DANIEL UGOCHUKWU ONOVO

UB81898CO91115

BACHELOR IN COMPUTER SCIENCE

COURSE: AGILE SOFTWARE DEVELOPMENT

ATLANTIC INTERNATIONAL UNIVERSITY

HONOLULU, HAWAII U.S.A

JUNE, 2026

TABLE OF CONTENTS

1.0 INTRODUCTION TO AGILE SOFTWARE DEVELOPMENT

Agile software development is an umbrella term for a family of iterative and incremental methodologies that emphasize flexibility, collaboration, and delivering working software frequently. Rather than following a rigid, sequential plan, Agile teams embrace change, adapt continuously, and maintain a relentless focus on customer value.

In a simple term, Agile is a way of doing a big job by breaking it into small pieces, working together, checking often, and improving as you go.

1.1 A Brief History

To appreciate Agile, it helps to understand what it was reacting against. In the 1970s, Winston Royce described what later became known as the Waterfall model — a sequential, phase-gated approach to software development. While Royce actually warned against this model in his original paper, it became industry standard practice for decades.

By the 1990s, practitioners across the industry were independently developing lighter, more adaptive approaches: Kent Beck created Extreme Programming (XP) in 1996; Ken Schwaber and Jeff Sutherland presented Scrum at OOPSLA in 1995; and Alistair Cockburn articulated Crystal methodologies. Each recognized the same fundamental problem: heavyweight processes were failing teams and customers.

1.2 Waterfall vs. Agile

Understanding the contrast between Waterfall and Agile is essential context for everything that follows.

Waterfall	Agile
Sequential phases: Requirements > Design > Build > Test > Deploy	Iterative cycles delivering working software every 1-4 weeks
Requirements frozen at project start	Requirements evolve and are welcomed throughout the project
Customer involved mainly at the beginning and end	Customer is a continuous, active collaborator
Testing is a late-stage gate	Testing is continuous and integral to every iteration
Value delivered at project end (months or years away)	Value delivered incrementally from the very first sprint
Change is controlled and expensive	Change is embraced and managed at low cost
Success measured by adherence to plan	Success measured by working software and business outcomes
Risk is highest at project end (big-bang integration)	Risk is distributed and managed continuously throughout

1.3 Why Agile? The Business Case

The Standish Group's CHAOS reports have tracked software project success rates for decades. Their data consistently shows that large, plan-driven projects fail at alarmingly high rates — projects are canceled, massively over budget, or delivered late with reduced scope. Agile approaches correlate with dramatically better outcomes, particularly for complex and uncertain work.

The core business rationale for Agile:

- Faster time-to-market through incremental delivery of working features
- Reduced risk because problems are discovered and addressed early
- Higher customer satisfaction through continuous collaboration and feedback loops
- Improved team morale from autonomy, mastery, and clear purpose
- Better quality through continuous testing and technical excellence
- Adaptability to market changes, competitor moves, and new information

1.4 Benefits of Agile Methodology

Agile emerged as an alternative to the traditional software development approaches that were widely used during the 1990s. It was developed to address the limitations of rigid, sequential methods such as the Waterfall model. By emphasizing adaptability, collaboration, and continuous enhancement, Agile enables teams to respond effectively to changing project demands.

Some of the key benefits of Agile include:

✓ *Adaptability to Change:* Agile welcomes change throughout the project lifecycle. Since planning and development occur in short cycles, teams can regularly review priorities, update requirements, and adjust the product backlog to meet evolving business needs.

✓ *Suitable for Unclear or Evolving Requirements:* Agile is particularly effective when project objectives are not fully defined at the outset. As development progresses and more information becomes available, the team can refine goals and adjust the product direction accordingly.

✓ *Faster Delivery with Improved Quality:* Projects are divided into smaller, manageable iterations that allow teams to focus on delivering functional components quickly. Continuous testing and frequent reviews help identify defects early, resulting in higher-quality outputs and reduced rework.

✓ *Enhanced Team Collaboration:* Regular communication and close interaction among team members are fundamental principles of Agile. Frequent discussions, meetings, and collaborative activities help improve coordination and problem-solving.

✓ *Greater Customer Involvement:* Agile encourages active stakeholder participation throughout the project. Customers can review progress regularly, provide feedback, and influence product development, ensuring that the final solution aligns with their expectations.

✓ *Ongoing Learning and Improvement:* Continuous feedback from customers, stakeholders, and team members allows Agile teams to identify areas for improvement. Lessons learned during each iteration can be applied to future work, enhancing both processes and outcomes.

1.5 Limitations of Agile Methodology

Although Agile offers significant flexibility and responsiveness, it also presents certain challenges. Factors such as changing priorities, limited documentation, and evolving requirements can sometimes create difficulties in project management and delivery.

Some of the common drawbacks of Agile include:

- *Less Predictable Planning and Scheduling:* Because priorities are frequently reassessed and adjusted, it may be difficult to establish fixed delivery dates. New requirements can emerge during development, leading to additional iterations and potential extensions of the project timeline.
- *Dependence on Highly Skilled Team Members:* Agile teams are often small and cross-functional, requiring members to possess a broad range of competencies. Success depends heavily on the team's expertise, experience, and understanding of Agile principles and practices.
- *Increased Time Commitment*: Agile relies on continuous collaboration, regular meetings, and active stakeholder engagement. These activities can require more time and effort from developers and project participants compared to traditional development approaches.
- *Risk of Insufficient Documentation:* One of Agile's core values emphasizes working software over extensive documentation. While this can accelerate development, inadequate documentation may create challenges for maintenance, onboarding, knowledge transfer, and long-term project support. Therefore, organizations must strike an appropriate balance between documentation and direct communication.

2. THE AGILE MANIFESTO

In February 11-13, 2001 — Seventeen software practitioners gathered at The Lodge at Snowbird ski resort in Utah and produced the Agile Manifesto, a landmark document that named and unified these lightweight approaches.

The Agile Manifesto is a short but profound document. Every serious Agile practitioner should be able to recite its four values and understand its twelve principles. It is not a process or a prescription — it is a set of values and beliefs that guide decision-making in every situation.

2.1 The Four Core Values

VALUE 1: Individuals and interactions over processes and tools

This value does not dismiss processes and tools — it prioritizes the human element. The best process in the world will fail in the hands of a team that does not communicate, collaborate, and trust each other. Invest in your people; the tools will follow their lead.

VALUE 2: Working software over comprehensive documentation

Documentation has its place, but the primary measure of progress is software that actually runs and delivers value. Excessive documentation can become a substitute for real communication and create a false sense of progress. Write just enough to serve its purpose — no more, no less.

VALUE 3: Customer collaboration over contract negotiation

Traditional contracts attempt to specify everything upfront and then litigate when reality diverges from the plan. Agile replaces this adversarial dynamic with a collaborative partnership. The customer is not a requirements-source at the start — they are an ongoing partner throughout the entire development lifecycle.

VALUE 4: Responding to change over following a plan

In complex domains, plans are hypotheses. Reality continually provides feedback that should update those hypotheses. Teams that cling rigidly to outdated plans sacrifice outcomes on the altar of predictability. Plans are necessary — but they must remain subordinate to emerging reality and new information.

2.2 The Twelve Principles

The Manifesto's twelve principles provide more actionable guidance to teams:

1. Satisfy the customer through early and continuous delivery of valuable software.
2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference for the shorter timescale.
4. Business people and developers must work together daily throughout the project.
5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
7. Working software is the primary measure of progress.
8. Agile processes promote sustainable development. All parties should be able to maintain a constant pace indefinitely.
9. Continuous attention to technical excellence and good design enhances agility.
10. Simplicity — the art of maximizing the amount of work not done — is essential.
11. The best architectures, requirements, and designs emerge from self-organizing teams.
12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behaviour accordingly.

3. Agile Frameworks

The Agile Manifesto describes values and principles, not practices. Various frameworks have emerged to provide concrete structures for teams working in an Agile way. No single framework is universally superior — the best choice depends on team size, domain, organizational culture, and the nature of the work.

Framework	Origin	Best For	Key Idea
Scrum	Schwaber & Sutherland, 1995	Product dev teams (5-9 people)	Short sprints with defined roles, events, and artifacts
Kanban	Toyota / Anderson ~2007	Operational and flow-based work	Visualise work, limit WIP, improve flow continuously
XP (Extreme Programming)	Kent Beck, 1996	Teams needing technical excellence	Pair programming, TDD, CI, simple design
SAFe	Dean Leffingwell, 2011	Large enterprises (100+ people)	Scaling Agile across programs and portfolios
LeSS	Larman & Vodde, 2013	Medium-large Scrum scaling (2-8 teams)	One PO, one backlog, multiple Scrum teams
Nexus	Ken Schwaber, 2015	Scrum scaling (3-9 teams)	Integration team to manage cross-team dependencies
Crystal	Alistair Cockburn, 1990s	Small to medium teams	People-centric; adjust formality to team size
DAD	Scott Ambler, 2012	Enterprises needing flexibility	Hybrid toolkit for context-appropriate practices

4.0. SCRUM

Scrum is the most widely adopted Agile framework. The State of Agile reports consistently show Scrum or Scrum-hybrid approaches used by the large majority of Agile teams worldwide. Scrum is intentionally incomplete — it provides a framework of roles, events, and artifacts, but deliberately leaves implementation details to the team.

DEFINITION: Scrum is a lightweight framework that helps people, teams and organizations generate value through adaptive solutions for complex problems. — The Scrum Guide, 2020

4.1 The Three Scrum Accountabilities (Roles)

- Product Owner

The Product Owner (PO) is accountable for maximizing the value of the product and the work of the Scrum Team. This is a single person, not a committee. Key responsibilities include:

- Creating and communicating the Product Goal — the north star that guides the team
- Creating, ordering, and maintaining the Product Backlog with clarity and priority
- Making the Product Backlog transparent, visible, and understood by all
- Deciding what to release and when — accepting or rejecting completed work based on the Definition of Done

A common anti-pattern is a Product Owner who is absent, indecisive, or who delegates too much to developers. The PO must be empowered to make decisions and actively engaged with the team daily.

