Al-Farabi Kazakh National University

UDC 004.421(043.3) Under manuscript copyright

MUKHIT ZHANUZAKOV

Research of fault-tolerance algorithms for reliable information processing

8D06103 – Computer Science

Dissertation prepared for the degree of
Doctor of Philosophy (PhD)

Domestic scientific supervisor:
Doctor of physical and mathematical sciences, professor
Balakaeva Gulnar Tultaevna
Al-Farabi Kazakh National University,
Almaty, Kazakhstan

Foreign scientific supervisor:
PhD, Paul Ezhilchelvan
Newcastle University, UK

Kazakhstan, Almaty, 2026

CONTENT

REGULATORY REFERENCES 4

ABBREVIATIONS AND DEFINITIONS 5

INTRODUCTION 6

1 CURRENT SITUATION OF THE RELIABILITY OF INFORMATION PROCESSING IN ENTERPRISE DISTRIBUTED SYSTEMS 10

1.1 Fault tolerance algorithms 12

1.2 Load balancing algorithms 16

1.2.1 Complexity analysis of load balancing algorithms 18

1.3 Rollback-recovery with checkpointing fault-tolerance algorithm 22

1.4 Architecture of enterprise distributed systems 26

1.5 Failures and Assumptions 29

1.5.1 Server applications crash 29

1.5.2 Physical server crash 30

2. DEVELOPMENT AND DESIGNING ALGORITHMS OF ROLLBACK-RECOVERY BASED ON CHECKPOINTING 32

2.1 Development of model based algorithm for calculating optimal checkpointing interval 32

2.2 Development of an algorithm for calculating optimal checkpointing interval based on queuing theory 35

3. IMPLEMENTATION AND TESTING OF CHECKPOINTING BASED ALGORITHMS IN ENTERPRISE DISTRIBUTED SYSTEMS 40

3.1 System architecture 40

3.2 System model 41

3.3 Performance evaluation setup 42

3.4 Experiment 1: Implementation and Testing of Rollback-Recovery Algorithms Using Virtual Machines in Real Distributed Enterprise Systems 43

3.5 Experiment 2: Implementation and Testing of Rollback-Recovery Algorithms Using Physical Machines in Real Distributed Enterprise Systems 45

3.6 Comparitive analysis and discussion of testing results 47

3.6.1 Results of Performance Evaluation in a Virtualized Environment (Experiment 1) 47

3.6.2 Results of Performance Evaluation in a Physical Environment (Experiment 2) 49

3.6.3 Discussion of comparison analysis results 50

3.7 Performance overhead analysis: Model based solution for computing checkpointing interval 51
3.8 Performance overhead analyses: Optimal checkpointing based on queuing theory .53
3.8.1 Evaluation Metrics 54
3.8.2 Failure Simulation Scenarios 56
3.8.3 Discussion of results and Bottleneck Analysis 58
4. DEVELOPMENT OF DIGITAL SCALABLE FAULT-TOLERANT ENTERPRISE SYSTEM USING AI TECHNOLOGIES 60
4.1 Application of fault-tolerance mechanisms 60
4.2 System description 61
4.3 Related works on development of enterprise systems incorporating AI and BigData tools 63
4.4 Assessing the reliability of information, identifying erroneous data: anomalies, duplicates, contradictions, empty values, correcting identified errors 67
4.5 Development of a business process for coordination and approval of documents....68
4.6 Scalability and application of load-balancing strategies 80
4.7 Description of technologies used in the development of enterprise system 84
CONCLUSION 88
REFERENCES 89
APPENDIX A 98
APPENDIX B.....99

REGULATORY REFERENCES

In this dissertation, references were used in accordance with the following standards:

"Guidelines for Dissertation Formatting," Committee for Control in Education and Science of the Ministry of Education and Science of the Republic of Kazakhstan, September 28, 2004, No. 377-3.

GOST 7.32-2001. Report on Scientific Research Work. Rules and Structure of Formatting.

GOST 7.1-2003. Bibliographic Record. Bibliographic Description.

ABBREVIATIONS AND DEFINITIONS

HPC – High-Performance Computing
AI – Artificial Intelligence
IoT – Internet of Things
BPM – Business process management
FT – Fault tolerance
SA – Server Application
CDF - Cumulative Distribution Function
USD - United States dollar
BoT - Bags of Tasks
WSSS - Weakly-Supervised Semantic Segmentation
TCC - Time-aware Checkpointing Coordination
LRT - Least Response Time
IP - Internet Protocol
MTTF - mean time to failure
I/O - Input/Output
AWS - Amazon Web Services
HTTP - Hypertext Transfer Protocol
CPU - central processing unit
FIFO - First in, first out
LAN - Local Area Network
Mbps - megabits per second
MTTR - Mean Time to Repair
SVM - support vector machine
CNN - convolutional neural network
HSI - hyperspectral image
BERT - Bidirectional Encoder Representations from Transformers
CTR - Connection Termination Rate
BFT - Byzantine Fault Tolerance

INTRODUCTION

Relevance of the research.

The increasing complexity of computing systems introduces higher risks of failures. Existing fault-tolerance algorithms often suffer from scalability issues, high computational overhead, or limited adaptability to diverse failure scenarios. This research aims to develop and evaluate algorithms that overcome these limitations, providing robust solutions for reliable information processing.

In the contemporary business environment, the impact of digitalization on enterprises is a transformative force. Integrating digital technologies such as AI, IoT, and data analytics into organizational operations leads to increased efficiency, innovation, and competitiveness. However, this transition is accompanied by obstacles like as data consistency, security, privacy, and the need for a digitally competent workforce. This dissertation attempts to give precise insights into the strategic components of company digitalization through case studies and trend analysis. Understanding these implications is critical for firms crossing the digital frontier and guaranteeing long-term development and competitiveness in today's fast-paced business environment.

The significance of fault-tolerance lies in mitigating the risks associated with system failures, ensuring uninterrupted operations, and safeguarding critical data [1]. Scalability, on the other hand, is crucial for accommodating increased workloads and sustaining optimal performance as enterprises evolve.

When a physical server crashes, applications running on the server become unresponsive. If suspended applications are not quickly restored, the organizations that own them will suffer loss of revenue and reputation. As a result, it is important that applications are built with resiliency in mind to reduce downtime and provide customers with near uninterrupted service. The ability of a system to continue operating in the face of failures or errors determines the crash tolerance of the system [3]. Crash-tolerant methods detect and recover from errors, ensuring that the system continues to function according to its pre-failure behavior.

In this context, incorporating resiliency provisions into server applications is critical to ensure that customers receive a continuous and consistent experience. They must be able to ensure data consistency, prevent data loss, and recover from hardware or software errors.

The purpose of the dissertation work. The aim of this research is to develop and analyze a fault-tolerant information processing methods for distributed systems, in particular enterprise servers, that minimizes performance overhead in failure-free operation while ensuring fast and reliable rollback-recovery algorithms through an optimized checkpointing interval.

Research tasks:

1) Research of fault-tolerance algorithms for reliable information processing

2) Development of a model-based solution and designing an algorithm for computing checkpointing interval for fault-tolerant rollback-recovery algorithm in distributed enterprise systems

3) Designing and implementation of an algorithm for computing checkpointing

interval for fault-tolerant rollback-recovery algorithm based on queuing theory

4) Implementation and testing of checkpointing-based rollback-recovery algorithms in a real distributed enterprise systems

Research object. Distributed systems, Queuing theory, Fault-tolerance algorithms, Checkpointing interval calculation, Enterprise information systems

Research subject. Fault tolerance algorithms for reliable information processing

Research methods.Theoretical analysis of fault-tolerance algorithms; enterprise distributed system modelling; designing and testing of rollback-recovery fault-tolerance algorithms.

Research novelty:

The novelty of the research work is the development of new and optimized solutions for calculating checkpointing interval for rollback-recovery algorithm. This includes following:

1) First implementation of optimal checkpointing interval for rollback-recovery algorithm based on queuing theory

2) Model-based solution for computing checkpointing interval for fault-tolerant rollback-recovery in distributed enterprise systems

Theoretical and practical significance of the work.

The study develops algorithms for reliable information processing in distributed enterprise systems. It extends existing theories of rollback-recovery mechanisms by introducing approaches for optimal checkpointing interval computation under varying system loads and failure rates. The research contributes to the theoretical understanding of the trade-off between system performance and fault tolerance overhead. The developed algorithms are implemented and tested in both simulated and realistic the distributed systems, showing the practical significance of the work.

Main provisions to be defended.

Development of new models and algorithms for calculating optimal checkpointing interval for the fault-tolerant rollback-recovery algorithm in distributed enterprise systems.

Development and application of an algorithm for calculating the optimal checkpointing interval based on queuing theory, which demonstrated a system availability of up to 96.27% and achieved an improvement in overall system efficiency of approximately 3–4% compared to traditional algorithms.

The degree of reliability and the results of approbation.

7 articles were published on the topic of the dissertation. The results of the study were discussed at scientific seminars at the Department of Computer Sciences of Al-Farabi kaznu, as well as at seminars with the participation of a foreign supervisor, and reported at the following international conferences:

1. Balakayeva G., Zhanuzakov M., Darkenbayev D. Comparative analysis of fault tolerance algorithms for fault handling during digitalization of enterprises // Lecture Notes in Networks and Systems. – London: Springer, 2026.

2. Balakayeva G., Nurlybayeva K., Zhanuzakov M. Development of software complex for digitalization of enterprise activities // International Journal of Electrical and Computer Engineering. – 2023. – Vol. 17, No. 12. – P. 315–318.

Articles published in the publication recommended by the Committee for Quality

Assurance in the field of Science and Higher Education of the Ministry of Science and Higher Education of the Republic of Kazakhstan:

1. Zhanuzakov M., Balakayeva G., Ezhilchelvan P. Model based solution for computing checkpointing interval for fault-tolerant rollback-recovery in enterprise servers // Journal of Problems in Computer Science and Information Technologies. – 2025. – Vol. 3, No. 1. – P. 25–33. – DOI: 10.26577/jpcsit20253103.

2. Zhanuzakov M., Balakayeva G. A comparative study of load-balancing algorithms for reliability of enterprise information systems // Bulletin of Abai KazNPU. Series of Physical and Mathematical Sciences. – 2024. – Vol. 87, No. 3. – P. 146–153. – DOI: 10.51889/2959-5894.2024.87.3.012.

3. Zhanuzakov M., Balakayeva G. Prediction of employee promotion based on ratings using machine-learning algorithms // Bulletin. Series Physical and Mathematical Sciences. – 2022. – DOI: 10.51889/2022-1.1728-7901.14.

Scientific articles in journals indexed in the Scopus database:

1. Balakayeva G., Zhanuzakov M., Kalmenova G. Development of a digital employee rating evaluation system (DERES) based on machine learning algorithms and 360-degree method // Journal of Intelligent Systems. – 2023. – Vol. 32, No. 1. – Art. 20230008. – DOI: 10.1515/jisys-2023-0008.

2. Balakayeva G., Zhanuzakov M., Zhapbasbayev U., Nurlybayeva K. An intelligent enterprise system with processing and verification of business documents using big data and AI // Journal of Intelligent Systems. – 2025. – Vol. 34, No. 1. – Art. 20240446. – DOI: 10.1515/jisys-2024-0446.

3. Balakayeva G., Darkenbayev D., Zhanuzakov M. Development of a software system for predicting employee ratings // Informatyka, Automatyka, Pomiary w Gospodarce i Ochronie Środowiska. – 2023. – Vol. 13, No. 3. – P. 121–124. – DOI: 10.35784/iapgos.3723.

This work was carried out as part of the scientific project AP09259208 entitled «Development of a scalable fault-tolerant information system for enterprise digitalization using Big Data technologies» (0121RK00369).

Personal contribution of the researcher. The researcher independently solved all the tasks of the dissertation work. Modern information and programming technologies were used. Algorithms were developed and implemented for practical application in distributed systems of enterprises. The objectives of the research were proposed by local and foreign scientific advisors.

Volume and structure of the work. The dissertation consists of an introduction, four sections, a conclusion and a list of references. The total volume of the dissertation work: 100 pages of written text, including 35 Figures, 11 tables, sources of literature 102, appendices.

Introduction section determines the relevance of the work and the indicates issues related to the topic. It describes the idea of the work, the goals and objectives of the study, the scientific novelty and practical value of the study and research methods.

The first chapter provides a review of scientific papers and a detailed analysis of fault-tolerance and load-balancing algorithms. It also presents the research gap, introduces the design challenges.

The second chapter describes algorithms for calculating optimal checkpointing

interval for rollback-recovery method developed during research. It includes theoretical justification and algorithm design.

The third chapter describes the implementation and testing of the developed algorithms in both simulated and production-like enterprise environment setup. This section provides comparative analysis of traditional algorithms and developed algorithms.

In conclusion section, the main results and conclusions of the dissertation research are outlined.

1 CURRENT SITUATION OF THE RELIABILITY OF INFORMATION PROCESSING IN ENTERPRISE DISTRIBUTED SYSTEMS

This section describes current situation of the reliable information processing in the world, as well as existing fault-tolerance methods and problem identification.

Currently, reliable information processing is becoming relevant with the increase of digital information. The increase in the volume of digital information was due to the exponential growth of data mining and the widespread introduction of digital technologies. In 2024, more than 463 exabytes (10^{18} bytes) of data are estimated to be generated worldwide every day.

Almost all information is processed on servers. According to researchers [1], the enterprise server market is projected to be USD 87.96 billion in 2024 and USD 129.42 billion by 2029, an average increase of 8.03% during the forecast period (2024-2029). Figure 1.1 shows this data in a graph format for clear understanding.

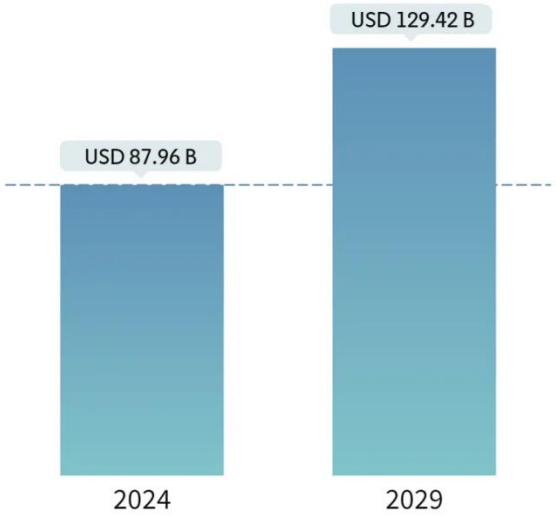


Figure 1.1 - Enterprise Server Market size predictions

Enterprise servers are crucial indicators of a country's level of digitalization. Figure 1.2 displays growth of enterprise server market by regions. We can see that most developing countries in Asia and Australia show high increase in enterprise server market including Kazakhstan. This suggests that the relevance and importance of the reliable enterprise servers in these regions will increase as well.

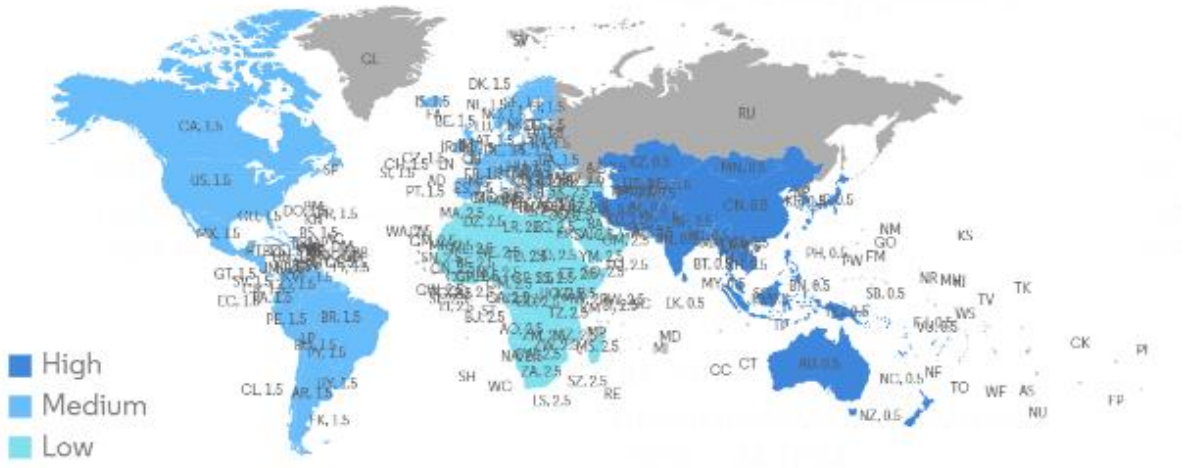


Figure 1.2 - Enterprise server market growth rate by region

Enterprise servers can fail while processing any information, leading to data loss and disruption of the entire system. Other researchers from Uptime Institute [2] analyzed outage of servers from 2021 to 2023. An outage is a period of time when a computer system, service, network, or infrastructure is unavailable or not functioning properly. Figure 1.3 shows that according to their research, average cost per outage increased from 100 000 USD to 200 000 USD during the examined period. Therefore, it is very important for enterprise systems to have methods and techniques that increase the system's resistance to failures.

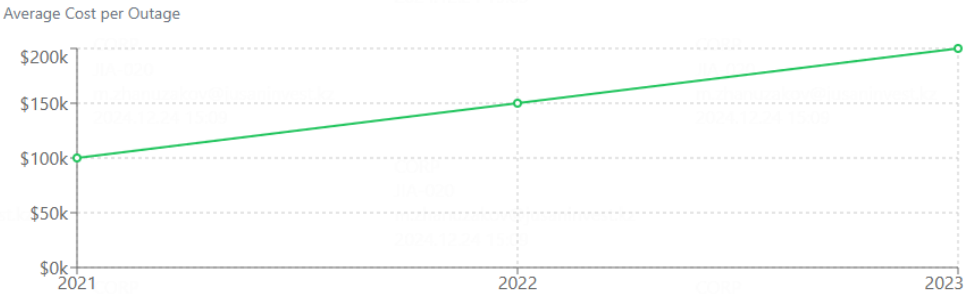


Figure 1.3 - Average cost per server outage

A popular solution for handling faults and malfunctions is the integration of fault-tolerance algorithms. Fault tolerance is the ability to eliminate unexpected malfunctions that occur during the operation of the system [3]. Using fault tolerance algorithms, one can achieve reliable operation of the entire system. This is especially important for enterprise information systems that are highly dependent on the operation of servers and perform high computing processes that require high availability.

Rollback-recovery algorithm is one of the fault-tolerance algorithms that allows to recover the server/system to a consistent state using checkpointed states called “checkpoints”. These checkpoints are stored in stable storage [3]. One of the hyperparameters of the algorithm is checkpointing interval, that is how often checkpointing should be taken. In this work, we aim to compute the optimal interval between each checkpoint while maintaining minimal impact on performance.

1.1 Fault tolerance algorithms

Fault is the condition of a component (server), when it fails to function as defined by the system. It can occur because of various reasons: network error, hardware crashes, application error, etc.

Fault tolerance is the ability to handle unexpected faults that occur during the system`s work [4]. By utilizing fault tolerance algorithms, we can achieve reliability and availability of the whole system. It is especially important for digitalization of enterprises that highly depend on servers` work and perform high computing processes that require high availability [5].

Fault tolerance in digitalization of enterprises helps ensure that the organization and its business operations run continuously. A system is fault-tolerant if it can continue to function despite failures. The standards for such systems get increasingly strict year after year, determining the overall quality of service. It is important to understand that tiny periods of subsystem downtime have an exponential effect on the overall functioning of the system, and this fault in the system only gets worse with time.

Fault tolerance is defined as the number of single failures of system components or pieces that sustain the system's operability. If performance deteriorates, it will be proportional to failure in an inadequately built system, where a little failure can lead to the system failing completely.

There are several algorithms and methods that are aimed at solving this problem. In this section, we make a comparative analysis of different types of fault-tolerance algorithms for enterprises.

In the work [6], authors emphasize on the importance of fault-tolerance in cloud computing and provide policies of the implementation of these methods. They also offer architectures using these policies with the implementation of fault detection and fault recovery techniques. The results show that the discussed models benefited greatly from fault tolerance.

In a conference work [7], author has conducted an experiment to demonstrate how fault tolerance can be achieved by determining how requests are assigned to clustered nodes. The researcher improves the web application with load balance technology configured with weighted round-robin algorithm. With the load balance technology configured with weighted round-robin algorithm, the strength and accessibility of the server is improved significantly.

Authors in the survey [8], provide insights into the existing fault tolerance approaches and future possible challenges. The work reviews promising methods used for efficient solutions and identifies important research directions in this area.

Work by Kirti et al. [9] systematically reviews fault tolerance methods in both distributed and cloud computing and with discussion of their taxonomy. They classify fault-tolerance approaches into four types: reactive, proactive, adaptive, and hybrid approaches. The article aims to give a better understanding of handling faults using suitable approaches and further compare them on various parameters.

Another work [10] is dedicated to a comprehensive and detailed description of

the various faults kinds, factors, & different methods to fault tolerance used in the cloud. It is noted that approaches of fault tolerance directed to checkpoint restart and replication are mainly included to address the crash faults in the cloud.

Kumari et al. [4] propose checkpointing protocols tailored for Bags of Tasks (BoT) and interdependent applications, utilizing edge switch storage to reduce resource consumption. Their simulation results demonstrate improved recovery efficiency compared to traditional approaches. However, such methods are often optimized for specific workload types and may not generalize well to heterogeneous enterprise systems with dynamic execution patterns.

Another research direction addresses fault tolerance under complex failure conditions, including Byzantine faults. Chinnathambi et al. [5] introduce the WSSS scheduling algorithm and the TCC checkpoint optimization technique. WSSS dynamically selects high-performance servers within virtual clusters, while TCC adjusts checkpoint intervals based on delay variations to minimize failure impact. Although these approaches improve reliability in distributed environments, they introduce additional system complexity and monitoring overhead, which may limit scalability in large-scale enterprise infrastructures.

A number of studies focus on theoretical modeling of checkpointing and recovery processes. Dimitriou [6] investigates retrial queue systems with checkpointing and rollback using stability conditions and steady-state analysis. Such models provide valuable insights into system behavior and performance optimization, particularly for mobile and wireless systems. Nevertheless, these approaches often remain abstract and lack direct applicability to real-world enterprise architectures without further system-level integration.

Using recent study [11], we can divide fault-tolerance approaches into four main types:

1. Reactive - provide a preventive measure after the occurrence of faults in the system;
2. Proactive - prevent the system or minimize failure effects by predicting in advance;
3. Adaptive - predict, learn, and adapt the changes to deal with new faults in the system;
4. Hybrid – all other approaches.

Reactive Algorithms:

Replication (Primary-Backup)

The primary-backup replication is a model where one server acts as a master server that receives and processes all the requests, and one or more servers are backup or standby servers [11]. This means that in the event the primary server is not operational, then one of the backup servers assumes the role of serving the services. This method ensures high availability and low server unavailability due to the presence of a failover system in the case of a server outage.

Checkpointing and Rollback Recovery

This technique involves partial or complete saving of the state of a process at certain intervals, then in the event of a failure one can revert the process to the state of the last saved check point [12]. In coordinated checkpointing techniques, all the

processes take their checkpoints with reference to a certain point in time while on the other hand uncoordinated checkpoint allows individual and does not necessarily have to take them at the same time. This is another important method of ensuring that the system has some form of fault tolerance in case of a detected a fault, then the system can go back to what is familiar to it.

Byzantine Fault Tolerance

Byzantine Fault Tolerance (BFT) deals with cases whereby some of the components may fail and either provide wrong or even fake details. Faults of such an arbitrary or malicious nature are addressed by algorithms like the PBFT (Practical Byzantine Fault Tolerance) by achieving consensus among the distributed components [13]. This approach is very important in the fields where trust and correctness are paramount, for example in the financial systems or the blockchain networks.

Load Balancing

Load balancing algorithms basically involve spreading the load within the various servers to avoid a situation whereby one server is heavily loaded [1]. Some forms of load balancing include round robin with health check and least connections with health check, which makes it a reactive load balancing technique since it constantly redistributes the load to the servers depending on the health of the servers and the amount of load that is placed on them. These algorithms help in cases when an unhealthy server is identified; the system doesn't route traffic to it or distributes load to other less-loaded healthy servers to ensure the system's efficiency and reliability.

Proactive Algorithms:

Replication (State Machine Replication)

State machine replication entails the executing of multiple instances of a service concurrently but in an order that makes their state identical [15]. This is a proactive approach that averts problems since if an instance is not able to handle requests, there is another one that can take over immediately. It further guarantees very high availability and consistency by anticipating the likely failure points and providing the necessary backup.

Self-Healing Systems

Self-healing computer systems apply methods such as heartbeats and self-diagnostic to identify and prevent faults. Heartbeats are a process of regularly sending signals between the components so that the status of their functioning can be checked. Self-monitoring components are also able to check its health and can even self-start or self-repair in cases of problems. These are preventive measures that help avoid faults turning into failures since faults are corrected before they become failures [16-19].

Adaptive Algorithms:

Consensus Algorithms

Consensus algorithms like Paxos and Raft ensure agreement among distributed systems, even in the presence of failures [20, 21]. Paxos is a famous and the stable leader in the field of fault-tolerant consensus, which uses the majority rule approach by electing multiple leaders thus, ensuring consistency even when some of the nodes are down. Raft expedites the consensus process and assigns clear roles such as leaders and followers, allowing the system to respond to changes or leader fatalities. Moreover, the algorithms are able to learn and re-learn the network and the node conditions by usage

of flexible algorithms, hence the fault tolerance property is unchanged

Hybrid Algorithms:

Quorum-Based Systems

In a system that is quorum-based they are taking in combinations of methods in order to be sure in terms of fault tolerance and consistency [22-24]. They need a minimum number of nodes (quorum) to agree on a transaction before it is committed. It can be seen that the latter approach is more appealing in terms of consistency and fault tolerance, in that, even if some nodes are out of the business, the system can still reach a decision and proceed with operations. These types of systems leverage majority agreement and other mechanisms to provide better fault tolerance and data integrity across distributed systems[25-27].

Measurement metrics

1. Performance refers to an algorithm's efficiency in terms of latency and throughput (from low to high).
2. Reliability: capacity to maintain system availability and accuracy in the presence of defects (from low to high).
3. Scalability: how effectively an algorithm can manage an increasing number of nodes or components (from low to high).
4. Complexity: the complexity in implementing and maintaining the algorithm (from low to high).
5. Fault Tolerance (FT): type and quantity of defects that the algorithm can handle (types of faults the method is able to handle).
6. Consistency: the algorithm's capacity to keep data consistent across the system(from low to high).
7. Resource utilization: the amount of computing and network resources utilized by the method(from low to high).

We analyzed the methods and algorithms, the results of these finding are presented in Table 1.1.

Table 1.1 - Comparison of different existing fault-tolerance algorithms

Algorithm	**Performance**	**Scalability**	**Complexity**	**Fault types**	**Consistency**	**Resource utilization**
Replication	high	high	moderate	hardware failures	high	high
Rollback-recovery with	high	high	moderate	Software/ hardware faults	high	moderate

checkpointing						
Consensus Algorithms	low	moderate	high	arbitrary faults including network partitions	high	high
Self-Healing System	high	high	high	hardware, software, transient faults	moderate	moderate
Load Balancing with Health Checks	high	high	moderate	server failures	not applicable	moderate
Byzantine Fault Tolerance	low	moderate	high	malicious faults	high	high
Quorum-Based Systems	moderate	high	moderate	node failures	high	moderate

Results of the comparison show that almost all methods are reliable, however, the Rollback-Recovery with checkpointing provide best reliability and performance.

The comparison of features show that each method has its advantages and disadvantages over others. Replication methods show high consistency but can be resource intensive. Although, checkpointing offers high reliability with fair scalability, it can introduce latency during rollbacks adding small performance issues. Self-healing systems provide high reliability and scalability but require complex monitoring and recovery mechanisms increasing complexity of the implementation. Load balancing with health checks is simple and scalable, yet it does not consider consistency of information. Byzantine Fault Tolerance is the most robust against most of the faults butcosts performance and complexity. Quorum-based systems balance reliability and consistency with moderate complexity overhead and require more resources.

1.2 Load balancing algorithms

The current enterprise information systems are designed to feed many users and execute complicated data calculations that cannot be performed by a single machine. Nowadays, enterprise systems are distributed systems comprising several nodes, each node is considered as an independent computer. When the primary server is busy handling resourceful calculations, requests from other clients keep coming to that server and all these requests are put into the queue. The idea of the load balancing algorithm is to distribute these requests to other servers, so that the load on each server is near equal [28].

Load balancing algorithms can be divided into two main types: the operated media [29]. The goal of a static load balancing algorithm is uniformly spreading incoming requests to a pool of existing servers which is achieved by defining particular parameters or using fixed set of rules. In contrast, the dynamic load balancer is continuous and real-time, which monitors inbound traffic and adjusts the servers dynamically by determining their load value, for instance, CPU, memory or number of active connections.

In the article by Beniwal P. et al., authors compare load-balancing algorithms by different parameters such as overhead, response time etc. Another review done by Sidra Aslam et al., reviews different methods for load-balancing between 2004-2015 and its relevance in cloud computing. Similar research was carried out by M. Randles et al., where the authors investigate Honeybee Foraging Behaviour, Biased Random Sampling and Active Clustering methods in cloud computing.

