2 Sep 2005 | 3:18 GMT

Why Software Fails

We waste billions of dollars each year on entirely preventable mistakes

By **Robert N. Charette (/author/charette-robert-n)**

Photo: Graham Barclay/Bloomberg News/Landov

Market Crash: After its new automated supply-chain management system failed last October, leaving merchandise stuck in company warehouses, British food retailer Sainsbury's had to hire 3000 additional clerks to stock its shelves.

Have you heard the one about the disappearing warehouse? One day, it vanished—not from physical view, but from the watchful eyes of a well-known retailer's automated distribution system. A software glitch had somehow erased the warehouse's existence, so that goods destined for the warehouse were rerouted elsewhere, while goods at the warehouse languished. Because the company was in financial trouble and had been shuttering other warehouses to save money, the employees at the "missing" warehouse kept quiet. For three years, nothing arrived or left. Employees were still getting their paychecks, however, because a different computer system handled the payroll. When the software glitch finally came to light, the merchandise in the warehouse was sold off, and upper management told employees to say nothing about the episode.

This story has been floating around the information technology industry for 20-some years. It's probably apocryphal, but for those of us in the business, it's entirely plausible. Why? Because episodes like this happen all the time. Last October, for instance, the giant British food retailer J Sainsbury PLC had to write off its US $526 million investment in an automated supply-chain management system. It seems that merchandise was stuck in the company's depots and warehouses and was not getting through to many of its stores. Sainsbury was forced to hire about 3000 additional clerks to stock its shelves manually [see photo above, "Market Crash"].

(/image/1436123)

Sources: *Business Week*, *CEO Magazine*, *Computerworld*, *InfoWeek*, *Fortune*, *The New York Times*, *Time*, and *The Wall Street Journal.**Converted to U.S. dollars using current exchange rates as of press time.+Converted to U.S. dollars using exchange rates for the year cited, according to the International Trade Administration, U.S. Department of Commerce.**Converted to U.S. dollars using exchange rates for the year cited, according to the *Statistical Abstract of the United States, 1996*.

YEAR	COMPANY	OUTCOME (COSTS IN US $)
2005	Hudson Bay Co. [Canada]	Problems with inventory system contribute to $33.3 million* loss.
2004–05	UK Inland Revenue	Software errors contribute to $3.45 billion* tax-credit overpayment.
2004	Avis Europe PLC [UK]	Enterprise resource planning (ERP) system canceled after $54.5 million† is spent.
2004	Ford Motor Co.	Purchasing system abandoned after deployment costing approximately $400 million.
2004	J Sainsbury PLC [UK]	Supply-chain management system abandoned after deployment costing $527 million.†
2004	Hewlett-Packard Co.	Problems with ERP system contribute to $160 million loss.
2003–04	AT&T Wireless	Customer relations management (CRM) upgrade problems lead to revenue loss of $100 million.
2002	McDonald's Corp.	The Innovate information-purchasing system canceled after $170 million is spent.
2002	Sydney Water Corp. [Australia]	Billing system canceled after $33.2 million† is spent.
2002	CIGNA Corp.	Problems with CRM system contribute to $445 million loss.
2001	Nike Inc.	Problems with supply-chain management system contribute to $100 million loss.
2001	Kmart Corp.	Supply-chain management system canceled after $130 million is spent.
2000	Washington, D.C.	City payroll system abandoned after deployment costing $25 million.
1999	United Way	Administrative processing system canceled after $12 million is spent.
1999	State of Mississippi	Tax system canceled after $11.2 million is spent; state receives $185 million damages.
1999	Hershey Foods Corp.	Problems with ERP system contribute to $151 million loss.
1998	Snap-on Inc.	Problems with order-entry system contribute to revenue loss of $50 million.
1997	U.S. Internal Revenue Service	Tax modernization effort canceled after $4 billion is spent.
1997	State of Washington	Department of Motor Vehicle (DMV) system canceled after $40 million is spent.
1997	Oxford Health Plans Inc.	Billing and claims system problems contribute to quarterly loss; stock plummets, leading to $3.4 billion loss in corporate value.
1996	Arianespace [France]	Software specification and design errors cause $350 million Ariane 5 rocket to explode.
1996	FoxMeyer Drug Co.	$40 million ERP system abandoned after deployment, forcing company into bankruptcy.
1995	Toronto Stock Exchange [Canada]	Electronic trading system canceled after $25.5 million** is spent.
1994	U.S. Federal Aviation Administration	Advanced Automation System canceled after $2.6 billion is spent.
1994	State of California	DMV system canceled after $44 million is spent.
1994	Chemical Bank	Software error causes a total of $15 million to be deducted from 100 000 customer accounts.
1993	London Stock Exchange [UK]	Taurus stock settlement system canceled after $600 million** is spent.
1993	Allstate Insurance Co.	Office automation system abandoned after deployment, costing $130 million.
1993	London Ambulance Service [UK]	Dispatch system canceled in 1990 at $11.25 million**; second attempt abandoned after deployment, costing $15 million.**
1993	Greyhound Lines Inc.	Bus reservation system crashes repeatedly upon introduction, contributing to revenue loss of $61 million.
1992	Budget Rent-A-Car, Hilton Hotels, Marriott International, and AMR [American Airlines]	Travel reservation system canceled after $165 million is spent.

