Architecture-based Availability Prediction and Service Recommendation for Cloud Computing

Von der Fakultät Informatik, Elektrotechnik und
Informationstechnik der Universität Stuttgart
zur Erlangung der Würde eines Doktors der Naturwissenschaften
(Dr. rer. nat.) genehmigte Abhandlung

vorgelegt von

Otto Bibartiu

aus Baia Mare (Neustadt), Rumänien

Hauptberichter:	Prof. Dr. rer. nat. Dr. h. c. Kurt Rothermel
Mitberichter:	Prof. Dr. rer. nat. Lars Grunske
Mitprüfer	Prof. Dr.-Ing. Steffen Becker
Tag der mündlichen Prüfung:	06.06.2023

Institut für Parallele und Verteilte Systeme (IPVS)
der Universität Stuttgart
2023

For Simi.
Eternally grateful for your patience and unconditional love.

Attributions

This work uses icons of the Microsoft Azure Cloud known as “Azure-Architektursymbols” by Microsoft Corporation and is licensed under `https://docs.microsoft.com/de-de/azure/architecture/icons/`.

This work also uses “Cloud Computing Pattern Icons” by `http://www.cloudcomputingpatterns.org` licensed under the Creative Commons Attribution 4.0 International (CC BY 4.0)

Acknowledgments

In the moments of uncertainty and hardship, I had the privilege to be guided by many exceptional people to whom I would like to express my sincere gratitude for the help and valuable lessons they gave.

To Prof. Kurt Rothermel from whom I learned to be a critical thinker and precise writer, tapping into unknown potential that was otherwise not possible without his guidance.

To my supervisor Dr. Frank Dürr for his patient and endurance in teaching me to be a good researcher.

To my supervisors at Bosch, Dr. Beate Ottenwälder and Dr. Andreas Grau for teaching me the necessary technical insights to complete this work. On this occasion, I would also like to acknowledge and thank Robert Bosch GmbH for funding a major part of this project.

To my dear friend Henriette Röger for her generosity to finance parts of my study when in need, and from whom I learned valuable skills in self-management.

To all my dear colleagues Eva Strähle, Ben Carabelli, Johannes Kässinger, Naresh Nayak, Ahmad Slo, Sukanya Bhowmik, David Hellmanns, Jonathan Falk, Heiko Geppert and Michael Schramm from whom I learned the importance of trust in team work, and with whom I shared the best moments when cleaning up the *Lager*.

To my parents Maria and Otto Bibartiu for supporting me with the work whenever possible.

To my parents-in-law Mirela and Costel Mititelu for taking care of my family when I had to work during the family visits.

To Dr. Howard J.T. Steers, whose sincerity and humility taught me faith, and to Gheorghe Bîlgăr, who ignited the drive in me to become a researcher.

To my wife and daughter, who taught me the value of family, providing the love that kept me going.

But most importantly, to my wife Simina Bibartiu, from whom I learned to be patient and humble. Thanks for supporting me.

Contents

Abstract

Cloud computing has become an essential pillar in the development of modern IT systems. IoT solutions such as smart factories and connected cars, or finance and e-commerce systems, just to name a few, have formed a strong availability dependency to the cloud. Consequently, it becomes increasingly important to assess the availability of a cloud application during its development phase to choose suitable cloud services to assure high service quality and to define robust service-level agreements with clients. However, one single cloud data center is already a highly complex system of hard- and software components, making availability assessments of cloud applications a challenging task.

While researchers acknowledge the significance of infrastructure and communication faults in the cloud as important aspects of the availability prediction, they usually model either the (compute) infrastructure or the communication part of a cloud service, disregarding that a cloud application consists of multiple interconnected services with potentially different replication degrees and common cause failures. Especially with the introduction of the Function-as-a-Service (FaaS) paradigm, which have a large number of redundant service instances, availability models become computationally infeasible when modeling k-out-of-n services with large n.

Usually, availability assessments guide design choices by probing for different cloud services that fit availability requirements or other constraints such as cost simultaneously. However, this task becomes also increasingly challenging in its own right due to the vast amount of potential service offerings and individual configuration options. As a result, recent developments in cloud application modeling have introduced a wide range of technology-agnostic cloud modeling languages, introducing abstractions or patterns that do not require ample knowledge on concrete cloud services anymore. However, the initial challenge remains: the more abstract a pattern, the larger the solution space, and the longer the time to find a suitable solution.

This thesis addresses both challenges. First, it proposes a hierarchical availability model implementing a novel availability model utilizing Bayesian networks for

the prediction task. Second, it presents a service recommendation system based on a novel pattern-based cloud modeling language called Clams, which provides a framework for custom search criteria in cooperation with meta-heuristics.

In detail, the availability model enables developers to model cloud applications at any preferred level of component and network granularity, accounting for cascading infrastructure and communication faults, including the individual replication semantics of services. Moreover, this work introduces scalable Bayesian network structures to enable the modeling of FaaS offerings, or large-scale replicated services, with many instances. The presented service recommendation system provides a generic approach of utilizing meta-heuristics to exploit a component-based architectural description of a cloud application. Cloud computing patterns are used as architectural placeholders while at the same time encoding the solution space of concrete services. Combining the service recommendation system with the availability model, this work demonstrates its results by implementing a system that suggests services with minimal operational cost, while adhering to availability constraints. To show the feasibility of the modeling concepts, this thesis analyses a set of thirty-one real-life architectural examples. Performance evaluations show that the service recommendation system can return a near-to-optimal solution in a feasible time.

Zusammenfassung

Cloud-Computing ist zu einem wesentlichen Bestandteil moderner IT-Systeme geworden. Es ermöglicht eine automatische Provisionierung von skalierbaren und kosteneffiziente Rechen- und Speicherkapazitäten, ohne dabei selbst die Infrastruktur betreiben zu müssen. Das Internet der Dinge (Internet of Things, IoT), sowie Smart-Factories und Connected-Cars, oder auch Finanz- und E-Commerce-Systeme haben eine starke Abhängigkeit in ihrer Verfügbarkeit zur Cloud entwickelt. Folglich wird es immer wichtiger die Verfügbarkeit einer Anwendung in der Cloud schon während ihrer Entwicklung zu bestimmen, um vorab geeignete Cloud-Dienste auszuwählen, die später, während der Produktionsphase, eine hohe Dienstgüte gewährleisten können. Dabei sollten nachträgliche Kosten durch Anpassungen wegen einer zu geringen Servicequalität vermieden werden. Jedoch ist ein einziges Rechenzentrum bereits schon ein hochkomplexes System aus Hard- und Softwarekomponenten, das eine Verfügbarkeitsbewertung von Cloud-Anwendungen umso relevanter macht.

Während die Wissenschaft die Bedeutung von Infrastruktur- und Kommunikationsfehlern in der Cloud als wichtige Merkmale der Verfügbarkeitsvorhersage anerkannt haben, beschreiben sie jedoch öfters Modellierungsansätze, die nur den Infrastruktur- oder Kommunikationsteil eines Cloud-Dienstes berücksichtigen. Dabei besteht eine Cloud-Anwendung aus mehreren miteinander verbundenen Diensten mit potenziell unterschiedlichen Replikationsgraden und gemeinsamen Fehlerursachen. Insbesondere mit der Einführung des Function-as-a-Service-Paradigmas, das im Wesentlichen eine funktionale Replikation mit einer großen Anzahl von Instanzen darstellt, werden existierende Verfügbarkeitsmodelle an ihre Grenzen gebracht. Solche hochskalierbare Dienste können nur durch ebenso skalierbare Verfügbarkeitsmodelle gelöst werden.

Einerseits ist es wichtig, dass Verfügbarkeitsmodelle verwendet werden, um zu bestimmen, ob Verfügbarkeitsanforderungen erfüllt werden, anderseits müssen auch solche Cloud-Dienste gewählt werden, die möglichst geringe Kosten verursachen. In der Regel werden Architekturentscheidung öfters durch Verfügbarkeitsbewertungen geleitet. Dabei wird nach Cloud-Diensten, die bestimmten Verfüg-

barkeitsanforderungen oder anderen Qualitätsmerkmalen entsprechen, gesucht. Jedoch wird durch das mittlerweile große Angebot an Cloud-Diensten und individueller Konfigurationsmöglichkeiten die Suche nach geeigneten Lösungen immer anspruchsvoller. Infolgedessen haben die jüngsten Entwicklungen in der Modellierung von Cloud-Anwendungen eine breite Palette an technologieunabhängigen Modellierungssprachen (Cloud-Modeling-Languages) eingeführt, die keine umfassenden Kenntnisse über konkrete Cloud-Dienste mehr erfordern. Jedoch sind Modellierungsabstraktionen weiterhin nur Stellvertreter für bestimme Lösungen. Je abstrakter das Modell, desto größer der Lösungsraum an möglichen Diensten und Konfigurationsmöglichkeiten, was wiederum die Suche nach kostengünstige Diensten, die die Verfügbarkeitsanforderung der Cloud-Anwendung erfüllt, erschwert.

Diese Arbeit befasst sich dementsprechend mit den Herausforderungen der Verfügbarkeitsmodellierung und Bewertung von Cloud-Anwendungen, um verfügbarkeitsbasierte Architekturentscheidungen durch Empfehlungen von optimalen Cloud-Diensten zu ermöglichen. Hierfür wird ein neuartiges Verfügbarkeitsmodell vorgestellt, das Bayes'sche Netze als mathematisches Modell zur Verfügbarkeitsvorhersage verwendet. Das Verfügbarkeitsmodell ermöglicht Entwicklern, ihre Cloud-Anwendungen auf unterschiedlichen Ebenen der Komponenten- und Netzwerkgranularität zu modellieren. Dabei kann das Modell abhängige Infrastruktur- und Kommunikationsfehler, als auch die individuellen Replikationsgrade der verschiedenen Dienste berücksichtigen. Um auch moderne Cloud-Dienste beschreiben und bewerten zu können, schlägt diese Arbeit skalierbare Bayes'sche Netzstrukturen vor, die die Modellierung von Cloud-Diensten mit hohen Replikationsgrad erst ermöglicht und berechenbar macht.

Um die zweite Herausforderung der Kostenoptimierung zu lösen, präsentiert diese Arbeit eine auf Metaheuristik basierte Empfehlungssystem, das auf die hier vorgestellte musterbasierten Cloud-Modellierungssprache namens Clams aufbaut. Das Empfehlungssystem verfolgt das Ziel, Cloud-Dienste mit minimalen Betriebskosten vorzuschlagen, während die Gesamtverfügbarkeit der Cloud-Anwendung über einem bestimmten Schwellenwert liegen muss. Validiert wurden die vorgeschlagenen Konzepte durch die Analyse von einunddreißig realen Architekturbeispielen. Schlussendlich zeigen ausführliche Leistungsbewertungen, dass das Verfügbarkeitsmodell große Cloud-Anwendungen modellieren und bewerten kann, als auch, dass das Empfehlungssystem eine nahezu optimale Lösung für die Architekturbeispiele in angemessener Zeit finden kann.

List of Figures

List of Tables

List of Algorithms

Acronyms

AWS Amazon Web Services. 2, 8

Azure Microsoft® Azure. 8, 10, 12

BPMN Business Process Model and Notation. 42

Clams Cloud Application Modeling Solution. 9, 10, 12, 13, 15, 16, 18, 19, 21–25, 27, 31, 32, 35, 37, 42, 43, 45, 64, 102, 103, 142, 145, 150, 157, 158

CPT Conditional Probability Table. 77, 78, 82, 84–87, 93, 100, 114, 118–121, 123, 124, 127–130, 132, 133, 136, 137

CTMC Continuous-time Markov Chain. 57, 117

DAG Direct Acyclic Graph. 67, 77

DCaaS Data-center-as-a-Service. 7

DTMC Discrete-time Markov Chain. 5, 10, 22, 24, 57, 157

ETL Extract, Transform and Load. 169

FaaS Function-as-a-Service. 7, 8, 22, 61, 64, 120

FMEA Failure Modes and Effects Analysis. 4

IaaS Infrastructure-as-a-Service. 5, 7, 22, 56, 60, 117

IEC International Electrotechnical Commission. 2, 4

IEEE Institute of Electrical and Electronics Engineers. 2

IoT Internet of Things. 7, 48

ISO International Organization for Standardization. 2

ITU International Telecommunication Union. 23

MCUB Min-Cut-Upper-Bound. 104

MDA Model-Driven Architecture. 16, 17, 42

MSC Message Sequence Chart. 18, 20, 21, 23

MTTF Mean Time to Failure. 117

PaaS Platform-as-a-Service. 7, 60

PCM Palladio Component Model. 5, 117, 155

QoS Quality of Service. 3, 6, 7, 12, 22, 142–145, 147, 155, 156, 158

SaaS Software-as-a-Service. 7

SLA Service Level Agreement. 1, 62, 89, 104, 105

SRGM Software Reliability Growth Model. 4, 5

UML Unified Modeling Language. 17, 42, 117

VM Virtual Machine. 5, 7, 8, 21, 22, 30, 56, 63, 117

XaaS Everything-as-a-Service. 7

List of Symbols

$\mathcal{C}$, $\boldsymbol{\mathcal{C}}$	Clams component and set of components
$\mathcal{S}$, $\boldsymbol{\mathcal{S}}$	Scenario model and set of scenarios
$\mathcal{U}$	Usage Profile model
$A_{\mathcal{S}}$	Availability value of a scenario
$\{F, T\}$	Availability states (F =faulty, T =up)
$\boldsymbol{M}, \boldsymbol{Q}$	Adjacency matrices
$\boldsymbol{P}$	Steady-state availability matrix
$A_{\mathcal{U}}$	Availability of usage profile
$P(\dots)$	Probability distribution
$P(\dots \mid \dots)$	Conditional probability distribution
$\boldsymbol{C}$	Set of soft- and hardware components
$\boldsymbol{I}$	Set of service instances
G_{FD}	Fault dependency graph model
FT	(Static) Fault tree
H	Set of hosts
G_{NET}	Network graph
E	Set of Edges
C_{A-B}	Channel between component A and B
R	Network route
S, $\boldsymbol{S}$	Service model and service set
$G_{\boldsymbol{S}}$	Service topology
$D(.)$	Deployment function
$\mathcal{G}$	Set of network entry points
Q	Set of instance combinations
$\mathcal{A}$	Cloud application/scenario model
BN	Bayesian network
X	Bayesian network nodes
$pa(.)$	Parent function
$\boldsymbol{A}$	Architecture (vector of components)
$\mathcal{L}$	Solution space
L	Loss function

Chapter 1.

Introduction

Cloud computing has become the de facto standard in modern Internet-based services. Consequently, incidents in cloud computing have a more extensive reach on businesses than ever before. Since cloud faults are no longer the exception, cloud service failures cannot be entirely avoided. Therefore, designing reliable cloud applications remains a challenge. As a result, actively assessing the availability of a cloud application during its development stages becomes a common practice in application development. Developers must be cautious in selecting appropriate cloud services that fit their availability requirements but are still within their budget. Failing to use appropriate services can have a negative influence on the expected availability in production, even increasing development costs when architectural decisions need to be revised later. The consequences are a decrease in service quality and customer satisfaction or potential financial loss due to violations of service-level agreements (SLA). Hence, this thesis proposes a novel availability model that considers dependent faults of cloud services and a heuristic approach that recommends appropriate cloud services from the myriads of possible service plans and offerings, intending to aid developers in implementing best practices that reduce operative costs.

This chapter provides an introduction to the work and its contributions. It begins with a motivation on cloud incidents in Section 1.1. Afterward, Section 1.2 provides a brief outline of the state-of-the-art in software availability prediction. Section 1.3 introduces modeling challenges. And finally, Section 1.4 outlines the contributions, followed by a description of the thesis structure in Section 1.5.

1.1. Motivation

Assessing the availability of a cloud application as early as possible during its development phase enables developers to properly select appropriate cloud services

that best fit their availability requirements. Nowadays, cloud computing forms the backbone of many IT systems, including, but not limited to, smart home solutions, connected cars services, smart cities, payment services, social networks, and logistic management. These IT systems directly or indirectly influence our day-to-day life and are responsible for comfort and security. Without a doubt, if a cloud provider encounters an outage, multiple dependent businesses will be unavailable simultaneously. This new magnitude of dependence on the cloud is the primary motivation why developers should be deliberate in selecting appropriate cloud services for their applications. The following selection of incidents exemplifies how dependent businesses are on cloud computing nowadays:

- The outage of Amazon.com in 2013, with a downtime of 30 minutes, led to an estimated loss of 2 million dollars in sales revenue [1].
- The outage of Amazon Kinesis in 2020, a service that enables real-time event (stream) processing, led to the unavailability of multiple customers and other dependent Amazon Web Services (AWS) offerings [2].
- The OVHcloud data center fire incident led to the unavailability of thousands of e-commerce businesses [3].
- The Google App Engine has encountered about 17 hours of downtime in the year 2021, and about 15 hours of downtime in 2020, causing service unavailabilities for several thousand dependent business [4].
- The Facebook outage in 2021, caused by a miss-configuration of the backbone routers [5], led to an estimated loss of 65 million dollars in revenue [6].

These incidents clearly show the financial consequences and impact of cloud computing on their clients. Especially, the Facebook outage shows how network faults have an equally important role in perceiving a service as available or not. As Bailis and Kingsbury [7] showed, it is not reasonable to believe that the network is reliable. Regarding a cloud application as working and reachable from the end-user perspective is in line with the availability definition of the ISO/IEC/IEEE International Standard on systems and software engineering. The standard defines *availability* as the "degree to which a system or component is operational and accessible when required" [8]. As such, availability analysis plays an integral role in developing dependable systems. Especially in software reliability engineering, examining the architecture to infer the availability of a

potential software system is a recurring research topic due to the continuously growing complexity of software systems.

Site reliability engineers and operation teams give their best to build and operate reliable cloud services. However, developers choose what services to use and how to interconnect them to build their cloud applications. As the incidents show, cloud services are not independent since they share infrastructure and communication components with other services. Cascading faults and common cause failures in the infrastructure, such as node failures, or communication faults, such as switch or firewall failures, can render multiple services unavailable simultaneously. To mitigate the influence of such outages, whether major or minor, developers must maintain best architectural practices and cultivate an availability-driven mindset to select those cloud services that best fit their availability requirements.

Availability is just one aspect of dependability, and it is just one property of a more extensive set of quality of service (QoS) attributes. However, availability is one of those attributes that developers or architects can assess initially, making it possible to guide design decisions and implement best architectural practices. Applying best architectural practices from the beginning strengthens the application's service quality. It increases maintainability and reduces the odds that availability requirements are not met during testing or are insufficient during production. In addition, since software projects have a narrow development schedule and a tight budget, avoiding architectural changes in later development stages is beneficial. Therefore, developers should address availability as early as possible during the application's design stage to avoid additional development costs due to a misdemeanor in selecting the correct services and mitigating risks during production.

As a result, this thesis proposes a novel cloud modeling language to design cloud applications with the help of cloud computing patterns, introducing a novel Bayesian network availability model to target specific modeling challenges that enable the assessment of large-scale cloud services. With the help of the new cloud modeling language and availability model, this thesis proposes a recommendation system that suggests cost-minimal cloud services that adhere to given availability constraints in order to assist developers in designing cloud applications accordingly.

1.2. Software Availability and Reliability Analysis

Software reliability engineering is a specialized domain of reliability engineering [9]. Due to historical reasons, reliability engineering strongly emphasizes hardware systems, such as industrial appliances and electrical components, where failures are mainly caused by ware-out, human error, physical damage, temperature, pressure, and humidity. Quantitative analysis approaches such as Bayesian [10] and Markov-based [11] models aim to estimate failure rates, whereas qualitative analysis approaches aim to identify failure modes. This includes models such as fault trees [12], reliability block diagrams [13], or Failure Modes and Effects Analysis (FMEA). FMEA has been also adapted for software systems [14] known as software FMEA and standardized under IEC 60812. Its main goal is to uncover potential software defects during development.

According to J. D. Musa [9], Software systems fail systematically, i.e., the same usage circumstances lead to the same failures when not fixed. Therefore, assuming we have systems without hardware faults, the software does not fail due to wear-outs as hardware components tend to do, making it challenging to apply reliability and availability models from the classical field of reliability engineering. Nevertheless, software systems still require probabilistic models to estimate failure rates since the fault circumstances are often unknown due to uncertain user usage or activation patterns caused by the runtime and execution environment [15, 16]. These behavioral properties led to the novel research area of *software reliability engineering*.

Software systems tend to get better when defects are fixed when they occur, which increases the software's maturity over time. Models that account for this maturity process are so-called software reliability growth models (SRGM) [9, 17, 18]. The goal of SRGM is to estimate how many defects might still exist in the software after a given time. SRGM is mostly based on data collection from software testing or during operations. Hence, this approach is suitable for cloud applications that are finished and in production or in their final development stages when extensive testing is possible. Reliability engineers can apply SRGM at the service level to compute failure probabilities of individual services from history data and later compute the availability at the application level from the failure probabilities of the services. While SRGM is a potential solution to estimate the reliability of individual cloud services, cloud incidents are often caused

by external failure causes, which also need to be considered. Therefore, software reliability engineers often use SRGM to assess individual service instances to compute failure rates, which they then can use as input parameters for more sophisticated availability models, that form a hierarchy of multiple availability models [19].

As Michael R. Lyu noted [20], it is not sufficient to assess the reliability or availability of a software system in isolation, but considering the execution (operational) environment, like the compute and network infrastructure, is equally essential to create detailed models. Hence, multiple availability/reliability models have been proposed throughout the decades to predict the failure probabilities of software systems at various stages of the development process. In cloud computing, Jammal et al. [21] introduce an availability model for multi-tier cloud applications. They model fault propagation within a strict hierarchical compute infrastructure, considering only the failures of the data center, servers, and virtual machines (VMs). They provide little space to improve the model with new failure modes.

Ghosh et al. [22] provide a more general approach, where they explicitly address the scalability issue of large Infrastructure-as-a-Service (IaaS) offerings. Instead of building one large availability model, they propose a hierarchical model of multiple smaller stochastic rewards nets. However, a cloud application is a collection of interconnected services. Therefore, to assess the availability of a cloud application, one needs models that can consider all service offerings, such as replicated database services.

There exist few models that also combine network and infrastructure failure modes. For example, Pitakrat et al. [23] use Bayesian networks for online failure predictions of microservice architectures, which address the communication between multiple dependent services. However, they do not address the challenge of modeling replicated services. Similarly, for the Palladio Component Model (PCM) [24,25], which is a powerful performance analysis model, supporting complex user usage profiles, network communication, and hardware models. Brosch et al. [24] extended the capabilities of PCM to support reliability analysis, using discrete-time Markov chains (DTMC) as underlying mathematical formalism. However, the PCM modeling approach does not support service replication. Pérez and Casal [26] enhanced PCM with cloud-specific metrics to support also performance assessments of cloud services. For example, modeling the elasticity of stateless-compute services, a service category we also identified as a modeling challenge in the domain of availability analysis.

Through the decades, multiple availability models have been published and successfully applied [11, 19], among which Bayesian networks have gained significant acceptance within the industry and research [27–30]. This work will mainly use Bayesian networks as a mathematical framework to predict the availability of cloud applications, due to their powerful modeling formalism to express complex fault dependencies and uncertainty between components [31–33]. Nevertheless, similar to the *state explosion problem* of Markov models, Bayesian networks suffer from the *exponential memory blow-up* problem [34], where the memory space grows exponentially in the edge degree of the nodes in the network. This problem limits modeling capabilities, making it infeasible to model large-scale cloud services.

Guiding architectural decisions and proposing concrete services is a recurring research problem in the web service community, known as the *web service composition problem*. Research on web service composition is broad and ranges from tooling and language support to suitable matching criteria and search algorithms to find potential solutions rapidly [35]. The goal is to find and connect web services to one (web) solution automatically, so the union of the individual web services covers the desired functional requirements. Many proposed solutions also include QoS constraints in their composition process. However, availability plays a marginal role in most solutions and is often evaluated with a simple reliability block diagram of the system [36–40]. Few proposals consider agnostic QoS functions in their matching procedures, leaving it to the developers to interface with their availability prediction method [41]. Availability prediction is not the primary concern in the composition problem. Therefore, most models only assume independent fault probabilities between web services and do not provide flexibility to change the granularity of the components. Needless to say, the cloud service composition problem is strongly related to the web service composition problem [42, 43]. The differences are subtle and mainly focus on service semantics, discovery, and additional QoS attributes specific to cloud computing, such as elasticity. However, approaches, as suggested by Bekkouche et al. [41], where they regard availability as an agnostic function in their service matching routine, provide untapped potential to use advanced availability models.

The web service community has rapidly realized that the resulting solution space to compose a web service is getting large. Therefore, they experimented with different search heuristics to increase performance by finding near-optimal solutions in a feasible time. Their year-long experience with meta-heuristics such as tabu search [44], genetic algorithms [36, 38], ant colony optimization [39], har-

mony search [41,45], or hybrid algorithms [46] to name a few, showed that meta-heuristics provide good performance in approximating near to optimal solutions.

This begs the question, how to integrate meta-heuristics in the architectural decision process for cloud computing? On the one hand, we have a rich set of solutions to solve the web service composition problem. On the other hand, we have a rich set of powerful modeling languages to design cloud applications. Bergmayr et al. [47] published an in-depth survey on cloud modeling languages, discussing models such as CAML [48,49], Blueprint [50,51], GENTL [52], and Tosca [53,54]. Most of these models focus on application orchestration and deployment and do not provide matching mechanisms that consider QoS constraints and (operative) cost simultaneously.

1.3. Challenges

Cloud computing has largely impacted how developers design and implement software nowadays. Advances in virtualization technologies and novel architecture paradigms like cloud-native applications and microservice architectures are just some of the contributions brought by cloud computing. As such, cloud computing divides its service offerings into three main categories: IaaS, which offers container and VM services; Platform-as-a-Service (PaaS), which offers pre-configured and deployed platform applications such as database systems and runtime environments; and Software-as-a-Service (SaaS), which offers complete software solutions that run out of the box. Intermediate service categories such as Function-as-a-Service (FaaS) or Data-center-as-a-Service (DCaaS) also exist. As a result, Everything-as-a-Service (XaaS) has become an umbrella term for all service offerings.

As discussed in the previous section, related work has focused mainly on providing specialized availability models for IaaS and PaaS offerings, considering only failure modes caused by network communication or the cloud infrastructure. The challenge is building a cloud availability model that generally supports XaaS. This means the model should be able to express and incorporate failure modes caused by communication and infrastructure faults, while also addressing the fault tolerance semantics of the services that constitute the cloud application.

However, with the popularity of e-commerce, the Internet of Things (IoT), and video streaming, the FaaS paradigm has emerged. FaaS are stateless-compute services. Their service instances can be started and killed rapidly, making it possible to react fast to dynamic load demands. The availability definition of FaaS

is strongly tied to performance and is a special case when discussing how FaaS implements fault tolerance. It is clear that when the service fails, i.e., no service instance can be reached, the service is unavailable. However, how does availability change when no additional service instances start in the face of an increasing load? The remaining service instances will receive the additional load, increasing the service time until, eventually, the service overloads. From an availability perspective, FaaS provides redundant instances, which are semantically similar to redundant systems where at least k-out-of-n instances need to work to sustain the load. Transaction-oriented database systems tend to have between three or five replicas. However, FaaS offerings, or even key-value (NoSQL) data stores, tend to have tenths or hundreds of instances, primarily for performance and data locality reasons [55]. This new order of magnitude in the number of service instances apparently influences existing availability models. The resulting state-space of the availability model might become too large to solve. Reliability engineers generally circumvent this problem by aggregating the model or simplifying the modeling assumptions, reducing the expressiveness of the model in favor of computability. Hence, we need scalable availability models to model large-scale cloud services.

Assessing the availability of cloud applications based on their architectures is just a preliminary step. Developers and architects would use such models to probe for different architectural alternatives and to search for services that best fit their availability requirements. Providing an adequate cloud modeling language that enables the automatic search for cloud services with the help of a recommendation system is a desirable goal; however, it introduces new challenges. In the early beginning of cloud computing, major competitors offered few services. As for the year 2021, AWS offers over 200 services [56], Microsoft® Azure (Azure) provides about 200 services [57], and Google Cloud offers about 170 services [58], which does not include the various service plans and configuration options of each service. For example, Azure and AWS offer more than 400 instance types for their VM service, while simultaneously offering container services as well. Hence, it becomes a tedious task to search for the correct service plan. Since complex cloud applications use more than one service, finding services that best fit, even in an automatic manner, becomes a computationally challenging task, due to the potential large number of configuration combinations.

1.4. Contributions

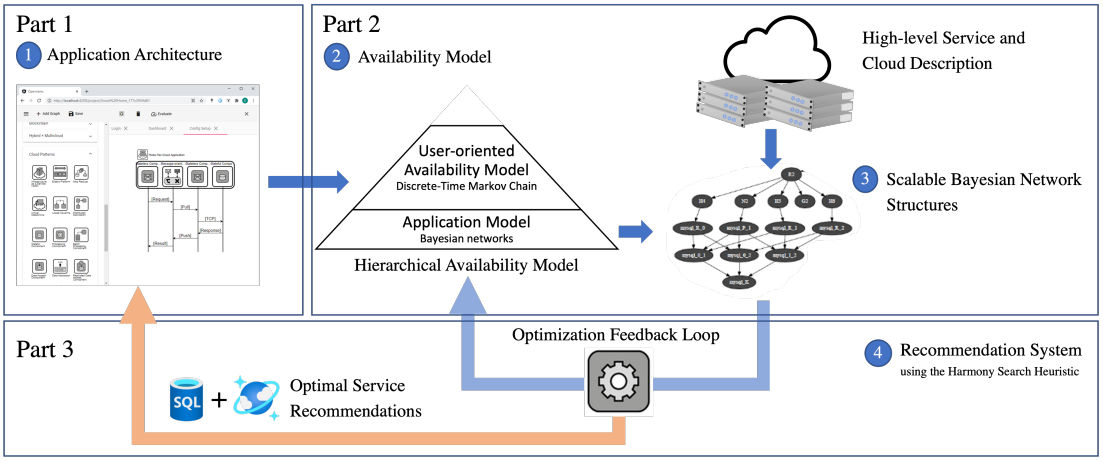


Figure 1.1.: The solution workflow.

This thesis is a combination and extension of the contributions presented in [59–63]. This work has two goals. First, it aims to provide methodologies for predicting the availability of cloud applications during the early stages of the development process. Second, this work should support cloud developers and architects in their architectural decision-process by designing and implementing a recommendation system that suggests cost-optimal cloud services that fulfill given availability constraints. To solve these goals, this work proposes a three-part solution workflow, depicted in Figure 1.1. This is a high-level roadmap on how to attain the thesis goals. The workflow is divided into three parts. The first part focuses on providing modeling tools to model cloud applications. The application models form then the input for the availability model in part two. The second part focuses on predicting the availability of a cloud application, whereas the third part focuses on the recommendation system that uses the results of the second part to search for fitting services that fulfill the optimization criteria. Once the recommendation system has found fitting services, it returns its results to the architectural design tool. Blue circles highlight the contributions in the corresponding blocks. In the following, we discuss each contribution in detail:

1. The first contribution is a novel cloud modeling language called Clams, a **CL**oud **A**pplication **M**odeling **S**olution, which was published in [59]. Clams is a scenario-based cloud modeling language to model application architectures that act as input for the availability model and recommendation system. Clams combines message sequence charts with cloud computing

patterns [64] to describe application-level functionalities, referring to it as application *scenarios*. Cloud computing patterns are used as structural components to represent architectural placeholders. This provides the flexibility to specify functional requirements on a higher level. A refinement process will then replace the patterns with services that provide a concrete solution to the high-level functional requirements. As discussed later, the recommendation system will perform the refinement. It will search for cost-minimal services for a given availability threshold. An additional contribution is an in-depth case study on cloud application architectures, which consists of thirty-one real-live architectural examples from the Azure cloud. The results show that refining architectural patterns to concrete services can result in a high number of possible service matches, influencing the performance when searching for optimal services in general. Finally, this work also presents tooling support for Clams, called OpenClams[1], which offers a visualization tool to create and interact with Clams models. It provides a programmable interface for external applications to implement complex evaluation workflows like our recommendation system. Overall, the author of this thesis has contributed 90% of the scientific content of the paper.

2. The second contribution is a novel availability model to assess the availability of cloud applications. The model addresses the availability of cloud applications from the end-user perspective by computing the availability of the user's usage profile, i.e., a set of application scenarios that users invoke in a specific order to attain their desired outcome. The Clams model provides concepts to model usage profiles and scenarios. Modeling scenarios as an interconnected set of cloud services that represent an application-level functionality. To compute the availability of a usage profile, this work proposes a hierarchical availability model as shown in Figure 1.1, where the root model of the hierarchy uses an adaptation of the user-oriented software readability model by Cheung [65]. This is a DTMC representation of the usage profile, to compute its availability value. The DTMC model takes the availability values of the corresponding scenarios that are part of the usage profile as input parameters. Computing the availability of scenarios is subject to the second layer of the hierarchical availability model, which forms the principal contribution in the second part. Here, the second con-

[1] https://github.com/openclams [Last visited: 5 Mai 2022]

tribution presents a novel Bayesian network availability model that predicts the availability of cloud services. The model considers a flexible number of failure modes caused by the network and the cloud infrastructure at any preferred level of component granularity.

Especially in the case of replicated services, the model will also account for the likelihood that sufficient replicas can communicate with each other in the present of network partition failures, where sufficient refers to the specific fault tolerance properties of the service, such as majority sets, for example. The model supports multiple replication specifications such as voting-based, weighted-voting and the special case of read-one/write-all replication. Overall, the author of this thesis has contributed 80% of the scientific content of the corresponding paper [62], which primarily introduced the Bayesian network model to assess a single cloud service. Additionally, the author introduced the hierarchical availability model and extended the Bayesian network model to support scenarios in this thesis.

3. So far, Bayesian networks exhibit an exponential memory growth in the edge degree of their nodes, making Bayesian networks infeasible to model large-scale cloud service. To overcome these memory limits and to model large-scale cloud services, scalable Bayesian network structures are needed. The proposed availability model from the second contribution uses a k-out-of-n voting gate representation in its Bayesian network model. Bayesian network nodes representing these gates are susceptible to the exponential memory growth problem [34]. The corresponding conditional probability table of the random variable that encodes the boolean expression of the k-out-of-n voting gate grows exponentially with the number of input events. This work proposes a scalable k-out-of-n voting gate representation for Bayesian networks published in [60, 61], which can be used with any standard (exact or approximate) inference algorithm, making it universally applicable in other Bayesian network models as well. Numerical evidence shows that the scalable k-out-of-n voting gate reduced the memory demand from initial exponential to polynomial in the number of input events, making it possible to model large-scale replicated service. Overall, the author of this thesis has contributed 90% of the scientific content of the papers. Also, the author has introduced the scalable *noisy* k-out-of-n model for Bayesian networks in this thesis.

4. Predicting the availability of a cloud application based on its architecture

is just one core topic of this thesis. The next step is applying the findings to design and implement a service recommendation system. The recommendation system uses a meta-heuristic approach that allows for a broader set of optimization constraints, including but not limited to availability and cost. As shown in Figure 1.1, the third part of the high-level overview shows that the recommendation system uses the availability model to implement a feedback loop that probes for different service combinations. Once the termination criteria is reached, the recommendation system returns the most fitting services it has found. Generally, the recommendation system refines a given high-level specification of a component-based architecture to an architecture consisting of concrete service plans. The proposed refinement algorithm uses refinement trees to encode the potential solution space of a cloud pattern, where this work shows how to get from a refinement tree notation to a suitable input for a meta-heuristic to solve the refinement problem. In general, meta-heuristics strive to optimize a given search objective stated as a loss function. Hence, this work shows how to design a suitable loss function to minimize cost while also considering QoS constraints, i.e., availability. Evaluations based on the case study with thirty-one architectural examples from the Azure cloud validate the feasibility of the recommendation system, by using the Harmony Search [66] algorithm as a meta-heuristic for the optimization approach. Overall, the author of this thesis has contributed 85% of the scientific content of the corresponding paper [63].

Finally, a number of student theses [67–73] have also contributed in parts to the implementation of OpenClams and the search for suitable availability models.

1.5. Document Structure

The document structure of this thesis follows the solution workflow shown in Figure 1.1.

Chapter 2 This chapter introduces Clams. Clams uses cloud computing patterns as structural components to model abstract architectures. Developers can use these abstractions to express architectural intentions, which work as placeholders for later refinements. Examples show how to use Clams and how to model a cloud application.

Chapter 3 This chapter introduces the hierarchical availability model to compute the availability of a usage profile.

Chapter 4 This chapter introduces the Bayesian network model to predict the availability of application scenarios.

Chapter 5 This chapter discusses scalable Bayesian network structures and introduces the scalable k-out-of-n voting gate representation for Bayesian networks.

Chapter 6 This chapter introduces the general framework to design and implement the service recommendation system, providing an interface to express custom search constraints. The system uses Clams as input to extract abstract components, build the solution space, and propose appropriate services according to a given loss function.

Chapter 7 The final chapter concludes this thesis and provides an outlook on future research topics.

Chapter 2.

Modeling Cloud Applications

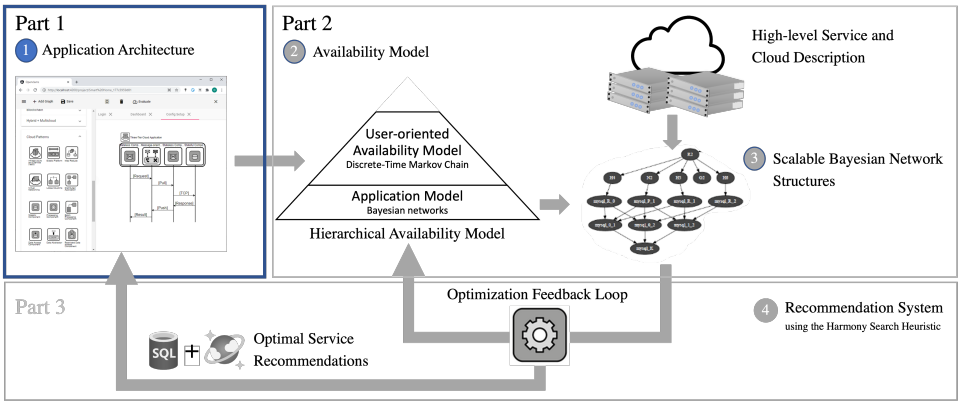


Figure 2.1.: This chapter focuses on part one of the solution workflow.

To predict the availability of a cloud application during design time, we need to describe the application in a formal language first. This modeling language forms the beginning of the proposed workflow, as depicted in Figure 2.1. The model should help developers express their architectural intentions, by using structural elements that act as placeholders. Existing cloud-specific models, such as cloud modeling languages, emphasize strongly on application deployment and little on design-time evaluations and recommendations of cloud services. Therefore, this chapter introduces a novel cloud modeling language called Clams, a **CL**oud **A**pplication **M**odeling **S**olution [59]. Clams is a scenario-based modeling language using cloud computing patterns as abstract components to model service uncertainty and to guide developers in implementing best architectural practices.

To understand the implications of patterns on the actual solution size of concrete services, this chapter also introduces a case study with real-live applications examples, which forms the baseline for cloud applications in this work.

Section 2.1 presents the motivation behind Clams, while Section 2.2 delivers a brief overview of the main modeling concepts. Section 2.3 introduces the system

model, followed by Section 2.4, which expands on the formal definition of Clams. Next, Section 2.5 provides a case study, where we analyze application examples from the Azure cloud. Afterwards, Section 2.6 describes the tooling support and development framework of Clams. Section 2.7 discusses related work. Finally, Section 2.8 summarizes this chapter.

2.1. Introduction

The previous chapter introduced the research goals and outlined a solution based on the workflow shown in Figure 2.1. This chapter focuses on the first part of the workflow, namely, the modeling language to describe the architecture of a cloud application. We need a modeling language that enables developers to describe their architecture in order to assess its availability and suggest cloud services that suite application scenarios. Ideally, this modeling language should support the developer in modeling a cloud application efficiently, in a cloud agnostic manner, and enable multi-cloud application architectures. The model should comprise sufficient information to evaluate its availability and offer an interface to suggest cloud services by a recommendation system. As a result, the modeling language needs to not only support service placeholders but also indicate what services make semantically sense within the context of the architecture since the recommendation system must propose those services which are sensible candidates to substitute the architectural placeholders.

Considering the need for architectural placeholders and cloud agnostic modeling, the model-driven architecture paradigm (MDA) provides the general concept of realizing the interface for this work. MDA distinguishes between platform independent and dependent models, whereby the former gets refined into the later [74]. Several component-based architecture modeling solutions have adopted this idea by introducing the notion of abstract components, which get replaced by components that represent concrete solutions [75–77]. Since the second research goal is to suggest cloud services, applying the MDA concept to architectural refinement is reasonable. A developer models an application in a platform-independent model using architectural components, representing abstractions of the concrete service plans. The followup refinement can then be used to recommend services that fulfill a given optimality criteria. How to realize this refinement and how to build the recommendation system is part of Chapter 6. This chapter focuses on the modeling language to design platform-independent and platform-dependent architectural models.

Cloud modeling languages provide a promising approach in modeling cloud applications in contrast to general-purpose modeling languages such as UML, or architecture description languages. Bergmayr et al. [47] have surveyed a large body of cloud modeling languages, arguing that a domain-specific approach focusing on cloud computing vocabulary increases the modeling power by introducing more semantics to a model that is otherwise difficult to express with, e.g., UML. Some cloud modeling languages make use of cloud computing patterns [64], or pattern languages in general [78], as structural components to model abstractions that later get refined into concrete services. This approach fits the aforementioned MDA concept. However, current cloud modeling languages focus on deployment and orchestration of cloud applications, implying the existence of finished application artifacts that are usually not available in the early development phases. Moreover, since existing cloud modeling languages model an application from a deployment perspective, it is difficult to infer how end-users perceive an application from an availability point of view. A deployment model contains all the services that comprise the application, while users typically issue requests to invoke application-level functionalities that involve a subset of the deployed services. So far, cloud modeling languages do not per se give rise to application-level functionalities. Hence, assessing the availability that all services are up and reachable can deviate from the availability of individual application-level functionalities.

Users exhibit behavior patterns in how they interact with cloud applications. A user can be a human, an IoT device, or even some other cloud application from a third party that consumes an API interface of the current application. This work will refer to these entities simply as (end-)users or customers. The behavior of a user can contain requests to different application-level functionalities. For example, let us consider a simple web application with user authentication. First, a user logs in, which might involve a front-end service such as AWS Lambda and a database service. Afterward, they tend to gather information via a home screen, e.g., a product catalog from an e-commerce online store or statistics for a business case loaded from an analytics service. The home screen might involve different services to acquire the desired information independent of the use case. Suppose the home-screen calls for action, and the user has to request yet again a different application-level functionality that involves another subset of services. Calling or invoking subsequent actions can be modeled as a usage profile, including probabilities between the transition from one request to another to model nondeterministic user behaviors. Generally, we will regard the service subset comprising an application-level function as a *scenario*. Each user request

invokes a specific scenario in the application. The usage profile is a graph-based representation of a user's behavior, where nodes refer to scenarios and directed edges represent the transition from one scenario to another. Consequently, the services that form the application deployment are the union of all the services in all application scenarios.

So far, cloud modeling languages do not include the user perspective, but as argued early, individual application-level functionalities can result in different availability values. Hence, a usage profile is important to assess the availability of a given behavior pattern. Modeling applications that combine user behavior and scenarios are subject to several standards, such as the UML sequence and communication diagram notions or the basic message sequence chart (MSC) and high-level MSC standard by the International Telecommunication Union (ITU). The latter has been used by Rodrigues et al. [79] to model and assess the reliability of generic systems with complex usage profiles. Their model uses basic MSCs to define scenarios and high-level MSCs as usage profiles. The basic MSC represents the scenario as a message sequence chart where instances are (cloud service) components, using message events to interact with other components. The usage profile is a Markov chain model of the transitions between the scenarios. A transformation algorithm translates all scenarios and the usage profile into a labeled transition system [80], which then gets again translated into the existing user-oriented software reliability model proposed by R. C. Cheung [65]. Consequently, we could substitute components with cloud services and apply the same transformations concepts. However, we need a modeling language that enables platform-independent modeling with emphasis on cloud computing so that a recommendation system has the required semantics to suggest optimal services. This raises the questions of how to realize architectural placeholders and how to implement a cost model that considers the wide variety of configuration options of each cloud provider? To answer these questions, the standard MSC notion has to be significantly extended with domain-specific elements for cloud computing. However, the promising results from Rodrigues et al. inspired this work to investigate the potential solution of extending the MSC notion.

Therefore ,the final contribution of this chapter is the extension of the MSC notion to a novel cloud modeling language called Clams, a **CL**oud **A**pplication **M**odeling **S**olution. Clams is a scenario-based cloud modeling language, which uses basic MSCs to model scenarios, i.e., application-level functionalities, and usage profiles to enable a user-oriented availability assessment of the application. Moreover, Clams uses cloud computing patterns [64] as structural components to

represent architectural placeholders, which also guide developers in implementing best practices in application development. A refinement process can then replace these patterns with concrete service configurations that constitute the solution of the respective cloud computing pattern. The refinement process uses the functional semantics defined by the patterns as filtering criteria to propose only those services with matching functional semantics. However, Clams is not only a theoretical language. It has been implemented as part of an open source project called OpenClams[1], which offers a visualization tool to create and interact with Clams models. It provides a programmable interface to external applications, such as a recommendation system, to implement complex evaluation workflows for the refinement process. So, the contributions of this work also include the design and implementation of tooling support. An additional contribution is a case study of 31 cloud architectural examples from the Azure cloud to gather statistics on the average size of existing architectures and the expected solution size when refining architectures. These are important reference parameters for the followup chapters when we build and evaluate availability models.

2.2. Overview

This section introduces Clams with the help of an example to provide an overview of the main modeling concepts. A formal definition that covers all modeling elements of Clams follows the next sections. Here, we discuss the architecture of a back-end application for a smart home solution, where a mobile client application consumes the back-end's API. The back-end offers the following application scenarios:

Login User authentication.

Dashboard Overview of all available smart home devices and their current state.

Change State Change device state, such as setting the temperature for a smart thermostat.

Logout Closing the current user session.

Let us assume the application uses cloud services from the Azure cloud. Figure 2.2 depicts the Clams model of the cloud application, showing four scenarios

[1] https://github.com/openclams

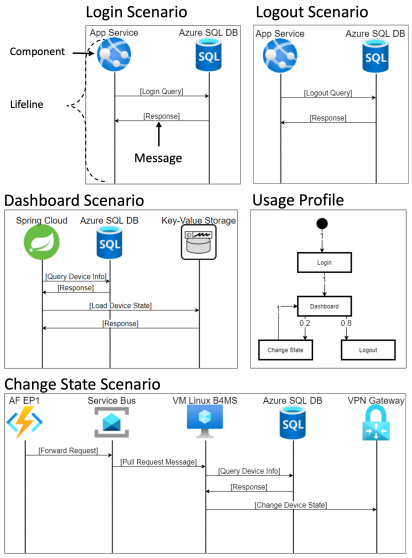


Figure 2.2.: Scenarios and usage profile of the smart home back-end application.

represented by the four MSCs and one usage profile in the middle-right part. First, let us start with the scenarios. The first box in the upper-left corner of Figure 2.2 depicts the login scenario, which consists of an MSC with two services. The scenario uses an instance of the *App Service* for the login logic, which queries an *Azure SQL database* to check if the user credentials are correct. Each *component* represents a concrete service or abstraction, such as a cloud computing pattern, and has a lifeline along which they exchange messages with other services. The components of the logout scenario use the same services as the login scenario. However, the dashboard scenario uses different components since the developers wish to implement a more sophisticated dashboard that accesses device logs from a key-value storage. Hence, they decided to use *Azure Spring Cloud* to build the dashboard logic. The service queries the same database used by the login and logout scenarios to retrieve a list of the user's devices and then load the corresponding device logs from the key-value storage. Now assume that the developers might be uncertain about what key-value storage to use during the current design phase. So they choose to express their design intention by using the key-value storage pattern as an abstract component. This pattern acts, on the one hand, as a placeholder and, on the other hand, as a functional description of a service that should replace it so that in a followup step a recommendation system can refine the key-value storage pattern to a concrete service. However, abstract components are not limited to pattern expressions alone but can also represent service offerings at a categorical level. For example, consider the change scenario from Figure 2.2 where an *Azure Function* captures user requests and

forwards them to the *Service Bus*. The Service Bus component represents a wide range of potential configuration options to instantiate a concert Service Bus offering. Hence, this component can be further refined to a specific service plan. In contrast, the VM instance is already a concrete service, consisting of a Linux (Ubuntu) machine with the configuration option B4MS.

Usage profiles describe a user's interaction with the application as a Markov process to address the stochastic nature that constitutes a user's behavior with regard to invoking scenarios. Each node in the usage profile refers to a scenario. Each edge contains a transition probability to indicate the likelihood for a user to transition to the next scenario after the previous scenario was successful. A start dot indicates the initial scenario. For example, in Figure 2.2, the login scenario is the first to happen, followed directly by the dashboard scenario. Afterward, the user changes with 80 % probability the state of a device and goes back to the dashboard or logs out with a 20% chance. In the context of availability, those users who log out directly after taking a glimpse at he dashboard would not notice if the change state scenario is unavailable due to a VM failure. Hence, availability is subject to the user's interaction with the application.

In summary, the additional component semantics distinguishes Clams from the general MSC notion. It includes cloud-specific information such as associations of services to cloud providers and their regions (not shown in the figure) and a uniform (monetary) service cost model. Moreover, Clams also supports abstract modeling with patterns. Patterns are structural elements in Clams and encode the solution space of potential service configurations. They can have different levels of abstractions, from highly abstract, which narrows the service selection little, to less abstract, narrowing the service selection to a few concrete service plans. A less abstract pattern can result in a set of concrete services that are a reasonable solution for a more abstract pattern. In this case, the less abstract pattern is a refinement of the more abstract pattern. Clams organizes and encodes this refinement relation as a tree graph, also called a *refinement tree*. The refinement tree is an implicit structure hidden behind each component in the MSC and used to navigate between different levels of abstractions. A detailed description of the refinement tree notion follows in the upcoming sections.

2.3. System Model

This section introduces the main system assumptions with regard to cloud applications. The model assumes that a cloud application is an interconnected set

of one or more cloud services distributed across multiple cloud regions of one or more cloud providers, essentially defining a multi-cloud application. The model only considers those services that are visible to developers. That is, a service is *visible* only if developers perceive the service as an active part of their application, self-serving interface, or if it can be booked from the service catalog.

Components have semantics that contain details about the services they represent. *Concrete* components contain sufficient information to identify a concrete service configuration, including its cost plan, resource requirements, and QoS information. However, the model also supports abstractions. An *abstract* component contains the (abstract) functional specification of a service. Therefore, multiple service plans can match the description of an abstract component. For instance, abstract components can be structural patterns [64,81,82] which model and express the best architectural practices for recurring problems. Services that act as a platform to execute the customer code-base of the business logic are components as well. These are the VM or container services that are part of the IaaS layer or the runtime environments that are part of FaaS offerings. This work does not consider the custom code-base or image artifacts that are deployed to the services. The application model only accounts for the underlying cloud services required to execute them.

The model does not further consider a division of service components into their specific instances. For example, a database service that consists of three replica instances is represented by one logical component in the model, which entails the configuration that the replication degree is three for this service. Therefore, the same database service with five replicas is a different configuration and represents a different component in Clams. Note that the model also allows multiple instantiations of the same component if required to denote that the same service plan has been booked several times.

Moreover, the model assumes that user behavior is stochastic, modeling the transfer of control between scenarios as a DTMC, which this work refers to as a *usage profile*. Hence, the invocation of a scenario depends solely on the previously invoked scenario. A usage profile encompasses an average user interaction with the application. A usage profile has an initiating scenario that starts the session and at least one scenario that acts as its final state, forming the termination of the session. Reaching and executing the final scenario without failure means that the application is available from a user's perspective.

2.4. Clams Model

The previous sections have introduced Clams informally. This section continues with the formal description of Clams, introducing the underlying architectural model and important concepts like cloud components, templates, and refinement trees to design application architectures.

2.4.1. Scenarios and Usage Profiles

Cloud modeling languages often use existing modeling or architectural description languages as their formal foundation and extend them by employing domain-specific elements from cloud computing. As mentioned earlier, Clams uses the MSC notation by the ITU as a formal foundation. More precisely, it uses a slightly adapted version of the scenario and usage profile model used by Uchitel et al. [80, 83].

Let us start by defining $\boldsymbol{\mathcal{C}}$ as the set of all available abstract and concrete cloud components to model a scenario. The core element of a scenario is a component, which consists of a lifeline.

Definition 1 (MSC Component) *A component is a three-tuple* $\mathcal{C} = (ID, \boldsymbol{e}, <_e) \in \boldsymbol{\mathcal{C}}$ *which has a reference key ID that identifies a specific service plan/configuration or abstract component such as a cloud computing pattern.* $\boldsymbol{e} = e_{in} \cup e_{out}$ *is a set of input and output events that have a linear ordering according to a set relation* $<_e \subseteq (\boldsymbol{e} \times \boldsymbol{e})$*, which represent the temporal ordering of the events along the life-line.*

Each output event needs to be connected with one input event of another component via an arrow representing a *message*. Each message event $e \in \boldsymbol{e}$ can also be annotated by a set of key-value attributes to enrich the message with additional semantics if required. This feature has been introduced to enable a richer set of analysis possibilities, apart from the here discussed availability analysis methodologies.

Next, we group components to scenarios. A scenario is a vector of components that form a message sequence chart. Each scenario should ideally represent one application-level functionality.

Definition 2 (Scenario) *A scenario is a structure* $\mathcal{S} = (\langle ID_i, \boldsymbol{e}_i, <_e^i \rangle_{i=1}^n, \mathrm{f}_e)$*, containing a finite set of components and* f_e *a bijective function*

$$\mathrm{f}_e : \cup_{i=1}^{n} e_{\mathrm{out},i} \rightarrow \cup_{j=1}^{n} e_{\mathrm{in},j}$$

that maps the output events $e_{\text{out},i} \in \boldsymbol{e}_i$ *of the* i*-th component with the input events* $e_{\text{in},j} \in \boldsymbol{e}_j$ *of a different component* j*, such that the passing of messages between output and input events are causally ordered.*

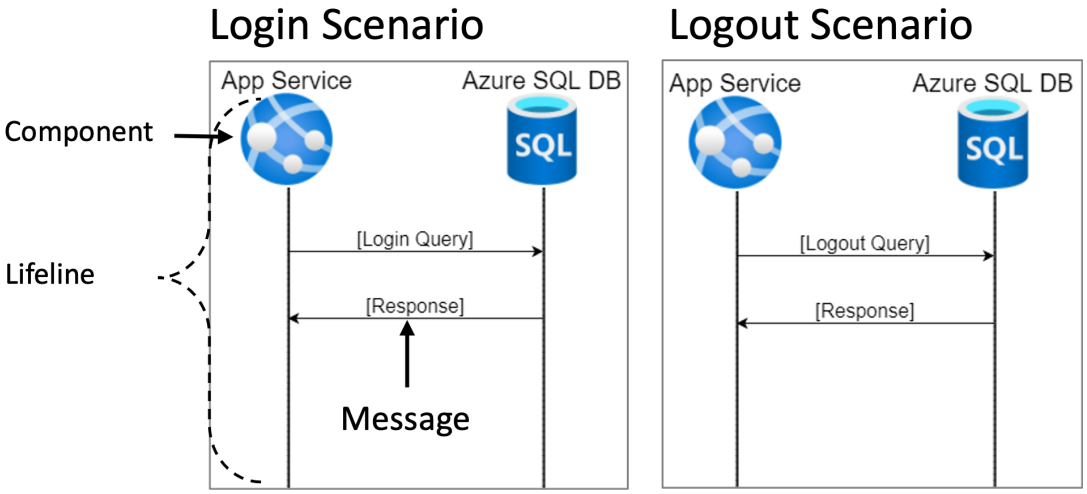


Figure 2.3.: Scenario examples with two component.

