TU WIEN Informatics

Reliability of Edge Offloading

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doktor der Technischen Wissenschaften

by

Mag. Josip Zilic
Registration Number 0123456

to the Faculty of Informatics

at the TU Wien

Advisor: Univ. Prof. Dr. Ivona Brandic

The dissertation has been reviewed by:

Forename Surname	Forename Surname

Vienna, January 1, 2001

Josip Zilic

Erklärung zur Verfassung der Arbeit

Mag. Josip Zilic

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.
Ich erkläre weiters, dass ich mich generativer KI-Tools lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss überwiegt. Im Anhang „Übersicht verwendeter Hilfsmittel“ habe ich alle generativen KI-Tools gelistet, die verwendet wurden, und angegeben, wo und wie sie verwendet wurden. Für Textpassagen, die ohne substantielle Änderungen übernommen wurden, haben ich jeweils die von mir formulierten Eingaben (Prompts) und die verwendete IT- Anwendung mit ihrem Produktnamen und Versionsnummer/Datum angegeben.

Wien, 1. Jänner 2001

Josip Zilic

Previous Publications

This thesis is based on work published in peer-reviewed scientific journals, conferences, and workshops. The following papers build the foundation of this thesis. They are listed here once and will generally not be explicitly referenced again. Parts of these papers are contained in verbatim.

Refereed Publications in Journals (Under Submission)

1. Josip Zilic, Vincenzo de Maio, Shashikant Ilager, Ivona Brandic, *FRESCO: Fast and Reliable Edge Offloading using Hybrid Smart Contracts*. IEEE Transactions on Services Computing (Under Submission), 2024., DOI: https://doi.org/10.48550/arXiv.2410.06715

Refereed Publications in Conference Proceedings

1. Josip Zilic, Vincenzo De Maio, Atakan Aral, Ivona Brandic, *Edge Offloading of Microservice Architecture: A Failure-Aware Approach*. Proceedings of the 5th International Workshop on Edge Systems, Analytics and Networking (EdgeSys), 2022. DOI: https://doi.org/10.1145/3517206.3526266.

2. Josip Zilic, Atakan Aral, Ivona Brandic. *EFPO: Energy-Efficient and Failure Predictive Offloading for Edge Computing*. Proceedings of the IEEE/ACM International Conference on Utility and Cloud Computing (UCC), 2019. DOI: https://doi.org/10.1145/3344341.3368818.

Abstract

Cloud computing enabled scalable computational and data storage services to become ubiquitous and available on-demand globally around the world for all internet-capable devices including those with mobile and wireless connections. The resource-abundant cloud environment equipped with hundreds and thousands of commodity off-the-shelf servers, located in so-called data centres, enabled developers and companies to develop much more complex enterprise applications and systems tailored to satisfy customer needs and requirements. Cloud allowed businesses to deliver and scale business operations globally and reach out to all corners of global markets, fuelling economic growth and development. Newly developed cloud applications and services contained a richer set of features and were accessible by all kinds of mobile devices that did not have sufficient resources to execute them. Furthermore, with the hardware development breakthrough of system-on-chip technology, the first smartphones emerged that supported local software functions where some application features could be executed locally to reduce response time for a better user experience, especially where user-facing interactions and input are required. However, newly introduced classes of applications have emerged such as autonomous vehicles and GPS navigation which require low latency response up to a few milliseconds to perform real-time decisions but simultaneously require huge amounts of resources to achieve such fast performance that mobile devices cannot provide. Violating strict time requirements can lead to hazardous and even life-threatening situations. These strict requirements cannot be satisfied by simply moving complex applications and their data from mobile devices to computationally scalable cloud data centres which exhibit long latencies due to geographical distance and high-volume internet traffic load. Fortunately, edge computing was introduced as a viable and promising middle-ground solution between cloud and mobile to provide sufficient resources for achieving high-performance. Cloud data processing and storage capabilities were partly moved from the centralized cloud data centres to the servers at the edge of the network in the proximity of mobile devices, thus reducing geographical distance and bypassing high-volume internet traffic.

To exploit the full benefits of edge computing for enabling real-time execution of resource-intensive applications in resource-constrained environments, the offloading concept was introduced. Mobile applications are partitioned into smaller units called tasks and offloaded from resource-constrained mobile devices to resourceful nearby edge servers for reducing application response time and saving energy supplies on battery-powered mobile devices. Every application task has its own time and resource requirements. Low-latency

application tasks are offloaded to the nearby edge servers while computational-intensive tasks are offloaded to computationally scalable cloud data centres. However, the edge servers are resource-limited compared to their cloud counterparts having less hardware redundancy, a lack of supporting systems (e.g. backup power generators), and are deployed across many geographical sites which are exposed to different environmental factors which makes them more failure-prone. Offloading tasks on such unreliable edge servers would postpone or prolong the offloading process making it less efficient and potentially hazardous which can lead to life-threatening situations, degraded performance, or substantial economic cost. With the intent of addressing the aforementioned reliability problem in the edge offloading setting, this thesis introduces an efficient and reliable offloading framework that predicts failures on the edge and cloud infrastructure to mitigate failures during offloading. Furthermore, we developed an experimental prototype for real-world deployment that improves the prediction of reliability by employing a machine learning approach for more efficient proactive offloading decision-making. We also propose an edge offloading framework that formally guarantees the feasibility of offloading decisions while balancing performance and reliability objectives in a distributed and failure-prone edge environment for latency-sensitive applications. Proposed solutions are evaluated using real-world datasets, preliminary simulations, emulators, real experimental testbeds, and synthetic applications. Presented works are not just theoretical but also practical for latency-sensitive applications in resource-constrained edge environments. Our contributions make edge offloading more efficient and reliable, driving progress toward a sustainable edge offloading ecosystem.

Contents

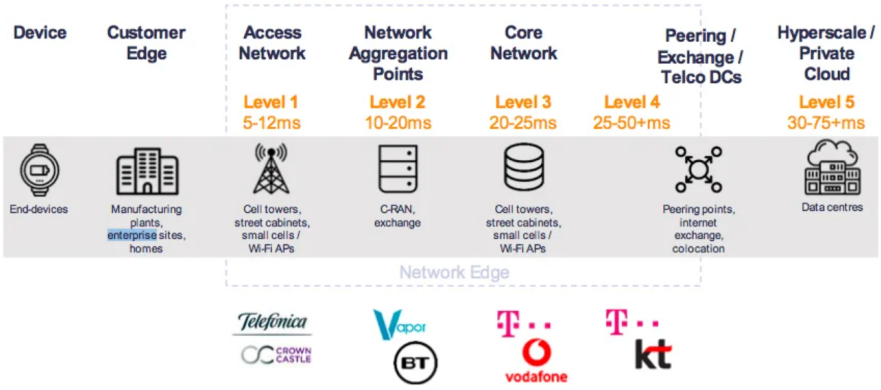
CHAPTER 1

Introduction

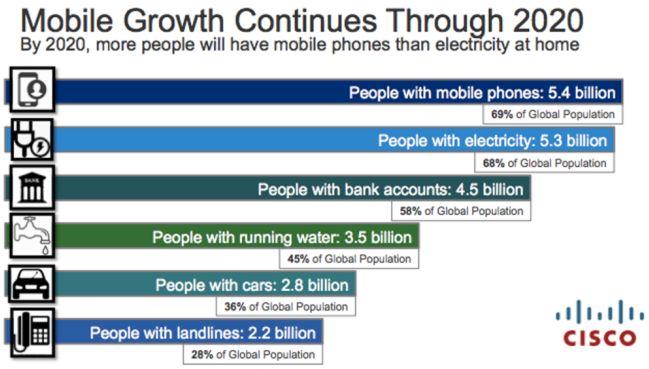
Nowadays, mobile devices become more popular and face increased user demands. Portability, mobility, and small physical dimensions of mobile devices enhance their usability and practicality. Moreover, wireless connections to remote and distant infrastructure make services ubiquitous and available on demand through mobile applications. Mobile hardware capabilities are enhanced to cope with such complex mobile applications. However, they still cannot handle resource-intensive mobile applications that require large data storage, high computation speed, or frequent wireless communication. Managing resource consumption properly will yield benefits for the user. For instance, prolonging the mobile device's battery life is considered the most important feature [20].

To overcome resource limitations, mobile cloud computing (MCC) is introduced. which allows mobile applications to be partially executed on mobile devices and partially on cloud data centers for better efficiency. However, this solution did not consider the high network latency and low bandwidth availability that come with distant cloud data centers. Figure 1.1a compares cloud latency with latencies observed at other parts of the network, ranging from cell towers to telco data centers, and shows that cloud latency outstands the most and varies between 30-75 milliseconds excluding processing latency on the cloud server and downlink latency from the cloud to end devices. This implies that round-trip latency can easily scale up to several hundreds of milliseconds or more depending on network congestion and processing workload. The cloud latency timescale exceeds strict real-time requirements of latency-sensitive applications such as virtual reality (VR) [KLBK24] and autonomous vehicles [DCH+23] where latencies should not go beyond 20 and 50 milliseconds respectively. Furthermore, the Cisco 2020 report shown in Figure 1.1b, outlines that in 2020 the number of mobile subscribers (5.4 billion) already has surpassed the number of people using electricity (5.3 billion), bank accounts (4.5 billion), running water (3.5 billion), cars (2.8 billion), and landlines (2.2 billion) due to massive growth in the adoption of mobile devices. In another report by Visual Capitalist [Vis22], the number of mobile devices has surpassed a number of people on the planet in

2023. This causes a big surge in traffic and workload on network and cloud infrastructure and a huge strain on managing the centralized cloud properly to deliver agreed service quality to end users. Furthermore, the mobile application market which values in 2023 at around 252.89 billion has been forecasted that it will continue its booming trend with an annual growth rate of 14.3% until 2030 [Gra23]. As previously mentioned, some of the new emergent mobile applications, like AV/VR and autonomous driving, are resource-intensive and require large amounts of resources to function in real-time where responses have to be within dozens of milliseconds which makes it almost impossible for resource-limited mobile devices or high-latency cloud to deliver. To conclude, we need innovative and efficient viable solutions for coping with real-time and resource-intensive mobile applications executed in resource-constrained mobile environments.



(a) Network latencies



(b) Mobile growth

Figure 1.1: Network and mobile statistics

To achieve energy efficiency on mobile devices and fast application response time, edge computing is introduced, where data processing is moved from the cloud on the network's edge in the users' proximity. Figure 1.2 taken from [AO18] shows edge computing architecture where edge servers are embedded between resource-limited user devices and far-distant massive cloud data centers. The geographical proximity of edge servers to

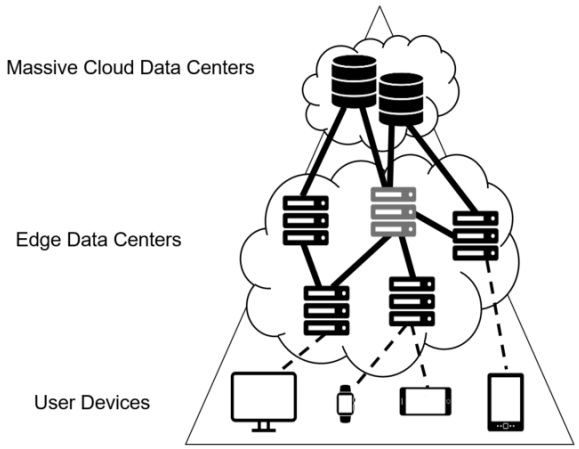
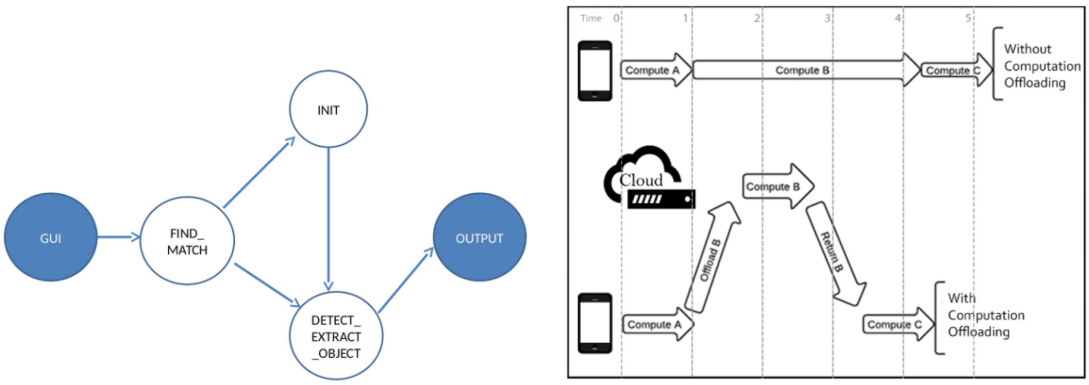


Figure 1.2: Edge computing architecture

user devices eliminates issues such as long network latency and unnecessary bandwidth consumption towards the cloud. Edge nodes have superior storage and computational capabilities compared to the mobile device, but inferior to the Cloud data center. Edge node can be any network device that has computational, storage, and network capabilities such as micro data center, radio base station, router device, gateway node, or similar.

To harness the benefits of nearby edge servers for better performance efficiency and energy savings on mobile devices, the offloading concept is introduced. The offloading is also known as cyber foraging which is a mimic of biological foraging where animals search and exploit nearby resources opportunistically for survival and flourishing. Analogously, the same applies to edge offloading where mobile devices exploit nearby edge resources to flourish in terms of enhancing their performance efficiency and improving energy savings. To offload mobile applications on the edge similarly to mobile cloud computing (MCC), the application has to be partitioned into smaller chunks called tasks. Each task is a computational and data workload that is part of a larger application and represents its smaller and interdependent computational unit. In real-world practice, tasks can be classes, processes, threads, methods, or functions, depending on the programming model applied (e.g. object-oriented) and on which system level the application is executing (e.g. platform processes). Some tasks are offloadable and other tasks are not. For instance, if an application task has a dependency on a physical function of the mobile device like a camera or sensor, then the application task is not offloadable. The mobile application's interdependent structure can be modeled via directed acyclic graphs (DAG) shown in Figure 1.3a of the face recognizer application example. Vertices represent tasks while edges represent task interdependencies. Blue-colored vertices represent non-offloadable tasks while transparent tasks represent offloadable tasks. All tasks have their resource demands where some tasks are computationally intensive requiring higher CPU frequencies or a larger number of CPU cores like DETECT_EXTRACT_OBJECT task. Other tasks are data-intensive tasks that require large disk storage capacities and



(a) Directed acyclic graph of face recognizer application

(b) Task offloading

Figure 1.3: Mobile application and offloading examples

faster network transmission rates like FIND_MATCH task. DAG captures application execution order where current tasks cannot be executed if all prior dependent tasks are not executed completely. To illustrate the offloading execution of tasks, Figure 1.3b shows the simple execution of the application which consists of tasks A, B, and C which are sequentially ordered. Task A is non-offloadable, task B is computationally-intensive and task C is lightweight in terms of resource demands. The upper example in the figure shows when the entire application is executed locally on the device without offloading. In this case, the response time is longer and drains more battery energy. The lower example shows the offloading case where non-offloadable task A is executed locally, computational-intensive task B is offloaded and executed on a cloud/edge server, and lightweight task C is executed locally. The offload case due to offloading of computation-intensive task B to the remote server, speeds up application response time and saves battery energy on mobile devices.

Edge offloading decision-making process is a complex process where many objectives and constraints have to be accounted for, ranging from device constraints, network conditions, cost, and latency, to resource capacities, server loads, and availability. In real-world settings, edge server resources are diverse and heterogeneous, and geographically distributed to many sites that adhere to different environmental conditions. Resource-wise, edge servers are smaller and cheaper compared to cloud servers and thus have fewer hardware redundancies and sometimes lack supporting systems (e.g. power backup) which impacts reliability. All aforementioned factors come into play during the offloading decision process. Because of that the goal and motivation of this work is to ensure an efficient and reliable edge offloading decision-making process that can deliver (near-) real-time response to latency-sensitive applications.

In the following Section 1.1, we describe the research problems and motivate our research direction of offloading decision-making in distributed and resource-constrained edge environments. Section 1.2 discusses the efficiency and reliability concerns of edge offloading

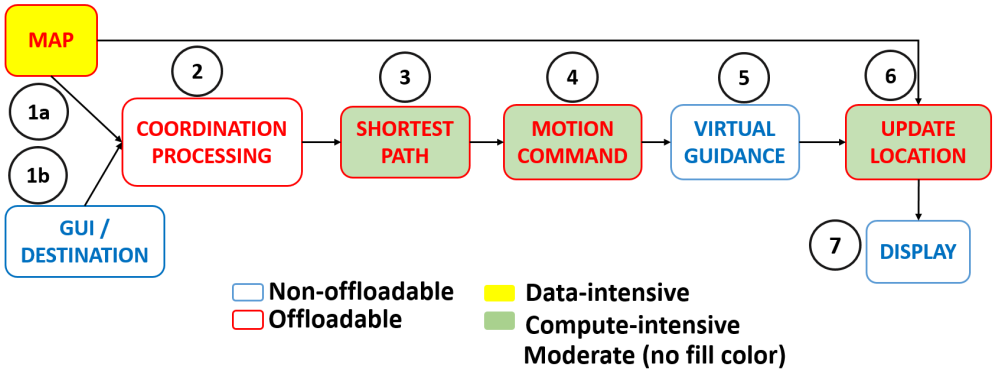


Figure 1.4: Motivational use case scenario with mobile augmented reality

and how research problems in this thesis relate to such concerns. In Section 1.3, we provide an overview of addressed research questions, while Section 1.4 presents the main scientific contributions. We discuss the importance of the research and contributions to the field in Section 1.5. The thesis organization is outlined in Section 1.6.

1.1 Problem Statement

In this section, we examine the motivational use case scenario of a mobile augmented reality application, for better and easier framing of research questions studied in this thesis. Latency-sensitive mobile applications are usually resource-intensive and require low-latency execution. An example is a Mobile Augmented Reality (MAR), where any significant delay hinders the users' experience [RGW+21, WYS21]. One of the typical MAR applications is personal live navigation called NaviAR. NaviAR supports users with real-time navigation by displaying virtual path information over the physical environment.

Figure 1.4 shows a typical NaviAR execution flow [WYS21]. The application is represented as a Directed Acyclic Graph (DAG) to model execution order and task interdependencies. The NaviAR consists of heterogeneous tasks where some are offloadable while others are not due to dependency on local device functions (e.g. camera). First, the destination location is taken as input from the user, upon which the map is loaded (steps 1a and 1b). Afterward, the geo-information is processed to identify the current and destination locations on the map (step 2). Then, the shortest route is calculated (step 3) based on which motion commands (i.e., left, right) are generated to navigate the user (step 4). Motion commands are rendered visually to guide the user in the physical environment (step 5). Finally, the user location is constantly updated (step 6) on the display until the user reaches the final destination (step 7).

Resource-intensive tasks (e.g. `MAP`, `SHORTEST PATH`) require offloading to the nearby edge servers to achieve desired performance [WYS21]. Failures on edge servers can affect offloading, causing additional delay [LMFH23]. Identifying reliable edge servers is of paramount importance to ensure a good user experience. Furthermore, during

mobility, mobile devices connect to different edge servers which have varying levels of resources and reliability. A reliable offloading solution is required to be robust to changing environmental conditions.

We address the challenges of achieving efficient and reliable edge offloading following:

- **Limited and heterogeneous resources**: Edge nodes have limited CPU cores, memory bandwidth, and storage capacity due to physical and economic constraints making efficient resource management even more critical compared to other distributed systems. Additionally, edge resources are heterogeneous because edge devices can be manufactured by different vendors (e.g. ARM-based CPU cores), deployed on geographical sites that can have unique environmental conditions (e.g. harsh cold climate), communicate with networks that have flexible topologies to accommodate different needs (e.g. mesh peer-to-peer for file sharing), and executing applications that depend on special hardware (e.g. GPUs for AI inference).
- **Volatile workloads**: Task offloading has to be highly adaptable to dynamic workload changes that stem out of intermittent connections (e.g. physical obstacles breaking line of sight), dynamic user activities (e.g. daily patterns, stadium events), mobility (e.g. joining and leaving radio cells) and diverse set of applications executed on mobile devices (e.g. AR/VR, video streaming). Edge resource-limited infrastructure makes it more susceptible to workload volatility and, if handled inefficiently can cause task failures like dropping.
- **Failures and reliability assesment**: Edge servers are more susceptible to failures compared to cloud counterparts due to a lack of supporting systems (e.g. backup power generators, fan units), physical damage amid less stable environments (e.g harsh cold climate), intermittent power supply (e.g. faulty power lines), network-wise disruptions (e.g. packet loss), and software errors (e.g. compatibility issues between adjacent microservices). Fault-tolerance methods applied in the cloud such as replication and checkpoint are less feasible in the resource-limited edge. Hence, the requirement for proactive failure-aware methods is a necessity, especially for latency-sensitive mobile applications where failures cause unacceptable delays.

1.2 Emerging Fields in Edge Offloading

Efficiency and reliability are the main concerns regarding edge offloading [JCG+19]. Efficiency is usually related to minimizing application response time, energy consumption on mobile devices, and monetary costs, which are induced due to outsourcing task execution on edge and cloud servers that are in ownership of resource providers. Resource providers rent their own resources for executing tasks of devices that do not have sufficient resources for executing tasks to satisfy desired requirements. Optimizing multiple objectives can be tricky since sometimes they can conflict with each other where lower response time can induce higher energy consumption[yCLlL23] and monetary

cost. For instance, offloading tasks from mobile devices on an edge server can accelerate application response time but can induce higher energy consumption if wireless network conditions are unfavorable causing longer data transmissions. The conflict arises due to changing environmental conditions which are typical for edge. Additionally, the reliability goal imposes requirements for maintaining continuous and consistent systems performance despite varying environmental conditions that causes failures and disruptions. Ensuring reliability in an edge offloading environment, requires failure mitigation techniques and predictive-based methods which can sustain application service availability and performance amid edge dynamism and volatility. In essence, edge offloading solutions should integrate adaptive and predictive methods into their framework to balance both efficiency and reliability aspects of the edge offloading systems.

Previously presented research challenges can be addressed by balancing between efficiency and reliability aspects. Available resource capacities and their allocation have to be optimized because limited and heterogenous resources are constraining the capability to handle serious failures adequatenly. Task offloading must be resource- and energy-aware to output optimized offloading decisions that respect resource limitations and strict latencies imposed by latency-sensitive applications. Also, taking into account the reliability levels of underlying resources for mitigating less reliable ones which can disrupt task offloading executions. Volatile workloads can contribute to worse efficiency where resource demands of diverse application tasks can quickly consumes limited edge resources and cause unpredictable performance for other hosted tasks. The inconsistent performance leads to a poorer reliability level and can deter further offloading decisions, leading to suboptimal performance. And lastly, proactively managing edge failures can enable their mitigation and counteracting them before they impact the system's performance. Addressing all aforementioned concerns will enhance edge offloading efficiency and reliability, leading to a more sustainable, scalable, adaptable, and fault-tolerant edge offloading ecosystem and landscape.

1.3 Research Questions

We provide an overview of our research questions that will guide us in our future research work.

1. *How to select Edge server that will satisfy performance by taking offloading failures into account? (RQ1)*
   Performance has a wide range of objectives that can be managed optimally in the sense of delivering Edge services to the user or smart devices to perform near real-time decisions. Depending on the type of application, various resources have to be allocated and available for application execution. Data-intensive applications require a big amount of network bandwidth and data storage resources, while computational-intensive application emphasizes more on CPU processing resources. Resource failures are appearing frequently because of the increasing functionality

and complexity of Edge services. Due to resource-limited capacity and prone to failures, offloading strategies have to be developed that can cope with the performance and reliability issue.

2. *Which failure mitigation or predictive approaches can be used to avoid Edge offloading failures? (RQ2)*
   As the systems' complexity grows, also the number of failures increases. Proactive prediction methods are required to improve service availability and decrease response time by avoiding failures. In case of failure, recovery time can be an overkill for a systems' availability, which gives the motivation to investigate failure prediction methods. Estimating failure probability is challenging in distributed systems due to the correlation between various types of failures. A single failure can lead to multiple failures in a short period. Temporal and spatial failure dependability can be exploited in such manner, that task offloading on unreliable nodes can be mitigated or minimized.

3. *Which resilience techniques can we utilize to enhance the reliability of the Edge Computing system? (RQ3)*
   Enhancement of reliability is considered as one of the crucial requirements for Quality-of-Service (QoS) guarantee [YLL15]. Resilience techniques such as replication are one of the possible solutions that can bring fault-tolerance and reliability to the system. They are needed for Edge services to fulfill strict requirements for latency-sensitive applications and avoid service interruptions. Violating those requirements can cause failover to redundant resources or recovery backup which can introduce additional delay and overhead in response. Failure forecasting methods that contain learning and inference features can deliver valuable input information to resilient techniques to perform informed decision-making that ensures service availability with a minimum cost of redundancy or failures.

4. *How can failure-aware Edge resource provisioning enhance systems' performance and reliability? (RQ4)*
   Run time decision support for resource management is challenging to ensure QoS parameters such as performance, availability, response time and reliability. Resource scalability and failure prediction are factors that need to be considered for accurate and optimal resource allocation and provisioning. It should handle Edge resource effectively to mitigate SLA (Service Level Agreements) violations that are negotiated between the Edge service provider and the user. Over-provisioning and under-provisioning have to be avoided to provide fairness to the stakeholders. Resource provisioning algorithms should consider both minimizing monetary cost for users and maximizing financial profit for a service provider.

1.4 Scientific Contributions

The state-of-the-art systems lack efficient and reliable edge offloading strategies for reliable edge offloading, especially targeting latency-sensitive solutions for (near-)real-time mobile applications. In this thesis, we address (i) novel edge offloading concepts and strategies showing the theoretical contributions and (ii) their practical applicability on real-world testbeds. In this context, driven by performance efficiency and reliability requirements of latency-sensitive and resource-intensive mobile applications, scientific contributions of this thesis are the following.

1. *Energy-efficient and failure predictive edge offloading (SC1)*
   Edge computing, due to distributed architecture that contains diverse resource and reliability characteristics, is prone to server and network failures that can postpone or prevent offloading thus affecting the overall system performance. We proposed a novel solution to model the energy consumption of mobile device and application response time assuming the resource and reliability diversity of the Edge Computing system. The model adopts the Markov Decision Process (MDP), which provides a formal framework for capturing stochastic and non-deterministic behavior of Edge offloading. We propose the Energy Efficient and Failure Predictive Edge Offloading (EFPO) framework based on a model checking solution called Value Iteration Algorithm (VIA). EFPO determines the feasible offloading decision policy, which should yield a near-optimal system performance. Evaluation is performed by offloading various mobile applications modeled as Directed Acyclic Graphs (DAG).

2. *Edge offloading of microservice-based applications (SC2)*
   Edge offloading is widely used to support the execution of near real-time mobile applications. However, offloading on edge infrastructures can suffer from failures due to the absence of supporting systems and environmental factors. We propose a fault-tolerant offloading method modeled as a Markov Decision Process (MDP) based on predictions performed through Support Vector Regression (SVR). SVR is used to estimate offloading service availability, which is used by MDP for offloading decisions. Our approach is implemented in a real-world test-bed and compared with the default Kubernetes scheduler augmented with hybrid fault-tolerance.

3. *Fast and reliable edge offloading using reputation-based hybrid smart contracts (SC3)*
   Mobile devices offload latency-sensitive application tasks to edge servers to satisfy applications' Quality of Service (QoS) deadlines. Consequently, ensuring reliable offloading without QoS violations is challenging in distributed and unreliable edge environments with diverse resource and reliability levels. We propose FRESCO, a fast and reliable edge offloading framework that utilizes a blockchain-based reputation system, which enhances the reliability of offloading in the distributed edge. The distributed reputation system tracks the historical performance of edge servers, while blockchain through a consensus mechanism ensures that sensitive reputation information is secured against tampering. However, blockchain consensus

typically has high latency, and therefore we employ a Hybrid Smart Contract (HSC) as a *reputation state manager* that automatically computes and stores reputation securely on-chain (i.e., on the blockchain) while allowing fast offloading decisions off-chain (i.e., outside of blockchain). The *offloading decision engine* uses a reputation score from HSC to derive fast offloading decisions, which are based on Satisfiability Modulo Theory (SMT). The SMT can formally guarantee a feasible solution that is valuable for latency-sensitive applications that require high reliability. With a combination of an on-chain HSC reputation state manager and an off-chain SMT decision engine, FRESCO offloads tasks to reliable servers without being hindered by blockchain consensus.

