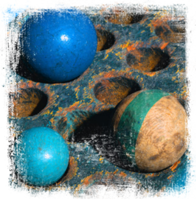

Tom's Top Ten Things Executives Should Know About Software

THOMAS A. LIMONCELLI

SOFTWARE ACUMEN IS THE NEW NORM

A friend of mine is an accountant at a large company. The CEO and other executives don't know what accounting is, and that's OK. Everyone works around it.

OK, that's a lie. No company like that exists.

I do have a friend, however, who is a software engineer at a large company where the CEO and other executives don't understand software. They don't understand what is reasonable to expect software to do, how it is made, how software projects are managed, or how a web-based service is run.

That isn't something that employees can "work around."

Maybe that was OK years ago, but it isn't OK now. In fact, my advice to this friend was to start sending out her resume.

Many companies that don't think of themselves as software companies are finding that software is a key component of their operations. If executives and management do not understand how software is made, they will be ineffective compared with those who do. This will either limit their careers or negatively affect their company's performance. Either way, they're doomed. (You don't have to take my word for it: Gartner predicted that 50 percent of CIOs who haven't transformed their organization's capabilities by 2020 will be displaced.[9])

In this column I list the things that "executives who get software" understand in an effort to help those executives and managers who have found themselves in this new world. The list is not exhaustive, as the full list could fill multiple books, but it is based on a very unscientific poll of my friends in the industry.

SOFTWARE ATE THE WORLD

In 2011 Marc Andreessen[1] wrote an article predicting that "software will eat the world." By that he meant two things. First, many traditional businesses are being replaced by software companies. Second, all other companies are finding that the value they deliver is increasingly a result of software.

When Andreessen wrote his article none of the 10 biggest companies (by market value) were in software-driven businesses. Today 6 of the 10 biggest companies are primarily driven by software. The others are ripe for a transformation.

The first category is easy to understand. Your local music store has probably been replaced by online music stores such as Apple's iTunes. A physical location is eliminated in favor of one solely defined by software.

The second category is a subtler change. For example, while automobiles haven't been replaced by websites, what makes auto companies succeed is increasingly a result of their software acumen. Their supply chain, manufacturing, marketing, and sales processes are controlled by software. The typical car has 10 to 100 million lines of code in it. The majority of what differentiates car models comes from software-powered features such as the dashboard and audio system. Everything I love about my new car is software; everything I dislike is software that could have been better.

I recently purchased a very low-tech refrigerator. As far as I can tell, it has zero software inside of it. Not long after the purchase, however, I received an email offering a water-filter-replacement subscription. That system is entirely software-driven. Considering the high price of the subscription filters, I presume they are responsible for more profit than the original purchase.

If you don't control what makes your product valuable, then you aren't much of a company.

If you don't control what makes your product valuable, then you aren't much of a company. Therefore, executives and managers must now understand the software-delivery lifecycle.

Both Stack Overflow Talent and LinkedIn now list more software engineering job advertisements for nontechnical companies than the tech industry itself.[11] This is a major shift in the economy and indicates that companies are ramping up their software practices.

THE LIST

Here are my top ten things I believe all executives and managers need to know about software:

1. Software is not magic

Software is not magic. Often it looks like magic, or is magical, but it isn't magic. Every element was designed by a human and has its basis in math or a process that can be explained in human words.

Unlike magic, software isn't conjured out of thin air. It needs to be designed, built, and operated. Just as a house has layers of systems that work together (foundation, structure, plumbing, rooms, furniture, and so on), software has layers and subsystems that create the whole. It can be designed well or badly, and a fast design is rarely a lasting one.

If you can't describe in words what it will do (both the desired outcome and how it can get there), then a computer can't do it. The "how" is called an *algorithm* and is not magic.

A web search for pictures of chairs doesn't actually show pictures of chairs. It shows images that frequently appear on web pages that mention the word *chairs*. The difference is subtle, and it took years for someone to think of that trick and perfect the technique. Yet, it isn't magic.

Your email system's spam detection looks pretty magical, but it is not magic. Bayesian statistics and other mathematical models work under the hood to achieve the behavior you see.

Autocorrect feels magical (try turning it off for a day), but the best autocorrect systems process trillions of data points from the past to create a database of precomputed autocorrections to use in the future.

