

CLOSING THE LOOP

Putting Business Automation Back in the Hands of the
People Who Know the Business

A White Paper by Keith Hudson, J.D.

October 2026

***“The closer the person building the automation is to the person
who understands the work, the shorter the feedback loop.”***

Copyright © 2026 Keith Hudson. All rights reserved.

AI assistance disclosure: This white paper was conceived and directed by Keith Hudson and developed with research, drafting, fact-checking, and editorial assistance from OpenAI's ChatGPT.

Keith Hudson reviewed and approved the argument, interpretations, and final text and is responsible for the published work.

Executive Summary

Organizations invest heavily in software, yet many repetitive business processes remain manual because the distance between the people who understand the work and the people or tools that can automate it is too great. This white paper examines that distance as a feedback-loop problem.

The historical arc begins with early computing, when many programmers worked close to both the machine and the scientific, mathematical, or engineering problem being solved. As programming became a specialized profession, organizations gained valuable technical expertise but also accumulated handoffs between subject-matter experts, analysts, developers, testers, managers, and users. Large-system methods such as those described by Winston Royce attempted to control the risk of late discovery through preliminary design, extensive documentation, formal testing, customer involvement, and, when necessary, early pilot versions. Agile later emphasized working software and frequent feedback as another way to reduce the cost of misunderstanding.

For an important class of SharePoint-centered business automation, DFFS can shorten that loop further. Modern DFFS in SharePoint Online and Classic DFFS in on-premises SharePoint can allow subject-matter experts to implement rules themselves or work beside a developer who can prototype and revise solutions rapidly. When the subject-matter expert is also the programmer, the feedback loop can exist inside one person's head.

The opportunity is not simply faster form building. Mature use requires governance, testing, DEV-UAT-PROD discipline, appropriate documentation, and a clear understanding of when a solution has become complex enough to require deeper software engineering. Modern DFFS 2.x strengthens the technical foundation, but organizations still need the knowledge, habits, examples, and community required to turn a tool into an automation capability.

Central thesis: Organizations create value when they minimize unnecessary distance between the people who understand a business process and the people or tools that automate it.

Contents

Chapter 1. When the Programmer Was the Expert

Chapter 2. How Programming Became a Profession - and the Loop Got Longer

Chapter 3. Before Rapid Prototyping: Royce's Answer to Software Risk

Chapter 4. What Agile Was Really Trying to Fix

Chapter 5. The Shortest Feedback Loop Is Inside One Head

Chapter 6. When the Expert Isn't the Developer

Chapter 7. Low Code Without a Low Ceiling

Chapter 8. Speed Without Discipline Is Just Faster Risk

Chapter 9. The Economics of a Short Feedback Loop

Chapter 10. Automation Behind the Firewall

Chapter 11. From SharePoint Form to Organizational Automation Capability

Chapter 12. We Have Come Full Circle

Chapter 13. The Tool Is Only the Beginning

Sources

Chapter 1 - When the Programmer Was the Expert

By the end of this chapter, you will understand why some of the earliest computer programs had one organizational advantage that many modern software projects struggle to reproduce: the person who understood the problem was often very close to the person changing the machine.

When we picture the earliest computers, we tend to focus on the machines: rooms filled with equipment, blinking lights, patch cables, paper tape, and far less computing power than the phone in your pocket. But there is another part of the story that matters just as much: who was doing the programming.

Programming was not yet a mature occupation with a standard career path. Many of the people doing it came from mathematics, science, engineering, and the earlier profession of human computation. Two of the best-known ENIAC programmers, Frances Bilas and Betty Jean Jennings, both held mathematics degrees. The people working on early machines were often close to the scientific or engineering questions the machines were intended to solve.

That does not mean early programming was easy. Quite the opposite. Before modern high-level languages, programming required extraordinary attention to the details of the machine. ENIAC initially had to be configured through switches, cables, and function tables. In the early 1950s, John Backus later recalled, almost all programming was done in machine language or assembly language. Much of the programmer's effort went into dealing with the limitations of the hardware itself.

Then abstraction began to change the work. FORTRAN, developed at IBM in the 1950s, allowed programmers to express scientific calculations in a notation much closer to the mathematics of the problem and let the compiler translate that notation into machine instructions. The programmer could begin spending less time thinking about the machine and more time thinking about the problem.

Here is the distinction that matters for our story. Early programming was not simply modern software development carried out with primitive tools. The organizational relationship was often different. The subject expert and the implementer were frequently the same person, or at least worked very close to one another. The feedback loop between 'I need the computer to do this' and 'let me change what the computer does' could be remarkably direct.

As computing spread, that relationship changed. Businesses, governments, universities, and other organizations discovered more and more uses for computers. Programming became a profession of its own. That specialization was necessary and enormously productive. It also introduced something that did not have to exist when the problem expert and the programmer were one person: a translation step.

Someone now had to explain the business problem to someone else, who had to translate it into software. If the explanation was incomplete, the error could remain hidden until much later. The computer might execute the instructions perfectly while solving the wrong problem.

That is the idea we will follow throughout this white paper. The history of software is partly a history of increasing technical capability. But it is also a history of managing the distance between knowledge and implementation.

Modern tools have made programming incomparably easier than it was in the 1940s and 1950s. The remaining question is organizational: can we use those tools to get implementation capability close to the people who actually understand the work?

For certain kinds of business automation, especially inside SharePoint, I believe the answer is yes.

The machines have become easier to program. The opportunity now is to bring the programmer closer to the problem again.