1.5 Significance of the Study

This study represents a quantum leap toward an efficient and reliable edge offloading ecosystem for latency-sensitive mobile applications amid of distributed and unreliable edge environments. The output of the study is critical findings that contribute to the broader ecosystem and landscape of distributed computing systems by addressing challenges (i) the emergence of resource-intensive and latency-sensitive mobile applications and unprecedented global mobile growth adoption, (ii) limited edge heterogeneous resources and their failure-prone and volatile behavior, and (iii) need for reliability-aware edge offloading frameworks for enhancing systems performance.

The end results of the study are benefiting system engineers, solution architects, software developers, and business decision-makers which are included in the entire process of designing, managing, developing, and deploying edge systems and projects. For integrating adaptive and proactive edge offloading frameworks in edge computing systems, the study provides deep insights for optimizing and maintaining consistent performance of latency-sensitive mobile applications and their reliability levels.

The study shows that theoretical research can be bridged with practical work by developing an experimental prototype that incorporates adaptive and proactive edge offloading frameworks. The prototype is validated by being deployed in a real-world environment instead of solely relying on simulations. Used methodologies enhances the offloading decision-making process and thus can be used as a platform for further development of efficient and reliable edge offloading systems.

The significance of conducted research around edge offloading goes beyond merely academic scientific contributions. It can pave the way for practical development and deployment of edge offloading solutions in real commercial environments that adds business value to cloud providers, telco operators, and industrial partners. The study supports further evolution of distributed computing systems by enhancing user experiences, cutting operational costs, and making supported systems more sustainable.

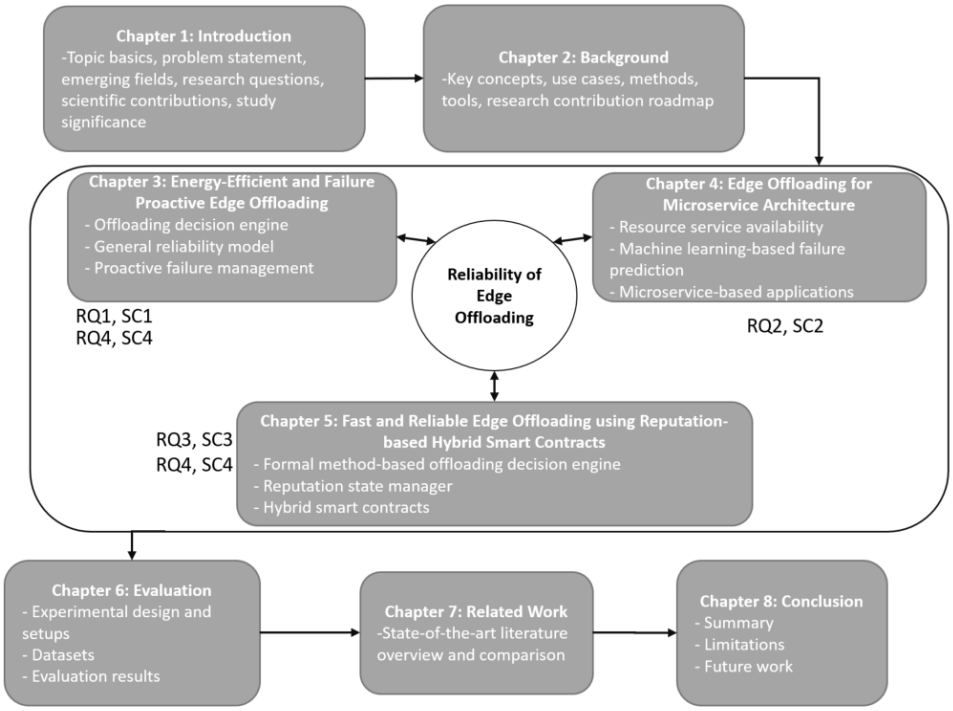


Figure 1.5: Thesis organization

1.6 Thesis Organization

The thesis is organized into eight chapters, presenting research problems, research solutions, and experimental evaluation results which are contained through benchmarking against state-of-the-art baseline solutions that are taken from the literature.

- *Chapter 1: Introduction*
  The chapter gives an overview of edge offloading, including problem statements, emerging fields, research questions, scientific contributions, and elaborations on the significance of the study.

- *Chapter 2: Background*
  The chapter provides background technical information on used methodologies, including tools and frameworks. A research contribution roadmap is also presented which outlines the main research outcomes of this work.

- *Chapter 3: Energy-Efficient and Failure Predictive Edge Offloading*
  The chapter presents an offloading decision engine that optimizes performance by predicting failure rates over time. The offloading solution is based on a general reliability bathtub curve model for edge servers, and their proactive management to mitigate failures before they occur.

- *Chapter 4: Edge Offloading for Microservice Architectures*
  The chapter introduces an offloading solution that focuses on resource service availability which is critically needed for offloading and executing microservice applications. The proposed solution includes a machine learning-based availability prediction model for edge resources by predicting their availability probabilities that vary over time.

- *Chapter 5: Fast and Reliable Edge Offloading Using Reputation-based Hybrid Smart Contracts*
  The chapter unveils an offloading solution that employs a formal method-based offloading decision engine which formally guarantees the feasibility of offloading decisions, and reputation state manager encoded as a hybrid smart contract that tracks edge server historical performance to provide reliable and trusted offloading service.

- *Chapter 6: Evaluation*
  The chapter describes experimental design, setup, schematics, datasets, and evaluation results for benchmarking proposed research solutions. Metrics, objectives, and benchmarking against state-of-the-art baseline solutions are explained in great detail.

- *Chapter 7: Related Work*
  The chapter gives a holistic review of the state-of-the-art literature by comparing our proposed research solutions to existing baseline solutions by addressing research gaps.

- *Chapter 8: Conclusion*
  The final chapter provides a summary of the main contributions of the thesis, and its limitations, and points out the direction for future research work.

The structured approach ensures an exploration of efficient and reliable edge offloading, and aligning theoretical knowledge with practical design and implementation.

CHAPTER 2

Background

In this chapter, we present background technical information about fundamentals of edge offloading and mobile applications, research focus, used research methods ranging from formal methods to machine learning, real-world testbed and infrastructures, software tools for research solution and experiment development, and research contributions overview which highlights research outcomes of the thesis.

Mobile cloud computing (MCC)
Cloud computing emerged as the need to provision on-demand scalable resources for high-complexity application execution with large data volumes. Traditional on-premise data centers had struggled to tackle resource consumption spikes with limited scalability due to fixed resource capacity. Also, on-premise data centers require a high upfront capital investment and in-house technical skills to manage such data centers. Hence, IT companies migrated software and applications on hyperscaler cloud data centers (e.g. Google Cloud, Amazon Web Services, Microsoft Azure), to gain elastic scalability to cope with resource-intensive applications and lower their total cost of ownership without requiring data center maintenance which is under hyperscaler cloud provider responsibility. It shifts costs from capital to operational expenses making it more cost-efficient. Also, cloud data centers support multitenancy by enabling application and service execution of multiple users on shared cloud infrastructure instead of purchasing separated dedicated servers making it also resource-efficient.

Mobile devices, on the other hand, had significant hardware advancements (e.g. system-on-chip) which enabled them to execute more complex mobile applications than just sending messages and conducting voice phone calls. Mobile applications grew in code complexity over the years to satisfy the diverse needs of users by delivering a richer set of features (e.g. graphical animations, live navigation, real-time syncing). To cope with resource-intensive mobile applications, the mobile cloud computing (MCC) concept was introduced and enables the execution of resource-intensive mobile applications on resource-limited mobile devices. MCC partitions the application into smaller tasks

and loads resource-intensive tasks on a cloud while lightweight tasks remain on mobile devices. However, a new class of applications emerged in recent years like AR/VR, cloud gaming, autonomous vehicles, and smart factories which are both resource-intensive and latency-sensitive. Offloadithosehat kind of tasks on the cloud or keeping them on a mobile device would be performance-wise counterproductive. Centralized cloud data centers are usually located in locations far away from end-users and are accessible through public internet which can have high levels of network congestion and reduced network bandwidth. This disqualifies cloud data centers from executing latency-sensitive application tasks. Hence, MCC suffers from high latency and network congestion, and the need for near-instantaneous processing has driven the evolution from MCC towards edge computing, where computation is offloaded to edge servers closer to mobile users. This transition from MCC to edge-based offloading represents a key shift in optimizing low-latency, high-performance mobile applications.

Edge offloading
Although MCC augments the computation and storage capabilities of mobile devices by offloading application tasks on resourceful but distant cloud servers, it fails to meet the strict time requirements of latency-sensitive mobile applications which are measured in milliseconds. Reasons for high-latency responses lay in physical distance (e.g. network hops), reduced network bandwidth, and high traffic volume. Therefore, edge computing was introduced as an intermediate middle layer embedded between cloud and mobile layers as a decentralized computing infrastructure. Edge computing provides cloud services and capabilities but on edge servers that are deployed in the vicinity of mobile users. Nearby edge servers reduce overall latency by executing application tasks near the task sources instead of executing tasks on far-distant cloud servers.

Edge offloading, on the other hand, is a resource management technique primarily enabled by the edge computing concept to enable real-time execution of real-time mobile applications on resource-limited mobile devices. Computationally and data-intensive tasks are dynamically offloaded from mobile devices to nearby resourceful edge servers with the objectives of reducing application response time, conserving battery energy supplies on mobile devices, and minimizing the monetary resource utilization costs of using rented edge resources from edge resource providers for task offloading and execution.

The most prominent edge computing instance is mobile (multi-access) edge computing (MEC) which targets edge computing frameworks on telecommunication networks and infrastructure. MEC extends cloud services on radio access networks where edge servers are deployed on radio network controllers, radio base stations, and aggregation points. This kind of deployment enables application execution at the edge of the network close to mobile users, thus dramatically reducing latency for latency-sensitive mobile applications. Recently, there have been plenty of MEC-enabled applications using 5G ultra-reliable and low-latency cellular connections like real-time video analytics, vehicular networks (V2X), mobile gaming, and AI-driven network optimization. Due to high-speed cellular connections, MEC provides (near-) real-time task offloading and execution thus enabling a great mobile user experience.

Latency-sensitive mobile applications
Cloud-based task executions fail to satisfy the strict time requirements of latency-sensitive mobile applications that require real-time response. To tackle the latency issue, the offloading technique has emerged as a critical solution for achieving high performance.

Latency-sensitive applications have pervaded a variety of application domains including a few representative examples:

- **Mobile augmented and virtual reality (AR/VR)** – Low-latency response is critical for AR/VR applications to provide an immersive and interactive user experience. Tasks like object detection, graphics rendering, and instant interaction require immediate reaction where any significant delays can degrade performance and experience. AR/VR application tasks offloaded on edge servers could reduce processing latencies to ensure smooth operations and save energy supplies of battery-powered AR/VR glasses.
- **Traffic safety** - Real-time communication is critical where autonomous vehicles communicate with nearby vehicles and roadside infrastructure to avoid collisions, and hazardous situations and regulate traffic flows. Edge offloading would minimize the communication latency necessary for enabling real-time decisions without compromising the safety of all included traffic actors.
- **Real-time image processing** - Facial recognition, medical imaging, and remote sensing are just a fragment of image processing applications that require real-time response. They can be both computational and data-intensive consuming CPU and network bandwidths at scale. Offloading such application tasks would reduce communication towards cloud servers and enable timely response.

In all aforementioned application domains, there are plenty of use cases where edge offloading can play a dominant role in enabling better performance, energy efficiency, and high reliability. Some of the mentioned applications are also used as a workload for evaluating proposed edge offloading solutions and benchmarked against baseline solutions.

2.1 Research Focus

The research focus of this dissertation is on adaptive and failure-aware edge offloading mechanisms that improve the performance and reliability of latency-sensitive applications in distributed and unreliable edge environments. The low-latency application execution requirement was normalized in past years due to the increasing emergence of real-time applications. Consequently, edge computing has co-emerged as a viable solution to handle resource-intensive mobile applications in real time. However, distributed edge environments are unreliable and failure-prone due to heterogeneous and limited infrastructure, and intermittent network connections. We focus on adaptive and failure-aware offloading techniques to improve both the performance and reliability of edge

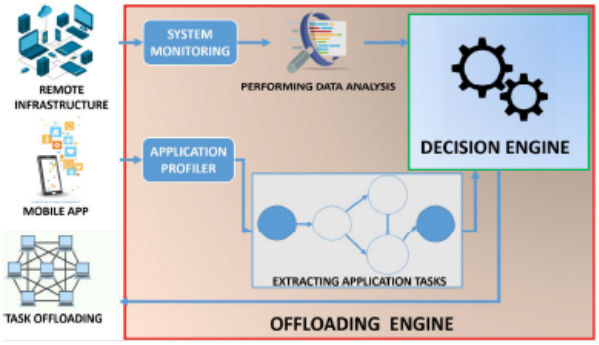


Figure 2.1: Edge offloading decision-making model

computing by selecting high-performing and reliable edge servers that minimize offloading failure rate.

The main focus of reliable edge-offloading approaches is on selecting the optimal target server while performing trade-offs and balancing between timing deadlines, resource constraints, performance objectives, energy efficiency, monetary resource utilization costs, and reliability factors. Hence, we investigate and explore formal methods, machine learning, and decentralized analytical approaches such as blockchain-based reputation systems to improve the performance and reliability of edge offloading systems in distributed and unreliable edge environments.

The research is organized around three key aspects:

- Failure predictive edge offloading - employing Markov Decision Process (MDP) for offloading decision-making and encoding failure probabilities via general reliability bathtub curve model for estimating failure events.
- Microservice-based edge offloading - Offloading of microservice applications based on Support Vector Regression (SVR) which assesses the availability probabilities of edge servers and forwards them to the MDP decision-maker agent for selecting edge servers for offloading.
- Blockchain-based reputation-aware edge offloading -The reputation system tracks the historical performance of edge servers and stores them on blockchain against any malicious tampering. The reputation scores are queried by mobile devices which offloads tasks accordingly.

The general offloading decision-making framework is illustrated in Figure 2.1 which consists of the following components:

- Application profiler: Extracts application structure and tasks' resource requirements from mobile applications.

- System monitoring: Observes available resource capacities on the infrastructure.
- Decision engine: Based on data inputs from the profiler and monitor, it determines the target server as an offloading location.

The goal of the presented research focus is to invent edge-offloading frameworks that can produce efficient and reliable offloading decision policies by improving the performance and reliability of edge-offloading systems. Also, the intent is to transfer theoretical offloading findings into real-world prototypes with breakthrough impact for latency-sensitive mobile applications.

2.2 Methods

Edge computing systems are designed to enable real-time and latency-sensitive decision-making by distributing computational tasks across edge nodes. These systems require robust and adaptive methodologies to optimize task offloading strategies, ensure data integrity, and enhance overall system reliability. In this section, we introduce the methodologies employed in this dissertation, each addressing different challenges in energy-efficient and failure-predictive edge offloading.

2.2.1 Markov decision processes (MDP)

One of the core challenges in edge offloading is determining the optimal decision policy for task execution—whether tasks should be executed locally on mobile devices or offloaded to edge/cloud servers. This decision-making process needs to consider device resource constraints (CPU, memory, and battery life), network conditions (latency and bandwidth), and application deadlines.

Markov Decision Processes (MDP) provide a formalized mathematical framework for sequential decision-making under uncertainty. In an MDP-based approach, the state represents the system's current computational and network resource availability, while actions correspond to offloading decisions (e.g., execute locally, offload to a specific edge node, or delegate to the cloud). The transition function models network fluctuations and dynamic workload conditions, and the reward function ensures energy efficiency, task completion success, and latency minimization.

MDP solvers can be implemented using dynamic programming or reinforcement learning, enabling the system to learn and adapt offloading decisions over time. By leveraging MDP-based optimization, the edge infrastructure can efficiently manage tasks and dynamically allocate computational resources, ensuring both reliability and responsiveness.

An example of formally modeling the offloading decision-making process under uncertainty is presented in Figure 2.2. MDPs provide a mathematical framework where decisions are made sequentially based on the current system state, available actions, and probabilistic transitions to future states. The MDP state space consists of various execution locations for an offloaded task, including:

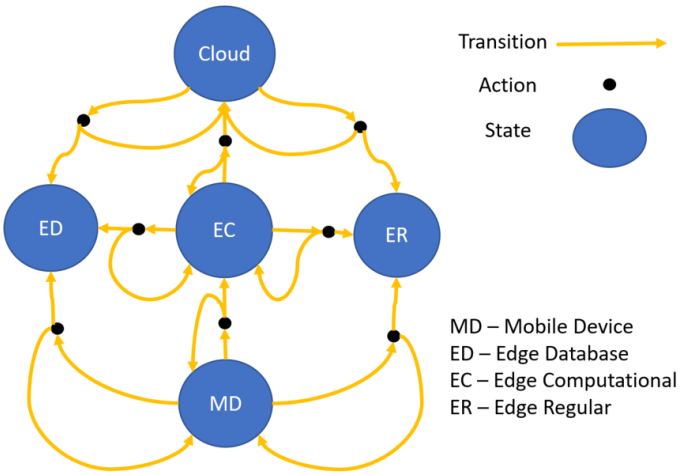


Figure 2.2: Markov decision process example for offloading decision-making

- MD (Mobile Device): Execution occurs locally on the user's mobile device.
- EC (Edge Computational Node): A high-performance edge node optimized for computational tasks.
- ED (Edge Database Node): A storage-focused edge node handling data-intensive processing.
- ER (Edge Regular Node): A general-purpose edge node that offers intermediate processing capabilities.
- Cloud: A remote cloud data center with abundant resources but higher latency

Each state transition occurs due to offloading decisions (actions), influenced by network conditions, device resource availability, and task complexity. These transitions are depicted in the MDP model visualization (Figure above), where states are represented as blue circles, corresponding to available execution locations. Actions (black dots) indicate decisions to offload tasks between execution nodes. Transitions (yellow arrows) illustrate the probabilistic movement between states based on network conditions and system constraints.

2.2.2 Support vector regression (SVR)

Support Vector Regression (SVR) is a machine learning technique widely used for predictive modeling across diverse domains where accurate estimation and forecasting are essential. SVR is particularly effective in capturing complex, non-linear relationships between input features and target variables, making it a robust choice for applications that require precise predictions with controlled error margins.

Unlike traditional regression methods, which minimize overall prediction error, SVR operates by defining a margin of tolerance (epsilon-insensitive zone) within which predictions are considered acceptable. By leveraging kernel functions, SVR maps input data into a higher-dimensional space, allowing it to identify underlying patterns and generalize well to unseen data.

SVR applies to a broad range of predictive tasks, including but not limited to:

- Failure Prediction: Estimating the likelihood of system failures in manufacturing, IT infrastructure, healthcare devices, and autonomous systems.
- Performance Forecasting: Predicting processing times, energy consumption, or service delays in various industries, from logistics to cloud computing.
- Anomaly Detection: Identifying irregularities in sensor data, cybersecurity threats, or equipment malfunctions.
- Demand Estimation: Forecasting resource utilization, consumer demand trends, or supply chain fluctuations for efficient planning.
- Environmental and Sensor Data Prediction: Anticipating weather patterns, air pollution levels, or seismic activity for real-time monitoring applications.

SVR enables adaptive and data-driven decision-making by providing accurate forecasts that guide system optimization, resource allocation, and risk mitigation. By training on historical data, SVR can proactively identify trends and anticipate potential disruptions, allowing for informed decision-making in uncertain or dynamic environments. Its ability to generalize across different domains and data types makes SVR a versatile tool for predictive analytics, enhancing reliability, efficiency, and performance optimization in complex systems.

2.2.3 Satisfiability modulo theory (SMT)

Satisfiability Modulo Theories (SMT) is a powerful formal method used for constraint solving, decision-making, and system verification across various computational domains. SMT solvers extend traditional Boolean satisfiability (SAT) solving by incorporating theories such as arithmetic, bit-vectors, arrays, data structures, and temporal logic, allowing for more expressive and complex constraint representations.

SMT is particularly useful in domains that require rigorous constraint validation and automated reasoning, enabling efficient problem-solving where multiple constraints must be simultaneously satisfied. By encoding problems as logical formulas, SMT solvers determine whether a given set of constraints is satisfiable and, if so, provide feasible solutions. SMT solvers offer several advantages, making them valuable across multiple disciplines:

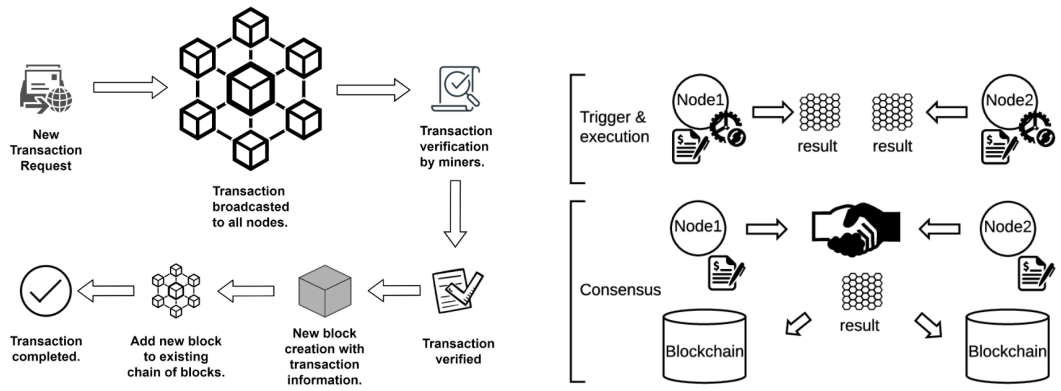
- Expressive Constraint Representation: They support rich mathematical theories beyond simple Boolean logic, making them ideal for complex decision-making scenarios.
- Automation and Scalability: They enable automated reasoning and constraint resolution, reducing the need for manual rule enforcement in large-scale systems.
- Efficient Search for Optimal Solutions: Unlike brute-force approaches, SMT solvers efficiently prune the search space, finding solutions faster and handling large problem instances.
- Formal Verification Guarantees: They provide provable correctness in systems where safety, security, or compliance is critical.

By integrating SMT-based reasoning into computational frameworks, organizations, developers, and researchers can ensure that systems operate within defined constraints, maintain robust security, and optimize performance in constrained environments.

2.2.4 Blockchain and smart contracts

Blockchain is a decentralized network that secures transactions through consensus, where all participating nodes agree on the current blockchain state. It is difficult to tamper transactions without compromising with a majority of nodes on a large-scale public blockchain (e.g. Ethereum). Thus, a public blockchain with consensus ensures tampering-resistance. Figure **??** [PNMH21] visually represents the fundamental process of how transactions are executed and validated within a blockchain network. It follows the key stages: (i) *new transaction request*, a user initiates a transaction which is the request is submitted to the network for processing, such as sending cryptocurrency. (ii) *transaction broadcasted to all nodes*, the transaction is propagated across all participating nodes in the blockchain network where nodes act as validators and maintain decentralized consensus, (iii) *transaction verification by miners* is the transaction undergoes verification, typically through a proof-of-work (PoW) or proof-of-stake (PoS) mechanism, where miners or validators ensure the transaction is legitimate by checking digital signatures, double-spending issues, and network rules. (iv) *transaction verified*, once validated, the transaction is deemed correct and ready for block inclusion. (v) *new block creation with transaction information* a miner (or validator) compiles verified transactions into a new block which is added to the blockchain through consensus mechanisms like PoW (in Bitcoin) or PoS (in Ethereum 2.0), (vi) *add new block to existing chain of blocks*, the new block is linked to the previous block, forming a tamper-resistant ledger which follows the chain structure that ensures security, immutability, and distributed record-keeping, (vii) *transaction completed* the transaction is finalized, permanently recorded in the blockchain, and considered immutable.

A smart contract is a self-executing program that automatically enforces agreed rules when certain events or conditions on the blockchain are met. Thanks to the tamper-resistant property of the blockchain, smart contracts can securely execute transactions that



(a) Blockchain mining process (b) Smart contract execution

Figure 2.3: Blockchain and smart contract

include sensitive information. However, the blockchain imposes long latencies and limits functionalities that smart contracts can provide by excluding non-deterministic operations (e.g. floating-point arithmetic)[BID21]. Additionally, blockchain is self-contained and accepts only transactions that occur on-chain. Overall, it is unsuitable for complex, latency-sensitive, and off-chain applications.

Figure 2.3b [DSX+18] illustrates how smart contracts are executed and validated in a blockchain-based system. It is divided into two main phases the execution phase and the consensus phase. Node1 and Node2 represent participants executing a smart contract. Both nodes independently compute the contract's logic and generate a result. This phase ensures each participant processes the same computation but does not yet reach a consensus. After execution, nodes synchronize their results to reach a consensus. The handshake symbol represents an agreement among nodes before finalizing the transaction. Once consensus is achieved, the results are recorded on the blockchain. This ensures all nodes have an identical state, preventing double-spending, fraud, or inconsistencies.

2.2.5 Reputation systems

A distributed reputation system operates without a central authority. Instead of relying on a single entity to maintain reputation data, it allows all participants to contribute, evaluate, and distribute trust scores. This ensures that no single entity can control or manipulate the reputation landscape, making it more resistant to bias or failure. At the core of such a system is the process of trust evaluation and reliability measurement. Reputation is built based on past interactions, where participants provide feedback on their experiences. The system collects this feedback and aggregates it into a reputation score, which helps others make informed decisions. However, reputation is more than just a numerical score—it is a reflection of both trustworthiness and reliability. A high reputation suggests not only that an entity has acted fairly in the past but also that it can be expected to fulfill its commitments consistently in the future. For a distributed

reputation system to function effectively, it must facilitate the sharing and verification of reputation information across a decentralized network. This requires a method for participants to exchange feedback, validate its authenticity, and ensure that reputation scores remain accurate over time. Unlike centralized reputation models, where the authority ensures data integrity, distributed systems must rely on mechanisms where trust emerges organically from the interactions between participants.

Reputation systems serve as a critical component in decision-making. Whether in online transactions, decentralized financial services, or collaborative networks, participants need a way to assess risk before engaging with others. Reputation helps reduce uncertainty, allowing participants to confidently enter agreements, knowing that past behaviors provide a reliable indication of future actions. Beyond trust, reputation systems also influence incentives and behavior. A well-functioning reputation system encourages honest and reliable participation while discouraging fraudulent or irresponsible behavior. When reputation is tied to benefits—such as access to better services, higher privileges, or stronger network influence—participants are naturally incentivized to maintain a good reputation. In contrast, those who act dishonestly or fail to meet expectations risk damaging their reputation, which can limit their ability to participate in the system effectively.

As distributed reputation systems grow in complexity, the challenge of maintaining trust in a decentralized environment becomes more pronounced. This is where blockchain technology presents a valuable integration. Blockchain offers immutability, transparency, and decentralization, aligning well with the principles of distributed reputation systems. One of the key reasons for integrating reputation systems with blockchain is ensuring data integrity. In a decentralized system where reputation data is collectively maintained, the risk of data manipulation or loss must be mitigated. Blockchain provides a tamper-resistant ledger where reputation records are securely stored and verifiable by all participants. Additionally, blockchain enhances reputation verification by eliminating the need for intermediaries. Since reputation scores influence decision-making, ensuring their authenticity is crucial. Blockchain allows for cryptographic verification of reputation data, making it easier for participants to assess trustworthiness without relying on a third party.

2.3 Testbeds and infrastructure

This section presents the hardware and network infrastructure utilized in our testbed for evaluating edge offloading in microservice architectures. The setup consists of heterogeneous devices, including low-power edge nodes, mobile devices, and high-performance cloud servers. The selected devices are Raspberry Pi 3B+, Huawei P Smart Z, and an AMD64 cloud-class server, each playing a distinct role in the distributed processing workflow.

2.3.1 Hardware and networking equipment

Raspberry Pi 3B+
The Raspberry Pi 3B+ serves as an edge node within our testbed, acting as an intermediary between mobile clients and more powerful processing resources. It is equipped with a Quad-core ARM Cortex-A53 processor clocked at 1.4GHz, 1GB RAM, and 64GB of local storage. These characteristics make it suitable for lightweight data processing tasks, sensor aggregation, and executing small-scale machine learning models. Furthermore, it provides wireless connectivity to nearby mobile devices and is configured with local DHCP and DNS services for managing mobile IP address allocation.

Huawei P smart Z
This mobile device is used for testing real-time application offloading and data transmission efficiency. It features a Quad-core ARM Cortex-A53 processor at 1.7GHz, 4GB RAM, and 64GB of storage. The device is particularly relevant for evaluating computational offloading strategies and network latency in mobile-edge computing scenarios. Offloading requests from the Huawei P Smart Z are performed over HTTP, leveraging a Flask-based web service deployed on edge nodes.