ML (machine learning) and other AI techniques are not magic. ML is prediction based on data, instead of explicit rules or instructions. It is monkey-see-monkey-do using linear algebra. You have a million pictures known to have bananas, and a million pictures without bananas, and a trained ML system will look at a new photo and tell you if it looks like the first or second group based on what it has learned from the previous photos. Not magic. ML is enormously useful across many domains, but sometimes it can act like *The Sorcerer's Apprentice*. For example, using ML to sort resumes based on past hiring decisions can amplify a racist hiring history, even without any intended bias.[2]

2. Software is never "done"

Software is never "done." It is an iterative process with many revisions and updates shipping over its lifetime. Your job is to create an environment that recognizes this.

Likewise, we never expected marketing and customer acquisition to be "done". They too are iterative processes. In each iteration we learn and grow as we continue to deliver value to the business. We don't ever plan to "stop" doing any of those things, even once we find something that is a successful launch.

It would be nice if software could be finished in one release, but that is not reality. Requirement documents are full of ambiguity. The first release of software is full of "oh, that's what I wrote but it wasn't what I meant" moments. The best software inspires new ideas and new features. Seeing that the new sales-management system is more efficient inspires even more efficiencies. If you don't plan on future releases that will incorporate the best ideas of your employees, you have built a system that just solves yesterday's problems. The world changes, your competitors offer new features, people have new ideas. There are always bugs to be fixed—maybe in your code, or in the underlying software frameworks and systems that it is built upon. Your software may be perfect, but I assure you that over time people will find security holes in the platform it is built on.

It is your job to expect an organization that recognizes this.

The way we recognize this is to build an organization that confidently produces new software releases at regular intervals. When fully automated testing and other engineering disciplines are in place, we build confidence. This confidence creates the ability to eschew multiyear release cycles and instead ship high-quality software quarterly, monthly, or even weekly. The particular frequency isn't important; confidence is. Confidence leads to faster innovation.

This also means rejecting project plans that involve one perfect release, then no more. Or plans that do not involve sufficient testing, or eliminate the beta-test period, or allow developers to make changes to live production systems instead of having an approved and tested path to release. Features that make software more shippable should not be left until the end; ease of shipping has business value.

Lastly, let's stop with software projects that are allowed to run for multiple years before showing any progress. Release early and often. Require a minimum viable product to launch, followed by periodic releases that add features. The first release might be just the basic framework or support only a few edge cases. Each release provides an opportunity to get feedback and change course. Early releases might run only in a beta area, inaccessible to real users. At least you've started the feedback cycle. Beta testing saves lives—and careers.

Of equal importance, behind-the-scenes operations now have a chance to begin developing their processes and procedures, build and vet infrastructure, and test the invisible foundation that supports everything else. Imagine if the Obamacare website had first supported only Rhode Island, then added support for states one at a time. The experience from each iteration would have propelled it forward and made it a success from the start.

3. Software is a team effort; nobody can do it all

Software is a team effort. The developer is neither product manager nor UX (user experience) designer nor quality assurance analyst nor security guru nor technical writer nor operations engineer. You need them all.

No executive would propose that each salesperson do his or her own marketing, or that the sales force should be fired because marketing understands the product and can do sales, too. Marketing and sales are related but different. Therefore, a division of labor exists between the two.

Likewise, software teams need separate people for requirements gathering, quality assurance and test engineering, technical writing, and so on.

There is a myth of the developer who "does it all," known as the "full stack developer" or "10x engineer." This doesn't exist outside of the smallest company. Yes, a very small company may have a single person who does both marketing and sales, but you probably don't work for a company that tiny. Neither do your engineers.

Yes, your 12-year-old son made a website all by himself. Don't let that make you think that it can't be that difficult, or that coding is "just typing." I assure you Johnny's website isn't processing billions of financial transactions per hour.

When I was 10 years old I built a "robot" out of cardboard boxes. My parents were smart enough to take that as an indication that I was interested in engineering, not to think I could skip calculus.

Which reminds me: Dear Parents, Just because your child is "good at Facebook" doesn't mean he or she will be the next Zuckerberg. Stop saying that at cocktail parties. It's embarrassing. (P.S. No teens use Facebook any more. Your kids post to Facebook only as a decoy so that you don't try to find where they actually hang out. I'm sorry you had to learn that here.)

4. Design isn't how something looks; it is how it works

Steve Jobs famously said, "Design is not just what it looks like and feels like. Design is how it works." UX designers don't sit around trying to decide what color the menus will be, or if buttons will be round or square. They determine what the workflow and interactions will be.

Will the user be presented with one screen with three choices, or will the choices be presented one screen at a time? The decision requires psychology, empathy for the user, and testing, testing, testing.

