4. Designing for Failure, Operating to Learn [Book]

Digital infusion confronts post-industrial businesses with ever more complex environments. Traditional boundaries, such as those between back-office and front-office functions,[1] break down. Even the idea that IT is separate from the rest of the business, and from the means of customer engagement, comes under scrutiny. Brand management becomes exponentially harder when it has to reflect the operation of an entire service organization.

The continuously self-designing organization represents the triumph of complexity over complicatedness. Its autopoietic process is the ultimate expression of emergent structure. The organization and its components continually create, adapt, and rearrange one another. The shape of its structure becomes a dynamic process that unfolds over time. The history of that process reflects the history of the organization's interactions with its customers and the market.

The flexibility that arises from continuous design dooms attempts at complicated-systems control. The digital conversational medium allows organizations to conduct intimate conversations with their customers. In the process, however, it forces digital businesses to reimagine their understanding of success and failure.

Industrial-era control mechanisms strive to identify and eradicate opportunities for failure. The ideal production workflow is error-free. It turns out products that are free from defects and identical to one another. In addition to banishing error, this approach also banishes innovation and adaptation. Particularly in complex environments where component-level failure is unavoidable, it results in brittleness instead of robustness.

Cybernetic control mechanisms, on the other hand, incorporate failure within themselves. They make the calculation and correction of error part of steady-state operations. They provide a basis for rethinking the meaning of failure in a way that can accommodate the sloppiness of complex systems. This capability for accommodation is critical to leveraging the resilience that accompanies that sloppiness.

The cybernetic model of control that underlies the digital conversational medium lets us trade the precise robustness of complicated systems for the sloppy resilience of complex systems. In the process, it defines the relationship between success and failure in less dualistic terms. From a second-order cybernetic perspective, one might rephrase "calculation and correction of error" as "recognition of and response to difference." The ability to adopt a more unified understanding of success and failure is indispensable for successful post-industrial management.

Complexity confounds our traditional understanding of the meaning of success and failure. Instead of a binary decision at a point in time, it becomes a dynamic, evolving

process that can be evaluated only in hindsight. We generally equate *success* with *correctness*. Post-industrialism brings that equivalency into question.

These days failure is all the rage. Blog posts and tweets announce the good news: failure is valuable, failure is necessary, failure should be encouraged, Google developed all of its best services out of failures, and so on. The idea that failure is good, though, seems to imply that somehow it leads to success. If that's the case, is it really failure anymore? What does it really mean to say that "failure is good"?

By itself, failure is anything but good. Making the same mistake over and over again doesn't help anyone. Failure only leads to success when we learn from it by changing our behavior in response to it. Even then, it's impossible to guarantee the accuracy of any given response. Its validity can only be evaluated in hindsight. The environment to which you're trying to adapt doesn't stop changing just because you've declared victory. Yesterday's success can turn into tomorrow's failure.

We need a new definition of failure that shifts our focus from momentary events to unfolding processes. This shift is especially important in the context of complex systems that evade traditional control. Component-level failure is inevitable in complex systems, yet the systems themselves can still thrive and even improve. Conversely, component-level events can combine to cause systemic breakdowns without themselves being considered failures. Once again, failure is in the eyes of the future beholder.

We seem to be stuck in a catch-22. On one hand, we can't be sure our actions won't make things worse. On the other hand, inaction isn't an option; the situation arose in the first place because our current state is unsatisfactory. To resolve this conundrum, we need a less dualistic, binary approach to success and failure. Complexity forces us to redefine success as "a useful conversation with one's environment."

Post-industrial IT provides the conversational medium by which self-steering organizations navigate uncertain, unknowable, continuously changing environments. Put another way, it helps them have useful conversations with those environments. Because complex systems are resistant to industrial management techniques, twenty-first-century businesses need a new strategy. Instead of trying to tame their environments by engineering out the possibility for breakdowns, they need to develop the ability to continuously repair them.

A conversation is "an interchange of information." A useful conversation exchanges information in a way that continually makes sense and offers value to all involved parties. Imagine the following counterproductive exchange:

> Speaker 1: Let's have Indian food for dinner.

> Speaker 2: I don't like Indian food.
>
> Speaker 1: Do you prefer the Indian place on Grand or the one on Summit?

Speaker 1 isn't really listening to his counterpart. He's not leading the conversation in a direction that has anything to offer to Speaker 2. In fact, Speaker 1 is wandering off into the weeds by way of non sequitur.

