Manual Work is a Bug

**A.B.A: ALWAYS
BE AUTOMATING**

THOMAS A. LIMONCELLI

Let me tell you about two systems administrators I know. Both were overloaded, busy IT engineers. Both had many repetitive tasks to do. Both wanted to automate these tasks. After observing these two people for a year, I noticed that one made a lot of progress, while the other one didn't. It wasn't a matter of skill—both were very good software engineers. The difference was their approach, or mindset.

I'd say that the successful one had a mindset of always thinking in terms of moving toward the goal of a better automated system. Imagine an analog gauge that points to the left when measuring that a process is completely manual but slides to the right as progress is made toward a fully autonomous system. The developer mindset is always intent on moving the needle to the right.

The less successful person didn't write much code, and he had excellent reasons why: I'm too busy! The person who made the request can't wait! I have 100 other things to do today! Nobody's allocating time for me to write code!

The successful person had the same pressures

but somehow managed to write a lot of code. The first time he did something manually, he documented the steps. That may not be code in the traditional sense, but writing the steps in a bullet list is similar to writing pseudocode before writing actual code. It doesn't run on a literal computer, but you run the code in your head. You are the CPU.

Automation is putting process into code. A bullet list in a process document is code if it is treated that way.

The second time the successful engineer did something manually, he followed his own documentation. This might seem strange since he knew the process well enough to document it, but by following his own documentation, he found opportunities to improve it. He made corrections, augmented the command-line snippets he had recorded, and so on.

As he repeated this process over and over, the document grew to be much better. The command-line examples were replaced with parameterized commands, using variables instead of examples. Ambiguous statements such as "make sure everything is ok" were replaced by checklists of things to be tested, which were soon augmented by commands that performed the tests directly.

Soon this manual process was feeling more and more like real automation. There was less thinking, more following orders. Doing the process "manually" was more like copying and pasting command-line snippets from the document and pasting them in his terminal window. I call this PasteOps.

By doing this in the open, with collaborative document systems such as a wiki or Git repository, coworkers are

able to join in: Mary fixes a command line that broke on a certain class of machine. Joe does some web searches and soon a step that previously required a mouse click is replaced by a command.

As more and more coworkers adopt this work style, the entire team contributes to the constant goal of better automation.

This engineer, who began with the same time pressure and other obstacles as the other less successful engineer, has not yet written any traditional code, so to speak, but the process has become much more automated.

In the future these smatterings of command-line snippets will be combined into one big program that automates the entire process. This tool will be used as the basis for a web-based self-service portal. This will allow users to do the task on demand, seven days a week, even when the sysadmins are asleep.

Meanwhile, the other engineer, the one who was “too busy to write code,” is no closer to getting started.

The difference between these two engineers is that one is willing to do work manually just to get the task done. The other is willing to do work manually only as a mechanism for generating artifacts (documentation and code snippets) that “move the needle” toward an automated world.

A CULTURE OF AUTOMATING

People who are successful at automating tasks tend to work this way in every aspect of their jobs. It is just how they work; it is part of their culture.

Manual work should not be tolerated unless it generates an artifact or improves an existing one.

The successful engineer has a quick way to create documents for new procedures and to find existing procedures. People with this mindset avoid the cognitive load of deciding whether or not a task is worth documenting, because they document everything. On finding a potential improvement, they are not slowed by the speed bump of switching from a document viewer to a document editor because they work from the editor at the start. Heck, they have a dedicated second monitor just for their editing app!

People with this culture revise documents in realtime. Meanwhile, the less successful engineer has a stack of notes that he honestly plans on entering into a document someday soon—perhaps the same “someday” when he will start writing code.

The successful engineer realizes that the earlier he starts collaborating, the sooner others can contribute. Together they can create a culture of documentation that spreads throughout the team. Thus, every project is collaborative and has a “stone soup” feeling, as all are invited to bring their skills and insights. The more people who embody this culture, the more success it has.

