Case Study 1: Reducing Toil in the Datacenter with Automation

Note

Toil reduction strategies highlighted in Case Study 1:

* Engineer toil out of the system
* Start small and then improve
* Increase uniformity
* Use SLOs to reduce toil
* Assess risk within automation
* Use feedback to improve
* Automate toil response

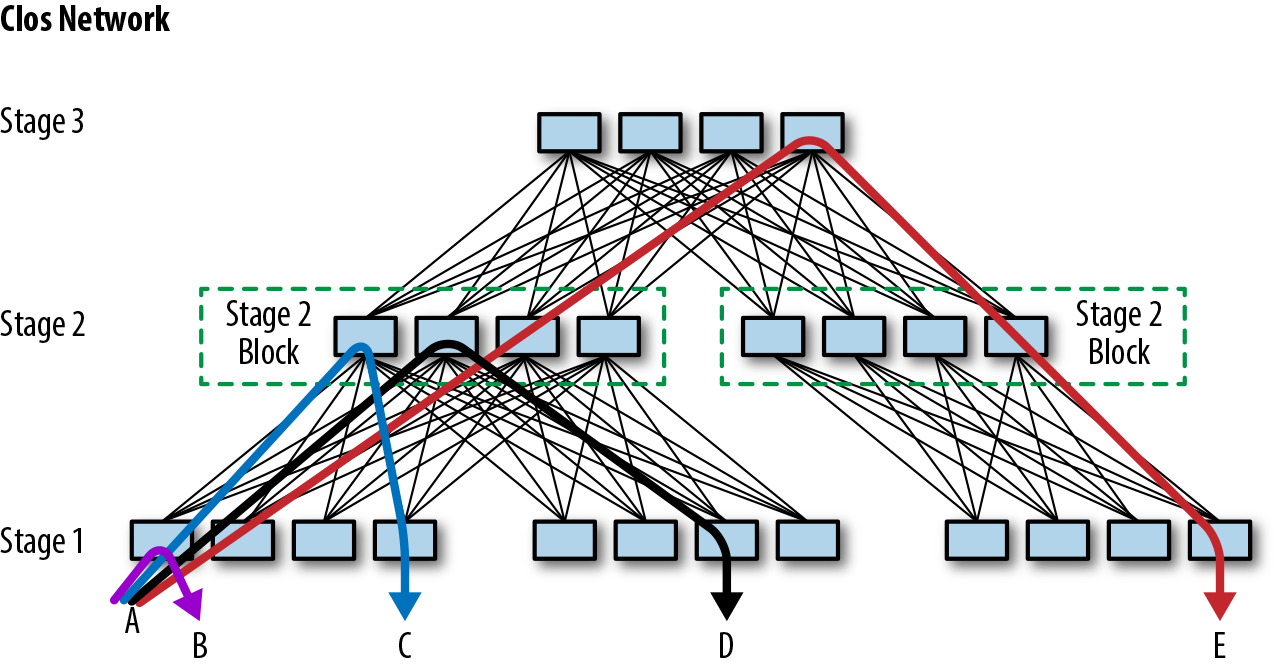
Background

This case study takes place in Google’s datacenters. Similar to all datacenters, Google’s machines are connected to switches, which are connected to routers. Traffic flows in and out from these routers via links that in turn connect to other routers on the internet. As Google’s requirements for handling internet traffic grew, the number of machines required to serve that traffic increased dramatically. Our datacenters grew in scope and complexity as we figured out how to serve a large amount of traffic efficiently and economically. This growth changed the nature of datacenter manual repairs from occasional and interesting to frequent and route—two signals of toil.

When Google first began running its own datacenters, each datacenter’s network topology featured a small number of network devices that managed traffic to a large number of machines. A single network device failure could significantly impact network performance, but a relatively small team of engineers could handle troubleshooting the small number of devices. At this early stage, engineers debugged problems and shifted traffic away from failed components manually.