In this section, we present a comparative review of the most popular load-balancing algorithms and make a complexity analysis of each. In the methods section, we outline the methods used to compare and analyze the algorithms. The results of the analysis are described in the results sections. The discussion part underlines the interpretations of the findings and suggests further scientific directions of the research.

The compared features are the time complexity of the algorithm for handling one request, support of fault-tolerance and type of the algorithm (static or dynamic). This section will provide a list of algorithms that are chosen for analysis and gives a description for each one.

Round Robin

The Round Robin algorithm is a basic load balancing mechanism that balances incoming traffic between a number of servers by round-robin fashion [30]. The distribution pattern in this case is static, not changeable by the status of the system or various conditions affecting workload. After this sequences for task assignment are decided, they run indefinitely without any change that may come as a result of workload or resource availability variation.

Random

The Random load balancing algorithm uses a random number generator to select a server from the available pool for each incoming request.

Threshold Algorithm

The Threshold Algorithm for load balancing in distributed systems operates by assigning threshold values to each processor, monitoring their workload continuously, and dynamically adjusting thresholds to redistribute tasks when necessary [31]. By

setting acceptable load levels for each processor, the algorithm aims to prevent overload situations while maximizing resource utilization. When a processor's load surpasses its threshold, tasks are migrated to other processors with lower loads to maintain balance. Periodic updates to thresholds ensure adaptability to changing system conditions and workload patterns [32]. This dynamic nature allows the Threshold Algorithm to adapt to fluctuations in workload, changes in system resources, and variations in processing capabilities among different processors.

IP hash

The IP Hash load balancing algorithm is a method used to distribute incoming requests or traffic across multiple servers or resources based on the hash value of the source IP address of each request. The basic idea is to consistently map each unique IP address to a particular server, ensuring that requests from the same IP address are always directed to the same server [33]. This helps maintain session affinity or "sticky sessions" for clients, which can be important for stateful applications or sessions that need to be maintained across multiple requests.

Least Connections

The Least Connections algorithm is a popular load balancing technique that aims to distribute incoming requests evenly across a pool of servers. It achieves this by directing each request to the server with the fewest active connections at the time [34, 35]. This method helps prevent overloading any specific server and ensures efficient utilization of resources.

When a new request arrives, the load balancer examines the connection count of all available servers. It then selects the server with the lowest number of active connections and forwards the request to that server.

Least Response Time

The Least Response Time (LRT) algorithm is a dynamic load balancing technique that aims to distribute incoming requests to servers based on their average response times [36]. This ensures that requests are directed to the servers that can process them the fastest, ultimately reducing overall user wait times and improving application performance.

1.2.1 Complexity analysis of load balancing algorithms

Round Robin

Pseudo code of the algorithm is as follows:

```
servers = [server1, server2, ..., serverN]
pointer = 0
function handleRequest(request):
    server = servers[pointer]
    pointer = (pointer + 1) % len(servers)
    sendRequestToServer(request, server)
while True:
    request = receiveRequest()
    handleRequest(request)
```

The Big O complexity of the provided Round Robin algorithm depends on the number of requests and the number of servers. Initialization step simply involves creating a list of servers and initializing a pointer. It runs in constant time regardless of the number of servers. So, it has a time complexity of O(1). The time complexity of handling a single request depends on two factors: Accessing the server pointed to by the pointer, which takes constant time O(1)). Updating the pointer, which involves adding 1 to the pointer and performing a modulo operation with the length of the servers list. Both of these operations are constant time as well O(1). Therefore, the overall time complexity of handling a single request is O(1). The main loop continuously receives and handles requests using the handleRequest function. Since the time complexity of handleRequest is O(1), the time complexity of the main loop depends on the number of requests received. If there are N requests, the time complexity of the main loop would be O(N). Combining all these factors, the overall time complexity of the provided Round Robin algorithm is O(N), where N is the number of requests.

Random

Pseudo code of the algorithm:

```
function choose_server(servers):
  random_index = generate_random_number(0, len(servers) - 1)
  return servers[random_index]
```

The time complexity of the Random load balancing algorithm is O(1) (constant). This means the time it takes to select a server for a request doesn't depend on the number of servers in the pool.

Threshold Algorithm

Pseudo code of the algorithm:

```
resources = list of available resources
overloadThreshold = predefined threshold value
underloadThreshold = predefined threshold value

while True:  // Continuously monitor and balance load
   for each resource in resources:
      currentLoad = measureLoad(resource)  // Get current load on resource

      // Overloaded condition
      if currentLoad > overloadThreshold:
         // 1. Stop assigning new tasks to this resource
         // 2. Optionally, try to migrate existing tasks to underloaded resources
         underloadedResource = findUnderloadedResource()
         if underloadedResource is not None:
            migrateTasks(resource, underloadedResource)

      // Underloaded condition
      if currentLoad < underloadThreshold:
         // Attempt to attract tasks from overloaded resources
         attractTasks(resource)
```

The threshold algorithm for load balancing typically has a constant time complexity (O(1)) for monitoring load and comparing it to thresholds. However, the complexity of taking action (e.g., finding a suitable resource for migration) can vary depending on the implementation. In ideal scenarios with centralized resource lists, it might remain O(1). However, real-world scenarios involving distributed systems and searching for resources can introduce communication overhead, potentially pushing the complexity towards O(N) in the worst case. In essence, the overall complexity hinges on the efficiency of finding target resources.

IP hash

Pseudo code of the algorithm:

```
function hash(ipAddress)
  result = 0
  for each byte in ipAddress:
    result = result XOR byte
  return result

function distributeRequest(clientIpAddress, servers)
  hashValue = hash(clientIpAddress)
  serverIndex = hashValue % number_of_servers
  return servers[serverIndex]

clientIp = "10.0.0.1"
servers = [server1, server2, server3]
chosenServer = distributeRequest(clientIp, servers)
```

Choosing a suitable hash function plays a crucial role here. While a simple XOR function requires iterating through each byte of the IP address, leading to O(n) complexity (n being the number of bytes), using a more robust option like SHA-256 offers constant time complexity (O(1)) on average, even though it involves internal operations. Other steps like calculating the hash value and selecting the server are generally constant time (O(1)) due to their simplicity.

The algorithm itself requires constant space (O(1)) for variables like the hash value and server index. However, the space complexity of the server list itself depends on the chosen data structure. For instance, an array used to store the servers has a space complexity of O(n), where n is the number of servers.

Least Connections

Pseudo code of the algorithm:

```
function get_connection_count(server):
  #retrieve the actual number of connections on the server
  return connection_count
function choose_server(servers):
  min_connections = 0
  chosen_server = null
  for server in servers:
    connection_count = get_connection_count(server)
    if connection_count < min_connections:
```

```
        min_connections = connection_count
        chosen_server = server
    return chosen_server
servers = [server1, server2, server3]
incoming_request = "Request data"
chosen_server = choose_server(servers)
// Forward the request to the chosen server
```

The function get_connection_count could have varying complexity depending on the implementation. Iterating through servers and comparing connection counts is typically O(n) in the worst case, where n is the number of servers. The algorithm itself uses constant space (O(1)) as it only involves temporary variables. The server list might have its own space complexity depending on the data structure used (e.g., an array has O(n) space complexity).

Least Response Time

Pseudo code of the algorithm:

```
function get_average_response_time(server):
  // retrieve the server's average response time
  return average_response_time
function choose_server(servers):
  min_response_time = 0
  chosen_server = null
  for server in servers:
    response_time = get_average_response_time(server)
    if response_time < min_response_time:
      min_response_time = response_time
      chosen_server = server
  return chosen_server
```

Iterating through servers and comparing response times is similar to Least Connections, typically O(n) in the worst case. Space complexity is similar to Least Connections.

Several load-balancing algorithms incorporate fault-tolerance mechanisms to ensure service availability even when individual servers encounter failures. Round Robin with Health Checks includes health checks to remove and re-add servers from the pool based on their availability. Least Connections with Failover prioritizes servers with fewer connections but can be extended to automatically failover to a backup server if the primary server fails.

Table 1.2 - Complexity analysis results

Algorithm	Time complexity for single request	Time complexity for N requests	Space complexity	Fault - tolerant	Type
Round robin	O(1)	O(N)	O(n)	yes	static

Continuation of Table 1.2

Random	O(1)	O(N)	O(n)	no	static
Threshold Algorithm	O(n)	$O(N^2)$	O(n)	no	dynamic
IP hash	O(n)	$O(N^2)$	O(n)	no	static
Least Connections	O(n)	$O(N^2)$	O(n)	yes	dynamic
Least Response Time	O(n)	$O(N^2)$	O(n)	no	dynamic

The complexity analysis results are presented in Table 1.2. Static load-balancing algorithms are simple to implement and effective if the system has a predictable number of users. Random and Round Robin static algorithms are both effective in terms of speed, however only Round Robin is considered fault-tolerant. The static load balancing algorithm leverages randomness for simplicity. While it aims for even distribution across servers in the long run, its limitations lie in potential inefficiency with varying server capacities, occasional creation of temporary hotspots, and the lack of dynamic adaptation to individual server health or workload [37].

Dynamic load-balancing algorithms on the other hand are adaptive to changes in the number of active users. The disadvantage of these algorithms is the difficulty of their implementation and complexity. The complexity of all algorithms is quadratic, thus, we have to choose the one with reliability and fault-tolerance. The Least Connection algorithm has an implemented fault-tolerance, which ensures the system will continue to operate even with the occurrence of failure. This fully satisfies the requirement we have set for choosing a load-balancing algorithm.

1.3 Rollback-recovery with checkpointing fault-tolerance algorithm

Rollback-recovery with checkpointing is one of the most widely used fault-tolerance techniques in distributed and high-performance computing systems. Its core idea is to periodically save the state of an application (checkpoint) so that, in the event of a failure, the system can roll back to the most recent consistent state instead of restarting execution from the beginning. This approach significantly reduces computation loss and improves system reliability, especially in long-running and large-scale applications [38].

The idea of *checkpointing* is to periodically store the state of a computation called checkpoints in stable storage. Checkpoints are established while system is running without failure. This saved state can be used to restart the process from the last

checkpointing state, rather than completely restarting the process, which would involve lengthy, repetitive processing and repeated output [38, 39].

Figure 1.4 shows how checkpointing works: during the processing of information, the current state of progress is saved in store with a particular interval. These saved states are called “checkpoints”.



Figure 1.4 - Checkpointing during process execution

In this method, the execution timeline is divided into intervals separated by checkpoints. During normal operation, the system performs computation and periodically writes checkpoint data to stable storage. When a failure occurs, the system initiates a recovery phase, restoring the last saved checkpoint and re-executing only the lost portion of computation. The total execution time can therefore be represented as a combination of useful computation time, checkpoint overhead, recovery time, and rework due to failures. The efficiency of the algorithm largely depends on selecting an optimal checkpoint interval that balances checkpointing cost and expected rollback loss.

Checkpointing strategies can be classified into static and dynamic approaches. Static checkpointing uses fixed intervals determined analytically or empirically, often based on system parameters such as mean time to failure (MTTF) and checkpoint overhead. While simple to implement, static methods may not adapt well to changing system conditions. In contrast, dynamic checkpointing adjusts checkpoint intervals at runtime based on observed failure behavior, workload characteristics, or predictive models. This adaptability allows dynamic approaches to achieve better performance in environments with variable failure rates.

Rollback-recovery mechanisms also differ in their coordination schemes. Coordinated checkpointing ensures that all processes save a globally consistent state, simplifying recovery but introducing synchronization overhead. Uncoordinated checkpointing, on the other hand, allows processes to checkpoint independently, reducing coordination costs but potentially leading to the domino effect, where cascading rollbacks occur. Hybrid approaches attempt to combine the benefits of both methods [40].

The main advantages of rollback-recovery with checkpointing include simplicity, general applicability, and effectiveness in reducing recomputation after failures. It is particularly suitable for systems where failures are relatively infrequent but costly.

The main issue of checkpointing method is to find the optimal interval for taking checkpoints. Because, *writing data to both storage types at the same time is expensive*. That is why checkpointing too often will increase overhead and response time during uptime periods and will require large storage capacity. On the other hand, if the

checkpointing is performed rarely, the rollback duration must be long, which subsequently increases the duration of service unavailability after a failure. Thus, the effectiveness of the system's fault tolerance will decrease.

Numerous studies have proposed static, dynamic checkpointing methods to balance these trade-offs, taking into account system behavior, failure patterns, and computational costs. This section reviews key contributions in this field, highlighting their approaches, advantages, and limitations, and sets the stage for evaluating how our proposed solutions improve upon existing techniques.

The article by Tiwari et. al. [38] proposes adaptive checkpointing techniques to reduce overhead in large-scale computing systems by exploiting the fact that failures tend to occur in clusters rather than randomly. It introduces methods like Lazy Checkpointing, which dynamically adjusts checkpoint intervals based on recent failures, improving efficiency compared to fixed periodic approaches. The main advantage is reduced I/O overhead and better performance, while the main drawback is increased complexity and reliance on specific failure patterns. Overall, the work demonstrates that understanding real failure behavior can significantly optimize fault-tolerance strategies.

Frank et. al. [39] investigates advanced checkpointing strategies for improving fault tolerance in large-scale computing systems, focusing on reducing performance overhead caused by frequent state saving. It proposes optimized or adaptive checkpointing methods that better align with system behavior and failure patterns, leading to more efficient execution compared to traditional periodic approaches. The main advantages include improved performance, reduced I/O costs, and better scalability, while the disadvantages involve increased system complexity and dependence on accurate modeling of failures. Overall, the work highlights the importance of intelligent checkpoint scheduling in enhancing reliability and efficiency in high-performance computing environments.

The work by John T. Daly. [40] develops a higher-order analytical model to determine the optimal checkpoint interval that minimizes total execution time in systems with failures, improving upon earlier first-order approximations. It introduces a more accurate cost function that accounts for restart overhead, failure timing, and multiple failures, and derives both exact and approximate solutions using advanced mathematical methods. The main advantage is significantly improved accuracy (with very low error compared to real execution), while the disadvantage is increased mathematical complexity and reduced simplicity for practical use. Overall, the work provides a more precise and reliable foundation for optimizing checkpointing in high-performance computing systems.

The reviewed pap**e**rs are compared with our proposed solutions in terms of their use of dynamic or static checkpointing approaches, as well as whether they were evaluated in simulation or real-world environments (see Table 1.3).

Table 1.3 – different solutions for computing optimal checkpointing interval for rollback-recovery algorithm

Work	Dynamic checkpointing	Static checkpointing	Tested in simulation	Tested in real-world environment
Tiwari et. al.	+		+	
Frank et. al.		+	+	
John et. al.		+	+	
Current work: Model based solution for computing checkpointing interval	+		+	+
Current work: Optimal checkpointing based on queuing theory	+		+	+

The Table 1.3 shows a clear distinction between existing studies and the proposed work in terms of checkpointing approaches and validation methods. First, prior works demonstrate different focuses: Tiwari et al. employ dynamic checkpointing but validate only through simulation, while Frank et al. and John et al. rely on static checkpointing and are also limited to simulation-based evaluation. None of the reviewed studies include testing in real applications, indicating a gap between theoretical development and practical validation.

In contrast, the current work combines dynamic checkpointing with both simulation and real-world testing. This represents a significant advancement, as it not only adopts a more adaptive approach but also validates its effectiveness in practical environments. Additionally, the inclusion of a queuing-theory-based model further strengthens the methodological contribution, suggesting improved realism and applicability compared to existing solutions.

In conclusion, existing studies either focus on optimizing checkpointing techniques, enhancing fault tolerance in specific environments, or developing theoretical recovery models. However, there is a lack of integrated approaches that combine analytical optimization, practical system design, and scalability considerations within a single framework. Limited attention has been given to balancing checkpointing overhead with recovery efficiency while ensuring system scalability and resilience in enterprise digitalization contexts. This research addresses this gap by proposing a unified framework that integrates an optimal checkpointing model with a scalable and fault-tolerant system architecture.

1.4 Architecture of enterprise distributed systems

Enterprise systems typically have physical servers located either locally or in the cloud by providers (AWS, Google Cloud, etc.). Each of these servers have multiple applications (in this work we call them “server applications”), each of which hosts a set of M information processing task types (e.g., authentication, data modification, etc.). Basic types, special types, multiple instances of a given task type. Task instances are in general called “server applications”, abbreviated as SA.

Figure 1.5 describes an Enterprise system, which consists of N, N>1, physical servers; each server has number of server applications (SA). The system has one database (DB) for storing/retrieving data. Figure 1 shows how the system works with client requests in a crash free scenario: clients send a HTTP request to any server in system.

In Figure 1.5, $Server_1$ receives a request from $Client_1$, makes queries to database, receives results, processes information, stores result in the database and communicates with $Server_2$; $Server_2$ further processes information and updates database, sends result to $Server_3$; $Server_3$ adds modifications to the result and updates database, sends response back to the client. Note that any server might send results back to previous servers for additional processing, thus creating cycles (e.g. $Server_3$ might send result to $Server_2$). Figure 1.5 also shows that some processes may require only one server. For example, $Client_2$ sends request to $Server_N$, which processes the data, updates database and sends response to the client.

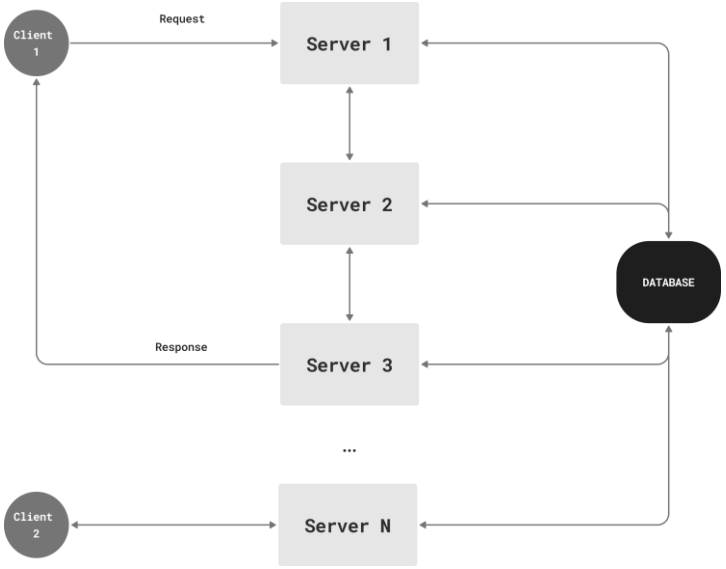


Figure 1.5 - Enterprise system architecture

As a prototype process for implementing and validating purposed ideas we use *document processing workflow* that is a part of our enterprise information system (see Appendix A). The workflow of document processing can be seen in Figure 1.6. Client request lands at $Server_1$, where SA_1 executes the request. The results and the request are passed on to $Server_2$ where two applications SA_2 and SA_3 process it in that order.

The output from SA_3 is sent to SA_4 in $Sersver_3$ and then from SA_4 to SA_5 in $Server_1$, which sends out the response to the Client. This particular workflow involves a total of 5 SAs distributed on three servers.

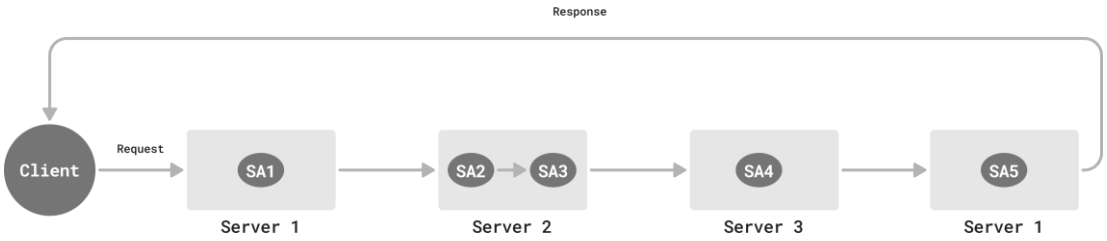


Figure 1.6 - Workflow of document processing

To reduce delays in the processing of requests it is important to develop appropriate crash-tolerant methods that do not significantly reduce performance in the absence of crashes, that is, the costs of ensuring crash tolerance should be minimal by design [43]. To achieve that we utilize *rollback-recovery with checkpointing* algorithm.

The idea of recovery is straightforward: when SA crashes, the available SA must recover the data of the crashed one and continue the process from the last checkpoint.

We follow the *dual redundant strategy* of using both the local disk and a remote database to store checkpoints. Figure 1.7 shows how the proposed strategy works: each physical server has Checkpointing Service that is designed to periodically save checkpoints of all SAs of the server into both local disk and remote database.

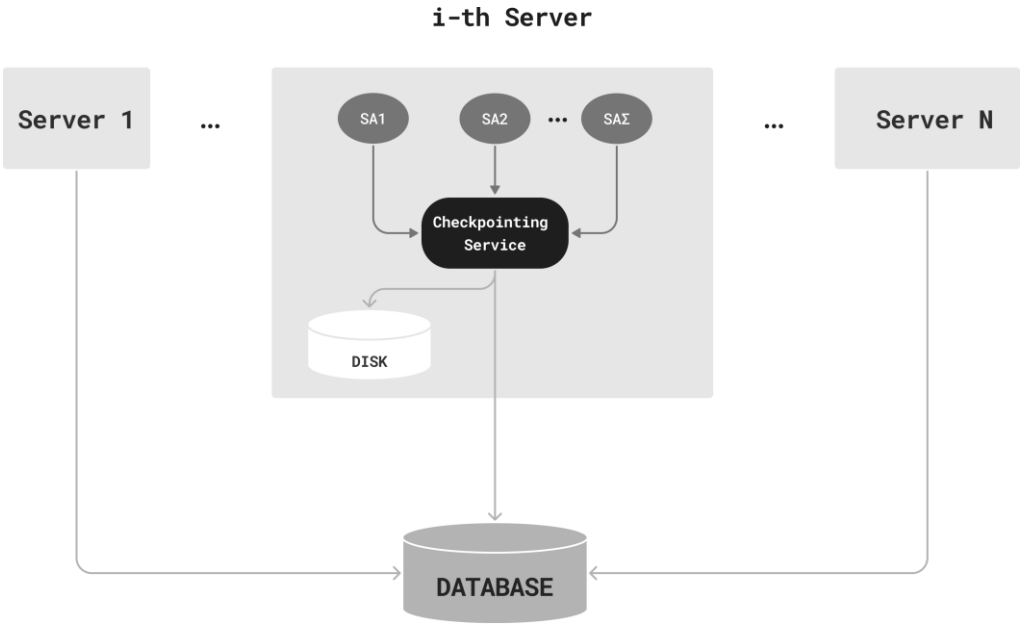


Figure 1.7 - Dual redundant strategy for checkpointing states

To solve the SA crash scenario, we use *checkpointing-recovery* method described above. When a server application crashes, available SA recovers the crashed one's state using last checkpoint that is stored in the server disk (see Figure 5). This way, the recovered SA will not start the process from beginning, but from the last

checkpoint which was taken before SA crashed.

For the second crash type that assumes the physical server crash, we add backup server for each server. This way, when primary server crashes, backup server has to recover all the SAs of primary server using last checkpoint saved in a remote database (see Figure 1.7).

Figure 1.8 describes an example of how this solution is applied in a proposed workflow in our system. When $Server_2$ receives a request from $Server_1$, it crashes. As soon as it does, its backup server picks up the last checkpoint from remote database and recovers $Server_2$ with all the SAs (SA_2 and SA_3 are recovered using respective checkpoints). After processing the request, it sends result further to $Server_3$, thus continuing the workflow.

We assume that both the main server and backup server do not crash. At most one of them will be operative at any given time.

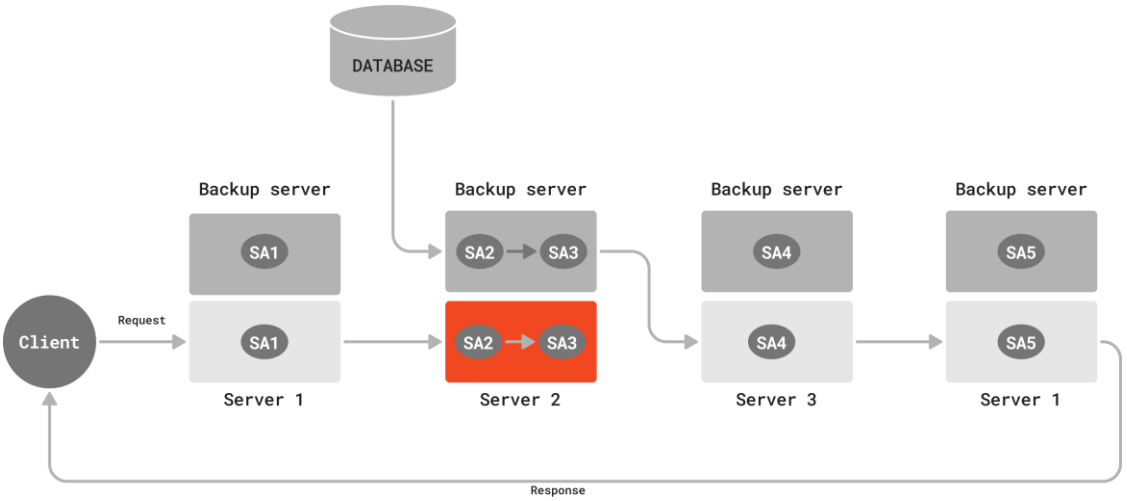


Figure 1.8 - Checkpointing-recovery approach when server crashes

One of the issues of the rollback-recovery method is the way of storing checkpointed process states. The main requirement is using reliable and stable storage methods.

One option is the use of server disk storage. Using a server disk as a store provides greater control, accessibility, cost-effectiveness, and customization for an organization. A server disk can be accessed from within the organization's network, providing fast and reliable access. However, it also has limitations, including limited accessibility, vulnerability, maintenance requirements, and scalability issues. Server disks can be vulnerable to hardware failures or damage, natural disasters, and security breaches, which can result in data loss or corruption.

Another option is storing state in a remote database. Remote databases are usually equipped with backup and recovery options, which provide an extra layer of security. Remote databases can be accessed from anywhere with an internet connection, allowing access to state of the process even if the servers are down. However, remote databases can be expensive to set up and maintain, especially if they require a high level of security or if large amounts of data are being stored.

Storing checkpointed state on a local (physical) server disk is effective and useful if one of the server applications crashes, the process could be recovered by

retrieving checkpoint data from the local disk. However, if the physical server itself crashes then the disk becomes inaccessible. Thus, there is a need to migrate all server applications hosted by crashed server to a new physical server using the recovery data. For this purpose, the checkpointing data is also stored in the remote database. Thus, we follow the *dual redundant strategy* of using both the local disk and a remote database (see Figure 1.7).

1.5 Failures and Assumptions

Let's consider different failure assumptions that might occur in an individual server of the proposed system when interacting with client(s). We assume the following two failure types:

- server application failure within an operative server;
- crash of a server itself.

We assume that modern databases are built with high standards, reliability and backup stores. That is why in this work we do not focus on database failures.

1.5.1 Server applications crash

SA failure can be of two types: either SA does not respond (freezes) even though it is utilizing resources, or it simply crashes. *Any number of SAs can crash at any time*. These failures can be due to various software errors: segmentation fault, buffer overflow, null reference errors, memory leaks, unhandled exceptions, stack overflow, deadlocks, resource exhaustion, etc. [41, 42]. SA crashes are costly and lead to performance issues as they *prolong the execution of requests*.

Figure 3 shows an example of this failure type: one server with 6 server applications = {SA_1, SA_2, SA_3, SA_4, SA_5, SA_6}, among these: SA_1 and SA_5 are crashed and unavailable. Note that some SAs send results to each other, particularly SA_2 to SA_3, SA_3 to SA_6, SA_4 to SA_5.

Let’s look at failed SAs that are painted in red color in Figure 1.9. As SA_5 failed, SA_4 has to wait for the operational system to recover it. When it recovers, the process must *restart* in a new SA, because the progress of SA_5 was lost. Same happens with SA_1, any other available SA will pick up its request and restart the process. This adds delays to request processing, thus clients have to wait longer times.

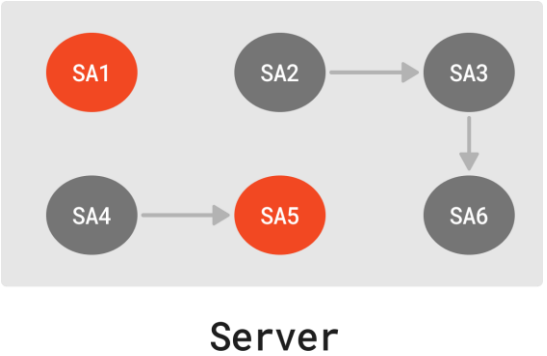


Figure 1.9 - SA(s) failure assumption

1.5.2 Physical server crash

Another failure can occur when physical servers crash. *Note that any number of servers can crash at any time*. This failure is primarily caused by hardware issues such as network connection, power loss, etc. This crash leads to failure of all server applications in the crashed server. Upon this failure, the crashed server(s) is replaced, and all the SAs hosted by the server(s) are restarted which will drastically increase delay.

