FLOW

Why Software Teams Need a New Operating Model

A whitepaper by Joe Hochrein | PSM I, PSM II, PSFS | Scrum.org Certified

May 2026 | Version 1.0

This paper presents FLOW as a proposed operating model. FLOW is a new framework and has not yet been tested at scale. The claims herein are grounded in established research on lean thinking, systems theory and flow-based delivery.

Table of Contents

Executive Summary

Software teams are drowning in ceremony. The frameworks meant to bring discipline to software delivery have become a primary source of inefficiency: hours lost to meetings, velocity metrics that measure activity rather than value, and sprint commitments that create pressure without producing accountability.

This paper makes the case for a fundamentally different approach. FLOW is a continuous delivery operating model built on lean thinking and systems theory. It is designed to eliminate unnecessary overhead, make constraints visible, and create a culture of collective ownership.

This paper examines why sprint-based delivery frameworks are struggling in practice, what FLOW does differently, what the established research suggests about flow-based delivery, and how to evaluate whether your organization is ready to make the shift.

Key Findings

1. Sprint-based delivery frameworks impose ceremony overhead estimated at over $98,000 per year for a ten-person team at mid-market US salaries.[8]
2. Only 23 percent of organizations report that Agile has fully met their expectations, according to the 2023 State of Agile Report. Satisfaction with Agile dropped from 71 to 59 percent in a single year. A KPMG survey found that only 13 percent of top management fully supports their Agile transformation. Jeff Sutherland, co-creator of Scrum, estimates that 47 percent of all Agile transformations fail.[3, 27, 28]
3. Velocity, Scrum's primary performance metric, is widely acknowledged to be gameable. Scrum.org's own published research documents how teams inflate estimates to hit velocity targets without delivering more value.[5]
4. Research on continuous delivery practices shows that teams deploying more frequently have lower change failure rates and faster recovery times than teams releasing on fixed schedules. Continuous flow is not just faster. It is safer.[12]
5. Research on interruption recovery suggests that it takes an average of 23 minutes and 15 seconds to fully regain focus after a significant interruption. Daily standups, sprint planning sessions, and retrospectives are not just time costs. They are focus costs that compound across the entire engineering organization.[9]
6. Developer dissatisfaction is driven primarily by too many meetings, unclear priorities, context switching, and being measured on things outside their control. FLOW is designed to address all four structurally.[16]
7. Psychological safety is one of the strongest predictors of team performance in organizational research. FLOW is designed to create it structurally by treating work stalls as system signals rather than personal failures.[11]
8. The average cost to replace a software engineer is estimated at 1.5 to 2 times their annual salary. Organizations that reduce the structural drivers of developer dissatisfaction retain engineers at meaningfully higher rates.[22]

The Problem With How We Build Software Today

The software industry adopted Agile in the early 2000s for good reasons. Waterfall projects were failing at alarming rates. Requirements changed faster than multi-year plans could accommodate. Teams needed a way to deliver incrementally, inspect their work, and adapt.[1]

Scrum became the dominant Agile implementation. By 2023, the majority of Agile teams reported using Scrum or a Scrum hybrid.[2]

Scrum brought real improvements: iterative delivery, regular inspection points, and cross-functional teams. For many organizations it was a significant step forward from what came before.

But Scrum was designed for a different era of software delivery. The sprint timebox, the commitment model, and the fixed ceremony structure made sense when deployment was expensive and infrequent. Today, continuous delivery is not just possible, it is expected. The sprint is an artificial container in a world that has outgrown containers.

The data on Scrum's practical performance is sobering. The 17th State of Agile Report found that only 11 percent of respondents expressed high satisfaction with Agile, with overall satisfaction dropping from 71 to 59 percent in a single year.[3]

Developer satisfaction surveys consistently cite too many meetings, unclear priorities, and being measured on things outside their control as top sources of dissatisfaction.[4]

The problem is not that teams are doing Scrum wrong. The problem is that Scrum was designed for a world that no longer exists. Scrum.org itself acknowledged in 2025 that the Scrum Guide was written in a time when releasing software to production was a cumbersome, risky process. That world has changed fundamentally.[23]

What Sprint-Based Delivery Gets Wrong

The sprint is a batching mechanism in a world that needs flow

Scrum's sprint timebox was designed to create a predictable rhythm and a commitment window. In practice it creates three problems that compound each other.