Our next-generation datacenter had significantly more machines and introduced [software-defined networking (SDN) with a folded Clos topology](https://sreworkbook.page.link/B4tU) which greatly increased the number of switches. [Figure 6-2](https://sre.google/workbook/eliminating-toil/#480-machines-attached-below-Stage) shows the complexity of traffic flow for a small datacenter Clos switch network. This proportionately larger number of devices meant that a larger number of components could now fail. While each individual failure had less impact on network performance than before, the sheer volume of issues began to overwhelm the engineering staff.

In addition to introducing a heavy load of new problems to debug, the complex layout was confusing to technicians: Which exact links needed to be checked? Which line card[6](https://sre.google/workbook/eliminating-toil/#ch06fn6) did they need to replace? What was a Stage 2 switch, versus a Stage 1 or Stage 3 switch? Would shutting down a switch create problems for users?

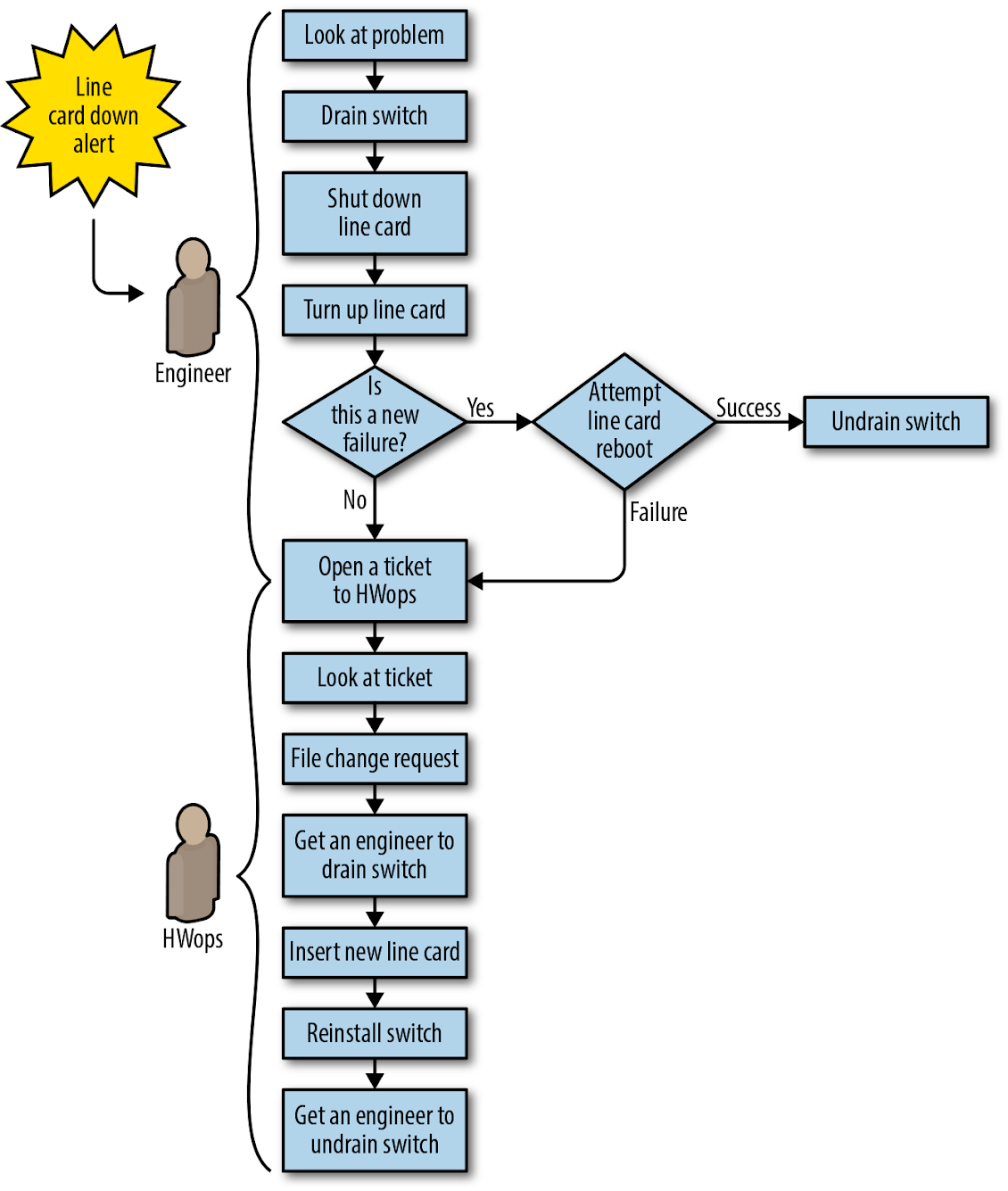
Figure 6-2. A small Clos network, which supports 480 machines attached below Stage 1