For example, Figure 2.3 shows the login and logout scenario from the smart home application discussed at the beginning of the introductory section. Each scenario has two components referring to cloud services from the Azure cloud: an App Service and an Azure SQL database service. Since both App Service components have the same name, Clams regards this as using the same service component. Hence, component names in scenarios decide whether or not the same service is used or if a separate service is meant.

Clams uses usage profiles to model the behavior of users. A usage profile is a DTMC with an additional *Init* state to mark the start state. Nodes reference scenarios, whereas the edges define transition probabilities from one scenario to another. For example, Figure 2.4 shows the usage profile of the smart home example. The profile defines the sequence of potential user requests with appropriate scenario invocations. Hence, in this example, it is certain that a user session always begins with the authentication followed by the display of the dashboard. Afterward, it is uncertain whether or not the user logs out with an 80% chance or continuously interacts with the smart devices with a 20% chance. Formally, a usage profile is defined as follows.

Definition 3 (Usage Profile) *A usage profile is a graph structure* $\mathcal{U} = (\boldsymbol{\mathcal{S}}, \mathit{Init}, E_{\mathcal{U}}, P_{\mathcal{U}})$*, where:*

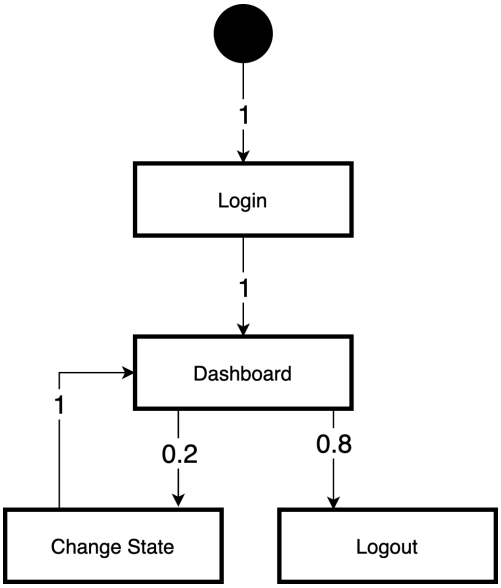


Figure 2.4.: Usage profile example of the smart home application.

- $\boldsymbol{\mathcal{S}}$ *is a finite set of scenarios defining the nodes of the graph.*
- $Init \in \boldsymbol{\mathcal{S}}$ *is the starting scenario that initiates the profile.*
- $E_{\mathcal{U}} \subseteq \boldsymbol{\mathcal{S}} \times \boldsymbol{\mathcal{S}}$ *are the edges between scenarios.*
- $P_{\mathcal{U}} : E_{\mathcal{U}} \rightarrow \mathcal{R}$ *is a probability distribution where the probabilities of all outgoing edges of a node sum up to one, i.e.*

$$\forall \mathcal{S}_i \forall \mathcal{S}_j : (\mathcal{S}_i, \mathcal{S}_j) \in E_{\mathcal{U}} \implies \sum_j P_{\mathcal{U}}((\mathcal{S}_i, \mathcal{S}_j)) = 1.$$

The usage profile allows for self and back-references. However, depending on the evaluation and analysis tools, they might impose constraints with regard to loops in the usage profile or might even require at least one absorbing state, i.e., a scenario that eventually terminates the session.

2.4.2. Component Model

Figure 2.5 shows the class diagram of the component meta-model. Clams organizes components by cloud providers. A component is either a concrete service plan, also referred to as a concrete service, an abstract component, which is a categorical grouping of services of the same type, or, even more generic, a cloud computing pattern as introduced by Fehling et al. [64]. Moreover, a component can also be a template, which is a set of nested components. Cloud computing

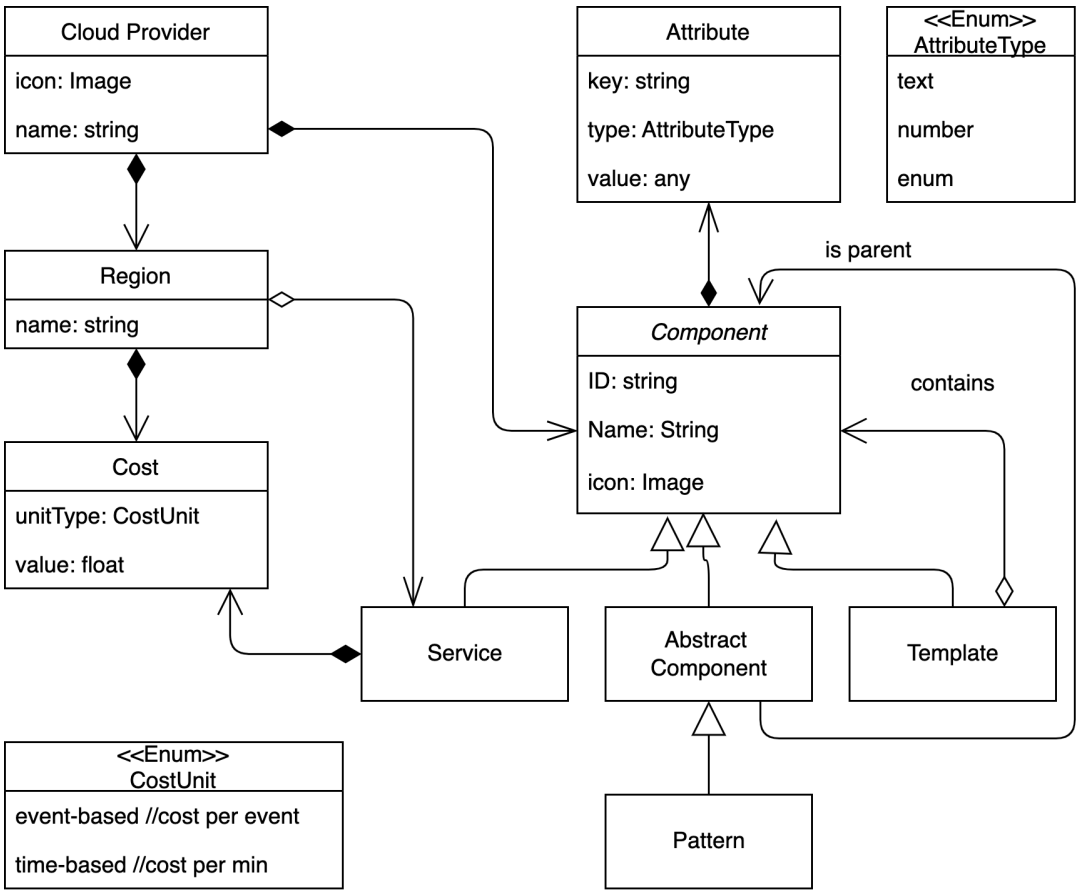


Figure 2.5.: Cloud component meta-model

patterns can be either structural or behavioral. This work only considers structural patterns since they relate to concrete services. As for [84], a pattern consists of an icon, a name, and implicitly of the description of the recurring problem, the context in which the pattern can be applied, and the appropriate solution. Abstract components, patterns, and templates are independent of cloud regions. However, services might be offered only in certain regions; therefore, a service has a reference to all regions where the service is available. In Figure 2.5, cloud providers, regions, and components use a name and an ID key for identification within scenarios. Cloud providers and components also have an image field to display the component in the modeling application or other tools if required. Additional fields are also possible, but not essential here. Moreover, the component class can have multiple attributes. An *attribute* is a simple key-value pair, which developers can use for component annotations. These annotations can provide component descriptions or add more semantics to a cloud component if needed.

Component Refinement

Component refinement is an essential feature of Clams which transforms technology-independent architectures into technology-dependent models. This work uses the refinement definition by Davies et al. [85], which states that a component $\mathcal{C}_i \in \boldsymbol{\mathcal{C}}$ is a refinement of $\mathcal{C}_j \in \boldsymbol{\mathcal{C}}$, if and only if the concrete components that match the functional specification of $\mathcal{C}_i$ also match $\mathcal{C}_j$. Clams implements this refinement notion by arranging components in so-called *refinement trees*, as introduced by Falkenthal et al. [82] in the context of pattern refinement.

The cloud component meta-model contains an aggregation between the abstract component class and the component class to implement the popular composite design pattern, known from object-oriented programming [86]. This aggregation enables the construction of tree structures where cloud components form a parent-child dependency representing the refinement relation. The tree structure should contain cloud components starting from the most abstract component and provide more specific solutions with every subsequent component. Parent nodes represent solutions to abstract architectural problems, whereas child nodes provide a more concrete problem description and a more specific solution set. Thus, abstract components are the inner nodes of the refinement tree, and concrete services are the leave nodes since they provide a concrete solution and cannot be further refined.

For example, Figure 2.6 shows a small excerpt of a possible refinement tree for stateful services. It has the stateful component pattern as the root node, addressing the general intention to persist state. A more specific solution is the blob storage or databases pattern, which has further child components that narrow down the initial problem of persisting data. Next, one can refine the database pattern using the relational database pattern or a key-value storage pattern. These components have a more specific solution for the initial problem. At this point, the refinement tree becomes more concrete. At the lower levels of the tree, it further divides the services into their sub-categories. For example, for the MariaDB components, further refinements distinguish between specific service categories, such as MariaDB Basic, MariaDB General Purpose, and MariaDB Memory Optimized. Finally, the leaves contain concrete service plans.

Abstract components do not make any assumptions about the (programming and communication) interfaces offered by their successor nodes in the refinement tree. An inner node, possibly a cloud computing pattern, defines the general intent to solve a particular recurring problem according to best practice, explicitly

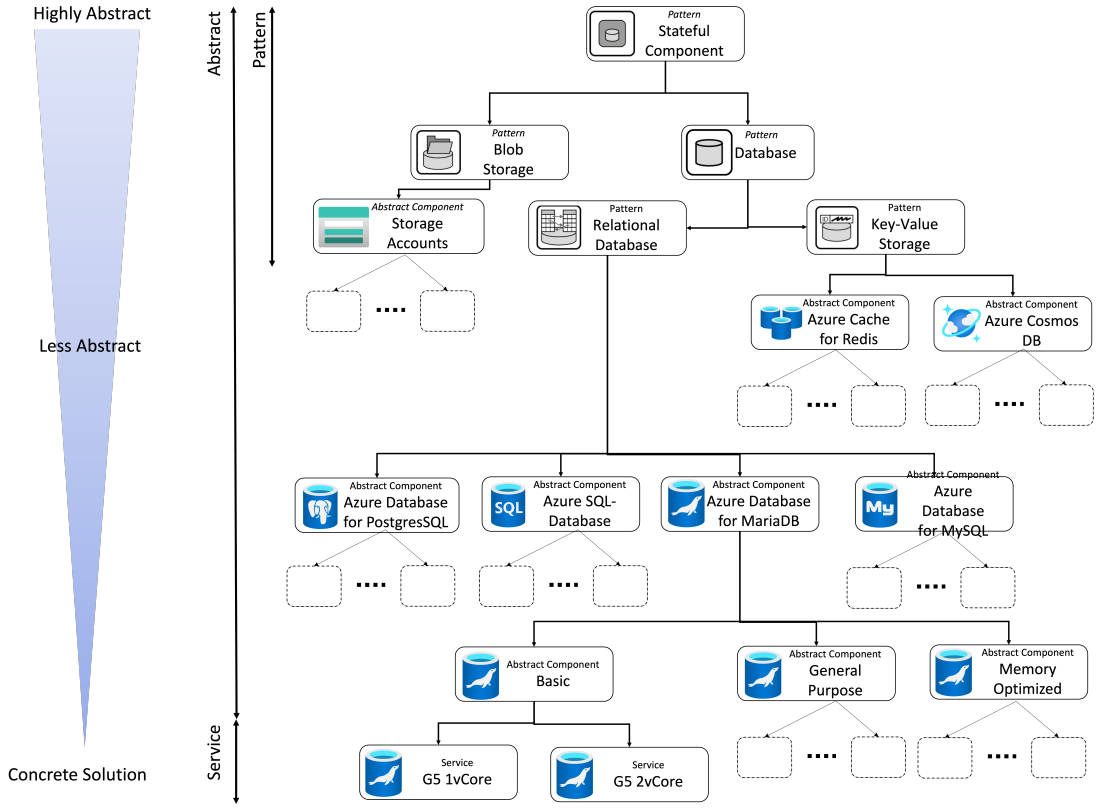


Figure 2.6.: Refinement tree for stateful services in the context of the Azure cloud

or implicitly defining the solution as functional requirements according to the corresponding level of abstraction. Child nodes represent potential solutions to the parent's problem. These nodes can be patterns again, which refine the stated functional requirement of the parent node more explicitly, or a service plan that implements the solution directly.

Each node in the refinement tree is a component that can be used when modeling a scenario. The main goal of the refinement tree is to implement a pattern language [78], which encodes the refinement association efficiently. It can be used to navigate different levels of component abstractions, and constrain the search space for service suggestions. The leave nodes in the refinement sub-tree starting at the abstract component are the possible solution candidates. Hence, if the developer uses the stateful component, a recommendation algorithm can consider all the leaves of the refinement tree as its possible solution space. If the developer selects the generic MariaDB component, then the selection algorithm only considers specific MariaDB service plans. Consequently, developers can use abstract components to express their uncertainty with regard to what service to use. Depending on the design stage, a developer might not have decided yet

what database system to use. Patterns can represent this decision uncertainty according to their abstraction level. If a developer chooses a service directly, the decision is certain. No further search is required. The more abstract a component, the higher the selection uncertainty and the larger the potential solution space. A potential measure to quantify the selection uncertainty is to compute the complement of the likelihood of selecting an optimal service when refining a component $\mathcal{C} \in \boldsymbol{\mathcal{C}}$.

$$u = 1 - \frac{|\boldsymbol{\mathcal{C}}_{\text{opt}}|}{|\boldsymbol{\mathcal{C}}_{\text{ref}}|}$$

Here, $\boldsymbol{\mathcal{C}}_{\text{ref}} \subset \boldsymbol{\mathcal{C}}$ is the set of service candidates resulting from the refinement of $\mathcal{C}$, and $\boldsymbol{\mathcal{C}}_{\text{opt}} \subseteq \boldsymbol{\mathcal{C}}_{\text{ref}}$ the set of *optimal* services. However, an upper bound would be to assume that only one service fits best the abstract component, i.e., $|\boldsymbol{\mathcal{C}}_{\text{opt}}| = 1$. The optimality criteria is subject to the requirements of the application and can vary accordingly.

Component Templates

The aggregation between the template class and the component class in Figure 2.5 implements the composite design pattern again. This second aggregation organizes components into predefined architectural templates. A template is a nested component containing multiple prearranged components. To exemplify this visually, Figure 2.7 shows a scenario using the three-tier cloud application template. It portrays the template as a black border containing a name and an image in the left corner. The template body contains the components that are typically required by the three-tier architecture: a stateless component for the front end, a message-passing middleware, a stateless component for the back end, followed by a stateful component to persist data. Templates are specific to cloud providers. For example, two cloud providers might group different components or use perhaps fewer or more components to implement a three-tier cloud application. The advantage of using templates as a modeling device is two-fold. The cloud provider can express best practices for achieving specific architectural designs, whereas the developer can use one component to insert a series of components into a scenario. Developers can rely on the template containing the components considered best practice by the cloud provider.

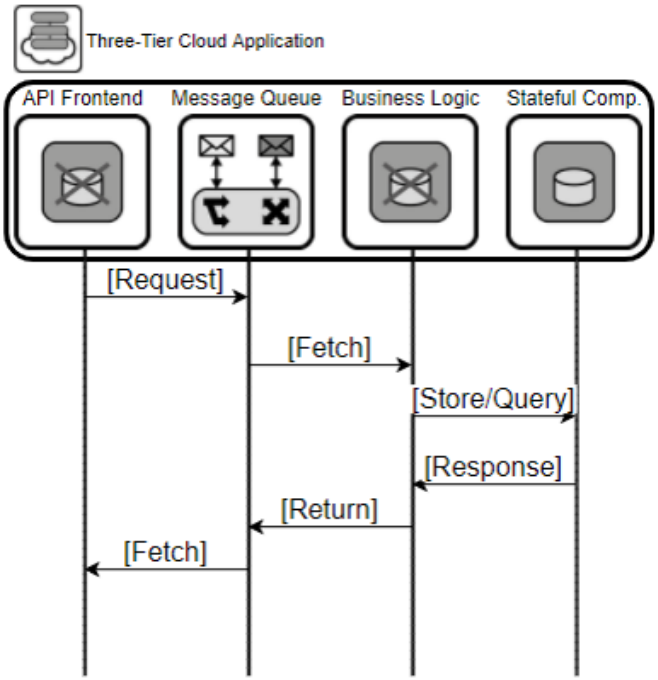


Figure 2.7.: The three-tier architecture pattern is a template consisting of four services.

Cost Model

As shown in the component meta-model in Figure 2.5, the cost of a service depends on the selected service plan, which in turn depends on the cloud region where the developer wishes to use the service. Hence, one needs to create a component for every possible service plan, i.e., configuration option, and assign the appropriate cost depending on the region. For example, consider the smallest VM instance type at Azure. The cost of the *A0 VM* is the product of multiple configuration options, such as its type (*Standard*, *Basic*, or *Low Priority*), and its operating systems (Windows or Linux). Depending on the selection of these properties, the cost varies significantly. In order to account for the actual cost considering these properties, one can create a component for each combination of the parameters and assign the corresponding cost accordingly. In this example, six cloud components represent the different configuration options: *VM A0 Basic Windows*, *VM A0 Low Priority Window*, etc., essentially computing the Cartesian product. The region property is not part of the Cartesian product since it is already part of the service component. So, the component contains the cost for each region where the service is offered.

Computing the Cartesian product for services with multiple configuration op-

tions can lead to an exponential blow-up of components. While the discretization of the configuration space leads to a large number of possible components, it reduces the complexity of the cost model in exchange.

There are two solutions to mitigate the exponential increase of components when discretizing the configuration space. Both solutions assume that a large portion of developers uses just a small set of possible standard configurations per service (long-tail distribution). The first solution is to use intelligent parameter sampling. Suppose a service offers the option to extend its resources by an unbounded parameter such as memory. In this case, creating separate components becomes infeasible, where each component just adds one more unit. Therefore, selecting a subset of standard or popular values, e.g., 1GB, 10GB, and 100GB for storage space, is sensible to address the most common cases. Moreover, approaches such as MICKY [87] might be useful in finding optimal configuration options for different user loads, reducing the set of possible configurations in general. The second solution is to ignore rarely changed parameters and set them to their default values. For example, consider the *A0 VM* example from above, an additional cost-factor is the support type (*Included*, *Developer*, *Standard*, *Professional*, and *Direct*), which might not be of interest in the early development stages.

2.5. Case Study

This section evaluates the feasibility of the Clams approach by introducing a case study on real-live architectural examples from the Azure cloud. This case study answers the general questions on how many components has a major cloud provider and how many services and scenarios are used on average in an application. It also discusses the performance to query refinement trees to collect all service plans that form the solution space.

2.5.1. Data Acquisition and Preparation

It is a challenging task to collect information on real-life architectures. Companies usually avoid disclosing their architecture to protect their intellectual property or have security concerns. However, the documentation page of the Azure Cloud provides a large and diverse set of architectural examples. These examples are a general starting point for developers when developing new cloud applications. They are publicly available and guarantee the reproducibility of the experiments.

Note that the presented concepts are not limited to Azure but also support other cloud providers. Unfortunately, other cloud providers did not provide such a data set at the time of this writing. Therefore, due to the extensive documentation of cloud architectures and an easy-to-access price calculator to extract service components, this work chooses Azure for this case study.

The documentation page of the Azure Cloud provides 385 architectural examples [88]. All architectural examples are unstructured text pages, describing the example in detail and providing, in many cases, a list of components like a recipe. As part of this research, this work used a web crawler to gather and extract the component data from the component list, resulting in 213 architectures with an identifiable component list. The average number (and median) is six components per architecture. The standard deviation is three services. The numbers only consider how many different services are present in the architecture. This does not include the number of instances of a particular service. However, most architectures have two instances of the same service at most. Moreover, some architectural examples also provide a *data flow description* list for user interactions, which we interpret as informal scenario descriptions. These high-level descriptions have insufficient information to extract the exact service interactions to build a scenario. Whenever an example has such a list, the crawler returns the list's cardinality as the number of scenarios for the architecture. Note that these data flow descriptions only include those scenarios important for the application's core features. Scenarios for standard features, such as changing user profiles or passwords, were not considered. The final result is a set of 31 architectural examples in which the component list had identifiable cloud services that only use services from the Azure product catalog. The excluded examples contained third-party components which could not be mapped to concrete service offerings. Appendix A provides a complete list of all architectures with their components.

A primary feature of Clams is its refinement capabilities which are only helpful when cloud components are available. To collect components, this work sampled the cloud service catalog of Azure by crawling its price calculator [89]. Each cloud service has different configuration options, ultimately influencing the final operational cost. The component extraction procedure of the crawler regards each service configuration as one component. For example, the Azure VM service has more than 400 instance types, each with three different tier types, resulting in about 1200 components for the VM service alone. The crawler only considered the four most common values for specific configuration parameters, which require

numerical values such as the expected number of event calls or device connections, e.g., 1, 10, 100, and 1000 for device connections. In total, the crawler extracted 18127 concrete service configurations with prices for four cloud regions. Each configuration represents a component, which is stored in one database table. The table has a self-referencing foreign key to implement the parent-child relationship for the component refinement. The resulting component database and refinement trees are publicly available as part of the tooling support, which will be discussed in the next section.

2.5.2. Architecture Analysis

The scope of this work is to predict the availability of a cloud application and to build a recommendation system that suggests an optimal set of concrete services. Considering that the optimization criteria is operational cost with regard to availability constraints, it can be stated with confidence that assessing the availability of just one refinement candidate results in a non-trivial computation time. Assuming that there is more than one refinement possibility, the second goal can become a challenging task. The refinement of an abstract architecture can result in a high number of valid service combinations. The next step is to analyze the expected solution size, essentially the number of different service combinations that form the potential refinement for a given abstract architecture. The finding should indicate whether an exhaustive search is feasible to find an optimal service combination or if an approximation approach is necessary later when designing and implementing the recommendation system. To enumerate the number of service combinations, it is sufficient to count the number of leaf nodes in the corresponding refinement tree of each abstract component and compute their product.

Table 2.1 provides a summary of the architectural examples, showing the number of components, the number of estimated scenarios, and the number of service combinations to refine the architecture, i.e., the solution space. The table is sorted according to the size of the solution space. The average number of services and scenarios is five, and the number of refinement solutions per architecture ranges from two (App 1) to over a quadrillion (App 31). The highest number of scenarios has App 16 with ten scenarios. Looking at the data, we see a positive correlation between the number of components and the solution space size. This is no surprise since more components lead to more service combinations. However, one factor, which is not shown in Table 2.1 for readability reasons, is the

Table 2.1.: Summary of the Azure case study.

Short	Architecture Example	# Components	# Scenarios	# Solutions
App 1	Speech Services	2	7	2
App 2	Unlock Legacy Data with Azure Stack	2	4	5
App 3	HPC System and Big Compute Solutions	2	6	131
App 4	SMB disaster recovery with Double-Take DR	3	1	180
App 5	Loan Credit Risk + Default Modeling	2	-	690
App 6	Predicting Length of Stay in Hospitals	2	-	690
App 7	Sharing location (...) low-cost serverless Azure services	3	-	8460
App 8	Retail and e-commerce using Cosmos DB	4	-	42000
App 9	DevTest Image Factory	3	6	76608
App 10	Tier Applications & Data for Analytics	3	7	188640
App 11	Archive on-premises data to cloud	2	1	220800
App 12	Container CI/CD using Jenkins and Kubernetes on AKS	5	9	340200
App 13	SMB disaster recovery with Azure Site Recovery	4	1	414000
App 14	Adding a mobile front-end to a legacy app	2	8	666729
App 15	Custom Data Sovereignty & Data Gravity Requirements	5	3	4.0×10^6
App 16	Design Review Powered by Mixed Reality	3	10	6.4×10^6
App 17	Demand Forecasting + Price Optimization	4	3	1.8×10^7
App 18	Defect prevention with predictive maintenance	5	-	2.0×10^7
App 19	Enterprise-scale disaster recovery	6	1	4.6×10^7
App 20	Hybrid ETL with Azure Data Factory	4	5	2.1×10^8
App 21	Discovery Hub with Cloud Scale Analytics	5	5	5.5×10^8
App 22	Modern Data Warehouse Architecture	6	4	7.4×10^8
App 23	Master Data Management powered by CluedIn	5	8	9.3×10^9
App 24	Advanced Analytics Architecture	7	6	7.4×10^{10}
App 25	Anomaly Detector Process	6	6	9.4×10^{11}
App 26	Personalized marketing solutions	8	-	1.1×10^{12}
App 27	Quality Assurance	7	8	1.1×10^{12}
App 28	Predictive Aircraft Engine Monitoring	7	-	2.3×10^{12}
App 29	Build web and mobile applications	7	-	2.2×10^{13}
App 30	Predictive Insights with Vehicle Telematics	8	-	2.7×10^{15}
App 31	Real Time Analytics on Big Data Architecture	8	7	2.7×10^{15}

number of concrete services per component refinement (c.f. Appendix A). Depending on the abstraction level of the components, the solution space can vary accordingly. Consider App 14 and 11 in contrast to App 1 and 2, which have only two components. The component list of App 1 and 2 has one concrete service and a low-level abstract component, whereas App 14 and 11 use two high-level abstract components. For example, App 14 uses the virtual machines pattern and the abstract SQL Database components. The virtual machines pattern already results in almost 1200 concrete service plans while the abstract SQL Database component results in 660 concrete services. As a result, two factors determine the size of the solution space. The first is the number of components, and the second is the abstraction level of the components.

Consequently, the data in Table 2.1 indicates that an exhaustive search for an optimal refinement is not feasible in general. When we search for a specific service combination, the recommendation algorithm inevitably has to compute the availability of the architecture for each service combination. As shown in the next chapters, computing the availability takes milliseconds to seconds, depending

on the architecture size. However, an average application with five components can already have a solution size of billions (App 23). Hence, an exhaustive search for an optimal refinement becomes infeasible for such examples. As a result, we need to adopt approximation and accept a near-to optimal solution, as long as we get a reasonable result within a feasible time frame. This conclusion is the main driver for designing and implementing a recommendation system based on approximation, which will be introduced in Chapter 6.

2.5.3. Performance Analysis

The preliminary step for the refinement process is to load the service candidates that match each abstract component. Hence, an important aspect in evaluating and refining abstract architectures is the performance in collecting all service candidates from a component database. Therefore, the final part of this analysis focuses on Clams and its ability to load concrete services. We need to recall that Clams uses a homogeneous cost model, which assigns one price to each service plan (configuration option) for a given cloud region. Consequently, in the worst case, the number of components can increase exponentially in the number of configuration options per service. For example, although the Azure cloud offers about 200 different services in its official product catalog [57], the crawler has collected 18127 concrete services due to the cost model. The high number of components can significantly influence the performance to load services for the refinement process.

As indicated in the data preparation section, all components are stored in one database table. This work uses a breadth-first search to collect the leaf nodes in the refinement tree. From a technical perspective, the schema of the component table has a parent attribute, which contains a reference key to the corresponding parent component to encode the refinement relation. In the following, we analyze the performance to query the database for the leaf nodes of the refinement tree. All experiments were performed on a 64-bit machine with 64 Intel(R) Xeon(R) CPU E7-4850 v4 at 2.10GHz and 1 TB of main memory, running Arch Linux 5.13.12 with GCC 11.1.0, and Python 3.9.6, using MySQL 8.0 to store the component table.

Figure 2.8 shows the mean execution time to query the leaf nodes for a subset of the architectures. For the sake of readability, the figure includes only those architectures where the solution space size is more than one thousand. What is interesting is that App 14 has the highest query time compared to all other

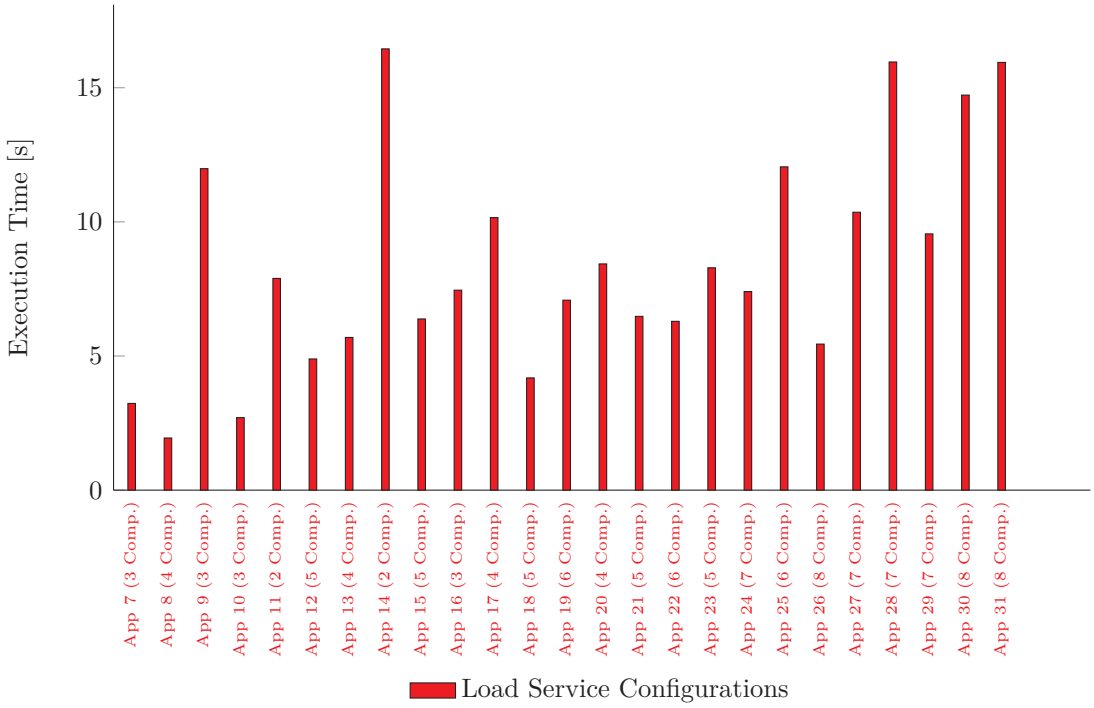


Figure 2.8.: Execution time for querying service refinements for different architectures.

architectures but has only two components. The second and third highest architectures are App 28 and App 31. One might have expected that only the number of components influences the query time, but apparently, a different factor plays an important role as well. App 9 and 14 use highly abstract components. This indicates that the traversal depth of the refinement tree influences the query time. For example, the average traversal depth for App 14 is six, whereas App 31 has an average traversal depth of about three, although it has eight components.

The correlation coefficient between query time and average tree depth is 0.56. This confirms our assumption that tree depth positively influences query time. However, the correlation coefficient between query time and the number of components is -0.33, which indicates an inverse relation. It appears that having more components leads to shorter query times. However, this should not be regarded as the general case. When we analyze the component lists of the larger architectures, we will see that they tend to use less abstract components. Hence the average tree depth is shorter compared to architectures with less components. For example, App 12 has five components, but the average traversal depth to load the service candidates is 1.4. As a result, we can conclude that the architectural examples from the Azure cloud tend to be more generic the smaller the

architecture and more specific the larger the architecture.

In summary, this case study shows that Clams can feasibly handle a large number of components from a major cloud provider. Thanks to the architectural examples of the Azure cloud, we were able to gather insides on the expected size of real-life examples and on the query time to prepare the solution space for the refinement process. Finally, the overall query time to load the service candidates depends on the number of components and the abstraction level of those components.

2.6. Tooling Support

Tooling support is essential for any cloud modeling language. A graphical modeling environment and a framework that eases custom tooling extensions for model evaluation and analysis are important to provide sustainable modeling and development experience. Hence, an additional contribution of Clams is the design and implementation of appropriate tooling to support model creation and management. The tooling implements the first part of the workflow shown in Figure 2.1 on the first page of this chapter.

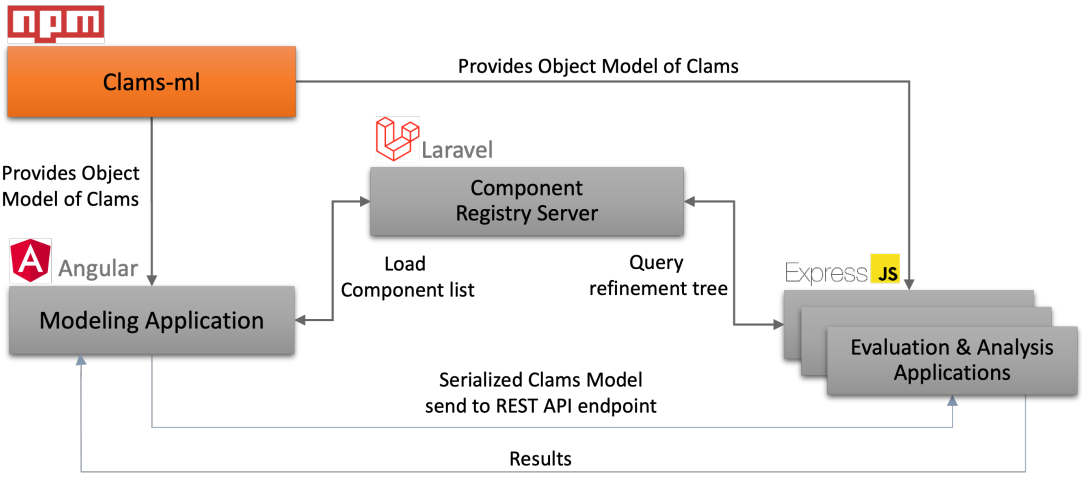


Figure 2.9.: OpenClams architecture with its tooling support.

The result is OpenClams[2], a collection of open-source tools to manage and interact with Clams models. OpenClams includes a graphical *web application* to create and edit models and a *component registry server* to store, manage, and publish cloud components to the web application or any other application that

[2]https://github.com/openclams

processes Clams models and needs to discover and refine components. Additionally, OpenClams provides a development library to build custom tools for further evaluation and analysis of models.

Figure 2.9 shows an overview of the OpenClams architecture. The architecture consists of three parts. These are the web application for graphical interactions, a component registry server, and a set of one or more custom evaluation and analysis tools. These evaluation and analysis tools are web services with a common REST API endpoint, so the web application can uniformly interface with them and process their results. In our case, the evaluation tools will be responsible for availability prediction and service recommendation. However, these custom services, as well as web applications, require knowledge of existing components. Therefore, a component registry server stores and publishes components centrally to provide a consistent view of the existing components. Furthermore, to ease the development of new tools, a Node.js package, called *clams-ml*, provides a common programming framework to serialize and deserialize Clams models when communicating with the web application.

The web application uses Angular[3] for basic user interactions and to render scenarios and usage profiles. Figure 2.10 shows the web application displaying a scenario consisting of four cloud computing patterns. The right side contains the main modeling pane for scenarios and usage profiles, whereas the left side depicts the list of cloud components. The web application loads the component catalog from the component registry server. This includes cloud computing patterns and abstract components representing available service offerings. If developers wish to use a concrete service plan, the web application offers a navigation dialog to manually traverse the refinement tree until the required service plan is found.

The main advantage of the architecture in Figure 2.9 is that evaluation and analysis applications are loosely coupled with the modeling interface. The web application contains a configuration menu where developers can register their extensions. Any custom tooling needs to implement *clams-ml* and expose a REST API endpoint to receive models. The developer can then register the API URL in the web application. Multiple endpoints can be registered, allowing for multiple extensions simultaneously. The web application expects as a result a summary of the evaluation and optionally a list of concrete service components to replace the abstract components. Figure 2.11 shows the web application displaying the result of a recommendation service with its suggestions. For instance, the recommendation service suggests to replace the key-value storage pattern with a

[3]https://angular.io/

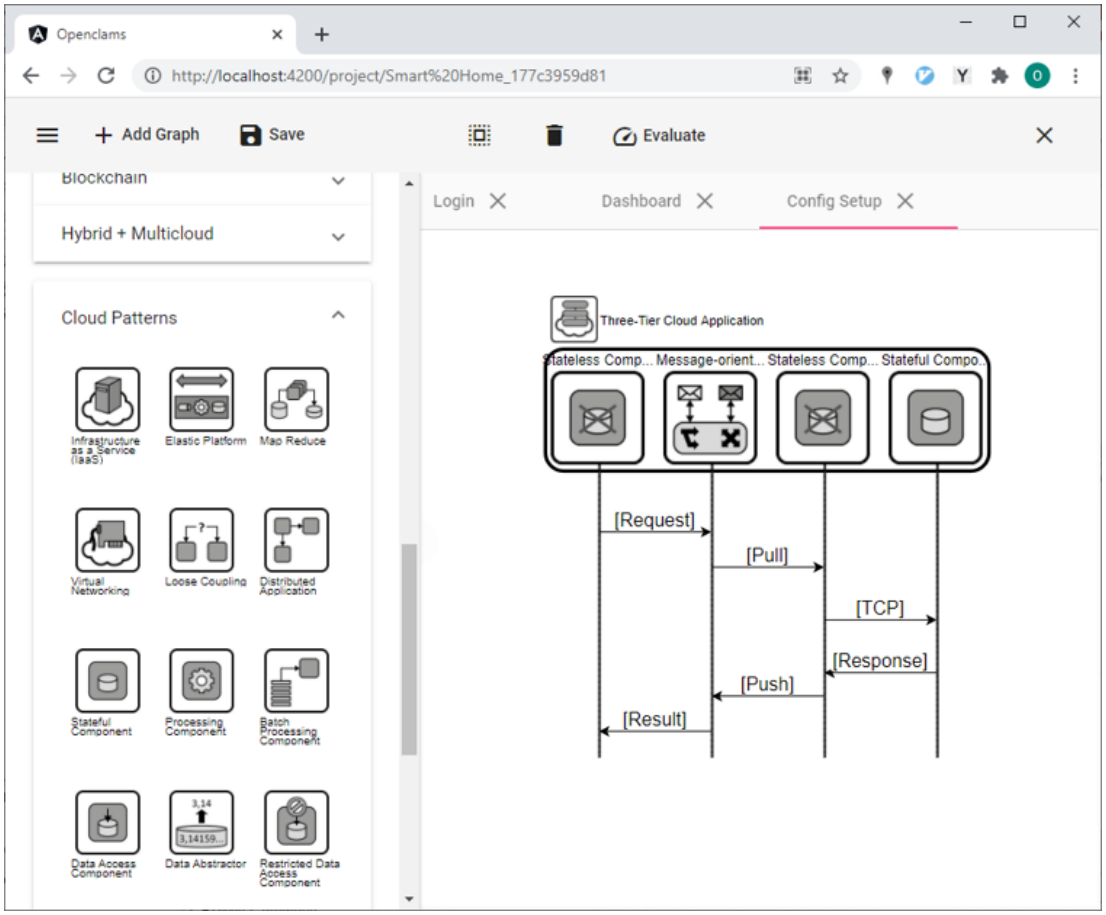


Figure 2.10.: The web application supports the viewing and editing of Clams models.

Cosmos DB Gen 5 service. Due to the loose coupling of the web and the evaluation applications, a cloud provider could host its evaluation application that recommends optimal cloud services with regard to cost and availability. The evaluation application would then use (internal) availability information of the cloud provider, which is otherwise not accessible by a developer, to enrich the submitted model and to assess its availability. As a result, it only returns a list of service candidates without having to expose any critical internal metrics.

The component registry server is a database that contains a list of all available cloud components. It provides a user interface to create, edit, and delete components. It is written with the Laravel Framework[4] and contains a design tool to organize components into refinement trees. For example, Figure 2.12 shows an excerpt of the component registry server, where one can organize patterns, services, and other abstract cloud components within nested lists to express the

[4]https://laravel.com

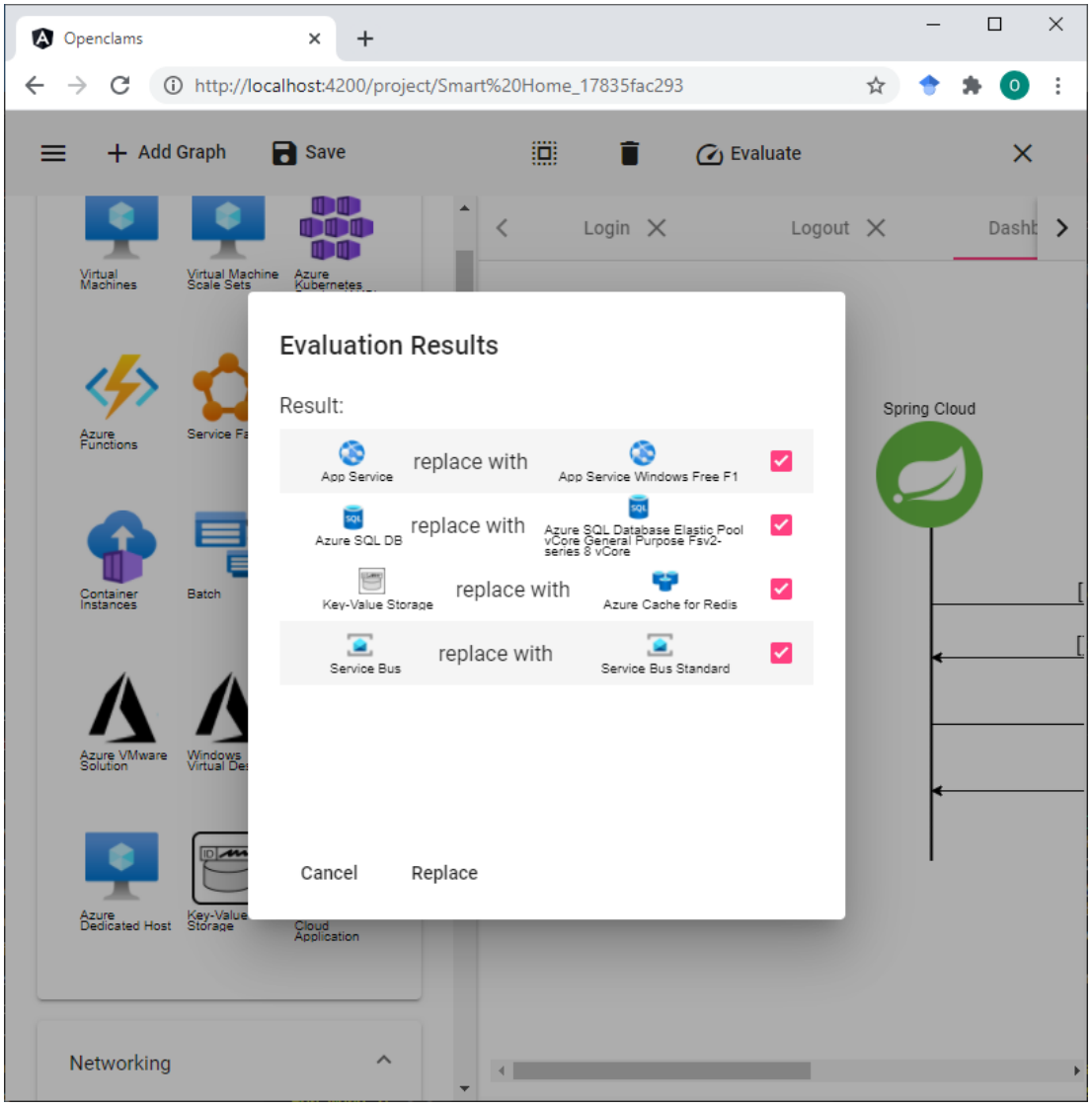


Figure 2.11.: The component replacement dialog from the web application.

refinement relation efficiently. With the help of the design tool, which is part of the component registry server, the author could include cloud computing patterns as abstract components and curate the refinement trees within a feasible time frame.

Any developer can curate the component registry server, but it can also be hosted as a service by a cloud provider or the IT department of a company. Cloud providers can offer the component registry server as a service that publishes their components and patterns representing best practices for their cloud environments. This has two advantages. The cloud provider can indirectly influence the quality of architectures depending on what components they publish. At the same time, developers can use these components to design architectures

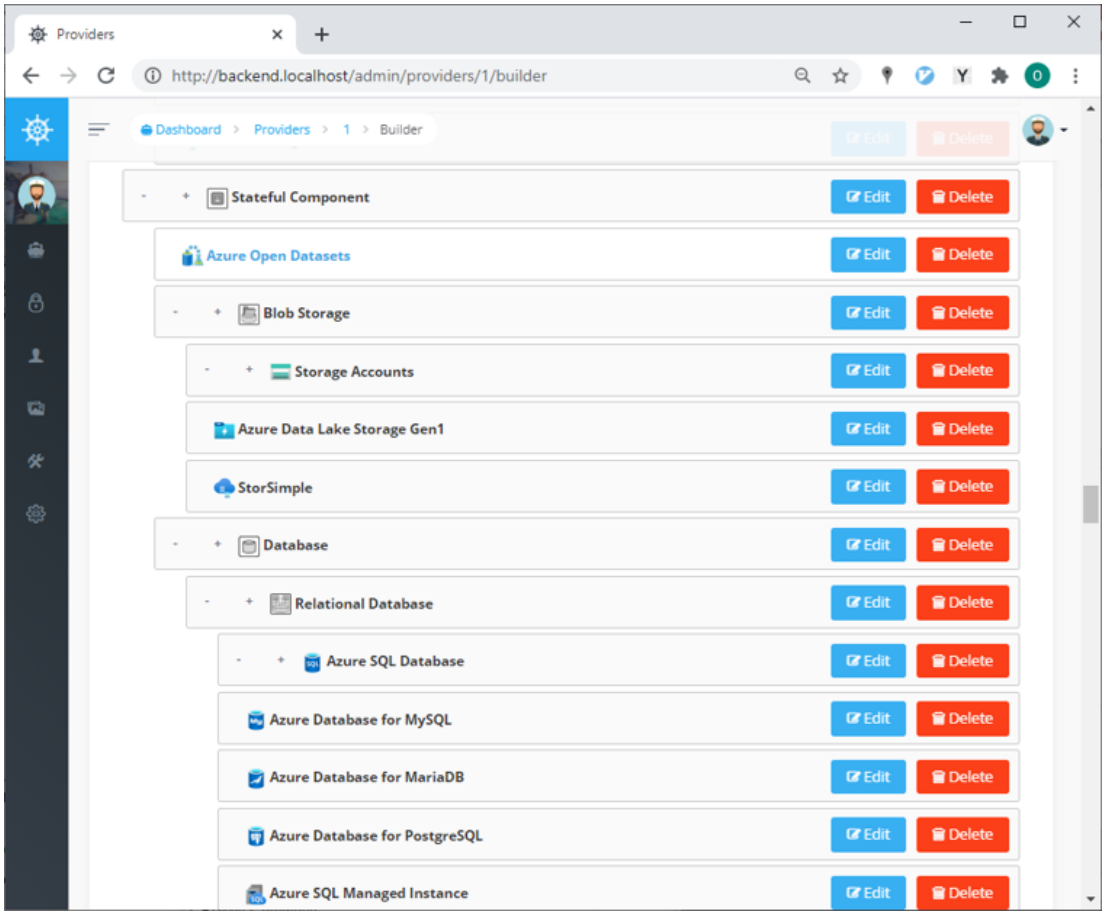


Figure 2.12.: User interface to manage component refinements.

that meet the standards of the cloud provider without needing ample knowledge of the existing services. Within an enterprise company, the IT department can host their own on-premise component registry server and only make those cloud components accessible, which comply with the company's development and security policies. The company is free to include custom patterns that are specialized in their field of business or limit the possible set of available cloud components accordingly.

In summary, developers model their architecture with the help of the web application. At the same time, cloud providers can offer a custom evaluation service that can (a) predict the availability based on the architecture defined in Clams, and (b) suggest optimal services if required. To manage this choreography, OpenClams offers the *clams-ml* package that handles the necessary boilerplate code to read Clams models and to return results and the service suggested in an appropriate format.

2.7. Related Work

Recent developments in cloud application modeling have increased the need for technology-agnostic modeling tools to enable developers to design applications without the explicit knowledge of the vast landscape of cloud service offerings [90–92]. One solution among many is the MDA approach, which introduces the notion of platform-independent modeling [74]. In MDA, platform-independent models get refined into platform-dependent models. Several component-based architecture modeling solutions have adopted this idea by introducing the notion of abstract components, which get replaced by components that represent concrete solutions [75–77]. Clams follows this notion and provides a MDA approach, which focuses on cloud application modeling.

Bergmayr et al. [47] provided an in-depth survey on cloud modeling languages. Based on their proposed cloud modeling language taxonomies, CAML [48, 49], Blueprint [50, 51], GENTL [52], and Tosca [53, 54] have the most similarities to Clams with regard to their component model. CAML and Blueprint do not consider component abstractions. They expect that developers already know which services they use. Moreover, current cloud modeling languages focus on deployment, orchestration, or migration of cloud applications. Therefore, a scenario-based cloud modeling language like Clams would complement existing cloud modeling languages by providing developers with a user-oriented modeling perspective to guide architectural design decisions in the early development stages.

Fraj et al. [93] suggest a model-driven approach for service orchestration in workflows, where they describe functional views of a workflow using the Business Process Model and Notation (BPMN) notation and the behavioral view with UML statechart diagrams. Their primary focus is cloud service orchestration and flexible service adaptation according to dynamic changes in the target cloud. Their use of BPMN provides an alternative modeling concept similar to the scenario approach of Clams.

A large and growing body of literature has investigated patterns and pattern refinement in software architecture and service composition [42, 82, 94, 95]. So far, current research uses architectural refinements for pattern-based deployment models [77, 90, 91]. Specifically, Harzenetter et al. [90, 96] use Cloud Computing Patterns [64] and Enterprise Integration Patterns [81] as abstract concepts to express agnostic deployment models. They introduce *pattern refinement models*, which are mappings from patterns to concrete deployment plans. Like existing cloud modeling languages, Herzenteer et al. regard a cloud application as a graph

of connected services [97], thus they do not distinguish between application-level functionalities within the architecture.

2.8. Summary

Cloud computing continues to have a significant impact on the development of modern software systems. As a result, researchers have introduced cloud modeling languages to support developers in their endeavor to design high-quality applications. However, current cloud modeling languages focus mostly on application deployment and orchestration. A modeling language that resonates on early design stages while guiding architectural design decisions is missing. Therefore, this chapter introduced Clams, a scenario-based cloud modeling language, that models application-level functionalities as scenarios, using cloud computing patterns as building blocks to describe cloud architectures abstractly. Finally, a case study analysis with 31 architectural examples from the Azure cloud showed that Clams could feasibly handle the size of real-live architectural examples and manage a large set of components of a major cloud provider.

The next chapter shows how to translate a Clams model into its appropriate availability model.

Chapter 3.

Hierarchical Availability Model

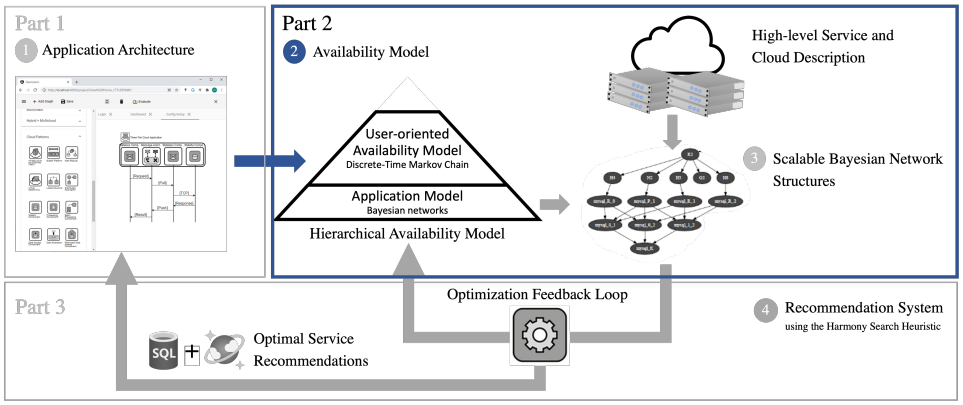


Figure 3.1.: This chapter focuses on part two of the solution workflow.

This chapter introduces a hierarchical availability model to compute the availability of a cloud application modeled in Clams. It initiates the second part of the solution workflow as indicated in Figure 3.1, where we take a Clams model as input for further processing. We focus on designing and implementing an availability prediction mechanism. In particular, this chapter introduces a hierarchical availability model and discusses only the root model of the hierarchy in detail. The model of the lower layer will be the topic of the next chapter.

Section 3.1 provides an overview of the proposed availability model. Sections 3.2 discusses usage profiles that exhibit different degrees of structural complexities that influence the availability prediction. Afterwards, Section 3.3 discuss the availability model, while Section 3.4 provides a sensitivity analysis of the model concerning generic Clams models. Section 3.5 introduces related work. Finally, Section 3.6 summarizes this chapter.

3.1. Introduction

The previous chapter introduced Clams, a modeling language to describe cloud architectures with the help of scenarios and usage profiles. This chapter introduces a hierarchical availability model [19] to compute the availability of a usage profile, which we refer to as the user-oriented availability model. A usage profile is available when a user can execute the final scenario. Hence, the availability of a usage profile is the likelihood of reaching and executing a distinguished end-scenario.

The proposed availability model is a hierarchical composition of two models. The root model uses the well-known *user-oriented software reliability model* introduced by R. C. Cheung [65] to account for the user behavior. In contrast, the lower level uses a Bayesian network availability model, discussed in Chapter 4, to compute the availability for the individual scenarios in the usage profile. Rodrigues et al. [79] also used the user-oriented software reliability model as a mathematical formalism to compute the reliability of usage profiles. They first translate their scenarios and usage profile into a labeled transition system, which they then translated into the user-oriented software reliability model for further assessments. Without going into too much detail, their labeled transition system approach assumes fault-independent services, whereas we want to account for common and cascading failures, and service replication. A labeled transition system is a process algebra that does not support these modeling requirements. However, this work adapts the idea of the user-oriented software reliability model as part of a hierarchical availability model.

Figure 3.2 shows an overview of the hierarchical availability model. The model consists of five granularity levels, where each level includes additional fault aspects to the overall availability assessment. The bottom level considers failure modes of individual components. At this level, we model common-cause and cascading failures propagating through a system. For example, if a data center catches fire, we are confident that all racks and hosts are also failing. The next level builds upon the first level and includes the availability model of individual service instances, computing the likelihood of observing an instance as up and reachable, introducing failure aspects caused by the network. The third level considers service replication. Here, the level accounts for network partition failures to address the likelihood that replicas cannot agree anymore upon user requests. The next level computes the availability of a scenario as a function of the previously modeled services. This level includes additional aspects of communication failures

between the services. Finally, at the top level, we translate the usage profile into the user-oriented software reliability model using availability values, that were computed at the previous level. The first levels will be handled by a Bayesian network availability model, which we will introduce next chapter in detail. This chapter focuses on level five, assuming the input parameters are given for now.