First, it delays delivery. Work that is completed on day three of a sprint cannot be released until the sprint ends. Value that is ready sits waiting for an artificial boundary.

Second, it creates commitment pressure that produces sandbagging. Teams learn quickly that sprint commitments are evaluated. The rational response is to estimate conservatively to protect the commitment. The result is a velocity number that tells leadership what they want to hear, not what is actually happening.[5]

Third, it makes change expensive. Mid-sprint priority changes disrupt commitments and create friction between engineering and product teams. The sprint becomes a protection mechanism rather than a delivery mechanism.

Velocity measures activity, not value

Story points were designed to help teams forecast capacity. They were never intended to be a performance benchmark.[6]

Research on estimation accuracy is consistent: developers at all experience levels systematically underestimate software complexity. Estimation sessions consume real time to produce numbers that are unreliable by design.[7]

When a developer estimates three points and a ticket takes thirteen, the gap frequently becomes a record of underperformance against a number that was never intended as a performance benchmark. Practitioners and the Scrum Alliance itself have documented how this misuse of estimation data creates fear and dysfunction within teams.[24]

Sprint-based delivery costs more than you think

A standard Scrum team of ten people running two-week sprints spends approximately 7 hours per engineer per sprint in ceremonies that FLOW eliminates entirely: planning, standups, review, retrospective, and capacity planning. That is 182 hours per engineer per year spent in meetings that produce no working software. Refinement is retained in FLOW as the one necessary meeting. At mid-market US engineering salaries, the eliminated ceremonies represent over $98,000 per year in recovered capacity, before accounting for the cognitive cost of frequent context switching.[8]

Research on interruption recovery suggests that knowledge workers require an average of 23 minutes to return to full focus after a significant interruption. Daily standups, sprint planning sessions, and retrospectives are not just time costs. They are focus costs that compound across the engineering organization.[9]

The cost of task switching compounds this further. Research from the American Psychological Association found that shifting between tasks can consume up to 40 percent of a knowledge worker's productive time.[25,15] Sprint-based teams often carry multiple concurrent active tickets per developer. If even a fraction of that productivity loss is attributable to high WIP, the financial impact for a ten-person engineering team at mid-market US salaries is worth doing the math on. FLOW's soft WIP ceiling of two to three active tickets per engineer is designed to directly address this drain.

Finishing one thing is always more valuable than starting three. High WIP is the enemy of flow.

The Case for Continuous Flow

Lean manufacturing demonstrated in the 1980s and 1990s that small batch sizes, pull-based production, and visible constraints produce dramatically better outcomes than large batch, push-based systems. Toyota's production system, which inspired lean thinking, achieved quality and throughput improvements that transformed manufacturing.[10]

The principles transferred to knowledge work. The core insight is that high work in progress creates context switching, and context switching creates cognitive overhead that reduces code quality and increases defect rates. Finishing work before starting new work, constrained by a soft WIP ceiling, maintains focus and reduces the compounding cost of partially completed work.

Psychological safety, the shared belief that a team is safe for interpersonal risk-taking, is directly linked to learning behavior, which in turn predicts team performance. When engineers feel safe raising blockers and admitting uncertainty early, problems surface before they compound. Sprint commitments, by design, create the pressure that suppresses exactly that behavior.[11]

When aging indicators surface constraints at the board level rather than the person level, psychological safety is built into the system itself.

Connection to Six Sigma and Process Improvement Science

FLOW draws on a broader body of process improvement science that extends beyond Agile and lean manufacturing. Six Sigma, which emerged from Motorola and was scaled by General Electric in the 1980s and 90s, shares several foundational principles with FLOW.

Variation reduction is a defining principle of Six Sigma. Inconsistent processes produce inconsistent outputs. FLOW's WIP limits, aging indicators, and pull-based execution are all variation reduction mechanisms applied to software delivery. Cycle time variability, which FLOW tracks as a primary health signal, is precisely what Six Sigma would work to eliminate in any measurable process.