This is only one of the latest in a long, dismal history of IT projects gone awry [see table above, “Software Hall of Shame” for other notable fiascoes]. Most IT experts agree that such failures occur far more often than they should. What’s more, the failures are universally unprejudiced: they happen in every country; to large companies and small; in commercial, nonprofit, and governmental organizations; and without regard to status or reputation. The business and societal costs of these failures—in terms of wasted taxpayer and shareholder dollars as well as investments that can’t be made—are now well into the billions of dollars a year.

The problem only gets worse as IT grows ubiquitous. This year, organizations and governments will spend an estimated $1 trillion on IT hardware, software, and services worldwide. Of the IT projects that are initiated, from 5 to 15 percent will be abandoned before or shortly after delivery as hopelessly inadequate. Many others will arrive late and over budget or require massive reworking. Few IT projects, in other words, truly succeed.

The biggest tragedy is that software failure is for the most part predictable and avoidable. Unfortunately, most organizations don’t see preventing failure as an urgent matter, even though that view risks harming the organization and maybe even destroying it. Understanding why this attitude persists is not just an academic exercise; it has tremendous implications for business and society.

SOFTWARE IS EVERYWHERE. It’s what lets us get cash from an ATM, make a phone call, and drive our cars. A typical cellphone now contains 2 million lines of software code; by 2010 it will likely have 10 times as many. General Motors Corp. estimates that by then its cars will each have 100 million lines of code.

The average company spends about 4 to 5 percent of revenue on information technology, with those that are highly IT dependent—such as financial and telecommunications companies—spending more than 10 percent on it. In other words, IT is now one of the largest corporate expenses outside employee costs. Much of that money goes into hardware and software upgrades, software license fees, and so forth, but a big chunk is for new software projects meant to create a better future for the organization and its customers.

Governments, too, are big consumers of software. In 2003, the United Kingdom had more than 100 major government IT projects under way that totaled $20.3 billion. In 2004, the U.S. government cataloged 1200 civilian IT projects costing more than $60 billion, plus another $16 billion for military software.

Any one of these projects can cost over $1 billion. To take two current examples, the computer modernization effort at the U.S. Department of Veterans Affairs is projected to run $3.5 billion, while automating the health records of the UK's National Health Service is likely to cost more than $14.3 billion for development and another $50.8 billion for deployment.

Such megasoftware projects, once rare, are now much more common, as smaller IT operations are joined into "systems of systems." Air traffic control is a prime example, because it relies on connections among dozens of networks that provide communications, weather, navigation, and other data. But the trick of integration has stymied many an IT developer, to the point where academic researchers increasingly believe that computer science itself may need to be rethought in light of these massively complex systems.