This culture can be summarized in two sentences: (1) Every manual action must have a dual purpose of completing a task and improving the system. (2) Manual work should not be tolerated unless it generates an artifact or improves an existing one.

FOUR PHASES

Traditional software development involves requirements gathering and so on. In the culture of automation, we

wiggle and iterate among four overlapping phases: [1] document the steps; [2] create automation equivalents; [3] create automation; and [4] create self-service and autonomous systems.

Phase 1: Document the steps

At the start, developers perform a task manually to learn the process. They keep good notes and record what they do for each step. This is often exploratory or may require interviewing experts on how to do the process. They produce an artifact—written documentation describing how the process is done.

Beginning programmers are taught to write a program in pseudocode first, then turn each line of pseudocode into actual code. The same applies to automation: if you can't describe the process in writing, you can't automate it.

Documentation is automation. Following a step-by-step guide is automation: you are the CPU; you are following the instructions. As with any prototyping language, you should not expect perfection, but learning. You have the benefit of being able to spot and fix problems along the way. You are a CPU that improves the code as it executes!

There is no reason to wait for the document to be perfect before moving onto the next phase. All that is required is that the people involved gain the minimum necessary confidence in the document to move forward.

Phase 2: Create automation equivalents

As the document matures, manual action generates a new kind of artifact: command-line snippets. The document is augmented with automated equivalents for each step.

At first, you simply paste the command line used to perform the step into the document as is. The next time you manually perform the task, you improve it—perhaps by rewriting it. Over time these command examples become fully functional code snippets.

Other improvements happen. Mouse clicks are replaced by commands. Ambiguous statements such as “make sure everything is ok” are replaced by checklists of what to test for, which are soon augmented by commands that perform the tests directly.

Mouse clicks and other GUI actions that have no API or command-line equivalent are noted. Bugs are filed with the vendor and the bug ID is added to the document. (As a manager, I am unsatisfied when engineers tell me, “Oh, the vendor knows that’s a problem,” but can’t show me a bug ID. I say, “Bug ID, or it didn’t happen.”)

Yes, the process is still being done manually, but now each manual iteration is done by setting variables and pasting lines of commands into the terminal window. Each manual iteration tests the accuracy of the snippets and finds new edge cases, bugs, and better ways of verifying the results.

Phase 3: Create automation

Soon these command-line snippets are turned into longer scripts. Like all good code, this is kept in a source-code repository. The artifacts begin looking more like real software.

Perhaps the code is performing only certain steps or works in only a narrow set of circumstances. Each manual iteration, however, expands the code to cover new

use cases. No manual iteration should leave the scripts unimproved. In fact, it should be the other way around. Each manual iteration is simply a test for the most recent improvements. You should look forward to finding an edge case that breaks the code because this is an opportunity to fix the problem.

Often the entire process is more complex than is appropriate for a scripting language. Turning snippets of PowerShell or Bash into stand-alone scripts is easy, but it is often better to write larger programs in languages such as Python, Ruby, or Go. The individual snippets usually translate easily, and when they don't, a reasonable stopgap measure is to have the program "shell out" to run the command line. These can be "downcoded" into the native language later as needed.

Since February 2015, the SRE [site reliability engineering] team at Stack Overflow has switched from a mixture of Python and Bash to Go. Even though Go isn't a scripting language, for small programs it compiles and runs nearly as fast as Python takes to start. At Stack Overflow we tend to prefer compiled, type-checked languages for large programs, especially when multiple people are collaborating, and, therefore, no one person is familiar with every line of code. Our policy was that Bash scripts couldn't be larger than 100 lines and Python programs couldn't be larger than 1,000 lines. Those seemed like reasonable limits. Rewriting scripts when they grew beyond the limit, however, was a lot of work. It was better to start in Go and avoid the conversion.

