations (and software development was subsumed by Cloud operations). But more on this later.

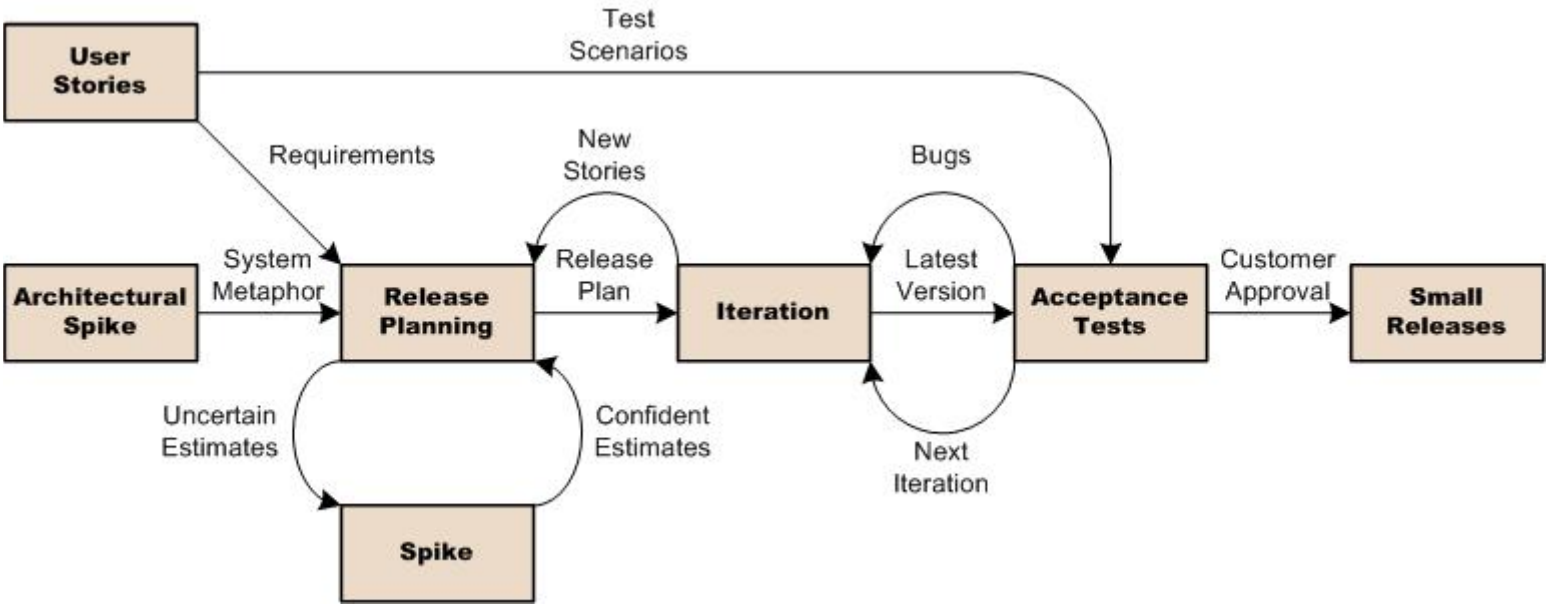

Figure 1. **Extreme Programming** (XP) – Lightweight one-piece workflow system (used for quarterly planning)

(Phase II) **Scrummerfalling**

One of the early challenges with agile frameworks was that product disciplines missed the memo on successfully applying XP's tactical CI/CD (DevOps) practices. Rather than incorporate the requirements of various product disciplines as automated test chunks which developers had to instantly satisfy in days or weeks, these disciplines still demanded manual practices. UX, security, test, quality, performance, etc. engineers demanded to conduct manual activities. And, if not done manually, refused to participate in the agile paradigm at all. Worse yet, reject them altogether and revert to traditional requirements-driven staged waterfall practices which took decades and billions of dollars. Again, the first approach was to have separate user stories (PBIs) and representation for each product discipline on a small agile team. That is, have a requirements, design, test, CM, security, UX, performance, quality, etc. engineer and user stories for each discipline. For example, let's say, a user story said, "As a customer, I want to find, order, pay for, and receive a product from your catalog." In the agile approach, a programmer codes a catalog, search, order, payment, and shipping function (which sends you a notice when it has been delivered). A small team of XP