When a project fails , it jeopardizes an organization's prospects. If the failure is large enough, it can steal the company's entire future. In one stellar meltdown, a poorly implemented resource planning system led FoxMeyer Drug Co., a $5 billion wholesale drug distribution company in Carrollton, Texas, to plummet into bankruptcy in 1996.

IT failure in government can imperil national security, as the FBI's Virtual Case File debacle has shown. The $170 million VCF system, a searchable database intended to allow agents to "connect the dots" and follow up on disparate pieces of intelligence, instead ended five months ago without any system's being deployed [see "Who Killed the Virtual Case File? (/computing/software/who-killed-the-virtual-case-file)" in this issue].

IT failures can also stunt economic growth and quality of life. Back in 1981, the U.S. Federal Aviation Administration began looking into upgrading its antiquated air-traffic-control system, but the effort to build a replacement soon became riddled with problems [see photo, "Air Jam," at top of this article]. By 1994, when the agency finally gave up on the project, the predicted cost had tripled, more than $2.6 billion had been spent, and the expected delivery date had slipped by several years. Every airplane passenger who is delayed because of gridlocked skyways still feels this cancellation; the cumulative economic impact of all those delays on just the U.S. airlines (never mind the passengers) approaches $50 billion.

Worldwide, it's hard to say how many software projects fail or how much money is wasted as a result. If you define failure as the total abandonment of a project before or shortly after it is delivered, and if you accept a conservative failure rate of 5 percent, then billions of dollars are wasted each year on bad software.

For example, in 2004, the U.S. government spent $60 billion on software (not counting the embedded software in weapons systems); a 5 percent failure rate means $3 billion was probably wasted. However, after several decades as an IT consultant, I am convinced that the failure rate is 15 to 20 percent for projects that have budgets of $10 million or more. Looking at the total investment in new software projects—both government and corporate—over the last five years, I estimate that project failures have likely cost the U.S. economy at least $25 billion and maybe as much as $75 billion.

Of course, that $75 billion doesn't reflect projects that exceed their budgets—which most projects do. Nor does it reflect projects delivered late—which the majority are. It also fails to account for the opportunity costs of having to start over once a project is abandoned or the costs of bug-ridden systems that have to be repeatedly reworked.

Then, too, there's the cost of litigation from irate customers suing suppliers for poorly implemented systems. When you add up all these extra costs, the yearly tab for failed and troubled software conservatively runs somewhere from $60 billion to $70 billion in the United States alone. For that money, you could launch the space shuttle 100 times, build and deploy the entire 24-satellite Global Positioning System, and develop the Boeing 777 from scratch—and still have a few billion left over.

Why do projects fail so often?

Among the most common factors:

- Unrealistic or unarticulated project goals
- Inaccurate estimates of needed resources
- Badly defined system requirements
- Poor reporting of the project's status
- Unmanaged risks
- Poor communication among customers, developers, and users
- Use of immature technology
- Inability to handle the project's complexity
- Sloppy development practices
- Poor project management
- Stakeholder politics
- Commercial pressures

Of course, IT projects rarely fail for just one or two reasons. The FBI's VCF project suffered from many of the problems listed above. Most failures, in fact, can be traced to a combination of technical, project management, and business decisions. Each dimension interacts with the others in complicated ways that exacerbate project risks and problems and increase the likelihood of failure.

Consider a simple software chore: a purchasing system that automates the ordering, billing, and shipping of parts, so that a salesperson can input a customer's order, have it automatically checked against pricing and contract requirements, and arrange to have the parts and invoice sent to the customer from the warehouse.

The requirements for the system specify four basic steps. First, there's the sales process, which creates a bill of sale. That bill is then sent through a legal process, which reviews the contractual terms and conditions of the potential sale and approves them. Third in line is the provision process, which sends out the parts contracted for, followed by the finance process, which sends out an invoice.

Let's say that as the first process, for sales, is being written, the programmers treat every order as if it were placed in the company's main location, even though the company has branches in several states and countries. That mistake, in turn, affects how tax is calculated, what kind of contract is issued, and so on.