One of the biggest challenges of UX design is that once you know the system well, you lose the ability to predict what a new user will expect. The person who designed the system is automatically disqualified for predicting what a new user will need.

I remember the first time I had the opportunity to watch users through a one-way mirror as they used a product I was involved with. "It's the button on the left! Look to the left! Oh god, why aren't they clicking the button on the left!" Then reality sets in. Our brilliant placement of buttons was brilliant only because we knew the system so well.

Therefore, testing with real users is required. This could be as simple as recruiting a coworker who has yet to see the system, or as complex as using a double-blind study with one-way mirrors and eye-tracking systems.

I also remember watching someone test Google Maps. The user was asked to route from New York Penn Station to a particular hotel. After that task was completed, the UX designer asked, "What do you think you'd want to do now?" The person responded, "Once I'm checked in, I find a restaurant." Soon after, Google Maps added a "find a nearby restaurant" feature. That's a good UX designer!

A UX may be beautiful and elegant and comparable to a piece of art, but asking a UX designer to change the background to a picture of a sailboat is not helpful.

It is your job to trust testing data over opinions, to create an environment that plans for multiple revisions before the product ships and expects further refinement after.

Do not confuse a UX designer with a graphic designer. A graphic designer develops layouts to inspire and inform in a variety of media from brochures to websites. Asking a UX designer to design the company holiday card is as much of a faux pas as asking the technical writer to write the company newsletter. These are all different skills.

5. Security is everyone's responsibility

You are in the security business whether you know it or not, and whether you want to be or not. All software has security requirements and potential security vulnerabilities. The systems involved in producing your software have security requirements and vulnerabilities, too. While security infrastructure components such as firewalls and intrusion detection are necessary, they are not sufficient: you must also design, implement, and operate your software platforms with built-in security controls. Security is as much about good process as it is good technology.

If you think you are not a target, then you are wrong. All computer systems are targets, as the prize is not just the information in them but the mere fact that it is a computer. For example, a system with no information of value is a cybersecurity target because it can be used to relay an attack on other computers, or mine bitcoin, or store someone's pirated video library.

Security is not an on/off switch. There are many shades of gray. You don't build a system, then press the "make it secure" button.

Security is about risk and your tolerance level for risk. Encrypting communication between two points doesn't make it secure, but it enhances the security such that only a superpower has the resources to crack the code. Mitigating risk in one area doesn't help in other areas. Securing the network doesn't prevent physical security issues. An employee propping a door open enables someone else to steal your backup tapes.

As Gene Spafford famously stated, "The only truly secure system is one that is powered off, cast in a block of concrete and sealed in a lead-lined room with armed guards—and even then, I have my doubts."[3]

Compliance with security standards such as NIST

CSF (National Institute of Standards and Technology Cybersecurity Framework), PCI DSS (Payment Card Industry Data Security Standard), and SOC 2 (Service Organization Control report) quantifies risk and, when done right, reduces risk. These standards do not assure perfect security; such a thing does not exist. More importantly, they provide guidance on how to respond responsibly and report the inevitable security breach. Being honest, forthright, and public is my recommendation.

Security is best designed in from the start. Bolting it on after the fact is expensive and often ineffective. You wouldn't build a boat and then "add in" a way for it to float.

Today the most common vector for security issues is not the sexy high-tech security hole some elite hacker discovered last night. It is the old, boring, everyone-else-fixed-it-years-ago issue that goes unnoticed. You would be stunned at how many systems are calcified and can't be updated because updates are impossible, expensive, or unavailable. They may have been considered (relatively) secure when new, but now new vulnerabilities have been discovered. Software, left alone, grows stale like bread.

It is your job to balance security paranoia with reality, and budget time and resources appropriately.

6. Feature size doesn't predict developer time

Feature size (as perceived by users) is entirely unrelated to how long it will take to create said feature. Small features can take days or years. Big features (as perceived by users) can take days or years.

It is your job to create and support a software development process that accepts this and does not second-guess engineering's work estimates. Producing the work estimate itself may take a surprisingly long time.

Negotiation is encouraged. The engineers may reply with a surprisingly long work estimate but offer changes to the requirements that will cut the time significantly. Remember to include time for testing, training, deployment, and unexpected family or medical leave.

Never promise a feature without consulting with engineering for a work estimate. It is not a sign of your corporate power to promise a feature by a certain deadline on the spot. I assure you that what people find more impressive is a professional process where their request is taken seriously, work estimates are produced, and the request is delivered on time (or rejected for honest reasons).