Phase 4: Self-service and autonomous systems

In the next phase the script becomes a stand-alone tool, which then becomes part of a larger system, usually with a web-based front end. Ideally, some kind of self-service portal can be created so that users can activate the automation themselves. Even better is to create an autonomous system. The difference between *automated* and *autonomous* is the difference between a *tool* that someone can use to create new user accounts, and a *system* that monitors the HR database and creates and deletes accounts without human intervention. Autonomous systems eliminate the human task.

Depending on how frequently the task is needed, this phase may not be worth the effort. The return on investment may indicate that stopping at the tool stage is sufficient. CI (continuous integration) systems such as Jenkins and runbook automation systems such as Rundeck, however, make it easy to create simple, RBAC (role-based access control) restrained, self-service portals.

DISCIPLINE

Maintaining this culture and not backsliding takes discipline. Every manual iteration must move you closer to better automation.

It is tempting to revert to the old methods or skip updating the documentation “just this once” because you are in a hurry, or you’ll fix it next time, or the new system is broken, or you’re not in a good mood today. The developer mindset, however, resists such temptations and treats every manual iteration as an opportunity that should not be squandered.

Doing something manually “because it is faster” is often a sign that engineers feel pressure, but they do not realize they are mortgaging their future. In reality, the old way may feel faster only because they are more comfortable with it. Often the time pressure they feel does not actually exist. Will the person who asked the engineer to do this particular task notice that it took 20 minutes instead of 5? If the person is in the middle of a two-hour meeting, he or she certainly won’t notice. A few extra minutes spent improving the system, however, pays off in all future iterations.

On the contrary, I’ve often warned requesters that their requests may take a little longer because we’re testing some new automation that is buggy but I hope to debug in realtime. I’ve found that whether or not the requesters are technical, they generally get excited and ask to watch. Sometimes the process is in a broken state and I’ve been waiting for the next request as an opportunity to reproduce and fix the bug. In that case, I’ve warned the requester ahead of time that I’ll timebox the debugging process and revert to the old way after a certain amount of time.

If it is tempting to revert to the old way for expediency’s sake, it is useful to remind yourself that the benefit of automation is not always speed. Automation that is slower but less error-prone can be a net gain if the errors take a long time to fix. Preventing a single error that requires a day of restoring data from backups could be invaluable. Because I’m fat-fingered and easily distracted, this is a major motivation for me.

Another benefit is the consistency that automation can bring. Increased variation increases the cost of support

and makes other automation projects more burdensome by increasing the number of edge cases. For example, at one site I discovered that half the Linux systems used raw disk partitions, while the others used Linux LVM (Logical Volume Manager) to manage disk storage. This complicated the monitoring system (which now had to handle both variations), procedure documentation (which had to be written and tested with both variations), and so on. Tasks that should have taken minutes took hours (or days) on the machines that could not benefit from LVM's flexibility. The two variations did not exist for technical reasons. The installation process was not automated, and the manual process resulted in what I'll politely call "creativity," where we would have preferred conformity.

Automation and documentation democratizes the work, lowering the bar so that others may do the task. Any positive progress through the four phases enables more people on a team to do a task, thus enabling you to distribute work among your peers and reduce single points of failure. You might be the only person with the knowledge and experience to do the task, but a little documentation can empower others to do the task instead, even if they don't have a deep understanding of the technology. Even if the documentation covers only the most common situation and is full of warnings such as, "This procedure won't work if the user has [insert technical details]" or "If you get the following error, don't try to fix it yourself. Call Mary or Bob." Future updates to the document can cover those edge cases. You don't need everyone on the team to have your years of experience, just the wisdom to follow directions and contact you if they get stuck.

These benefits save you time in ways other than just making the process faster. They make you more efficient, reduce the work for the entire team by reducing the complexity that must be managed, or create a workforce multiplier that enables other people to take work off your plate.

By creating a culture of continuous improvement, constantly taking baby steps along the four phases, the work becomes less stressful and easier to manage. While the other reasons listed here are quite logical, what motivates me to maintain this discipline is more emotional: I want to reduce stress and have more time for creativity and joy.