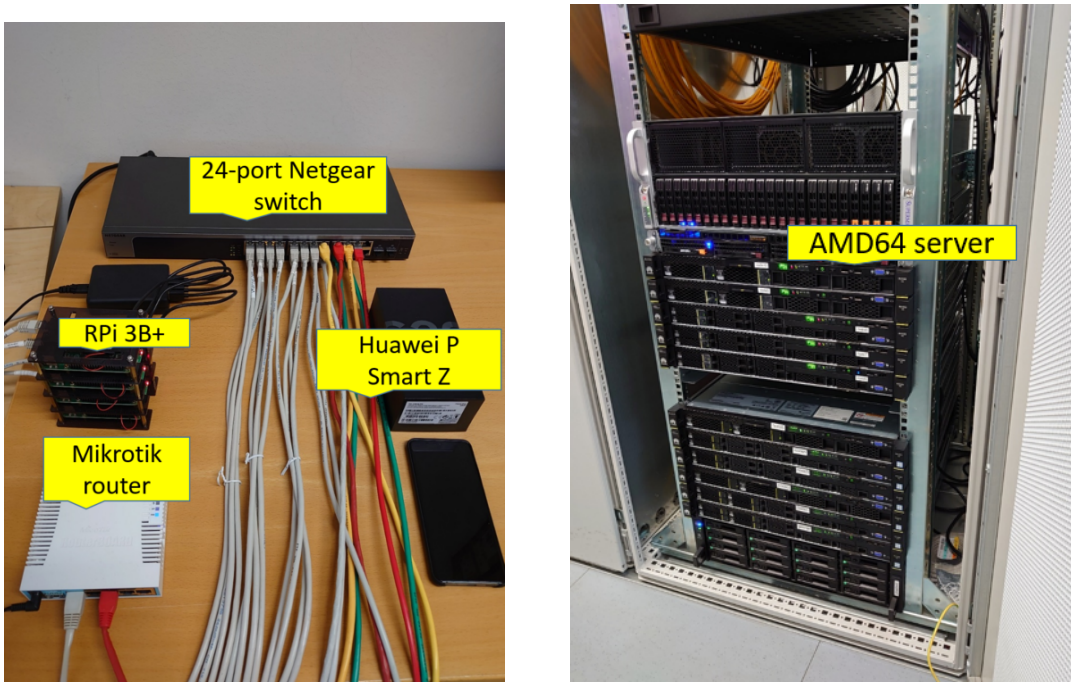
AMD64 cloud server
To simulate cloud infrastructure, we utilize an AMD64 server equipped with a 48-core Intel Xeon E5-2650 v4 processor running at 2.2GHz, 128GB RAM, and 1TB of storage. The cloud server plays a pivotal role in handling computationally intensive workloads that exceed the processing capabilities of edge nodes. It supports Kubernetes-based container orchestration for microservice deployment, and its high processing power allows us to benchmark edge-to-cloud task migration strategies.

Netgear 24-Port switch
The Netgear 24-port switch is a high-speed networking solution designed for environments that require low-latency, high-bandwidth data transfer. With Gigabit Ethernet across all ports, it ensures efficient communication between distributed computing nodes, sensors, and connected devices. Managed models offer VLAN segmentation and QoS, optimizing traffic flow and enabling network isolation for different workloads.

For scalability and reliability, the switch supports Link Aggregation (LACP) to increase bandwidth and Spanning Tree Protocol (STP) for redundancy, ensuring uninterrupted operation in dynamic network environments. Power over Ethernet (PoE) capabilities simplify infrastructure by delivering power to connected edge devices, while remote management options provide flexibility for configuring and monitoring network performance.

MikroTik router
The MikroTik router is a versatile networking device that enables efficient traffic management, security enforcement, and adaptive routing in distributed environments. Powered by RouterOS, it provides VLAN support, QoS policies, and traffic shaping, ensuring optimal resource allocation and seamless network performance.

With hardware-accelerated VPN support (WireGuard, IPSec, OpenVPN), the router



(a) Edge cluster equipment (b) AMD64 cloud class-server

Figure 2.4: Lab testbed

secures data transmission across distributed nodes, while its firewall and intrusion prevention system enhances security in dynamic network conditions. Load balancing and failover mechanisms improve network resilience, ensuring reliable operation in environments where connectivity and performance must be maintained without interruption.

The entire overall testbed used in our edge offloading evaluation is illustrated in Figures 2.4a and 2.4b. The first figure shows edge devices whereas the second figure shows a cloud server in TU Wien University's local data center.

2.3.2 Software

Python programming language
Python is a high-level, interpreted programming language widely used for automation, web development, data analysis, and system orchestration. Its extensive standard library and third-party ecosystem support rapid development across various domains. Python's readability and dynamic typing make it an ideal choice for scripting, system integration, and application prototyping. With frameworks like Flask, Python enables the development of lightweight web APIs and microservices, commonly deployed in distributed environments. The language also integrates seamlessly with asynchronous event loops (asyncio), parallel computing (multiprocessing), and containerized execution, making it well-suited for modern cloud-native applications.

Docker containers
Docker provides a lightweight, portable containerization platform that encapsulates applications and their dependencies into isolated runtime environments. It enables consistent deployment across different infrastructures, improving scalability and reducing compatibility issues. With support for multi-container applications via Docker Compose, networking between services, and resource constraints for CPU and memory, Docker ensures efficient workload distribution and isolation in cloud and edge computing environments. Its layered filesystem and image caching enhance deployment efficiency, making it a fundamental tool for modern DevOps workflows.

Kubernetes orchestrator
Kubernetes is a container orchestration platform designed for automated deployment, scaling, and management of containerized applications. It provides dynamic workload scheduling, self-healing capabilities, and rolling updates, ensuring high availability and efficient resource utilization. With features like Ingress controllers for traffic routing, persistent storage management, and service discovery, Kubernetes facilitates scalable and fault-tolerant microservices architectures. Built-in load balancing, autoscaling, and node affinity policies further enhance system reliability, making it a preferred choice for managing complex distributed applications.

Figure 2.5 [Ada21] illustrates the architecture of a Kubernetes cluster, showing how the master node (control plane) interacts with worker nodes to manage and deploy applications. The master node contains key components like the API server, scheduler, controller manager, etc., which work together to orchestrate workloads. Worker nodes, each running kubelet and kube-proxy, are responsible for executing the assigned workloads by running containers inside pods. In the task scheduling process, the kube-scheduler in the master node selects an appropriate worker node for a new pod based on resource availability and scheduling constraints. The API server manages all communication, ensuring that the scheduler's decisions are enforced. Once a node is assigned, the API server notifies the kubelet running on that worker node, which then pulls the necessary container images and starts the pod. Kube-proxy configures networking to allow communication between the new pod and other services, while the controller manager continuously monitors the cluster state to ensure that the desired number of replicas is maintained.

Ganache blockchain emulator
Ganache is a personal Ethereum blockchain emulator that enables rapid smart contract development and testing in a controlled environment. It provides a local blockchain instance with adjustable gas fees, instant block mining, and customizable account balances, allowing developers to simulate real-world transactions without incurring network costs. With JSON-RPC support, event logging, and integrated debugging tools, Ganache streamlines the development and testing of decentralized applications (dApps). It is commonly used in Ethereum-based smart contract development workflows, complementing tools like Truffle and Hardhat.

Solidity Ethereum smart contracts
Solidity is a statically typed programming language designed for writing Ethereum

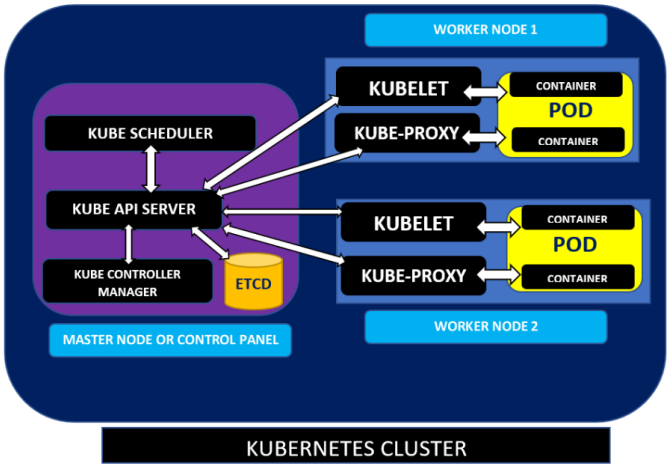


Figure 2.5: Kubernetes cluster components and their interactions

smart contracts that execute on the Ethereum Virtual Machine (EVM). It supports contract inheritance, function modifiers, and event-driven programming, enabling secure and efficient blockchain interactions. With built-in features like reentrancy protection, access control mechanisms, and gas optimization, Solidity ensures the secure and cost-effective execution of decentralized applications. It integrates with tools like Remix, Truffle, and Ganache for testing and deployment, facilitating the development of trustless, decentralized systems.

Z3 SMT solver
Z3 is a high-performance Satisfiability Modulo Theories (SMT) solver developed by Microsoft Research, designed for formal verification, constraint solving, and automated reasoning. It enables symbolic execution, theorem proving, and model checking, making it a powerful tool for program analysis, cryptographic verification, and security testing. With support for integer arithmetic, bit-vectors, and logical formulas, Z3 allows for automated decision-making in optimization problems, formal verification of smart contracts, and constraint-based software testing. It is widely used in AI, formal methods, and cybersecurity applications, ensuring correctness in complex computational workflows.

2.4 Research Contributions Overview

Figure 2.6 illustrates the research contributions of this thesis, which are structured into three main chapters. Chapter 3 introduces an energy-efficient and failure-predictive edge offloading framework, incorporating proactive failure prediction and adaptive decision-making for reliable task execution. Chapter 4 focuses on application-specific offloading strategies, optimizing computational, data-intensive, and latency-sensitive workloads for enhanced efficiency. Chapter 5 presents a trust-based offloading mechanism, integrating blockchain-based reputation systems and formal decision models to ensure reliable and

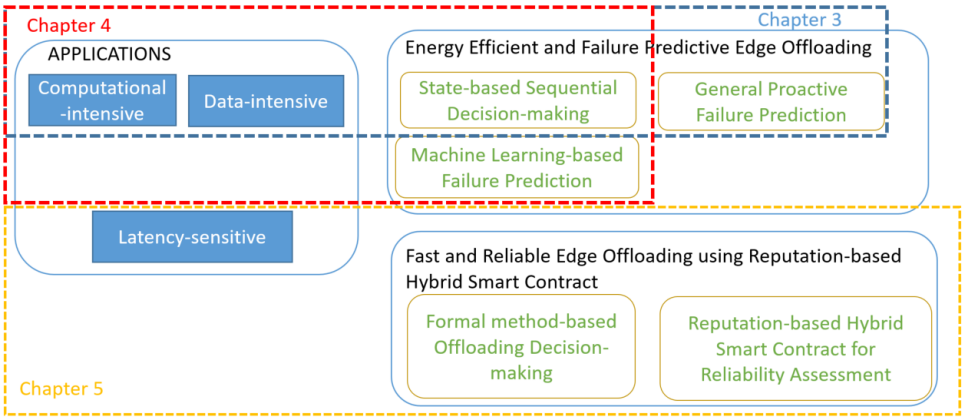


Figure 2.6: Research contributions overview

secure edge task execution. These contributions collectively improve efficiency, reliability, and adaptability in distributed edge environments.

CHAPTER 3

Energy Efficient and Failure Predictive Edge Offloading

In the previous chapter we established the key concepts and challenges inherent to edge offloading—particularly the issues of energy consumption, resource constraints, and reliability in distributed edge environments. In this chapter, we revisit the work presented in our conference paper "EFPO: Energy Efficient and Failure Predictive Edge Offloading" to further advance our investigation into reliable offloading mechanisms. This study introduces a novel framework that integrates formal modeling and verification techniques to address the dual challenges of performance efficiency and failure mitigation in heterogeneous edge computing systems.

Specifically, the conference paper formulates the offloading decision problem as a Markov Decision Process (MDP), capturing the stochastic behavior of failures alongside non-deterministic offloading choices. We employ the Value Iteration Algorithm (VIA) within the Energy Efficient and Failure Predictive Edge Offloading (EFPO) framework to derive near-optimal offloading policies. The main contributions of this work include (a) a comprehensive energy and time response model that accounts for the diversity in computational and network resources between mobile devices and edge nodes, (b) the development of the EFPO framework, which integrates MDP-based modeling with formal verification to address offloading failures systematically, (c) experimental evaluations demonstrating improvements in application response times.

The remainder of this chapter is organized as follows. Section 3.1 outlines the system architecture and MDP formulation underlying the EFPO framework. Section 3.2 details the implementation of the Value Iteration Algorithm and the offloading decision process. In Section 3.3, we present and discuss the experimental evaluation and its implications for both performance and reliability. Finally, Section 3.4 offers insights into future research directions for enhancing fault-aware edge offloading strategies.

3.1 MDP formulation and system architecture

MDP is a mathematical framework for modeling decision making in situations where outcomes are partly probabilistic and partly under the control of a decision-maker. This kind of framework can be used for modeling systems that exhibit probabilistic and non-deterministic behavior. Offloading decisions can be resolved in a non-deterministic manner as a *optimal decision policy*. An optimal decision policy describes the best action for each state in the MDP model, which yields optimal performance for the modeled system under given conditions. It can be obtained by formally verifying the MDP model using the model checking solution Value Iteration Algorithm (VIA). VIA algorithm focuses critically on expected value, in contrast to safety properties that are focused on the worst-case scenario. This allows us to exploit sampling and approximation more aggressively. There are other model checking solutions for MDP including Policy Iteration Algorithm, but VIA is preferred due to its theoretical simplicity and ease of implementation [Put14].

MDP is defined as a labeled transition system with state space S, where each state represents system configuration, action space A, where probabilistic transitions defines state trajectory from previous state s' to current state s and a reward function R which determines immediate reward (or cost) for taken action a while in state s. Therefore, MDP can be formally defined as a tuple $< S, A, P, R >$:

- S state space,
- A action set,
- $P(s, s', a)$ transition probability by taking action a in state s will lead to state s',
- $R(s, s', a)$ immediate reward received after transition from current state s to next state s' by taking action a.

The system architecture that we model with MDP is illustrated in Figure 3.1. It consists of five offloading sites, a single mobile device (MD), three Edge servers, and a single Cloud data center (CD). The scenario is that ODE is running on the mobile device and offloads the tasks from a current executed mobile application on Edge servers or Cloud data center. Alternatively, it is performed locally on the device. The mobile device has inferior computation and data storage resources comparing it with Edge and Cloud. Edge servers, on the other hand, have inferior resources when comparing to the Cloud data center. We introduce three Edge server types: (i) *Edge database server* (E1) has larger data storage capabilities and network transmission rates for faster data transfer for handling data-intensive applications such as Facerecognizer, (ii) *Edge computational server* (E2) has larger computational capabilities as CPU processing speed suitable for computational-intensive applications such as Chess, and (iii) *Edge regular server* (E3) has intermediate resources suitable more for typical applications that do not have large requirement for computation or data storage, such as social media applications. Mesh

network topology is used in the architecture due to advantages such as system robustness in case of server or network failures. The system architecture is extendable for employing multiple instances of each aforementioned offloading site. MDP model checking solutions support scalability. This is an important feature which can cope with verifying larger system models. However, the price of larger system models is a larger state space that can cause state space explosion which disrupts the model checking process. In this study, we use five offloading site instances as illustrated in Figure 3.1.

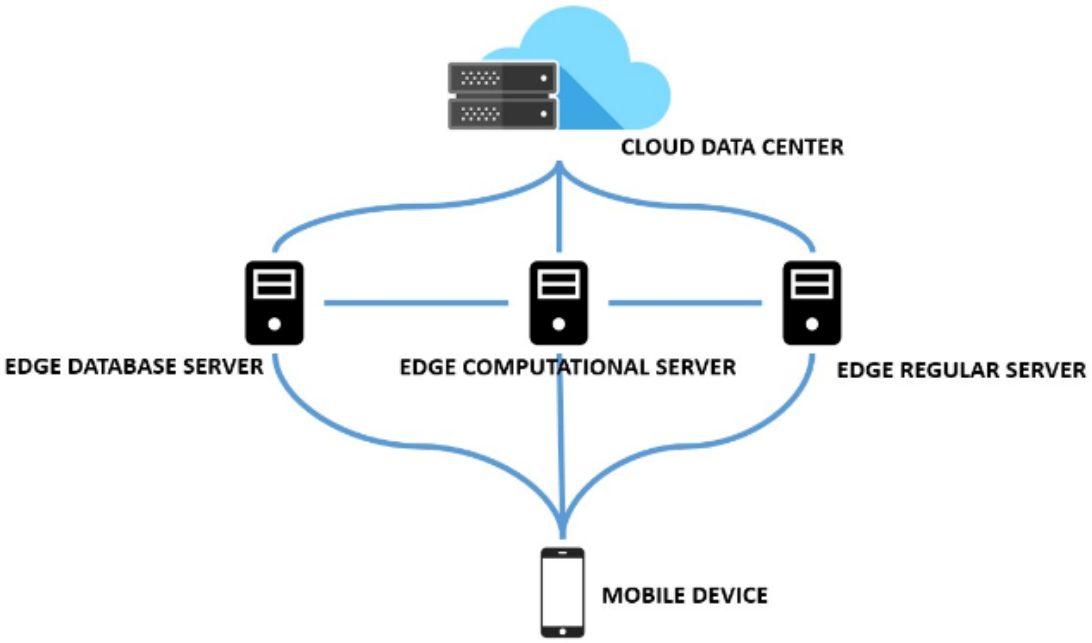


Figure 3.1: System architecture

Modeling of the mentioned system architecture as a MDP is as follows: (a) **The state space** S is defined as $S = \{MD, E1, E2, E3, CD\}$, where elements correspond to the aforementioned offloading sites, (b) **The action space** A is defined as $A = S$ where $a \in A$ represents offloading site decisions (c) **The discrete decision epochs** represents discrete time events when offloading actions are performed and defined as $T = \{0, 1, ..., n\}$, (d) **The transition probabilities** for each state $s(t)$ and action $a(t)$, gives quantitative information that the next state will be $s(t + 1)$, (e) **The reward function** used in this work considers two objectives, energy consumption and application response time, resulting in two reward functions, $R_{\mathrm{e}}(s, a)$ and $R_{\mathrm{t}}(s, a)$, which are combined in the overall reward function $R(s, s', a)$, (f) **Value iteration algorithm** (VIA) performs an approximation to the optimal value function for each state as in Equation (3.1) and yields optimal actions which represents our optimal offloading decision policy as shown in Equation (3.2). $\pi(s)$ represents offloading decision policy with initial state s and γ is

discount factor $[0, 1]$ which guarantees algorithm output convergence.

$$V(s) = \sum_{s'} P_{\pi(s)}(s, s')(R_{\pi\ (s)}(s, s') + \gamma V(s')) \tag{3.1}$$

$$\pi(s) = \underset{a}{\operatorname{argmax}} \sum_{s'} P_{\pi(s)}(s, s')(R_{\pi\ (s)}(s, s') + \gamma V(s')) \tag{3.2}$$

With Edge offloading, a mobile application (or a part of it) is offloaded from a mobile device to remote network infrastructure. In Figure 4.1, we summarize the offloading process. We assume that offloading is performed by a software unit running on the mobile device called the offloading decision engine (ODE). ODE is responsible for offloading application tasks on remote servers. It decides on which Edge server or Cloud data center each application task shall be offloaded, taking into account applications' and infrastructures' resource requirements and capacities. Once offloading is completed, the infrastructure executes tasks and sends the results to the location where the next application task will be executed. This process repeats until the application terminates.

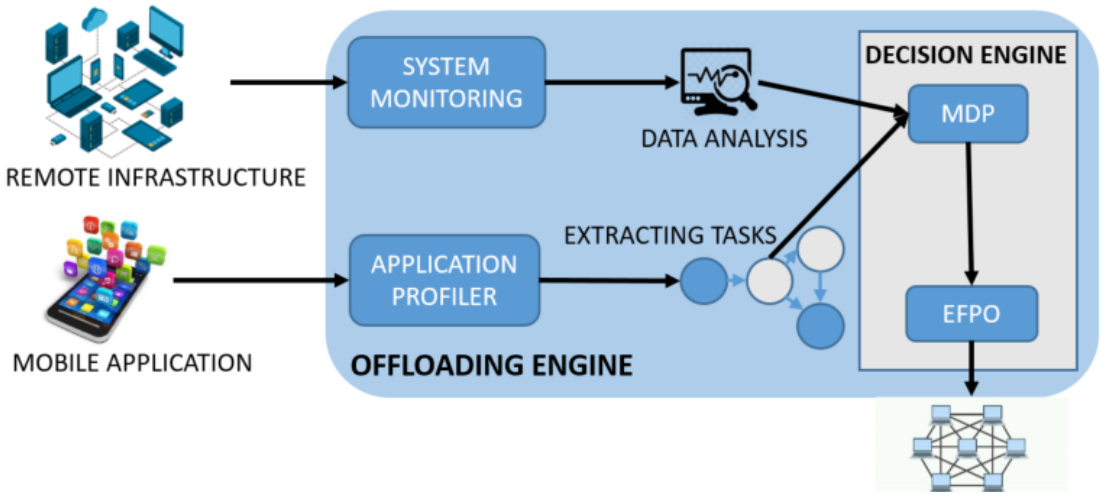


Figure 3.2: Edge offloading model

Main components of offloading engine in Figure 4.1 are: (1) *Application profiler* that profiles and extracts DAG structure and identifies tasks requirements and dependencies; (2) *System monitoring* that monitors data about remote infrastructure, and (3) *Decision engine* that collects data from the other two and performs offloading decisions. In this work, we focus on the decision engine. We assume that the decision engine has already collected the application DAG model annotated with requirements, the offloading possibility for each task from the Application profiler, as well as the information about remote infrastructure from System monitoring. The entire system in the decision engine is viewed as an MDP model, and by verifying it, we obtain optimal decision policy that is being used in the EFPO algorithm to determine which application tasks will be

offloaded on which offloading sites. MDP framework is used for modeling systems that exhibit probabilistic and non-deterministic behavior. Edge offloading fits to this scenario as offloading failures occur probabilistically and offloading decisions can be resolved in a non-deterministic manner. Executing ODE on the mobile device consumes energy that may be intolerable in some cases. The alternative is to execute ODE on a remote server and store the result on the device. In case of unstable connectivity, execution can continue on the device until the connection to the remote server is restored.

3.2 Offloading Model

Offloading sites has hardware characteristics based on which offloading and failure cost are computed. A site is defined as the vector $q = (f, ram, stor, r, l)$, where f is the CPU processing speed in millions of cycles per second, ram is the memory storage, $storage$ is the data storage capacity, r is the network bandwidth and l is the network latency. Application tasks are defined similarly, $v = (w, ram, d^{in}, d^{out}, off)$, where w is the CPU processing speed in millions of cycles per second, ram is the memory consumption, d^{in} is the input data size, d^{out} is the output data size and off is the binary variable that indicates whether the task offloadable.

When offloading application tasks on remote infrastructure, resource constraints must be respected. Defining valid offloading, the following conditions must be satisfied:

- $\sum_{v \in O_q(t)}(\omega_v) \leq f_q$
- $\sum_{v \in O_q(t)}(ram_v) \leq ram_q$
- $\sum_{v \in O_q(t)}(d_v^{in} + d_v^{out}) \leq stor_q$
- $\phi(v) = \emptyset$

$O_q(t)$ represents a set of application tasks that are executed in time moment t on offloading site q and $\phi(v)$ represents a set of application tasks that are input dependencies for task v. First, three conditions, validate that CPU, RAM and data storage capacities of offloading site q are not exceeded. The last condition validates that all input dependable tasks are executed before task v is ready for offloading and execution. The notation used for our simulation model is listed in Table 3.1.

3.2.1 Time Response Model

Time response cost is defined as $T_v = \{t_{md}, t_{e1}, t_{e2}, t_{e3}, t_{cd}\}$, where each element denotes the time response cost to execute component v on each of the offloading sites. $f_{md}, f_{e1}, f_{e2}, f_{e3}$ and f_{cd} are defined as the CPU clock speeds (cycles/second) of offloading sites. The total CPU cycles needed to execute the application task v is ω_v and f_i denotes

Table 3.1: Simulation parameters

Simulation parameters		
Offloading sites	q_i	i-th offloading site
Energy	e_{v_i}	Energy cost of task v if executed on offloading site q_i
parameters	t_{v_i}	Total time cost of task v if executed on offloading site q_i
Time parameters	$t^c_{v_i}$	Computational time spent executing task v on site q_i
	$t^i_{v_i}$	Time spent receiving input data to the task v on site q_i
	$t^o_{v_i}$	Time spent sending output data to the task v on site q_i
Hardware parameters	f_i	CPU clock speed (cycles/second) of offloading site q_i
	w_v	Total CPU cycles needed by the instructions of task v
	ram_q	RAM memory storage of offloading site q
	$stor_q$	Data storage capacity of offloading site q
Data parameters	d^{in}_v	Input data received by task v
	d^{out}_v	Output data sent by a task v
Power parameters	p_u	Mobile power consumption for uplink transmission
	p_d	Mobile power consumption for downlink transmission
	p_e	Mobile power consumption for local execution
	p_{idle}	Mobile power consumption at idle
Network parameters	r_{ij}	Network bandwidth rate between site q_i and q_j
	l_{ij}	Network latency between site q_i and q_j
Weight factors	ω_e	Weight factor for energy consumption reward function $R_e(s,a)$
	ω_t	Weight factor for time response reward function $R_t(s,a)$
Failure parameters	λ_{q_i}	Binary flag that indicates did failure occured on offloading site q_i
	$MTBF$	Mean time between failures
	$c^t_{v_i}$	Failure time cost on the offloading site q_i where task v is offloaded
	$c^e_{v_i}$	Failure energy cost on the offloading site q_i where task v is offloaded

CPU frequency of q_i offloading site. $t^c_{v_i}$ denotes the computational time of executing task v on site q_i and defined as:

$$t^c_{v_i} = \frac{\omega_v}{f_i}, \forall v \in V, \forall i \in [0,k] \tag{3.3}$$

Input and output data of task v are denoted as d^{in}_v and d^{out}_v, respectively. Also, t^{in}_{vi} and t^{out}_{vi} are defined as the communication time spent for input and output data transmission between q_i and q_j offloading sites, given by both equations:

$$t^{in}_{v_i} = \frac{d^{in}_v}{r_{ij}} + l_{ij}, \forall v \in V, \forall i,j \in [0,k], i \neq j \tag{3.4}$$

$$t_{v_i}^{out} = \frac{d_v^{out}}{r_{ij}} + l_{ij}, \forall v \in V, \forall i, j \in [0, k], i \neq j \tag{3.5}$$

r_{ij} and l_{ij} represents bandwidth and latency between offloading sites q_i and q_j respectively. t_{v_i} is the total time cost of task v to be executed on site q_i and it is defined as:

$$t_{v_i} = t_{v_i}^c + t_{v_i}^{in} + t_{v_i}^{out} \tag{3.6}$$

In case that successive tasks are executed on the same offloading site then according to Equation (3.6), total time cost is $t_{v_i} = t_{v_i}^c$, without data transmission costs from Equations (3.4) and (3.5) .

3.2.2 Energy Consumption Model

Energy consumption cost is defined as $E_v = \{e_{md}, e_{e1}, e_{e2}, e_{e3}, e_{cd}\}$, where each element denotes the energy cost to execute task v on the offloading sites. Energy consumption is considered only from mobile device perspective. Supplies on infrastructure are perceived as unlimited. It is assumed that the energy consumption e_v is computed as the amount of energy a mobile device spends while executing the application task or waiting for the application task to be executed on remote offloading sites. Energy consumption of a task v is then defined by e_{v_i} in Equation (3.7) where p_c is the mobile power consumption for local computation, p_d is the mobile power consumption when downloading data, p_u is the mobile power consumption when uploading data, and p_{idle} is the mobile power consumption in idle mode when application task is executed on remote infrastructure (Edge or Cloud).

$$e_{v_i} = \begin{cases} t_{v_i}^c \times p_c + t_{v_i}^{in} \times p_d + t_{v_i}^{out} \times p_u \\ t_{v_i} \times p_{idle} \end{cases} \tag{3.7}$$

The first case considers offloading from the mobile device, and the second when task is migrated on the remote infrastructure. Assumption about mobile power parameters when computing total energy consumption cost is considered as $p_u > p_d > p_c > p_{idle}$ from [KL10], where transmission consumes more energy then local computation or idle mode.

