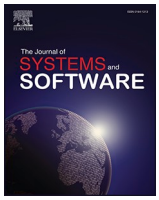
The Journal of Systems and Software 217 (2024) 112168

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss



Real-Time rejuvenation scheduling for cloud systems with virtualized software spares

Joshua R. Carberry, John Rahme, Haiping Xu *

Computer and Information Science Department, University of Massachusetts Dartmouth, Dartmouth 02747, USA

ARTICLE INFO

Keywords:
Cloud systems
Virtualized software spares
Rejuvenation schedule
Reliability analysis
Dynamic fault tree

ABSTRACT

With the increasing popularity of cloud services, there is a growing demand for high reliability and availability of cloud computing. As viable solutions, virtualized software spares and rejuvenation scheduling have been used to maintain highly reliable software platforms and combat Mandelbugs in cloud systems. However, developing real-time rejuvenation schedules for software components with dynamic reliability models has been a challenging task. In this paper, we propose a hybrid approach that integrates preventive and automatic failover strategies to mitigate the harmful effects of Mandelbugs. The approach allows selecting reliability models based on the state of virtualized software components, performing reliability calculations for Software SPare (SSP) gates with up to two virtual hot spares, and scheduling software rejuvenation in real time for cloud systems. Furthermore, the use of Dynamic Fault Tree (DFT) analysis supports the compositional modeling of complex and interconnected systems, alleviating the problem of state-space explosion. Finally, we present a case study of a cloud system with virtualized software spares to demonstrate how rejuvenation schedules can be generated and updated in real time.

1. Introduction

Software reliability and dependability are important issues for software systems, especially those responsible for critical applications. For example, momentary interruptions in electronic banking, military, or healthcare applications can have serious consequences. Therefore, reliability and dependability have become an important consideration in the design of critical software systems. In recent years, cloud computing has grown in scope and popularity, becoming an integral part of almost all popular applications (Morkos, 2023). The cloud computing model is an attractive and increasingly accessible option whenever remote storage or computation is required. The use of cloud computing in applications as diverse as social media and healthcare systems attests to the steady growth in its applicability. As cloud computing continues to evolve as a computing and services paradigm, it is increasingly being used in critical applications such as healthcare systems, online banking systems, railroad/aircraft operations and control systems, and power systems (Boudlal et al., 2022; Vinoth et al., 2022; Yao and Hao, 2023; Fang et al., 2016). For those applications, guaranteed dependability and Quality of Service (QoS) is essential. To fulfill this need, a great deal of research has focused on enhancing fault tolerance and improving the availability of cloud systems. One of the main aspects, i.e., hardware reliability, has been extensively explored and addressed through various techniques (Rausand and Høyland, 2004). The most common strategy is to use redundant hardware nodes in order to take over and ensure high availability in case of failure. However, while hardware-level redundancy is effective in mitigating permanent hardware failures, transient and intermittent failures still pose a risk, leading to data corruption and other problems. Similarly, achieving software reliability for complex systems remains a major challenge. Despite extensive theoretical foundations and practical advances in this area, software systems remain vulnerable to software bugs, unexpected interactions, and environmental changes that can compromise their reliability (Pham, 2006; Somani and Vaidya, 1997). Addressing these issues requires continuous innovation and rigorous validation to ensure that software components function consistently and correctly under various operating conditions.

According to software reliability engineering (SRE) practices, software bugs are usually divided into two groups: Bohrbugs and Mandelbugs (Grottke and Trivedi, 2005; Grottke et al., 2016). Bohrbugs are deterministic bugs caused by design faults, which are easily reproducible, and as such are best approached with testing and manual debugging as part of software maintenance. On the other hand, Mandelbugs

* Corresponding author.
E-mail address: hxu@umassd.edu (H. Xu).

https://doi.org/10.1016/j.jss.2024.112168
Received 15 February 2024; Received in revised form 25 May 2024; Accepted 25 July 2024
Available online 26 July 2024

are nondeterministic bugs of unknown and complex origin, which are difficult to reproduce and fix using traditional testing and debugging methods. As systems become increasingly complex, their vulnerability to Mandelbugs can increase. This is especially true for cloud systems with many interacting components, where the possible system states grow exponentially, increasing the likelihood of Mandelbugs. Therefore, it is crucial to address Mandelbugs in cloud computing using software reliability techniques such as failover and software rejuvenation.

To increase the fault tolerance of software systems, we adopt virtualized software spares that can be used to replace failing software components. We also employ software rejuvenation, another means of bolstering system reliability, which may effectively prevent the occurrence of Mandelbugs including aging-related bugs. This process calls for the termination of affected applications or the restart of virtual machines running in a software environment to reset their internal states and revert any undesirable conditions such as software aging that may have occurred. For cloud systems supporting critical applications such as e-commerce and online banking, software rejuvenation is essential to combat software aging in cloud computing despite the need for high uptime. Although it does not introduce any changes to the underlying software, the reset provided by rejuvenation can alter the software environment sufficiently to prevent critical Mandelbugs (Qiu et al., 2021). The main complication of applying software rejuvenation is *when* to use it. A shortage of rejuvenations can cause a system to fall below desired thresholds for reliability and QoS. However, excessive rejuvenations can also be costly, consuming both system and potentially human resources to restart the software system. In addition, a zero-downtime rejuvenation requires an old component to continue running to complete its current task; meanwhile, a new replacement component boots and starts to accept new tasks. The simultaneous execution of the old and new components will inevitably incur additional computational overhead until the current task is completed. A reasonable rejuvenation schedule must dynamically adjust to changing system conditions to maintain high fault tolerance of the system while avoiding the needless expense of excessive rejuvenations. A feasible strategy is to rejuvenate the software system when its reliability falls below a given threshold, e.g., 99%. Necessarily, this strategy requires calculating the reliability of the system in order to compare it with the given threshold. For this reason, a comprehensive system reliability analysis is at the core of the approach. With these calculations and schedules in place, a system gains a numerical measure and a guarantee of reliability, as required by critical systems in need of quantifiably and consistently high reliability.

In this paper, we propose a novel approach to handling and preventing the appearance of Mandelbugs using virtualized software spares and real-time rejuvenation scheduling. The main contribution of the research is to develop a systematic approach to integrating preventive and automatic failover strategies and mitigating the harmful effects of Mandelbugs in software systems running on the cloud. This is done by incorporating multiple software spares, selecting dynamic reliability models based on the states of virtualized software components, and scheduling software rejuvenations in real time. Previous work introduced reliability analysis methods and rejuvenation scheduling for software systems with virtualized software spares (Rahme and Xu, 2017a; Rahme and Xu, 2015). This work was later extended to account for components with non-constant failure rates, thus greatly improving the applicability of the approach in the real world (Rahme and Xu, 2017b). However, these studies all assumed *non-varying* lifetime distributions of the software components involved in the analysis. Once the original calculations and schedules are completed, they cannot be dynamically updated to reflect runtime changes. In practice, reliability models for software components often change due to workload or other operational conditions, including the harmful effects of Mandelbugs. Therefore, in our new approach to software system reliability analysis, we consider *dynamic* reliability models for virtualized software components. During runtime, reliability models are chosen according to the workload of software components and the system resource usage measurements to properly reflect changing reliability conditions. The adjusted reliability models are then combined using highly scalable Dynamic Fault Tree (DFT) analysis (Aslansefat et al., June 2020) to calculate the reliability of a complex software system. Unlike other software reliability modeling approaches, DFT analysis does not suffer from the state-space explosion problem associated with many-component systems, making it particularly suitable for modeling cloud systems, which often have complex architectures. Using dynamic reliability models of virtualized software components, rejuvenations can be optimally scheduled to keep system reliability above a desired threshold, save resources and maximize uptime, and thereby enhance standards of dependability and QoS of software systems like cloud services. The key novelties of this paper are summarized as follows:

1) We present a systematic approach to mitigating the harmful effects of Mandelbugs in cloud systems by integrating preventive and automatic failover strategies.
2) We use *dynamic* reliability models of virtualized software components, which were chosen using a measurement-based approach.
3) We develop an analytical method to calculate the reliability of cloud systems with virtualized software spares.
4) Finally, based on the calculated dynamic system reliability, we use a case study to demonstrate how rejuvenation schedules can be developed in real time for a cloud system with virtualized software spares.

The rest of the paper is organized as follows. Section 2 reviews the existing work related to this research. Section 3 describes a detailed procedure for calculating the reliability of cloud systems that employs dynamic reliability modeling with normal state changes and critical events, the results of which provide a foundation for effective scheduling of rejuvenation. Section 4 presents a case study to demonstrate the effectiveness and utility of our approach in various scenarios. Section 5 concludes the approach and mentions future work.

2. Related work

Bohrbugs and Mandelbugs are two major categories of bugs in software reliability engineering (Grottke and Trivedi, 2005; Grottke et al., 2016). Bohrbugs are deterministic and can be addressed with traditional debugging techniques. Mandelbugs are apparently non-deterministic bugs that present challenges for direct debugging. Software-aging-related bugs are commonly classified as a subtype of Mandelbugs, which can be the result of the accumulation of any number of initially harmless errors (Grottke and Trivedi, 2005). Such errors could be leaked memory, unterminated processes and threads, or any unknown resource leakage. Note that aging-related bugs usually do not result in software failure right away; thus, they are often not detected in the software testing stage. In practice, constant vigilance during the maintenance phase is necessary to prevent and mitigate the emergence of Mandelbugs including aging-related bugs that jeopardize the reliability of software services. As explored by Grottke et al., an effective way to prevent Mandelbugs from occurring is to increase the diversity of software environments (Grottke et al., 2016). Mandelbugs are often associated with problematic situations in the environment surrounding the software, and thus by diversifying the software environment, we may reduce the chances of getting caught up in, or trapped in, these problematic environmental states. While the authors presented important definitions and tools for Mandelbug prevention and recovery, they did not directly address practical details such as how to schedule rejuvenations or implement software-level failover. In this section, we examine related work that addresses software reliability issues through various strategies designed to mitigate Mandelbugs-related problems.

Preventive approaches in SRE have been one of the major approaches to mitigating the impact of Mandelbugs in software systems

(Vaidyanathan et al., 2002; Carrozza et al., 2013). A preventive approach diagnoses or predicts issues within the system before they can lead to the emergence of a failure-causing Mandelbug, attempting to reduce the risk of Mandelbugs that have not yet occurred. Bobbio et al. proposed a fine-grained software degradation model assuming the presence of a measurable "degradation index" by which to schedule optimal rejuvenations (Bobbio et al., 2001). In their approach, two different rejuvenation policies were used, and their effectiveness was assessed. Tantri and Murulidhar demonstrated techniques for the estimation of such an index, namely the reliability of a piece of software over time, through the analysis of failure data (Tantri and Murulidhar, 2016). They used failure data to estimate the parameters of a gamma lifetime distribution, which could then be used to predict the software's reliability over time. Vaidyanathan et al. used semi-Markov modeling to schedule system inspections, with different states representing various stages of deterioration (Vaidyanathan et al., 2002). Based on the observed level of system deterioration, the system would or would not be rejuvenated. Bruneoet al. developed a method utilizing symbolic algebra and a non-Markovian method for timing virtual machine monitor (VMM) rejuvenations, taking into account the variability in workload that changes the rate of aging of some cloud-based software systems (Bruneoet al., 2013). Dohi et al. investigated a method for scheduling rejuvenations based on interval reliability criteria (Dohi et al., 2018). They defined rejuvenation policies to maximize reliability over certain intervals and produce optimal rejuvenation schedules based on reliability analysis. The aforementioned works analyze reliability only at the system level, thus ignoring the internal complexity of systems with many components. In contrast, our approach defines a reliability model for each individual software component as well as for a set of components including a primary component and up to two spare software components. In a complex cloud system consisting of multiple interconnected components, our approach combines the computed reliability values for each connected component through DFT analysis to derive its system reliability.

There are also several related research efforts on developing a system reliability model based on component-level models. For example, Liu et al. proposed a software rejuvenation-based fault tolerance scheme to manage a multi-component cloud system (Liu et al., 2015). In their approach, rejuvenations for interrelated software components were scheduled based on aging severity or failure occurrences. Nguyen et al. modeled the reliability of a cloud center using reliability graphs, fault tree modeling, and stochastic reward nets (Nguyen et al., 2019). Using a robust theoretical framework, they were able to analyze and quantify the cloud center's performance, informing design decisions with respect to reliability and dependability. More recent research pays even closer attention to individual cloud components and how they interact, specifically to address the high count of modules and interactions present in growing paradigms like Internet of Things (IoT). Nguyen et al. conducted a similar analysis of an IoT system using a hierarchical fault tree representation (Nguyen et al., 2021). Through a well-designed theoretical framework, they analyzed and quantified the reliability of IoT systems using a range of metrics. While preventive approaches such as these can effectively model failures and uptime, they do not provide proactive measures to collect new data, reanalyze, and respond to the ever-changing conditions encountered during the operation of a typical cloud service. Unlike the existing approaches, our approach supports automatic failover modeling in cloud computing and, more importantly, incorporates dynamic reliability models and supports real-time scheduling of software rejuvenations.