THE LEFTOVER PRINCIPLE

Focusing on automating the easy parts means the work left for humans is the difficult stuff. That means automation just made life worse for you.² Ironical, eh? Weren't computers supposed to make life easier? This is called the Leftover Principle, as discussed in this column in 2015.³

The solution to this is the Compensatory Principle: people and machines should each do what they are good at and not attempt what they don't do well. That is, each group should compensate for the other's deficiencies.¹

Therefore, rather than focusing on automating what's easy, focus on automating the boring parts (unlike you, computers love repetition), the difficult parts (reduce error-prone steps), and the parts that need to happen when you would rather be asleep. As a human, you are better than computers at improvisation and being flexible,

exercising judgment, and coping with variations. So, don't fret over not being able to automate deciding which of four paths to take when that decision is purely a judgment call. Instead, automate the four paths but leave the selection process to you!

Documentation as automation lowers the bar for what can be automated, enabling you to improve tasks you would have avoided in both the Leftover Principle and the Compensatory Principle.

AMBIGUOUS REQUIREMENTS

The computer scientists reading this piece might be wondering why I'm not recommending a formal requirements-gathering stage or other more rigorous software-engineering best practices.

The reality is that an organization's IT environment is usually so opaque and amorphous that requirements cannot be written beyond a basic statement of desired results. The first time one attempts to use an API call is more a matter of trial and error than following instructions. Nothing works the first time. It is hours (or days) of guesswork, exploration, and discovery. Nearly every operating system, framework, and IT system contribute to this mess. IT does not live in a world of high-school physics where one has the luxury of an infinitely large, flat, frictionless surface. IT lives in a world that is a squishy swamp of vendor promises and "damned if you do, damned if you don't" choices, all made worse by authentication systems that seem to be designed to work only on sunny days.

Early in the discovery process it is not obvious exactly

what to do, what will work, or how long it will take to figure out how to make something work. It is more exploration than rote execution. It reminds me of a framed sign that hung in my father's chemistry lab that read, "If we knew what we were doing, it wouldn't be called research."

As a result, an incremental and iterative approach is required. Early phases are more exploratory, and later phases are more confident. You start by working on the low-hanging fruit, not because they are easy, but because if you are honest with yourself, you have to admit to having no idea how the more difficult parts could conceivably ever be implemented. By doing the easier parts, however, you gain the experience that makes the other parts possible. Initial experiences inform later decisions, build confidence, and give you the fortitude to continue. Soon the impossible parts of the project become possible.

Therefore, working in a waterfall approach is untenable. Maintaining a lockstep workflow through the phases would mean never leaving the first gate. Some steps may be ready for full automation, while others lag behind. You cannot wait for the documentation to be perfect before moving to the next phase. You may not have figured out a command-line equivalent for step 46, but the other steps can move forward. I once used a system that was pretty darn automated, except someone had to be there to click "ok" at one point, thanks to an installation process from a vendor with a unique definition of what "unattended mode" should mean. It took months to resolve that last issue. I'm glad we didn't wait.

ENABLE EARLY COLLABORATION

An iterative structure improves your ability to work collaboratively. If the documentation is on a wiki or similar system, everyone can contribute and update the documentation. Once the basic infrastructure is in place, everyone can fill in the missing pieces by adding support for new edge cases, improving testing, and so on. Good engineers build the initial framework but make it easy for others to contribute. I call this the “stone soup” method of software development: you bring the cooking pot and everyone else fills it.

The earlier you share, the better. The earlier you can enable this collaboration, the sooner more people can contribute. For example, by keeping the documentation in something easy to edit, such as a wiki or Git repository, everyone on the team can “be the CPU,” not only testing the algorithm, but also contributing improvements. The sooner the software is packaged in a way that everyone can use, the sooner feedback is available. Someone with a developer mindset treats the documentation and code a lot like an open-source project: available and easy to contribute to.

