Planning for Software System Recovery by Knowing Design Limitations of Cloud-native Patterns

Alireza Hakamian[1], Floriment Klinaku[1], Sebastian Frank[1], André van Hoorn[2] and Steffen Becker[1]
[1]*Software Quality and Architecture, University of Stuttgart, Germany*
[2]*Software Engineering and Construction Methods, University of Hamburg, Germany*

Keywords: Cloud-native Patterns, Formal Method, TLA$^+$.

Abstract: *Context.* Application designers use cloud-native architectural patterns such as Circuit Breaker that come with third-party implementations to improve overall system reliability. *Problem.* Important quality decisions are hidden in the codebase and are usually not documented by third-party implementations. Runtime changes may invalidate, e.g., pattern's decision assumption(s) and cause the reliant service to face unacceptable quality degradation with no recovery plan. *Objective.* The primary goal of this study is to derive important quality decisions of patterns independent of a particular implementation. *Method.* To achieve our objective, we perform exploratory research on two architectural patterns, (1) Circuit Breaker and (2) Event Sourcing, which come with different third-party implementations and that application designers often use. We formally specify the design and the guarantees of each pattern using Temporal Logic of Actions (TLA) and verify the guarantees, which guide us in deriving important quality decisions. *Result.* To show the usefulness of our method, we systematically generate failure scenarios for third-party implementations of Circuit Breaker and Event Sourcing patterns that compromise *Hystrix*' and *Kafka*'s guarantees on preventing further degradation of protected services and the loss of committed messages, respectively. *Conclusion.* The result suggests that important quality decisions derived from formal models of the patterns help application designers prepare for unacceptable system quality degradation by knowing when a third-party implementation of the architectural patterns fails to maintain its guarantees.

1 INTRODUCTION

Continuous change of system configurations, e.g., new deployments, is the important characteristic of cloud-native software systems (Davis, 2019). To cope with the impact of changing environments, application designers use architectural patterns, e.g., Circuit Breaker, or Event Sourcing implemented by *Resilience4j* (Resilience4j Contributors, 2021) (for Java programmers) and Kafka (Kafka Team, 2021) respectively. However, important quality decisions and related assumptions remain hidden in the codebase or informal description (high-level language) in the respective official third-party documentation, books, or blogs. Real-world incidents such as (Lianza and Snook, 2020) have shown a lack of understanding of pattern's quality decisions, and related assumptions implemented by a third-party component come with the consequence of unacceptable system quality degradation.

Therefore, our primary goal is to derive important quality decisions in cloud-native architectural patterns independent of a particular implementation, which benefits the application designers in planning to retain the system quality when a third-party implementation of an architectural pattern fails to maintain the pattern's guarantee. Kazman et al. (Kazman, Rick et al., 2000) described that important quality decisions are decisions that have been made by architects on sensitivity points, which are design properties that are critical to quality achievement. For example, service availability can be sensitive to the number of working nodes and the repair rate. Moreover, Kazman et al. describe the situation when no decision has been made on sensitivity points as architectural risk.

In our study, we specify and use the formal models of the architectural patterns to derive sensitivity points and their related decisions. We perform exploratory research on two architectural patterns, (1) Circuit Breaker and (2) Event Sourcing. Our research method consists of (1) selecting cloud-native patterns that come with a variety of implementations and often are used by application designers, (2) formally specifying the design and requirements of each pattern (independent of a particular implementation) using Temporal Logic of Actions (TLA)$^+$, (3) veri-

Hakamian, A., Klinaku, F., Frank, S., van Hoorn, A. and Becker, S.
Planning for Software System Recovery by Knowing Design Limitations of Cloud-native Patterns.
DOI: 10.5220/0011061500003200
In *Proceedings of the 12th International Conference on Cloud Computing and Services Science (CLOSER 2022)*, pages 215-222
ISBN: 978-989-758-570-8; ISSN: 2184-5042

fying the requirements against the design specification, which guide us in deriving sensitivity points, and (4) reviewing the design specification of the patterns for respective decision on each sensitivity point.

