Computing (2022) 104:73–93
https://doi.org/10.1007/s00607-020-00895-6

SURVEY ARTICLE

Tutorial on systems with antifragility to downtime

Kjell Jørgen Hole[1]

Received: 5 May 2020 / Accepted: 22 December 2020 / Published online: 7 January 2021
© The Author(s) 2021

Abstract
An antifragile system of software and stakeholders, including designers, developers, and operators, learn from incidents how to avoid outages and maintain high uptime. This tutorial article reviews how to design and operate such socio-technical systems with antifragility to downtime. It documents the importance of four design principles and two operational principles by exploring the polar opposite anti-principles and the interplay between the principles and the anti-principles. The design principles mandate a software design of separate and isolatable processes with sufficient diversity and redundancy. The processes should communicate asynchronously over an external network. The operational principles imply that the software development teams should repeatedly inject artificial failures into the production system to understand its behavior and detect and mitigate vulnerabilities as the system and its environment change.

Keywords Antifragility · Distributed systems · Design principles · Uptime

Mathematics Subject Classification 68-01, 68M14, 68M15

1 Introduction

In the context of socio-technical systems of software and stakeholders, a *principle* is a fundamental proposition or rule for stakeholders to create or manage the software such that it behaves according to specifications. The opposite of a principle is an *anti-principle*, which leads to misbehaving software. This tutorial-style review studies four design principles and two operational principles for software systems with high uptime. Rather than focusing primarily on the principles, the article mostly considers the corresponding anti-principles. There is a rewarding interplay between the six principles and their polar opposite anti-principles. When we study both principles and

✉ Kjell Jørgen Hole
hole@simula.no

[1] Simula UiB, Thormøhlens gate 53D, 5006 Bergen, Norway

anti-principles, it becomes evident how stakeholders can design and operate software systems with high uptime.

The tutorial considers how to avoid downtime in socio-technical systems due to natural incidents like software bugs, design flaws, configuration mistakes, and hardware failures. A socio-technical system is fragile, robust, or antifragile to downtime caused by these natural events [1,2]. The discussed anti-principles lead to fragility, while the opposite principles provide antifragility to downtime. The study of the anti-principles strengthens earlier claims that the principles are fundamental to achieving high uptime in socio-technical systems [2]. The principles are often called patterns in computer science. Although most of the principles or patterns are well known to the software community, it is much less known how to combine them to create antifragility to downtime.

The design and operational principles lead to distributed software systems proactively maintained by their stakeholders. The design principles mandate a software system of separate and isolatable processes with adequate redundancy and diversity. The processes should communicate over an external network using asynchronous communication. The operational principles imply that engineering teams should regularly induce failures in a production system to learn how to improve it and maintain a low downtime. Together, the principles and the polar opposite anti-principles define a technology-agnostic foundation for the design and operation of socio-technical systems with antifragility to downtime. Although programming languages, development techniques, and database technologies change, this foundation should be stable in the foreseeable future.

The literature discusses many types of antifragility in, for example, cybersecurity [3–5], software [6–8], complex adaptive systems [9–11] and biology [12]. However, the current tutorial focuses solely on avoiding downtime in socio-technical systems that change over time. The primary audience is students, practitioners, and computer scientists wanting to learn about adaptive systems with high uptime. (If you are an experienced Erlang programmer or an expert on microservices, the article may not be for you). The author searched extensively for well-written books, review articles, and tutorials on antifragility and related topics such as the design of distributed systems, parallel programming, systemic risk, and complex adaptive systems. Together, the cited works and their references constitute a starting point for readers to learn more about creating, maintaining, and analyzing antifragile systems.

The organization of the tutorial is as follows. Section 2 provides the system view used in the article. Section 3 introduces the anti-principles, and Sect. 4 discusses the polar opposite principles. A case study in Sect. 5 illustrates how to use both the anti-principles and the principles. Section 6 introduces three design choices and one operational choice obtained by applying the principles to distributed software systems of separate processes. Finally, Sect. 7 examines when and how to create socio-technical systems with antifragility to downtime and discusses a possible need for more principles.

2 System view

This section sets the stage for the discussion of the anti-principles. It introduces two classes of software designs; discusses vulnerabilities and failures in software systems; defines fragile, robust, and antifragile socio-technical systems; and examines the need for system monitoring to learn about a system's behavior.

2.1 Software designs

A design defines a software system's components, interfaces, data formats, data flows, and storage solutions. While software systems consist of both hardware and software, we mostly consider software design. A typical *monolithic* software system has a layered composition with at least three layers that handle user interaction, business logic, and data storage. Each layer contains units that provide the functionality of the system. The units can be subroutines, functions, or classes. They are compiled and combined into a single executable file, resulting in a deployment monolith independent of other applications. If a system needs to support many simultaneous users, it deploys one or more load balancers that distribute user requests to multiple servers (physical machines), each running a copy of the executable.

A *distributed* software system consists of modules, where each module contains a cohesive set of units. Separate processes realize the functionality of one or more modules in a distributed system. Each process is a set of operations directed toward a goal. An active process executes its operations; else, it is inactive. A process may store data during inactive periods. The processes run on multiple servers and communicate over an external network to complete various tasks. Each process decides how to respond to a request from another process, including ignoring it altogether. Since processes are autonomous decision-makers that cooperate voluntarily, no process can force another process to do anything [13].

A system with millions of users may have thousands of processes that run on many servers. The processes can even move between machines during system operation. A process exists until it crashes or the system terminates it. There are different types of distributed systems with various kinds of processes [14], including Erlang processes [15,16] and microservices [17–22]. Here, we focus on the fundamental properties of processes and their communication in large distributed systems.

2.2 Vulnerabilities and failures

A *vulnerability* in a software system is a design flaw, an implementation bug, a mistake in the system configuration, or a hardware malfunction. Experience shows that all large software systems have vulnerabilities that lead to failures. A local failure partly damages a system's functionality, while a system failure destroys (nearly) all functionality. We must prevent inevitable local failures from propagating and causing intolerable system failures. In this paper, we consider how to avoid system downtime due to natural incidents caused by vulnerabilities. Two earlier papers [5,23] discuss how to prevent downtime due to attackers that intentionally exploit vulnerabilities.