7. Greatness comes from thousands of small improvements

Greatness comes from thousands, perhaps millions, of small improvements done over a long stretch of time. The effect of each change is measured, and the change is rolled back if the outcome is negative.

Google wasn't built in a day. Google's search engine is the result of millions of individual improvements. Once a week the search-quality panel meets. Engineers step up to the podium and present their proposed changes. They show how much of an improvement would be made based on simulations. The committee debates and votes it up or down. Weeks later the measurements are reviewed, and the change is either kept or rolled back.

Google search is the triumph of iterative development over "big bang" thinking. You can't make a good search engine on your first attempt.

Only in Hollywood movies does a brilliant young mind come up with an amazing new idea that is implemented and works perfectly the first time. In the real world it takes years to create an overnight success.

This is true whether the greatness you are trying to achieve is a system that provides better service to customers, is more efficient, has fewer errors, or just organizationally runs more smoothly.

It is your job to require systems to be designed to make it easy to try new things and to define pertinent KPIs (key performance indicators) that can easily be measured before and after changes. Most importantly, there must be a process by which the results are examined and a decision is made to keep or roll back the change. A rollback should not be considered a failure or be punished. What is learned from each rollback is as valuable as what is learned in each change that is retained.

Thomas Edison claimed to have tested 1,000 filaments on the way to creating his light bulb. When a reporter asked, "How did it feel to fail 1,000 times?" he replied, "I didn't fail 1,000 times. The light bulb was an invention with 1,000 steps."

This is another reason why software systems need to support rapid releases.

The biggest improvements come from working across silos and involving all stakeholders. If there is no collaboration across teams, then each team will optimize their area, often to the detriment of the efficiency of the other teams. By working across teams, you develop empathy and can create the most impactful changes.

I recently read about a U.S. company that stayed ahead of foreign competition through efficiency. It was able to achieve this advantage by constantly examining the end-to-end process. In one case large amounts of materials and manufacturing time were being spent on plastic covers. A major customer was removing and disposing of those covers because they got in the way. If the manufacturer hadn't visited the customer, it would never have realized it could improve efficiency by selling a model without that cover.

Likewise, both the process of building software and the process in which the software is used must be under constant revision brought about by end-to-end examination.

8. Technical debt is bad but unavoidable

Technical debt is the work you will need to do in the future because you chose an easy solution now instead of using a better approach that would take longer. Any software project of reasonable size has technical debt.[7] Technical debt makes all forward progress slower, and it snowballs the more you ignore it.

I fear that executives with a finance background hear "debt" and think it is an investment that will pay off in the future. Technical debt is the opposite. It is toxic and painful. It is a ticking time bomb. Caskey L. Dickson[4] compares it with "naked call options," future obligations that could arise at any time, without advance notice, and having an unlimited downside.

In 1972 Fram ran a TV commercial for its oil filters in which an auto mechanic explained that a customer tried to save $4 by not replacing a filter; later the customer had to pay $200 for an expensive main bearing replacement. The mechanic concluded, "You can pay me now, or pay me later."[8]

Once I was involved in a software project with a subsystem that communicated to a supplier. Initially the system talked to only one supplier, so that was pretty easy. Then a second was grafted on. Then another. Some features had to be implemented three times, once for each supplier. This was not sustainable.

When asked to support a fourth supplier, the developers revolted. Yes, they could graft it on in about a month, but the software was starting to creak like an old house in a hurricane. The quick fixes had accumulated technical debt.

Their proposal was to spend two months refactoring (reworking) the supplier architecture so that it was a plug-in system. New suppliers could then be added in a week, not a month.

The executives weren't happy. Why would this next supplier take more than two months to add when previous suppliers were added in one month?

The two months invested in paying down technical debt would make future additions faster, stabilize the code base, and make it easier to add new features. It is difficult to measure the exact benefits.

You can pay me now, or pay me later.

It is your job to allocate time to pay down technical debt. Runaway technical debt slows down the ability to add other features, and it leads to unstable software. Paying down technical debt should be tied to business goals, similar to nonfunctional features.

9. Software doesn't run itself

While vendors and developers may try to tell you otherwise, software doesn't just run itself. Any software-based system (websites and web applications, in particular) requires operational staff and processes; otherwise, it just sits there like a closed book. Someone has to turn it on, care for it, and tend to its needs.

I assert that operations is more important than software development itself. Code is written once but runs millions of times. Therefore, operations are, by that rough measure, millions of times more important.