In Figure 1.6, we described how our system processes document workflow. Let σ denoted the total number of SAs involved in processing given client request, and n denote the total number of servers that host all these σ SAs. Note that (σ, n) may also depend on to which server the request was initially sent to. In Figure 2, σ=5, n=3; (σ=3, n=3) and (σ=1, n=1) hold for $Request_1$ and $Request_2$ respectively. Note that σ can be large and 1≤n≤minimum {σ, n}.

Let R_{SA} be the reliability of any SA, i.e., the probability that the SA will not fail while executing a task in that workflow. Note that $R_{SA} < 1$.

A workflow with (σ, n) will fail at one SA failure with probability 1 - R_{SA}^{σ}.

That is, a workflow with (σ, n) will experience additional delay due to SA failure(s) with probability:

$$1 - R_{SA}^{\sigma}$$

Note that the above probability approaches 1 as σ increases.

Similarly, a workflow with (σ, n) will experience long extra delays due to Server crash with probability:

$$1 - R_s^n$$

where R_s is the probability that a server will not fail during the workflow execution.

This chapter introduced the situation around the reliable information processing in distributed enterprise systems. It also covered existing fault-tolerance and load-balancing methods including their advantages and disadvantages. At the end of the chapter, we described different crash scenarios in distributed systems and identified research problems.

In conclusion, during literature review, we analyzed different fault-tolerance algorithms, and concluded that Rollback-Recovery with checkpointing provides highest reliability and performance. This algorithm was further studied and different modifications of the algorithm were analyzed. During this analysis, we identified following drawbacks of existing solutions: lack of integrated approaches that combine analytical optimization, practical system design, and scalability considerations within a single framework; limited attention to balancing checkpointing overhead with recovery efficiency while ensuring system resilience in enterprise digitalization

contexts. Next chapters will focus on our solutions that tries to eliminate these drawbacks and improve this algorithm by calculating optimal value of checkpointing interval.

2. DEVELOPMENT AND DESIGNING ALGORITHMS OF ROLLBACK-RECOVERY BASED ON CHECKPOINTING

In this chapter, we describe the development of new algorithms and methods for fault-toleance algorithms based on checkpointing, specifically: calculating optimal value for checkpointing. We propose two solutions for this issue: empirical and queuing theory-based algorithms.

The main issue of checkpointing method is to find the optimal interval for taking checkpoints. Because, *writing data to both storage types at the same time is expensive*. That is why checkpointing too often will increase overhead and response time during uptime periods and will require large storage capacity. On the other hand, if the checkpointing is performed rarely, the rollback duration must be long, which subsequently increases the duration of service unavailability after a failure. Thus, the effectiveness of the system's fault tolerance will decrease.

2.1 Development of model based algorithm for calculating optimal checkpointing interval

To achieve consistency of data processed during process execution, we use a rollback-recovery algorithm. This algorithm mainly works using “checkpoints”.

The idea of a checkpointing is to periodically store the state of a computation in stable storage [44 - 47]. Checkpoints are established while the process is running without failure. This saved state can be used to restart the process, rather than completely restarting the process, which would involve lengthy, repetitive processing and repeated output.

The main question here will be to find the optimal interval for taking these control points. Because taking checkpoints too often will increase overhead and response time during uptime periods and will require larger storage capacity. On the other hand, if the checkpointing is performed less frequently, the rollback duration must be long, which subsequently increases the duration of service unavailability after a failure. Thus, the effectiveness of the system's fault tolerance will decrease.

The ideal way to calculate the optimal check interval is to estimate it by simulating the system along with all relevant parameters, which include the request arrival rate, server processing speed, server failure probability, etc. Such modeling work has been extensively done by many authors [47 - 51]. This ideal way of determining the optimal interval is not feasible when we are prototyping a proof-of-concept system because the corresponding values for all model parameters are unknown. Therefore, we decided to estimate the checkpointing interval heuristically [52], as described below.

Intuitively, the frequency of checkpointing will depend on the average execution of the process as well as on the frequency of the CPU. This is because: more processor power means shorter execution times. This is derived from fundamental computer architecture and performance modeling equations by John L. Hennessy et. al. [53].

Equation 1.

The equation by John L. Hennessy et. al. describes how total execution time is

inversely proportional to the CPU frequency, if we assume the number of instructions and cycles per instruction (CPI) remain constant.

$$T = \frac{I*CPI}{f} \tag{2.1}$$

Where:
T = total execution time
I = number of instructions
CPI = cycles per instruction
f = CPU frequency (gigahertz)

This was based on fundamental principles established by computer architects and engineers over decades, including Amdahl's Law [54] and pipelining concepts in modern CPUs. Using this model, we propose a solution for calculating checkpointing interval that is based on CPU frequency and mean execution time.

Definition 1. T_c is the time in seconds between two checkpoints during a fail-free computational process.

Definition 2. The mean execution time T_p refers to the average time required to complete a given computational process on a single-threaded CPU. It is calculated as the difference between the total area of the cumulative distribution function (CDF) and the area under the CDF.

Definition 3. CPU frequency - f defines how many cycles per second the CPU can execute in gigahertz. This parameter varies depending on CPU of each individual server.

Equation 2.

We use the following equation (2.2) to determine the checkpointing interval (T_c):

$$T_c = \frac{T_p}{f} \tag{2.2}$$

Where:
T_p= mean execution time on a single threaded CPU (seconds)
f = CPU frequency (gigahertz)

The higher CPU frequency the lower checkpointing interval value. Subsequently, lower checkpointing interval means more checkpoints can be established during processing with minimal impact on performance [55].

Equation 3.

The area under the CDF is computed using the Trapezoidal Rule:

$$\text{area under CDF} = \sum_{i=1}^{N-1}(x_{i+1} - x_i) * \frac{(y_i + y_{i+1})}{2} \tag{2.3}$$

Where:
h_i = i-th height of trapezoid
x_i = i-th width of trapezoid

N = total number of trapezoids

This method ensures a fast and accurate approximation of execution time probabilities over time. CDF will also be used to evaluate the effectiveness of checkpointing in the next sections.

Equation 4.

T_p is calculated using the following equation:

$$T_p = \text{total area CDF - area under CDF} \tag{2.4}$$

Where:

Total area CDF represents the longest execution time.

The area under the CDF is the result of equation (2.3).

Using equations (2.2 - 2.4) we can construct psudocode of the alogirthm which can be used to calculate checkpointing interval:

Input: Response time sample X={$x_1, x_2, \ldots, x_n$}; constant f
Compute empirical probabilities using rank normalization:
  Construct data frame from (X)
  Rank elements using maximum-rank method
  Normalize ranks: $p_i = \frac{rank(x_i)}{n}$
  Store probability array P = {$p_1, p_2, \ldots p_n$*}*
Initialize cumulative area under CDF: Area_CDF = 0
For (i = 1) to (n-1) do
  Compute height: $\Delta x = x_{i+1} - x_i$
  Compute average base: $\Delta p = \frac{p_i + p_{i+1}}{2}$
  Compute trapezoidal area: $A_i = \Delta p \cdot \Delta x$
  Update cumulative area: Area_CDF = Area_CDF + A_i
End For
Determine total CDF area bound: Total_Area_CDF = x_n
Compute mean response time: Tp=Total_Area_CDF−Area_CDF
Compute optimal checkpointing interval: $I = \frac{T_p}{f}$
Return I

The block-schema of the algorithm can be viewed in Figure 2.1. The full program code can be seen in Appedix A.

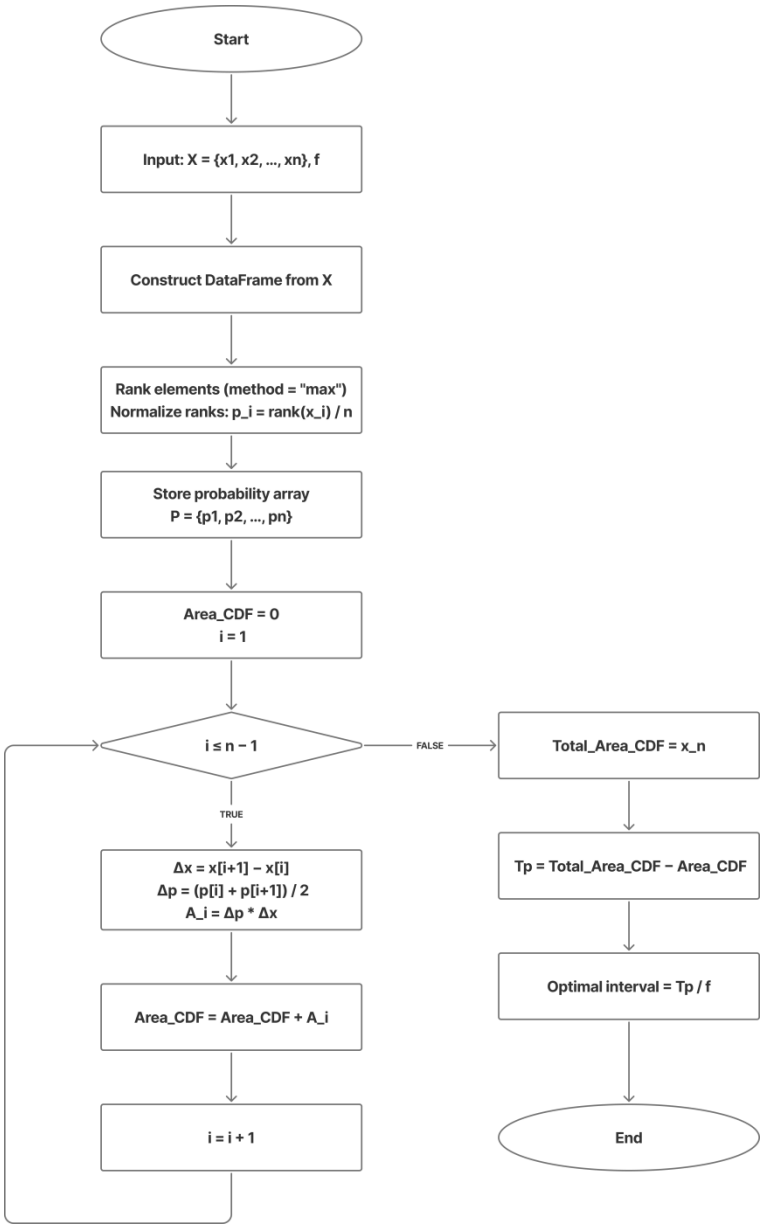


Figure 2.1 – Block scheme of algorithm 1

Overall, this solution is aimed at providing basic checkpointing implementation for applications that are less prone to long term breaks as the solution is not theoretically optimal. However, this should be enough for small to medium-sized enterprises with limited technical capabilities and specialists.

2.2 Development of an algorithm for calculating optimal checkpointing interval based on queuing theory

In this section, the optimal checkpointing interval is calculated using model from recent research [55]. Table 2.1 lists all the necessary parameters needed to calculate optimal checkpointing interval using this model [56 – 58].

Table 2.1 - Parameters for calculating optimal checkpointing interval

Parameter name	Symbol
phase	k
arrival rate	λ
number of service phases	K
probability of phase k	q
average service requirement in phase k	1/μ
average back-up interval	1/β
breakdown rate	ξ
average repair (replacement) interval	1/η
average service interval in phase k	1/ν
average interruptable back-up interval	1/σ

The optimal interval value that is computed using this model will be submitted to each Checkpointing Service, which will then utilize it for checkpointing. We computed each parameter using experiments and the values are presented in the next sections.

The first step is to calculate the necessary parameters for the checkpointing model. In Table 2.2, we make assumptions before calculating other parameters.

Table 2.2 - Assumptions and measurements of parameter values

Parameter name	Definition	Value
phase	k	1
arrival rate	λ	to be decided
number of service phases	K	1 (since we assume k = 1)
probability of phase k	q	1 (since we assume k = 1)
average service requirement in phase k	1/μ	to be calculated
average back-up interval	1/β	to be calculated
breakdown rates	ξ	0.001, 0.01, 0.1
average repair (replacement) interval	1/η	0 (we assume the repair is instantaneous)
average service interval in phase k	1/ν	to be calculated
average interruptable back-up interval - 1/σ;	1/σ	to be calculated

In order to find an optimal checkpointing interval, we have to plot different

values of α against load (L). For that, the parameters described below are calculated (2.5 – 2.14).

Jobs arrive to the system according to a Poisson process with intensity λ. Any back-up operation follows an exponential distribution (2.5).

$$\sigma = \beta + \xi \tag{2.5}$$

Any service interval is exponentially distributed:

$$\nu = \mu + \alpha + \xi \tag{2.6}$$

Calculate expected values (first and second moments of a random variable V related to service interval) using equations (2.7 – 2.8). V is defined as the time interval from the moment of failure (breakdown) to the restart of the service:

$$E(V) = \frac{1}{\nu} \tag{2.7}$$

$$E(V^2) = \frac{1}{\nu^2} \tag{2.8}$$

Calculate expected time considering backup intervals and breakdowns (2.9 – 2.10). Y is the time interval from the start of the attempt to create a checkpoint (checkpoint) to the resumption of the service.

$$E(Y) = \frac{1}{\sigma}(1 + \beta * \mathrm{E}[\mathrm{V}]) \tag{2.9}$$

Calculate variance:

$$E(Y^2) = \frac{1}{\sigma^2}(1 + \beta * \mathrm{E}[\mathrm{V}]) + \frac{\beta}{\sigma}\mathrm{E}[\mathrm{V}^2] \tag{2.10}$$

Similarly, calculate expected service time and second moment of the service time (2.11 – 2.12). X is a random variable that represents the full time from the beginning of the service of the task to its completion.

$$E(X) = \frac{1}{\mu} * \frac{\alpha + \sigma}{\sigma} * \frac{\nu + \beta}{\nu} \tag{2.11}$$

$$E[X^2] = 2 * E[X]^2 + \frac{1}{\mu}(1 + \alpha * \mathrm{E}[Y^2] + \beta * \mathrm{E}[\mathrm{V}^2]) \tag{2.12}$$

The system is considered to be stable if the following is true:

$$\lambda * \mathrm{E}[\mathrm{X}] < 1 \tag{2.13}$$

The condition ensures that the long-term average workload imposed on the

system is less than its processing capacity. In other words, the server is able to handle incoming jobs faster than they arrive on average.

If the stability condition is satisfied, the load L is calculated using Pollaczek–Khinchine formula (2.14).

$$L = \lambda * \mathrm{E}[\mathrm{X}] + \frac{\lambda^2 * E[X^2]}{2(1-\lambda * \mathrm{E}[\mathrm{X}])} \quad (2.14)$$

This equation (2.14) gives the expected number of jobs in the system (both in service and waiting), derived from the Pollaczek–Khinchine formula for the M/G/1 queue.

The description of the proposed algorithm for finding optimal checkpointing interval can be viewed bellow in the pseudocode or in Figure 2.2 in the form of block scheme. The full program code can be seen in Appedix A.

Algorithm 1: To find the checkpoint interval *I* for which the system load *L* L is minimal, we compute all values and select the smallest valid one (ignoring unstable cases).

Generate a set of candidate intervals: Intervals = Linspace(I_min, I_max, N)
Initialize minimal load: Minimal_Load = +∞
Initialize optimal interval: Optimal_Interval = null
Initialize empty list of loads: Loads = []
For each I in Intervals do:
  Compute checkpointing rate: α = 1 / I
  Compute system load: L = Calculate_Load(λ, μ, α, avg_backup_interval, β)
  Append load to list: Append L to Loads
  If L is defined AND L < Minimal_Load then:
    Update minimal load: Minimal_Load = L
    Update optimal interval: Optimal_Interval = I
  End If
End For
Return Optimal_Interval

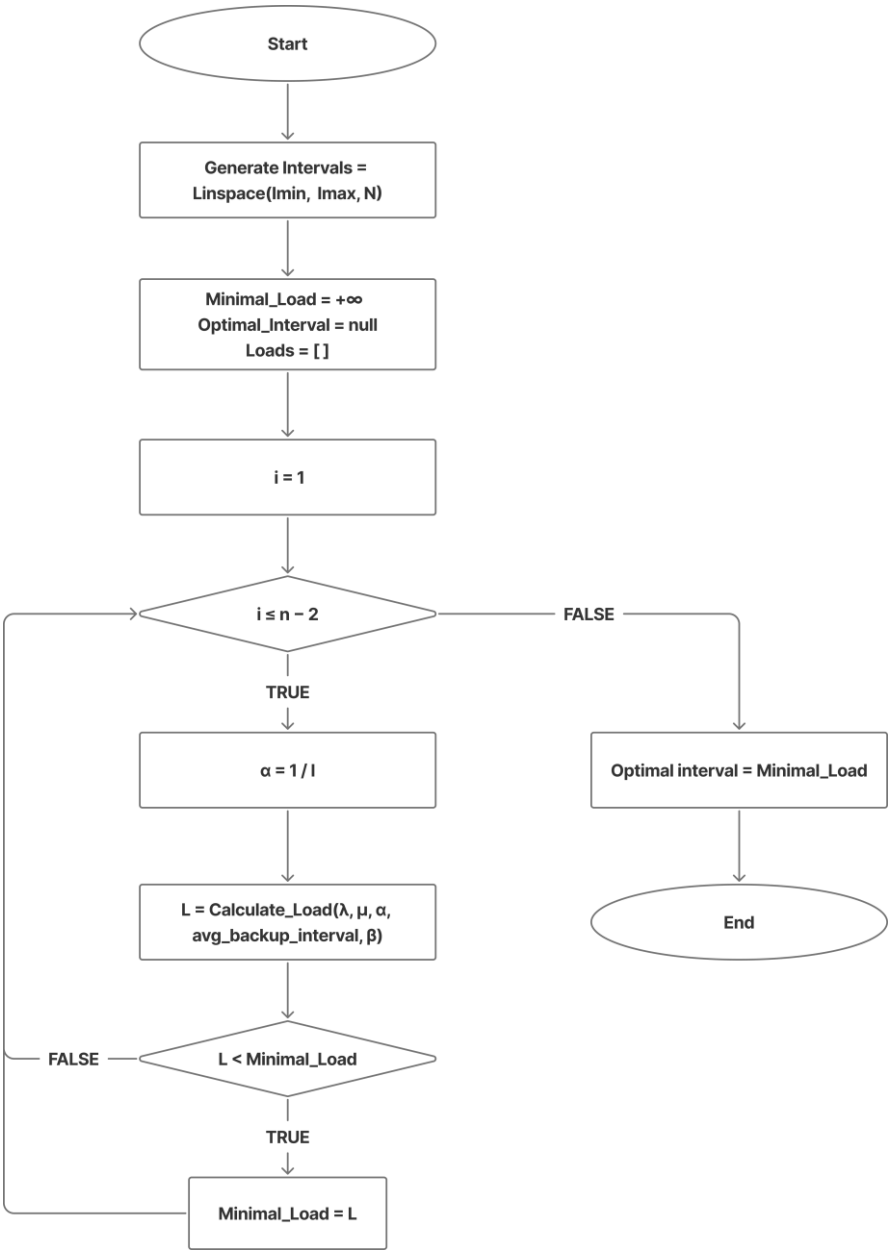


Figure 2.2 - Finding optimal checkpointing interval block scheme (Algorithm 2)

3. IMPLEMENTATION AND TESTING OF CHECKPOINTING BASED ALGORITHMS IN ENTERPRISE DISTRIBUTED SYSTEMS

This chapter illustrates the evaluation and implementation of the developed algorithms in both simulated and real-world environments.

In order to ensure high reliability and availability of enterprise distributed systems, checkpointing-based rollback-recovery algorithms were implemented and tested. These algorithms are designed to minimize data loss and maintain system consistency in the presence of failures, including hardware crashes, network disruptions, and software exceptions. The implementation focuses on both static and dynamic checkpointing strategies to balance performance overhead and fault-tolerance guarantees.

The implementation of the developed algorithms was carried out on a simulated enterprise environment and subsequently deployed in a controlled production-like setup. Checkpointing was integrated at multiple levels: application layer, service orchestration layer, and data storage layer. Dynamic checkpointing intervals were adjusted based on system workload and observed failure rates, while static checkpointing was used as a baseline for performance comparison.

3.1 System architecture

Figure 3.1 illustrates the proposed fault-tolerant architecture composed of five main components: the application layer, checkpoint manager, storage layer, monitoring service, and recovery orchestrator. Applications periodically save their states through the checkpoint manager, which handles checkpointing and rollback control. The storage layer ensures reliable persistence using both local and remote storage. The monitoring service tracks performance metrics and detects failures, while the recovery orchestrator coordinates system restoration based on the latest consistent checkpoint. This integrated design enables efficient recovery with minimal overhead and supports scalability in distributed enterprise environments.

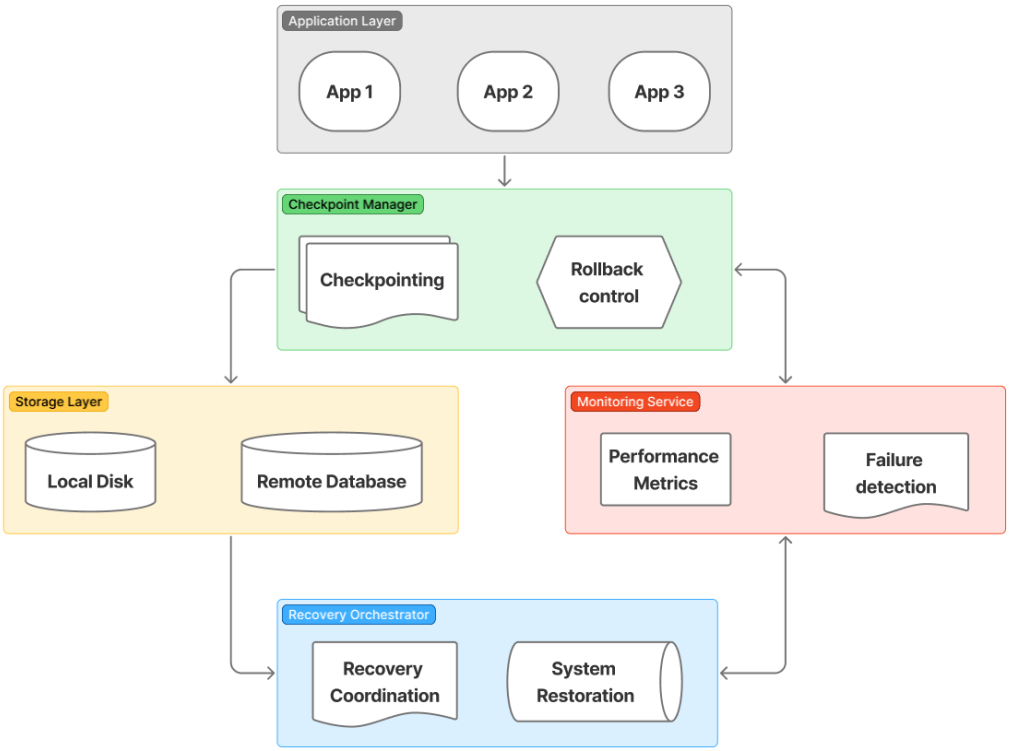


Figure 3.1 - Architecture diagram of fault-tolerant enterprise system

3.2 System model

The proposed fault-tolerant framework operates in a distributed environment and is defined through the following components: nodes, processes, failure model, recovery model, and consistency model.

The system consists of a set of physical servers: $N = \{S_1, S_2,...,S_n\}$, $n>1$, where each server represents a computational node capable of processing client requests. Each node hosts multiple server applications (SAs) and is connected to a shared database and remote storage. For fault tolerance, each primary server is associated with a backup server that can take over in case of failure.

Processes in the system are modeled as server applications (SAs) executing tasks within enterprise workflows. Let $P = \{SA_1, SA_2,...,SA_\sigma\}$ denote the set of SAs involved in processing a client request, where σ is the total number of applications participating in a workflow. These processes may be distributed across multiple servers and can interact sequentially or cyclically, as shown in the document-processing workflow. Each process periodically saves its execution state using checkpointing.

The reliability of a process is defined as R_{SA}, representing the probability that the SA completes execution without failure.

The system considers two primary types of failures:

1. Server Application (SA) Failure. Individual processes may crash or become unresponsive due to software-related issues (e.g., memory leaks, exceptions). Such failures affect only specific components of the workflow but may propagate delays.

2. Physical Server Failure. A complete server crash results in the failure of all SAs hosted on that server. This type of failure is typically caused by hardware faults, power loss, or network issues. The reliability of a server is denoted as R_s. The

probability of workflow disruption increases with the number of participating processes m, highlighting the need for efficient fault-tolerance mechanisms. The model assumes a fail-stop behavior, where failed components cease operation without producing incorrect outputs.

The system adopts a checkpointing-based rollback-recovery model. Each SA periodically saves its state at an optimal checkpointing interval $1/\alpha$. Recovery is performed as follows:

In case of SA failure, another available SA retrieves the latest checkpoint from the local disk and resumes execution. In case of server failure, a backup server retrieves checkpoints from the remote database and restores all affected SAs. The dual-redundant storage strategy (local + remote) ensures both fast recovery and resilience against server crashes.

The system ensures checkpoint consistency to enable correct recovery without loss of computation or data corruption. A global state is considered consistent if all checkpoints correspond to a causally valid execution without missing or orphan messages. Given the enterprise workflow characteristics, the model employs:

1. Coordinated checkpointing within workflows, ensuring that dependent SAs maintain consistent states;

2. Deterministic recovery, where processes resume from the latest valid checkpoint without violating execution order.

This approach prevents the domino effect and guarantees that the system can safely resume operation after failures.

3.3 Performance evaluation setup

To evaluate the effectiveness of the proposed checkpointing-based fault-tolerant framework, a structured experimental methodology was designed. The evaluation focuses on quantifying system performance under failure-prone conditions using a set of well-defined metrics.

Number of checkpoints

This metric represents the total number of checkpoint states created by all distributed nodes during the experiment. It reflects the frequency of state-saving operations and is directly related to checkpointing overhead and recovery granularity. A higher number of checkpoints may improve recovery accuracy but increases storage and I/O costs.

Throughput

Throughput measures the number of jobs successfully processed by the system per second. It is a key indicator of overall system performance under the given workload (arrival rate of 0.3 jobs/sec) and reflects the efficiency of task distribution and execution across the nodes.

Average service time

This metric denotes the average time required to complete a single job. It includes processing time as well as any delays caused by failures and recovery operations. It is critical for evaluating the impact of rollback-recovery mechanisms on system responsiveness.

System Availability

System availability measures the proportion of time the system remains operational, as in (21).

$$\text{Availability} = \frac{T_{uptime}}{T_{uptime} + T_{downtime}} * 100\ \% \quad (3.1)$$

where, downtime includes both failure detection and recovery periods. This metric reflects the overall reliability of the system.

Efficiency

Efficiency measures how effectively system resources (CPU, memory, energy, etc.) are utilized for productive computation. It is defined as:

$$Efficiency = \frac{Useful\ time}{Total\ time} * 100\% \quad (3.2)$$

where "Useful Time" corresponds to the time spent performing actual job processing, excluding overhead such as checkpointing and recovery. This metric is essential for understanding the trade-off between fault tolerance and resource utilization.

Together, these metrics provide a comprehensive evaluation framework for analyzing the performance, fault tolerance, and operational efficiency of rollback-recovery algorithms in the proposed distributed enterprise system.

To ensure a comprehensive and realistic evaluation of the proposed framework, experiments are conducted using three categories of benchmark workloads that reflect typical enterprise digitalization scenarios.

Synthetic workloads are generated to simulate controlled enterprise processing conditions. These workloads consist of sequences of tasks representing common operations such as authentication, data processing, and document generation.

Each workload is parameterized by:

— Number of nodes σ

— Arrival rate λ

— Failure rate ξ

This benchmark allows controlled variation of system parameters and enables systematic evaluation of checkpointing interval optimization. It is particularly useful for analyzing system scalability and sensitivity to increasing workload size and failure frequency.

3.4 Experiment 1: Implementation and Testing of Rollback-Recovery Algorithms Using Virtual Machines in Real Distributed Enterprise Systems

In this experiment, rollback-recovery algorithms based on checkpointing mechanisms were implemented and evaluated within a realistic distributed enterprise environment. The experimental infrastructure was deployed using multiple virtual machines instantiated as containerized services. These containers were orchestrated and interconnected using Docker Compose, enabling the simulation of a scalable and

fault-prone distributed system architecture. The resource allocation of the containers (nodes) in a tested computer are as follows:

1) Node 1: RAM: 512 Mb; GPU cores: 1;
2) Node 2: RAM: 512 Mb; GPU cores: 1;
3) Node 3: RAM: 512 Mb; GPU cores: 1;
4) Node 4: RAM: 512 Mb; GPU cores: 1;
5) Node 5: RAM: 512 Mb; GPU cores: 1;
6) Backup Node: RAM: 512 Mb; GPU cores: 1;
7) Ngnix: RAM: 256 Mb; CPU cores: 0.5;

The workload was generated with an average arrival rate (λ) of 0.3 jobs per second, representing a moderate but continuous processing demand typical for enterprise systems. The total duration of the experiment was set to 10 hours to ensure sufficient observation time for system behavior under sustained operational conditions.