By this definition, success is less about what you do at any given point in time than how you process the environment's response to it. The following conversation is perfectly constructive:

> Speaker 1: Let's have Indian food for dinner.
>
> Speaker 2: I don't like Indian food.
>
> Speaker 1: Do you like Mexican food?
>
> Speaker 2: I love it!
>
> Speaker 1: I know a great Mexican place on Grand.
>
> Speaker 2: That sounds good. But wait, won't Grand be congested tonight because of the game?
>
> Speaker 1: Good point. There's another good place on Summit.
>
> Speaker 2: Let's go there.

By redefining success in this way, we've enabled ourselves to evaluate our progress in real time. As long as we're (a) speaking, that is, acting by trying something new, (b) listening to the environment's response to our action, and (c) guiding our future action by the response to our past action, then we are succeeding. When we stop engaging in any of those three steps, we have failed.

Whether it be a vendor and its customer, a finance department and a project management department, or an authorization microservice and a login microservice, conversation always takes place between agents that differ from one another. Regardless of the level of empathy, perfect mutual understanding is never feasible. As one can see even from the simplistic example of the conversation about where to have dinner, breakdowns in understanding are inevitable and continual. Success through conversation is a matter of continual repair.

By its very cybernetic nature, conversational IT enables useful conversation through continual repair. Feedback loops continually correct conversational misunderstanding. In fact, one could think of self-steering as another word for a "useful conversation with

one's environment." If self-steering doesn't generate a meaningful conversation, it doesn't help the system stay alive.

The Agile practice of sprint demos perfectly illustrates this mechanism. The sprint demo's ostensible purpose is to *demonstrate* the development team's progress to customers or their proxy. Its real purpose, however, is to get feedback about errors in understanding. It gives customers an opportunity to say "That's not really how we wanted it to work," or "That's not really what we need."

Complex systems are enigmatic and rife with failure. Their properties of emergence, resilience in the face of component-level failure, cascading failures, and sensitivity to history make them continually surprise us. Their resistance to top-down management makes conversational *control* the only feasible method. Continuous repair requires the willingness to listen and to be led; in Glanville's words, "to control by being controlled by the controlled."

The futility of trying to control complex systems through traditional means is not without its benefits. People tout the value of failure because it contributes to learning. Instead of trying to keep complex systems from failing, we can use their failures as learning opportunities. The willingness to listen to and be led by failure means that we can build and operate systems and organizations that improve over time, rather than just not degrading.

To use continuous design as a strategy for steering through complexity, we need to augment it with an additional principle: *design for failure; operate to learn*.

Cutting-edge organizations are espousing this principle through a variety of practices, including:

- MTTR over MTBF
- Design-for-fail
- Game days
- *Chaos monkeys*
- Blameless postmortems
- Operational transparency

These practices typically appear inside of IT. They don't, however, only apply to problems of technical resilience. The challenges of complexity confront digitally infused service as a whole. The entire service organization needs to design for failure and operate to learn. After describing specific practices for steering through

complexity in the context of IT, this chapter will examine them from an overall service business perspective.

Traditionally, IT exerts tremendous energy trying to maximize mean time between failures (MTBF). It strives to make underlying systems (hardware, network, databases, etc.) as robust as possible. This strategy allows higher-level systems and applications to take a naïve approach to failure. They only need to concern themselves with it at their own level. They can assume lower-level failures either won't happen or will be contained and hidden.

Maximizing MTBF doesn't work in complex environments with large numbers of independent, rapidly changing agents. At some point, failure becomes inevitable by simple fact of arithmetic. When you have thousands of systems and are continuously deploying many small changes, trying to maximize MTBF no longer suffices. As Adrian Cockcroft, former Chief Cloud Architect at Netflix, says, "speed at scale breaks everything."

One might think that Netflix's scalability challenges don't apply to normal enterprises. Netflix does, after all, account for 33% of all Internet traffic on a typical Friday night. A large enterprise, though, operates hundreds if not thousands of applications. Those applications run on thousands of production server instances and process millions of transactions against many terabytes of data. By decomposing applications into more fine-grained modules, service-oriented architectures multiply the number of software objects needing management. Netflix's lessons are thus more relevant to most enterprises than one might realize.

The alternative to struggling to maximize MTBF is to shift your strategy toward minimizing mean time to repair (MTTR). If it's sufficiently quick, easy, and safe to repair faults, you become less afraid of them. Continuous Delivery, fearless releases, infrastructure as code, and microservices architectures are all components of an MTTR-centric fault-management strategy.

