Automation Should Be Like Iron Man, Not Ultron

THE “LEFTOVER PRINCIPLE” REQUIRES INCREASINGLY MORE HIGHLY-SKILLED HUMANS.

TOM LIMONCELLI

Q: Dear Tom: A few years ago we automated a major process in our system administration team. Now the system is impossible to debug. Nobody remembers the old manual process and the automation is beyond what any of us can understand. We feel like we’ve painted ourselves into a corner. Is all operations automation doomed to be this way?

A: The problem seems to be that this automation was written to be like Ultron, not Iron Man.

Iron Man’s exoskeleton takes the abilities that Tony Stark has and accentuates them. Tony is a smart, strong guy. He can calculate power and trajectory on his own. However, by having his exoskeleton do this for him, he can focus on other things. Of course, if he disagrees or wants to do something the program wasn’t coded to do, he can override the trajectory.

Ultron, on the other hand, was intended to be fully autonomous. It did everything and was, basically, so complex that when it had to be debugged the only choice was [spoiler alert!] to destroy it.

Had the screenwriter/director Joss Whedon consulted me [and Joss, if you are reading this, you really should have], I would have found a way to insert the famous Brian Kernighan quote, “Debugging is twice as hard as writing the

code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.”

THE SOURCE OF THE PROBLEM:

Before we talk about how to prevent this kind of situation, we should discuss how we get into it.

The first way we get into this trap is by automating the easy parts and leaving the rest to be done manually. This sounds like the obvious way to automate things and, in fact, is something I generally encouraged until my awareness was raised by John Allspaw’s excellent two-part blog post “A Mature Role for Automation” (<http://www.kitchensoap.com/2012/09/21/a-mature-role-for-automation-part-i>).

You certainly shouldn’t automate the difficult cases first. What we learn while automating the easy cases makes us better prepared to automate the more difficult cases. This is called the Leftover Principle. You automate the easy parts and what is “left over” is done by humans.

In the long run this creates a very serious problem. The work left over for people to do becomes, by definition, more difficult. At the start of the process, people were doing a mixture of simple and complex tasks. After a while the mix shifts more and more towards the complex. This is a problem because people aren’t getting smarter over time. Moore’s Law predicts that computers will get more powerful over time, but sadly there is no such prediction about people.

Another reason the work becomes more difficult is

that it becomes rarer. Easier work, done frequently, keeps a person's skills fresh and keeps us ready for the rare but difficult tasks.

Taken to its logical conclusion, this paradigm results in a need to employ impossibly smart people to do impossibly difficult work. Maybe this is why Google's recruiters sound so painfully desperate when they call about joining their SRE team.

One way to avoid the problems of the leftover principle is called the Compensatory Principle. There are certain tasks that people are good at that machines don't do well. Likewise there are other tasks that machines are good at that people don't do well. The compensatory principle says that 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.

Machines don't get bored, so they are better at repetitive tasks. They don't sleep, so they are better at tasks that must be done at all hours of the night. They are better at handling many operations at once, and at operations that require smooth or precise motion. They are better at literal reproduction, access restriction, and quantitative assessment.

People are better at improvisation and being flexible, exercising judgment, and coping with variations in written material, perceiving feelings.

Let's apply this principle to a monitoring system. The monitoring system collects metrics every five minutes, stores them, and then analyzes the data for the purposes of

alerting, debugging, visualization, and interpretation.

A person could collect data about a system every five minutes, and with multiple shifts of workers they could do it around the clock. However, the people would become bored and sloppy. Therefore it is obvious that the data collection should be automated. Alerting requires precision, which is also best done by computers. However, while the computer is better at visualizing the data, people are better at interpreting those visualizations. Debugging requires improvisation, another human skill, so again people are assigned those tasks.

John Allspaw points out that only rarely can a project be broken down into such clear-cut cases of functionality this way.

DOING BETTER

A better way is to base automation decisions on the complementarity principle. This principle looks at automation from the human perspective. It improves the long-term results by considering how people's behavior will change as a result of automation.

For example, the people planning the automation should consider what is learned over time by doing the process manually and how that would be changed or reduced if the process was automated. When a person first learns a task, they are focused on the basic functions needed to achieve the goal. However, over time, they understand the ecosystem that surrounds the process and gain a big-picture

view. This lets them perform global optimizations. When a process is automated the automation encapsulates learning thus far, permitting new people to perform the task without having to experience that learning. This stunts or prevents future learning. This kind of analysis is part of a cognitive systems engineering [CSE] approach.

The complementarity principle combines CSE with a joint cognitive system [JCS] approach. JCS examines how automation and people work together. A joint cognitive system is characterized by its ability to stay in control of a situation.

In other words, if you look at a highly automated system and think, “Isn’t it beautiful? We have no idea how it works,” you may be using the leftover principle. If you look at it and say, “Isn’t it beautiful how we learn and grow together, sharing control over the system,” then you’ve done a good job of applying the complementarity principle.

Designing automation using the complementarity principle is a relatively new concept and I admit I’m no expert, though I can look back at past projects and see where success has come from applying this principle by accident. Even the blind squirrel finds some acorns!

For example, I used to be on a team that maintained a very large (for its day) cloud infrastructure. We were responsible for the hundreds of physical machines that supported thousands of virtual machines.