Distributed software systems tend to fail in unclear and surprising ways [24]. A system failure can be due to a vulnerability in a single process or multiple processes failing at the same time because they share a common vulnerability. However, a system can fail without there being anything wrong with its processes per se. To see why we view a process as a function that maps an input vector to an output vector. We define the function for a set of input vectors and a set of output vectors.

A vector outside the defined set of input vectors can occur in a system because network problems change the communication. When the input vector to a process is a combination of output vectors from other processes, an unusual combination—not foreseen by the system designers—can also result in a vector outside the input vectors set. Processes that receive input vectors they were not designed to handle may very well generate vectors outside the defined set of output vectors. These surprising output vectors can lead to extreme system behavior, including prolonged downtime.

2.3 Fragility, robustness, and antifragility

According to the established view, the opposite of a fragile system is a robust system. While stressors and perturbations easily damage fragile systems, robust systems tolerate rough treatment (up to a point). In 2012, Nassim N. Taleb published his groundbreaking book [1] *Antifragility: Things that Gain from Disorder*, pointing out that the opposite of a fragile system is a system which needs stressors to thrive. Unlike robust systems, antifragile systems learn from failures how to adjust themselves to limit the impact of future failures and become stronger in a continually changing environment. The human immune system is an example of an antifragile system, as it becomes stronger from regular exposure to germs.

Presently, we do not know how to make software systems that learn reliably without human intervention. Therefore, it is necessary to involve at least software developers and system operators to learn from failures. We model an antifragile system that consists of a distributed software system and stakeholders with significant interests in the system outcome as a complex adaptive system [2, Sec. 1.1], [25–27]. Since the collaboration between the many entities in this socio-technical system can cause undesirable emergent behavior, the entities must adapt to each other and the environment to allow the system to survive failures with potentially substantial negative impact.

In practice, complex adaptive systems of software and stakeholders have a varying degree of antifragility. Since it is not possible to foresee all rare, large-impact incidents in complex adaptive systems, antifragile systems must be able to restrict the adverse effects of events with unknown causes [2, Ch. 2]. Furthermore, systems have to become robust before they can become antifragile. Finally, no system can be antifragile to the impacts of all kinds of incidents. It is essential to understand what types of impacts are intolerable for a particular system and then design it to limit these effects.

The author's book *Anti-fragile ICT Systems* discusses antifragility to malware spreading [2, Ch. 8–10]. Here, we focus on making a socio-technical system antifragile to downtime. It is not enough to develop a system that is robust to known failures to achieve high uptime. Such a system gradually becomes more fragile to downtime as the system and its environment change in unpredictable ways. Fragility inevitably

accumulates below the surface of any robust system, and ultimately the system goes down, perhaps for a long time [24]. An antifragile socio-technical system must limit the impact of all failures with the potential to reduce the uptime. It is vital that developers and operators continuously learn from small-impact failures how to change and improve the technical system to keep it running.

2.4 System monitoring

It is necessary to monitor a system's behavior to achieve antifragility to downtime. In particular, it is crucial to detect local failures early, preferably before they spread and cause system failures. To learn what went wrong when a monolith misbehaves or crashes, software developers study data logs and the single executable's source code. It is often not clear what code to analyze when a distributed system fails. The software consists of many separate processes that run on multiple servers and communicate in convoluted ways over an external network. Thus, sophisticated real-time monitoring of both individual processes and their interplay is needed to detect unusual or undesirable behaviors.

It is not enough to know that a process is alive. We also need to verify that the process is responding the way it should. The monitoring software can test processes, or the processes can report to the monitoring software regularly. It must also be possible to track requests as they move through a system. Note that system monitoring, learning from failures, and mitigating problems is not only a technical challenge. It requires the right organizational structures and support from management to ensure that developers and operators learn about problems early and fix them quickly [17–22].

3 Anti-principles

The author has previously discussed the four design principles and the first operational principle listed in the left column of Table 1 for socio-technical systems with antifragility to downtime [2, Ch. 4]. The table introduces one additional operational principle in the last row. Rather than directly arguing the importance of the six principles, this section provides a general discussion of the fragility to downtime caused by applying the polar opposite anti-principles in the table's right column to sizeable software systems. The next section uses the anti-principles to explain the importance of the listed principles.

3.1 Monolith

Software monoliths are simple to develop, test, and deploy, and they scale by running multiple copies behind a load balancer. However, while massive monoliths with high uptime exist, a monolith's single executable incurs fragility of downtime. A local failure caused by an internal mistake in a unit (for example, a memory leak) or an abnormal interaction between two units is likely to halt accurate data processing and terminate the executable. While we can reduce the fragility to downtime by running

several executables, significant fragility remains if the instances contain the same vulnerabilities. Since any substantial application includes vulnerabilities, a massive deployment monolith is fragile to downtime. It is not a recommended design choice when a very high uptime is essential to a software system's long-term success.

3.2 Inseparable

Consider a distributed software system containing two communicating processes A and B. If A's functionality significantly degrades when B stops working correctly, then A is strongly dependent on B. To illustrate, let the process A depend on the functionality of the process B to produce output data from input data. If A generates incorrect output data when B misbehaves, then A is strongly dependent on B. If a system contains processes that are strongly dependent on a process B, then it is hard to isolate B by taking down its connections to other processes without severely damaging the system's functionality. When B misbehaves, it can thus adversary affect other processes and eventually take down the whole system. We say that a process B is *inseparable* from the system when other processes strongly depend on B.

There are disadvantages associated with inseparable processes. It is challenging to modify an inseparable process's functionality because it is necessary to change other processes simultaneously. We need to plan and execute a coordinated software release rather than make a code change in a single process. Many inseparable processes in a system lead to infrequent and expensive software updates. Furthermore, it becomes too hard and costly to remove old software technologies. While legacy systems may appear to be stable with long periods of uninterrupted operation, they also tend to experience unplanned downtime due to maintenance problems. These maintenance problems increase over time as it becomes harder to find individuals with intimate knowledge of the deployed legacy technologies.