Automatic failover allows for the transfer of service to a duplicate standby server upon a server failure or a service event to preserve its availability (Boles, 2011). Automatic failover helps bolster a software system against Mandelbugs with the provision of redundant components to act as replacements for components failed due to Mandelbugs. This technique is effective for Mandelbug recovery but implies a cost depending on where and how redundancy is implemented. Clustering is a popular approach that links together many physical machines to improve processing power and reliability. Ibe et al. provided a pioneering investigation of cluster availability for a system with failover capabilities (Ibe et al., 1989). Following the failure of one processor, another could take over. A continuous-time Markov chain (CTMC) was used to model availability, where the optimal number of processors can be selected. Mendiratta provided a reliability analysis approach for a clustered system using an irreducible Markov chain (Mendiratta, 1998). The major goal of the approach was to model the expected downtime of a telecommunications system with cluster failover. Koutras and Platis explored the application of software rejuvenation in a two-cluster system, employing both hot and cold standby spares (Koutras and Platis, 2006). In their approach, system modeling was carried out using a CTMC. Yang et al. examined another multi-component system using clustering, with a particular emphasis on failure dependencies between components (Yang et al., 2010). They implemented failover and scheduled rejuvenations to minimize a system cost measurement obtained through a stochastic reward net analysis. Redundancy and automatic failover through traditional clustering improve reliability, but can be expensive, as they necessitate not only software but hardware redundancy. While the above approaches address software aging bugs (a significant subtype of Mandelbugs) through automatic failover, the additional hardware expenditure is not ideal. Unlike the above approaches, our approach treats virtual machines (VMs) as low-cost *Software SPares* (SSPs) that can be readily switched on to take over the jobs of failed components. Furthermore, our approach explicitly addresses the issue of when to schedule these software components to be rejuvenated, which can be affected by events that occur during the runtime after the initial analysis.

Considerable work has been done on using VMs as low-cost SSPs. Cully et al. addressed component failures by virtualizing protected components and mirroring their states to backup hosts (Cully et al., 2008). In the event of a component failure, backup hosts could seamlessly take over execution of the compromised component, resetting to a recent checkpoint. Du and Yu proposed a technique to encapsulate running components in VMs and replicate their states frequently to a backup physical host (Du and Yu, 2009). Thus, when a primary VM crashes, a copy of the VM on the backup host can immediately take over the failed primary VM. Thein et al. used Markov modeling to calculate the availability of a virtualized server using standby VM spares and rejuvenation (Thein et al., 2007). VMs were rejuvenated on an as-needed basis, with the detection of this need left undiscussed. This work was later extended by using virtual clustering to structure rejuvenating systems to ensure high availability (Thein et al., 2008). Chang et al. assumed an exponential aging distribution and used a stochastic Petri net to study the impact of VM rejuvenation, failover, and migration on availability (Chang et al., 2018). In their approach, different combinations of rejuvenation, failover, and migration are automatically evaluated to find the most useful approach. However, parameters such as time-based rejuvenation intervals are only set according to existing standards. In this paper, we incorporate virtualized SSPs and seek to optimize the rejuvenation timer to keep the reliability of the software system above the desired minimum reliability. Our approach accounts for changes that may occur at runtime, such as the failure of a certain software component or usage fluctuations. With the occurrence of these events in mind, schedules can be recalculated in real time to adapt to the current system state. Furthermore, to mitigate the state space explosion problem in complex systems, our analysis employs extended DFTs (Boudali et al., 2009) that allows for compositional modeling of dynamic relationships, including the allocation of spare components to a primary one.

In addition to the preventive approaches previously described, much work has also been done on measurement-based approaches. Unlike preventive approaches, which operate on long-term forecasts based upon initial parameters, measurement-based approaches utilize measurements and diagnostics of software components during runtime to

improve reliability analysis. Some methods include assessing the quantity and quality of user requests or internal events. Guo et al. predicted reliability degradation due to software aging by examining the workload generated by user requests to the server (Guo et al., 2010). They calculated the damage of each component due to user requests by multiple linear regression. Meng et al. used a cumulative damage model (CDM) to analyze a system degradation process due to damage shocks and failure shocks (Meng et al., 2017). In their research, damage shocks were either internal faults in a VM or external security attacks from malicious cloud users. Other approaches take internal system measurements like CPU and memory usage, to predict ongoing trends or approaching failures. Jia et al. and Sudhakar et al. used multiple linear regression and artificial neural networks, respectively, to predict failures based upon resource usage statistics (Jia et al., 2017; Sudhakar et al., 2014). Yangzhen et al. used an improved Long Short-Term Memory (LSTM) network to predict software reliability using classical software reliability datasets (Yangzhen et al., 2017). Grottke et al. analyzed the effects on available memory of time-varying system workloads on a web server and used time series modeling to predict the future resource usage (Grottke et al., 2006). Fantechi et al. proposed an approach called Reliability Based Monitoring (RBM) for flexible runtime monitoring of software reliability in complex systems (Fantechi et al., 2022). In their approach, two reliability estimation techniques, i.e., stochastic Petri nets and reliability block diagrams (RBD), are used to estimate system reliability at runtime in order to schedule maintenance interventions. While our real-time rejuvenation scheduling approach incorporates system resource usage measurements to tune the reliability models of virtualized software components, the major focus of this research is to generate rejuvenation schedules for a complex cloud system with multiple virtualized software components. In this sense, our approach complements existing measurement-based failure detection and prediction methods and provides a scalable solution to maintain the required reliability of complex cloud systems.

3. Software reliability and analysis

Our approach is grounded in software reliability analysis, allowing us to predict the reliability of software components and systems. In the subsequent sections, we will present probabilistic models for estimating the reliability of individual components and modules, including hot sparing, for further compositional modeling using DFT.

3.1. Software reliability model

Since the Weibull distribution is one of the most widely used distributions in SRE for flexible representation of failure data, in this paper, we use the Weibull distribution, as defined in (1), to model the lifetime of virtualized software components. In particular, the Weibull distribution allows us to model components with non-constant failure rates, such as aging software components, which become more prone to failure over time.

$$f(t) = \begin{cases} \kappa\lambda^{\kappa} t^{(\kappa-1)} e^{-(\lambda t)^{\kappa}} & t \geq 0 \\ 0 & t < 0 \end{cases} \tag{1}$$

where κ is a positive shape parameter, and λ is a positive scale parameter. The parameters can be estimated using the regression model implemented in R's *survival* library as *survreg()*, which provides a maximum likelihood estimation. This estimation delivers the set of parameters $\{\kappa, \lambda\}$ with the maximum likelihood L of producing the set of n observed failure times $\boldsymbol{y} = \{y_1, y_2, \ldots, y_n\}$ as in (2).

$$L = f(\boldsymbol{y}|\kappa,\lambda) = \prod_{i=1}^{n} f(y_i|\kappa,\lambda) = \prod_{i=1}^{n} \kappa\lambda^{\kappa} y_i^{(\kappa-1)} e^{-(\lambda y_i)^{\kappa}} \tag{2}$$

The reliability and hazard functions for the Weibull distribution are given as in (3) and (4).

$$R(t) = \int_{t}^{\infty} f(t)dt = e^{-(\lambda t)^{\kappa}} \tag{3}$$

$$h(t) = \frac{f(t)}{R(t)} = \kappa\lambda^{\kappa} t^{(\kappa-1)} \tag{4}$$

Note that when $\kappa = 1$, $h(t) = \lambda$, indicating a special case where the probability density function (pdf) $f(t)$ becomes an exponential distribution, where λ is a constant failure rate.

3.2. Regular reliability calculations with virtualized software spare parts

Existing methods that use Markov modeling to assess system reliability (e.g., (Mendiratta, 1998; Koutras and Platis, 2006)) suffer from scalability issues due to the high computational cost of matrix operations necessary to derive analysis results. On the other hand, the DFT approach used in this work supports compositional modeling and is more scalable for reliability analysis using binary decision concepts. In this section, we introduce a new dynamic gate to improve the expressiveness of DFT to analyze cloud-based complex systems. We first define two types of SSP, namely the *Cold Software Spare* (CSS) and the *Hot Software Spare* (HSS) (Rahme and Xu, 2017a). A CSS is a virtual image by which a software component can be deployed and redeployed, while an HSS is a virtual machine instance instantly available to take over the operations of a failed component. This type of failover prevents the system from downtime in the event of a software component failure due to the occurrence of Mandelbugs. While any software system (cloud or local) can employ SSPs to improve fault tolerance, this paper focuses exclusively on the use of SSPs in the context of cloud computing. For a cloud server component, a request queue can be shared between the primary component and an HSS; the HSS can only start processing requests stored in the queue if the primary component fails. Since the HSS is usually kept local to its primary component, the communication overhead required to maintain this shared synchronized queue is minimal and should not affect the user experience. As the software running environment of an HSS does not have to be identical to that of the primary component, its use also increases the diversity of the software environment, making the reemergence of Mandelbugs less likely. We consider a system where the primary component P has two HSSs H_1 and H_2. To model this relationship, we define a special DFT gate: the SSP gate, as shown in Fig. 1. With the SSP gate, the primary component P is active at the start; when it fails, the first HSS H_1 is switched on. When both P and H_1 have failed, the secondary HSS H_2 is switched on. The SSP gate fails when all three of its components fail. For this reason, our probability calculations must account for the failures of each component individually, including the HSSs, which may fail before or after they are switched on. Due to increased workload, the pdf of H_1 (or H_2) changes when it has been switched on, in which case the component H_1 (or H_2) is renamed to H_1^* (or H_2^*). Consequently, both hot spares will have some pdf f_H to represent their initial "off" states with lower failure rates as well as some f_{H^*} to represent their later "on" states with much higher failure rates.

Let τ_1, τ_2 and τ_3 be the failure times of components P, H_1 and H_2,

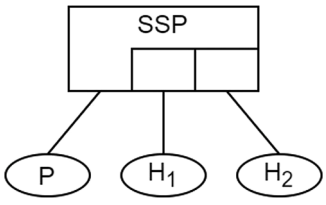


Fig. 1. An SSP gate with a primary component and two HSSs.

respectively. We use the notation $f_P(t)$ to refer to the primary component P's pdf, as in (5).

$$f_P(t) = \kappa_P \lambda_P^{\kappa_P} t^{(\kappa_P - 1)} e^{-(\lambda_P t)^{\kappa_P}} \tag{5}$$

Similarly, the notations $f_{H_1}(t)$ and $f_{H_2}(t)$ denote the initial pdfs of spare components H_1 and H_2, respectively. Once H_1 (or H_2) is switched on, it becomes H_1^* (or H_2^*) with a new pdf $f_{H_1^*}(t)$ (or $f_{H_2^*}(t)$). Based on the order of the component failures, there are six possible disjoint failure sequences (denoted as σ_1 to σ_6) that lead to the failure of the SSP gate (Rahme and Xu, 2017a; Rahme and Xu, 2015). We define a failure sequence σ as $C_1 \prec C_2 \prec C_3$, where C_1, C_2, and C_3 denote the first, second, and third components to fail, respectively. The failure sequences σ_1 to σ_6 are formally described as follows.