Mendonça et al. (Mendonça et al., 2020) use probabilistic models for resilience patterns aiming to predict the impact of a different configuration of resilience patterns on the performance and availability of the overall system. Our primary difference is in our pragmatism behind modeling architectural patterns. In order to derive important quality decisions, we consider all relevant context in the design of the architectural patterns, e.g., modeling threads in the Circuit Breaker pattern.

We derive two sensitivity points, namely, (1) the number of active threads and (2) how up-to-date is the new node joining a leader-based replication group in the design of Circuit Breaker and Event Sourcing patterns, respectively. To validate the usefulness of our method, we create failure scenarios for the *Hystrix* and *Kafka* implementations of the Circuit Breaker and Event Sourcing patterns, respectively. Our insight from the study is that: *Application designers can use the result from our study to plan for recovery from system quality failure when a third-party implementation of an architectural pattern fails to maintain its guarantees.*

We summarize our contributions as follows:

- A method to derive sensitivity point(s) from the design of cloud-native patterns systematically.
- Formal design specification of the Circuit Breaker and Event Sourcing patterns.
- Validating the usefulness of our method by constructing failure scenarios. We apply failure scenarios against *Hystrix* and *Kafka* implementations of the Circuit Breaker and Event Sourcing patterns respectively.

Due to space limitations, the paper only discusses the application of the method to the Circuit Breaker pattern and provides a condensed summary of the results regarding the Event sourcing pattern. The complete discussion on Event Sourcing pattern is available as an online appendix (Hakamian, A. et al., 2022).

2 FOUNDATION: TLA$^+$

Lamport introduced TLA mainly for design specifications of distributed and concurrent systems. The core idea is to use the language to specify both the design and the intended properties. The modeler should think of the system in terms of states and actions to specify the system's design (abstracted from the implementation details). The modeler defines the state of the system by defining one or more variables. The state of the system is defined by the values of those variables at each instance of time. An action makes the system transition from one state to the next by changing the value of one or more variables. A sequence of states is called behavior.

TLA$^+$ (Lamport, 2002) is an extension to TLA (Lamport, 1994), which makes the language suitable for writing modular specifications. In order to specify architectural pattern's goals in the form of (1) what the system must always/never do and (2) what must eventually happen, we choose TLA$^+$ as our formal language. Moreover, the language comes with tooling support, called TLA$^+$ Toolbox (Lamport, 2021).

To show the language constructs in specifying actions and states, we use the simple hour clock example provided by Lamport in his book (Lamport, 2002). We represent the hour by the variable *hour*, which takes values $[1-12]$. A sequence of states for hour clock is $hour = 1, hour = 2, \ldots$. For specifying the hour clock system, we specify the initial state of the system $Init \triangleq hour \in (1..12)$, which means the system can start at any hour. After initial state, we specify the action *tick*, which creates the sequence of intended behavior as follow: $Tick \triangleq hour' = IF\ hour \neq 12\ THEN\ hour + 1\ ELSE\ 1$. The formula is an action because it changes the value of the variable hour. The change is shown by using the $'$ symbol. We specify the complete hour clock system using the Formula $HClock \triangleq Init \wedge \Box Tick$, which describes all valid behaviors. The symbol $\Box$ is a temporal formula that asserts *tick* is true for every step in the behavior.

3 SENSITIVITY POINTS IN ARCHITECTURAL PATTERNS

To derive sensitivity points of architectural patterns systematically, we devise a method consisting of four steps (as illustrated in Figure 1), (1) pattern selection and review, (2) pattern specification, (3) verification, and (4) pattern specification review. We apply the method to the two patterns Circuit Breaker and Event Sourcing. Figure 1 shows the four steps that engineers who are experts in formal methods perform for each pattern. The outcome of the method is (1) a verifiable model and a (2) description of risk(s) and a decision on sensitivity point(s), which will be stored in the repository of formal models of the architectural patterns. In the rest of the section, we explain each step for the Circuit Breaker pattern.