- Scrum Master

The Scrum Master is a servant-leader accountable for the Scrum Team's effectiveness. They are not a project manager, not a team lead, and not a status reporter. The Scrum Master:

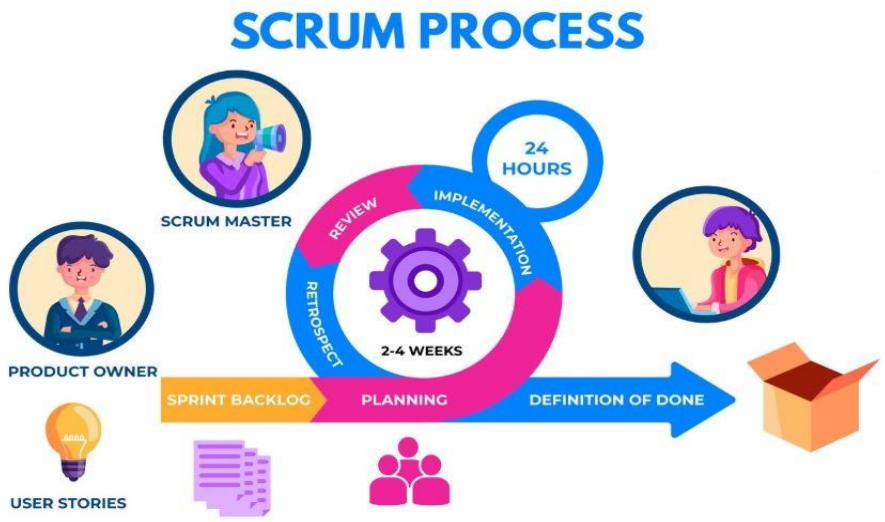
- Coaches the team in self-management and cross-functionality
- Helps the team focus on creating high-value Increments within the Definition of Done
- Removes impediments to the team's progress and shields them from external interference
- Ensures all Scrum events are positive, productive, and kept within their timebox
- Helps the broader organization understand and implement Scrum and empiricism

- Developers

Developers are the people committed to creating any aspect of a usable Increment each Sprint. They are responsible for creating the Sprint plan (Sprint Backlog), instilling quality by adhering to the Definition of Done, adapting their plan daily toward the Sprint Goal, and holding each other accountable as professionals.

4.2 The Five Scrum Events

Event	Purpose & Key Details
Sprint (1-4 weeks)	The heartbeat of Scrum. A fixed-length iteration creating a Done, usable Increment. Duration never changes once set. Only the Product Owner can cancel a Sprint.
Sprint Planning (max 8hrs/4-week sprint)	Answers: What can be done this Sprint (Sprint Goal + selected items)? How will chosen work get done (task decomposition into Sprint Backlog)?
Daily Scrum (15 min daily)	Developers inspect progress toward Sprint Goal and adapt the Sprint Backlog. Not a status report. Three questions: What did I do? What will I do? What is blocking me?
Sprint Review (max 4hrs/4-week sprint)	Team and stakeholders inspect the Increment and adapt the Product Backlog. Collaborative working session — not just a demo. Gathers feedback for next sprint.
Sprint Retrospective (max 3hrs/4-week sprint)	Inspect how the Sprint went: people, interactions, processes, tools, DoD. Identify and commit to improvements. The most impactful and most often skipped event.



4.3 The Three Scrum Artifacts

In Scrum, **artifacts** are the documents or information sources that provide transparency about the work being done. They help the Scrum Team and stakeholders understand what needs to be built, what is being worked on, and what has been completed.

According to the official Scrum Guide, there are three Scrum artifacts:

1. Product Backlog

An ordered list of everything that is known to be needed in the product. It is never complete — it evolves as the product and environment evolve. Items at the top are more refined, smaller, and ready for sprint selection. Items lower down are larger and represent future possibilities. Managed by the Product Owner.

2. Sprint Backlog

Composed of the Sprint Goal (the why), the selected Product Backlog items (the what), and an actionable plan for delivering the Increment (the how). A real-time picture of work Developers plan to accomplish during the Sprint. Belongs entirely to the Developers.

3. Increment

A concrete stepping stone toward the Product Goal. Each Increment is additive to all prior Increments and thoroughly verified — all Increments must work together. Multiple Increments may be created within a Sprint. The sum of all Increments is presented at the Sprint Review.

4.4 Definition of Done (DoD)

The Definition of Done is a formal description of the state of the Increment when it meets the quality measures required for the product. It creates shared understanding and transparency. If a Product Backlog item does not meet the DoD, it cannot be released or even presented at Sprint Review.

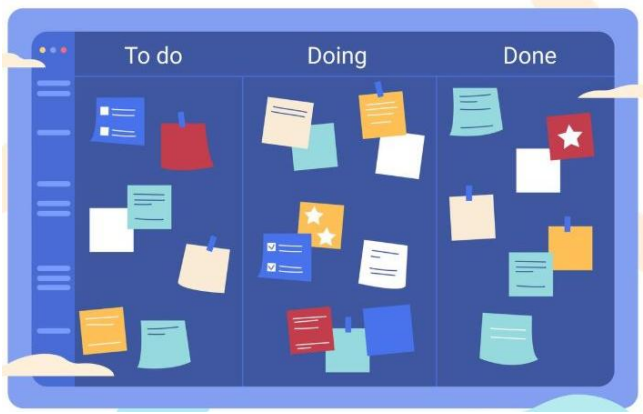
A sample Definition of Done for a software product team:

- All code peer-reviewed and approved via pull request
- Unit tests written and passing with >80% code coverage
- Integration tests passing in the CI pipeline
- No known defects of severity High or Critical
- Code merged to main branch and deployed to staging environment
- Acceptance criteria verified by Product Owner
- Relevant documentation updated (API docs, user guide, README)

Performance benchmarks met and security scan completed

5. KANBAN

Kanban is a method for managing and improving the flow of work. Unlike Scrum, it prescribes no specific roles, events, or iteration cycles. It starts from wherever you are and focuses on visualising work, limiting work-in-progress, and continuously improving flow. It was adapted for knowledge work by David J. Anderson around 2007, drawing from Toyota's production system.



5.1 The Six Core Kanban Practices

13. Visualize the Workflow — Map all stages of your process and represent every work item as a card on the board. Make invisible work visible.
14. Limit Work-in-Progress (WIP) — Set explicit limits on how many items can be in each stage simultaneously. This is the most powerful and counter-intuitive Kanban practice.
15. Manage Flow — Monitor, measure, and report on the flow of work. Look proactively for bottlenecks, blockers, and variability.
16. Make Policies Explicit — Define and publish the rules of your system. When does an item move? What are the entry and exit criteria for each stage?
17. Implement Feedback Loops — Review the system regularly using cadences: daily stand-up, replenishment meeting, service delivery review, operations review.
18. Improve Collaboratively, Evolve Experimentally — Use models and the scientific method to drive incremental, evidence-based improvement.

5.2 Understanding WIP Limits

WIP limits are the heart of Kanban. Little's Law from queuing theory states: Average Cycle Time = Average WIP divided by Average Throughput. If WIP is high, cycle time will be long. Reducing WIP directly reduces cycle time and improves predictability.

ANALOGY: Imagine a highway. Light traffic flows at full speed. As traffic increases, throughput briefly rises — but then gridlock sets in and throughput collapses. WIP limits

prevent the team from entering gridlock. The counterintuitive insight: doing less work simultaneously gets more done.

5.3 Kanban Metrics

Metric	Definition & Use
Lead Time	Time from when a request enters the system to when it is delivered. Represents the customer's experience of how long something takes.
Cycle Time	Time from when work actively begins on an item to when it is complete. Represents the team's internal delivery capability.
Throughput	The number of items completed per unit of time (e.g., items per week). Use to forecast future delivery rates with Monte Carlo simulation.
Work-in-Progress (WIP)	The count of items currently being worked on. High WIP is a leading indicator of long cycle times and is the primary lever for improvement.
Cumulative Flow Diagram	A chart showing count of items in each state over time. Widening bands indicate growing queues; flat lines indicate stoppages.
Aging WIP	How long current in-progress items have been active. Old items are risks; investigate and escalate proactively.

6. EXTREME PROGRAMMING (XP)

Extreme Programming is an Agile software development methodology created by Kent Beck in the mid-1990s, first applied on the Chrysler Comprehensive Compensation (C3) project in 1996, and formally published in 'Extreme Programming Explained: Embrace Change' in 1999. XP takes proven software engineering practices and turns their dials up to eleven — hence 'extreme'. It is the methodology most focused on technical excellence and the human side of software development simultaneously.

While Scrum is a project management framework that says little about engineering, XP is an engineering methodology that provides rich, specific guidance on how to write code. The two are highly complementary — many teams use Scrum's management structure with XP's engineering practices.

KEY INSIGHT: *XP was one of the first methodologies to recognize that the cost of change in software does not have to rise exponentially over time. With the right technical practices (automated tests, refactoring, continuous integration), the cost curve can be kept nearly flat — making late changes affordable rather than catastrophic.*

6.1 The Five XP Values

XP is grounded in five core values that shape every practice and decision:

XP Value	Meaning in Practice
Communication	Problems often stem from lack of communication. XP promotes face-to-face collaboration, pair programming, and practices that force knowledge sharing acr oss the team.
Simplicity	Do the simplest thing that could possibly work. Avoid speculative generality. Build what you need today, not what you might need tomorrow. Simpler code

	is easier to change.	
Feedback	Fast, frequent feedback from the system (tests), the customer (acceptance tests, demos), and the team (retrospectives). Feedback drives learning and adaptation.	
Courage	Courage to tell the truth about estimates and progress; courage to refactor code that needs it; courage to throw away code and start again; courage to raise concerns openly.	
Respect	Team members respect each other's contributions and expertise. Management respects that developers are professionals. Respect enables open communication and genuine collaboration.	