3.2.3 Reward Functions

Reward functions are used to model utilities or objectives which we want maximize or minimize through state sequence sampling. We define the overall reward function $R(s, a)$ which contains reward function for energy consumption $R_e(s, a)$ and reward function for application response time $R_t(s, a)$:

$$R(s, a) = \omega_e \times R_e(s, a) + \omega_t \times R_t(s, a) \tag{3.8}$$

ω_e and ω_t are defined as the weight factors for energy consumption and response time. The weight factors constraints are given as $\sum_m \omega_m = 1$ where $m = \{e, t\}$ represents

objectives such that $0 \leq \omega_e \leq 1$ and $0 \leq \omega_t \leq 1$. Both aforementioned reward functions are defined as follows:

$$R_e(s,a) = \frac{1}{1+e^{e_v}} \tag{3.9}$$

$$R_t(s,a) = \frac{1}{1+e^{t_v}} \tag{3.10}$$

In the evaluation, we used $\omega_e = \omega_t = 0.5$ which gives equal importance to both objectives. Weight factors can be altered to optimize the trade-off but this is out of this work's scope.

3.2.4 Failure Model

Failures can occur on Edge and Cloud infrastructure on the server or the network level. Server failures can be hardware faults (aging factor, power outage, hard disk failure, etc.) and software faults (OS failure, application crash, etc.), while network failures occur on network physical connections or network interface. Both failure types in this study are considered as an offloading failure.

Offloading failure occurrence in the simulation model is considered as a failure event $\lambda_{q_i}(t)$, which can occur in any discrete-time epoch t and offloading site q_i except mobile device which is considered as failure-free. Use case scenario of our most interest is that task offloading is performed on offloading site q_i at the same time moment t when failure event $\lambda_{q_i}(t)$ occurred. This interrupts offloading and execution process and causes additional cost in energy and time as well as forcing ODE to select other offloading sites q_j.

Offloading failure costs for time is defined as:

$$c^t_{v_i} = \begin{cases} t^{in}_{v_i} \\ t^{in}_{v_i} + t^{c}_{v_i} \\ t^{in}_{v_i} + t^{c}_{v_i} + t^{out}_{v_i} \\ 0 \end{cases} \tag{3.11}$$

The first case is when a failure occurs during input data transmission, the second case is when a failure occurs during execution, thus, input data transmission and computation time cost are included, thirdly, failure occurs during output data transmission, thus, all three time cost components are included. Finally, the last case is when there are no failures observed. Next, the offloading failure cost for energy is defined as:

$$c^e_{v_i} = \begin{cases} t^{in}_{v_i} \times p_{idle} \\ (t^{in}_{v_i} + t^{c}_{v_i}) \times p_{idle} \\ (t^{in}_{v_i} + t^{c}_{v_i} + t^{out}_{v_i}) \times p_{idle} \\ 0 \end{cases} \tag{3.12}$$

Cases are the same as in the previous equation. All time components are multiplied with p_{idle} since all failures are occurring only on remote infrastructure and during that period failure-free mobile device is in idle mode. Simulating failure events $\lambda_{q_i}(t)$ is done by Poisson distribution similar to [Wu18b]. As a rate parameter, we use Mean Time Between Failures (MTBF). It is a measure that gives quantified information about product reliability, defined as:

$$MTBF = \frac{T}{R} \tag{3.13}$$

T denotes total time and R number of failures. MTBF can be expressed in hours, days or any other time unit. The longer the MTBF, the product reliability is higher. It is an opposite measure of failure rates. Using it as a rate parameter in Poisson distribution, we obtain the number of discrete epoch events until failure event $\lambda_{q_i}(t)$ occurs on offloading site q_i. The final issue is predicting failure events. Failure predictability is defined as probability estimation:

$$P(t) = 1 - e^{-t/MTBF} \tag{3.14}$$

3.2.5 EFPO Algorithm

Algorithm 1 shows the EFPO algorithm for obtaining an energy-efficient offloading decision policy with failure predictability. The algorithm obtains an optimal policy from the VIA algorithm by exploring every state in the state space and selects the action with the lowest energy and time cost to be the optimal action. It performs this operation until it finds a feasible action that can be performed on the offloading site that did not experience offloading failure. After exploring the state space, the EFPO algorithm determines the feasible offloading decision policy that is then used to make an efficient decision for every future state the system encounters. EFPO algorithm is illustrated in Algorithm 1.

In line 1 we obtain optimal offloading decision policy from the VIA algorithm. However, optimal actions from VIA does not guarantee that they are feasible due to failures that happen during the runtime. We need to iterate all states in the MDP state space to obtain feasible optimal actions. In line 5, $\lambda_{T(s,a)}$ is a boolean variable which indicates whether offloading failure occurred on the offloading site $T(s, a)$ or not. If it is true, then we consider other offloading sites from vector Q which contains all action-state values returned from the VIA algorithm. Based on those values, we obtain the next action a. The algorithm continues to iterate until it finds an action that is feasible to offload on site which did not experience failure. Otherwise, the algorithm terminates on line 7. When a feasible action is found, it is stored in ω vector in line 12 and returned in line 17.

The EFPO algorithm finds the efficient offloading decision with an algorithmic complexity of $O(SA)$ per task offloading, where S is the state space and A is the action set. This algorithmic complexity does not reflect the complexity of the VIA algorithm. Although the EFPO algorithm can be considered to be a computationally expensive operation for resource-limited mobile devices, an alternative can be made so that the feasible offloading

Algorithm 1 Energy Efficient and Failure Predictive Edge Offloading Algorithm

1: $\langle \pi^*, Q \rangle \leftarrow \text{VIA}(S, A, P, R, s_0)$ ▷ VIA algorithm
2: **for** each state s in S **do**
3: $a \leftarrow \pi^*(s)$ ▷ Get optimal action
4: **while** True **do**
5: **if** $\lambda_{T(s,a)}$ **then** ▷ Escaped parenthesis
6: $Q \leftarrow Q \setminus \{(s, a)\}$
7: **if** $Q = \emptyset$ **then**
8: **return** "No solution"
9: **end if**
10: $a \leftarrow \arg\max_a Q(s, a)$
11: **continue**
12: **else**
13: $\omega \leftarrow \omega \cup \{(s, a)\}$
14: **break**
15: **end if**
16: **end while**
17: **end for**
18: **return** ω

decision policy is performed by the remote server. Therefore, mobile devices only store the matrix form of the results.

CHAPTER 4

Edge Offloading for Microservice Architectures

Building upon our previously established offloading framework, we now present a comprehensive investigation into the reliability challenges of microservice-based edge offloading. In the first phase, we focus on the architectural requirements and design principles for resilient offloading services that seamlessly integrate predictive failure management with real-time decision-making. This paper contributes:

- a detailed analysis of fault-tolerant offloading mechanisms tailored for microservice architectures, incorporating proactive failure prediction via Support Vector Regression (SVR) and a Markov Decision Process (MDP)-based decision engine
- a specification of essential engineering principles for deploying highly resilient, containerized offloading services on edge nodes, ensuring optimized performance and energy efficiency under dynamic network conditions.

In Section 2, we review the state-of-the-art in edge offloading and highlight key reliability issues. Section 3 details the design and implementation of our proposed fault-tolerant offloading framework, while Section 4 presents experimental evaluations and performance analyses. Finally, Section 5 discusses future research directions and concludes the paper.

4.1 Edge Offloading Framework

We envision an Edge Offloading Framework with the following components: (*i*) *Decision engine*, which computes the offloading decision policy; (*ii*) *Prediction engine*, which estimates future service availability on the remote offloading sites based on local historical failure trace logs; (*iii*) *Failure monitor*, which monitors failures of local system operations

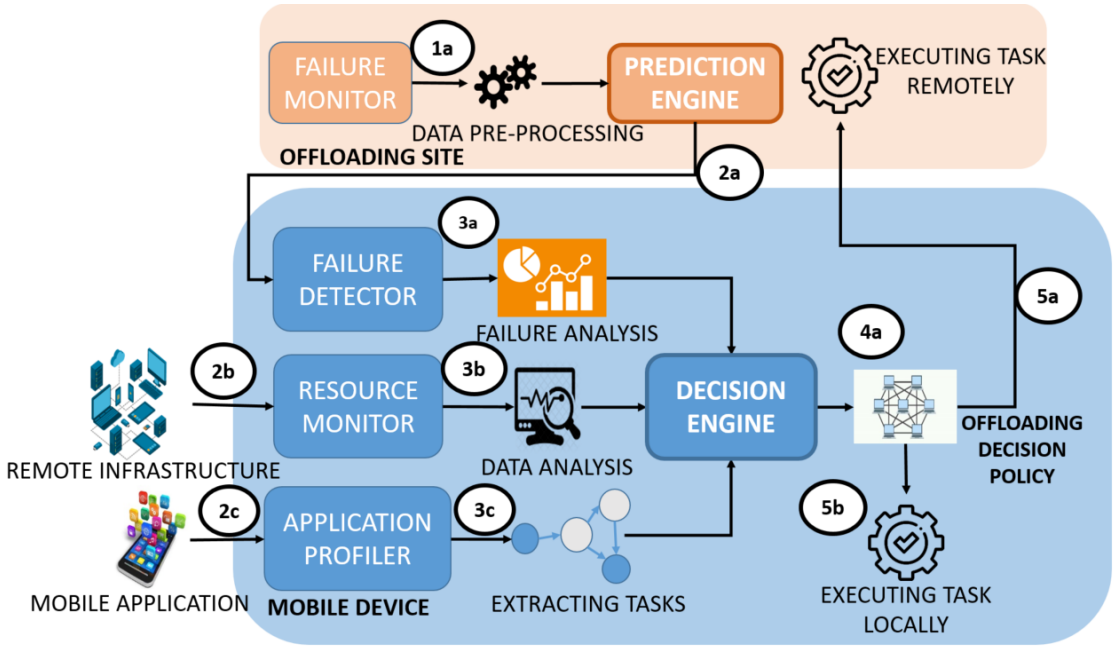


Figure 4.1: Edge Offloading Framework

on remote offloading sites; (*iv*) *Failure detector*, which detects failures during execution on remote offloading sites and collects the failure estimation data from prediction engine; (*v*) *Resource monitor*, which collects resource information about remote infrastructure; (*vi*) *Application profiler*, which profiles resource requirements of underlying mobile applications. Components are partitioned between mobile device and remote offloading sites as summarized in Figure 4.1.

The edge offloading process is described as following. First, the failure monitor collects historical failure traces and forwards them to the prediction engine (step 1a), which estimates service availability of each offloading site and sends these data to a mobile device (step 2a). Simultaneously, the application profiler and the resource monitor collect data about mobile application requirements and remote infrastructure capabilities (steps 2b and 2c). These data are used by the decision engine (steps 3a, 3b, and 3c) for offloading decisions (Step 4a), based on which tasks are offloaded (Step 5a and 5b).

However, decision engine micro-service can be a potential resource-intensive operation that can consequently reverse the offloading benefits. There is an alternative that the decision engine can be placed on a remote dedicated server and the mobile device acts as a thin client. This is also discussed in section 6.

4.1.1 Application Requirements

We focus on response time (RT) and mobile device battery lifetime (BL) which are mathematical models to estimate the aforementioned measurements that are necessary

for the decision engine to make informative offloading decision-making as in [ZAB19]. RT is defined as the sum of local computation time, uploading, and downloading data transfer time. Local computation time is defined as a ratio between CPU Millions of Instructions per Second (MIPS) and the number of task's instructions. Data transfer time is defined as a ratio between data size and network bandwidth plus the network latency.

BL is defined as the difference between total battery capacity and runtime energy consumption. Energy consumption is defined as the sum of local, upload, and download energy consumption. Each energy consumption component is equal to the multiplication of time and its power coefficient. We assume $p_u > p_d > p_c > p_i$, respectively power consumption for upload, download, local computation and idle [SSX+12]. Idle mode is the case when a mobile device is in an operational state but not used for executing application tasks. It has become necessary to model energy consumption since it is common that low-level energy information is usually not accessible. We pick the BL metric instead of direct energy consumption measurements since energy supplies on the infrastructure are perceived as unlimited while on the mobile device are limited despite power-saving modes. It is defined as a difference between total mobile battery capacity and total energy consumption during runtime.

4.1.2 Offloading Sites

We assume the infrastructure setup of [ZAB19], which allows to address diverse application requirements, i.e., data-intensive, computational-intensive, and moderate applications. We assume three Edge nodes types: (i) *Edge database server* (ED), with large data storage capabilities and fast network transmission rates for data-intensive applications; (ii) *Edge computational server* (EC) with greater computational power to support computational-intensive applications such as games and AI, and (iii) *Edge regular server* (ER) with intermediate resources suitable for applications that do not require a large amount of computation or data storage capabilities, such as live traffic navigation or posting updates on Facebook. Edge nodes are clustered together with the cloud data center (CD).

4.1.3 Failure Monitor

Failure monitor collects historical system trace logs on remote offloading sites for availability estimation. However, to detect failures on remote offloading sites it is required to employ a suitable failure detection strategy. The failure detector micro-service container on the mobile device is based on heartbeat-like failure detection. We employ heartbeat failure detection [hea] to collect traces. This approach sends ping messages to remote offloading sites at a fixed time interval. Offloading site is considered to be unavailable if the ping is not answered before timeout. When the message is received, then consequently, the site is removed from the list. This approach is responsible for maintaining the integrity of the entire system architecture. Recommended configuration settings for heartbeat protocols are time intervals of 150 ms and 10 timeouts [hea]. Therefore, the offloading site

Algorithm 2 Edge Offloading Algorithm

```
1: procedure EDGE_OFF_ALGO(S, A, train_dataset, tasks)
2:     energy_vector ← array()                          ▷ Store energy consumption of each off. site
3:     time_vector ← array()                            ▷ Store response time for each offloading site
4:     for each state v in tasks do
5:         for each state q in S do
6:             energy ← compute_energy(v, q)
7:             time ← compute_time(v, q)
8:             energy_vector.append(energy)
9:             time_vector.append(time)
10:        end for
11:    end for
12:    svr_avail_predict ← SVR(train_dataset)           ▷ Predict availability
13:    P ← compute_P_matrix(svr_avail_predict)
14:    R ← compute_R_matrix(energy_vector, time_vector)
15:    < π*, Q > ← PIA(S, A, P, R, s_0)                 ▷ PIA returns offloading decision policy
16:    return < π*, Q >
17: end procedure
```

is considered to be unavailable after 1.5 seconds, which captures the network variability due to different network delays between nodes.

Service Availability Estimator

We select the SVR algorithm for availability predictions, which provides prediction accuracy above 90% [MAUY19] and requires a small training dataset [dCMZLD11] as opposed to deep neural networks. The algorithm takes as input historical failure traces as input and its accuracy depends on hyper-parameters C and ϵ. C is a regularization parameter that models the ability to generalize the unseen data as a trade-off between the training and testing phases. ϵ parameter determines the level of regressor accuracy by controlling the width of the ϵ-insensitive area in the loss function $L(y, \hat{y})$ which measures the difference between actual data y and estimated data $\hat{y}$. Due to the near real-time requirements of our scenario, we use [CM04] parameter selection algorithm to reduce response time. C is defined in Equation 4.1 and ϵ in Equation 4.2,

$$C = max(|\bar{y} + 3\sigma|, |\bar{y} - 3\sigma|) \tag{4.1}$$

$$\epsilon = 3\sigma\sqrt{\frac{ln(m)}{m}} \tag{4.2}$$

where y is availability dataset, $\bar{y}$ represents the arithmetic mean, m is a dataset sample size and σ represents the standard deviation of the dataset. As a kernel solution, we use the Gaussian RBF kernel function which can estimate time-series data that exhibit non-linear behavior such as failures.

Algorithm 3 Edge Offloading Process

```
1: procedure EDGE_OFF_PROC(train_dataset, tasks)
2:     S ← (q_md, q_ed, q_ec, q_ed, q_cd)                          ▷ Offloading sites
3:     A ← (a_md, a_ed, a_ec, a_ed, a_cd)                          ▷ Action decisions
4:     < π*, Q >← EDGE_OFF_ALGO(S, A, train_dataset, tasks)
5:     for each state s in S do
6:         a ← π*(s)                                               ▷ for state s get best action a
7:         while True do
8:             if λ_T(s,a) then                                    ▷ if offloading fails then consider another action a
9:                 Q ← Q − {(s, a)}
10:                if Q = ∅ then return "No feasible solution"
11:                end if
12:                a ← argmax_a[Q(s, a)]                           ▷ get next best action a
13:                continue
14:            else
15:                ω = ω + {(s, a)}                                ▷ store feasible action a
16:                break
17:            end if
18:        end while
19:    end for
20:    return ω                                                    ▷ return feasible offloading policy
21: end procedure
```

4.2 Proposed Method

4.2.1 MDP offloading model

We employ offloading MDP in [ZAB19], MDP is a formal mathematical framework used for modeling discrete-time non-deterministic and stochastic control processes. It is defined as a labeled transition system in tuple form of $\langle S, A, P, R\rangle$ yields the following modeling assumption: (i) state-space $S = \{MD, ED, EC, ER, CD\}$ representing offloading site where a current task is offloaded, (ii) action set $A = \{MD, ED, EC, ER, CD\}$ represents the offloading site where to offload next task, (iii) P probabilistic state transition matrix representing offloading service availability, and (iv) R matrix of rewards associated with RT and BL. The goal is to maximize rewards by minimizing RT and maximizing BL. We use Policy Iteration Algorithm (PIA) [Put14] to iterate MDP and find a feasible offloading policy and yield offloading decision policy based on which offloading decisions are performed.

4.2.2 Edge Offloading Algorithm

The Algorithm 4 describes the edge offloading while Algorithm 3 executes offloading decisions. In Algorithm 4, the loop on lines 4-9 iterates over application tasks and computes BL and RT for each offloading site. In line 12, the SVR algorithm estimates offloading service sites' availability, based on the probability matrix P which is constructed on line 13. On line 14, the reward matrix R is computed and forwarded together with MDP's states, actions, and P to PIA, which synthesizes the offloading policy π (line 15). Policy π is executed during runtime by the Algorithm 3. Within the for loop (lines 5-19) offloading is performed. If the target offloading site fails during runtime, offloading is

classified as failed (line 8) and the next alternative is considered (line 12). The algorithm terminates when offloading is successful (lines 15-16) and returns a feasible offloading policy (line 20) or when no service site is available (line 10) and returns an error message.

4.3 Prototype Implementation

4.3.1 Cluster Networking

Wireless connectivity between the mobile device and the Kubernetes offloading cluster has to be assured to enable task offloading. The Raspberry Pi (RPi) single-hop away edge nodes provide wireless connectivity to nearby mobile devices. Configuring it as a Wi-Fi access point is possible due to the Wi-Fi shield component which is integrated into the RPi board and enables Wi-Fi protocol communication with remote nodes. Configuration requires installation of local DHCP and DNS servers which provide control over mobile IP address space. Moreover, it is required to re-configure the local NAT table of the RPi edge node to mask the mobile IP address with its' own so the packets are not discarded during packet routing.

Deploying the Kubernetes cluster over the public and private IP subnets is not straightforward. To address firewall and NAT translation issues, we deploy the private virtual networking solution called OpenVPN, which provides point-to-point communication and shared virtual IP address space. However, deploying the Kubernetes cluster which is composed out of private edge IP nodes and public cloud IP server yields firewall and NAT translations issues. We deploy the private virtual networking solution called OpenVPN which provides point-to-point communication and shared virtual IP address space between edge and cloud.

4.3.2 Micro-service Containerization

We developed our microservices using Python 3.6 programming language and containerized them using Docker. Containerizing Python micro-service applications using Docker makes the software deployment process much easier due to their modularity and fine granularity. Docker container images are created by bundling together Python micro-service source code and libraries that are necessary to run an isolated and independent container instance. Although Docker solves portability and system dependency issues, it can still be a challenge to build the Docker multi-CPU architecture container image for the underlying RPi ARMv7 and AMD64 architecture. Cloud-class host server to build a consistent Docker container image can be a significant obstacle for interoperability. We use the buildx command-line interface (CLI) plugin that utilizes machine processor emulator QEMU to build a common Docker container image for both CPU architectures available in the cluster, i.e., RPi ARMv7 and AMD64.

Micro-services on the mobile device (resource monitor, failure detector, application profiler, and decision engine) are developed using Python Kivy mobile cross-platform framework [**?**]. We developed it as a Python application for Android OS mobile devices.

These microservices do not have to be containerized. However, microservices can be placed on the dedicated offloading site instead (as part of the Kubernetes cluster) to reduce mobile devices' resource consumption. In this case, micro-services have to be containerized to be executed on the edge offloading cluster.

4.3.3 Service Deployment

Since offloading requests are performed by mobile devices through HTTP, we deploy Flask web service on each offloading site. Flask provides necessary web services without additional third-party components. We instantiate it as an additional microservice on the remote offloading site, together with the failure monitor and prediction engine, on a single Kubernetes pod. Each pod has its unique virtual IP address dispatched by the Flannel Container Networking Interface (CNI) plugin. Usually, Docker containers are not equipped with network interfaces so subsequent CNI plugin installation is required. We installed the Flannel CNI plugin due to its ease of deployment. We also employ the NGINX reverse proxy to redirect HTTP requests to appropriate offloading services. Combining NGINX web service on the Kubernetes cluster level with Flask micro web services on the offloading site, we can expose offloading sites to mobile devices.

CHAPTER 5

Fast and Reliable Edge Offloading using Reputation-based Hybrid Smart Contracts

In the previous sections we examined the challenges of offloading latency-sensitive mobile applications in unreliable edge environments, where conventional methods struggle to meet strict Quality of Service (QoS) requirements due to volatile resource availability and unpredictable failures. In this paper, we introduce FRESCO—a fast and reliable edge offloading framework that leverages a blockchain-based reputation system combined with hybrid smart contracts to ensure trust and timely offloading decisions. FRESCO addresses the inherent trade-offs between security and performance by maintaining sensitive reputation information on-chain via a Hybrid Smart Contract (HSC) while executing offloading decisions off-chain using a Satisfiability Modulo Theory (SMT)-based decision engine.

The main contributions of FRESCO include:

- A formal offloading decision process that guarantees feasibility by incorporating critical resource constraints and timing requirements, ensuring that offloading decisions satisfy latency and energy objectives
- The integration of a blockchain-enabled reputation system re-purposed to assess edge server reliability in terms of failure metrics, thereby mitigating the risks posed by volatile edge environments
- A hybrid approach that bypasses the high latency of blockchain consensus during decision-making while preserving tamper-resistant reputation data, thus achieving significant improvements in response time, energy efficiency, and QoS compliance

The remainder of the paper is organized as follows. Section II outlines the underlying methodologies and motivations for our approach. Section III details the system model, including the design of the HSC reputation state manager and the SMT-based offloading decision engine. In Section IV, we formalize the offloading optimization problem and present the FRESCO algorithm. Section V discusses the experimental setup, evaluation results, and a comparative analysis with baseline methods. Finally, Section VI reviews limitations and outlines future research directions.

5.1 System Model

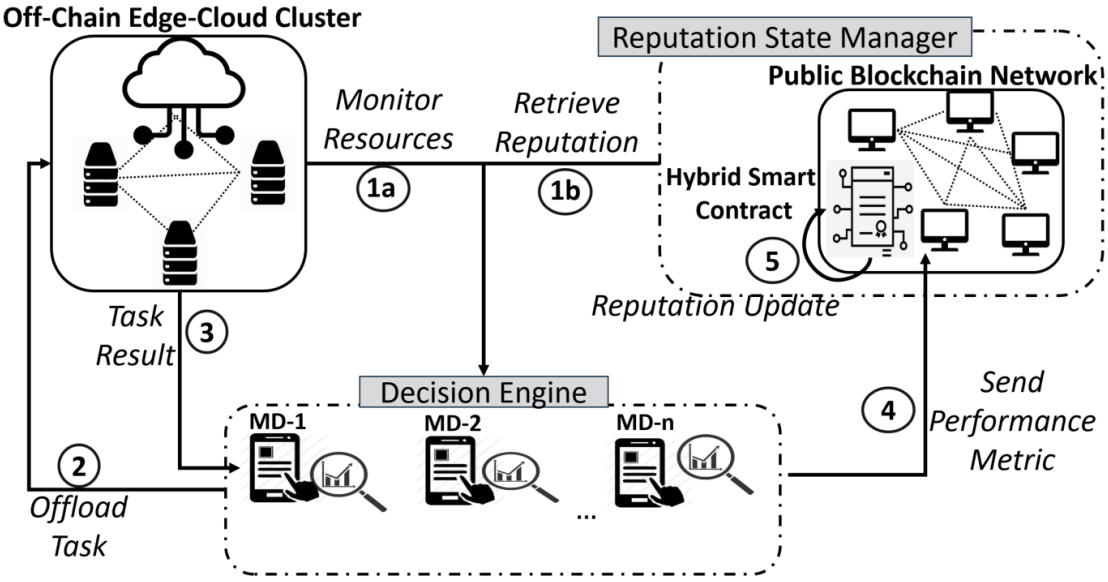


Figure 5.1: Edge offloading lifecycle model

Figure 5.1 illustrates the edge offloading lifecycle model, which manages offloaded tasks and estimates the reliability level of edge servers based on monitored performance. The two main components of our solution are the *reputation state manager* and the *offloading decision engine*. The reputation state manager is deployed as an HSC on the public blockchain network, estimates the critical reliability level of edge servers as a reputation score, and stores it securely on a public blockchain thanks to a consensus mechanism. The decision engine offloads tasks to an off-chain cluster based on reputation scores retrieved from the reputation state manager. The decision engine is often exposed as an intermediate central third-party service [LZC+20], making it vulnerable as a single point of failure in an unreliable environment. In our system, the decision engine is deployed on the mobile device, therefore its design choices should ensure a limited overhead, to guarantee fast decision time even on limited-resource mobile devices.

The lifecycle is executed as follows. In steps 1a and 1b, the mobile device retrieves the reputation score from HSC and monitors resources on the off-chain cluster. Based on

the procured information, the mobile device calculates offloading decisions and offloads tasks to the off-chain cluster in step 2. Task results are returned to the mobile device after execution in step 3. The mobile device records the performance metric (e.g., response time) and sends it to the HSC on the blockchain for evaluation in step 4. Finally, HSC compares the received performance metric to the deadlines and updates the reputation score accordingly. The lifecycle is repeated until the application is terminated. Noteworthy to mention, is that blockchain consensus is triggered only upon reputation update but not at reputation retrieval, which makes cached reputation score accessible in (near-)real-time.

5.1.1 Queuing and response time

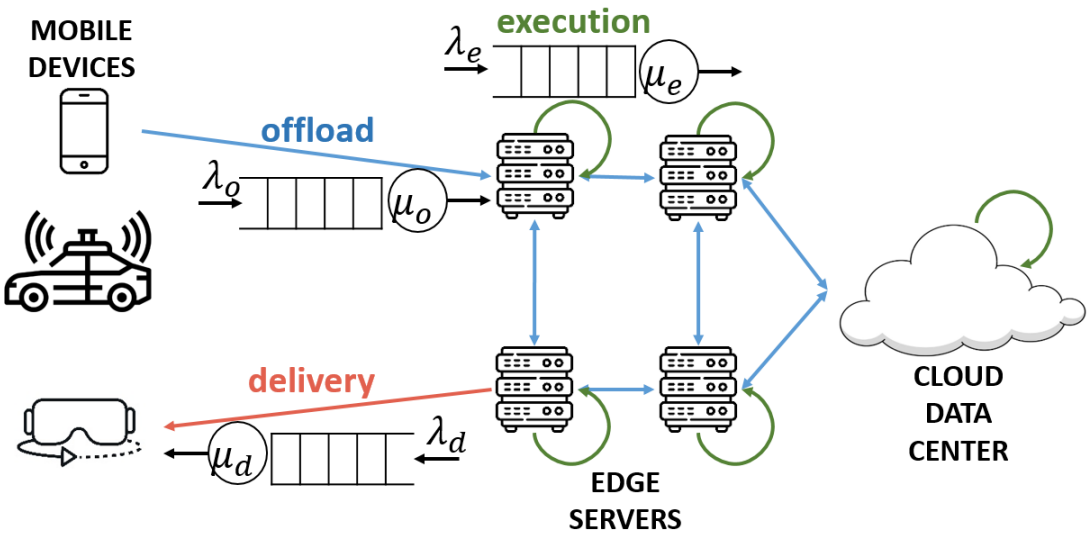


Figure 5.2: Dynamic queuing workload model

The workload on the shared infrastructure can be highly dynamic, where response times are hard to predict due to heterogeneous resources and tasks. To describe such dynamic behavior, we employ a queuing theory. Figure 5.2 illustrates the dynamic queuing workload model at the edge, which consists of three queuing parts. The *task offloading queue* models the task offloading, where multiple mobile task sources generate and offload tasks to remote servers through a shared communication channel. The *task execution queue* model the execution of the task, where the servers share their resources to execute multiple tasks. The *task result delivery* queue models the delivery of task results, where the results are sent back to the sources through the shared channel. The response time is defined as $RT(v,t,h) = T_o(h,t) + T_e(v,t) + T_d(h,t)$ where t is a task, h is a communication channel between pairs that can correspond to devices and servers, $v \in V$ where $V = N \cup \{m\}$ is a set of task execution nodes where a task can be executed on remote edge and cloud servers N and local mobile device m, $T_o(h,t)$ is offloading latency,

$T_e(v,t)$ is execution latency and $T_d(h,t)$ is delivery latency. We assume that the channels are on distinct frequencies to avoid interference [FA18] and employ a nonpreemptive First-Come-First-Served queuing policy that makes the performance predictable, which is important for high-reliability applications.

