2022-04-18

How Failures Cascade in Software Systems

Barbara W. Chamberlin
Brigham Young University

How Failures Cascade in Software Systems

Barbara W. Chamberlin

A thesis submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Master of Science

Jonathan Sillito, Chair
Casey Deccio
Xinru Page

Department of Computer Science

Brigham Young University

ABSTRACT

How Failures Cascade in Software Systems

Barbara W. Chamberlin
Department of Computer Science, BYU
Master of Science

Cascading failures involve a failure in one system component that triggers failures in successive system components, potentially leading to system wide failures. While frequently used fault tolerant techniques can reduce the severity and the frequency of such failures, they continue to occur in practice. To better understand how failures cascade, we have conducted a qualitative analysis of 55 cascading failures, described in 26 publicly available incident reports. Through this analysis we have identified 16 types of cascading mechanisms (organized into eight categories) that capture the nature of the system interactions that contribute to cascading failures. We also discuss three themes based on the observation that the cascading failures we have analyzed occurred in one of three ways: a component being unable to tolerate a failure in another component, through the actions of support or automation systems as they respond to an initial failure, or during system recovery. We believe that the 16 cascading mechanisms we present and the three themes we discuss, provide important insights into some of the challenges associated with engineering a truly resilient and well-supported system.

Keywords: software design, cascading failure, graceful degradation, fault tolerance, graceful recovery

ACKNOWLEDGMENTS

Thanks go to my advisor, Dr. Jonathan Sillito, and the members of the software engineering lab.

Thanks also to my husband, Gordon, and my children, Lincoln, Celeste, Jocelyn and Genevieve for their patience and support.

Table of Contents

Chapter 1

In preparation: How Failures Cascade in Software Systems

This manuscript has not yet been accepted for publication.

How Failures Cascade in Software Systems

***Abstract*—Cascading failures involve a failure in one system component that triggers failures in successive system components, potentially leading to system wide failures. While frequently used fault tolerant techniques can reduce the severity and the frequency of such failures, they continue to occur in practice. To better understand how failures cascade, we have conducted a qualitative analysis of 55 cascading failures, described in 26 publicly available incident reports. Through this analysis we have identified 16 types of cascading mechanisms (organized into eight categories) that capture the nature of the system interactions that contribute to cascading failures. We also discuss three themes based on the observation that the cascading failures we have analyzed occurred in one of three ways: a component being unable to tolerate a failure in another component, through the actions of support or automation systems as they respond to an initial failure, or during system recovery. We believe that the 16 cascading mechanisms we present and the three themes we discuss, provide important insights into some of the challenges associated with engineering a truly resilient and well-supported system.**

I. Introduction

In the discussion of software failures, and cascading failures in particular, we define a software *failure* as "an event that occurs when the delivered service deviates from correct service." [2] A *cascading failure* is a "kind of failure in a system comprising interconnected parts, in which the failure of a part can trigger the failure of successive parts." [20] When a software system experiences a failure, responders are notified to investigate and mitigate the problem. In this paper, we refer to such an event as an *incident*. After an incident is mitigated, organizations may conduct a postmortem analysis of the incident and produce an *incident report* [5].

Work in multiple software engineering research areas can be seen, at least partially, as aiming to reduce the likelihood of failures or cascading failures (see for example work on defect detection and prediction [16]). Existing fault-tolerance techniques, such as load balancing, circuit breakers, caching and queuing, aim to reduce cascading failures by engineering a system component so that it can tolerate failures in its dependencies or dependents [2]. Cascading failures have also been studied in fields such as civil engineering and complexity theory [28]. In that work, it is clear that the larger and more complex the system, the more likely it is to suffer cascading failures, and insufficient slack in the system will make the cascading failure worse [1].

Missing from previous work is a comprehensive and detailed look at the ways that failures cascade between software system components. So, to fill that gap, the goal of our ongoing research is to identify and analyze in detail what we call *cascading mechanisms*. Our research also aims to explore the challenges associated with engineering a truly resilient and well-supported system. We believe that this line of research has the potential to improve our ability to engineer systems in which a failure in one component is less likely to cascade through other components.

As a step toward that research goal, we have conducted a qualitative analysis of 26 incidents from a database of incident reports which was collected as part of a previous research project [30]. The unit of analysis for the research is a *failure pair*, which is two failures either in distinct components, or in the same component, separated by time, where one failure is understood (by the authors of the incident report) as being the cause of the other. In our review of the 26 incident reports, we identified 55 failure pairs. Using a process similar to that described by Corbin and Strauss [6], we first coded and categorized each failure pair, then identified key cascading failure related themes. For more details on our analysis, see section III.

In reporting our analytic results in this paper, we make the following contributions. First, we describe 16 types of cascading mechanisms, organized into eight categories, that we identified in our analysis. These capture the nature of the system interactions that contribute to cascading failures and provide insights into some of the challenges associated with engineering a truly resilient and well-supported system. Second, we describe three higher-level themes identified in our analysis, which we argue suggest alternative ways to design and build systems that are truly resilient.

II. Background and Related Work

This research draws from a variety of different fields. The name "cascading failure" comes from engineering and networking, and there is a large body of engineering research pertaining to it [25]. In that context, a cascading failure occurs in networks where each node has essentially the same function as all other nodes, and there are hundreds or thousands of nodes, such as electrical grids, water and sewer pipes, traffic, telecommunications, and the internet. When one node fails, it requires other nodes to take up its share of the load, and if the network as a whole has insufficient slack, then other nodes fail, causing a chain reaction [1], [19].

However, in the context of software systems and in our work, cascading failures are defined more broadly as a "kind of failure in a system comprising interconnected parts, in which the failure of a part can trigger the failure of successive parts." [20] So, in the following, we discuss relevant software engineering research in the areas of failures and prevention (see Sections II-A), fault tolerance (Section II-C) and graceful degradation (Section II-B). We also discuss previous work related to complex systems and complexity theory, because it

addresses unexpected behavior emerging from the interactions of connected components (see Section II-D).

A. Categorizing and Preventing Failures

In a defect tracking system, a software organization may categorize failures by priority, severity, effect, location/type, and detection activity [33]. Previous research on failure categorization has tended to categorize failures based on the defects that cause the initial failure, and generally do not directly consider cascading effects. For example, Grottke et al. categorize defects into Bohrbugs, Mandelbugs, and aging-related bugs [14], [7].

Similarly, previous research has considered how to prevent various categories of failures. Ganesh et al. looked at ways to prevent aging-related bugs using techniques such as preemptive migration and software rejuvenation [11]. Ghosh et al. and Shin et al. investigated a variety of self-healing techniques [12], [29]. Monperrus looked at using AI to automatically generate code to fix failures [24]. Fault injection is currently an active area of research. It is used to analyze the reliability of specific systems [17] and to try to predict likely failure rates [32][3]. While some of these techniques are likely relevant to preventing cascading failures, they have not yet been applied in that context. We hope the research we are reporting in this paper can identify ways that previous prevention research can be expanded to consider cascading failures.

B. Graceful Degradation

When a system loses some functionality and is unable to automatically fix it, the system still needs to keep working to some extent. Graceful degradation means that while the system is not fully operational, it is handling its tasks as well as possible under the circumstances [15], [10]. Current work on graceful degradation tends to focus on scheduling tasks for multi-processor systems based upon their priority level [13]. Some of the failures in our data set may have been preventable if graceful degradation had been designed into the system.