The sooner the omission is detected and corrected, the better. It's kind of like knitting a sweater. If you spot a missed stitch right after you make it, you can simply unravel a bit of yarn and move on. But if you don't catch the mistake until the end, you may need to unravel the whole sweater just to redo that one stitch.

If the software coders don't catch their omission until final system testing—or worse, until after the system has been rolled out—the costs incurred to correct the error will likely be many times greater than if they'd caught the mistake while they were still working on the initial sales process.

And unlike a missed stitch in a sweater, this problem is much harder to pinpoint; the programmers will see only that errors are appearing, and these might have several causes. Even after the original error is corrected, they'll need to change other calculations and documentation and then retest every step.

In fact, studies have shown that software specialists spend about 40 to 50 percent of their time on avoidable rework rather than on what they call value-added work, which is basically work that's done right the first time. Once a piece of software makes it into the field, the cost of fixing an error can be 100 times as high as it would have been during the development stage.

If errors abound, then rework can start to swamp a project, like a dinghy in a storm. What's worse, attempts to fix an error often introduce new ones. It's like you're bailing out that dinghy, but you're also creating leaks. If too many errors are produced, the cost and time needed to complete the system become so great that going on doesn't make sense.

In the simplest terms, an IT project usually fails when the rework exceeds the value-added work that's been budgeted for. This is what happened to Sydney Water Corp., the largest water provider in Australia, when it attempted to introduce an automated customer information and billing system in 2002 [see box, "Case Study #2"]. According to an investigation by the Australian Auditor General, among the factors that doomed the project were inadequate planning and specifications, which in turn led to numerous change requests and significant added costs and delays. Sydney Water aborted the project midway, after spending AU $61 million (US $33.2 million).

All of which leads us to the obvious question: why do so many errors occur?

Software project failures have a lot in common with airplane crashes. Just as pilots never intend to crash, software developers don't aim to fail. When a commercial plane crashes, investigators look at many factors, such as the weather, maintenance records, the pilot's disposition and training, and cultural factors within the airline. Similarly, we need to look at the business environment, technical management, project management, and organizational culture to get to the roots of software failures.

Chief among the business factors are competition and the need to cut costs. Increasingly, senior managers expect IT departments to do more with less and do it faster than before; they view software projects not as investments but as pure costs that must be controlled.

Political exigencies can also wreak havoc on an IT project's schedule, cost, and quality. When Denver International Airport attempted to roll out its automated baggage-handling system, state and local political leaders held the project to one unrealistic schedule after another. The failure to deliver the system on time delayed the 1995 opening of the airport (then the largest in the United States), which compounded the financial impact manyfold.

Even after the system was completed, it never worked reliably: it chewed up baggage, and the carts used to shuttle luggage around frequently derailed. Eventually, United Airlines, the airport's main tenant, sued the system contractor, and the episode became a testament to the dangers of political expediency.

A lack of upper-management support can also damn an IT undertaking. This runs the gamut from failing to allocate enough money and manpower to not clearly establishing the IT project's relationship to the organization's business. In 2000, retailer Kmart Corp., in Troy, Mich., launched a $1.4 billion IT modernization effort aimed at linking its sales, marketing, supply, and logistics systems, to better compete with rival Wal-Mart Corp., in Bentonville, Ark. Wal-Mart proved too formidable, though, and 18 months later, cash-strapped Kmart cut back on modernization, writing off the $130 million it had already invested in IT. Four months later, it declared bankruptcy; the company continues to struggle today.

Frequently, IT project managers eager to get funded resort to a form of liar's poker, overpromising what their project will do, how much it will cost, and when it will be completed. Many, if not most, software projects start off with budgets that are too small. When that happens, the developers have to make up for the shortfall somehow, typically by trying to increase productivity, reducing the scope of the effort, or taking risky shortcuts in the review and testing phases. These all increase the likelihood of error and, ultimately, failure.