Chapter 2 - How Programming Became a Profession - and the Loop Got Longer

By the end of this chapter, you will see how one of computing's greatest successes - specialization - also created a new problem: the distance between the person who knows what the software should do and the person who can change it.

As computers became useful to more organizations, programming could no longer remain a craft practiced mainly by a relatively small number of people working close to scientific and engineering problems. Businesses wanted payroll systems. Banks wanted transaction systems. Governments wanted records and reporting. Airlines wanted reservations. Manufacturing wanted planning and control. The demand for software expanded faster than the old model could support.

Programming became a profession.

That was progress. Specialized programmers could learn languages, data structures, operating systems, databases, testing techniques, and eventually the growing discipline of software engineering. Larger systems also required coordination among people with different expertise. Analysts, architects, developers, testers, database specialists, security specialists, operations staff, and project managers all brought skills that one person could not reasonably possess.

But specialization has a cost that is easy to overlook: every handoff can lengthen the feedback loop.

Imagine a business expert who understands a process thoroughly. She knows why one question must be asked only when another answer is 'yes.' She knows which exceptions matter, which approvals are merely customary, and which ones are mandated. She knows what users will misunderstand because she has watched them misunderstand it for years.

Now imagine that knowledge traveling through a chain: subject-matter expert to analyst, analyst to specification, specification to developer, developer to tester, tester to deployment, deployed system back to the subject-matter expert.

No one in that chain is doing anything wrong. In fact, each person may be highly competent. The problem is structural. Knowledge that begins richly contextual has to be compressed into words, diagrams, tickets, acceptance criteria, and meetings. Each translation creates an opportunity for delay or loss.

Robert C. Martin - one of the seventeen signers of the Agile Manifesto - has told a provocative version of this history in his talk 'The Future of Programming.' He describes the growth of a young programming profession and the increasing use of formal management processes as organizations tried to impose discipline on rapidly expanding software teams. His explanation is an interpretation, not a complete history of project management, which existed in other engineering fields as well. But the underlying observation is useful: as the population of programmers grew and software projects became organizational undertakings, management structures grew around them.

The important distinction is between specialization and separation. Specialization is often valuable. Separation is what becomes expensive. A security specialist should specialize in security. A professional developer should understand software engineering. But the organization pays a price when the people implementing a solution are so far from the business problem that every discovery must travel through a queue before it can change the software.

This is why the speed of the computer is often irrelevant to the speed of automation. A modern cloud platform can execute a rule in milliseconds, yet an organization may take three months to agree on what the rule should be.

SharePoint creates an interesting opportunity because the business content, the users, and the automation surface often already live in the same environment. A subject-matter expert may already own a list, understand the workflow around it, and know exactly what makes the default form inadequate.

If that person can also make the form intelligent - or can sit directly beside someone who can - the organizational chain can become much shorter.

The next question is what happens when organizations try to control long chains through process. That brings us to one of the most misunderstood documents in software history: the paper associated with the waterfall model.

Chapter 3 - Before Rapid Prototyping: Royce's Answer to Software Risk

By the end of this chapter, you will understand why one of the foundational papers in software engineering placed extraordinary emphasis on documentation - and why modern rapid-development tools give us another way to solve the problem Royce was confronting.

In 1970, Winston W. Royce published "Managing the Development of Large Software Systems." The paper is often associated with what later became known as the waterfall model: requirements followed by analysis, design, coding, testing, and operations.

But reading the paper through modern eyes can obscure just how different Royce's development environment was.

Royce was concerned with very large software systems, including software associated with spacecraft mission planning, commanding, and post-flight analysis. A serious design mistake discovered during testing could send a project back toward its beginning, producing enormous schedule and cost overruns.

His response was to try to discover as much as possible before the expensive parts of construction and testing.

That meant design early.

And it meant documentation - lots of it.

Royce called for a very high degree of documentation and described rigorous enforcement of documentation requirements as the first rule of software-development management. His comparison was striking. He estimated that controlling a \$5 million hardware procurement might require a 30-page specification, while an equivalent software procurement might justify roughly 1,500 pages.

That is about fifty times as many pages.

Why? Because before coding began, Royce viewed documentation, specification, and design as essentially the same thing. If the design had not been expressed clearly enough to document it, he did not consider the design complete. Documentation also allowed information to survive the transfer from designer to tester, operator, maintainer, and eventually the person modifying the system years later.

His documentation plan called for six documents, with five of them updated and current at final delivery. Documentation was not an afterthought. It was part of how Royce expected management to control a large and expensive software effort.

That makes considerable sense in the environment Royce was addressing.

Imagine trying to discover the requirements for a massive spacecraft software system in 1970. You could not sit beside a business user, rearrange a form, change three rules, refresh a browser, and ask, "Is this what you meant?"

Changing working software was expensive.

Consequently, organizations attempted to move discovery earlier in the process. Requirements were studied. Designs were developed. Decisions were documented. People tried to find mistakes on paper before those mistakes became software.

But Royce did not rely on documentation alone.

One of his five additional recommendations was essentially: build it twice.

For software containing significant new or uncertain elements, he advocated building an early pilot model or simulation. That first version could expose assumptions about timing, storage, interfaces, and other characteristics before the organization committed itself to the final implementation.

For a project expected to take 30 months, Royce suggested that this preliminary effort might consume about 10 months. Even for a 12-month project, he contemplated perhaps three months for the pilot.

That tells us something important.

Royce understood the value of learning from something more concrete than requirements documents.

The problem was the cost of producing that learning.