As a result, it is your job to expect operations to be part of any software-based system. It must be planned for, budgeted, managed, and run efficiently just like anything else.

Operational features (usually called nonfunctional features) are invisible to users except as second-order effects. Data backup is a good example of a nonfunctional feature. No user requests data to be backed up. Users do, however, ask for deleted data to be restored. Sadly, there can be no restore without a backup. A restore is a functional feature; a backup is an operational (nonfunctional) feature.

Features that make a software service easy or efficient to operate are never requested by users. They do, however, enjoy the benefits of a system that is cost effective and reliable. Customers leave unreliable websites and don't come back.

Software must be scaled, monitored, updated, and so on. Wikipedia has an excellent list of nonfunctional requirements that drive such features.[13] Operations are in a constant battle to improve efficiency. This often requires new code.

The need for continuous improvement includes not just new features, but new nonfunctional features. Therefore, it is your job to allocate resources not only for the features that customers demand, but also for operational features. Striking a balance between the two competing needs is difficult.

A successful product is the negotiated union of business and operational requirements.

10. Complex systems need DevOps to run well

A complex system is best improved through DevOps. This has many definitions, but I prefer to think of DevOps as accelerating the delivery of value (features, bug fixes, process improvements, and so on) by rapid iteration. To achieve this, everyone involved must participate. That is, they must work across silos. The name *DevOps* comes from the movement to remove the wall between developers and operations (IT), which is absolutely required to achieve rapid releases. Great DevOps environments, however, extend this to work across all silos end to end.

DevOps has been misinterpreted to mean developers perform operations. This "you build it, you run it" strategy is one way of working across silos (eliminating them), but it isn't the only way. More on that later.

The system that builds and continuously improves your software is a machine. Every time you turn the crank, a new (hopefully improved) release of software pops out and goes into production.

Delivering your product to customers is also a machine. Your marketing, sales, logistics, billing, and other systems all work together. Every time you turn the crank your product is delivered.

Either kind of machine is a complex system with many dependencies. To run well, a complex system needs three things: a good process, good communication by all the people involved, and the ability to try new things.

These are codified as the Three Ways of DevOps:

- *The First Way is "System Thinking" or "Flow."* The focus here is on improving the entire system, not specific silos, as described in item 7 above. The First Way is about driving improvements that move you from a process that sucks to one that is awesome. In pathological cases the process is nonexistent—each silo improvising and guessing its way through the process each time the crank turns. The result of the First Way is improved velocity and reduced defects. Things work better.
- *The Second Way is "Amplify Feedback Loops."* The focus is on improving communication among the people and components within the system. Communication is a feedback loop and should be bidirectional, responsive, transparent, and blameless. A system can't work well without the ability of the people involved to learn, share, and grow. The Second Way is about driving improvements that move you from communication that is lacking to communication that is comprehensive. In pathological cases communication is punished. The result of the Second Way is understanding, empathy, and responsiveness to customers both internal and external. Knowledge is where it is needed.

- *The Third Way is a "Culture of Continual Experimentation and Learning."* This is where you focus on creating a culture where you try new things, evaluate the results, and decide whether to keep or revert the change. The Third Way is about going from a culture where change is resisted to one where change is constant. Risk is accepted. Rituals reward teams for taking risks and learning from failure. In pathological cases the organization is calcified: change isn't possible, suggestions for change are rejected or possibly punished. The result of the Third Way is evolutionary change over time, punctuated by major leaps and innovation.

WAIT, THERE'S MORE

There are volumes more that an executive should know about software. Sadly, cultural pressure and David Letterman say I should stop at 10.

Here are some bonus items:

Bonus Item 1. Uptime is never perfect.

Asking for 100 percent uptime makes you look ignorant. Each order of magnitude of improvement costs ludicrously more than the level prior: 99.0 percent uptime is fine for plenty of systems; 99.999 percent is more expensive than you can afford. Punishing people for downtime sends the wrong message. Instead, ask "What did we learn?" If your organization learned something, the downtime was a gift. Recommended reading: *Beyond Blame, Learning from Failure and Success*, by Dave Zwieback,[14] and Wikipedia's page on High Availability.[12]

Bonus Item 2. Spammers and abusers ruin everything.
Fighting spam and abuse is an arms race. If you can build an online app in a week, you'll spend a year figuring out how to prevent spammers from ruining it. Google Sheets has anti-abuse detection because criminals make spreadsheets full of links to scams and then send the links to people who think any link that mentions Google is safe. The amount of anti-abuse work required to run online communities such as Twitter, Facebook, or other social networks would make you cry.