Value stream mapping is a tool used in both lean manufacturing and Six Sigma that identifies where value is created and where waste accumulates across a process. The FLOW board is a live value stream map. Every stage is a step in the value stream. Aging indicators flag where waste is accumulating in real time, without requiring a separate mapping exercise.[17]

The Cost of Poor Quality, a Six Sigma concept that quantifies the financial impact of defects, rework, and process failures, applies directly to software delivery. The ceremony overhead calculation in this paper is a Cost of Poor Quality analysis applied to process waste rather than product defects. Acceptance testing rejection rates, bug escape rates, and rework caused by poor refinement are all Cost of Poor Quality in FLOW terms.[18]

Voice of the Customer, another Six Sigma principle, centers delivery on customer-defined quality rather than internal process metrics. FLOW's Business Acceptance accountability is the structural equivalent. The customer confirms that the work delivers intended value before it is released. That is Voice of the Customer built into the operating model rather than treated as a separate validation activity. The acceptance testing rejection rate becomes a natural quality metric, surfacing requirements problems upstream before they compound into larger delivery failures.

The Theory of Constraints asserts that every system's performance is limited by its slowest constraint. Improving anything other than the constraint produces no meaningful gain in overall throughput. FLOW's aging indicators are a direct application of this principle. They exist specifically to surface the constraint so the team can focus improvement efforts exactly where they will have the most impact.[26]

What FLOW Does Differently

FLOW is not a modification of Scrum. It is a maturation of Agile, a continuous delivery operating model that draws on lean thinking and systems theory rather than extending or modifying any existing framework. It honors what Agile got right and builds on it. Where Agile defined the values, FLOW operationalizes them.

Scrum	FLOW
Sprint commitment	Pull-based execution
Velocity	Cycle time and throughput
Capacity planning	Aging indicators and WIP limits

Sprint review	Continuous release on readiness
Sprint retrospective	SPAR: triggered by a system signal
Fixed roles	Four accountabilities, any team size
Story points	Abstract sizing: Small, Medium, Large
Daily standup	Board visibility and ticket comments

Pull replaces push

Work is never pushed onto engineers to start before they have capacity. Engineers pull from a prioritized backlog when they have capacity, starting with the highest priority item. This eliminates capacity planning overhead, removes pre-loaded sprint backlogs, and ensures that work always flows to available capacity rather than waiting for a timebox. Research on flow-based software delivery consistently points to shorter lead times, lower defect rates, and higher team satisfaction compared to sprint-based commitment models when WIP is constrained and work is pull-based.[12]

WIP limits replace sprint commitments

Rather than committing to a fixed scope for a fixed period, FLOW constrains how much work can be in progress at any time. A soft ceiling of two to three active tickets per engineer ensures that **finishing is always prioritized over starting**. The board makes WIP visible in real time, without a planning meeting. Constraining WIP reduces cycle time variability and improves throughput predictability, two outcomes that velocity-based planning consistently struggles to deliver.[13]

Abstract sizing replaces estimation

FLOW uses three abstract sizes: Small, Medium, and Large. These are assigned during refinement before a ticket enters the ready queue. They are not time estimates, they are not story points, and they should not be converted to hours or days. Doing so reintroduces the estimation problem FLOW is designed to remove. Abstract sizes are a shared signal about relative effort that serves two purposes.

The first is prioritization. A Small ticket with high business value should move ahead of a Large ticket with moderate value. Without any sizing signal, prioritization is purely gut feel. Abstract sizing gives the Prioritization accountability a concrete input for ordering the backlog by value to effort ratio. Teams that use structured prioritization frameworks such as RICE, which scores ideas by Reach, Impact, Confidence, and Effort, will find abstract sizing maps naturally to the Effort component of that formula, making FLOW compatible with discovery practices already in use.

The second is refinement quality. When engineers cannot agree on ticket sizing, that disagreement is very likely a requirements problem. The conversation the disagreement triggers is more valuable than the size label it produces. Ambiguity surfaced during refinement is cheap. Ambiguity discovered during build is expensive.