Failure Sequence σ_1: $P \prec H_1 \prec H_2$. In σ_1, H_1 does not fail during $(0, \tau_1]$ and H_2 does not fail during $(0, \tau_2]$. When P fails at τ_1, H_1 takes over the workload and becomes H_1^*; similarly, when H_1^* fails at τ_2, H_2 takes over the workload and becomes H_2^*. The probability that the SSP gate fails during (0, t] in failure sequence σ_1 can be calculated as in (6.1) – (6.5).

$$\Pr_{\sigma_1}(T \le t) = \int_0^t \int_{\tau_{H_1^*}}^{t_1} \int_{\tau_{H_2^*}}^{t_2} f_P(\tau_1) f_{H_1^*}(\tau_2) f_{H_2^*}(\tau_3) d\tau_3 d\tau_2 d\tau_1 \ , \tag{6.1}$$

$$\tau_{H_1^*} = \frac{(\lambda_{H_1} \tau_1)^{\left(\frac{\kappa_{H_1}}{\kappa_{H_1^*}}\right)}}{\lambda_{H_1^*}} \tag{6.2}$$

$$\tau_{H_2^*} = \frac{\left(\lambda_{H_2}\left(\tau_2 + \tau_1 - \tau_{H_1^*}\right)\right)^{\left(\frac{\kappa_{H_2}}{\kappa_{H_2^*}}\right)}}{\lambda_{H_2^*}} \tag{6.3}$$

$$t_1 = t - \tau_1 + \tau_{H_1^*} \tag{6.4}$$

$$t_2 = t - \left(\tau_2 + \tau_1 - \tau_{H_1^*}\right) + \tau_{H_2^*} \tag{6.5}$$

Note that the time-shifting for some limits of the integration is necessary to ensure that the unreliability of the HSSs is calculated properly, taking into account *both* of the pdfs of a given HSS (before and after being switched on) (Rahme and Xu, 2015). Using H_1 as an example, when it transitions from the *standby* state to an *active* state at τ_1, the pdf of H_1 changes from f_{H_1} to $f_{H_1^*}$. Without incorporating time-shifting, the unreliability after the state change would not be accurately represented, resulting in a total failure probability of less than 1.0 as $t \to \infty$. Fig. 2 illustrates the inaccuracies of this naïve (unshifted) approach. Since $f_{H_1^*}$ is a valid Weibull pdf, the probability under $f_{H_1^*}$ must be 1.0. However, using the naïve approach, the shaded probability in Fig. 2 is less than the probability under $f_{H_1^*}$, as indicated in (6.6). This discrepancy arises because, during [0, τ_1], the probability under f_{H_1} is less than that under $f_{H_1^*}$.

$$\int_0^{\tau_1} f_{H_1}(t)dt + \int_{\tau_1}^{\infty} f_{H_1^*}(t)dt < \int_0^{\tau_1} f_{H_1^*}(t)dt + \int_{\tau_1}^{\infty} f_{H_1^*}(t)dt = 1 \tag{6.6}$$

Note that a total failure probability of less than 1.0 implies a non-zero probability for the component to live *indefinitely*, which is unrealistic since a component is certain to fail eventually. To ensure that the total failure probability approaches 1.0 as $t \to \infty$, calculations involving the new pdf must be time-shifted. This adjustment can be made by modifying the integral limits for the new pdf, as shown in Fig. 3, where the new limit $\tau_{H_1^*}$ can be determined from (6.7).

$$\int_0^{\tau_{H_1^*}} f_{H_1^*}(t)dt = \int_0^{\tau_1} f_{H_1}(t)dt \tag{6.7}$$

The integration limit of $f_{H_1^*}$ in (6.1) is adjusted leftward to the new limit $\tau_{H_1^*}$, ensuring that the total failure probability (shaded area in Fig. 3) approaches to 1.0 as $t \to \infty$, as depicted in (6.8).

$$\int_0^{\tau_1} f_{H_1}(t)dt + \int_{\tau_{H_1^*}}^{\infty} f_{H_1^*}(t)dt = \int_0^{\tau_{H_1^*}} f_{H_1^*}(t)dt + \int_{\tau_{H_1^*}}^{\infty} f_{H_1^*}(t)dt = 1 \tag{6.8}$$

In the case of H_2, $\tau_{H_2^*}$ can be calculated in a similar way. However, since the limits for the integration of $f_{H_1^*}$ have been adjusted by $\left(\tau_1 - \tau_{H_1^*}\right)$, when calculating $\tau_{H_2^*}$, the activation time of H_2 shall be first restored using $\tau_{2\_restored} = \left(\tau_2 + \tau_1 - \tau_{H_1^*}\right)$ before applying the time shifting. The new limit $\tau_{H_2^*}$ can be calculated by solving the equation in (6.9), which leads to (6.3).

$$\int_0^{\tau_{H_2^*}} f_{H_2^*}(t)dt = \int_0^{\tau_{2\_restored}} f_{H_2}(t)dt \tag{6.9}$$

Note that to keep the same interval lengths for the integrations of H_1^* and

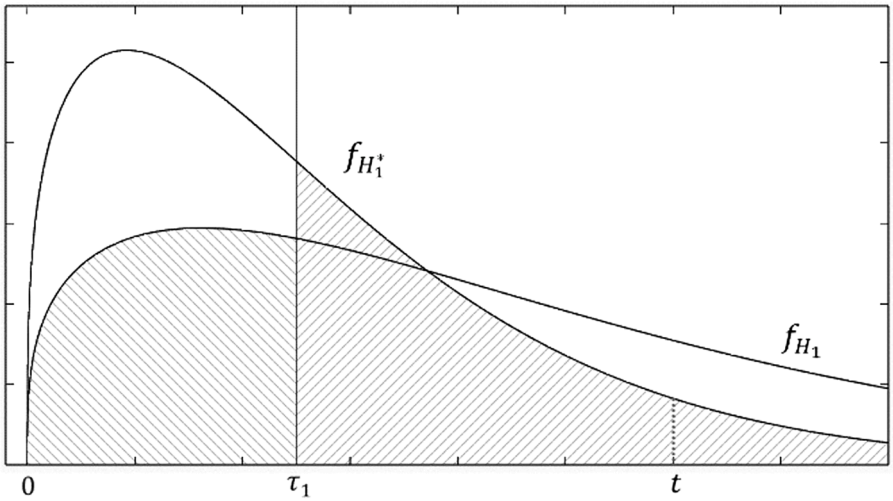


Fig. 2. The naïve (unshifted) approach for calculating the unreliability of an HSS.

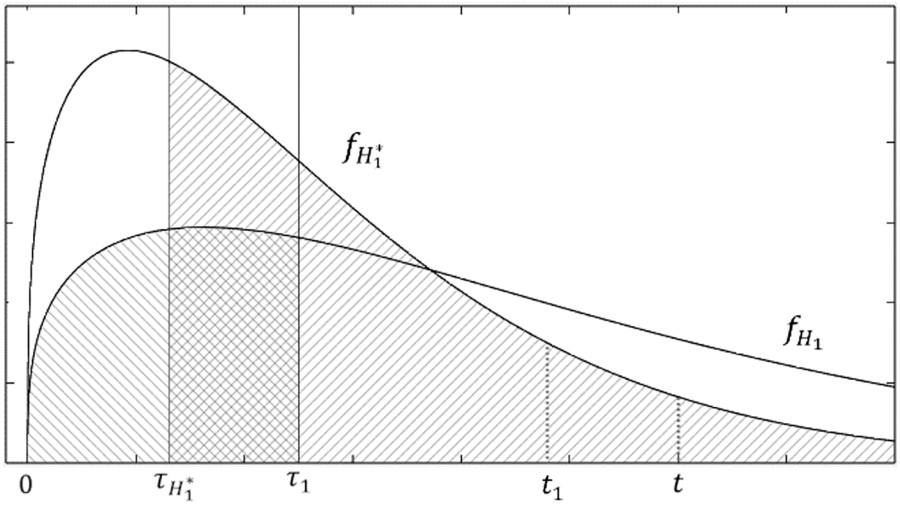


Fig. 3. The correct shifted approach for calculating the unreliability of an HSS.

H_2^*, the upper limits of integration t_1 and t_2 have also been shifted accordingly as in (6.4) and (6.5).

Failure Sequence σ_2: $P \prec H_2 \prec H_1$. In σ_2, H_2 does not have a chance to be switched on as it fails before H_1 fails. Since in σ_2, H_1^* does not fail while H_2 is still alive, the integration of H_1^* requires shifting the lower limit from $\tau_{H_1^*}$ to t_3 as in (7.2). The probability that the SSP gate fails during $(0, t]$ in σ_2 can be calculated as in (7.1) – (7.2), with $\tau_{H_1^*}$ defined in (6.2) and t_1 defined in (6.4).

$$\Pr_{\sigma_2}(T \le t) = \int_0^t \int_{\tau_1}^t \int_{t_3}^{t_1} f_P(\tau_1) f_{H_2}(\tau_3) f_{H_1^*}(\tau_2) d\tau_2 d\tau_3 d\tau_1 \tag{7.1}$$

$$t_3 = \tau_{H_1^*} + (\tau_3 - \tau_1) \tag{7.2}$$

Failure Sequence σ_3: $H_1 \prec P \prec H_2$. In σ_3, H_1 does not have a chance to be switched on, and H_2 is switched on when P fails at τ_1. Thus, the probability that the SSP gate fails during $(0, t]$ can be calculated as in (8.1) – (8.3).

$$\Pr_{\sigma_3}(T \le t) = \int_0^t \int_{\tau_2}^t \int_{\tau'_{H_2^*}}^{t_4} f_{H_1}(\tau_2) f_P(\tau_1) f_{H_2^*}(\tau_3) d\tau_3 d\tau_1 d\tau_2 \tag{8.1}$$

$$\tau'_{H_2^*} = \frac{(\lambda_{H_2}\tau_1)\left(\frac{\kappa_{H_2}}{\kappa_{H_2^*}}\right)}{\lambda_{H_2^*}} \tag{8.2}$$

$$t_4 = t - \tau_1 + \tau'_{H_2^*} \tag{8.3}$$

Note in (8.2), $\tau'_{H_2^*}$ differs from $\tau_{H_2^*}$ as in (6.3) because in σ_3, H_2 fails after τ_1 when P fails; while in σ_1, H_2 fails after $\tau_{2\_restored}$ when H_1 fails.

Failure Sequence σ_4: $H_1 \prec H_2 \prec P$. In σ_4, neither H_1 nor H_2 has a chance to be switched on, and P does not fail during $(0, \tau_3]$. Thus, the probability that the SSP gate fails during $(0, t]$ can be calculated as in (9).

$$\Pr_{\sigma_4}(T \le t) = \int_0^t \int_{\tau_2}^t \int_{\tau_3}^t f_{H_1}(\tau_2) f_{H_2}(\tau_3) f_P(\tau_1) d\tau_1 d\tau_3 d\tau_2 \tag{9}$$

Failure Sequence σ_5: $H_2 \prec P \prec H_1$. In σ_5, H_2 does not have a chance to be switched on, and H_1 is switched on when P fails at τ_1. Thus, the probability that the SSP gate fails during $(0, t]$ can be calculated as in (10), with $\tau_{H_1^*}$ and t_1 defined in (6.2) and (6.4), respectively.

$$\Pr_{\sigma_5}(T \le t) = \int_0^t \int_{\tau_3}^t \int_{\tau_{H_1^*}}^{t_1} f_{H_2}(\tau_3) f_P(\tau_1) f_{H_1^*}(\tau_2) d\tau_2 d\tau_1 d\tau_3 \tag{10}$$

Failure Sequence σ_6: $H_2 \prec H_1 \prec P$. In σ_6, neither H_1 nor H_2 has a chance to be switched on, and P does not fail during $(0, \tau_2]$. Thus, the probability that the SSP gate fails during $(0, t]$ can be calculated as in (11).

$$\Pr_{\sigma_6}(T \le t) = \int_0^t \int_{\tau_3}^t \int_{\tau_2}^t f_{H_2}(\tau_3) f_{H_1}(\tau_2) f_P(\tau_1) d\tau_1 d\tau_2 d\tau_3 \tag{11}$$

The reliability function for a SSP gate with 2 HSSs is $R(t) = 1 - U(t)$, where $U(t)$ is the unreliability given by the summation of the unreliability values of the failure sequence σ_1 to σ_6 at time t, as defined in (12).

$$U(t) = \sum_{i=1}^{6} \Pr_{\sigma_i}(T \le t) \tag{12}$$

In a special case when the primary component P has only one hot spare H, illustrated in Fig. 4, we can perform a similar derivation to obtain the unreliability equations.