6.2 The Thirteen XP Practices

XP defines a rich set of complementary practices. They are designed to reinforce each other — removing any one weakens the others. Beck groups them into primary and corollary practices:

Primary Practices

- Sit Together — Co-locate the team as much as possible. Physical proximity dramatically improves communication and collaboration quality.
- Whole Team — Include all the skills needed on the team: developers, testers, business analysts, and a real customer representative. No silos.
- Informative Workspace — Make the team's work visible. Use physical boards, charts, and information radiators so anyone walking past can understand project status in 15 seconds.
- Energised Work — Work only as many hours as you can be productive and sustain. Tired developers make more mistakes, which cost more to fix. 40-hour weeks, no death marches.
- Pair Programming — Two developers, one keyboard. The driver writes code; the navigator reviews, thinks ahead, catches errors. Roles rotate regularly throughout the day.
- Stories — Express requirements as short user stories on cards. Stories are placeholders for conversations, not complete specifications. Detail emerges through dialogue.
- Weekly Cycle — Plan work one week at a time. At the start of each week, review progress, select stories for the week, and break stories into tasks. Deliver working software by week's end.
- Quarterly Cycle — Every quarter, reflect on where the team is, identify bottlenecks, plan the quarter's theme, and ensure the team is seeing the big picture.
- Slack — Always include some low-priority tasks in each week's plan that can be dropped without jeopardizing the week's commitment if things run long.
- Ten-Minute Build — The entire system should build and all tests should run in under 10 minutes. Slow builds are a symptom of poor architecture and discourage frequent integration.

- Continuous Integration — Integrate and test code changes multiple times per day. Never leave integration more than a few hours. A passing build is the heartbeat of the team.
- Test-First Programming (TDD) — Write an automated failing test before writing any production code. Red, Green, Refactor. Tests drive design and provide a safety net for change.
- Incremental Design — Design the system every day. Make design decisions just-in-time with the best available information. Keep the design as simple as the current requirements demand.

Corollary Practices (Advanced)

- Real Customer Involvement — An actual customer sits with the team to write stories, answer questions, and review results. Not a proxy — a real person with real authority.
- Incremental Deployment — Replace legacy systems gradually, one small piece at a time, rather than in a big-bang cutover.
- Team Continuity — Keep effective teams together. The value of a team increases over time; constantly breaking up high-performing teams destroys intellectual capital.
- Shared Code — Anyone on the team can improve any part of the code at any time. Collective code ownership eliminates knowledge silos and bottlenecks.
- Root-Cause Analysis — When a defect is found, fix it AND find its cause. Write an automated test that would have caught it. Change the process that allowed it to be created.

6.3 XP Roles

XP Role	Responsibilities
Customer	Writes stories, sets priorities, defines acceptance tests, answers questions immediately. Must be genuinely available to the team — not a distant stakeholder.
Developer	Estimates stories, writes production code and tests, pairs with other developers, integrates continuously, owns code quality collectively.
Tester	Helps the customer write acceptance tests; runs and monitors acceptance test results; helps the team understand what quality means in this context.
Coach	An experienced XP practitioner who guides the team in adopting XP practices, recognizes when the team is straying, and helps them course-correct. Often a senior developer.
Tracker	Monitors team progress against commitments; keeps velocity records; provides visibility into whether the project is on track without interfering with the team's work.

6.4 XP vs Scrum — Key Differences

Dimension	Scrum	Extreme Programming (XP)
Primary Focus	Project management and team coordination	Software engineering practices and technical quality
Iteration Length	1–4 weeks (fixed, called Sprint)	1–2 weeks (weekly or biweekly cycle)
Engineering Practices	Not prescribed (framework is silent)	Highly prescriptive: TDD, pair programming, CI, 10-min build
Roles	Product Owner, Scrum Master, Developers	Customer, Developer, Tester, Coach, Tracker
Scope Changes	Not during a Sprint; PO controls backlog	Customer can change priorities between weekly cycles
Code Ownership	Not specified (often individual)	Collective code ownership — everyone owns everything
Best Used When	Team needs coordination structure and cadence	Team needs to improve technical practices and quality

6.5 When to Choose XP

XP is the right choice when:

- The team is struggling with technical debt, low test coverage, or brittle code that is expensive to change
- Requirements are genuinely unclear and likely to change frequently, even during development
- The team has access to an engaged, available customer or customer representative
- The team is small (typically 2–12 developers) and co-located or willing to collaborate intensely
- The organization values technical excellence and is willing to invest in pairing and TDD despite short-term velocity impacts

NOTE: One does not need to adopt all 13 XP practices at once. Most teams start with TDD and Continuous Integration, which provide the safety net that makes other practices possible. Add practices incrementally as the team builds confidence and capability.

7. LEAN SOFTWARE DEVELOPMENT

Lean Software Development adapts the principles of Lean Manufacturing — pioneered by Toyota in their Toyota Production System (TPS) — to the domain of software development. It was introduced to the software world by Mary and Tom Poppendieck in their 2003 book 'Lean Software Development: An Agile Toolkit'. Lean is not a prescriptive process; it is a way of thinking about work — a philosophy that drives the elimination of waste, the amplification of value, and the continuous improvement of the system.

Lean predates the Agile Manifesto, but its thinking deeply influenced Agile. The Kanban method (Section 7) is one of the most direct software implementations of Lean thinking. SAFe's portfolio level draws heavily on Lean portfolio management. Understanding Lean is understanding the intellectual roots of much of modern Agile practice.

ORIGIN: The Toyota Production System was developed primarily by Taiichi Ohno and Shigeo Shingo from the 1940s through the 1970s. James Womack and Daniel Jones named and codified it as 'Lean' in their 1990 book 'The Machine That Changed the World'. The Poppendiecks translated these ideas into software development practice.

7.1 The Seven Lean Principles

Mary and Tom Poppendieck identified seven principles that translate Lean manufacturing thinking into software development:

Lean Principle	Software Development Application
1. Eliminate Waste	Anything that does not add value to the customer is waste. In software: partially done work, extra features, relearning, handoffs, delays, task switching, and defects are all waste to be identified and eliminated.
2. Amplify Learning	Software development is a learning process. Use short feedback cycles, automated tests, and iteration reviews to accelerate learning. Write code to discover requirements; do not over-specify before building.
3. Decide as Late as Possible	Defer irreversible decisions until the last responsible moment — when you have the most information. Avoid committing to a design, architecture, or feature before it is necessary.

4. Deliver as Fast as Possible	Speed reduces waste (carrying cost), accelerates feedback (learning), and increases value delivery. Fast delivery is achieved through small batch sizes, WIP limits, and automation — not by rushing or cutting corners.
5. Empower the Team	The people doing the work know the work best. Push decisions down to the people with the best information. Management sets direction and removes obstacles; teams decide how to do the work.
6. Build Integrity In	Perceived integrity: the product must feel right and coherent to the customer. Conceptual integrity: a simple, consistent architecture. Quality is built in through practices (TDD, code review), not inspected in at the end.
7. See the Whole	Optimise the whole system, not individual parts. Local optimization often degrades global performance. Use systems thinking; measure end-to-end flow (lead time), not activity at individual stages.

7.2 The Seven Wastes of Software Development

In Lean Manufacturing, waste (Japanese: 'muda') is classified into seven categories (TIM WOOD). Mary Poppendieck translated these into software equivalents, providing a powerful vocabulary for identifying and eliminating waste in software development:

Manufacturing Waste	Software Equivalent	Examples in Software	Impact
Inventory	Partially Done Work	Features coded but not tested, not deployed, not validated by users	Carries risk and cost with no delivered value
Over-production	Extra Features	Building features users did not ask for and will not use (the '64% never used' finding)	Wasted development effort; increases complexity
Over-processing	Relearning	Revisiting decisions, rediscovering requirements, re-doing analysis	Indicates poor knowledge management and documentation
Transportation	Handoffs	Throwing work 'over the wall' between dev, QA, ops, security	Each handoff loses information and adds delay
Waiting	Delays	Waiting for approvals, waiting for environments, waiting for decisions, blocked items	Kills flow; WIP accumulates; cycle time grows

Motion	Task Switching	Developers context-switching between multiple projects or stories simultaneously	Multitasking has 20-40% productivity cost per switch
Defects	Defects	Bugs found late in the process, requiring rework, bug triage, and customer escalations	The later a defect is found, the exponentially more expensive to fix

7.3 Value Stream Mapping

Value Stream Mapping (VSM) is a Lean tool for visualizing the end-to-end flow of value from customer request to customer delivery. It is one of the most powerful diagnostic tools available to software teams seeking to understand where waste accumulates in their processes.

A VSM exercise involves:

1. Define the value stream — the sequence of steps required to deliver a feature from concept to production.
2. Walk the board — trace the journey of a real recent work item through every step, from first request to live in production.
3. Record both process time (time actively being worked on) and lead time (total elapsed time including waiting) at each step.
4. Calculate efficiency — Process Time divided by Lead Time multiplied by 100. Most software teams are shocked to discover efficiency below 15%, meaning more than 85% of elapsed time is waste (waiting).
5. Identify the biggest sources of waiting and delay (the largest gaps between process time and lead time).
6. Design a future-state value stream that eliminates or reduces the identified waste sources.
7. Implement changes and re-measure to verify improvement.

FINDING: Software teams that perform their first Value Stream Map commonly find that a feature taking 6 weeks to deliver has less than 3 days of actual development work. The remaining 4+ weeks are wait states: waiting for approval, waiting for a deployment slot, waiting for QA to be available, waiting for stakeholder review. These findings are transformative.

7.4 The Concept of Flow

Lean's central operational concept is flow — work should flow smoothly and continuously through the value stream with minimal interruption, waiting, or batching. Flow is the opposite of the

traditional batch-and-queue approach where large amounts of work are accumulated before being passed to the next stage.

Principles for achieving flow in software development:

- Small batch sizes — deliver one story at a time rather than batching features into large releases
- Limit WIP — fewer items in progress simultaneously means less context switching and faster completion of each item
- Eliminate blockers actively — a blocked item is stopped flow; removing blockers is the highest priority
- Reduce handoffs — each handoff is a queue; co-locate or automate to eliminate unnecessary handoffs
- Visualise flow — a Kanban board makes flow (and its absence) visible to everyone

7.5 Lean vs. Agile — Relationship and Distinctions

Dimension	Lean	Agile
Origin	Toyota manufacturing system (1940s-70s)	Software industry practitioners (1990s-2001)
Scope	Entire organisation and value stream	Primarily software development teams
Focus	Eliminating waste, optimising flow, reducing cycle time	Iterative delivery, customer collaboration, adaptability
Level of Prescription	Principles and tools; highly adaptable	Varies by framework (Scrum is structured; Kanban is flexible)
Measurement Primary	Lead time, cycle time, value stream efficiency	Velocity, burndown, delivered business value
Change Philosophy	Kaizen — continuous, incremental, relentless improvement	Inspect and adapt — empirical process control
Relationship	Lean thinking informs and underpins many Agile practices	Agile is one specific application of Lean principles to software

7.6 Kaizen — Continuous Improvement

Kaizen (Japanese: 'change for better') is the Lean philosophy of continuous, incremental improvement. It is not about radical transformation but about every person on every team improving their work by a small amount every day. Aggregate over months and years, kaizen produces dramatic results.