Continuous Delivery minimizes the latency of change and thus repair. Imagine that a bug has been discovered in production. Without Continuous Delivery, a bug fix has to be attached to a release. The latency with which that release moves from development to production is proportional to the complexity of all the changes contained within it. With Continuous Delivery, on the other hand, the bug fix can be released to production as quickly as a single change can be coded and tested.

Fearless releases are an extension of Continuous Delivery. Having practiced, automated, and wrung the difficulty out of the change deployment process, you no longer fear it. The prospect of having to change production in response to a failure no longer makes you hesitate. On both physical and psychological levels, fearless releases

help reduce deployment latency and thus reduce time-to-repair.

Infrastructure-as-Code makes it possible to apply Continuous Delivery and fearless releases to infrastructure as well as application changes. Software-as-a-Service, with its inseparable requirements for functionality and operability, necessitates treating infrastructure and application changes inseparably in terms of quality and speed. Failures happen at all levels of the IT stack; we need the ability to minimize MTTR throughout.

Microservices increase the surface area of code by decomposing applications into small, loosely coupled services. Reducing size and coupling shrinks internal complexity and the scope of impact of change. Combined with Continuous Delivery, microservices can dramatically reduce the duration and cost of failure. Imagine a monolithic application with a monthly release cycle. A bug in production will impact all users for, on average, 15 days. If we assume 1,000 users, the average cost of the bug is 15,000 user-days.

Now imagine a microservices architecture with a Continuous Delivery release process. One service has a bug. That service impacts 1/10th of the user base. Releasing a change to production takes on average one hour. The average cost of this bug is 100 user-hours, or approximately four user days. The combination of microservices and Continuous Delivery has thus reduced the cost of failure by a factor of nearly 4,000!

Complexity makes it infeasible to assume robustness on the part of the systems upon which one relies. Instead, one needs to design systems and applications at any given layer in the IT stack based to assume the possibility of failure at lower levels. Design-for-fail necessitates thinking about fault tolerance at all layers. No longer can application developers confine themselves to thinking about functionality. They must also consider the question of how to provide that functionality in the face of database, network, or other outages.

Design-for-fail impacts not just code but also design. Amazon's use of a microservices architecture for its website means that the site as a whole nearly never goes down. Individual services, on the other hand, experience outages and degraded performance all the time. Amazon has built its user interface to gracefully degrade in the face of service failures. If a given service is misbehaving, the website will remove that service from the user interface. It essentially takes degraded functionality out of production from the user's point of view.

It's this level of design-for-fail that makes the microservices cost-of-failure math truly work. In order to say that a bug only impacts a subset of your customers, you have to be able to render the bug invisible to the rest of your customers. If, for example, Amazon's customer review service is misbehaving and the company removes it from

the visible interface, it only impacts customers who want to read reviews during the period of the service outage.

Design-for-fail promises the ability to gracefully survive component failure. No design for a complex system, however, can completely prevent visible errors. No matter how well we design complex socio-technical systems, things inevitably go wrong. We need to know how to respond when components and systems fail in order to fix them. What do we do if a database crashes? Perhaps more interestingly, what do we do if the database failover system or the data backup and restore system fail? Most importantly, how do we even know what's liable to fail?

In 1958, Ross Ashby, one of the pioneers of cybernetics, identified a principle he called the Law of Requisite Variety. Simply put, the law states that a control mechanism must be as rich as the system it's controlling. If an IT system can fail in any of 100 different ways, the IT organization needs the ability to recognize and respond to all 100 kinds of failures. In order to develop an appropriately rich failure management mechanism, the organization needs to identify the possible failure scenarios along with adequate responses to them.

Game days evolved as a means of generating requisite variety for IT failure management. Game days simulate real, game-time situations. These simulations intentionally generate failures in test environments in order to find out what might go wrong and to test possible responses. Game day facilitators will do things like shut down databases, disconnect networks, and delete data without telling the simulation participants. These exercises assume that control systems and operational systems can fail. By uncovering and accounting for failure modes in simulation mode, Game days can help prevent real failures in production.

The 2008 U.S. presidential election illustrated the power of game days. The Obama and Romney campaigns both relied on volunteers going door to door on Election Day to maximize voter turnout in their favor. The effectiveness of this strategy depended on volunteers' ability to accurately target which doors on which streets to knock.

Each campaign developed software that allowed volunteers to access voter information in order to guide them as they walked about their neighborhoods. The Obama campaign ran game days during the weeks leading up to election day. The Romney campaign didn't finish developing its application until just before the election; as a result, there wasn't time to test it in advance. When Election Day came, the Romney campaign's application melted down. The Obama campaign's application, on the other hand, ran without a hitch. It would be overstating the situation to claim that the different won the election for Obama; it did, however, make a nontrivial difference.