Communication latency

The shared channels in offloading and delivery are modeled as M/M/1 queues, which emulates practical transmission due to fair sharing and different bandwidths [FA18]. We will use the term communication and symbol $T_c(h,t)$ when referring to offloading and delivery.

The arrival process relates to task generation, modeled as Poisson with an arrival rate $\lambda_c(s)$ where $s \in S$ is a node in the task load generation set $S = G \cup N \cup \{m\}$ which consists of mobile devices G that have the sole function of generating tasks, remote servers N and mobile device m which has decision engine deployed. Task sizes are sampled from the exponential distribution with task size rate $data(t)$, accounting for the diversity of tasks' t resources. Generated tasks occupy shared resources, where bandwidth utilization $U_c(h,t)$ is defined as a ratio of generated tasks $\lambda_c(s) \cdot data(t)$ and total bandwidth $bw_{total}(h)$:

$$U_c(h,t) = \sum_{s \in D_c} \frac{\lambda_c(s) \cdot data(t)}{bw_{total}(h)} \tag{5.1}$$

where $D_c \subset S$ represents a subset of load generators in a specific communication direction, like $D_o = G \cup \{m\}$ represents task generators and mobile devices that generate tasks in the offloading channel $c = o$, or only remote servers $D_d = N$ on a delivery channel when task results are delivered after execution $c = d$.

The waiting time $w_c(h,t)$ is a delay due to resource sharing between tasks, which is a ratio of the current enqueued tasks and the available bandwidth (difference between total and utilized bandwidth):

$$w_c(h,t) = \sum_{s \in D_c} \frac{\lambda_c(s) \cdot data(t)}{bw_{total}(h) - bw_{util}(h)}, \tag{5.2}$$

The communication service time $\mu_c(h,t)$ models the actual transmission between devices and remote servers. Communication is subject to the Shannon-Hartley theorem [RM14] which defines the maximum data transmitted over a noisy link. Hence, the communication service time $\mu_c(h,t)$ is:

$$\mu_c(h,t) = \frac{data(t)}{bw_{avail}(h) \cdot \log_2(1 + \frac{p_c(h)}{n_0 \cdot bw_{avail}(h)})} \tag{5.3}$$

where n_0 is the noise spectral density and p_c channel transmission power. Finally, the total communication latency $T_c(h,t)$ is a sum of communication waiting time $\mu_c(h)$ and service time $w_c(h)$ such as $T_c(h,t) = \mu_c(h,t) + w_c(h,t)$.

Execution latency

The execution on the shared infrastructure is represented as a queuing M/M/1 network. Each server is a queue with independent rates and is interconnected with other queues to form a network [Bra08]. Remote server $n \in N$ utilization is accumulated load and is defined as:

$$U_e(v) = \sum_{s \in S} \sum_{t \in \mathcal{T}(v)} \frac{\lambda_e(s) \cdot MI(t)}{MIPS(v)}, \tag{5.4}$$

where $MI(t)$ is the number of instructions for task t and $MIPS(v)$ represents the capacity in terms of millions of instructions per second, $\lambda_e(v)$ is the arrival rate of the task and $\mathcal{T}(v)$ is a set of tasks assigned to the server v.

The waiting time $w_e(v)$ is the delay in task execution due to resource contention and is defined as:

$$w_e(v,t) = \frac{\sum_{s \in S} \lambda_e(s) \cdot MI(t)}{1 - U(v)} \tag{5.5}$$

The actual execution is defined as the service time $\mu_e(n,t)$, which is the ratio between the task load t and the server capacity v:

$$\mu_e(v,t) = \frac{MI(t)}{MIPS(v)}, \tag{5.6}$$

Finally, we define execution latency $T_e(v,t)$ based on the execution waiting and service times as $T_e(v,t) = w_e(v,t) + \mu_e(v,t)$.

5.1.2 Battery lifetime

Mobile devices are battery-powered, thus energy saving is critical. We introduce energy models of local execution and network transmission, major drivers of mobile energy consumption[TE+16]. We assume a mobile multicore execution power model [ASV+16] with power states [ZLLL17]:

$$p_{comp} = \sum_{i=0}^{cores} (\beta_{U_i} U_i) + \beta_{base} \tag{5.7}$$

where *cores* is the number of CPU cores, U_i utilization per core, β_{U_i} and β_{base} are energy coefficients for the operating state and idle power state when the workload is absent. However, the mentioned computation model does not capture switching overhead when transitioning between power states and multicore energy baselines, which are not the same as in single-core systems. Therefore, we expand the computational power model in [ZLLL17] to capture both aforementioned effects:

$$p_e(m) = b_{cores(m)} + \sum_{i=0}^{cores(m)} (\beta_{U_e(m)} \cdot U_e(m)) + \beta_{base} \cdot \frac{T_{idle}}{C} \tag{5.8}$$

where $cores(m)$ is the number of CPU cores on mobile device m, $U_e(m)$ utilization per core, $\beta_{U_e(m)}$ and β_{base} are energy coefficients for the operating and idle power states, $b_{cores(m)}$ is a CPU power baseline for a specific number of cores, T_{idle} and C are idle state time duration and number of power state transitions. The ratio of T_{idle} and C captures state switching overhead. Multiple deep power-saving idle states exist, but switching to deeper states induces longer latencies [ZLLL17] which prolongs the execution of latency-sensitive applications. Hence, we only consider the zero-level power-saving idle state with minimum switching overhead which induces negligible latency[ZLLL17]. The energy consumption of task local computation is based on Equation 5.8 and task execution latency:

$$E_{comp} = T_e * p_{comp} \tag{5.9}$$

The power model for network transmission $p_c(h_m)$ is derived from the Shannon-Hartley theorem:

$$p_c(h_m) = n_0 \cdot bw_{avail}(h_m) \cdot (2^{\frac{Ch(h_m)}{bw_{avail}(h_m)}} - 1). \tag{5.10}$$

where $bw_{avail}(h_m)$ is the bandwidth on the channel h_m of the mobile device m, and $Ch(h_m)$ is a channel capacity that is an effective limit on bandwidth due to noise. Subsequently, we can define the total energy consumption on the mobile device as the sum of local execution and transmission energy consumption models, defined as $E(v,t,m,h_m) = T_e(v,t) \cdot p_e(m) + T_c(h_m) \cdot p_c(h_m)$. Finally, the battery lifetime of the device $BL(v,t,m,h_m)$ is defined as the ratio between the differentiation of the full battery *bcap* and the total energy consumption until the time instant τ and full battery capacity as $BL(v,t,m,h_m) = \frac{bcap - \sum_\tau E(v,t,m,h_m)}{bcap}$. Subsequently, we can define the energy model of network transmission, which applies both in offloading and delivery cases:

$$E_{net} = T_{net} * p_{net} \tag{5.11}$$

.

$$E = T_e * p_e + T_c * p_c \tag{5.12}$$

where T_{net} can represent offloading T_o or delivery latency time T_d and T_e is execution time as defined in equation **??**.

5.1.3 Resource utilization cost

Edge and cloud are commercial services that bring monetary costs to mobile users who utilize resources owned by resource providers. Including the monetary objective in the decision-making is important to validate the approach in practical commercial environments where budgetary constraints can impact performance. The utilization cost is defined as:

$$PR(v,t) = \begin{cases} 0 & \text{if local} \\ T_e(v,t) \cdot cost_r(v,t) & \text{if cloud} \\ T_e(v,t) \cdot (cost_r(v,t) + cost_e) & \text{if edge} \end{cases} \tag{5.13}$$

The first case of local execution has no cost since no remote resources are rented. The second case brings cost when cloud resources are rented for task execution latency time $T_e(v,t)$. The cloud price $cost_r(v,t)$ for the execution task t on the target server v is defined:

$$cost_r(v,t) = cost_{cores}(v) \cdot MI(t) + cost_{stor}(v) \cdot data(t) \tag{5.14}$$

where $cost_{cores}(v)$, $cost_{stor}(v)$, and $data(t)$ represent cost units for CPU and data storage. The third case accounts for renting edge servers for execution latency time $T_e(v,t)$ where the price includes edge price penalty $cost_e$ for using low-latency service [DMB19].

5.2 Problem Formulation

In this section, we formulate our problem statement and provide a solution algorithm for fast and reliable edge offloading.

5.2.1 Reputation state manager

The blockchain-based reputation state manager distributes task incentives to encourage servers' participation in resource-sharing and successful task completion. Task incentives are computed based on the task completion time, meaning that shorter completion results in higher rewards. The rewards stimulate servers to perform task executions reliably and efficiently and compete with each other by offering better performance. The task incentive $inc_\tau(v,t,h)$ at time instant τ is defined as:

$$inc_\tau(v,t,h) = max\{\frac{\nabla - RT(v,t,h)}{\nabla}, 0\} \tag{5.15}$$

where ∇ is a timing constraint. If ∇ is violated, then no incentive is distributed to the server. The incentive is proportional to the difference between ∇ and the execution time when task execution is successful. The task incentive is normalized $[0,1]$ to prevent potential blockchain overflow.

The reputation model has to adhere to blockchain consensus restrictions. The consensus requires that on-chain updates are deterministic, to reach an agreement between blockchain nodes. Therefore, stochastic and floating-point arithmetic is not allowed on-chain [BID21]. Also, resource and time consumption on the blockchain is limited to prevent resource saturation. To address the consensus determinism requirement and limited resource consumption, we define a linear reputation model:

$$R_\tau(v,t,h) = R_{\tau-1}(v,t,h) \cdot (1-\omega) + \omega \cdot inc_\tau(v,t,h) \tag{5.16}$$

where $R_\tau(v,t,h)$ is the current reputation score, $R_{\tau-1}(v,t,h)$ is a previous reputation score, and ω is a weight factor to balance between new and old reputation. Although the model stores only the last score reputation, implicitly it accounts for multiple past values. It can be expanded to the equivalent formula, which tracks historical reputation performance by storing past reputation scores:

$$\begin{aligned} R_\tau(v,t,h) = {} & inc_1(v,t,h) \cdot (1-\omega)^{\tau-1} + \\ & \sum_{i=0}^{\tau-2} \omega(1-\omega)^i inc_{\tau-1}(v,t,h) \end{aligned} \tag{5.17}$$

The expanded reputation formula explicitly requires storing multiple reputation scores and several computational steps to output updated reputation score.

Reputation scoring ensures that only reliable servers are selected for offloading. Combining both incentives and reputation scores ensures a balanced trade-off where reputation is used as a long-term performance indicator and incentives as immediate short-term rewards to stimulate continuous improvement in server reliability and prioritize reliable servers.

To summarize, the presented reputation-incentive dual approach is encoded as an HSC on the blockchain to asses the reliability level of servers based on past performance. It also ensures trust against reputation malicious tampering for gaining incentives unfairly. The reputation update is according to the presented reputation model based on provided time measurements that are acquired off-chain from mobile devices which allows performance-sensitive offloading decisions to be made off-chain.

5.2.2 Offloading decision engine

Our goal is to efficiently offload tasks to minimize application response time and resource costs and maximize device battery. Therefore, we transform these individual objectives as a constraint optimization problem:

$$\begin{aligned} \min \quad & \sum_{t\in A}\sum_{v\in V} RT(v,t,h) \\ \max \quad & \sum_{t\in A}\sum_{v\in V} BL(v,t,m,h_m) \\ \min \quad & \sum_{t\in A}\sum_{v\in V} PR(v,t) \\ \text{s.t.} \quad RT(v,t,h) \leq \nabla, \quad & \forall v \in V, t \in A \\ BL(v,t,m,h_m) > 0, \quad & \forall v \in V, t \in A \\ PR(v,t) \leq pr, \quad & \forall v \in V, t \in A, \\ AP_\tau \leq D & \end{aligned} \tag{5.18}$$

where A is a task set of certain application, and AP_τ is an overall application response time until τ time instant. ∇, D and pr represent task timing constraint, application time deadline, and price constraint that can be application-dependant (e.g., 1500 ms reaction time in a traffic safety [LMP+21]), user-defined, or defined by developers for testing purposes. Battery lifetime is limited on mobile device m; thus, the goal is to avoid total discharge. Therefore, our main objective function is a linear combination of the objectives mentioned earlier:

$$\begin{aligned} score(v,t,m,h) = \alpha(RT(v,t,h) - RT(\hat{v},t,h))+ \\ \beta(BL(v,\hat{t},m,h_m) - BL(v,t,m,h_m)+ \\ \gamma(PR(v,t) - PR(\hat{v},t))) \end{aligned} \tag{5.19}$$

where α, β, and γ are user-defined weight factors for response, battery, and resource cost respectively ($\alpha + \beta + \gamma = 1$). Objectives with caret symbols are local optimum values. The goal is to find server n that minimizes the value of the *score*. The weight factors can be fine-tuned according to user preferences and subject to sensitivity analysis. However, we fix the weight factors and justify them accordingly in our experimental evaluation, in the subsection 6.3.2.

SMT encoding

Encoding is necessary to translate Equations 5.18 and 5.19 into a form, known as SMT formulas, that a target solver can automatically solve. The SMT combines first-order Boolean logic with constraint programming to express resource constraints and deadlines

of real-time system [CZTL16]. The SMT is lighter than machine learning solutions that are usually exposed as central third-party services [LZC+20], and it is suitable for less powerful devices [ATD21]. Additionally, we encode infrastructure capacities, task requirements, and servers' reputation as in Equation 5.20. It combines them all and uses an SMT solver to find a reliable edge server.

$$\begin{aligned} reputation &: (R_\tau(v,t,h) \geq rp) \wedge (0 \leq rp \leq 1) \\ batteryLife &: (BL(v,t,m,h_m) \cdot bcap - E(v,t,m,h_m)) \geq 0 \\ storageLimit &: \sum_{t \in O_\tau} data(t) \leq stor(v) \\ cpuLimit &: \sum_{t \in O_\tau} MI(t) \leq cpu(v) \\ memoryLimit &: \sum_{t \in O_\tau} mem(t) \leq mem(v) \\ taskReady &: \sum_{t \in O_\tau} (\delta_{in}(t) = \emptyset \wedge t \notin O_{<\tau}) \end{aligned} \tag{5.20}$$

The *reputation* constraint refers to server reputation which has to be above a certain threshold. To determine the reputation threshold rp, we apply k criteria from [IMRN20] where top k servers with the highest reputation score will be considered. We take a reputation score, which is minimum among k servers as the reputation threshold rp. Here, the *batteryLife* constraint verifies that the mobile device's battery is not drained completely. The *storageLimit* constraint verifies that the input and output data of all offloaded tasks O_τ until time instant τ does not exceed storage capacity on v target server. Similarly, CPU and memory capacities are labeled as *cpuLimit* and *memoryLimit* respectively. Finally, the *taskReady* label indicates that the application task is ready for offloading only when tasks' input dependencies $\delta_{in}(t)$ on prior tasks are completed (i.e., empty set) and the current task t was not part of a previous executed task set $O_{<\tau}$ before time instant τ. Finally, we combine Equation 5.18, 5.19, and 5.20 with logical AND operator into a single SMT logical formula. The final result of verifying the formula should be a reliable server location for offloading. However, solving the optimization function in Equation 5.18 is NP-hard, which is very time-consuming and impractical for real-time systems. We propose an online algorithm based on a heuristic in the next section, which can find a feasible solution in a reasonable amount of time.

5.2.3 FRESCO Algorithm

The offloading algorithm needs to solve the objective function and respect application deadlines, task timing constraints, and resource constraints. Therefore, we propose the FRESCO algorithm (Algorithm 4) for performing reliable edge offloading decisions. Inputs are the list of candidate servers, the server where the previous task was executed (*currSite*), the reputation scores per server, a list of tasks, a list of constraints, and user-defined weights. First, we declare a transaction list recording every offloading

Algorithm 4 FRESCO Algorithm

```
1: procedure FRESCO(candList, currSite, reps, tasks, constr, α, β, γ)
2:     transactions = list()
3:     for each task in tasks do
4:         for each candSite in candList do
5:             if RT(task, candSite, currSite) ≤ optRT then
6:                 optRT = RT(task, candSite, currSite)
7:             end if
8:             if EC(task, candSite, currSite) ≤ optEC then
9:                 optEC = EC(task, candSite, currSite)
10:            end if
11:            if PR(task, candSite, currSite) ≤ optPR then
12:                optRT = PR(task, candSite, currSite)
13:            end if
14:        end for
15:        for each candSite in candList do
16:            score(candSite) = α(RT(task, candSite, currSite) −
               optRT) + β(EC(task, candSite, currSite) − optEC) +
               γ(PR(task, candSite, currSite) − optPR)
17:        end for
18:        if candList.empty() then
19:            break
20:        end if
21:        selSite = SMTSOLVING(score, candList, reps, constr)
22:        if OFFLOAD(selSite, task) then
23:            d = compTaskConstrMeasure(selSite, task)
24:            transactions.append((d, selSite))
25:            break
26:        end if
27:        transactions.append((0, selSite))
28:        score.pop(selSite)
29:        candList.pop(selSite)
30:        reps.pop(selSite) True
31:    end for
32: return transactions
33: end procedure
```

attempt and its associated constraint (line 2). Then, we compute local optima for each objective (lines 6-12), which are used to calculate servers' optimization score (line 16) (first *for* loop on line 3). We iterate until the candidate list is empty or the task is successfully offloaded (*do-while* loop on line 17). If the candidate list is empty (line 18) then it exits from the *do-while* loop and returns accumulated transactions. Otherwise, the SMT solver on line 21 selects the server. If offloading fails, then the server is removed from the candidate list and its associate objective values (lines 27-30) and loops back on line 17. If offloading succeeds, the difference between execution time and the constraint ∇ is computed (line 23) and appended to the transaction list (line 24). The offloading transaction list is returned (line 32) and forwarded to the HSC for reputation update.

5.2.4 Complexity Analysis

The computational complexity of the FRESCO depends on $|T|$, which is the cardinality of the set of application tasks T, and $|N|$, which is the cardinality of the set of nodes N. This can be seen by the *for* loop on line 3, that is executed $|T|$ times, and *for* loops on

lines 4, 15, that iterate over N set. Also, *do-while* loop on line 17 is executed $|N|$ times in the worst case. However, the most impacting factor on FRESCO complexity is the complexity of the SMT solver (SMTSOLVING function on line 21). Since SMT solving generalizes the boolean satisfiability problem (SAT), which is known to be NP-complete, solving SMT is NP-hard. The SMT solving complexity depends on multiple factors, such as heuristic space search, clause learning, and problem size and structure [RKG18]. Therefore, the selection of an SMT solver has a strong impact on performance [Høf14]. Works like [ATD21] show the applicability of SMT solvers to latency-critical settings such as mobile edge offloading. Application to edge offloading and it was shown in the edge offloading literature that it can be placed on the mobile device as a decision engine [ATD21]. We will empirically evaluate FRESCO overhead, including the SMT solver in our experiment. Lastly, the complexity of the SMT solver is a $\Omega(n \log n)$ where n is the number of clauses (e.g. *batteryLife*, *storageLimit*, etc.) in the SMT formula [RKG18].

CHAPTER 6

Evaluation

This chapter presents the experimental evaluation of the proposed approaches described in the previous chapters. We first evaluate the energy-efficient and failure-predictive edge offloading framework in Section 6.1, featuring three core mechanisms: adaptive failure prediction (Section 6.1.2), dynamic resource management (Section 6.1.3), and optimal offloading policy synthesis (Section 6.1.4). In Section 6.2, we assess the technology and system components implemented for robust offloading in microservice-based architectures, detailing the design choices and real-world prototype deployment. Finally, in Section 6.3, we present the experimental evaluation of the reputation-based hybrid smart contracts approach, including an in-depth analysis of testbed configurations, input datasets, and performance metrics related to decision latency, energy efficiency, and QoS compliance.

6.1 Energy Efficient and Failure Predictive Edge Offloading Evaluation

6.1.1 Experimental Setup

At the time we write, there are several state-of-the-art model checker tools available such as UPPAAL [B+06] and PRISM [K+11]. UPPAAL verifies non-deterministic and time-critical systems. Moreover, UPPAAL Statistical Model Checking (SMC) [DL+15] extension supports modeling and verification of the systems which exhibit both probabilistic and timed behavior. Nevertheless, the tool does not support MDP models. PRISM, on the other hand, is a tool for formal modeling and analysis of systems that exhibit random or probabilistic behavior. MDP modeling and verification are supported but PRISM modeling language requires that every aspect of the system, including simulation utilities (e.g. DAG models, Poisson distribution, parsing dataset, etc.) must be abstracted as state machines, which limit the expressiveness of our simulation. This can also cause time overhead in the verification process, for instance, the state space explosion. For

this reason, we use our simulation framework implemented in Python which provides relatively simple syntax, diverse mathematical sampling distributions available as simple application programming interface (API) calls and MDP solver toolbox [CC+14] for MDP modeling and verification. This framework supports DAG mobile application scheduling, energy consumption, time response, offloading failure models, simulation and distribution of failures, and Edge/Cloud infrastructure model. Moreover, it can be expended with other objectives due to modular architecture. The input of our simulation framework is the infrastructure model which includes hardware specifications of the computational nodes, network characteristics, and mobile application setup.

The evaluation scenario is as follows. ODE on the mobile device decides on which offloading site shall application task be offloaded and executed. Energy consumption and response time are affected by hardware characteristics of the site as well as network links between the sites. It is assumed that only data is offloaded from site to site, while computation is replicated on each of them. Efficient selection of replica sites in a large network is left as future work. Here, we only consider a local partition of the system. Additionally, failures on sites where tasks are offloaded prolong time and energy and ODE is forced to offload tasks on another site possibly without failure. Energy and time cost after offloading failure are defined by Equations (3.11) and (3.12). Failure can occur on links and servers. Depending on which part failure occurred, energy failure cost $c^t_{v_i}$ and time failure cost $c^e_{v_i}$ are computed accordingly.

Computational nodes

The infrastructure model used in our simulation includes five computational nodes. We have a mobile device, three Edge servers, and a Cloud data center. The mobile device is the start and endpoint of any mobile application execution. Resources are very limited when comparing to the hardware specifications of Edge servers and Cloud data centers. We assume that the hardware specifications of the aforementioned nodes do not change during the runtime. The contemporary CPU processing power of the mobile device is typical between 1.8 - 2.2 GHz but everything above 1 GHz is acceptable [mob]. Cloud data center CPU processing power with modern technological achievements can be boosted with 56 cores with a base frequency of 2.6 GHz and turbo 3.8 GHz [clo]. Using 20 GHz in the simulation is to reflect the computational superiority of the Cloud server compared to other counterparts (mobile and Edge). Values from Table 6.1 are selected as moderate to reflect the magnitude of the computational power ratio between complementary parts of the network. Also, Edge and Cloud servers due to unreliability (server and network failures) must have larger resource capacities to stay competitive.

Concerning the mobile device, we need to consider an energy consumption model. The parameters used in energy model are $p_u = 1.3$W, $p_d = 1.0$W, $p_c = 0.9$W, $p_{idle} = 0.3$W where condition is assumed $p_u > p_d > p_c > p_{idle}$ as in [KL10] and used in Equations (3.7) and (3.12).

Table 6.1: Hardware specifications

Node	**CPU (GHz)**	**RAM (GB)**	**Storage (GB)**
Edge database server	5	8	500
Edge computational server	8	8	250
Edge regular server	5	8	250
Cloud data center	20	128	1000
Mobile device	1	8	16

Table 6.2: Network specifications

Links		**Latency (ms)**	**Bandwidth (Mbps)**
Mobile	Edge	15	5.5/20
Mobile	Cloud	$54 + \varphi(\mu, \sigma)$	20
Edge	Cloud	$15 + \varphi(\mu, \sigma)$	100/987
Edge	Edge	10	100/987

Network Infrastructure

Network parameters that can influence offloading results are network latency and bandwidth. Network latency is the amount of time that takes the data to transmit between two points which is dependable on physical distance. This fits our model since the Cloud data center is geographically more distanced from the end-user which scale up the network latency. The latency on wireless links between the mobile device and Edge servers should be less due to geographical proximity. Bandwidth, on the other hand, is the rate of data transfer between the two points. This fits the DAG mobile application model where task dependency between the tasks is achieved based on input and output data transmission via network links. Overview of network latency and bandwidth distribution is shown in Table 6.2. Latency distribution is similar to work [DMB18], while bandwidth for wireless links (first and the second row of the table) are actual speed limits of IEEE 802.11 wireless standard (802.11a, 802.11b, 802.11g) and for fixed network links (third and fourth row of the table) are Fast Ethernet and 1Gbit Ethernet link standards. Mentioned wireless standards are used since speed is lower when comparing to other IEEE 802.11 wireless standards which could yield to higher price and operational cost. For a fixed network, we selected Ethernet links since we are assuming that Edge servers will be localized near each other. Higher bandwidth values from the Table 6.2 are associated with Edge database server network connection characteristics due to high demand in data transmission.

Network links between Edge servers and the Cloud data center are much faster when compared to wireless links. It is a reasonable assumption since it is well known that data transmission on telecommunication or Internet network is highly demanding, thus larger bandwidth rates should be provided, similar to the assumption in [TL$^+$16]. $\varphi(\mu, \sigma)$ function models Internet latency on Cloud data center due to transmission delay that according to [DP$^+$13] is estimated to be between 100 and 300ms. It is modeled by

employing a Gaussian distribution with mean $\mu = 200$ and standard deviation $\sigma = 33.5$ to obtain the values in the aforementioned range.

Mobile application setup

We use DAG models of mobile applications as described in works [DMB18, DMB19]. These applications are suitable for our use case scenarios since we are more interested in more typical and commercialized applications that will be more probable used by the average user. The DAG structure used for this paper comes from the description of each application in the aforementioned works. We select three applications: (i) *Facebook*, that models the behaviour of posting pictures on Facebook, which represents typical mobile application, (ii) *Facerecognizer*, models the image processing application which recognizes face on the picture, and represents data-intensive application due to large database of face images and (iii) *Chess*, that models the behavior of chess game between the user and AI software and represents computational-intensive mobile application due to large and complex computations for anticipating next game moves.

Denoting mobile application as typical, data or computational-intensive, does not imply that all tasks in the application are of the same intensity. Table 6.3 shows application task sizes in terms of CPU, input and output data size. CI and DI refer to as computational-intensive and data-intensive respectively. Moderate stands for application tasks that do not have emphasized computational or data components. Similar application task distribution is used in work [TL$^+$16]. Task specifications of aforementioned mobile applications that are used in this experiment are listed in Tables 6.4, 6.5 and 6.6.