A state-of-the-art travel reservation system spearheaded by a consortium of Budget Rent-A-Car, Hilton Hotels, Marriott, and AMR, the parent of American Airlines, is a case in point. In 1992, three and a half years and $165 million into the project, the group abandoned it, citing two main reasons: an overly optimistic development schedule and an underestimation of the technical difficulties involved. This was the same group that had earlier built the hugely successful Sabre reservation system, proving that past performance is no guarantee of future results.

After crash investigators consider the weather as a factor in a plane crash, they look at the airplane itself. Was there something in the plane's design that caused the crash? Was it carrying too much weight?

In IT project failures, similar questions invariably come up regarding the project's technical components: the hardware and software used to develop the system and the development practices themselves. Organizations are often seduced by the siren song of the technological imperative—the uncontrollable urge to use the latest technology in hopes of gaining a competitive edge. With technology changing fast and promising fantastic new capabilities, it is easy to succumb. But using immature or untested technology is a sure route to failure.

In 1997, after spending $40 million, the state of Washington shut down an IT project that would have processed driver's licenses and vehicle registrations. Motor vehicle officials admitted that they got caught up in chasing technology instead of concentrating on implementing a system that met their requirements. The IT debacle that brought down FoxMeyer Drug a year earlier also stemmed from adopting a state-of-the-art resource-planning system and then pushing it beyond what it could feasibly do.

A project's sheer size is a fountainhead of failure. Studies indicate that large-scale projects fail three to five times more often than small ones. The larger the project, the more complexity there is in both its static elements (the discrete pieces of software, hardware, and so on) and its dynamic elements (the couplings and interactions among hardware, software, and users; connections to other systems; and so on). Greater complexity increases the possibility of errors, because no one really understands all the interacting parts of the whole or has the ability to test them.

Sobering but true: it's impossible to thoroughly test an IT system of any real size. Roger S. Pressman pointed out in his book Software Engineering, one of the classic texts in the field, that "exhaustive testing presents certain logistical problems....Even a small 100-line program with some nested paths and a single loop executing less than twenty times may require 10 to the power of 14 possible paths to be executed." To test all of those 100 trillion paths, he noted, assuming each could be evaluated in a millisecond, would take 3170 years.

All IT systems are intrinsically fragile. In a large brick building, you'd have to remove hundreds of strategically placed bricks to make a wall collapse. But in a 100 000-line software program, it takes only one or two bad lines to produce major problems. In 1991, a portion of ATandamp;T's telephone network went out, leaving 12 million subscribers without service, all because of a single mistyped character in one line of code.

Sloppy development practices are a rich source of failure, and they can cause errors at any stage of an IT project. To help organizations assess their software-development practices, the U.S. Software Engineering Institute, in Pittsburgh, created the Capability Maturity Model, or CMM. It rates a company's practices against five levels of increasing maturity. Level 1 means the organization is using ad hoc and possibly chaotic development practices. Level 3 means the company has characterized its practices and now understands them. Level 5 means the organization quantitatively understands the variations in the processes and practices it applies.

As of January, nearly 2000 government and commercial organizations had voluntarily reported CMM levels. Over half acknowledged being at either level 1 or 2, 30 percent were at level 3, and only 17 percent had reached level 4 or 5. The percentages are even more dismal when you realize that this is a self-selected group; obviously, companies with the worst IT practices won't subject themselves to a CMM evaluation. (The CMM is being superseded by the CMM-Integration, which aims for a broader assessment of an organization's ability to create software-intensive systems.)

Immature IT practices doomed the U.S. Internal Revenue Service's $4 billion modernization effort in 1997, and they have continued to plague the IRS's current $8 billion modernization. It may just be intrinsically impossible to translate the tax code into software code—tax law is complex and based on often-vague legislation, and it changes all the time. From an IT developer's standpoint, it's a requirements nightmare. But the IRS hasn't been helped by open hostility between in-house and outside programmers, a laughable underestimation of the work involved, and many other bad practices.