In software development, kaizen manifests through:

- Sprint Retrospectives — dedicated time to reflect on process and commit to one specific improvement
- Blameless Post-mortems — after incidents or failures, focus on systemic causes, not individual blame
- Improvement Backlogs — track technical debt and process improvements as first-class backlog items alongside features
- Metrics and Measurement — measure the current state honestly, establish a baseline, track improvement over time

8. AGILE PLANNING

A common misconception is that Agile means no planning. In reality, Agile teams plan extensively — but at multiple levels of detail, and they revise plans continuously as they learn. The key insight is that all long-range plans are hypotheses, and hypotheses become more accurate as the team learns more through delivery.

6.1 The Planning Onion — Multiple Horizons

Level	Horizon | Who | Focus
Vision / Strategy	1-3 years | Executive, Product Leadership | What problem are we solving and for whom? What market are we in?
Portfolio / Roadmap	3-12 months | Product Management, Stakeholders | Which initiatives will we pursue and in what order? What are the outcomes we seek?
Release Planning	1-6 months | Product Owner, Team | What will we deliver in the next release? What are our key milestones and commitments?
Sprint / Iteration Planning	1-4 weeks | Scrum Team | What will we deliver in this Sprint? What is our Sprint Goal and backlog selection?
Daily Planning	24 hours | Developers | What will I work on today? What is blocking me? How do we adapt our plan to reach the Sprint Goal?

6.2 User Story Maps

A User Story Map (invented by Jeff Patton) organizes user stories in a two-dimensional grid to give the team a shared understanding of the user's journey. The horizontal axis represents the user's workflow (the 'backbone'). The vertical axis represents depth — alternative and detailed ways to complete each activity. Horizontal slices represent releases.

Story maps are invaluable because they reveal the full scope in a human-centred way, facilitate conversations about what to build and why, make it easy to identify the minimum viable slice of each release, and help teams see what they are deliberately not building.

6.3 Estimation Techniques

- Story Points

Story points are a relative measure of effort, complexity, and uncertainty — not hours. A story that is twice as complex as another gets twice as many points. The Fibonacci sequence (1, 2, 3, 5, 8, 13, 21) is commonly used because the gaps between numbers reflect the inherent uncertainty in larger estimates.

- Planning Poker

A consensus-based estimation technique. Each team member privately selects a card representing their estimate. All cards are revealed simultaneously. If estimates diverge significantly, the highest and lowest estimators explain their reasoning, and the team estimates again. This surfaces hidden assumptions and produces more accurate estimates through dialogue.

- T-Shirt Sizing

A lightweight technique using relative sizes (XS, S, M, L, XL, XXL) to quickly estimate large backlogs. Useful for high-level roadmap planning where relative ordering and feasibility matter more than precision.

- Affinity Mapping / Group Estimation

Teams silently sort stories into relative size groups, then discuss groupings. Faster than planning poker for large backlogs; useful for initial sizing of an entire product backlog in a single session.

9. USER STORIES AND BACKLOG MANAGEMENT

9.1 What Is a User Story?

A user story is a short, simple description of a feature told from the perspective of the person who desires the capability. Mike Cohn popularised the standard template: 'As a [type of user], I want [some goal] so that [some reason/benefit].'

EXAMPLE: As an online shopper, I want to save items to a wish list so that I can purchase them at a later date when I have budget available.

Bill Wake's INVEST criteria define a well-formed user story:

- Independent — Can be developed and delivered independently of other stories (ideally)
- Negotiable — The story is not a contract; details are negotiated between PO and developers
- Valuable — Delivers value to a user or stakeholder; avoid purely technical stories with no user benefit
- Estimable — The team can estimate the work; if not, it needs more discussion or splitting
- Small — Can be completed within a single Sprint by the team
- Testable — Acceptance criteria can be defined that determine when the story is done

9.2 Writing Acceptance Criteria (Given-When-Then)

Acceptance criteria define the conditions under which a story is considered complete. The most common format is Given-When-Then (Gherkin syntax), which describes system behaviour from a user perspective:

EXAMPLE: GIVEN I am a logged-in user viewing my wish list | WHEN I click 'Add to Cart' on a wish list item | THEN the item is added to my shopping cart AND the item is removed from my wish list AND I see a success confirmation message

9.3 Epics, Themes, and the Story Hierarchy

Level	Description & Example
Theme	A strategic objective or broad business capability. Example: 'Customer Self-Service Portal'
Epic	A large feature spanning multiple sprints. Example: 'Account Management'. Must be split into stories before development.
Story	A unit of work completable in one sprint. Example: 'As a user, I can reset my password via email link.'
Task	A technical sub-task of a story measured in hours. Example: 'Implement password reset email HTML template.'

9.4 Backlog Refinement (Grooming)

Backlog refinement is an ongoing process where the Product Owner and Developers review the Product Backlog. During refinement the team: adds detail and estimates to upcoming items; splits large epics into smaller, actionable stories; reviews and updates estimates as understanding improves; removes items that are no longer relevant; and clarifies acceptance criteria to resolve open questions.

The Scrum Guide recommends spending no more than 10% of the team's capacity on backlog refinement. Many teams schedule a dedicated weekly refinement session of 60-90 minutes, typically mid-sprint.

10. AGILE ENGINEERING PRACTICES

The Agile Manifesto's ninth principle states: 'Continuous attention to technical excellence and good design enhances agility.' Agile is not just about process — the engineering practices that enable teams to deliver high-quality software quickly and sustainably are equally important. Without technical excellence, Agile becomes 'Fragile.'

10.1 Test-Driven Development (TDD)

TDD is a development practice where tests are written before the code they test. The cycle is:

8. RED — Write a failing test for the desired behaviour (it must fail first to be valid)
9. GREEN — Write the minimum code necessary to make the test pass (not more)
10. REFACTOR — Improve the code's design without changing its external behaviour

KEY INSIGHT: TDD is not primarily about testing — it is a design technique that uses tests to drive better software design. The tests are a valuable by-product; the improved design and reduced coupling are the primary benefits.

TDD benefits: forces clear thinking about requirements before coding; creates a comprehensive regression suite automatically; encourages small, focused, well-designed code units; provides immediate feedback when changes break existing behaviour; and documents intended system behaviour through executable tests.

10.2 Continuous Integration (CI)

Continuous Integration is the practice of frequently merging code changes into a shared repository — ideally multiple times per day. Each integration is verified by an automated build and test suite, allowing teams to detect problems early when they are cheapest to fix.

Key CI practices and rules:

- Maintain a single source repository with a mainline (trunk) branch that is always deployable
- Automate the build — anyone can build the product with a single command, no manual steps
- Make the build self-testing — the build runs all automated tests automatically
- Every commit to mainline triggers the CI pipeline within minutes
- Fix broken builds immediately — a broken build is the team's absolute top priority
- Keep the build fast — a slow build (over 10 minutes) discourages frequent integration

10.3 Continuous Delivery and Deployment (CD)

Practice	Description & Implications
Continuous Integration	Code integrated and verified automatically on every commit. Goal: detect integration failures immediately.
Continuous Delivery	Code is always in a releasable state. Deployment to production is a business decision, not a technical constraint. Requires comprehensive automated testing.
Continuous Deployment	Every change that passes all automated tests is deployed to production automatically with no human intervention. Requires very high test coverage and feature flags.

10.4 Pair Programming

Pair programming is an XP practice where two developers work together at one computer. One drives (writes code) while the other navigates (reviews, thinks about design and edge cases). Roles are switched frequently. Benefits: continuous real-time code review; knowledge sharing across the team; higher design quality through two minds; collective code ownership; and reduced bus factor.

10.5 Refactoring

Refactoring is improving the internal structure of existing code without changing its external behaviour. It is not rewriting — it is disciplined, incremental improvement guided by comprehensive tests. Common refactorings include Extract Method, Rename for Clarity, Extract Class, and Replace Conditional with Polymorphism.

RULE: Refactoring requires a comprehensive automated test suite to be safe. Never refactor without tests — you have no way to verify you have not changed behaviour. This is why TDD and refactoring go hand-in-hand.

11. AGILE METRICS

Measuring the wrong things is worse than measuring nothing — it creates perverse incentives that destroy value. Agile metrics must drive improvement, not create gaming behaviour. Velocity, for example, is a planning tool, not a performance metric. A team that games its velocity to appear more productive is actively destroying the team and organisation.

11.1 Velocity

Velocity is the amount of work (in story points) a team completes in a Sprint. It is used for capacity planning and release forecasting only. Rules: never compare velocity between teams; use a rolling average of the last 3-5 sprints for planning; never pressure teams to increase velocity — it leads to estimate inflation, not real improvement.

11.2 Burndown and Burnup Charts

A Sprint Burndown chart shows remaining work in a Sprint over time. A straight diagonal from Sprint start to zero at Sprint end is the ideal line. Real teams produce irregular lines — this is expected and useful information about how work was discovered and completed.

A Release Burnup chart shows progress toward a release goal over time. Unlike burndown, it makes scope changes explicitly visible — when the product backlog grows, the burnup chart shows this clearly, making scope creep impossible to hide from stakeholders.

11.3 Cumulative Flow Diagram (CFD)

The CFD is the most powerful Kanban metric. It shows the count of items in each workflow state over time. Reading a CFD: the vertical distance between two lines = WIP in that stage; the horizontal distance = average cycle time; a widening band indicates a growing queue or bottleneck; a flat horizontal line indicates work has stopped flowing through that stage.

11.4 Key Metrics at a Glance

Metric	What It Measures	Primary Use	Warning Sign
Velocity	Story points per sprint	Sprint capacity planning	Increasing without real improvement
Lead Time	Request to delivery duration	Customer SLA management	Consistently above agreed target
Cycle Time	Active start to done duration	Process efficiency measure	High variance (unpredictability)
Throughput	Items completed per period	Delivery rate forecasting	Declining trend over time
Escaped Defects	Bugs found post-release	Quality and DoD health	Any rising trend
Deployment Frequency	How often code ships to prod	CI/CD health indicator	Less than once per sprint

12. SCALING AGILE

Agile was originally conceived for small, co-located teams. Most real enterprises have hundreds or thousands of engineers working on large, complex systems. Scaling frameworks attempt to extend Agile principles to these contexts without losing the core benefits of adaptability, feedback, and team autonomy.