3.3 Uniformity

In agriculture, a monoculture is a widespread practice of growing a single crop in an area. A continuous monoculture, where the same plant is cultivated year after year, can cause a buildup of pests and diseases to spread rapidly because the whole harvest is susceptible. Computer science has adopted the term monoculture to describe

Table 1 Four design principles and two operational principles to ensure antifragility to downtime, as well as the corresponding anti-principles

	Principle	Anti-principle
Design	Separate processes	Monolith
	Isolatable	Inseparable
	Diversity	Uniformity
	Redundancy	Uniqueness
Operational	Fail fast	Fail slow
	Skin in the game	No skin in the game

networked computers that run the same software. A monoculture could be vulnerable to catastrophic failure of all its machines due to prevalent poor design, software bugs, or misconfigurations. In practice, not all computers that run the same software are vulnerable since they reside on different networks with varying management and router policies. The machines on a particular network also have different configurations and patch levels, making a fraction of the computers immune to the consequences of a universal vulnerability.

A distributed system of networked computers that run a standard operating system, communication protocol, or application software has a degree of *uniformity* when the computers share vulnerabilities. The degree of uniformity is equal to the largest fraction of computers with the same vulnerability. Note that we consider physical machines and not individual processes. The reason is that when multiple processes run on the same machine, they may all fail when one crashes. The degree of uniformity is equal to one when each machine in a monoculture runs an instance of the same vulnerable process. In general, a detailed study of a system is needed to determine the degree of uniformity. The degree varies over time as the system changes. While it is hard to measure the exact degree of uniformity, it should be clear that a highly uniform system is susceptible to failure due to widespread vulnerabilities.

3.4 Uniqueness

A process is unique in a distributed software system if it is the only one providing a particular functionality. A distributed system's degree of *uniqueness* is the fraction of unique processes. A system with the ability to run multiple instances of each process can vary the degree of uniqueness. As we reduce the degree of uniqueness by adding multiple copies of more kinds of processes, the system becomes increasingly redundant. A distributed system loses functionality when a unique process crashes. If a system has many unique processes, some of these processes are likely to be single points of failure that take down the whole system when one fails.

3.5 Fail slow

An antifragile system learns from failures how to adapt and improve over time. While the learning is mostly carried out by humans today, a software system may deploy artificial intelligence in the future. Whether humans or machines learn, it is necessary to detect failures to learn from them. Although it is evident that failure occurs when a process or a whole system crashes, it can be challenging to identify partial failures where processes continue to run, but the system produces delayed or the wrong output data.

Designers and developers make assumptions about how a distributed software system fails; in particular, they make assumptions about how communication delay impacts a system's operation. The assumptions are often incomplete. Latency tends to affect a distributed system in ways that are very hard to foresee. When misbehaving processes are allowed to run for a long time, they can adversarially impact other processes. This failure propagation can create system failures, including prolonged

downtime, with substantial negative impact. We say that a process *fails slowly* when the communication latency is unusually large, or the process generates erroneous output data for a long time without human operators or the software system itself detecting any problems. Slow failures impede learning from mistakes and make a system fragile to unplanned downtime.

3.6 No skin in the game

In a hard-to-understand complex adaptive system like the international financial system, many operators benefit from the upside when the system behaves well without paying for the downside when it misbehaves [28]. Bankers and corporate executives receive bonuses for positive performance but do not have to pay out reverse bonuses for negative performance—only the investors take a loss. This transfer of risk to the customers gives bankers and executives an incentive to hide risk, thereby delaying financial blowups and making them larger. Risk hiding was a primary underlying reason for the financial crisis of 2008 [29–31].

Stakeholders of a software system have *no skin in a game*, or no dog in the fight when they do not share the risk associated with the system. People take less ownership of a system and its behavior when they have no skin in the game. In particular, a team of software developers without any responsibility for system failures and the resulting downtime does not take extra care to avoid problems. It does not spend time learning about the system behavior to detect and rectify vulnerabilities as the system and its environment change. Instead, the team ignores technical debt and starts to add new functionality to the system. Over time, the tendency to ignore technical debt while adding complexity leads to fragility to downtime.

To better understand the consequences of the anti-principle of no skin in the game, we study an example. Any shoddy code in a software product may expose users to risk unknown to them. As a thought experiment, consider when a programming mistake causes a popular spreadsheet program to make wrong calculations on rare occasions. A vast number of users and the complicated spreadsheet models used by businesses make it likely that the financial damage is substantial before the mistake is detected and corrected. Some users suffer considerably, but the cost to the software company that made the spreadsheet program is limited since it has no liability for its product under current laws. In other words, the company has little or no skin in the game.

4 From anti-principles to principles

This section introduces a six-step procedure consisting of four steps to create a technical system of software and hardware with robustness to downtime, followed by two additional steps to develop an antifragile socio-technical system, where stakeholders maintain and improve the technical system over time. The procedure uses the discussed anti-principles in Table 1 to provide general explanations for why the tabulated opposite principles are necessary to design and operate antifragile systems. Since

companies tend to build software monoliths that they later transform into distributed software systems [22], the procedure assumes a monolith exists.

4.1 Design principles

A monolithic software system with a single executable is likely to go down when one of its units experiences a problem. Since all sizeable monoliths contain vulnerabilities, they are fragile to downtime. The **first step** to increase uptime is to divide the functionality of a monolith into *separate processes* that run on a set of networked machines. This system must have at least two physical computers to observe each other's activity and restart crashed processes. If there is only one machine, it is impossible to reboot a process that collapsed due to a hardware or power failure. When there is a second machine, it can run the crashed process. If one computer goes down and all its processes die, another computer can restart these processes and run them until the crashed computer recovers.