Abstract sizing is never used to measure engineer output or forecast delivery dates. That boundary is what separates it from story points or specific estimates. A ticket that takes longer than its size suggested is a signal worth examining, not a performance record.

Refinement

Refinement in FLOW is not a scheduled ceremony. It is triggered when the ready queue falls below a defined threshold, typically two to three tickets per engineer. The threshold is a team decision, not a FLOW prescription.

Who attends depends on what the work requires. Some tickets need the full team. Others need only the Requirements accountability and the engineer most familiar with the domain. When a ticket requires deeper technical expertise, the relevant subject matter expert is pulled in. The goal is to get tickets to ready with the minimum people necessary, not to run a meeting for its own sake.

A ticket is ready when it has three things: exit criteria, acceptance criteria, and an abstract size. Teams are free to add to that minimum. Definition of done, QA approach, and dependency notes are common additions. FLOW only prescribes the floor.

Refinement is the one meeting FLOW retains because requirements quality is not a ceremony problem. It is a delivery problem. Ambiguity that enters build as a poorly defined ticket compounds into rework, expanded scope, technical debt, and acceptance testing rejections. Refinement is where that cost is prevented.

Aging indicators replace velocity

When a ticket exceeds its stage threshold, the board flags it. The signal tells the team where the constraint lives: in requirements, in build, or in business validation. The response is a swarm, a focused team intervention to unblock the constraint, not a performance conversation. The system surfaces the bottleneck and the team addresses it. Teams in which work stalls are treated as system signals rather than personal failures show measurably higher rates of early problem reporting, which correlates with lower defect escape rates and higher overall team performance.[14]

When Tickets Expand in Scope

Not all aging tickets represent process failures. When a ticket ages due to expanded scope, the first job of the swarm is to identify the cause before determining the response.

Expanded scope has two distinct origins. The first is a requirements failure. The ticket was not defined well enough before it entered the build stage. The engineer is building against a moving or incomplete target. The right response is to return the ticket to refinement, tighten the requirements, and re-enter the queue with a more accurate size.

The second is a technical discovery. The requirements were sound but the engineer found something in the code that changed the scope entirely. A missing dependency. A data model that does not support the feature. A refactor that has to happen before the work can proceed. No amount of better refinement would have surfaced this before someone was in the code. It is an engineering reality, not a process failure.

Both origins age the ticket. Both trigger a swarm. The response is the distinction.

For requirements failures, the ticket returns to refinement with new scope and reprioritization or sent to the backlog. For technical discoveries, the team chooses one of two paths: break the ticket and ship what is complete while creating a new ticket for the remaining work, or accept the expanded scope and continue with reset expectations.

In either case, the Prioritization accountability is notified. The original value to size ratio used to rank the ticket is no longer accurate. For teams using a structured prioritization framework such as RICE, the Effort component has changed materially and the score should be

recalculated before the ticket continues. A ticket that entered build as a Small but is behaving as a Large may no longer deserve its position in the queue. That is a prioritization decision and it needs to be made consciously and transparently.

Collaborative work modes

FLOW defines three collaborative work modes for situations where a ticket needs more than one engineer to move forward. These modes can be triggered proactively by team choice or reactively when an aging indicator signals a constraint. They represent an escalating level of intervention.

- **Swarming** is the first response. When a ticket is aging or a constraint is identified, one or more engineers stop pulling new work and focus on the blocked ticket until the constraint is resolved. The team focuses on clearing the constraint before pulling anything new. Swarming is temporary. Once the constraint is resolved the team returns to normal pull-based execution.
- **Co-development** is two engineers working a single ticket together. It can be chosen proactively when a ticket is complex enough to benefit from a second perspective, or reactively when swarming alone is not enough to move the work forward. Co-development is also a natural training opportunity. Pairing a senior engineer with a junior engineer on complex or unfamiliar work accelerates knowledge transfer in a way that code reviews and documentation rarely achieve. Co-development is conceptually aligned with pair programming but is not prescriptive about how the two engineers divide the work. The ticket remains a single unit of delivery.
- **Takeover** is a full reassignment. When an engineer is unable to continue on a ticket due to capacity, a skillset gap, or a re-prioritization that pulls them to higher value work, the ticket transfers to another engineer in place. It does not return to the queue. It does not restart. A formal handoff is required. The outgoing engineer documents current state, progress, and any known blockers directly on the ticket before reassignment. The incoming engineer picks up from where the work stands, inherits the ticket's current age, and assumes full responsibility for seeing it through to completion.