WARNING: Scaling frameworks are complex and expensive to implement. Before scaling, first ensure individual teams are genuinely Agile. Scaling dysfunction amplifies existing problems rather than solving them. Fix the team level first.

12.1 SAFe — Scaled Agile Framework

SAFe is the most widely adopted large-scale Agile framework. It operates at four levels:

- Team Level — Scrum teams deliver working software every sprint (2 weeks typically)
- Program Level — Agile Release Trains (ARTs) align up to 12 teams around a shared 10-week Program Increment (PI)
- Large Solution Level — Multiple ARTs coordinated for very large, complex solutions
- Portfolio Level — Strategic themes and Lean-Agile Budgets align investment with strategy

SAFe's central innovation is PI Planning — a two-day face-to-face event where all teams on an ART plan together, identify dependencies, and commit to shared objectives. SAFe's critics note it can become overly prescriptive and heavyweight, recreating waterfall at scale.

12.2 LeSS — Large-Scale Scrum

LeSS takes a minimalist approach: rather than adding roles and processes, it scales by applying basic Scrum principles more broadly. Key ideas: one Product Owner manages one Product Backlog for all teams; each team runs their own Scrum events; an Overall Retrospective brings all teams together; no additional management layers or coordination roles are added.

12.3 The Spotify Model (Squads, Tribes, Chapters, Guilds)

- Squads — Small, autonomous, cross-functional teams (~8 people, similar to a Scrum team)
- Tribes — A collection of squads working in related areas (fewer than 100 people)
- Chapters — Horizontal groupings of people with the same specialty across squads (e.g., all backend engineers)

- Guilds — Voluntary communities of interest spanning the entire organisation

IMPORTANT: Spotify has publicly stated they no longer use the 'Spotify Model' as described in their 2012 blog posts. Organisations should not copy the model — they should understand the underlying principles of autonomy, alignment, and trust, then design their own system contextually.

13.0 SCRUM IN PRACTICE

Building TaskFlow Pro

A Complete Scrum Case Study

PHP • MySQL • JavaScript

3 Sprints • 6 Weeks • Full Development Lifecycle

Let's delve into a complete, end-to-end case study demonstrating how Scrum methodology is applied to the real-world development of TaskFlow Pro — with a web-based project and task management application built with PHP, MySQL, and JavaScript. Every Scrum artifact, ceremony, role, and concept introduced in the tutorial is applied concretely here, with actual user stories, sprint plans, code snippets, and retrospective outcomes.

Application: TaskFlow Pro

TaskFlow Pro is a team productivity application that allows small organisations (5–50 users) to create projects, assign tasks, track progress on Kanban-style boards, and receive deadline notifications. It is a realistic, non-trivial application with authentication, a relational data model, a RESTful API backend, and a dynamic frontend — exactly the kind of project where Scrum delivers measurable value.

Dimension	Details
Application Name	TaskFlow Pro — Team Task & Project Management
Technology Stack	Backend: PHP 8.2 (OOP, MVC pattern) | Database: MySQL 8.0 | Frontend: Vanilla JavaScript (ES6+), HTML5, CSS3
Architecture	RESTful API (PHP) + Single-Page Application (JavaScript) + MySQL relational DB
Target Users	Small business teams (5–50 users) needing collaborative task management
Key Features	User authentication, project management, task boards, task assignment, due date tracking, email notifications, reporting dashboard
Project Duration	6 weeks | 3 sprints of 2 weeks each
Team Size	5 people (Product Owner, Scrum Master, 3 Developers)
Definition of Done	Code reviewed, unit tested, merged to main, deployed to staging, AC verified by PO

Part 1: Team Setup & Product Vision

13.1.1 The Scrum Team

Before the first sprint begins, the Scrum Team is formed and each accountability is established. The team is small enough to communicate effectively but carries all the skills needed to deliver a complete, working product increment each sprint.

Role	Name	Background	Key Responsibilities
Product Owner	Sarah Chen	Business analyst with 4 years experience in SaaS products. Deep knowledge of the target user base.	Owns and prioritises the Product Backlog; writes and accepts user stories; defines Sprint Goals; liaison with stakeholders.
Scrum Master	James Okafor	Certified PSM I practitioner with 3 years of Scrum coaching experience.	Facilitates all Scrum events; removes impediments; coaches team on Scrum values; protects team from external disruptions.
Developer — Backend	Priya Sharma	PHP developer, 5 years experience. Specialist in MVC architecture and REST API design.	Implements PHP backend: API endpoints, business logic, database layer, authentication.
Developer — Frontend	Carlos Reyes	JavaScript developer, 3 years experience. Strong in ES6+, DOM manipulation, and CSS.	Implements JavaScript SPA: UI components, API integration, state management, UX.
Developer — Full Stack	Aisha Okonkwo	2 years experience. PHP and JavaScript. Handles database design, testing, and integration.	Database schema design and migrations; integration testing; supports both backend and frontend as needed.

13.1.1 Team Working Agreements

Before Sprint 1, the team establishes working agreements — explicit norms that guide behaviour and prevent misunderstandings:

- Daily Scrum: 9:00 AM every weekday, 15 minutes maximum, standing up
- Core hours: All team members available 10:00 AM – 4:00 PM for collaboration
- Sprint length: 2 weeks (10 workingS days)
- Definition of Done: see Section 1.2 below — applies to every story, every sprint
- Branching strategy: feature branches off main; PRs require one approval before merge

- No story is 'done' until it passes all acceptance criteria AND the Definition of Done
- Blockers must be raised at the Daily Scrum or in the team Slack channel immediately — never held until the next day

13.1.2 Definition of Done (DoD)

TEAM DOD: A story is Done when: (1) all acceptance criteria pass, (2) code is peer-reviewed and PR merged to main, (3) PHPUnit tests written and passing, (4) no PHP warnings or JS console errors, (5) feature deployed to staging environment, (6) Product Owner has verified the story in staging.

13.2. Product Vision & Product Backlog

13.2.1 Product Vision Statement

FOR small teams that struggle with scattered task management,

TaskFlow Pro IS A web-based project management tool

THAT provides a centralised, real-time view of all team tasks and deadlines,

UNLIKE spreadsheets and email threads, **OUR PRODUCT** reduces missed deadlines by 60% in 90 days.

13.2.2 Initial Product Backlog (Prioritised)

The Product Owner works with the team during initial backlog refinement to create, estimate, and prioritise the following user stories. Story points use the Fibonacci scale. Priority is MoSCoW: Must Have (M), Should Have (S), Could Have (C), Won't Have this release (W).

ID	User Story	Priority	Points	Sprint
US-01	As a new user, I can register an account with email and password so that I can access the application.	M	3	1
US-02	As a registered user, I can log in and log out securely so that my data is protected.	M	3	1
US-03	As a team member, I can create a new project with a name and description so that I can organise my work.	M	5	1
US-04	As a team member, I can create tasks within a project with a title, description, and due date.	M	5	1

US-05	As a task creator, I can assign a task to any project member so that ownership is clear.	M	3	1
US-06	As a team member, I can view all tasks in a Kanban board (To Do / In Progress / Done) so that status is visible.	M	8	2
US-07	As a team member, I can drag and drop tasks between Kanban columns to update their status.	M	5	2
US-08	As a team member, I can add comments to a task so that the team can communicate in context.	S	5	2
US-09	As a project manager, I can invite team members to a project via email so that collaboration is possible.	M	5	2
US-10	As a team member, I receive an email notification when a task is assigned to me.	S	3	2
US-11	As a team member, I can filter tasks by assignee, status, or due date so that I can find relevant tasks quickly.	S	5	3
US-12	As a project manager, I can view a dashboard showing project progress, overdue tasks, and team activity.	S	8	3
US-13	As a team member, I can upload a file attachment to a task (max 10MB).	C	5	3
US-14	As a project manager, I can set task priority (Low / Medium / High / Critical).	S	3	3
US-15	As an admin, I can deactivate a user account so that former team members lose access.	S	3	3

BACKLOG NOTE: Total estimated backlog = 70 story points. Team average velocity (based on similar past projects) is estimated at 20-24 points per sprint. Three sprints of 2 weeks each should deliver the full MVP. Backlog will be refined at the start of and during each sprint.

Part 2: Sprint 1 — Foundation (Weeks 1–2)

Sprint 1: Foundation — Auth, Projects & Tasks

Sprint 1 Planning

Sprint 1 Details	
Sprint Goal	Users can register, log in, create projects, create tasks, and assign tasks — the core data model and authentication layer are working end-to-end in staging.
Duration	2 weeks: Day 1–10
Selected Stories	US-01, US-02, US-03, US-04, US-05
Committed Points	19 story points
Sprint Planning Date	Monday, Week 1, 9:00 AM — 2 hours
Key Risk	Database schema changes late in the sprint could invalidate earlier work — mitigated by finalising schema on Day 1.