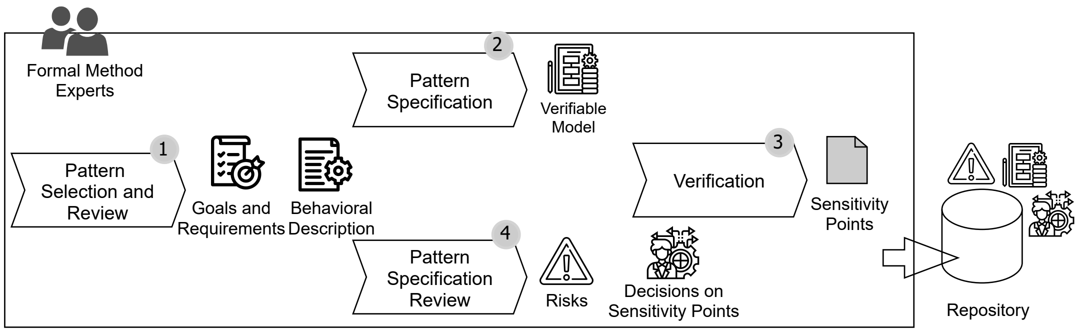

Figure 1: An overview of the method.

3.1 Pattern Selection and Review

We chose the two patterns based on the following three criteria: (1) application designers frequently use the pattern in the cloud-native domain, (2) the pattern's design requirements are not readily evident from the pattern description, and (3) the pattern should have different implementations to compare their important quality decisions. In the following we provide a high-level description of the Circuit Breaker and Event Sourcing patterns (Richardson, 2021).

Circuit Breaker. In a distributed environment, it is likely that calling a service fails, e.g., due to network issues. Resources of the caller get wasted for a non-responding service, which impacts the service's quality. The problem is known as cascading failure. The Circuit Breaker pattern solves this problem by disallowing further calls to the target service if a certain failure threshold is reached.

Event Sourcing. Data in cloud-native software systems reside on multiple and possibly different databases. Consequently, maintaining a consistent view of the whole system is problematic. Event Sourcing solves the problem by providing a single source of truth called *event storage*. Services emit events related to changes in entities and store projected data from the *event storage* locally.

After selecting the patterns, we review the behavioral description of each pattern and derive their goals and requirements. We use the Microsoft cloud-native patterns collection (Microsoft, 2022) and the *Release It!* book (Nygard, 2018) to derive high-level goals and behavioral descriptions of the Circuit Breaker pattern. The first goal is: *The pattern shall protect the client's resources from becoming exhausted.* The first goal states that the client resources shall not get wasted on requests that will most likely fail. The second goal is: *The pattern shall protect the already degraded service from further degradation.* We derive a list of requirements for the pattern considering the high-level goals and the behavioral descriptions of the Circuit Breaker pattern. However, from the list, we discuss one requirement that is important for this paper's discussion. The requirement of the Circuit Breaker is: *In the Half-Open state, the system permits only a few calls to test if the service is back.*

3.2 Pattern Specification

High-level goals, requirements and behavioral descriptions are the inputs to the pattern specification step. We chose TLA$^+$ because of the tooling support, and its adoption at industry such as Amazon (Newcombe, 2014) and MongoDB (Schvimer et al., 2020).

Regarding the specification, we present modeled variables, *initial* state formula, and requirements. The complete formal specification for each pattern is available as an online appendix (Hakamian, A. et al., 2022).

The following list describes the variables used to model the behavior of the Circuit Breaker pattern.

calls is a function that maps an OS thread $t1$ to the record of the form *[reply, permitted, permittedInState]*. The *reply* field is either *TRUE*, which means the service returns no error, *FALSE*, which means the service returns an error, or *NULL*, which means the thread has not sent any request yet. The *permitted* field shows whether (or not) the thread obtained permission to send a request. Finally, the *permittedInState* field tracks in which state the thread $t1$ obtained a permission.

circuitBreaker takes either the *CLOSED*, *OPEN*, or *HALF* value to represent the current state of the Circuit Breaker.

recordedCalls is a sequence of recorded *TRUE*, or *FALSE* when the Circuit Breaker was in either the *OPEN* or *CLOSED* state.

recordedCallsInHalf a same description as *recordedCalls* but now for the *HALF* state.

executed is a sequence of all calls against the service.