  The re-prioritization trigger is worth noting specifically. Takeover in FLOW is not a sign that something went wrong. It is a sign that the team is responding to what matters most. The ticket continues. The engineer moves to where they are needed.

Accountabilities replace roles

FLOW defines four accountabilities: Prioritization, Requirements, Building, and Business Acceptance. These map to dedicated roles on large teams and compress onto fewer people on small teams. A solo developer holds all four. FLOW is designed to work at every scale, from a solo developer to a full cross-functional team.

Continuous release replaces sprint reviews

Work is released when it is ready, based on business value, customer need, and strategic timing. The release queue builds continuously as tickets complete. There is no sprint boundary to wait for and no review ceremony required to unlock delivery. Released work is transparent by default. What shipped and when is visible on the board and should be communicated to stakeholders through a release summary, a shared changelog, or whatever reporting mechanism the organization already uses. FLOW does not prescribe the format. It does prescribe that transparency does not require a ceremony to produce it.

FLOW's continuous release model assumes that the technical infrastructure to support it exists. Teams with mature CI/CD pipelines can release on demand as the release queue builds. Teams without that infrastructure should treat pipeline maturity as a parallel investment alongside FLOW adoption. The board discipline FLOW brings is valuable at any level of pipeline maturity, but the full promise of continuous delivery requires both the operating model and the infrastructure to support it.

SPAR replaces the retrospective

Rather than a fixed ceremony at the end of every sprint, FLOW replaces the retrospective with SPAR, a pulled meeting triggered by a specific system signal that follows four steps: Signal, Problem, Action, and Result. A SPAR is pulled when there is something specific to address, not when the calendar says it is time. Triggers include a production bug, a rising acceptance testing rejection rate, or a stage that is aging consistently. Those familiar with Root Cause Analysis or post-mortem practices will recognize the structure. SPAR applies the same disciplined thinking but ties it to a system signal rather than a scheduled review.

Team Decision Meetings

Not every team conversation is a SPAR and not every process question belongs in refinement. Teams will occasionally need to make collective decisions: a coding style change, a tooling addition, a process improvement proposal. These are legitimate conversations that benefit from the whole team.

FLOW handles them the same way it handles everything else. Someone surfaces the proposal, identifies who needs to be in the room, and pulls the conversation when there is something specific to decide. If the decision requires action it becomes a ticket, gets sized, and enters the backlog like any other work.

The pull principle still applies. Given what is known about the cost of meetings, the bar for pulling any team conversation should be clear. A specific proposal, the right people, and a decision at the end. If any of those three are missing the meeting should wait until they are not.

How FLOW Relates to Existing Frameworks

FLOW did not emerge in a vacuum. It borrows deliberately from Kanban, Scrum, and lean manufacturing, and it is aware of what Scrumban and Extreme Programming offer. Understanding where FLOW sits relative to these frameworks is important for teams who are already working within one of them.

FLOW takes what works from each, removes what does not, and adds what is missing. It is not a rejection of any framework. The components are proven. The combination is new.

Kanban

Kanban, formalized by David Anderson in 2010, is a change management method built on visualizing work, limiting WIP, managing flow, and making process policies explicit. It is deliberately non-prescriptive, designed to be overlaid on an existing process rather than replacing it.[13]

FLOW's board mechanics come directly from Kanban. Pull-based execution, WIP limits, aging indicators, and the emphasis on cycle time over velocity are all Kanban thinking applied to software delivery. FLOW does not reinvent this foundation. It builds on it.

Kanban's non-prescriptive nature is its strength and its limitation. It gives teams flexibility but leaves critical questions unanswered. Who owns prioritization? What does done mean at each stage? When does the team stop and examine a systemic problem? Kanban is silent on all three.