Let τ_1 and τ_2 be the failure times of components P and H, respectively. Since there are only two failure sequences, namely σ_1 (P fails before H fails) and σ_2 (H fails before P fails), the corresponding equations can be derived as in (13.1) – (13.2) and (14) – (15).

$$\Pr_{\sigma_1}(T \le t) = \int_0^t \int_{\tau_{H^*}}^{(t-\tau_1+\tau_{H^*})} f_P(\tau_1) f_{H^*}(\tau_2) d\tau_2 d\tau_1 \tag{13.1}$$

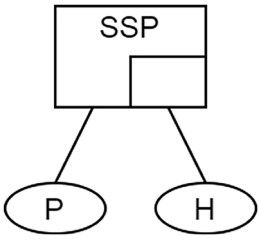


Fig. 4. An SSP gate with a primary component and one HSS.

$$\tau_{H^*} = \frac{(\lambda_H\ \tau_1)^{\left(\frac{\kappa_H}{\kappa_{H^*}}\right)}}{\lambda_{H*}} \tag{13.2}$$

$$\Pr_{\sigma_2}(T \le t) = \int_0^t \int_{\tau_2}^t f_H(\tau_2) f_P(\tau_1) d\tau_1 d\tau_2 \tag{14}$$

$$U(t) = \Pr_{\sigma_1}(T \le t) + \Pr_{\sigma_2}(T \le t). \tag{15}$$

3.3. Calculations following a normal state change

Due to the changing demands from user requests as well as the potential non-fatal effects of Mandelbugs in a cloud system, a virtualized software component may face a consistent change in computing usage. When this happens, we should update the lifetime distribution pdf of the virtualized software component to reflect this change and redo the reliability analysis for the cloud system. For this purpose, we define three states of a virtual software component, namely, *LOW, MED*, and *HIGH*, based on measures of low, medium and high *computing usage* such as CPU utilization and memory usage. The following example shows how such measurement can be done. Suppose the measurement considers only CPU utilization, memory usage, and disk i/o usage. With predefined thresholds CPU_{MED}, CPU_{HIGH}, MEM_{MED}, MEM_{HIGH}, $DISK_{MED}$, and $DISK_{HIGH}$, the raw numerical measures are discretized into the levels *LOW, MED*, and *HIGH* as defined in (16.1) – (16.3).

$$CPU(\alpha) = \begin{cases} HIGH, & if\ \alpha \ge CPU_{HIGH}, \\ MED, & if\ CPU_{HIGH} > \alpha \ge CPU_{MED} \\ LOW, & otherwise. \end{cases} \tag{16.1}$$

$$MEM(\beta) = \begin{cases} HIGH, & if\ \beta \ge MEM_{HIGH}, \\ MED, & if\ MEM_{HIGH} > \beta \ge MEM_{MED} \\ LOW, & otherwise. \end{cases} \tag{16.2}$$

$$DISK(\gamma) = \begin{cases} HIGH, & if\ \gamma \ge DISK_{HIGH}, \\ MED, & if\ DISK_{HIGH} > \gamma \ge DISK_{MED} \\ LOW, & otherwise. \end{cases} \tag{16.3}$$

where the parameters α, β, and γ represent the raw numerical measurements of CPU, memory, and disk usage, respectively. These discretized measures are integrated into a single overall measure of computing usage, where the highest individual measure determines the overall computing usage level (*HIGH, MED* or *LOW*) as defined in (16.4).

$$COMPUTING\_USAGE = \max(CPU, MEM, DISK) \tag{16.4}$$

where the function max(*CPU, MEM, DISK*) computes the highest computing usage level among the CPU, memory, and disk usage. For each virtualized component (primary component or HSS), an appropriate lifetime distribution model is then selected based on its highest computing usage level and applied in the reliability analysis and subsequent rejuvenation scheduling.

In this section, we consider the effect of a primary component state change on the reliability analysis for an SSP gate with two HSSs. Say *P*'s computing usage switches from *LOW* to *HIGH* at time t^*. After t^*, we expect reliability to degrade more quickly; however, at t^*, the reliability of *P* shall remain the same as it was in the original calculation, as the component has not yet spent any time at the higher and more unreliable level of usage. To satisfy these conditions and accurately report reliability, we split the integrals into two parts: one *before* t^*, using *P*'s old pdf $f_P(t)$, and one *after* t^*, using the pdf associated with the new usage level, denoted $f_{P^*}(t)$. To accommodate this state change, we shift $f_{P^*}(t)$ by a time interval *s*, ensuring that its area is the same as $f_P(t)$ before t^*. This is the same principle as the time shifting discussed in Section 3.2, and we will use a similar strategy here. As before, our goal is to time shift the new pdf $f_{P^*}(t)$ so that the unreliability under the original pdf f_P before the state change at t^* is correctly incorporated, and the total unreliability sums to 1.0 as $t \to \infty$. However, instead of shifting the integral limits for $f_{P^*}(t)$, we now take a simpler approach by time shifting the function $f_{P^*}(t)$ through a new function $f_{P_s^*}(t)$. To use this approach, we first equalize the unreliability of two pdfs before the state changes at t^* and move f_{P^*} by a time-shifting amount *s* as defined in (17.1).

$$\int_0^{t^*} f_P(t)dt = \int_0^{t^*} f_{P^*}(t-s)dt \tag{17.1}$$

For a component whose reliability is characterized by a Weibull distribution, we can solve Eq. (17.1) to determine the appropriate shift amount *s*. The calculated shift amount *s*, as shown in (17.2), can be applied to $f_{P^*}(t)$, as in (17.3), to produce the time-shifted pdf $f_{P_s^*}(t)$.

$$s = t^* - \frac{(\lambda_P t^*)^{\frac{\kappa_P}{\kappa_{P^*}}}}{\lambda_{P^*}} \tag{17.2}$$

$$f_{P_s^*}(t) = \begin{cases} f_{P^*}(t-s), & t \ge s \\ 0 & t < s \end{cases} \tag{17.3}$$

Fig. 5 shows the initial pdf $f_P(t)$ next to the new shifted pdf $f_{P_s^*}(t)$, with (17.1) being satisfied, i.e., the area under each curve preceding t^* being made equal. Note that the same approach applies when a component switches from *HIGH* to *LOW* usage. In this case, the resulting shift amount *s* from (17.2) is a *negative value*, causing $f_{P_s^*}(t)$ to shift to the left of the origin, as shown in Fig. 6. We can now modify the previously formulated failure sequence probability equations to integrate over f_P before t^* and over $f_{P_s^*}$ after t^* .

Let *P** be the state change event, occurring at t^* before *P* fails. For failure sequence σ_1 ($P \prec H_1 \prec H_2$), we use the notation σ_1^* to denote σ_1 with the state change event *P**, i.e., $P^* \prec P \prec H_1 \prec H_2$. With the state change, *P*'s behavior follows the initial pdf $f_p(t)$ in interval $(0, t^*]$ and the new shifted pdf $f_{P_s^*}(t)$ in interval $(t^*, t]$. Thus, the probability that the SSP gate fails during $(0, t]$ can be calculated as in (18), with t_1 and t_2 defined in (6.4) and (6.5), respectively.

$$\begin{aligned} \Pr_{\sigma_1^*}(T \le t) = &\int_0^{t^*} \int_{\tau_{H_1^*}}^{t_1} \int_{\tau_{H_2^*}}^{t_2} f_P(\tau_1) f_{H_1^*}(\tau_2) f_{H_2^*}(\tau_3) d\tau_3 d\tau_2 d\tau_1 \\ &+ \int_{t^*}^{t} \int_{\tau_{H_1^*}}^{t_1} \int_{\tau_{H_2^*}}^{t_2} f_{P_s^*}(\tau_1) f_{H_1^*}(\tau_2) f_{H_2^*}(\tau_3) d\tau_3 d\tau_2 d\tau_1 \end{aligned} \tag{18}$$

For failure sequence σ_2 ($P \prec H_2 \prec H_1$), denoted as σ_2^* with the state change before *P* fails, i.e., $P^* \prec P \prec H_2 \prec H_1$. The probability that the SSP gate fails during $(0, t]$ can be calculated in a similar way as in (19), with t_1 and t_3 defined in (6.4) and (7.2), respectively.

$$\begin{aligned} \Pr_{\sigma_2^*}(T \le t) = &\int_0^{t^*} \int_{\tau_1}^{t} \int_{t_3}^{t_1} f_P(\tau_1) f_{H_2}(\tau_3) f_{H_1^*}(\tau_2) d\tau_2 d\tau_3 d\tau_1 \\ &+ \int_{t^*}^{t} \int_{\tau_1}^{t} \int_{t_3}^{t_1} f_{P_s^*}(\tau_1) f_{H_2}(\tau_3) f_{H_1^*}(\tau_2) d\tau_2 d\tau_3\ d\tau_1 \end{aligned} \tag{19}$$

In failure sequences where *P* does not fail first, the derivations become more complex. Because *P* can change state at any time before it fails, *P** might occur *after* the failure of one or both HSSs. This leads to multiple unique sub-sequences arising from the original failure sequence due to the insertion of the state change event *P**, which can happen at various points within the original sequence. Consider failure sequence σ_3 ($H_1 \prec P \prec H_2$). Let σ_3^* be the failure sequences of σ_3 with the state

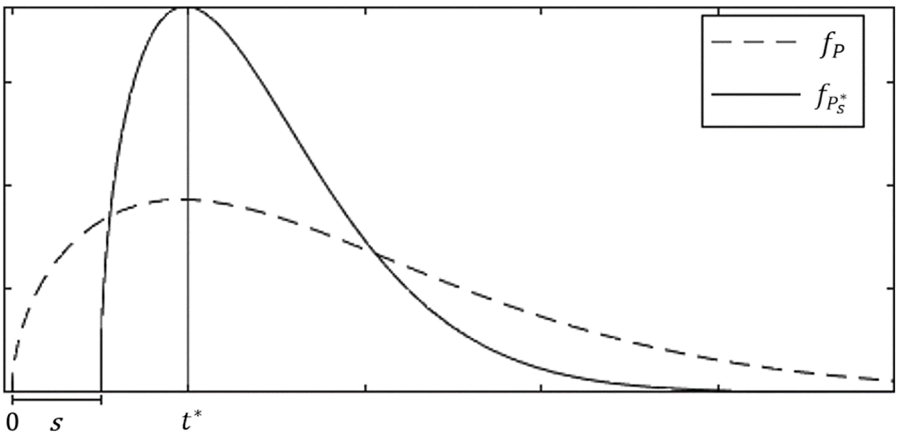


Fig. 5. P's initial pdf f_P and its new, time-shifted pdf f_{Ps^*}, after shifting from a lower usage to a higher usage state.

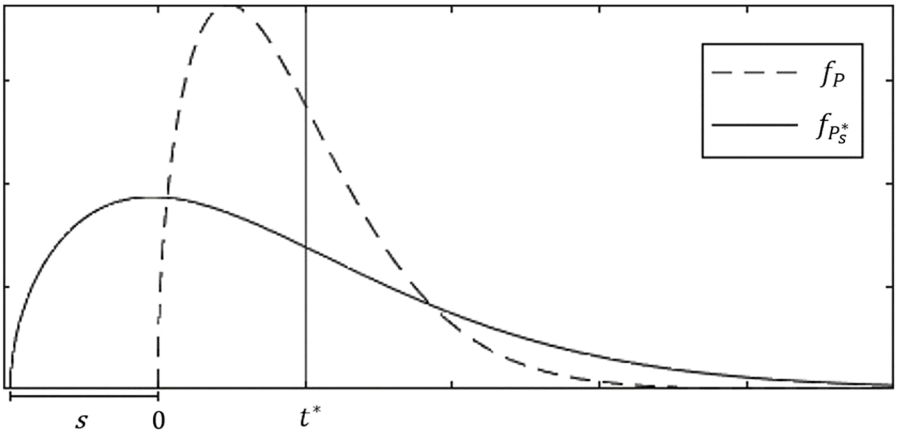


Fig. 6. P's initial pdf f_P and its new, time-shifted pdf f_{Ps^*}, after shifting from a higher usage to a lower usage state.

change event P^*. In σ_3^*, H_1 may fail before t^*, as in the sub-sequence $H_1 \prec P^* \prec P \prec H_2$. After the failure of H_1 at τ_2, P may fail in interval $(\tau_2, t^*]$ following the initial pdf $f_P(t)$, or in interval $(t^*, t]$ following the new shifted pdf $f_{P_s^*}(t)$. On the other hand, H_1 may fail after t^*, as in the sub-sequence $P^* \prec H_1 \prec P \prec H_2$. In this case, after the failure of H_1 at τ_2, P may only fail in interval $(\tau_2, t]$ following the new shifted pdf $f_{P_s^*}(t)$. Based on the above scenarios, the probability that the SSP gate fails during $(0, t]$ in σ_3^* can be calculated as in (20), with $\tau'_{H_2^*}$ and t_4 defined in (8.2) and (8.3), respectively.

$$\begin{aligned}\Pr_{\sigma_3^*}(T \le t) &= \int_0^{t^*}\int_{\tau_2}^{t^*}\int_{\tau'_{H_2^*}}^{t_4} f_{H_1}(\tau_2) f_P(\tau_1) f_{H_2^*}(\tau_3) d\tau_3 d\tau_1 d\tau_2 \\ &+ \int_0^{t^*}\int_{t^*}^{t}\int_{\tau'_{H_2^*}}^{t_4} f_{H_1}(\tau_2) f_{P_s^*}(\tau_1) f_{H_2^*}(\tau_3) d\tau_3 d\tau_1 d\tau_2 \\ &+ \int_{t^*}^{t}\int_{\tau_2}^{t}\int_{\tau'_{H_2^*}}^{t_4} f_{H_1}(\tau_2) f_{P_s^*}(\tau_1) f_{H_2^*}(\tau_3) d\tau_3 d\tau_1 d\tau_2\end{aligned} \tag{20}$$