Repairing failed datacenter line cards was one obvious growing work backlog, so we targeted this task as our first stage of creating datacenter network repair automation. This case study describes how we introduced repair automation for our first generation of line cards (named Saturn). We then discuss the improvements we introduced with the next generation of line cards for Jupiter fabrics.

As shown in [Figure 6-3](https://sre.google/workbook/eliminating-toil/#Datacenter-Saturn-line-card-repair), before the automation project, each fix in the datacenter line-card repair workflow required an engineer to do the following:

1. Check that it was safe to move traffic from the affected switch.
2. Shift traffic away from the failed device (a “drain” operation).
3. Perform a reboot or repair (such as replacing a line card).
4. Shift traffic back to the device (an “undrain” operation).

This unvarying and repetitive work of draining, undraining, and repairing devices is a textbook example of toil. The repetitive nature of the work introduced problems of its own—for example, engineers might multitask by working on a line card while also debugging more challenging problems. As a result, the distracted engineer might accidentally introduce an unconfigured switch back to the network.

Figure 6-3. Datacenter (Saturn) line-card repair workflow before automation: all steps require manual work

Problem Statement

The datacenter repairs problem space had the following dimensions:

* We couldn’t grow the team fast enough to keep up with the volume of failures, and we couldn’t fix problems fast enough to prevent negative impact to the fabric.
* Performing the same steps repeatedly and frequently introduced too many human errors.
* Not all line-card failures had the same impact. We didn’t have a way to prioritize more serious failures.
* Some failures were transient. We wanted the option to restart the line card or reinstall the switch as a first pass at repair. Ideally, we could then programmatically capture the problem if it happened again and flag the device for replacement.
* The new topology required us to manually assess the risk of isolating capacity before we could take action. Every manual risk assessment was an opportunity for human error that could result in an outage. Engineers and technicians on the floor didn’t have a good way to gauge how many devices and links would be impacted by their planned repair.

What We Decided to Do

Instead of assigning every issue to an engineer for risk assessment, drain, undrain, and validation, we decided to create a framework for automation that, when coupled with an on-site technician where appropriate, could support these operations programmatically.

Design First Effort: Saturn Line-Card Repair

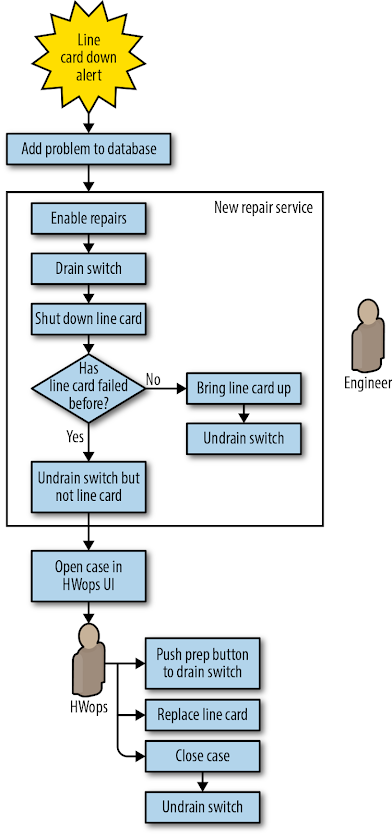
Our high-level goal was to build a system that would respond to problems detected on network devices, rather than relying on an engineer to triage and fix these problems. Instead of sending a “line card down” alert to an engineer, we wrote the software to request a drain (to remove traffic) and create a case for a technician. The new system had a few notable features:

* We leveraged existing tools where possible. As shown in [Figure 6-3](https://sre.google/workbook/eliminating-toil/#Datacenter-Saturn-line-card-repair), our alerting could already detect problems on the fabric line cards; we repurposed that alerting to trigger an automated repair. The new workflow also repurposed our ticketing system to support network repairs.
* We built in automated risk assessment to prevent accidental isolation of devices during a drain and to trigger safety mechanisms where required. This eliminated a huge source of human errors.
* We adopted a strike policy that was tracked by software: the first failure (or strike) only rebooted the card and reinstalled the software. A second failure triggered card replacement and full return to the vendor.