FLOW answers them. Four accountabilities replace undefined ownership. Exit criteria make quality gates explicit at every stage. Acceptance Testing becomes a named delivery stage with defined accountability and an escalation path. Swarming, co-development, and takeover give teams explicit practices for unblocking constrained work. SPAR replaces retrospection with a signal-driven meeting pulled when the system says something is wrong.

A team practicing Kanban is closer to FLOW than a team practicing Scrum. The foundation is shared. What FLOW adds is structure where Kanban is intentionally silent.

Scrum

Scrum, defined in the Scrum Guide maintained by Ken Schwaber and Jeff Sutherland, is a lightweight framework for developing and sustaining complex products. It defines three roles, five events, and three artifacts. It is the most widely adopted Agile framework in the world.[2]

FLOW does not entirely reject Scrum. It borrows from it selectively. The emphasis on iterative delivery, cross-functional accountability, and regular inspection all carry forward. The refinement meeting FLOW retains is directly descended from Scrum's backlog refinement event.

The departures are deliberate:

- Scrum requires a sprint timebox. FLOW eliminates it entirely. Delivery is continuous.
- Scrum defines three required roles. FLOW defines four accountabilities that compress or expand based on team size.
- Scrum uses velocity and story points. FLOW uses cycle time, throughput, and abstract sizing.
- Scrum requires daily stand-ups, sprint planning, sprint review, and sprint retrospective. FLOW retains only the refinement meeting and adds SPAR as a pulled replacement for the retrospective.
- Scrum treats acceptance testing and business validation as external to the framework. FLOW makes them a required delivery stage with defined accountability, aging indicators, and a clear escalation path when the business does not engage.

The most important distinction is philosophical. Scrum asks teams to commit to a scope for a fixed period. FLOW asks teams to commit to finishing work before starting new work. One creates sprint pressure. The other creates flow discipline. Both are forms of commitment that produce different cultures.

Scrumban

Scrumban was originally described by Corey Ladas as a way to transition teams from Scrum to Kanban by gradually introducing Kanban principles into a Scrum foundation. It retains Scrum's sprint timebox and roles while incorporating Kanban's WIP limits and flow optimization.[20]

Scrumban is a transitional model. It is what happens when a Scrum team wants more flow but is not ready to leave the sprint container entirely. It is a reasonable stepping stone.

FLOW is a destination model. It is what a team builds when they are ready to move past the sprint container entirely and operate on lean thinking and systems theory as independent foundations.

The practical differences:

- Scrumban typically retains the sprint timebox, even if shortened or made optional. FLOW eliminates it.
- Scrumban retains Scrum roles to varying degrees. FLOW replaces them with accountabilities that scale.
- Scrumban inherits Scrum's ceremony overhead to varying degrees depending on implementation. FLOW eliminates all ceremonies except refinement.
- Scrumban does not define Acceptance testing or business validation as a constraint. FLOW does.
- Scrumban does not define collaborative work modes. FLOW does.

Teams do not need Scrumban to adopt FLOW. The transition from Scrum to FLOW is a deliberate choice, not a gradual drift.

Extreme Programming

Extreme Programming, defined by Kent Beck in 1999, is an Agile methodology that emphasizes engineering practices including test-driven development, pair programming, continuous integration, refactoring, and small releases. Of the major Agile frameworks, XP is the most prescriptive about how engineers write and test code.[21]

FLOW shares XP's emphasis on quality gates, collaborative coding, and continuous delivery. Co-development in FLOW is conceptually aligned with XP's pair programming practice. FLOW's exit criteria reflect XP's emphasis on building quality in, rather than finding defects after the fact.

Where FLOW differs from XP is in scope. XP prescribes specific engineering practices. FLOW deliberately avoids prescribing how engineers write, test, or review code. FLOW defines what must be true at each stage exit without dictating the technical means of getting there. Teams practicing XP can practice FLOW. The two are compatible rather than competing.

FLOW draws on what the research supports in each framework, deliberately removes what the evidence suggests creates overhead without value, and defines practices that existing frameworks leave unaddressed.