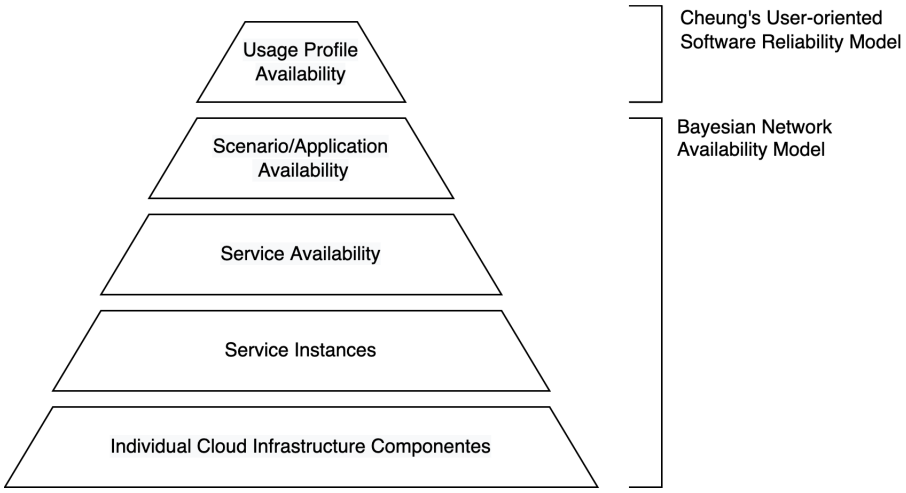


Figure 3.2.: Overview of the hierarchical availability model.

While the user-oriented software reliability model is already known in the literature, this chapter will briefly introduce and show how to translate a given usage profile into this model. However, an essential aspect of this work is prediction performance since the goal is to build a service recommendation system that considers availability constraints. Hence, computing the availability will be the most time-consuming activity in the search process for optimal services. An contribution of this chapter is a sensitivity analysis to understand which modeling parameters influence the performance of the hierarchical model in the end. There are two dimensions to this analysis. The first dimension considers the structural complexity of the usage profile, where we analyze two structural examples, whereas the second dimension considers the availability parameters. We first start by introducing the usage profiles, which will be the subject of this analysis

3.2. Serial and Fully Connected Usage Profiles

To compute the availability of a cloud application from the user's perspective, this work introduced Clams, which uses the notion of usage profiles to model the behavior of a user. Since usage profiles are graph models, they can vary in structural complexity. To understand how usage profiles influence the performance of

computing the availability, we will consider two families of usage profiles, which form two extremes concerning structural complexity, so that every other usage profile is within the spectrum of these two graph families.

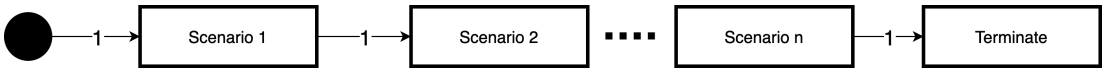


Figure 3.3.: A serial usage profile that defines a fixed order of scenario execution.

The first family of usage profiles represents the basic assumptions that users invoke scenarios in a deterministic order. Such usage profiles are serial graphs of $n + 1$ scenarios as shown in Figure 3.3. It starts with the first scenario and ends with a distinguished termination scenario. Each successor scenario will be invoked with probability of one until the termination state is reached. For example, such usage profiles might be used in the domain of IoT, where sensors frequently send state updates to a back-end cloud application, exhibiting a deterministic invocation behavior at the back-end's API.

The second family of usage profiles are fully connected graph models (complete graphs), representing a user with a random behavioral pattern. Figure 3.4 depicts the graph containing n scenarios and one termination scenario as the absorbing state. Each node has a self-referencing loop and an outgoing edge to the remaining $n - 1$ nodes. Due to the self-referencing, the transition probability from one scenario to the next is $\frac{1}{n}$. The only exception is scenario n, which has an additional state transition to the termination scenario. Consequently, its transition probability is $\frac{1}{n+1}$ for every edge. The termination scenario has to be an absorbing state for the user to terminate its random session eventually; otherwise, we cannot identify if the profile has ended successfully. Finally, the independent variable is the number of scenarios in both graphs.

3.3. User-oriented Availability Model

This section introduces the central concept of computing the availability of a cloud application from a user's perspective, namely the probability of successfully executing a usage profile by reaching a distinguished termination state. This work uses the user-oriented software reliability model proposed by Cheung [65] as a root model for the hierarchical model to address the problem of computing the availability of a usage profile as the function of the availability values of its scenarios.

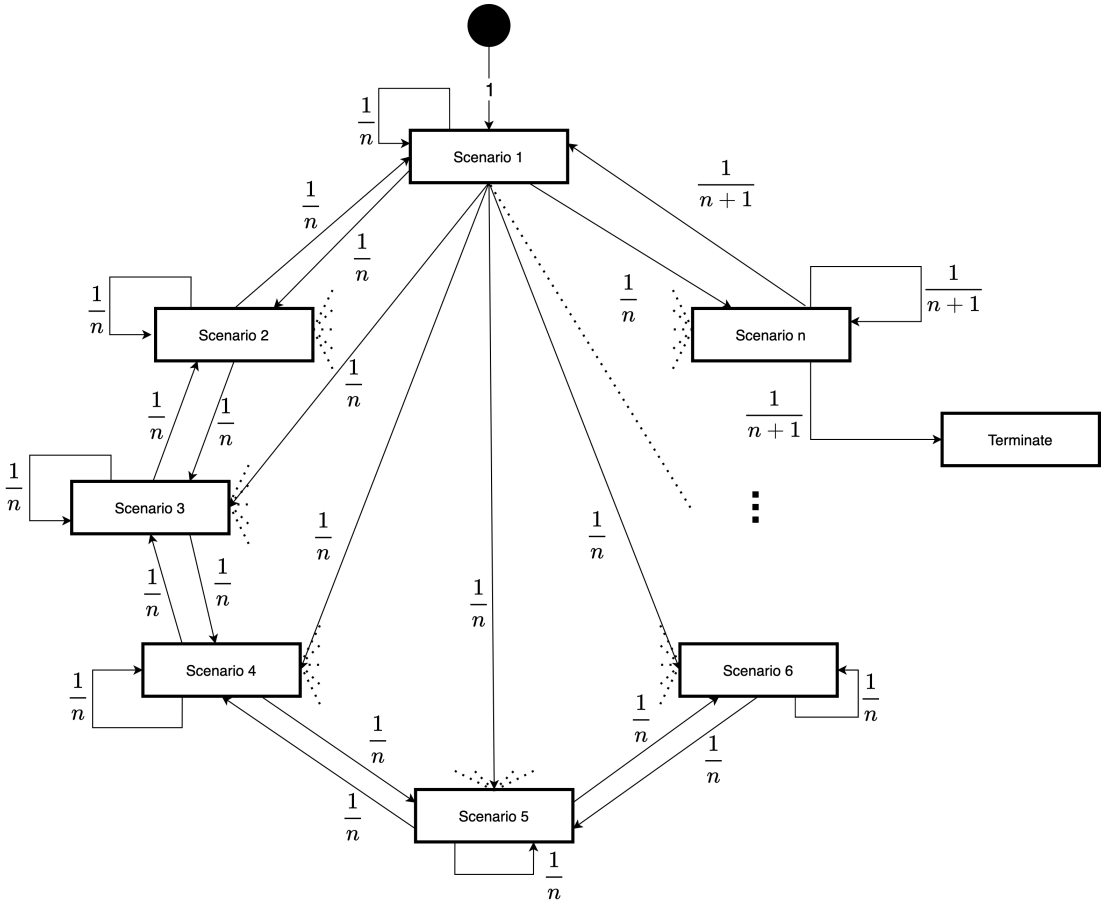


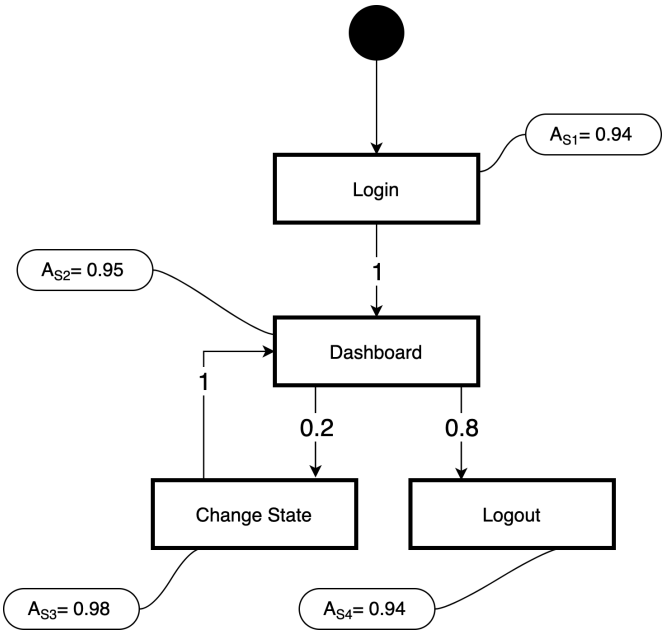
Figure 3.4.: An usage profile that defines a random execution of scenarios with eventual termination.

The user-oriented software reliability model is originally meant for module-based software systems, where a user's input invokes different sequences of software modules. The model assumes that user behavior is stochastic, modeling the transfer of control as a Markov process between modules. Hence, the reliability of a module is independent of the transition probability between modules. Therefore, if a module fails, it does not necessarily lead to a system failure if the user invokes a sequence of modules that do not require the former. In this work, usage profiles have scenarios as nodes. Cheung's model implies two modeling assumptions. First, it assumes that the availability of a scenario is independent of others. Hence, even though some scenarios might have components in common, the model does not account that if a common component fails, all other dependent scenarios also fail. This is an approximation since the usage profile does not indicate the time between transitioning from one scenario to another. The longer the waiting period between scenario invocations, the higher the chances

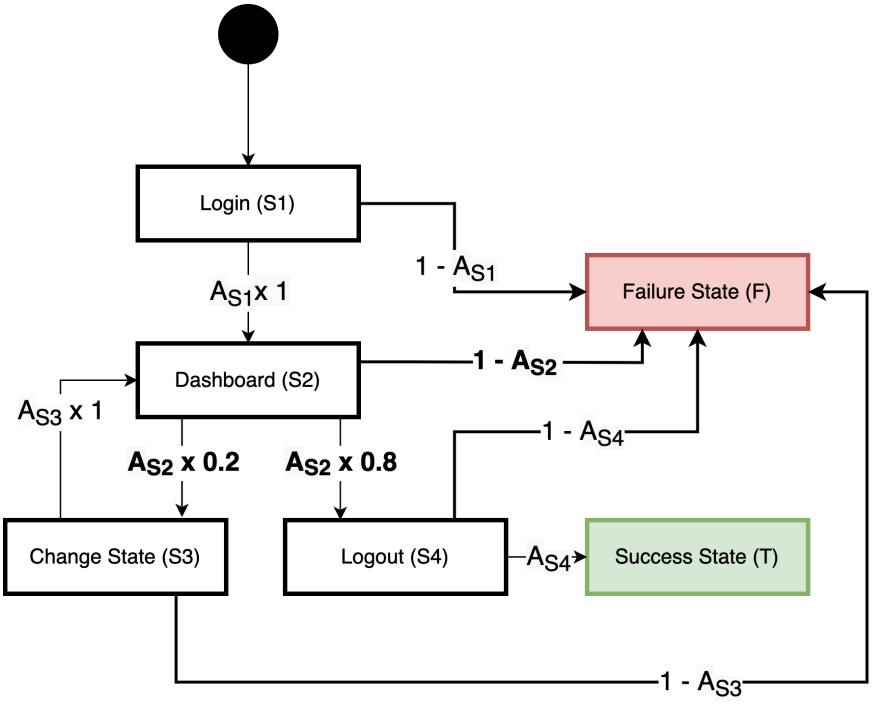
that a failed component gets repaired during the waiting period, so that in the end, the availability of a scenario, i.e., the likelihood to observe its services as up and reachable when requested, is independent of other scenarios. For example, users that consume web content typically exhibit such behavior where they take their time to interact with the content, e.g., reading a website or streaming music, before issuing a new request for new content to the cloud application. The second assumption of Cheung's model presumes that transferring from one scenario to the next is a Markov process. Hence, a user decides the next scenario depending on the current scenario and not based on the history of previously invoked scenarios. This is again an approximation based on the idea that the web uses mainly HTTP for content delivery or to consume REST APIs, where navigation is done by hyperlinks that usually do not take into account the history of previous requests.

Let us assume an usage profile has n scenarios, $\mathcal{S}_1$ to $\mathcal{S}_n$, with corresponding availability values $A_{\mathcal{S}_1}$ to $A_{\mathcal{S}_n}$. For instance, Figure 3.5a shows the usage profile from the example in Section 2.2 with its corresponding availability annotations. In the following, nodes are referred to as *states* to emphasize the availability state of a scenario. The next step is to extend the usage profile with an additional failure (F) and success (T) state and to include a new edge from every other state to the failure state with the transition probability $1 - A_{\mathcal{S}_i}$, respectively. Those final states that represent a successful execution have an additional edge to the success state with a probability of $A_{\mathcal{S}_j}$. In order to normalize the transition probabilities, the intermediate transitions need to be scaled accordingly to their availability values of the source state. For example, Figure 3.5b shows the corresponding transformation for the usage profile from Figure 3.5a. For node $\mathcal{S}_2$, the transition probabilities of the outgoing edges need to be scaled by $A_{\mathcal{S}2}$ due to the additional edge leading to the failure state. The resulting augmented Markov chain has the following general adjacency matrix.

$$\boldsymbol{M} = \begin{array}{c} \\ T \\ F \\ \mathcal{S}_1 \\ \mathcal{S}_2 \\ \vdots \\ \mathcal{S}_n \end{array} \begin{array}{c} \begin{array}{cccccc} T & F & \mathcal{S}_1 & \mathcal{S}_2 & \cdots & \mathcal{S}_n \end{array} \\ \begin{pmatrix} 1 & 0 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 - A_{\mathcal{S}_1} & 0 & A_{\mathcal{S}_1} \times P_{\mathcal{S}_1,\mathcal{S}_2} & \cdots & A_{\mathcal{S}_1} \times P_{\mathcal{S}_1,\mathcal{S}_n} \\ 0 & 1 - A_{\mathcal{S}_2} & 0 & A_{\mathcal{S}_2} \times P_{\mathcal{S}_2,\mathcal{S}_2} & \cdots & A_{\mathcal{S}_2} \times P_{\mathcal{S}_2,\mathcal{S}_n} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ A_{\mathcal{S}_n} & 1 & 0 & 0 & \cdots & 0 \end{pmatrix} \end{array}$$



(a) Annotated usage profile.



(b) Augmented graph model with availability values.

Figure 3.5.: Usage profile transformation.

Assume $\mathcal{S}_1$ is the initial state and $\mathcal{S}_n$ is the final state. The goal is to determine the probability of reaching the success state. According to the user-oriented software reliability model, to compute the steady-state availability we can use the transition matrix with:

$$\boldsymbol{Q} = \boldsymbol{M}_{2:n,2:n}$$

Here, $\boldsymbol{Q}$ is the result of excluding the rows and columns of the T and F states. To compute the steady-state probabilities, one needs to compute the complement.

$$\boldsymbol{P} = (\mathbf{1} - \boldsymbol{Q})^{-1}$$

Finally, the scalar $P_{1,n}$ represents the probability to reach state $\mathcal{S}_n$ from $\mathcal{S}_1$. Hence, the steady-state availability is the product of reaching $\mathcal{S}_n$ times its availability $A_{\mathcal{S}_n}$, leading to the final result:

$$A = \boldsymbol{P}_{1,n} \times A_{\mathcal{S}_n}$$

For example, the following adjacency matrix represents the transformed graph from Figure 3.5b.

$$\boldsymbol{M} = \begin{array}{c} \\ T \\ F \\ \mathcal{S}_1 \\ \mathcal{S}_2 \\ \mathcal{S}_3 \\ \mathcal{S}_4 \end{array} \begin{array}{c} \begin{array}{cccccc} T & F & \mathcal{S}_1 & \mathcal{S}_2 & \mathcal{S}_3 & \mathcal{S}_4 \end{array} \\ \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 - A_{\mathcal{S}_1} & 0 & A_{\mathcal{S}_1} & 0 & 0 \\ 0 & 1 - A_{\mathcal{S}_2} & 0 & 0 & A_{\mathcal{S}_2} \times 0.2 & A_{\mathcal{S}_2} \times 0.8 \\ 0 & 1 - A_{\mathcal{S}_3} & 0 & A_{\mathcal{S}_3} & 0 & 0 \\ A_{\mathcal{S}_4} & 1 & 0 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

Next, consider the sub matrix $\boldsymbol{Q}$ with:

$$\boldsymbol{Q} = \begin{array}{c} \\ \mathcal{S}_1 \\ \mathcal{S}_2 \\ \mathcal{S}_3 \\ \mathcal{S}_4 \end{array} \begin{array}{c} \begin{array}{cccc} \mathcal{S}_1 & \mathcal{S}_2 & \mathcal{S}_3 & \mathcal{S}_4 \end{array} \\ \begin{pmatrix} 0 & A_{\mathcal{S}_1} & 0 & 0 \\ 0 & 0 & A_{\mathcal{S}_2} \times 0.2 & A_{\mathcal{S}_2} \times 0.8 \\ 0 & A_{\mathcal{S}_3} & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \end{array} = \begin{pmatrix} 0 & 0.94 & 0 & 0 \\ 0 & 0 & 0.19 & 0.76 \\ 0 & 0.98 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

Finally, the model computes the steady-state probabilities $\boldsymbol{P}$ for all states:

$$\boldsymbol{P} = (\mathbf{1} - \boldsymbol{Q})^{-1} \begin{pmatrix} 1 & 1.155 & 0.219 & \mathbf{0.877} \\ 0 & 1.228 & 0.233 & 0.933 \\ 0 & 1.204 & 1.228 & 0.915 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

The steady-state availability is the probability of successfully reaching and executing the last state of the usage profile $\mathcal{U}$.

$$A_{\mathcal{U}} = \boldsymbol{P}_{1,4} \times A_{\mathcal{S}_4} = 0.877 \times 0.94 = 0.824$$

3.4. Sensitivity Analysis

Next, we analyze the influencing factors with regard to performance and results. In brief, all experiments were performed on a 64-bit machine with 64 Intel(R) Xeon(R) CPU E7-4850 v4 at 2.10GHz and 1 TB of main memory, running Arch Linux 5.13.12 with GCC 11.1.0, Python 3.9.6, and Numpy 1.20.3.

The first set of evaluations focuses on the influence of input parameters, which are the resulting availability values of the scenarios. For now, we assume the values are given. However, in the follow-up chapter, this work introduces the Bayesian network model to compute these values. For the analysis, we will use the previously introduced serial and fully-connected usage profiles with two levels of availability characteristics. The first availability level represents scenarios with availability values sampled from a beta distribution, such that the average downtime is one hour per year. That is, every scenario $\mathcal{S}_i$ has a value $A_{\mathcal{S}_i} \sim \text{Beta}(8760, 1)$. The second availability level considers scenarios with an average downtime of ten hours per year, i.e., $\mathcal{S}_i$ has $A_{\mathcal{S}_i} \sim \text{Beta}(8760, 10)$. We expect to observe a significant availability difference between the profiles that use higher availability values compared to those with lower availability values.

Figure 3.6 shows the availability results for the serial usage profile for scenarios with one (blue plot) and ten hours (red plot) average downtime per year. With increasing scenarios, the serial usage profile exhibits a linear decrease in availability. Clearly, the longer the chain in the profile, the higher the risk that a scenario will fail during its execution before reaching the final state. This is, in particular, true since we can argue that the serial usage profile resembles a serial reliability

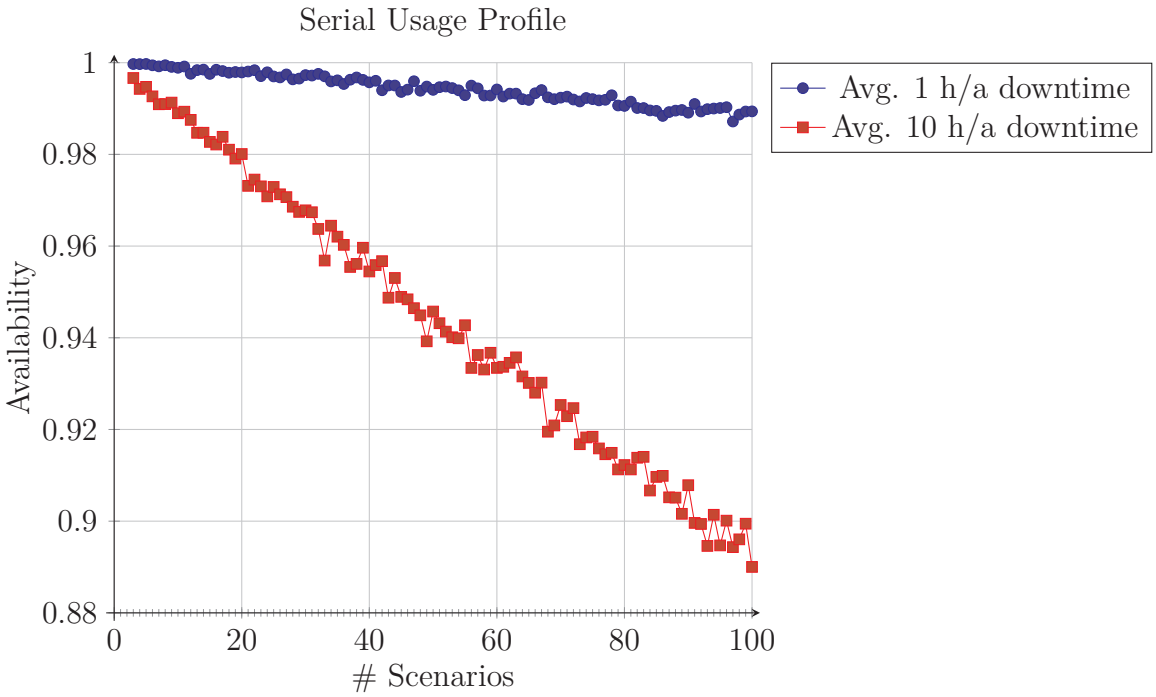


Figure 3.6.: Availability results for the increasing number of scenarios for the serial usage profile.

block diagram, where all blocks (scenarios) need to work for the usage profile to be available, i.e., executed successfully. Moreover, as expected, the availability drops faster for usage profiles with a lower availability level.

Next, Figure 3.7 shows the availability results for the fully-connected usage profile. With an increasing number of scenarios, the usage profiles with lower availability values drop faster than the equally sized serial profiles. The fully-connected usage profile resembles a user who randomly calls scenarios. The random calls lead, on average, to a higher number of scenario executions than in the serial profile. Hence, it takes longer to reach the final state, increasing the likelihood of observing a failed scenario. The final availability is, therefore, significantly lower for the fully connected usage profile than for the serial profile.

At the beginning of the plot, availability drops fast for profiles with less than 60 scenarios. Afterward, the slope gradually flattens. This flattening can be argued as follows. The shortest path to the final stage in the fully connected usage profile has three scenarios, independent of the number of scenarios. As a result, the availability begins to slowly converge for a larger number of scenarios. So, based on the analysis of the serial and fully-connected usage profiles, we can conclude that the input parameters, as well as the structural complexity, influences the outcome.

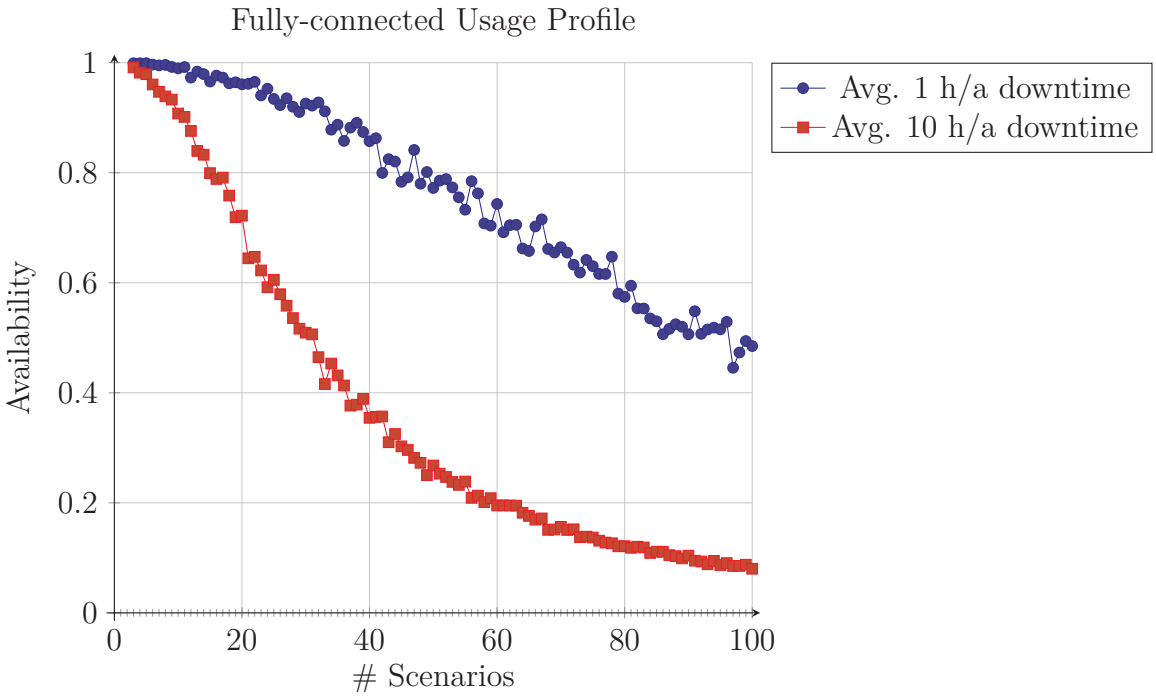


Figure 3.7.: Availability results for the increasing number of scenarios for the fully-connected usage profile.

Next, we analyze how input parameters and structural complexity influence the performance to compute the final availability outcome. Figure 3.8 shows the time to compute the availability for the fully-connected usage profile. The plot shows that the computation time increases polynomially with an increasing number of scenarios. All experiments were repeated 40 times to compute their 95% confidence intervals. Thus, it can be stated with 95% confidence that there is no significant difference between equally sized profiles for both availability levels. The fully connected usage profile generally consists of a dense transition matrix. The availability values only change the values in the matrix and have no influence on the actual computation time.

The largest usage profile has 100 scenarios, one order of magnitude larger than the scenarios in the Azure case study. It took about 1ms to compute the availability for the largest usage profile, which can be considered low. Hence, concerning the proposed hierarchical model, the user-oriented software reliability model can be excluded as the limiting factor in the overall prediction time. Computing the availability of a single scenario will have a higher contribution to the overall computation time in the end.

The serial usage profile has no time plot due to its trivial computation. Since the serial usage profile resembles a serial reliability block diagram, we can express

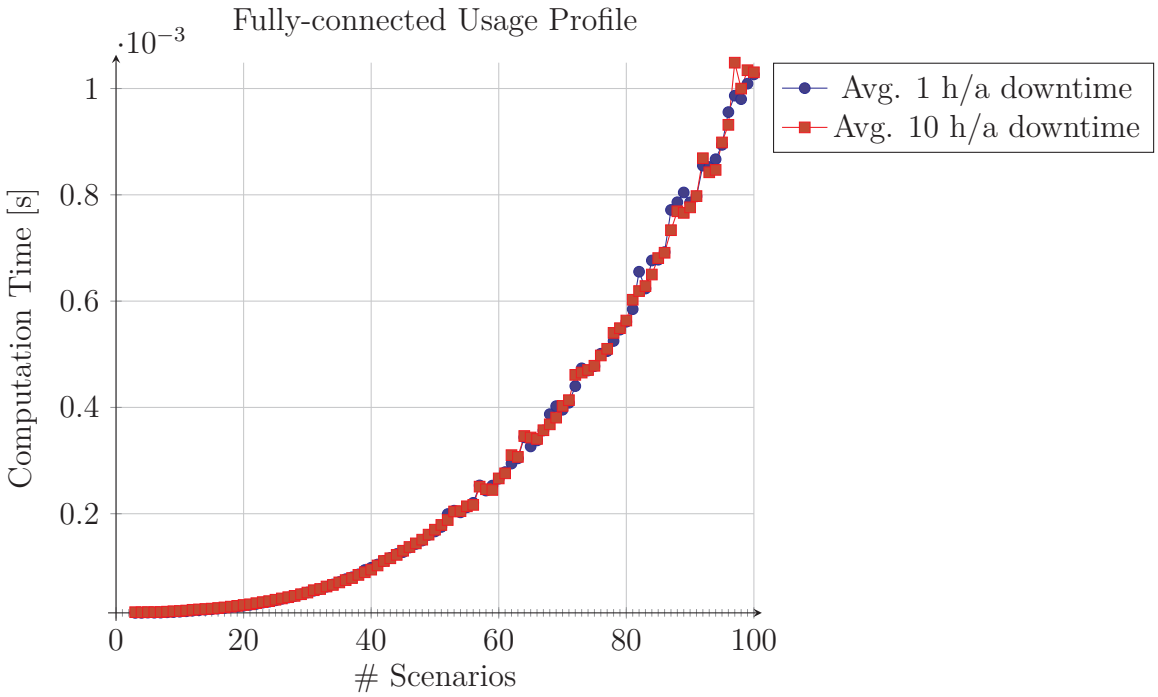


Figure 3.8.: Computation time for fully-connected usage profiles.

it in a closed form as the product of all its availability values.

In summary, the sensitivity analysis showed that the outcome is influenced by the input parameters and the structural complexity, whereas structural complexity only influences the performance of the assessment.

3.5. Related Work

Hierarchical compositions have proven to be a useful technique for computing the availability of complex systems [98–100]. Availability and reliability engineering distinguishes between non-state-space, such as fault trees [12] and reliability block diagrams [101], and state-space modeling approaches, such as continuous-time Markov chains and stochastic rewards nets [19], where the latter is a more powerful modeling formalism than the former [102,103]. Nevertheless, both modeling approaches have limitations. Multi-level models such as hierarchical compositions have the potential to overcome these limitations, as shown in [19].

For example, Ghosh et al. [22] introduced a multi-level model for assessing VM pools in IaaS clouds, using stochastic rewards nets to model individual subcomponents and an *import graph* as a root model to represent the pools. They showed a significant performance difference between the multi-level and mono-

lithic models of the same system, with the former being significantly faster than the latter. Smith et al. [99] use a two-level hierarchical composition to assess the availability of IBM blade clusters. They use as root model a non-state-space model, namely fault trees, and continuous-time Markov chains (CTMCs) to account for the dynamic behavior of the individual sub-systems, which then act as input for the fault tree model. Similarly, Trivedi et al. [100] also use a composition of fault-tress and CTMCs to model availability of the session initiation protocol [104] used again in the context of IBM blade clusters.

As already stated, Cheung [65] has initially proposed the user-oriented software reliability model, using DTMC to model user behavior in module-based software systems. Dazhi and Kishor [98] extended and generalized the approach of Cheung, proposing a hierarchical composition that uses DTMC to account for the behavior of the user and CTMCs to model the availability of individual components in the system. Similarly, Rodrigues et al. [79] employed the scenario definitions by Utchitel et al. [80] to transform scenarios into a labeled transition system first and then applied Cheung's user-oriented reliability model to compute the reliability of the system.

3.6. Summary

This chapter introduced a hierarchical availability model to assess the availability of Clams models, using usage profiles as a foundation for a user-oriented availability perspective. In particular, this chapter focuses on the root model of the hierarchical composition, which uses the user-oriented software reliability model by Cheung [65]. We discussed how to translate a usage profile into Cheung's model and analyzed the main factors influencing the prediction outcome and the performance. The analysis showed that the model behaves as expected when varying the availability inputs or when increasing the profile size. Moreover, the analysis also showed that the prediction performance only depends on the structural complexity of the profile. Finally, evaluating large usage profiles is in the range of milliseconds when considering the root model alone. Hence, the root model will not be the main limiting factor in our availability prediction task when we consider the computation of the lover level model, which will be discussed next.

Chapter 4.

Availability of Cloud Services and Scenarios

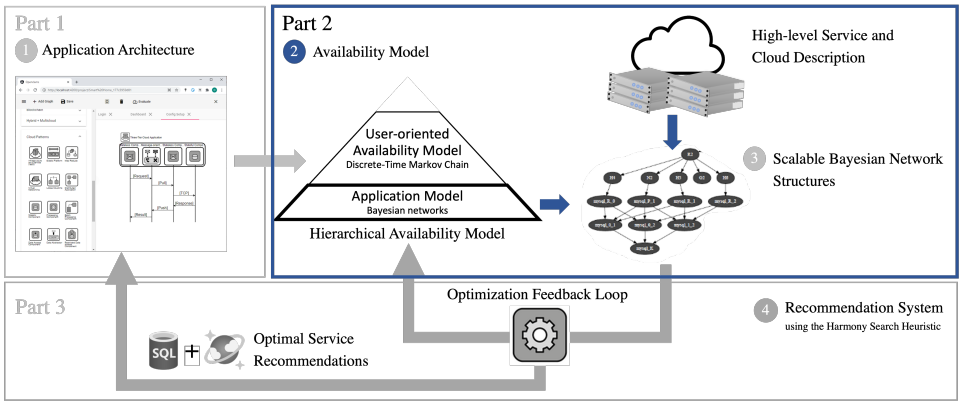


Figure 4.1.: This chapter continues with part two of the solution workflow.

This chapter introduces a Bayesian network availability model to predict the availability of scenarios. Similar to the previously introduced hierarchical availability model, this model also belongs to part two of the solution flow shown in Figure 4.1. Manually modeling complex systems with Bayesian networks can become error-prone and time-consuming. Hence, we first introduce a high-level modeling formalism to describe cloud applications and their corresponding services. Afterward, we design and implement an algorithmic approach to translate the high-level model into its corresponding Bayesian network. Evaluations will then verify if the Bayesian network model correctly implements our system descriptions and analyze prediction performance.

Section 4.1 introduces this chapter. Section 4.2 provides the system model. Next, Section 4.3 introduces a high-level model to describe cloud services and applications. Section 4.4 shows how to build a Bayesian network availability model based on the high-level model. Section 4.5 evaluates and discusses the model.

Afterwards, Section 4.6 discusses related work. Finally, Section 4.7 summarizes this chapter.

4.1. Introduction

The previous chapter introduced a hierarchical availability model, where the root model assesses the availability of a given usage profile, taking the availability of scenarios as input. Consequently, in this chapter, we discuss how to model and compute the availability of a scenario.

Bayesian networks have proven helpful in computing the availability of complex systems since they provide a powerful modeling formalism to express complex fault dependencies between components [31–33]. They support a rich set of efficient inference algorithms suitable for fault diagnostic [105] and availability prediction. This work argues that Bayesian networks can assess the availability of large-scale redundant and replicated services.

Many availability models use the independence assumption to implement efficient prediction algorithms. However, with the growing complexity of modern data centers, common cause failures are not the exception anymore; they are the norm [106, 107]. Common cause and cascading failures can lead to multi-node failures [108], while network failures can lead to network partitioning or total loss of communication [7, 109]. Therefore, the independence assumption does not represent real-life systems as such. While researchers acknowledge the significance of infrastructure and communication faults [110, 111], they usually model either the infrastructure [21, 101, 105], or the communication [112–114] part of a cloud service. Most of these models are highly specialized in emphasizing individual IaaS or PaaS services and rarely consider an application consisting of multiple services.

In this work, we design and implement a Bayesian network availability model that accounts for common-cause and cascading failures in the infrastructure and the network between the services. However, modeling the availability of large and complex systems with Bayesian networks is time-consuming and error-prone when done manually. In order to ease the modeling procedure, a contribution of this chapter is a high-level modeling formalism, which abstracts away the underlying details and required knowledge necessary to model the availability of a scenario with Bayesian networks. In detail, the high-level model is a conjunction of three sub-models. Each sub-model defines different types of failure modes in the cloud. The first sub-model represents the fault dependencies between components, the

second model accounts for communication failures, and the third model represents the availability requirements of the cloud service.

Another contribution is translating the high-level model into a Bayesian network. This consists of designing and implementing a procedure that can automatically translate the high-level model into its corresponding Bayesian network model. However, building the Bayesian network has to overcome its own modeling challenges. Since network partitioning failures or total loss of communication are frequent occurrences in cloud computing [7, 109], communication becomes a critical modeling aspect. In general, we distinguish between stateless and stateful services. Stateless services do not persist user states across multiple sessions, such as AWS Lambada, Azure Functions, or FaaS offering in general. In contrast, stateful services can persist user requests across multiple sessions, e.g., database or storage services. Replication is a common technique to increase fault tolerance for many stateful services. Replicas are not just redundant copies of the same instance; replication also involves communication between the instances to agree upon the next state of the service. In the case of stateless services, multiple copies of the same instance can be started and stopped on demand. These copies do not need to communicate with each other in order to agree upon a global state.

We say that a service is available as long it is reachable and working. In general, we call a service *redundant* when the service employs multiple copies of its instances without the necessity to implement a replication protocol. In contrast, we call a service *replicated* if its instances use a replication protocol to attain agreement on the next system state. Thus, replication also involves the reachability between sufficient working instances, where "sufficient" refers to the necessary number of available instances to implement the state-consistency requirements of the service, e.g., majority sets for services that implement strong consistency. As a result, we have two communication schemas. For redundant services, we only regard the reachability from client to instances, and for replicated services we also consider that replicas can communicate with each other. Hence, one important modeling contribution in this work is the implementation of redundant and replicated services with Bayesian networks, so we can assess the availability of services as the probability that their instances are working and are reachable given the likelihood of infrastructure and network failures. Moreover, we also present a modeling approach to express custom fault tolerance requirements for which a service is considered as up. Here, we provide efficient implementations for voting, weighted-voting, and read-one/write-all replication schemes.

The last contribution is an evaluation, where we empirically verify the Bayesian

network approach and analyze its limitations concerning performance and memory. The model verification compares the Bayesian network model with a fault tree model that implements the same system description. Given that the fault tree model uses a different mathematical formalism to compute the availability of a scenario, and assuming that the fault tree correctly implements the scenarios under test, evaluations show that the Bayesian network also implements the scenario description correctly by comparing their availability results. Additionally, the evaluations also show that the Bayesian network model exhibits a faster prediction performance for large services and scenarios than the fault tree model.

4.2. System Model

The model considers a cloud application as a set of interconnected cloud services that consist of individual instances. Service Instances are assumed to run on virtual or physical hosts linked by a communication network. The host is an aggregation of the operating system and the necessary runtime environment to execute an instance. It is placed within the infrastructure of one or more data centers. In detail, an instance is the aggregation of all software processes and data objects that are run and stored on a host. A redundant service is a service consisting of multiple stateless instances. Stateless instances are compute services that do not share or store the application state across multiple user sessions. In contrast, a replicated service consists of multiple stateful instances that implement a replication protocol to achieve state consistency. An instance is said to be available if it is up and reachable through the communication network by a client application. Although the instances are stateless, a redundant service might need more than one instance to be available. A redundant service is available as long k-out-of-n instances are available, where k is a parameter that can be chosen accordingly. If less than k instances are available, the system might be considered as overloaded or not able to fulfill aspects of its SLA. In the case of replicated services, it is assumed that instances cooperate to process a client request in a transaction-oriented manner. A replicated service is considered available when sufficient replicas are available. Furthermore, the model does not assume the concurrency control method or the particular replication protocol. Either there are enough replicas available that can reach a decision for some client request, or too many replicas crashed, such that the remaining replicas cannot act upon incoming requests.

A particular placement configuration of instances to virtual or physical hosts is

called a “deployment” and is known beforehand. Instances do not migrate. If an instance crashes, it does not recover on a different host, and it gets re-instantiated or restarted back at its former host. If a host crashes, all its instances can only recover when the host recovers. The model makes no restrictions on the number of instances per host. Instances and client applications can communicate with each other by exchanging messages via the communication network. The network is assumed to consist of components such as switches, routers, and middleboxes, e.g., firewalls, which are placed within the same infrastructure as the hosts. The hosts and the communication network are part of the infrastructure, forming a complex component-based system consisting of *infrastructure components*, such as data centers, racks, power supplies, VMs, and network appliances. The model assumes that hard – and software – components, including the instances, have a crash-recovery model. As soon a component encounters a failure, it crashes, stops, and eventually recovers. Here, the model does not assume a particular recovery mechanism. The only assumption is that the component is ”good-as-new“ after the recovery, i.e., the component has returned to a working state with the same failure probability as before.

The end-to-end communication between a service instance to a client application and a service instance to another can be synchronous or asynchronous. A route along the network graph realizes the communication. A route becomes unavailable when at least one network component along the route crashes. If multiple potential routes exist to realize the communication, then an instance is considered reachable as long as one route is available. Client applications might be placed outside of the known infrastructure. In this case, the model considers the paths starting from the network appliance that constitutes the entry point to the data center; or its hosts if the client application is within the data center. Moreover, all service instances have one or more dedicated network components, e.g., firewalls or load balancers, that act as a *gateway* to communicate with the instances. Hence, every client application has to enter the cloud network through some defined entry points and then communicate with the desired service through one of its defined gateways.

4.3. Cloud Application Model

Modeling the availability of a large and complex system with Bayesian networks can become increasingly time-consuming and error-prone when done manually. A common technique is to introduce a higher-level model that is domain-specific

and focuses on the system description rather than on enumerating failure modes. Afterward, a translation procedure transforms the high-level model into its specific Bayesian network model. Therefore, this section begins with the high-level cloud application model as part of the overall availability prediction task. The next section will then show how to translate this model into a Bayesian network to compute an availability value.

4.3.1. Overview Example

Let us start with an example to introduce the high-level model, which we will use throughout this chapter to demonstrate and explain how to build the respective Bayesian network model.

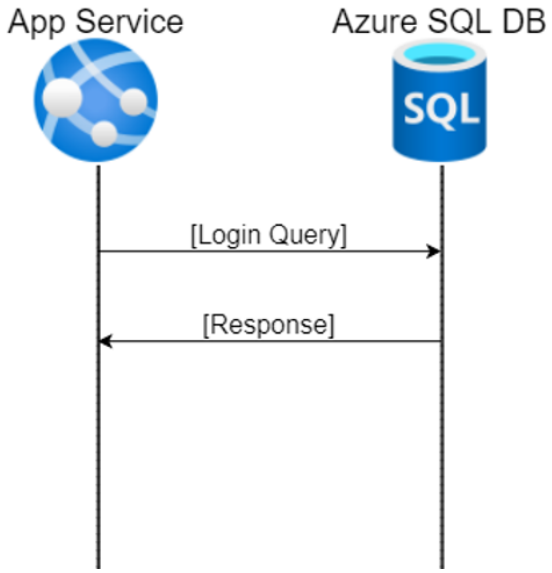


Figure 4.2.: Example of a login scenario with two services.

Let us assume a developer designs a login feature with two Azure services in Clams, as shown in Figure 4.2. The login feature consists of an App Service and an Azure SQL Database instance. The latter is a transaction-oriented relational database system using replication to increase fault tolerance. In contrast, the former is a FaaS offering, which employs redundant instances to scale up vertically. The Clams model does not contain sufficient information to compute an availability prediction in its initial form. Additional information needs to be added to build the proper model that gives rise to the necessary failure modes. While application models in Clams consider services as logical units, the availability model requires a different granularity level of the system. Hence, the high-level model offers knowledge about service instances and their execution environments,

including additional components such as cloud infrastructure and network components. How do we get from a scenario model to this enhanced model? We can simply implement this enhancement as part of a preprocessing procedure for our availability evaluation service using the tooling offered in OpenClams. For example, cloud providers can offer such an evaluation service since they have ample knowledge of their cloud. However, for now, we assume that we have this knowledge. We know how all relevant cloud components are connected and where service instances are deployed. Hence, we assume the Azure SQL Database has seven replicas denoted by the instance names $I_{\mathrm{DB},1}$ to $I_{\mathrm{DB},7}$, requiring the majority of instances to be available, whereas the App Service runs two instances $I_{\mathrm{AS},1}$ and $I_{\mathrm{AS},2}$. If one App Service instance fails, the remaining one can still handle the load demand to provide an available service.

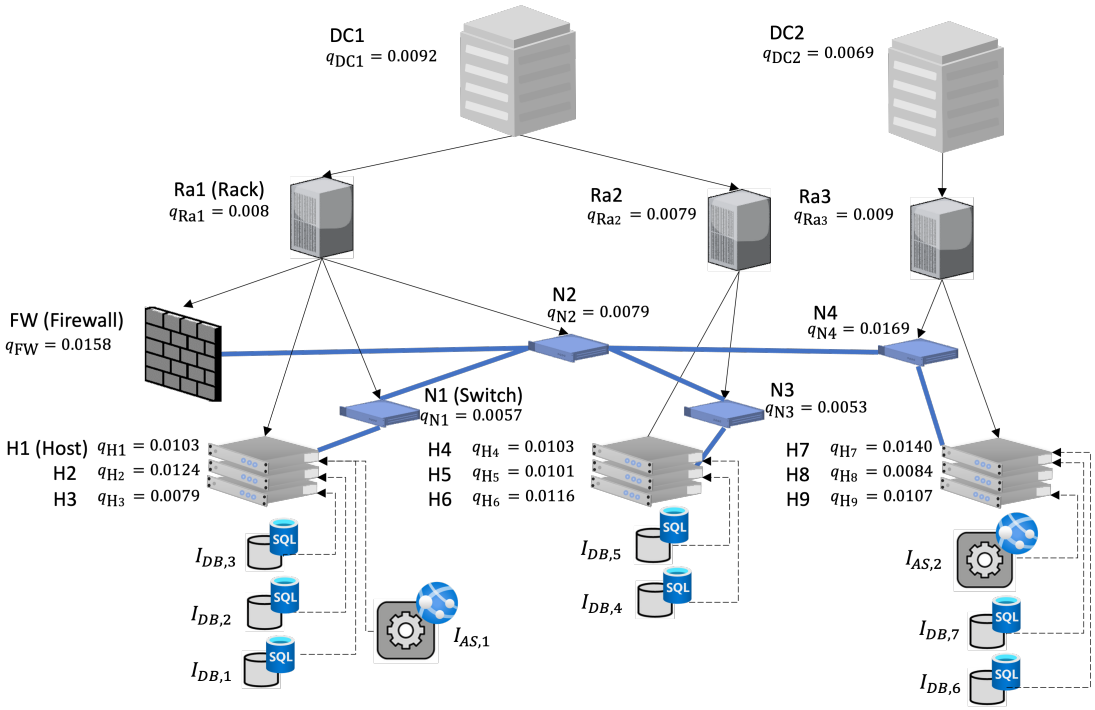


Figure 4.3.: Compute and network infrastructure to execute the login scenario

In order to account for different failure modes, the replicas of the database system and the instances of the App Service need to be mapped within a cloud infrastructure model. Figure 4.3 shows the resulting cloud infrastructure model as a graph of components and service instances. The infrastructure model consists of two data centers DC_1 and DC_2, three racks Ra_1 to Ra_3, network appliances like the firewall FW and switches N_1 to N_4, nine hosts H_1 to H_9, and the previously mentioned service instances. Although the cloud infrastructure might be much larger in real life, we only consider those components which serve the service

instances. Also, this model does not regard the whole application, which might contain more services, but rather those required to implement this scenario. If developers wish to assess the full architecture, they can simply design one scenario containing all services. So far, all instances are assigned to their potential hosts as if the services were deployed. Black arrows denote external failure causes, and blue edges represent the communication network. For instance, individual instances might fail due to some intrinsic fault or if their hosts fail. A host is susceptible to fail on its own, e.g., operating system failure, or if their rack fails, possible due to a power outage. If rack Ra_1 would fail, all built-in components, such as the hosts H_1 to H_3, the firewall, and the switches N_1 and N_2 would also fail. Hence, the replicas $I_{\mathrm{DB},1}$ to $I_{\mathrm{DB},3}$, and the App Service process $I_{\mathrm{AS},1}$ would fail as well, since Ra_1 is a common cause failure for all of them.

The final availability model needs to consider the network to account for communication failures. Therefore, Figure 4.3 has blue edges, representing communication links between network components. Let us assume that every instance has the potential to communicate with every other instance. Consequently, on the arrival of a user request at an instance of the App Service, the instance only needs to communicate with at least one replica of the database service. However, replicas need to be able to communicate with at least three other replicas to execute the replication protocol. A communication channel between any two replicas is available if the two replicas that form the channel's endpoint are working and at least one route within the network graph with working network components exist. Therefore, if switch N_1 fails, the remaining instance can still execute the login feature. However, if switch N_2 fails, then the network gets partitioned into three parts, in which none of them have sufficient instances or database replicas to realize the login feature. Moreover, suppose that the firewall FW represents all client applications suited outside the cloud infrastructure, communicating with the services through the firewall. If FW fails, the whole scenario becomes unavailable from the user's perspective because no service is reachable. However, the internal network and all instances are still working and can communicate with each other. The goal is to formalize what we just discussed as a high-level model so that the developer can describe the cloud application without the need to express all failure modes explicitly.

4.3.2. Fault Dependency Graph

As shown in Figure 4.3, the high-level model has to account for fault dependencies between cloud components. Hence the core element is, first of all, the component, which represents any virtual or physical entity in the cloud. Afterward, we describe the fault dependencies with the help of a graph model.

Definition 4 (Component) *A component $C \in \boldsymbol{C}$, from the finite set of all considered components of the cloud system $\boldsymbol{C} = \{C_1, C_2, \dots\}$, is an indivisible hard or software entity with the observable states $\{F, T\}$, and a Bernoulli probability distribution $P(C = F) = q_i$ to observe the component as faulty and $P(C = T) = 1 - q_i$ to observe the component as working when required.*

$\boldsymbol{C}$ is the set of all components, which also contains the set of all service instances. So, we define $\boldsymbol{I} = \{I_1, \dots, I_n\} \subset \boldsymbol{C}$ as the set of instances. The remaining components are infrastructure and network components.

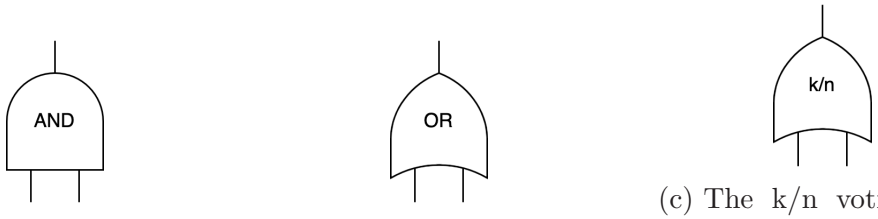
Components can fail on their own or due to external causes. These external causes arise from their dependency on another component, as indicated by the black arrows in Figure 4.3, which we simply refer to as fault dependencies.

Definition 5 (Fault Dependency Graph) *Given the set of all components $\boldsymbol{C}$, the model defines the fault dependency graph as a Direct Acyclic Graph (DAG) $G_{FD} = (\boldsymbol{C}, E_{INF}, FT)$, with edges $E_{INF} \subseteq \boldsymbol{C} \times \boldsymbol{C}$, and an associated fault tree model FT for every component in $\boldsymbol{C}$.*

Directed edges are tuples (C_i, C_j), where C_i is said to be a parent component of C_j, and C_j is said to be a child component of C_i. Components can have multiple parent components. In this case, we need to define which specific failure combination triggers a failure at the child component. Is it the failure of all parent components? Is it the failure of just one parent component or a completely different component combination? For example, if a host has two redundant power supplies, then it is certain to fail when both power supplies fail. To encode complex failure relations between child and parent components, the fault dependency graph associates each node with a fault tree $FT(C_i)$ that describes the failure semantics of a component C_i as a boolean expression of the observed failure state of the parent components. These (usually small) fault tree models are local for each component and only consider parent components.

Fault trees are graphs that describe how certain combinations of component faults, known as *base events*, can lead to an undesired system failure, known

as the *top event*. Gates can be used to create intermediate events by forming a Boolean expression to describe what combinations of base events lead to a system failure [115]. Each base event has a probability value to indicate the likelihood of observing a component failure. There exist multiple fault tree variations in literature. This work will only focus on the three basic gate types, which have all fault tree variations in common: the AND, OR, and the k-out-of-n voting gate, as shown in Figure 4.4. Gates accept as input either base events or output events of other gates. The AND gate triggers a fault event if all its inputs receive a failure event. Conversely, the OR gate triggers a fault event when at least one input registers a failure event. The voting gate triggers a fault event when more than k-out-of-n inputs register a failure event. Hence, the voting gate is suitable to model groups of redundant components, where a group is considered working as long as no more than $n - k + 1$ components are working.



(a) The AND gate propagates a fault if all input events trigger a fault.

(b) The OR gate propagates a fault when one input event has triggered a fault

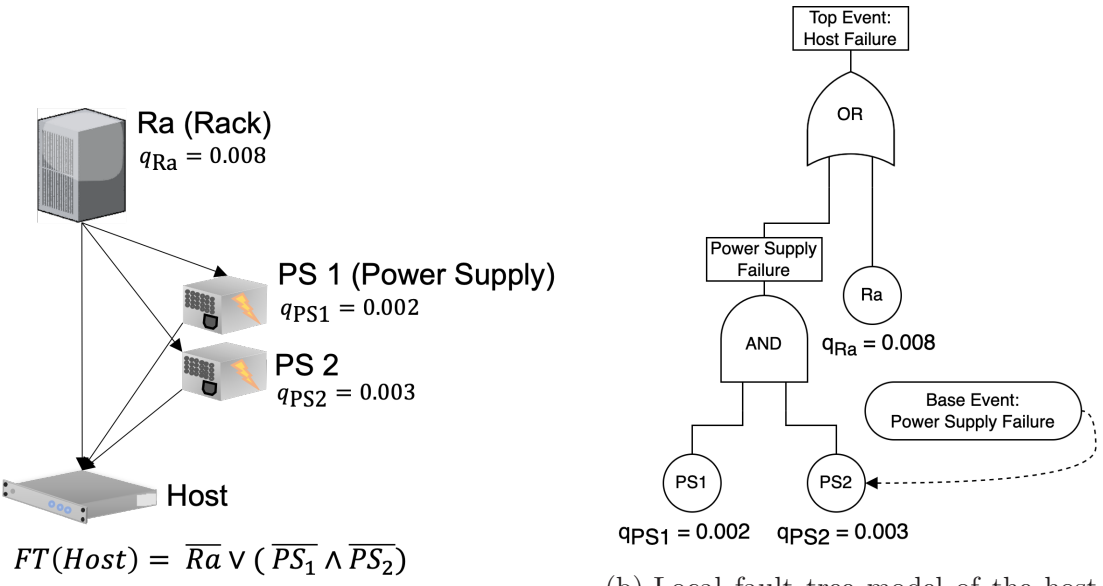
(c) The k/n voting gate propagates a fault when more than k-out-of-n input events are fault events.

Figure 4.4.: Summary of the basic fault tree gates.

Figure 4.5a shows a fault dependency graph with a host component that depends on three parent components to illustrate how to apply $FT(.)$. The host fails if the rack breaks, e.g., catching fire, or both power supplies stop working. $FT(\text{host})$ encodes this failure relation at the host component, as shown in Figure 4.5a. Hence, the corresponding fault tree shown in Figure 4.5b has the power supplies and the rack as basic input events and the host failure as the top event. The fault tree uses an OR gate to trigger the top event. The hosts fails when rack fails, or both power supplies fail, represented by the AND gate. The fault tree is part of the host component to determine the likelihood of a host failure due to an external cause, which depends solely on the failure probabilities of the parent components. Therefore, each component has its own individual fault tree model.