Implementation

The new automated workflow (shown in [Figure 6-4](https://sre.google/workbook/eliminating-toil/#Saturn-line-card-repair-workflow-with-automation)) proceeded as follows:

1. The problematic line card is detected and a symptom is added to a specific component in the database.
2. The repair service picks up the problem and enables repairs on the switch. The service performs a risk assessment to confirm that no capacity will be isolated by the operation, and then:
   1. Drains traffic from the entire switch.
   2. Shuts down the line card.
   3. If this is a first failure, reboots the card and undrains the switch, restoring service to the switch. At this point, the workflow is complete.
   4. If this is the second failure, the workflow proceeds to step 3.
3. The workflow manager detects the new case and sends it to a pool of repair cases for a technician to claim.
4. The technician claims the case, sees a red “stop” in the UI (indicating that the switch needs to be drained before repairs are started), and executes the repair in three steps:
   1. Initiates the chassis drain via a “Prep component” button in the technician UI.
   2. Waits for the red “stop” to clear, indicating that the drain is complete and the case is actionable.
   3. Replaces the card and closes the case.
5. The automated repair system brings the line card up again. After a pause to give the card time to initialize, the workflow manager triggers an operation to restore traffic to the switch and close the repair case.

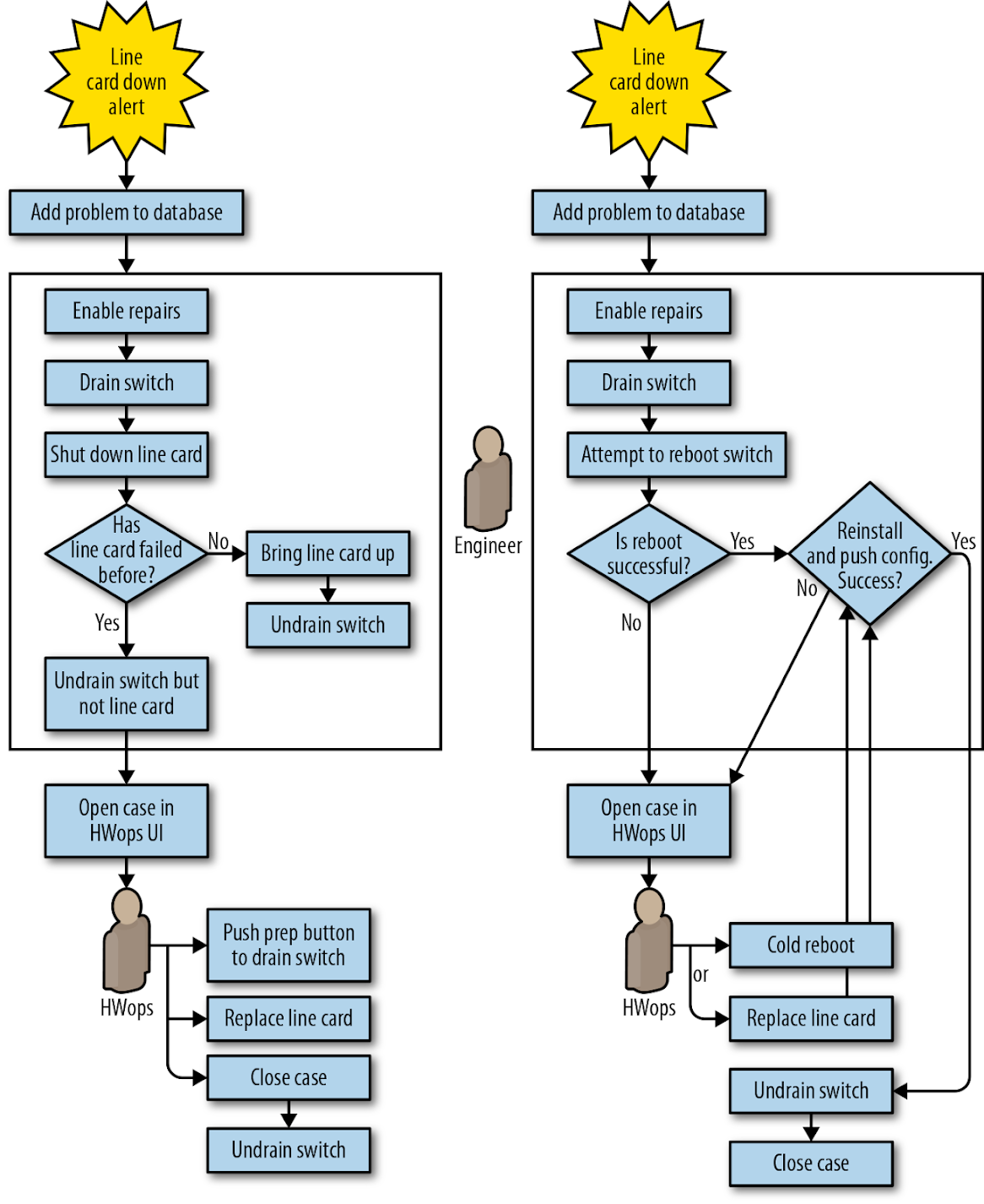
Figure 6-4. Saturn line-card repair workflow with automation: manual work required only to push a button and replace the line card