THE PILOT'S ACTIONS JUST BEFORE a plane crashes are always of great interest to investigators. That's because the pilot is the ultimate decision-maker, responsible for the safe operation of the craft. Similarly, project managers play a crucial role in software projects and can be a major source of errors that lead to failure.

Back in 1986, the London Stock Exchange decided to automate its system for settling stock transactions. Seven years later, after spending $600 million, it scrapped the Taurus system's development, not only because the design was excessively complex and cumbersome but also because the management of the project was, to use the word of one of its own senior managers, "delusional." As investigations revealed, no one seemed to want to know the true status of the project, even as more and more problems appeared, deadlines were missed, and costs soared [see box, "Case Study #3"].

The most important function of the IT project manager is to allocate resources to various activities. Beyond that, the project manager is responsible for project planning and estimation, control, organization, contract management, quality management, risk management, communications, and human resource management.

Bad decisions by project managers are probably the single greatest cause of software failures today. Poor technical management, by contrast, can lead to technical errors, but those can generally be isolated and fixed. However, a bad project management decision—such as hiring too few programmers or picking the wrong type of contract—can wreak havoc. For example, the developers of the doomed travel reservation system claim that they were hobbled in part by the use of a fixed-price contract. Such a contract assumes that the work will be routine; the reservation system turned out to be anything but.

Project management decisions are often tricky precisely because they involve tradeoffs based on fuzzy or incomplete knowledge. Estimating how much an IT project will cost and how long it will take is as much art as science. The larger or more novel the project, the less accurate the estimates. It's a running joke in the industry that IT project estimates are at best within 25 percent of their true value 75 percent of the time.

There are other ways that poor project management can hasten a software project's demise. A study by the Project Management Institute, in Newton Square, Pa., showed that risk management is the least practiced of all project management disciplines across all industry sectors, and nowhere is it more infrequently applied than in the IT industry. Without effective risk management, software developers have little insight into what may go wrong, why it may go wrong, and what can be done to eliminate or mitigate the risks. Nor is there a way to determine what risks are acceptable, in turn making project decisions regarding tradeoffs almost impossible.

Poor project management takes many other forms, including bad communication, which creates an inhospitable atmosphere that increases turnover; not investing in staff training; and not reviewing the project's progress at regular intervals. Any of these can help derail a software project.

The last area that investigators look into after a plane crash is the organizational environment. Does the airline have a strong safety culture, or does it emphasize meeting the flight schedule above all? In IT projects, an organization that values openness, honesty, communication, and collaboration is more apt to find and resolve mistakes early enough that rework doesn't become overwhelming.

If there's a theme that runs through the tortured history of bad software, it's a failure to confront reality. On numerous occasions, the U.S. Department of Justice's inspector general, an outside panel of experts, and others told the head of the FBI that the VCF system was impossible as defined, and yet the project continued anyway. The same attitudes existed among those responsible for the travel reservation system, the London Stock Exchange's Taurus system, and the FAA's air-traffic-control project—all indicative of organizational cultures driven by fear and arrogance.

A recent report by the National Audit Office in the UK found numerous cases of government IT projects' being recommended not to go forward yet continuing anyway. The UK even has a government department charged with preventing IT failures, but as the report noted, more than half of the agencies the department oversees routinely ignore its advice. I call this type of behavior irrational project escalation—the inability to stop a project even after it's obvious that the likelihood of success is rapidly approaching zero. Sadly, such behavior is in no way unique.

In the final analysis , big software failures tend to resemble the worst conceivable airplane crash, where the pilot was inexperienced but exceedingly rash, flew into an ice storm in an untested aircraft, and worked for an airline that gave lip service to safety while cutting back on training and maintenance. If you read the investigator's report afterward, you'd be shaking your head and asking, "Wasn't such a crash inevitable?"

So, too, the reasons that software projects fail are well known and have been amply documented in countless articles, reports, and books [see sidebar, To Probe Further]. And yet, failures, near-failures, and plain old bad software continue to plague us, while practices known to avert mistakes are shunned. It would appear that getting quality software on time and within budget is not an urgent priority at most organizations.