programmers can create half a dozen or more of these modularized functions in as little as two weeks. They may not be pretty, but they work (i.e., catalog products, search products, order products, pay for products, ship products, send notices, create account, customer service, etc.). In the Scrummerfall approach, an analyst documents the requirements ad-nausea, an architect creates an elaborate design, a tester creates a test plan, a coder implements the design, and an operations person deploys the code, etc.). In this manually intensive scenario, each user story would need a small team of product disciplines, fewer user stories could be delivered in a two-week sprint, or sprints would have to be longer (i.e., one or more months). And, of course, UX and security engineers would never be satisfied because they'd want months or years to plan and execute their approaches to perfection (i.e., diminishing returns). And, of course, all of this would be manual and none of this would be automated in the form of CI, CD, and DevSecOps pipelines.

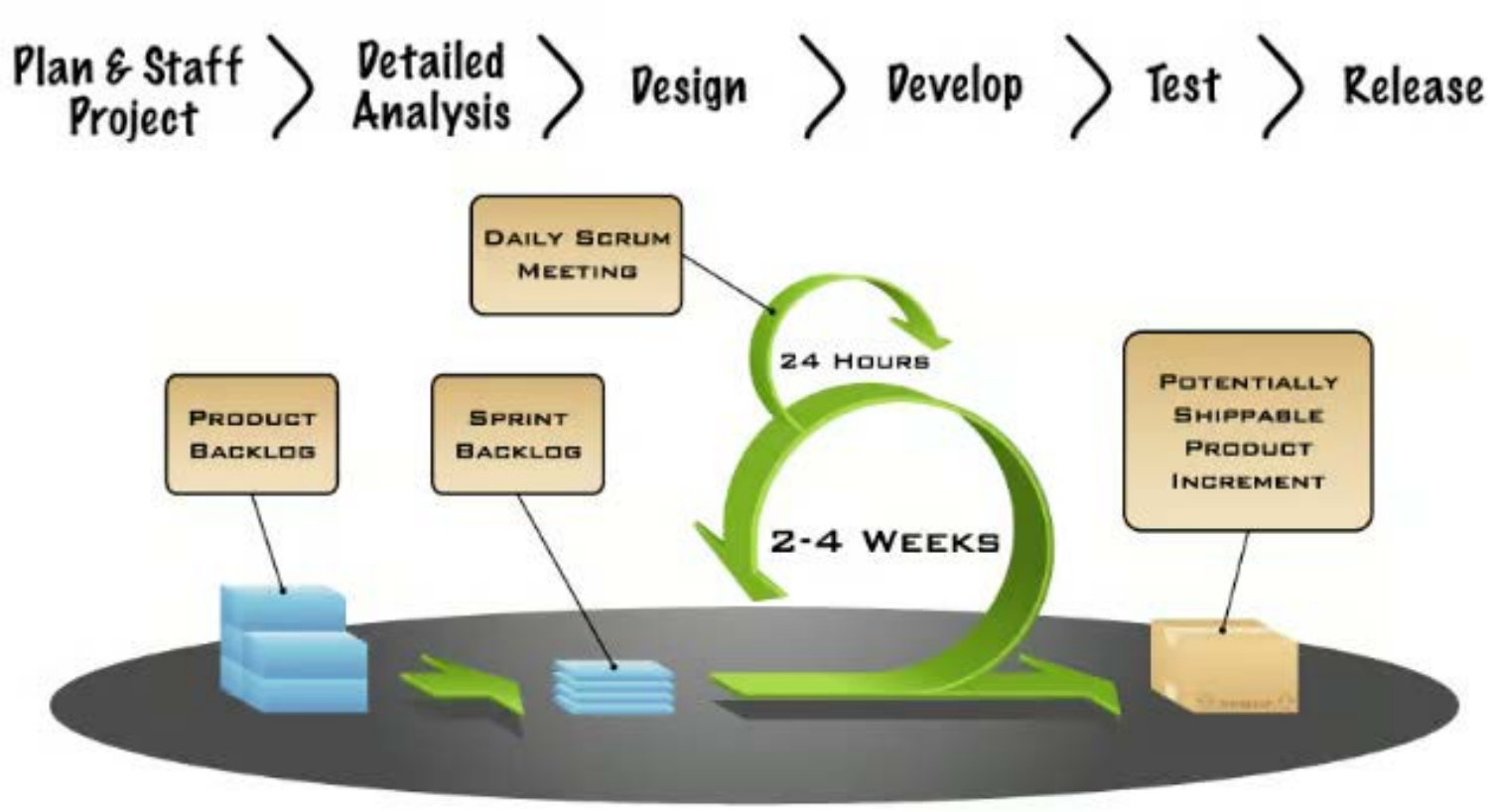

Figure 2. **Scrum** – Also a lightweight one-piece workflow system (used for Sprint Waterfalling)