The new system freed the engineering team from a large volume of toilsome work, giving them more time to pursue productive projects elsewhere: working on Jupiter, the next-generation Clos topology.

Design Second Effort: Saturn Line-Card Repair Versus Jupiter Line-Card Repair

Capacity requirements in the datacenter continued to double almost every 12 months. As a result, our next-generation datacenter fabric, Jupiter, was more than six times larger than any previous Google fabric. The volume of problems was also six times larger. Jupiter presented scaling challenges for repair automation because thousands of fiber links and hundreds of line cards could fail in each layer. Fortunately, the increase in potential failure points was accompanied by far greater redundancy, which meant we could implement more ambitious automation. As shown in [Figure 6-5](https://sre.google/workbook/eliminating-toil/#Saturn-line-card-down-automation) we preserved some of the general workflow from Saturn and added a few important modifications:

* After an automated drain/reboot cycle determined that we wanted to replace hardware, we sent the hardware to a technician. However, instead of requiring a technician to initiate the drain with the “Push prep button to drain switch,” we automatically drained the entire switch when it failed.
* We added automation for installing and pushing the configuration that engages after component replacement.
* We enabled automation for verifying that the repair was successful before undraining the switch.
* We focused attention on recovering the switch without involving a technician unless absolutely necessary.

Figure 6-5. Saturn line-card down automation (left) versus Jupiter automation (right)

Implementation

We adopted a simple and uniform workflow for every line-card problem on Jupiter switches: declare the switch down, drain it, and begin a repair.