Sprint 1 — Sprint Backlog (Task Breakdown)

Story	Task	Owner	Est. Hours	Status Day 10
US-01: Register	Design MySQL users table schema	Aisha	2h	Done
US-01: Register	Build POST /api/auth/register endpoint (PHP)	Priya	4h	Done
US-01: Register	Create registration form UI (HTML/JS)	Carlos	3h	Done
US-01: Register	Client-side validation (JS) + server-side validation (PHP)	Carlos / Priya	3h	Done
US-01: Register	Write PHPUnit tests for register endpoint	Aisha	2h	Done
US-02: Login	Build POST /api/auth/login — JWT token response	Priya	4h	Done

US-02: Login	Implement PHP session / JWT middleware	Priya	3h	Done
US-02: Login	Login form UI + token storage in localStorage	Carlos	3h	Done
US-02: Login	Logout — clear token + redirect	Carlos	1h	Done
US-03: Projects	Design projects table + project_members junction table	Aisha	3h	Done
US-03: Projects	CRUD API: GET/POST /api/projects, GET /api/projects/{id}	Priya	5h	Done
US-03: Projects	Projects list page + create project modal (JS)	Carlos	4h	Done
US-04: Tasks	Design tasks table (FK to projects, assignee)	Aisha	2h	Done
US-04: Tasks	CRUD API: GET/POST /api/tasks, PUT /api/tasks/{id}	Priya	5h	Done
US-04: Tasks	Create task form UI with due date picker (JS)	Carlos	4h	Done
US-05: Assign	Add assignee_id FK to tasks; update API	Priya	2h	Done
US-05: Assign	Assignee dropdown populated from project members	Carlos	2h	Done
US-05: Assign	Integration tests: create project → task → assign	Aisha	3h	Done

Sprint 1 — Key Technical Implementations

Database Schema (MySQL)

Aisha finalised the core schema on Day 1, ensuring all developers could work in parallel without dependency conflicts. The schema covers authentication, projects, and tasks:

```
-- users table
CREATE TABLE users (
  id          INT AUTO_INCREMENT PRIMARY KEY,
  name        VARCHAR(100)        NOT NULL,
```

```
  email       VARCHAR(150)        UNIQUE NOT NULL,
  password    VARCHAR(255)        NOT NULL,  -- bcrypt hash
  role        ENUM('admin','member') DEFAULT 'member',
  is_active   TINYINT(1)          DEFAULT 1,
  created_at  TIMESTAMP           DEFAULT CURRENT_TIMESTAMP
);

-- projects table
CREATE TABLE projects (
  id          INT AUTO_INCREMENT PRIMARY KEY,
  name        VARCHAR(200)        NOT NULL,
  description TEXT,
  owner_id    INT                 NOT NULL,
  created_at  TIMESTAMP           DEFAULT CURRENT_TIMESTAMP,
  FOREIGN KEY (owner_id) REFERENCES users(id) ON DELETE CASCADE
);

-- project_members junction table
CREATE TABLE project_members (
  project_id  INT NOT NULL,
  user_id     INT NOT NULL,
  joined_at   TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  PRIMARY KEY (project_id, user_id),
  FOREIGN KEY (project_id) REFERENCES projects(id) ON DELETE CASCADE,
  FOREIGN KEY (user_id)    REFERENCES users(id)    ON DELETE CASCADE
);

-- tasks table
CREATE TABLE tasks (
  id          INT AUTO_INCREMENT PRIMARY KEY,
  project_id  INT           NOT NULL,
  title       VARCHAR(200)  NOT NULL,
  description TEXT,
  assignee_id INT,
  status      ENUM('todo','in_progress','done') DEFAULT 'todo',
  priority    ENUM('low','medium','high','critical') DEFAULT 'medium',
  due_date    DATE,
  created_by  INT           NOT NULL,
  created_at  TIMESTAMP     DEFAULT CURRENT_TIMESTAMP,
  updated_at  TIMESTAMP     DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
  FOREIGN KEY (project_id)  REFERENCES projects(id)  ON DELETE CASCADE,
  FOREIGN KEY (assignee_id) REFERENCES users(id)     ON DELETE SET NULL,
  FOREIGN KEY (created_by)  REFERENCES users(id)     ON DELETE CASCADE
);
```

PHP Authentication API (JWT-based)

Priye implemented the authentication layer using PHP 8.2 with JWT tokens. The register endpoint demonstrates the pattern used across all API endpoints — input validation, prepared statements, and JSON responses:

```php
<?php
// src/Controllers/AuthController.php
namespace App\Controllers;

use App\Models\User;
use App\Services\JwtService;
use App\Helpers\Response;
use App\Helpers\Validator;

class AuthController {

    public function register(array $data): void {
        // 1. Validate input
        $errors = Validator::check($data, [
            'name'     => 'required|min:2|max:100',
            'email'    => 'required|email|unique:users',
            'password' => 'required|min:8|confirmed',
        ]);
        if ($errors) {
            Response::json(['errors' => $errors], 422);
            return;
        }
        // 2. Hash password & persist
        $user = User::create([
            'name'     => htmlspecialchars($data['name']),
            'email'    => strtolower($data['email']),
            'password' => password_hash($data['password'], PASSWORD_BCRYPT),
        ]);
        // 3. Issue JWT token
        $token = JwtService::issue($user->id, $user->email);
        Response::json(['token' => $token, 'user' => $user->toArray()], 201);
    }

    public function login(array $data): void {
        $user = User::findByEmail($data['email'] ?? '');
        if (!$user || !password_verify($data['password'] ?? '', $user->password)) {
            Response::json(['error' => 'Invalid credentials'], 401);
            return;
        }
        $token = JwtService::issue($user->id, $user->email);
        Response::json(['token' => $token, 'user' => $user->toArray()]);
    }
```

```
}
```

JavaScript API Client

Carlos built a reusable JavaScript API client module that all frontend code uses to communicate with the PHP backend, handling JWT token injection and error responses centrally:

```
// src/js/api.js  — centralised API client
const API_BASE = '/api';

const api = {
    async request(method, endpoint, body = null) {
        const token = localStorage.getItem('taskflow_token');
        const options = {
            method,
            headers: {
                'Content-Type': 'application/json',
                ...(token && { 'Authorization': `Bearer ${token}` })
            },
            ...(body && { body: JSON.stringify(body) })
        };
        const response = await fetch(`${API_BASE}${endpoint}`, options);
        if (response.status === 401) {
            localStorage.removeItem('taskflow_token');
            window.location.href = '/login';
            return;
        }
        return response.json();
    },
    get:    (endpoint)       => api.request('GET',    endpoint),
    post:   (endpoint, body) => api.request('POST',   endpoint, body),
    put:    (endpoint, body) => api.request('PUT',    endpoint, body),
    delete: (endpoint)       => api.request('DELETE', endpoint),
};

export default api;
```

Sprint 1 — Daily Scrum Snapshots

The Daily Scrum is the team's daily 15-minute opportunity to synchronise and adapt. Here are snapshots from three key days that illustrate how the event drives transparency and rapid problem-solving:

Day	Done Yesterday	Plan Today	Impediment
Day 2	Aisha: schema finalised and reviewed. Priya: register endpoint skeleton. Carlos: HTML registration form.	Priya: complete validation logic. Carlos: connect form to API. Aisha: begin writing PHPUnit test suite.	None — good start. Priya notes JWT library choice needs team agreement (resolved in 5 min post-standup).
Day 5	US-01 fully done (DoD met, deployed to staging). US-02 login endpoint complete.	Carlos: login UI + token storage. Priya: start projects CRUD API. Aisha: test register + login end-to-end.	BLOCKER: Local MySQL version mismatch causing different behaviour. James raised with DevOps — Docker config fix applied same day.
Day 8	US-02, US-03 done. US-04 tasks API 80% complete.	Priya: complete tasks API + error handling. Carlos: task creation modal. Aisha: write integration tests for project+task flow.	Carlos: CSS framework conflict slowing UI work. Team decision: drop framework, use custom CSS. 2h lost, recovered quickly.
Day 10 (last)	All Sprint 1 stories done and staged. Priya: completed US-05 assignee feature. Carlos: assignee dropdown polished.	Sprint Review prep — Aisha runs final regression. All: demo rehearsal with James.	None. Sprint Goal achieved. Velocity: 19 points delivered.

Sprint 1 — Sprint Review

The Sprint Review is held on the last afternoon of the sprint. Sarah (Product Owner) invites two stakeholders — the CEO and one target user — to participate. The team demonstrates the working application in the staging environment.

Review Item	Outcome
Demonstrated	User registration → login → create project → create tasks → assign tasks to team members — full end-to-end flow in staging
Stakeholder Feedback	CEO: 'The login is clean — exactly what I envisioned.' User: 'Can the task form show which project it belongs to?' (Added to backlog as US-16 enhancement)
Stories Accepted	US-01 ✓ US-02 ✓ US-03 ✓ US-04 ✓ US-05 ✓ — All 5 stories accepted by Product Owner
Velocity	19 story points delivered
Product Backlog Update	US-16 added: 'As a team member, I can see the project name on the task detail page.' (2 pts, Sprint 2)
Next Sprint Preview	Sarah walks through Sprint 2 priorities: Kanban board (US-06, US-07), comments (US-08), team invitations (US-09), notifications (US-10)

Sprint 1 — Sprint Retrospective

The Sprint Retrospective is the team's 90-minute private reflection session. James facilitates using the Start / Stop / Continue format:

Category	Observations	Action Item
Continue ✓	Daily Scrum at 9 AM is working well — short, focused, useful. Schema-first approach on Day 1 prevented dependency conflicts.	Keep schema design on Day 1 of every sprint. Keep Daily Scrum cadence.
Continue ✓	Feature branching + PR reviews catching bugs early (3 bugs caught in review in Sprint 1).	Maintain mandatory PR review policy — consider adding a PR checklist template.
Stop ✗	Ad-hoc Slack questions breaking deep work. Carlos lost 2+ hours to context switching from mid-day questions.	Introduce focus hours: 10 AM–12 PM and 2 PM–4 PM — no interruptions except blockers. Questions queued for Daily Scrum or focus breaks.
Start +	No automated integration testing pipeline — bugs only found manually in staging.	Aisha to set up GitHub Actions CI pipeline in Sprint 2 Day 1–2. All PRs must pass CI before merge.
Start +	No shared component library — Carlos duplicating CSS across pages.	Carlos to create shared components.css in Sprint 2. Saves estimated 4+ hours over remaining sprints.

Part 3: Sprint 2 — Collaboration Features (Weeks 3–4)

Sprint 2: Collaboration — Kanban Board, Comments & Notifications

Sprint 2 Planning

Sprint 2 Details	
Sprint Goal	Team members can visualise and manage all project tasks on a real-time Kanban board, invite colleagues, communicate via task comments, and receive assignment notifications by email.
Duration	2 weeks: Day 11–20
Selected Stories	US-06, US-07, US-08, US-09, US-10, US-16
Committed Points	28 story points

Sprint Planning Duration	2.5 hours (longer — more complex stories)
Key Risk	US-06 (Kanban board) is the most technically complex story. If it slips, it blocks US-07. Broken into front-end and back-end sub-tasks to allow parallel work.