Since each separate process consists of tightly integrated units, a whole process goes down when one of its units develops a problem. If the processes are inseparable, then the entire system stops functioning correctly. The **second step** to reduce downtime is, therefore, to prevent a process with a problem from negatively affecting other processes. Consider two communicating processes A and B. If the process A's functionality is nearly unaffected when process B malfunctions, then A is weakly dependent on B. When all processes that interact with B are weakly dependent on B, then B is *isolatable*. All processes in a distributed system should be isolatable to prevent any process with a problem from taking down the whole system.

The **third step** toward robustness to downtime is to avoid a distributed system with a high degree of uniformity, that is, a system whose servers run software with the same vulnerability. A widespread weakness can cause many processes to crash at the same time. Software *diversity* alleviates general vulnerabilities. We can use compilers with "diversity engines" to add diversity during the compilation of the processes and the underlying software of a distributed system [32,33], [2, Ch. 8–10]. The servers can also run different versions or even completely different implementations of the processes.

The **fourth** and last **step** to ensure robustness to downtime is to avoid a distributed system with a high degree of uniqueness. Such a system has many processes with unique functionality. Some of these processes are likely to be single points of failure that take down the whole system when they fail. System designers must avoid single points of failure, and system operators must introduce *redundancy* by running multiple instances of processes to alleviate the uniqueness problem. Software developers often realize processes as virtual machines or (Docker) containers, which are isolated and replaced when they misbehave. The use of virtualization makes it economically viable to deploy redundant and diverse services to limit the impact of failures.

4.2 Operational principles

Software systems with large-scale distributed designs have hardware and software components that fail all the time [21]. It is impossible to avoid these failures, especially when a system changes daily. The only real choice is whether a distributed system should fail fast or slow. A monolith fails quickly when a unit fails because the application runs as a single executable. Hence, the tendency to fail fast is a problem when we consider uptime of monoliths.

The situation is the opposite for distributed systems of separate and isolatable processes. The **fifth step** mandates that individual processes *fail fast* to facilitate the detection of failures, isolation of the processes, and activation of countermeasures to limit further the damage, such as default replies when processes are isolated. Increasingly slow failure detection, or even worse silent failures of processes, complicates stakeholders' learning about the system behavior and lead to failure propagation with growing consequences. Note that while the individual processes should fail fast, the overall system should, ideally, not fail at all.

A software system designed to be robust to downtime caused by known failures becomes fragile over time as the system and its environment change in unpredictable ways. Software developers and system operators must learn about new vulnerabilities and improve the system to maintain robustness to downtime. Since it is impossible to predict all future failures [2, Ch. 2], the system must limit the impact of failures with unknown origins. It is necessary to focus on preventing different types of damage rather than describing all possible failure scenarios and their initiating causes.

The **sixth step** requires that developers and operators share the responsibility for downtime and remove vulnerabilities; that is, they must have *skin in the game* [28,34,35]. Software developers creating a system should be responsible for mitigating problems with their code, and operators should make sure that the system runs without severe hiccups. The stakeholders should have operational responsibilities not to punish them when things go wrong, but to make sure that they learn from mistakes how to maintain and improve the system. The stakeholders must continue to mitigate weaknesses as the system and its environment change to achieve antifragility to downtime.

4.3 Overview of principles

In summary, the four design principles of *separate processes*, *isolatable*, *diversity*, and *redundancy* outline how to create a distributed software system that is robust to downtime. The two operational principles of *fail fast* and *skin in the game* describe how software developers and system operators can make the same system antifragile to downtime by continuously learning how to improve the system. The quality and speed of the never-ending learning process determine the degree of antifragility.

In addition to the logical justifications for the six principles, empirical evidence supports these propositions' importance. Netflix has applied the fundamental ideas to develop and operate a microservice-based media streaming solution in the cloud with antifragility to downtime [36–38]. Later, the paper discusses tools and techniques

developed by Netflix to learn about vulnerabilities from injected failures in a production system. There is also evidence that other companies, including Google, Microsoft, LinkedIn, Slack, and Uber, have implemented the discussed principles to attain high uptime [21,39,40]. The following case study illustrates how the (anti-) principles could be used.

5 Case study: e-government system

The principles and anti-principles in Table 1 guide the design and operation of socio-technical systems with antifragility to downtime. Here, we apply the anti-principles to determine fragilities in an electronic government (e-government) system and the polar opposite principles to make it antifragile [2]. The case study illustrates the advantage of considering both anti-principles and principles to attain antifragility. The discussion is partly inspired by two analyses of the Norwegian e-government platform Altinn as it appeared in 2012 [41,42]. The author makes no claims about Altinn's design after 2012.

5.1 Fragility and robustness to downtime

Figure 1 models the architecture of an e-government system with services developed by governmental entities to serve commercial companies and private citizens. The model contains identical copies of a *monolith*, where each executable runs on a separate server. While the use of several servers with identical executables introduces *redundancy*, the degree of *uniformity* is equal to one when the executables share at least one vulnerability. The *unique* load balancer in Fig. 1 is a single point of failure.

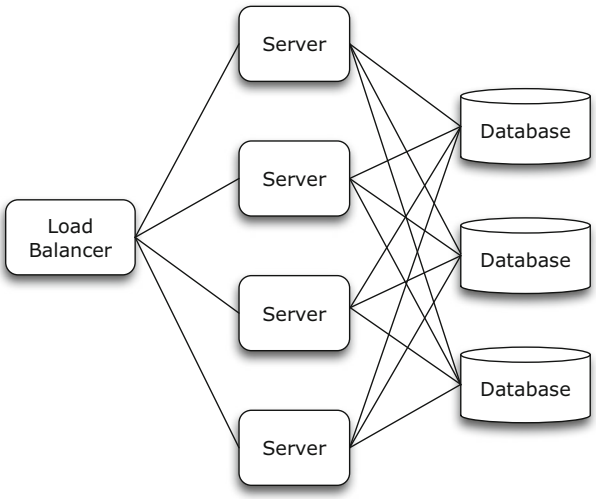

Fig. 1 E-government architecture

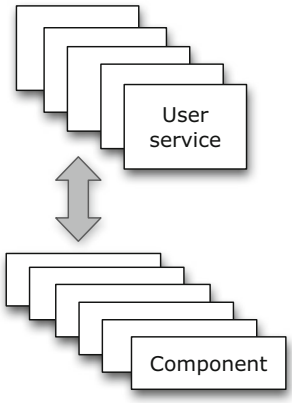

Fig. 2 Layers of user services and components

To increase the system's robustness to downtime, we divide the monolithic application into *separate processes* and then distribute the processes over the available servers. The distributed system achieves *redundancy* by running multiple instances of each process as needed, and *diversity* using "diversity engines" at compile-time, or by allowing the servers to run different versions, or even radically different implementations, of the processes. The clients communicate with the distributed system via dedicated edge services running on the servers. The clients must know how to contact these edge services and how to switch to another should one service fail [13, Ch. 7].