C. Fault Tolerance

Fault tolerance in general refers to a variety of techniques that compensate for or tolerate failures in system components, often with the goal of preventing cascading failures. Sometimes fault tolerant systems are able to fix the problem and continue working as if it never happened [9]. Other times graceful degradation, as discussed above, is the best that can be achieved. Several software design patterns have been developed specifically for the purpose of adding resiliency to software systems [22]. Some examples include bulkhead, retries, circuit breaker, redundancy, load balancers, load levelers, load shedding, health monitoring, and auto scaling. We will briefly describe each of these techniques below.

The bulkhead pattern is used to isolate one or more subsections of the system to prevent failures from affecting other sections [31]. Retries are a standard method for dealing with transient faults or failures that may occur. However, if something causes the system to go down for a longer period of time, retries will not help and may stress the system further. In this case, the circuit breaker pattern can prevent some or all of the traffic to the struggling system to allow time for it to recover on its own or to be repaired by engineers [31], [26]. When the reliability of the hardware is important, the normal approach is to provide redundant hardware. When redundancy is used, it is sometimes necessary, when the primary resource provider goes down, for the backups to elect a new primary. Duplicate hardware may also be used to provide additional worker processes. In that case a load leveler is needed to distribute the work evenly. The queue-based load-leveling pattern may be used to even out the flow to an acceptable distribution [26]. If the load leveling is insufficient, the system may need to implement load shedding, either at random or by priority. If part or all of the system is located in the cloud, then health monitoring is an important technique to determine if the system is behaving as expected. Auto scaling is also used in cloud computing to provide the correct number of resources based on the current load of traffic to the system [22].

Naturally, various fault-tolerance techniques are relevant to cascading failures. As these techniques are in use in many systems today, including systems in our data set, our research has the potential to improve our understanding of their effectiveness and of their limitations in the context of cascading failures. We also hope that our work can inform and inspire the design of novel fault-tolerance techniques.

D. Complex Systems

Emergent behavior is defined as originating in the relationship between components. It is "behavior that is not attributed to any individual agent, but is a global outcome of agent coordination. ... Emergent behavior is a collective behavior." [18] Cascading failure is a type of emergent behavior that is often present in complex systems [23].

In the book *Meltdown*, Clearfield et.al. discuss several cases of catastrophic cascading failure and general principles that may help prevent such severe failures in complex systems large and small [4]. Cascading failure is driven by the connection between parts, not by the parts themselves. Small failures that are understood well in isolation can work together in unexpected ways [19] to create a situation that is not well understood. Charles Perrow, author of *Normal Accidents: Living with High-Risk Technologies*, described complex interactions as those of "unfamiliar sequences, or unplanned and unexpected sequences, and either not visible or not immediately comprehensible [sic]" [27]. One hallmark of complex systems is that the interactions, which are not visible, must be determined by indicators. Because we can't look directly into the system, we can't know enough to predict all the ramifications of a failure, even a small one. Linear systems, by contrast, have connections that are clear and easy to investigate and understand. They are much like an assembly line. Failure of one component means that the next component will not be able to start, but it doesn't have any hidden effects on any other component.

III. Study Setup

Our source of data for this study is a collection of publicly published incident reports, which are accounts of incidents written by the company where they occurred. Our analytic aim is to understand the ways that failures cascade between system components, so the unit of analysis for this project is a *failure pair*, which we define as two failures in distinct components, or in the same component separated by time, where the first failure is described in the incident report as being the cause of the second failure.

We describe our data source in more detail below and then describe our analytic approach. Broadly speaking, our analysis is based on grounded theory, which generally proceeds in three phases [6], [8], [21]. The first phase, *open coding*, involves segmenting and categorizing data. As an example, our open coding involved identifying and naming different types of relationships between failures. The second phase, *axial coding*, involves further analysis of the coded data, including exploring how various categories relate to each other and possibly creating hierarchical categories. The final phase, *selective coding*, involves identifying one or more overarching themes from the categories identified in earlier phases and refining those to tell a cohesive story. Iterative in nature, rather than strictly sequential, these three phases can channel back into previous phases if necessary.

A. Data Source

The *Incident Database* (IDB) is a collection of publicly published incident reports from a variety of companies. These reports are the product of internal postmortem analyses and generally describe significant system failures along with explanations of how and why they occurred and how they were mitigated. The IDB contains structured summaries of these incidents and is stored in a GitHub repository.[1] The incidents used in this analysis are summarized in Table I.

These incident reports were found by searching the web for existing lists of postmortems using the search terms "list of tech postmortems." An incident report was included in IDB if it was in English, published in 2010 or later, and had sufficient detail to determine system architecture and the details of the failures. We added incidents that we identified from these searches (in a random order) to IDB. For this research we read the first 35 incident reports in IDB to identify failure pairs as described in the next section.

B. Data Collection

As mentioned above, the unit of analysis for this project is a *failure pair*, which we define as two failures in distinct components, or the same component separated by time, where the first failure is described as being the cause of the second failure. In order to be used in our analysis, a failure pair's description in the incident reports had to meet three conditions: (1) the two cascading failures must be clearly described, (2) the relationship between the two components must be included, and (3) the way that the first failure cascaded to the second must be described.

To collect failure pairs for our analysis, we examined incident reports in the IDB, identifying failure pairs and all relevant report text. This collection step counts as a first level of analysis, so we kept as much of the original wording as possible to ensure accuracy in our summaries. Concretely, a failure pair in our data set included four parts: an *initial failure* which is the first in the pair of failures, a *subsequent failure* which is caused by the first, *architectural details* describing the relationship between the components that failed, and a description of the *relationship between failures* describing how the initial failure is believed to have caused the subsequent failure. Rarely, there are two initial failures which are independent of each other, but are both required to bring about the subsequent failure. The output from this analysis step is a summary with those four parts.

As an example of what our collected data includes, incident report BK describes database performance problems that were caused by a downgrade of the database and "started a cascade of other issues." In our review of that incident, we identified a failure pair (identified as *BK.1* in this paper) that we summarized as follows:

- *Architecture:* A service with a load balancer and autoscaler that depends on a hosted PostgreSQL database.
- *Initial failure:* Under daily peak load, database CPU utilization maxed out. Some transactions were slow and others failed.
- *Resulting failure:* Healthy servers (that were failing health checks) were removed from the fleet, causing a complete outage.
- *Relationship:* The health checks running on the servers attempted to connect to the database that was experiencing performance issues and so began returning 500 responses, triggering the load balancer to remove servers from the fleet.

Some incidents had failures that we were not able to use since the cascading mechanism was unclear. For example, in incident *D1* we identified three failure pairs. We also found an additional failure pair where we could identify the initial failure and the resulting failure, but not the relationship between them. Even the engineers could not identify it at the time of publication. "The exact cause of why this initial ... outage cascaded into a full system failure is not known at this time." Although we reviewed a total of 35 incident reports, nine of the reports[2] did not include cascading failures that met our inclusion criteria and will not be discussed further in this paper.

C. Analysis Process

For our analysis, we followed a grounded theory approach [6]. As noted above, the data collection was the first analysis step, and we began with 14 incidents. For these

[1] https://github.com/BYU-SE/idb

[2] The excluded incident reports are numbers 2, 4, 17, 20, 23, 24, 26, 29, 31 in the incident database.