Is Your Organization Ready for FLOW?

FLOW is a new model and has not yet been tested at scale. It is built directly on a substantial and well-established research base, and the mechanisms it employs are grounded in findings that have been replicated across knowledge work and software delivery contexts. The specific outcomes in your organization will depend on your context, your team, and how faithfully the model is adopted.

FLOW is not for every context. It requires three conditions to hold.

A backlog that is maintained and prioritized

FLOW's pull mechanism only works if there is something worth pulling. If the backlog is a dumping ground rather than a prioritized list ordered by business value, engineers will have nothing meaningful to pull. A poorly maintained backlog will undermine FLOW before it starts.

Business participation in validation

FLOW treats acceptance testing as a required delivery stage, not an optional final step. Work is not released unvalidated. When the business does not participate within the aging threshold, tickets move to a Hold status rather than returning to the active queue. They are not defective. They are waiting. When the business is ready to validate, the ticket moves back into acceptance testing. Successful adoption of FLOW requires a business that is willing to engage in validation on a predictable and timely basis.

Leadership willing to measure flow health rather than sprint commitments

The shift from velocity reporting to cycle time and throughput is the hardest organizational change FLOW requires. The data FLOW produces is designed to be more honest and more stable than velocity over time, but it requires a period of transition and tolerance for ambiguity during adoption.

How to Adopt FLOW

You do not sell FLOW. You demonstrate it.

Adoption can start with a single engineer practicing three behaviors: pulling one ticket at a time, checking the board before pulling new work, and commenting on tickets when blocked rather than sitting silently on stalled work. These behaviors, practiced consistently, become visible to teammates and managers. That visibility is what opens the door to team-level adoption.

Team-level adoption follows naturally when the behavior is noticed. A manager sees tickets finishing consistently. The board tells the story without anyone having to explain it.

Expanding to a team requires agreement on the board stage structure, soft WIP targets visible on the board, and aging indicators configured in the tool.

The answer to resistance is never argument. It is continued demonstration. Cycle time improves. Tickets finish. Quality signals get cleaner. The board tells the truth. Let the data respond.

Starting conditions

- One engineer, one board, one prioritized list. No process change required from anyone else
- Three behaviors: check the board first, pull one ticket, comment when blocked

Expanding to a team

- Agreement on board stage structure
- Soft WIP targets visible on the board
- Aging indicators configured in the tool
- As little as a half-day of setup

Conclusion

The software industry adopted Scrum because it was better than what came before. But the gap between what sprint-based delivery promises and what it delivers in practice is worth examining honestly. Ceremony overhead is growing. Developer satisfaction is declining. Velocity is a managed number that leadership has learned not to trust.

Developer attrition research consistently identifies the same top drivers of dissatisfaction: too many meetings, unclear priorities, being measured on things outside their control, and constant context switching. FLOW is designed to address all four structurally.[16]

The research on psychological safety adds a dimension that delivery metrics alone cannot capture. When work stalls are treated as system signals rather than personal failures, engineers raise problems early, surface blockers freely, and collaborate without fear of judgment. FLOW is designed to create that environment structurally. Not through culture initiatives or values statements, but through the mechanics of how work moves and how constraints are surfaced.

The pace of software delivery is accelerating. AI-assisted development is producing code faster than sprint-based frameworks were designed to release it. Security vulnerabilities that once fit comfortably inside a sprint boundary now need to move from discovery to production in hours, not days. FLOW's continuous release model is built for exactly this environment. Work is released when it is ready. There is no sprint boundary standing between a critical fix and your users.

FLOW is built on lean thinking and systems theory rather than sprint-based commitments. Pull-based execution, constrained WIP, aging indicators, and system-level accountability are designed to produce more credible delivery signals, more sustainable engineering culture, and measurably better outcomes. The research that underpins the model is substantial. The practical test is in your organization.

One engineer, one board, one prioritized list. That is enough to start.

The FLOW Guide and this whitepaper are available as free downloads at flowframework.dev

References

1 Standish Group. CHAOS Report. 1994.

2 Digital.ai. State of Agile Report, 17th Annual Edition. 2023. Reports that more than 80 percent of Agile teams use Scrum or a Scrum hybrid as their primary methodology.