(Phase III) **Dual Track Development**

A later, counterintuitive approach emerged just after Scrummerfalling, which was a minor refinement. One of the early complaints was that agile teams were not taking UX into consideration. Product owners were haphazardly capturing customer needs as user stories in product backlogs without proper UX disciplines. UX, based on UCD, was a long multidisciplinary discovery process with hundreds of techniques. UX may involve complex customer surveys, customer interviews, mockups and prototypes, journey maps, design thinking, empathy maps, etc. UX engineers wanted to observe customers in the wild, identify unsolved problems or pain points that would make customer lives easier, and then suggest solutions to these critical customer problems with high value. They wanted to delight customers with previously unsolved solutions. Henry Ford once said, "If you ask a customer what they want, they'll ask for a faster buggy." Similarly, UX engineers would observe the buggy and suggest an automobile, train, or plane to get from point A to point B faster, easier, and more efficiently and effectively. In this paradigm, the customer may not even know they have the problem with the buggy, but the UX engineer may be able to independently spot the need and say, "We

need an automobile, not a more comfortable buggy." Again, this process may take years. But, if UX engineers could direct agile teams toward high value solutions rather than random stories suggested by (disconnected) product owners, then developers may produce better code (i.e., stop producing code customers don't need). So, one solution was to have multiple parallel, perhaps, concurrent tracks. That is, have UX engineers take the customer temperature and feed high value user stories to developers. Perhaps, these would take place in alternating sprints. First, UX identifies high value stories in one sprint and developers code the high value stories in the next sprint, while UX developers are collecting more high value stories for future sprints. This is a somewhat superior approach to low-value product backlogs and stories produced by disconnected product owners. Of course, there are challenges with this approach (i.e., UX engineers want long lead times, UX engineers have a hard time doing UX in a sprint, and product managers don't want to invest in independent UX teams). You're basically doubling your costs to increase your value, while traditional product managers value producing lower value code quickly without UX discovery PBIs.