TABLE I
A SUMMARY OF THE 26 INCIDENTS INCLUDED IN OUR ANALYSIS, ALONG WITH THE NUMBER OF FAILURE PAIRS IDENTIFIED IN EACH.

ID	Company	Report	Incident summary	#
A1	AWS	go.aws/3BiCVzd	Bandwidth disruption led to re-mirroring storm and then spread to control plane	4
A2	AWS	go.aws/2YsdwUQ	Missing control data led to errors in customer data and degraded performance	2
A3	AWS	go.aws/3oSKJnS	Accidental removal of servers from multiple data store clusters led to outages and backlogs	2
BK	Buildkite	bit.ly/2xTnliK	Database downgrade and connection pool configuration change led to outage at daily peak load	3
CC	Circle CI	bit.ly/3DgMfnB	Pause in web hooks led to backlog and site degradation	2
CF	Cloudflare	bit.ly/2RoVfBu	Configuration deployment led to an increase in CPU usage and a global outage	1
D1	Discord	bit.ly/2Yo61yY	Automated migration of primary datastore + a failover defect led to cascading outages	3
D2	Discord	bit.ly/3lfgoxj	Errors on one cluster node + a defect led to a split cluster and delays for dependencies	3
E1	Elastic	bit.ly/3DxZy34	Disconnections between two application layers led to excessive resync requests and an outage	2
EG	Epic Games	bit.ly/2XqzFjR	High traffic caused 6 outages due to exceeding database, loadbalancer, memcached limits	3
G1	Google	bit.ly/3ljaT0W	Configuration error caused network control jobs to be descheduled and a network outage	1
GC	GoCardless	bit.ly/39UxK9R	Cluster management misconfiguration + coincident disk failure and process crash led to outage	1
GH	Github	bit.ly/3AhoEkT	Network interruption led to misconfigured database clusters, affecting latency and availability	3
GH2	Github	bit.ly/3DG3rTZ	Power outage led to multiple servers unable to restart	1
GL	GitLab	bit.ly/2wpqMMq	Database replication failure due to high load and accidental data deletion led to database outage	2
HE	Heroku	bit.ly/3am0smP	Maintenance operation that silently failed led to routing errors for application containers	1
JO	Joyent	bit.ly/3Df7YMM	Database deadlock due to autovacuum conflict with application transactions	1
MC	MailChimp	bit.ly/2yPjFOl	Database maintenance processes failed due to high load, causing database shutdown	1
PL	Parse.ly	bit.ly/3c7sFwq	Usage growth exceeded network capacity of nodes in a cluster and led to a cascading outage	2
RD	Redit	bit.ly/3lcBGM9	Autoscaler, unintentionally running during upgrade, terminated healthy servers leading to outage	2
SF	Salesforce	sforce.co/3mtPFNd	High traffic exposed a firmware defect, leaving a database in a corrupt state	3
ST	Stripe	bit.ly/39Z5dQu	Existing configuration defect + multiple database failures caused a failed failover and an outage	3
T2	TravisCI	bit.ly/3CipBu5	Accidental (and permanent) deletion of virtual machine images broke build jobs	1
TC	TravisCI	bit.ly/3oEsDWA	Clean up process was not updated with new passwords and failed silently, leading to resource leak	3
TS	Tarsnap	bit.ly/3iDnlXm	Errors in a dependency caused excessive logging and full disks, crashing colocated applications	1
TW	Twilio	bit.ly/3BjL3iQ	Network disruption led to database failure and repeated customer credit card charges	4

first 14 incidents, both researchers independently reviewed the incidents and identified the cascading failure pairs. They then came together to discuss them until agreement was reached on which failures to include/exclude and how to describe them both accurately and succinctly. Open coding the failure pairs, which is a categorizing exercise, involved looking at failure pairs individually, though in the context of the incident, and finding words or short phrases that described the elements of interest in the pair. Both researchers coded the failures individually, and we discussed the coding until an initial and provisional set of codes was developed. As an example, for the failure pair above (*BK.1*), we had the following potential codes: Coupling, Load and Capacity/Time based, Automation/Health Checks, Automation/Relentless and indiscriminate, and Defect.

During the second round of analysis, we reviewed and identified failure pairs in an additional 12 incidents. One researcher identified the failure pairs while the other summarized the incidents. Both researchers then reviewed and refined the other's work. So, both researchers read and analyzed the incidents, but in sequence instead of in parallel. We reached saturation, that is we reached a point where we were not finding new mechanisms, at 26 incident reports and 55 failure pairs.

During the axial coding phase, which involved grouping codes, interpreting them, and reflecting on their meanings [21], we compared and contrasted failure pairs and, in the process, fine-tuned our coding scheme. We chose to concentrate on the relationships between failures, specifically on the cascading mechanism which is primarily located in the relationship descriptions in our data. Each mechanism, which explains how the initial failure cascaded to additional failures even across component boundaries, is different in the details, but often similar in the broad outlines. The cascading mechanism in the failure pair described above, *BK.1*, was ultimately named *Destructive health check actions*, and we found three other failure pairs in our data set with that same type of mechanism. In this way we identified 16 types of cascading mechanisms, which we then grouped into eight categories.

Finally, we identified three key themes that were central to understanding how failures cascade, and how those cascading events occur in different stages of an incident's life cycle: Ungraceful degradation, Automating failure, and Ungraceful recovery. Our mechanisms and categories are described in detail in Section IV, and our themes are discussed in Section V.

D. Risks to Validity and Limitations

IDB data will be biased towards incidents that are significant enough to merit being published publicly. We believe the consequence of this bias is that our data set is comprised of

incidents of greater than average significance. However, as we are not attempting statistical generalization nor attempting to capture the day to day work of engineers, the sample we have collected suits our research objectives. It would be useful to conduct a follow up research project aimed at understanding how frequently each of the cascading mechanisms occur, as part of, for example, understanding which are most important to protect against.

The concepts (mechanisms, categories and themes) we have identified are grounded in the 55 failure pairs we have analyzed, and through our analysis we have refined them to the point that they are stable across those pairs. Toward the end of our analysis we were not adding new concepts, but refining and adding new examples of already discovered concepts. These concepts should not be viewed as comprehensive. Though our set of incidents is relatively diverse, it is probable that with a different set of incidents, and therefore a different set of failure pairs, other concepts would emerge. We hope that future research, by us or others, may extend our findings based on more data.

The accuracy of our analysis of each failure pair, and specifically the cascading mechanisms, depends on the accuracy of the authors of the incident reports. When the authors were not certain about the cause of a failure, or more specifically, the relationship between two failures, we excluded the failure pair from our analysis. If the authors were incorrect or incomplete in their explanation of the relationship, it is possible that our analysis is similarly incorrect, which should be considered when interpreting our results. However, in our experience, significant effort is put into postmortem analyses, and we believe that errors will be rare.

IV. Analytic Results

"Incidents are usually comprised of several systems–both technical and human–interacting and overlapping in unexpected ways,"[3] and it is those interactions that are at the heart of cascading failures. In our analysis we have found that these interactions include the actions of automation or support systems, such as cluster management software and auto-scaling groups; actions that are appropriate in some contexts but unexpectedly detrimental in other contexts. We have also found that as responders work to restore the system to its normal state by resolving the initial failure and any cascading failures, their actions have the potential to introduce additional cascading failures as the system moves through unusual states on its way back to normal.