3 Digital.ai. State of Agile Report, 17th Annual Edition. 2023. Reports that only 23 percent of organizations state that Agile has fully met their expectations, with overall satisfaction dropping from 71 to 59 percent in a single year.

4 Stack Overflow. Developer Survey. 2023.

5 Wolpers, S. Gaming Velocity: How Not to Measure Success and What to Avoid. Scrum.org. 2024. Documents velocity gaming behavior aggregated from more than 50 Scrum Master and Product Owner training classes. Goodhart's Law: When a measure becomes a target, it ceases to be a good measure.

6 Cohn, M. Agile Estimating and Planning. Prentice Hall. 2005.

7 Jorgensen, M. A Review of Studies on Expert Estimation of Software Development Effort. Journal of Systems and Software. 2004.

8 Based on standard Scrum ceremony durations for a two-week sprint minus the retained refinement meeting, applied to a representative ten-person team at mid-market US engineering salaries 2025. Salary assumptions derived from industry benchmarks in 2025.

9 Mark, G. Attention Span: A Groundbreaking Way to Restore Balance, Happiness and Productivity. Hanover Square Press. 2023. Documents the average 23 minute and 15 second recovery time after interruption based on longitudinal research at UC Irvine.

10 Womack, J., Jones, D., Roos, D. The Machine That Changed the World. Free Press. 1990.

11 Edmondson, A. Psychological Safety and Learning Behavior in Work Teams. Administrative Science Quarterly. 1999.

12 Forsgren, N., Humble, J., Kim, G. Accelerate: The Science of Lean Software and DevOps. IT Revolution Press. 2018, winner of the Shingo Publication Award.

13 Anderson, D. Kanban: Successful Evolutionary Change for Your Technology Business. Blue Hole Press. 2010.

14 Edmondson, A. The Fearless Organization. Wiley. 2018.

15 Rubinstein, J.S., Meyer, D.E., and Evans, J.E., Executive Control of Cognitive Processes in Task Switching. Journal of Experimental Psychology: Human Perception and Performance. 2001.

16 Stack Overflow. Developer Survey. 2023 and 2024.

17 Rother, M., Shook, J. Learning to See: Value Stream Mapping to Add Value and Eliminate Muda. Lean Enterprise Institute. 1998.

18 Juran, J.M. Juran on Quality by Design. Free Press. 1992. Articulation of Cost of Poor Quality as a framework for quantifying process waste.

20 Ladas, C. Scrumban: Essays on Kanban Systems for Lean Software Development. Modus Cooperandi Press. 2009.

21 Beck, K. Extreme Programming Explained: Embrace Change. Addison-Wesley. 1999.

22 SHRM. Benchmarking data on employee replacement costs. Society for Human Resource Management. 2022. Documents average replacement cost of 1.5 to 2 times annual salary for professional roles including software engineers.

23 Hinshelwood, M. Delivery is the Only Measure of Progress in Scrum. Scrum.org. June 2025.

24 Scrum Alliance. Story Point Estimation. scrumalliance.org. 2023. Notes that management use of story points as a performance metric leads to employee burnout and developers inflating estimates to show growth.

25 American Psychological Association. Multitasking: Switching Costs. apa.org. 2006. Citing research by Meyer et al. documenting up to 40 percent productive time loss from task switching.

26 Goldratt, E.M., Cox, J. The Goal: A Process of Ongoing Improvement. North River Press. 1984.

27 KPMG Belgium. Survey on Agility: Agile Transformation. KPMG. 2019. Survey of more than 120 organizations across 17 countries.

28 Sutherland, J. Why 47% of Agile Transformations Fail. Scrum Inc. 2020.

About the Author

Joe Hochrein is a software engineering leader with 28 years of engineering experience and 20 years in leadership roles. He holds PSM I, PSM II, and PSFS certifications from Scrum.org, and has led engineering organizations across e-commerce, SaaS, and enterprise software. FLOW grew out of his direct experience applying and evolving Scrum in production engineering environments.