Bonus Item 3. Malleability is expensive.
Some changes to software require a new release, while other changes can happen while the system is running. The latter is expensive. It would be easy for Facebook profiles to store only your name, location, and a few other facts. The ability to store any field is an expensive engineering task. Be careful when asking for flexibility. It affects testing, security, usability, and a lot more.

CONCLUSION
Software is eating the world. To do their jobs well, executives and managers outside of technology

Related articles

The Age of Corporate Open Source Enlightenment
Paul Ferris
Like it or not, zealots and heretics are finding common ground in the open source holy war.
https://queue.acm.org/detail.cfm?id=945124

Managing Technical Debt
Shortcuts that save money and time today can cost you down the road.
Eric Allman
https://queue.acm.org/detail.cfm?id=2168798

Why Cloud Computing Will Never Be Free
The competition among cloud providers may drive prices downward, but at what cost?
Dave Durkee
https://queue.acm.org/detail.cfm?id=1772130

will benefit from understanding some fundamentals of software and the software-delivery process.

Further resources

If you are an executive who wants software acumen, there are many resources. The first is your VP of engineering or CTO. Ask the person in one of these jobs what you should learn.

I also highly recommend reading *The Phoenix Project: A Novel about IT, DevOps, and Helping Your Business Win*, by Gene Kim, Kevin Behr, and George Spafford.[10] It provides an inside view of IT and a practical understanding of how to use DevOps techniques to manage it.

I also recommend *Accelerate: The Science of Lean Software and DevOps: Building and Scaling High-Performing Technology Organizations*, by Nicole Forsgren, Jez Humble, and Gene Kim.[6] It provides a CEO view of the science that makes DevOps work.

Acknowledgments

Lynn Ballard, Nicole Forsgren, Tim Bell, Lance A. Brown, Jennifer Davis, Trent R. Hein, Mark Henderson, Steve VanDevender, Harald Wagener

References

1. Andreessen, M. 2011. Why software is eating the world. *The Wall Street Journal* (August 20); https://www.wsj.com/articles/SB10001424053111903480904576512250915629460.
2. Buranyi, S. 2017. Rise of the racist robots—how AI is learning all our worst impulses. *The Guardian* (August

8]; https://www.theguardian.com/inequality/2017/aug/08/rise-of-the-racist-robots-how-ai-is-learning-all-our-worst-impulses.

3. Dewdney, A. K. 1989. Computer recreations: of worms, viruses and core war. *Scientific American* 260(3), 110.
4. Dickson, C. L. 2015. Why your manager loves technical debt and what to do about it. *Proceedings of the Usenix LISA (Large Installation System Administration) Conference*; https://www.usenix.org/conference/lisa15/conference-program/presentation/dickson.
5. Fong-Jones, L. 2018. Twitter; https://twitter.com/lizthegrey/status/1052636505712275458?lang=en.
6. Forsgren, N., Humble, J., Kim, G. 2018. *Accelerate: The Science of Lean Software and DevOps: Building and Scaling High Performing Technology Organizations*. IT Revolution Press.
7. Fowler, M. 2003. Technical debt. Martinfowler.com; https://martinfowler.com/bliki/TechnicalDebt.html.
8. Fram Oil Filter commercial. 1972; https://www.youtube.com/watch?v=OHugOAIhVoQ.
9. Gartner. 2016. Gartner predicts; https://www.gartner.com/binaries/content/assets/events/keywords/infrastructure-operations-management/iome5/gartner-predicts-for-it-infrastructure-and-operations.pdf.
10. Kim, G., Behr, K., Spafford, G. 2013. *The Phoenix Project: A Novel About IT, DevOps, and Helping Your Business Win.* IT Revolution Press; https://www.goodreads.com/book/show/17255186-the-phoenix-project.
11. Snover, J. 2018. Digital transformation: thriving through the transition; DevOps Enterprise Summit; https://www.

youtube.com/watch?v=qHxkcndCQoI.

12. Wikipedia. High availability; https://en.wikipedia.org/wiki/High_availability.
13. Wikipedia. Non-functional requirement; https://en.wikipedia.org/wiki/Non-functional_requirement.
14. Zwieback, D. 2015. *Beyond Blame: Learning from Failure and Success*. O'Reilly Media; https://www.goodreads.com/book/show/23237459-beyond-blame.

Thomas A. Limoncelli *is the SRE 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.*