In failure sequence σ_4 ($H_1 \prec H_2 \prec P$), both H_1 and H_2 may fail before t^*, as in the sub-sequence $H_1 \prec H_2 \prec P^* \prec P$. After the failure of H_2 at τ_3, P may fail in interval $(\tau_3, t^*]$ following the initial pdf $f_p(t)$, or in interval $(t^*, t]$ following the new shifted pdf $f_{P_s^*}(t)$. On the other hand, H_1 may fail before t^* with H_2 failing after t^*, as in the sub-sequence $H_1 \prec P^* \prec H_2 \prec P$, or both H_1 and H_2 may fail after t^*, as in the sub-sequence $P^* \prec H_1 \prec H_2 \prec P$. In the above two scenarios, after the failure of H_2 at τ_3, P may only fail in interval $(\tau_3, t]$ following the new shifted pdf $f_{P_s^*}(t)$. Based on the above scenarios, the probability of the SSP gate fails during $(0, t]$ in σ_4^* with the state change, can be calculated as in (21).

$$\begin{aligned}Pr_{\sigma_4^*}(T \le t) &= \int_0^{t^*}\int_{\tau_2}^{t^*}\int_{\tau_3}^{t^*} f_{H_1}(\tau_2) f_{H_2}(\tau_3) f_P(\tau_1) d\tau_1 d\tau_3 d\tau_2 \\ &+ \int_0^{t^*}\int_{\tau_2}^{t^*}\int_{t^*}^{t} f_{H_1}(\tau_2) f_{H_2}(\tau_3) f_{P_s^*}(\tau_1) d\tau_1 d\tau_3 d\tau_2 \\ &+ \int_0^{t^*}\int_{t^*}^{t}\int_{\tau_3}^{t} f_{H_1}(\tau_2) f_{H_2}(\tau_3) f_{P_s^*}(\tau_1) d\tau_1 d\tau_3 d\tau_2 \\ &+ \int_{t^*}^{t}\int_{\tau_2}^{t}\int_{\tau_3}^{t} f_{H_1}(\tau_2) f_{H_2}(\tau_3) f_{P_s^*}(\tau_1) d\tau_1 d\tau_3 d\tau_2\end{aligned} \tag{21}$$

The failure sequence σ_5 ($H_2 \prec P \prec H_1$) is similar to σ_3, as in both cases, P fails second. In σ_5^* with the state change, H_2 may fail before t^*, as in the sub-sequence $H_2 \prec P^* \prec P \prec H_1$. After the failure of H_2 at τ_3, P may fail in interval $(\tau_3, t^*]$ following the initial pdf $f_P(t)$, or in interval $(t^*, t]$ following the new shifted pdf $f_{P_s^*}(t)$. On the other hand, H_2 may fail after t^*, as in the sub-sequence $P^* \prec H_2 \prec P \prec H_1$. In this case, after the failure of H_2 at τ_3, P may only fail in interval $(\tau_3, t]$ following the new shifted pdf $f_{P_s^*}(t)$. Based on the above scenarios, the probability of the SSP gate fails during $(0, t]$ in σ_5^* can be calculated as in (22), with $\tau_{H_1^*}$ and t_1 defined in (6.2) and (6.4), respectively.

$$\Pr_{\sigma_5^*}(T \le t) = \int_0^{t^*}\int_{\tau_3}^{t^*}\int_{\tau_{H_1^*}}^{t_1} f_{H_2}(\tau_3) f_P(\tau_1) f_{H_1^*}(\tau_2) d\tau_2 d\tau_1 d\tau_3$$
$$+ \int_0^{t^*}\int_{t^*}^{t}\int_{\tau_{H_1^*}}^{t_1} f_{H_2}(\tau_3) f_{P_s^*}(\tau_1) f_{H_1^*}(\tau_2) d\tau_2 d\tau_1 d\tau_3 \quad (22)$$
$$+ \int_{t^*}^{t}\int_{\tau_3}^{t}\int_{\tau_{H_1^*}}^{t_1} f_{H_2}(\tau_3) f_{P_s^*}(\tau_1) f_{H_1^*}(\tau_2) d\tau_2 d\tau_1 d\tau_3$$

Finally, failure sequence σ_6 ($H_2 \prec H_1 \prec P$) is likewise similar to σ_4, as in both cases, P fails third. In σ_6^* with the state change, both H_2 and H_1 may fail before t^*, as in the sub-sequence $H_2 \prec H_1 \prec P^* \prec P$. After the failure of H_1 at τ_2, P may fail in interval $(\tau_2, t^*]$ following the initial pdf $f_P(t)$, or in interval $(t^*, t]$ following the new shifted pdf $f_{P_s^*}(t)$. On the other hand, H_2 may fail before t^* with H_1 failing after t^*, as in the sub-sequence $H_2 \prec P^* \prec H_1 \prec P$, or both H_2 and H_1 may fail after t^*, as in the sub-sequence $P^* \prec H_2 \prec H_1 \prec P$. In the above two scenarios, after the failure of H_1 at τ_2, P may only fail in interval $(\tau_2, t]$ following the new shifted pdf $f_{P_s^*}(t)$. Based on the above scenarios, the probability of the SSP gate fails during $(0, t]$ in σ_6^* can be calculated as in (23).

$$\Pr_{\sigma_6^*}(T \le t) = \int_0^{t^*}\int_{\tau_3}^{t^*}\int_{\tau_2}^{t^*} f_{H_2}(\tau_3) f_{H_1}(\tau_2) f_P(\tau_1) d\tau_1 d\tau_2 d\tau_3$$
$$+ \int_0^{t^*}\int_{\tau_3}^{t^*}\int_{t^*}^{t} f_{H_2}(\tau_3) f_{H_1}(\tau_2) f_{P_s^*}(\tau_1) d\tau_1 d\tau_2 d\tau_3$$
$$+ \int_0^{t^*}\int_{t^*}^{t}\int_{\tau_2}^{t} f_{H_2}(\tau_3) f_{H_1}(\tau_2) f_{P_s^*}(\tau_1) d\tau_1 d\tau_2 d\tau_3 \quad (23)$$
$$+ \int_{t^*}^{t}\int_{\tau_3}^{t}\int_{\tau_2}^{t} f_{H_2}(\tau_3) f_{H_1}(\tau_2) f_{P_s^*}(\tau_1) d\tau_1 d\tau_2 d\tau_3$$

The total unreliability of the SSP gate can then be calculated as in (12), with $\Pr_{\sigma_i}(T \le t)$ replaced by $\Pr_{\sigma_i^*}(T \le t)$. Note that if P experience more than one state change before a rejuvenation, the same procedure can be applied again. Furthermore, since an HSS does not take workload until it is switched on, and this type of state change is always tied to the failure of another component, such cases have already been covered in the presented failure sequences.

3.4. Calculations following a critical event

In some cases, a critical event may arise necessitating the restarting of *one* of the components of an SSP gate. For example, one of the components could unexpectedly crash, and an administrator could decide to restart it before the next scheduled rejuvenation. In addition, an anomaly could be detected in its computing usage indicating some serious issues. A detected state change may signal the appearance of a bug or other component malfunction. For example, a primary component could read a high memory usage after a Mandelbug leaks a large block of memory. Likewise, a Mandelbug could cause a still-inactivated HSS to read an abnormally high computing usage; but we expect HSSs to always remain at very low usage before activation, as they are only responsible for request queue management or data mirroring. These critical events, if detected in time, can be addressed by manually restarting the affected component to avoid the increased risk of degraded system performance and reliability. However, after a single component is manually restarted, the other components belonging to the same SSP gate will no longer share the same start time, which must be considered in the analysis. This section covers the procedure for calculating the unreliability of an SSP with two HSSs after one of its components has been restarted.

When a single component of an SSP has been restarted, the pdfs of the surviving components need to be time-shifted accordingly. Let the restarting time of this SSP component be t^*, which is now reset to 0. The pdfs of the two surviving components must be time shifted so that they start at -t^*. In this way, the same formulas can be used as in the previous sections. However, as shown in Fig. 7, the area of a surviving component's shifted pdf $f^*(t)$=$f(t + t^*)$, denoted as a solid curve, will now not approach 1 in $(0, \infty)$, as some area under pdf $f^*(t)$ has been moved to the region $(-t^*, 0]$.

To ensure the area of the pdf $f^*(t)$ from $(0, t]$ approaches 1 as t approaches infinity, we must *scale* the pdf $f^*(t)$ by a factor q defined as in (24).

$$q = \frac{1}{1 - \int_{-t^*}^{0} f^*(t)dt} = \frac{1}{1 - \int_0^{t^*} f(t)dt}. \quad (24)$$

This characterization of the factor q comes from an application of the Bayes law to $f^*(t)$, which gives the probability that the component fails during $(0, t]$ given the condition that it does not fail during $(-t^*, 0]$.

Now having the shift amount and scaling factor, we apply them to the component's original pdf $f(t)$ to obtain the new pdf $f^*_{scaled}(t)$, defined in (25).

$$f^*_{scaled}(t) = \begin{cases} q \cdot f(t + t^*) & t \ge 0 \\ 0 & t < 0 \end{cases} \quad (25)$$

This new pdf $f^*_{scaled}(t)$ for a surviving component can now be used in the unreliability formulas in the previous sections. Note that as the original restart time t^* now serves as time 0 for our new analysis, the original pdf of the restarted component is used without any adjustment.

4. Case study

To demonstrate how rejuvenation schedules can be generated and updated in real time, we present a number of typical scenarios for a simulated cloud-based electronic health record (EHR) system that requires high uptime and QoS. A cloud-based EHR system would allow users with appropriate permissions to remotely access digital copies of healthcare records, including past diagnoses and medical histories, and where properly implemented can form a major part of the healthcare workflow (Meingast et al., 2006; Bahga and Madisetti, 2013). The application may have a seasonal and/or yearly fluctuation in activity corresponding to the fluctuation of demand for personal medical information. This fluctuation of demand will naturally correspond to a fluctuation in computing usage, which calls for adaptive reliability analysis. Furthermore, with medical records and histories being such vital parts of treatment, this system would need a very high reliability to ensure the prompt treatment of patients. Therefore, we set a high system reliability threshold of 0.99; whenever the system reliability drops below this level, the system is rejuvenated.

4.1. System architecture and initial simulation

Fig. 8 shows the general architecture of the simulated cloud-based EHR system, which receives user requests via a proxy server. The proxy server sends the same stream of requests to the primary EHR application server *PA* and two of its HSSs *HA*1 and *HA*2, so that in the event of a failure, a spare component can turn on and begin to work on the currently queued requests.

As shown in the figure, the application server accesses a QoS-guaranteed database service provided by a third party, which is not considered for software reliability analysis in this case study. A performance/security monitor *PS*, equipped with a single HSS *HS*, monitors transactions, ensuring that permissions are respected, and security is maintained. The performance/security monitor is crucial to enforce the EHR system to follow the stringent regulations imposed by legislature like the health insurance portability and accountability act (HIPAA). It is

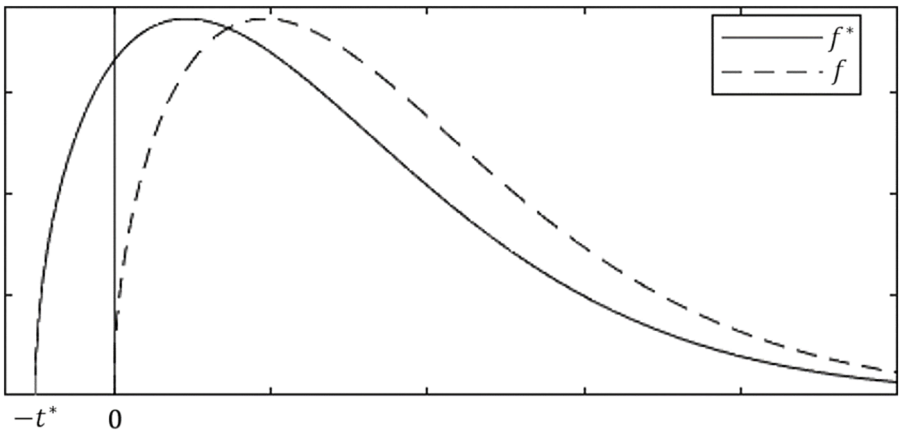


Fig. 7. The pdf of a surviving component before and after shifting.