It didn't seem to be at Oxford Health Plans Inc., in Trumbull, Conn., in 1997. The company's automated billing system was vital to its bottom line, and yet senior managers there were more interested in expanding Oxford's business than in ensuring that its billing system could meet its current needs [see box, "Case Study #1"]. Even as problems arose, such as invoices' being sent out months late, managers paid little attention. When the billing system effectively collapsed, the company lost tens of millions of dollars, and its stock dropped from $68 to $26 per share in one day, wiping out $3.4 billion in corporate value. Shareholders brought lawsuits, and several government agencies investigated the company, which was eventually fined $3 million for regulatory violations.

Even organizations that get burned by bad software experiences seem unable or unwilling to learn from their mistakes. In a 2000 report, the U.S. Defense Science Board, an advisory body to the Department of Defense, noted that various studies commissioned by the DOD had made 134 recommendations for improving its software development, but only 21 of those recommendations had been acted on. The other 113 were still valid, the board noted, but were being ignored, even as the DOD complained about the poor state of defense software development!

Some organizations do care about software quality, as the experience of the software development firm Praxis High Integrity Systems, in Bath, England, proves. Praxis demands that its customers be committed to the project, not only financially, but as active participants in the IT system's creation. The company also spends a tremendous amount of time understanding and defining the customer's requirements, and it challenges customers to explain what they want and why. Before a single line of code is written, both the customer and Praxis agree on what is desired, what is feasible, and what risks are involved, given the available resources.

After that, Praxis applies a rigorous development approach that limits the number of errors. One of the great advantages of this model is that it filters out the many would-be clients unwilling to accept the responsibility of articulating their IT requirements and spending the time and money to implement them properly. [See "The Exterminators (/computing/software/the-exterminators)," in this issue.]

Some level of software failure will always be with us. Indeed, we need true failures—as opposed to avoidable blunders—to keep making technical and economic progress. But too many of the failures that occur today are avoidable. And as our society comes to rely on IT systems that are ever larger, more integrated, and more expensive, the cost of failure may become disastrously high.

Even now, it's possible to take bets on where the next great software debacle will occur. One of my leading candidates is the IT systems that will result from the U.S. government's American Health Information Community, a public-private collaboration that seeks to define data standards for electronic medical records. The idea is that once standards are defined, IT systems will be built to let medical professionals across the country enter patient records digitally, giving doctors, hospitals, insurers, and other health-care specialists instant access to a patient's complete medical history. Health-care experts believe such a system of systems will improve patient care, cut costs by an estimated $78 billion per year, and reduce medical errors, saving tens of thousands of lives.

But this approach is a mere pipe dream if software practices and failure rates remain as they are today. Even by the most optimistic estimates, to create an electronic medical record system will require 10 years of effort, $320 billion in development costs, and $20 billion per year in operating expenses—assuming that there are no failures, overruns, schedule slips, security issues, or shoddy software. This is hardly a realistic scenario, especially because most IT experts consider the medical community to be the least computer-savvy of all professional enterprises.

Patients and taxpayers will ultimately pay the price for the development, or the failure, of boondoggles like this. Given today's IT practices, failure is a distinct possibility, and it would be a loss of unprecedented magnitude. But then, countries throughout the world are contemplating or already at work on many initiatives of similar size and impact—in aviation, national security, and the military, among other arenas.

Like electricity, water, transportation, and other critical parts of our infrastructure, IT is fast becoming intrinsic to our daily existence. In a few decades, a large-scale IT failure will become more than just an expensive inconvenience: it will put our way of life at risk. In the absence of the kind of industrywide changes that will mitigate software failures, how much of our future are we willing to gamble on these enormously costly and complex systems?

We already know how to do software well. It may finally be time to act on what we know.

About the Author

Robert N. Charette is president of ITABHI Corp., a risk-management consultancy in Spotsylvania, Va. An IEEE member, he is the author of several books on risk management and chair of the ISO/IEEE committee revising the 16085 standard on software and systems engineering risk management.