Formula 1 is the specification of the initial state of the Circuit Breaker. In the *Init* state specification, the assignment of values does not require the prime ($'$) notation over the variables in the left side of the assignment. The Circuit Breaker starts in the *Init* state, where the variable *circuitBreaker* has the *CLOSED* value.

$$\begin{aligned} Init \triangleq\ & calls = [t \in Threads \mapsto [reply \mapsto NULL, \\ & \quad permitted \mapsto NULL, \\ & \quad permittedInState \mapsto NULL] \\ & \wedge circuitBreaker = CLOSED \wedge recordedCalls = \langle\ \rangle \\ & \wedge count = 0 \wedge recordedCallsInHalfOpen = \langle\ \rangle \end{aligned} \tag{1}$$

3.3 Verification

After the specification step, we instantiate a model in the TLA$^+$ Toolbox and verify the model against requirement(s). In the following, we present the verification result for the Circuit Breaker pattern.

Formula 2 is the specification of the requirement: *In the Half-Open state, the system permits only a few calls to test if the service is back.* The requirement belongs to the second high-level goal (the Circuit Breaker shall protect the already-degraded service). The name of the formula is *NoFurtherDegradation* as the second goal suggests. The output of the *ExecInHalf* operator in Formula 2 is a sequence of all calls when the Circuit Breaker is in the *Half-Open* state. The property $NoFurtherDegradation$ checks the number of calls in the *Half-Open* state against the total number of permitted calls during the *Half-Open* state. We instantiate a model with two threads and one permitted call during a *Half-Open* state. The time to completion of the model checker is only a few seconds as the state space of the model is not large. The model checker shows that the design specification **does not** satisfy the property $NoFurtherDegradation$.

$$\begin{aligned} & ExecInHalf \triangleq LET\ Count(c) \triangleq c = HALF \\ & \qquad IN\ SelectSeq(executed, Count) \\ & NoFurtherDegradation \triangleq \\ & \quad Len(ExecInHalf) \leq PermittedCallsInHalfOpen \end{aligned} \tag{2}$$

The counterexample shows that thread $t1$ asks for permission before the Circuit Breaker switches to the *Open* state. The Circuit Breaker switches to the *Half-Open* state. The Circuit Breaker permits the thread $t2$ to gain permission during the *Half-Open* state and, together with the already permitted thread $t1$, call the protected service. Hence, the total number of executions during the *Half-Open* state is now 2, which is greater than the constant parameter *PermittedCallsInHalfOpen*.

There are two options at this stage: either changing the design to satisfy the property or relaxing the requirement, which means changing the property to pass the current design. We decided not to change the design specification as there was no notion of restricting execution in the general description of the Circuit Breaker. Formula 3 is the specification of the relaxed requirement. The *Cardinality* operator gives the number of elements in the set Threads, which is a constant. A translation of the property into the English language is *In Half-Open, the number of executions to the service is at most equal to the number of active Threads in the client.* Therefore, the edge case in the design of the Circuit Breaker pattern is that: *In a highly concurrent environment, all the active Threads gain permission before the Circuit Breaker switches to the Open state and hence further degrade the service.*

$$\begin{aligned} & NoFurtherDegradation \triangleq Len(ExecInHalf) \leq \\ & \quad PermittedCallsInHalfOpen + Cardinality(Threads) \end{aligned} \tag{3}$$

According to the definition of the sensitivity point and the help from the identified edge case, we derive the sensitivity point in the design of the Circuit Breaker pattern as:

The number of active threads.

3.4 Pattern Specification Review

We review the behavioral description and the specification of the patterns to find out about design decisions on sensitivity points. In Circuit Breaker, there is no decision on the sensitivity point, which is the risk in the design of the pattern.

4 VALIDATING USEFULNESS

This section presents a systematic generation of a failure scenario guided by the derived sensitivity point against the *Hystrix* implementation of the Circuit Breaker pattern. The aim is to validate the derived sensitivity point of the Circuit Breaker pattern. For the experiment, we cloned an example application publicly hosted on GitHub (Davis, 2020), which is provided by the author of the cloud-native patterns book (Davis, 2019). Moreover, we discuss the result of our investigation on the documentation of the third-party implementation of the Circuit Breaker pattern.

The aim is to check if the documentation explains the sensitivity point and the related decision.