Sprint 2 — Key Technical Implementations

Kanban Board — JavaScript Drag & Drop

The Kanban board (US-06 + US-07) is the most complex feature of Sprint 2. Carlos implemented a pure JavaScript drag-and-drop Kanban board that communicates task status changes to the PHP API in real time. No external libraries were used — a team decision to minimise dependencies:

```
// src/js/kanban.js — Kanban board with drag-and-drop
class KanbanBoard {
    constructor(projectId) {
        this.projectId = projectId;
        this.columns = { todo: [], in_progress: [], done: [] };
    }

    async init() {
        const data = await api.get(`/projects/${this.projectId}/tasks`);
        data.tasks.forEach(task => this.columns[task.status].push(task));
        this.render();
        this.bindDragEvents();
    }

    render() {
        const statuses = ['todo', 'in_progress', 'done'];
        const labels   = ['To Do', 'In Progress', 'Done'];
        const board    = document.getElementById('kanban-board');
        board.innerHTML = '';
        statuses.forEach((status, idx) => {
            const col = document.createElement('div');
            col.className   = 'kanban-column';
            col.dataset.status = status;
            col.innerHTML = `<h3>${labels[idx]} <span
class='count'>${this.columns[status].length}</span></h3>`;
            this.columns[status].forEach(task => {
                const card = this.createCard(task);
                col.appendChild(card);
            });
            board.appendChild(col);
        });
    }
```

```
    createCard(task) {
        const card = document.createElement('div');
        card.className     = `kanban-card priority-${task.priority}`;
        card.draggable     = true;
        card.dataset.taskId = task.id;
        card.innerHTML = `
            <p class='task-title'>${task.title}</p>
            <p class='task-due'>Due: ${task.due_date ?? 'No deadline'}</p>
            <span class='assignee'>${task.assignee_name ?? 'Unassigned'}</span>
        `;
        return card;
    }

    bindDragEvents() {
        document.addEventListener('dragstart', e => {
            if (e.target.classList.contains('kanban-card'))
                e.dataTransfer.setData('taskId', e.target.dataset.taskId);
        });
        document.querySelectorAll('.kanban-column').forEach(col => {
            col.addEventListener('dragover',  e => e.preventDefault());
            col.addEventListener('drop', async e => {
                e.preventDefault();
                const taskId    = e.dataTransfer.getData('taskId');
                const newStatus = col.dataset.status;
                await api.put(`/tasks/${taskId}`, { status: newStatus });
                await this.init();  // re-render board
            });
        });
    }
}
```

Email Notification Service (PHP + MySQL Queue)

US-10 (email notifications on task assignment) required a lightweight email queuing system to avoid blocking the API response. Priya implemented a database-backed email queue with a PHP cron job processor:

```
-- email_queue table (added in Sprint 2 migration)
CREATE TABLE email_queue (
  id          INT AUTO_INCREMENT PRIMARY KEY,
  to_email    VARCHAR(150) NOT NULL,
  subject     VARCHAR(255) NOT NULL,
  body        TEXT         NOT NULL,
  status      ENUM('pending','sent','failed') DEFAULT 'pending',
  attempts    INT          DEFAULT 0,
```

```
  created_at  TIMESTAMP    DEFAULT CURRENT_TIMESTAMP,
  sent_at     TIMESTAMP    NULL
);

// src/Services/NotificationService.php
class NotificationService {
    public static function queueAssignmentEmail(Task $task, User $assignee): void
{
        $subject = "New task assigned: {$task->title}";
        $body    = "Hi {$assignee->name},\n\n"
                 . "You have been assigned a new task: {$task->title}\n"
                 . "Project: {$task->project->name}\n"
                 . "Due: {$task->due_date}\n\n"
                 . "View task: " . APP_URL . "/tasks/{$task->id}";
        DB::insert('email_queue',
            ['to_email','subject','body'],
            [$assignee->email, $subject, $body]
        );
    }
}

// cron/process_email_queue.php — runs every 5 minutes via cron
// * * * * * php /var/www/taskflow/cron/process_email_queue.php
$pending = DB::query(
    'SELECT * FROM email_queue WHERE status=? AND attempts < 3 LIMIT 20',
    ['pending']
);
foreach ($pending as $email) {
    $result = mail($email['to_email'], $email['subject'], $email['body']);
    DB::update('email_queue', $email['id'], [
        'status'   => $result ? 'sent' : 'failed',
        'attempts' => $email['attempts'] + 1,
        'sent_at'  => $result ? date('Y-m-d H:i:s') : null,
    ]);
}
```

Sprint 2 — Impediment Log

The Scrum Master tracks all impediments raised during Daily Scrums and ensures they are resolved quickly. Unresolved impediments are escalated:

Day	Impediment Raised	Raised By	Resolution	Days to Resolve
Day	CI pipeline (GitHub	Aisha	Aisha fixed YAML config.	0 (same day)

12	Actions) failing on PHP 8.2 — wrong Docker image version in workflow YAML		Pipeline green by end of Day 12.	
Day 14	Drag-and-drop (US-07) not working in Safari — HTML5 Drag API has Safari quirks	Carlos	Carlos researched Safari-specific fix (pointer events polyfill). 3h cost. James moved US-07 acceptance to Monday to give recovery time.	1
Day 15	SMTP server not configured on staging — US-10 emails not sending	Priye	James raised with infrastructure team. Staging SMTP configured by Day 16 morning.	1
Day 17	US-09 (invite by email) blocked: invite token needs email delivery, which needs SMTP fix	Priye	SMTP fix unblocked this. Priya resumed US-09 on Day 16. No net delay.	0 (unblocked)

Sprint 2 — Sprint Review

Review Item	Outcome
Demonstrated	Full Kanban board with drag-and-drop; task comments in real time; team invitation flow; email notification on assignment
Stakeholder Feedback	User (tester): 'The board is exactly what we needed — it loads fast and drag feels natural.' CEO: 'Email notification email needs the company logo — can we add that?' (Added to Sprint 3 backlog as US-17)
Stories Accepted	US-06 ✓ US-07 ✓ US-08 ✓ US-09 ✓ US-10 ✓ US-16 ✓ — All 28 points accepted
Velocity	28 story points — team velocity improving with shared components and CI pipeline
Cumulative Velocity	Sprint 1: 19 pts | Sprint 2: 28 pts | Running total: 47 pts delivered

Sprint 2 — Sprint Retrospective

Category	Observation	Action Item
Continue ✓	CI pipeline added in Sprint 2 caught 2 regressions before they reached staging. Enormous value.	Keep CI mandatory for all PRs. Add code coverage report to CI output.
Continue ✓	Focus hours (from Sprint 1 retro) worked. Carlos reported measurably fewer interruptions.	Keep focus hours. Add to working agreements document.
Stop ✗	PR descriptions are too terse — reviewers spending time asking basic questions that a good description would answer.	Add PR template to repo with required sections: What changed? Why? How to test? Screenshots (if UI).
Start +	No Definition of Ready — stories sometimes arrive at Sprint Planning without clear acceptance criteria, causing lengthy discussion.	Sarah to write acceptance criteria for all Sprint 3 stories 3 days before Sprint Planning. Team reviews async and flags gaps.

Part 4: Sprint 3 — Analytics, Polish & Release (Weeks 5–6)

Sprint 3: Analytics, Polish & Production Release

Sprint 3 Planning

Sprint 3 Details	
Sprint Goal	Project managers have a real-time analytics dashboard, tasks support priorities and file attachments, admin user management is complete, and TaskFlow Pro is deployed to production with monitoring.
Duration	2 weeks: Day 21–30
Selected Stories	US-11, US-12, US-13, US-14, US-15, US-17 (logo in emails)

Committed Points	27 story points
Key Decision	Team adds a production deployment checklist as a Sprint 3 artifact — this becomes part of the Definition of Done for the release.
Non-functional NFRs	Page load < 2 seconds; SQL queries < 100ms; all forms CSRF-protected; input sanitisation audit complete.

Sprint 3 — Key Technical Implementations

Analytics Dashboard (PHP + JavaScript Charts)

US-12 (the analytics dashboard) requires aggregated MySQL queries and a JavaScript charting component. Priya built the API endpoints with optimised SQL and Aisha built the chart rendering using the native Canvas API to avoid library dependencies:

```
// PHP API: GET /api/projects/{id}/analytics
public function analytics(int $projectId): void {
    $this->authorize($projectId);  // check membership

    // Task status breakdown
    $statusBreakdown = DB::query(
        'SELECT status, COUNT(*) as count FROM tasks
         WHERE project_id = ? GROUP BY status',
        [$projectId]
    );

    // Overdue tasks (due_date in past and not done)
    $overdueTasks = DB::query(
        'SELECT t.*, u.name as assignee_name FROM tasks t
         LEFT JOIN users u ON t.assignee_id = u.id
         WHERE t.project_id = ? AND t.due_date < CURDATE()
         AND t.status != "done" ORDER BY t.due_date ASC',
        [$projectId]
    );

    // Team activity — tasks completed per member in last 30 days
    $teamActivity = DB::query(
        'SELECT u.name, COUNT(*) as completed FROM tasks t
         JOIN users u ON t.assignee_id = u.id
         WHERE t.project_id = ? AND t.status = "done"
         AND t.updated_at >= DATE_SUB(NOW(), INTERVAL 30 DAY)
         GROUP BY u.id ORDER BY completed DESC',
        [$projectId]
```

```
    );

    Response::json([
        'status_breakdown' => $statusBreakdown,
        'overdue_tasks'    => $overdueTasks,
        'team_activity'    => $teamActivity,
    ]);
}
```

```
// src/js/dashboard.js — render charts with Canvas API
async function loadDashboard(projectId) {
    const data = await api.get(`/projects/${projectId}/analytics`);

    // Status doughnut chart
    drawDoughnut(
        document.getElementById('status-chart'),
        data.status_breakdown.map(s => s.status),
        data.status_breakdown.map(s => s.count),
        ['#6366F1','#F59E0B','#10B981']   // todo, in_progress, done
    );

    // Team activity bar chart
    drawBarChart(
        document.getElementById('activity-chart'),
        data.team_activity.map(m => m.name),
        data.team_activity.map(m => m.completed)
    );

    // Overdue tasks list
    const overdueList = document.getElementById('overdue-list');
    data.overdue_tasks.forEach(task => {
        const li = document.createElement('li');
        li.className = 'overdue-item';
        li.innerHTML = `<strong>${task.title}</strong> — Due: ${task.due_date}
                        <span class='assignee'>${task.assignee_name ??
'Unassigned'}</span>`;
        overdueList.appendChild(li);
    });
}
```