In our analysis we identified 16 mechanisms, grouped into eight categories, that capture the nature of the interactions that contribute to cascading failures, as summarized in Table II and discussed below. The mechanisms and categories presented provide insights into some of the challenges associated with engineering a truly resilient and well-supported system. Other higher-level insights or themes that emerged during our analysis are discussed in Section V.

[3]https://mailchimp.com/what-we-learned-from-the-recent-mandrill-outage

TABLE II
Cascading mechanisms by category and the number of failure pairs in each.

	Categories and Mechanisms	#
A	**Deferring work and cascading failures**	**6**
	Exhausting resources while deferring work	3
	Overwhelming with deferred work	3
B	**Troubles (re)connecting**	**9**
	Failing to reconnect to restored service	3
	Overwhelming with reconnections and resynchronizations	6
C	**Resource competition and starvation**	**8**
	Competing for thread and CPU resources	4
	Competing for database resources	4
D	**Failing to handle an unsupported state**	**6**
	Failing to handle database state	3
	Failing to handle unexpected state	3
E	**Unsupportive support systems**	**8**
	Destructive load-balancing and health-checking	4
	Cascading failures through control processes	4
F	**A cluster of cascading failures**	**6**
	Cluster failing over to error state	4
	Cluster reaching resources limits	2
G	**Cascading while recovering**	**7**
	Adventures in rolling forward and back	2
	Troubles restarting under duress	5
H	**Responders cascading failures**	**5**
	Errors by incident responder	2
	Calculated tradeoffs and deliberate shutdowns	3

A. Deferring Work and Cascading Failures

A class of fault-tolerance techniques that have the potential to prevent some kinds of cascading failures involve deferring work during a failure, with the idea of handling that work when the failure is resolved. For example, a component that sends messages to a dependency may queue failed messages for later retry. Interestingly, in six failure pairs in our data set, these techniques were actually central to the way that a failure cascaded. This was generally because the failure lasted longer, or required deferring more work, than the techniques were designed to handle, suggesting that in some cases the technique may simply delay or change the nature of the cascading failure, but not prevent it. In our analysis we identified two specific cascading mechanisms in this category.

1) Exhausting resources while deferring work: While waiting for a dependency to recover, a dependent component that defers work may exhaust the resources used to store that deferred work, as occurred in three failure pairs (*TS.1*, *D2.3*, *E1.2*). For example, in *D2.3* there was a mechanism in place to handle network communication interruptions using an in-memory "buffering mechanism" which "works great," but when the interruption persisted, the messages "quickly filled the in-memory buffers [...] and caused many of [the

servers] to run out of memory and fail." In this case, a mechanism to tolerate a likely failure (network interruptions) also introduced other, potentially more serious failures (out of memory). Similarly, in *TS.1*, on failed writes to a datastore, the data to be written was logged to disk, eventually filling the disk and causing a cascading failure.

2) Overwhelming with deferred work: On the other hand, once a dependency recovers, the dependency (or other downstream systems) may be overwhelmed with traffic when the dependent component begins retrying that work, as occurred in three failure pairs (*SF.1*, *TC.3*, *A3.2*). For example, in failure pair *SF.1*, in response to a power failure, responders moved an application from one data center to another. A backlog of "traffic" built up in the interim, leading to an "increased traffic volume" which triggered a "latent firmware bug" once the mitigation was complete. The consequence was data corruption and a database outage. Similarly, in *TC.3*, the system had so many queued items that when it started processing them, it overwhelmed the CPU by starting too many at once and immediately failed. In all three cases, the downstream dependencies had sufficient capacity for the usual volume of traffic but not the temporarily accelerated rate of traffic.

B. Troubles (Re)Connecting

In some distributed systems, client components may maintain a long-lived network connection with a server component. In some cases, this connection is used to keep data in a synchronized state. When an initial failure interrupts a long-lived connection or results in other changes to the server component (promoting a new primary, for example), issues associated with reestablishing that connection, or the load associated with multiple clients attempting to reconnect or resynchronize data, can lead to cascading failures.

1) Failing to reconnect to restored service: We identified three failure pairs where the cascade was due to clients of a cluster not responding appropriately once a cluster failed over (*D1.1*, *D1.3*, *PL.2*). For example, in failure pair *D1.1*, a remote database cluster "properly failed over to the secondary." However, due to a defect in the client side components, it appeared to the application that the cluster had dropped offline. Therefore the clients were unable to "properly handle the failover." As a result, failure in the cluster led to failures in the clients. Similarly, in failure pair *D1.3*, in response to an initial failure responders were required to restart a cluster, causing a failure for the clients as they were not able to reconnect to the restarted cluster due to a defect with a configuration "setting for allowing users to quickly and safely reconnect" to the cluster.

2) Overwhelming with reconnections and resynchronizations: Six of our failure pairs featured clients that maintain a long-lived connection to a server or service. When an initial failure interrupted those connections, a cascading failure occurred as all of the clients simultaneously attempted to reconnect, overwhelming the server as a result (*EG.3*, *D2.2*, *A1.1*, *A1.3*, *TW.1*, *E1.1*). In these cases, the capacity of the system was sufficient to maintain all connections and incrementally synchronize data, but not sufficient to reestablish all of those connections concurrently. For example, in failure pair *D2.2*, one node in a cluster failed, millions of clients lost a connection to the cluster, and a "thundering herd" of reconnections caused the cluster to crash completely. Similarly, in failure pair *A1.1*, a large number of primary nodes in a highly redundant data store lost connection to their secondary nodes and attempted to establish new secondaries, leading "to a re-mirroring storm" that overwhelmed the disk volumes and network. Then, that re-mirroring storm (along with a latent defect, associated with closing network connections) led to a vicious cycle of connections being exhausted and more node failures.

C. Resource Competition and Starvation

One consequence of an initial failure can be changes to load, capacity, or resource utilization in one or more components, and these changes can result in cascading failures. In this subsection we discuss two such cascading mechanisms, both related to shared resources, where a change in resource utilization for one activity can cause resource starvation for another.

1) Competing for thread and CPU resources: In four failure pairs, a shared computational resource, such as a shared thread pool or a server's CPU, was central to the cascade of a failure (*EG.1*, *ST.2*, *CF.1*, *A1.2*). In all four cases, an initial failure caused an increase in computational resources used for one activity, leading to a second activity failing due to resource starvation. For example, in failure pair *ST.2*, the initial failure was an outage of one database shard. On dependent components, calls to that shard were blocked until a timeout was reached, so "the unavailability of this shard starved compute resources [...] and cascaded into a severe [...] degradation" in dependent components. It is interesting to note that the threads that were consumed were not actually doing any work, as they were simply waiting for a response, but they were nevertheless unavailable to other activities.

2) Competing for database resources: In four failure pairs, the shared resource that was central to the cascade of a failure was a database or database cluster, where an initial failure led to an increase in resources consumed by one activity, which in turn led to other activities being starved for database resources or database failure (*D1.2*, *GL.1*, *CC.1*, *TW.2*). For example, in failure *GL.1*, relatively less important maintenance transactions contributed to a database overloading, causing a cascading failure in clients of the database. Another consequence was that "the database replication process" stopped, due to resource starvation, and some data was permanently lost.