If we know that our systems are rife with failure, what should we do about it? Should

we try to eradicate it even though we know the attempt will likely backfire? Should we ignore it and hope for the best? Or should we instead go looking for failure and try to expose it to the light of day. That way we can learn from it and adapt to it, making our systems better as a result.

Netflix pioneered exactly this approach with its so-called *chaos monkey*. The *chaos monkey* is an automated software system that intentionally generates component failures in production. The *chaos monkey* in effect runs game days during the real game.

One of the challenging characteristics of complex systems is their sensitivity to history. This characteristic tells us we can never hope to perfectly test a system outside of production. Production will always be a little different; if not in some subtle architectural or configuration difference, then in the patterns and volume of user behavior and information flow. The *chaos monkey* operates on the principle that the only way to test production is in production.

One might think that a successful *chaos monkey* run is one that doesn't cause any visible production problems. If the production systems can survive the *chaos monkey*, that sounds like a positive result. According to Adrian Cockcroft, though, that's not the case. If you know that failure is lurking within your systems, success involves finding that failure. Netflix considers successful *chaos monkey* runs to be ones that visibly break the application. Those runs are the ones that show the company how to improve its systems.

After an outage has come and gone, organizations generally conduct postmortems. A postmortem's ostensible purpose is to identify the root cause for the outage and understand how to prevent it from happening again. Unfortunately, they too often devolve into assignment of blame. Focusing on blame is counterproductive. It discourages the openness necessary to understand the true causes of an outage and thus inhibits learning.

Blameful postmortems also miss the point that complex systems failures often lack a single root cause or even multiple linear causal chains. Searching for someone to blame is thus not only counterproductive but also futile and misguided. Furthermore, if failure is inevitable, as it is in complex systems, then prevention is a fool's errand. Learning is the only viable purpose for a postmortem.

Learning from mistakes requires fearlessness. This imperative leads to the practice of blameless postmortems, pioneered by John Allspaw at Etsy. A blameless postmortem encourages engineers to expose their assumptions, actions, and mistakes without fear of retribution. It encourages participants to share the maximum, rather than the minimum, possible information about their role in the outage. Not only does exposing

problems, assumptions, and limitations increase opportunities to improve systems and procedures; it also drives institutional learning by letting everyone witness one another's thought processes.

Finally, blameless postmortems have an important psychological and organizational effect. They treat engineers as intelligent people trying to do the right thing rather than as untrustworthy, potentially defective cogs in the machine. By treating team members with respect, it puts another nail in the coffin of the Taylorist industrial approach to employee management. Complex socio-technical systems require human initiative and creative decision making. Blameless postmortems make a critical contribution to sustaining that spirit within IT organizations.

Operational transparency takes blameless postmortems one step further by exposing the details of outages to all interested parties, including customers. One would think that's the last thing service providers would want to do. It would seem to cripple their credibility in their customers' eyes. The fact is, though, that customers know there was an outage. The provider isn't saving themselves any embarrassment by hiding the details. Instead, they are presenting themselves as an organization that believes in honesty, learning, and continuous improvement on its customers' behalf.

We all know from daily life that service doesn't always work perfectly. In fact, we often value a service provider's efforts to fix something for us. If the situation is handled well, we can end up thinking more highly of the service provider after a problem than before. Operational transparency follows this philosophy. It reflects the growing understanding that distributed, digital services are in fact complex and do in fact fail in ways that can neither be foreseen nor prevented.

The need to accommodate and seek out failure goes beyond the confines of IT. Complexity characterizes the entirety of a digital business and its interaction with customers and the market. In the post-industrial economy, brand management becomes a matter of continuous repair.

No organization can perfectly predict market or customer behavior. Brand failure comes in many flavors. It can take the form of anything from a website outage on Black Friday to a poorly designed new offering. The digital conversational medium helps us better understand and serve customers; it does not, however, guarantee success.

Post-industrial businesses need to take the lessons of complexity to heart on every level. Valuing MTTR over MTBF, for example, makes it possible to minimize the negative brand impact of failure. The power of practices such as Continuous Delivery and microservices transcends purely technical repair. Their greatest benefit is minimizing mean time to repair. A bug corresponds to a unit of customer dissatisfaction; from that perspective, any gap between current and preferred is a bug.

Perfectly continuous customer satisfaction is unachievable. Post-industrial business requires the ability to continuously repair dissatisfaction. One might even claim that the transition from industrialism to post-industrialism involves a shift from MTBF (manufacture millions of identical cars) to MTTR (fluidly respond to disruption).