To assess fault tolerance capabilities, failures were artificially introduced into the system at a failure rate (ξ) of 0.1 failures per second. These induced faults emulate real-world instability scenarios, such as hardware malfunctions, network disruptions, or software crashes. The rollback-recovery algorithms were evaluated based on their ability to maintain system consistency, minimize data loss, and ensure rapid recovery under these failure conditions.

The experimental setup allows for comprehensive analysis of system performance, recovery latency, checkpointing overhead, and overall reliability in a controlled yet realistic distributed enterprise environment.

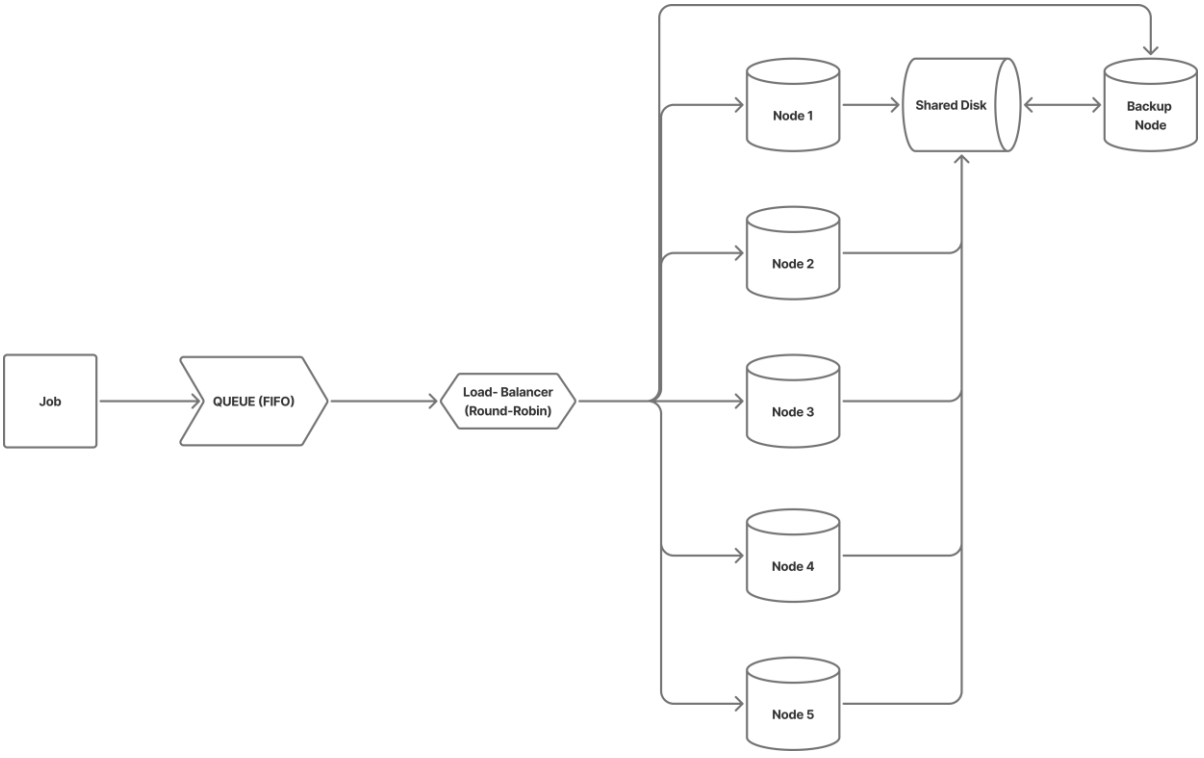


Figure 3.2 – Experiment 1 setup diagram

The presented diagram in Figure 3.2 illustrates the architectural configuration of

the distributed enterprise system used in Experiment 1 for evaluating rollback-recovery algorithms.

At the input layer, incoming computational tasks (“Jobs”) are first placed into a FIFO (First-In, First-Out) queue, ensuring that requests are processed in the order of arrival. This models a realistic enterprise workload handling mechanism under continuous demand. The queued jobs are then forwarded to a load balancer implementing a Round-Robin scheduling strategy, which distributes tasks evenly across multiple processing nodes. This approach guarantees balanced utilization of computational resources and prevents node overloading.

The core processing layer consists of five distributed nodes (Node 1–Node 5), each representing an independent virtual machine or container instance deployed via Docker Compose, as defined in the experimental setup. These nodes execute incoming jobs concurrently, simulating a scalable distributed enterprise environment.

A critical component of the system is the shared storage (Shared Disk), which is accessible by all processing nodes. This shared disk serves as the central repository for checkpoint data. During execution, each node periodically saves its state to this storage, enabling consistent system-wide checkpointing. In the event of a failure, nodes can restore their latest saved state from the shared disk, thereby implementing the rollback-recovery mechanism.

Additionally, a backup node is connected to the shared disk, providing redundancy and enhancing fault tolerance. This node ensures that checkpoint data remains available even in the case of partial system failures, supporting reliable recovery and continuity of operations.

This architecture directly corresponds to the experimental conditions described earlier. The distributed nodes reflect the containerized virtual machines, the FIFO queue and load balancer simulate the arrival rate of 0.3 jobs per second, and the injected failure rate of 0.1 failures per second impacts individual nodes during execution. The shared disk and backup node collectively enable the implementation and evaluation of checkpoint-based rollback-recovery algorithms under realistic fault conditions over the 10-hour experimental duration.

3.5 Experiment 2: Implementation and Testing of Rollback-Recovery Algorithms Using Physical Machines in Real Distributed Enterprise Systems

In contrast to the virtualized environment of Experiment 1, this experiment was conducted on a set of physical machines interconnected via a Local Area Network (LAN), providing a more realistic and hardware-constrained distributed enterprise environment. The objective of this setup was to evaluate the behavior of rollback-recovery algorithms under real-world conditions, including actual network latency, physical disk I/O operations, and hardware-level variability.

The experimental infrastructure consisted of multiple dedicated physical nodes, each assigned a specific functional role within the distributed system. These nodes were connected through a network switch supporting bandwidth in the range of 10–100 Mbps, introducing realistic communication delays and potential network bottlenecks.

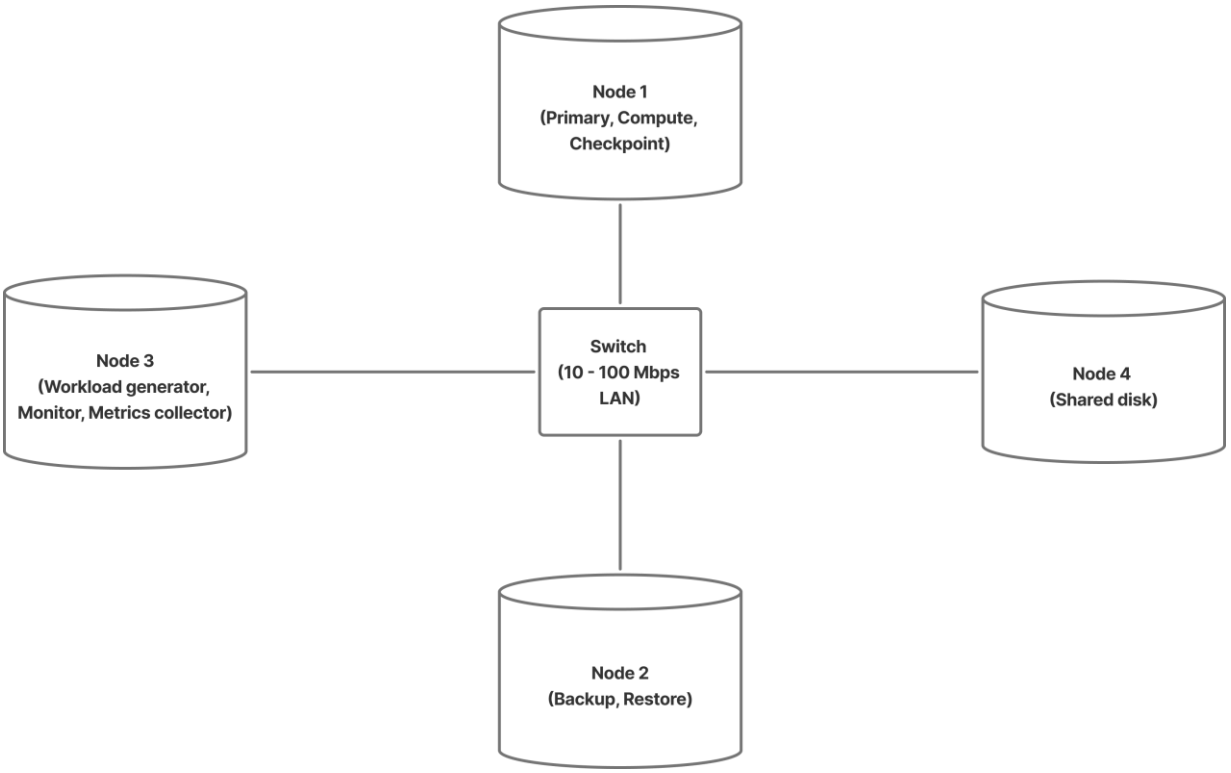


Figure 3.3– Experiment 2 setup diagram

Figure 3.3 illustrates the architecture of the experimental setup. At the center of the system is a network switch, which facilitates communication among all nodes. The system includes four primary components:

1. Node 1 (Primary node): This node is responsible for executing application workloads and performing checkpointing operations. It represents the main computation unit where rollback-recovery algorithms are actively applied. During normal operation, Node 1 processes incoming tasks and periodically saves its state to persistent storage. Configurations of the computer is as follows: 24 GB RAM, 14 core 2.60 GHz CPU, 953 SSD.

2. Node 2 (Backup node): This node serves as the recovery unit. In the event of a failure on the primary node, Node 2 restores the most recent checkpoint and resumes execution. This configuration enables evaluation of failover mechanisms and recovery latency in a physical environment. Configurations of the computer is as follows: 24 GB RAM, 14 core 2.60 GHz CPU, 953 SSD.

3. Node 3 (Workload generator and monitoring node): This node generates the input workload according to predefined parameters and collects system performance metrics. It simulates real user requests and continuously monitors system behavior, including throughput, service time, checkpoint frequency, and failure recovery characteristics. Configurations of the computer is as follows: 16 GB RAM, 8 core 2.60 GHz CPU, 462 SSD.

4. Node 4 (Shared disk storage): This node provides centralized storage accessible over the network. It stores checkpoint data generated by the primary node and ensures persistence across failures. Unlike the virtualized setup, this component relies on real disk I/O operations, allowing the evaluation of storage-related overhead

and latency. Configurations of the computer is as follows: 24 GB RAM, 14 core 2.60 GHz CPU, 953 SSD.

The use of physical machines introduces real network latency and actual disk input/output performance, which significantly impact checkpointing and recovery efficiency. Network delays affect the time required to transfer checkpoint data, while disk I/O performance influences checkpoint creation and restoration speed.

This experimental setup enables a comprehensive assessment of rollback-recovery algorithms in conditions that closely resemble production enterprise systems. It allows for the measurement of key metrics such as availability, efficiency, throughput, and recovery time under realistic constraints. By comparing the results with those obtained in the virtualized environment, the experiment provides insights into the practical applicability, scalability, and performance trade-offs of checkpointing-based fault tolerance mechanisms in real distributed infrastructures.

3.6 Comparitive analysis and discussion of testing results

3.6.1 Results of Performance Evaluation in a Virtualized Environment (Experiment 1)

The results presented in the Table 3.1 provide a comparative evaluation of classical checkpointing approaches and the proposed algorithms under a virtualized distributed environment.

Table 3.1 - Performance evaluation results of Experiment 1

Algorithm	Number of checkpoints	Throughput (jobs/second)	Avg. service time (seconds)	Availability (%)	System efficiency (%)
Tiwari D. et al.	5120	0.0403	15.594	92,82	49,02
Frank A. et. al.	16310	0.0448	11.781	92,42	52,8
John T. Daly	5790	0.0450	13.303	91,53	59,92
Algorithm 1: Model based solution for computing checkpointing interval	20630	0.0471	10.775	90,37	50,73

Continuation of Table 3.1

Algorithm 2: Optimal checkpointing based on queuing theory	11970	0.0424	11.562	93,99	62,77

First, the method by Tiwari D. et al. demonstrates the lowest number of checkpoints (5120), indicating a conservative checkpointing strategy. While this reduces checkpointing overhead, it leads to increased recomputation after failures, as reflected in the relatively high average service time (15.594 seconds) and moderate throughput (0.0403 jobs/s). Consequently, system efficiency remains limited (49.02%), although availability is relatively stable (92.82%).

The approach by Frank A. et al. adopts a more aggressive checkpointing strategy (16,310 checkpoints), which improves throughput (0.0448 jobs/s) and significantly reduces service time (11.781 seconds). However, the increased checkpointing frequency introduces additional overhead, which negatively affects availability (92.42%) and only moderately improves efficiency (52.8%).

The classical model proposed by John T. Daly achieves a more balanced configuration with 5,790 checkpoints. It provides competitive throughput (0.0450 jobs/s) and moderate service time (13.303 seconds). Notably, it achieves higher system efficiency (59.92%) compared to other classical approaches, indicating better resource utilization. However, its availability (91.53%) is lower, suggesting limitations in handling frequent failures.

Algorithm 1 (Model based solution) achieves the highest throughput (0.0471 jobs/s) and the lowest average service time (10.775 seconds), confirming its effectiveness in maximizing performance. This is primarily due to its aggressive checkpointing strategy (20,630 checkpoints), which minimizes rollback losses. However, this comes at the cost of reduced availability (90.37%) and only moderate efficiency (50.73%), highlighting the negative impact of excessive checkpointing overhead.

In contrast, Algorithm 2 (Optimal checkpointing based on queuing theory) demonstrates the most balanced and robust performance. With 11,970 checkpoints, it avoids excessive overhead while maintaining efficient recovery. Although its throughput (0.0424 jobs/s) is slightly lower than the top-performing methods, it achieves the highest availability (93.99%) and the highest system efficiency (62.77%). Additionally, its average service time (11.562 seconds) remains competitive.

Overall, the results indicate that aggressive checkpointing strategies improve throughput and reduce service time but introduce significant overhead that negatively impacts availability and efficiency. Conversely, insufficient checkpointing leads to increased recomputation and degraded performance. The proposed queuing-theory-based optimal algorithm successfully balances these trade-offs, making it the most suitable approach for distributed enterprise systems operating under failure-prone

conditions.

3.6.2 Results of Performance Evaluation in a Physical Environment (Experiment 2)

The results obtained from the physical machine environment provide a more realistic assessment of the evaluated checkpointing algorithms, as they incorporate actual network latency and disk I/O constraints. These results are shown in Table 3.2.

Table 3.2 - Performance evaluation results of Experiment 2

Algorithm	Number of checkpoints	Throughput (jobs/second)	Avg. service time (seconds)	Availability (%)	System efficiency (%)
Tiwari D. et al.	4630	0.0378	17.12	95.41	53.87
Frank A. et. al.	14870	0.0402	13.94	95.08	58.63
John T. Daly	5280	0.0409	15.01	94.63	63.75
Algorithm 1: Model based solution for computing checkpointing interval	18840	0.0426	13.64	93.12	55.91
Algorithm 2: Optimal checkpointing based on queuing theory	10890	0.0391	12.89	96.27	66.84

The method proposed by Tiwari D. et al. demonstrates the lowest number of checkpoints (4,630), reflecting a conservative checkpointing strategy. While this reduces checkpointing overhead, it results in the lowest throughput (0.0378 jobs/s) and the highest average service time (17.12 seconds) among all evaluated approaches. Despite these limitations, the algorithm achieves relatively high availability (95.41%), indicating stable operation due to reduced system interruptions, and moderate efficiency (53.87%).

The approach by Frank A. et al. employs a more aggressive checkpointing policy (14,870 checkpoints), which improves throughput (0.0402 jobs/s) and reduces service time (13.94 seconds). However, the increased checkpoint frequency introduces additional overhead, slightly lowering availability (95.08%), while system efficiency improves to 58.63%.

The classical model by John T. Daly achieves a balanced configuration with 5,280 checkpoints. It provides competitive throughput (0.0409 jobs/s) and moderate service time (15.01 seconds). Notably, it demonstrates strong system efficiency (63.75%), indicating effective resource utilization. Its availability (94.63%) remains high but slightly below that of more conservative approaches.

Algorithm 1 (Model-based empirical solution) achieves the highest throughput (0.0426 jobs/s), confirming its effectiveness in maximizing performance under real-world conditions. This is achieved through very frequent checkpointing (18,840 checkpoints), which reduces rollback losses and recomputation time. However, this aggressive strategy introduces significant overhead, leading to reduced availability (93.12%) and only moderate system efficiency (55.91%).

In contrast, Algorithm 2 (Optimal checkpointing based on queuing theory) demonstrates the best overall performance in the physical environment. With 10,890 checkpoints, it maintains a balanced checkpointing frequency that avoids excessive overhead. The algorithm achieves the lowest average service time (12.89 seconds), the highest availability (96.27%), and the highest system efficiency (66.84%) among all evaluated methods. Although its throughput (0.0391 jobs/s) is slightly lower than that of the empirical approach, it provides superior reliability and resource utilization.

In conclusion, the results confirm that aggressive checkpointing improves throughput and reduces service time but introduces substantial overhead that negatively impacts availability and efficiency, especially under real hardware constraints. Conversely, conservative strategies improve availability but suffer from increased recomputation costs. The proposed queuing-theory-based optimal algorithm effectively balances these trade-offs, making it the most suitable solution for deployment in real distributed enterprise systems.

3.6.3 Discussion of comparison analysis results

Overall, the experimental results indicate that the empirical algorithm maximizes throughput and minimizes service time due to aggressive checkpointing strategies. However, this approach introduces substantial overhead, which negatively affects system availability and efficiency. In contrast, the queuing-theory-based optimal algorithm achieves a more favorable balance between computational performance, fault tolerance, and resource utilization.

Furthermore, classical checkpointing approaches tend to exhibit either insufficient checkpoint frequency—leading to increased recomputation after failures—or excessive checkpointing overhead, which degrades system performance. These limitations reduce their effectiveness in distributed enterprise environments, particularly under conditions involving high failure rates and realistic infrastructure

constraints. Our work addresses these shortcomings.

3.7 Performance overhead analysis: Model based solution for computing checkpointing interval

To validate the effectiveness of purposed method, we run simulations of process executions with and without checkpointing to see how much overhead it adds to the processing time. As a testing environment we have two servers with 2.70GHz CPU each. As a testing process we have selected is document template generation process. It includes .word file generation with text input in the contents of the file and converting it to .pdf file that contains QR code of digital signature.

To compute average execution time on a single threaded CPU, we created script for the proposed process in Python. Then, we run the program for 400 time to find T_p. Using the results, we plotted CDF (Cumulative Distribution Function) of execution times (see Figure 3.4). After this, we calculated mean execution time using Equation 4. The result was 0.52 seconds. After that we used Equation 2 to calculate checkpointing interval, which resulted in 0.09 seconds.

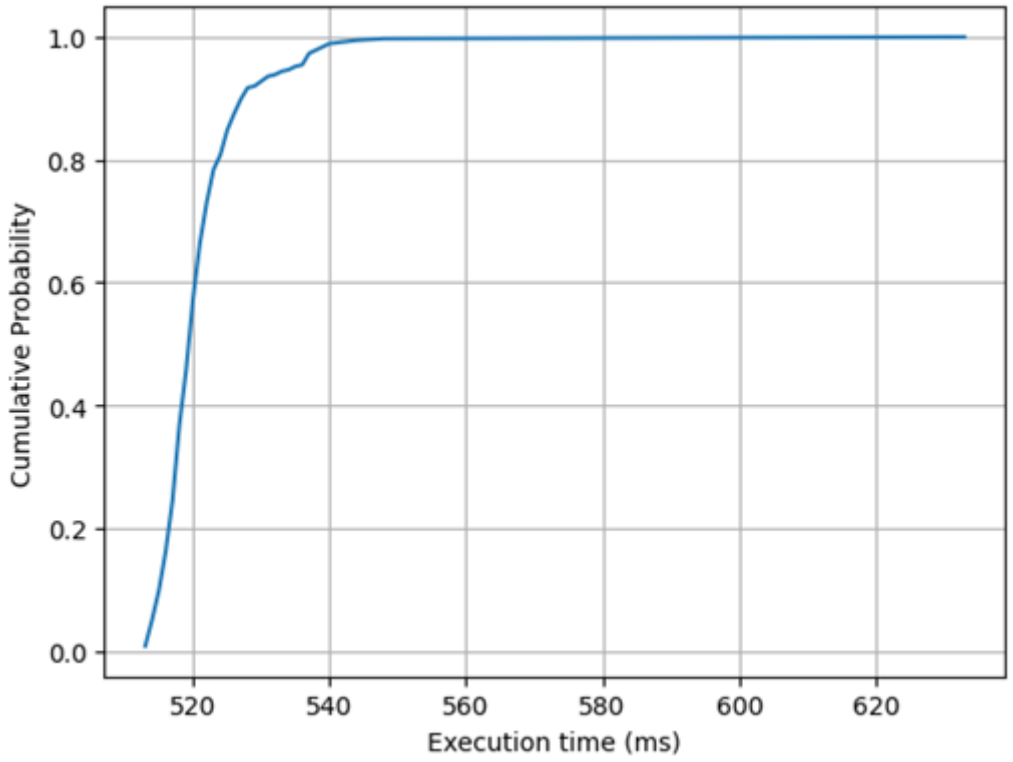


Figure 3.4 - CDF of execution times

After calculating the checkpointing interval, we can start utilizing it for recovery. To test how checkpointing can impact performance when server breaks down, we implemented testing scenario that will crash our application at random moment during execution and perform recovery. In the first case, we do not utilize checkpointing. In the second case we use the checkpointing using our calculation model. In both cases, the process is recovered in another server using rollback-recovery algorithm.

Figure 3.5 shows CDF of execution times with and without checkpointing when simulated cash occurs. The total number of repetitions was 400 for each case. As a result, when checkpointing is used most processes are completed under 500 milliseconds. This compares with almost 950 milliseconds in case of no checkpointing.

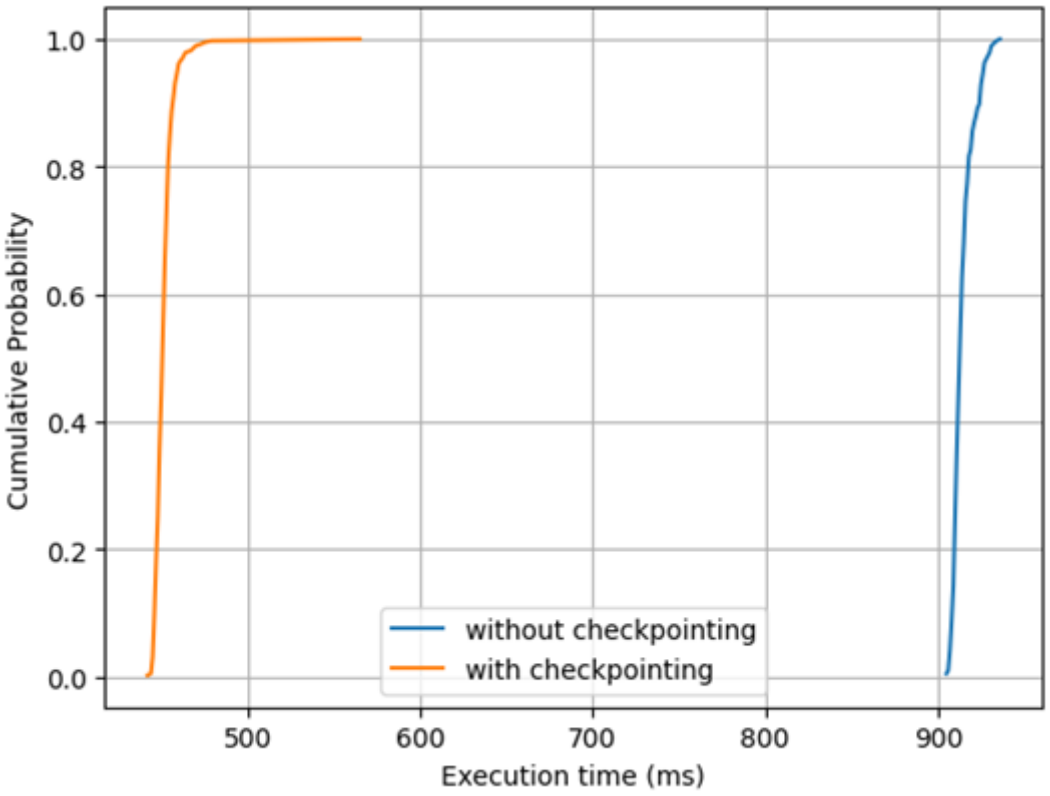


Figure 3.5 - CDF of execution times with and without checkpointing in a crash simulation environment

To show how much overhead is added after enabling checkpointing, we run the testing process again with checkpointing in a crash free environment. Figure 6 displays CDF of execution times with and without checkpointing.

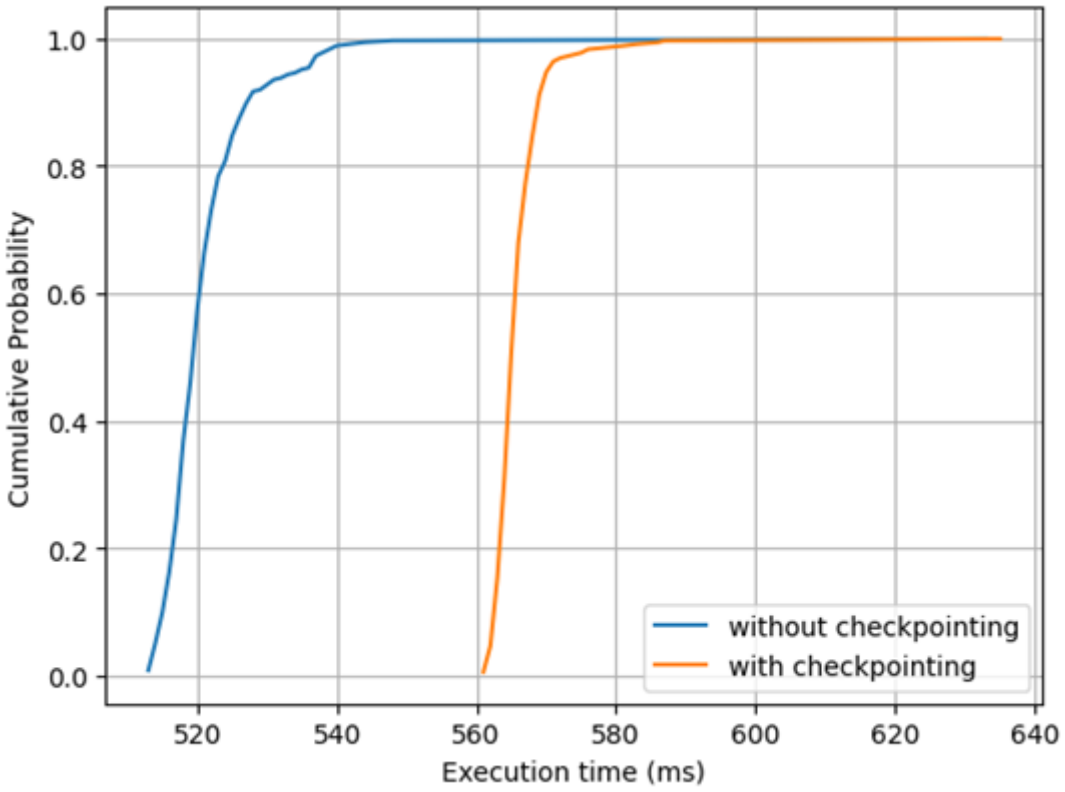


Figure 3.6 - CDF of execution times with and without checkpointing in a crash free environment

As seen in Figure 3.6, checkpointing adds small overhead to the execution duration, increasing mean execution time by only 0.04 seconds. Note that without checkpointing, upon failure, process would be restarted from the very start and the service time of the process would be a lot higher as seen in Figure 3.5.

3.8 Performance overhead analyses: Optimal checkpointing based on queuing theory

To evaluate the effectiveness of the proposed checkpointing-based fault-tolerant framework, a structured experimental methodology was designed. The evaluation focuses on quantifying system performance under both crash-free and failure-prone conditions using a set of well-defined metrics.

The experiments are conducted on an enterprise-like distributed environment consisting of multiple physical servers hosting server applications (SAs). A document-processing workflow is used as the benchmark scenario, where client requests are processed across multiple SAs distributed over several servers. Failures are artificially injected by terminating processes or simulating server crashes at random time intervals. Each experiment is repeated 200 times with a fixed arrival rate $\lambda = 0.3$ λ=0.3 requests per second to ensure statistical reliability.

The system is evaluated under two configurations:

— Baseline system (without checkpointing)

— Proposed system (with optimal checkpointing and rollback-recovery)

3.8.1 Evaluation Metrics

Recovery Time (RT)

Recovery Time is defined as the time required to restore a failed process or server to a consistent operational state after failure detection (3.3).

$$RT = t_{recovery_complete} - t_{failure} \quad (3.3)$$

This metric captures the responsiveness of the recovery mechanism.