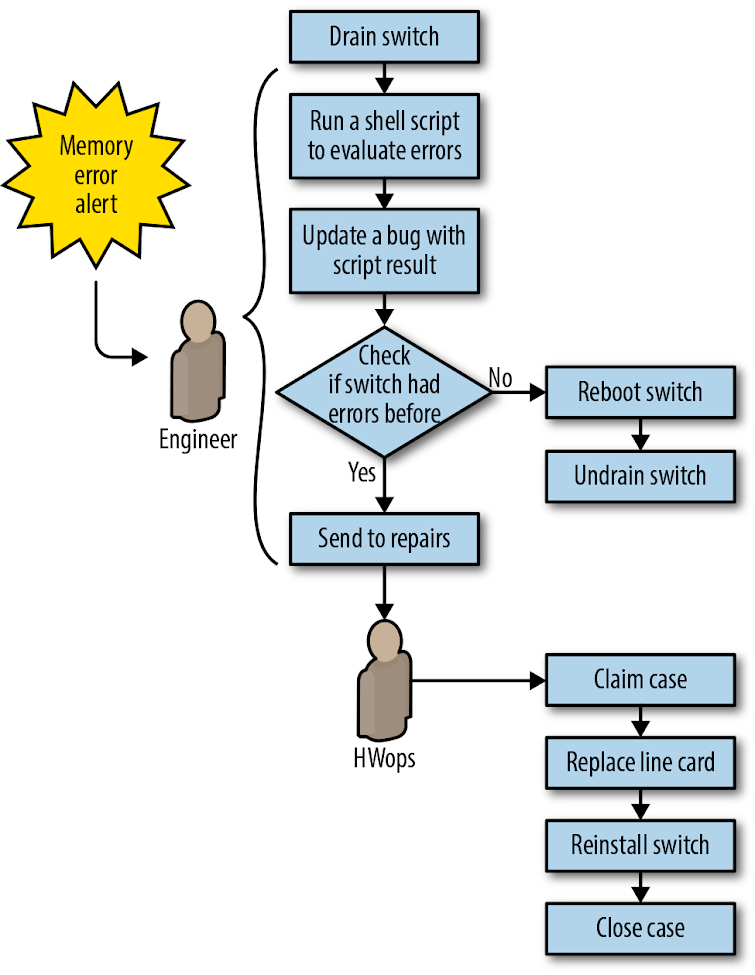
The automation carried out the following:

1. The problem switch-down is detected and a symptom is added to the database.
2. The repair service picks up the problem and enables repairs on the switch: drain the entire switch, and add a drain reason.
   1. If this is the second failure within six months, proceed to step 4.
   2. Otherwise, proceed to step 3.
3. Attempt (via two distinct methods) to power-cycle the switch.
   1. If the power-cycle is successful, run automated verification, then install and configure the switch. Remove the repair reason, clear the problem from the database, and undrain the switch.
   2. If preceding sanity-checking operations fail, send the case to a technician with an instruction message.
4. If this was the second failure, send the case directly to the technician, requesting new hardware. After the hardware change occurs, run automated verification and then install and configure the switch. Remove the repair reason, clear the problem from the database, and undrain the switch.

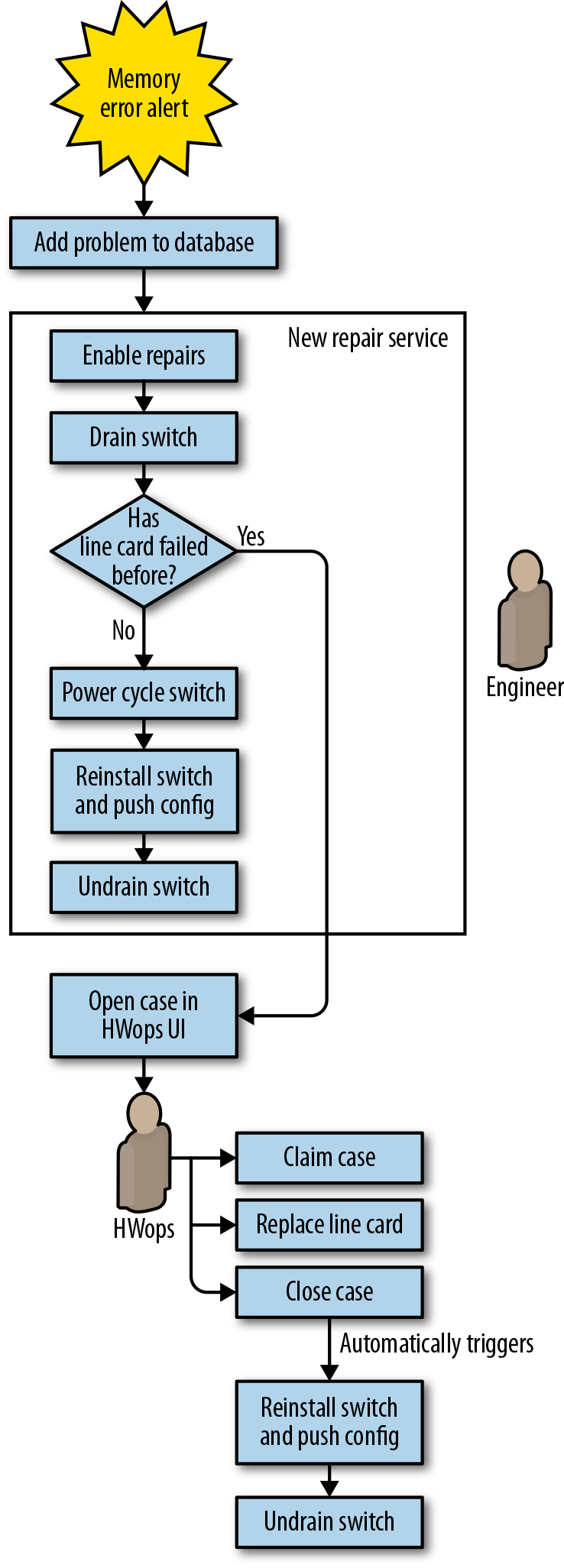
This new workflow management was a complete rewrite of the previous repair system. Again, we leveraged existing tools when possible:

* The operations for configuring new switches (install and verify) were the same operations we needed to verify that a switch that had been replaced.
* Deploying new fabrics quickly required the ability to BERT[7](https://sre.google/workbook/eliminating-toil/#ch06fn7) and cable-audit[8](https://sre.google/workbook/eliminating-toil/#ch06fn8) programmatically. Before restoring traffic, we reused that capability to automatically run test patterns on links that had fallen into repairs. These tests further improved performance by identifying faulty links.

The next logical improvement was to automate mitigation and repair of memory errors on Jupiter switch line cards. As shown in [Figure 6-6](https://sre.google/workbook/eliminating-toil/#Jupiter-memory-error-repair-workflow), prior to automation, this workflow depended heavily on an engineer to determine if the failure was hardware- or software-related, and then to drain and reboot the switch or arrange a repair if appropriate.

Figure 6-6. Jupiter memory error repair workflow before automation

Our automation simplified the repair workflow by no longer attempting to troubleshoot memory errors (see [Sometimes imperfect automation is good enough](https://sre.google/workbook/eliminating-toil/#Sometimes-imperfect-automation-is-good-enough) for why this made sense). Instead, we treated memory errors the same way we handled failed line cards. To extend automation to memory errors, we simply had to add another symptom to a config file to make it act on the new problem type. [Figure 6-7](https://sre.google/workbook/eliminating-toil/#Jupiter-memory-error-repair-workflow-with-automation) depicts the automated workflow for memory errors.

Figure 6-7. Jupiter memory error repair workflow with automation

Lessons Learned

During the several years we worked to automate network repair, we learned a lot of general lessons about how to effectively reduce toil.

UIs should not introduce overhead or complexity

For Saturn-based line cards, replacing a line card required draining the entire switch. Draining the entire switch early in the repair process meant losing the working capacity of all line cards on the switch while waiting for replacement parts and a technician. We introduced a button in the UI called “Prep component” that allowed a technician to drain traffic from the entire switch right before they were ready to replace the card, thereby eliminating unnecessary downtime for the rest of the switch (see “Push prep button to drain switch” in [Figure 6-5](https://sre.google/workbook/eliminating-toil/#Saturn-line-card-down-automation)).

This aspect of the UI and repair workflow introduced a number of unexpected problems:

* After pressing the button, the technician did not get feedback on drain progress but instead simply had to wait for permission to proceed.
* The button didn’t reliably sync with the actual state of the switch. As a result, sometimes a drained switch did not get repaired, or a technician interrupted traffic by acting upon an undrained switch.
* Components that did not have automation enabled returned a generic “contact engineering” message when a problem arose. Newer technicians did not know the best way to reach someone who could help. Engineers who were contacted were not always immediately available.

In response to user reports and problems with regressions caused by the complexity of the feature, we designed future workflows to ensure the switch was safe and ready for repair before the technician arrived at the switch.

Don’t rely on human expertise

We leaned too heavily on experienced datacenter technicians to identify errors in our system (for example, when the software indicated it was safe to proceed with repairs, but the switch was actually undrained). These technicians also had to perform several tasks manually, without being prompted by automation.