This gives us a more useful historical contrast than simply arguing about whether Royce did or did not invent "waterfall." Royce was attacking a problem that still exists today: How do we discover that our understanding of the requirement is wrong before correcting it becomes enormously expensive?

His available answers included extensive analysis, extensive documentation, preliminary design, formal customer involvement, and, when the uncertainty justified it, building an early version of the system.

Modern development tools give us another answer.

Make the first working version cheap enough that we can afford to learn from it almost immediately.

Consider a SharePoint process.

A subject-matter expert might explain that a particular field should appear only when another answer has a certain value. The developer can document that requirement carefully: "If Condition A is true, display Field B except when Condition C is also true, unless the current user belongs to Group D."

We can review that sentence. Approve it. Add it to the requirements document.

Or, using a tool such as DFFS, we may be able to implement the rule, show the form to the subject-matter expert, and ask: "Is this what you mean?"

The expert may immediately say: "Yes - but when the request comes from Accounting, that field should stay hidden."

We have just discovered a requirement that might never have emerged from another hour of documentation.

The distinction matters.

Documentation records knowledge.

Rapid prototyping can help create knowledge.

Those are not competing activities. Good software development often needs both.

For complex, regulated, long-lived, or high-risk systems, documentation remains indispensable. Future developers need to understand decisions. Testers need expected behavior. Operations personnel need instructions. Organizations need auditability and institutional memory. Royce was right to recognize that undocumented knowledge trapped inside one developer's head creates enormous risk.

But documentation should not have to carry the entire burden of discovering what users actually need.

This is where modern DFFS changes the economics of the feedback loop for an important class of SharePoint automation.

Royce recommended spending significant effort before final construction because discovering mistakes late was devastatingly expensive.

More than fifty years later, we have another option for many business processes: Make experimentation inexpensive.

Build something. Show it to the person who understands the work. Learn. Change it. Show it again.

And document what needs to survive after the conversation is over.

The objective has not changed since Royce wrote his paper.

Discover that we are wrong while being wrong is still cheap.

What has changed is how quickly we can find out.

Chapter 4 - What Agile Was Really Trying to Fix

By the end of this chapter, you will be able to separate Agile's core idea from the meetings, roles, and vocabulary that later accumulated around it.

Ask ten people what Agile means and you may hear ten different answers: sprints, standups, Scrum Masters, story points, backlogs, retrospectives, Kanban boards, product owners, velocity, or two-week releases.

Most of those things are not in the Agile Manifesto.

In February 2001, seventeen software practitioners met in Snowbird, Utah, looking for common ground among several approaches that were then being described as 'lightweight' alternatives to heavyweight, documentation-driven software processes. The resulting Manifesto for Agile Software Development expressed four preferences: individuals and interactions, working software, customer collaboration, and responding to change were valued more than the corresponding alternatives of processes and tools, comprehensive documentation, contract negotiation, and following a plan.

Notice what is not there. No standup. No sprint length. No points. No prescribed job titles.

The deeper idea is feedback.

Working software gives people something real to evaluate. Customer collaboration reduces the distance between the person defining the problem and the person implementing the solution. Responding to change acknowledges that understanding improves as the work progresses. Individuals and interactions recognize that software development is not simply the execution of documents by interchangeable resources.

That is why Agile belongs in our story. Royce's generation tried to reduce the cost of late discovery through preliminary design, extensive documentation, formal testing, customer involvement, and, where necessary, an early pilot. Agile pushed the feedback idea much further: shorten the interval between building working software and learning from it.

There is an important fence here. Agile is not the same thing as Scrum. Scrum can be one way of organizing work, but Agile's underlying principles can exist without a Scrum ceremony, and Scrum ceremonies can exist without much genuine agility. A team that holds daily standups but takes three months to show working software to the business has preserved the vocabulary while losing the point.

Agile also does not mean an absence of discipline. The Manifesto explicitly says there is value in the items on the right; it simply values the items on the left more. Good Agile teams still design, test, document, secure, and govern their systems. They simply try to learn earlier and more continuously.

Now consider the remaining distance. Even in a very good Agile team, the business expert and the developer are often different people. The customer can collaborate frequently with the programmer, but knowledge still has to cross from one mind to another.

For large systems, that is inevitable. For many small and medium business-process automations, it may not be.

Modern SharePoint environments are full of subject-matter experts who already structure information in lists, maintain process documentation, and understand the exceptions that make their work difficult. If the automation tool is accessible enough, some of those people can move from describing rules to implementing them. Others can work with a developer who changes the solution while they are still in the conversation.

That takes the Agile idea one step further. Agile asks, 'How quickly can we get feedback from the customer?' A citizen-development or rapid-prototyping model can sometimes ask a more radical question: 'How much of the customer-to-developer boundary do we need at all?'

That question leads to the central proposition of this white paper: the shortest feedback loop is inside one head.

Chapter 5 - The Shortest Feedback Loop Is Inside One Head

By the end of this chapter, you will understand why citizen development can sometimes shorten the feedback loop beyond what even an excellent Agile team can achieve.

Every organization has processes that are held together by human memory.

Someone knows that if the request is for this type of customer, another field must be completed. Someone knows that approvals under a certain amount can skip a level. Someone knows that a date cannot be earlier than another date, that three questions are irrelevant unless a previous answer is yes, or that an email has to go to a particular group whenever a status changes.

Often the process works because an experienced employee remembers all of that.

Now imagine that employee is using a SharePoint list. The list captures the data, but the default form does not know the process. It presents every field. It allows combinations that make no business sense. It leaves validation and routing to the user's memory.

Traditional development says: explain the rules to a developer.