To better understand why *inseparable* processes constitute a problem, we return to the 2012 Altinn platform. As illustrated in Fig. 2, Altinn consisted of user services that ran on top of standardized components. Various government entities developed the services, while Altinn provided the components. Any service that no longer functions when a component crashes is strongly dependent on the component. Since multiple user services depended on the same standardized component in Altinn, it was hard to isolate a misbehaving component to shield the services from a component problem.

When components on different servers share databases, as illustrated in Fig. 1, there are high dependencies between the services and the databases accessed via the components. If a database goes down, all the dependent services stop working. Furthermore, when a service A depends on another service B to update information in a database, then A is strongly dependent on B. Strong dependencies make it hard to isolate services and lead to failure propagation and system failures. We need *isolatable* components and services. In the next section, we discuss how to avoid strong dependencies by exclusively letting processes own their data.

5.2 Antifragility to downtime

Classical testing of new software functionality mostly occurs before it is added to the production system, while stakeholders of an antifragile system must continuously test the production system itself. Even with classical testing, it is more challenging to test an e-government platform with many national services than to test a single enterprise application. The Altinn organization did not have an acceptable test environment and

test procedures in 2012 [41,42]. The testing was unacceptable in all phases of the development process, and the ability to correct discovered mistakes was limited. While Altinn tested the components, the service owners tested the services developed on top of the components. The testing tools available to the service owners were inadequate. Because of insufficient testing, many bugs were not detected in the code before it went into production. Due to budgetary constraints, Altinn did not correct many known defects in the production code [41].

The Altinn experience demonstrates the need to learn from comprehensive and repeated testing before and after code is in production. Learning from one's own and others' failures requires an open and blame-free culture, where developers and operators responsible for vulnerabilities are encouraged to explain why and how these weaknesses occur and how they were mitigated. Parts of the software industry have embraced DevOps, a methodology combining Development and Operations to facilitate an open and blame-free environment where building, testing, and releasing software happens rapidly, frequently, and more reliably than with previous methodologies. DevOps teams creating and operating an e-government platform must focus on testing to achieve antifragility to downtime. The teams must realize the *fail fast* operational principle to detect problems rapidly and have *skin in the game* [26] to learn from the issues.

6 Design and operational basis

We have argued at length the fundamental importance of the principles *separate processes*, *isolatable*, *diversity*, *redundancy*, *fail fast*, and *skin in the game* in Table 1. Here, we introduce three design choices and one operational choice by applying the principles to distributed software systems with many separate processes that communicate over unreliable networks. The selections form a technology-agnostic design and operational basis for systems with antifragility to downtime.

6.1 Local persistent storage

As discussed earlier, when there are weak dependencies between the processes in a distributed system, the processes are isolatable, making it possible to detach a troublesome process from the system. Here, we assume that the system has enough diversity and redundancy to cope with the loss of the process. The e-government case study illustrated that a central database creates strong dependencies between processes in a distributed system. We, therefore, study how to achieve weak dependencies when a system uses persistent storage.

A process with a mutable or changeable state, often referred to as a stateful process, alters information in a database or an in-memory data structure that persists between process activations. Consider a distributed system where multiple processes have states in a central database. If the database goes down, these processes are unable to carry out their tasks. In other words, the processes are strongly dependent on the database. Furthermore, the database is a single point of failure. Finally, a process A is strongly

dependent on a process *B* when *A* depends on *B* to update information in the database. Hence, weakly dependent processes cannot share a database. A similar argument shows that weakly dependent processes cannot share any in-memory data structure. It follows that any stateful process must maintain the state locally and not make it directly available to any other process.

Since a centralized database creates strong dependencies between processes, we need to build distributed systems of weakly dependent processes with local persistent data storage; that is, stateful processes must maintain individual databases even if several processes must store similar data. Processes can store data in flat files, SQL databases, or NoSQL databases, depending on their needs. While processes should never share a database, they may share a distributed and highly redundant cloud-based database solution like the Cassandra column-family store or the Mongo DB document database [19, Ch. 5], given that different processes only access isolated portions of the solution. More information about distributed NoSQL databases is available in the book *NoSQL Distilled* by Pramod J. Sadalage and Martin Fowler [43]. It is particularly important to understand the data consistency issues that emerge when we use distributed database solutions.

6.2 Asynchronous message passing

There is no process in charge of the system behavior in a software system of weakly dependent or isolatable processes. A message from one process to another is not a command but a request for some data or action. The receiving process may or may not produce a reply and send it back to the requesting process. If the receiving process refuses to carry out the request, then there is little the requesting process can do to force the receiving process to fulfill the request. Hence, a fundamental property of a truly distributed system without central control is the need for cooperation between processes.

Thus, to avoid shared mutable state and strong dependencies between processes, they need to exchange messages via application programming interfaces (APIs). When an application's processes have well-defined APIs, we can update the processes independently of each other as long as we do not change the APIs. To change a process without affecting its API, we should hide most of the functionality by making the API small. The messages must have a standard format and contain task-specific content, not arbitrary "objects" to avoid complicated data parsing and processing. The processes may not send pointers to internal functionality or data because this would again create strong dependencies between them.