D. Failing to Handle an Unsupported State

One category of cascading mechanism that we identified in our data revolved around error or otherwise unsupported states. Essentially, such cascading failures occur when an initial failure leaves a component in an unsupported or error state,

and a subsequent failure occurs as other components interact with the component in an error state. Such an unsupported state is generally different than an outage in terms of its effects on other parts of a system. In the following, we describe two such cascading mechanisms.

1) Failing to handle database state: In three failure pairs, an initial failure left a database in a state that other components in the system did not support, resulting in cascading failures (*JO.1*, *MC.1*, *TW.4*). The three unsupported database states were *dead locked*, *safety shutdown* and *read-only*. For example, in failure pair *TW.4*, an initial failure left a database with incorrect data (financial account balances set to 0) and the database in read-only mode. A second failure occurred as the billing system began to repeatedly and erroneously charge customer credit cards, because successful charges could not be recorded in the read-only database, until responders "shut down the billing system to prevent further charges."

2) Failing to handle unexpected state: In three failure pairs, the cascading mechanism was that one failure left a component in an unsupported state or with corrupt data, and a second failure occurred as other components interacted with that component (*TC.2*, *GH.2*, *SF.2*). For example, in failure pair *TC.2* an initial failure left many virtual machines in an unusual state as they failed to power on. A support process then failed to clean them up, as it was configured only to clean up virtual machines that terminated normally, which led to a cascading failure due to the subsequent resource exhaustion. In the case of failure pair *GH.2*, a database cluster failed over to a new primary, but that primary was located in a distant data center, causing cascading failures as applications were "unable to cope with the additional latency introduced by a cross-country round trip for the majority of their database calls." During failure pair *SF.2*, a primary database cluster failed with "file inconsistencies" and, before it failed, those "discrepancies had been replicated" to the secondary, which then also failed and could not be used as a fallback to restore service.

E. Unsupportive Support Systems

In addition to the core systems involved in cascading failures, we also identified multiple examples of support or auxiliary systems taking actions that led to a cascading failure. Interestingly, these systems that can cause cascading failures are often designed to prevent or mitigate failures. In this section, we discuss cascading failures involving load balancers and control systems. In the next section, we discuss cascading failures associated with clusters and cluster management systems (see Section IV-F). Note that in other sections we also discuss failure pairs that relate to support systems, such as *PL.1*, *D1.2*, *EG.1*, *EG.3* and others.

1) Destructive load-balancing and health-checking: In four failure pairs, an initial failure led to a load balancer removing healthy servers and causing a cascading failure (*BK.1*, *BK.3*, *EG.2*, *RD.1*). For example, in failure *BK.1*, existing servers failed health checks because the health check attempted to establish a connection to an overloaded dependency. The consequence was that the load balancer "automatically started removing servers" but failed to replace them. "Servers kept coming up, failing health checks, and going down again," eventually exhausting the provider-enforced limit on the number of servers that could be provisioned (*BK.3*) and hampering responders' efforts to later restore service. It is worth noting that the removal of the healthy servers made a partial system outage a complete system outage.

2) Cascading failures through control processes: Networking (and other) systems differentiate between control systems and data systems. The *data plane* transmits data through the network. The *control plane* manages and configures the individual devices on the network. In four failure pairs, an initial failure overloaded a control system or left one in an error state, and, after some delay, that process's absence or inability to perform its functions caused a secondary failure (*G1.1*, *A2.1*, *HE.1*, *A1.4*). For example, in failure pair *A1.4*, a network was suffering a "re-mirroring storm," and each re-mirror operation involved a call to the control plane, eventually causing a "brown out" and more re-mirror failures in a vicious cycle. In failure pair *A2.1*, due to an accidental manual deletion of configuration data, the support system used to manage load balancers was in a degraded state. As it made changes to load balancers, in response to user requests through an API, those load balancers became "improperly configured" and degraded in performance. As a final example, in failure pair *G1.1*, when an initial failure stopped the network control software, the network eventually became congested, as routing information changed and the network was not updated accordingly, causing widespread cascading failures.

F. A Cluster of Cascading Failures

As mentioned above, various support systems are involved in managing clusters, such as database clusters. One cluster-related cascading mechanism that we have already discussed occurs when clients of a cluster fail to reconnect to a cluster after it fails over to a new primary node or recovers from other failure scenarios (see IV-B). In this section we discuss two other cluster-related cascading mechanisms.

1) Cluster failing over to error state: Cluster management software is designed to tolerate various failure scenarios by, for example, failing over to a new primary node. However, in four failure pairs (*GC.1*, *ST.1*, *GH.1*, *D2.1*), the cluster management software was unable to tolerate a particular scenario and so an initial failure (say of one node) led to the failure of the cluster, generally with additional cascading consequences for services that depend on the cluster. For example, in failure pair *ST.1*, due to a latent defect in the database cluster software, only manifest in the presence of two "stalled" nodes, when the original primary failed, the cluster was not able to promote a new primary, though candidate nodes were available. In failure pair *GH.1*, a network failure (a network partition event between two data centers) led a database cluster to fail over to a topology that was invalid and also a database state that could not be reconciled automatically, as there were un-replicated writes in each data center.

2) Cluster reaching resource limits: In two failure pairs, an initial failure caused a cluster to reach a resource limit, resulting in a cascading failure (*PL.1*, *TC.1*). In incident *PL.1* one cluster node hit "operating system network limits" and failed, so the network load on the remaining nodes increased until eventually all nodes failed in a "classic cascading failure." Interestingly, though the owners of the cluster monitored various resource utilization metrics for the cluster, including network related metrics, they were unaware of what the limit was. In failure pair *TC.1*, a defect in the cluster management system responsible for cleaning up unused virtual machines caused a resource leak and eventually the "cluster was unable to satisfy the resource reservations," which led to a complete failure of the cluster.

G. Cascading While Recovering

Some cascading failures we identified occurred due to automated or manual actions taken while responding to the failure. Such actions are often taken with limited testing, may represent rare system operations, and may bring the system into unusual states, so it is perhaps not surprising that there are risks associated with the actions. In this section we discuss two such cascading mechanisms, and in the next section we discuss other actions responders take that can lead to cascading failures (see Section IV-H).

1) Adventures in rolling forward and back: When a source code or configuration change introduces a defect and causes an incident, responders or automated systems may return a component back to a previous version (called rolling back). Alternatively, responders may roll out a new version of the component with a fix (called rolling forward). In either case, when these occur during an incident, opportunities for testing are limited, and the new version combinations can lead to additional cascading failures, as occurred in two failure pairs in our data set (*ST.3*, *T2.1*). In failure pair *ST.3*, one component was rolled back to a previous version, due to a defect, and the "rolled-back [component] interacted poorly with a recently-introduced" configuration change, leading to an additional cascading failure. On the other hand, in failure pair *T2.1*, because some deleted virtual-machine images were unrecoverable, the responders rolled scheduled jobs forward to newer virtual-machine images, resulting in failed jobs and weeks of "all hands support" to fix all of the issues that were introduced. In a related scenario, the failure pair *RD.1* was triggered by an automated rollback. In this failure pair, the rollback was not the cascading mechanism, but was instead the instigating event. The engineers had manually disabled the load balancer in order to reconfigure the system. Another support system noticed the manual change and automatically removed it (rolled it back), thereby causing the subsequent chain of cascading failures.