Note that the fault dependency model is a DAG, disallowing cyclic fault dependencies since it leads to cycles in the final Bayesian network graph, which is

not allowed by definition. However, in Section 4.4.2 we will discuss the modeling conditions under which one can model cycles.



(a) Fault dependencies of a host.

(b) Local fault tree model of the host component.

Figure 4.5.: Fault Dependency Graph Example.

4.3.3. Network Graph

To account for communication failures, e.g., network partitioning failures, the high-level model needs to consider the network. In the network model, the host acts as an interface for instances to communicate with other instances or client applications. Network appliances and hosts form the nodes of the network topology. In our model, a host can be a virtual or physical machine that constitutes the execution environment of the service instance aggregating several sup-components such as the operating system, the execution runtime, and other processes that are required to sustain and run the service instance.

Definition 6 (Network Graph) *Given the set of host components $H \subset \boldsymbol{C}$, a set of network components $N \subset \boldsymbol{C}$, and their union $C_{NET} = H \cup N$, the network is a unidirectional graph $G_{NET} = (C_{NET}, E_{NET})$, where the edges $E_{NET} \subseteq C_{NET} \times C_{NET}$ define the communication links between any two network components.*

Network components consist of infrastructure components representing network appliances such as switches, routers, load-balancers, and firewalls. Network

components are also part of the fault dependency graph, so the fault of infrastructure components can influence the fault of network components, which can lead to communication failures in return. Unlike the fault dependency graph, the network graph can have cycles.

With this graph notion, the reliability engineer can decide the granularity of the network model. Suppose the reliability engineer has little or no knowledge of the network. In that case, he can represent the network as 'one switch' connecting all replicas and aggregating all potential failures in one *super* component. However, reliability engineers can describe a more complex network graph with ample network knowledge, improving the assessment due to a more realistic system representation.

4.3.4. Service Model

So far, this section has introduced two graph models. These are a fault dependency graph to infer fault relations between components and a network graph to infer network communication failures. The service model needs to consider three aspects to address the availability definition of "working" and "reachable". First, the service's deployment in the cloud infrastructure, which defines the external failure causes of service instances, and second, how the service implements fault tolerance. Lastly, how client applications or other instances of different services reach the service through the network.

Definition 7 (Cloud Service) *A cloud service* $S = (I, D, \mathcal{G}, Q, c)$ *is a five-tuple consisting of the following elements:*

- *The set* $I = \{I_{S,1}, \ldots, I_{S,n}\} \subseteq \boldsymbol{I}$ *of* n *instances that constitutes the service.*
- *The deployment function* $D : I \to H$ *that maps instances to hosts.*
- *A set of network components that constitute the entry points (gateways)* $\mathcal{G} \subseteq C_{NET}$ *to access the instances.*
- *A set of instance combinations* $Q \subseteq 2^I$ *(power set) for which the service is considered working in the presence of instance failures.*
- *The communication scheme between instances as a Boolean value* $c \in \{false, true\}$*, where* $c = true$ *signifies that instances communicate with each other, to indicate a replicated service.*

An instance aggregates all software processes and data objects to instantiate and serve the service from one virtual or physical host. The set I and the function D cover the first aspect of inferring failure causes of instances. Each instance is connected to one host according to the deployment function D. In this manner, instances become part of the fault dependency graph, connecting the instances to the previous two models.

The set $\mathcal{G}$ covers the aspect of reachability by containing network components that act as an entry point to the service. Commonly, cloud services use firewalls and load balancers to protect instances of unauthorized connections and overloading. Some services might even have their own sub-network so that client requests must pass a router to reach the service. We assume that every client application that wishes to connect to some instance of the service has to pass through such a network appliance, which we simply call a *gateway*. Once we know the set of gateways, the translation producer computes only channels from client to instances that traverse those network components. These channels then form the basis for computing the reachability property in the final availability model.

The last two parameters define the fault tolerance aspect of a service. Generally, we assume instances can be either stateful or stateless. A stateful instance can persist service state or user data across multiple user sessions, e.g., storage or database services. Consequently, stateless instances do not persist state across a user request. For example, HTTP servers are stateless by definition due to their use of the HTTP protocol. FaaS offerings like AWS Lambda and Azure's Functions are also stateless.

A service is replicated if it has stateful instances. This implies the execution of a replication protocol, meaning instances need to communicate with each other, increasing the risk of communication failures. Hence, depending on the replication requirements, network partitioning failures can lead to the unavailability of the service by creating partitions where insufficient instances can communicate. Stateless instances do not communicate with each other to exchange states. Hence, their availability assessment only considers the communication from the client to the instances. Hence, the parameter c defines whether or not the instances communicate with each other. The parameter Q defines all instance combinations for which the service is considered in a working state in the presence of instance failures. This definition implies the enumeration of all instance combinations to build Q. For example, let us assume a service has three instances $I = \{I_1, I_2, I_3\}$ and the service works as long as two instances are up.

As a result, Q is the enumeration of all combinations with at least two instances $Q = \{\{I_1, I_2\}, \{I_1, I_3\}, \{I_2, I_3\}, \{I_1, I_2, I_3\}\}$. In standard terms Q is a (minimal) path set of the service instances.

The enumeration of all instance combinations can become inefficient, especially when considering services with tenths of instances. To alleviate this burden, this work introduces an implicit construction method for k-out-of-n redundancy and voting-based replication models, as well as for the special cases of read-one and write-all replication. As a result, for these specific models, it is sufficient to define Q as a tuple (V, t), where $V = (v_1, \ldots, v_n)$ are instance votes and t a threshold value. The availability model will then account for the probability of observing sufficient working instances such that their votes exceed the threshold. For example, we can express the previous examples as $Q = ((1, 1, 1), 2)$ to implement the majority set without enumerating all possible set combinations.

If the service has different thresholds, that is, different quorum size requirements per operation, like read-one/write-all replication, which has $t = 1$ for the read operation and $t = n$ for the write operation, then the service definition refers to one specific operation. Multiple operations can be supported by defining a service model for each operation separately and compute their availability values. At this point, it is up to the developers or reliability engineers to decide on how to aggregate the availability of the different operations. They can use the lowest resulting value as a means to assess the probability of the worst-case service model, or they could compute the (weighted) average availability across all operations. Independently of what aggregation method a developer or reliability engineer chooses, this work shows how to build the availability model for the operation accordingly.

4.3.5. Scenario Model

Our goal is to build a high-level model as an intermediary between the scenario notation of Clams and the availability model that forms the second layer in the hierarchical composition, c.f. Section 3. In the following, we discuss the scenario model that encompasses the previously defined models into one definition. A scenario consists of one or more services in a message sequence graph communicating to serve an application-level functionality. Based on the message exchange in the message sequence graph, the services form a communication graph, i.e., a service topology. The communication between the services introduces a new source of potential failures, which requires additional information to build the appropri-

ate availability model of the scenario. First, for a scenario to be reachable and working, all services must be working and communicate according to the service topology. Second, the client application that invokes a scenario needs a location in the infrastructure to infer the channels from the client to the instances of the first service that initially receives incoming client requests. In general, client applications are normally suited outside the observable infrastructure, e.g., mobile devices, which is why the model uses the network appliance that provides an entry point into the cloud network as a representative for the client. Multiple entry points are possible, and if the client is suited within the cloud infrastructure, the model simply uses the client's host as an entry point to the network. These two aspects and the previously defined models lead to the final model definition of a scenario.

Definition 8 (Application Scenario Model) *An application scenario*

$$\mathcal{A} = (\boldsymbol{C}, \boldsymbol{S}, G_{FD}, G_{NET}, G_{\boldsymbol{S}}, \mathcal{G}_{entry}, S_{init})$$

is a seven-tuple consisting of the following elements:

$\boldsymbol{C}$ *The set of all cloud components.*

$\boldsymbol{S}$ *The set of n cloud services* $\boldsymbol{S} = \{S_i\}_{i=1}^{n}$.

$G_{\boldsymbol{FD}}$ *The fault dependency graph.*

$G_{\boldsymbol{NET}}$ *The network graph.*

$G_{\boldsymbol{S}}$ *The service topology* $(\boldsymbol{S}, E_{SERVICE})$*, where services are nodes and the edges* $E_{SERVICE} \subseteq \boldsymbol{S} \times \boldsymbol{S}$ *define the communication links.*

$\mathcal{G}_{\boldsymbol{entry}}$ *The set of network components that act as entry point for client applications* $\mathcal{G}_{entry} \subseteq C_{NET}$.

$S_{\boldsymbol{init}}$ *The initiating service* $S_{init} \in \boldsymbol{S}$ *that handles incoming client requests.*

This scenario definition is not limited to Clams, but can also be regarded as a general structure to model architecture-based cloud models. The key parameter is the service topology $G_{\boldsymbol{S}}$, which essentially defines the application architecture at the service level.

The first three parameters of the scenario model will help us define the fault relations between components and the instances of the services. The Bayesian

network model will use these parameters to infer the probability of observing service instances as working given the probability of cascading and common cause failures of other cloud components. The last four parameters account for the communication between client applications and services and between the services themselves. Based on these parameters, the availability model infers the probability of reaching service instances. Moreover, the scenario model has two different sets of gateways. The network components in $G_{\boldsymbol{S}}$ that define the entry points for client applications into the cloud network, and the gateways of the services in $\boldsymbol{S}$, which define the entry point to the instances of the specific services.

Example Model

In the following, the remainder of this section shows how to model the login example from Section 4.3.1 with the scenario definition. The model has the following form:

$$\mathcal{A}_{\text{Login}} = (\boldsymbol{C}, \boldsymbol{S}, G_{\text{FD}}, G_{\text{NET}}, G_{\boldsymbol{S}}, \mathcal{G}_{\text{entry}}, S_{\text{init}})$$

First, we define the set of all components and assign the fault probabilities of observing the components as unavailable.

$$\begin{aligned}\boldsymbol{C} =\{&DC_1, DC_2, Ra_1, Ra_2, Ra_3, FW, N_1, N_2, N_3, N_4, H_1, H_2, H_3, H_4, H_5, H_6,\\ &H_7, H_8, H_9, I_{\text{DB},1}, I_{\text{DB},2}, I_{\text{DB},3}, I_{\text{DB},4}, I_{\text{DB},5}, I_{\text{DB},6}, I_{\text{DB},7}, I_{\text{AS},1}, I_{\text{AS},2}\}\end{aligned}$$

$$\begin{aligned}P(DC_1 = F) = 0.0092 \quad P(DC_2 = F) = 0.0069 \quad P(Ra_1 = F) = 0.008\ldots\\ P(H_7 = F) = 0.0140 \quad P(H_8 = F) = 0.0084 \quad P(H_9 = F) = 0.0107\end{aligned}$$

For the sake of readability, we assume that instances do not fail due to intrinsic faults. Hence, they have an availability of one.

Afterwards, we describe the set of all services which contains the database service and the App Service.
Hence, the scenario consists of $\boldsymbol{S} = \{S_{\text{DB}}, S_{\text{AS}}\}$.

$$S_{\text{DB}} = (\{I_{\text{DB},1}, I_{\text{DB},2}, I_{\text{DB},3}, I_{\text{DB},4}, I_{\text{DB},5}, I_{\text{DB},6}, I_{\text{DB},7}\}, D_{\text{DB}}, \{FW\}, Q_{\text{DB}}, c = \mathit{true})$$

The database service has seven replicas, i.e., stateful instances, that can tolerate three replica failures. With $c = \mathit{true}$, the service model will also consider communication between instances. Finally, we use the shorthand notation

$Q_{\mathrm{DB}} = ((1,1,1,1,1,1,1),4)$ to define a voting-based replication scheme, where each instance has one vote and a threshold of four to signify the necessity of a majority set for a working service.

$$S_{\mathrm{AS}} = (\{I_{\mathrm{AS},1}, I_{\mathrm{AS},2}\}\}, D_{\mathrm{AS}}, \{FW\}, Q_{\mathrm{AS}}, c = \mathit{false})$$

The App Service has two stateless instances, where one instance is sufficient for a working service. Hence, $Q_{\mathrm{AS}} = \{\{I_{\mathrm{AS},1}\}, \{I_{\mathrm{AS},1}\}, \{I_{\mathrm{AS},1}, I_{\mathrm{AS},2}\}\}$ contains all instance combinations, where at least one instance is working to define the working state of the service. Moreover, since they are stateless, no communication is needed between the two instances, i.e., $c = \mathit{false}$. Finally, both services use the firewall component FW as a gateway to access their instances, which are placed as follows:

$$\begin{aligned} D_{\mathrm{DB}}(I_{\mathrm{DB},1}) = H_1 \quad D_{\mathrm{DB}}(I_{\mathrm{DB},2}) = H_2 \quad D_{\mathrm{DB}}(I_{\mathrm{DB},3}) = H_3 \\ D_{\mathrm{DB}}(I_{\mathrm{DB},4}) = H_4 \quad D_{\mathrm{DB}}(I_{\mathrm{DB},5}) = H_5 \quad D_{\mathrm{DB}}(I_{\mathrm{DB},6}) = H_7 \\ D_{\mathrm{DB}}(I_{\mathrm{DB},7}) = H_7 \quad D_{\mathrm{AS}}(I_{\mathrm{AS},1}) = H_1 \quad D_{\mathrm{AS}}(I_{\mathrm{AS},2}) = H_9 \end{aligned}$$

Next, we build the fault dependency graph, where the instances use the deployment function to identify their host within the fault dependency graph.

$$G_{\mathrm{FD}} = (\boldsymbol{C}, E_{\mathrm{INF}}, FT)$$

$$\begin{aligned} E_{\mathrm{INF}} = \ & \{(DC_1, Ra_1), (DC_1, Ra_2), (DC_2, Ra_3), (Ra_1, FW), \dots \\ & (D_{\mathrm{DB}}(I_{\mathrm{DB},6}), I_{\mathrm{DB},6}), (D_{\mathrm{DB}}(I_{\mathrm{DB},7}), I_{\mathrm{DB},7}), \\ & (D_{\mathrm{AS}}(I_{\mathrm{AS},1}), I_{\mathrm{AS},1}), (D_{\mathrm{AS}}(I_{\mathrm{AS},2}), I_{\mathrm{AS},2})\} \end{aligned}$$

Figure 4.6 shows the resulting graph of components and their failure relations.

In this example, we assume that a component automatically fails when its parent component fails. So, FT is a simple mapping of the failure event of the parent component $pa(C)$ of C. If a component has no parent, e.g. DC_1, then $pa(.)$ returns the empty set. Consequently, we get for all components that $\forall C \in \boldsymbol{C} : FT(C) = \bigwedge_{C_i \in pa(C)} \overline{C_i}$.

The network graph $G_{\mathrm{NET}} = (C_{\mathrm{NET}}, E_{\mathrm{NET}})$ contains the network components and hosts. Figure 4.7 depicts the communication links as blue edges.

$$C_{\mathrm{NET}} = \{FW, N_1, N_2, N_3, N_4, H_1, H_2, H_3, H_4, H_5, H_6, H_7, H_8, H_9\}$$

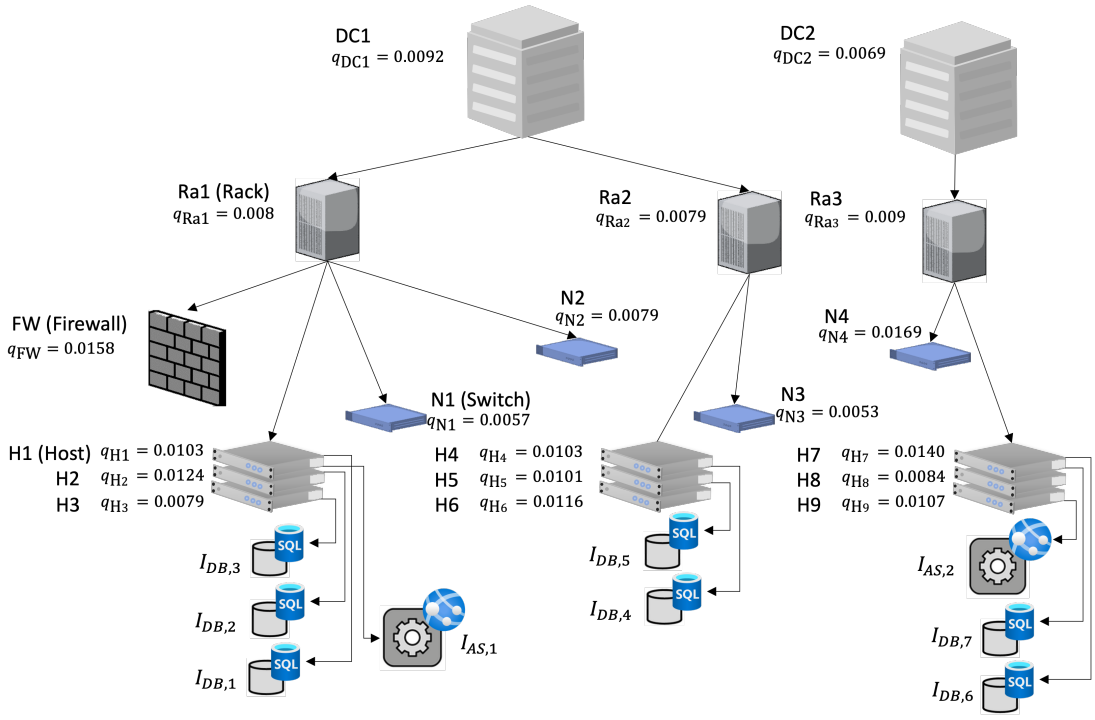


Figure 4.6.: Scenario model with all components and their fault dependencies.

$$E_{\text{NET}} = \{\{FW, N_2\}, \{N_2, N_1\}, \{N_2, N_3\}, \{N_2, N_4\}, \ldots \\ \{N_4, H_7\}, \{N_4, H_8\}, \{N_4, H_9\}\}$$

In this example, we assume that the entry point is the firewall $\mathcal{G}_{\text{entry}} = \{FW\}$ for all client applications.

Finally, the first service that accepts incoming requests is the App Service $S_{\text{init}} = S_{\text{AS}}$, which then communicants with the database service, resulting into a service topology $G_{\boldsymbol{S}} = (\boldsymbol{S}, E_{\text{SERVICE}})$ with one edge.

$$E_{\text{SERVICE}} = \{\{S_{\text{DB}}, S_{\text{AS}}\}\}$$

Figure 4.7 shows the final service topology in the upper-left corner. This concludes the scenario model. The next section shows how to translate this high-level model into a Bayesian network availability model.

4.4. Bayesian Network Availability Model

The translation procedure consists of three steps. First, the procedure builds a Bayesian network model of the fault dependency graph. Afterward, in the case of replicated services, the procedure extends the Bayesian network model to account

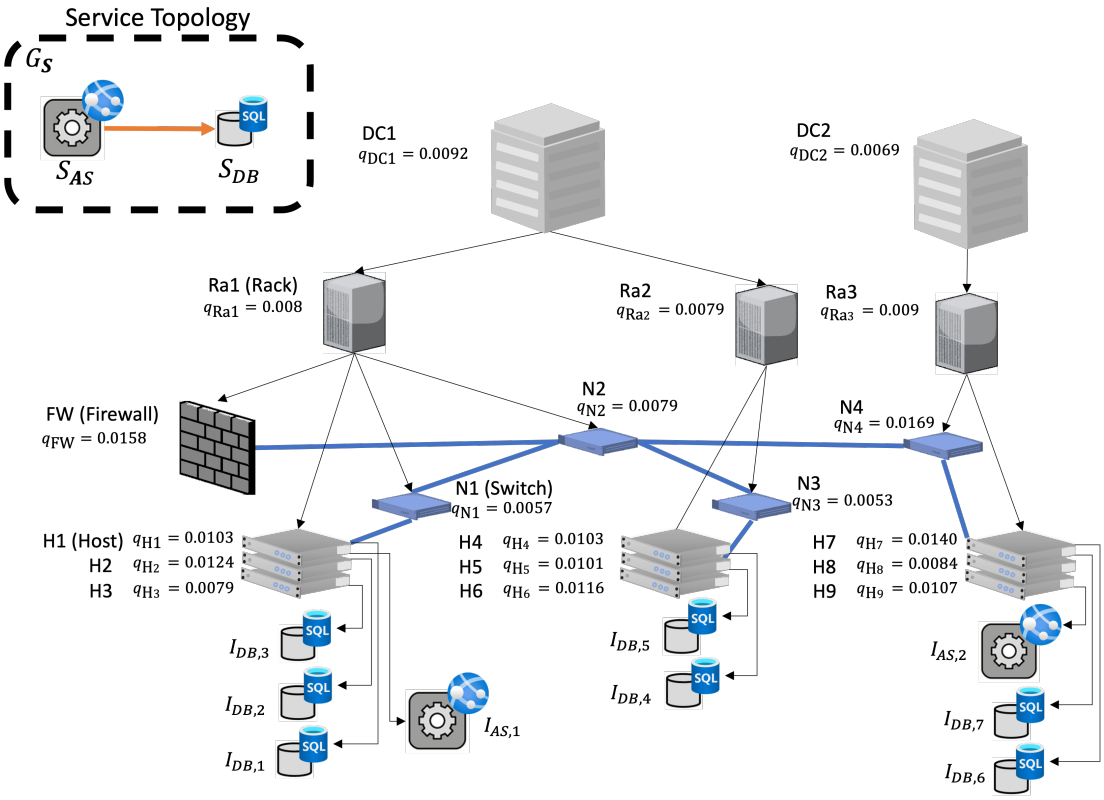


Figure 4.7.: Scenario model with service topology and network links.

for failures in communication between instances. Finally, the third step includes the failure modes for client-to-service and service-to-service communication.

4.4.1. Background

Throughout this work, the availability model will primarily use the Bayesian network formalism. Hence, this section provides the necessary background to understand the Bayesian network notation and their equivalent fault tree representation.

A discrete Bayesian network [116] is a DAG $G = (X, E)$ that represents a joint probability distribution $P(X)$ over the set of discrete random variables $X = \{X_1, X_2, \ldots X_n\}$. The term *variable* or *node* are used interchangeably to denote the vertices of the Bayesian network graph. For every edge $(X_i, X_j) \in E$ between the nodes X_i and X_j, X_i is said to be a parent node of X_j, and X_j is a child node of X_i. We use the parent function $\text{pa}(X_i) = \{X_p : \forall(X_p, X_i) \in E\}$ to denote the set of all parent nodes for a given child node X_i. Each variable has a conditional probability distribution $P(X_i = x_i|pa(X_i))$ expressed as a conditional probability table (CPT). The CPT contains the probability of observing a certain

state $X_i = x_i$ given the observed state combinations of its parent nodes. Nodes without parents are called root nodes and have a prior probability distribution $P(X_i = x_i)$. Figure 4.8 shows on the right side a Bayesian network with three root nodes and an inner node with their corresponding CPTs.

A Bayesian network entails the full joint probability distribution as the product of all the nodes' conditional probability distributions:

$$P(X) = \prod_{x \in X} P(x|\text{pa}(x)) \tag{4.1}$$

With the help of the joint probability distribution, we can use existing inference algorithms to compute the posterior distribution $P(Y|X')$ of some query $Y \subset X$ of uncertain variables from a given subset $X' \subset X \backslash Y$ of observations of the remaining variables.

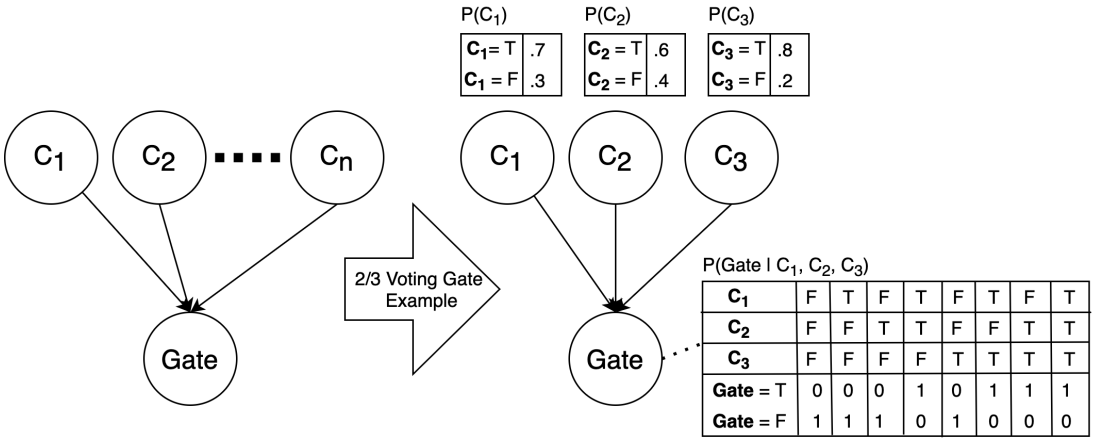


Figure 4.8.: Basic Bayesian network to represent the AND/OR, or k-out-of-n fault tree gate (left). Example instance of a Bayesian network k-out-of-n model (right).

The Bayesian network availability model will use CPTs to model probability distributions that behave equivalent to fault tree gates to model the system's failure probability [117, 118]. Bobbio et al. [117] introduced the general approach to represent fault tree gates with the help of Bayesian networks, which this work will use as building blocks. Next, we discuss how to represent the AND/OR and the k-out-of-n voting gate as probability distributions with the help of CPTs.

Figure 4.8 shows the main Bayesian network structure on the left side to realize the AND/OR and the k-out-of-n voting gate. The general structure has one or more components (or base events) C_1 to C_n with prior probabilities represented by their eponymous binary random variables with states $\{F, T\}$, observing the

component as either faulty or available, respectively. The individual semantics of the gates are encoded as conditional probability distributions within the CPT of the *Gate* node.

Fault tree gates are defined over the fault state of their input events. Whenever an input triggers a fault event, the event takes the state *true* to indicate that a fault occurred. However, in our model, the fault state of a component uses the state value F instead, making, for example, the AND semantic for component failures counter-intuitive to the basic AND expression in Boolean algebra. Nevertheless, to stick to the original intention of the fault tree gates, the Boolean expression of AND/OR and the voting gate acts upon the fault state of the components. An example follows next when discussing the gates in detail.

AND Model

Figure 4.8 shows on the left side the basic structure of a Bayesian network to model the AND failure semantic. For every state combination of the parent nodes, we will observe Gate $= F$ if all parent nodes are observed to be in state F. Hence, for example, if we have two parent nodes C_1 and C_2, the CPT for the Gate node results in the following shorthand definition, accounting for both states:

$$
\begin{aligned}
&P(\text{Gate} = F | C_1 = F, C_2 = F) = 1, \quad P(\text{Gate} = T | C_1 = F, C_2 = F) = 0 \\
&P(\text{Gate} = F | C_1 = F, C_2 = T) = 0, \quad P(\text{Gate} = T | C_1 = F, C_2 = T) = 1 \\
&P(\text{Gate} = F | C_1 = T, C_2 = F) = 0, \quad P(\text{Gate} = T | C_1 = T, C_2 = F) = 1 \\
&P(\text{Gate} = F | C_1 = T, C_2 = T) = 0, \quad P(\text{Gate} = T | C_1 = T, C_2 = T) = 1
\end{aligned}
$$

As mentioned earlier, the AND acts upon the fault state of the parent nodes, resulting into a logical OR evaluation to observe the state T. The resulting conditional probability distribution has a certain outcome for every observed state combination of C_1 and C_2, which can be generalized for an arbitrary number of parent nodes by asserting every state combination of the parent nodes:

$$
\begin{aligned}
&P(\text{Gate} = F | \forall C \in pa(\text{Gate}) : C = F) = 1, \quad P(\text{Gate} = T | \forall C \in pa(\text{Gate}) : C = F) = 0 \\
&P(\text{Gate} = F | \exists C \in pa(\text{Gate}) : C = T) = 0, \quad P(\text{Gate} = T | \exists C \in pa(\text{Gate}) : C = T) = 1
\end{aligned}
\tag{4.2}
$$

OR Model

Again we use Figure 4.8 as a basic structure to express the OR failure semantic. For every state combination of the parent nodes, we will observe Gate $= F$ if at least one parent node is in state F. For example, if we have two parent nodes C_1 and C_2, the CPT for the Gate node results in the following definition:

$$\begin{aligned}
&P(\text{Gate} = F | C_1 = F, C_2 = F) = 1, \quad P(\text{Gate} = T | C_1 = F, C_2 = F) = 0 \\
&P(\text{Gate} = F | C_1 = F, C_2 = T) = 1, \quad P(\text{Gate} = T | C_1 = F, C_2 = T) = 0 \\
&P(\text{Gate} = F | C_1 = T, C_2 = F) = 1, \quad P(\text{Gate} = T | C_1 = T, C_2 = F) = 0 \\
&P(\text{Gate} = F | C_1 = T, C_2 = T) = 0, \quad P(\text{Gate} = T | C_1 = T, C_2 = T) = 1
\end{aligned}$$

This can be again generalized for an arbitrary number of parent nodes as follows:

$$\begin{aligned}
&P(\text{Gate}=F | \forall C \in pa(\text{Gate}) : C=T)=0, \quad P(\text{Gate}=T | \forall C \in pa(\text{Gate}) : C=T)=1 \\
&P(\text{Gate}=F | \exists C \in pa(\text{Gate}) : C=F)=1, \quad P(\text{Gate}=T | \exists C \in pa(\text{Gate}) : C=F)=0
\end{aligned} \tag{4.3}$$

k-out-of-n Model

The k-out-of-n voting gate triggers a fault event when k or more input events are in a faulty state. Hence, the CPT of the corresponding Bayesian network model has to address every observable state combination of the parent nodes. Whenever a combination has more than k parent nodes in state F, the conditional probability distribution returns Gate $= F$ with probability one. For example the CPT of Gate in Figure 4.8 shows the implementation of a two-out-of-three voting gate. Again, we can express the CPT for arbitrary numbers of parent nodes. We will use the indicator function $\mathbf{1}_F(x)$ when providing the CPT definition.

$$\mathbf{1}_F(x) := \begin{cases} 1 & \text{if } x = F, \\ 0 & \text{otherwise.} \end{cases}$$

This function returns one whenever a random variable takes on the state $X = F$, otherwise zero. In the following, we will use the short hand notation c_i instead of $C_i = c_i \in \{F, T\}$ to indicate that the random variable is in a specific state. For every state combination of the parent nodes $c_1, \ldots, c_n$, the CPT of the k-out-of-n

model has the following formal definition:

$$\forall c_1, \ldots, c_n \in \{F, T\}^n$$
$$P(\text{Gate} = F | c_1, \ldots, c_n) = \begin{cases} 1 & \sum_{i=1}^{n} \mathbf{1}_F(c_i) \geq k \\ 0 & \text{otherwise} \end{cases}$$
$$P(\text{Gate} = T | c_1, \ldots, c_n) = 1 - P(\text{Gate} = F | c_1, \ldots, c_n) \tag{4.4}$$

4.4.2. Infrastructure Failure

We begin with the Bayesian network model of the fault dependency graph. Perhaps it is not apparent why the fault dependency graph forms the beginning. However, due to the cause-effect semantics of Bayesian networks, it is essential to start with root causes first and then successively attach the effects, which themselves are failure causes for other components. Hence, infrastructure failures form the initial failure causes of the application. Given a scenario model $\mathcal{A}$, the first step is to build the Bayesian network for infrastructure failures.

Component Failure Model

A component $C \in \boldsymbol{C}$ fails either because of an intrinsic failure or an external fault caused by its parent components. First, we define the Bayesian network structure of a single component. This structure will then be used as a building block for the upcoming Bayesian network representation of the fault dependency graph.

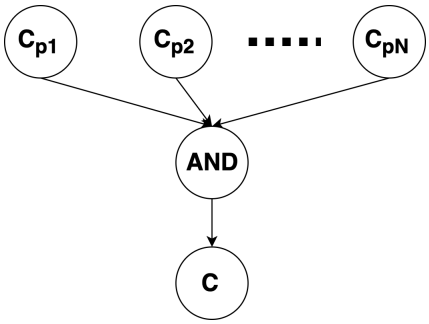


Figure 4.9.: AND fault relation between infrastructure components.

First, the procedure creates a binary random variable for every component in $\boldsymbol{C}$ with the states $\{F, T\}$, where each variable defines the probability of observing the eponymous component as faulty or available. The procedure applies to each component C the Bayesian network transformation of $FT(C)$ according to [117],

c.f. Section 4.4.1, where the fault of C is the top event, and C's parent components (from the fault dependency graph) are the base events. For example, Figure 4.9 shows the Bayesian network representation of a component C that expresses its dependability to its parent components C_{p_1} to C_{p_N} as a fault tree with one AND gate represented as a node in the structure. Hence, the CPT uses the previously introduced AND model from Equation 4.2. To also account for C's intrinsic failure probability q, we implement its CPT with a *noisy*-AND model.

$$\begin{aligned} P(C = F|\mathbf{AND} = F) &= 1 \quad P(C = T|\mathbf{AND} = F) = 0 \\ P(C = F|\mathbf{AND} = T) &= q \quad P(C = T|\mathbf{AND} = T) = 1 - q \end{aligned} \tag{4.5}$$

The noisy-AND model is the same as the AND model for $q = 0$. However, for $q > 0$, we introduce an additional probability to observe the component $C = F$ even though the AND node is observed to be in state T, i.e., all parent components are working.

Translating the Fault Dependency Graph

For a given fault dependency graph G_{FD}, we design a procedure that repeats the above approach for each component. Algorithm 1 describes the translation procedure to transform a given fault dependency graph G_{FD} into a Bayesian network. The notion $(\boldsymbol{C}, E_{\text{INF}}, FT) \leftarrow G_{\text{FD}}$ means that a structure, say G_{FD}, provides its elements to the outer scope. In the context of functional programming, this is known as pattern matching. First, Algorithm 1 creates the eponymous nodes for all components (line 4). Then it creates their corresponding Bayesian network fault tree representation defined in $FT(C)$ (line 8). Afterward, it adds the node representation of the instances to the host nodes according to a predefined deployment from D (line 15).

Applying Algorithm 1 to the login example from Section 4.3.5 leads to the preliminary Bayesian network shown in Figure 4.10. Here, without loss of generality and for the sake of readability, the AND node can be combined with the noisy AND model to one node. With this simplification, the Bayesian network corresponds in its shape to the fault dependency graph, as illustrated in Figure 4.7. Moreover, to visually assist the translation procedure, the nodes in Figure 4.10 are rearranged. All network components are on the left, and all hosts with their processes are on the right side (gray dashed box).

This preliminary Bayesian network model can already be used to infer the availability of individual service instances given the fault influenced by their sur-

Algorithm 1 Building the Bayesian network infrastructure model.

procedure CreateFailureDependencyGraph(G_{FD},D,$\boldsymbol{I}$)
  $BN = (X, E)$ with $X = \{\}$ and $E = \{\}$ ▷ Initialize Bayesian network.
  $(\boldsymbol{C}, E_{\text{INF}}, FT) \leftarrow G_{\text{FD}}$
  for $C \in \boldsymbol{C}$ **do**
    $X = X \cup C$ ▷ Create node C with binary state $\{F, T\}$
  end for
  for $C \in \boldsymbol{C} \setminus \boldsymbol{I}$ **do**
    $BN_C = FT(C)$ ▷ Create Bayesian network model of $FT(C)$ according to [117]
    for $C_{pj} \in \text{pa}(C)$ **do**
      $E = E \cup (C_{pj}, BN_{C,j})$ ▷ Add edge from C_{pj} to corresponding base event node $BN_{C,j}$
    end for
    $E = E \cup (TE(BN_C), C)$ ▷ Add edge from the top event (TE) node from BN_C to C
    add CPT to C using Eq. 4.5 with $q = P(C = F)$)
  end for
  for $I_i \in \boldsymbol{I}$ **do**
    $E = E \cup (D(I_i), I_i)$ ▷ Add edge from host node $D(I_i) = H$ to I_i
    add CPT to I_i using Eq. 4.5 with $q = P(I_i = F)$
  end for
  return BN
end procedure

rounding infrastructure.

Discussion on Circular Fault Dependencies

The Bayesian network formalism does not support cycles in general. This is why the fault dependency graph is a DAG. Its translation would lead to cycles in the corresponding Bayesian network availability model. However, under certain conditions, one might translate a cyclic fault dependency into a Bayesian network without compromising the Bayesian network formalism.

Figure 4.11a shows a fragment of a fault dependency graph where two components C_i and C_j have a cyclic fault dependency Both components C_i and C_j might also have one or more parent components, shown by the incoming edges. This fault dependency graph indicates that if C_i has a fault, then C_j and C_r fail. Symmetrically, if C_j has a fault, then also C_i and C_r fail. Two random variables cannot be parent nodes to each other. To solve this problem, helper variables can break the cycle by detouring the parent edges of the variables that form the cycle.

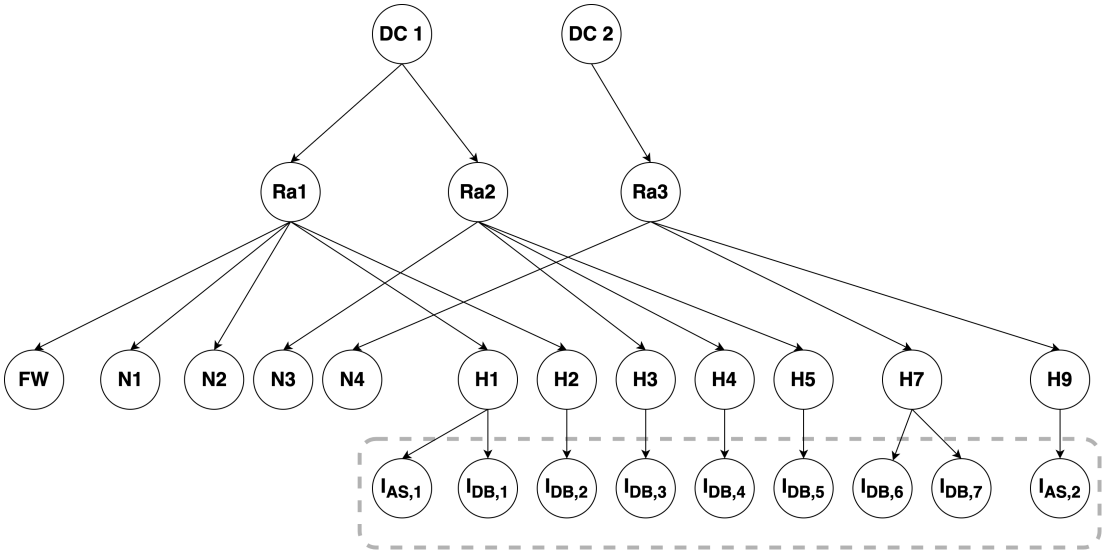
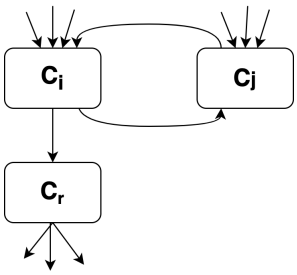


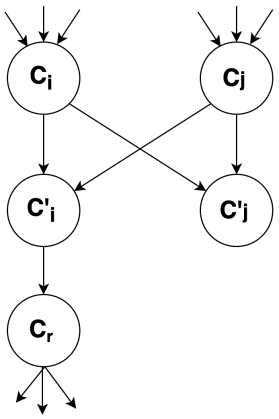
Figure 4.10.: Bayesian network infrastructure model of the login example.

Figure 4.11b shows the resulting Bayesian network, where the helper variables C_i' and C_j' break the cycle. The parent variables of C_i and C_j are connected with the original nodes, whereas the parent component that forms the cycle has an edge with the helper variable C_i' in the Bayesian network representation. However, this solution only provides the qualitative structure to handle fault dependency cycles. The next question is how to split the CPTs of C_i and C_j?

A cycle can be broken only when the causal impact of the component in the fault dependency cycle is independent of the causal impact of the remaining parent components. For example, C_i is a child component of C_j. Therefore, from C_i's point of view, C_j is one of the multiple parent components. In the acyclic fault dependency graph, the parents of a component C_i would have also been parents or ancestors of the Bayesian network. The CPT of C_i would contain some fault semantics that define exactly which combination of parent states leads to a failure. However, in the presence of a cycle, the model splits the original node into two nodes. Together, they represent the actual component C_i. However, the parent nodes that form the cycles are now connected to the helper variable C_i', resulting in two CPTs. The first CPT belongs to C_i, which defines the conditional probability distribution for the parent components with no cyclic dependencies. The second CPT belongs to C_i', which defines the conditional probability distribution for those parent components that are part of the fault dependency cycle. Technically, the helper variable, say C_i', has a CPT that provides either an OR model, where both parent variables, C_i and C_j, need to be in a working state for C_i' to be in state T, or it provides an AND model, where C_i or C_j need to be in a



(a) Direct fault dependency cycle.



(b) Structural representation to model a cycle in a Bayesian network.

Figure 4.11.: The Bayesian network solution to solve cyclic dependencies in the fault dependency graph.

working state for C_i' to work. Any other CPT definition, would either ignore the edge (C_i, C_i') or (C_j, C_i'), which ignores the cycle in the fault dependency graph.

In general, if C_i has no parents, then the transformation is valid. The variable C_i represents just the availability of its corresponding component. Hence C_i' will mirror and propagate the fault of C_i. However, when C_i has parent nodes, it depends on the fault semantic defined in the fault dependency graph for C_i given its parents. If the fault semantic of the parent nodes makes no difference between the fault influence of the parent components that belong to the cycle and those that do not, then the CPT cannot be split in two. If the underlying fault tree semantics that constitutes the boolean expression of the CPT can be separated by an AND/OR gate at the top event, then the CPT can be split, and the cycle can be broken. Consequently, the helper variables implement this last AND/OR gate as the final logical operation.

4.4.3. Channels

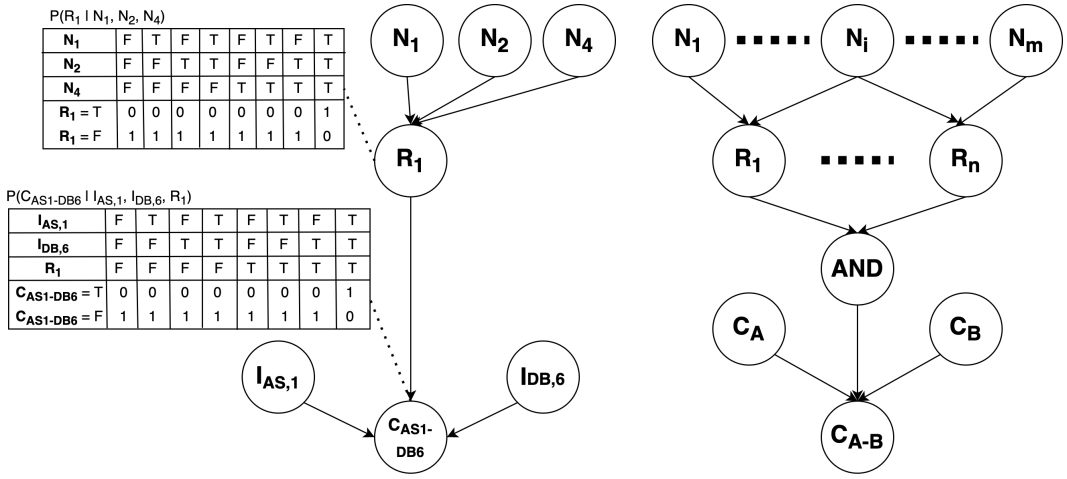
According to our availability definition, an instance is available as long it is up and reachable. The reachability aspect depends on the capability of the network to realize a communication channel between the client application and the particular instance. In the context of the network infrastructure, a channel is realized along a route of switching and routing devices. Layer three network protocols such as IP can tolerate switch failures as long there is an alternative route to detour the communication channel. Hence, from an availability perspective, the channel

is the probability that two instances can communicate in the presence of route failures. A channel is interrupted when there is no route available. A route fails when at least one network component along the path of the route fails. For example, let us examine the channel between the service instances $I_{AS,1}$ and $I_{DB,6}$ in the network graph shown in Figure 4.7. The channel consist of one route, which contains the network components N_1, N_2, and N_4. The failure modes that can impair the communication are when one of the two instances fail, or the route fails. A route fails when one or more network components along the route fail. Figure 4.12a shows the Bayesian network structure that encompass these failure modes. The node $C_{AS1-DB6}$ represents the probability of observing a channel failure between the instances $I_{AS,1}$ and $I_{DB,6}$. For readability, this section refers to the final node $C_{AS1-DB6}$, simply as a *channel node*. Its conditional probability distributions depends on the nodes of the instances and the node R_1 that represents the probability of observing a route failure. Similarly, the conditional probability distribution of R_1 depends on the component nodes that form the route. Node R_1 has a CPT that implements an OR model. Hence, we observe the route working as long as all network components are working. Similarly, the channel node $C_{AS1-DB6}$ implements the OR model to indicate that a channel works as long as both instances and routes are working.

This example can be generalized for channels that have multiple routes to their disposal. Figure 4.12b shows the Bayesian network structure that contains the channel node C_{A-B}, representing the probability of a communication failure between two channel endpoints C_A and C_B. The channel node is conditionally dependent on three nodes: an AND node and two nodes for the endpoints. The AND node represents the failure probability that no route is working by implementing the eponymous AND model as its CPT. As mentioned in the example before, the CPT of the channel node entails an OR model, defining the probability of observing a channel failure. Note, the notation C_{A-B} represents a channel and is not to be confused with C_i or C, which refers to components.

The nodes that define the failure of the endpoints, i.e., C_A and C_B, do not necessarily need to be instance nodes. They could also represent different failure causes that indirectly affect the channel. This could be the host of the client or a common endpoint of a second channel. The latter is essential to chain two channels, which is important for the replicated service model later.

Finally, nodes R_1 to R_n define the failure probabilities of routes. All route nodes implement an OR model and are conditionally dependent on the network components N_1 to N_m that are part of the appropriate route in the network



(a) Channel Model of instances $I_{AS,1}$ and $I_{DB,6}$.

(b) Bayesian network representation of a general channel.

Figure 4.12.: Bayesian network representation of a single communication channel.

graph. This model also considers shared route failures. For example, if N_i fails, route R_1 and R_n are interrupted and thus unavailable, simultaneously.

Algorithm 2 describes the construction of a channel as a procedure. Inputs are source C_{src} and destination C_{dst} components and a pair of Bayesian network nodes X_{src} and X_{dst}, which represent the failure causes of the channel's endpoints. As discussed briefly, the model distinguishes between the components for which it computes the channels and the parent nodes that provide the failure causes at the channel's endpoints. The node $AND_{src-dst}$ indicates that the AND node belongs to the channel $C_{src-dst}$ to distinguish the AND nodes between multiple channels. First, the procedure computes all routes based on the network graph at line 3. Afterward, line 4 to 8 connects the channel node with its parents nodes. Line 9 iterates over the list of routes and determines if the route has existed as a node in the Bayesian network graph yet or not. If yes, then the corresponding route node is directly added to the channel as shown in line 17. If not, then lines 11 to 13 create the new route node and connect it with its corresponding network components. The remainder of the procedure finalizes the CPT of the channel node and returns it as a reference.

Undoubtedly, the most resourceful operation is to compute all routes between the two endpoints. The number of routes can get intractably large. In this case, one might resort to simplifying the network graph or consider a limited number of routes. However, while these simplifications increase performance, it comes at the expense of a reduced model representation of the real system.

Algorithm 2 Create channel model.

1: **procedure** CREATECHANNEL(BN, G_{NET}, $C_{src} \in C_{\text{NET}}$, $C_{dst} \in C_{\text{NET}}$, $X_{src} \in X$, $X_{dst} \in X$)
2: $(X, E) \leftarrow BN$
3: *routes* := compute all routes from C_{src} to C_{dst} in G_{NET}
4: $X = X \cup C_{src-dst}$ ▷ Create channel node $C_{src-dst}$ with binary state $\{F, T\}$
5: $X = X \cup AND_{src-dst}$ ▷ Create node $AND_{src-dst}$ with binary state $\{F, T\}$
6: $E = E \cup (AND_{src-dst}, C_{src-dst})$ ▷ Add edge from node $AND_{src-dst}$ to $Ch_{src-dst}$
7: $E = E \cup (X_{src}, C_{src-dst})$
8: $E = E \cup (X_{dst}, C_{src-dst})$
9: **for** R in *routes* **do**
10: **if** $R \notin X$ **then** ▷ Check if no other route exists that represents the same path.
11: $X = X \cup R$ ▷ Create node R with binary state $\{F, T\}$
12: **for** $C \in R.components$ **do**
13: $E = E \cup (C, R)$
14: **end for**
15: add OR model to CPT of R
16: **end if**
17: $E = E \cup (R, AND_{src-dst})$
18: **end for**
19: add OR model to CPT of $C_{src-dst}$
20: add AND model to CPT of $AND_{src-dst}$
21: **return** $C_{src-dst}$
22: **end procedure**

4.4.4. Redundant Services

First, we focus on modeling one service, as defined in Definition 7. We do not consider the placement of clients or other services yet but pretend that communication starts from the service's gateways.

Redundant services have stateless instances. However, even in the context of stateless instances, a service might only be able to tolerate a certain number of instance failures or even none at all. For example, let us assume we have a compute service to process numerical data in a parallel manner. Every instance is a worker task that performs the same operations on different data sets. The results are aggregated later. For the aggregation to finish successfully, all instances must finish. Therefore, n-out-of-n instances need to be available for the service to be available. Conversely, assume we have an HTTP service that employs multiple instances for web content distribution. One instance is enough to maintain availability if the user load is sufficiently low. As a result, the HTTP service can tolerate $n - 1$ instance failures. However, suppose user-load is high, and latency requirements are part of the SLA. In that case, the service might have a threshold, which requires that at least k-out-of-n instances need to be available in order to provide a service that fulfills its service-level objectives. As a result, even for redundant services, sufficient instances need to be available to consider the service as available. Here, "sufficient" is defined by the instance path set Q, which contains all valid instance combinations necessary to regard the service as working.

The communication pattern for redundant services entails that clients directly communicate with instances through at least one gateway. Since gateways form the entry points, we create channels from every gateway to every instance. This simulates the behavior that every client can reach every instance through any gateway. So, if a gateway fails, then all channels from that gateway to every instance fail. However, the service might still be available through another gateway. Hence, we assess the reachability property for each gateway individually and infer the probability of observing a service as available through its gateways. For example, in the login scenario shown in Figure 4.7, the firewall component is the gateway to the database service. A channel starts at the firewall to every instance. According to the channel model, a channel fails when either one of the endpoints or all its routes fail. Let us assume the database implements the special case of the read-one write-all replication protocol, so no inner-instance communication is required, and the client can communicate with each instance

directly. For the read operation, the service is available as long as one channel starting at the firewall is working. For the write operation, the service is available as long all channels are working. Therefore, the Bayesian network model aims to infer the probability of observing at least one working channel for the read operation, or all channels for the write operation.

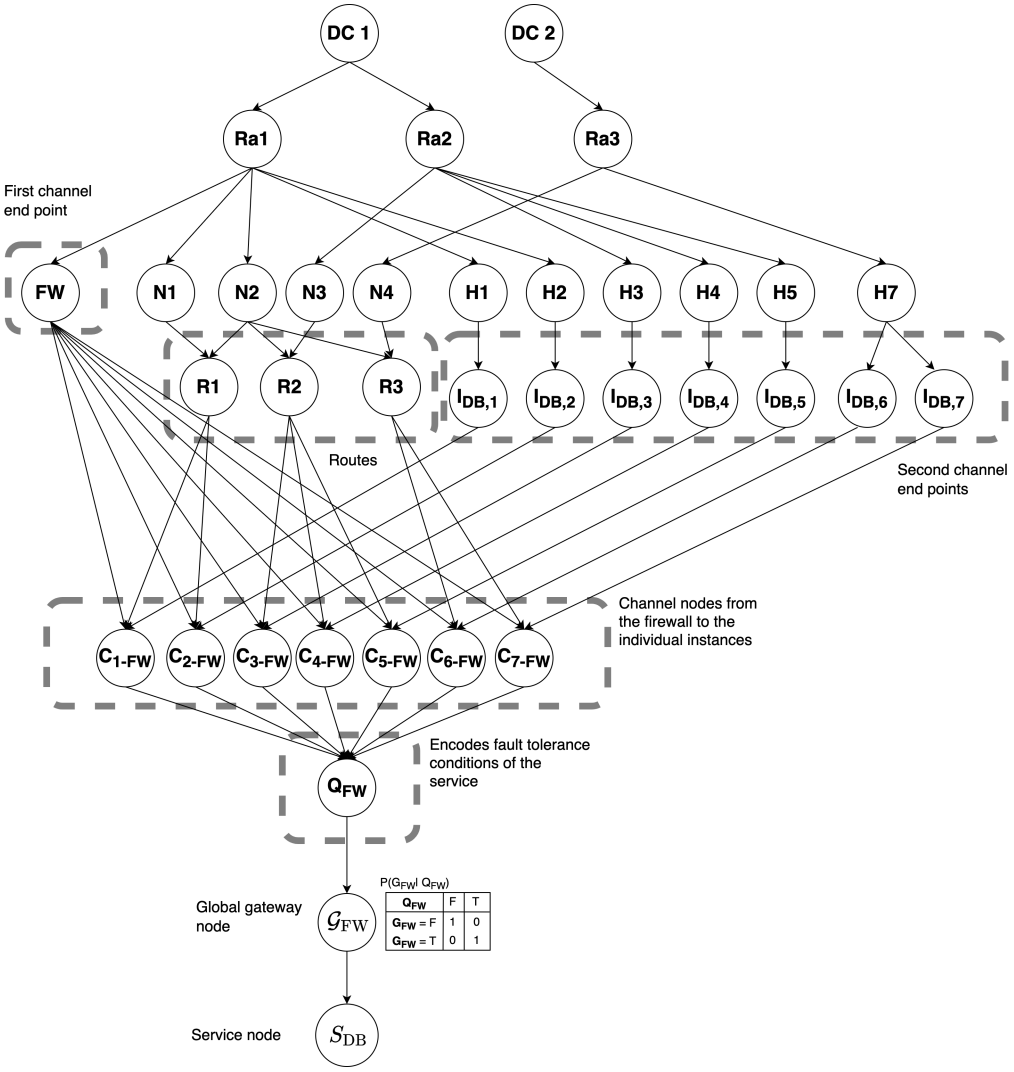


Figure 4.13.: The Bayesian network of the database service using the redundant service model.

Before we discuss how to implement the Bayesian network model in detail, let us discuss the main concepts based on an example. Figure 4.13 shows the Bayesian network model of the database service S_{DB} embedded within the Bayesian network model of the fault dependency graph. All clients communicate with the instance via the firewall (represented by node `FW`). There are three routes R_1 to R_3, which are shared by all seven channels, emphasized by the dashed box.

Each channel is connected to the firewall node and an instance. To assess the probability that at least one channel is working for the read-one operation, we need a node Q_{FW}, which is a child node to all channels of a gateway. The CPT of Q_{FW} implements an AND model to describe the probability that no instance is reachable through the firewall component. Conversely, suppose we assess the write-all operation. In that case, the CPT of Q_{FW} has to implement an OR model to infer the probability that all instances are working and reachable since all channel nodes need to be observed in a working state.