Table 6.3: Application task specifications

Type	**CPU**	**Input data**	**Output data**
DI	100-200 M cycles	15-20 KB	25-30 KB
CI	550-650 M cycles	4-8 KB	4-8 KB
Moderate	100-200 M cycles	4-8 KB	4-8 KB

Table 6.4: Facebook task specifications

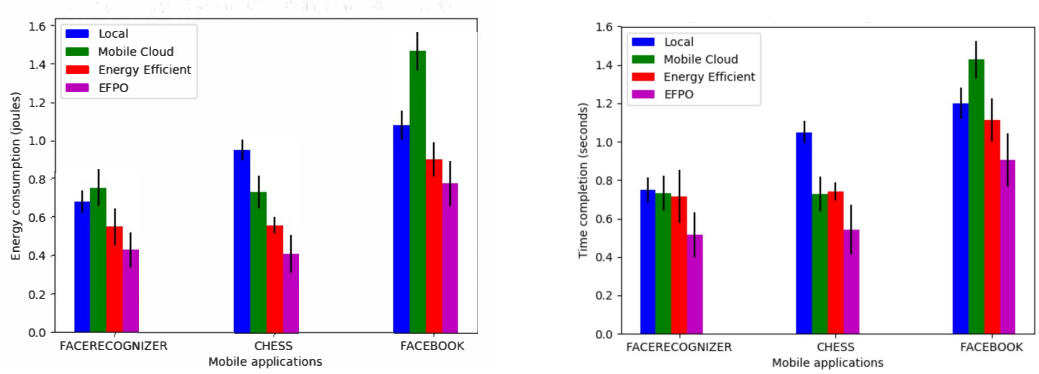
Task	**Type**	**RAM**	**Offloadable**
FACEBOOK_GUI	Moderate	1 GB	False
GET_TOKEN	Moderate	1 GB	True
POST_REQUEST	Moderate	2 GB	True
PROCESS_RESPONSE	Moderate	2 GB	True
FILE_UPLOAD	DI	2 GB	False
APPLY_FILTER	DI	2 GB	True
FACEBOOK_POST	DI	2 GB	False
OUTPUT	Moderate	1 GB	False

Table 6.5: Facercognizer task specifications

Task	Type	RAM	Offloadable
GUI	DI	1 GB	False
FIND_MATCH	DI	1 GB	True
INIT	DI	1 GB	True
DETECT_FACE	DI	1 GB	True
OUTPUT	DI	1 GB	False

Table 6.6: Chess task specifications

Task	Type	RAM	Offloadable
GUI	Moderate	1 GB	No
UPDATE_CHESS	Moderate	1 GB	Yes
COMPUTE_MOVE	CI	2 GB	Yes
OUTPUT	Moderate	1 GB	No



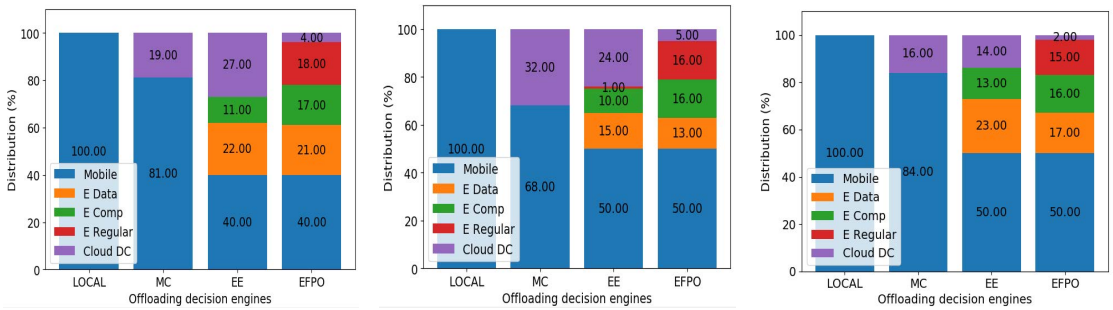
(a) Energy consumption with different applications.

(b) Response time with different applications.

Figure 6.1: Performance metrics

Failure dataset

Failure dataset is vital for MTBF computing for simulating failures through sampling via Poisson distribution (as explained in subsection 3.2.4) and probability estimation of failures that are encoded in the probability matrix of the MDP model. There does not exist an Edge Computing failure dataset that is available for scientific research at present due to the novelty of technology in the field. Consequently and similar to the previous work [AB18], we adopt failure traces from other real-world distributed systems to the edge computing scenario. Our simulation divides and maps real-world failure traces into simulation nodes that have distinctive characteristics as depicted in Figure 3.1. Failure dataset is needed for failure simulation and computing transition probabilities that are used for failure predictability in the EFPO algorithm. Dataset is made publicly

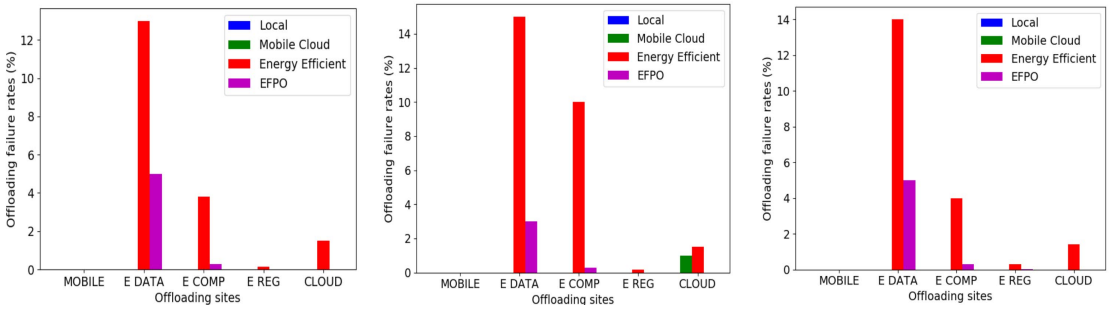


(a) Offloading distribution with Facerecognizer application.

(b) Offloading distribution with Chess application.

(c) Offloading distribution with Facebook application.

Figure 6.2: Offloading distribution



(a) Offloading failure rates with Facerecognizer application.

(b) Offloading failure rates with Chess application.

(c) Offloading failure rates with Facebook application.

Figure 6.3: Offloading failure rates

available by Pacific Northwest National Laboratory (PNNL). Although the PNNL dataset is not collected on an Edge infrastructure, it possesses certain properties that suit our evaluation scenario. The number of computational nodes is large, they are distributed in different geographical locations, and they contain different hardware characteristics.

The dataset contains 4652 failure logs between 2003-2007. Failure logs are collected from the HPC (High-Performance Computing) system that consists of 980 computational nodes. Nodes are classified into several categories according to hardware characteristics. Categories are (i) *fat node*, they are 570 of them, where each contains 430 GB local disk and 10 GB RAM, (ii) *thin node*, 378 nodes, where each contains 10GB local disk and 10 GB RAM, and (iii) *Lustre servers*, they are 34 of them. Every node uses Itanium-2 processor 1.5 GHz and all are interconnected with Quadrics QsNetll.

Simulated failures are applied in the simulation model on all nodes except mobile devices, which is assumed to be failure-free. Before simulating failures, we need to map the failures of node categories from the dataset into the nodes of our simulation model. Fat nodes are suitable for Edge database server due to larger local disk capacity, Lustre servers are considered as Cloud data centers, where due to high performances are used in

Cluster computing, and thin nodes are divided between Edge computational and regular server. A small portion of thin nodes is test and login nodes that have only a few failures. Those node failures are mapped on a regular server while the majority of thin nodes are mapped to a computational server. This setup gives regular servers more reliability than a computational server. With this setting, we want to explore how EFPO performs in a scenario where we have resourceful servers that are less reliable with reliable servers that are less resourceful. Another scenario is where resourceful servers are more reliable, but the EFPO algorithm could have similar performance as other state-of-the-art decision engines since offloading failures are occurring much less on resourceful servers which are more attractive for offloading. Concerning failures, server failures are identified by hardware identifier which is easy to map it on the particular nodes. Network failures, on the other hand, cannot be mapped since they do not contain information on which nodes they are connected to. Thus, network failures are distributed to node categories in proportion to the frequency of failures.

6.1.2 Evaluation Results

Besides the mentioned mobile applications that are used in the experiment, we implemented three additional ODEs to compare performance with EFPO. These are (i) *Local*, which considers only mobile device as an execution site, (ii) *Mobile Cloud* (MC), which considers only mobile device and Cloud data center, and (iii) *Energy Efficient* (EE), which considers offloading application tasks on all offloading sites but without considering failure probability. Considering the reliability, after mapping failures from the PNNL dataset to simulation nodes as explained in 6.1.1, the Edge database and computational server are less reliable then Edge regular and Cloud. The main goal here is to evaluate, whether EFPO boosts the system performance by offloading application tasks on more reliable servers in certain periods by mitigating offloading failures on more resourceful servers. This can extend the execution time but it is still less harmful than offloading failures.

Figures 6.6b and 6.6a show energy consumption and response time per ODE for a single mobile application execution along with the standard deviation. Single mobile application execution is sampled 100,000 times that gives validity and statistical significance to our experimental results. We also consider an experiment, where we have successive mobile application executions, but increasing the number of application executions linearly increases both energy and time. Deviation in application executions is measured and detected but does not change the conclusion of the results. This justifies that sampling a single mobile application execution is sufficient for the evaluation and less time consuming. In both figures, the EFPO algorithm outperforms all other ODE engines in all three mobile application cases. EE engine does not yield better performance since it does not contain failure predictability feature, which is shown in Figures 6.3a, 6.3b and 6.3c where offloading failure rates are the highest. EE engine always prefers those sites that have superior resources capacities without considering failure probabilities. Thus, as shown in Figures 6.2a, 6.2b and 6.2c, EE considers Cloud as most attractive and Edge regular

server as less attractive offloading site. Consequently, this yields bias in task offloading towards those sites which are resource superior but less reliable, which leads to more frequent offloading failures and increased energy consumption as well as response time. Edge regular server, on the other hand, is more attractive for EFPO due to low failure probability and forces offloading distribution to utilize Edge servers more frequently to exploit the advantages of Edge Computing in lower network latency and better network bandwidth. EFPO utilizes Edge servers in 56%, 45% and 48% of task offloading cases with Facerecognizer, Chess and Facebook application respectively. Non-offloadable tasks are executed on the mobile device, where 4%, 5%, and 2% end up in the Cloud data center in Facerecognizer, Chess and Facebook, respectively.

Local ODE outperforms MC ODE in terms of energy, in Facerecognizer and Facebook application cases, while for time response, only in the Facebook case as shown in Figures 6.6b and 6.6a. This is due to high latency between mobile device and Cloud (geographical distance and Internet transmission delay) and the fact that the majority of the application tasks are DI requiring more expensive data transmissions due to larger input and output data sizes. However, in Chess application case, MC ODE outperforms Local since Chess contains CI application task *COMPUTE_MOVE* where Cloud is more suitable due to superior computational capacity and less expensive data transmission for small input and output data size. Cloud data center utilization in Chess application case for MC ODE is 32% (Figure 6.2b), while in Facerecognizer (Figure 6.2a) and Facebook (Figure 6.2c) cases are 19% and 16%, respectively. However, the majority of applications tasks are executed on the mobile device. In Chess case, two out four application tasks are non-offloadable which explains the high distribution of task execution on the device. In the other two application cases, besides a high proportion of non-offloadable tasks, the majority of tasks are more data expensive due to larger input and output data size, which causes the majority of tasks to remain on the device. The mobile device is the site where the majority of application tasks are distributed, from 40% for EE and EFPO ODEs in the Facerecognizer application case up to 100% for all three application cases when Local ODE is performing.

6.2 Edge Offloading for Microservice Architecture Evaluation

6.2.1 Experimental Setup

We evaluate our edge offloading framework on the test-bed described in Table 6.7. Infrastructure setup is summarized in Figure 6.4: Huawei P Smart Z is a mobile device; RPis are edge nodes, deployed as in Figure **??** and AMD64 in Figure 2.4b is used to simulate a cloud data center. Resource heterogeneity is simulated by defining hardware and network limitations, as in [ZAB19]. They are parameterized in the clusters' PostgreSQL database as experimental input parameters. When offloading micro-service is deployed on the Kubernetes cluster, it connects to the database and retrieves the resource information based on which resource capacity of the underlying site is specified. One RPi is configured

Table 6.7: Experimental Setup

Hardware specifications			
NODE TYPE	CPU	RAM [GB]	STORAGE [GB]
Huawei P Z (mob.)	Quad-core ARM Cortex-A53 1.7 GHz	4	64
RPi 3B+ (master)	Quad-core ARMv7 at 1.4GHz	1	64
RPi 3B+ (ED)	Quad-core ARMv7 at 1.4GHz	1	64
RPi 3B+ (EC)	Quad-core ARMv7 at 1.4GHz	1	64
RPi 3B+ (ER)	Quad-core ARMv7 at 1.4GHz	1	64
AMD64 (cloud)	48-core Intel Xeon E5-2650 v4 @ 2.2GHz	128	1000

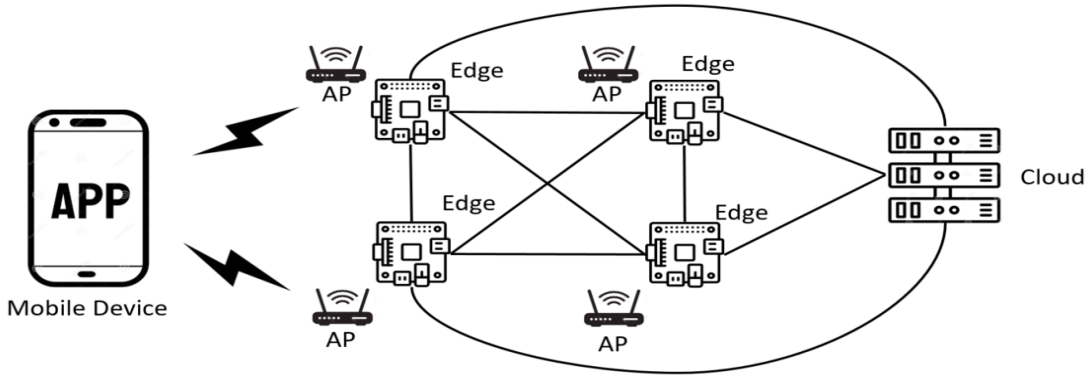


Figure 6.4: Infrastructure Schematics

as a master node and the others as worker nodes in the Kubernetes cluster. Their setup is illustrated in Figure 6.5.

Edge nodes and the cloud server are integrated into a single Kubernetes cluster while a mobile device is implemented as an external user. One of the RPi edge nodes is configured as a master node and the other nodes are configured as worker nodes where offloading micro-services are deployed and implemented as Docker containers. They are deployed according to the node labeling system. Each node in the Kubernetes cluster has a certain label that represents a node type. For instance, if we want to mark a certain node as an edge database server for handling data-intensive applications, the node is labeled as edge database, and inserted into Kubernetes deployment manifest file.

The mobile applications used in the evaluation are Directed Acyclic Graphs (DAGs) taken from [DMB18, ZAB19], namely (i) *Facebook*, as a use case scenario of posting posts on Facebook, (ii) *GPS navigation*, that navigates the traffic drivers to their destination (iii) *Facerecognizer*, as the image processing application which recognizes facial structures and patterns in the images, (iv) *Antivirus*, that scans the software and compares potential software virus signatures with the registered ones in the database, and (v) *Chess.* as an interactive game where AI software agent tries to anticipate player chess moves.

The mobile applications are sampled according to a probability distribution taken from

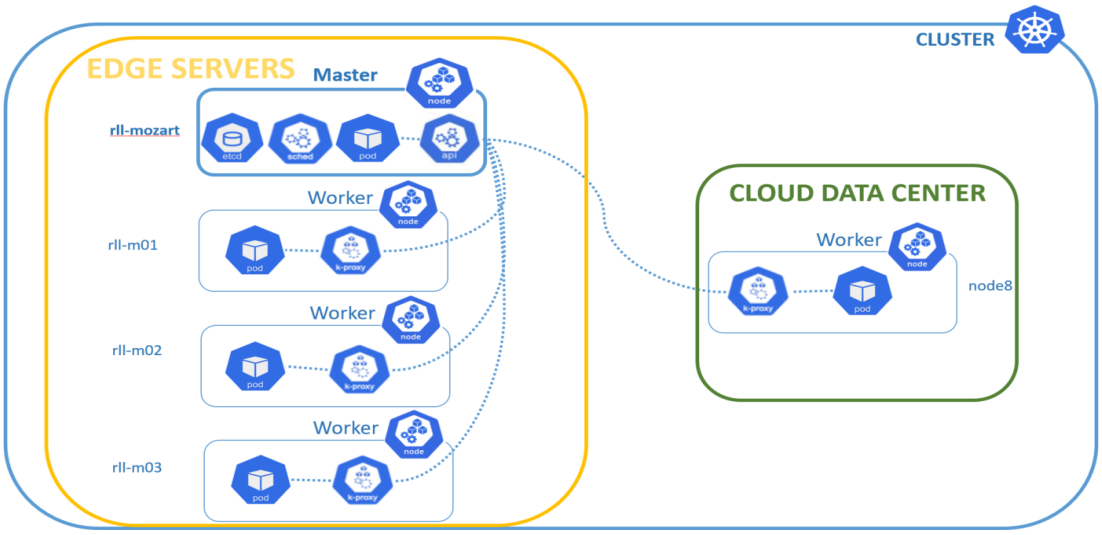


Figure 6.5: Kubernetes Schematics

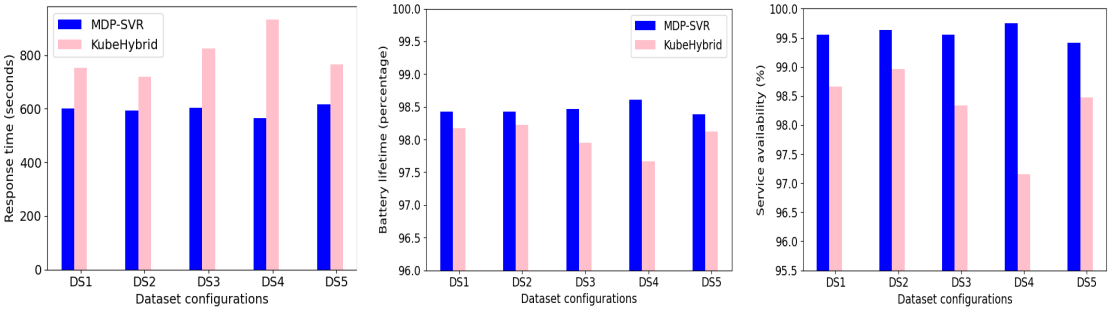
[DMB19]. The simulated workload is utilized since the real application would require application partitioning and profiling mechanisms, which are out of the scope of this paper. To offload the simulated DAG workload on the remote Kubernetes offloading service site, the JSON serialization is performed. It converts task objects into byte strings which are necessary to transfer the data via a network to the target offloading service site. On the recipient site, JSON deserialization is performed to acquire the original task object from which it extracts all necessary resource information.

Hardware and software failures are some of the main threats to the availability of production systems. Software-controlled failure injection during runtime can stress the system when real-world failures are hard to reproduce. This kind of software testing approach often requires custom-developed failure injection software separated from the system under test, especially for distributed systems such as edge computing. This is out of the scope and thus, instead, to simulate failures on remote offloading sites, we implement a two-state Markov state machine. This kind of on/off (failure/non-failure) model is used to simulate network intermittent channels where simplicity is preferred over complexity [BL00]. Although this cannot fully replicate spontaneous failure behaviors from the real-world but at least can aid us to get an insight into edge failure proactive performance.

The probabilistic availability distribution is extracted from the local failure dataset Los Alamos National Laboratory (LANL) for HPC clusters [SG09]. We adopted this dataset since it shares some characteristics with edge computing, i.e., distributed architecture, a large number of nodes, and heterogeneous resources. Possible limitation of using the HPC dataset for the edge is that it probably cannot replicate the edge behavior completely. HPC cluster nodes usually have superior resources, equipped with additional support systems (e.g. fan units, backup power generators) and interlinked with high-speed network

Table 6.8: Dataset configurations

	Dataset configurations				
Service	**DS1**	**DS2**	**DS3**	**DS4**	**DS5**
ED	HA (7_1)	MA (5_158)	HA (5_165)	HA (5_243)	HA (5_48)
EC	LA (19_1)	MA (19_11)	MA (19_4)	HA (19_8)	HA (20_41)
ER	HA (3_0)	HA (16_80)	MA (4_55)	MA (4_1)	HA (4_3)
CD	HA (22_0)	HA (22_0)	HA (22_0)	HA (22_0)	HA (22_0)



(a) Application response time (b) Mobile battery lifetime (c) Service availaiblity

Figure 6.6: System performance

connections where in edge could not be the case. We pick several nodes from the dataset to compute availability distributions for each offloading service (Table 6.8). The nodes are categorized according to their availability levels as low (LA), medium (MA), and high (HA) based on failure rates, and mean and deviation of their availability distribution. Their hardware characteristics are the second selection criteria. For instance, nodes from systems 5 and 7 are selected for the ED edge node due to a large number of nodes (larger data storage). Additionally, a high volatile (HV) node is also present which presents a node that is highly available but exhibits a larger variance due to a few severe failures which are observed as an outlier. The nodes are named *<systemID_nodenumber>* where both index numbers are obtained from the original dataset. They are split into train and test data in a proportion of 80%-20% as the general rule of thumb practiced in ML community. The nodes from systems 5 and 7 are most suitable to the ED edge node due to a large number of nodes (larger data storage). The EC node is sampled from nodes of systems 19 and 20 which have a higher ratio of processors per node (higher computational power). ER edge node is sampled from 3, 4, and 16 systems due to a lower processor per node ratio, a minimum quantity of network interface cards, and a moderate number of nodes compared relatively to the ED and the EC nodes. The cloud is sampled only from 22 system since it has a single node with the highest processor per node ratio and RAM capacity in the entire dataset.

For statistical significance, we set application runs to 1000 and average results of 100 executions. Results are compared with the solution in [ZAB19], which emulates default Kubernetes greedy multi-criteria decision-making (with adjusted parameter tuning) and estimates availability levels through mean-time-between-failures (MTBF). Moreover,

it is augmented with re-computing and check-pointing and named KubeHybrid as a Kubernetes hybrid (proactive-reactive) decision-maker. The source is available online[1].

6.2.2 Results

Figures 6.6a, 6.6b and 6.6c illustrates results for RT, BL and availability respectively. Our solution outperforms the KubeHybrid in all three objectives. There is a strong correlation between the three objectives since higher service availability increases BL and decreases RT. This is explained by the necessity of re-transmitting offloading tasks in case of offloading failures, which consumes additional mobile devices' resources. Hence, higher availability ensures more BL and shorter RT. In our evaluation, we consider also offloading distribution, i.e., the number of tasks offloaded per offloading service site.

Figures 6.6a, 6.6b and 6.6c depict that for DS1 configuration our solution achieves around 600 seconds RT, 98.4% BL, and 99.6% availability against the KubeHybrid with 760 seconds RT, 98.15% BL, 98.6% availability. EC is marked as a LA service while its counterparts on other offloading sites are marked as HA service. According to offloading distribution, our solution offloaded around 50% of tasks to EC, completely avoiding Cloud (0% distribution) while ER receives less than 0.1% distribution. Other tasks are offloaded either on a mobile device or an ED service. Despite lower availability, the prediction engine predicts service availability accurately enough to select EC service for timely task offloading. Moreover, 50% implies that not only CI-intensive tasks are offloaded but also moderate tasks. This is because ER has a lower CPU than EC. The KubeHybrid algorithm, on the other hand, relies on a cloud distribution of 2.9%, while edge services are consumed proportionally to their resource availability. ED is the most used (31.7%), ER is moderately utilized (17.8%) while EC is the least used edge service (7.9%). KubeHybrid depends on an average MTBF, which reduces the prediction accuracy. The SVR algorithm, on the other hand, generally, did yield in our experiment the prediction accuracy between 55% and 90% measured in R2, so-called the good-of-fitness metric. It is widely used in statistics to measure accuracy in predicting future outcomes and usually preferred since it is more intuitive and informative then other metric alternatives.

For DS2 and DS5 configurations, our solution achieves offloading distribution and prediction accuracy similar to DS1, which indicates adaptability towards different availability distributions. However, in DS4 configuration findings are unlike the previously mentioned. Instead of overwhelmingly offloading tasks on EC service, our solution proposal replicates the DS3 availability setting where ED service is tagged HA, while EC and ER services are MA tagged. ED is the most utilized service, with 37% offloading distribution, due to its high availability and resource capabilities. The second most utilized service is EC since it has more hardware capabilities than ER service. In our approach, none of the tasks are offloaded to the Cloud. The KubeHybrid approach, instead, prefers ED service the most (26%) but cloud service is the second most utilized (15%). When ED

[1] https://github.com/jzilic1991/edge-offloading/tree/master

service is unavailable, data intensive tasks are offloaded to the cloud. However, the higher latency results in its worst performance of around 950 seconds RT, 97.5% BL, and 97% availability.

6.3 Evaluation of Fast and Reliable Edge Offloading using Reputation-based Hybrid Smart Contract

6.3.1 Implementation and testbed

Simulated off-chain edge cloud clusters and decision engine are developed in Python and evaluation is performed on a laptop machine with a dual-core CPU of 2.8GHz and 16 GB RAM. The Ganache blockchain emulator and HSC reputation state manager contract are deployed on an AMD64 server with a 40-core 1.80GHz CPU and 128Gb RA. The basic distributed testbed setting reflects our edge offloading lifecycle model presented in **??**. The decision engine connects to the Ganache when reputation needs to be updated and stored during runtime. Ganache executes the blockchain consensus and returns confirmation. The infrastructure is simulated based on the OpenCellID dataset [ope] which contains radio cell tower locations geographically distributed over vast areas. The workload on the nodes is simulated through the queuing network (Section 5.1.1). For SMT solving, we use Z3 as SMT solver [Høf14]. We used the Ganache [2] blockchain emulator as a blockchain and implemented a real-world HSC in the Solidity[3]. Using an emulator instead of a real blockchain is due to the limited number of Ethereum tokens available, which prevents repeated experiments for statistical significance. We assume Proof-of-Authority (PoA) consensus, popular in both private and public Ethereum whose consensus delay is around 4 seconds [etha]. Developers usually use this type of consensus to get easy access and fast feedback. We have open-sourced our prototype publicly[4].

6.3.2 Experimental design and setup

Computing and networking infrastructure

Table 6.9 shows the target infrastructure configuration. It reflects our infrastructure's configurations of different edge, cloud, and mobile devices. We classified servers into several classes to capture resource heterogeneity. The mobile device has limited resources compared to other nodes. The ED is an edge database server that has fast-speed network access and large data storage capacity to handle data-intensive (DI) tasks; the second one represents a computational-intensive server (EC) that has a high number of CPU cores to cope with computational-intensive (CI) tasks, and the third one represents an edge regular server (ER) with moderate resource capacities. The cloud server is the most resourceful one but has higher latency.

[2] https://archive.trufflesuite.com/ganache/
[3] https://soliditylang.org/
[4] https://github.com/jzilic1991/hybrid-edge-blockchain

Table 6.9: Computing infrastructure

Node class	**CPU cores**	**CPU (GHz)**	**RAM (GB)**	**Storage (GB)**
ED server	8	2100	8	300
EC server	16	2800	16	150
ER server	4	1800	8	150
CD server	64	2400	128	1000
Mobile device	2	1800	8	16

Table 6.10: Empirical latency measurements as constraints and deadlines from real-world applications in milliseconds

	Intra(D=108)		**MobiAR(D=400)**		**NaviAR(D=800)**	
∇	Proc	Net	Proc	Net	Proc	Net
Edge	18	15	2-20	15	250-300	300-400
Cloud	2-20	90	1	300	2-20	1000-1500
Mobile	300	0	300	0	800	0

We adopt processing and network latencies as application QoS deadlines from three real-world use cases, described in Table 6.10. In Table 6.10, "Proc" indicates the processing timing constraint, while "Net" is the networking timing constraint. Note that we distinguish task timing constraint ∇ from application deadline D. We measure QoS violations against application deadlines.

Mobile DAG applications

Mobile applications are modeled as DAGs which is a common method of mobile application modeling [ATD21, ZDMAB22]. These applications exhibit a pipeline workflow structure, which is typical for AI-based applications. Table 6.11 specifies task categories from which the applications are constructed, while Tables 6.12, 6.13, and 6.14 describe structures of selected applications. We selected the following applications because they are latency-sensitive, and are part of an emerging market where edge computing is a key technology enabler.

(i) Intrasafed: It is a traffic safety application [LMP+21], which employs an AI-based object detection that detects pedestrians at intersections, notifying drivers in real-time to prevent accidents. We simulated the application in our simulator with latency measurements from the original work, presented in Table 6.10. It has a deadline of $D =$ 108 ms for the average drivers' notification latency via 5G networks. **(ii) MobiAR:** It is a generic AR object detection application [RGW+21], which we extracted its application structure and executed in our simulator. The real latency measurements are extracted from the work and presented in Table 6.10. The application requires a deadline of D

Table 6.11: Task specifications

Type	**CPU**	**Input data**	**Output data**
DI	100-200 M cycles	15-20 KB	25-30 KB
CI	550-650 M cycles	4-8 KB	4-8 KB
Moderate	100-200 M cycles	4-8 KB	4-8 KB

Table 6.12: Intrasafed task specifications

Task	**Type**	**RAM**	**Offloadable**
LOAD_MODEL	Moderate	1 GB	False
UPLOAD	DI	1 GB	True
ANALYZE	CI	4 GB	True
AGGREGATE	CI	2 GB	True
SEND_ALERT	Moderate	1 GB	True

Table 6.13: MobiAR task specifications

Task	**Type**	**RAM**	**Offloadable**
UPLOAD	Moderate	1 GB	False
EXTRACT	CI	2 GB	True
PROCESS	CI	2 GB	True
DATA	DI	1 GB	True
DOWNLOAD	DI	1 GB	False

= 400 ms to meet the applications' inference latency. **(iii) NaviAR:** It is an AR live navigation executed on AR HoloLens glasses [WYS21]. We simulated the structure in our simulator backed by latency measurements as constraints listed in Table 6.10. It requires a deadline of 800 ms which is equal to the local execution time on AR glasses.