Good Agile development says: explain the rules, let the developer build a small piece, look at it quickly, and keep iterating.

But there is an even shorter loop available in the right circumstances: give the person who already understands the rules a tool simple enough to implement them.

When the subject-matter expert is also the programmer, the feedback loop is inside the programmer's head.

That sentence is deliberately provocative, so it needs a fence. Citizen development is not simply professional software development performed by amateurs. The distinctive advantage is that some business knowledge no longer has to be translated at all. The person who knows why the rule exists can encode the rule directly.

Imagine the difference. In the traditional model, the SME says, 'When the request type is X and the amount is over Y, make the justification required, except for category Z.' The developer writes that down, interprets it, builds it, and brings it back. In the SME-as-builder model, the thought can become a rule while the context is still present in the same mind.

Modern DFFS is well suited to this class of problem because a SharePoint form can be changed visually and through rules before conventional code is required. Fields can be organized into sections, tabs, or steps. Rules can control visibility, validation, values, and actions. The person creating the solution does not need years of computer-science training just to make a business form behave intelligently.

This does not mean every employee should be given unrestricted authority to build mission-critical systems. Security, scale, integration, regulatory obligations, data architecture, and

business criticality all matter. The amount of professional engineering and governance should rise with risk.

But there is a large middle territory between 'do it manually forever' and 'fund a conventional software project.' That territory contains thousands of repetitive processes in organizations that already use SharePoint.

The citizen developer does not need to become a full-stack engineer. The organization needs to help the subject-matter expert recognize an automation opportunity, understand basic data and rule design, test the result, and know when the solution has crossed the threshold where professional help is appropriate.

That is why the phrase 'citizen developer' can sometimes undersell the idea. The important asset is not amateur programming skill. It is the combination of domain knowledge and enough implementation power to act on that knowledge without waiting for a long translation chain.

The goal is not to turn everyone into a software engineer. The goal is to remove unnecessary distance between knowledge and automation.

Chapter 6 - When the Expert Isn't the Developer

By the end of this chapter, you will see why the value of a short feedback loop does not disappear when a professional developer builds the solution.

The strongest version of the feedback-loop idea is easy to remember: when the subject-matter expert is also the programmer, the feedback loop is inside one head.

But that will not always be practical - and it does not have to be.

A subject-matter expert may have no interest in configuring forms. A process may require JavaScript, APIs, complex integrations, or design skills beyond what the SME wants to learn. An organization may deliberately separate development responsibilities for governance reasons.

The next best thing is to put the SME and the developer in front of the same working solution.

That changes the nature of requirements gathering.

In a traditional handoff, a developer may ask, 'Tell me what you want.' The answer becomes notes, requirements, tickets, mockups, or acceptance criteria. The developer then leaves the conversation and turns those representations into software.

With rapid prototyping, the question becomes, 'Is this what you mean?'

That is a fundamentally different conversation. The working form becomes a shared language between the business expert and the developer.

Suppose an SME says, 'Those three fields should only show for an exception request.' In a conventional cycle, that statement enters a backlog. In a rapid DFFS session, the developer can create or change the visibility rule, refresh the form, and ask the SME to try it. The SME may immediately notice something that was not included in the original statement: 'Actually, one of those fields is also needed for emergency requests.'

Nothing failed. The prototype did exactly what good prototypes are supposed to do: it surfaced knowledge that had not yet been expressed.

The fence is important. Rapid prototyping is not merely typing faster. Its advantage is not that one developer can produce more configuration per hour than another. Its advantage is that implementation becomes part of the communication process. The interval between statement, interpretation, implementation, and correction can shrink from days or weeks to minutes.

DFFS supports this particularly well because much of the behavior of a SharePoint form can be changed through configuration and rules without rebuilding an application from the ground up. When deeper behavior is needed, professional developers can extend the solution with JavaScript, CSS, REST data, and other capabilities rather than abandoning the rapid-feedback model.

There are limits. A working form cannot reveal every requirement. Security architecture, performance, data retention, integration failure modes, accessibility, compliance, and long-term

maintainability still require deliberate analysis. Rapid feedback should complement engineering judgment, not replace it.

But for the part of software development that is notoriously difficult - understanding what users actually need - showing something that works is extraordinarily powerful.

This also changes the role of the developer. The developer is no longer merely a translator who disappears with requirements. The developer becomes a collaborator who can expose possibilities, suggest simpler designs, identify risks, and help the SME refine the process while it is being automated.

In that model, expertise on both sides becomes more valuable, not less. The SME brings deep business context. The developer brings technical depth. DFFS reduces the friction between them.

The shortest feedback loop may be inside one head. The next best loop may be two heads looking at the same working form.

Chapter 7 - Low Code Without a Low Ceiling

By the end of this chapter, you will understand why the most useful low-code tools are not necessarily the ones that eliminate code, but the ones that postpone it until code actually adds value.

Low-code platforms make an attractive promise: let people build useful applications without requiring them to become conventional programmers.

That promise solves one problem and can create another. What happens when the easy visual tool reaches a requirement it was never designed to handle?

Some platforms have a low floor but also a low ceiling. They make simple things easy, then force the builder into awkward workarounds when the application becomes more sophisticated. Other tools are powerful but expose so much complexity at the beginning that the subject-matter expert never gets started.

The better model is progressive complexity.

Start with the simplest mechanism that can express the business requirement. Move deeper only when the requirement justifies it.