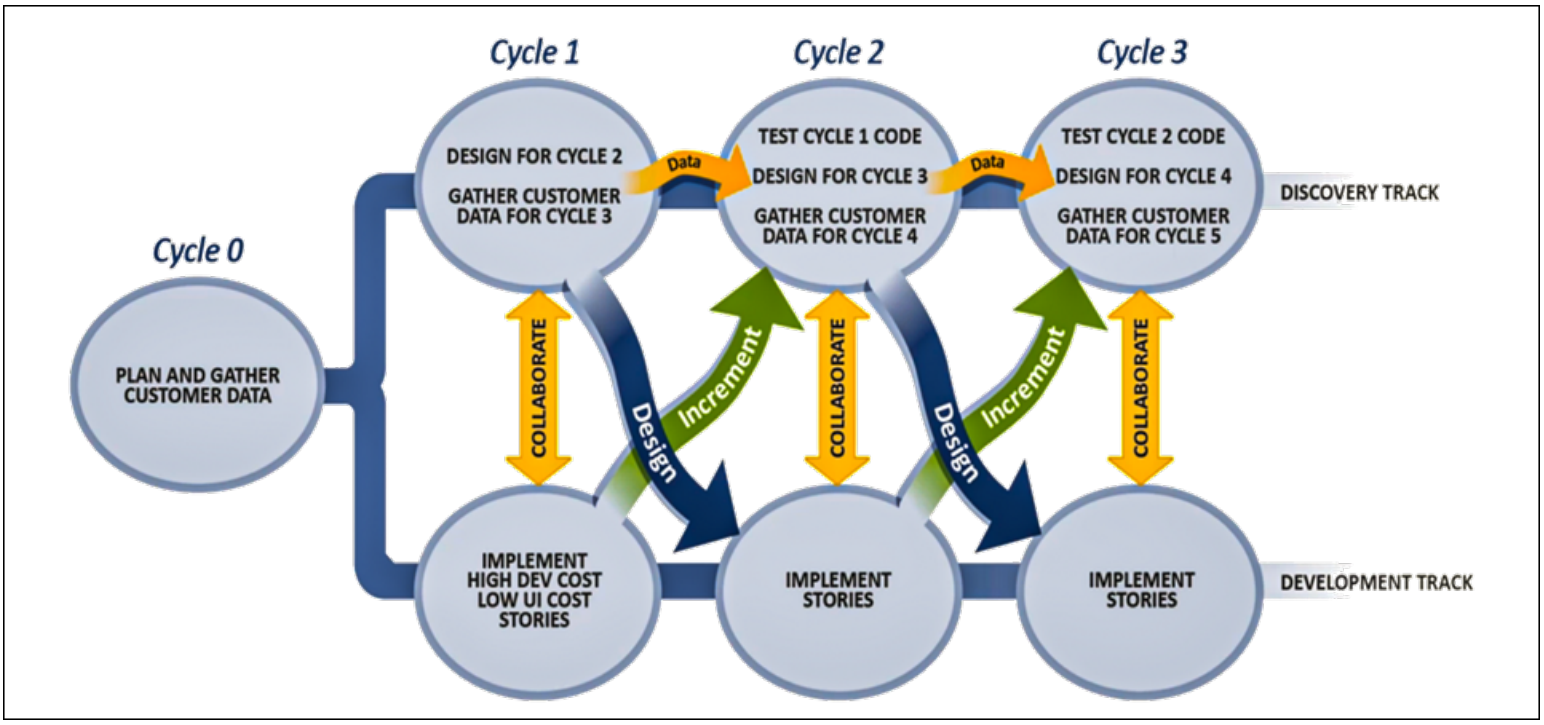

Figure 3. **Dual Track Development** (alternating discovery cycles to insert high value just-in-time UX PBIs)

The same was true of pair programming (i.e., "Two heads are better than one"). Two programmers can produce better code faster, but product managers didn't wanna pay for two programmers. They would rather pay for one developer to produce lower value code (or take a little longer if necessary). Two programmers produce great code in a few hours. Individual programmers working alone may take months or years to do the same job. Product managers feel like two programmers aren't worth the investment, but don't realize that individual programmers working alone cost more than a team of pair programmers. Worse yet, they might ask individual programmers to work on multiple user stories at one time, or even ask the programmers to work on multiple projects. In mathematical queuing theory, multitasking is more expensive, takes longer, and results in lower quality. In reality, so little time is spent on each user story or product that no user story will ever get done (i.e., it is completely blocked). Hence, disinvestment in pair programming, multitasking, and over allocation looks great in Excel, but is a failed concept in practice. You might ask, why this digression into pair programming economics? Dual track development is similar, we're gonna invest in a little collaborative UX discovery to speed up and increase quality and customer value. How does slowing down to apply UX practices speed up development. Well, if the developer arrives at a better solution sooner, then fewer product improvement cycles are necessary (i.e., do it right the first time). But there are still a few

more challenges to overcome (i.e., should UX engineers have their own agile team, backlog, or UX stories, should UX engineers be integrated directly into the agile team, and what happens if UX stories are bigger or have varying sizes?). For instance, what if a UX story is, "Conduct a 90-day survey so that customer needs are captured" (i.e., a UX PBI may take an entire release vs. sprint). One solution, is acknowledge the importance of UX stories, apply just-in-time UX stories that can be applied in a single sprint, integrate the UX engineers into the agile team, and integrate UX stories into the backlog). Furthermore, short, quick, and just-in-time UX chunks could be part of the definition-of-done, acceptance criteria, or subtasks in a user story or PBI.