For readability reasons, the figure only contains one gateway. In the case of multiple gateways, we would have a channel from each gateway to each instance. All channels would be connected to a separate node Q_i, representing the probability of observing the service as available from the i-th gateway. In the end, Q_i implements the fault tolerance condition defined by the service parameter Q, which we will discuss in detail at the end of this subsection. For now, let us assume we assess the availability of the read-one operation where at least one channel needs to work, so each CPT of Q_i implements an AND model. Hence, we separately assess the service availability starting from each gateway. Consequently, to infer the final service availability, we connect all Q_i nodes with a *service node* S. The CPT of the S node implements an AND model so that we can infer the probability $P(S = T)$ of observing an available service through at least one gateway. For example, in Figure 4.13, where we have one gateway, the node Q_{FW} is connected with the service node S_{DB} through an intermediary node $\mathcal{G}_{FW}$, which we discuss now.

Algorithm 3 describes how to extend the previously created Bayesian network infrastructure model with the redundant service model. It introduces a new set of binary random variables $\{Q_i\}_{i=1}^{m}$, with $m = |\mathcal{G}|$, which will ultimately be used to encode the requirements posed by the service parameter Q. A node Q_i represents the probability of observing the service as available through the i-th gateway, given that sufficient instances are working. At line 11, the model creates the channel nodes for each gateway to every service instance. It takes the network component that acts as a gateway, the host of the instance as defined by its deployment, and the two nodes that represent the failure of the channel's endpoints. Line 14 introduces a global gateway node $\mathcal{G}_i$ that represents the probability of accessing sufficient instances through the i-th gateway G_i. These global *gateway nodes* are intermediaries and used to model the communication between services when building the scenario model later. Finally, Algorithm 3 finishes by introducing the final node S. This node accounts for the probability that no gateway has

Algorithm 3 Bayesian network model for redundant services.

```
1: procedure RedundantServiceModel(BN,G_NET,S)
2:     (X, E) ← BN
3:     (I, D, G, ., .) ← S
4:     X = X ∪ S                                   ▷ Create service node S
5:     m = |G|
6:     for i ∈ [1, m] do
7:         X = X ∪ Q_i    ▷ Create a set of binary random variables Q_1, ..., Q_m
8:     end for
9:     for G_i ∈ G do
10:        for I_i ∈ I do
11:            C_{G_i−I_i} := CreateChannel(BN,G_NET,G_i,D(I_i),G_i,I_i)
12:            E = E ∪ (C_{G_i−I_i}, Q_i)
13:        end for
14:        X = X ∪ G_i
15:        E = E ∪ (Q_i, G_i)                      ▷ Add edge from Q_i to G_i
16:        E = E ∪ (G_i, S)                        ▷ Add edge from G_i to S
17:    end for
18:    add AND mode to CPT of S
19: end procedure
```

sufficient working channels to communicate with the service instances S. At this point, one can compute the probability of a single service failure as the marginal $P(S = F)$ or its availability $P(S = T)$.

When we discussed the service model, we introduced the parameter Q, c.f. Definition 7, as the set of instance combinations where the service is considered to work according to specification. Moreover, we argued that for k-out-of-n voting schemes, we could use the shorthand definition $Q = (V, t)$, where V is a set of votes per instance, and t is the threshold. In the following, we discuss implementing both definitions of Q as part of the CPT belonging to every Q_i.

The main advantage of the set definition of Q is that we can define any combination of instances to express the working state of a service, but with the potential drawback of explicitly stating each combination. A specific node Q_i already addresses every instance combination as part of its conditional probability distribution through the channel nodes. A gateway has the same amount of channels as instances. Since a channel fails when its endpoints are not working, we can use the channel to represent the availability of an instance, i.e., a working channel means that the instance is reachable and working. Let us, assume $n = |pa(Q_i)|$ is the number of instances connected to Q_i. A channel node represents a random variable with the binary states $C_{i-j} = c_{i-j} \in \{F, T\}$, where we

use c_{i-j} to distinguish a specific state from its random variable C_{i-j}. For every state combination of channel nodes $c_{i-1}, \ldots, c_{i-n}$ and set Q, we define the CPT of Q_i by checking all instances that are reachable by those channels that are in state T in the combination. $P(Q_i = T|c_{i-1}, \ldots, c_{i-n})$ is 1, whenever the set of reachable instances are part of Q. Formally:

$$\begin{gathered}\forall c_{i-1}, \ldots, c_{i-n} \in \{F, T\}^n \\ P(Q_i{=}T|c_{i-1}, \ldots, c_{i-n}){=}\begin{cases}1 & (\bigcup_{j=1}^{n}\{I_j : c_{i-j} = T\}) \in Q \\ 0 & \text{otherwise}\end{cases} \\ P(Q_i{=}F|c_{i-1}, \ldots, c_{i-n}){=}\,1 - P(Q_i{=}T|c_{i-1}, \ldots, c_{i-n})\end{gathered} \tag{4.6}$$

The notation $\bigcup_{j=1}^{n}\{I_j : c_{i-j} = T\}$ refers to the union of instances I_j where their corresponding channels have the state $c_{i-j} = T$ within the given channel combination. If this set of instances exists in Q then the conditional probability distribution of $P(Q_i = T|c_{i-1}, \ldots, c_{i-n})$ is 1 for that specific combination of channel states. Conversely, $P(Q_i = F|c_{i-1}, \ldots, c_{i-n})$ is simply the counter probability of the former.

However, if $Q = (V, t)$ has the shorthand notation, with $V = (V_1, \ldots, V_n)$ containing the votes of each instance and t defining the threshold, then the CPT of Q_i has the following formal construction description:

$$\begin{gathered}\forall c_{i-1}, \ldots, c_{i-n} \in \{F, T\}^n \\ P(Q_i = T|c_{i-1}, \ldots, c_{i-n}) = \begin{cases}1 & \sum_{j=1}^{n} \mathbf{1}_T(c_{i-j})V_j \geq t \\ 0 & \text{otherwise}\end{cases} \\ P(Q_i = F|c_{i-1}, \ldots, c_{i-n}) = 1 - P(Q_i = T|c_{i-1}, \ldots, c_{i-n})\end{gathered} \tag{4.7}$$

Again, $\mathbf{1}_T$ is the indicator function that returns one when the variable has the value T. For every state combination $c_{i-1}, \ldots, c_{i-m}$, the model builds the weighted sum of those channels with the state T in that specific combination and checks if the result is above the threshold. If the weighted sum of the accessible votes are above the threshold, then we set the probability distribution
$P(Q_i = T|c_{i-1}, \ldots, c_{i-n}) = 1$, otherwise zero.

If we describe a k-out-of-n voting scheme where each instance has one vote, then Equation 4.7 can be substituted with the k-out-of-n definition from Equation 4.4, acting on state T instead of state F.

$$\forall c_{i-1}, \ldots, c_{i-n} \in \{F, T\}^n$$

$$P(\mathrm{Q_i} = T | c_{i-1}, \ldots, c_{i-n}) = \begin{cases} 1 & \sum_{j=1}^{n} \mathbf{1}_T(c_{i-j}) \geq k \\ 0 & \text{otherwise} \end{cases}$$

$$P(\mathrm{Q_i} = F | c_{i-1}, \ldots, c_{i-n}) = 1 - P(\mathrm{Q_i} = T | c_{i-1}, \ldots, c_{i-n}) \tag{4.8}$$

In special cases, we can use the AND model for the 1-out-of-n voting scheme, which models the probability that all channels might fail. Also, we can use the OR model for the n-out-of-n voting scheme to define that the service is unavailable when at least one channel fails.

4.4.5. Replicated Services

Next, we discuss the replicated service model. This model is similar to the previous model; however, it has a different communication pattern. Again, gateways form the entry points to the service. A replicated service is available as long a client can reach an instance that, in turn, can reach sufficient other instances. The first condition is that a client can reach at least one instance, which is the same model as for the redundant service model. Hence, the Bayesian network model will have a set of channel nodes per gateway to assess the likelihood of reaching at least one instance. This assessment must be combined with a second assessment, where the model includes the likelihood of reaching sufficient remaining instances starting from the first initiating instance. Consequently, the Bayesian network will contain an additional set of channels per instance to every other instance, behaving as if every instance is a potential gateway to reach the remaining instances. To exemplify this communication pattern and the replicated service model in general, we analyze the Bayesian network model of the database service from the login scenario shown in Figure 4.7.

Figure 4.14 emphasizes the main structural aspects of the Bayesian network of the database service. The top of the Bayesian network represents the fault dependency graph. Again, the gateway component, i.e., the entry point, is the firewall component represented by node `FW`. Due to the cause-effect modeling paradigm of the Bayesian network formalism, we build the network in a top-down fashion so that "down" is the service node that represents the probability of observing the service as available, whereas "top" is the root cause, such as components failures. We need to assess communication failures once we get past

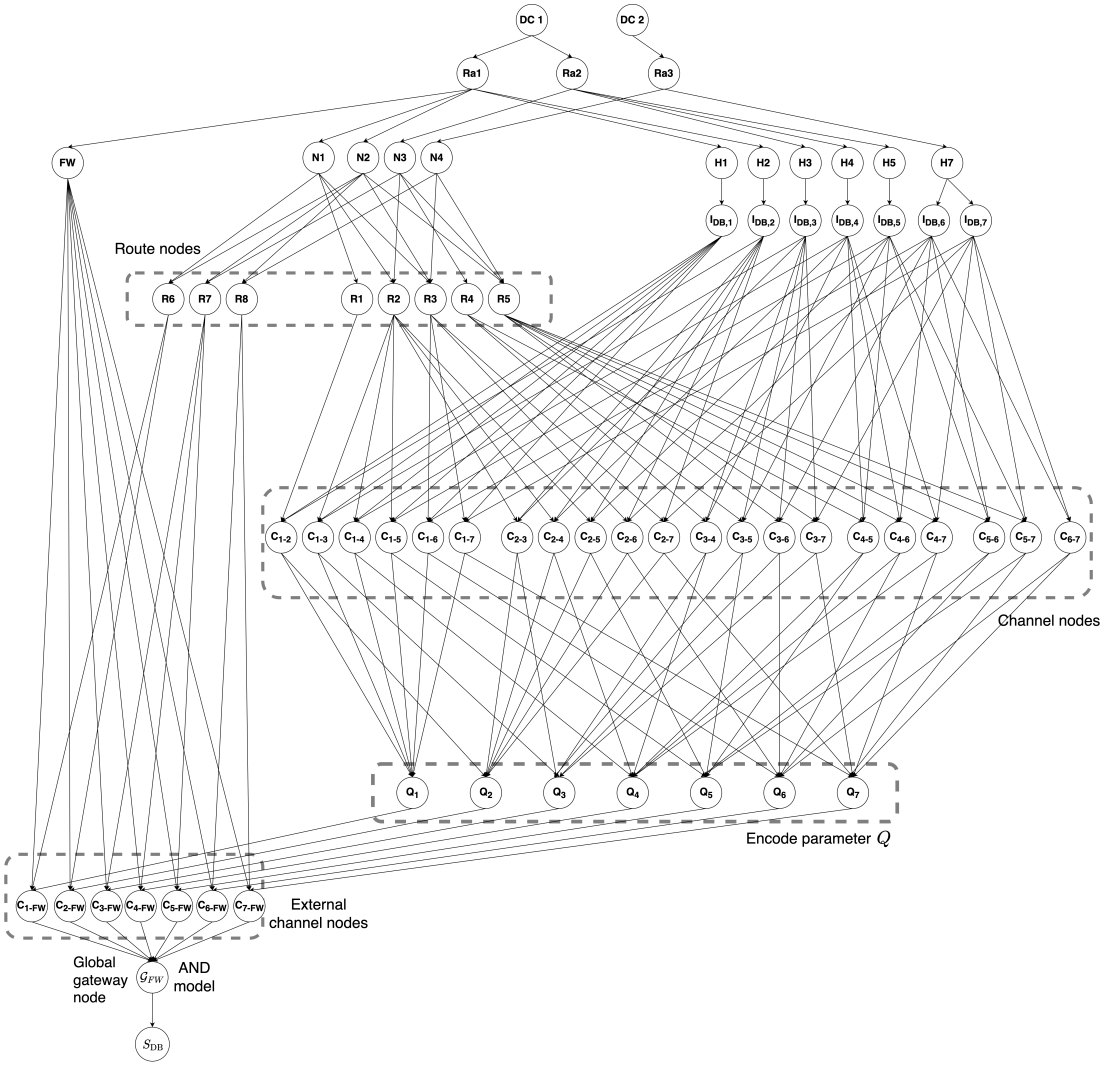


Figure 4.14.: The Bayesian network of the database service using the replicated service model.

component failures. The root cause of communication failure are route failures. For example, the network graph gives rise to eight routes that are shared by all channels. The next higher cause of a communication failures is that instances cannot communicate with each other, denoted by the dashed box in the middle, showing the channel nodes. Here, we have a channel for every instance to every other instance. Afterward, the next level is that instances cannot communicate with sufficient other instances. Here, we introduce again the state nodes Q_i, which have a conditional probability distribution that encodes all state combinations of reachable instances for which the service is considered working. So, we have a node for every instance.

Consequently, the Bayesian network is the same as the redundant service model

so far, but the instances act as gateways themselves in this model. So, every node Q_i of the i-th instance encodes whether or not an instance can communicate with sufficient other instances. Next, we need to assess that we can communicate with at least one such instance, starting from our gateway. Instead of referring to the instance nodes as the opposing endpoints directly, we use the nodes Q_1 to Q_7 as representatives of the instances instead to account for the additional availability aspect that they can communicate with sufficient other instances. We call this second set of channels *external channels*.

Algorithm 4 Bayesian network infrastructure model for replicated services.

1: **procedure** ReplicatedServiceModel(BN,G_{NET},S)
2: $(X, E) \leftarrow BN$
3: $(I, D, \mathcal{G}, \cdot, \cdot) \leftarrow S$
4: $X = X \cup S$ ▷ Create service node S
5: $n = |I|$
6: **for** $i \in [1, n]$ **do**
7: $X = X \cup Q_i$ ▷ Create a set of binary random variables $Q_1, \ldots, Q_m$
8: **end for**
9: **for** (I_i, I_j) in $I \times I$ **do** ▷ First Step
10: **if** $C_{I_i - I_j} \notin X$ and $C_{I_j - I_i} \notin X$ **then** ▷ Only consider a channel between two processes once
11: $C_{I_i - I_j}$:= CreateChannel(BN,G_{NET},$D(I_i)$,$D(I_j)$,I_i,I_j)
12: $E = E \cup (C_{I_i - I_j}, Q_i)$ ▷ Add edge to Q_i
13: $E = E \cup (C_{I_i - I_j}, Q_j)$ ▷ Add edge to Q_j
14: **end if**
15: **end for**
16: **for** $\mathcal{G}_i \in \mathcal{G}$ **do** ▷ Second Step
17: $X = X \cup G_i$
18: **for** j=1; j < n; j++ **do**
19: $C_{\mathcal{G}_i - P_j}$:= CreateChannel(BN,G_{NET},$\mathcal{G}_i$,$D(I_j)$,$\mathcal{G}_i$,Q_j)
20: $E = E \cup (C_{\mathcal{G}_i - I_j}, \mathcal{G}_i)$ ▷ Add edge from channel node to S
21: **end for**
22: add AND model to CPT of $\mathcal{G}_i$
23: $E = E \cup (\mathcal{G}_i, S)$ ▷ Add edge from $\mathcal{G}_i$ to S
24: **end for**
25: add AND model to CPT of S
26: **end procedure**

Algorithm 4 implements the Bayesian network representation of a replicates service. Every state node Q_1 to Q_n is a child node of $n-1$ channel nodes (line 12 and 13). Next, the procedure builds a channel node for every gateway $\mathcal{G}_i$ to every instance I_i by using Q_i as failure cause (line 19). Instead of directly addressing the failure probability of an instance, the model uses Q_i to represent the process

I_i. Hence, in the likelihood of a network partitioning failure, Q_i would contain the probability that I_i can still access sufficient instances in its partition.

All channel nodes between gateways and instances are connected to a gateway node $\mathcal{G}_i$, representing the probability of successfully initiating a request through the i-th gateway. Hence, the CPT of every node $\mathcal{G}_i$ implements an AND model. Finally, node S accounts for the failure probability that no client can access the service through any gateway (line 25). Hence, one can now predict the availability of the service by computing the marginalization $P(S = T)$.

However, one question remains. How do we define the CPT of the state nodes Q_1 to Q_n? Since the sub-graph that contains this nodes is the same graph from the redundant service model, but using instances as gateways instead, the CPTs of every Q_i are built the same way when using the set definition of the service parameter Q. However, there is one detail to consider. Since the instance is considered as a gateway, both channel endpoints must be considered when building the set of reachable instances in Equation 4.6. For the general case, when using the shorthand notation $Q = (V, t)$, with $V = (V_1, \ldots, V_n)$ containing the votes of each instance and t defining the threshold, the CPT of Q_i has the vote of the i-th instance implicitly, since it initiates the communication to the remaining instances. Hence, we need to reduce the threshold by the individual vote of the i-th instance when building the CPT of Q_i. This leads to the following definition:

$$\forall c_{i-1}, \ldots, c_{i-n} \in \{F, T\}^n$$

$$P(Q_i = T | c_{i-1}, \ldots, c_{i-n}) = \begin{cases} 1 & \sum_{j=1}^{n} \mathbf{1}_T(c_{i-j}) V_j \geq \mathbf{t} - \mathbf{V_i} \\ 0 & \text{otherwise} \end{cases}$$

$$P(Q_i = F | c_{i-1}, \ldots, c_{i-n}) = 1 - P(Q_i = T | c_{i-1}, \ldots, c_{i-n}) \tag{4.9}$$

Lastly, if all instances have unit votes, and the replicated system uses a classical k-out-of-n voting scheme, e.g., majority sets where at least $k = \lfloor \frac{n}{2} \rfloor + 1$ instances need to be available, we can use Equation 4.8 without modification. Again, since the initiating instance acts like a gateway, it already contributes with its vote when available. Every instance has $n - 1$ channels to the reaming instances, so when the replicated service uses a voting scheme, any reachable instance needs to have at least (k-1)-out-of-(n-1) working channels. As a result, we can use Equation 4.8 to implement the CPT with a k'-out-of-n' model for every Q_i, with $k' = k - 1$ and $n' = n - 1$.

4.4.6. Scenario Model

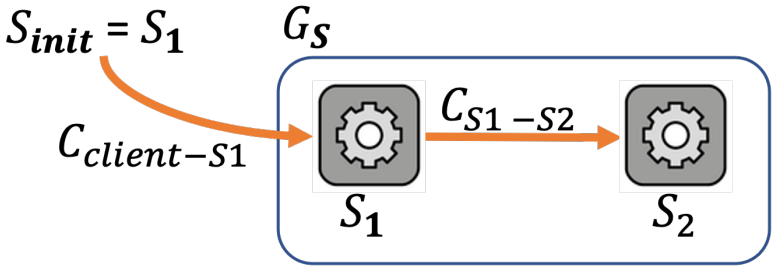
So far, we have discussed how to model a single redundant or replicated service. The next step is to fully extend the availability model to support the scenario notation provided in Definition 8. To recap, an application scenario is a structure $\mathcal{A}$ with the following elements:

$$(\boldsymbol{C}, \boldsymbol{S}, G_{\text{FD}}, G_{\text{NET}}, G_{\boldsymbol{S}}, \mathcal{G}_{\text{entry}}, S_{\text{init}}).$$

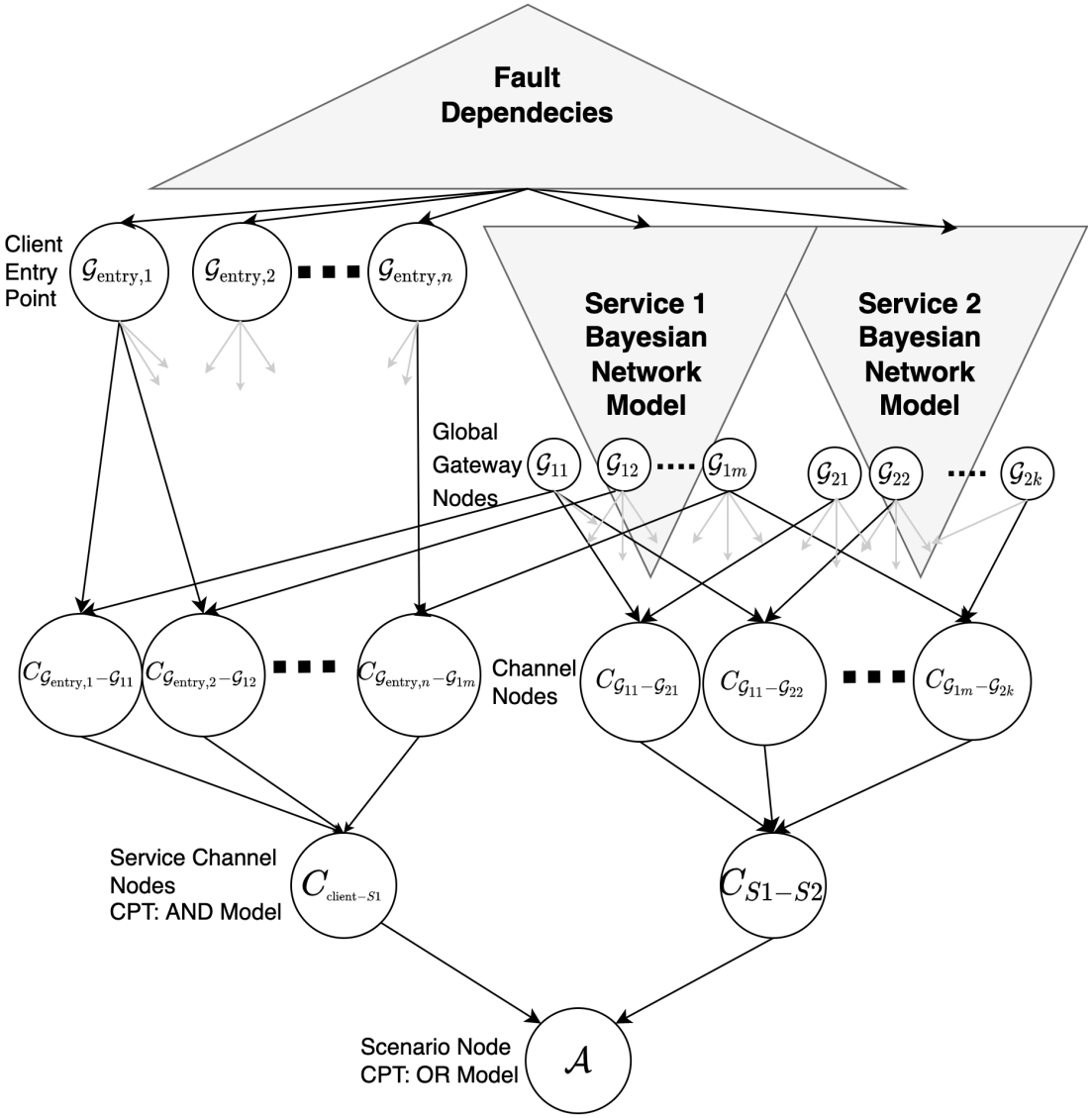
Until this point, we did not consider the set of services $\boldsymbol{S}$, G_{FD}, the service topology $G_{\boldsymbol{S}}$, the set of client entry points, i.e. gateways, $\mathcal{G}_{\text{entry}}$, and the service that is first initiated by the client S_{init}.

First, we start off with an example to discuss the main modeling concepts. Figure 4.15a shows the service topology of a scenario, consisting of two services S_1 and S_2. In this scenario model, S_1 is first contacted by the client; then, a message is passed from S_1 to S_2. The communication between clients and S_1 uses the entry points defined by the components in the gateway set $\mathcal{G}_{\text{entry}}$. A scenario is available as long all services are reachable and working. This depends in large on the communication defined in the service topology. In our example, the client needs to reach S_1 first. The availability model needs to account for the probability of reaching the gateways from S_1, starting from $\mathcal{G}_{\text{entry}}$. This probability is the likelihood of observing at least one working channel from some component in $\mathcal{G}_{\text{entry}}$ to a component in the gateway set from S_1. The cloud might provide multiple entry points for clients to enter the network. Consequently, we need to account for every channel from each component in $\mathcal{G}_{\text{entry}}$ to every other component in the gateway set from S_1. We define this aggregation of multiple channels from client to service or from service to service as a *service channel*. So, the resulting communication channels from $\mathcal{G}_{\text{entry}}$ to the gateways from S_1 are aggregated as the service channel $C_{\text{client}-S_1}$. Similarly, all channels from the gateways S_1 to S_2 are aggregated as the service channel $C_{S_1-S_2}$.

Without going into further detail, let us first assume that a service channel has the same semantics as an ordinary channel. It is *up* as long the connected endpoints are working. A scenario is available when all service channels are working. We introduced the global gateways nodes in our Bayesian network model, c.f. Figure 4.13 and 4.14, to represent the probability of observing a service as available through a specific gateway. As a result, using these global gateway nodes, we model service availability from a gateway perspective. It is sufficient only to regard channels between gateways and use the global gateway



(a) Service topology of a scenario with two services.



(b) The Bayesian network of the scenario model.

Figure 4.15.: Transforming a scenario into a Bayesian network availability model.

nodes as failure causes of the channel's endpoints instead.

The final Bayesian network is shown in Figure 4.15b. It summarizes the pre-

viously introduced Bayesian network model as gray areas, only displaying the service channels. First, we have the binary random variable $\mathcal{A}$, which represents the probability of observing a given scenario as available. The CPT of node $\mathcal{A}$ implements an OR model to define the conditional probability that at least one service channel is not working, represented by the nodes $C_{\text{client}-S_1}$ and $C_{S_1-S_2}$. These service channels are higher-order channel representations, so to speak, since they have conditional dependencies solely on channel nodes. The CPT of the service node $C_{\text{client}-S_1}$ implements a AND model, representing the conditional probability of all corresponding channels. Similarly, for the service channel $C_{S_1-S_2}$, which depends on the channel nodes of the gateways of S_1 and S_2, and is conditional dependent on the corresponding global gateway nodes. The global gateway nodes are part of the Bayesian network sub-structure that contains the service availability model, which is part of the larger structure containing the fault dependency graph.

Based on the example shown in Figure 4.15, we can design a procedure to build the Bayesian network for an arbitrary scenario. Algorithm 5 finalizes the construction of this Bayesian network model. It brings together all previously discussed procedures. The routine consists of three blocks. The first block builds the base structure that includes the fault dependency graph and all services (line 2 to 14). The second block includes the channel nodes that consider the reachability of the first service (up to line 25). The last block includes the service channels to model the service graph $G_{\boldsymbol{S}}$ (line 39). Finally, one can infer a scenario's availability by inferring the marginalization $P(\mathcal{A} = T)$.

4.5. Evaluation and Discussion

For this evaluation, we will use two families of scenarios. The first family of scenarios implements a multi-tier service architecture and the second family of scenarios implement a fully-connected service topology. We perform experiments based on these two scenario models to verify if the Bayesian network model implements the system assumptions correctly. We will do this by comparing it to a fault tree model that implements the same system assumptions, using a different mathematical formalism, namely binary decision trees, and check if they compute the same availability value. Afterward, we focus on performance, where we analyze what the main influencing parameters are.

Algorithm 5 Building the application scenario model

procedure CreateApplicationModel($\mathcal{A}$)
  $(\boldsymbol{C}, \boldsymbol{S}, G_{\text{FD}}, G_{\text{NET}}, G_{\boldsymbol{S}}, \mathcal{G}_{\text{entry}}, S_{\text{init}}) \leftarrow \mathcal{A}$
  $D := \bigcup_{S \in \boldsymbol{S}} D_S$ ▷ Gather all deployment functions
  $\boldsymbol{I}$:= getAllInstances($\boldsymbol{C}$)
  BN := CreateFailureDependencyGraph(G_{FD},D,$\boldsymbol{I}$)
  $(X, E) \leftarrow BN$
  for $\mathcal{S} \in \boldsymbol{S}$ **do** ▷ Include individual services
    $(\cdot, \cdot, \cdot, \cdot, \cdot, c) \leftarrow S$
    if c **then**
      ReplicatedServiceModel(BN,G_{NET},S)
    else
      RedundantServiceModel(BN,G_{NET},S)
    end if
  end for
  $X = X \cup \mathcal{A}$ ▷ Create application node
  $X = X \cup C_{\text{client}-S_{\text{init}}}$ ▷ Create client channel node
  for $\mathcal{G}_{\text{entry},i} \in \mathcal{G}_{\text{entry}}$ **do**
    $(\cdot, \cdot, \mathcal{G}, \cdot, \cdot) \leftarrow S_{\text{init}}$
    for $\mathcal{G}_j \in \mathcal{G}$ **do**
      $C_{\mathcal{G}_{\text{entry},i}-\mathcal{G}_j}$:= CreateChannel(BN,G_{NET},$\mathcal{G}_{\text{entry},i}$,$G_j$,$\mathcal{G}_{\text{entry},i}$,$G_j$)
    end for
    $E = E \cup (C_{\mathcal{G}_{\text{entry},i}-\mathcal{G}_j}, C_{\text{client}-S_{\text{init}}})$
  end for
  add AND model to CPT of $C_{\text{client}-S_{\text{init}}}$
  $E = E \cup (C_{\text{client}-S_{\text{init}}}, \mathcal{A})$
  $(\cdot, E_{\text{SERVICE}}) \leftarrow G_{\boldsymbol{S}}$
  for $(S_{src}, S_{dst}) \in E_{\text{SERVICE}}$ **do** ▷ Include service graph
    $(\cdot, \cdot, \mathcal{G}_{src}, \cdot, \cdot) \leftarrow S_{src}$
    $(\cdot, \cdot, \mathcal{G}_{dst}, \cdot, \cdot) \leftarrow S_{dst}$
    $X = X \cup C_{src-dst}$
    for $\mathcal{G}_i \in \mathcal{G}_{src}$ **do**
      for $\mathcal{G}_j \in \mathcal{G}_{dst}$ **do**
        $C_{\mathcal{G}_i-\mathcal{G}_j}$:= CreateChannel(BN,G_{NET},$\mathcal{G}_i$,$\mathcal{G}_j$,$\mathcal{G}_i$,$\mathcal{G}_j$)
        $E = E \cup (C_{\mathcal{G}_i-\mathcal{G}_j}, C_{src-dst})$
      end for
    end for
    add AND model to CPT of $C_{src-dst}$
    $E = E \cup (C_{src-dst}, \mathcal{A})$
  end for
  add OR model to CPT of $\mathcal{A}$
  return $\mathcal{A}$
end procedure

4.5.1. Scenario Setup

The complexity of a scenario is a potential influencing factor with regard to performance and the availability outcome. Hence, we will focus on two different scenario families with different degrees of structural complexity with regard to their service topology. Three dimensions constitute the complexity of a scenario. First, the number of services, second, the service topology, and last, the size of the service solution space when refining the model. The latter will receive a dedicated chapter and will not be further discussed here. This section solely focuses on the structural characteristics of a scenario.

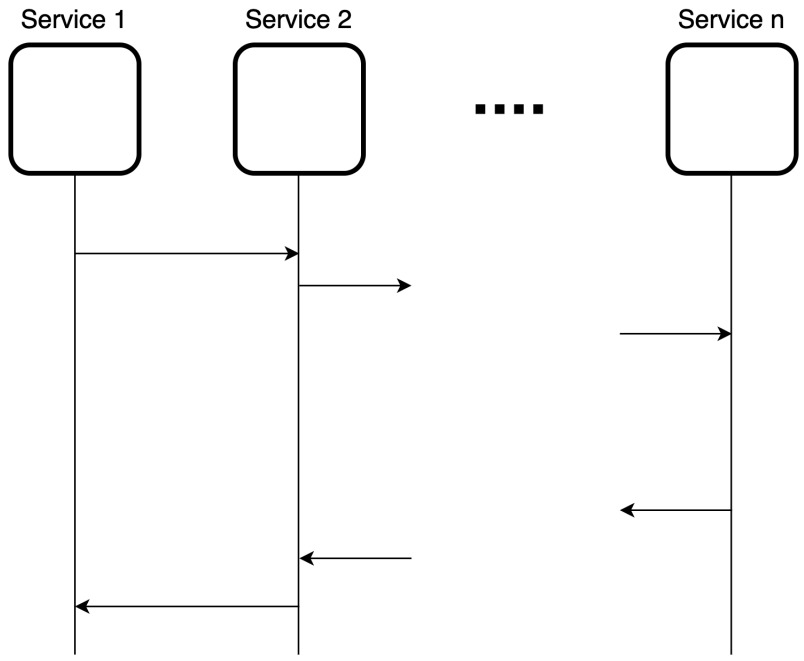


Figure 4.16.: A scenario with n services forming a multi-tier application.

This work considers two service typologies that cover a variety of common scenario cases. The first scenario implements a generalization of the well-known multi-tier service architecture. Figure 4.16 shows a Clams scenario as a message sequence between n services, where the first service initiates a new request to the second service, the second service to the third service until the last service is reached. Afterward, the last service returns its response to the previous service until the first service receives its response from the second service. This scenario has $2n$ messages in total. Hence, the number of messages grows linearly with the number of services. If the number of messages grows sub-linearly, then some components would not have in- and outgoing messages, meaning they are not used and can be ignored in the scenario. Consequently, the multi-tier architecture is

the most basic and non-trivial model. As a result, the corresponding service topology is a serial graph.

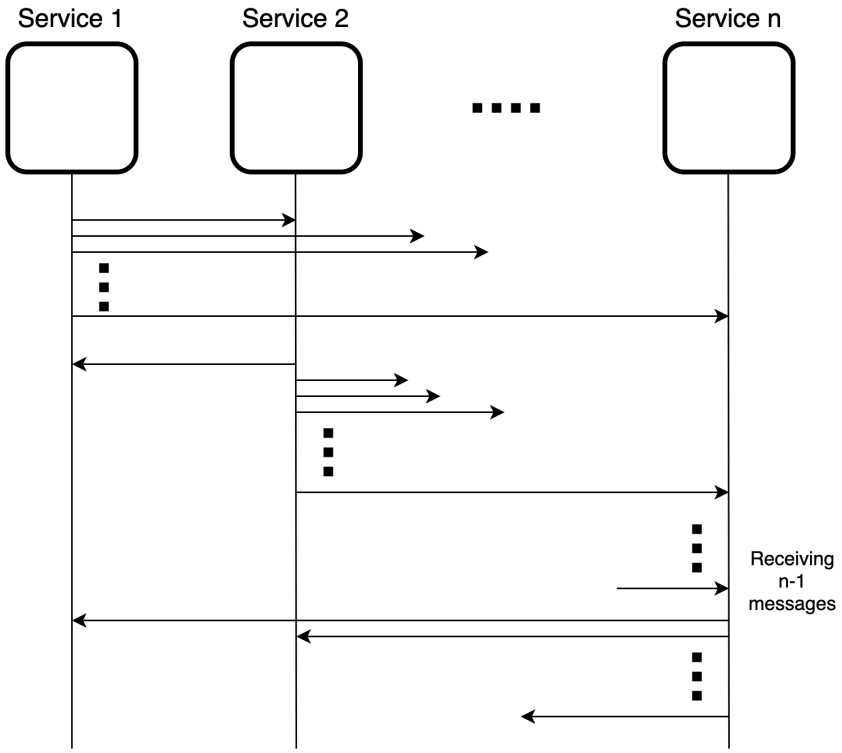


Figure 4.17.: A scenario where every service performs a broadcast to every other service.

At the other end of the complexity spectrum, the second Clams scenario implements an architecture where every service has to communicate with every other service, forming a fully connected communication graph. Figure 4.17 depicts the corresponding message exchange. The first service broadcasts messages to the remaining services. Afterward, the second service sends its messages to the other services until the last service receives its $n-1$ messages and then sends its responses back. The number of messages is $n \times (n-1)$. This is a hypothetical scenario; however, the resulting service topology is a fully connected graph. Thus, every other service topology is a sub-graph of this model.

In summary, these two scenarios represent two completely different structural aspects, which are important in determining what parameters influence the availability prediction performance. The first model is apt to evaluate the influence of the increasing number of components. In contrast, the second scenario offers a proper evaluation of the influence of the service connectivity density.

4.5.2. Model Verification

First, we focus on whether the Bayesian network availability model correctly implements and solves models based on the defined system assumptions. Trivedi et al. [19] provide a list of possible verification techniques, from which two approaches are suitable for verifying the Bayesian network model. The first approach is the *logical flow check*, where we assess if the results of the model are reasonable for a given input. The second approach is to compare the Bayesian network model with an alternative model, using a different mathematical formalism that implements the same system as described by the system model. Here, we will use *fault trees with repeated events* (FTRE) as an alternative model. All experiments will use the SCRAM risk analysis tool (ver. 0.16.2) for assessing FTREs. SCRAM uses binary decision trees for exact inference and Min-Cut-Upper-Bound (MCUB) for approximate computation [119]. Strictly speaking, (static) fault trees cannot reuse the same input or base events multiple times in a tree structure, e.g., having the same input event connected at two different gates. However, FTRE extends the static fault tree notation by allowing precisely that. Since, in our case, failure events can influence multiple components simultaneously, we have to use the FTRE formalism. FTREs are essentially fault trees; hence, we will refer to them as fault trees for the remainder of this chapter for readability reasons.

This evaluation performs an empirical verification by comparing the fault tree results with the results of the Bayesian network model for the multi-tier and fully-connect service topologies. The evaluation uses the *simple* cloud infrastructure shown in Figure 4.7 and a *large* cloud infrastructure model with 440 components as a basis for the fault dependency graph. The large infrastructure model consists of three data centers with 40 hosts each, using a random network topology with 20 network components to connect hosts and data centers. Moreover, each data center has 100 additional infrastructure components influencing hosts and network components. Evaluations that use the simple infrastructure examples will use the failure probabilities shown in Figure 4.3, whereas evaluations with the large infrastructure will use similar availability values as encountered in the industry.

Cloud providers usually do not disclose insights on actual component dependencies and availability values due to security or intellectual property reasons. To roughly infer what availability values are standard in industry, this work uses values based on the availability promises published in the SLA statements for

the virtual machine and storage offering of the three major cloud providers. The Google Cloud aims to provide an uptime of 99.95% per month for the Google App Engine and the blob storage service [120, 121]. AWS promises 99.99% for instances running in their Elastic Compute Cloud (EC2) [122] and 99.90% for their simple storage service (S3) [123]. Azure aims to provide an uptime of 99.90% for its StorSimple service [124], and an uptime of 99.90% for its virtual machine instances with standard solid states drives, and 95% with hard disk drives, respectively [125]. So, major cloud providers aim to offer a service uptime in the order of "three-nines" (99.90% to 99.95%) for their basic platform services. However, in real life, the Google App Engine service had accumulated a downtime of about 17 hours in 2021, which resulted in an average availability of 99.8%. The lowest availability was in Mai, 2021, when the service encountered a downtime of 10 hours, resulting in an availability of about 98.6% for that month [4].

Since a cloud service consists of more than one instance and infrastructure component, it is reasonable to expect that the overall availability of the infrastructure is higher than the resulting service availability. Throughout this work, we will use the beta distribution as a source to sample availability values at random since the distribution has a suitable value interval $[0, 1]$. The components of the large infrastructure will have an availability value sampled from a beta distribution with $C \sim \text{Beta}(10000, 1)$, which results in an average availability of "four-nines" 99.99%, which are higher than the SLA availability values. The standard deviation is 0.009%; hence, sampling from this beta distribution will also result in availability values commonly used to model high availability components.

We built a translation procedure that translates the high-level model into an FTRE model, implementing the same system failure assumptions used for the Bayesian model. In general, the procedure is similar to the algorithm shown in Listing 5, but instead of Bayesian network nodes, it uses fault tree gates. The changes are straight forward. The full implementation is available as open source in OpenClams[1]. Nevertheless, there is one modeling restriction. Due to the voting gate semantics in fault trees, we can only model voting-based replication schemes, i.e., no weighted voting schemes are possible. With this limitation in mind, this verification aims to show that the results of both models have no significant difference in the performed experiments. This should be true for the exact and the approximate computation methods. All experiments were performed on a 64-bit machine with 64 Inter(R) Xenon(R) CPU E7-4850 v4 at 2.10 GHz and 1TB of main memory, running Arch Linux 5.13.12 with GCC 11.1.0, Python 3.9.6,

[1] https://github.com/openclams/bn-availability-model

and Numpy 1.20.3. If not stated otherwise, inference in Bayesian networks is performed with approximate inference using the forward sampling method [34]. In some cases, experiments will also use exact inference. Exact inference will be performed with the Lauritzen-Spiegelhalter Algorithm [126] implemented in the gRain 1.3.2 package [127, 128].

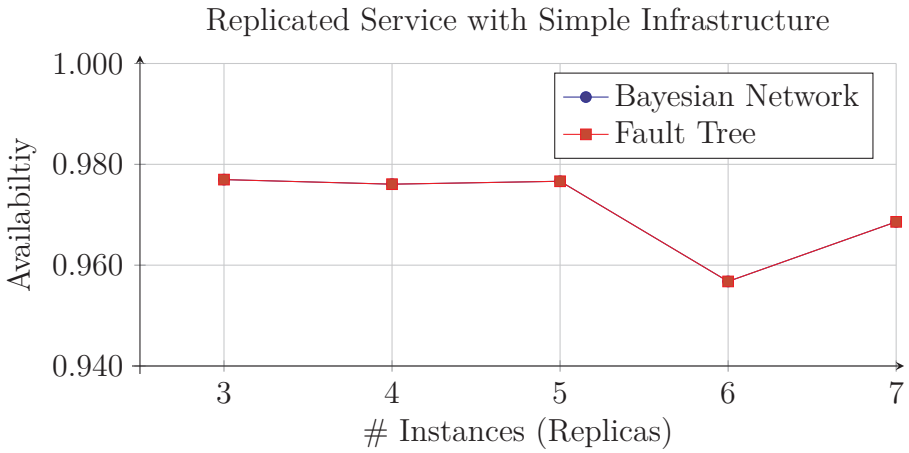


Figure 4.18.: Comparing the availability of a replicated service with exact inference methods using the simple infrastructure model.

We begin by comparing the fault tree model with the Bayesian network model of a replicated service, using the simple infrastructure model for the fault dependency graph. Figure 4.18 shows the availability results for an increasing number of instances. Here, the service is available as long as the majority of instances are available. Instances are deployed in round-robin, starting from the first host. We will generally focus on replicated services since they form the most complex model instances, containing the redundant service model as a sub-problem. The experiments use exact inference for the Bayesian network and fault tree models. However, availability assessments were only possible for at most seven instances; afterward, exact inference for the Bayesian network model became infeasible due to memory limitations. A detailed discussion on performance follows at the end of this section. Note that the results of the fault tree model overlap the data points of the Bayesian network model in the plot, which is why only the data points of the fault tree model are visible. The largest absolute error is 3×10^{-8}, which can be regarded as an insignificant difference between the prediction results since high availability is commonly in the magnitude of 10^{-5}. When analyzing the plot, one might notice that the model for six instances exhibits the lowest availability, contrary to the general assumption that a higher replication degree increases availability. The reason for this effect is the influence of the common

cause failures posed by the potential faults of the infrastructure. All six instances are placed in the first data center, requiring at least four instances for an available service. This means that both racks and their corresponding network switches need to work for the service to be available. In contrast, when a service has five instances, it is sufficient if the first or second rack and their network switches are working. Thus, the service with five instances can tolerate more infrastructure faults, increasing its availability, resulting in a higher availability than for a service with six instances.

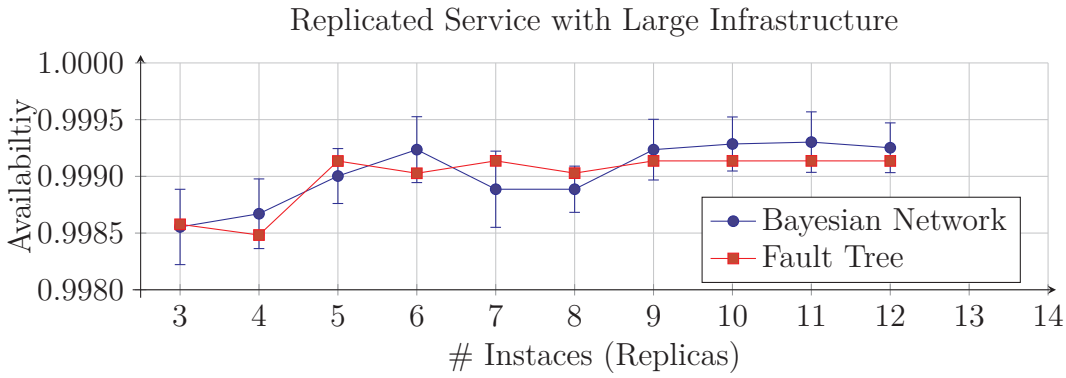


Figure 4.19.: Comparing the availability of a replicated service with approximate inference methods using the large infrastructure model.

Next, we perform the same experiments using the large infrastructure model. Figure 4.19 shows the resulting availability predictions for an increasing number of instances, using approximate computation methods for both models, including confidence intervals. Predictions were only possible for up to 13 instances with the fault tree model; afterward, assessing the fault tree model became infeasible with SCRAM, due to an exponential increase in run-time. We repeated each experiment 20 times and computed its 95% confidence interval with the help of the Student's t-distribution since the result of approximate inference in Bayesian networks is non-deterministic and varies with every execution. However, approximate computation in fault trees is deterministic, so the plot does not show confidence intervals for the fault trees. When comparing the approximate inference results, it becomes clear that all data points of the fault tree experiments are within the 95% confidence intervals of the corresponding Bayesian network experiments. Hence, it can be stated with 95% confidence that there is no significant difference between the outcomes of the Bayesian network model and the fault tree model for these experiments. Moreover, when analyzing the availability progression in Figure 4.19, one might notice that availability increases until

it converges towards 99.92%. Although the current instance deployment did not utilize all hosts of the infrastructure, we do expect that availability does not converge arbitrarily near 100% by just adding sufficient replicas. The benefits of replication will eventually fade due to the influence of common cause failures. This behavior follows the logical flow that, at some point, common cause failures will balance out the benefits of replication.

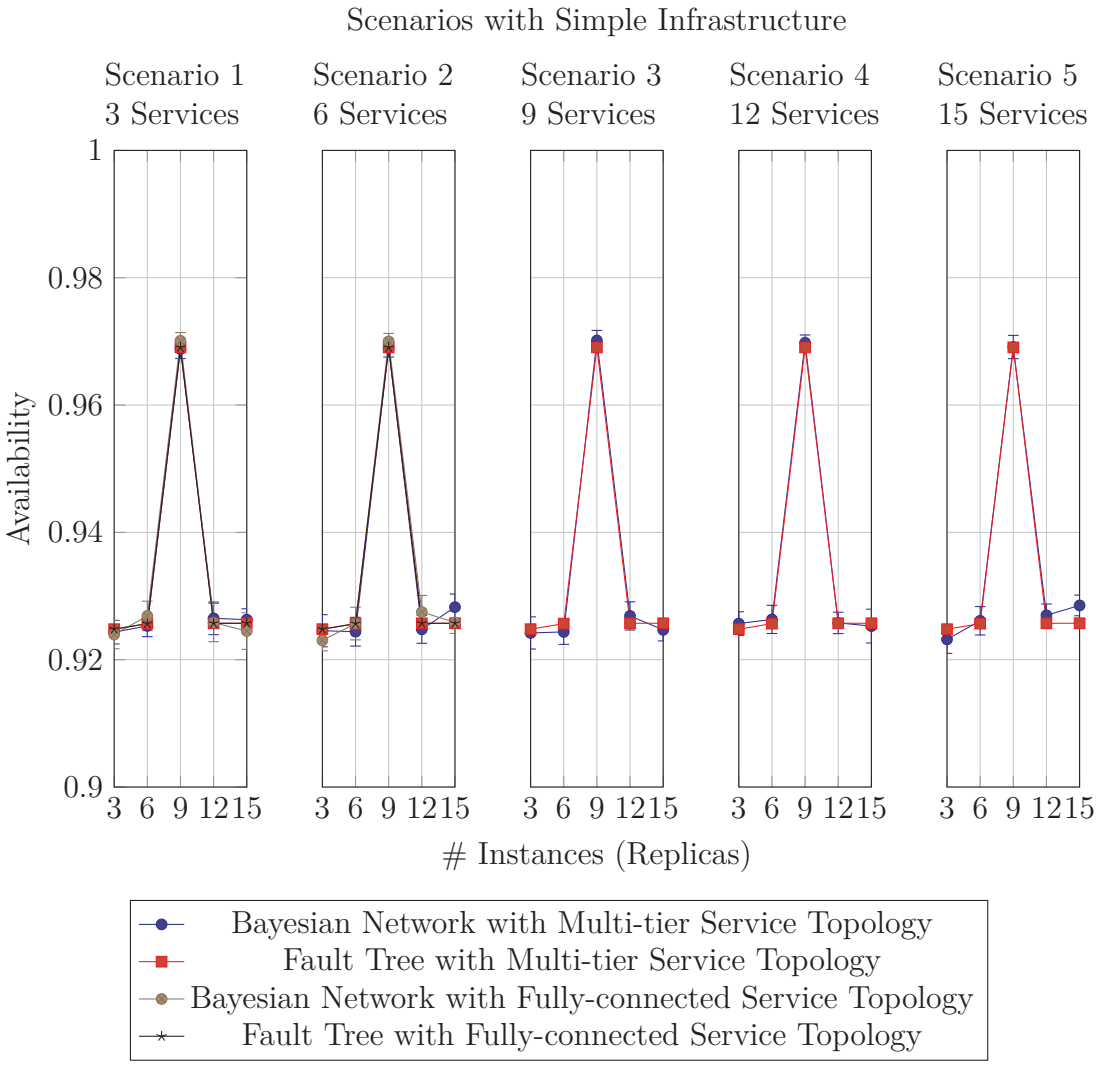


Figure 4.20.: Comparing the availability for different scenarios and different number of replicas, using the simple infrastructure model.

Next, we assess different scenarios with an increasing number of services. Figure 4.20 shows five scenarios where each scenario has a different number of services, which have different numbers of instances. All services use replication and have the same amount of instances as indicated by the x-axis. A service is available as long the majority of its instances are available. All scenarios use the simple

infrastructure model for their fault dependency graph. The service instances are deployed in round-robin across all hosts, starting with the first service at the first host and continuing with the first instance of the next service where the previous service left off. For each scenario, we compare the availability results between the Bayesian network model and the fault tree model for the multi-tier and the fully-connected service topology. Hence, the smallest scenario has three services, each with three replicas, resulting in nine instances. In contrast, the largest scenario has 15 services with 15 replicas per service, resulting in 225 instances. So far, approximate inference in both models became infeasible for scenarios that use the fully-connected service topology when they have more than six services. However, inference was still possible for the remaining scenarios that used the multi-tier service topology.

Again, we repeated each experiment 20 times and computed its 95% confidence interval with the help of the Student's t-distribution, shown as interval bars in the plots. All data points of the fault tree experiments are within the 95% confidence intervals of the corresponding Bayesian network experiments, except for the last scenario with 15 services and 15 instances per service. Here, the model size might have led to floating-point errors, which could have caused the discrepancy. Nevertheless, with the exception of this outlier, it can be stated with 95% confidence that there is no significant difference between the outcomes of the Bayesian network model and the fault tree model.

The most interesting observation is the peak in every scenario when all services have nine replicas, independent of the service number. Due to the round-robin deployment scheme and the simple infrastructure model, which has nine hosts, each service equally distributes its instances on all hosts from start to end. Only five instances need to be available for each service; hence, services can tolerate the fault of one data center or one full rack of hosts. Conversely, scenarios with services where the replication degree is less than nine cannot tolerate a data center failure. In contrast, a replication degree above nine cannot tolerate a rack failure. Consequently, the deployment scheme with nine instances can tolerate more component failures, which results in a higher availability.

Finally, Figure 4.21 shows the availability results for scenarios that use the large infrastructure model in combination with the multi-tier and fully-connected service typologies. Here, we analyze scenarios with three, six, and nine replicated services with varying instances. Again, we use approximate inference to compute the respective availability values. Availability predictions with fault trees were only possible for at most nine services with nine instances per service for the multi-

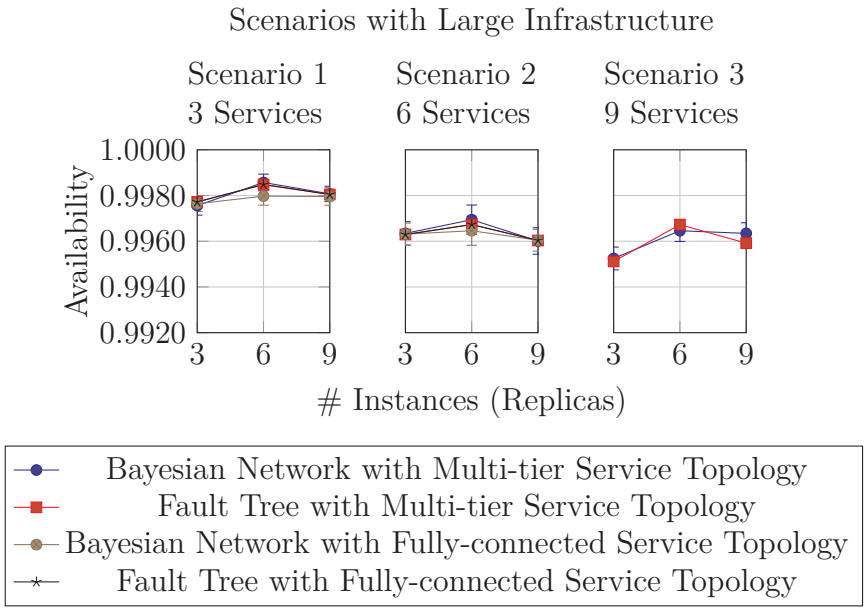


Figure 4.21.: Comparing the availability for different scenarios and different number of replicas, using the large infrastructure model.

tier service topology; afterward, computation became infeasible. For scenarios with the fully-connected service topology, computation with the Bayesian network and fault tree model was only possible for six services with up to nine instances per service. All data points of the fault tree experiments are within the 95% confidence intervals of the corresponding Bayesian network experiments. Hence, it can be stated with 95% confidence that there is no significant difference between the outcomes of the Bayesian network model and the fault tree model.

In summary, this empirical verification showed no significant difference between the results of the fault tree model and Bayesian network model in these experiments. Assuming that the fault tree model has correctly implemented the system description of the scenario models, it can be concluded that the Bayesian network has also correctly implemented the same model description.

4.5.3. Performance

The goal is to build a recommendation system that suggests cost-minimal cloud services with respect to availability constraints. Hence, prediction performance is important in building a recommendation system that suggests optimal services within a manageable time frame. In the following, we will analyze the performance of the previous experiments and discuss their main limiting factors.