Modern DFFS follows that pattern. A maker can begin with a SharePoint list and use a visual form builder to arrange fields, create tabs or steps, and add visibility and validation rules. Professional capabilities extend the rule set and add features such as writing to other lists, integration actions, multi-language support, related items, repeating sections, REST data sources, and a broader field toolkit. When a requirement genuinely needs code, Professional also exposes custom JavaScript and CSS and a CRUD API.

That gives us the fence for this chapter: low code is not the same thing as no code.

'No code' treats the absence of programming as the objective. That can become a constraint. A better objective is to solve each problem at the lowest appropriate level of complexity while preserving an escape hatch for requirements that deserve conventional code.

Think of it as a ladder. A subject-matter expert might solve a problem with layout and rules. A power user might add integrations. A professional developer might add JavaScript around an unusual business requirement. The organization does not have to replace one development model with another; it can select the appropriate rung.

Modern DFFS 2.x has pushed that idea further with an AI Assistant that can assemble or modify form configuration from natural-language prompts. Importantly, DFFS does not bundle an AI model or token allowance; the organization brings its own supported AI access or uses a copy-and-paste workflow with an AI tool it already has. The significance is not that AI replaces developers. It is that another layer of friction can be removed between describing an intended form behavior and producing a configuration that can be reviewed and tested.

The qualification matters as much as the opportunity. Once a DFFS solution contains substantial JavaScript, integrations, shared code, or business-critical logic, it is software. It deserves version

awareness, testing, naming conventions, documentation, security review, and controlled deployment. Low-code origins do not exempt a solution from engineering discipline.

That is one reason the 'citizen developer versus professional developer' debate can be misleading. They are not necessarily competing roles. A well-designed platform lets work begin close to the SME and move toward deeper engineering only as the problem demands it.

The organization benefits twice. Easy requirements do not consume expensive development capacity unnecessarily. Difficult requirements do not have to be abandoned merely because the first version was created visually.

The ideal low-code platform does not eliminate code. It delays code until code earns its keep.

Chapter 8 - Speed Without Discipline Is Just Faster Risk

By the end of this chapter, you will understand why rapid development makes disciplined testing and controlled deployment more important, not less.

One of the great attractions of a tool such as DFFS is speed. A form can sometimes be changed in minutes instead of waiting for a conventional development cycle.

That creates a temptation: if a change takes five minutes, why not make it directly in production?

Because a five-minute change can still create a five-day problem.

Rapid development and uncontrolled development are not the same thing. That is the fence around this chapter. Speed describes how quickly we can create or modify a solution. Discipline describes how safely we determine whether that solution should be used by everyone else.

Organizations that have used SharePoint for years often have a curious maturity gap. Their professional application teams may use development, test, user-acceptance, and production environments, while departmental SharePoint solutions are edited directly where users work. A person can be called a SharePoint developer without ever having worked routinely in a DEV-UAT-PROD promotion model.

Modern DFFS 2.x creates an opportunity to close that gap without sacrificing the speed that makes rapid development attractive.

The Modern DFFS Admin tool can inventory and back up configurations and promote them through defined environment chains such as DEV to SIT to UAT to PRD. Before overwriting a target, it can snapshot the existing configuration, and promotion includes checks for whether required fields exist in the target list. Modern DFFS 2.1 also introduced saved form tests, and 2.3 expanded the ability to run tests on the real form.

Those capabilities matter because they make software-engineering discipline less expensive to apply to low-code work.

A sensible pattern can be simple. Build in DEV, where experimentation is safe. Move to UAT, where the people who understand the process confirm that the form behaves correctly. Then promote to production in a controlled step. After later changes, rerun the tests that describe what the form is supposed to do.

This is not bureaucracy for its own sake. It is a way to preserve the short feedback loop while protecting real users from experiments.

The amount of control should match the risk. A small team list used as a convenience may not need the same approval process as a form that controls financial decisions or regulated records. Governance should be proportional, not ritualistic.

But the principle should remain: fast construction does not imply immediate release.

This also improves the relationship between citizen developers and IT. Many governance conflicts begin when IT sees low-code development as invisible, untested production change. A

lightweight DEV-UAT-PROD discipline makes the work more observable and supportable. It creates a shared language between rapid builders and traditional software professionals.

The result is a more mature automation capability. The organization can still exploit the fact that a DFFS rule takes minutes to change, but it can also prove that the rule was tested, accepted, promoted deliberately, and recoverable if something goes wrong.

That is the difference between moving quickly and gambling quickly.

Speed is valuable. Repeatable speed is an organizational capability.

Chapter 9 - The Economics of a Short Feedback Loop

By the end of this chapter, you will understand why the cost of an automation platform cannot be evaluated by comparing license prices alone.

When organizations compare automation platforms, the conversation often begins with licensing. That is understandable because licenses are visible, quotable, and easy to put into a spreadsheet.

But the license is only one part of the cost of automation.

Consider the full chain: requirements gathering, meetings, development, testing, rework, deployment, support, licensing, and the business cost of waiting while people continue performing the process manually. A product with a low license price can still produce an expensive solution if every change requires a long development cycle. A product with a higher initial price can be economical if it collapses months of labor into days.

The most important economic effect of a short feedback loop is not that the developer types faster. It is that misunderstandings are discovered while they are still cheap.

Now add licensing architecture to the picture.

Microsoft 365 includes limited Power Apps and Power Automate use rights intended for scenarios using Microsoft 365 data and standard connectors. That can make Power Apps an excellent and economical option for many SharePoint solutions. But Microsoft also classifies capabilities such as premium connectors, Dataverse use in custom applications, on-premises gateways, and custom APIs as premium scenarios that can require standalone Power Apps licensing. Microsoft currently lists Power Apps Premium at \$20 per user per month through the end of 2026, with a published increase to \$22 beginning January 1, 2027. Microsoft also lists a lower price for very large qualifying volumes.