4.1 Experiment on the *Hystrix*

We designed an experiment to evaluate how a protected service by the Circuit Breaker behaves in a highly concurrent environment. We evaluate whether the protected service by Circuit Breaker further degrades during the Open state in high concurrency or not. To measure further degradation during the time the Circuit Breaker is Open, we use *the number of processes waiting for CPU time in the protected service* as the metric. Another metric we use is *the average system load* that is measured by the Linux operating system. The only processes running on the database container are the MySQL process and *sar* that collects system activity information and is a tool from *sysstat* (System performance tools for the Linux operating system).

We installed *MicroK8s* on our Ubuntu machine hosted in a private cloud. The machine has 16 GB of RAM, 8 VCPUs, and 12 GB of storage. We deployed two containerized services, (1) a *Post service* responsible for creating and retrieving posts and (2) a MySQL *database service* responsible for storing and retrieving database queries. The call to retrieve posted items from the *Post service* is protected by a *Hystrix* Circuit Breaker. The configuration for the Circuit Breaker is:

- SleepTime 10 seconds. SleepTime shows how long Circuit Breaker remains open until allowing one test call.
- ThreadTimeout 1.5 seconds. The variable shows how long a thread waits for a reply from the database service.
- ThreadSize 30. We allow up to 30 concurrent threads to be activated.

We created the following scenario, which is the expected use case of a Circuit Breaker:

Scenario. We degrade the *database service* by loading the *Post service* with around 12 requests per second. The chosen number is to keep the concurrency low but at the same time degrade the database to trigger the Circuit Breaker trip between the Open and Closed states. We measure the defined metrics, and afterward, we create a highly concurrent environment by sending more than 50 requests per second. We measure the defined metrics for a highly concurrent environment.

Our hypothesis from the scenario is that:

Hypothesis. We hypothesize that when many concurrent threads are active, many threads can send requests to the database service and cause further degradation of the service.

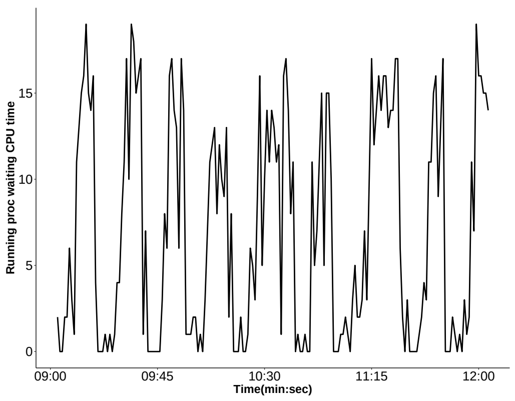

Figure 2: Queue size in low concurrent environment (constant rate 12 reqs/sec).

Figure 2 shows the behavior of the Circuit Breaker for 3 minutes in a low concurrent environment. The load to the post service's API is at a constant rate of 12 requests per second. The *Y-axis* represents the number of processes waiting for CPU time. The number drops to 0 whenever the Circuit Breaker switches to *Open*. From Figure 2 it is visible that the number of processes awaiting CPU time drops to 0 every time the Circuit Breaker is open, which means the Circuit Breaker achieves its goal that is protecting the degraded service from further degradation. However, in high concurrency(Figure 3), the line graph shows that most of the time when the Circuit Breaker is open, the number of processes awaiting CPU time is above 0.

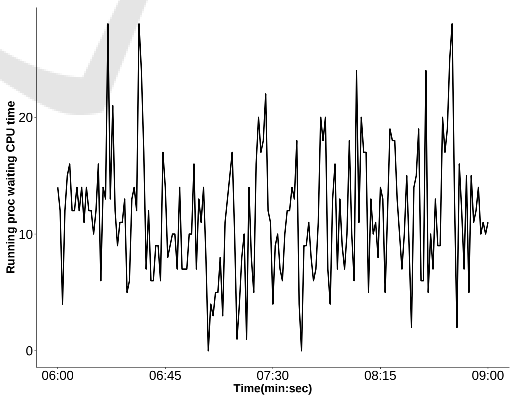

Figure 3: Queue size in highly concurrent env (> 50 req/s).