We needed to automate the process of repairing the physical machines. When there was a hardware problem, virtual machines had to be moved off the physical machine,

the machine had to be diagnosed, and a request for repairs had to be sent to the hardware techs in the data center. After the machine was fixed, it needed to be re-integrated into the cloud.

The automation we created abided by the complementarity principle. It was a partnership between human and machine. It did not limit our ability to learn and grow. The control over the system was shared between the automation and the humans involved.

In other words, rather than creating a system that took over the cluster and ran it, we created one that partnered with humans to take care of most of the work. It did its job autonomously, but we did not step on each other's toes.

The automation had two parts. The first part was a set of tools that the team used to do the various related tasks. Only after these tools had been working for some time did we build a system that automated the global process, and it did so more like an exoskeleton assistant than like a dictator.

The repair process was functionally decomposed into five major tasks, and one tool was written to handle each of them. The tools were (a) Evacuation: any virtual machines running on the physical machine needed to be migrated live to a different machine; (b) Revivification: an evacuation process required during the extreme case where a virtual machine had to be restarted from its last snapshot; (c) Recovery: attempts to get the machine working again by simple means such as powering it off and on again; (d) Send to Repair Depot: generate a work order describing what needs to be fixed and send this information to the data

center technicians who actually fixed the machine; (e) Re-assimilate: once the machine has been repaired, configure it and re-introduce it to the service.

As the tools were completed, they replaced their respective manual processes. However the tools provided extensive visibility as to what they were doing and why.

The next step was to build automation that could bring all these tools together. The automation was designed based on a few specific principles:

- It should follow the same methodology as the human team members.
- It should use the same tools as the human team members.
- If another team member was doing administrative work on a machine or cluster (group of machines), the automation would step out of the way if asked, just like a human team member would.
- Like a good team member, if it got confused it would back off and ask other members of the team for help.

The automation was a state-machine-driven repair system. Each physical machine was in a particular state: normal, in trouble, recovery in progress, sent for repairs, being re-assimilated, and so on. The monitoring system that would normally page people when there was a problem instead alerted our automation. Based on whether the alerting system had news of a machine having problems, being dead, or returning to life, the appropriate tool was activated. The tool's result determined the new state assigned to the machine.

If the automation got confused, it paused its work on that

machine and asked a human for help by opening a ticket in our request tracking system.

If a human team member was doing manual maintenance on a machine, the automation was told to not touch the machine in an analogous way to how human team members would be, except people could now type a command instead of shouting to their coworkers in the surrounding cubicles.

The automation was very successful. Previously whoever was on call was paged once or twice a day. Now we were typically paged less than once a week.

Because of the design, the human team members continued to be involved in the system enough that they were always learning. Some people focused on making the tools better. Others focused on improving the software release and test process.

As stated earlier, one problem with the leftover principle is that the work left over for humans requires increasingly higher skill levels. At times we experienced the opposite! As the number of leftover tasks was reduced, it was easier to wrap our brains around the ones that remained. Without the mental clutter of so many other tasks, we were better able to assess the remaining tasks. For example, the most highly technical task involved a particularly heroic recovery procedure. We re-evaluated whether or not we should even be doing this particular procedure. We shouldn't.

The heroic approach risked data loss in an effort to avoid rebooting a virtual machine. This was the wrong priority. Our customers cared much more about data loss than about a quick reboot. We actually eliminated this leftover task by

replacing it with an existing procedure that was already automated. We would not have seen this opportunity if our minds had still been cluttered with so many other tasks.

Another leftover process was building new clusters or machines. It happened infrequently enough that it was not worthwhile to fully automate. However, we found we could Tom Sawyer the automation into building the cluster for us if we created the right metadata to make it think that all the machines had just returned from repairs. Soon the cluster was built for us.

Processes requiring ad hoc improvisation, creativity, and evaluation were left to people. For example, certifying new models of hardware required improvisation and the ability to act given vague requirements.

The resulting system felt a lot like Iron Man's suit: enhancing our skills and taking care of the minutiae so we could focus on the big picture. One person could do the work of many, and we could do our jobs better thanks to the fact that we had an assistant taking care of the busy work. Learning did not stop because it was a collaborative effort. The automation took care of the boring stuff and the late-night work, and we could focus on the creative work of optimizing and enhancing the system for our customers.

I don't have a formula that will always achieve the benefits of the complementarity principle. However, by paying careful attention to how people's behavior will change as a result of automation and by maintaining shared control over the system, we can build automation that is more Iron Man, less Ultron.

FURTHER READING

John Allspaw's article "A Mature Role for Automation."
[\[http://www.kitchensoap.com/2012/09/21/a-mature-role-for-automation-part-i\]](http://www.kitchensoap.com/2012/09/21/a-mature-role-for-automation-part-i).

David Woods and Erik Hollnagel's book *Joint Cognitive Systems: Foundations of Cognitive Systems Engineering*.
 Taylor and Francis, Boca Raton, FL, 2005.

Chapter 12 of *The Practice of Cloud System Administration*,
 by Thomas A. Limoncelli, Strata R. Chalup, and Christina J.
 Hogan.[\[http://the-cloud-book.com\]](http://the-cloud-book.com)

LOVE IT, HATE IT? LET US KNOW feedback@queue.acm.org

Thomas A. Limoncelli *is an author, speaker, and system administrator. He is an SRE at Stack Exchange, Inc in NYC. His books include The Practice of Cloud Administration [the-cloud-book.com] and Time Management for System Administrators. He blogs at EverythingSysadmin.com*