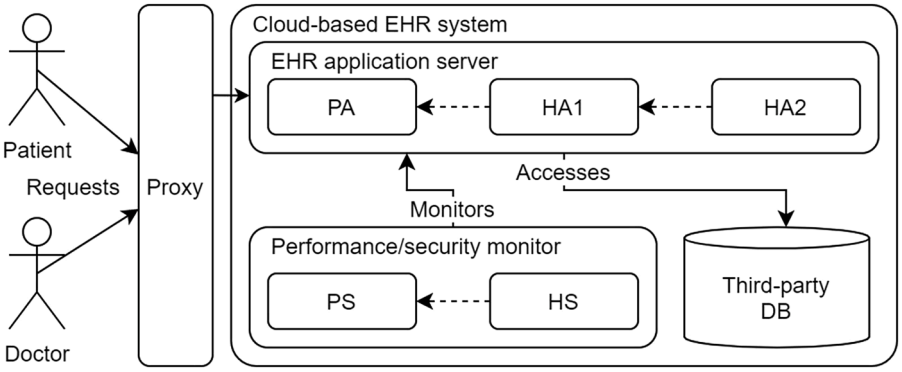


Fig. 8. The general structure of a cloud-based EHR system.

also responsible for taking measurements of the application server's computing usage (e.g., CPU, memory, and disk i/o). These measurements allow us to change the reliability models associated with the application server's primary component *PA*, based upon its current level of stress. Note that since the performance/security monitor has a constantly high workload, in this case study, *PS* does not change state and *HS* changes its state from *LOW* to *HIGH* only when it is switched on. Using maximum likelihood estimation as in (2), we conducted initial simulations on the virtualized components *PA, HA*1, *HA*2, *PS* and *HS*, and estimated Weibull parameters used in this case study as shown in Table 1.

In the case study, the distributions listed in Table 1 will be chosen based upon runtime measurements of computing usage to provide the most accurate reliability analysis. Note that some parameters are shared between the primary components and their hot spares as they have the same specifications. Further, note that as failures are very difficult to simulate with *LOW* usage, where the component experiences little to no traffic, constant failure rate distributions have been assumed.

As mentioned in Section 3, reliability calculations for each gate will be combined using DFTs. Fig. 9 shows an extended DFT for the cloud-based EHR system during stage 1 of rejuvenation. The DFT indicates that, if the application server *or* the security/performance manager fail, the system will fail as a whole.

Let *S*1 be the SSP gate with *PA, HA*1 and *HA*2, and *S*2 be the SSP gate

Table 1
Simulated lifetime distribution parameters.

	Usage	Shape κ	Scale λ
Component *PA*			
	LOW	1.000	0.0020
	MED	1.395	0.0068
	HIGH	1.422	0.0176
Components *HA*1, *HA*2			
	LOW	1.000	0.0020
	HIGH	1.422	0.0176
Component *PS*			
	HIGH	1.771	0.0021
Component *HS*			
	LOW	1.000	0.0010
	HIGH	1.771	0.0021

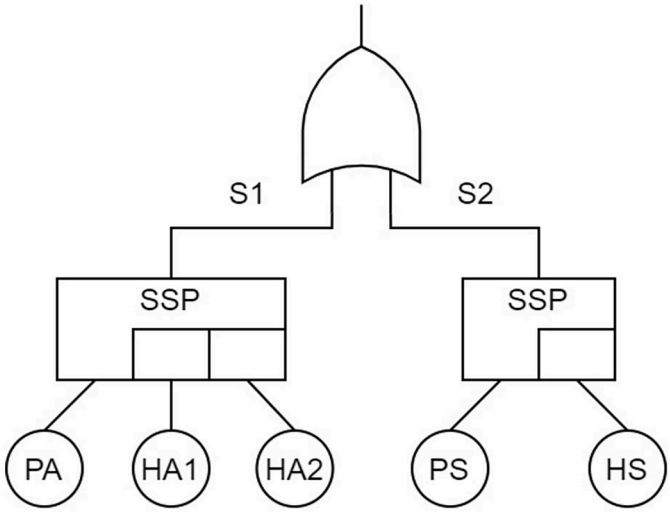


Fig. 9. DFT for the cloud-based EHR system.

with *PS* and *HS*. The individual unreliabilities $U_{S1}(t)$ and $U_{S2}(t)$ are calculated as described in Section 3, and then combined as in (26).

$$R(t) = 1 - (U_{S1}(t) + (1 - U_{S1}(t)) * U_{S2}(t)) \quad (26)$$

To demonstrate the efficacy of the approach, we first compare its impact on uptime to a system without rejuvenation and software spare support. Then, we demonstrate the adaptability of our approach under various scenarios. Specifically, the adaptive behavior of the cloud-based EHR system is shown through a typical year and rescheduling of rejuvenation when faced with several different situations.

4.2. Comparative analysis

In this section, we compare a system supported by HSSs and software rejuvenation described in our approach with a system that does not use these reliability techniques. We assume both systems experience high computing usage. We refer to the system using our approach and the system without rejuvenation and HSS support as system *A* and system *B*, respectively. Fig. 10 shows reliability forecasts for both systems. The reliabilities calculated for each system differ in two ways. First, the reliability of system *A* reverts to 1 immediately after each rejuvenation. When the reliability drops below the threshold 0.99 every approximately 31 days, the system is rejuvenated, resetting its reliability to 1. Second, the reliability of system *A* is higher than that of system *B*, even before the first rejuvenation. This general advantage is conferred by the failover capability provided by the HSSs attached to the primary components of system *A*.

To further contextualize the advantage of our approach over a system without rejuvenation and HSS support, we allowed both systems to be repairable in the event of a failure and performed Monte Carlo simulations to estimate uptime and downtime for both systems over a 100-day period. We assumed a repair time of 45 min after a system failure, and 10,000 trials of each system were performed for 100 days each, yielding the averages in Table 2.

On average, system *B* has more than 60 times the downtime of system *A*, with an average downtime of over an hour. This means that in a given 100-day period, system *B* will have at least one downtime of up to 45 min for maintenance and repairs. In healthcare, it is often necessary to review a patient's records before administering medication or performing a procedure. In this way, doctors can avoid causing further complications that could threaten a patient's life. Such a high average downtime is a major cause for concern as health issues can be very time sensitive. On the other hand, system *A*, which uses hot software spares and rejuvenation according to our method, has an average downtime of only about 1 min. Out of a total of 10,000 trials using system *A*, there were 9747 with zero downtime. In a healthcare setting, this means that doctors have access to patient records reliably and patients can receive informed care with much higher safety and less delays in the long run. The average uptime of system A meets and exceeds the "five nines" availability standard for emergency response systems (Reynolds, 2020). For critical applications, such as an EHR system, the major improvements in availability and QoS of our approach can be achieved despite the small overhead associated with software failover and reasonable rejuvenation schedules.

Table 2
QoS measures for simulated systems.

Measure	System *A*	System *B*
Avg. Uptime	0.99999206 (99 d, 23 hr, 58.86 min)	0.99945316 (99 d, 22 h, 41.25 min)
Avg. Downtime	0.00000794 (1.14 min)	0.00053175 (1 hr, 18.75 min)
0-Downtime Trials	9747	1001

4.3. Scenarios

In this section, we will follow the system as it adapts to the typical and expected scenarios throughout a period of usage, as shown in Fig. 11. Note that although overall "computing usage" is taken as the highest of the three discretized measures of usage (memory, CPU, and disk i/o usage), it is visualized in Fig. 11(a) and (b) as a single, continuous curve to aid understanding.

Scenario 1: Suppose our system begins the year with *PA* experiencing medium computing usage. Following the regular formulas in Section 3.2, the system shall be rejuvenated once roughly every 43 days, as shown in Fig. 12.

Now at some point in the year, one of the measures of computing usage rises and stays above the threshold for high usage as shown in Fig. 11(a). For example, the emergence of a new disease or the recurrence of a seasonal illness may lead to a sustained increase in the utilization of the EHR system. To confirm such a seasonal change, it is important to repeatedly observe the increased computing usage over a period of time (e.g., a week or more). This approach prevents a system from prematurely adjusting for "false positives" caused by temporary spikes in usage such as daily updates, backup processes, or transient traffic spikes. When the system enters a period of high usage, we should change the reliability model for *PA* and subsequently the overall reliability analysis. To calculate a rejuvenation schedule that takes this change of state into account, we use the procedure described in Section 3.3 to recalculate *PA*'s unreliability. Meanwhile, the unreliabilities for the other components *HA*1, *HA*2, *PS,* and *PA* remain the same, as only *PA* has undergone a state change requiring a recalculation. These unreliabilities are combined according to the construction of our DFT (shown in Fig. 9) to produce the new reliability forecast, from which the next rejuvenation is scheduled.

Suppose component *PA* changes state from medium to high usage at $t^* = 8$ days from the previous rejuvenation. With a threshold of 0.99, the following rejuvenation dates can be calculated as in Fig. 13. As shown in

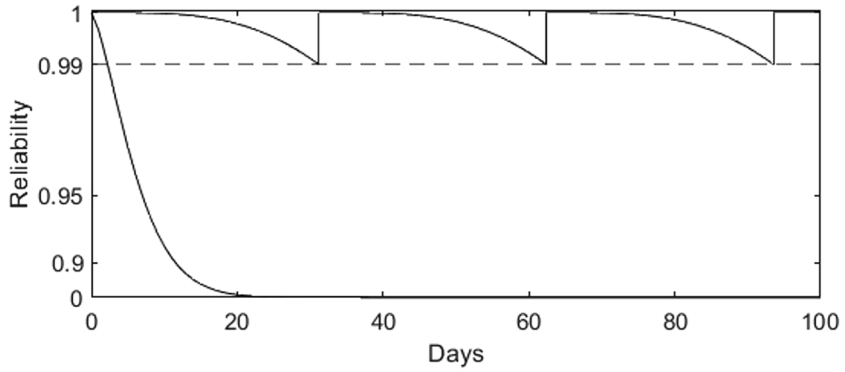


Fig. 10. Reliability forecast for system *A* and *B* (using the medium parameter set for *PA*).

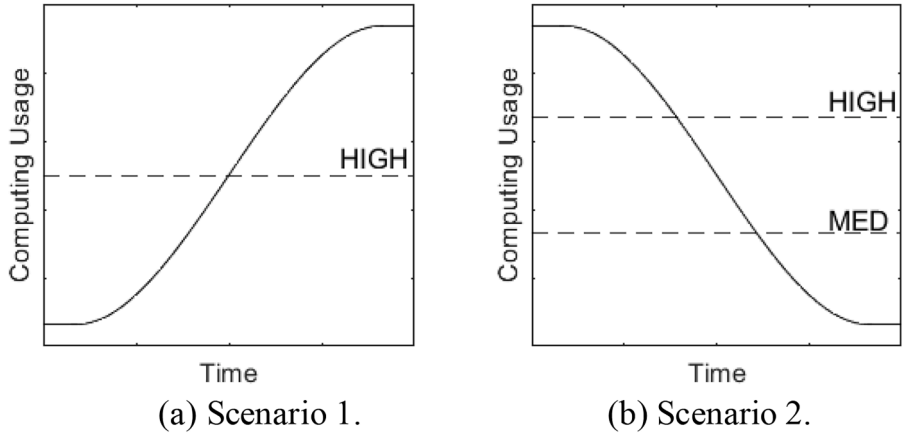


Fig. 11. Graphs of computing usage in Scenarios 1 and 2.

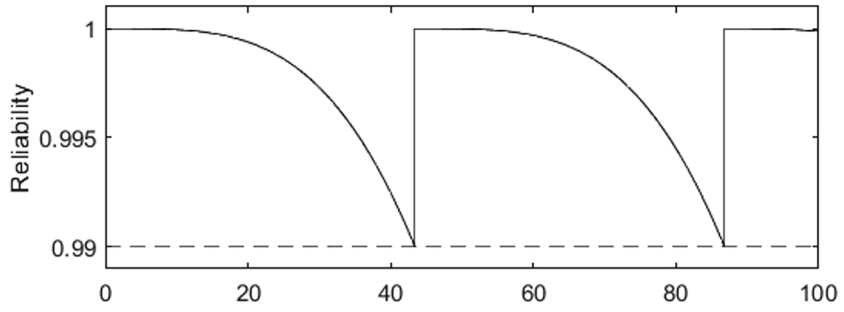


Fig. 12. Reliability forecast at the beginning of Scenario 1 (using the medium parameter set for PA).