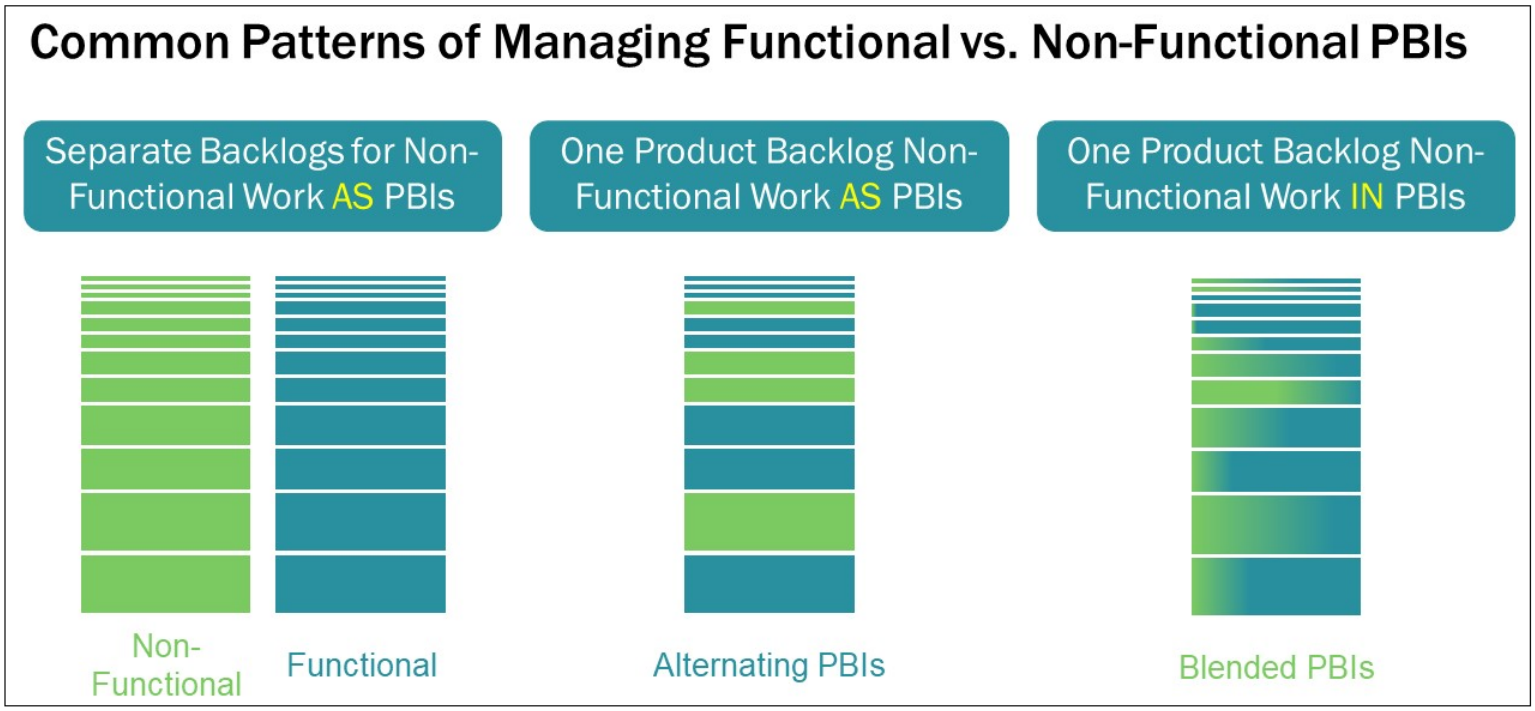

Figure 4. **Product Backlog Item** (chunking, integration, and optimization to insert high value discovery PBIs)

Of course, all we've talked about is UX up until this point. What about other product disciplines (i.e., testing, QA, performance, security, operations, etc.). The same paradigm for integrating better UX may be applied. Have all product disciplines participate in alternating, parallel, or concurrent sprints; integrate these disciplines directly into the agile teams; or, better yet, integrate the needs of all necessary product disciplines into the definition-of-done, acceptance criteria, or subtasks. Likewise, all product disciplines must develop bite-sized chunks that can fit into a single sprint (i.e., like asking UX engineers to apply bite-sized just-in-time UX chunks that fit into a single sprint). Like UX engineers, all product disciplines must trade off long lead time activities, for shorter bite-sized chunks. And, again, like 90-day UX customer surveys, some product discipline activities may be longer term epics with multiple stories performed over multiple sprints before they realize value. The even better solution is for all product disciplines to incorporate their requirements into the Cl, CD, and DevSecOps pipeline as automated test chunks. This applies to all disciplines like UX, test, quality, security, performance, operations, etc.). As developers check-in code into the version control repository, automated test chunks representing all disciplinary requirements, needs, and concerns should be executed. And, like early tactical XP Cl/CD practices, the code isn't done if the test chunks don't pass, or the build is broken. This forces developers to fix problems early when the cost of fixing non-conformances is low. Humans aren't perfect, mistakes will be made, and continuous improvement is always necessary. If a product discipline non-conformance slips through the DevSecOps pipeline, then a new automated test chunk must be added to prevent the release of non-conformant code. The answer is not in long-lead time product disciplines, but rather bite-sized just-in-time product discipline chunks and personnel integrated into agile teams and backlogs as definition-of-done, acceptance criteria, tasks, and occasionally