Finally, regarding the protected database service, by comparing the average system load (averaged every 1 minute) in the low concurrency and high concurrency scenarios (shown in Figure 4), we conclude that our hypothesis is true and the Circuit Breaker is, in fact, sensitive to the number of active threads de-

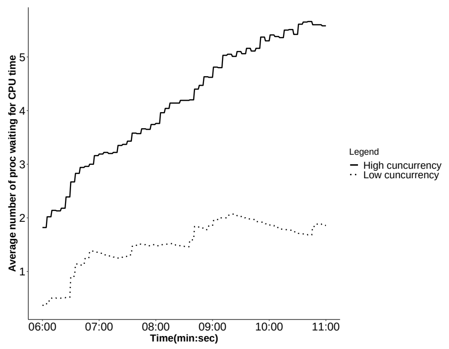

Figure 4: System load average (every 1min)).

rived from our formal analysis.

The design of the Circuit Breaker does not make any assumption on the number of active threads. Hence, failure in protecting the target service from further degradation due to the high number of active threads is possible, and application designers should have a recovery plan. Depending on the context, limiting the number of active threads may not be possible. For such a case, it is essential to first detect an error by looking at indicators such as the number of trips to Open and Closed states in the Circuit Breaker. Second, eliminate the error by using a rate limiter at the protected service during the high load.

4.2 Investigating the Documentation of Third-party Implementations

For the Circuit Breaker pattern, we looked at the documentation of *Resilience4j*, *Polly*, and *Hystrix* concerning the Circuit Breaker pattern.

The documentation of *Resilience4j* refers to a different pattern — bulkhead — to limit the number of concurrent threads. However, the documentation does not explain the sensitivity point in the Circuit Breaker design pattern and the consequences of a high concurrency level of threads, which overloads an already degraded service. Hence, *Resilience4j* has a design limitation, which is controlling the number of active threads. The documentation of *Polly* on the Circuit Breaker pattern does not provide any notion on the number of concurrent threads, its consequence, and even referring to other patterns such as bulkhead. On the other hand, the *Hystrix* design, limits the number of threads through a configuration parameter. However, documentation does not explain the consequence of a high number of active threads.

5 EVENT SOURCING PATTERN

This section summarizes the application of our proposed method on the Event Sourcing pattern.

We studied the Event Sourcing pattern from books (Kleppmann, 2016; Davis, 2019). The event storage (aka event log) design is the heart of the Event Sourcing pattern. Therefore, our focus in event sourcing is on the design of the event storage. We derive the goal that: *The event storage must guarantee the integrity of the data.* Regarding the behavioral description of the event storage, we choose the *PacificA* protocol proposed by Lin et al. (Lin et al., 2008). The main reason is that Kafka's design (Kafka Team, 2021) is based on *PacificA*. Kafka is one of the few existing platforms developers use for the Event Sourcing pattern, mainly because of Kafka's no loss of committed data guarantee that aligns with the event storage goal.

After the specification and verification steps, we derived the sensitivity point in the design of the event storage as

How up-to-date is the new node joining the replicaGroup.

Guided by the sensitivity point, we experimented on a three-node Kafka cluster by testing what happens when all nodes fail and the most recent leader does not have the most recent committed messages. In reality, it happens that data is wiped out due to hardware error (Junqueira, 2015), or human operators error (Shea, 2017). After running the experiment, we observed that the other two nodes became followers and removed all their committed data to the index of the current leader. Hence, data integrity is violated.

6 THREATS TO VALIDITY

Conclusion Validity. Deriving a set of high-level goals and specifying a design that satisfies them involves human subjective opinion. Therefore, repeating the method by someone else may result in different abstractions for individual patterns and, hence, different sensitivity point(s). To counteract this threat, we did not use our subjective opinion for deriving high-level goals and the design. Instead, we use the literature for deriving goals and the design of each pattern.

Internal Validity. One threat is that the derived goals and requirements for the two architectural patterns are not valid. To counteract this threat, we chose the well-known books *Release It!* (Nygard, 2018; Kleppmann,

2016) for Circuit Breaker, and Event Sourcing patterns, respectively.

Construct Validity. The main threat under this category is mono-method bias, which means we did not use other methods e.g., reviewing codebases to derive sensitivity point(s). We can not entirely rule out this threat.