Modern DFFS uses a different licensing model. Its current pricing is based on site collections or an organization-wide license rather than charging every form user a DFFS per-user fee. As of this writing, Professional is \$1,100 for a site collection in year one with a 20 percent annual renewal; company-wide Professional pricing ranges by organization size. Enterprise adds central governance and included features such as the AI Assistant, while individual add-ons are also available on Professional.

Those numbers should not be turned into a simplistic claim that DFFS is always cheaper. That would be wrong.

The proper fence is this: application economics are not the same thing as license economics. Power Apps using only capabilities already included in an organization's Microsoft 365 entitlement may involve little or no incremental platform license cost. A DFFS solution may require a paid DFFS license. In a different scenario, a large user population that needs premium Power Apps rights can make per-user licensing a major part of total cost while DFFS licensing remains tied to the site or organization.

The right comparison therefore begins with the process, not the product. How many users? What data sources? What integrations? What development skills? How many changes per year? How

much time is spent translating requirements? How quickly must the process evolve? What governance is required?

Then measure total cost over time.

This is where DFFS can create a particularly strong value proposition for organizations already committed to SharePoint. The data may already live in lists. Users already know the environment. The form can often be automated without introducing a new application stack. The SME may build part of the solution directly, or a developer may prototype with the SME in real time.

Licensing can be a bonus. The deeper economic advantage is removing expensive distance from the development process.

The cheapest license does not necessarily produce the cheapest automation. The most economical solution is the one that satisfies the requirement with the least total complexity, delay, rework, and recurring cost.

Chapter 10 - Automation Behind the Firewall

By the end of this chapter, you will see why the short-feedback development model is not inherently dependent on the public cloud.

Modern technology conversations often treat modernization and cloud migration as though they were the same thing.

They are not.

Some organizations cannot place every workload in a public cloud. Defense, intelligence, regulated infrastructure, sensitive research, and other environments may operate disconnected networks, tightly controlled on-premises systems, or other architectures where internet access and cloud services are unavailable or inappropriate.

Those organizations still have repetitive processes. They still have forms. They still have users who understand business rules that could be automated. And they still benefit from short feedback loops.

This is where the distinction between Modern DFFS and Classic DFFS matters. Modern DFFS is built for modern SharePoint and Microsoft 365. SPJSWorks explicitly directs on-premises SharePoint customers to Classic DFFS, which remains available for older SharePoint environments.

The fence around the idea is simple: cloud modernization and process modernization are different things.

A team does not have to move its data to SharePoint Online in order to improve a SharePoint form, make it respond intelligently to user input, or reduce repetitive manual steps. The same organizational principle applies on premises: keep the automation work close to the people who understand the process.

That can be especially important in environments where development resources are scarce or deployment cycles are expensive. If every small business-process improvement requires a conventional custom application, many worthwhile automations will never compete successfully for resources. A configurable forms layer can lower the threshold at which improvement becomes practical.

Classic DFFS also gives the DFFS story continuity. The central idea did not begin with AI or SharePoint Framework. DFFS has been extending SharePoint forms since 2012. Modern DFFS carries that experience into SharePoint Online, while Classic DFFS remains relevant where the infrastructure itself must remain on premises.

There is an important qualification. Saying that Classic DFFS can run in an on-premises SharePoint environment does not mean that any particular classified, air-gapped, regulated, or security-sensitive organization is authorized to deploy it. Each organization has its own approval, accreditation, supply-chain, and configuration-control requirements. Capability is not the same thing as authorization.

But the broader point remains useful. The feedback-loop principle is independent of hosting model.

A subject-matter expert using an on-premises SharePoint system can still collaborate directly with a developer. A developer can still prototype rules and form behavior quickly. A business process can still be improved without creating an entirely separate application stack.

That makes this more than a cloud product story. It is a development-model story.

Good ideas survive changes in infrastructure. Shortening the distance between expertise and implementation is one of them.

Chapter 11 - From SharePoint Form to Organizational Automation Capability

By the end of this chapter, you will understand how a series of small SharePoint improvements can become something larger: a repeatable organizational ability to automate work safely.

A successful automation often begins with a very small question.

Can we hide these fields unless they are relevant? Can this value fill itself in? Can the form stop users from submitting an impossible combination? Can we route this request automatically?

Then something predictable happens. People see the improved form and ask, 'Can we do that with this other process too?'

That is the point at which an organization can either accumulate isolated solutions or begin building an automation capability.

Those are not the same thing.

A collection of automated forms is simply a collection. An organizational capability means the next solution becomes easier, safer, and more predictable because the organization learned from the previous one.

One way to think about maturity is as a progression.

At the first level, the process is largely manual: email, spreadsheets, documents, and human memory. At the second, information becomes structured in SharePoint lists. At the third, forms become dynamic and adapt to context. At the fourth, rules, calculations, notifications, and other automation reduce repetitive work. At the fifth, solutions integrate with other systems where needed. At the sixth, development is governed through DEV, UAT, production, repeatable testing, and recoverable releases. At the seventh, patterns, components, documentation, and expertise are deliberately reused across the organization.

The fence is repeatability. A clever form built by one talented person is not yet an organizational automation capability. The capability exists when another team can solve a similar problem faster because the organization has reusable knowledge, governance, examples, and tools.