File Upload Implementation (US-13)

Aisha implemented secure file uploads using PHP with MIME type validation, randomised filenames, and MySQL tracking:

```
// POST /api/tasks/{id}/attachments
public function uploadAttachment(int $taskId): void {
    $file     = $_FILES['attachment'] ?? null;
    $maxBytes = 10 * 1024 * 1024;  // 10 MB
    $allowed  = ['image/jpeg','image/png','application/pdf',
                 'text/plain','application/zip'];

    // Validate file
    if (!$file || $file['error'] !== UPLOAD_ERR_OK)
        Response::json(['error' => 'Upload failed'], 400);
    if ($file['size'] > $maxBytes)
        Response::json(['error' => 'File exceeds 10MB limit'], 413);

    // MIME check (finfo is safer than trusting $_FILES['type'])
    $finfo    = new \finfo(FILEINFO_MIME_TYPE);
    $mimeType = $finfo->file($file['tmp_name']);
    if (!in_array($mimeType, $allowed))
        Response::json(['error' => 'File type not permitted'], 415);

    // Save with randomised name to prevent path traversal
    $ext      = pathinfo($file['name'], PATHINFO_EXTENSION);
    $filename = bin2hex(random_bytes(16)) . '.' . $ext;
    $dest     = UPLOAD_DIR . '/' . $filename;
    move_uploaded_file($file['tmp_name'], $dest);

    // Record in DB
    DB::insert('task_attachments', [
        'task_id'       => $taskId,
        'original_name' => htmlspecialchars($file['name']),
        'stored_name'   => $filename,
        'mime_type'     => $mimeType,
        'file_size'     => $file['size'],
        'uploaded_by'   => $this->authUser->id,
    ]);
    Response::json(['message' => 'Attachment uploaded', 'filename' => $filename],
201);
}
```

Sprint 3 — Production Release Checklist

As part of Sprint 3, the team defined and executed a Production Release Checklist. This becomes part of the Definition of Done for the Sprint 3 release increment:

Category	Checklist Item	Owner	Status

Security	All user inputs sanitised with htmlspecialchars() / prepared statements	Priya	Done
Security	CSRF tokens implemented on all state-changing forms	Priya	Done
Security	JWT secret key stored in .env (not in codebase)	Priya	Done
Security	File upload MIME validation and size limits enforced	Aisha	Done
Security	Admin panel routes protected by role middleware	Aisha	Done
Performance	MySQL indexes on tasks(project_id), tasks(assignee_id), tasks(due_date)	Aisha	Done
Performance	All API endpoints respond in < 100ms (tested with Apache Bench)	All	Done
Performance	Frontend assets (JS/CSS) minified and cached (Cache-Control headers)	Carlos	Done
Quality	PHPUnit test suite: 87% code coverage, all tests green	Aisha	Done
Quality	Cross-browser testing: Chrome, Firefox, Safari, Edge	Carlos	Done
Quality	Mobile responsive: tested on iPhone SE and Samsung Galaxy S22	Carlos	Done
Deployment	Environment variables configured on production server	James	Done
Deployment	Database migrations run on production MySQL	Aisha	Done
Deployment	SSL certificate installed and HTTPS enforced	James	Done
Deployment	Error logging configured (PHP errors → log file, not browser)	Priya	Done
Monitoring	Uptime monitoring (UptimeRobot) configured — alert on 5-min downtime	James	Done

Sprint 3 — Sprint Review (Release Demo)

Review Item	Outcome
Demonstrated	Full analytics dashboard with charts; task filtering by assignee/status/priority; file attachments; admin user management; production deployment live on custom domain
Stakeholder Feedback	CEO: 'This is better than I expected at this stage. The dashboard alone justifies the investment.' User group: 3 users onboarded live during review — no critical issues. One UX suggestion: 'make the filter bar sticky' (logged for next sprint).
Stories	US-11 ✓ US-12 ✓ US-13 ✓ US-14 ✓ US-15 ✓ US-17 ✓ — All 27 points

Accepted	accepted
Velocity	27 story points
Final Product	TaskFlow Pro v1.0 is LIVE in production. Full MVP delivered across 3 sprints.

Sprint 3 — Final Retrospective

Category	Observation	Action Item / Learning
Continue ✓	Production release checklist was invaluable — zero critical post-release bugs in first 48 hours. This is exceptional.	Make release checklist a permanent artifact in the team's process. Update it every sprint.
Continue ✓	Definition of Ready (introduced in Sprint 2) eliminated planning ambiguity completely. Sprint 3 planning took 90 minutes vs. 2.5 hours in Sprint 2.	Keep DoR as a permanent team practice. Sarah to maintain it per sprint.
Stop ✗	Some tasks were estimated individually rather than as a team — a few estimates were badly wrong (US-13 took 2x estimated time).	All estimates to be done as a group. Use planning poker even for 'small' tasks — surprises happen.
Start +	No knowledge documentation — tribal knowledge in team members' heads. If someone leaves, knowledge is lost.	Start a team wiki (Confluence or Notion) for architectural decisions, API contracts, and deployment procedures.

Part 5: Project Summary & Scrum Lessons Learned

5. End-of-Project Summary

5.1 Delivery Summary

Metric	Sprint 1	Sprint 2	Sprint 3	Total
Stories Delivered	5	6	6	17
Story Points Delivered	19	28	27	74

Planned Points	19	28	27	74
Forecast Accuracy	100%	100%	100%	100%
Bugs Escaped to Production	0	0	0	0
PHPUnit Test Coverage	62%	74%	87%	87% (final)
Impediments Raised	1	4	1	6
Impediments Resolved	1	4	1	6 (all resolved)

RESULT: TaskFlow Pro v1.0 was delivered on time, within the 6-week timebox, with all 17 planned stories accepted, 74 story points delivered, zero escaped production bugs, and 87% automated test coverage. The application was live in production by the end of Day 30.

5.2 How Scrum Created Value on This Project

Scrum Mechanism	Concrete Value Delivered on TaskFlow Pro
Sprint Goals gave focus	Each sprint had a clear, single outcome. The team never lost sight of 'what does done look like this sprint?'
Sprint Reviews caught misalignment early	The CEO's feedback about email logo in Sprint 2 Review — caught with 2 weeks remaining, not after launch.
Retrospectives drove real improvement	3 retros produced 8 concrete action items, all implemented. Sprint velocity grew from 19 to 28 pts (47% increase) in part due to these improvements.
Daily Scrums surfaced blockers fast	The SMTP misconfiguration blocker (Day 15, Sprint 2) was surfaced and resolved in 1 day. In Waterfall, it might not have been found until QA phase weeks later.
Definition of Done enforced quality	Zero production bugs in first 48 hours — the DoD's requirement for staging verification and test coverage created a quality gate the team trusted.
Backlog transparency built trust	Stakeholders could see the full backlog, understand priorities, and make informed decisions about what to include/exclude. No surprises at launch.

5.3 Scrum Artefacts Produced

Artifact	Description & Location
Product Backlog	17 user stories with priorities, estimates, and sprint assignments. Maintained by Product Owner throughout project.
3x Sprint Backlogs	Per-sprint task breakdowns with owner and status tracking. Created at Sprint Planning, updated daily.
Definition of Done	Team-owned quality checklist. Evolved across sprints — v1 (Sprint 1) had 6 items; v3 (Sprint 3) had 10 items.
3x Sprint Goals	One clear statement per sprint guiding all team decisions about scope and priority.
Impediment Log	6 impediments raised and resolved across 3 sprints. Maintained by Scrum Master.
3x Sprint Review Notes	Stakeholder feedback, accepted/rejected stories, backlog updates. Maintained by Product Owner.
3x Sprint Retrospective Notes	Improvement actions with owners. James tracked completion of previous retro actions at each new retro.
Production Release Checklist	16-item checklist created in Sprint 3. Zero critical production bugs attributed in part to this artifact.
API Documentation	Endpoint reference maintained in team wiki (added as Sprint 3 retro action).

— End of Case Study —

TaskFlow Pro v1.0 | PHP 8.2 + MySQL 8.0 + JavaScript ES6+ | Delivered in 3 Sprints

14.0 CONCLUSION

Agile software development is not a destination — it is a journey of continuous improvement guided by clear values, empirical evidence, and genuine respect for people. The organizations and teams that thrive with Agile are those that embrace its spirit, not just its ceremonies. Agile is a mindset before it is a method.

The most important thing you can do after reading this tutorial is to try something, inspect the results honestly, and adapt. Start small. Pick one practice, implement it rigorously, observe what happens, and improve. The path to agility is paved with experiments and learning — not with frameworks and certifications.

BIBLIOGRAPHY

1	Beck, K. (1999). Extreme Programming Explained: Embrace Change. Addison-Wesley Professional. — The original and definitive text on XP, written by its creator. Essential reading for anyone serious about engineering practices.
2	Poppendieck, M., & Poppendieck, T. (2003). Lean Software Development: An Agile Toolkit. Addison-Wesley Professional. — Translates Toyota's Lean Manufacturing principles into software development. Introduces the seven wastes of software and value stream thinking.
3	Schwaber, K., & Sutherland, J. (2020). The Scrum Guide: The Definitive Guide to Scrum — The Rules of the Game. Scrum.org. Retrieved from https://scrumguides.org — The authoritative, freely available reference for Scrum. Every Scrum practitioner must read this in full.
4	Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., … Thomas, D. (2001). Manifesto for Agile Software Development. Retrieved from https://agilemanifesto.org — The founding document of the Agile movement. Short but enormously consequential. Read the values, principles, and the authors' backgrounds.
5	Anderson, D. J. (2010). Kanban: Successful Evolutionary Change for Your Technology Business. Blue Hole Press. — The foundational book on the Kanban Method as applied to knowledge work. Covers WIP limits, flow, metrics, and the six core practices in depth.
6	Cohn, M. (2004). User Stories Applied: For Agile Software Development. Addison-Wesley Professional. — The definitive guide to writing, splitting, and managing user stories. Covers the INVEST criteria, story mapping, and estimation with planning poker.
7	Leffingwell, D. (2011). Agile Software Requirements: Lean Requirements Practices for Teams, Programs, and the Enterprise. Addison-Wesley Professional. — The intellectual foundation for SAFe. Covers Agile requirements at scale: features, epics, portfolio epics, and program increment planning.