Experience is difficult to replicate. In one particularly high-impact episode, a technician decided to expedite the “press button and wait for results” experience by initiating concurrent drains on every line card waiting for repairs at the datacenter, resulting in network congestion and user-visible packet loss. Our software didn’t anticipate and prevent this action because we didn’t test the automation with new technicians.

Design reusable components

Where possible, avoid monolithic designs. Build complex automation workflows from separable components, each of which handles a distinct and well-defined task. We could easily reuse or adapt key components of our early Jupiter automation for each successive generation of fabric, and it was easier to add new features when we could build on automation that already existed. Successive variations on Jupiter-type fabrics could leverage work done in earlier iterations.

Don’t overthink the problem

We overanalyzed the memory error problem for Jupiter line cards. In our attempts at precise diagnosis, we sought to distinguish software errors (fixable by reboots) from hardware errors (which required card replacement), and also to identify errors that impacted traffic versus errors that did not. We spent nearly three years (2012–2015) collecting data on over 650 discrete memory error problems before realizing this exercise was probably overkill, or at least shouldn’t block our repair automation project.

Once we decided to act upon any error we detected, it was straightforward to use our existing repair automation to implement a simple policy of draining, rebooting, and reinstalling switches in response to memory errors. If the problem recurred, we concluded that the failure was likely hardware-based and requested component replacement. We gathered data over the course of a quarter and discovered that most of the errors were transient—most switches recovered after being rebooted and reinstalled. We didn’t need additional data to perform the repair, so the three-year delay in implementing the automation was unnecessary.

Sometimes imperfect automation is good enough

While the ability to verify links with BERT before undraining them was handy, BERT tooling didn’t support network management links. We added these links into the existing link repair automation with a check that allowed them to skip verification. We were comfortable bypassing verification because the links didn’t carry customer traffic, and we could add this functionality later if verification turned out to be important.

Repair automation is not fire and forget

Automation can have a very long lifetime, so make sure to plan for project continuity as people leave and join the team. New engineers should be trained on legacy systems so they can fix bugs. Due to parts shortages for Jupiter fabrics, Saturn-based fabrics lived on long after the originally targeted end-of-life date, requiring us to introduce some improvements quite late in Saturn’s overall lifespan.

Once adopted, automation may become entrenched for a long time, with positive and negative consequences. When possible, design your automation to evolve in a flexible way. Relying on inflexible automation makes systems brittle to change. Policy-based automation can help by clearly separating intent from a generic implementation engine, allowing automation to evolve more transparently.

Build in risk assessment and defense in depth

After building new tools for Jupiter that determined the risk of a drain operation before executing it, the complexity we encountered led us to introduce a secondary check for defense in depth. The second check established an upper limit for the number of impacted links, and another limit for impacted devices. If we exceeded either threshold, a tracking bug to request further investigation opened automatically. We tuned these limits over time to reduce false positives. While we originally considered this a temporary measure until the primary risk assessment stabilized, the secondary check has proven useful for identifying atypical repair rates due to power outages and software bugs (for one example, see [“Automation: Enabling Failure at Scale”](https://sre.google/sre-book/automation-at-google/) in *Site Reliability Engineering*).

Get a failure budget and manager support

Repair automation can sometimes fail, especially when first introduced. Management support is crucial in preserving the project and empowering the team to persevere. We recommend establishing an error budget for antitoil automation. You should also explain to external stakeholders that automation is essential despite the risk of failures, and that it enables continuous improvement in reliability and efficiency.

Think holistically

Ultimately, the complexity of scenarios to be automated is the real hurdle to overcome. Reexamine the system before you work on automating it—can you simplify the system or workflow first?

Pay attention to all aspects of the workflow you are automating, not just the aspects that create toil for you personally. Conduct testing with the people directly involved in the work and actively seek their feedback and assistance. If they make mistakes, find out how your UI could be clearer, or what additional safety checks you need. Make sure your automation doesn’t create new toil—for example, by opening unnecessary tickets that need human attention. Creating problems for other teams will increase resistance to future automation endeavors.