2) Troubles restarting under duress: At times, a component of a system is restarted in response to an initial failure. In five failure pairs, that restart was central to a cascading failure (*BK.2*, *TW.3*, *GH2.1*, *RD.2*, *A3.1*). For example, in *TW.3*, a database was rebooted manually, a rare operation for that system, but it read a bad configuration file and started up in read-only mode and with no data in the customer accounts, leading to a cascading failure as other components attempted to interact with the database (see also the discussion in Section IV-D). Similarly, in *GH2.1*, the initial error was a power outage, and, once power was restored, several components failed to restart because "the boot path of [the associated] application code" had a "hard dependency" on other components that were also still recovering. It is also possible for a cascading failure to occur after a restart, as in failure pair *RD.2* where, after "all servers were restored ... [the] new caches were still empty," leading to latency-related failure for clients. In addition to these five failure pairs, we have previously discussed cascading failures that occurred as clients attempt to reconnect to a newly restarted component (see Section IV-B).

H. Responders Cascading Failures

Building on the discussion in the previous section about cascading failures that occur during incident recovery (see Section IV-G), here we discuss situations where responders, deliberately or not, may directly cause cascading failures. While these mechanisms might not be considered cascading failures in the traditional sense of the word, they are included here for completeness.

1) Errors by incident responder: In two failure pairs, we identified actions taken by responders, in response to an initial failure, as unintentionally leading to a cascading failure (*GL.2*, *CC.2*). Note that some actions taken during an incident may not necessarily follow typical operational procedures and may be done under a time pressure. For example, in incident *GL*, while attempting to repair a data replication process between a primary and secondary database, a responder deleted production data on the primary database "errantly thinking they were doing so on the secondary" (*GL.2*). Due to a coincident failure, no recent backups of the database were available, leading to permanent data loss.

2) Calculated tradeoffs and deliberate shutdowns: Three failure pairs in our data set are examples of responders deliberately choosing to cause or allow a cascading failure (*SF.3*, *GH.3*, *A2.2*). Each of these failure pairs feature responders making application-specific tradeoffs, often between the lesser of two or more evils. For example, in failure pair *SF.3*, responders performed a time-consuming and resource-intensive data restore process to mitigate an initial failure, and a "backlog of built up jobs" developed. Responders decided "to not impact a second day of customer activity" so they abandoned the data restore process, "halted several internal jobs and [...] staggered initial customer activity." Taken together, these resulted in several cascading failures but avoided another general failure due to excess database load. In failure pair *GH.3*, the responders "made an explicit choice to partially degrade site usability" to "prioritize data integrity over site usability and time to recover." Similarly, in failure pair *A2.2*, due to the error state of a control system, changes by users to their load balancer would result in "degraded performance

and errors," so responders disabled various "control plane workflows" to prevent this.

V. Themes

In this section we discuss higher-level themes that emerged from our analysis, and in particular our selective coding. The discussion of these ideas is organized around three stages of an incident's life cycle, based on the simple observation that broadly speaking cascading failures can happen in one of three ways. First, and possibly most expected, a cascading failure may be due simply to one system component failing in some way and another component being unable to tolerate that failure leading to ungraceful degradation (see Section V-A). Second, a cascading failure may occur when support systems or automation systems responding to an initial failure, and introduce an additional failure (see Section V-B). Finally, cascading failures may occur as responders attempt to mitigate or resolve a failure, intentionally or unintentionally, leading to more failures and an ungraceful recovery (see Section V-C). As we discuss these higher-level themes, we conclude each of the following subsections with a discussion of the implications these themes may have for the engineering of resilient systems.

A. Ungraceful Degradation

Many cascading failures in our data set are examples of subsystems failing *abruptly*, in contrast to degrading gracefully. A lack of resources, or a mismatch between load and capacity (demand being greater than supply) are at the heart of the reason these failures cascaded, and that mismatch can occur in many different ways during an incident. When resources are shared between two or more activities, an initial failure with one activity may starve the other activity, such as when regular load, automated maintenance activities, and backups competed for database resources, leading to failure pair *GL.1*. When a work load is shared between multiple components, a load-related failure of one component leaves more load for the remaining components, in a "classic cascading failure" [*PL.1*]. Similarly, in response to some types of failures, clients of a component may need to reconnect, re-mirror, or retry requests, leading to excessive load, possibly on a component already in distress, and a subsequent failure such as occurred in *E1.1*.

Naturally, if demand is greater than supply, we would hope the system would at least be able to handle some of that demand, rather than simply fail abruptly and handle none of it. Various fault-tolerance or graceful degradation techniques are used in systems, including the systems in the incident reports we analyzed, to guard against some resource-related failure scenarios. In some cases, a technique such as load shedding may accomplish this, but it is not applicable to all of the cases we describe above and is also a "blunt tool," indiscriminately shedding demand. Other techniques also have limits and, in some cases, may simply delay a resource-related cascading failure or even cause one. For example, an in-memory "buffering mechanism" for failed-message sends can cause components "to run out of memory and fail" [*D2.3*]. Even when deferred work does not exhaust available resources, like with retries or reconnections, catching up with that deferred work can cause excessive load on a component after an initial failure is resolved, causing a cascading failure and delaying the full recovery of the system.

One perspective on the nature or cause of these resource-related cascading failures is that in many ways, systems are engineered based on an implicit assumption of abundance. That is, an engineer may assume there are enough threads, memory, CPU cycles, et cetera, and engineer the system accordingly, with some attempts to handle specific exceptional scenarios. When that assumption does not hold–that is, during moments of scarcity–the system may fail completely rather than degrade gracefully. Similarly, we may assume that we can automatically scale up and that it is always cost-effective to do so. From our analysis, it is clear that the assumption of abundance does not always hold as systems degrade and fail. Nor does it always hold as systems recover from failures, with components operating at less than full capacity for some period of time. We have also seen that "adding a few more nodes" may not always be "that simple" [*PL.1*].

To develop more graceful alternatives, it may be valuable to ask the question: *How could we engineer a system that does not assume abundance, but simply makes effective use of the available resources?* If the typical approach is to assume we have an abundance of resources, then that assumption is supported with resource allocation and scaling strategies. Often, excess resources are allocated to account for the time required to add resources, for example. In other situations, as our data shows, excess resources might be allocated because of the risk of complete failure when demand exceeds supply, even by a small amount. The proposed alternative is to engineer the system to make do with what is available at the moment, with the result that allocation decisions become more flexible and perhaps more efficient in normal operation. Our analysis suggests that such an approach may lead to systems that are more resilient during incidents.

Arguably, that approach to engineering systems may be well supported by changes to how we think about system requirements. Often, system requirements are expressed as binary. For example, consider a system that generates a complex product page on a retail site. The requirements might say that the page request must return in less than 500ms, but there is no information that would inform decision making if the system is degraded. This observation applies to both functional and non-functional system requirements. Though not all work a system component performs is of equal value, functional requirements do not always express the relative value of that work. A performance requirement expressed as a maximum response time, for example, is not rich enough to inform decisions about queuing work or shedding load when trade-offs need to be made. To demonstrate this concept, consider the product page mentioned above. It might normally show a name, description, picture, price, user reviews, similar items, manufacturer, and a host of other information. If the system is degraded, should it return fewer elements, or should it take longer to returns all elements? Does it treat all client

requests equally? Perhaps the product page above would give less data to some customers, or take a longer time for other customers, or meet the stated requirements for some while ignoring others. Requirements that capture the costs of delay or decaying utility over time may better support engineering a system that can make appropriate trade-offs under distress. For a different example, if a cache has a specified time to live (TTL), but the database is slow or is not responding, is it better to empty the cache or extend the TTL?