larger epics, features, and user stories or PBIs. Agile leadership teams should not be averse to discovery epics, features, stories, and PBIs that do not result in working code. Even developers need an occasional discovery and design story before diving into working code (before a series of working-code-only-PBIs). Some bite-sized just-in-time product discipline chunks (like one-week design sprints) should be executed in advance of implementation sprints (resulting in working software). And, of course, DevSecOps automated testing is the ultimate goal with a smattering of manual activities (i.e., the keyword here is, "smattering").

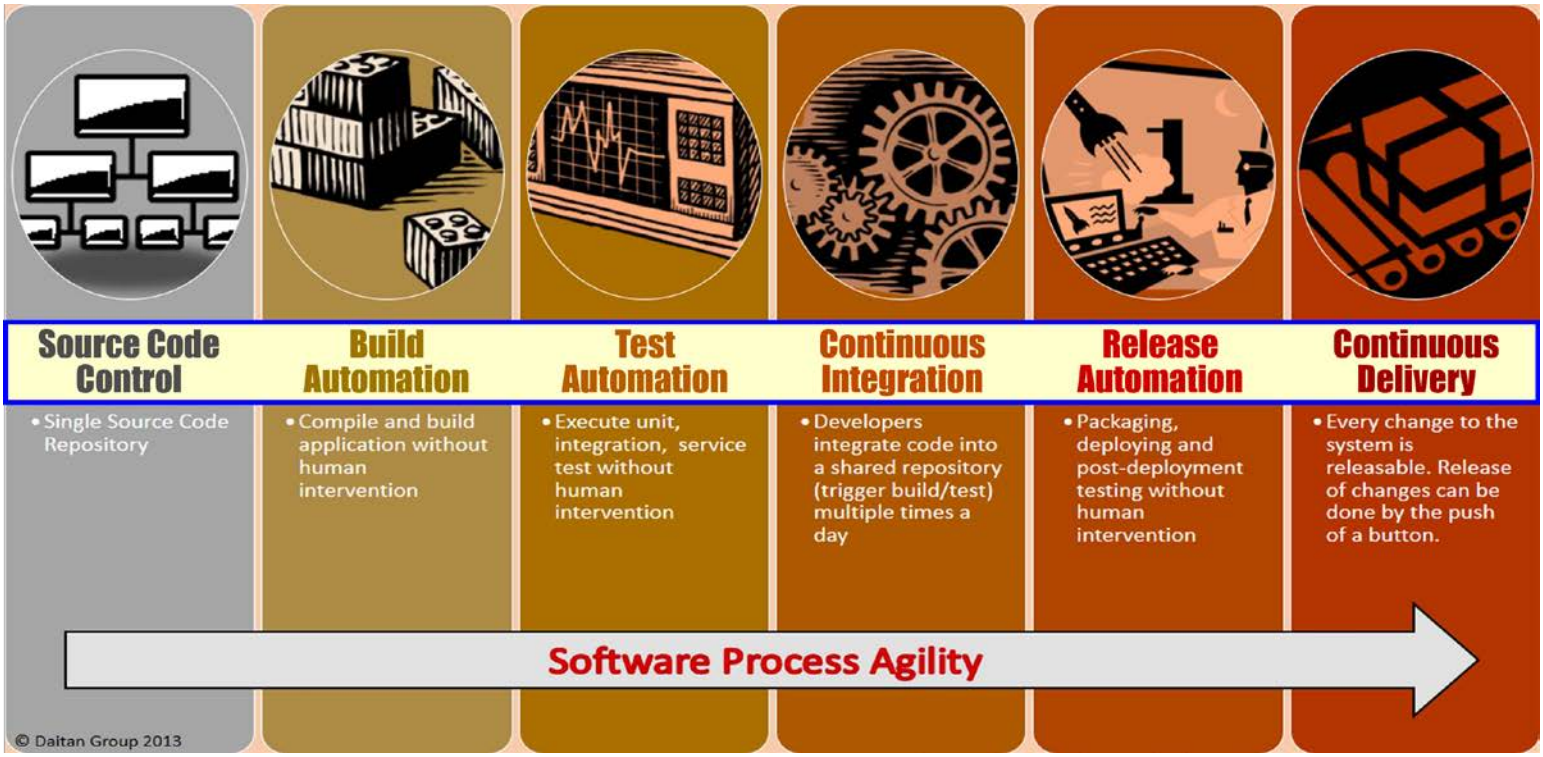

Figure 5. **DevSecOps Pipelines** (automated execution if tests representing bite-sized disciplinary chunks)

Summary

What's the bottomline? Well, the fate of next-century solution delivery is at stake, especially as it pertains to lightweight lean-agile thinking approaches like Scrum, XP, SAFe, DevSecOps, etc. That is, are lightweight lean-agile thinking frameworks efficient and effective (i.e., can sufficiently detailed solutions be rapidly created which incorporate the needs of multiple product disciplines and stakeholders like UX, testing, quality, performance, security, operations, etc.)? **Yes**. Do we need to revert back to old-school, traditional staged waterfall models which take decades and billions of dollars (so that product disciplines like UX, testing, quality, performance, security, operations, etc. have time to executive long-lead time activities)? **No**. Should there be two linear tracks for lean-agile frameworks (i.e., a first track for product disciplines like UX, testing, quality, performance, security, operations, etc. and a second track for implementation sprints to work off large backlogs of user stories or PBIs)? **No**, not necessarily, maybe, etc. In some cases, a product may need some implementation discovery sprints to collect leading innovation metrics before scaling to larger cost-intensive production implementation sprints). Should there be two parallel or concurrent tracks where product disciplines like UX, testing, quality, performance, security, operations, etc. conduct analysis, design, and backlog grooming, and a second track for production implementation sprints (with alternating sprints where the first track feeds user stories PBIs to implementation teams). **No**, not necessarily, maybe, etc. Some people believe Product Owners and Scrummasters are a first, parallel or concurrent track doing continuous backlog grooming (or the agile teams themselves should be doing ongoing grooming). Some believe quarterly or release planning events should be emergent with little ongoing preplanning (where