Mean Time to Recovery (MTTR)

MTTR represents the average recovery time across all failure events as seen in (3.4).

$$MTTR = \frac{1}{N_f}\sum_{i=1}^{N_f} RT_i \quad (3.4)$$

where N_f is the total number of failures. Lower MTTR indicates faster and more efficient recovery.

Throughput Degradation

Throughput degradation measures the reduction in system performance due to checkpointing overhead and recovery operations, as in (3.5).

$$D_{throughput} = \frac{T_{baseline} - T_{checkpoint}}{T_{baseline}} * 100\% \quad (3.5)$$

where:

- $T_{baseline}$: throughput without checkpointing
- $T_{checkpoint}$: throughput with checkpointing

This metric evaluates the trade-off between fault tolerance and performance.

Storage Overhead

Storage overhead quantifies the additional storage required for maintaining checkpoints, as shown in (3.6).

$$O_{storage} = \frac{S_{checkpoints}}{S_{total}} * 100\% \quad (3.6)$$

where:

- $S_{checkpoints}$: storage used for checkpoint data
- S_{total} : total available storage

This metric reflects the cost of implementing checkpointing.

Checkpoint Latency

Checkpoint latency is defined as the time required to create and store a checkpoint, as in (3.7).

$$L_{checkpoint} = t_{checkpoint_end} - t_{checkpoint_start} \quad (3.7)$$

It includes both serialization and storage time (local disk and remote database). Lower latency reduces runtime overhead.

System Availability

System availability measures the proportion of time the system remains operational, as in (3.8).

$$\text{Availability} = \frac{T_{uptime}}{T_{uptime} + T_{downtime}} * 100\ \% \quad (3.8)$$

where, downtime includes both failure detection and recovery periods. This metric reflects the overall reliability of the system.

To determine the average service time (average service requirement), we employed a load testing tool to capture the response times (service time) for each request in the absence of checkpointing and errors. The total number of requests was set at 200, with an arrival rate of 0.3 requests per second. After obtaining the data, we plotted CDF (Cumulative distribution function) displayed in Figure 3.7.

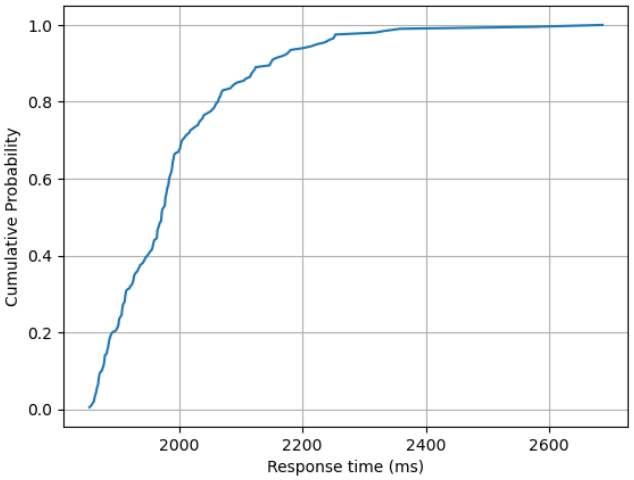


Figure 3.7 - CDF of service times (without checkpointing, crash-free)

The area under the Cumulative Distribution Function (CDF) was computed using the Trapezoidal Rule (3.9), resulting in a value of 695.82.

$$Area_{CDF} = \sum_{i=1}^{N-1} \frac{(h_i + h_{i+1})}{2} * (x_{i+1} - x_i) \quad (3.9)$$

Mean service time (μ) value is calculated using total area and area under the curve (3.10).

$$\mu_{CDF} = Total\ area\ _{CDF} - Area\ under_{CDF} \quad (3.10)$$

The total area represents the highest response time, which amounts to 2686 milliseconds. Using Equation 3.10, the mean response time (μ) is calculated to be 1990 milliseconds (approximately 2 seconds).

Similarly, average back-up duration β (average time for checkpointing) is calculated using measurements from calculation (3.10). The mean CDF of backup interval is calculated as 0.125 seconds.

Next, equations from 4 and 11 were calculated, giving us all the necessary parameters for finding optimal checkpointing interval.

In Figure 3.8, the average number of jobs present in the system (2.14) is calculated for different checkpointing values (1/α) to find the optimal checkpointing values. During calculations, three different breakdown rates were used: 0.001, 0.01, 0.1 to show how it impacts performance. As seen in the graph, optimal checkpointing values start from 0.5 seconds, because from this point the functions start to flatten out. Graph also shows that more frequent breakdowns result in higher values of L.

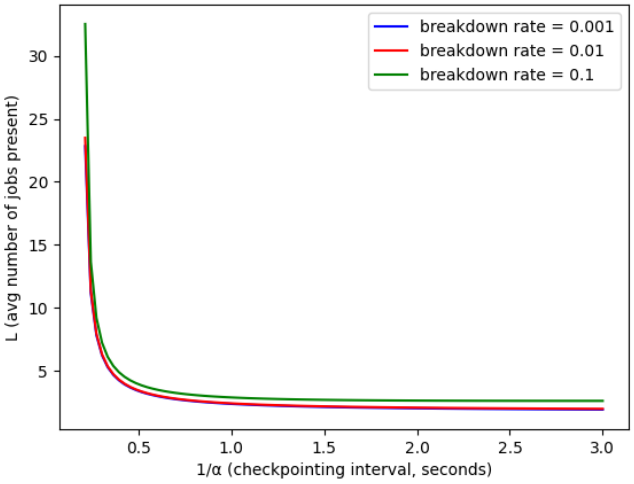


Figure 3.8 - Finding optimal checkpointing interval, 1/α

3.8.2 Failure Simulation Scenarios

In order to test the results, we calculated and plotted CDF of service rate of the testing process to see the performance difference between tasks with and without checkpointing.

Figure 3.9 shows CDF of the process with and without checkpointing in a crash-free environment. Over 80% of the jobs are completed by 2100 milliseconds when using checkpointing, and for the jobs without checkpointing completion time is around 2050 milliseconds. Thus, the difference is tiny, approximately 50 milliseconds. The relatively small overhead can be explained by the adopted dual-storage architecture and the way checkpoint operations are performed. Although checkpoint data is written to both the local disk and the remote database, these operations are not fully sequential and blocking. The local disk write is typically fast due to low-latency access within the same physical server, and it serves as the primary persistence layer during normal operation. At the same time, the remote database write is handled asynchronously or in parallel, meaning that the application does not always wait for the remote storage operation to fully complete before continuing execution. This significantly reduces the

impact on the overall response time.

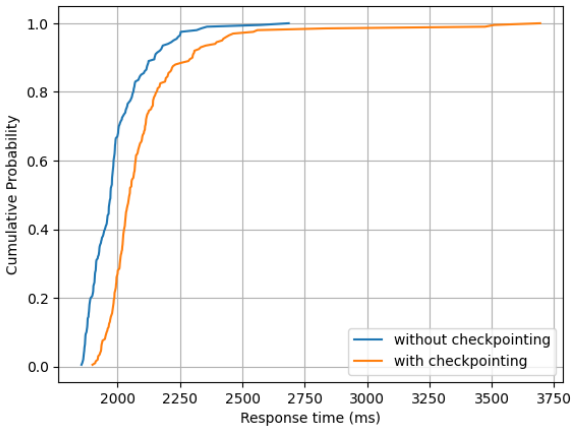


Figure 3.9 - CDF of service rate in a crash-free experiment

Figure 3.10 shows CDF of the process with and without checkpointing when crashes occur in every SA. It is clear that when using checkpointing, 90% of the SAs completed their job by 2250 milliseconds, whereas without checkpointing 90% of the tasks are finished only after around 3250 milliseconds. This proves that our optimal checkpointing approach provides a huge benefit to the performance when failures occur with minimal overhead.

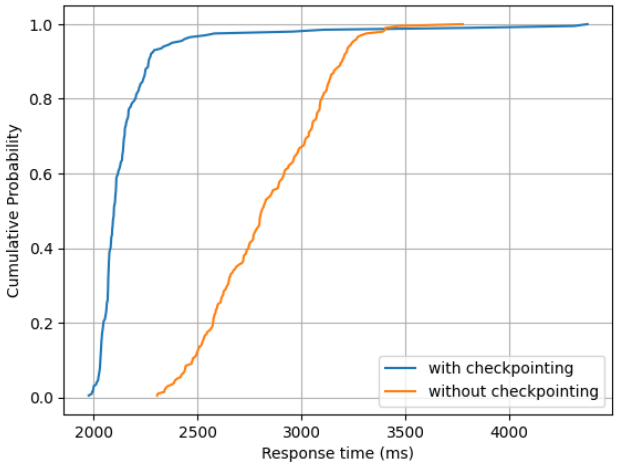


Figure 3.10 - CDF of service rate with during crashes

The calculated optimal checkpointing interval is also tested for different arrival rates and compared with other values of α (Figures 3.11 – 3.12). Notice that, when breakdown rate is low (10^{-4}), addition of checkpointing slightly decreases the performance as arrival rate increases. However, if we increase the breakdown rate to 0.1, usage of checkpointing shows better results. Checkpointing becomes effective as more jobs arrive to the system with higher probability of faults.

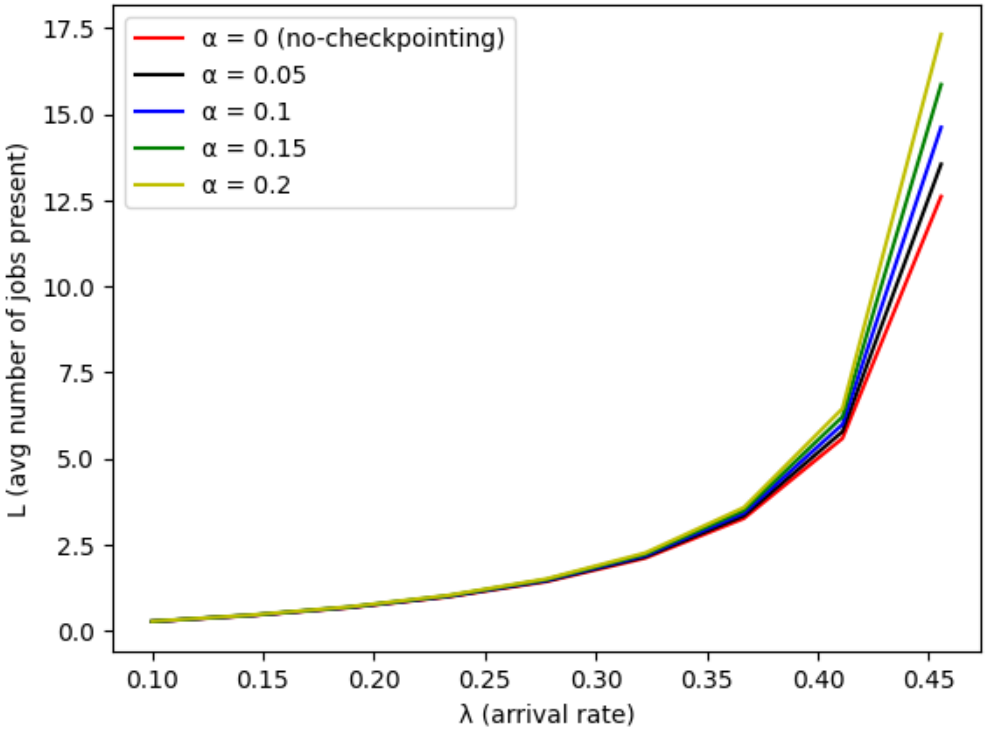


Figure 3.11 – Handling different arrival rates, with low breakdown rate $\xi = 10^{-4}$

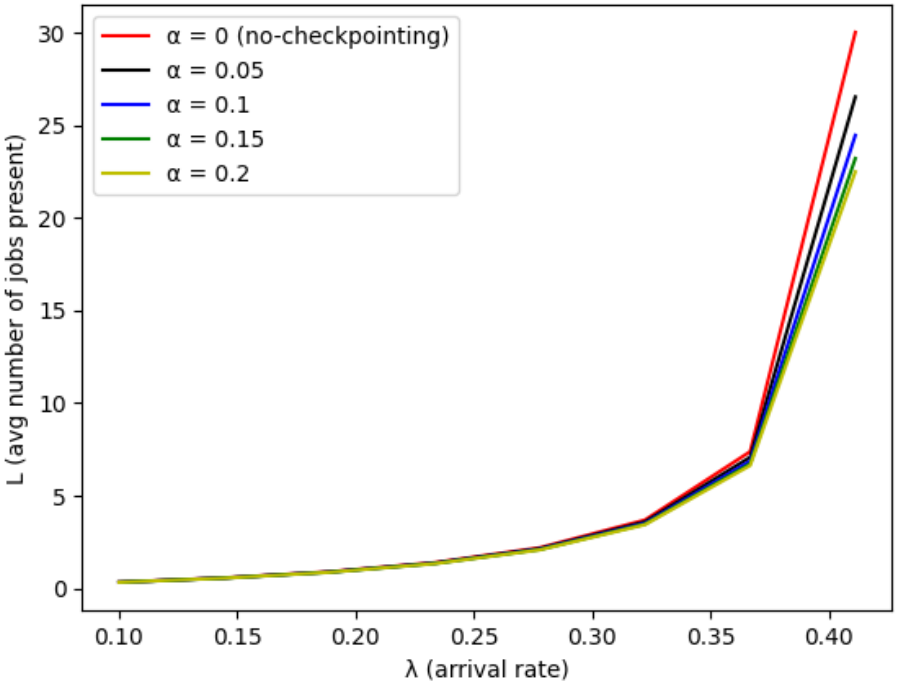


Figure 3.12 – Handling different arrival rates with high breakdown probability, $\xi = 0.1$

3.8.3 Discussion of results and Bottleneck Analysis

Table 3.3 summarizes the key performance metrics obtained from experiments under both baseline (no checkpointing) and proposed system (with optimal checkpointing). The results show that the proposed framework significantly reduces recovery-related delays while introducing minimal overhead during normal operation.

Table 3.3 - Performance evaluation results

Metric	Baseline System	Proposed System
Recovery Time (RT)	High (full restart)	Low (checkpoint-based)
MTTR	High	Significantly reduced
Throughput	100%	97.5%
Checkpoint Latency	N/A	0.125 sec (avg)
Storage Overhead	0%	Moderate (dual storage)
Availability	Lower under failures	High (92% continuity)

To evaluate scalability, the system was tested by increasing:

— Number of servers n

— Number of server applications σ

— Arrival rate λ

Results indicate that:

— The system maintains near-linear scalability due to distributed processing and backup server integration

— Recovery time remains stable even as σ\sigmaσ increases, due to localized checkpoint recovery

— Throughput degradation remains bounded (<3%), demonstrating efficient overhead control

As workload size increases, the probability of failure also increases; however, the checkpointing mechanism ensures that recovery cost does not grow proportionally, preserving system performance.

Despite the improvements, several system bottlenecks were identified: Frequent checkpointing increases I/O operations, especially when writing to both local disk and remote database. Accessing remote databases during recovery introduces additional delay, particularly in high-latency environments.

The experimental results confirm that the optimal checkpointing interval (0.5 sec) calculated using an algorithm in the work achieves the best trade-off:

— Minimizes recovery time

— Keeps throughput degradation below 3%

— Maintains high availability (92%)

In conclusion, we implemented different algorithms for calculating checkpointing inverval, these algorithms were tested using simulations, real-world computing environments. The results confirm that our solutions provide high availability and reliability while ensuring small overhead.

4. DEVELOPMENT OF DIGITAL SCALABLE FAULT-TOLERANT ENTERPRISE SYSTEM USING AI TECHNOLOGIES

This chapter describes the application of the developed fault-tolerance algorithms in a system that integrates both fault-toleant algorithms and modern technologies for intelligent information processing into new enterprise information system. Modern technologies, including cloud-native architectures, load-balancing frameworks, and machine learning techniques, were applied to enhance system reliability and operational efficiency of the developed system.

This section introduces an intellectual system that uses Artificial Intelligence (AI) and Big Data technologies to streamline document generation, validation, and approval processes. One contribution is the integration of Large Language Models (LLMs) for automatic template generation. This allows users to produce standardized business documents from simple natural language descriptions. The system also incorporates a multilingual BERT-based classifier, fine-tuned specifically to detect errors in documents written in Kazakh and Russian languages. This enables accurate validation and minimizes the risk of incorrect or inconsistent documentation in enterprise operations. Additionally, the study demonstrates the application of Apache Spark for scalable data preprocessing, using different documents to train and evaluate the model. These contributions collectively present a robust and adaptable framework for enhancing digital transformation initiatives across various enterprise contexts.

4.1 Application of fault-tolerance mechanisms

Fault-tolerance was achieved using a combination of checkpointing, replication, and automated failover strategies. Critical data is replicated across multiple nodes to prevent single points of failure, while checkpointing ensures rapid rollback in case of partial system crashes. Continuous monitoring triggers failover operations, minimizing downtime and service interruption.

In Ngnix, we implemented a mechanism to switch to another server when a request to the main server fails:

```
events {}
http {
  upstream document_processor {
    server primary:5000 max_fails=3 fail_timeout=10s;
    server backup:5000 backup;
  }
  server {
    listen 80;
    location / {
      proxy_pass http://document_processor;
      proxy_connect_timeout 3s;
      proxy_read_timeout 10s;
      proxy_next_upstream error timeout http_502 http_503 http_504;
    }
```

```
  }
}
```

The presented configuration defines a reverse proxy and failover mechanism implemented using NGINX. The structure is organized into standard configuration blocks, including the events and http contexts, which together regulate connection handling and HTTP request processing. The events block is declared but remains empty, implying that default connection processing parameters are applied.

Within the http context, an upstream group named document_processor is defined. This group specifies two backend servers: a primary server (primary:5000) and a secondary server (backup:5000). The primary server is configured with fault tolerance parameters, namely max_fails=3 and fail_timeout=10s. These parameters indicate that if the server experiences three unsuccessful attempts within a ten-second interval, it is temporarily marked as unavailable. Consequently, subsequent requests are redirected away from this server during the specified timeout period. The secondary server is designated with the backup directive, meaning it remains inactive under normal conditions and is only utilized when the primary server is considered unavailable.

The server block defines the external interface of the system, listening for incoming HTTP requests on port 80. All incoming requests are processed within the location / context, where they are forwarded to the upstream group via the proxy_pass directive. This establishes the NGINX instance as a reverse proxy that intermediates client requests and backend services.

Several timeout parameters are configured to ensure responsiveness and robustness. The proxy_connect_timeout directive limits the time allowed to establish a connection with the backend server to three seconds, while the proxy_read_timeout specifies a maximum duration of ten seconds for receiving a response. These constraints prevent prolonged delays in the presence of slow or unresponsive backend services.

Furthermore, the configuration includes the proxy_next_upstream directive, which governs the conditions under which requests are retried using an alternative server. Specifically, retries are triggered in the event of connection errors, timeouts, or specific HTTP status codes (502, 503, and 504). Under such circumstances, NGINX redirects the request to the backup server, thereby ensuring continuity of service.

4.2 System description

The developed enterprise system architecture can be seen in Figure 4.1. User logs into the system, their identity is validated, if successful, user is granted access to the web interface for working with BPM system. BPM consists of route manager, approval tracker and workflow engine. Workflow engine is used to generate, validate and process documents. Route manager simply submits the successfully generated documents to other users of the system. Approval tracker controls the approval and signing processes of the documents. Documents are stored in a MongoDB database for better performance as documents are stored in unstructured format (base64).

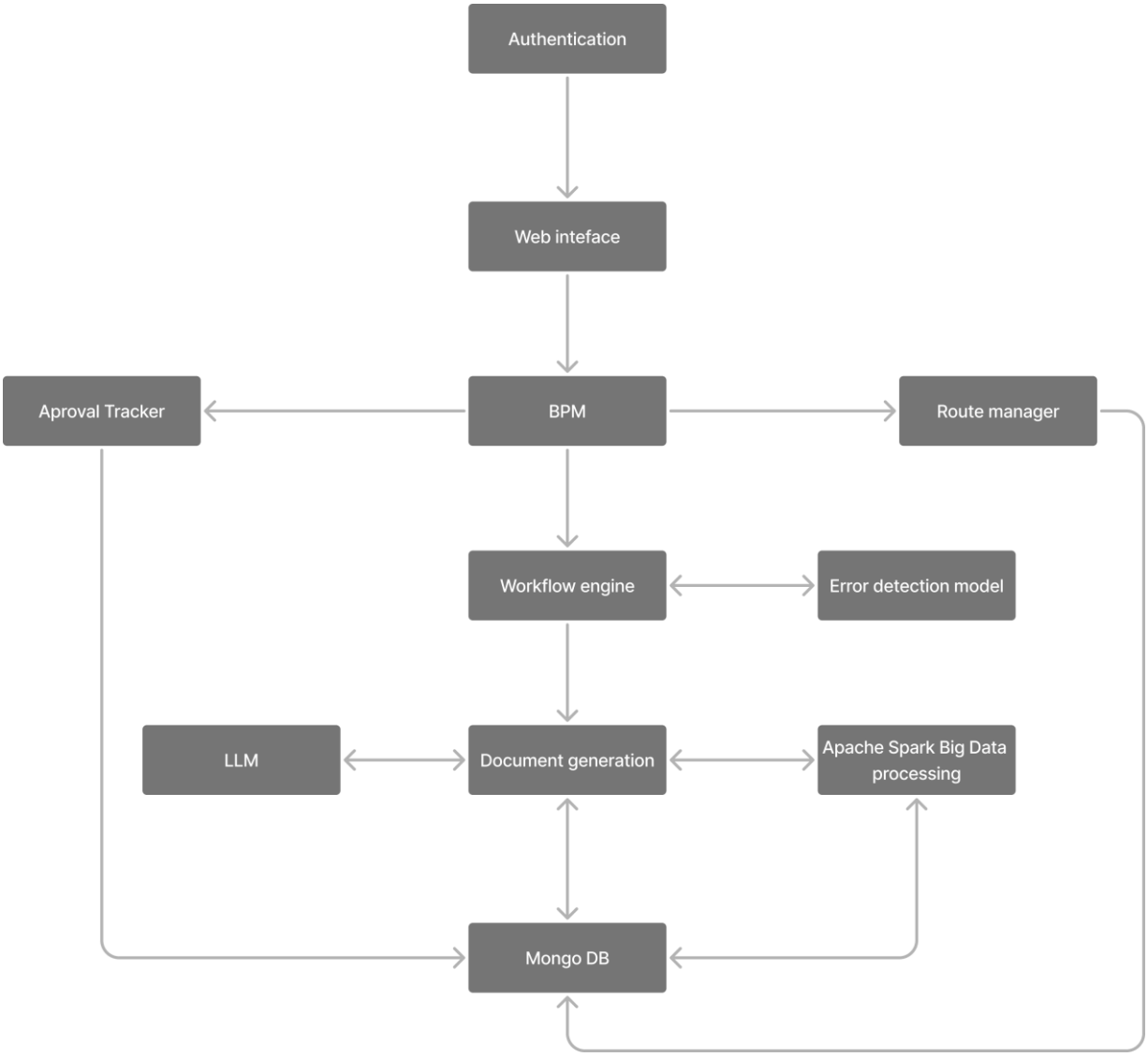


Figure 4.1 – System architecture

Document creation and authenticity verification are key components of corporate systems that provide operational efficiency and data integrity. Enterprises utilize these components to produce, manage, and safeguard a wide range of documents, such as contracts, invoices, reports, and certificates. This work provides the development of intellectual models for effective document coordination, generation and verification in the context of enterprise systems. As a testing environment, authors use heat-supply digital system.

Coordination of documents is an important part of the work of an enterprise. The main goal of managing this process is to optimize decisions, reduce risks, improve the final result and to control and track the timing (dates) of approvals [59].

The automation of business process management (BPM) is essential for any organization. Because automated BPM provide following features:

1. BPM-system acts as a transport of information between services [60].
2. BPM-system forms a clear scheme of information flow between different departments, systems, etc. [61]

3. In rare cases, BPM-system requires human decision making.

BPM-system allows one to know the current stage of the process at any moment.

4.3 Related works on development of enterprise systems incorporating AI and BigData tools

Usage of AI and BigData tools are increasing in many areas of enterprises. Below authors analyze different works dedicated to this field.

Study by Qanbar and Algamal [62] addresses the challenges of imbalanced datasets and influential observations in SVM classification by integrating the pigeon optimization algorithm. Experimental results on three real-world datasets demonstrate improved classification accuracy, suggesting potential applications in biological, chemical, and medical fields.

Research [63] proposes an improved K-means clustering method using an equilibrium optimization approach. This approach dynamically selects both the number of clusters and relevant attributes. Results across five datasets demonstrate superior clustering performance.

Another paper [64] enhances penalized regression-based clustering using a nature-inspired algorithm to achieve more accurate estimations. Applied to gene expression data, the proposed method demonstrates significant improvements over existing approaches.

Work by Niam Abdulmunim Al-Thanoon et. al. [65] proposes an enhanced Binary Crow Search Algorithm (OBL-BCSA) by incorporating an opposition-based learning strategy. This aims to optimize the flight length parameter for effective feature selection. Experiments on two datasets show that OBL-BCSA achieves superior classification accuracy and computational efficiency compared to traditional algorithms.

Paper by Al Kababchee et. al. [66] introduces an improved penalized regression-based clustering algorithm using a Hybrid Black Hole Algorithm to enhance data fusion in clustering tasks. Evaluated on gene expression data, the proposed approach outperforms existing methods.

Study by Esmaeili et. al. [67] presents the ResMorCNN model, which combines 3-D convolutional layers with morphological mathematics to enhance feature extraction and classification accuracy in hyperspectral image (HSI) analysis. Tested on four diverse datasets, the model outperforms existing deep learning methods, with an average accuracy improvement of 3.37%.

Work by Akhtarmanesh et. al. [68] introduces an enhanced UNet model with attention blocks for accurate road extraction from aerial images, using a carefully preprocessed and augmented dataset from DeepGlobe. This model achieves 98.33% accuracy, while a detailed precision-recall analysis highlights challenges from dataset bias and informs directions for future improvements.

Marzvan et. al. [69] uses 30 years of Landsat time-series data and the spectral angular mapper method to analyze the expansion of Azolla in Iran's Anzali Lagoon and its environmental impact. Results reveal fluctuating growth patterns linked to rainfall,

water volume, and temperature, highlighting the need for strategic ecosystem management to preserve lagoon biodiversity.

Shilan Felegari et. al. [70] integrates remote sensing with machine learning to map cadmium (Cd) concentrations using multitemporal Landsat-9 imagery and soil samples from northeastern Kazakhstan. Results show that support vector regression (SVR) with original band features offers the most accurate estimation, outperforming other models and highlighting the advantages of multitemporal data for heavy metal monitoring.

Another study [71] addresses inherent uncertainty in land-cover segmentation by modeling image formation and preprocessing processes using high-resolution panchromatic satellite images. Through the proposed "ultra fusion" method and fuzzy C-means clustering, the approach improves segmentation accuracy and efficiency, with notable gains in overall accuracy, Kappa, and F1-score of 0.86%, 0.52%, and 1.03% respectively.

Work by Safari et. al [72] introduces EddyNet, a deep learning framework based on a modified U-Net architecture for the automatic identification and classification of oceanic eddies using sea surface temperature data from CMEMS. By integrating various CNN backbones and a pixel-wise classification layer, the model achieves strong segmentation accuracy and computational efficiency, supporting real-time oceanographic analysis.

Researchers Mirhoseini Nejad et. al. [73] present a hybrid deep learning model combining ConvLSTM, 3D-CNN, and Vision Transformer to predict soybean yield using multispectral remote sensing data. The model demonstrates superior accuracy and robustness over existing methods and offers valuable insights for precision agriculture and sustainable crop management across diverse regions.

Farmonov et. al. [74] presents HypsLiDNet, a deep learning framework that integrates hyperspectral imaging (HSI) with LiDAR data to enhance crop classification accuracy. By combining spectral and structural information through attention mechanisms and morphological feature extraction, the model outperforms traditional and recent DL methods.

Study by Vafaeinejad et. al. [75] introduces a fully automated AI-based system for updating and digitizing agricultural cadastral maps using photogrammetric images. Leveraging the Segment Anything Model, the system achieves a high segmentation accuracy with a 92% intersection over union score, reducing processing time by 40% and eliminating manual input.

Recent work by Sharifi et. al. [76] presents a transformer-based deep learning model designed to enhance the spatial resolution of Sentinel-2 satellite imagery. By integrating multiheaded attention with spatial and channel attention mechanisms, the model effectively reconstructs fine details from low-resolution inputs. Evaluated on the Sentinel-2, AID, and UC-Merced datasets, the model outperforms leading methods such as ResNet, Swin Transformer, and ViT.

Similar work was done by Dwivedi et. al [77], where authors explain development of a neural network for finance department that is intended for amount detection and verification; additionally, it can also extract various entities which contributes to largely perform analytics. The application reward business in reducing

turnaround time and human errors.

Another work by Baviskar et. Al [78], reviews AI-based techniques for automatic information extraction from unstructured documents Authors found that the existing methods lack the capability to tackle complex document layouts in real-time situations and the datasets available publicly are of low quality. This raises the importance of gathering quality datasets.