External Validity. One aspect of external validity is to what degree the proposed method is applicable for other cloud-native architectural patterns. We use TLA$^+$ for the design specification, and hence we support reliability and availability kind of quality guarantees in architectural patterns. Although, using other modeling languages should not be an issue. However, it requires further investigation. Another aspect is finding a suitable design abstraction that the model solver solves in a reasonable time and space.

7 RELATED WORK

We derived sensitivity points of the two architectural patterns systematically. Research areas that we fit into are (1) requirements engineering and documenting system design for the purpose of quality analysis, and (2) reliability testing of software systems by fault injection.

Regarding the first area we mention applying formal methods to reason about software design properties. Vergara et al. (Vergara et al., 2020) propose formalizing microservice architectural patterns such as Circuit Breaker using the Event-B language. They argue that describing a pattern's behavior through imprecise language may lead to ambiguity. Therefore, they use Event-B to formalize few well-known patterns in a microservice-based application, including Circuit Breaker. In the paper, the authors only discussed the Circuit Breaker pattern. However, in their online appendix, there is also the specification of service registry pattern.

Mendonça et al. (Mendonça et al., 2020) propose a model-based analysis of resilience patterns such as Retry and Circuit Breaker. The authors argue that an informal description of resilience patterns causes difficulties for application designers to know a set of proper configurations for each pattern, which improves the availability and performance of the microservice-based system. The core idea is to (1) specify the design of resilience patterns using Continuous Time Markov Chain (CTMC) in the PRISM tool (Marta Z. Kwiatkowska et al., 2011), (2) specify the availability and performance (execution time and contention time) of an exemplary system through reward functions, and finally (3) simulate the final model with different values of model parameters to obtain availability and performance measurements.

Very close to Mendonça et al. is the work by Jagadeesan et al. (Jagadeesan and Mendiratta, 2020). The authors raise the challenge in trade-off analysis in microservice-based applications where service meshes such as *Istio* (Istio Contributors, 2021) are being used. The paper focuses on the Circuit Breaker implementation in service mesh and discusses the necessity in the analysis of different trade-offs concerning the impact of slow or quick detection of a failure in downstream services and activating the Circuit Breaker on availability and the number of processed requests. To this end, the authors propose a modeling framework for such analysis based on CTMC in PRISM.

Compared to the three previous works, our primary difference is our pragmatism behind modeling architectural patterns. The modeling goal in the first work is to make the specification non-ambiguous. In contrast, the goal in the second paper is to predict the impact of a different configuration of resilience patterns on performance and availability. In the last work, the goal is a trade-off analysis of microservice-based applications. Our work provides *different abstraction*. The goal of the modeling is to derive important quality decisions in the design of the architectural patterns. We consider all assumptions that individual architectural patterns make regarding the environment, e.g., modeling threads in the Circuit Breaker pattern.

The importance of explicit assumptions in the process of documenting requirements and architecture has been discussed by Lago et al. (Lago and van Vliet, 2005). The authors present an exploratory research to investigate the importance of documenting assumptions explicitly. They suggest three important applications of having explicit assumptions in the documentation process of the architecture that are Traceability, Assessment and Knowledge management. The authors propose feature modeling for documenting assumptions explicitly. However, their method to come to assumptions most likely is brainstorming sessions.

Regarding the second research area, the core idea is to inject fault(s) into the system implementation while assuming that the system holds specific properties after the injection phase. An example is *Jepsen* (Jepsen Contributors, 2022), which have been mainly used by application designers for testing the correctness of distributed systems. The approach has successfully found edge cases in production systems such as etcd, Consul (Kingsbury, 2014) and even the old version of Kafka.

8 CONCLUSION

We discussed our systematic method in deriving sensitivity points of the Circuit Breaker and Event Sourcing, patterns, independent of a particular implementation. Through experiments on *Hystrix* and *Kafka*, we demonstrated the usefulness of our method in deriving important quality decisions of architectural patterns. The repository of the formal models of the architectural patterns helps the application designers prepare for unacceptable system quality degradation when a third-party implementation of the architectural patterns fails to maintain its guarantees.

We extend our method to perform a conformance check of third-party implementations against their formal design as ongoing work.