others advocate a first, parallel or concurrent track for product disciplines like UX, testing, quality, performance, security, operations, etc. to conduct analysis, design, and backlog grooming). Some of this may depend on the scope, scale, size, and duration of the product, service, or service product development initiative. That is, a small agile team may be able to incorporate product disciplines directly into user stories or PBIs as acceptance criteria or tasks, or even independent non-coding stories). In some cases, small, shared services teams may be formed for critical product disciplines. Ideally, user stories or PBIs can be formed by small teams with the necessary product discipline attributes, quickly implemented and deployed, and user feedback immediately measured. In conclusion, we'd argue that dual-track development principles are necessary for successfully applying light-lean agile frameworks without resorting to long-lead item activities, frameworks, and teams. Some throttling back of coding velocity may be necessary to improve the value of user stories and PBIs with investment in product disciplines like, but not limited to UX. In other words, two heads ARE better than one. Product discovery is not the anathema it has become to high-utilization leaders (i.e., product discovery is ongoing and never stops, so it must be properly managed with dual-track agile delivery principles and practices). Successful discovery and development of innovatively new products and services is a team sport (i.e., put the "we" back in "I"). High utilization leaders are experts at deconstructing teams to substitute short term cost optimization for greater customer value which is the goal of lean-agile thinking principles, practices, and tools. The ROI for integrated product disciplines is extremely high.

Further Reading

- **Adapting Usability Investigations for Agile User-centered Design** (May 2007)
https://uxpajournal.org/wp-content/uploads/sites/7/pdf/JUS_Sy_May2007.pdf

- **Dual Track Development is not Duel Track** (May 10, 2017)
https://jpattonassociates.com/dual-track-development

— *Dual-track Sprint/Iteration Planning Recipe* (Nov 1, 2019)

— *Dual-track Daily Standup* (Nov 1, 2019)

— *Dual-track Team Review and Retrospective* (Nov 1, 2019)

— *Dual-track Stakeholder Review* (Nov 1, 2019)