Modern DFFS 2.x contains several ingredients that can support that progression. The Admin tool can inventory configurations and promote them across environments. Enterprise centralization can provide a common license, global code and styling, reusable content, and centralized AI access patterns. Reusable Content can allow centrally maintained HTML, JavaScript, and CSS to be referenced by many forms instead of copied repeatedly.

But technology cannot manufacture maturity by itself. The organization still needs conventions. When should a solution use a separate DEV list or site? What must be tested? Who approves production promotion? Which shared JavaScript is centrally maintained? When should an SME ask a professional developer for help? How are abandoned applications identified?

It also needs restraint. Not every process should be automated. A broken process can become a faster broken process after automation. A task performed once a year may not justify the maintenance burden of a custom solution. A process that changes weekly may need redesign before it needs software.

The qualification, therefore, is that automation maturity includes knowing when not to automate.

For suitable processes, however, the payoff compounds. The first DFFS project teaches form building. The second teaches reusable rule patterns. The third exposes a governance need. The fourth benefits from a shared component. Eventually the organization stops treating each request as an isolated project and begins treating business automation as a learned capability.

That is a much larger value proposition than a better SharePoint form.

The real payoff is an organization that gets progressively better at automating itself.

Chapter 12 - We Have Come Full Circle

By the end of this chapter, you will see how more than seventy years of software history points back toward a surprisingly simple idea: put implementation capability close to the people who understand the problem.

We began this white paper with the earliest programmers - people working close to the machine and, often, close to the scientific or engineering problem the machine was intended to solve.

Then computing expanded. Programming became a profession. Specialization made far larger and more capable systems possible, but it also created distance between the person who knew the work and the person who changed the software.

Organizations responded with process. Royce's 1970 paper reflects one influential answer: perform preliminary design early, document extensively, plan and control testing, involve the customer, and, for uncertain systems, build a pilot before the final version. Later development methods would place increasingly greater emphasis on obtaining feedback from working software sooner and more often.

Agile pushed in the other direction. Working software, customer collaboration, interactions, and responsiveness to change all shortened the learning cycle. Instead of assuming we could understand everything before development began, Agile made learning part of development.

Low-code tools shortened the cycle again by reducing how much specialized programming knowledge was required to implement many business rules.

Now consider where that trajectory leads in SharePoint.

For an important class of business-process automation, a subject-matter expert can already own the data and understand the process. With an accessible tool, that person may also be able to configure the form and rules. When more technical depth is required, a developer can work beside the SME and prototype changes quickly rather than disappearing into a long development queue.

That looks, in one narrow but important respect, like a return to the beginning.

The fence matters. Coming full circle does not mean returning to 1950s technology, programming practices, or organizations. Modern systems are vastly more complex, and professional software engineering remains indispensable. The similarity is much more specific: domain knowledge and implementation capability can once again occupy the same head - or at least the same conversation.

DFFS is not the only technology that can support that idea, and it should not be used for every problem. But it is particularly interesting inside organizations that already use SharePoint because it begins where the work already lives. The list does not have to be moved into a new platform before the form can become intelligent. A simple requirement can be handled visually. Advanced requirements can move into rules, integrations, or code. Classic DFFS keeps the model available on premises, while Modern DFFS extends it in Microsoft 365.

The qualification is equally important. Departmental and business-process automation is not the whole software world. Enterprise platforms, safety-critical systems, high-scale applications, complex transactional architectures, and specialized products will continue to require experienced engineers and dedicated architectures.

But enormous amounts of organizational work do not belong in those categories. They are requests, approvals, inventories, inspections, reviews, exceptions, onboarding steps, checklists, case records, and dozens of other repetitive processes that are important enough to improve but often too small to win a conventional development budget.

For those processes, the most important innovation may not be another programming language or another application platform. It may be the restoration of proximity.

The person who understands the problem should be able to participate directly in creating the solution.

Perhaps the future of programming is, in one important respect, a return to its beginning.

Chapter 13 - The Tool Is Only the Beginning

By the end of this chapter, you will understand why the release of Modern DFFS 2.x creates an important opportunity for SharePoint organizations - and why realizing that opportunity depends more on people and practices than on software alone.

On August 9, 2026, Modern DFFS 2.0 was released.

The date is worth marking because 2.0 did more than add another group of form features. It established a new 2.x line with three license tiers, an AI Assistant, tenant-wide governance capabilities, additional developer tools, richer form elements, and a stronger path from individual form building toward centrally governed automation. The releases that followed moved quickly: 2.1 on September 12 added form tests; 2.2 arrived September 21; and 2.3 on September 24 expanded testing and other capabilities further.

Those features strengthen the value proposition we have been exploring throughout this white paper: shorten the distance between the person who understands the work and the means of automating it.

But releasing the technology does not create that capability inside an organization.

That is the fence for the final chapter: a powerful development tool is not the same thing as an organizational development capability.

Modern DFFS can provide a visual builder, rules, advanced actions, custom JavaScript and CSS, reusable content, AI-assisted configuration, configuration promotion, backups, testing, and tenant-level governance. It cannot, by itself, teach a subject-matter expert how to recognize a good automation opportunity. It cannot decide how much testing a process needs. It cannot create good requirements-discovery habits. It cannot make a developer collaborate closely with the business. It cannot decide where DEV ends and UAT begins. Those are human and organizational disciplines.

This is why I believe the work ahead is primarily educational.