Parameters

Parameters used in our experiment are defined in Table 6.15. The Poisson task arrival rate λ range is selected so it can scale to different workload intensities. α, β, and γ values are selected as a representative case of the user's preferences about preferring fast response and willingness to pay a higher price for it ($\alpha > \beta > \gamma$) since we target latency-sensitive applications. BL is the initial device battery capacity. Reputation weight factor ω is taken from [BID21] which accounts for a relatively conservative reputation system to mitigate volatility in a crowdsourced system. *cost* coefficients for CPU and storage are taken from Google Cloud [gcl] which is one of the most commonly used cloud providers. Energy coefficients of β_{base}, β_{U_e} and p_{cores} are taken from [ASV+16, ZLLL17] which are validated against real mobile equipment.

Table 6.14: NaviAR task specifications

Task	**Type**	**RAM**	**Offloadable**
MAP	DI	1 GB	True
GUI	Moderate	1 GB	False
COORDINATION	CI	4 GB	True
SHORTEST_PATH	CI	2 GB	True
MOTION_COMMAND	CI	1 GB	True
VIRTUAL_GUIDANCE	Moderate	1 GB	False
RUNTIME_LOCATION	CI	1 GB	True
DISPLAY	Moderate	1 GB	False

Table 6.15: Simulation and algorithmic parameters

Parameter	**Value**
λ	[10, 20]
α	0.5
β	0.4
γ	0.1
BL	1000
ω	0.3
$cost_{cores}$	0.023
$cost_{stor}$	0.776
β_{base}	625.25 10^{-3}
β_{U_e}	6.9305 10^{-3}
p_{cores}	0.073 10^{-3}

Datasets

We employed the Skype availability dataset[GD05] to model the system's availability. The motivation for selecting Skype dataset is because it shares edge characteristics like geo-distribution, heterogeneity, large number of nodes, and it constitutes the middle ground in availability ratio (60-70%) and latency (up to ∼50 ms) compared to other infrastructure[AB20]. Traces are collected over 2,081 servers for 400 days and contain availability time intervals that are associated with each node. Nodes have different lifespans and hence they are normalized within the $[0, 1]$ time range interval. Adopting such availability datasets from distributed systems that share similar characteristics is common in edge computing research[AB20, SMKP23] due to the lack of publicly available datasets.

Edge and cloud deployment follow cellular base station locations from OpenCellID. OpenCellID is an open cellular database containing datasets of cell tower geolocations that mobile operators publicly publish. It is used in generating infrastructure topologies under edge computing settings [XEMDN21]. We selected a dataset that contains around

3,500 cell tower locations and randomly filtered them out to match the number of 2,081 Skype nodes for one-to-one availability trace mapping. We clustered the entire network into 30 cell clusters using the k-means clustering algorithm as illustrated in Figure 6.7. In such a deployment, location-based mobility is simulated where a mobile device visits each cell cluster and offloads tasks on remote servers. Mobile device dwelling time in each cell is evenly distributed throughout the entire simulation. Each cell cluster location has edge node classes such as ER, ED, and EC which are randomly associated with OpenCellID nodes and a single accessible cloud server. Remote servers have an associated reputation score, which is stored on a public blockchain that is globally accessible.

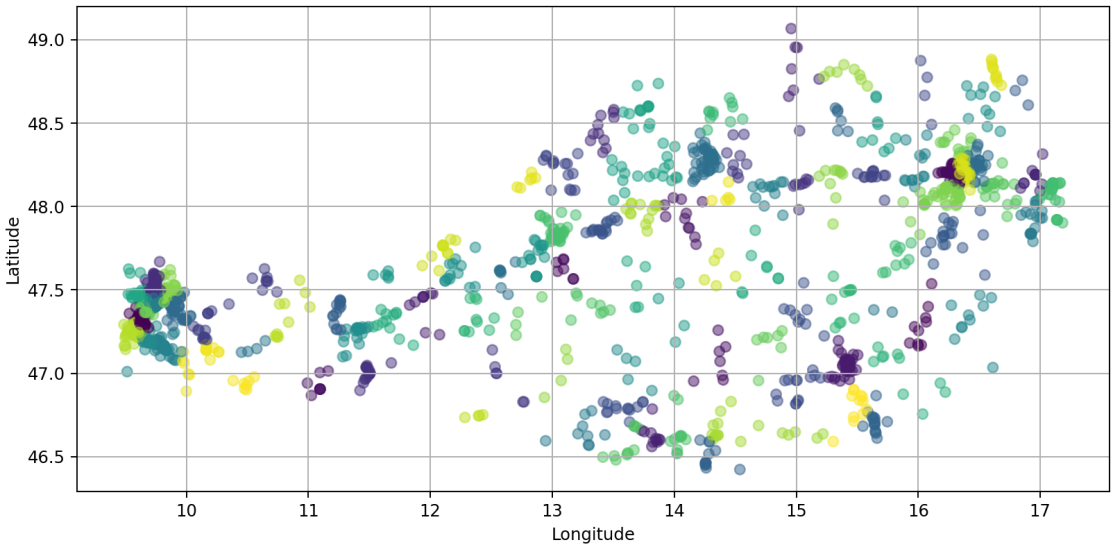
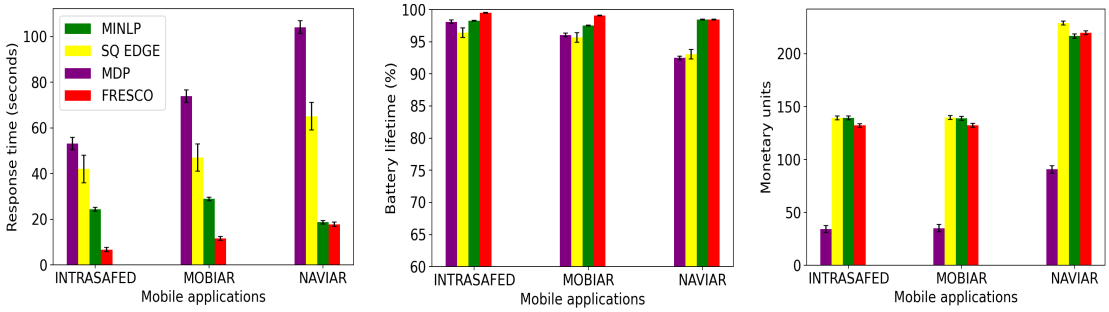


Figure 6.7: Cell tower locations from OpenCellID dataset [ope]

Baselines

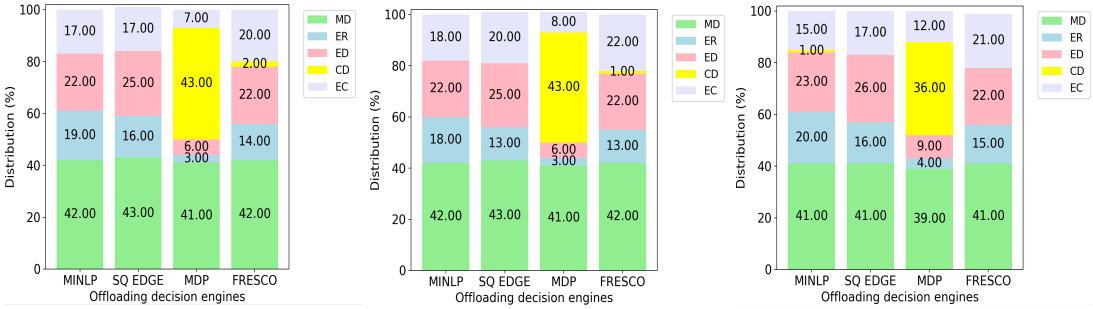
We compare *FRESCO* with the following three baseline algorithms.

- ***MINLP*** is a mixed integer non-linear programming-based method that formulates constraint offloading optimization problems without reputation. The MINLP approach is the most common modeling method for offloading optimization [FHZ+22].
- ***SQ EDGE*** [IMRN20] considers reputation and queuing time on edge nodes, and it is utilized in blockchain-based vehicular ad-hoc networks. The method considers only local and edge offloading, as in naive offloading approaches used when resources are limited for decision-making.
- ***MDP*** is a common method for modeling offloading [LZC+20]. Reputation is encoded as transition probability, remote servers represent states, and objectives are modeled as reward functions. The modeling is similar to existing work that targets reliable offloading[ZDMAB22].



(a) Response time (b) Battery lifetime (c) Monetary cost

Figure 6.8: Performance main objectives



(a) Intrasafed (b) MobiAR (c) NaviAR

Figure 6.9: Offloading distribution

6.3.3 Analysis of results

For each experimental run, we execute 100 applications sequentially and average results over 100 runs to obtain statistically significant results.

Response time

Figure 6.8a illustrates the response time performance of offloading decision engines in Intrasafed, MobiAR, and NaviAR applications respectively. The worst-performing decision engine is MDP whose response time is 53.11, 73.86, and 104.06 seconds for three applications, respectively. Whereas the SQ EDGE decision engines have average response times of 41.99, 46.96, and 65.12 seconds. However, SQ Edge has a higher deviation in its response time compared to others (6.16, 4.03, and 5.99 seconds). Although the SQ EDGE approach is reputation-aware, its primary target is to identify malicious servers instead of reliable offloading in terms of QoS violations caused by failed or high-loaded sites. Thus, this leads to more volatile performance as observed. The MINLP decision engine yielded the second-best approach with 24.34, 28.91, and 18.72 seconds. The best performance was achieved with the NaviAR application (18.72 seconds) which is unexpected since NaviAR has the most complex structure. The possible explanation is that the edge servers in the

last visited cells were more loaded which limits resource capacity. It could deter MINLP from taking offloading decisions on the edge and rather opt for local execution or select a far-distant cloud. 75% of the offloading attempts in the last cell were concentrated on cloud and mobile. Although MINLP does not perceive reputation, selecting both mobile devices and the cloud are safe for offloading and avoids offloading failures on the failure-prone edge which in the last two cells have limited availability ($12 - 25\%$). Offloading failure would impose a longer response time as seen in Intrasafed and MobiAR applications. Lastly, our FRESCO solution outperforms other decision engines due to frequent offloading on more reliable servers which resulted in response time performance of 6.75, 11.61, and 17.81 seconds.

Battery lifetime

Figure 6.8b illustrates the battery performance of offloading decision engines in all three applications. The SQ EDGE decision engine drains the device battery the most, with 96.38%, 95.33%, and 93.34%. A higher rate of failed offloading attempts drains the energy more than the longer response time (Figure 6.8a) in MDP whose battery lifetimes are 98.08%, 95.64%, and 93.03%. MINLP and FRESCO, on the other hand, have battery lifetimes that reflect response time performance from Figure 6.8a. MINLP battery lifetimes are 98.28%, 97.54%, and 98.42% while FRESCO has the highest battery lifetime of 99.47%, 99.06%, and 98.43%.

Resource utilization cost

Figure 6.8c shows the resource utilization cost of all four approaches. Here. MDP incurs lower resource utilization costs (ranging from 34.4 to 90.69 monetary units) although it has poor response time (see Figure 6.8a). This is because it offloads to a highly available cloud node which has a lower utilization cost. SQ EDGE is the most expensive solution due to failed offloading attempts and fixed k parameter which does not scale with several nodes, and thus limits alternative servers for offloading consideration and can lead to potentially higher costs. According to SQ EDGE, best k sites are the most reputable ones but can be highly loaded and limit re-offloading alternatives in case of failed or untimely execution. SQ EDGE monetary costs are 139.3, 139.16, and 228.23 units for three applications. When comparing MINLP and FRESCO, FRESCO emerged as the second-best cost-effective solution and cheaper than MINLP in Intrasafed (132.22 vs 139.33 units) and MobiAR (132.38 vs 139.08 units) use cases but worse when offloading NaviAR (219.89 vs 216.76 units). In NaviAR's case, MINLP is slightly cheaper than FRESCO because it offloaded a minor portion of tasks on the cloud which is cheaper than Edge. FRESCO did not yield the overall best cost-effectiveness because hyperparameters are tuned so it prefers faster and energy-efficient solutions rather than low-cost sites.

Offloading distribution

Figure 6.9a) shows the offloading distribution analysis for three applications. This analysis shows the distribution of application tasks to different nodes in our infrastructure. In the Intrasafed use case, the MDP was worse-performant because most of the offloading decisions were targeting highly available cloud (43%) instead of expensive edge servers and thus is the cheapest solution (Figure 6.8c) and not necessarily the least energy-efficient (Figure 6.8b). SQ EDGE, with a similar offloading distribution composition to MINLP, only considers k sites with the highest reputation and selects the shortest queue waiting time. k parameter is fixed and does not scale with several nodes which can limit the number of alternative servers and exclude viable ones that are less loaded and sufficiently reliable. MINLP, on the other hand, does not restrict its offloading options. FRESCO, comparably to the previous two aforementioned baselines, is more flexible and utilizes all site types, including cloud (2%) for CI tasks when edge servers are less reliable, also less reliant on moderate ER sites (14% compared to 19% and 16% in MINLP and SQ EDGE respectively), and utilizes resource-rich EC sites more frequently (20% compared to 17% in MINLP and SQ EDGE). FRESCO's offloading distribution composition is similar in the MobiAR case (Figure 6.9b) and reflects FRESCO's higher performance in both applications. In the NaviAR case (Figure 6.9c), MINLP and FRESCO have different offloading distribution compositions but performance-wise are comparable (Figure 6.8a). FRESCO balances reliability and performance, where the most reputable servers are not necessarily efficient ones. MINLP is reputation-oblivious but selecting the most efficient servers can be sometimes beneficial if the underlying infrastructure is more reliable and has fewer failures or less volatile load.

QoS violations

Figure 6.10 illustrates QoS violation results. MDP has the highest violation rate because of frequent offloading on highly available clouds. Leading to fewer failures but frequent violations. The next better performant solution is SQ EDGE, with a violation rate between 18.9% (in the NaviAR case) and 15.9% (in the MobiAR case). MINLP shows better performance with violation rates of 12.3%, 9.2%, and 0.1% in Intrasafed, MobiAR, and NaviAR respectively. FRESCO has the lowest violation rates in Intrasafed and MobiAR use cases with 7.1% and 3.8% and has a violation rate of 0.4% with a standard deviation of 0.48% in the NaviAR case which is comparable with MINLP.

HSC overhead

HSC blockchain usage costs are expressed as gas consumption and are called Wei. The results are presented in Table 6.16 for each function. Where range is expressed, it refers to executing from 1 to 30 offloading transactions, as multiple offloading transactions are typically executed for those functions. All HSC functions consume slightly above $21,000$ Wei, which is typical on the Ethereum [ethb].

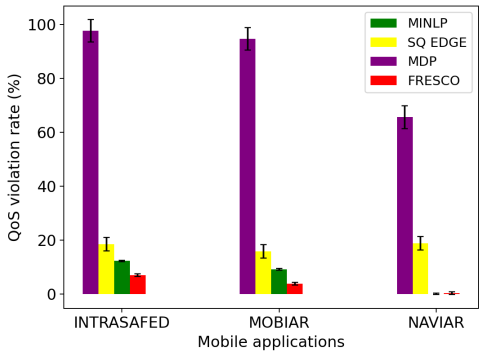


Figure 6.10: QoS violations

Table 6.16: Hybrid smart contract usage cost on Ethereum

Function name	**Gas consumption (Wei)**
registerNode	21,503 Wei
unregisterNode	21,204 Wei
getNodeCount	21,604 Wei
getNode	21,204 Wei
updateNodeReputation	21,638-29,984 Wei
getReputationScore	21,204 Wei
resetReputation	21,484-25,544 Wei

Offloading decision overhead

Figure 6.11 illustrates offloading decision time overhead across different infrastructure sizes on a logarithmic scale. The SQ EDGE is the least complex algorithm since selecting the first k nodes and computing their estimated queue waiting time is relatively straightforward in comparison to other decision engines. The average decision time overhead is 0.048 milliseconds. FRESCO and MINLP decision time overhead are 5.05 milliseconds and 6.57 milliseconds with standard deviations of 5.16 milliseconds and 3.07 milliseconds respectively, making them comparable. MDP has the highest overhead, which is an average of 1373.83 milliseconds, due to state space explosion when offloading on a larger number of nodes. In summary, FRESCO has an average decision time overhead of 5.05 milliseconds, which makes it suitable for offloading latency-sensitive applications.

Summary: FRESCO decreases average response time up to 7.86x, and increases battery up to 5.4% compared to baselines. It also achieves a low deadline violation rate of 0.4% while maintaining competitive utilization costs. With approximately typical blockchain consumption ($\approx 21,500$ Wei) and low average decision time overhead (5.05 milliseconds), FRESCO is suitable for offloading latency-sensitive applications.

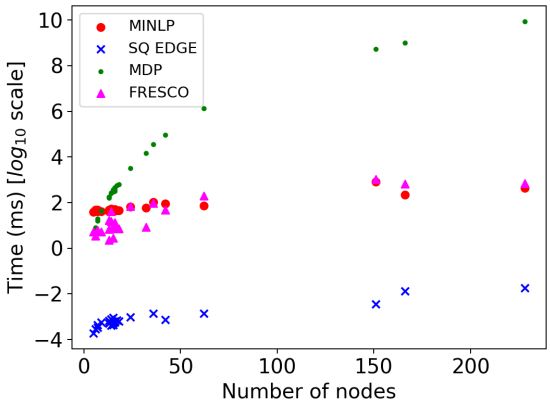


Figure 6.11: Offloading decision time in logarithmic scale

CHAPTER 7

Related Work

In this chapter, we provide a structured overview of the state-of-the-art in edge offloading, focusing on key challenges and advancements in heterogeneous and resource-constrained environments. We analyze existing approaches and their limitations across five main aspects:

- Energy-efficient and failure-aware edge offloading
- Microservice-based edge offloading
- Blockchain-enhanced reliability and offloading mechanisms

Through this analysis, we highlight critical gaps in the literature and discuss emerging solutions that improve efficiency, fault tolerance, and decision-making in edge computing.

7.1 Energy-efficient and failure-aware edge offloading

Offloading was considered as a suitable solution for tackling energy efficiency and application response time issues as summarized in [Wu18a, AGH18] for MCC infrastructure. Most of those works introduced computation offloading frameworks and multi-objective decision-making algorithms. Similarly, in survey work [MB17] for MEC, a lot of literature work about offloading frameworks and architectures are systemized to overcome the offloading limitations where offloading decision-making, computation resource allocation, and mobility management are addressed as key areas. Currently, some researchers in the Edge Computing area are coping with offloading challenges through multi-objective optimization algorithms as [DMB18, DMB19] inspired by offloading frameworks in MCC as [C$^+$11, KA$^+$12, C$^+$10] where energy consumption, application run time and/or monetary costs are considered as primary objectives. None of these works considers the effect of offloading failures on systems' performance.

Table 7.1: Overview of state-of-the-art literature for microservices and offloading

Publication	MSA	OFF	PRO	ORCH	REAL
Suk et al. [SHBZ19]			✓	✓	✓
Aral et al. [AB18]			✓		
Zilic et al. [ZAB19]		✓	✓		
Dupont et al. [DGC17]		✓		✓	✓
Tang et al. [TYLG20]	✓	✓		✓	
Samanta et al. [SLE19]	✓			✓	✓
Wu et al. [WTAK20]	✓			✓	✓
Jimenez et al. [JS19]	✓			✓	✓
Samanta et al. [ST20]	✓			✓	
This work	✓	✓	✓	✓	✓

Research works about fault-tolerant offloading systems that exist for mobile wireless environments such as [Wu18b, OWYZ08] using M/M/1 queue model and checkpointing mechanism respectively. Work as [WWW13] performs a trade-off between local re-execution and offloading on remote infrastructure in case of offloading failure using the timeout mechanism, while work [SPJ17] considers recovery mechanism by finding alternative paths via ad-hoc relay nodes through Fyold-Warshall algorithm in case of offloading failure occurrence on the shortest path. None of this works provides formal verification of performance and reliability for Edge Computing. A desirable solution for achieving both goals is formal verification. Work [ZNW15] was using the MDP algorithm for obtaining optimal offloading decision policy in a wireless mobile environment where uncertainty in wireless connections and user mobility can cause offloading failures. This work was adopted for Cloudlet systems. There exist works [TL+16] and [AGA18] for MCC and Edge Computing, which use the MDP algorithm to obtain optimal offloading decision policy but without considering offloading failures. Also, the MDP reward optimization technique is used for Edge/Cloud offloading but in the context of data stream analytics [dSV+19].

7.2 Microservice-based edge offloading

Mostly reactive failure management techniques has been discussed in the related edge computing literature thus far. The authors in [IGAK+15] perform container checkpointing at the edge to ensure high service availability while [OWYZ08] checkpoints the applications offloaded on the offloading sites. Another work [WWW13] locally re-computes offloaded tasks on a mobile device when task offloading fails. Research conducted both in simulated [DMB18, DMB19, HXW+20] and real-world edge environment [TYLG20] do not consider proactive failure mitigation. Failure prediction approaches such as [MAUY19, dCMZLD11] proved the effectiveness of proactive failure management, but these approaches are neither applied at the edge nor on a real-world test-bed.

There exists few studies focusing on proactive failure management. They propose risk based [SHBZ19], learning based [AB18, AB20], or formal verification based [ZAB19] solutions. Nevertheless, none of these consider microservices. We summarise our literature review in Table 7.2. The works are selected according to whether they focus on mi-

Table 7.2: Overview of state-of-the-art literature for blockchain-based reputation and offloading

Publication	OFF	(H)SC	BLOCK(-REP)	REL
[RGW+21, ZEMR23, ZWSZ21]	✓	✗	✗	✗
[INMR21, IMRN20, MYZ+24, SDS+23, ZLJZ21]	✓	✗	✓	✗
[ZDMAB22, LMFH23, LL23]	✓	✗	✗	✓
[MJSS+18, SWS21]	✗	✓	✗	✗
[SYCC21, YLG+20, KXL+21, ZWY+21]	✗	✗	✓	✓
[DCZ+20]	✓	✗	✓	✗
FRESCO	✓	✓	✓	✓

croservice architecture (MSA), edge offloading (OFF), proactive failure prediction (PRO), container orchestration (ORCH), and real-world implementation (REAL). We conclude that to the best of our knowledge, none of the selected works covers all aforementioned objectives.

7.3 Blockchain-enhanced reliability and offloading mechanisms

Table 7.2 compares FRESCO with the literature concerning offloading OFF, smart contracts $(H)SC$, blockchain-enabled or based-reputation $BLOCK(-REP)$, and reliability REL.

Offloading survey [ZEMR23] shows that applying deep reinforcement learning solutions (DRL) and their variants has become common recently due to their adaptability to changing and dynamic conditions. They do not consider edge failures and are deployed as an intermediary between devices and servers which is risky in the unreliable environment that can lead to a single-point-of-failure, sometimes requiring model re-training when topology or environment changes drastically[ZWSZ21]. Also in the experimental evaluations, they are usually trained on limited infrastructure, probably due to convergence issues requiring extensive experience.

Deep reinforcement learning solutions (DRL) are common in edge offloading areas[ZEMR23] due to their adaptability to changing and dynamic conditions. However, we did not opt for such a solution because in most cases, the deep RL is deployed as an intermediary between devices and servers which is risky in the unreliable environment that can lead to single-point-of-failure. Also, model re-training is required when topology or environment changes drastically[ZWSZ21] which is common in our scenario with failure-prone infrastructure and devices moving between different cells. Distributed multi-agent DRL offloading solutions can mitigate the single-point-of-failure issue and adapt to local changes. However, they are usually trained on limited infrastructure due to convergence issues requiring large experience and do not consider reliability problems[ZYP+22, YYC+23, HLMZ21].

Work [MUB19] employs an SMT solver for computing offloading by respecting QoS constraints. Also, an adaptive offloading framework (ACTOR) based on artificial neural

networks was developed in [RGW+21] that offloads latency-sensitive applications such as AR that we used in our evaluation. None of the aforementioned works are considered for offloading reliability problems.

Blockchain-enabled edge offloading approaches [MYZ+24, SDS+23, ZLJZ21] enhance reliability and efficiency in mobile edge computing and vehicular edge networks where blockchain and smart contracts make offloading more secure and trustworthy. They do not provide specifics about real-world smart contract implementation or they deploy an offloading framework as a smart contract which is unrealistic since smart contracts prohibit nondeterministic and stochastic computations that conflict with the determinism requirement of blockchain consensus protocols. Blockchain-based reputation offloading was proposed for specific scenarios that require fast responses like vehicular networks[IMRN20] and IIoT[INMR21]. Solutions are applied in private environments (e.g. factories, enterprises) while ignoring the consensus overhead, and using reputation against malicious actors rather than against failures in unreliable environments. Also, other blockchain-based reputation works are applied for selecting reliable edge servers, ranging from IoT [YLG+20], federated edge learning [KXL+21] to vehicular networks [SYCC21]. However, they did not target reliability in edge offloading in terms of edge failures and do not employ HSC, which forces them to compute reputation off-chain rather than on-chain which can lead to potential risks (e.g. collusion) [DLWZ20].

Some edge offloading approaches [PZBX22, LMFH23, LL23] tried to solve reliability issues in terms of failure and recovery probabilistic models, or predict edge failures based on historical data [ZDMAB22]. The aforementioned works did not prove or evaluate their solutions in distributed unreliable edge scenarios where the device moves and reacts to different environments.

Works [MJSS+18, SWS21] implemented smart contracts on hybrid blockchain architecture to reconcile conflicting objectives such as trust on one side and performance on the other side. The works are not applied in edge offloading context.

Works like [DCZ+20, DSRS19] applied incentive-based offloading in e-commerce and service provisioning in IoT environments based on blockchain-based reputations. Both solutions do not account for reliability issues and neglect the consensus delay impact on decision-making, which makes it not suitable for (near-)real-time environments.

In conclusion, none of the works applied a blockchain-reputation system for enhancing reliability in the edge offloading context. FRESCO uniquely ensures trust for sensitive reputation information on-chain and allows fast performance for latency-sensitive applications off-chain.

CHAPTER 8

Conclusion

8.1 Summary

Edge offloading has been positioned as a key technique for enabling efficient and reliable real-time execution of resource-intensive mobile applications in distributed and unreliable resource-limited edge environments. The core focus of this thesis was the investigation and exploration of the reliability impact on edge offloading by addressing research challenges like heterogeneous and limited edge resources, volatile workloads, and failures and reliability assessment of edge servers. Many new emerged latency-sensitive mobile applications relies on offloading due to its adaptive and failure-aware capabilities to cope against dynamic environmental conditions such as mobile augmented and virtual reality, traffic safety, and facial image processing.

This research study has demonstrated through development and evaluation of proposed edge offloading frameworks how adaptive and failure-aware offloading decision policies can contribute to fast and reliable system performance, which is needed for latency-sensitive mobile applications. Proposed edge offloading solutions integrated formal methods, machine learning,g and decentralized analytical approaches to derive dynamic offloading policies for assessing reliability levels of edge and cloud servers with the purpose of predicting and mitigating failrues over time. The evaluation results emphasized the role of ensuring that unreliable edge servers were less likely to be selected for offloading and task execution, which can be prolonged or postponed due to occurred failures.

In this study, we evaluated methodologies used both in simulation and real-world settings and empirically validated theoretical offloading findings practically. Evaluation results have shown that our proposed offloading solutions did minimize application response time and failures rates, prolong battery lifetime and optimize monetary resource utilization costs compared to cloud-based solutions and baselines methodologies taken from the literature. These critical findings should incentivize further research to find other

offloading alternatives that balances performance with other objectives like trustworthiness and credibility, to make offloading more reliable and resilient.

8.2 Limitations

Although this study have shown promising preliminary research results, the proposed solutions still do not eliminate uncertainties completely. Monitored resource data and performance metrics can have lower quality due to noise and error-prone measurements. Also, during runtime in large-scale systems, it is almost impossible to have a complete information corpus about the infrastructure state. This compels offloading decision-making to be performed upon partial and limited data. The robustness of offloading policies and reliability predictions is critical in such dynamic and large-scale edge systems.