- **Dual-Track Agile** (Jan 7, 2020)
https://aktiasolutions.com/dual-track-agile

- **Dual Track Agile: The Secret Sauce to Outcome-based Development** (Feb 22, 2020)
https://medium.com/@daviddenham07/dual-track-agile-the-secret-sauce-to-outcome-based-development-601f6003ea73

- **Dual-Track Agile: The What, Why, Pros, and Cons** (Dec 7, 2020)
https://www.bornfight.com/blog/dual-track-agile-the-what-the-why-the-pros-the-cons

- **Dual Track Agile** (Jun 19, 2022)
https://agilefirst.io/dual-track-agile

Case Study

A new Scrum team of expert Java developers was formed to convert manual test procedures into automated tests in Java. Up until this point, some manual tests had been automated in a custom vendor specific user-friendly scripting language optimized for test automation. Product leadership teams wanted to free themselves from commercial scripting languages and test frameworks by directly coding the tests in Java. Few people on the large product team knew how to do this, so the new Scrum team was stacked with the top Java developers possible. That is, leadership knew this was a tough-nut-to-crack and were willing to allocate their best development talent to cracking-the-code of converting complex manual test cases into vanilla Java without the aid of user-friendly vendor specific scripting languages, frameworks, and testing libraries. The large product team itself had invested millions of dollars in constructing manual test cases which took thousands of staff hours to write and execute every 90 days to show customers the complex portfolio of products and services were thoroughly tested. Furthermore, product leadership was utilization driven meaning they wanted complex manual tests to be converted into automated tests in a single user story and two-week sprint. The leadership mantra was code as quickly as possible and do discovery on your own dime if necessary. That is, developers could spend all night doing discovery activities without pay, but daylight hours could only be billed or charged for coding activities. While the expert Java developers were not averse to spending some of their free unpaid time for product discovery activities, they weren't ready to invest 80 hours of their own time each sprint for free unpaid discovery activities. Thus, the new Java development team was at an impasse (i.e., how does each developer convert a complex manual test into an automated Java test with a single user story and two-week sprint). To add insult to injury, the development environment was highly volatile and unstable, so the development system was not available for half of each sprint (forcing developers to automate manual tests in only one week). The large product team's expert Java developers were befuddled while product leadership teams impatiently waited for the conversion to begin. The Scrummaster who was familiar with progressive agile techniques suggested each user story or PBI be implemented by pair programming which helped, and developers automate the tests in alternating sprints (i.e., developers would spend one sprint on discovery and the next sprint on implementation). This approach was an anathema to both product leadership teams as well as the expert Java developers who had full utilization concepts baked into their brains (i.e., each user story should result in validated working software at the end of each sprint by individual Java developers). However, due to the large degree of uncertainty by the expert Java developers, they reticently agreed to apply the Scrummaster's suggestions. Each team of pair programmers used the first sprint to analyze complex manual tests of which they had no prior knowledge. They also developed an architecture and design for implementing the tests as well as a reusable library of test widgets. This was a user interface intensive service product, so the test widgets allowed the developers to target the elements of the user interface (i.e., eliminate the need for costly manual user interface testing). In the second sprint, the pair programmers converted the manual tests into automated Java tests along with a test widget library. While awkward at first for a decade long team used to programming like automatons due to a utilization driven product leadership team, that is when the magic or impossible happened. The expert Java agile teams began producing new automated Java tests in droves like baby rabbits popping out of a top hat. The need for alternating discovery sprints ended after the first discovery sprint as the expert Java developers intuitively knew how to automate the next manual test without the fanfare of discovery user stories or PBIs. Furthermore, pair programming was no longer necessary, as individual developers accumulated enough expertise to automate complex manual tests alone which increased output exponentially. Furthermore, each sprint added new test widgets to the Java library, and the test widgets were hardened and generalized with each new test case. The test widgets and Java implementation approach, architecture, design, and architectural runway enablers also helped achieve yet another product leadership team goal. The teams developing the application itself could also write their own automated tests, utilize and harden the test widget library, and deliver tested code at the end of each release without the need for independent manual testing. So, both sets of teams were eliminating the need for expensive decade long manual independent testing (i.e., the expert Java teams automating the manual tests as well as the application developers writing new automated tests for new application code). Furthermore, the entire approach also enabled another product leadership goal (i.e., wrapping all tests into the product's first DevSecOps pipeline).