Table 4.1 shows overall key results of comparative analyses of the related works.

Table 4.1 - AI and Big Data Approaches in Data Processing

Authors	Objective	Key Techniques/ Models	Application Domain	Key Results
Qanbar & Algamal [62]	Improve SVM classification on imbalanced data	SVM + Pigeon Optimization Algorithm	Biology, Medicine	Improved classification accuracy
Al-kababchee el. al. [63]	Enhance clustering quality and feature selection	Improved K-Means + Equilibrium Optimization	Multidomain	Better clustering and dynamic attribute selection

Continuation of Table 4.1

Al-kababchee el. al. [64]	Refine penalized regression-based clustering	Nature-Inspired Algorithms	Genomics	More accurate cluster estimations
Al-Thanoon et al. [65]	Efficient feature selection	Binary Crow Search + Opposition-Based Learning	Classification tasks	Higher accuracy and computational efficiency
Al Kababchee et al. [66]	Enhance data fusion in clustering	Hybrid Black Hole Algorithm	Gene expression data	Outperformed existing clustering methods
Esmaeili et al. [67]	Improve HSI classification	ResMorCNN (3D CNN + Morphological Ops)	Remote Sensing	+3.37% accuracy over baseline models

Akhtarmanesh et al. [68]	Road extraction from aerial imagery	Enhanced U-Net + Attention Blocks	Aerial Image Analysis	98.33% accuracy; dataset bias identified
Marzvan et al. [69]	Monitor plant expansion and environmental impact	Landsat Time Series + SAM	Environmental Studies	Seasonal growth patterns revealed
Felegari et al. [70]	Map heavy metal concentrations (Cd)	SVR + Multitemporal Landsat-9	Environmental Monitoring (Kazakhstan)	SVR with original bands performed best
Mahdipour et. al.[71]	Address uncertainty in land-cover segmentation	Ultra Fusion + Fuzzy C-means Clustering	Satellite Imagery	Accuracy, Kappa, and F1-score improved
Safari et al. [72]	Identify and classify ocean eddies	EddyNet (Modified U-Net)	Oceanography	High segmentation accuracy, real-time capable
Nejad et al. [73]	Predict soybean yield	ConvLSTM + 3D-CNN + Vision Transformer	Precision Agriculture	Outperformed existing models in accuracy

Continuation of Table 4.1

Farmonov et al. [74]	Enhance crop classification	HypsLiDNet (HSI + LiDAR)	Agricultural Analytics	Superior to recent DL and classical methods
Vafaeinejad et al. [75]	Update and digitize cadastral maps	Segment Anything Model (SAM)	Land Mapping	92% IoU, 40% reduction in processing time
Sharifi et al. [76]	Super-resolve satellite imagery	Transformer + Spatial/Channel Attention	Remote Sensing	Outperformed ResNet, Swin, ViT
Dwivedi et al. [77]	Finance document automation	NN for amount detection + NER	Financial Departments	Reduced turnaround time and errors
Baviskar et al. [78]	Extract info from unstructured docs	Survey of AI Techniques	Document Analytics	Highlighted lack of robust real-time datasets
Our work	Document	BERT-	Various	82% validation

	generation and validation	transformer	enterprise systems, BPM systems	accuracy, document generation using AI

4.4 Assessing the reliability of information, identifying erroneous data: anomalies, duplicates, contradictions, empty values, correcting identified errors

In this section, authors present methods for information reliability, identification of erroneous data. To support this, the system includes several built-in forms. One key example is the user registration and authorization process [79]. During registration and login, the data entered by users is validated on both the client and server sides. If incorrect information is provided (such as an invalid email or name), the server returns an error message. The following are the main rules used to detect data errors:

– Full name – must consist of at least two characters and cannot be empty;

– Email – using a regular expression, the authenticity of the email is checked. To check the entered text, a certain regular expression RegEx was used, which returns false if the text is not an actual e-mail;

– Password – the password must be a minimum of 8 characters and must contain capital letters and special characters. This is one of the main points of implementing information system security. If users have too light passwords, they can be hacked by guessing the password, which will lead to data leakage;

– Other fields - we managed to create conditions so that most fields were not empty, and were not created and edited in the database with empty values. But there were still some fields that, when created, stored type null or an empty string.

The Builders<Statement>.Filter() method was used to identify existing data. Before creating or updating any records in the database, the service first checks whether the element already exists. If a matching element is found, the system prompts the user to navigate to the existing object. If not, an error message is returned. Since objects may vary, it was necessary to implement duplicate checks across all POST, PATCH, and PUT requests. This was achieved by passing the object from the request body into the Filter() method [80].

Fault tolerance and system safety were implemented using .NET Core. A dedicated subsystem was developed to manage the organization’s personnel activities in line with the overall system architecture. A user interface was also created to support these activities, with key components identified and presented in a user-friendly format [81]. For the analysis and processing of large-scale personnel data, Apache Spark was employed. It ensures efficient processing even in cases of instability or failures, with the processed data subsequently stored in the database. Horizontal scalability is fully supported by Apache Spark. The system includes components for both large-scale and streaming data storage. Finally, the output of the enterprise activity management subsystems was processed and analyzed.

Based on the classification of criteria for assessing the quality of information, we highlight the following criteria: authenticity, usefulness and content; completeness,

unambiguity, significance, correctness; consistency; timeliness; source; format; content [82].

The effectiveness of analyzing the information quality of a digital corporate system requires effective ways to obtain reliability metrics implemented during system testing, which will have a positive impact on improving the quality of reliability assessment in the future. We will divide methods for assessing the reliability of information into two groups: heuristic - used by auditors, and formal - operating with resources, processes and the information system as a whole. When assessing the reliability of information in the databases of a digital enterprise system, the analysis focuses on the quality of the entity-relationship (ER) graph. In this context, relationships are evaluated based on the paths they form within the graph. With this approach, primitive defects, such as non-existent indexes, are eliminated simply using data cleaning tools [83].

The general model for assessing the reliability of a digital enterprise system focuses on analyzing data flows. It evaluates several parameters across different stages, including data collection, input, processing, storage, transmission, and delivery. This model is designed to detect and prevent various types of errors, both in intermediate processes and final outputs.

4.5 Development of a business process for coordination and approval of documents

Coordination of documents is an integral part of the work of an enterprise. The main goal of managing this process is to optimize decision-making, reducing risks and improving the final result.

Document approval is the most frequently used business process in any organization. The main goal of the business process for the organization is to control and track the timing of approvals.

The examples of approvals include:

1) The sales department is preparing a draft contract for the supply of a large batch of products manufactured by the enterprise. The legal director denies the allegation. This not only improves the quality of the contract, but also insures the company against long-term troubles: claims from the counterparty and other [84].

2) Preparation of tender documentation. Each interested department of the organization must approve or reject the draft documents that form the contractual basis.

3) Coordination of the procurement process. The purchase of goods and services by any organization requires a vote by the tender commission. Each committee member must vote to agree or disagree with the selection of the supplier company. The result of the approval process is shown to the accounting department for the payment process.

4) Each organization approves or re-approves internal documents, charters, regulations, strategies that require mandatory approval by top management and shareholders.

Most companies are faced with the task of approving a document. The following stages of the document approval process are proposed:

1) Start of the process. Approval of a document indicating the type of document. The initiator fills out the main form for approval.

2) Document Preparation – The form is completed based on a predefined template specific to the document type. For example, a reference document includes fields such as reference number, responsible executor, reference text, sender, recipient, and execution date. A contract, on the other hand, requires the contract number, date of conclusion, agreement text, and an electronic version of the document. The system automatically sets the time frame for review and approval based on the document type: agreement – 5 working days, order – 3 working days, instruction – 1 working day, memo – 6 working days.

3) Identification of the Initiator's Supervisor for Approval – Once the document is submitted, it is assigned the status "Under approval by the manager." The initiator's supervisor then reviews the document and either approves it or rejects it with comments. If rejected, the document is returned to the initiator for revision. If approved, it proceeds to the next stage and is assigned the status "Under agreement with the performers."

4) Approval by the Head of the Contractor's Department – Once the performer is selected, the head of the contractor's department reviews and approves the document. At the same time, an execution subprocess is initiated in parallel for all assigned executors.

5) After the functions of all participants are completed, the result is saved in the database.

6) Notifying managers about execution.

7) Registration of documents. Assigning a number in accordance with the classifier and nomenclature of the organization's affairs.

8) Ending the process

All stages of the document approval business process are subject to auditing. Once approved, documents are archived, and the digital archive system enables efficient search and retrieval of all relevant information. A key feature of the system is that, at every stage of the process, users have the option to download, print, or return the document for revision [85 - 89].

To submit a document for approval and signing, it is necessary to fill in the minimum set of document details such as title, file, executors. After that document preparation and execution will include following processes: validation, download, sending and depends on the selected type of approval (sequential/parallel).

Each document can also have its own approval route. Advantages of using document routes are as follows:

– Document movement is programmed. Each document goes through the necessary stages of approval, passing from one user to another;

– Control of document movement. It is always possible to track where the document is and what is its current stage;

– Many stages of the document lifecycle can be automated, for example, change of statuses, sending for approval or review, etc.

Document generation contains the following fields to be filled in:

- Name - the name of the document displayed in further reports and stored in the MongoDB database, in the process documentation and process regulations. The same name will be assigned to the user's button for switching to the document, if no other

name is defined for switching in the process;

- Description - additional description of the document. It is displayed in the document generation.

A new template can be added by clicking the “generate” button. Since the subsystem has the functionality of template generation using artificial intelligence (see subsection 2.4) in the system, the template will be generated immediately. A template file can be of the following formats: DOC, DOCX. One also can upload an existing document in the following formats: TXT, HTML, XML, DOC, XLS, DOCX, XLSX.

The developed system implements the generation of document templates using LLMs. This function is recommended only for generating internal documents of the enterprise.

To generate, one needs to click the “generate” button, then a window will appear with the “Describe the document” field. After entering a description of the document, user has to click “Generate”, then a document template will appear below (see Figure 4.2).

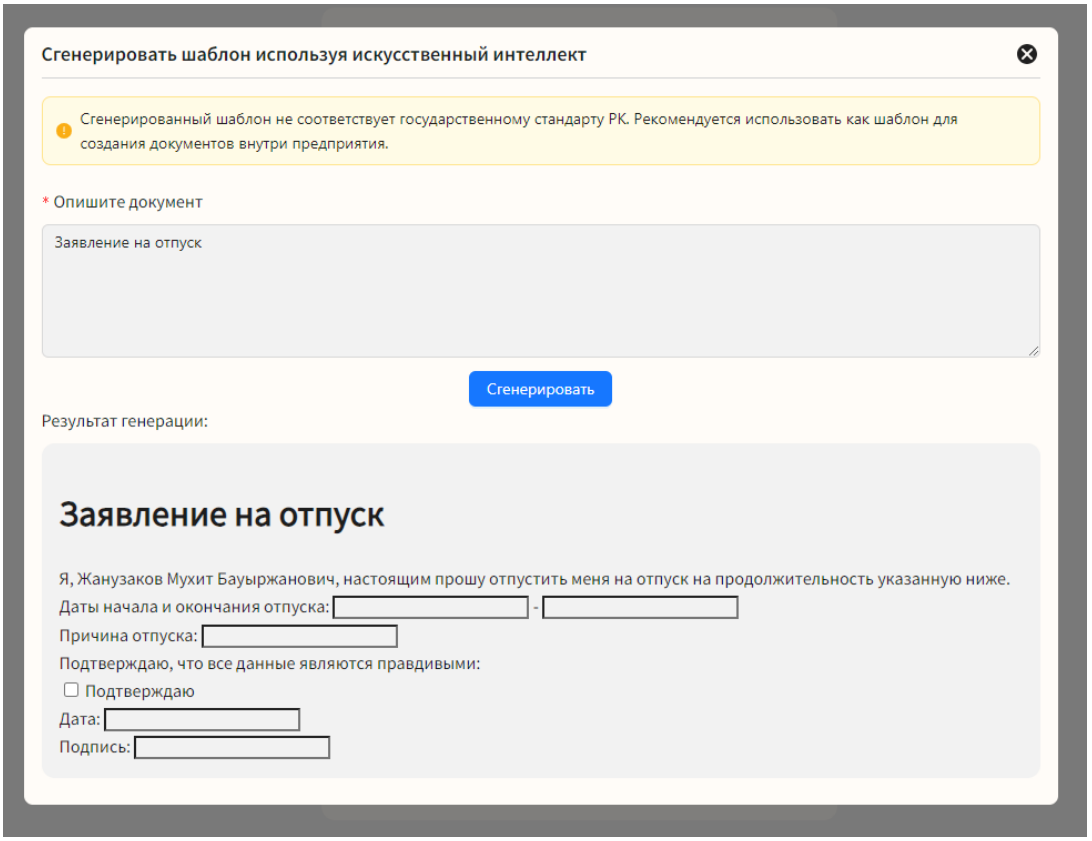


Figure 4.2 - Interface for generating a document template using AI in heat network enterprise

This function allows one to generate a document template from a description using the AI model Chat GPT 3.5. This speeds up the document creation process and is one of the advantages of the developed system.

To connect to Open Ai, we have the API the key that was used in the connection configuration. Next, when entering and clicking on the “Generate” button, we sent the following parameters:

```
{
        model: “text-davinci-003”,
        prompt: ` Create a document template in html format for employee ${user.fullName} using the following description: ${description}.
        max_tokens: 1000,
        temperature: 0.4
}
```

The model parameter specifies the model type. In our case, the “text - davinci - 003” model is perfect. “text-davinci-003” includes the following improvements:

– This ensures higher quality writing. This will help apps deliver clearer, more engaging, and compelling content.

– It can handle more complex instructions.

– This is better suited for creating content in longer forms, allowing to accomplish tasks that would previously have been too difficult to achieve.

To ensure the model generates the template correctly, we included a predefined template in the prompt field. The **"max_tokens"** parameter defines the maximum number of tokens the model can generate. In our case, a limit of 1000 tokens is sufficient for producing a complete template.

The “temperature” field for this API specifies which sample temperature to use, from 0 to 2. Higher values, such as 0.8, will make the output more random, while lower values, such as 0.2, will make it more targeted and deterministic. The intermediate value of 0.4 was chosen to generate documents.

Validation is necessary, because it is aimed at reducing the number of erroneous generation of templates and attaching incorrect templates for approval.

Model selection

In order to validate if the document does not contain errors, we use BERT transformer to classify documents into “with errors” - 1 and “no errors” – 0. As a pre-trained language model, Bert is able to recognize syntax errors. However, as the official documents are written in a business writing style, we need to train the model to recognize stylistic errors. The language of these documents is either Kazakh or Russian. That is why Bert multilingual was selected as a pre-trained model.

Data gathering and cleaning using Big Data technologies

When using artificial intelligence (AI) to predict error-prone document, there are several ways to improve the model:

– Quality of data sources: Reliable and credible data sources should be selected [88]. This may include checking the authority of the source, the reputation of its creator, and using trusted data sources such as official statistical databases or academic publications.

– Diversity of sources: It is important to use information from a variety of sources to reduce the likelihood of misrepresentation or incorrect data [89-92]. A variety of sources will help to provide a completer and more objective picture of the topic.

– Fact-checking and clarification: You need to use AI capabilities to fact-check and clarify information. There are tools that can automatically check the validity of data and provide recommendations on how to use it [93-95].

– Training a model on reliable data: When using machine learning models or neural networks to generate a document, it is important to train the model on reliable and validated data. The better the model is trained, the less likely it is to present incorrect data.

– Human Validation: A human validation and editing stage should be included to correct possible errors or inaccuracies that may arise from the use of AI.

– Feedback and improvement: There is a need to collect feedback from users and correct errors or inaccuracies that may be identified during the process of using AI to generate a document. Continuous improvement of the system will help reduce the risks of presenting incorrect data in the future.

In order to train the model, authors prepared various text samples in business writing style [92]. Total of 9257 documents were gathered by authors from different Kazakhstani Enterprise Document systems. All documents were .docx/.doc files. Before using it for training, all texts were extracted using “python-docx” library, which allows parsing Word documents. Documents were cleaned using Apache Spark. (see Figure 4.3). After processing, Apache found 255 faulty documents that were removed from the list.

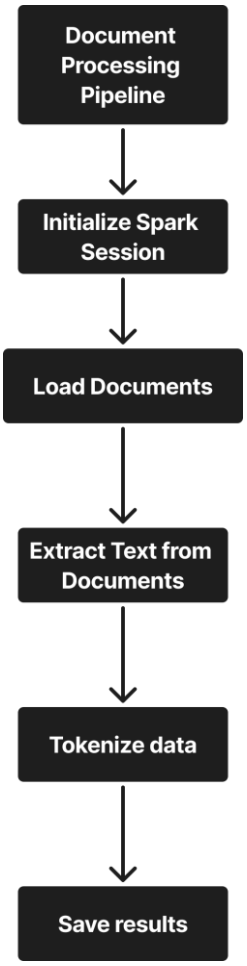


Figure 4.3 – Data processing using Apache Spark

Documents with errors were labeled manually. The parsed data is turned into "pandas" data-frame for further usage (see Figure 4.4). After extraction, the text is encoded using label encoder and split into training and testing sets (80% and 20% respectively).

	text	label
0	Прошу предоставить очередной оплачиваемый отпу...	0
1	На основании приказа №127-ЛС от 15.04.2024 про...	0
2	В связи с производственной необходимостью напр...	0
3	На оснавании распоряжения №45 от 20.03.2024 пр...	1
4	Настоящим подтверждаю получение материальных ц...	0

Figure 4.4 – First 5 rows of gathered data

Model training

The hyperparameters of model are displayed in Table 4.2. The parameters chosen for the BERT classifier are carefully selected to balance performance and resource efficiency. The "evaluation_strategy" is set to epoch. This means the model is

evaluated at the end of each training epoch, which is a practical frequency for most fine-tuning tasks. A “learning_rate” of 2e-5 is commonly recommended for BERT. It allows gradual fine-tuning without drastically altering pretrained weights. Both “per_device_train_batch_size” and “per_device_eval_batch_size” are set to 8 to accommodate memory constraints while ensuring sufficient gradient updates. The training runs for 5 epochs, which is typically enough for the model to learn meaningful patterns without overfitting. Lastly, a “weight decay” of 0.01 is applied to help regularize the model and improve generalization by penalizing large weights.

Table 4.2 – Hyperparameters for training Bert

Parameter	**Value**
output_dir	./results
evaluation_strategy	epoch
learning_rate	2e-5
per_device_train_batch_size	8
per_device_eval_batch_size	8
num_train_epochs	5
weight_decay	0.01
logging_dir	./logs

Model architecture

Figure 4.5 illustrates the architecture of a BERT-based binary classification model, divided into three main stages: Input Processing, BERT Processing, and the Classification Head. In the Input Processing phase, raw text is first tokenized into subword units, then converted to numerical token IDs that the BERT model can understand. Special tokens such as CLS and SEP are added to signal the beginning and end of input sequences. Padding is then applied to ensure that all inputs are the same length, and an attention mask is created to help the model ignore padded positions during processing. The prepared input is passed into the BERT Processing block, where it first goes through an embedding layer, followed by 12 transformer layers that generate contextual hidden states for each token. From these, the pooled output, typically derived from the CLS token, is extracted to represent the whole sequence. This pooled output is sent into the Classification Head, which includes a dropout layer with a dropout rate of 0.3 to prevent overfitting, a linear layer that maps features to output dimensions, and a SoftMax function that converts the output into probability scores. The final result is a binary classification output indicating either an error or no error, which can be applied to tasks such as error detection or sentiment analysis.

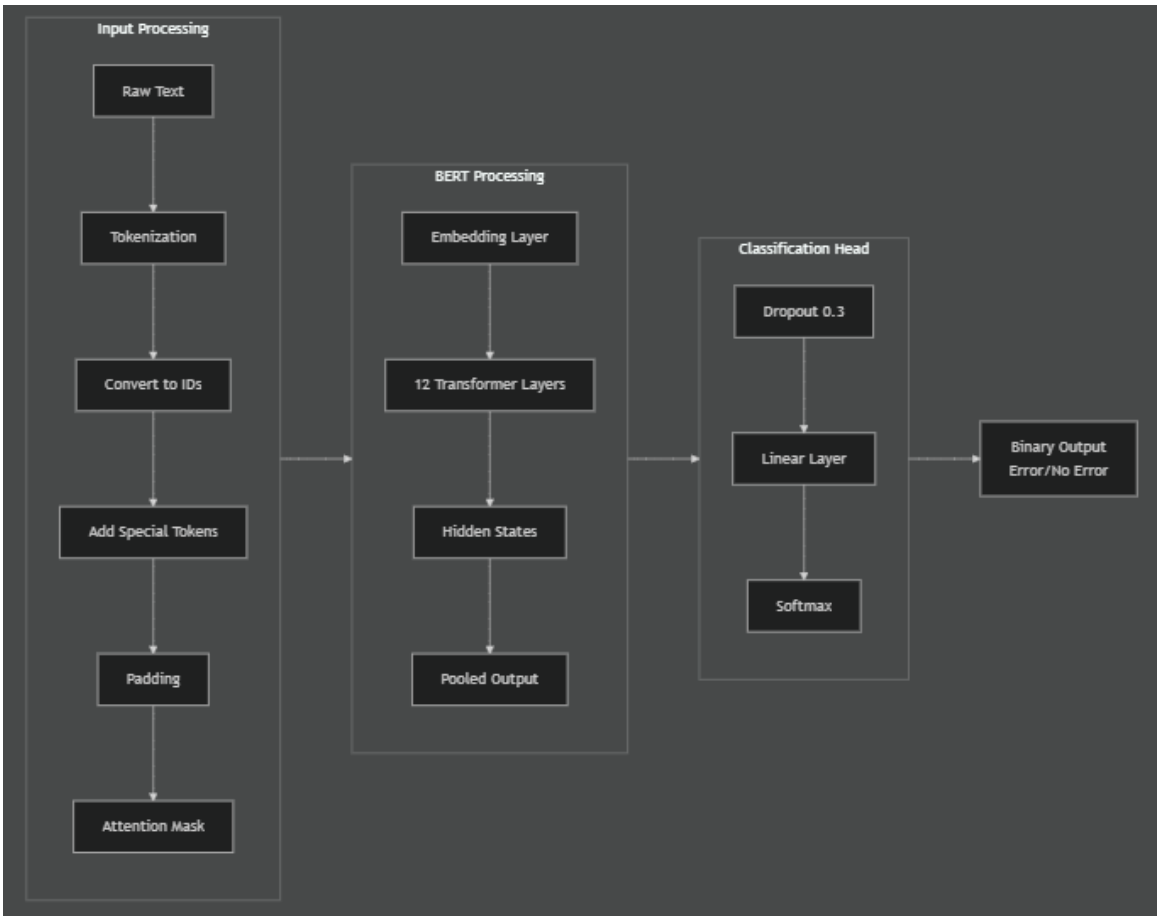


Figure 4.5 – Model architecture

Coordination of documents is an integral part of the work of an enterprise. The main goal of managing this process is to optimize decisions made, reduce risks and improve the final result [96].

Document approval is the most frequently requested business process in any organization. The main goal of the business process for the organization is to control and track the timing (dates) of approvals.

Examples of agreement:

1) The sales department is preparing a draft contract for the supply of a large batch of products manufactured by the enterprise. The legal director denies the allegation. This not only improves the quality of the contract, but also insures the company against long-term troubles: claims from the counterparty and other.

2) Preparation of tender documentation. Each interested department of the organization must approve or reject the draft documents that form the contractual basis.

3) Coordination of the procurement process. The purchase of goods and services by any organization requires a vote by the tender commission. Each committee member must vote to agree or disagree with the selection of the supplier company. The result of the approval process is shown to the accounting department for the payment process.

4) Each organization approves or re-approves internal documents, charters, regulations, strategies that require mandatory approval by top management and shareholders.

Most companies are faced with the task of approving a document. The following stages of the document approval process are proposed:

1) Start of the process. Approval of a document indicating the type of document -> The initiator fills out the main form for approval

2) Preparation of the document -> the form is filled in with data in accordance with the template for the type of document (for references -> number, executor, reference text, from (whom), to whom (whom), execution date, for contracts -> contract number, conclusion date contract, text of the agreement, electronic version of the document, etc.). Depending on the type of document, the period within which the participants in the process must agree on the document is automatically determined: agreement - 5 working days, order - 3 working days, instruction - 1 working day, memo - 6 working days, etc.

3) Identification of the immediate superiors of the initiator for approval. The document is assigned the status "Under approval by the manager". The initiator's manager either rejects the contract, adding comments, or agrees. In the first case, the contract is sent to the initiator for revision; in the second case, it moves to the next stage and receives the status "By agreement with the performers."

4) Approval of the document by the head of the contractor's department. The performer has been selected. In parallel, an execution subprocess is launched for all executors.

5) After the functions of all participants are completed, the result is calculated.

6) Notifying managers about execution.

7) Registration of documents. Assigning a number in accordance with the classifier and nomenclature of the organization's affairs.

8) Ending the process

All stages of the business process of approval and approval of documents must be audited, documents are archived, and the digital archive system allows you to search and find all the necessary information about the document. A special feature is that at each stage one can download, print and return the document for revision.

Process of document signing

To add additional layer of security for document approval, the ability to approve using an EDS (electronic digital signature) was added. For this purpose, methods for verifying and signing data using RSA keys were added and created. To sign a document, the user clicks on the "Sign with digital signature" button, after which a modal appears to confirm the action (see Figure 4.6).

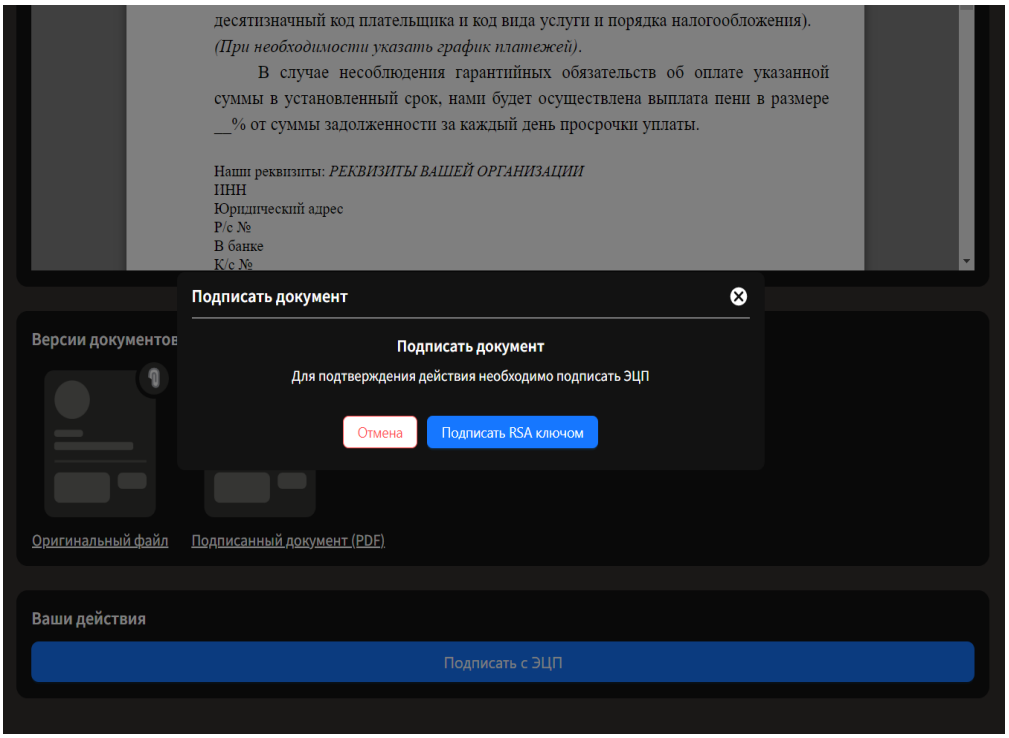


Figure 4.6 – Modal window for signing a document

After clicking on the “Sign with RSA key” button, the “handleStoreSelect” function is executed. This function calls a file selection window with the extension “ p 12” only if the connection to the socket is successful wss :// localhost :13579 (see Figure 4.7) .

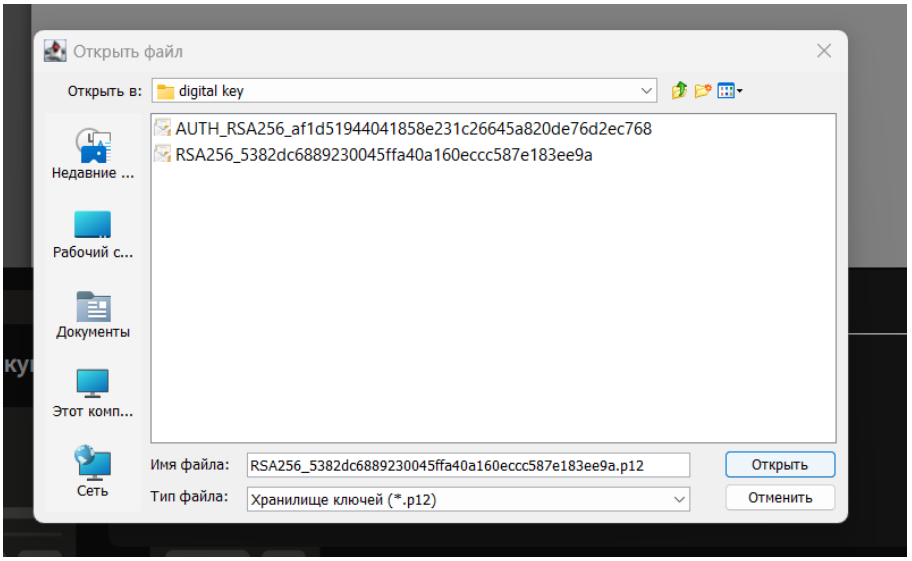


Figure 4.7 – Digital signature file selection window

After selecting the correct digital signature of the RSA file, a form appears where one needs to enter the password for the digital signature (see Figure 4.8).