To enable collaboration, use the same tools people are already using. If your team uses Git, keep the documentation in Git. Repurpose the team’s wiki, Google docs structure, CI system, or whatever will lower the bar to contributions.

The anti-pattern is to work privately and plan on releasing the documentation and code to the rest of the team “next week.” Next week never comes. It is a red flag when I hear someone say that “the code isn’t ready to

share with other people” or “I can’t show the document to the team until the next round of edits.” The opposite is true. If you release something that you think “works only for you,” it enables others to figure out how to make it run for them. How can you know what parts work only for you if you haven’t let other people try it?

It is important for managers to create a structure where projects are easily sharable from the start, and to provide (gentle) pressure to move projects into that structure when they aren’t. I try to role-model the release-early attitude by starting my documentation and code in an open Git repository, unabashedly inserting comments such as, “This code sucks and needs to be replaced,” or by indicating which parts are missing or could use improvement. Do not shame people for releasing broken code; reward them for transparency and promoting collaboration.

CONCLUSION

Some IT engineers never have time to automate their work. Others have the same time constraints but succeed in creating the preconditions (documentation, code snippets) that enable automation.

As you work, you have a choice. Will each manual task create artifacts that allow you to accelerate future work, or do you squander these opportunities and accept the status quo?

By constantly documenting and creating code-snippet artifacts, you accelerate future work. That one-shot task that could never happen again, does happen again, and next time it moves faster. Even tasks that aren’t worth

automating can be improved by documenting them, as documentation *is* automation.

Every IT team should have a culture of constant improvement—or movement along the path toward the goal of automating whatever the team feels confident in

automating, in ways that are easy to change as conditions change. As the needle moves to the right, the team learns from each other's experiences, and the system becomes easier to create and safer to operate.

A good team has a structure in place that makes the process frictionless and collaborative—plus, management that rewards and encourages the developer's mindset. Always be automating.

Acknowledgments

This article benefited from feedback from John Allspaw (Adaptive Capacity Labs), Nicole Forsgren (DORA: DevOps Research and Assessment LLC), and Jason Shantz (Stack Overflow, Inc.).

Related articles

➡ Automation Should Be Like Iron Man, Not Ultron

The “Leftover Principle” requires increasingly more highly skilled humans.

Thomas A. Limoncelli

<https://queue.acm.org/detail.cfm?id=2841313>

➡ Swamped by Automation

Whenever someone asks you to trust them, don't.

Kode Vicious

<https://queue.acm.org/detail.cfm?id=2440137>

➡ Automated QA Testing at EA:

Driven by Events

A discussion with Michael Donat, Jafar Husain, and Terry Coatta

<https://queue.acm.org/detail.cfm?id=2627372>

References

1. Allspaw, J. 2013. A mature role for automation, part II. Kitchen Soap; <https://www.kitchensoap.com/2013/08/20/a-mature-role-for-automation-part-ii/>.

2. Bainbridge, L. 1983. Ironies of automation. *Automatica* 19(6): 775-779; <https://pdfs.semanticscholar.org/0713/bb9d9b138e4e0a15406006de9b0cddf68e28.pdf>.
3. Limoncelli, T. A. 2015. Automation should be like Iron Man, not Ultron. *acmqueue* 13(8); <https://queue.acm.org/detail.cfm?id=2841313>.

Thomas A. Limoncelli is the site reliability engineering manager at Stack Overflow Inc. in New York City. His books include *The Practice of System and Network Administration* (<http://the-sysadmin-book.com>), *The Practice of Cloud System Administration* (<http://the-cloud-book.com>), and *Time Management for System Administrators* (<http://shop.oreilly.com/product/9780596007836.do>). He blogs at *EverythingSysadmin.com* and tweets at @YesThatTom. He holds a B.A. in computer science from Drew University.

Copyright © 2018 held by owner/author. Publication rights licensed to ACM.