ACKNOWLEDGMENT

This research has received funding from the Baden-Württemberg Stiftung (Orcas project) and the German Federal Ministry of Education and Research (Software Campus 2.0 — Microproject: DiSpel).

REFERENCES

Davis, C. (2019). *Cloud Native Patterns: Designing Change-tolerant Software*. Manning Publications.

Davis, C. (2020). Cloud-native example application. https://github.com/cdavisafc/cloudnative-abundantsunshine. Accessed: 2022-January.

Hakamian, A. et al. (2022). Formal models of cloud-native patterns. https://doi.org/10.5281/zenodo.5905810.

Istio Contributors (2021). istio documentation. https://istio.io/latest/docs/. Accessed: 2022-January.

Jagadeesan, L. J. and Mendiratta, V. B. (2020). When failure is (not) an option: Reliability models for microservices architectures. In *ISSRE Workshops*, pages 19–24. IEEE.

Jepsen Contributors (2022). Jepsen. https://github.com/jepsen-io/jepsen. Accessed: 2021-January.

Junqueira, F. (2015). Disk error. https://fpj.systems/2015/05/28/dude-wheres-my-metadata/. Accessed: 2022-January.

Kafka Team (2021). Kafka design documentation. https://kafka.apache.org/documentation/#design. Accessed: 2022-January.

Kazman, Rick et al. (2000). Atam: Method for architecture evaluation. Technical report, Carnegie-Mellon Univ Pittsburgh PA Software Engineering Inst.

Kingsbury, K. (2014). Jepsen: etcd and consul. https://aphyr.com/posts/316-call-me-maybe-etcd-and-consul. Accessed: 2021-January.

Kleppmann, M. (2016). *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. O'Reilly.

Lago, P. and van Vliet, H. (2005). Explicit assumptions enrich architectural models. In *27th International Conference on Software Engineering (ICSE 2005)*, pages 206–214. ACM.

Lamport (2021). The TLA+ toolbox. https://lamport.azurewebsites.net/tla/toolbox.html. Accessed: 2022-January.

Lamport, L. (1994). The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16(3):872–923.

Lamport, L. (2002). *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley.

Lianza, T. and Snook, C. (2020). Cloudflare incident report. https://blog.cloudflare.com/a-byzantine-failure-in-the-real-world/. Accessed: 2022-January.

Lin, W., Yang, M., Zhang, L., and Zhou, L. (2008). Pacifica: Replication in log-based distributed storage systems.

Marta Z. Kwiatkowska et al. (2011). PRISM 4.0: Verification of probabilistic real-time systems. In *Computer Aided Verification - 23rd International Conference, CAV 2011*, pages 585–591. Springer.

Mendonça, N. C., Aderaldo, C. M., Cámara, J., and Garlan, D. (2020). Model-based analysis of microservice resiliency patterns. In *2020 IEEE International Conference on Software Architecture, ICSA 2020, Salvador, Brazil, March 16-20, 2020*, pages 114–124. IEEE.

Microsoft (2022). Microsoft classification of design patterns in cloud-native application domain. https://docs.microsoft.com/en-us/azure/architecture/patterns/index-patterns. Accessed: 2022-January.

Newcombe, C. (2014). Why amazon chose TLA +. In *Abstract State Machines, Alloy, B, TLA, VDM, and Z - 4th International Conference, ABZ 2014*, pages 25–39. Springer.

Nygard, M. T. (2018). *Release it!: design and deploy production-ready software*. Pragmatic Bookshelf.

Resilience4j Contributors (2021). Resilience4j documentation. https://resilience4j.readme.io/docs. Accessed: 2022-January.

Richardson, C. (2021). Patterns for microservice architectural style. https://microservices.io/patterns/. Accessed: 2021-October.

Schvimer, J., Davis, A. J. J., and Hirschhorn, M. (2020). extreme modelling in practice. *Proc. VLDB Endow.*, 13(9):1346–1358.

Shea, C. (2017). Gitlab incident report. https://about.gitlab.com/blog/2017/02/10/postmortem-of-database-outage-of-january-31/. Accessed: 2022-January.

Vergara, S., González, L., and Ruggia, R. (2020). Towards formalizing microservices architectural patterns with event-b. pages 71–74. IEEE.