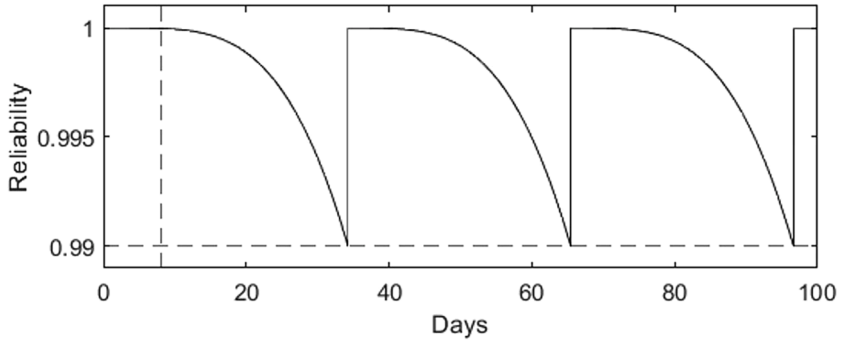


Fig. 13. Reliability forecast for Scenario 1 when PA switches from the medium to the high usage state at $t^* = 8$ days.

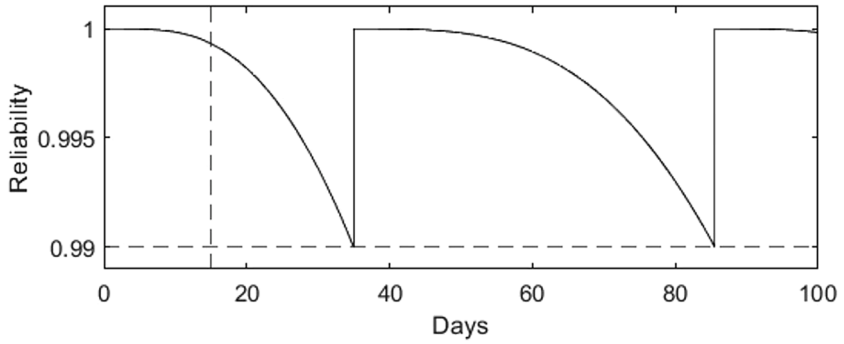


Fig. 14. Reliability forecast for Scenario 2 when PA switches from the high to the low usage state at $t^* = 15$ days.

the figure, this recalculation causes the rejuvenation to be pushed backward from about day 43 to about day 34, which fits with our expectation that the system should be more unreliable after it switches to a higher usage state. Subsequent reliability calculations follow the regular procedure (described in Section 3.2) using the high-usage parameter sets for *PA*. Following the higher computing usage and thereby higher expected failure rate of *PA*, the rejuvenations scheduled according to our reliability analysis are now more frequent. These more frequent rejuvenations are required for the system to maintain high reliability during busy periods for use by doctors and patients; without the rejuvenation schedule adjustment, the system would fall below the acceptable threshold for reliability and risk dangerous outages. Using the high usage parameter sets, we now rejuvenate roughly every 31 days; thus, the next rejuvenation date will be on day 65.

Scenario 2: The system is now running at a high usage for component *PA*, which schedules a rejuvenation every roughly 31 days, as shown in Fig. 14. Suppose our system now experiences the opposite situation where *PA*'s computing usage steadily declines and stays in a *LOW* computing usage state, as shown in Fig. 11(b). Similar to Scenario 1, we wait for repeated measurements of low usage to confirm that the *LOW* computing usage state is a steady one. In this case, we switch the high usage parameters out for the low usage parameters, which are used to recalculate overall system reliability. Once again, we will follow the procedure described in Section 3.3 to recalculate the reliabilities of the SSP gates and subsequently the entire system, from which we will schedule our next rejuvenations. Suppose we detect this consistent change of state from high to low at $t^* = 15$ days from the previous rejuvenation. With a threshold of 0.99, the following rejuvenation dates can be calculated as in Fig. 14. In this situation, we expect our recalculated reliability analysis to call for less frequent rejuvenations as the system is no longer under the stress of prolonged high computing usage. As shown in the figure, with the stress on our components decreasing at day 15, the next rejuvenation is pushed forward from about day 31 to about day 35. Following this first rejuvenation, we will now rejuvenate roughly every 50 days rather than every 31 days. Note that the rescheduling helps avoid excessive reboot overhead while still maintaining reliability above the desired threshold. For example, if this period of low stress lasts 160 days, by adjusting our models, we will have reduced our rejuvenations from 5 to 3.

Scenario 3: When a primary component or an HSS experiences a critical event, such as a failure, it may be detected by the performance/security monitor if the critical event occurs in the application server, or by an administrator if it occurs in the monitor. When such a critical event is detected, the ideal practice is to restart only the failed component. In this way, system reliability can be restored without introducing the overhead of an unnecessary system-wide reboot, which can result in longer service response times for doctors and patients using the EHR system. When a single component is restarted, the reliability of the entire system must be recalculated, and rejuvenations must be rescheduled. Following the procedure detailed in Section 3.4, we time-shift the pdf of the restarted component to its new start time and recalculate the unreliability as a conditional probability using the observation that the other components have *not* yet failed. The next rejuvenation is scheduled according to this recalculation.

Suppose *PA* is running with medium computing usage, with regular rejuvenations occurring roughly every 43 days. Suppose a critical event is detected in component *HA*1 at $t^* = 20$ days. *HA*1 can be restarted, and the system reliability is recalculated as shown in Fig. 15.

From the figure, we can see that when component HA_1 is restarted, the entire system reliability jumps back up. Because this component is fresh, the entire system is more reliable and takes longer to rejuvenate after $t^* = 20$ days. The first rejuvenation is delayed by 12 days and occurs at roughly 55 days. Subsequent rejuvenations are scheduled as normal at roughly every 43 days.

Scenario 4: Computing usage crosses the threshold for a state change, but only momentarily, as shown in Fig. 16. These momentary changes *should not* affect reliability calculations, as they do not represent a true change in the running circumstances of a component. Our measures of computing usage (e.g., CPU and disk i/o usage) can often show short-term spikes. Therefore, our approach requires waiting for a certain number of repeated measurements to confirm a true shift to a new usage level. The number of elevated measurements required to confirm a true state change varies from system to system, but some general guidelines can be helpful. For example, a system that needs to adapt to seasonal traffic may require daily measurements for a couple of weeks to confirm a true seasonal state change. On the other hand, for a system that needs to adapt to traffic over a shorter period of time (e.g., holidays), hourly measurements over several days may be helpful. However, confirmation should wait long enough to exclude expected regular usage periods such as routine updates or backups. This ensures that we do not make frequent model changes and recalculations due to "false positives" or momentary changes that do not reflect underlying usage trends.

Finally, when a component fails in a way that is not immediately recognizable, it shall not result in a system reliability that is below the threshold. This is because an SSP gate allows its component to fail, and such scenarios have already been covered in the possible failure sequences while calculating the system reliability. This being said, for unrecognizable component failure, no recalculation of system reliability would be required, as the system reliability will always be maintained above the threshold with its current rejuvenation schedule.

5. Conclusions and future work

In this work, we explored a hybrid approach to reliability analysis and rejuvenation scheduling using preventive and automatic failover strategies. This approach makes use of both runtime measurements and DFT analysis to forecast reliability in order to schedule future rejuvenations for software systems with virtualized software spares. We presented the procedures to calculate reliabilities of an SSP gate with up to 2 HSSs, which allows rejuvenations to be rescheduled following state

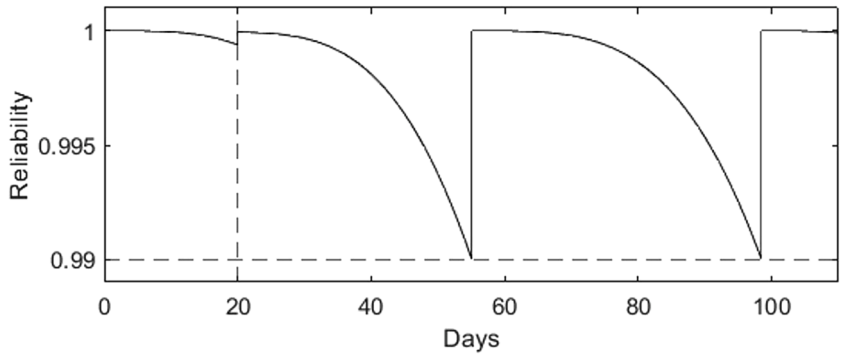
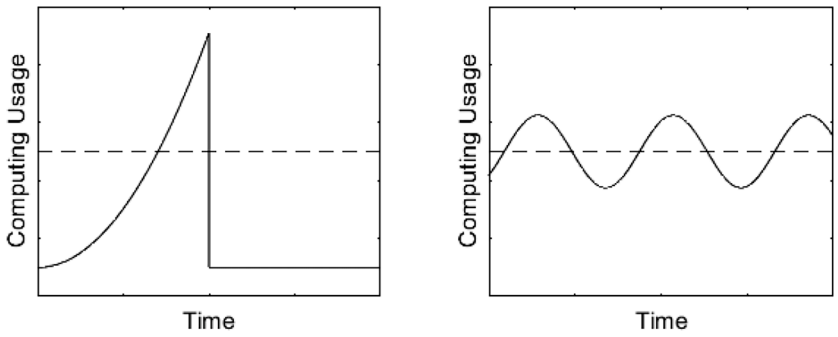


Fig. 15. Reliability forecast for Scenario 3, where component *HA*1 is restarted manually following a critical state change at $t^* = 20$ days.



(a) Usage 1 (b) Usage 2

Fig. 16. Possible computing usage in Scenario 4.

changes of virtualized software components in a cloud system. The general effectiveness of this approach has been demonstrated using a simulated cloud system, with initial reliability data estimated using simulations on virtualized software components and reliability analysis based on modular and easily designable DFTs. In addition, the approach has been demonstrated to allow adaptation to the unexpected rebooting of a component. The specific scenarios presented in the case study may serve as examples for the types of typical situations to be handled by the proposed approach.

For future work, we will further demonstrate how to provide a scalable solution to maintain required levels of reliability for a complex cloud system. We plan to study the measurement-based approaches for failure detection and prediction in real time, as well as the effective way to incorporate the findings into our approach. Since the changes in lifetime distributions of software components in a cloud system are typically due to changes in computing resource usage, they can be viewed as concept drifts that describe the changes of data streams over time due to their underlying factors (Lu et al., 2019; Widmer and Kubat, 1996). It is envisioned that concept drift detection techniques may provide an effective way to support real-time monitoring of state changes of virtualized software components in cloud systems. Another vital component of future work will be the incorporation of studies on a real implementation, using cloud services like Microsoft Azure, Google Cloud, or Amazon Web Services. These implementations will provide support for the approach's viability beyond theoretical analyses. To extend the applicability of the work, future research could also investigate the application of our rejuvenation scheduling approach in critical non-cloud systems, as many of the methods involved (e.g., SSPs and DFT analysis) can be broadly applied beyond cloud environments.

CRediT authorship contribution statement

Joshua R. Carberry: Writing – review & editing, Writing – original draft, Methodology, Formal analysis, Conceptualization. **John Rahme:** Writing – review & editing, Writing – original draft, Methodology, Formal analysis, Conceptualization. **Haiping Xu:** Writing – review & editing, Writing – original draft, Methodology, Formal analysis, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this article.

Data availability

No data was used for the research described in the article.

Acknowledgment

We thank the editors and all anonymous referees for their valuable time in reviewing this paper. We also thank UMass Dartmouth for their financial support to the first author of this paper in completing this work.

References

Aslansefat, K., Kabir, S., Gheraibiam, Y., Papadopoulos, Y., June 2020. Dynamic fault tree analysis: state-of-the-art in modelling, analysis and tools. In: Garg, H., Ram, M. (Eds.), Reliability Management and Engineering: Challenges and Future Trends, Edition 1. Chapter 4, CRC Press.

Bahga, A., Madisetti, V.K., 2013. A cloud-based approach for interoperable electronic health records (EHRs). IEEe J. Biomed. Health Inform. 17 (5), 894–906. Sept.

Bobbio, A., Sereno, M., Anglano, C., 2001. Fine grained software degradation models for optimal rejuvenation policies. Performance Eval. 46 (1), 45–62.

Boles, J., 2011. What you need to know about automatic failover and disaster recovery automation. Search Disaster Recovery. Taneja Group. June 22Retrieved on June 1, 2020 from. https://searchdisasterrecovery.techtarget.com/.

Boudali, H., Nijmeijer, A., Stoelinga, M., 2009. DFTSim: a simulation tool for extended dynamic fault trees. In: Proceedings of the 2009 Spring Simulation Multiconference (SpringSim'09), pp. 1–8. MarchArticle No.

Boudlal, H., Serrhini, M., Tahiri, A., 2022. Cloud computing for healthcare services: technology, application and architecture. In: Proceedings of the 2022 11th International Symposium on Signal, Image, Video and Communications (ISIVC). El Jadida, Morocco, pp. 1–5.