Figure 4.22 depicts the computation time to predict the availability of a repli-

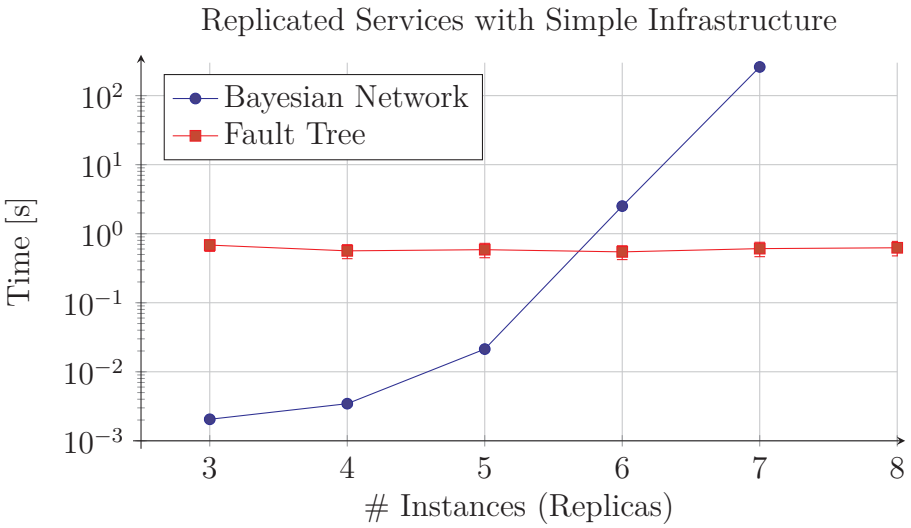


Figure 4.22.: Comparing the prediction time of a replicated service with increasing number of replicas, using the simple infrastructure model and exact inference techniques.

cated service with an increasing number of instances, using the simple infrastructure model. The evaluation uses exact inference for the fault tree and Bayesian network models. As previously said, inference for the Bayesian network model was only possible for services with up to seven instances; afterward, memory limitations made further assessments infeasible. In contrast, the fault tree model exhibits a computation time uncorrelated by the number of instances. Exact inference in Bayesian networks is faster than its equivalent fault tree model for service with three to five instances. However, after five instances, the computation time increases exponentially, so the fault tree model is faster for services with six to eight instances.

Nevertheless, the fault tree model also exhibits an exponentially increase in computation time once we use the large infrastructure model. Figure 4.23 shows the corresponding plots of the computation time. Predictions with the fault tree model was only possible for services with up to 12 instances. Afterward, the assessment became infeasible. All experiments were repeated 20 times to compute their 95% confidence intervals. Since the confidence intervals overlap for services with three to six instances, it can be stated with 95% confidence that there is no significant difference between the computation time of the Bayesian network model and the fault tree model. For services with seven and more instances, the computation time of the fault tree model increases exponentially, whereas the Bayesian network model remains below one second. For services with three

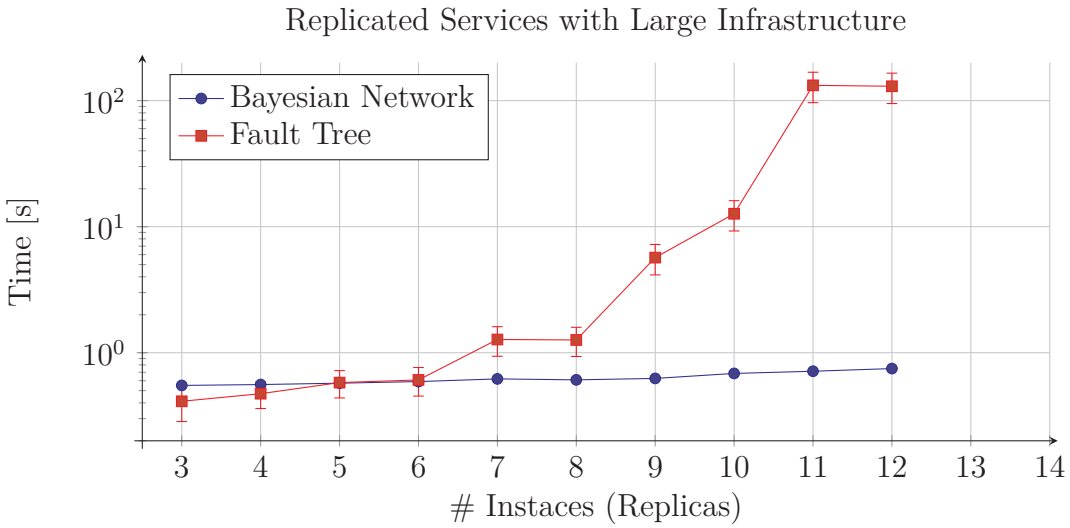


Figure 4.23.: Comparing the prediction time of a replicated service with increasing number of replicas, using the large infrastructure model and approximate inference techniques.

to six instances, the fault tree model exhibits the same time behavior as in the experiments shown in Figure 4.22. However, once the models get sufficiently large, the computation time increases accordingly.

Next, we analyze scenarios with multiple services, using the simple infrastructure model as a basis. Figure 4.24 shows five scenarios with an increasing number of instances per service. Again, all services use replication with majority sets, having the same amount of instances as indicated by the x-axis. All five scenarios implement the multi-tier and fully-connected service topologies. The figure plots the 95% convenience intervals as bars at each data point. Bayesian network inference was only possible for the scenarios with up to six services and 15 instances for the fully-connected service topology; afterward, inference became infeasible due to memory limitations. However, predictions were still possible for all scenarios and service sizes with the multi-tier service topology. For both service topologies, the fault tree and the corresponding Bayesian network models show no significant time difference for scenarios with three and six services, as indicated by the overlapping confidence intervals. For scenarios with nine and more services, the Bayesian network model is initially faster than the fault tree model for services that have a replication degree of three. With an increasing number of replicas, the Bayesian network exhibits a slower performance than the corresponding fault tree models.

However, when comparing the computation time for the same experiments,

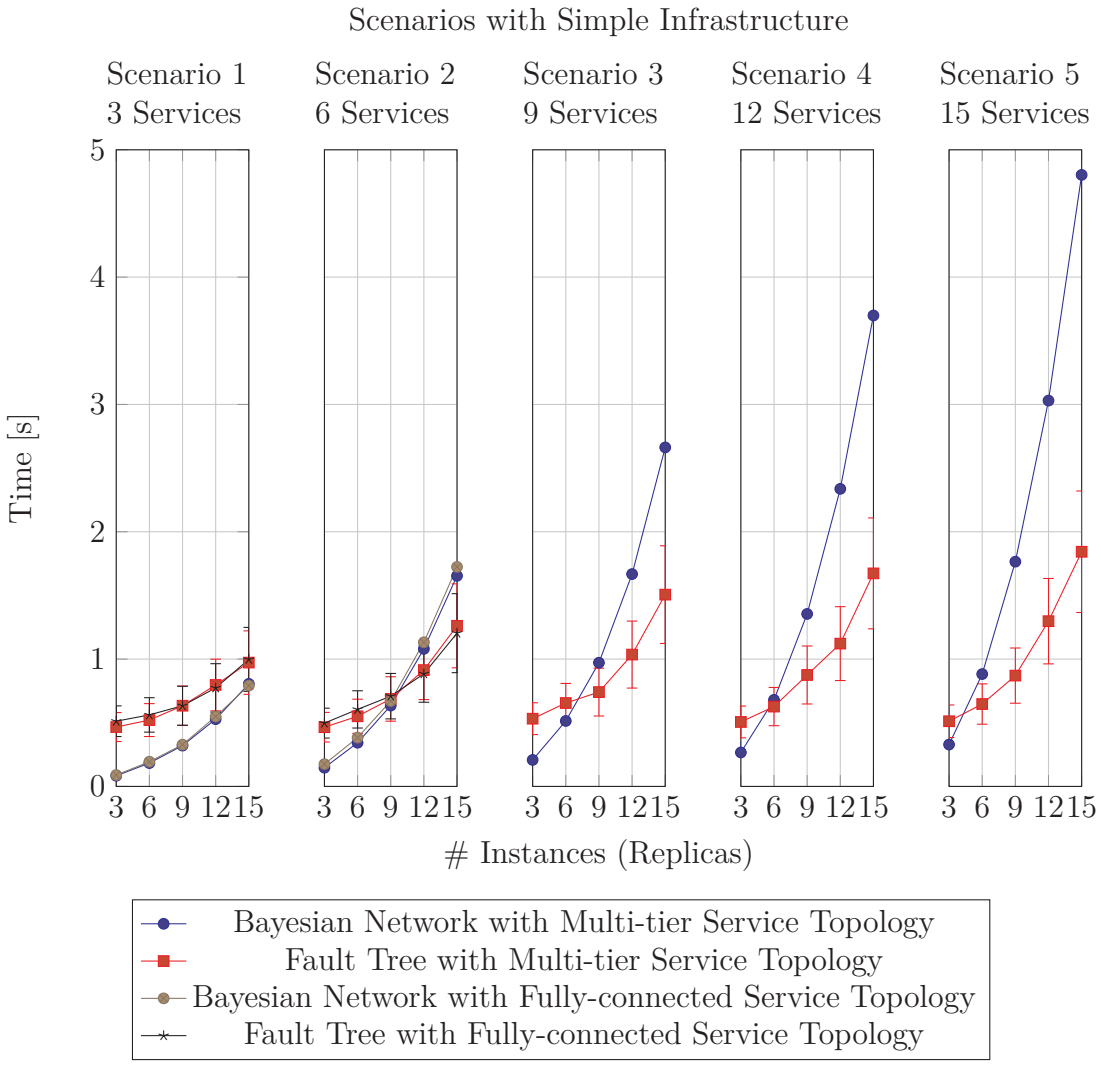


Figure 4.24.: Comparing the prediction time for scenarios with increasing number of services, using the simple infrastructure model and approximate inference techniques.

using the large infrastructure model, we observe a similar effect as for the single service experiments. The Bayesian network model gets up to two orders of magnitude faster than the fault model for large problems, as shown in Figure 4.25. Scenarios with twelve and fifteen services were impossible to compute in feasible time with both models. Similarly, scenarios with the fully-connected service topology could only be evaluated for up to six services with nine instances per service. We observe that with an increasing number of services, the differences between the fault tree and Bayesian network models increase significantly after six replicas per service in all three scenarios.

The evaluations show that Bayesian networks perform better than fault trees

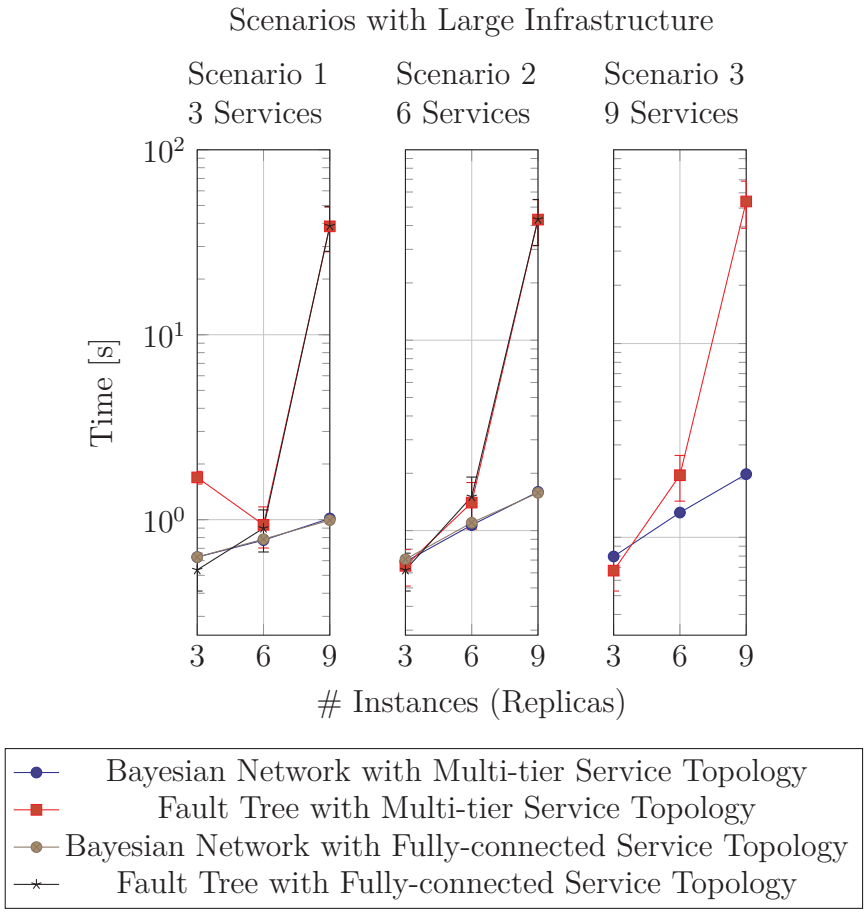


Figure 4.25.: Comparing the prediction time for scenarios with increasing number of services, using the large infrastructure model and approximate inference techniques.

for large availability models. Predictions with fault tree models outperformed Bayesian network models whenever experiments used the simple infrastructure model. In contrast, the Bayesian network models outperformed the fault tree models whenever the experiments used the large infrastructure model. Hence, we conclude that the size of the infrastructure model and the number of services are limiting factors for fault trees. However, the reason why larger scenario models could not be evaluated with Bayesian networks was memory. Here, the size of the services had a larger influence on the performance (until the evaluation system ran out of memory).

Due to the difference in the mathematical framework between fault trees and Bayesian networks, performance limitations have different causes. For Bayesian networks, the size of a CPT, i.e., the number of table entries, grows exponentially with the number of parent nodes. Hence, regardless of the content of the CPT,

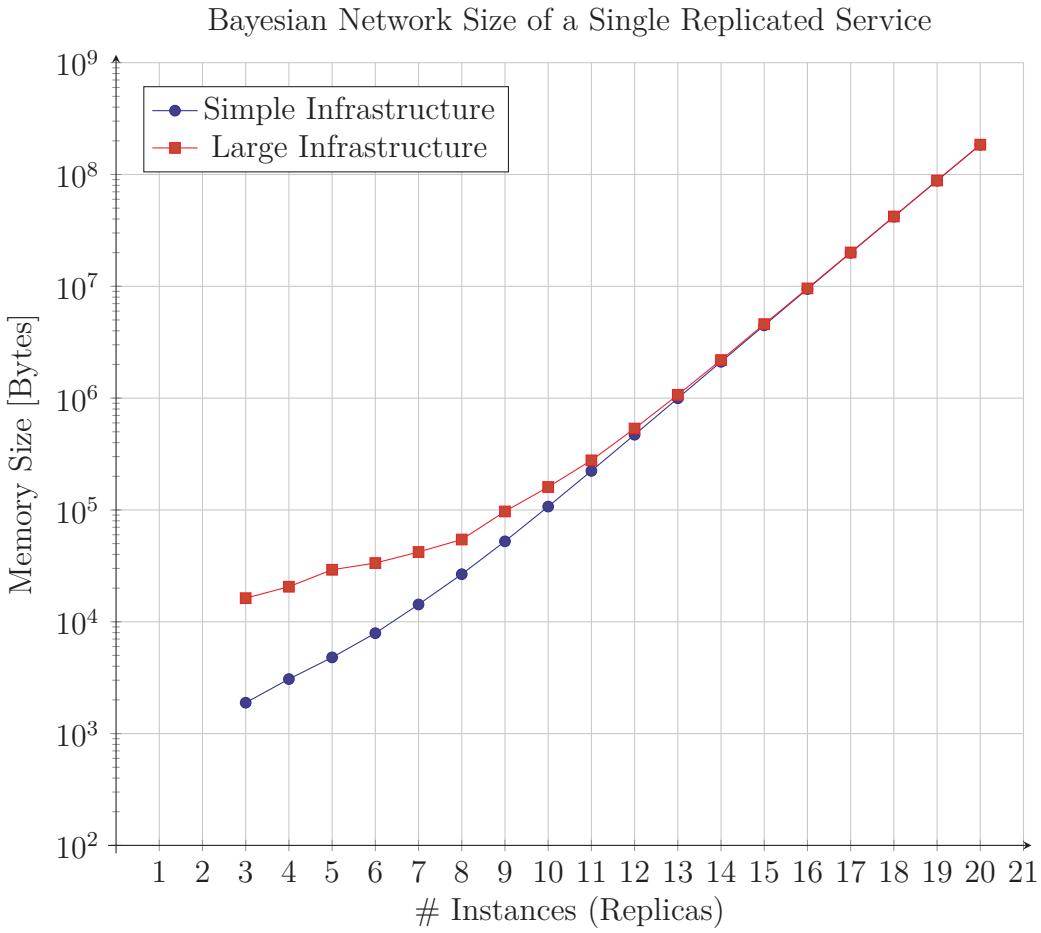


Figure 4.26.: The memory size required to store the Bayesian network availability model for a single service with increasing number of replicas, using the simple and large infrastructure.

its size will always have $O(2^n)$ for parent nodes with binary states. Figure 4.26 illustrates the overall size of the Bayesian network of the replicated service for an increasing number of replicas, using the simple and large infrastructure models. At first, as expected, models that use the large infrastructure require more memory space than those that use the simple infrastructure model. However, when adding more instances, the memory demand grows exponentially (linear grows in log plot) until the large and the simple infrastructure become almost alike. The main driving factors are those nodes Q_1 to Q_n, representing the probability that instances can communicate with sufficient remaining instances. Each node Q_i has $n-1$ parent nodes. So with increasing n, those nodes ultimately outweigh the size of the remaining Bayesian network. The presented infrastructure examples here have about 184MB in size when considering 20 replicas. With this growth rate, 30 instances would already require 16 gigabytes of memory, making it infeasible

to handle larger models.

However, fault trees use binary decision trees to compute minimal cut sets. The resulting cut sets are then used to compute the probability of the top event, either exact or via approximation with MCUB. Hence, for both solution approaches, the fault tree must first be translated into a binary decision tree. The size of the resulting binary decision tree and, therefore, the performance to find minimal cut sets depends on the ordering of the base events. As Tani et al. [129] proved, finding an optimal ordering is NP-complete. This explains why the size of the infrastructure model has a greater influence on the fault tree analysis because the infrastructure model contains all the components that form the base events of the fault tree.

In summary, fault tree models suffer in principal from a computational complexity issue, whereas the Bayesian network models suffer more from a representational issue. However, we found a solution to circumvent the representational issue for redundant and voting-based replicated services, which we will discuss in the next chapter. Nevertheless, the fact that Bayesian networks can also express more generic availability requirements and assess larger infrastructure models than fault trees makes Bayesian networks a suitable model for predicting the availability of large-scale cloud services.

4.6. Related Work

Several methods exist to evaluate a system's availability, among which Bayesian networks have gained large acceptance within the industry and research [27–30].

Bobbio et al. [117,130] demonstrated the applicability and superiority of Bayesian networks in modeling and evaluating equivalent fault trees [12]. Moreover, Boudali and Dugan [118, 131] showed how to use dynamic Bayesian networks to model dynamic fault trees, effectively proving that the Bayesian network formalism is powerful enough to cover all non-state space models.

In network modeling, Bennacer et al. [105] use Bayesian networks for network diagnostics by introducing a novel inference approach to increase diagnostic performance for large-scale Bayesian network models. Their method exemplifies the general approach in predicting QoS attributes for networks, considering common cause influences from network devices. Similarly, Giorgio and Liberati [132] use dynamic Bayesian networks to model the reliability of critical infrastructures such as power grids, where components have multi-valued fault states and dynamic fault behaviors. However, they only use a one-out-of-n semantic to model

redundancy.

As for cloud applications, Pitakrat et al. [23] use Bayesian networks for online failure predictions of microservice architectures. Their Bayesian network model represents the interconnection between services. They use an online monitoring tool to gather performance metrics from a real-live system to update the Bayesian network's probability tables and to infer service failures in advance. While they consider fault propagation between services, they do not consider replication as such.

Nevertheless, modeling cloud infrastructures is subject to various areas of reliability and availability engineering [101, 113, 133]. Cloud infrastructures can get large, consisting of several data centers and hundreds of hosts [134]. Hence, a monolithic availability model might become computationally intractable. Therefore, one approach to scale computation is using hierarchical composition modeling [19].

Kim et al. [133] introduce a hierarchical availability model for VMs running on host systems. Their hierarchical model uses fault trees at the root level and smaller CTMCs for the lower level to model the failure rates of individual components, e.g., CPU, memory, and power of the host system. As for this work, the hierarchical composition method from Section 3 can be further extended. One can combine the here proposed Bayesian network model with state-space methods. Individual cloud components can be assessed using state-space methods to compute failure rates or mean time to failure (MTTF). Afterward, their results can be used as input values for the Bayesian network model, representing the overall application model at a higher composition level.

Jammal et al. [21] provide an availability model for multi-tier cloud applications, where they describe applications in a domain-specific modeling language using UML. However, they only consider fault propagation within a strict hierarchical infrastructure model, accounting only for the failures of the data center, server, and VM. Their domain-specific model strongly emphasizes IaaS and does not consider network communication or the replication degree of services.

As for architecture-based reliability modeling, PCM [24, 25] provides a holistic modeling approach to evaluate the performance and reliability of complex software systems by unifying hard – and software – into one model. PCM supports complex usage profiles for different user roles. The reliability model translates components into CTMCs to compute the overall failure rates. However, the model provides limited support for replication, considering at most a one-out-of-n fault requirement.

4.7. Summary

This chapter introduced a Bayesian network availability model derived from a high-level scenario description. The Bayesian network model unifies the fault aspects from three sub-models: a fault dependency graph to express the failure relation between cloud services and the execution environment, a network model to address communication and network partitioning failures, and a service model to define individual availability requirements at the instance level, supporting two different communication patterns to express stateful and stateless services. Evaluations show the feasibility of the Bayesian network approach to represent services and scenarios containing hundreds of components. However, one issue in modeling larger applications is the general limitation imposed by the exponential memory growth of the CPTs. The next chapter tackles this problem by introducing scalable Bayesian network representations, reducing memory growth from exponential to polynomial.

Chapter 5.

Scalable Bayesian Network Structures

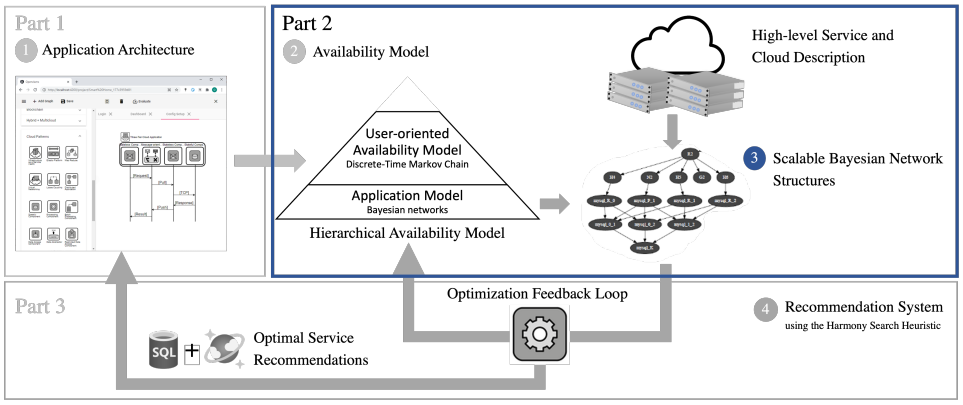


Figure 5.1.: This chapter focuses again on part two of the solution workflow, proposing a novel structural enhancement to model larger-scale services with Bayesian networks.

A significant challenge of the previously introduced Bayesian network availability model is the so-called *exponential memory blow-up problem*, where the CPTs grow exponentially in the number of parent nodes. So far, the AND/OR nodes already have scalable representations; however, a scalable k-out-of-n model that defies the exponential memory blow-up problem is missing. The k-out-of-n model is primarily used to implement the broad range of voting-based replicated services. Therefore, a large number of modeling cases would benefit from a scalable k-out-of-n model. Hence, this chapter focuses again on part two of the solution workflow as shown in Figure 5.1, introducing a scalable k-out-of-n model [60, 61] based on the temporal noisy adder [135] that reduces the required space size from exponential to polynomial in the number of parent nodes.

Section 5.1 introduces the exponential memory blow-up problem and motivates

the need for the scalable k-out-of-n model. Section 5.2 gives an overview of how to realize the scalable AND/OR models, followed by the main contribution of the scalable k-out-of-n model and its noisy representation. Next, Section 5.3 provides a memory and performance evaluation by comparing the previous modeling results from Chapter 4 with the results of equivalent models that use the scalable representation. Section 5.4 discusses related work. Finally, Section 5.5 concludes this chapter.

5.1. Introduction

Large-scale cloud services pose modeling challenges that require scalable availability models. Nowadays, the number of instances for redundant and replicated services usually exceeds the typical size of three and five instances. FaaS offerings such as AWS Lambda or key-value storage solutions such as Casandra [136] tend to have tenths or hundreds of replicated instances. The previously defined availability model mainly depends on the Bayesian network representation of fault tree gates. These Bayesian network nodes are so-called *converging nodes*, i.e., nodes in which multiple parent nodes converge. However, with an increasing number of parent nodes, the CPT grows exponentially in size [34]. Especially when modeling replicated services, the resulting CPTs of the state nodes Q_i grow exponentially in the number of instances.

So far, the Bayesian network representation for redundant and voting-based replicated service uses the k-out-of-n model. Let us assume we model a replicated service with n instances. The state nodes Q_1 to Q_n have a total CPT size of $O(n \times 2^n)$. Consequently, a node with $n = 30$ parent nodes already results in 30×2^{30} CPT entries, which amounts to several gigabytes of memory. Hence, a node with $n = 60$ parent nodes requires 60×2^{60} CPT entries, which amounts to several zettabytes of memory, making it impossible to store in today's state-of-the-art compute infrastructures. For small n, e.g., three or five, as known from transaction-oriented database systems, the memory is not the limiting factor. However, inference becomes intractable for larger $n > 30$ as used in key-value storage such as Casandra [136]. Therefore, a scalable k-out-of-n model is necessary to represent services with more replicas.

This chapter provides a solution to mitigate the exponential growth for the class of services that use the k-out-of-n model to represent redundant, and voting-based replicated services. Scalable Bayesian network structures already exist for the AND/OR models [135]; however, a scalable structure for the k-out-of-n model is

missing. Hence, the contribution of this chapter is a scalable k-out-of-n model [61], which is compatible with standard inference algorithms while simultaneously reducing the exponential growth of CPTs to polynomial growth. Moreover, further contributions also include an approach to automatically replace non-scalable k-out-of-n models with their scalable counterparts in existing Bayesian networks and concepts to implement a scalable *noisy* k-out-of-n model. Finally, evaluations show the superiority of the scalable k-out-of-n model by comparing it with performance results from the previous chapter.

5.2. Scalable Gate Structures

The following section introduces the formal construction of the scalable k-out-of-n model. It starts with a brief discussion on conditional independence, a key requirement for the transformation. Afterward, for the sake of completeness, this section briefly presents the scalable AND/OR models. Finally, this section shows how to derive the scalable k-out-of-n model from its non-scalable counterpart step-by-step.

5.2.1. Causal Independence

In general, input events of fault tree gates have an individual contribution to the output event. Hence, the AND/OR and k-out-of-n voting gate representations for Bayesian networks give rise to the possibility of exploiting causal independence, representing their nodes in a memory-efficient manner. In this work, causal independence refers to the "temporal definition of causal independence" as defined in [135, 137].

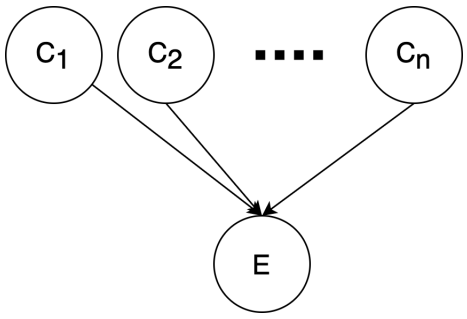


Figure 5.2.: Bayesian network representing an effect influenced by its causes

Consider the common case of a converging node E with parent nodes C_1 to C_n as shown in Figure 5.2. The parent nodes are commonly referred to as *cause*

variables, whereas the child node is the *effect variable*. The temporal definition of causal independence assumes that when cause variables randomly change their state over time, their temporal behavior has only an individual contribution to the probability distribution of the effect variable. Roughly speaking, the effect is the sum of its causes. In detail, assume $C_{j,t}$ refers to the cause variable C_j at time t, and E_t to the effect variable E at time t. Suppose the cause variable C_j changes its state within the time interval t and t', while the remaining cause variables remain unchanged. In that case, the effect variable changes its probability distribution depending solely on the state transition of C_j. Hence, the change of the final probability distribution of the effect variable is independent of the current states of the remaining cause variables. In order to determine if a variable is conditionally independent, Heckerman and Breese [137] propose the following assertion criteria.

Definition 9 (Causal Independence Assertion)

$$\forall t < t', C_j : \quad (E_{t'} \perp C_{1,t}, \ldots, C_{j-1,t}, C_{j+1,t}, \ldots, C_{n,t} | \\ E_t, C_{j,t}, C_{j,t'}, C_{k,t} = C_{k,t'} \text{ for } k \neq j)$$

This assertion reads as follows. Assume C_j changes its state, then the effect variable E is conditionally independent ($\perp$) of the remaining cause variables, given that the states of the remaining cause variables have not changed $C_{k,t} = C_{k,t'}$ for $k \neq j$, while $C_{j,t}$ makes its transition in the time interval t to t'.

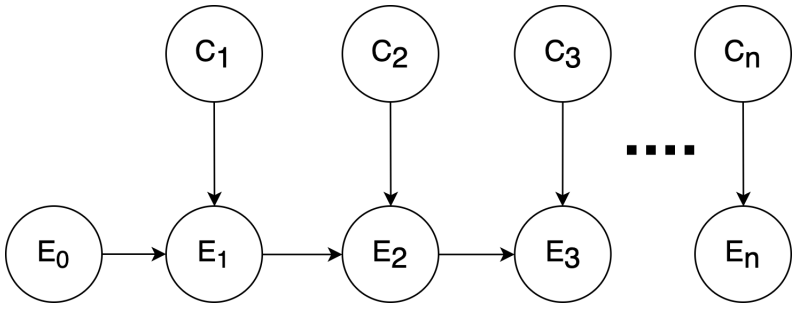


Figure 5.3.: The temporal Bayesian network structure.

For instance, the causal independence assertion can be fulfilled by a chain-like Bayesian network structure, as shown in Figure 5.3, also known as the *temporal representation*. The space reduction is evident. While a child node with n parents requires 2^n tables entries for binary variables, the temporal representation requires only $4n + 1$ (plus one for the initial variable E_0).

According to Heckerman and Breese [137], for the causal independence assertion to hold, the cause variables need to have a distinguished state that does not contribute to the effect. For example, for binary random variables with states $\{F, T\}$ that represent the availability of components in a system, the state T does not affect the system's availability. In contrast, changing a component's state from working to unavailable might affect the system's availability.

The variables E_0 to E_n are so-called *contribution variables*. The result of the final variable E_n represents the original effect variable E. A contribution variable E_i adds the individual contribution of its connected cause C_i to the overall contribution, as perceived by its former contribution variable E_{i-1}. If the cause variable is in the distinguished state, its corresponding contribution variable only forwards the effect of its predecessor. Hence, E_0 represents the effect when all-cause variables are in the distinguished state. The general goal of knowledge engineers is to find the appropriate CPT definitions for the contribution variables so that E_n is equivalent to E. To exemplify this task, we first discuss the temporal definition of the AND/OR models.

5.2.2. AND/OR Models

Appropriate definitions for scalable Bayesian network structures of the (noisy) AND/OR models already exist [135]. This section briefly introduces these models for completeness since they are also part of the performance evaluation at the end of this chapter.

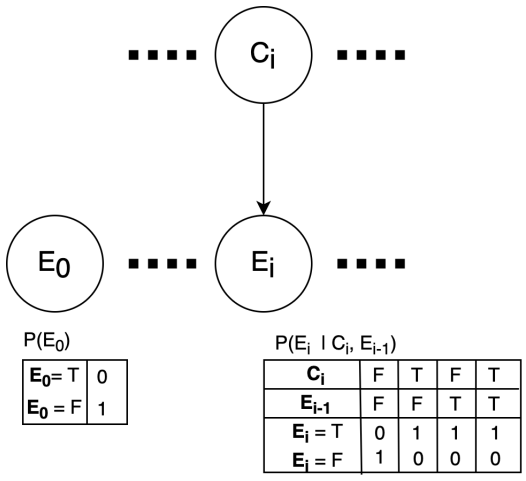


Figure 5.4.: The temporal AND model.

Figure 5.4 and 5.5 show the CPTs of the contribution variables to represent the AND/OR models from Equations 4.2 and 4.3 as temporal structures. Since

the contribution variables have at most two parent nodes, the CPT has only to account for four entries. The counter probability does not need to be stored since it can be computed from the first entries with $P(E_i = F|C_i, E_{i-1}) = 1 - P(E_i = T|C_i, E_{i-1})$. Thus, the temporal AND/OR models have linear memory growth in the number of parent nodes.

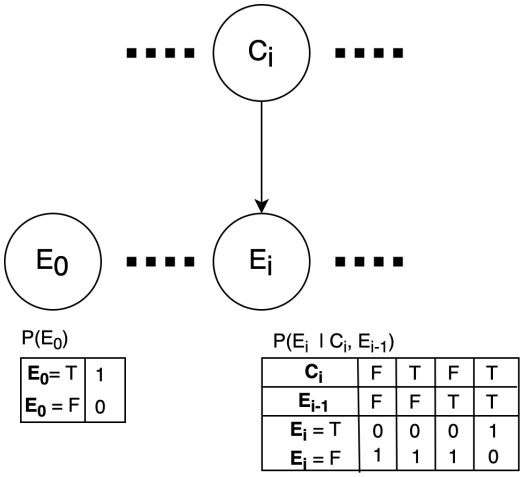


Figure 5.5.: The temporal OR model.

For the AND model shown in Figure 5.4 , the contribution variables implement a smaller AND model. E_0 initializes the chain by being in state F. Similarly, for the OR model shown in Figure 5.5, the contribution variables implement smaller OR nodes for the OR model, whereas E_0 initializes the chain by starting with state T.

Both models can also introduce a leak probability for the system to fail despite having sufficient working components available. This leak probability q_{leak} can be added as part of the last contribution variable. For the AND model, the system can fail with probability q_{leak} although some components might still work.

$$
\begin{aligned}
P(E_n = F|C_i = F, E_{n-1} = F) &= 1 \\
P(E_n = F|C_i = F, E_{n-1} = T) &= q_{leak} \\
P(E_n = F|C_i = T, E_{n-1} = F) &= q_{leak} \\
P(E_n = F|C_i = T, E_{n-1} = T) &= q_{leak}
\end{aligned}
$$

And for the OR model, the system can fail with probability q_{leak} although all

components still work.

$$
\begin{aligned}
&P(E_n = F | C_i = F, E_{n-1} = F) = 1 \\
&P(E_n = F | C_i = F, E_{n-1} = T) = 1 \\
&P(E_n = F | C_i = T, E_{n-1} = F) = 1 \\
&P(E_n = F | C_i = T, E_{n-1} = T) = q_{leak}
\end{aligned}
$$

Additionally, aside from the leak probability, the AND/OR model can also include *noise* to address the chances of the system being still available even in the advent of insufficient working components. Such events might have a low probability, but it accounts for the potential of possible fail-over mechanisms that have not been identified or are simply missing in the system design. For example, a DNS server fails to update its registry (a partial failure that still requires repair); however, cached requests can still be served, allowing some cloud services to function correctly for an uncertain amount of time.

For the temporal noisy AND, the model adds a probability value q^i_{noise} to its contribution variable E_i to model the chance of the system working even though C_i is in a faulty state.

$$
\begin{aligned}
&P(E_i = T | C_i = F, E_{i-1} = F) = q^i_{noise} \\
&P(E_i = T | C_i = F, E_{i-1} = T) = 1 \\
&P(E_i = T | C_i = T, E_{i-1} = F) = 1 \\
&P(E_i = T | C_i = T, E_{i-1} = T) = 1
\end{aligned}
\tag{5.1}
$$

Conversely, for the temporal noisy OR, the model adds a probability value q^i_{noise} to its contribution variable E_i to describe the chance of the system to work even if some C_i are observed in a faulty state given that the previous C_{i-1} components have not failed. However, if one component has failed at some point, the OR semantics require that the whole system is considered unavailable.

$$
\begin{aligned}
&P(E_i = T | C_i = F, E_{i-1} = F) = 0 \\
&P(E_i = T | C_i = F, E_{i-1} = T) = q^i_{noise} \\
&P(E_i = T | C_i = T, E_{i-1} = F) = 0 \\
&P(E_i = T | C_i = T, E_{i-1} = T) = 1
\end{aligned}
\tag{5.2}
$$

The temporal AND/OR models are special cases of the k-out-of-n model, with AND being an n-out-of-n model and OR being a one-out-of-n model. Neverthe-

less, other combinations for k are impossible with binary contribution variables. Therefore, next follows the concepts on how to implement a temporal k-out-of-n model.

5.2.3. k-out-of-n Model

The following section shows how to derive the scalable k-out-of-n model by gradually changing the non-scalable k-out-of-n model representation into its scalable form.

The foundation of the scalable k-out-of-n model is the *temporal noisy adder* [135], which is the scalable version of the general noisy adder. A *noisy adder* is a random variable representing a counter. Its probability distribution defines the likelihood of observing how many causes are in a certain dedicated state. For example, Figure 5.6 shows an adder model for the dedicated state F with three binary parent nodes C_1 to C_3. The effect node N has an Integer domain with the states $\mathrm{dom}(N) = [0, 3]$, which reflects the total count of causes that are in state F[1] within a particular instantiation. Consequently, for n causes, the domain of N would be $\mathrm{dom}(N) = [0, n]$.

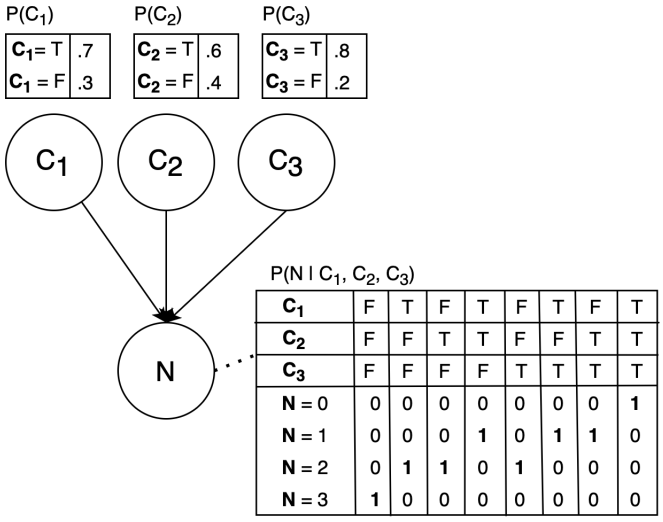


Figure 5.6.: The adder model.

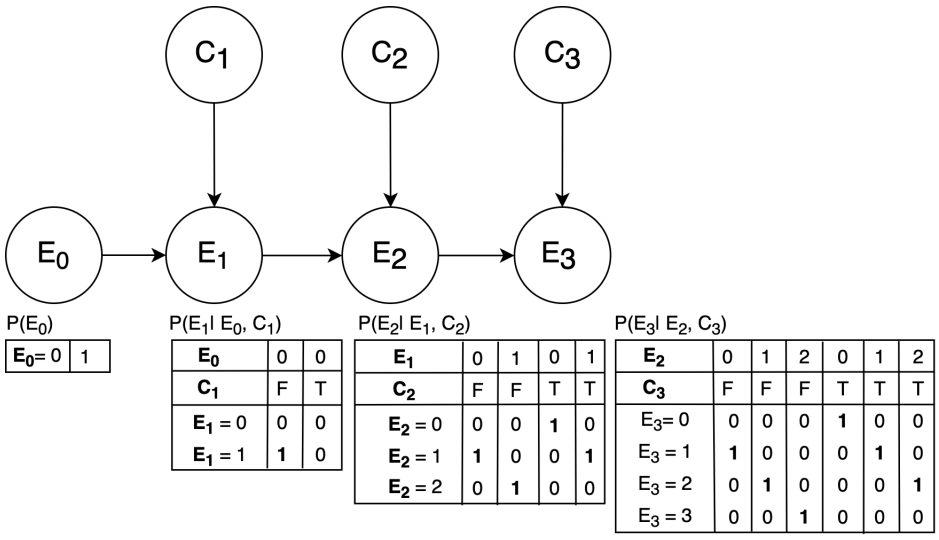
To solve the exponential memory growth, Heckerman [135] proposed the temporal (noisy) adder. For example, Figure 5.7 shows the corresponding temporal adder (with no noise) for the three cause variables, where the contribution variable

[1]The corresponding publication [61] considers the dedicated state T. However, w.l.o.g., in order to remain in line with the fault tree voting gate definition of the Bayesian network availability model, F has been chosen instead.

E_i stores the intermediate count of all cause variables that are in the dedicated state up to the i-th cause variable C_i. Thus, E_i can count at most i observations, hence its possible states are $\mathrm{dom}(E_i) = [0, i]$. Here, E_0 has no utility other than offsetting the counter if necessary. Consequently, the CPT of a contribution variable E_i (except for E_0) is as follows:

$$\begin{aligned} &\forall i \in [1, n], \forall m : m < i \\ &\qquad P(E_i = m+1 | E_{i-1} = m, C_i = F) = 1 \\ &\qquad\quad P(E_i = m | E_{i-1} = m, C_i = F) = 0 \\ &\qquad P(E_i = m+1 | E_{i-1} = m, C_i = T) = 0 \\ &\qquad\quad P(E_i = m | E_{i-1} = m, C_i = T) = 1 \end{aligned} \tag{5.3}$$

E_i changes its count from m to $m+1$ with probability one, whenever C_i is in state F. However, if $C_i = T$, then E_i propagates the current state of his parent node E_{i-1}. E_0 is initialized to $P(E_0 = 0) = 1$ so that the adder can start from zero. Finally, the last contribution variable E_n contains the count and is equivalent to the node N.



E_0	1
$E_0 = 0$	

E_0	0	0
C_1	F	T
$E_1 = 0$	0	0
$E_1 = 1$	1	0

E_1	0	1	0	1
C_2	F	F	T	T
$E_2 = 0$	0	0	1	0
$E_2 = 1$	1	0	0	1
$E_2 = 2$	0	1	0	0

E_2	0	1	2	0	1	2
C_3	F	F	F	T	T	T
$E_3 = 0$	0	0	0	1	0	0
$E_3 = 1$	1	0	0	0	1	0
$E_3 = 2$	0	1	0	0	0	1
$E_3 = 3$	0	0	1	0	0	0

Figure 5.7.: The temporal adder model.

Now consider the original (non-scalable) Bayesian network implementation of the k-out-of-n model as exemplified in Figure 4.8. For readability reasons, lets write again the CPT definition of the k-out-of-n model as introduced in Equation 4.4 from the previous chapter, assuming we have random variable K with a

conditional probability distribution based on the variables C_1 to C_n.

$$\forall c_1, \ldots, c_n \in \{F, T\}^n$$
$$P(K = F|c_1, \ldots, c_n) = \begin{cases} 1 & \sum_{i=1}^{n} \mathbf{1}_F(c_i) \geq k \\ 0 & \text{otherwise} \end{cases}$$
$$P(K = T|c_1, \ldots, c_n) = 1 - P(K = F|c_1, \ldots, c_n) \tag{5.4}$$

The conditional probability distribution $P(K = F|c_1, \ldots, c_n)$ is 1 if k and more instances of the parent variables are in state F, and 0 otherwise, where $c_1, \ldots, c_n$ represent one particular instance of states of the random variables C_1 to C_n.

Assuming causal independence, defining T as the distinguished neutral state, and F the dedicated state that increments the sum, $P(K = F|c_1, \ldots, c_2)$ can be represented as $P(K = F|N = m)$, where N is a random variable with the domain $[0, n]$. N reflects the total count of parent variables in state F. Thus, N implements an adder model since it counts the occurrences of a dedicated state of its parent variables.

$$\forall m \in [0, n], \forall c_1, \ldots, c_n \in \{F, T\}^n$$
$$P(N = m|c_1, \ldots, c_n) = \begin{cases} 1 & \sum_{i=1}^{n} \mathbf{1}_F(c_i) = m \\ 0 & \text{otherwise.} \end{cases}$$

The CPT of K has linear size now, due to its new conditional dependency to N.

$$\forall m \in [0, n]$$
$$P(K = F|N = m) = \begin{cases} 1 & m \geq k \\ 0 & \text{otherwise.} \end{cases}$$

Nevertheless, the issue of the exponentially growing CPT shifts to N, due to its conditional dependency on the variables C_1 to C_n. However, the conditional probability distribution of N represents an adder and, therefore, can be substituted with the temporal-adder model.

$$\forall m \in [0, n], \forall c_1, \ldots, c_n \in \{F, T\}^n$$
$$P(N = m|c_1, \ldots, c_n) = P(E_0 = 0) \prod_{i \in [1,n]} P(E_i = e_i|E_{i-1} = e_{i-1}, C_i = c_i)$$

Node K is now conditionally dependent on E_n instead of N, i.e. $P(K|E_n)$, concluding the scalable k-out-of-n model, which can be regarded as a temporal k-out-of-n model.

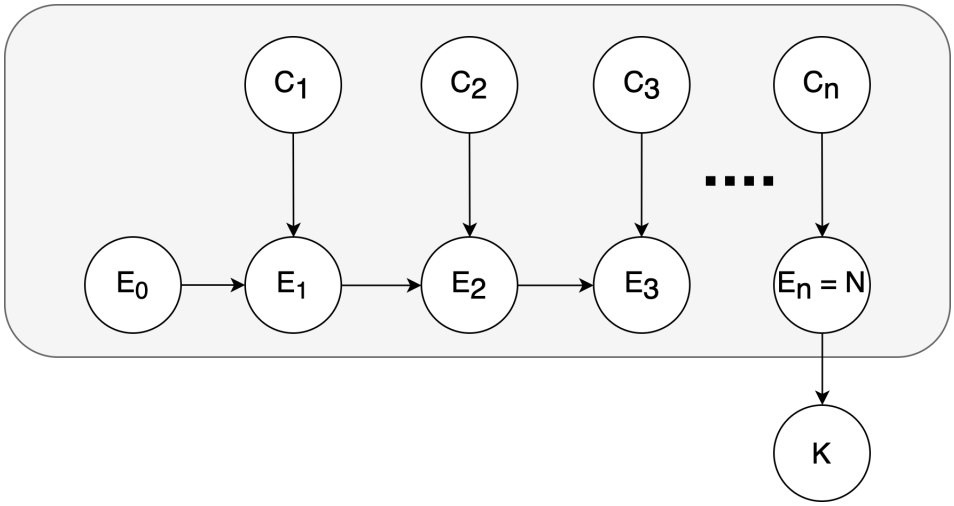


Figure 5.8.: Temporal representation of the k-out-of-n model.

Figure 5.8 depicts the resulting Bayesian network of the scalable k-out-of-n model as two parts. The first part (gray box) contains the temporal adder structure, whereas the second part is the node K, which encodes the k-out-of-n semantics of the model. Since E_n represents the total count of the causes C_1 to C_n, the conditional probability distribution of K can enforce the Boolean expression of the k-out-of-n voting gate. Whenever the count is k and more, we define K's conditional probability distribution as $P(K = F|E_n \geq k) = 1$, otherwise 0. Finally, this model does not deviate from the standard Bayesian network formalism, making it compatible with general-purpose inference algorithms.

The CPT of node K grows linearly in the number of cause nodes. However, the overall space complexity of the temporal noisy adder grows polynomially in the number of cause variables. The total number of contribution variables equals the number of cause variables. The table size of a contribution variable is at most $O(n^2)$. Hence, since there are in total n CPTs, the resulting space complexity is $O(n^3)$, constituting a polynomial growth in the number of causes, similarly to the computational complexity. The computational complexity of the temporal noisy adder is $O(n^3)$ according to Heckerman [135]. Hence, the overall computational complexity of the k-out-of-n model is $O(n^3)$.

5.2.4. Noisy k-out-of-n Model

For the scalable k-out-of-n model, there are two options to include noise. The first option introduces a leak to model systems that might fail although less than k components have failed. The second option introduces a probability to account for the observation uncertainty of components that trigger faults, although no faults have occurred (false positives). Both options can be combined to form the noisy k-out-of-n model.

The first option allows for m leak probabilities q^m_{leak} at node K, representing a system to fail even if less than $m < k$ failures occurred.

$$\forall m \in [0, n]$$
$$P(K = F|N = m) = \begin{cases} 1 & m \geq k \\ q^m_{leak} & m < k \end{cases}$$

Notice, that each possible count m can have its own leak probability q^m_{leak}. This enables sophisticated models, where the leak probabilities can be low for small m and increase for growing m, describing a faster degradation of a system, where the risk of premature failure increases the more components fail.

The second option is to implement the general temporal noisy adder for the k-out-of-n model. The temporal noisy adder considers noise by adding a probability q^i_{noise} to each contribution variable E_i to represent the chance of incrementing the counter when observing $C_i = F$. Consequently the CPT of a contribution variable E_i (except for E_0) is as follows:

$$\forall i \in [1, n], \forall m : m < i$$
$$P(E_i = m + 1|E_{i-1} = m, C_i = F) = q^i_{noise}$$
$$P(E_i = m|E_{i-1} = m, C_i = F) = 1 - q^i_{noise}$$
$$P(E_i = m + 1|E_{i-1} = m, C_i = T) = 0$$
$$P(E_i = m|E_{i-1} = m, C_i = T) = 1$$

5.2.5. Transformation Algorithm

Existing Bayesian networks that already contain the non-scalable implementation of the k-out-of-n model can be automatically changed into an equivalent scalable model. Assume that C_1 to C_n are parent nodes of a converging network structure with child node K. Then, the conversion of this Bayesian network into a scalable

k-out-of-n model includes the following steps:

1. For each C_i, a contribution variable E_i with the domain $[0, i]$ must be created.

2. Let E_i be a child of C_i and E_{i-1}. Also, let K be a child of E_n.
   For each E_i define the temporal adder model:

$$\forall m : m < i$$
$$P(E_i = m + 1 | E_{i-1} = m, C_i = F) = 1$$
$$P(E_i = m + 1 | E_{i-1} = m, C_i = T) = 0$$
$$P(E_i = m | E_{i-1} = m, C_i = T) = 1$$
$$P(E_i = m | E_{i-1} = m, C_i = F) = 0$$

3. Define the conditional probability distribution of $P(K|E_n)$ with:

$$\forall k \in [0, n]$$
$$P(K = F | E_n \geq k) = 1$$
$$P(K = T | E_n \geq k) = 0$$
$$P(K = F | E_n < k) = 0$$
$$P(K = T | E_n < k) = 1$$

This algorithm introduces new nodes into the Bayesian network. These nodes should not have any other parent nodes except for their associated cause variables. If the existing Bayesian network model has multiple voting gates, denoted by the nodes K_1 to K_m, then we apply the transformation similarly to each gate. We represent the contribution variable of the i-th cause node that belongs to the j-th voting gate model as $E_{i,j}$, to certify that the association to the voting gate is unambiguous.

5.3. Evaluation and Discussion

In the following, this section demonstrates the feasibility of the scalable k-out-of-n model by re-evaluating the largest problem instances from the last chapter. All experiments were performed on a 64-bit machine with 64 Intel(R) Xeon(R) CPU E7-4850 v4 at 2.10GHz and 1 TB of main memory, running Arch Linux

5.13.12 with GCC 11.1.0, Python 3.9.6, and Numpy 1.20.3. If not stated otherwise, Bayesian network inference is performed with approximate inference using the forward sampling method. Moreover, throughout this section, the scalable Bayesian network availability model will also use the scalable AND/OR models.

5.3.1. Replicated Services

First, we analyze the scalable k-out-of-n model in the context of a voting-based replicated service that requires majority sets. All experiments will use the large infrastructure example, c.f. Section 4.5, since it entails the most complex problem instance in this work. Other voting schemes would also be possible, but they would only change the content of the corresponding CPTs and do not influence their size.

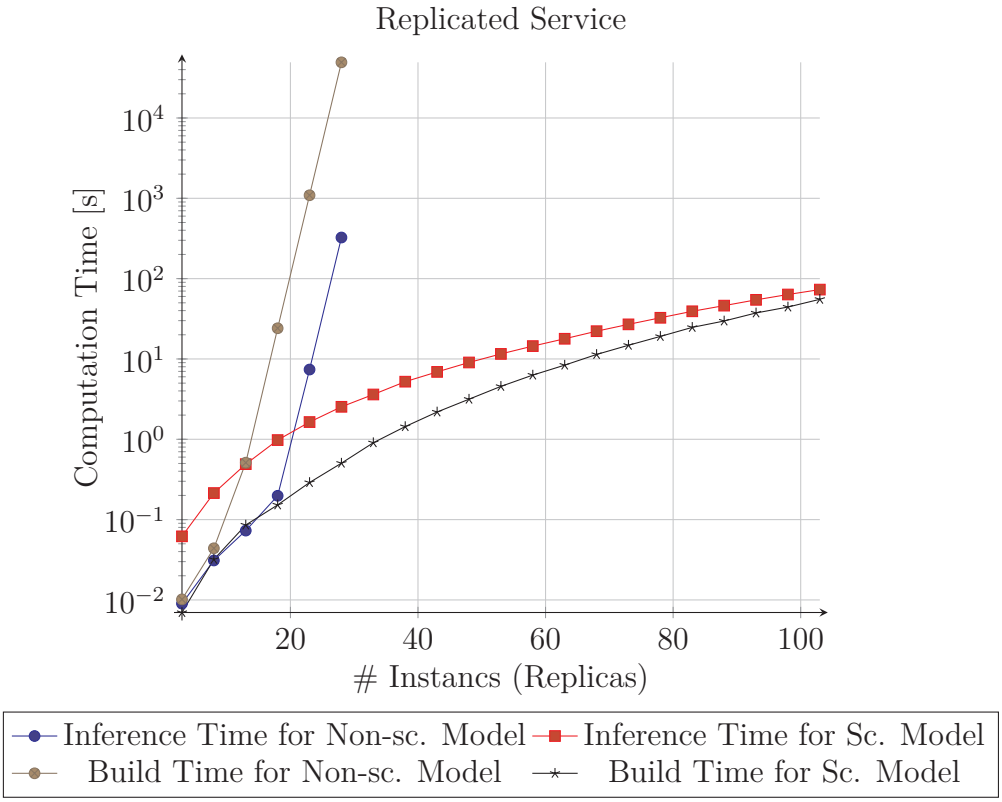


Figure 5.9.: Comparing the inference and build time between the scalable (sc.) and non-scalable (non-sc.) Bayesian network models to compute the availability of a replicated service for an increasing number of replicas using the large infrastructure model.

Figure 5.9 compares the build and inference time to compute the availability of a replicated service using the scalable and non-scalable Bayesian network model

for an increasing number of replicas. With the successive increase of replicas, the inference time and the build time of the non-scalable Bayesian network grow exponentially. What stands out is that the build time for a service with 27 replicas is four orders of magnitude larger than the build time of the scalable counterpart. It took 13h to build the non-scalable Bayesian network. Therefore, evaluations with more than 27 replicas became infeasible for the non-scalable Bayesian network. Nonetheless, it was still possible to continue the evaluations to 100 replicas for the scalable Bayesian network model.

Moreover, the non-scalable Bayesian network exhibits an exponential increase in built and inference time, while the scalable model shows a polynomial growth. Comparing the two results, it can be seen that the build time is always higher than the inference time for the non-scalable models, whereas its the exact opposite for the scalable model. Overall, these results show that even for 100 replicas, the scalable Bayesian network is three orders of magnitude faster in building and five times faster in computing the availability than the time required with the non-scalable network for a quarter of the replicas. This is primarily a result of the reduced number of CPT entries. Fewer CPT entries mean that the forward sampling algorithm needs fewer samples to converge since the maximum number of samples is a function based on the total number of CPT entries. If the network size grows exponentially, the number of required samples also grows exponentially, leading to an exponential increase in the inference time. Conversely, if the number of CPT entries grows polynomial, the inference time increases accordingly.