With *synchronous* communication, a process *A* on a server sends a request to another process *B* that may run on a different server. The request blocks the operation of a process *A* until the process *B* completes its calculations and returns a response to *A*. With *asynchronous* communication, process *A* sends a message stating that something happened and expects process *B* to act on this information. Process *A* does not wait for an answer from *B* but continues to carry out other tasks. Process *A* may receive a response from *B* later, but often no response is expected.

It is easier to reason about synchronous (request/response) communication than asynchronous (event-based) communication because we know if a request received an answer. However, synchronous communication requires highly reliable communication. When processes communicate synchronously over an external network between servers, communication failures cause processes to hang. There may be a significant delay before processes receive a response to a request even without any network failure. During this waiting time, the requesting processes sit idly, wasting computational resources. Hence, synchronous communication creates strong dependencies between processes, while asynchronous communication establishes process boundaries and help to ensure weak dependencies and, thus, isolatable processes [17, Ch. 4], [18, Sec. 9.4].

6.3 Supervisors and workers

Carl Hewitt has created the Actor Model, a mathematical model of concurrent computation that treats concurrent processes, called actors, as the universal primitives. The central ideas of the Actor Model [44–46] allow an antifragile system's processes to fail fast with limited impact. In practice, the processes run on a collection of networked servers. Each server runs an instance of a runtime system. It controls how the processes spawn other processes, assigns processes to a server's processing cores, and determines how the processes send asynchronous messages directly to one another. A process receives messages in a mailbox and processes one message at a time. When a system restarts a process, it need not delete the mailbox, thus, effectively hiding the restart from the other processes. The runtime system supports location transparency. A programmer may not know and do not care about what machine a process is running.

The Actor Model allows any process to generate another process on any server. The creator process is a *supervisor*, and the created process is a *worker*. A supervisor typically produces one or more workers and delegates tasks to them. Spawned workers run independently and concurrently. The supervisor sends asynchronous messages to the workers to complete a task. It also monitors the workers to handle any occurring problems. The supervisor may start a timer when it delegates a job to one of its workers. If a worker does not respond in time, then the supervisor decides how to proceed.

If a worker runs into a problem, it suspends itself and notifies the supervisor about the failure. When the supervisor receives the notification, it may continue without the worker's missing information, restart the suspended worker, move it into a valid state, spawn a new instance of the worker with an empty mailbox, or initiate some other action. A failing worker's ability to notify its supervisor allows processes on one server to monitor processes on different servers and recover from a complete server failure. No matter the underlying reason for a problem, the goal is to prevent a slow or crashed process from taking down the whole system. To avoid a process in trouble from being hammered by incoming messages, a programmer can wrap it in a circuit breaker pattern that stops the flow of requests [47]. Circuit breakers allow for smooth recovery and self-healing, stemming the flow of messages to a failed process to accelerate its recovery process.

In the Actor Model, the supervisors and workers reside in a hierarchy: A worker can spawn other workers. The first worker then becomes the supervisor of the new workers. One of the new workers can again become a supervisor by generating more workers. If a supervisor cannot handle a problem, then it escalates the issue to its supervisor. At the top of the supervisor-worker hierarchy is the runtime system. If the runtime system cannot handle a problem, then the whole system crashes. This hierarchical supervision strategy frees up the workers from handling their problems, which means that developers can focus on business logic and create workers with little or no error handling code.

6.4 Failure injection

Even if risk management of a socio-technical system has allowed stakeholders to avoid failures with severe consequences for a long time, there is a remaining risk of future failures with a sizeable negative impact. It is difficult to quantify this risk because large distributed software systems tend to fail in ways that are hard to foresee [2, Ch. 2], [24]. Furthermore, it is hard to learn from failures when there are only a few minor errors between rare significant malfunctions [48]. The operational principle of fail fast suggests that we can alleviate these problems by regularly injecting artificial failures into a system to test it. This failure injection should be done by stakeholders with skin in the game to ensure adequate learning. Sophisticated system monitoring is essential to detect and limit the consequences of injecting these artificial failures.

The customer-facing software systems belonging to Netflix, Facebook, Amazon, and Google require very high uptime. Stakeholders continuously mitigate problems caused by software and hardware failures, network latencies, configuration errors, and power outages. We focus on Netflix's cloud-based system that streams movies and television shows over the internet to millions of viewers. This distributed system has problematic behaviors and failure modes. It is impossible to predict all possible incidents with serious negative consequences because of the system's very many interacting components.

To uncover and rectify weaknesses, Netflix invented *Chaos Engineering* [36–38], a set of fault-injection tools and techniques to experiment on its system (https://principlesofchaos.org). Examples of events injected by the tools are transient network failures, surges in incoming requests, and malformed data inputs. Netflix's engineers use the tools to run fault-injection experiments to find design flaws, implementation bugs, and configuration errors before real system incidents occur. Thus, Chaos Engineering is an approach for learning about system behavior by applying empirical exploration. Unlike mere testing of a specific condition, Chaos Engineering generates new knowledge about the system.

Before running an experiment, the responsible engineering team defines expected behavior, denoted "steady state," as a measurable output of a system. The team hypothesizes that this state continues during the experiment. The manual or automated experiment then induces some adversarial events and collects observations about system behavior. Surprising and unwanted behavior disproves the steady-state hypothesis

and provides evidence that there exist one or more system vulnerabilities. The team acts on this information to mitigate undesirable behavior.

Netflix's system contains many microservices that run in a cloud divided into regions, each with multiple data centers. It is necessary to run experiments in this production system since it is impossible to reproduce all aspects of a distributed system within a test environment. The company initially created the tool *Chaos Monkey* to run experiments. It closes down randomly selected microservices to ensure that the production system tolerates this frequent failure without the customers experiencing problems.

The newer tool *Chaos Kong* simulates the outage of an entire region with several data centers, verifying the system's ability to transition service from one region to another. Netflix does not take a whole region offline. Instead, the system assumes that a region has gone offline and starts directing client requests to an alternative region. Netflix has many other tools. During these tools' design, the engineers considered tradeoffs between realistic events and potential harm to users. Furthermore, the engineers usually apply a tool to a subset of users to limit the impact if something goes wrong.