We need to help subject-matter experts notice the repetitive decisions they make every day and ask whether those decisions can be expressed as form behavior or rules. We need to teach developers to use rapid prototypes as part of requirements discovery rather than treating requirements as a document handed over before development starts. We need examples that begin simply and grow into sophisticated, governed solutions. We need repeatable DEV-UAT-PROD patterns that are light enough for departmental automation but disciplined enough to protect production. We need tests, naming conventions, reusable code, and clear escalation points for when a citizen-built solution has become complex enough to deserve deeper engineering.

We also need evidence.

If the claim is that this approach can reduce development time and cost, we should measure it. How many hours did the first version take? How many iterations occurred while the SME and developer were together? How long would a similar change have taken through the

organization's normal queue? How much rework was avoided by seeing the prototype early? What licenses would the alternative architecture have required?

And we need a community that spreads those lessons.

Modern DFFS remains a relatively little-known option compared with the enormous visibility of Microsoft's own Power Platform. That is not simply a marketing problem. It is a knowledge-transfer problem. People cannot choose a development model they do not know exists, and they cannot use it maturely if all they see is a list of product features.

The qualification is that Modern DFFS is not the answer to every automation problem. Some applications belong in Power Apps. Some need custom software. Some require specialized platforms or architectures. The opportunity is to become much better at recognizing the class of problems for which DFFS can dramatically shorten the path between 'we do this manually every day' and 'the system now does that for us.'

The historical journey began with problem experts working close to the machines. Specialization increased the distance. Documentation-heavy staged development tried to control the cost of late discovery. Agile shortened the loop. Low code can shorten it further. Modern DFFS 2.x gives SharePoint organizations another way to make that loop very short for the right problems.

But an opportunity is not an outcome.

The technology can shorten the feedback loop. Our job now is to develop the knowledge, examples, discipline, and community that teach people how to close it.

Sources

The following primary and authoritative sources support the historical, technical, and licensing claims used in this white paper. Product features and licensing can change; the information presented in this white paper was current as of October 1, 2026, to the best of the author's knowledge.

1. Patrick Winston, 'How to Speak' transcript, MIT OpenCourseWare

https://ocw.mit.edu/courses/15-371-innovation-teams-fall-2024/Unzc731iCUY_transcript.pdf

Primary source for empowerment promise, cycling, and the fence principle.

2. Manifesto for Agile Software Development

<https://agilemanifesto.org/>

Primary source for the four Agile values and the seventeen signatories.

3. History: The Agile Manifesto

<https://agilemanifesto.org/history.html>

Primary history of the February 2001 Snowbird meeting; notes Robert C. Martin's role in convening the meeting.

4. Robert C. Martin, 'The Future of Programming'

<https://www.youtube.com/watch?v=ecIWPzGEbFc>

Source of Martin's interpretation of programming demographics, waterfall, Agile, and the growth of management/process around the profession. Treat as Martin's interpretation, not sole historical authority.

5. Winston W. Royce, 'Managing the Development of Large Software Systems'

<https://cse.msu.edu/~cse435/Homework/HW3/royce1970.pdf>

Original 1970 paper. Primary source for Royce's preliminary-design emphasis, extensive documentation plan, pilot-model ('do it twice') recommendation, testing controls, and customer involvement.

6. John Backus, 'The History of FORTRAN I, II, and III'

<https://research.ibm.com/publications/the-history-of-fortran-i-ii-and-iii>

Primary retrospective explaining that before 1954 most programming was in machine or assembly language.

7. Computer History Museum, ENIAC programmers

<https://www.computerhistory.org/revolution/birth-of-the-computer/4/78/317>

Documents Frances Bilas and Betty Jean Jennings as ENIAC programmers with mathematics degrees.

8. IBM, History of FORTRAN

<https://www.ibm.com/history/fortran>

Background on FORTRAN and the abstraction of software from hardware.

9. SPJSWorks, Modern DFFS changelog

<https://spjsworks.com/changelog/>

Primary source for Modern DFFS 2.0 release on August 9, 2026 and subsequent 2.x releases.

10. SPJSWorks, Pricing & Editions

<https://spjsworks.com/pricing/>

Primary source for current editions, features, site-collection/company licensing, renewals, and Classic vs Modern positioning.

11. SPJSWorks, Modern DFFS Admin guide

<https://spjsworks.com/documentation/doc-admin-tool/>

Primary source for inventory, snapshots, DEV/SIT/UAT/PRD promotion chains, and field compatibility checks.

12. SPJSWorks, About Modern DFFS

<https://spjsworks.com/about/>

Primary source for DFFS history since 2012, Modern DFFS positioning, and Classic DFFS availability for on-premises SharePoint.

13. SPJSWorks, AI Assistant

<https://spjsworks.com/product/ai-assistant/>

Primary source for AI-assisted configuration and the bring-your-own-AI model.

14. Microsoft, Power Platform licensing FAQ

<https://learn.microsoft.com/en-us/power-platform/admin/powerapps-flow-licensing-faq>

Current Power Apps licensing details, including \$20/user/month Power Apps Premium and premium connector requirements.

15. Microsoft, Power Apps licensing FAQs

<https://learn.microsoft.com/en-us/power-platform/admin/powerapps-licensing-faq>

Current premium entitlement guidance and February 2027 enforcement notes.

16. Microsoft, Power Apps Premium price update

<https://www.microsoft.com/en-us/licensing/news/power-apps-premium-price-update>

Announces the increase from \$20 to \$22 per user/month effective January 1, 2027.

Copyright and AI assistance

Copyright © 2026 Keith Hudson. All rights reserved. Research, drafting, fact-checking, and editorial assistance were provided with OpenAI's ChatGPT under the author's direction. The author retains responsibility for the argument, interpretations, and published text.