Surprisingly, the non-scalable model shows a better performance for a small number of replicas, i.e., $n \leq 13$. These results are likely related to the additional nodes inserted into the scalable Bayesian network, which ultimately increase the inference time compared to the non-scalable counterpart. This finding has important implications for building the Bayesian network of a replicated service. If a replicated service has three or five replicas, it is best to use the non-scalable k-out-of-n model. However, if the service has more than 18 replicas, one should clearly use the scalable model instead.

5.3.2. Scenario Models

So far, we have analyzed the scalable k-out-of-n model and its impact on the performance of the Bayesian network for one replicated service. Next, we will analyze scenarios with three to fifteen replicated services. All evaluations will use the large infrastructure example. Additionally, the scenarios will use the

multi-tier and fully-connected service topology.

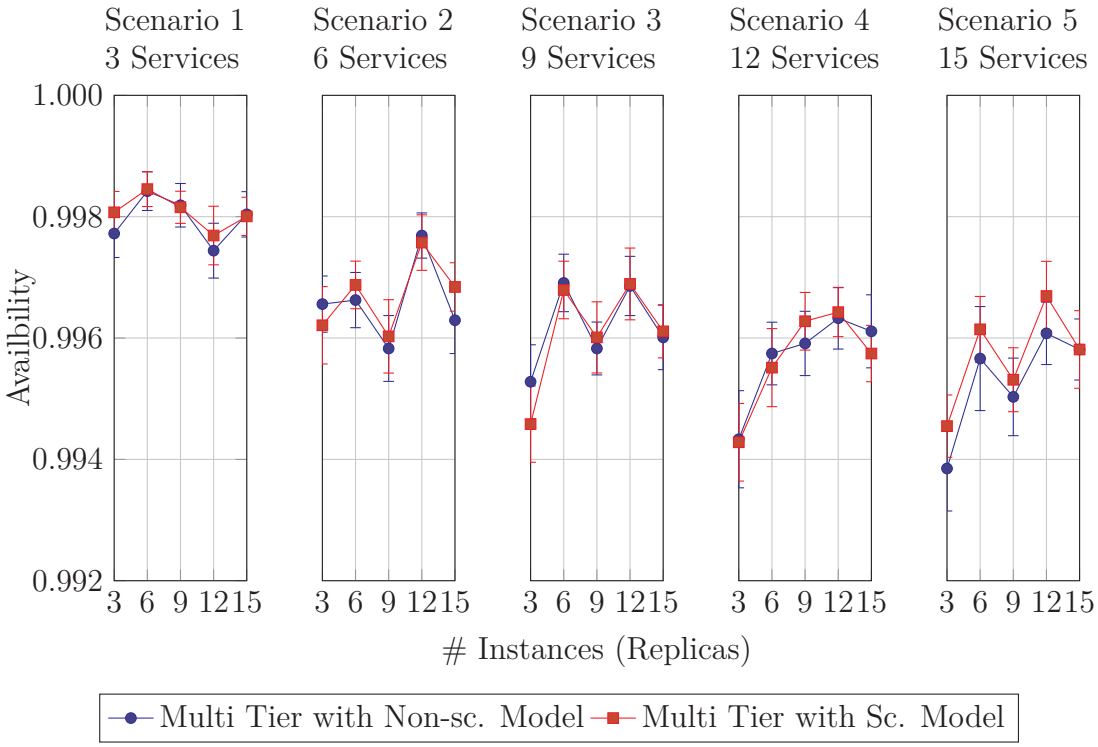


Figure 5.10.: Comparing availability results between the scalable and non-scalable models for an increasing number of services and an increasing number of instances per service, using multi-tier service topology and the large infrastructure model.

Figure 5.10 and Figure 5.11 show the availability results of the scenarios introduced last chapter in Section 4.5. All scenarios in Figure 5.10 use the multi-tier service topology, whereas the experiments in Figure 5.11 use the fully-connected service topology. The availability values of the non-scalable Bayesian network models are the same as in Section 4.5. Again, each service has the same number of replicated instances as indicated by the x-axis. What stands out in Figure 5.11 is that it is now possible to compute the availability for scenarios with a fully-connected service topology with more than six services. As shown in the last chapter, and here again, this was not possible with the corresponding non-scalable Bayesian network model. Moreover, all experiments were repeated 20 times to compute their 95% confidence intervals, shown as interval bars in the plots. Comparing the result of the scalable and non-scalable models, it can be stated with 95% confidence that there is no significant difference in the inference results for this evaluation, suggesting that the scalable and non-scalable models provide the same outcome. However, these results were expected since the scal-

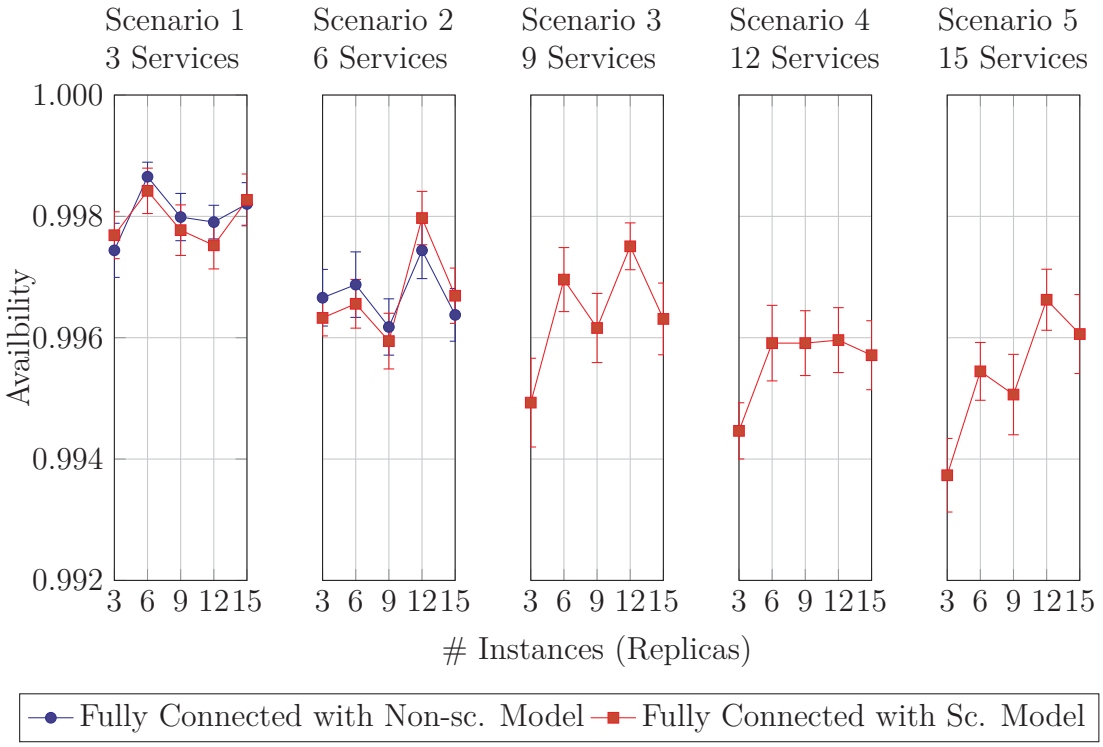


Figure 5.11.: Comparing availability results between the scalable and non-scalable models for increasing number of services and increasing number of instances per service, using fully-connected service topology and the large infrastructure model.

able k-out-of-n model is directly derived from the non-scalable without changing the semantics of the Bayesian network formalism.

Next, Figure 5.12 compares the corresponding inference times. Independent of the service topology, the timing results of the scalable models are significantly higher for nine replicas in every scenario. With successive increase of services and instances per service, the non-scalable models outperform the scalable models. However, these results are consistent with the experiments with one replicated service. The presented evaluation goes only as far as to model services with 15 replicas. As Figure 5.9 already showed, the non-scalable models are more suitable for a smaller number of replicas since they do not have the initial overhead of additional nodes compared to the scalable models. Therefore, applications with more than 15 replicas need to be analyzed to develop a complete picture of the timing behavior.

Figure 5.13 shows the performance results for scenarios with six services. Since this is the largest architectural size for which inference is still tractable for the non-scalable model. What stands out is the exponential increase of the build time

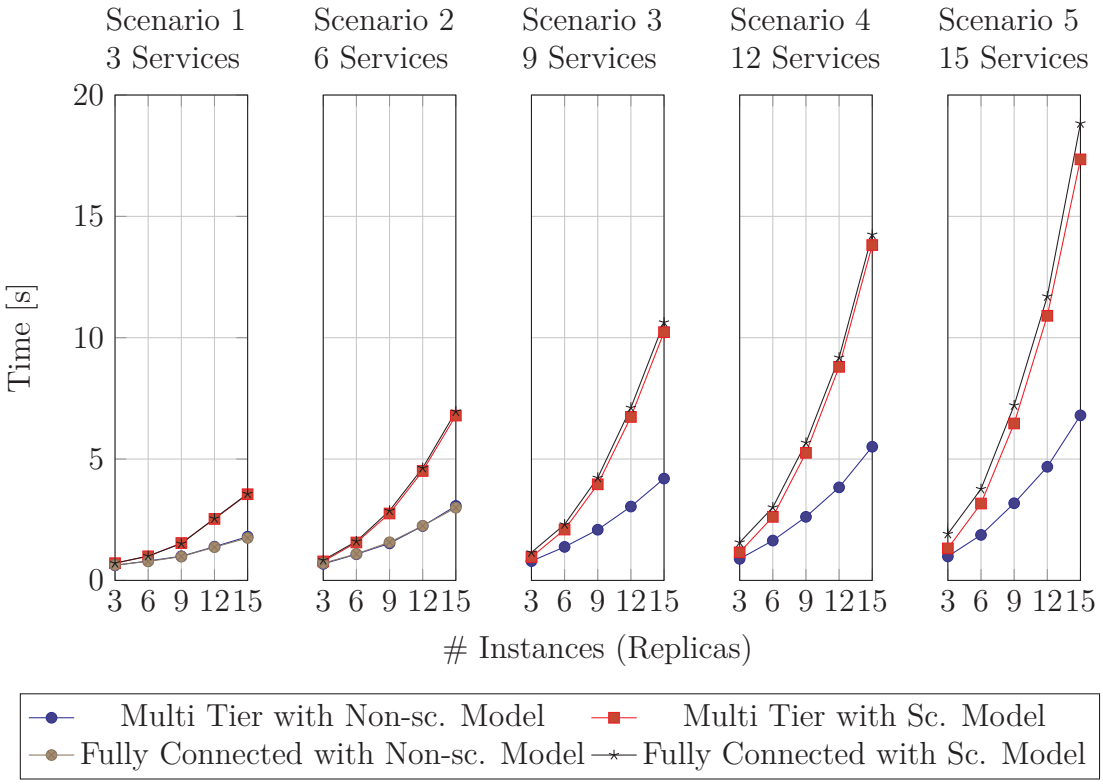


Figure 5.12.: Inference time between the scalable and non-scalable models for an increasing number of services and an increasing number of instances per service, using the multi-tier and fully-connected service topology.

contrasting a low inference time for the non-scalable model. While the inference time is surprisingly low compared to the built time, inference was only possible for up to 24 replicas. The most stunning result provides the scalable model, which enables inference with 60 replicas per service. This amounts to a scenario with an application architecture that has in total 360 instances that share communication and infrastructure faults.

In summary, the evaluation showed that the scalable k-out-of-n model reduces the total CPT size from exponential to polynomial in the number of instances for replicated services. This enables the Bayesian network representation for large-scale services while reducing the build time simultaneously.

5.4. Related Work

As mentioned in the previous chapter, Bobbio et al. [117] established the concepts for assessing static fault trees with Bayesian networks, introducing the CPT def-

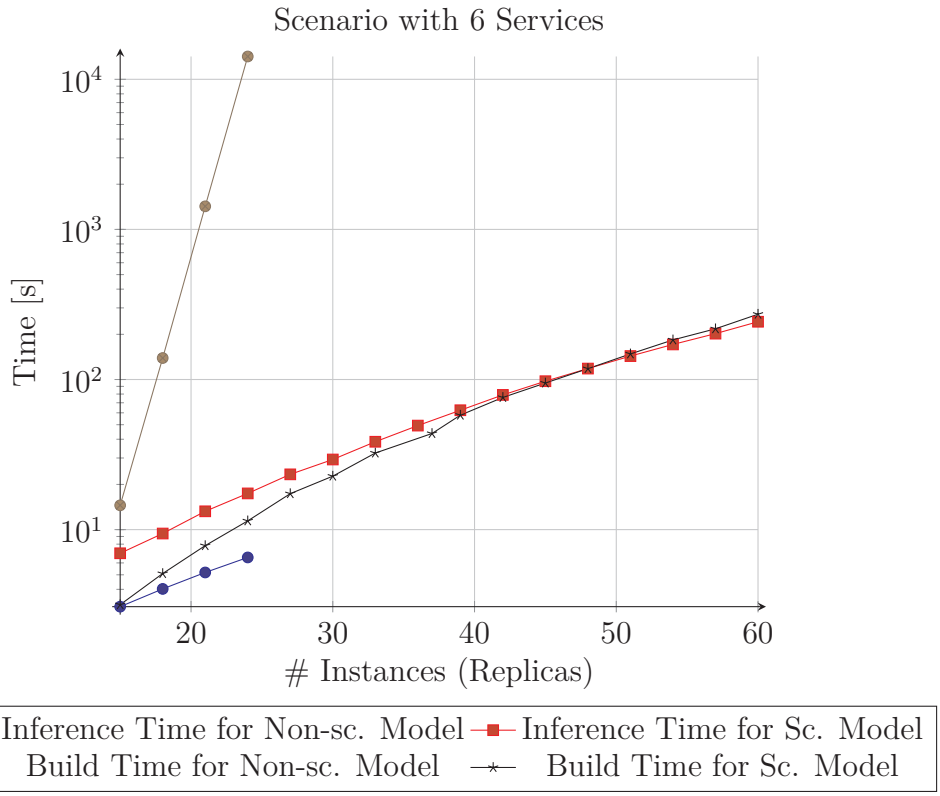


Figure 5.13.: Inference time for a scenario with six services and increasing number of replicas in conjunction with the large infrastructure example.

initions to model the AND/OR and k-out-of-n voting gates. These are the same definitions as used in the availability model from the last chapter. However, the CPT that implements the corresponding Boolean expression of the gates grows exponentially with the increasing number of input events. Boudali and Dugan [118, 131] extended the Bayesian network modeling concepts of Bobbio et al. to encompass also dynamic fault trees. They noticed the exponential memory growth and indirectly suggested the parent-divorce method at the fault tree level to generate a memory-efficient Bayesian network as part of the transformation process. They also noticed the exponential table growth of the k-out-of-n voting gate, but they did not provide a solution.

Iris and Kiureghian [138] have tackled the exponential memory growth problem of the k-out-of-n voting gate by proposing a lossless compression algorithm based on run-length encoding and Lempel Ziv compression to reduce the size of the CPT. Their evaluations indicated that their approach does not scale for systems with hundreds of redundant components. Moreover, their solution also requires a custom inference algorithm to handle the compression.

In the early beginnings of probabilistic graph modeling, knowledge engineers

introduced the notion of *causal independence* to design efficient Bayesian networks to ease knowledge acquisition and increase inference performance [116, 139, 140]. Casual independence is a property of the parent nodes describing their influence on the conditional probabilities of their child nodes. It assumes that each parent node has an individual contribution to the conditional probability distribution, independent of the remaining parent nodes.

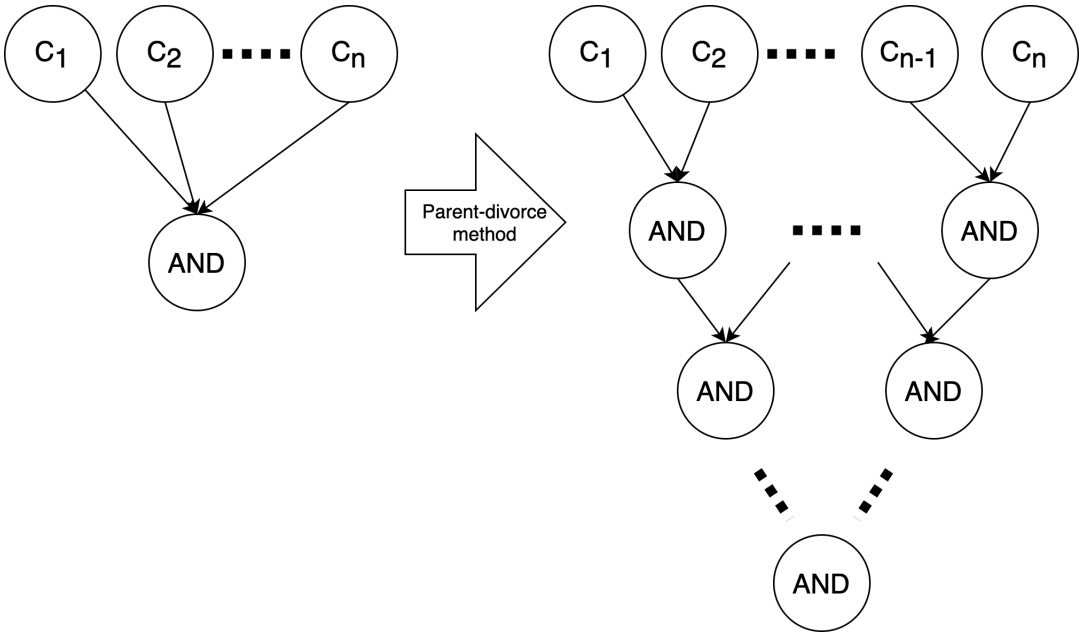


Figure 5.14.: The parent-divorce method applied to a converging AND node.

Efficient Bayesian network structures have already been proposed for the noisy AND/OR [116, 140–142], noisy MAX [143], and the noisy adder [144, 145] models by exploiting this causal independence property. In particular, Olesen et al. [139, 146] suggested the *parent-divorce method* as a solution to reduce the memory demand of the AND/OR models. The parent-divorce method is similar to the divide-and-conquer approach, where a converging node is split into smaller cascading nodes resulting in a lower number of parent nodes per child node. Each new node in the cascade structure implements the corresponding semantic depending on the original semantic of the initial node. For example, Figure 5.14 shows the resulting Bayesian network structure when applying the parent-divorce method to an AND model. As a result the final Bayesian network remains compatible with existing inference algorithms since only new nodes are added.

While the parent-divorce method works well for the AND/OR models, it is not apparent how to use this model to represent the k-out-of-n model. In contrast,

the temporal noisy adder proposed by Heckerman [135, 137] is more promising due to its much simpler representation and algorithmic realization. The temporal representation is a special case of the parent-divorce method. Instead of building a balanced tree structure, the temporal structure is a skewed tree, which eases construction and inference due to its pipeline-like processing of the causes.

5.5. Summary

The main goal of this chapter was to find a solution that mitigates the exponential growth of the Bayesian network availability model to enable the assessment of large-scale cloud applications. While there exists no general solution to this problem, this work found a solution for the class for redundant and voting-based replicated services, which are primarily implemented with the help of the k-out-of-n model. So far, the AND/OR models already have scalable representations; however, a scalable k-out-of-n model was missing. Therefore, this chapter tackled this problem and proposed a scalable k-out-of-n model based on the temporal noisy adder. The final solution effectively reduces the required space size of the Bayesian network from exponential to polynomial.

Empirical results showed a significant decrease in memory space while still computing the same availability and preserving its compatibility with standard inference algorithms. Nevertheless, experiments showed that the non-scalable Bayesian network models perform better for small service models with less than 13 instances. However, it is best to use the scalable model for large-scale services. Nonetheless, another important practical implication is that Bayesian network representations of static fault trees can now be implemented fully scalable since all basic gate types have scalable counterparts now. Hence, fault tree analysis can now profit from the scalable Bayesian network models to assess large-scale fault trees as well.

Chapter 6.

Service Recommendation System

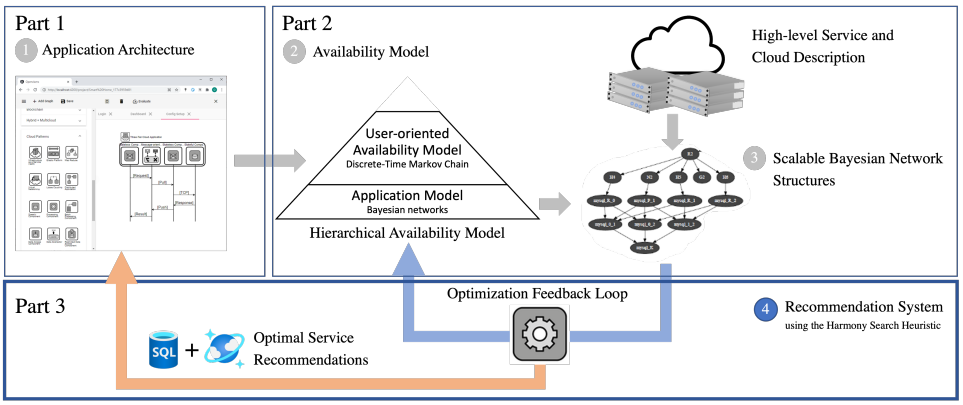


Figure 6.1.: This chapter focuses on part three of the solution workflow, proposing a service recommendation system to guide architectural design decisions.

This chapter introduces a service recommendation system to assist architectural design decisions by suggesting optimal cloud services that meet availability and cost constraints [63]. This forms the last part of the solution flow, as shown in Figure 6.1.

Section 6.1 introduces the problem statement. Afterward, Section 6.2 formulates the optimization problem. Next, Section 6.3 exemplifies the recommendation system with the help of the Harmony Search algorithm, analyzing the feasibility and performance of the approach based on the Azure case study. Section 6.4 discusses related work. Finally, Section 6.5 summarizes this chapter.

6.1. Introduction

Clams uses cloud computing patterns to express architectural intentions and to aid developers in implementing best practices. In order to compute the availabil-

ity of an architecture that consists of abstract components, the architecture needs to be refined to contain concrete services before any availability assessment can take place. Refining abstract components becomes complex when the refinement matches multiple services. This rises the question which matching services are best, when considering QoS and cost constraints? A simple solution would be to probe for all matches. However, due to the large number of today's service offerings, finding optimal services can become intractable for an exhaustive search, e.g., see the Azure case study summary in Table 2.1. Therefore, one might need approximation to find a near-to optimal solution in a feasible time. Researcher on web service composition face a similar problem when multiple web services match the same interface description. They propose the use of meta-heuristics to solve the web service composition problem [37, 40, 41, 45]. As a result, their findings inspired this work to use meta-heuristics to find optimal refinements in component-based architectures as used in the context of Clams.

The contribution of this chapter is the design and implementation of a refinement process that utilizes meta-heuristics to find optimal services for a given abstract architecture with subject to availability constraints. Clams encodes the design space of an abstract architecture in its corresponding refinement trees. Hence, necessary transformations are required to translate the design space of the potential service configurations into an appropriate input for the meta-heuristic. This work focuses on meta-heuristics that operate on loss functions where the main goal is to find a service configuration that minimizes the loss. Consequently, the loss function needs to encode the search objective.

In general, the refinement problem can be defined as follows in the context of this work. For a given usage profile $\mathcal{U}$ that consists of one or more scenarios $\boldsymbol{\mathcal{S}}$. We define an architecture as a vector of components $\boldsymbol{A} = \langle \mathcal{C}_1, \ldots, \mathcal{C}_n \rangle \in \boldsymbol{\mathcal{C}}^n$, where $\boldsymbol{A}$ is the union of all abstract and concrete service components in $\boldsymbol{\mathcal{S}}$. If an architecture has at least one abstract component, then the architecture is said to be *abstract*; otherwise, when all components are concrete services, then the architecture is said to be *concrete*. $\mathcal{L}$ is the set of all concrete architectures that result from the refinement of an abstract architecture $\boldsymbol{A}$. The main goal of the recommendation process is to find an architecture in $\mathcal{L}$ that minimizes a given loss-function L. However, since the solution space can become too large, we consider the problem of approximating the solution with the help of meta-heuristics, leading to the final problem definition:

$$\boldsymbol{A}_{min} \approx \arg\min_{\boldsymbol{A} \in \mathcal{L}}(L(\boldsymbol{A}))$$

Although this work emphasizes availability-driven design decisions, one can use the proposed recommendation process in the broader scope of QoS-driven service recommendations. In brief, this chapter shows how to transform the refinement trees into a suitable input for a search heuristic and define an appropriate QoS-aware loss function that minimizes operation cost. Evaluations will employ the Harmony Search algorithm [66] as a meta-heuristic to exemplify this recommendation approach and apply it to the Azure case study, searching for optimal services that are cost-minimal and above a given availability threshold.

6.2. Service Recommendation

This section describes the service recommendation system. It shows how to transform abstract components into an appropriate input for a search heuristic and express QoS-aware loss functions that minimize service cost. Finally, these sections introduce the Harmony Search algorithm as an implementation option to realize the recommendation system.

6.2.1. Overview

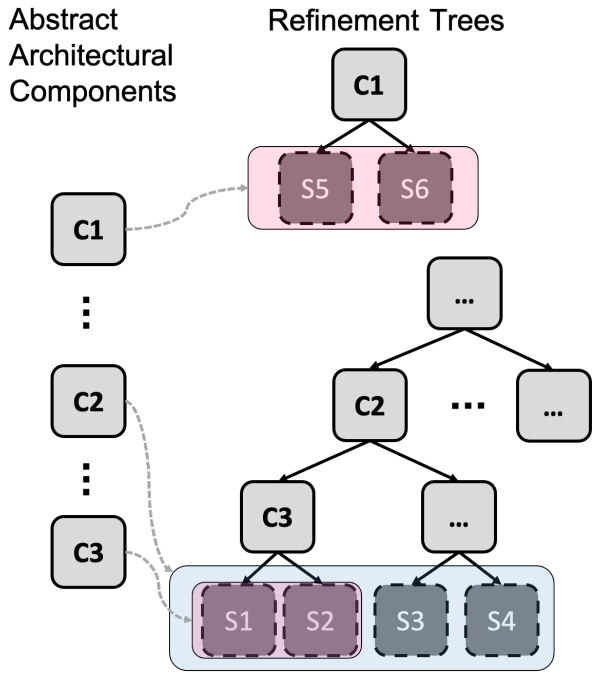


Figure 6.2.: Gathering the potential service configurations from the refinement trees.

This section provides an overview of the recommendation system at a high level. We assume a meta-heuristics takes in general as input a loss function and a set of variables, which is a collection of n variables $V = \{V_1, \ldots, V_n\}$ with a finite domain of values. In our case, variables correspond to abstract components containing concrete services (e.g. their component id reference) as values. Hence, the domain of a variable is the set of all concrete service components, which are the leaves in the corresponding refinement sub-trees. As a result, the overall solution space $\mathcal{L}$ is the Cartesian product $\mathcal{L} = \mathrm{dom}(V_1) \times \cdots \times \mathrm{dom}(V_n)$ of all the domains from the variables in V. For instance, Figure 6.2 shows a set of abstract components C1 to C3 of an architecture with their refinement trees. To build the value domains of the input variables V_{C1}, V_{C2}, and V_{C3}, we query the leaf nodes of the sub-trees accordingly. For C1, we traverse the first refinement tree; for C2, we traverse the second refinement tree starting at the second hierarchy level; and for C3, we traverse the same tree at the third level. As a result, the input for the meta-heuristic has the following values:

$$V_{C1} : \mathrm{dom}(V_{C1}) = \{S5, S6\}$$
$$V_{C2} : \mathrm{dom}(V_{C2}) = \{S1, S2, S3, S4\}$$
$$V_{C3} : \mathrm{dom}(V_{C3}) = \{S1, S2\}$$

Based on this input notion, we can design a recommendation system, as shown in Algorithm 6. The recommendation system takes as input an architecture $\boldsymbol{A} = \langle \mathcal{C}_1, \ldots, \mathcal{C}_n \rangle \in \boldsymbol{\mathcal{C}}^n$ with n components. It iterates over all components to collect their concrete services by querying the leaf nodes in the sub-trees using breadth-first search (line 5). Afterward, it provides the resulting list of input variables and a loss function to the search heuristic in order to compute the final solution $\boldsymbol{A}_{min} \in \mathcal{L}$, which only contains a set of concrete services. If some components in $\boldsymbol{A}$ are already concrete service components, then the leafNodesAt function will return that component as the sole solution for that variable. The remainder of this section will discuss how to design and implement the QoS-aware loss function L and provide more details on the SearchHeuristic with a focus on the Harmony Search algorithm.

6.2.2. QoS-aware Loss Function

To help developers design a suitable loss function, we discuss a class of loss functions that can be used as a framework to find cost-minimal services while adhering

Algorithm 6 High-level description of the service recommendation system.

1: **Input:** $\boldsymbol{A} = \langle C_1, \ldots, C_n \rangle$ ▷ Abstract architecture
2: $V := \langle\rangle$ ▷ Initialize empty list for variables
3: **for** $\forall C \in \boldsymbol{A}$: **do**
4: $V := [V, \langle V_C \rangle]$ ▷ Create new Variable V_C for C and append to V
5: $\mathrm{dom}(V_C) :=$ LEAFNODESAT(C) ▷ Add all concrete services based on the refinement tree of C to the domain of V_C
6: **end for**
7: $\boldsymbol{A}_{min} =$ SEARCHHEURISTIC(V,L)

to given QoS constraints. Algorithm 7 describes this class of loss function as a procedure consisting of three abstract methods that need to be implemented according to the specific model, which is Clams in our case. In detail, this procedure is called by the search heuristic and receives as input a concrete architecture $\boldsymbol{A}$ sampled from $\mathcal{L}$. The first step is to replace all components in the cloud application model under consideration, with the concrete service proposal in $\boldsymbol{A}$ (see line 4). Here, we use the abstract method SUBSTITUTE to indicate this replacement step. In the case of Clams, SUBSTITUTE would replace all component instances in the sequence message chart of each scenario with their corresponding concrete service in $\boldsymbol{A}$.

Next, the procedure takes the resulting (concrete) cloud application model as input to compute a QoS value with the EVALUATE method, which is, in our case, the hierarchical availability model presented in Section 3. Afterward, based on the cost of each component in $\boldsymbol{A}$, the procedure computes the total cost c of the model in line 6. At this stage, one might also consider more complex cost functions which take more properties into account. However, since components in Clams represent one specific configuration of a service offering with its cost, the procedure computes the total cost as the sum of the components.

Finally, the loss functions returns the cost c if the QoS value meet the QoS requirements (line 8), or, it returns *infinity* to rule out this set of concrete services if the requirements are not met. Here, the procedure defines a generic method CHECK, which compares the evaluation result with the given QoS requirements. This method needs to be implemented based on the according QoS metric. For example, suppose we wish to find a cost-minimal set of concrete services that result in an availability above a given availability threshold. In that case, we need to check if the availability result exceeds the threshold. Nevertheless, other

loss functions are also possible. Instead of returning a cost in line 8, a developer can use a different property or combination of multiple properties, including performance estimates and resource usage, to define different selection criteria.

Algorithm 7 Generic loss function for QoS-aware service selection with cost minimization.

```
1: procedure Loss(A)
2:     global model                                  ▷ Abstract architecture
3:     global QoSRequirements
4:     concreteModel := substitute(model, A)
5:     QoSValue := evaluate(concreteModel)
6:     c := ∑_{C∈A} cost(C)
7:     if check(QoSValue,QoSRequirements) then
8:         return c
9:     else
10:        return Infinity
11:    end if
12: end procedure
```

6.2.3. Harmony Search Algorithm

The search heuristic aims to approximate a solution $\boldsymbol{A} \in \mathcal{L}$. While multiple meta-heuristics are suitable for solving the refinement problem, this work uses the Harmony Search algorithm due to its algorithmic simplicity, small memory footprint, and a small number of tuning parameters as an example implementation to build the recommendation system. The Harmony Search algorithm is a meta-heuristic with an analogy in music improvisation [66]. Harmony Search mimics an orchestra that improvises *harmonies* until they find a fantastic harmony according to some aesthetic measure. Musicians improvise through pitch changes of their musical instruments, and while doing so, they remember only a finite number of good harmonies. In this analogy, music instruments translate to variables; each instrument can play a range of notes representing the possible domain of values. The aesthetic measure represents the loss function $L : \mathcal{L} \to \mathbb{R}$, which takes a harmony as input and computes a loss value. The description of the loss function represents the objective of the optimization problem.

The Harmony Search algorithm creates harmonies of length n, where n is the number of components in the architecture. Hence, a harmony is the set of concrete services. The Harmony Search algorithm does not consider the semantics of the values of its variables. This happens only in the loss function, which Harmony

Search generally regards as a black box. Therefore, developers need to provide an appropriate loss function to find a solution that meets their QoS and cost requirements.

The Harmony Search algorithm generates new harmonies by partially building them from previously calculated harmonies and randomly sampling new values. Algorithm 8 describes the Harmony Search algorithm in detail. The algorithm works as follows:

1. *Initialize harmony memory (HM).* A cache where the Harmony Search algorithm stores a collection of the best harmonies found during its execution (line 2). In the beginning, the HM has random harmonies. *The harmony memory size* (HMS) parameter defines the size of the HM.

2. *Improvise a new harmony.* Until the termination criteria is reached, the Harmony Search algorithm generates a new harmony by iterating through each variable and randomly choosing either an already used value from the HM or a random value from the whole domain of values. *The harmony memory consideration rate* (HMCR) $\in [0, 1]$ at line 7 defines the probability of selecting a value used in the HM or a value from the full domain (line 8 and 13). If the Harmony Search selects a value from the HM, it applies the pitch adjustment rate (PAR) $\in [0, 1]$ (line 9) to decide if a neighboring value (with fifty-fifty probability) should be chosen instead. After creating a new harmony, the Harmony Search algorithm computes the loss of the harmony.

3. *Test and Replace.* If the new harmony has a lower loss than the worst harmony in HM (line 19), the Harmony Search algorithm replaces the latter with the former to exclude the worst harmony from the HM.

4. *Termination.* Repeat steps 2 and 3 until the maximum number of iterations has been reached, and then return the harmony with minimal loss from the HM.

The Harmony Search algorithm has four parameters that influence the approximation quality: HMS, HMCR, PAR, and the maximum number of iterations. While the HMS defines the memory footprint, HMCR and PAR are probabilities that strive to escape local minima. We can also stop the algorithm anytime and retrieve the most current solution. However, the quality of the solution improves

Algorithm 8 Harmony Search Algorithm

```
1: procedure HarmonySearch(V,L)
2:     init HM[1 . . . HMS]                          ▷ Step 1: Initialize HM
3:     while not Terminated do
4:         harmony := new List(|V|)
5:                                                   ▷ Step 2: Create new harmony
6:         for each V_i ∈ V do
7:             if random() < HMCR then
8:                 v_i := randomFrom(HM[:,i])
9:                 if random() < PAR then
10:                    v_i := randomNeighbor(v_i)
11:                end if
12:            else
13:                v_i := randomFrom(dom(V_i))
14:            end if
15:            harmony.push(v_i)
16:        end for
17:                                                  ▷ Step 3: Update HM
18:        j := worstEntryIndex(HM)
19:        if L(harmony) < L(HM[j,:]) then
20:            HM[j] := harmony
21:        end if
22:    end while
23:                                                  ▷ Step 4: Return the best solution
24:    return arg min_{A∈HM} L(A)
25: end procedure
```

over time; the longer the execution, the higher the probability of finding an optimal solution. If all solutions are infeasible, then all harmonies in the memory will have a loss of infinity.

6.3. Evaluation and Discussion

This section evaluates the recommendation system with the help of the architectural examples from the Azure case study. This evaluation conducts a series of experiments to demonstrate the effectiveness and performance of the heuristic approach compared to the results of an exhaustive search for the optimal global solution.

6.3.1. Setup

All experiments were performed on a 64-bit machine with 64 Intel(R) Xeon(R) CPU E7-4850 v4 at 2.10GHz and 1 TB of main memory, running Arch Linux 5.13.12 with GCC 11.1.0, Python 3.9.6, and Numpy 1.20.3. The implementation of the Harmony Search and the exhaustive search algorithms are multi-threaded and use approximate inference to assess the corresponding Bayesian networks of the scenario models.

The Harmony Search algorithm requires four parameters. Instead of exhaustively searching for optimal parameters, this work will use frequently used parameter values from literature [41, 66], to minimize the parameter search space when performing hyperparameter optimization. Common values for the harmony memory size are the 10, 100, and 1000; for the HCR, the values 0.8, 0.85, 0.9, and 0.95; for the PAR, the values 0.05,0.1,0.2, and 0.4; and for the termination criteria, 1000, 5000, and 10000. As a result of the hyperparameter optimization based on a small subset of the architectural examples, this evaluation uses a harmony memory size of 10, an HCR value of 0.85, and a PAR value of 0.05. As for the termination criteria, the Harmony Search will stop after 10,000 iterations, i.e., after creating 10,000 samples.

This evaluation uses the architectural examples from the Azure case study presented in Section 2.5. Appendix A as well as Table 2.1 provide an overview of all architectures. The search objective is to find a set of concrete services with minimal operational cost while the overall availability estimation of the architectures is above a given threshold. All experiments will again use the simple and large cloud infrastructure models as described in Section 4.5. The threshold is highly

subjective and depends on the specific system at hand. Since the infrastructure models have different failure rates, the evaluation will use two different threshold values accordingly.

We will also consider failure probabilities for service instances for the large cloud infrastructure model. However, finding realistic failure probabilities is difficult since companies usually do not disclose availability information at the instance level. Nevertheless, Google has published the log traces of its Borg clusters, which we can use as guidance to roughly estimate the magnitude of the failure rates for individual instances [147]. Borg is a container orchestration system to schedule jobs in a compute cluster. Jobs can be part of any internal or external google service. Some jobs might be part of a MapReduce or PageRank execution, while other jobs represent continuous services like Gmail, or Google Docs [148]. The traces contain 29 days of log data with resource and scheduling information on about 24 million jobs. All jobs are anonymized and have no information about their concrete purpose. However, the data includes if jobs have finished successfully or failed. As a result, 1.3% of the jobs have failed during the trace period. Assuming that a job is the same as an instance in our model, we will sample the failure probability of instances again from a beta distribution with Beta$(100, 10000)$, which results in an average instance availability of 99.00% with a standard deviation of 0.09%.

All services in an architecture will have the same replication degree. Here, we will use two sets of experiments. The first set of experiments will have three replicas per service, and the second set will have fifteenth replicas per service. All instances are placed in round-robin across the hosts. Due to the relatively large failure rates of the simple cloud infrastructure model, we will use an availability threshold of 90% for those experiments. For the large infrastructure model, we will use an availability threshold of 99.50% respectively.

Algorithm 9 describes the corresponding loss function, using the framework shown in Algorithm 7. The function takes a set of concrete service components $\boldsymbol{A}$ and substitutes the corresponding components in a given Clams model with the help of the InsertServices method. Afterward, the loss function computes the availability according to the application model from Chapter 4, using the scalable Bayesian network structures from Chapter 5. Next, the computeCost method computes the operational cost, assuming the application is deployed in the same cloud region, i.e., Central US. Finally, the function checks if the availability exceeds the threshold.

Algorithm 9 Loss function to find a cost-minimal service combination for a given availability threshold.

```
1: procedure Loss(A)
2:     global ClamsModel
3:     global threshold
4:     concreteModel := InsertServices(S,ClamsModel)
5:     availability := computeAvailability(concreteModel)
6:     c := ∑_{C∈A} cost(C)
7:     if availability > threshold then
8:         return c
9:     else
10:        return Infinity
11:    end if
12: end procedure
```

6.3.2. Search Results

Table 6.1 summarizes the evaluation result by comparing the Harmony Search algorithm with the results of an exhaustive search. It shows the architecture name, the number of components, and the solution size, which is also the sorting order. Furthermore, it contains four columns that summarize the outcome of the experiments. Each column indicates if the evaluation used the *Simple* or *Large* infrastructure and the number of replicas per service. Each experiment was first performed with the search heuristic and afterward compared to the optimal result found by the exhaustive search. Due to tractability reasons, the evaluation only performed the exhaustive search for architectures with a search space size of less than ten million. All those architectures with larger search spaces or an execution time longer then 24h have a (-) minus sign. We repeated each experiment five times. A check mark indicates that the search heuristic has returned the same result as the exhaustive search for all repetitions, i.e., the Harmony search algorithm has returned a service combination with (globally) minimal cost. A cross indicates that the search heuristic has returned a different result as the exhaustive search repeatedly.

What stands out in the table is that for almost all architectures, where an exhaustive search was possible, the search heuristic's results matches the exhaustive search results. However, the Harmony Search has a termination criteria of 10.000 iterations. Thus, it is no surprise that it has found an optimal service combination for all architectural examples with a solution size smaller than the maximum number of search iterations. What stands out are those architectures

Table 6.1.: Summary of the Azure case study.

Short	# Services/ Abstract Components	# Solutions	Simple 3 Replicas	Simple 15 Replicas	Large 3 Replicas	Large 15 Replicas
App 1	2	2	✓	✓	✓	✓
App 2	2	5	✓	✓	✓	✓
App 3	2	131	✓	✓	✓	✓
App 4	3	180	✓	✓	✓	✓
App 5	2	690	✓	✓	✓	✓
App 6	2	690	✓	✓	✓	✓
App 7	3	8460	✓	✓	✓	✓
App 8	4	42000	✓	✓	✓	✓
App 9	3	76608	✓	✓	✓	✓
App 10	3	188640	✓	✓	✓	✓
App 11	2	220800	✓	✓	✓	✓
App 12	5	340200	✓	✓	✓	-
App 13	4	414000	✓	✓	✓	-
App 14	2	666729	✓	x	x	-
App 15	5	4.0×10^6	✓	-	-	-
App 16	3	6.4×10^6	✓	-	-	-
App 17	4	1.8×10^7	-	-	-	-
App 18	5	2.0×10^7	-	-	-	-
App 19	6	4.6×10^7	-	-	-	-
App 20	4	2.1×10^8	-	-	-	-
App 21	5	5.5×10^8	-	-	-	-
App 22	6	7.4×10^8	-	-	-	-
App 23	5	9.3×10^9	-	-	-	-
App 24	7	7.4×10^{10}	-	-	-	-
App 25	6	9.4×10^{11}	-	-	-	-
App 26	8	1.1×10^{12}	-	-	-	-
App 27	7	1.1×10^{12}	-	-	-	-
App 28	7	2.3×10^{12}	-	-	-	-
App 29	7	2.2×10^{13}	-	-	-	-
App 30	8	2.7×10^{15}	-	-	-	-
App 31	8	2.7×10^{15}	-	-	-	-

✓ Result of Harmony Search is equal to the exhaustive search.
- Exhaustive search evaluation was not possible.
x Result of Harmony Search is not equal to the exhaustive search.

with a solution size of a hundred thousand and more service combinations. Here, two experiments were unsuccessful on the first run, but subsequent repetitions returned the same results as the exhaustive search. Nevertheless, even when the Harmony Search did not find the optimal solution, the (monetary) loss of the search heuristic was two dollars off compared in those particular examples to the optimal loss from the exhaustive search.

6.3.3. Performance Analysis

So far, the results of the search heuristic are promising and have found the global minimum solution for most architectures. Next, we analyze the execution time between small and large application models, to identify the factors that influence performance.

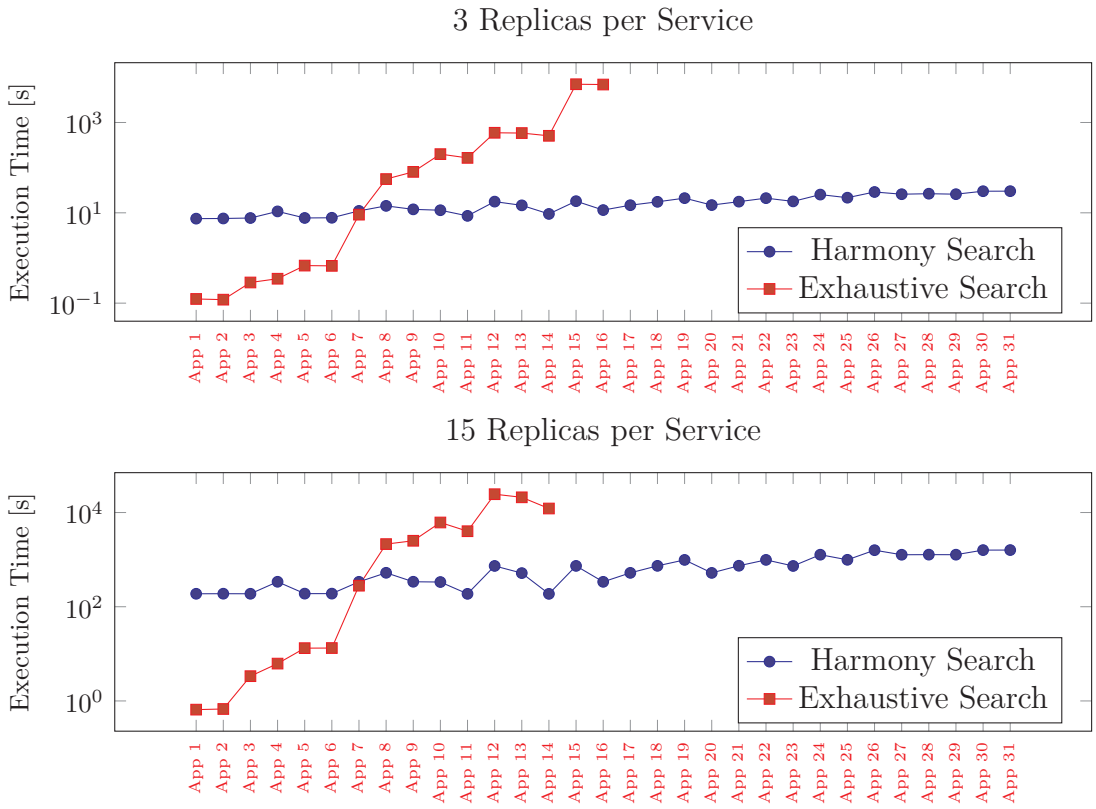


Figure 6.3.: Execution Time for finding an optimal solution for the simple architectural examples with three and fifteen replicas per service

Figure 6.3 shows the execution time of the Harmony search algorithm compared to the exhaustive search, using the simple infrastructure for three and 15 replicas per service. App 7 has a solution size of almost ten thousand service combinations, which is why it is the break-even point where the exhaustive search and the Harmony search algorithm have a similar execution time. The execution time of one iteration is proportional to the size of the Bayesian network model. Hence, when comparing App 12, which has five components, with App 14, which has two components, the execution time for the exhaustive search and Harmony Search algorithm of App 14 is less than the time for App 12. App 12 has a significantly larger Bayesian network model to evaluate than App 14, which in turn requires more time per iteration for the availability prediction of one service combination. So, although App 14 has a larger solution space than App 12, the Harmony Search and the exhaustive search were faster for App 14.

What stands out in the figure is that the execution time for services with 15 replicas is one order of magnitude higher than the same architectures with three

replicas per service. The Bayesian network model of a service with three replicas has nine channel nodes per service, whereas the corresponding service with 15 replicas has 210 channel nodes per service, which concludes the difference in magnitude of the execution time. The Harmony Search algorithm takes less than three minutes to evaluate the largest architecture (App 31). In contrast, the models with the simple infrastructure example required at most 30 seconds to return a solution. As a result, even for the largest architecture, the execution time can provide a near to real-time user experience, which is important when developers wish to evaluate multiple architectural alternatives in a timely manner.

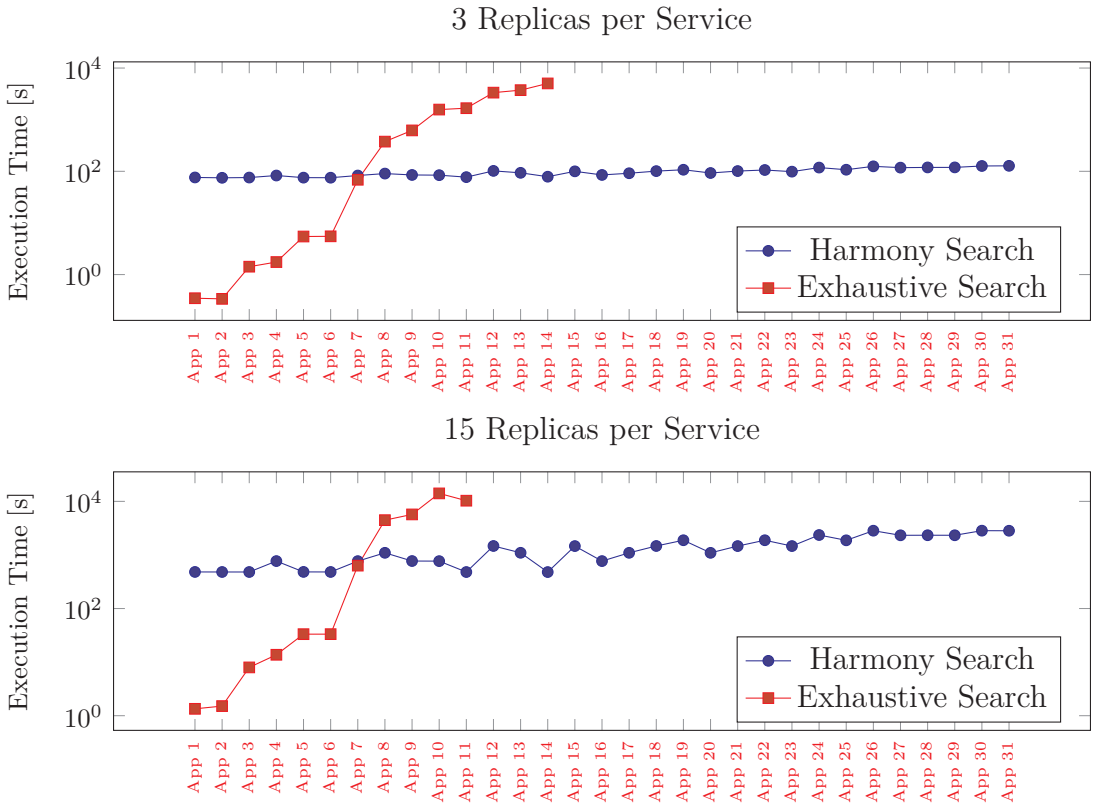


Figure 6.4.: Execution Time for finding an optimal solution for the large architectural examples with three and fifteen replicas per service

Figure 6.4 shows the repetition of the experiments shown in Figure 6.3, using the large infrastructure instead. The results show that the execution time has increased by one order of magnitude compared to the same experiments that used the simple infrastructure. The number of channel nodes is the same as for the models with the simple infrastructure, so the additional infrastructure components cause the increase in execution time.

In summary, comparing architectures with three replicas per service with 15 replicas per service shows that the replication degree is the primary performance driver, followed by the number of services and the infrastructure size. In all experiments, the Harmony Search algorithm used ten thousands iterations. Lowering the number of iterations reduces the execution time, but also the chances decrease to find the best service matches. Users might need to repeat the Harmony search algorithm several times and select the most frequent result in that case. Overall, this evaluation shows that the Harmony Search algorithm can find stable results by using ten thousands iterations for this particular architectural examples.

6.4. Related Work

Research on artificial intelligence has introduced many meta-heuristics such as tabu search [46], evolutionary algorithms [149, 150], or ant colony optimization [43, 151]; all of which are valid solutions for the here proposed refinement problem. Nevertheless, the Harmony Search algorithm [66] has successfully been applied to solve the web service composition problem [37, 40, 45]. Bekkouche et al. [41] provided a detailed performance evaluation on the Harmony Search algorithm and its various improvements for the web service composition problem, showing how to match semantic web services and create new compositions by considering multi-dimensional QoS requirements. As a result, their findings inspired the use of the Harmony Search algorithm as a concrete implementation to solve the refinement problem in this work.

Karpova and Buhnova [152] provide concepts on performance-driven architectural refinement within the context of PCM [25]. In the case of multiple refinement matches, they suggest using the solution with the best performance measure. They also mention the use of search heuristics to find solutions for more complex quality evaluations, but they did not present concrete examples or implementation suggestions.

A large body of literature has investigated patterns and pattern refinement in software architecture composition [42, 82, 94, 95]. Previous research [77, 90, 91] uses architectural refinements for pattern-based deployment models, but their selection mechanism takes the first concrete solution that fits the pattern. This does not necessarily mean it is optimal with regard to cost or QoS. Specifically, Harzenetter et al. [90, 96] use Cloud Computing Patterns [64] and Enterprise Integration Patterns [81] as abstract concepts to express agnostic deployment models. They introduce Pattern Refinement Models (PRMs), representing the

mapping from patterns to concrete deployment plans. So far, the refinement algorithm uses the first PRM that fits the pattern. Hence, the matching approach does not consider properties like QoS in its selection process.

6.5. Summary

In summary, this chapter introduced a service recommendation system to propose optimal services that minimizes a given QoS-aware search objective. The refinement step, which transforms an abstract architecture into a concrete architecture, might result in multiple possible solutions. This chapter showed how to translate the refinement solutions encoded in refinement trees into a suitable input for the search algorithm and introduced a framework for loss functions to aid developers in designing custom search objectives that minimize service cost while adhering to QoS requirements. The evaluation used the Harmony Search algorithm and applied it to the architectural examples from the Azure case study to exemplify and validate the accuracy of the approach. Comparing the results with the optimal solution of an exhaustive search showed that the Harmony Search algorithm finds the global optimum for all the architectures where an exhaustive search was possible and provides a significant performance advantage.

Chapter 7.

Conclusion and Outlook

In summary, this thesis addressed numerous challenges on availability prediction and availability-driven service recommendation for cloud computing. As a result, the contribution of this work is four-fold. First, this work introduced a novel cloud modeling language called Clams. Developers can use Clams to describe their architectural intentions and communicate these to the cloud provider for availability prediction. It is a scenario-based cloud modeling language that utilizes cloud computing patterns to design architectures according to best practices and express architectural design uncertainties. Developers can define usage profiles in Clams to describe the behavior of users and how they interact with scenarios. Scenarios represent application-level functionalities that define the sub-architecture consisting of cloud services necessary to implement the respective functionality.

Second, this work showed how to implement a hierarchical availability model to predict the availability of a cloud application as a function of the user behavior defined in the usage profile. This so called user-oriented availability model uses DTMC as a root model to express the availability of executing an usage profile successfully and a Bayesian network availability model at the lower level to compute the availability of the individual scenarios that constitute the usage profile. The Bayesian network availability model unifies the fault aspects, derived from a higher level description of the scenario that consists of three sub-models. These sub-models are a fault dependency graph to express the failure relations between cloud services and the execution environment, a network model to address communication and network partition failures, and a service model to define availability requirements as function of the service instances. Evaluations showed the feasibility of the Bayesian network approach to represent large applications with hundreds of fault dependencies. However, the replication degree of cloud services are constrained to exponential grow of memory size posed by the Bayesian network formalism in conjunction with the Bayesian network representation of

the k-out-of-n voting gate model.

Consequently, this work presents a scalable k-out-of-n model as a third contribution to express large replicated cloud services with Bayesian networks. The scalable k-out-of-n model is based on the temporal noisy adder model, which effectively reduces the required space size of the Bayesian network from exponential to polynomial.

Finally, this thesis introduced a service recommendation system to assist architectural design decisions by suggesting optimal cloud services that meet availability and cost constraints. Developers that cultivate an availability or QoS-driven mindset develop their architecture by trying alternative design variants to find cloud services that best fit their QoS requirements. However, nowadays, the large number of service offerings makes it almost impossible to perform an exhaustive search to check all configuration possibilities. Hence, this work introduced concepts to utilize meta-heuristics in combination with Clams to guide architectural decisions by proposing optimal cloud services. Here, we use the Harmony Search algorithm to exemplify the main concepts and applied our findings to a set of real-live architectural examples to demonstrate its feasibility.