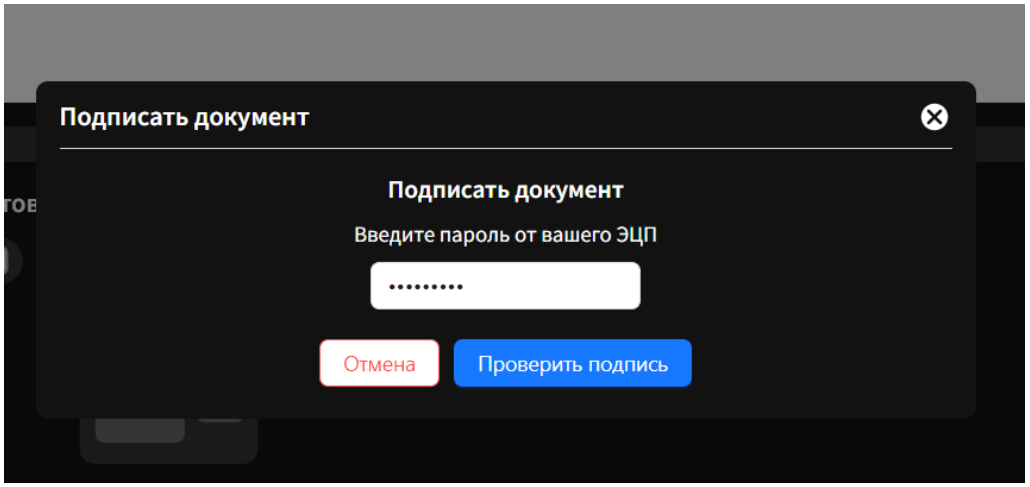


Figure 4.8 – Digital signature file selection window

After clicking on the “Sign document” button, a request is made to the server to sign the document. After signing complete, all participants in the document approval process have the opportunity to download the signed document (Figure 4.9).

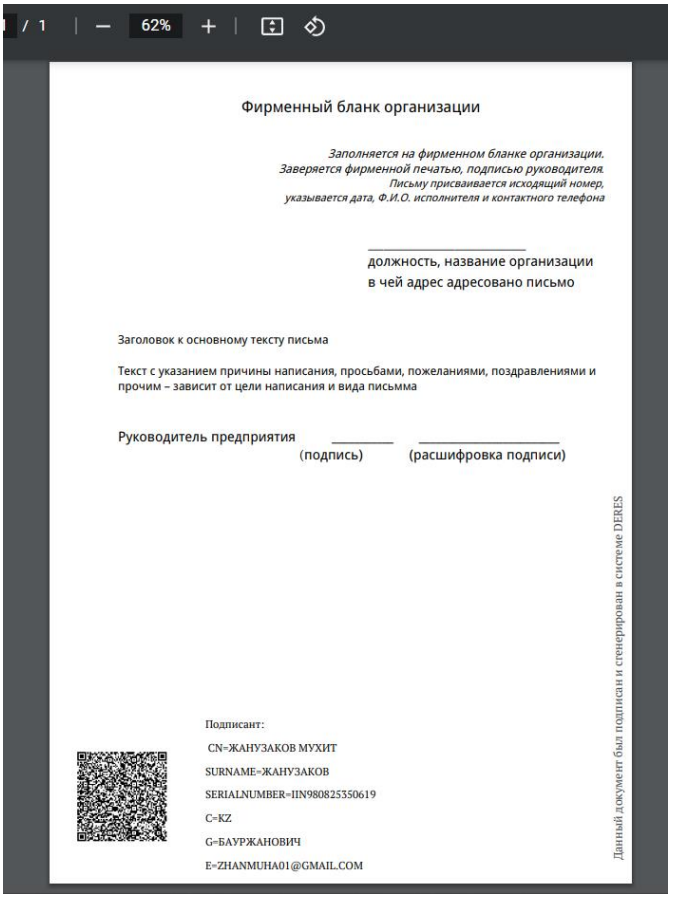


Figure 4.9 – Generated document with digital signatures

Generating a document template using generative AI

The developed administrative activity subsystem implements the generation of document templates using AI. This function is recommended only for generating

internal documents at the request of the enterprise.

To generate, you need to click the “Generate template using artificial intelligence” button, then a window will appear with the “Describe the document” field. After entering a description of the document, you must click “Generate”. Then a document template will appear below (see Figure 4.10).

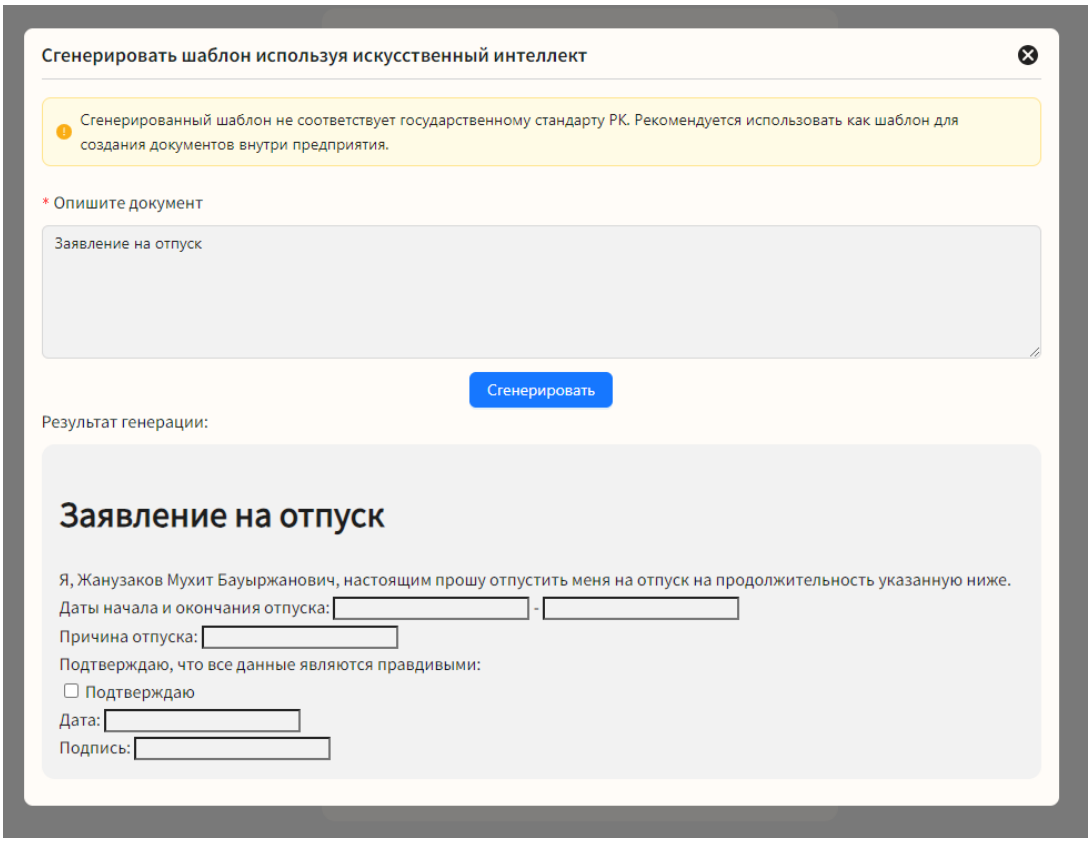


Figure 4.10 - Window for generating a document template with AI

This function allows you to generate a document template from a description using the AI model Chat GPT 3.5. This speeds up the document creation process and is one of the advantages of the developed system.

To connect to Open Ai, we have the API the key that was used in the connection configuration . Next, when entering and clicking on the “Generate” button, we sent the following parameters:

```
{
        model: “text-davinci-003”,
        prompt: ` Create a document template in html format for employee ${user.fullName} using the following description: ${description}.
        max_tokens: 1000,
        temperature: 0.4
}
```

The model parameter specifies the model type. In our case, the “ text - davinci - 003” model is perfect .

text-davinci-003 includes the following improvements:

– This ensures higher quality writing. This will help your apps deliver clearer, more engaging, and compelling content.

– It can handle more complex instructions .

– This is better suited for creating content in longer forms, allowing to accomplish tasks that would previously have been too difficult to achieve.

In order for the model to correctly generate the template, we added a template to the prompt field in advance.

The max_tokens field specifies the maximum number of tokens to be generated upon completion . In our case, 1000 will be enough to generate a template.

The temperature field specifies which sample temperature to use, from 0 to 2. Higher values, such as 0.8, will make the output more random , while lower values, such as 0.2, will make it more targeted and deterministic. The value 0.4 was chosen to generate documents.

4.6 Scalability and application of load-balancing strategies

In this section, we describe the development of an algorithm that improves the fault-tolerance of the enterprise information systems by adding additional functionality for saving connection ids to shared disk. This will help recovering the lost connections and redistributing them to healthy connections.

Problem of load-balancing algorithms is the absence of out of the box solution for handling incomplete connections when server is crashed all its connections are lost. Round Robin (RB) with Health checks is a simple yet effective load-balancing algorithm that handles server crashes [97]. The main drawback of this solution is that any requests or connections that were handled by the crashed server are completely lost and terminated. Thus we propose a new algorithm that redirects all connections to healthy servers increasing effectiveness of the system.

Round Robin with Health Checks is a fault-tolerant load balancing algorithm that distributes incoming requests across a pool of servers in a circular manner, ensuring each server gets an equal share of the workload. When combined with health checks, it becomes capable of detecting and responding to server failures or unresponsiveness.

It maintains a list or pool of available servers. It initializes a pointer to keep track of the current server in the rotation. When a new request arrives, directs it to the server currently pointed to by the pointer. Increments the pointer to point to the next server in the pool until the end of the pool.

The algorithm periodically performs health checks on each server in the pool to assess its availability and responsiveness. If a server is detected as unhealthy, marks it as such. When a server is marked as unhealthy, it is removed from the pool of available servers. It adjusts the rotation so that the pointer skips over the unhealthy server when distributing requests. It continuously monitors the health of unhealthy servers. Once a

unhealthy server is detected, remove it from the pool of available servers (Figure 2).

Our algorithm modifies the Round Robin with Health Checks by introducing caching of connection ip addresses to the shared disk. When a new request comes in, the algorithm assigns server to serve the connection in Round Robin fashion, however, before the server is assigned to handle request, the request/connection IP is saved in a hash table (key is the server id, value is an array of connection IPs).

When health checking, the process is similar to Round Robin with Health Checks, however, when server is found unhealthy, all its currently served connections are redirected to healthy servers in Round Robin way (see Figure 4.11).

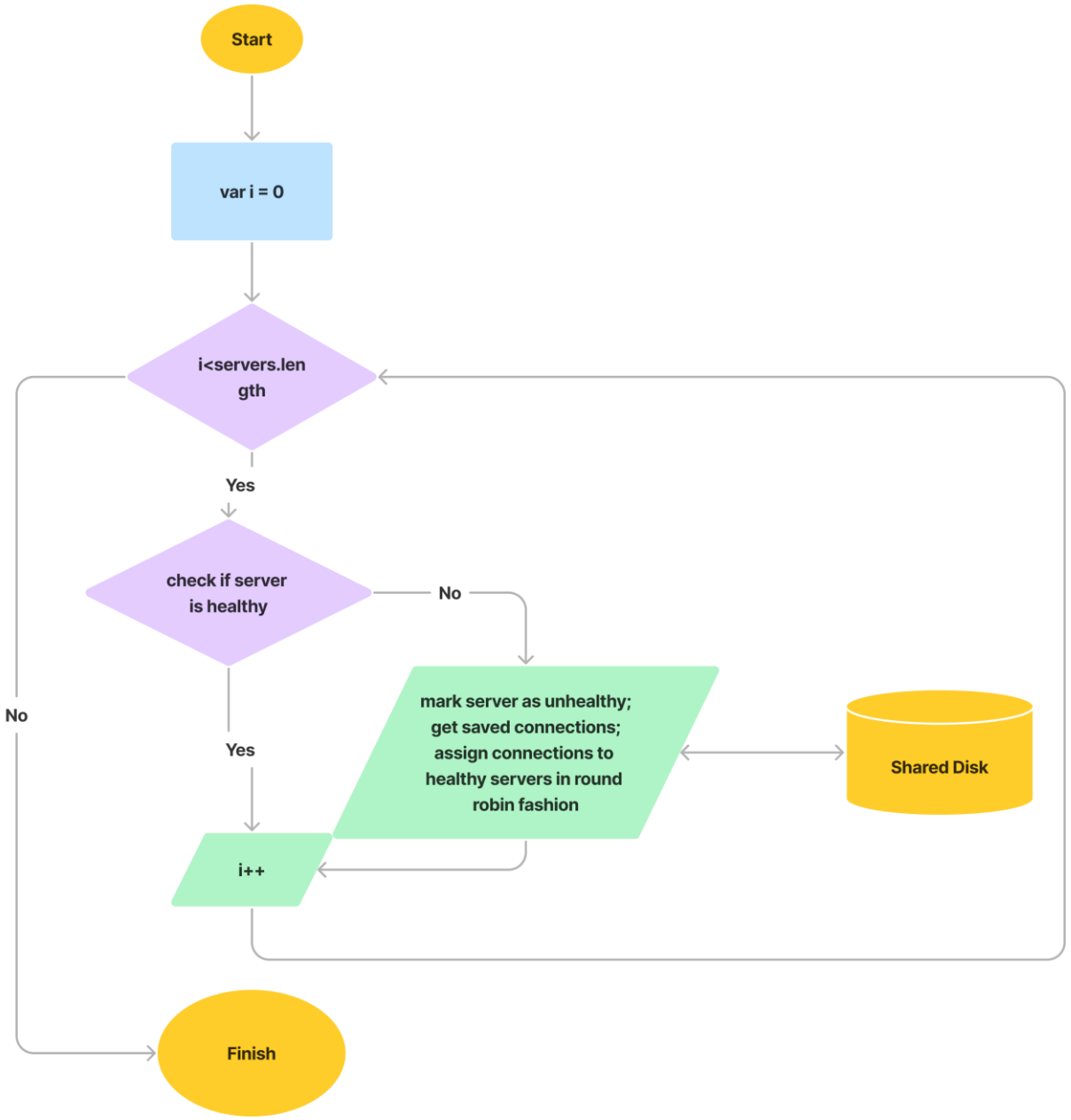


Figure 4.11 – Improved health checking block-scheme

When any connection is complete (server finished serving client), the connection ip is removed from the disk.

Complexity analysis

The complexity of Round Robin with Health Checks algorithm is linear - O (N) for both assigning server and health checking. Because, in the worst case, the algorithm needs to iterate over N servers for finding healthy server or unhealthy in the case of Health Checking.

Our algorithm is also linear for assigning server, as saving IP address to disk is considered constant time. However, when health checking, if one or more servers are found unhealthy, the connections (K) should be redistributed among healthy servers (N) in Round Robin fashion. Thus, its complexity in the worst case scenario would be O (N*K). But in the best case, when no server is defected, the complexity stays linear.

Measurement metrics

In order to measure the algorithms ability to handle incomplete requests, we introduce the following metrics.

1. Connection Termination Rate. It measures the percentage of connections that were terminated prematurely due to a server failure. This metric gives an indication of how effectively the algorithm handles incomplete requests. Lower termination rates indicate better handling of incomplete requests.

Connection Termination Rate (CTR) typically refers to the rate at which connections are terminated or closed within a system or network. It's often expressed as the number of connections terminated per unit of time.

$$CTR = \frac{Number\ of\ terminated\ connections}{Duration} \tag{4.1}$$

2. Probability of termination. It shows how likely the connection will be terminated.

$$P_{termination} = \frac{Terminated\ connections}{Total\ connections} \tag{4.2}$$

3. Response Time Distribution. We analyze the distribution of response times for requests that were affected by a server failure. It will show patterns such as increased latency or variability in response times during and after a failure event.

4. Service rate. How many connections can the system handle per second including timed out connections. More value means faster served connections.

In order to assess the proposed algorithm, we run a series of simulations for Round Robin with Health Checks and our solution. The simulation is done using 4 virtual servers and a proxy server ngnix. For load simulation an application called artillery was used. It creates virtual users that try to access our server in a set rate.

In our case, we set arrival rate as 2 (requests per second), during 100 seconds overall 200 requests were made, the health checking was done every 5 seconds. The simulation is done for both algorithms.

Server number 3 is shut down during each simulation after 30 seconds from the start to the program. This allowed to see how each algorithm handle the requests

assigned to crashed server.

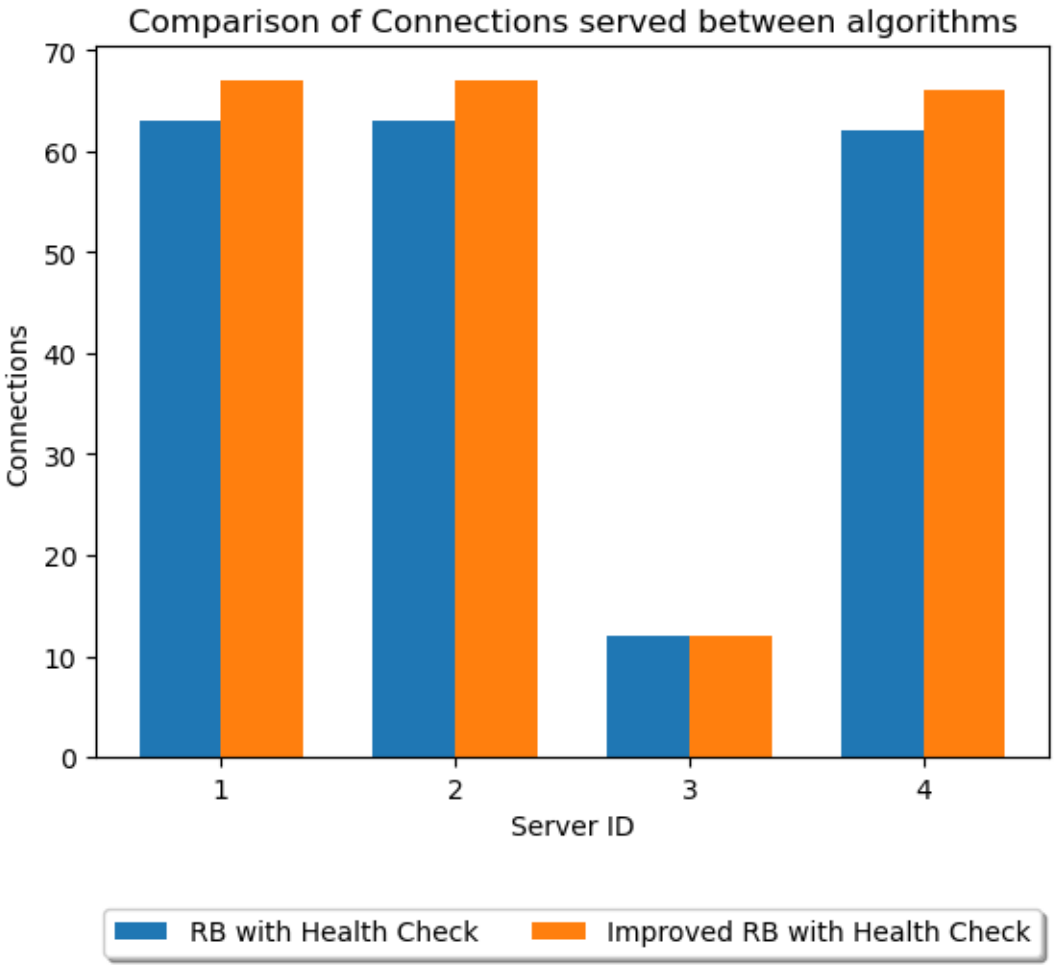


Figure 4.12 – Comparison of algorithms by the number of connections served (including incomplete connections)

In Figure 4.12, one can see that the server number 3 crashed after receiving 12 connections as planned. The RB with Health Checks terminated these requests after server crash. On the other hand, improved RB recovered these connections, redirected them to healthy servers. This is why server in our algorithm served more connections comparing to RB with Health Checks.

Both algorithms classic RB with Health Checks and our solution were assessed using CTR and probability of termination. The results are displayed in Table 4.3.

Table 4.3 - Comparison of load-balancing algorithms

Algorithm	Service rate (connections/second)	CTR	Probability of termination	Terminated connections	Total served connections
RB with Health Check	0.28	0.12	0.06	12	188

Continuation of Table 4.3

Improved RB with Health Check	171.6	0	0	0	200

As seen in Table 4.3, the improved RB showed much better results when handling the terminated connections due to the mechanism for saving, redirecting, recovering such connections. The reason the service rate of improved RB being extremely higher is the inclusion of timed out connections value. As known the default time out connection value in ngnix is 60. That is why all terminated connections take more than 60 seconds to complete if they are not handled by other servers. In conclusion, the proposed algorithm for handling incomplete connections showed better results in all measurements.

4.7 Description of technologies used in the development of enterprise system

The developed enterprise system is based on a set of modern, industry-proven technologies selected to ensure high levels of scalability, fault tolerance, performance, and extensibility. Each technology fulfills a specific functional role within the overall system architecture.

Containerization and Orchestration.

The system employs Docker and Kubernetes to support containerized deployment and automated infrastructure management. Docker enables the encapsulation of application components and their dependencies into isolated, lightweight containers, thereby ensuring environmental consistency across development, testing, and production stages. Kubernetes extends this capability by providing orchestration mechanisms, including automated deployment, scaling, service discovery, and self-healing [98]. This combination significantly enhances system reliability and simplifies the management of distributed services.

In addition, containerization facilitates modular system design by enabling the decomposition of the enterprise application into loosely coupled microservices. Each service can be independently developed, deployed, and updated without affecting other system components, which is essential for maintaining system flexibility and accelerating development cycles. Docker images serve as immutable artifacts, ensuring reproducibility and version control of application environments, which is particularly important in large-scale distributed systems.

Kubernetes further strengthens system robustness through advanced resource management and scheduling mechanisms. It dynamically allocates computational resources based on workload demands, thereby optimizing infrastructure utilization. Features such as horizontal pod autoscaling allow the system to automatically adjust the number of running instances in response to variations in incoming traffic, ensuring consistent performance under fluctuating loads. Moreover, Kubernetes provides built-in fault tolerance through automatic restarts, replication, and rescheduling of failed

containers across nodes, minimizing system downtime.

Another important aspect is the support for declarative configuration and infrastructure as code. Kubernetes enables the definition of system states using configuration files, which enhances maintainability, traceability, and automation of deployment processes. This approach also facilitates continuous integration and continuous deployment (CI/CD) pipelines, allowing rapid and reliable delivery of system updates.

Distributed Messaging and Event Streaming.

For asynchronous communication and event-driven processing, the system integrates Apache Kafka. Kafka serves as a high-throughput, fault-tolerant messaging backbone that decouples system components and enables real-time data streaming [99, 100]. Its partitioned log architecture ensures durability and scalability, while its publish–subscribe model facilitates efficient data exchange between microservices in the enterprise environment.

Kafka enables loose coupling between producers and consumers, allowing system components to operate independently without direct dependencies. This architectural approach enhances system flexibility and simplifies the integration of new services, which is particularly important in dynamically evolving enterprise environments. By persisting messages in distributed logs, Kafka guarantees data durability and supports replayability, enabling the system to reprocess events for recovery, auditing, or analytical purposes.

Kafka's distributed design provides horizontal scalability through topic partitioning, allowing data streams to be processed in parallel across multiple nodes. This ensures that the system can handle large volumes of data with low latency, which is critical for real-time analytics and decision-making processes. Additionally, built-in replication mechanisms enhance fault tolerance by maintaining multiple copies of data across different brokers, thereby preventing data loss in the event of node failures.

Another significant advantage of Kafka is its support for stream processing frameworks and integration with data processing pipelines. It can serve as a central data hub, enabling seamless interaction between transactional systems, analytical platforms, and machine learning modules. This capability facilitates the implementation of event-driven architectures, where system behavior is determined by continuous streams of events rather than synchronous request–response interactions.

Data Storage Technologies.

A hybrid data storage approach is implemented using PostgreSQL and Apache Cassandra. PostgreSQL is utilized for transactional data management, providing strong consistency, ACID compliance, and support for complex queries. In contrast, Cassandra is employed for handling large-scale, distributed datasets requiring high availability and horizontal scalability. This combination enables the system to efficiently process both transactional and analytical workloads, aligning with the requirements of modern enterprise applications.

The use of PostgreSQL ensures reliable management of structured data, particularly in scenarios requiring strict data integrity, referential constraints, and transactional consistency [101]. Its advanced indexing mechanisms, support for stored procedures, and extensibility features make it suitable for implementing core business

logic and executing complex analytical queries. Furthermore, PostgreSQL supports JSON and semi-structured data types, allowing a certain degree of flexibility within a relational model, which is advantageous in enterprise systems with evolving data schemas.

Cassandra is designed to operate in distributed environments where large volumes of data must be processed with minimal latency and without a single point of failure. Its decentralized architecture, based on a peer-to-peer model, ensures continuous availability even in the presence of node failures. Data is automatically replicated across multiple nodes and data centers, providing fault tolerance and supporting geographically distributed deployments. This makes Cassandra particularly effective for applications involving high write throughput, real-time data ingestion, and large-scale logging or event storage.

The integration of these two database paradigms reflects the polyglot persistence approach, where different data storage technologies are employed based on specific use-case requirements. Transactional operations and critical business data are handled by PostgreSQL, while high-volume, time-series, or unstructured data streams are managed by Cassandra. This separation of concerns enhances system performance and scalability while maintaining data consistency where necessary.

The hybrid storage architecture supports data lifecycle management and analytical processing. Data generated and stored in Cassandra can be periodically aggregated, transformed, and transferred to PostgreSQL or other analytical systems for reporting and decision-making purposes. This enables the implementation of both operational and analytical workflows within a unified enterprise system.

Machine Learning Frameworks.

The intelligent components of the system are developed using Python in conjunction with TensorFlow and scikit-learn [102]. Python provides a flexible and expressive programming environment, while TensorFlow supports the implementation of deep learning models and scalable numerical computations. Scikit-learn is utilized for classical machine learning algorithms, including classification, regression, and clustering. Together, these tools enable the development of predictive and analytical modules within the enterprise system [103].

Load Balancing and Reverse Proxying.

To ensure efficient request distribution and high availability, the system incorporates NGINX and HAProxy. NGINX functions as a reverse proxy, handling incoming client requests and routing them to appropriate backend services. HAProxy provides advanced load balancing capabilities, including fine-grained traffic distribution, health checks, and failover mechanisms. The combined use of these technologies improves system responsiveness and resilience under varying workloads.

Monitoring and Observability.

System monitoring and performance analysis are achieved Prometheus and Grafana. Prometheus collects and stores time-series metrics from system components, enabling real-time monitoring and alerting. Grafana complements this functionality by providing interactive dashboards and visualization tools for analyzing system performance, detecting anomalies, and supporting decision-making processes.

The integration of these technologies establishes a robust and flexible

architecture capable of supporting large-scale enterprise applications. Their combined use ensures high fault tolerance, efficient resource utilization, and adaptability to evolving business and technological requirements.

In conclusion, this section described development of intelligent enterprise system that has been developed as a prototype system for testing the integration of the developed fault-tolerance algorithms and mechanisms in a real-world application. During development of the system, new models for information validation were developed. The system also integrated modified Round Robin with Health Checks load-balancing algorithm.

CONCLUSION

The research presented in this dissertation addresses the critical challenge of ensuring fault-tolerant and reliable information processing in distributed enterprise systems. The increasing load on corporate servers and the complexity of modern enterprise environments create high risks of system failures, which can significantly disrupt operations, compromise data integrity, and impact organizational performance. This study has developed, implemented, and evaluated novel solutions for calculating the optimal checkpointing interval for rollback-recovery algorithms, that aim to balance system reliability, performance, and efficiency.

Two approaches were proposed: a model-based empirical algorithm and a queuing-theory-based optimal algorithm. Experimental evaluation in virtualized and physical environments demonstrated their respective strengths and trade-offs. The empirical algorithm achieved the highest throughput (0.0471 jobs/s in virtualized, 0.0426 jobs/s in physical environments) and the lowest average service time (10.775 s and 12.89 s, respectively) due to an aggressive checkpointing strategy (20,630 and 18,840 checkpoints). However, this approach introduced notable overhead, slightly reducing availability (90.37% and 93.12%) and system efficiency (50.73% and 55.91%). In contrast, the queuing-theory-based algorithm maintained a balanced checkpointing strategy (11,970 and 10,890 checkpoints), achieving competitive throughput (0.0424 and 0.0391 jobs/s), low average service time (11.562 s and 12.89 s), and the highest availability (93.99% and 96.27%) and system efficiency (62.77% and 66.84%) across both environments. These results demonstrate its superior ability to balance performance, fault tolerance, and resource utilization under realistic operational constraints.