To create failure injection experiments, the Netflix tool Failure Injection Testing (FIT) changes headers of requests at the edge of the microservice system. FIT adds a failure scenario to a percentage of the headers of a class of requests. When the modified requests move through the system, injection points between the microservices check for the failure scenario and act accordingly. FIT covers the impact space between the small failures generated by Chaos Monkey and the massive failures simulated by Chaos Kong. In 2016, Netflix launched the Chaos Automation Platform (ChAP) to run tool-based experiments automatically. To examine a microservice, ChAP launches experiment and control clusters of the service and applies a FIT scenario to the experimental group. ChAP then compares real-time metrics from the two groups to determine the impact of the FIT scenario.

Chaos Engineering allows stakeholders to learn about a technical system's behavior and build confidence in its ability to withstand adversarial events. Netflix's experience strongly suggests that comprehensive and frequent Chaos Engineering detects vulnerabilities, verifies robustness, and leads to antifragility. Problems must be removed promptly, for example, by adding countermeasures to limit the impact of future incidents. Although Chaos Engineering originated at Netflix, it has spread to other companies and may become a regular part of site reliability engineering [39,40].

7 Discussion

Any complex software system fails because there is always something that goes wrong, from self-inflicted outages caused by buggy processes or operators that misconfigure a system to incidents outside the control of the system operators like denial-of-service attacks or network failures. Furthermore, no matter how hard software developers try, they cannot build flawless applications. To ensure high uptime, we used known design and operational principles to develop a basis for socio-technical systems with antifragility to downtime. The modeled socio-technical solutions consisted of stakeholders and technical systems of software and hardware. This final section discusses

when and how to create such antifragile systems. It also considers the possible need for more principles.

7.1 When to build antifragile systems

After studying this paper, the reader may be under the impression that monolithic applications are always inferior to distributed systems. This view is wrong—monoliths are perfectly acceptable solutions in many cases. We should only create a distributed software system when a monolith would struggle to satisfy stated availability, performance, or scaling requirements [49]. The author focused on the six design and operational principles in Table 1 precisely because they support Internet-scale solutions, preferably implemented on cloud infrastructures with support for virtualization and distributed storage [2].

Not all socio-technical systems can be made antifragile to downtime. An antifragile solution requires a technical system with highly distributed architecture based on some variation of the Actor Model [44–46], where numerous processes run in parallel on multiple (multicore) machines. Many programs would slow down if we tried to parallelize them. A reader interested in designing distributed systems should study Domain-Driven Design [46,50] to understand better how to divide a solution into processes. The many books on microservices [17–22] describe how to realize these processes. In practice, companies often build monoliths that they later transform into microservice solutions when the need arises [22].

Since it is impossible to predict rare, consequential incidents in complex distributed systems, an organization wanting to achieve antifragility to downtime must develop skills to run failure-inducing processes to surface vulnerabilities. At the time of this writing, Chaos Engineering is the state-of-the-art approach. Books [36–38,40] and conference talks (on Youtube) document that companies embrace Chaos Engineering to reduce system downtime. Readers wanting to operate antifragile systems should study Chaos Engineering [36–38,40].

We can use numerous programming languages to create software that supports downtime antifragility, but some languages are particularly well suited to the task. The functional programming language Erlang and its runtime system [15,16] support the outlined design and operational basis for systems with antifragility to downtime. Other programming languages and runtime systems also support the fundament for antifragility. One option is the language Elixir (https://elixir-lang.org) that uses the same runtime system as Erlang. Another possibility is to use one of the languages Java or Scala for the Java Virtual Machine and the actor-based toolkit Akka (http://akka.io).

7.2 More principles?

Many published principles [2,5,7,11,51,52] provide foundations for various aspects of antifragility (see [5,7,11] for examples). Although this paper's author has only discussed principles for antifragility to downtime, the presented principles are valid for other types of antifragility, including antifragility to malware attacks [2]. There is

considerable overlap between the many published principles. Some of them are rather apparent consequences of more fundamental principles. Other propositions sound good, but it is unclear how to realize them. The author chose the basic design and operational propositions that are realizable and do not overlap. The discussion of the polar opposite anti-principles shows that the selected principles are necessary, but perhaps not sufficient, to achieve antifragility to downtime. Antifragility in socio-technical systems is an active area of research. Future work may determine more non-overlapping principles that clarify how to design or operate systems with different antifragility types.

Compliance with ethical standards

Conflict of interest The author declares that he has no conflict of interest.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Taleb NN (2012) Antifragile: things that gain from disorder. Random House, New York
2. Hole KJ (2016) Anti-fragile ICT systems. Springer
3. Uzunov AV, Nepal S, Chhetri MB (2019) Proactive antifragility: a new paradigm for next-generation cyber defence at the edge. In: Proceedings of the IEEE 5th international conference on collaboration and internet computing (CIC), Los Angeles, CA, USA, pp 246–255
4. Chhetri MB, Uzunov AV, Vo QB, Nepal S, Kowalczyk R (2019) Self-improving autonomic systems for antifragile cyber defence: challenges and opportunities. In: Proceedings of the IEEE international conference on autonomic computing (ICAC), Umeå, Sweden, pp 18–23
5. Køien GM (2020) A philosophy of security architecture design. Wirel Person Commun 113:1615–1639
6. Monperrus M (2015) Software that learns from its own failures. Technical report. arXiv:1502.00821
7. Monperrus M (2017) Principles of antifragile software. In: Proceedings of Salon des Refusés
8. Russo D, Ciancarini P (2017) Towards antifragile software architectures. Procedia Comput Sci 109:929–934
9. Tolk A, Johnson IV JJ (2013) Implementing antifragiles: systems that get better under change. In: Proceedings of the 34th international annual conference of the american society for engineering management (ASEM), Minneapolis, Minnesota, USA, pp 118–226
10. Bakhouya M, Gaber J (2015) Approaches for engineering adaptive systems in ubiquitous and pervasive environments. J Reliab Intell Environ 1(2–4):75–86
11. Bruijn H, Gröβler A, Videira N, (2019) Antifragility as a design criterion for modelling dynamic systems. Syst Res Behav Sci 37(1):23–37
12. Danchin A, Binder PM, Noria S (2011) Antifragility and tinkering in biology (and in business) flexibility provides an efficient epigenetic way to manage risk. Genes 2(4):998–1016
13. Burgess M (2015) Thinking in promises: designing systems for cooperation. O'Reilly, Sebastopol
14. Steen M, Tanenbaum AS (2016) A brief introduction to distributed systems. Computing 98(10):967–1009