There are several directions on how to extend this work. The remainder of this chapter provides an outlook on potential short and long-term topics for the future.

- Implement a Clams2TOSCA extension to deploy cloud architectures after the refinement process has found an optimal service configuration. TOSCA is already a well-established standard for cloud application orchestration, for which tooling support already exists [54, 153]. Hence, it is more advantageous to translate Clams to TOSCA and use its existing deployment engines.

- Consider the effects of long-running requests and their implications on component failures and recoveries. Long-running requests introduce new temporal aspects, which would require a dynamic Bayesian network approach to model the time dimension, which results in larger and more complex models. Consequently, an increase in model complexity might require scalable Bayesian network representations for dynamic fault tree gates. Dynamic fault trees provide new gates that have temporal properties. While dynamic Bayesian network solutions exist for dynamic fault trees, scalable implementations for the new gates do not exist.

- Predict the placement of cloud services within the infrastructure. In cloud

computing, modern placement schedulers often use explicit rules to constrain the placement of instances to decrease the risk of common cause failures. For example, a constraint might be to place instances or replicas in different availability zones or data center sites. The goal is to use the knowledge of the rules to predict which placements are likely to occur. This opens the un-tapped potential to determine the average availability by predicting the most probable placements. Or, if the worst-case availability is of interest, search for viable placements according to the rules that lead to a high number of shared components between processes.

Appendix A.

Azure Case Study

This chapter introduces all 31 architectural examples in detail. A crawler application extracted the architectures from the Microsoft Azure Documentation. The documentation offers 385 examples [1]. Only the here presented examples had an identifiable component list. However, due to the unstructured architectural descriptions, there is no guarantee that all components were captured in all examples. The here presented architectures consists of a title, the web source, a brief description, the number of potential scenarios when the example has a data flow description, and a table with the components. The table contains the component name, the number of potential service candidates that match the components if its abstract, the search depth when the breadth-first search algorithm searches for the leaf nodes, and the total number of traversal steps. The here presented data was last verified on November 3th, 2021.

App 1: Speech Services

Source:
`https://docs.microsoft.com/en-us/azure/architecture/solution-ideas/articles/speech-services`

Description:

An application to transcribe and analyze calls.

Number of Scenarios: 7

Components:

Component Name	# Matching services	RT Depth	Traversing Steps
Storage Accounts Managed Disks Premium Blob Storage	2	2	3
Azure Cognitive Services Speech Services	2	2	3

[1] https://docs.microsoft.com/en-us/azure/architecture/browse/

App 2: Unlock Legacy Data with Azure Stack

Source:
`https://docs.microsoft.com/en-us/azure/architecture/solution-ideas/articles/unlock-legacy-data`

Description:

A solution to extend on-premise legacy applications to the cloud.

Number of Scenarios: 4

Components:

Component Name	# Matching services	RT Depth	Traversing Steps
Virtual Network	5	2	6
VPN Gateway	1	0	0

App 3: HPC System and Big Compute Solutions

Source:
`https://docs.microsoft.com/en-us/azure/architecture/solution-ideas/articles/big-compute-with-azure-batch`

Description:

A solution for batch processing of large tasks like image rendering, or large scale simulations.

Number of Scenarios: 6

Components:

Component Name	# Matching services	RT Depth	Traversing Steps
Storage Accounts	131	6	222
Batch	1	0	0

App 4: SMB disaster recovery with Double-Take DR

Source:
`https://docs.microsoft.com/en-us/azure/architecture/solution-ideas/articles/disaster-recovery-smb-double-take-dr`

Description:

Disaster recovery of on-premise VMs and database services to the cloud with Double-Take DR.

Number of Scenarios: 1

Components:

Component Name	# Matching services	RT Depth	Traversing Steps
Traffic Manager	36	3	52
VPN Gateway	1	0	0
Virtual Network	5	2	6

App 5: Loan Credit Risk + Default Modeling

Source:
`https://docs.microsoft.com/en-us/azure/architecture/solution-ideas/articles/loan-credit-risk-analyzer-and-default-modeling`

Description:

An application for credit-risk analysis using machine learning to predict credit scores.

Number of Scenarios: unspecified

Components:

Component Name	# Matching services	RT Depth	Traversing Steps
Azure Machine Learning	115	1	115
Power BI Embedded	6	1	6

App 6: Predicting Length of Stay in Hospitals

Source:
`https://docs.microsoft.com/en-us/azure/architecture/solution-ideas/articles/predicting-length-of-stay-in-hospitals`

Description:

Prediction service to estimate length of stay in hospitals.

Number of Scenarios: unspecified

Components:

Component Name	# Matching services	RT Depth	Traversing Steps
Azure Machine Learning	115	1	115
Power BI Embedded	6	1	6

App 7: Sharing location in real time using low-cost serverless Azure services

Source:
`https://docs.microsoft.com/en-us/azure/architecture/example-scenario/signalr/`

Description:

A general solution to implement real-time messaging services to share live location, push notifications to mobile devices, or offer chat services.

Number of Scenarios: unspecified

Components:

Component Name	# Matching services	RT Depth	Traversing Steps
Service Bus	282	3	375
Azure SignalR Service	2	1	2
Azure Functions	15	2	17

App 8: Retail and e-commerce using Cosmos DB

Source:
`https://docs.microsoft.com/en-us/azure/architecture/solution-ideas/articles/retail-and-e-commerce-using-cosmos-db`

Description:

An e-commerce application with advanced product search options.

Number of Scenarios: unspecified

Components:

Component Name	# Matching services	RT Depth	Traversing Steps
Hybrid Multimedia Web Application	1	0	0
Azure Cosmos DB	100	4	154
Azure Data Lake Storage Gen1	60	3	80
Azure Cognitive Search	7	1	7

App 9: DevTest Image Factory

Source:
`https://docs.microsoft.com/en-us/azure/architecture/solution-ideas/articles/dev-test-image-factory`

Description:

Organizes images to assist teams in application development according to compliance and security requirements.

Number of Scenarios: 6

Components:

Component Name	# Matching services	RT Depth	Traversing Steps
Azure Lab Services	8	1	8
Virtual Machines	1197	3	1205
Azure DevOps	8	2	10

App 10: Tier Applications & Data for Analytics

Source:
https://docs.microsoft.com/en-us/azure/architecture/solution-ideas/articles/tiered-data-for-analytics

Description:

An best practice implementation of the three-tier architecture for the Azure Cloud.

Number of Scenarios: 7

Components:

Component Name	# Matching services	RT Depth	Traversing Steps
Storage Accounts	131	6	222
Azure Functions	15	2	17
Azure Stack Edge	96	6	189

App 11: Archive on - premises data to cloud

Source:
https://docs.microsoft.com/en-us/azure/architecture/solution-ideas/articles/backup-archive-on-premises

Description:

A solution to store back up data to the cloud.

Number of Scenarios: 1

Components:

Component Name	# Matching services	RT Depth	Traversing Steps
StorSimple	384	7	574
Blob Storage	575	8	879

App 12: Container CI / CD (...) on Azure Kubernetes Service(AKS)

Source:
https://docs.microsoft.com/en-us/azure/architecture/solution-ideas/articles/container-cicd-using-jenkins-and-kubernetes-on-azure-container-service

Description:

An solution that setups build environments to continuously build and release applications.

Number of Scenarios: 9

Components:

Component Name	# Matching services	RT Depth	Traversing Steps
Azure Kubernetes Service (AKS)	378	1	378
Container Registry	3	1	3
Azure Cosmos DB	100	4	154
Azure Monitor	3	1	3
Visual Studio Codespaces	1	0	0

App 13: SMB disaster recovery with Azure Site Recovery

Source:

`https://docs.microsoft.com/en-us/azure/architecture/solution-ideas/articles/disaster-recovery-smb-azure-site-recovery`

Description:

An solution for disaster recovery of on-premise VMs and database services to the cloud.

Number of Scenarios: 1

Components:

Component Name	# Matching services	RT Depth	Traversing Steps
Traffic Manager	36	3	52
Azure Site Recovery	4	2	6
Virtual Network	5	2	6
Blob Storage	575	8	879

App 14: Adding a mobile front - end to a legacy app

Source:

`https://docs.microsoft.com/en-us/azure/architecture/solution-ideas/articles/adding-a-modern-web-and-mobile-frontend-to-a-legacy-claims-processing-application`

Description:

A web and mobile front end solution to display data aggregated form one or more different business systems.

Number of Scenarios: 8

Components:

Component Name	# Matching services	RT Depth	Traversing Steps
Virtual Machines	1197	3	1205
Azure SQL Database	557	9	662

App 15: Custom Data Sovereignty & Data Gravity Requirements

Source:

`https://docs.microsoft.com/en-us/azure/architecture/solution-ideas/articles/data-sovereignty-and-gravity`

Description:

Solution to securely send and store data according to company policies.

Number of Scenarios: 3

Components:

Component Name	# Matching services	RT Depth	Traversing Steps
Virtual Network	5	2	6
VPN Gateway	1	0	0
Azure SQL Database	557	9	662
Azure Functions	15	2	17
Azure Stack Edge	96	6	189

App 16: Design Review Powered by Mixed Reality

Source:
`https://docs.microsoft.com/en-us/azure/architecture/solution-ideas/articles/collaborative-design-review-powered-by-mixed-reality`

Description:

An application that realizes the collaboration between users that use Microsoft HoloLens to work 3D holograms together.

Number of Scenarios: 10

Components:

Component Name	# Matching services	RT Depth	Traversing Steps
Azure Active Directory	112	3	196
Blob Storage	575	8	879
Azure Cosmos DB	100	4	154

App 17: Demand Forecasting + Price Optimization

Source:
`https://docs.microsoft.com/en-us/azure/architecture/solution-ideas/articles/demand-forecasting-price-optimization-marketing`

Description:

A solution for demand forecasting in retail to automatically adapt and optimize pricing of products.

Number of Scenarios: 3

Components:

Component Name	# Matching services	RT Depth	Traversing Steps
Azure Data Lake Storage Gen1	60	3	80
HDInsight	1224	4	1286
Data Factory	40	4	44
Power BI Embedded	6	1	6

App 18: Defect prevention with predictive maintenance

Source:

`https://docs.microsoft.com/en-us/azure/architecture/solution-ideas/articles/defect-prevention-with-predictive-maintenance`

Description:

Failure prediction with machine learning, based on real-time assembly line data.

Number of Scenarios: unspecified

Components:

Component Name	# Matching services	RT Depth	Traversing Steps
Azure Stream Analytics	2	1	2
Event Hubs	60	3	75
Azure Machine Learning	115	1	115
Azure Synapse Analytics	242	5	440
Power BI Embedded	6	1	6

App 19: Enterprise-scale disaster recovery

Source:

`https://docs.microsoft.com/en-us/azure/architecture/solution-ideas/articles/disaster-recovery-enterprise-scale-dr`

Description:

A solution where on-premise services use the Azure Cloud as failover.

Number of Scenarios: 1

Components:

Component Name	# Matching services	RT Depth	Traversing Steps
Traffic Manager	36	3	52
Azure Site Recovery	4	2	6
Blob Storage	575	8	879
Azure Active Directory	112	3	196
VPN Gateway	1	0	0
Virtual Network	5	2	6

App 20: Hybrid ETL with Azure Data Factory

Source:

`https://docs.microsoft.com/en-us/azure/architecture/example-scenario/data/hybrid-etl-with-adf`

Description:

A hybrid Extract, Transform and Load (ETL) application that also uses on-premises ETL services in the enterprise.

Number of Scenarios: 5

Components:

Component Name	# Matching services	RT Depth	Traversing Steps
Blob Storage	575	8	879
Data Factory Azure Data Factory V2 SQL Server Integration Services	38	2	40
Data Factory Azure Data Factory V2	39	3	42
Azure Synapse Analytics	242	5	440

App 21: Discovery Hub with Cloud Scale Analytics

Source:
`https://docs.microsoft.com/en-us/azure/architecture/solution-ideas/articles/cloud-scale-analytics-with-discovery-hub`

Description:

An application to analyze data lakes, build reports, and visualize the results in a dashboard.

Number of Scenarios: 5

Components:

Component Name	# Matching services	RT Depth	Traversing Steps
Azure Data Lake Storage Gen1	60	3	80
Azure Databricks	288	3	297
Azure Synapse Analytics	242	5	440
Azure Analysis Services	22	2	33
Power BI Embedded	6	1	6

App 22: Modern Data Warehouse Architecture

Source:
`https://docs.microsoft.com/en-us/azure/architecture/solution-ideas/articles/modern-data-warehouse`

Description:

A data warehouse solution supporting ETL and analytical services.

Number of Scenarios: 4

Components:

Component Name	# Matching services	RT Depth	Traversing Steps
Azure Synapse Analytics	242	5	440
Data Factory	40	4	44
Storage Accounts Managed Disks Premium Blob Storage	2	2	3
Azure Databricks	288	3	297
Azure Analysis Services	22	2	33
Power BI Embedded	6	1	6

App 23: Master Data Management powered by CluedIn

Source:

`https://docs.microsoft.com/en-us/azure/architecture/reference-architectures/data/cluedin`

Description:

An application for policy conform data management based on CluedIn.

Number of Scenarios: 8

Components:

Component Name	# Matching services	RT Depth	Traversing Steps
Azure SQL Database	557	9	662
Azure SQL Managed Instance	71	6	100
Azure Cosmos DB	100	4	154
Azure Data Lake Storage Gen1	60	3	80
Data Factory Azure Data Factory V2	39	3	42

App 24: Advanced Analytics Architecture

Source:

`https://docs.microsoft.com/en-us/azure/architecture/solution-ideas/articles/advanced-analytics-on-big-data`

Description:

An application to create and orchestrate custom machine learning models to derive insights from data at scale.

Number of Scenarios: 6

Components:

Component Name	# Matching services	RT Depth	Traversing Steps
Azure Synapse Analytics	242	5	440
Data Factory	40	4	44
Storage Accounts Managed Disks Premium Blob Storage	2	2	3
Azure Databricks	288	3	297
Azure Cosmos DB	100	4	154
Azure Analysis Services	22	2	33
Power BI Embedded	6	1	6

App 25: Anomaly Detector Process

Source:
https://docs.microsoft.com/en-us/azure/architecture/solution-ideas/articles/anomaly-detector-process

Description:

Detecting and monitoring abnormalities in time series data.

Number of Scenarios: 6

Components:

Component Name	# Matching services	RT Depth	Traversing Steps
Service Bus	282	3	375
Azure Databricks	288	3	297
Power BI Embedded	6	1	6
Storage Accounts	131	6	222
Azure Cognitive Services	34	2	47
Azure Logic Apps	432	4	518

App 26: Personalized marketing solutions

Source:
https://docs.microsoft.com/en-us/azure/architecture/solution-ideas/articles/personalized-marketing

Description:

A data analysis solution with recommendation engine to offer personalizing offers to clients.

Number of Scenarios: unspecified

Components:

Component Name	# Matching services	RT Depth	Traversing Steps
Event Hubs	60	3	75
Azure Stream Analytics	2	1	2
Azure Cosmos DB	100	4	154
Storage Accounts	131	6	222
Azure Functions	15	2	17
Azure Machine Learning	115	1	115
Azure Cache for Redis	68	3	80
Power BI Embedded	6	1	6

App 27: Quality Assurance

Source:
https://docs.microsoft.com/en-us/azure/architecture/solution-ideas/articles/quality-assurance

Description:

A solution for assembly lines to predict manufacturing failures, based on machine learning with data stemming from testing systems and domain knowledge.

Number of Scenarios: 8

Components:

Component Name	# Matching services	RT Depth	Traversing Steps
Event Hubs	60	3	75
Azure Stream Analytics	2	1	2
Azure Machine Learning	115	1	115
Storage Accounts	131	6	222
Azure Logic Apps	432	4	518
Azure Synapse Analytics	242	5	440
Power BI Embedded	6	1	6

App 28: Predictive Aircraft Engine Monitoring

Source:
`https://docs.microsoft.com/en-us/azure/architecture/solution-ideas/articles/aircraft-engine-monitoring-for-predictive-maintenance-in-aerospace`

Description:

An application for maintenance prediction and health monitoring of aircraft, based on time series data and machine learning at scale.

Number of Scenarios: unspecified

Components:

Component Name	# Matching services	RT Depth	Traversing Steps
Azure Stream Analytics	2	1	2
Event Hubs	60	3	75
Azure Machine Learning	115	1	115
HDInsight	1224	4	1286
Azure SQL Database	557	9	662
Data Factory	40	4	44
Power BI Embedded	6	1	6

App 29: Build web and mobile applications

Source:
`https://docs.microsoft.com/en-us/azure/architecture/solution-ideas/articles/webapps`

Description:

A microservice-based e-commerce solution with analytical platform and recommendation engine

Number of Scenarios: unspecified

Components:

Component Name	# Matching services	RT Depth	Traversing Steps
Azure Database for MySQL	82	4	116
Azure Cosmos DB	100	4	154
Azure Cache for Redis	68	3	80
Azure Kubernetes Service (AKS)	378	1	378
Event Hubs	60	3	75
Azure Databricks	288	3	297
Power BI Embedded	6	1	6

App 30: Predictive Insights with Vehicle Telematics

Source:
`https://docs.microsoft.com/en-us/azure/architecture/solution-ideas/articles/predictive-insights-with-vehicle-telematics`

Description:

An application to predict driver habits and vehicle health based on diagnostic events and information from vehicle telematics.

Number of Scenarios: unspecified

Components:

Component Name	# Matching services	RT Depth	Traversing Steps
Event Hubs	60	3	75
Azure Stream Analytics	2	1	2
Azure Machine Learning	115	1	115
Storage Accounts	131	6	222
HDInsight	1224	4	1286
Data Factory	40	4	44
Azure Synapse Analytics	242	5	440
Power BI Embedded	6	1	6

App 31: Real Time Analytics on Big Data Architecture

Source:
`https://docs.microsoft.com/en-us/azure/architecture/solution-ideas/articles/real-time-analytics`

Description:

An application for near-real time data processing of event streams.

Number of Scenarios: 7

Components:

Component Name	# Matching services	RT Depth	Traversing Steps
Azure Synapse Analytics	242	5	440
Data Factory	40	4	44
Azure Data Lake Storage Gen1	60	3	80
Azure Databricks	288	3	297
HDInsight	1224	4	1286
Azure Cosmos DB	100	4	154
Azure Analysis Services	22	2	33
Power BI Embedded	6	1	6

Bibliography

[1] K. Clay, "Amazon.com Goes Down, Loses $ 66,240 Per Minute," https://www.forbes.com/sites/kellyclay/2013/08/19/amazon-com-goes-down-loses-66240-per-minute/, Aug. 2013, [Online; accessed 11-Okt-2021].

[2] Amazon Web Services, Inc., "Summary of the Amazon Kinesis Event in the Northern Virginia (US-EAST-1) Region," https://aws.amazon.com/de/message/11201/, Nov. 2020, [Online; accessed 12-Okt-2021].

[3] M. Rosemain and R. Satter, "Millions of websites offline after fire at French cloud services firm," https://www.reuters.com/article/us-france-ovh-fire-idUSKBN2B20NU, Mar. 2021, [Online; accessed 12-Okt-2021].

[4] Google, Inc., "Google Cloud Status Dashboard," https://status.cloud.google.com/products/kchyUtnkMHJWaAva8aYc/history, [Online; accessed 12-Okt-2021].

[5] S. Janardhan, "Update about the october 4th outage," https://engineering.fb.com/2021/10/04/networking-traffic/outage/, Oct. 2021, [Online; accessed 12-Okt-2021].

[6] A. Brown, "Facebook Lost About $65 Million During Hours-Long Outage," https://www.forbes.com/sites/abrambrown/2021/10/05/facebook-outage-lost-revenue/, Oct. 2021, [Online; accessed 12-Okt-2021].

[7] P. Bailis and K. Kingsbury, "The Network is Reliable," *Queue*, vol. 12, no. 7, pp. 20–32, jul 2014. doi: 10.1145/2639988.2655736

[8] "Iso/iec/ieee international standard - systems and software engineering–vocabulary," *ISO/IEC/IEEE 24765:2017(E)*, pp. 1–541, 2017. doi: 10.1109/IEEESTD.2017.8016712

[9] J. D. Musa, *Software Reliability Engineering: More Reliable Software Faster and Cheaper*. Authorhouse, 2004. ISBN 1418493880

[10] M. S. Hamada, A. G. Wilson, C. S. Reese, and H. F. Martz, *Bayesian Reliability*. Springer New York, 2008.

[11] K. S. Trivedi, *Probability and Statistics with Reliability, Queuing and Computer Science Applications*. John Wiley & Sons, Inc., sep 2016.

[12] E. Ruijters and M. Stoelinga, "Fault tree analysis: A survey of the state-of-the-art in modeling, analysis and tools," *Computer Science Review*, vol. 15-16, pp. 29–62, feb 2015. doi: 10.1016/j.cosrev.2015.03.001

[13] A. Myers, *Complex System Reliability*. Springer London, 2010.

[14] D. J. Reifer, "Software Failure Modes and Effects Analysis," *IEEE Transactions on reliability*, vol. R-28, no. 3, pp. 247–249, aug 1979. doi: 10.1109/tr.1979.5220578

[15] M. Grottke and K. S. Trivedi, "Fighting bugs: remove, retry, replicate, and rejuvenate," *Computer*, vol. 40, no. 2, pp. 107–109, 2007. doi: 10.1109/MC.2007.55

[16] M. Grottke, H. Sun, R. M. Fricks, and K. S. Trivedi, "Ten fallacies of availability and reliability analysis," in *Service Availability*. Springer Berlin Heidelberg, 2008, vol. 5017, pp. 187–206.

[17] S. Yamada and S. Osaki, "Software Reliability Growth Modeling: Models and Applications," *IEEE Transactions on Software Engineering*, vol. SE-11, no. 12, pp. 1431–1437, dec 1985. doi: 10.1109/tse.1985.232179

[18] D. D. Hanagal and N. N. Bhalerao, *Literature Survey in Software Reliability Growth Models*. Singapore: Springer Singapore, 2021, pp. 13–26. ISBN 978-981-16-0025-8. [Online]. Available: https://doi.org/10.1007/978-981-16-0025-8_2

[19] K. S. Trivedi, A. Bobbio, and J. Muppala, *Reliability and Availability Engineering*. Cambridge University Press, 2017.

[20] M. R. Lyu, "Software Reliability Engineering: A Roadmap," in *Future of Software Engineering (FOSE '07)*, IEEE. IEEE, may 2007. doi: 10.1109/fose.2007.24 pp. 153–170.

[21] M. Jammal, A. Kanso, P. Heidari, and A. Shami, "A Formal Model for the Availability Analysis of Cloud Deployed Multi-tiered Applications," in *2016*

IEEE International Conference on Cloud Engineering Workshop (IC2EW), IEEE. IEEE, apr 2016. doi: 10.1109/ic2ew.2016.21 pp. 82–87.

[22] R. Ghosh, F. Longo, F. Frattini, S. Russo, and K. S. Trivedi, "Scalable analytics for IaaS cloud availability," *IEEE Transactions on Cloud Computing*, vol. 2, no. 1, pp. 57–70, jan 2014. doi: 10.1109/tcc.2014.2310737

[23] T. Pitakrat, D. Okanović, A. van Hoorn, and L. Grunske, "Hora: Architecture-aware online failure prediction," *Journal of Systems and Software*, vol. 137, pp. 669–685, mar 2018. doi: 10.1016/j.jss.2017.02.041

[24] F. Brosch, H. Koziolek, B. Buhnova, and R. Reussner, "Architecture-based reliability prediction with the palladio component model," *IEEE Transactions on Software Engineering*, vol. 38, no. 6, pp. 1319–1339, nov 2012. doi: 10.1109/tse.2011.94

[25] S. Becker, H. Koziolek, and R. Reussner, "The palladio component model for model-driven performance prediction," *Journal of Systems and Software*, vol. 82, no. 1, pp. 3–22, jan 2009. doi: 10.1016/j.jss.2008.03.066

[26] J. F. Pérez and G. Casale, "Assessing sla compliance from palladio component models," in *2013 15th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*. IEEE, 2013, pp. 409–416.

[27] J. G. Torres-Toledano and L. E. Sucar, "Bayesian Networks for Reliability Analysis of Complex Systems," in *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 1998, pp. 195–206.

[28] T. Ye, Y. Zhou, A. Chen, L. Liu, and S. Liu, "Extend GO methodology to support common-cause failures modeling explicitly by means of bayesian networks," *IEEE Transactions on Reliability*, vol. 69, no. 2, pp. 471–483, jun 2020. doi: 10.1109/tr.2019.2917752

[29] B. Cai, X. Kong, Y. Liu, J. Lin, X. Yuan, H. Xu, and R. Ji, "Application of bayesian networks in reliability evaluation," *IEEE Transactions on Industrial Informatics*, vol. 15, no. 4, pp. 2146–2157, apr 2019. doi: 10.1109/tii.2018.2858281

[30] O. Kammouh, P. Gardoni, and G. P. Cimellaro, "Probabilistic framework to evaluate the resilience of engineering systems using bayesian and dynamic bayesian networks," *Reliability Engineering & System Safety*, vol. 198, p. 106813, jun 2020. doi: 10.1016/j.ress.2020.106813

[31] H. Langseth and L. Portinale, "Bayesian networks in reliability," *Reliability Engineering & System Safety*, vol. 92, no. 1, pp. 92–108, jan 2007. doi: 10.1016/j.ress.2005.11.037

[32] R. xing Duan and H. lin Zhou, "A New Fault Diagnosis Method Based on Fault Tree and Bayesian Networks," *Energy Procedia*, vol. 17, pp. 1376–1382, 2012. doi: 10.1016/j.egypro.2012.02.255

[33] R. Pan and P. Yontay, "Reliability assessment of hierarchical systems with incomplete mixed data," *IEEE Transactions on Reliability*, vol. 66, no. 4, pp. 1036–1047, dec 2017. doi: 10.1109/tr.2017.2760802

[34] D. Koller and N. Friedman, *Probabilistic graphical models: principles and techniques*. MIT press, 2009.

[35] A. L. Lemos, F. Daniel, and B. Benatallah, "Web Service Composition: A Survey of Techniques and Tools," *ACM Comput. Surv.*, vol. 48, no. 3, pp. 1–41, feb 2016. doi: 10.1145/2831270. [Online]. Available: https://doi.org/10.1145/2831270

[36] H. Liu, F. Zhong, B. Ouyang, and J. Wu, "An approach for QoS-aware web service composition based on improved genetic algorithm," in *2010 International Conference on Web Information Systems and Mining*, vol. 1. IEEE, oct 2010. doi: 10.1109/wism.2010.128 pp. 123–128.

[37] M. Mohammed, M. A. Chikh, and H. Fethallah, "QoS-aware web service selection based on harmony search," in *2014 4th International Symposium ISKO-Maghreb: Concepts and Tools for knowledge Management (ISKO-Maghreb)*. IEEE, nov 2014. doi: 10.1109/isko-maghreb.2014.7033465 pp. 1–6.

[38] A. E. Yilmaz and P. Karagoz, "Improved genetic algorithm based approach for QoS aware web service composition," in *2014 IEEE International Conference on Web Services*. IEEE, jun 2014. doi: 10.1109/icws.2014.72 pp. 463–470.

[39] W. Zhang, C. K. Chang, T. Feng, and H. yi Jiang, "QoS-based dynamic web service composition with ant colony optimization," in *2010 IEEE 34th Annual Computer Software and Applications Conference*. IEEE, jul 2010. doi: 10.1109/compsac.2010.76 pp. 493–502.

[40] P. M. Esfahani, J. Habibi, and T. Varaee, "Application of social harmony search algorithm on composite web service selection based on quality attributes," in *2012 Sixth International Conference on Genetic and Evolutionary Computing*. IEEE, aug 2012. doi: 10.1109/icgec.2012.65 pp. 526–529.

[41] A. Bekkouche, S. M. Benslimane, M. Huchard, C. Tibermacine, F. Hadjila, and M. Merzoug, "QoS-aware optimal and automated semantic web service composition with user's constraints," *Service Oriented Computing and Applications*, vol. 11, no. 2, pp. 183–201, mar 2017. doi: 10.1007/s11761-017-0205-1

[42] B. D. Martino, G. Cretella, and A. Esposito, "Cloud Services Composition Through Cloud Patterns," in *Adaptive Resource Management and Scheduling for Cloud Computing*. Springer International Publishing, 2015, pp. 128–140.

[43] F. Dahan, "An effective multi-agent ant colony optimization algorithm for QoS-aware cloud service composition," *IEEE Access*, vol. 9, pp. 17 196–17 207, 2021. doi: 10.1109/access.2021.3052907

[44] C. B. Pop, M. Vlad, V. R. Chifu, I. Salomie, and M. Dinsoreanu, "A Tabu Search Optimization Approach for Semantic Web Service Composition," in *2011 10th International Symposium on Parallel and Distributed Computing*. IEEE, jul 2011. doi: 10.1109/ispdc.2011.49 pp. 274–277.

[45] N. Jafarpour and M.-R. Khayyambashi, "Qos-aware selection of web service compositions using harmony search algorithm." *J. Digit. Inf. Manag.*, vol. 8, no. 3, pp. 160–166, 2010.

[46] J. A. Parejo, P. Fernandez, and A. R. Cortés, "Qos-aware services composition using tabu search and hybrid genetic algorithms," *Actas de los Talleres de las Jornadas de Ingeniería del Software y Bases de Datos*, vol. 2, no. 1, pp. 55–66, 2008.

[47] A. Bergmayr, U. Breitenbücher, N. Ferry, A. Rossini, A. Solberg, M. Wimmer, G. Kappel, and F. Leymann, "A Systematic Review of Cloud Modeling Languages," *ACM Computing Surveys*, vol. 51, no. 1, pp. 1–38, apr 2018. doi: 10.1145/3150227

[48] A. Bergmayr, J. Troya Castilla, P. Neubauer, M. Wimmer, and G. Kappel, "UML-based cloud application modeling with libraries, profiles, and tem-

plates," in *CloudMDE 2014: 2nd International Workshop on Model-Driven Engineering on and for the Cloud co-located with the 17th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2014)(2014), p 56-65.* CEUR-WS, 2014.

[49] A. Bergmayr, U. Breitenbücher, O. Kopp, M. Wimmer, G. Kappel, and F. Leymann, "From architecture modeling to application provisioning for the cloud by combining UML and TOSCA," in *Proceedings of the 6th International Conference on Cloud Computing and Services Science.* SCITEPRESS - Science and Technology Publications, 2016. doi: 10.5220/0005806900970108 pp. 97–108.

[50] D. K. Nguyen, F. Lelli, Y. Taher, M. Parkin, M. P. Papazoglou, and W.-J. van den Heuvel, "Blueprint template support for engineering cloud-based services," in *Towards a Service-Based Internet.* Springer Berlin Heidelberg, 2011, pp. 26–37.

[51] D. K. Nguyen, F. Lelli, M. P. Papazoglou, and W.-J. van den Heuvel, "Blueprinting approach in support of cloud computing," *Future Internet*, vol. 4, no. 1, pp. 322–346, mar 2012. doi: 10.3390/fi4010322

[52] V. Andrikopoulos, A. Reuter, S. G. Sáez, and F. Leymann, "A GENTL approach for cloud application topologies," in *Advanced Information Systems Engineering.* Springer Berlin Heidelberg, 2014, pp. 148–159.

[53] OASIS (2013), "Topology and orchestration specification for cloud applications (TOSCA) Version 1.0."

[54] J. Bellendorf and Z. Á. Mann, "Cloud topology and orchestration using TOSCA: A systematic literature review," in *Service-Oriented and Cloud Computing*, K. Kritikos, P. Plebani, and F. de Paoli, Eds. Cham: Springer International Publishing, 2018, pp. 207–215. ISBN 978-3-319-99819-0

[55] A. Cockcroft and D. Sheahan, "Benchmarking cassandra scalability on aws - over a million writes per second," https://netflixtechblog.com/benchmarking-cassandra-scalability-on-aws-over-a-million-writes-per-second-39f45f066c9e, [Online; accessed 22-Feb-2022].

[56] Amazon Web Services, Inc., "Overview of Amazon Web Services," https://docs.aws.amazon.com/whitepapers/latest/aws-overview/introduction.html, [Online; accessed 12-Okt-2021].

[57] Microsoft Corporation, "Azure products," https://azure.microsoft.com/de-de/services/, [Online; accessed 12-Okt-2021].

[58] Google, Inc., "Google Cloud products," https://cloud.google.com/products, [Online; accessed 12-Okt-2021].

[59] O. Bibartiu, F. Dürr, and K. Rothermel, "Clams: A cloud application modeling solution," in *2021 IEEE International Conference on Services Computing (SCC)*, 2021.

[60] O. Bibartiu, F. Dürr, K. Rothermel, B. Ottenwälder, and A. Grau, "Towards Scalable k-out-of-n Models for Assessing the Reliability of Large-Scale Function-as-a-Service Systems with Bayesian Networks," in *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. IEEE, jul 2019. doi: 10.1109/cloud.2019.00095. ISSN 2159-6182 pp. 514–516.

[61] ——, "Scalable k-out-of-n models for dependability analysis with bayesian networks," *Reliability Engineering & System Safety*, vol. 210, p. 107533, jun 2021. doi: 10.1016/j.ress.2021.107533

[62] ——, "Availability analysis of redundant and replicated cloud services with bayesian networks," arXiv:2306.13334v1 [cs.DC], 2023.

[63] O. Bibartiu, F. Dürr, and K. Rothermel, "Optimal refinement for component-based architectures," in *2021 IEEE 25th International Enterprise Distributed Object Computing Conference (EDOC)*, 2021.

[64] C. Fehling, F. Leymann, R. Retter, W. Schupeck, and P. Arbitter, *Cloud computing patterns: fundamentals to design, build, and manage cloud applications*. Springer, 2014.

[65] R. Cheung, "A user-oriented software reliability model," *IEEE Transactions on Software Engineering*, vol. SE-6, no. 2, pp. 118–125, mar 1980. doi: 10.1109/tse.1980.234477

[66] Z. W. Geem, J. H. Kim, and G. Loganathan, "A new heuristic optimization algorithm: Harmony search," *SIMULATION*, vol. 76, no. 2, pp. 60–68, feb 2001. doi: 10.1177/003754970107600201

[67] M. Müller, "Design and evaluation of availability models for the rabbitmq messaging middleware," Master's thesis, University of Stuttgart, 2017.

[68] F. Pfeffer, "Verfügbarketisvorhersagen für paas komponente mittels maschinellem lernens," Bachelor's thesis, University of Stuttgart, 2018.

[69] F. Vogt, "Bayessche zuverlässigkeitsanalyse von mongodb," Bachelor's thesis, University of Stuttgart, 2018.

[70] M. Coskuner, "Designing and analyzing of hybrid cloud concepts for openclams," Master's thesis, University of Stuttgart, 2020.

[71] T. Linn, "Design of a rule-based formalism to predict vm placements in cloud infrastructures," Master's thesis, University of Stuttgart, 2020.

[72] M. Weilinger, "Design and implementation of a service recommendation system for clams," Bachelor's thesis, University of Stuttgart, 2021.

[73] S. Matejetz, "Extending modeling concepts of openclams to support performance analysis with layered queuing," Master's thesis, University of Stuttgart, 2021.

[74] A. G. Kleppe, J. Warmer, J. B. Warmer, and W. Bast, *MDA explained: the model driven architecture: practice and promise*. Addison-Wesley Professional, 2003.

[75] E. Ábrahám, F. Corzilius, E. B. Johnsen, G. Kremer, and J. Mauro, "Zephyrus2: On the fly deployment optimization using SMT and CP technologies," in *Dependable Software Engineering: Theories, Tools, and Applications*, M. Fränzle, D. Kapur, and N. Zhan, Eds. Cham: Springer International Publishing, 2016, pp. 229–245. ISBN 978-3-319-47677-3

[76] K. Kepes, U. Breitenbucher, F. Leymann, K. Saatkamp, and B. Weder, "Deployment of distributed applications across public and private networks," in *2019 IEEE 23rd International Enterprise Distributed Object Computing Conference (EDOC)*. IEEE, oct 2019. doi: 10.1109/edoc.2019.00036 pp. 236–242.

[77] V. Yussupov, U. Breitenbucher, C. Krieger, F. Leymann, J. Soldani, and M. Wurster, "Pattern-based modelling, integration, and deployment of microservice architectures," in *2020 IEEE 24th International Enterprise Distributed Object Computing Conference (EDOC)*. IEEE, oct 2020. doi: 10.1109/edoc49727.2020.00015 pp. 40–50.

[78] M. Falkenthal, J. Barzen, U. Breitenbücher, C. Fehling, and F. Leymann, "From pattern languages to solution implementations," in *Proceedings of the Sixth International Conferences on Pervasive Patterns and Applications (PATTERNS 2014)*, 2014, pp. 12–21.

[79] G. Rodrigues, D. Rosenblum, and S. Uchitel, "Using scenarios to predict the reliability of concurrent component-based software systems," in *Fundamental Approaches to Software Engineering*, M. Cerioli, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 111–126. ISBN 978-3-540-31984-9

[80] S. Uchitel, J. Kramer, and J. Magee, "Synthesis of behavioral models from scenarios," *IEEE Transactions on Software Engineering*, vol. 29, no. 2, pp. 99–115, feb 2003. doi: 10.1109/tse.2003.1178048

[81] G. Hohpe and B. Woolf, *Enterprise integration patterns: Designing, building, and deploying messaging solutions*. Addison-Wesley Professional, 2004.

[82] M. Falkenthal, J. Barzen, U. Breitenbücher, C. Fehling, F. Leymann, H. Schulze, and L. Ostwestfalen-Lippe, "Leveraging pattern application via pattern refinement," in *Proceedings of the International Conference on Pursuit of Pattern Languages for Societal Change (PURPLSOC)*, 2016, pp. 38–61.

[83] S. Uchitel, J. Kramer, and J. Magee, "Incremental elaboration of scenario-based specifications and behavior models using implied scenarios," *ACM Transactions on Software Engineering and Methodology*, vol. 13, no. 1, pp. 37–85, jan 2004. doi: 10.1145/1005561.1005563

[84] C. Alexander, *A pattern language: towns, buildings, construction*. Oxford university press, 1977.

[85] J. Davies, J. Gibbons, D. Milward, and J. Welch, "Compositionality and refinement in model-driven engineering," in *Lecture Notes in Computer Science*, R. Gheyi and D. Naumann, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 99–114. ISBN 978-3-642-33296-8

[86] E. Gamma, R. Helm, R. Johnson, J. Vlissides, and D. Patterns, "Elements of reusable object-oriented software," *Design Patterns. massachusetts: Addison-Wesley Publishing Company*, 1995.

[87] C.-J. Hsu, V. Nair, T. Menzies, and V. Freeh, "Micky: A cheaper alternative for selecting cloud instances," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. IEEE, jul 2018. doi: 10.1109/cloud.2018.00058 pp. 409–416.

[88] Microsoft Corporation, "Azure Architecture Documentation," https://docs.microsoft.com/de-de/azure/architecture/browse/, [Online; accessed 12-Dec-2021].

[89] ——, "Azure Calculator," https://azure.microsoft.com/de-de/pricing/calculator/, [Online; accessed 12-Dec-2021].

[90] L. Harzenetter, U. Breitenbucher, M. Falkenthal, J. Guth, C. Krieger, and F. Leymann, "Pattern-based deployment models and their automatic execution," in *2018 IEEE/ACM 11th International Conference on Utility and Cloud Computing (UCC)*. IEEE, dec 2018. doi: 10.1109/ucc.2018.00013 pp. 41–52.

[91] J. Guth and F. Leymann, "Pattern-based rewrite and refinement of architectures using graph theory," *SICS Software-Intensive Cyber-Physical Systems*, vol. 35, no. 1-2, pp. 115–126, aug 2019. doi: 10.1007/s00450-019-00416-7

[92] A. Brogi and J. Soldani, "Finding available services in TOSCA-compliant clouds," *Science of Computer Programming*, vol. 115-116, pp. 177–198, jan 2016. doi: 10.1016/j.scico.2015.09.004 Special Section on Foundations of Coordination Languages and Software (FOCLASA 2012) Special Section on Foundations of Coordination Languages and Software (FOCLASA 2013).

[93] I. B. Fraj, Y. B. Hlaoui, and L. BenAyed, "A control system for managing the flexibility in bpmn models of cloud service workflows," in *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*, 2020. doi: 10.1109/CLOUD49709.2020.00081 pp. 537–543.

[94] M. Falkenthal, U. Breitenbücher, and F. Leymann, "The nature of pattern languages," *cit. on*, p. 14, 2018.

[95] M. Falkenthal and F. Leymann, "Easing pattern application by means of solution languages," in *Proceedings of the 9th International Conference on Pervasive Patterns and Applications*, 2017, pp. 58–64.

[96] L. Harzenetter, U. Breitenbücher, M. Falkenthal, J. Guth, and F. Leymann, "Pattern-based Deployment Models Revisited: Automated Pattern-driven Deployment Configuration," in *Proceedings of the Twelfth International Conference on Pervasive Patterns and Applications (PATTERNS 2020)*. Xpert Publishing Services, Oct. 2020. ISBN 978-1-61208-783-2 pp. 40–49.

[97] D. L. Metayer, "Describing software architecture styles using graph grammars," *IEEE Transactions on Software Engineering*, vol. 24, no. 7, pp. 521–533, jul 1998. doi: 10.1109/32.708567

[98] D. WANG and K. S. TRIVEDI, "MODELING USER-PERCEIVED RELIABILITY BASED ON USER BEHAVIOR GRAPHS," *International Journal of Reliability, Quality and Safety Engineering*, vol. 16, no. 04, pp. 303–329, aug 2009. doi: 10.1142/s0218539309003411. [Online]. Available: https://doi.org/10.1142/S0218539309003411

[99] W. E. Smith, K. S. Trivedi, L. A. Tomek, and J. Ackaret, "Availability analysis of blade server systems," *IBM Systems Journal*, vol. 47, no. 4, pp. 621–640, 2008. doi: 10.1147/sj.2008.5386524

[100] K. Trivedi, D. Wang, D. J. Hunt, A. Rindos, W. E. Smith, and B. Vashaw, "Availability modeling of SIP protocol on IBM© WebSphere©," in *2008 14th IEEE Pacific Rim International Symposium on Dependable Computing*, IEEE. IEEE, dec 2008. doi: 10.1109/prdc.2008.50 pp. 323–330.

[101] M. C. Kim, "Reliability block diagram with general gates and its application to system reliability analysis," *Annals of Nuclear Energy*, vol. 38, no. 11, pp. 2456–2461, nov 2011. doi: 10.1016/j.anucene.2011.07.013

[102] M. Malhotra and K. Trivedi, "Power-hierarchy of dependability-model types," *IEEE Transactions on Reliability*, vol. 43, no. 3, pp. 493–502, 1994. doi: 10.1109/24.326452

[103] G. Bolch, S. Greiner, H. de Meer, and K. S. Trivedi, *Queueing Networks*. Wiley Online Library, 1998.

[104] Network Working Group, "SIP: Session Initiation Protocol," https://tools.ietf.org/html/rfc3261, [Online; accessed 12-Okt-2021].

[105] L. Bennacer, Y. Amirat, A. Chibani, A. Mellouk, and L. Ciavaglia, "Self-diagnosis technique for virtual private networks combining bayesian

networks and case-based reasoning," *IEEE Transactions on Automation Science and Engineering*, vol. 12, no. 1, pp. 354–366, jan 2015. doi: 10.1109/tase.2014.2321011

[106] D. Cotroneo, L. D. Simone, P. Liguori, R. Natella, and N. Bidokhti, "Enhancing Failure Propagation Analysis in Cloud Computing Systems," in *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, oct 2019. doi: 10.1109/issre.2019.00023. ISSN 1071-9458 pp. 139–150.

[107] P. H. Hochschild, P. Turner, J. C. Mogul, R. Govindaraju, P. Ranganathan, D. E. Culler, and A. Vahdat, "Cores that don't count," in *Proceedings of the Workshop on Hot Topics in Operating Systems*, ser. HotOS '21. New York, NY, USA: ACM, jun 2021. doi: 10.1145/3458336.3465297. ISBN 9781450384384 p. 9–16.

[108] E. Brewer, "Spanner, truetime and the cap theorem," Google Inc., Tech. Rep., 2017.

[109] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "Highly available transactions: Virtues and limitations," *Proceedings of the VLDB Endowment*, vol. 7, no. 3, pp. 181–192, nov 2013. doi: 10.14778/2732232.2732237

[110] P. Garraghan, R. Yang, Z. Wen, A. Romanovsky, J. Xu, R. Buyya, and R. Ranjan, "Emergent Failures: Rethinking Cloud Reliability at Scale," *IEEE Cloud Computing*, vol. 5, no. 5, pp. 12–21, sep 2018. doi: 10.1109/mcc.2018.053711662

[111] H. S. Gunawi, M. Hao, T. Leesatapornwongsa, T. Patana-anake, T. Do, J. Adityatama, K. J. Eliazar, A. Laksono, J. F. Lukman, V. Martin, and A. D. Satria, "What bugs live in the cloud? a study of 3000+ issues in cloud systems," in *Proceedings of the ACM Symposium on Cloud Computing*, ACM. ACM, nov 2014. doi: 10.1145/2670979.2670986 pp. 1–14.

[112] A. Epstein, E. K. Kolodner, and D. Sotnikov, "Network aware reliability analysis for distributed storage systems," in *2016 IEEE 35th Symposium on Reliable Distributed Systems (SRDS)*. IEEE, sep 2016. doi: 10.1109/srds.2016.042 pp. 249–258.

[113] D. Ford, F. Labelle, F. I. Popovici, M. Stokely, V.-A. Truong, L. Barroso, C. Grimes, and S. Quinlan, "Availability in Globally Distributed Storage Systems." in *Osdi*, vol. 10, 2010, pp. 1–7.

[114] M.-C. Chiang, C.-Y. Huang, C.-Y. Wu, and C.-Y. Tsai, "Analysis of a fault-tolerant framework for reliability prediction of service-oriented architecture systems," *IEEE Transactions on Reliability*, vol. 70, no. 1, pp. 13–48, mar 2021. doi: 10.1109/tr.2020.2968884

[115] M. Stamatelatos, W. Vesely, J. Dugan, J. Fragola, J. Minarick, and J. Railsback, "Fault tree handbook with aerospace applications," *Office of safety and mission assurance NASA headquarters*, 2002.

[116] J. Pearl, "Probabilistic reasoning in intelligent systems: Networks of plausible reasoning," *Morgan Kaufmann Publishers, Los Altos*, 1988.

[117] A. Bobbio, L. Portinale, M. Minichino, and E. Ciancamerla, "Improving the analysis of dependable systems by mapping fault trees into Bayesian networks," *Reliability Engineering & System Safety*, vol. 71, no. 3, pp. 249–260, mar 2001. doi: 10.1016/s0951-8320(00)00077-6

[118] H. Boudali and J. Dugan, "A discrete-time bayesian network reliability modeling and analysis framework," *Reliability Engineering & System Safety*, vol. 87, no. 3, pp. 337–349, mar 2005. doi: 10.1016/j.ress.2004.06.004

[119] J. D. Esary and F. Proschan, "Coherent structures of non-identical components," *Technometrics*, vol. 5, no. 2, pp. 191–209, 1963.

[120] Google, Inc., "Google App Engine SLA," https://cloud.google.com/appengine/sla, [Online; accessed 12-Okt-2021].

[121] ——, "Google Storage SLA," https://cloud.google.com/storage/sla, [Online; accessed 12-Okt-2021].

[122] Amazon Web Services, Inc., "Amazon Web Services: EC2 SLA ," https://d1.awsstatic.com/legal/AmazonComputeServiceLevelAgreement/Amazon%20Compute%20Service%20Level%20Agreement_German_2020-07-22.pdf, [Online; accessed 12-Okt-2021].

[123] ——, "Amazon Web Services: S3 SLA," https://docs.aws.amazon.com/whitepapers/latest/aws-overview/introduction.html, [Online; accessed 12-Okt-2021].

[124] Microsoft Corporation, "Azure StorSimple SLA," https://azure.microsoft.com/de-de/support/legal/sla/summary/, [Online; accessed 12-Okt-2021].

[125] ——, "Azure Virtual Machine SLA," https://azure.microsoft.com/de-de/support/legal/sla/virtual-machines/v1_9/, [Online; accessed 12-Okt-2021].

[126] S. L. Lauritzen and D. J. Spiegelhalter, "Local Computations with Probabilities on Graphical Structures and Their Application to Expert Systems," *Journal of the Royal Statistical Society: Series B (Methodological)*, vol. 50, no. 2, pp. 157–194, jan 1988. doi: 10.1111/j.2517-6161.1988.tb01721.x

[127] S. Højsgaard, "Bayesian networks in R with the gRain package," *Rel téc Aalborg University*, pp. 1–15, 2015.

[128] S. H. jsgaard, "Graphical independence networks with thegRainPackage forR," *Journal of Statistical Software*, vol. 46, no. 10, pp. 1–26, 2012. doi: 10.18637/jss.v046.i10

[129] S. Tani, K. Hamaguchi, and S. Yajima, "The complexity of the optimal variable ordering problems of shared binary decision diagrams," in *Algorithms and Computation*, K. W. Ng, P. Raghavan, N. V. Balasubramanian, and F. Y. L. Chin, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993. ISBN 978-3-540-48233-8 pp. 389–398.

[130] A. Bobbio, L. Portinale, M. Minichino, and E. Ciancamerla, "Comparing fault trees and bayesian networks for dependability analysis," in *Computer Safety, Reliability and Security.* Springer Berlin Heidelberg, 1999, pp. 310–322.

[131] H. Boudali and J. Dugan, "A new bayesian network approach to solve dynamic fault trees," in *Annual Reliability and Maintainability Symposium, 2005. Proceedings.* IEEE, Jan 2005. doi: 10.1109/rams.2005.1408404. ISSN 0149-144X pp. 451–456.

[132] A. D. Giorgio and F. Liberati, "A Bayesian Network-Based Approach to the Critical Infrastructure Interdependencies Analysis," *IEEE Systems Journal*, vol. 6, no. 3, pp. 510–519, sep 2012. doi: 10.1109/jsyst.2012.2190695

[133] D. S. Kim, F. Machida, and K. S. Trivedi, "Availability modeling and analysis of a virtualized system," in *2009 15th IEEE Pacific Rim Inter-*

national Symposium on Dependable Computing, IEEE. IEEE, nov 2009. doi: 10.1109/prdc.2009.64 pp. 365–371.

[134] I. Narayanan, A. Kansal, and A. Sivasubramaniam, "Right-Sizing Geo-distributed Data Centers for Availability and Latency," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, IEEE. IEEE, jun 2017. doi: 10.1109/icdcs.2017.118 pp. 230–240.

[135] D. Heckerman, "Causal independence for knowledge acquisition and inference," in *Uncertainty in Artificial Intelligence*. Elsevier, 1993, pp. 122–127.

[136] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, apr 2010. doi: 10.1145/1773912.1773922

[137] D. Heckerman and J. S. Breese, "A New Look at Causal Independence," in *Uncertainty Proceedings 1994*. Elsevier, 1994, pp. 286–292.

[138] I. Tien and A. D. Kiureghian, "Algorithms for Bayesian network modeling and reliability assessment of infrastructure systems," *Reliability Engineering & System Safety*, vol. 156, pp. 134–147, dec 2016. doi: 10.1016/j.ress.2016.07.022

[139] K. G. Olesen and S. Andreassen, "Specification of models in large expert systems based on causal probabilistic networks," *Artificial Intelligence in Medicine*, vol. 5, no. 3, pp. 269–281, jun 1993. doi: 10.1016/0933-3657(93)90029-3

[140] J. Kim and J. Pearl, "A computational model for causal and diagnostic reasoning in inference systems," in *Proceedings of the Eighth International Joint Conference on Artificial Intelligence (I)*, vol. 1, 1983, pp. 190–193.

[141] K. Laskey, "Sensitivity analysis for probability assessments in Bayesian networks," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 25, no. 6, pp. 901–909, jun 1995. doi: 10.1109/21.384252

[142] S. Srinivas, "A generalization of the noisy-or model," in *Uncertainty in Artificial Intelligence*. Elsevier, 1993, pp. 208–215.

[143] F. J. Diez, "Parameter adjustment in Bayes networks. The generalized noisy OR–gate," in *Uncertainty in Artificial Intelligence, 1993*. Elsevier, 1993, pp. 99–105.

[144] P. Dagum and M. Luby, "Approximating probabilistic inference in Bayesian belief networks is NP-hard," *Artificial Intelligence*, vol. 60, no. 1, pp. 141–153, mar 1993. doi: 10.1016/0004-3702(93)90036-b

[145] P. Dagum and A. Galper, "Additive belief-network models," in *Uncertainty in Artificial Intelligence*. Elsevier, 1993, pp. 91–98.

[146] K. G. OLESEN, U. KJAERULFF, F. JENSEN, F. V. JENSEN, B. FALCK, S. ANDREASSEN, and S. K. ANDERSEN, "A MUNIN NETWORK FOR THE MEDIAN NERVE-a CASE STUDY ON LOOPS," *Applied Artificial Intelligence*, vol. 3, no. 2-3, pp. 385–403, jan 1989. doi: 10.1080/08839518908949933

[147] M. Tirmazi, A. Barker, N. Deng, M. E. Haque, Z. G. Qin, S. Hand, M. Harchol-Balter, and J. Wilkes, "Borg: The next generation," in *Proceedings of the Fifteenth European Conference on Computer Systems*, ser. EuroSys '20. New York, NY, USA: ACM, apr 2020. doi: 10.1145/3342195.3387517. ISBN 9781450368827. [Online]. Available: https://doi.org/10.1145/3342195.3387517

[148] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at google with borg," in *Proceedings of the Tenth European Conference on Computer Systems*, ser. EuroSys '15. New York, NY, USA: ACM, apr 2015. doi: 10.1145/2741948.2741964. ISBN 9781450332385. [Online]. Available: https://doi.org/10.1145/2741948.2741964

[149] M. Cremene, M. Suciu, D. Pallez, and D. Dumitrescu, "Comparative analysis of multi-objective evolutionary algorithms for QoS-aware web service composition," *Applied Soft Computing*, vol. 39, pp. 124–139, feb 2016. doi: 10.1016/j.asoc.2015.11.012

[150] S. Mistry, A. Bouguettaya, H. Dong, and A. K. Qin, "Metaheuristic optimization for long-term IaaS service composition," *IEEE Transactions on Services Computing*, vol. 11, no. 1, pp. 131–143, jan 2018. doi: 10.1109/tsc.2016.2542068

[151] M. Dorigo, M. Birattari, and T. Stutzle, "Ant colony optimization," *IEEE Computational Intelligence Magazine*, vol. 1, no. 4, pp. 28–39, nov 2006. doi: 10.1109/ci-m.2006.248054

[152] L. Kapova and B. Buhnova, "Performance-driven stepwise refinement of component-based architectures," in *Proceedings of the 2nd International Workshop on the Quality of Service-Oriented Software Systems - QUASOSS '10*, ser. QUASOSS '10. New York, NY, USA: ACM Press, 2010. doi: 10.1145/1858263.1858269. ISBN 9781450302395

[153] H. Brabra, A. Mtibaa, W. Gaaloul, B. Benatallah, and F. Gargouri, "Model-driven orchestration for cloud resources," in *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. IEEE, jul 2019. doi: 10.1109/cloud.2019.00074 pp. 422–429.