Bruneo, D., Distefano, S., Longo, F., Puliafito, A., Scarpa, M., 2013. Workload-Based software rejuvenation in cloud systems. IEEE Trans. Comput. 62 (6), 1072–1085. June.

Carrozza, G., Cotroneo, D., Natella, R., Pietrantuono, R., Russo, S., 2013. Analysis and prediction of mandelbugs in an industrial software system. In: Proceedings of the 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation, pp. 262–271.

Chang, X., Wang, T., Rodríguez, R.J., Zhang, Z., 2018. Modeling and analysis of high availability techniques in a virtualized system. Comput. J. 61 (2), 180–198.

Cully, B., Lefebvre, G., Meyer, D., Feeley, M., Hutchinson, N., Warfield, A., 2008. Remus: high availability via asynchronous virtual machine replication. In: Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI '08), pp. 161–174. April 16.

Dohi, T., Zheng, J., Okamura, H., Trivedi, K.S., 2018. Optimal periodic software rejuvenation policies based on interval reliability criteria. Reliab. Eng. Syst. Saf. 180, 463–475.

Du, Y., Yu, H., 2009. Paratus: instantaneous failover via virtual machine replication. In: Proceedings of the Eighth International Conference on Grid and Cooperative Computing. Lanzhou, China, pp. 307–312. August 27-29.

Fang, B., Yin, X., Tan, Y., Li, C., Gao, Y., Cao, Y., Li, J., 2016. The contributions of cloud technologies to smart grid. Renewable Sustain. Energy Rev. 59, 1326–1331. June.

Fantechi, A., Gori, G., Papini, M., 2022. Software rejuvenation and runtime reliability monitoring. In: Proceedings of the 2022 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW). Charlotte, NC, USA, pp. 162–169.

Grottke, M., Trivedi, K.S., 2005. Software faults, software aging, and software rejuvenation. J. Reliab. Eng. Assoc. Japan 27 (7), 425–438.

Grottke, M., Li, L., Vaidyanathan, K., Trivedi, K.S., 2006. Analysis of software aging in a web server. IEEE Trans. Reliab. 55 (3), 411–420.

Grottke, M., Kim, D.S., Mansharamani, R., Nambiar, M., Natella, R., Trivedi, K.S., 2016. Recovery from software failures caused by mandelbugs. IEEE Trans. Reliab. 65 (1), 70–87. March.

Guo, J., Ju, Y., Wang, Y., Li, X., Zhang, B., 2010. The prediction of software aging trend based on user intention. In: Proceedings of the IEEE Youth Conference on Information, Computing and Telecommunications. Beijing, China, pp. 206–209. November 28-30.

Ibe, O.C., Howe, R.C., Trivedi, K.S., 1989. Approximate availability analysis of VAXcluster systems. IEEe Trans. Reliab. 38 (1), 146–152. April.
Jia, S., Hou, C., Wang, J., 2017. Software aging analysis and prediction in a web server based on multiple linear regression algorithm. In: Proceedings of the IEEE 9th International Conference on Communication Software and Networks (ICCSN). Guangzhou, China, pp. 1452–1456. May 6-8.
Koutras, V.P., Platis, A.N., 2006. Applying software rejuvenation in a two node cluster system for high availability. In: Proceedings of the Int'l Conference on Dependability of Computer Systems. Szklarska, Poreba, pp. 175–182. May 25-27.
Liu, J., Zhou, J., Buyya, R., 2015. Software rejuvenation based fault tolerance scheme for cloud applications. In: Proceedings of the IEEE 8th International Conference on Cloud Computing. New York, NY, USA, pp. 1115–1118. June 27-July 2.
Lu, J., Liu, A., Dong, F., Gu, F., Gama, J., Zhang, G., 2019. Learning under concept drift: a review. IEEE Trans. Knowl. Data Eng. 31 (12), 2346–2363. December.
Meingast, M., Roosta, T., Sastry, S., 2006. Security and privacy issues with health care information technology. In: Proceedings of the International Conference of the IEEE Engineering in Medicine and Biology Society. New York, NY, USA, pp. 5453–5458. August 30–September 3.
Mendiratta, V.B., 1998. Reliability analysis of clustered computing systems. In: Proceedings of the Ninth International Symposium on Software Reliability Engineering (Cat. No.98TB100257). Paderborn, Germany, pp. 268–272. November 4-7.
Meng, H., Zhang, X., Zhu, L., Wang, L., Yang, Z., 2017. Optimizing software rejuvenation policy based on CDM for cloud system. In: Proceedings of the 12th IEEE Conference on Industrial Electronics and Applications (ICIEA). Siem Reap, Cambodia, pp. 1850–1854. June 18-20.
Morkos, R., 2023. Powering the growth of cloud computing: infrastructure challenges and solutions," Forbes newsletters. Forbes Technol. Council. July 24Retrieved on May 15, 2024 from. https://www.forbes.com/sites/forbestechcouncil/2023/07/24/powering-the-growth-of-cloud-computing-infrastructure-challenges-and-solutions/.
Nguyen, T.A., Min, D., Choi, E., Tran, T.D., 2019. Reliability and availability evaluation for cloud data center networks using hierarchical models. IEEe Access. 7, 9273–9313. January.
Nguyen, T.A., Min, D., Choi, E., Lee, J.W., 2021. Dependability and security quantification of an internet of medical things infrastructure based on cloud-fog-edge continuum for healthcare monitoring using hierarchical models. IEEE IoT J. 8 (21), 15704–15748. November.
Pham, H., 2006. System software reliability. Springer Series in Reliability Engineering. SpringerVerlag, London.
Qiu, K., Zheng, Z., Trivedi, K.S., Mura, I., 2021. Availability analysis of systems deploying sequences of environmental-diversity-based recovery methods. IEEE Trans. Reliab. 70 (3), 1126–1142.
Rahme, J., Xu, H., 2015. A software reliability model for cloud-based software rejuvenation using dynamic fault trees. IJSEKE 25, 1491–1513. Nos. 9 & 10.
Rahme, J., Xu, H., 2017a. Dependable and reliable cloud-based systems using multiple software spare components. In: Proceedings of the 14th IEEE International Conference on Advanced and Trusted Computing (IEEE ATC 2017). San Francisco, CA, USA, pp. 1462–1469. August 4-8.
Rahme, J., Xu, H., 2017b. Preventive maintenance for cloud-based software systems subject to non-constant failure rates. In: Proceedings of the 14th IEEE International Conference on Advanced and Trusted Computing (IEEE ATC 2017). San Francisco, CA, USA, pp. 1576–1581. August 4-8.
Rausand, M., Høyland, A., 2004. System Reliability Theory: Models, Statistical Methods, and Applications, Second Edition. John Wiley & Sons, Inc., Hoboken, New Jersey, USA.
Reynolds, R., 2020. Achieving 'five nines' in the cloud for justice and public safety. AWS Public Sector Blog. Amazon. March 17Retrieved on July 6, 2022 from. https://aws.amazon.com/es/blogs/publicsector/achieving-five-nines-cloud-justice-public-safety/.
Somani, A., Vaidya, N., 1997. Understanding fault tolerance and reliability. IEEE Comput. 30 (4), 45–50.
Sudhakar, C., Shah, I., Ramesh, T., 2014. Software rejuvenation in cloud systems using neural networks. In: Proceedings of the International Conference on Parallel, Distributed and Grid Computing. Solan, India, pp. 230–233. December 11-13.
Tantri, B.R., Murulidhar, N.N., 2016. Software reliability estimation of gamma failure time models. In: Proceedings of the 2016 International Conference on System Reliability and Science (ICSRS). Paris, France, pp. 105–109. November 15-18.
Thein, T., Chi, S.-D., Park, J.S., 2007. Availability analysis and improvement of software rejuvenation using virtualization. Economics and Applied Informatics. University of Galati, pp. 5–14.
Thein, T., Chi, S.-D., Park, J.S., 2008. Availability modeling and analysis on virtualized clustering with rejuvenation. IJCNS 8 (9), 72–80.
Vaidyanathan, K., Selvamuthu, D., Trivedi, K.S., 2002. Analysis of inspection-based preventive maintenance in operational software systems. In: Proceedings of the 21st IEEE Symp. on Reliable Distributed Systems (SRDS2002). Suita, Japan, pp. 286–295. October 13-16.
Vinoth, S., Vemula, H.L., Haralayya, B., Mamgain, P., Hasan, M.F., Naved, M., 2022. Application of cloud computing in banking and e-commerce and related security threats. Mater. Today 51 (8), 2172–2175.
Widmer, G., Kubat, M., 1996. Learning in the presence of concept drift and hidden contexts. Mach. Learn. 23 (1), 69–101.
Yang, M., Li, Z., Yang, W., Li, T., 2010. Analysis of software rejuvenation in clustered computing system with dependency relation between nodes. In: Proceedings of the 10th IEEE International Conference on Computer and Information Technology. Bradford, UK, pp. 46–53. June 29-July 1.
Yangzhen, F., Hong, Z., Chenchen, Z., Chao, F., 2017. A software reliability prediction model: using improved long short term memory network. In: Proceedings of the IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C). Prague, Czech Republic, pp. 614–615. July 25-29.
Yao, H., Hao, Z., 2023. Research on key technologies of design of flight test telemetry monitoring system based on private cloud computing. In: Proceedings of the 2023 2nd International Conference on Cloud Computing, Big Data Application and Software Engineering (CBASE). Chengdu, China, pp. 259–263.

Joshua R. Carberry (S'21) received the B.S. degree in Computer Science with a Minor in Mathematics and M.S. degree in Computer Science from the University of Massachusetts Dartmouth in 2020 and 2022, respectively. He is currently a Ph.D. student in the Engineering and Applied Science (EAS) Ph.D. program, Computer Science and Information System (CSIS) option at University of Massachusetts Dartmouth. His research interests include cloud-based software reliability, natural language processing, deep learning and automated medical coding. He is a student member of the IEEE and the IEEE Computer Society.

John Rahme received the B.S. degree in Computer and Communication Engineering from American University of Technology, Halat-Beirut, Lebanon, in 2009, the M.S. degree in Computer Science and the Ph.D. degree in Engineering and Applied Science (EAS), Computer Science and Information System (CSIS) option from University of Massachusetts Dartmouth, MA, in 2012 and 2018, respectively. He earned the first Ph.D. in the Computer and Information Science Department and is currently a Data Analyst II at Hanscom Air Force Base, MA, USA. He has expertise in stochastic calculus and formal modeling techniques. His current research interests include reliable cloud computing and mission-critical systems, resilient cybersecurity, and machine learning.

Haiping Xu (S'97–M'03–SM'07) received the B.S. degree in Electrical Engineering from Zhejiang University, Hangzhou, China, in 1989, the M.S. degree in Computer Science from Wright State University, Dayton, OH, in 1996, and the Ph.D. degree in Computer Science from the University of Illinois at Chicago, IL, in 2003. Prior to 1996, he successively worked with Shen-Yan Systems Technology, Inc. and Hewlett-Packard Co., as a Software Engineer, in Beijing, China. Since 2003, he has been with the University of Massachusetts Dartmouth, where he is currently a Professor and Chairperson at the Computer and Information Science Department, and a Director of the Concurrent Software Engineering Laboratory (CSEL). Dr. Xu has over 100 publications including 7 edited book, proceedings and special issues, as well as over 40 peer-reviewed journal papers. He has supervised over 100 M.S. projects, M.S. theses and Ph.D. dissertations at University of Massachusetts Dartmouth. He has expertise in distributed software engineering and formal methods, and his current research interests include cloud computing, software reliability, cybersecurity, electronic commerce, and deep learning. His research has been supported by the U.S. National Science Foundation (NSF), the U.S. Marine Corps, the Office of Naval Research (ONR), and the UMass President's Office. Dr. Xu is a senior member of both the IEEE and the Association of Computing Machinery (ACM). He has been an Associate Editor for many journals including the IEEE Transactions on Cloud Computing (TCC) and the International Journal of Software Engineering and Knowledge Engineering (IJSEKE). He was a recipient of the Editorial Excellence and Eminence Award for his outstanding contributions as an Associate Editor of the IEEE Transactions on Cloud Computing in 2019. Dr. Xu has served on program committees for over 100 international conferences, and organized and chaired many technical sessions. He is a leading Program Chair of the 27th International Conference on Software Engineering and Knowledge Engineering (SEKE 2015), the IEEE International Conference on Mobile Cloud Computing, Services, and Engineering (IEEE Mobile Cloud 2017), the IEEE International Conference on Smart City Innovations (IEEE SCI 2017), and the IEEE International Workshop on Blockchain and Smart Contracts (IEEE BSC 2020), as well as a General Chair of the SEKE 2016, IEEE Mobile Cloud 2018 and IEEE SCI 2018.