The research results confirm that the proposed algorithms enhance system resiliency by ensuring rapid rollback recovery, maintaining service continuity during failures, and reducing recomputation losses. Additionally, these methods are practical for small and medium enterprises seeking to implement reliable rollback-recovery mechanisms without imposing substantial performance penalties. Beyond traditional fault tolerance, the dissertation also explored the integration of AI and Big Data tools to improve enterprise processes, demonstrating the applicability of intelligent methods for validating and automating document generation, further enhancing operational reliability and efficiency.

In summary, the study makes a significant contribution to the theory and practice of fault-tolerant distributed systems by developing optimized checkpointing strategies, providing empirical and analytical validation, and demonstrating their applicability in realistic enterprise settings. The queuing-theory-based optimal checkpointing algorithm, in particular, offers a robust, scalable, and efficient solution suitable for deployment in high-load, failure-prone environments, thereby supporting the creation of resilient digital ecosystems for modern enterprises.

REFERENCES

1. Enterprise Server Market - Share, Trends & Analysis (Mordor Intelligence). Available: https://www.mordorintelligence.com/industry-reports/enterprise-servers-market. Accessed: Feb. 2, 2025.
2. Uptime Institute, "Annual Outage Analysis 2023." Available: https://datacenter.uptimeinstitute.com. Accessed: Feb. 2, 2025.
3. R. Koo and S. Toueg, "Checkpointing and rollback-recovery for distributed systems," *IEEE Transactions on Software Engineering*, vol. SE-13, no. 1, pp. 23-31, Jan. 1987, doi: 10.1109/TSE.1987.232562.
4. E. Dubrova, "Fault-tolerant design," *New York: Springer*. vol. 8, 2013, doi:10.1007/978-1-4614-2113-9
5. *V. Mbandu, "Enhancing Fault Tolerance in Web Application Using Weighted Round-Robin Algorithm,". Multidisciplinary Research Conference, p.39, 2018.*
6. M. Hasan, M.S. Goraya, "Fault tolerance in cloud computing environment: A systematic survey," *Computers in Industry*, vol. 99, pp.156–172, 2018, doi:10.1016/j.compind.2018.03.027
7. V. Mbandu, "Enhancing Fault Tolerance in Web Application Using Weighted Round-Robin Algorithm". Multidisciplinary Research Conference, p.39, 2018.
8. P. Kumari, P.Kaur, "A survey of fault tolerance in cloud computing,". Journal of King Saud University – Computer and Information Sciences, vol.33, no.10 pp.1159–1176, 2021, doi:10.1016/j.jksuci.2018.09.021
9. M.Kirti, A.K Maurya, R.S Yadav, "Fault-tolerance approaches for distributed and cloud computing environments: A systematic review, taxonomy and future directions," *Concurrency and Computation: Practice and Experience*. Advance online publication, 2024, doi:/10.1002/cpe.8081
10. M. A. Shahid, N.Islam, M.M. Alam, M.S.Mazliham, S. Musa, "Towards resilient method: An exhaustive survey of fault tolerance methods in the cloud computing environment," Computer Science Review, vol.40, 100398, 2021, doi:10.1016/j.cosrev.2021.100398
11. M.Ansari, M. Salehi, S. Safari, A. Ejlali, M.Shafique, "Peak-power-aware primary-backup technique for efficient fault-tolerance in multicore embedded systems," *IEEE Access*, vol.8, pp.142843 – 142857, doi: 10.1109/access.2020.3013721
12. M.Vargas-Santiago. *et.al.*, "Survey on Web Services Fault Tolerance Approaches Based on Checkpointing Mechanisms" *J. Softw.*, vol.12, no.7, pp. 507–525, 2017, doi: 10.17706/jsw.12.7.507-525
13. T. Distler, "Byzantine fault-tolerant state-machine replication from a systems perspective" *ACM Computing Surveys (CSUR)*, vol. 54, no.1, pp.1–38, 2021, https://doi.org/10.1145/3436728
14. S.M. Attallah, M.B.Fayek, S.M. Nassar, E.E.Hemayed, "Proactive load balancing fault tolerance algorithm in cloud computing," *Concurrency and*

Computation: Practice and Experience, vol.33, no.10, 2021, doi:10.1002/cpe.6172

15. C.Stathakopoulou, M. Pavlovic, M.Vukolić, "State machine replication scalability made simple", *In Proceedings of the Seventeenth European Conference on Computer Systems,* pp.17–33, 2022, doi:10.5281/zenodo.6344032
16. J.Nikolić, N. Jubatyrov, E. Pournaras, "Self-healing dilemmas in distributed systems: Fault correction vs. fault tolerance" IEEE *Transactions on Network and Service Management*, vol.18, no.3, pp. 2728–2741, 2021, doi: 10.1109/tnsm.2021.3092939
17. H. Howard, R. Mortier, "Paxos vs Raft: Have we reached consensus on distributed consensus?," *Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data*, pp.1–9, 2020, doi:10.1145/3380787.3393681
18. R. S. R. Putri. *et.al.,* "Implementation of Fault-Tolerance Mechanism in Quorum-Based Blockchain Provisioning in Cloud Infrastructure Using Replication and Monitoring Protocols,". *Proceedings of the 8th International Conference on Sustainable Information Engineering and Technology*, pp. 311–322, 2023, doi:10.1145/3626641.3626673
19. G. Balakayeva. et.al., "Digitalization of enterprise with ensuring stability and reliability," Informatyka, Automatyka, Pomiary w Gospodarce i Ochronie Srodowisk, vol.13, no.1, pp. 54–57, 2023, doi:10.35784/iapgos.3295
20. G. Balakayeva. *et.al.,* "Development of a software system for predicting employee ratings,"*Informatyka, Automatyka, Pomiary w Gospodarce i Ochronie Srodowisk*, vol.13, no.3, pp. 121–124, 2023, doi:10.35784/iapgos.3723
21. T. J.Shimeall, N. Leveson, "An empirical comparison of software fault tolerance and fault elimination," *IEEE Transactions on Software Engineering*, vol.17, 1991.
22. R.Roveicy, A.M. Bidgoli, "Comparative Analysis of Fault Tolerance Techniques in Cloud Computing: A Case of Armangerayan Co" *International Journal of Engineering & Technology,* vol.7, no.3, pp. 432–436, 2018, doi:10.14419/ijet.v7i3.14.18834
23. H.Agarwal, A. Sharma, "A Comprehensive Survey of Fault Tolerance Techniques in Cloud Computing," *Conference on Computing and Network Communications (CoCoNet'15),* 2015.
24. A.Ganesh, M. Sandhya, S. Shankar, "A Study on Fault Tolerance methods in Cloud Computing," *IEEE International Advance Computing Conference (IACC),*2016.
25. H.Ledmi, *et.al.,* "Fault Tolerance in Distributed Systems: A Survey," *3rd International Conference on Pattern Analysis and Intelligent Systems (PAIS), Tebessa, Algeria,* pp.1–5, 2018, doi:10.1109/PAIS.2018.8598484.
26. A.Tchana, *et.al.,* "Approaches to cloud computing fault tolerance". *Computer, Information and Telecommunication Systems (CITS), 2012 International Conference IEEE,* pp. 1–6, 2012.

27. S.Kumar, *et.al.,* “Fault Tolerance and Load Balancing algorithm in Cloud Computing: A survey” *International Journal of Advanced Research in Computer and Communication Engineering,* vol.4, no.7, 2015.
28. Beniwal P., Garg A. A comparative study of static and dynamic load balancing algorithms //International journal of advance research in computer science and management studies. – 2014. – Т. 2. – №. 12. – С. 1-7.
29. S. Aslam and M. A. Shah, "Load balancing algorithms in cloud computing: A survey of modern techniques," 2015 National Software Engineering Conference (NSEC), Rawalpindi, Pakistan, 2015, pp. 30-35, doi: 10.1109/NSEC.2015.7396341. keywords: {Load management;Heuristic algorithms;Cloud computing;Round robin;Servers;Classification algorithms;Time factors;cloud computing;load balancing;algorithms;load balancers},
30. Pradhan, Pandaba, Behera, Prafulla Ku., Ray, B.N.B. "Modified Round Robin Algorithm for Resource Allocation in Cloud Computing." *Procedia Computer Science*, vol. 85, 2016, pp. 878-890. ISSN: 1877-0509, https://doi.org/10.1016/j.procs.2016.05.278.
31. S. Chowdhury and A. Katangur, "Threshold Based Load Balancing Algorithm in Cloud Computing," 2022 IEEE International Conference on Joint Cloud Computing (JCC), Fremont, CA, USA, 2022, pp. 23-28, doi: 10.1109/JCC56315.2022.00011.
32. S. Malik, “Dynamic Load Balancing in a Network of Workstation”, 95.515 Research Report, 19 November, 2000.
33. Ghomi E. J., Rahmani A. M., Qader N. N. Load-balancing algorithms in cloud computing: A survey //Journal of Network and Computer Applications. – 2017. – Т. 88. – С. 50-71.
34. Vashistha J., Jayswal A. K. Comparative study of load balancing algorithms //IOSR Journal of Engineering. – 2013. – Т. 3. – №. 3. – С. 45-50.
35. Radojević B., Žagar M. Analysis of issues with load balancing algorithms in hosted (cloud) environments //2011 Proceedings of the 34th international convention MIPRO. – IEEE, 2011. – С. 416-420.
36. Xu M., Tian W., Buyya R. A survey on load balancing algorithms for virtual machines placement in cloud computing //Concurrency and Computation: Practice and Experience. – 2017. – Т. 29. – №. 12. – С. e4123.
37. Zhanuzakov M., Balakayeva G. A comparative study of load-balancing algorithms for reliability of enterprise information systems // Bulletin of Abai KazNPU. Series of Physical and Mathematical Sciences. – 2024. – Vol. 87, No. 3. – P. 146–153. – DOI: 10.51889/2959-5894.2024.87.3.012.
38. Tiwari, Devesh & Gupta, Saurabh & Vazhkudai, Sudharshan. (2014). Lazy Checkpointing: Exploiting Temporal Locality in Failures to Mitigate Checkpointing Overheads on Extreme-Scale Systems. Proceedings of the International Conference on Dependable Systems and Networks. 25-36. Doi: 10.1109/DSN.2014.101.
39. Frank A., Baumgartner M., Salkhordeh R., Brinkmann A. Improving checkpointing intervals by considering individual job failure probabilities //

Proceedings of the 35th IEEE International Parallel and Distributed Processing Symposium (IPDPS). — Portland, OR, USA, 2021. — P. 299–309. — DOI: 10.1109/IPDPS49936.2021.00038.
40. J.T. Daly, A higher order estimate of the optimum checkpoint interval for restart dumps, Future Generation Computer Systems, Volume 22, Issue 3, 2006, Pages 303-312, ISSN 0167-739X, https://doi.org/10.1016/j.future.2004.11.016.
41. Q. Wang, Y. Liu, T. Xu, X. Sun and J. Li, "Design of Software Fault Tolerant System for Autonomous Underwater Vehicles," 2021 IEEE 4th Advanced Information Management, Communicates, Electronic and Automation Control Conference (IMCEC), Chongqing, China, 2021, pp. 1456-1460, doi: http://doi.org/10.1109/IMCEC51613.2021.9482037.
42. Y. Liu, Y. Zhou, H. Zhang, Z. Chang, S. Xu, Y. Jia, W. Wang, and Z. Liu, "Rethinking software misconfigurations in the real world: An empirical study and literature analysis," *arXiv*, 2024. [Online]. Available: https://arxiv.org/abs/2412.11121
43. S. Isukapalli and S. Srirama, "A systematic survey on fault-tolerant solutions for distributed data analytics: Taxonomy, comparison, and future directions," *Comput. Sci. Rev.*, vol. 53, 2024, http://doi.org/10.1016/j.cosrev.2024.100660.
44. P. K. Jaggi and A. K. Singh, "Rollback recovery with low overhead for fault tolerance in mobile ad hoc networks," *Journal of King Saud University - Computer and Information Sciences*, vol. 27, no. 4, pp. 402-415, 2015.
45. S. Kumar T, M. HS, S. S. Mustapha, P. Gupta, and R. P. Tripathi, "Intelligent fault-tolerant mechanism for data centers of cloud infrastructure," *Mathematical Problems in Engineering*, vol. 2022, no. 1, p. 2379643, 2022.
46. M. Kirti, A. K. Maurya, and R. S. Yadav, "Fault-tolerance approaches for distributed and cloud computing environments: A systematic review, taxonomy and future directions," *Concurrency and Computation: Practice and Experience*, vol. 36, no. 13, p. e8081, 2024.
47. X. Wang, C. Zhang, J. Fang, R. Zhang, W. Qian, and A. Zhou, "A comprehensive study on fault tolerance in stream processing systems," *Frontiers of Computer Science*, vol. 16, pp. 1-18, 2022.
48. S. S. Nabavi and H. Farbeh, "A fault-tolerant resource locking protocol for multiprocessor real-time systems," *Microelectronics Journal*, vol. 137, p. 105809, 2023.
49. A. U. Rehman, R. L. Aguiar, and J. P. Barraca, "Fault-tolerance in the scope of cloud computing," *IEEE Access*, vol. 10, pp. 63422-63441, 2022.
50. S. Khan, I. A. Shah, K. Aurangzeb, S. Ahmad, J. A. Khan, and M. S. Anwar, "Energy-efficient task scheduling using fault tolerance technique for IoT applications in fog computing environment," *IEEE Internet of Things Journal*, vol. 11, no. 24, pp. 39009-39019, Dec. 15, 2024, doi: 10.1109/JIOT.2024.3403003.

51. P. Kumari and P. Kaur, "A survey of fault tolerance in cloud computing," *Journal of King Saud University - Computer and Information Sciences*, vol. 33, no. 10, pp. 1159-1176, 2021, doi: 10.1016/j.jksuci.2018.09.021.
52. Zhanuzakov M., Balakayeva G., Ezhilchelvan P. Model based solution for computing checkpointing interval for fault-tolerant rollback-recovery in enterprise servers // Journal of Problems in Computer Science and Information Technologies. – 2025. – Vol. 3, No. 1. – P. 25–33. – DOI: 10.26577/jpcsit20253103.
53. J. L. Hennessy and D. A. Patterson, Computer Architecture: A Quantitative Approach, 5th ed. Elsevier, 2011. ISBN: 9780123838735.
54. G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in Proc. AFIPS Spring Joint Comput. Conf., New York, NY, USA, Apr. 1967, pp. 483–485. doi: 10.1145/1465482.1465560.
55. P. Ezhilchelvan and I. Mitrani, "Checkpointing models for tasks of different types," *ACM Trans. Model. Perform. Eval. Comput. Syst.*, vol. 1, no. 1, Jan. 2024, https://doi.org/10.1145/3658667.
56. J. F. Shortle, J. M. Thompson, D. Gross, and C. M. Harris, *Fundamentals of Queueing Theory*, 5th ed. Hoboken, NJ: Wiley, 2018.
57. L. Bautista-Gomez, A. Benoit, S. Di, T. Herault, Y. Robert, and H. Sun, "A survey on checkpointing strategies: Should we always checkpoint à la Young/Daly?," *Future Gener. Comput. Syst.*, vol. 161, pp. 315–328, 2024, https://doi.org/10.1016/j.future.2024.07.022.
58. B. M. Oki and B. H. Liskov, "Viewstamped replication: A new primary copy method to support highly-available distributed systems," in *Proc. 7th Annu. ACM Symp. Principles Distrib. Comput.*, 1988, pp. 8–17, https://doi.org/10.1145/62546.62549
59. Sutherland Global. (2024). What is BPM. Retrieved May 27, 2024, from https://www.sutherlandglobal.com/insights/technology/what-is-bpm
60. Ravesteijin, P., & Zoet, M. (2010). A BPM-systems architecture that supports dynamic and collaborative processes. *Journal of International Technology and Information Management, 19*(3), Article 1.
61. Jung, J., Choi, I., & Song, M. (2007). An integration architecture for knowledge management systems and business process management systems. Computers in industry, 58(1), 21-34.
62. Qanbar, A. A., & Algamal, Z. Y. (2024). Improving Support vector machine for Imbalanced big data classification. Journal of Intelligent Systems & Internet of Things, 11(2).
63. Al-kababchee, S., Algamal, Z. & Qasim, O. (2023). Enhancement of K-means clustering in big data based on equilibrium optimizer algorithm. Journal of Intelligent Systems, 32(1), 20220230. https://doi.org/10.1515/jisys-2022-0230
64. Al-Kababchee, S. G. M., Qasim, O. S., & Algamal, Z. Y. (2021, May). Improving penalized regression-based clustering model in big data. In Journal of Physics: Conference Series (Vol. 1897, No. 1, p. 012036). IOP Publishing.

65. Al-Thanoon, N. A., Algamal, Z. Y., & Qasim, O. S. (2021). Feature selection based on a crow search algorithm for big data classification. Chemometrics and Intelligent Laboratory Systems, 212, 104288.
66. 8. Al Kababchee, S. G., Algamal, Z. Y., & Qasim, O. S. (2023). Improving Penalized-Based Clustering Model in Big Fusion Data by Hybrid Black Hole Algorithm. Fusion: Practice & Applications, 11(1).
67. Esmaeili, M., Abbasi-Moghadam, D., Sharifi, A., Tariq, A., & Li, Q. (2023). ResMorCNN model: hyperspectral images classification using residual-injection morphological features and 3DCNN layers. IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing, 17, 219-243.
68. Akhtarmanesh, A., Abbasi-Moghadam, D., Sharifi, A., Yadkouri, M. H., Tariq, A., & Lu, L. (2023). Road extraction from satellite images using attention-assisted UNet. IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing, 17, 1126-1136.
69. Marzvan, S., Moravej, K., Felegari, S. et al. (2021). Risk Assessment of Alien Azolla filiculoides Lam in Anzali Lagoon Using Remote Sensing Imagery. J Indian Soc Remote Sens 49, 1801–1809. https://doi.org/10.1007/s12524-021-01362-1
70. Felegari, S., Sharifi, A., Khosravi, M., & Sabanov, S. (2023). Using experimental models and multitemporal Landsat-9 images for cadmium concentration mapping. IEEE Geoscience and Remote Sensing Letters, 20, 1-4.
71. Mahdipour, H., Sharifi, A., Sookhak, M., & Medrano, C. R. (2024). Ultrafusion: Optimal fuzzy fusion in land-cover segmentation using multiple panchromatic satellite images. IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing, 17, 5721-5733.
72. Safari, M. M., Sharifi, A., Mahmood, J., & Abbasi-Moghadam, D. (2024). Mesoscale eddy detection and classification from sea surface temperature maps with deep neural networks. IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing.
73. Mirhoseini Nejad, S. M., Abbasi-Moghadam, D., & Sharifi, A. (2024). ConvLSTM-ViT: A Deep Neural Network for Crop Yield Prediction Using Earth Observations and Remotely Sensed Data. IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing.
74. Farmonov, N., Esmaeili, M., Abbasi-Moghadam, D., Sharifi, A., Amankulova, K., & Mucsi, L. (2024). HypsLiDNet: 3D-2D CNN model and spatial–spectral morphological attention for crop classification with DESIS and LiDAR data. IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing.
75. Vafaeinejad, A., Alimohammadi, N., Sharifi, A., & Safari, M. M. (2025). Super-Resolution AI-Based Approach for Extracting Agricultural Cadastral Maps: Form and Content Validation. IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing.
76. Sharifi, A., & Safari, M. M. (2025). Enhancing the Spatial Resolution of Sentinel-2 Images Through Super-Resolution Using Transformer-Based Deep

Learning Models. IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing.
77. Dwivedi, A., Vijayan, P., Gupta, R., & Ramdasi, P. (2021). Enhancing Enterprise Business Processes Through AI Based Approach for Entity Extraction–An Overview of an Application. In Recent Trends in Image Processing and Pattern Recognition: Third International Conference, RTIP2R 2020, Aurangabad, India, January 3–4, 2020, Revised Selected Papers, Part I 3 (pp. 373-380). Springer Singapore.
78. Baviskar, D., Ahirrao, S., Potdar, V., & Kotecha, K. (2021). Efficient automated processing of the unstructured documents using artificial intelligence: A systematic literature review and future directions. IEEE Access, 9, 72894-72936.
79. Rhem, A. J. (2023). Ethical use of data in AI applications. In Ethics - Scientific Research, Ethical Issues, Artificial Intelligence and Education [Working Title]. IntechOpen. https://doi.org/10.5772/intechopen.1001597
80. DeVerna, M. R., Yan, H. Y., Yang, K.-C., & Menczer, F. (2023, August 21). Fact-checking information generated by a large language model can decrease news discernment (v1).
81. Meng, F., & Wang, W. (2023). The impact of digitalization on enterprise value creation: An empirical analysis of Chinese manufacturing enterprises. Journal of Innovation & Knowledge, 8, 100385.
82. Gupta, A., Tung, Y. A., & Marsden, J. R. (2004). Digital signature: use and modification to achieve success in next generational e-business processes. Information & Management, 41(5), 561-575.
83. Chang, S. E., Chen, Y. C., & Wu, T. C. (2019). Exploring blockchain technology in international trade: Business process re-engineering for letter of credit. Industrial Management & Data Systems, 119(8), 1712-1733.
84. Pérez-Álvarez, J. M., Gómez-López, M. T., Eshuis, R., Montali, M., & Gasca, R. M. (2020). Verifying the manipulation of data objects according to business process and data models. Knowledge and Information Systems, 62(7), 2653-2683.
85. Balakayeva, G., Ezhilchelvan, P., Makashev, Y., Phillips, C., Darkenbayev, D., & Nurlybayeva, K. (2023). Digitalization of enterprise with ensuring stability and reliability. Informatyka, Automatyka, Pomiary W Gospodarce I Ochronie Środowiska, 13(1), 54–57. https://doi.org/10.35784/iapgos.3295
86. Viriyasitavat, W., Da Xu, L., Bi, Z., & Pungpapong, V. (2019). Blockchain and internet of things for modern business process in digital economy—the state of the art. IEEE transactions on computational social systems, 6(6), 1420-1432.
87. Zacharewicz, G., Diallo, S., Ducq, Y., Agostinho, C., Jardim-Goncalves, R., Bazoun, H., & Doumeingts, G. (2017). Model-based approaches for interoperability of next generation enterprise information systems: state of the art and future challenges. Information Systems and e-Business Management, 15, 229-256.

88. Chauhan, P., Verma, J. P., Jain, S., & Rai, R. (2021). Blockchain based framework for document authentication and management of daily business records. In Blockchain for 5G-Enabled IoT: The new wave for Industrial Automation (pp. 497-517).
89. Balakayeva G., Darkenbayev D., Zhanuzakov M. Development of a software system for predicting employee ratings // Informatyka, Automatyka, Pomiary w Gospodarce i Ochronie Środowiska. – 2023. – Vol. 13, No. 3. – P. 121–124. – DOI: 10.35784/iapgos.3723.
90. Mahmood, H. S. (2024). Conducting In-Depth Analysis of AI, IoT, Web Technology, Cloud Computing, and Enterprise Systems Integration for Enhancing Data Security and Governance to Promote Sustainable Business Practices. Journal of Information Technology and Informatics, 3(2).
91. Azman, N. A., Mohamed, A., & Jamil, A. M. (2021). Artificial intelligence in automated bookkeeping: a value-added function for small and medium enterprises. JOIV: International Journal on Informatics Visualization, 5(3), 224-230.
92. Balakayeva G., Zhanuzakov M., Zhapbasbayev U., Nurlybayeva K. An intelligent enterprise system with processing and verification of business documents using big data and AI // Journal of Intelligent Systems. – 2025. – Vol. 34, No. 1. – Art. 20240446. – DOI: 10.1515/jisys-2024-0446.
93. Yathiraju, N. (2022). Investigating the use of an artificial intelligence model in an ERP cloud-based system. International Journal of Electrical, Electronics and Computers, 7(2), 1-26.
94. Bharadiya, J. P. (2023). A comparative study of business intelligence and artificial intelligence with big data analytics. American Journal of Artificial Intelligence, 7(1), 24.
95. Balakayeva, G., Zhanuzakov, M. & Kalmenova, G. (2023). Development of a digital employee rating evaluation system (DERES) based on machine learning algorithms and 360-degree method. Journal of Intelligent Systems, 32(1), 20230008. https://doi.org/10.1515/jisys-2023-0008
96. Balakayeva G., Nurlybayeva K., Zhanuzakov M. Development of software complex for digitalization of enterprise activities // International Journal of Electrical and Computer Engineering. – 2023. – Vol. 17, No. 12. – P. 315–318.
97. Mahdi S. Almhanna, Tariq A. Murshedi, Firas S. Al-Turaihi et al. Dynamic Weight Assignment with Least Connection Approach for Enhanced Load Balancing in Distributed Systems, 07 August 2023, PREPRINT (Version 1) available at Research Square [https://doi.org/10.21203/rs.3.rs-3216549/v1]
98. Costa, J., Matos, R., Araujo, J., Li, J., Choi, E., Nguyen, T. A., Lee, J.-W., & Min, D. (2023). Software aging effects on Kubernetes in container orchestration systems for digital twin cloud infrastructures of urban air mobility. Drones, 7(1), 35. https://doi.org/10.3390/drones7010035
99. Mohammad, M. (2025). Analysis of design patterns and benchmark practices in Apache Kafka event-streaming systems. arXiv. https://arxiv.org/abs/2512.16146

100. Martín, C., Langendoerfer, P., Soltani Zarrin, P., Díaz, M., & Rubio, B. (2020). Kafka-ML: Connecting the data stream with ML/AI frameworks. arXiv. https://arxiv.org/abs/2006.04105
101. PostgreSQL Global Development Group. (n.d.). PostgreSQL: The world's most advanced open source database. https://www.postgresql.org/
102. Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Duchesnay, É. (2011). Scikit-learn: Machine learning in Python. Journal of Machine Learning Research, 12, 2825–2830
103. Zhanuzakov M., Balakayeva G. Prediction of employee promotion based on ratings using machine-learning algorithms // Bulletin. Series Physical and Mathematical Sciences. – 2022. – DOI: 10.51889/2022-1.1728-7901.14.

APPENDIX A - Program code for calculating optimal checkpointing interval using queuing theory based algorithm

```
import numpy as np

# Define Parameters
   lambda_arrival
   mu
   avg_backup_interval
   breakdown_rate
   I_min = 0
   I_max
   N

def calculate_load(arrival_rate, mu, alpha, tau, beta):
       sigma = tau + beta
       v = mu + alpha + beta
       EV = 1 / v
       EV2 = 2 / (v ** 2)
       EY = (1 / sigma) * (1 + beta * EV)
       EY2 = (2 / (sigma ** 2)) * (1 + beta * EV) + (beta / sigma) * EV2
       EX = (1 / mu) * ((alpha + sigma) / sigma) * ((v + beta) / v)
       EX2 = 2 * (EX ** 2) + (1 / mu) * (1 + alpha * EY2 + beta * EV2)
       if arrival_rate * EX < 1:
          L = arrival_rate * EX + (arrival_rate ** 2 * EX2) / (
             2 * (1 - arrival_rate * EX)
          )
          return L
       else:
          return None

   intervals = np.linspace(I_min, I_max, N)
   minimal_load = float("inf")
   optimal_interval = None
   loads = []
   for I in intervals:
       alpha = 1 / I
       L = calculate_load(lambda_arrival, mu, alpha,
                          avg_backup_interval, breakdown_rate)
       loads.append(L)
       if L is not None and L < minimal_load:
          minimal_load = L
          optimal_interval = I
```

APPENDIX B - Program code for calculating optimal checkpointing interval using model-based algorithm

```
import pandas as pd

def calc_probabilities(arr):
    df = pd.DataFrame(arr)
    df = df.rank(method='max') / len(df)
    return df[0].tolist()

def calc_area(h1, h2, x1, x2):
    delta_x = x2 - x1
    avg_h = (h1 + h2) / 2
    return avg_h * delta_x

def find_area_under_cdf(x_arr, y_arr):
    res_area = 0
    for i in range(len(x_arr)):
        if (i == len(x_arr) - 2):
            break
        res_area += calc_area(y_arr[i], y_arr[i + 1], x_arr[i], x_arr[i + 1])
    return res_area

def compute_Tp(total_area_CDF, area_CDF):
    return total_area_CDF - area_CDF


def compute_checkpoint_interval(T_p, N_t):
    return T_p / N_t


probabilities = calc_probabilities(response_times)
area_CDF = find_area_under_cdf(response_times, probabilities)
total_area_CDF = response_times[len(response_times) - 1]
T_p = compute_Tp(total_area_CDF, area_CDF)
optimal_checkpointing_interval = compute_checkpoint_interval(T_p, 2.7)
print(optimal_checkpointing_interval/1000)
```