B. Automating Failure

Secondary support systems and automation systems may unexpectedly contribute to a cascading failure. Examples of support systems include cluster management software that may respond to failed nodes in a cluster and closely related database redundancy systems. Automation systems in our data set include load-balancers performing health checks and removing servers that are in an error state, processes for automatically adding servers to a service when more capacity is needed, etc. Our analysis has identified two ways that such systems may contribute to cascading failures.

First, support systems are subject to failures just like other systems, and these failures can be cascading failures when they adversely affect the system they are supposed to be supporting. Examples include cluster management software that was unable to promote a new primary in a particular failure scenario "due to a bug that was fixed but not yet deployed" (*D2.1*) or when another did promote a new primary but the result was an invalid cluster topology, a database in a bad state, and failures in dependent services (*GH.1*). In that last example, the result was "degraded service for 24 hours" while responders repaired the damage caused by the cluster management software's handling of the failure. Cluster management (and similar) software are controlled by many configuration parameters or constraints, which need to be satisfied in each failure scenario. That can lead to "extremely unusual behavior [...] in certain failure conditions," (*GC.1*) such as being unable to "decide where the [new] primary should run," despite the appropriate choice being clear to the responders. The cascading mechanisms *cluster failing over to error state* (see IV-F1) and *failing to handle database state* (see IV-D1) involve cluster-management related failures. The cascading mechanisms *troubles (re)connecting* (see IV-B) and *cascading failures through control processes* (see IV-E2) contain examples of other automatic support systems where the initial failure had clearly been anticipated, and a suitable response had been designed and implemented, but the response did not produce the expected result.

Second, automation systems may take actions that, while reasonable in some contexts, are based on an incorrect view of the overall event and of the consequences of those actions. Automation played a role in all three of *BK*'s failure pairs. The initial failure began when a shared database "couldn't keep up [with load] and its CPUs started maxing out." Three load-balanced and auto-scaled services began experiencing some health check failures, because the health checks checked the server's connection to the degraded database, and the load balancer with the lowest configured threshold "automatically started removing servers," leading to a complete outage for that service. Attempts to add new servers failed due to a defect and those attempts continued until the cloud provider's "request limit" was reached, which later hampered responders' efforts to mitigate the issue. It is important to note that the removed servers were healthy, and removing them was counter productive, suggesting there was a mismatch between the signal from the health check and the action taken by the automation, or that the action ought to consider more context (such as the health of the database, in this case). Other automated actions that caused cascading failure in our data set were an automated rollback *RD.1*, a garbage collection system *TC.1*, and a web pre-processor *CF.1*, as well as the cluster managers and backup systems mentioned above.

One way to think about this problem is that the automated actions are taken based on assumptions, and the system may not always be checking to ensure that the assumptions hold. So even if the actual scenario unfolding during an incident is different in important ways from what was envisioned when the automation was put in place, the actions are carried out and are relentlessly continued even if they are doing harm. A simple improvement may be to be explicit about those assumptions when designing the automation. In some cases it may be useful to implement additional checks to ensure assumptions hold and define a *continue criteria* (or the reverse, a *back-out criteria*) for actions that would allow the automation system to determine when its actions are not having the intended effect and should be discontinued. This concept could be modeled as a measure of *certainty* that the assumption holds, possibly with multiple inputs influencing that measure.

As an example of this idea, consider a failure similar to *EG.2* in which an automated system removed servers from a fleet due to failing health checks, replacing those with new servers. The assumption being made is *servers are in a bad state and replacing them will lead to an increasing proportion of healthy servers*. The action taken based on that assumption is to *replace servers* and each replacement takes a known duration. A relevant *continue criteria* could be that *the ratio of healthy servers increases proportional to the number replaced*. Then the automation system could pace the action, checking the criteria as it went, and only continue the action as long as the criteria held. If it did not, as was the case in this failure, it could discontinue the action in favor of an alternative or simply wait for the responders to determine the best course of action, rather than cause a cascading failure and make the incident harder for responders to unwind.

C. Ungraceful Recovery

An initial failure during incident *PL* "led to a number of cascading failures throughout [their] plant, which took [their] team some time to fully unwind." As discussed under theme *ungraceful degradation* above, system components degrading can be dangerous for other components, and the longer the

degradation lasts, the more dangerous it is. Interestingly, we have also found that a component recovering can also be dangerous. Even during the manual or automated response to an incident, additional cascading failures can occur. There is messiness and risk associated with unwinding the error state of a system. The process may take time and include attempted fixes that fail (see categories IV-E1 and IV-G2) and may require deliberately degrading components until they can be safely restored (see category IV-H2).

For many of the complex systems in our data set, there was no straightforward way to bring the system from the error state they were in to a normal state. Certain restoration activities need to be carefully ordered, are delicate, or are rarely performed, as in incident *SF*, which featured a subsystem which, at the time of the incident, the responders had not "completely restarted [...] for many years." Since that time, business growth and other changes made the restart, and validating its integrity, take "longer than expected." During incident *D2*, the responders needed to restart a cluster, but needed to make some configuration changes first, as they were concerned about "simply letting the thundering herd that is millions of [...] clients reconnect to [the cluster] at once." And in other cases, such as during incident *RD*, responders simply need to wait. Incident *A1* is an extreme example of "hard to unwind" failures, because the failure included a storage cluster and the responders were attempting to avoid data loss. The process took multiple days and ongoing careful attention by the engineers and still was not completely successful.

In incident *PL* the recovery plan involved a series of stages that did not go as expected. Backlogs had to be processed or deleted; data had to be backfilled ("re-process a massive backlog of data" to catch up to real-time); servers had to be added to replace failed servers and new clusters put in operation. Clusters had to be expanded, even if responders were "not sure how gracefully it would handle dynamic cluster expansion" and still needed to be manually rebalanced even if they did handle it "gracefully." Such response activities needed to be carefully ordered and paced, so they "slowly flipped on downstream consumer systems," carefully tracking the results. Such multi-day recovery efforts often involve never- or rarely-experienced combinations of component states, and demonstrate ways that components had not been engineered to automatically recover when other components recover.

Despite the limitations we discussed above, techniques for tolerating failure and degrading gracefully are common considerations when designing systems. Our analytic results suggest there would be value in also considering closely related techniques for *tolerating recovery* or for *gracefully recovering*. We argue that a system that can gracefully recover has at least four properties. First, components can wait patiently while another component is in a distressed state and can return to normal operation when the subsystem recovers, ideally with no time limit and no intervention needed. Second, the order of recovery is unconstrained, meaning that if multiple components are in degraded states, they can recover (by restarting, for example) in an arbitrary order. Third, no subsystems need to be manually turned off and back on during response. Fourth, state changes (e.g. restarts) are handled gracefully, avoiding "resync storms" and "thundering herds."

VI. Summary

We have identified eight categories of mechanisms for cascading failure, with two cascading mechanisms in each. Some of these categories are no surprise to anyone. Others might be less obvious. By using these categories as reference points, we hope to make it easier for engineers to design, maintain, and improve systems with fewer cascading failures.