Additional limitations are used methodologies themselves including heuristics, formal methods, and machine learning models. Heuristic methods such as reputation systems can be limiting in tracking historical performance depending on weight factors, which can be biased in more diverse application and infrastructure setups. Formal methods maybe can give rigorous and guaranteed solutions, but lack flexibility and adaptivity for dynamic conditions and circumstances. Machine learning predictive models need continuous re-update and learning, and rely heavily on data quality, which can be compromised depending on the reliability of monitoring instrumentation.

The system architectures that we relied on in this study followed a three-tier mobile-edge-cloud architectural pattern. Other architectural patterns were not employed for evaluation purposes, such as mobile-edge without cloud connectivity, or device-to-device offloading, which is inevitable in infrastructureless or minimum infrastructure environments like vehicular networks. Also, extreme failure cases were not examined like natural disasters, where availability zones or regional locations are unavailable, to test the robustness and resilience. In our experiments we used real datasets, which do not contain such high-stress situations since they are rare and hardly reproducible.

8.3 Future Work

Based on theoretical findings, evaluation results, and identified limitations, we propose several research directions and venues to investigate and explore for further development of efficient, reliable, failure-aware, and adaptive edge offloading. First venue could be decentralized offloading decision-making in a distributed unreliable edge. Multi-agent deep reinforcement learning and stochastic game theory could be viable methods to copre with failures uncertainty in distributed settings. They can dynamically adapt to changeable environmental parameters and mutually coordinate offloading decisions to output optimal offloading strategies that should lead to welfare benefit of all effected offloading actors.

Second venue for future work considerations are lightweight predictive machine learning models. These kinds of solutions are deployed directly on resource-limited devices without imposing high resource consumption, and can dynamically predict future events and trends for more informed decision-making. Methods like personalized federated learning could be explored for predicting performance or reliability levels over time. The approach could improve real-time and adaptive decision-making without sacrificing mobile resources and potentially nullify the benefits of offloading.

A third venue for research considerations is further exploration of formal methods and their hybrid integration with machine learning models. The goal would be to combine formal methods' mathematical rigour with machine learnings' stochastic predictions to enable consistent and predictable high-performance, especially for latency-sensitive mobile applications that require high-reliability guarantees. Formal methods through formal verification and validation techniques can guarantee the feasibility and correctness of offloading decisions, while machine learning can assign probability levels to offloading decisions or locations about their performance impact or reliability assessment.

And the last venue would be more empirical and industrial-oriented edge offloading solutions, which are deployed in live production environments. The goal would be to expand the scalability of edge offloading solutions and identify in which concrete use cases would be most beneficial to deploy the offloading solution. Also, necessary software artifacts for development, deployment and runtime execution of edge offloading frameworks should be identified, including tools, frameworks, and libraries. It should be considered how to integrate offloading software development lifecycle into existing legacy systems and asses state-of-the-art technology stacks' adequacy. Otherwise, novel custome software artifacts should be invented to support edge offloading development lifecycles.

Overview of Generative AI Tools Used

GrammarlyGO and ChatGT-4o tools were used as an aid in the writing of this thesis. Both tools were used with intent to improve clarity, readability, and unambiguity of the thesis content. They improved text quality without changing the inherent semantics of original text version.

The GrammarlyGO tool was utilized for correcting grammatical mistakes, spelling errors, and sentence re-phrasing. The tool made writing clearer and easier to read. Identified word or sentence errors were highlighted together with text suggestions without generating new synthetic text. The accepted suggested improvements were only localized at the sentence level without altering the original meaning of the text.

ChatGPT-4o was mostly used for brainstorming around the structure and organization of the thesis, generating key points and concepts as a basis for storyline creation and content writing, and getting feedback about the text quality. It aided in text clarification and adjusting writing style for simplifying complex concepts, but without losing technical soundness.

The text generated by the AI tools was refined thoroughly without compromising the author's original intent and meaning. The final content version contains the original author's contributions by respecting academic ethical conduct.

List of Figures

List of Tables

List of Algorithms

Glossary

Artificial Intelligence (AI) The simulation of human intelligence in machines that can learn, reason, and solve problems, often used in predictive offloading and reliability modeling.

Blockchain A decentralized digital ledger technology used for secure transactions and trust management in distributed computing.

Central Processing Unit (CPU) The primary component of a computer that performs most of the processing inside a system.

Cloud Computing The delivery of computing services over the internet, allowing on-demand access to computing resources.

Edge Computing A distributed computing paradigm that brings computation and data storage closer to the location where it is needed to improve response times and save bandwidth.

Energy Efficiency A key factor in mobile and edge computing that optimizes resource consumption while maintaining performance.

Failure Prediction A technique that anticipates potential system failures in order to improve reliability and performance.

FRESCO A fast and reliable edge offloading framework that ensures optimized task distribution in edge computing environments.

Gigabyte (GB) A unit of digital information storage equal to 1,024 megabytes (MB).

IEEE Institute of Electrical and Electronics Engineers, an organization that sets standards for computing and telecommunications.

Kubernetes (Kub) An open-source container orchestration system for automating application deployment, scaling, and management.

Latency The delay before a data packet is transmitted and received, critical in real-time computing applications.

Markov Decision Process (MDP) A mathematical framework for modeling decision-making where outcomes are partly random and partly under the control of a decision-maker.

Markov Model A stochastic model that represents systems where future states depend only on the current state, commonly used in failure prediction.

Mobile Computing A technology that allows computation to be performed on mobile devices while offloading tasks to edge or cloud resources.

Quality of Service (QoS) A measure of performance guarantees such as latency, bandwidth, and availability in computing networks.

Reliability The ability of an edge computing system to perform consistently and avoid failures.

Reputation System A mechanism that evaluates the reliability of edge computing nodes based on historical data and trust scores.

Satisfiability Modulo Theory (SMT) A decision problem for logical formulas with respect to combinations of background theories, used in formal verification and constraint solving.

Task Offloading The process of transferring computational tasks from a mobile device to edge or cloud servers to optimize performance and energy consumption.

Bibliography

[AB18] Atakan Aral and Ivona Brandic. Dependency mining for service resilience at the edge. In *IEEE/ACM Symposium on Edge Computing (SEC)*, pages 228–242, 2018.

[AB20] Atakan Aral and Ivona Brandić. Learning spatiotemporal failure dependencies for resilient edge computing services. *IEEE Transactions on Parallel and Distributed Systems*, 32(7):1578–1590, 2020.

[Ada21] AdamTheAutomator. Kubernetes architecture diagram : Fits components together, 2021. Accessed: 13-Feb-2025.

[AGA18] Khalid R Alasmari, Robert C Green, and Mansoor Alam. Mobile edge offloading using mdp. In *Int'l. Conf. on Edge Computing*, pages 80–90. Springer, 2018.

[AGH18] Khadija Akherfi, Micheal Gerndt, and Hamid Harroud. Mobile cloud computing for computation offloading. *Applied Comp. and Inf.*, 14(1):1–16, 2018.

[AO18] Atakan Aral and Tolga Ovatman. A decentralized replica placement algorithm for edge computing. *IEEE Transactions on Network and Service Management*, 15:516–529, 2018.

[ASV+16] Farhan Azmat Ali, Pieter Simoens, Tim Verbelen, Piet Demeester, and Bart Dhoedt. Mobile device power models for energy efficient dynamic offloading at runtime. *Journal of Systems and Software*, 113:173–187, 2016.

[ATD21] Cosmin Avasalcai, Christos Tsigkanos, and Schahram Dustdar. Resource management for latency-sensitive iot applications with satisfiability. *IEEE Transactions on Services Computing*, 15(5):2982–2993, 2021.

[B+06] Gerd Behrmann et al. A tutorial on UPPAAL 4.0. Technical report, Department of computer science, Aalborg university, 2006.

[BID21] Ammar Battah, Youssef Iraqi, and Ernesto Damiani. Blockchain-based reputation systems: Implementation challenges and mitigation. *Electronics*, 10(3):289, 2021.

[BL00] Fulvio Babich and Giancarlo Lombardi. A markov model for the mobile propagation channel. *IEEE Transactions on Vehicular Technology*, 49(1):63–73, 2000.

[Bra08] Maury Bramson. *Stability of queueing networks*. Springer, 2008.

[C+10] Eduardo Cuervo et al. Maui: making smartphones last longer with code offload. In *Int'l. Conf. on Mobile Systems, Applications, and Services*, pages 49–62. ACM, 2010.

[C+11] Byung-Gon Chun et al. Clonecloud: elastic execution between mobile device and cloud. In *ACM Conference on Computer systems*, pages 301–314, 2011.

[CC+14] Iadine Chadès, Guillaume Chapron, et al. Mdptoolbox: a multi-platform toolbox to solve stoch. dyn. prog. problems. *Ecography*, 37(9):916–920, 2014.

[clo] Intel's new assault on the data center: 56-core xeons, 10nm fpgas, 100gig ethernet. `https://arstechnica.com/gadgets/2019/04/intels-new-assault-on-the-data-center-56-core-xeons-10nm-fpgas-` Accessed: 2019-09-05.

[CM04] Vladimir Cherkassky and Yunqian Ma. Practical selection of svm parameters and noise estimation for svm regression. *Neural networks*, 17(1):113–126, 2004.

[CZTL16] Zhuo Cheng, Haitao Zhang, Yasuo Tan, and Yuto Lim. Smt-based scheduling for multiprocessor real-time systems. In *2016 IEEE/ACIS 15th International Conference on Computer and Information Science (ICIS)*, pages 1–7. IEEE, 2016.

[DCH+23] Daniel Mawunyo Doe, Dawei Chen, Kyungtae Han, Yanpeng Dai, Jiang Xie, and Zhu Han. Real-time search-driven content delivery in vehicular networks for ar/vr-enabled autonomous vehicles. *2023 IEEE/CIC International Conference on Communications in China (ICCC)*, pages 1–6, 2023.

[dCMZLD11] Márcio das Chagas Moura, Enrico Zio, Isis Didier Lins, and Enrique Droguett. Failure and reliability prediction by support vector machines regression of time series data. *Reliability Engineering & System Safety*, 96(11):1527–1534, 2011.

[DCZ+20] Shuiguang Deng, Guanjie Cheng, Hailiang Zhao, Honghao Gao, and Jianwei Yin. Incentive-driven computation offloading in blockchain-enabled e-commerce. *ACM Transactions on Internet Technology (TOIT)*, 21(1):1–19, 2020.

[DGC17] Corentin Dupont, Raffaele Giaffreda, and Luca Capra. Edge computing in iot context: Horizontal and vertical linux container migration. In *2017 Global Internet of Things Summit (GIoTS)*, pages 1–4. IEEE, 2017.

[DL+15] Alexandre David, Kim G Larsen, et al. Uppaal smc tutorial. *International Journal on Software Tools for Technology Transfer*, 17(4):397–415, 2015.

[DLWZ20] Xiaoheng Deng, Jin Liu, Leilei Wang, and Zhihui Zhao. A trust evaluation system based on reputation data in mobile edge computing network. *Peer-to-Peer Networking and Applications*, 13:1744–1755, 2020.

[DMB18] Vincenzo De Maio and Ivona Brandic. First hop mobile offloading of dag computations. In *IEEE/ACM Int'l. Symp. on Cluster, Cloud and Grid Comp.*, pages 83–92, 2018.

[DMB19] Vincenzo De Maio and Ivona Brandic. Multi-objective mobile edge provisioning in small cell clouds. In *ACM/SPEC Int'l. Conf. on Perf. Eng.*, pages 127–138, 2019.

[DP+13] Mark DeVirgilio, W David Pan, et al. Internet delay statistics: Measuring internet feel using a dichotomous hurst parameter. In *IEEE Southeastcon*, pages 1–6, 2013.

[DSRS19] Mazin Debe, Khaled Salah, Muhammad Habib Ur Rehman, and Davor Svetinovic. Iot public fog nodes reputation system: A decentralized solution using ethereum blockchain. In *Proceedings of the IEEE International Conference on Communications (ICC)*, pages 1–10, 2019.

[dSV+19] da Silva Veith et al. Multi-objective reinforcement learning for reconfiguring data stream analytics on edge computing. In *International Conference on Parallel Processing*, page 106, 2019.

[DSX+18] Nour Diallo, Weidong Larry Shi, Lei Xu, Zhimin Gao, Lin Chen, Yang Lu, Nolan Shah, Larry Carranco, Ton Chanh Le, Abraham Bez Surez, and Glenn Turner. egov-dao: a better government using blockchain based decentralized autonomous organization. *2018 International Conference on eDemocracy & eGovernment (ICEDEG)*, pages 166–171, 2018.

[etha] ""ethereum test network" https://medium". "Ethereum Test network" https://medium.com/coinmonks/ethereum-test-network-21baa86072fa (Accessed: 2024-02-07).

[ethb] ""what is gwei? the cryptocurrency explained" https://www". "What Is Gwei? The Cryptocurrency Explained" https://www.investopedia.com/terms/g/gwei-ethereum.asp (Accessed: 2024-02-07).

[FA18] Qiang Fan and Nirwan Ansari. Towards workload balancing in fog computing empowered iot. *IEEE Transactions on Network Science and Engineering*, 7(1):253–262, 2018.

[FHZ+22] Chuan Feng, Pengchao Han, Xu Zhang, Bowen Yang, Yejun Liu, and Lei Guo. Computation offloading in mobile edge computing networks: A survey. *Journal of Network and Computer Applications*, 202:103366, 2022.

[gcl] ""cloud storage pricing" https://cloud". "Cloud Storage pricing" https://cloud.google.com/storage/pricing (Accessed: 2022-30-11).

[GD05] Saikat Guha and Neil Daswani. An experimental study of the skype peer-to-peer voip system. Technical report, Cornell University, 2005.

[Gra23] GrandViewResearch. Mobile application market size, share and trends analysis report by store (google store, apple store, others), by application (gaming, music and entertainment, health and fitness, social networking), and region, segment forecasts, 2024 - 2030, 2023. Accessed: 13-Feb-2025.

[hea] Network heartbeat configuration. `https://www.aerospike.com/docs/operations/configure/network/heartbeat/`. Accessed: 2020-09-02.

[HLMZ21] Xiaoyan Huang, Supeng Leng, Sabita Maharjan, and Yan Zhang. Multi-agent deep reinforcement learning for computation offloading and interference coordination in small cell networks. In *Proceedings of the IEEE International Conference on Communications (ICC)*, pages 1–6, 2021.

[Høf14] Andrea Høfler. Smt solver comparison. *Graz, July*, page 17, 2014.

[HXW+20] Miao Hu, Zixuan Xie, Di Wu, Yipeng Zhou, Xu Chen, and Liang Xiao. Heterogeneous edge offloading with incomplete information: A minority game approach. *IEEE Transactions on Parallel and Distributed Systems*, 31(9):2139–2154, 2020.

[IGAK+15] Bukhary Ikhwan Ismail, Ehsan Mostajeran Goortani, Mohd Bazli Ab Karim, Wong Ming Tat, Sharipah Setapa, Jing Yuan Luke, and Ong Hong Hoe. Evaluation of docker as edge computing platform. In *2015 IEEE Conference on Open Systems (ICOS)*, pages 130–135. IEEE, 2015.

[IMRN20] Sarah Iqbal, Asad Waqar Malik, Anis Ur Rahman, and Rafidah Md Noor. Blockchain-based reputation management for task offloading in micro-level vehicular fog network. *IEEE Access*, 8:52968–52980, 2020.

[INMR21] Sarah Iqbal, Rafidah Md Noor, Asad Waqar Malik, and Anis U Rahman. Blockchain-enabled adaptive-learning-based resource-sharing framework for iiot environment. *IEEE Internet of Things Journal*, 8(19):14746–14755, 2021.

[JCG+19] Congfeng Jiang, Xiaolan Cheng, Honghao Gao, Xin Zhou, and Jian Wan. Toward computation offloading in edge computing: A survey. *IEEE Access*, 7:131543–131558, 2019.

[JS19] Lara Lorna Jiménez and Olov Schelén. Docma: A decentralized orchestrator for containerized microservice applications. In *2019 IEEE Cloud Summit*, pages 45–51, 2019.

[K+11] M. Kwiatkowska et al. PRISM 4.0: Verification of probabilistic real-time systems. In *International Conference on Computer Aided Verification*, pages 585–591, 2011.

[KA+12] Sokol Kosta, Andrius Aucinas, et al. Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offl. In *IEEE Infocom*, pages 945–953, 2012.

[KL10] Karthik Kumar and Yung-Hsiang Lu. Cloud computing for mobile users: Can offloading computation save energy? *Computer*, (4):51–56, 2010.

[KLBK24] Eugene Korneev, M. Liubogoshchev, D. Bankov, and Evgeny M. Khorov. How to model cloud vr: An empirical study of features that matter. *IEEE Open Journal of the Communications Society*, 5:4155–4170, 2024.

[KXL+21] Jiawen Kang, Zehui Xiong, Xuandi Li, Yang Zhang, Dusit Niyato, Cyril Leung, and Chunyan Miao. Optimizing task assignment for reliable blockchain-empowered federated edge learning. *IEEE Transactions on Vehicular Technology*, 70(2):1910–1923, 2021.

[LL23] Chunhui Liu and Kai Liu. Toward reliable dnn-based task partitioning and offloading in vehicular edge computing. *IEEE Transactions on Consumer Electronics*, 70(1):3349–3360, 2023.

[LMFH23] Jingyu Liang, Bowen Ma, Zihan Feng, and Jiwei Huang. Reliability-aware task processing and offloading for data-intensive applications in edge computing. *IEEE Transactions on Network and Service Management*, 20(4):4668–4680, 2023.

[LMP+21] Ivan Lujic, Vincenzo De Maio, Klaus Pollhammer, Ivan Bodrozic, Josip Lasic, and Ivona Brandic. Increasing traffic safety with real-time edge analytics and 5g. In *Proceedings of the 4th International Workshop on Edge Systems, Analytics and Networking*, pages 19–24, 2021.

[LZC+20] Hai Lin, Sherali Zeadally, Zhihong Chen, Houda Labiod, and Lusheng Wang. A survey on computation offloading modeling for edge computing. *Journal of Network and Computer Applications*, 169:102781, 2020.

[MAUY19] Bashir Mohammed, Irfan Awan, Hassan Ugail, and Muhammad Younas. Failure prediction using machine learning in a virtualised hpc system and application. *Cluster Computing*, 22(2):471–485, 2019.

[MB17] Pavel Mach and Zdenek Becvar. Mobile edge computing: A survey on architecture and computation offloading. *arXiv preprint arXiv:1702.05309*, 2017.

[MJSS+18] Carlos Molina-Jimenez, Ioannis Sfyrakis, Ellis Solaiman, Irene Ng, Meng Weng Wong, Alexis Chun, and Jon Crowcroft. Implementation of smart contracts using hybrid architectures with on and off–blockchain components. In *2018 IEEE 8th International Symposium on Cloud and Service Computing (SC2)*, pages 83–90. IEEE, 2018.

[mob] The specs that really count when buying a phone. `https://smartphones.gadgethacks.com/how-to/specs-really-count-when-buying-phone-0171678/`. Accessed: 2019-09-05.

[MUB19] Vincenzo De Maio, Rafael Brundo Uriarte, and Ivona Brandic. Energy and profit-aware proof-of-stake offloading in blockchain-based vanets. In *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing (UCC)*, pages 177–186, 2019.

[MYZ+24] Jiayu Ma, Yuhan Yi, Wenqian Zhang, Yue Sun, and Guanglin Zhang. Blockchain-based task offloading for mobile edge computing networks with server collaboration. *2024 5th Information Communication Technologies Conference (ICTC)*, pages 221–226, 2024.

[ope] "opencellid, 2021, (https://opencellid". OpenCellID, 2021, (https://opencellid.org/).

[OWYZ08] Shumao Ou, Yumin Wu, Kun Yang, and Bosheng Zhou. Performance analysis of fault-tolerant offloading systems for pervasive services in mobile wireless environments. In *IEEE Int'l. Conf. on Communications*, pages 1856–1860, 2008.

[PNMH21] Tahmid Hasan Pranto, Abdullah Al Noman, Atik Mahmud, and Akm Bahalul Haque. Blockchain and smart contract for iot enabled smart agriculture. *PeerJ Computer Science*, 7, 2021.

[Put14] Martin L Puterman. *Markov Decision Processes.: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, 2014.

[PZBX22] Kai Peng, Bohai Zhao, Muhammad Bilal, and Xiaolong Xu. Reliability-aware computation offloading for delay-sensitive applications in mec-enabled aerial computing. *IEEE Transactions on Green Communications and Networking*, 6(3):1511–1519, 2022.

[RGW+21] Jie Ren, Ling Gao, Xiaoming Wang, Miao Ma, Guoyong Qiu, Hai Wang, Jie Zheng, and Zheng Wang. Adaptive computation offloading for mobile augmented reality. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 5(4):1–30, 2021.

[RKG18] Robert Robere, Antonina Kolokolova, and Vijay Ganesh. The proof complexity of smt solvers. In *Computer Aided Verification: 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II 30*, pages 275–293. Springer, 2018.

[RM14] Olivier Rioul and José Carlos Magossi. On shannon's formula and hartley's rule: Beyond the mathematical coincidence. *Entropy*, 16(9):4892–4910, 2014.

[SDS+23] Jinming Shi, Jun Du, Yuan Shen, Jian Wang, Jian Yuan, and Zhu Han. Drl-based v2v computation offloading for blockchain-enabled vehicular networks. *IEEE Transactions on Mobile Computing*, 22:3882–3897, 2023.

[SG09] Bianca Schroeder and Garth A Gibson. A large-scale study of failures in high-performance computing systems. *IEEE transactions on Dependable and Secure Computing*, 7(4):337–350, 2009.

[SHBZ19] Tonghoon Suk, Jinho Hwang, Muhammed Fatih Bulut, and Zemei Zeng. Failure-aware application placement modeling and optimization in high turnover devops environment. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pages 115–123, 2019.

[SLE19] Amit Samanta, Yong Li, and Flavio Esposito. Battle of microservices: Towards latency-optimal heuristic scheduling for edge computing. In *2019 IEEE Conference on Network Softwarization (NetSoft)*, pages 223–227, 2019.

[SMKP23] Zahra Najafabadi Samani, Narges Mehran, Dragi Kimovski, and Radu Prodan. Proactive sla-aware application placement in the computing continuum. In *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 468–479. IEEE, 2023.

[SPJ17] Dimas Satria, Daihee Park, and Minho Jo. Recovery for overloaded mobile edge computing. *Future Generation Computer Systems*, 70:138–147, 2017.

[SSX+12] Aki Saarinen, Matti Siekkinen, Yu Xiao, Jukka K Nurminen, Matti Kemppainen, and Pan Hui. Can offloading save energy for popular apps? In *Seventh ACM international workshop on Mobility in the evolving internet architecture*, pages 3–10, 2012.

[ST20] Amit Samanta and Jianhua Tang. Dyme: Dynamic microservice scheduling in edge computing enabled iot. *IEEE Internet of Things Journal*, 7(7):6164–6174, 2020.

[SWS21] Ellis Solaiman, Todd Wike, and Ioannis Sfyrakis. Implementation and evaluation of smart contracts using a hybrid on-and off-blockchain architecture. *Concurrency and computation: practice and experience*, 33(1):e5811, 2021.

[SYCC21] Lijun Sun, Qian Yang, Xiao Chen, and Zhenxiang Chen. Rc-chain: Reputation-based crowdsourcing blockchain for vehicular networks. *Journal of network and computer applications*, 176:102956, 2021.

[TE+16] Mohammad Tawalbeh, Alan Eardley, et al. Studying the energy consumption in mobile devices. *Procedia Computer Science*, 94:183–189, 2016.

[TL+16] Mati B Terefe, Heezin Lee, et al. Energy-efficient multisite offloading policy using mdp for mcc. *Pervasive and Mobile Computing*, 27:75–89, 2016.

[TYLG20] Jie Tang, Rao Yu, Shaoshan Liu, and Jean-Luc Gaudiot. A container based edge offloading framework for autonomous driving. *IEEE Access*, 8:33713–33726, 2020.

[Vis22] VisualCpaitalist. Charted: The rise of mobile device subscriptions worldwide, 2022. Accessed: 13-Feb-2025.

[WTAK20] Li Wu, Johan Tordsson, Alexander Acker, and Odej Kao. Microras: Automatic recovery in the absence of historical failure data for microservice systems. In *2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC)*, pages 227–236, 2020.

[Wu18a] Huaming Wu. Multi-objective decision-making for mobile cloud offloading: A survey. *IEEE Access*, 6:3962–3976, 2018.

[Wu18b] Huaming Wu. Performance modeling of delayed offloading in mobile wireless env. with failures. *IEEE Comm. Letters*, 22(11):2334–2337, 2018.

[WWW13] Qiushi Wang, Huaming Wu, and Katinka Wolter. Model-based performance analysis of local re-execution scheme in offloading system. In *IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 1–6, 2013.

[WYS21] Yue Wang, Tao Yu, and Kei Sakaguchi. Context-based mec platform for augmented-reality services in 5g networks. In *2021 IEEE 94th Vehicular Technology Conference (VTC2021-Fall)*, pages 1–5. IEEE, 2021.

[XEMDN21] Bin Xiang, Jocelyne Elias, Fabio Martignon, and Elisabetta Di Nitto. A dataset for mobile edge computing network topologies. *Data in Brief*, 39:107557, 2021.

[yCLlL23] Zheng yi Chai, Xu Liu, and Ya lun Li. A computation offloading algorithm based on multi-objective evolutionary optimization in mobile edge computing. *Eng. Appl. Artif. Intell.*, 121:105966, 2023.

[YLG+20] Yao Yu, Shumei Liu, Lei Guo, Phee Lep Yeoh, Branka Vucetic, and Yonghui Li. Crowdr-fbc: A distributed fog-blockchains for mobile crowdsourcing reputation management. *IEEE Internet of Things Journal*, 7(9):8722–8735, 2020.

[YLL15] Shanhe Yi, Cheng Li, and Qun A. Li. A survey of fog computing: Concepts, applications and issues. *Proceedings of the 2015 Workshop on Mobile Big Data*, 2015.

[YYC+23] Jian Yang, Qifeng Yuan, Shuangwu Chen, Huasen He, Xiaofeng Jiang, and Xiaobin Tan. Cooperative task offloading for mobile edge computing based on multi-agent deep reinforcement learning. In *Proceedings of the IEEE International Conference on Communications (ICC)*, pages 1–6, 2023.

[ZAB19] Josip Zilic, Atakan Aral, and Ivona Brandic. Efpo: Energy efficient and failure predictive edge offloading. In *12th IEEE/ACM International Conference on Utility and Cloud Computing*, pages 165–175, 2019.

[ZDMAB22] Josip Zilic, Vincenzo De Maio, Atakan Aral, and Ivona Brandic. Edge offloading for microservice architectures. In *Proceedings of the 5th International Workshop on Edge Systems, Analytics and Networking*, pages 1–6, 2022.

[ZEMR23] Zeinab Zabihi, Amir Masoud Eftekhari Moghadam, and Mohammad Hossein Rezvani. Reinforcement learning methods for computation offloading: a systematic review. *ACM Computing Surveys*, 56(1):1–41, 2023.

[ZLJZ21] Yutong Zhou, Xi Li, Hong Ji, and Heli Zhang. Blockchain-based trustworthy service caching and task offloading for intelligent edge computing. *2021 IEEE Global Communications Conference (GLOBECOM)*, pages 1–6, 2021.

[ZLLL17] Yifan Zhang, Yunxin Liu, Xuanzhe Liu, and Qun Li. Enabling accurate and efficient modeling-based cpu power estimation for smartphones. In *2017 IEEE/ACM 25th International Symposium on Quality of Service (IWQoS)*, pages 1–10. IEEE, 2017.

[ZNW15] Yang Zhang, Dusit Niyato, and Ping Wang. Offloading in mobile cloudlet systems with intermittent connectivity. *IEEE Transactions on Mobile Computing*, 14(12):2516–2529, 2015.

[ZWSZ21] Haibin Zhang, Rong Wang, Wen Sun, and Huanlei Zhao. Mobility management for blockchain-based ultra-dense edge computing: A deep reinforcement learning approach. *IEEE Transactions on Wireless Communications*, 20(11):7346–7359, 2021.

[ZWY+21] Zhili Zhou, Meimin Wang, Ching-Nung Yang, Zhangjie Fu, Xingming Sun, and QM Jonathan Wu. Blockchain-based decentralized reputation system in e-commerce environment. *Future Generation Computer Systems*, 124:155–167, 2021.

[ZYP+22] Nan Zhao, Zhiyang Ye, Yiyang Pei, Ying-Chang Liang, and Dusit Niyato. Multi-agent deep reinforcement learning for task offloading in uav-assisted mobile edge computing. In *Proceedings of the IEEE International Conference on Communications (ICC)*, pages 1–6, 2022.