Continuous repair is a key part of continuous design. As soon as we deliver the things we've designed, they escape our control and lead our customer conversations in unpredictable directions. In order for these conversations to remain useful, we need the ability to continuously repair breakdowns. Digitally infused service depends on IT to conduct conversations and thus to repair breakdowns in them. MTTR-centric IT practices make it possible to repair these breakdowns with minimal latency and therefore minimal customer dissatisfaction.

The need for design-for-fail goes beyond even the user-interface level. Service as a whole has to be able to survive failures of all kinds. If, for example, the customer management system experiences an outage, support representatives still have to answer the phone and provide meaningful information. If a snowstorm makes it impossible for them to get to work, the company still has to help customers with problems. Service design thus needs to consider systemic and human failure just as infrastructure design considers technical failure.

In addition to testing technical design-for-fail solutions, we also need to test human and social ones. Business operations faces similar challenges to technical operations. How will customer support provide meaningful information if the customer management system goes down? If your website fails on the most important shopping day of the year, how will your CEO respond to press inquiries? The only way to know for sure is to simulate potential failure scenarios.

Blamelessness and transparency are becoming business concerns as well as engineering concerns. Nontechnical organizations are beginning to adopt cybernetic approaches to their own work through such practices as Agile marketing. Companies are recognizing the importance of complex-systems structures such as open innovation and platform ecosystems. In order to leverage the power of conversational systems, they also need to transform their attitude toward failure. An Agile marketing campaign, for example, doesn't proceed by striving for perfection. In order to steer its way to success, it needs to analyze and respond to failure blamelessly.

Particularly as social media shifts brand power from the corporation to the customer, consumers have less and less patience for opaque business practices. It's no longer enough for a company to publicly acknowledge a security breach. Customers immediately want to know how and why the breach happened, and what the company is doing to prevent it from happening again. Attempts at information hiding quickly become negative brand moments. Brand repair depends first of all on acknowledging

that trust has been broken, and second on communicating the concrete internal activities that can restore trustworthiness.

Complexity's deepest lesson concerns the limitations of reductionist analysis. There is no way to guarantee the correctness of any possible choice. It's not even possible to guarantee that a reasonable-seeming choice won't make things worse. The best you can do is to try, and to be prepared to continually try again based on the results of each attempt. This approach epitomizes the cybernetic worldview.

If it isn't possible to identify solutions purely through analysis, how can you generate possibilities? Any attempt to manage complex systems needs to incorporate a nonanalytical component. Design thinking offers tremendous insight in this respect. Its humanistic heritage, drawing on philosophy, art, architecture, and literature, provides a powerful counterbalance to IT's engineering legacy.

As much as anything, complex systems force us to embrace new ways of knowing. We need a way to proceed through a world where there is no single root cause, and what causality we find is circular in nature. We need the ability to experiment through disciplined wandering.

The inevitability of failure ironically frees us from the anxiety of trying to "get it right." Instead, we can design for failure, optimize for mean time to repair, and build in feedback loops that bound our wandering. Within those bounds, we can use the designer's mind to choose where to go next. Digital businesses that infuse design thinking the most deeply into their post-industrial control mechanism are best positioned to respond to the challenges of service, infusion, complexity, and disruption.

Continuous design, in its guise as continuous repair, functions by continually transforming failure into success. Failure might take the form of a complex-systems outage or a gap between what the customer needs and what the company just produced. It might even manifest as the customer responding to a new feature by saying, "That's wonderful; now how about you make it do this?"

Whatever the cause, if a business stops modifying itself in ways that are meaningful to its customers, it will die, either from stasis or from wandering off into irrelevancy. Complex landscapes change as we move through them, making it impossible to predict the right destination or even the right direction. The only way we can navigate them without foundering on unforeseen rocks or reefs is through never-ending, cybernetic steering.

Conversational IT and the digital business it powers never sit still. They never assume they're done. They never stop listening or responding to what they've heard. The musician Laurie Anderson captured this dynamic in her song "Walking And Falling":

> You’re walking. And you don’t always realize it,
>
> but you’re always falling.
>
> With each step you fall forward slightly.
>
> And then catch yourself from falling.
>
> Over and over, you’re falling.
>
> And then catching yourself from falling.
>
> And this is how you can be walking and falling
>
> at the same time.

[1] “Front-office” and “back-office” correspond to what IT typically refers to as Systems of Record and Systems of Engagement.