We have also identified three themes that we believe can help engineers to design better systems. "Ungraceful degradation" discusses existing fault-tolerance strategies and some of their weaknesses. It also argues for the need to create requirements documents that detail how a system should handle a degradation in service. "Automating failure" reviews how auxiliary systems can initiate or contribute to cascading failures. It also reiterates the need to check assumptions before blindly implementing counterproductive measures, which may lead to additional cascading failures. "Unwinding from failure" covers the need to design components that are as independent as possible from the rest of the system, in order to simplify and accelerate recovery when a failure occurs.

All of these are tasks that initially fall to system designers. But as systems change over time, these principles also need to be checked regularly by the engineers to ensure continued conformity.

References

[1] Jeff Ash and David Newth. Optimizing complex networks for resilience against cascading failure. *Physica A: Statistical Mechanics and its Applications*, 380:673 – 683, 2007.

[2] Algirdas Avizienis, J-C Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE transactions on dependable and secure computing*, 1(1):11–33, 2004.

[3] João R. Campos and Ernesto Costa. Fault injection to generate failure data for failure prediction: A case study. In *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*, pages 115–126, 2020.

[4] Chris Clearfield and András Tilcsik. *Meltdown: What Plane Crashes, Oil Spills, and Dumb Business Decisions Can Teach Us About How to Succeed at Work and at Home*. Penquin Books, 2018.

[5] Bonnie Collier, Tom DeMarco, and Peter Fearey. A defined process for project post mortem review. *IEEE Software*, 13(4):65–72, 1996.

[6] Juliet Corbin and Anselm Strauss. *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory*. SAGE Publications, Inc, 3 edition, 2007.

[7] Domenico Cotroneo, Michael Grottke, Roberto Natella, Roberto Pietrantuono, and Kishor S. Trivedi. Fault triggers in open-source software: An experience report. In *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*, pages 178–187, 2013.

[8] John W. Creswell and Chreyl N. Poth. *Qualitative Inquiry '&' Research Design: Choosing Among Five Approaches*. Sage Publications, 2018.

[9] Eric M. Dashofy, André van der Hoek, and Richard N. Taylor. Towards architecture-based self-healing systems. In *Proceedings of the First Workshop on Self-Healing Systems*, WOSS '02, page 21–26, New York, NY, USA, 2002. Association for Computing Machinery.

[10] Murielle Florins and Jean Vanderdonckt. Graceful degradation of user interfaces as a design method for multiplatform systems. In *Proceedings of the 9th International Conference on Intelligent User Interfaces*, IUI '04, page 140–147, New York, NY, USA, 2004. Association for Computing Machinery.

[11] Amal Ganesh, M. Sandhya, and Sharmila Shankar. A study on fault tolerance methods in cloud computing. In *2014 IEEE International Advance Computing Conference (IACC)*, pages 844–849, 2014.

[12] Debanjan Ghosh, Raj Sharman, H. Raghav Rao, and Shambhu Upadhyaya. Self-healing systems – survey and synthesis. *Decision Support Systems*, 42(4):2164–2185, 2007. Decision Support Systems in Emerging Economies.

[13] Chris Gill, James Orr, and Steven Harris. Supporting graceful degradation through elasticity in mixed-criticality federated scheduling. In *Proc. 6th Workshop on Mixed Criticality Systems (WMC), RTSS*, pages 19–24, 2018.

[14] Michael Grottke, Allen P. Nikora, and Kishor S. Trivedi. An empirical investigation of fault types in space mission system software. In *2010 IEEE/IFIP International Conference on Dependable Systems Networks (DSN)*, pages 447–456, 2010.

[15] Maurice P. Herlihy and Jeannette M. Wing. Specifying graceful degradation. *IEEE Transactions on Parallel and Distributed Systems*, 2(1):93–104, 1991.

[16] Qiao Huang, Xin Xia, and David Lo. Supervised vs unsupervised models: A holistic look at effort-aware just-in-time defect prediction. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 159–170, 2017.

[17] Dominik Kesim, Andre van Hoorn, Sebastian Frank, and Matthias Häussler. Identifying and prioritizing chaos experiments by using established risk analysis techniques. In *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*, pages 229–240, 2020.

[18] Zhengping Li, Cheng Hwee Sim, and Malcolm Yoke Hean Low. A survey of emergent behavior and its impacts in agent-based systems. In *2006 4th IEEE International Conference on Industrial Informatics*, pages 1295–1300, 2006.

[19] Richard G. Little. Controlling cascading failure: Understanding the vulnerabilities of interconnected infrastructures. *Journal of Urban Technology*, 9(1):109–123, 2002.

[20] Yuanqing Xia Magdi S. Mahmoud, editor. *Networked Control Systems Chapter 9 - Cyberphysical Security Methods*. Butterworth-Heinemann, 2019.

[21] Sharan B. Merriam and Elizabeth J. Tisdell. *Qualitative Research: A Guide to Design and Implementation Fourth Edition*. Jossey-Bass, 2016.

[22] Microsoft. Resiliency patterns, 2017.

[23] Saurabh Mittal and Larry Rainey. Harnessing emergence: The control and design of emergent behavior in system of systems engineering. In *Proceedings of the conference on summer computer simulation*, pages 1–10, 2015.

[24] Martin Monperrus. Automatic software repair: A bibliography. *ACM Comput. Surv.*, 51(1), jan 2018.

[25] Upama Nakarmi, Rahnamay Mahshid Naeini, Md. Jakir Hossain, and Md. Abul Hasnat. Interaction graphs for cascading failure analysis in power grids: A survey. *Energies*, 13(9), 2020.

[26] Masashi Narumoto, Alex Homer, John Sharp, Larry Brader, and Trent Swanson. *Cloud Design Patterns: Prescriptive Architecture Guidance for Cloud Applications*. Microsoft, 2014.

[27] Charles Perrow. *Normal Accidents: Living with High-Risk Technologies*. Princton University Press, 1999.

[28] Ahmed Shehata and Ashraf EL Damatty. Numerical model of cascade failure of transmission lines under downbursts. *Engineering Structures*, 255:113876, 2022.

[29] Michael E. Shin and Daniel Cooke. Connector-based self-healing mechanism for components of a reliable system. In *Proceedings of the 2005 Workshop on Design and Evolution of Autonomic Application Software*, page 1–7, New York, NY, USA, may 2005. Association for Computing Machinery.

[30] Jonathan Sillito and Esdras Kutomi. Failures and fixes: A study of software system incident response. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 185–195. IEEE, 2020.

[31] Juan Alejandro Valdivia, A Lora-González, X. Limón, K. Cortes-Verdin, and Jorge Octavio Ocharán-Hernández. Patterns related to microservice architecture: a multivocal literature review. *Programming and Computer Software*, 46(8):594–608, 2020.

[32] Yong Yang, Yifan Wu, Karthik Pattabiraman, Long Wang, and Ying Li. How far have we come in detecting anomalies in distributed systems? an empirical study with a statement-level fault injection method. In *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*, pages 59–69, 2020.

[33] David Zubrow and Std Classification for Software Anomalies Working Group. IEEE standard classification for software anomalies. *IEEE Std 1044-2009 (Revision of IEEE Std 1044-1993)*, pages 1–23, 2010.