15. Armstrong J (2013) Programming Erlang: software for a concurrent world, 2nd edn. Pragmatic Bookshelf, New York
16. Cesarini F, Vinoski S (2016) Designing for scalability with Erlang/OTP: implement robust, fault-tolerant systems. O'Reilly, Sebastopol
17. Newman S (2015) Building microservices: designing fine-grained systems. O'Reilly, Sebastopol
18. Wolff E (2016) Microservices: flexible software architecture. Addison-Wesley, Boston
19. Nadareishvili I, Mitra R, McLarty M, Amundsen M (2016) Microservice architecture: aligning principles, practices, and culture. O'Reilly, Sebastopol
20. Richardson C, Smith F (2016) Microservices: from design to deployment. NGINX. https://www.nginx.com/resources/library/designing-deploying-microservices
21. Fowler SJ (2016) Production-ready microservices: building standardized systems across an engineering organization. O'Reilly, Sebastopol
22. Newman S (2020) Monolith to microservices: evolutionary patterns to transform your monolith. O'Reilly, Sebastopol
23. Hole KJ, Otterstad C (2019) Software systems with antifragility to downtime. IEEE Comput 52(2):23–31
24. Dekker S (2011) Drift into failure: from hunting broken components to understanding complex systems. CRC Press, Boca Raton
25. Mobus GE, Kalton MC (2015) Principles of system science. Springer, New York
26. Crutchfield JP (2009) The hidden fragility of complex systems—consequences of change, changing consequences. In: Ascione G (ed) Cultures of change: social atoms and electronic lives. Actard Publishers, New York, pp 98–111
27. Helbing D (2009) Systemic risks in society and economics. Available at SSRN. https://ssrn.com/abstract=2413205 or https://doi.org/10.2139/ssrn.2413205
28. Taleb NN, Sandis C (2016) The skin-in-the-game heuristic for protection against tail events. In: DeMartino GF, McCloskey DN (eds) The Oxford handbook of professional economic ethics. Oxford University Press, Oxford
29. Mandelbrot B, Hudson RL (2006) The misbehavior of markets: a fractal view of financial turbulence. Annotated edn. Basic Books, New York
30. Taleb NN (2010) The black swan: the impact of the highly improbable, 2nd edn. Random House, New York
31. Dekker S (2013) Drifting into failure: complexity theory and the management of risk. In: Banerjee S (ed) Chaos and complexity theory for management: nonlinear dynamics. IGI Global, Hershey, pp 241–253
32. Larsen P, Brunthaler S, Franz M (2015) Automatic software diversity. IEEE Secur Priv 13(2):30–37
33. Larsen P, Brunthaler S, Davi L, Sadeghi AR, Franz M (2015) Automated software diversity. Morgan and Claypool, San Rafael
34. Safire W (2006) Skin in the game. The New York Times Magazine, 17 Sept 2006
35. Taleb NN (2018) Skin in the game: hidden asymmetries in daily life. Random House, New York
36. Basiri A, Behnam N, Rooij R, Hochstein L, Kosewski L, Reynolds J, Rosenthal C (2016) Chaos engineering. IEEE Softw 33(3):35–41
37. Rosenthal C, Hochstein L, Blohowiak A, Jones N, Basiri A (2017) Chaos engineering: building confidence in system behavior through experiments. O'Reilly, Sebastopol
38. Rosenthal C, Jones N (2020) Chaos engineering: system resiliency in practice. O'Reilly, Sebastopol
39. Beyer B, Jones C, Petoff J, Murphy NR (2016) Site reliability engineering: how Google runs production systems. O'Reilly, Sebastopol
40. Miles R (2019) Learning Chaos Engineering: discovering and overcoming system weaknesses through experimentation. O'Reilly, Sebastopol
41. Det Norske Veritas (2012) Vurdering av Altinn II-platformen. Report in Norwegian commissioned by the Norwegian Ministry of Trade and Industry, version 1.1
42. Capgemini Norge (2012) Altinn—en plattform å satse på? Report in Norwegian commissioned by the Norwegian Ministry of Trade and Industry
43. Sadalage PJ, Fowler M (2013) NoSQL distilled: a brief guide to the emerging world of polyglot persistence. Addison-Wesley, New York
44. Murray J (2015) Introduction to the actor model for concurrent computation. video at https://www.youtube.com/watch?v=lPTqcecwkJg

45. McKee H (2016) Designing reactive systems: the role of actors in distributed architecture. O'Reilly, Sebastopol
46. Lightbend (2017) Modernization: the end of the heavyweight era. White paper
47. Nygard MT (2007) Release it!: resign and deploy production-ready software. Pragmatic Bookshelf, New York
48. Dekker S, Pitzer C (2016) Examining the asymptote in safety progress: a literature review. Int J Occup Saf Ergon 22(1):57–65
49. Wilder B (2012) Cloud architecture patterns: using Microsoft Azure. O'Reilly, Sebastopol
50. Vernon V (2016) Domain-driven design distilled. Addison-Wesley, New York
51. Verhulst E (2014) Applying systems and safety engineering principles for antifragility. In: Proceedings of 1st international workshop: antifragile. Hasselt, Belgium
52. Russo D, Ciancarini P (2016) A proposal for an antifragile software manifesto. Procedia Comput Sci 83:982–987

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.