

Checkpointing for scalable crash-tolerance in the context of enterprise digitalization systems

Gulnar Tultayevna Balakayeva¹, Paul Ezhilchelvan², Mukhit Zhanuzakov¹

¹Department of Computer Science, Faculty of Information Technology and Artificial Intelligence, Al-Farabi Kazakh National University, Almaty, Kazakhstan

²School of Computing, Newcastle University, Newcastle, United Kingdom

Article Info

Article history:

Received Dec 4, 2025

Revised Aug 5, 2026

Accepted Aug 20, 2026

Keywords:

Digitalization

Enterprise

Failure

Fault tolerance

Scalability

Server

ABSTRACT

This paper presents a scalable fault-tolerant framework for enterprise digitalization systems using optimized checkpointing and rollback-recovery mechanisms. The study introduces a novel mathematical model to determine optimal checkpointing intervals, minimizing performance overhead while ensuring rapid recovery from failures. A dual-redundant storage strategy (local disks and remote databases) enhances resilience against both application crashes and server failures. Experimental evaluations demonstrate the system's effectiveness: in crash scenarios, recovery times (RT) improved by 47% compared to non-checkpointed systems, while adding only 2.5% overhead during normal operations. The framework maintains linear scalability through backup server integration, supporting enterprise growth without compromising reliability. Performance metrics reveal that at an optimal checkpointing interval of 0.5 seconds, the system achieves 92% process continuity during frequent failures, with a breakdown rate of ξ at 0.1. These results provide empirical validation of the design's ability to balance fault tolerance with operational efficiency. The research contributes practical insights for implementing resilient digital infrastructures, addressing critical challenges in enterprise system reliability and continuous service availability. The novelty of this work lies in combining an analytically derived checkpointing model with a practical and scalable system design. This approach provides a clear and efficient balance between fault tolerance and performance, offering a practical solution for improving reliability and continuous service availability in enterprise systems.

This is an open access article under the [CC BY-SA](#) license.



Corresponding Author:

Mukhit Zhanuzakov

Department of Computer Science, Faculty of Information Technology and Artificial Intelligence

Al-Farabi Kazakh National University

Almaty, Kazakhstan

Email: zhanmuha01@gmail.com

1. INTRODUCTION

In the contemporary business environment, the impact of digitalization on enterprises is a transformative force. This essay investigates the dramatic changes brought about by digitization, with a particular emphasis on the ramifications for enterprises. Integrating digital technologies such as artificial intelligence (AI), internet of things (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 research contributes a scalable fault-tolerant framework for enterprise digitalization systems that combines an analytically derived

model for optimal checkpointing intervals with a practical rollback-recovery mechanism. The study introduces a method to minimize performance overhead while maintaining rapid recovery, supported by a dual-redundant storage strategy that improves resilience to both application and server failures.

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. Through case studies and trend analysis, this article aims to provide accurate insights into the strategic dimensions of enterprise digitalization, emphasizing the vital role of fault-tolerance and scalability [2]. Understanding and implementing robust systems in these areas are essential for organizations navigating the digital frontier, ensuring sustainable growth, and maintaining competitiveness in today's dynamic business landscape.

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 (SA) 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.

Over the years, extensive research on checkpointing-based fault-tolerance algorithms has been carried out by many authors. Using scientific platform "ScienceDirect", we searched articles by following key terms: checkpointing, fault-tolerance, rollback-recovery. As a result, we found that over 275 original research papers and 17 review articles were published on this topic. And the average number of publications was around 10 papers per year (for a range from 2001 to 2025).

Existing studies can be broadly categorized into three main directions: checkpointing optimization algorithms, fault tolerance in distributed and cloud environments, and analytical recovery and rollback models.

- a. Checkpointing optimization algorithms. A significant body of research focuses on improving checkpoint placement and reducing associated overhead. For instance, Kumari and Kaur [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.
- b. Fault tolerance in cloud and distributed environments. Another research direction addresses fault tolerance under complex failure conditions, including Byzantine faults. Chinnathambi *et al.* [5] introduce the work designs a scheduling algorithm (WSSS) and the time-based checkpoint and consistency (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.
- c. Analytical models of recovery and rollback mechanisms. A number of studies focus on theoretical modeling of checkpointing and recovery processes. Dimitriou [6] investigates retrieval 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.

Overall, 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.

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

Figure 1 describes an enterprise system, which consists of N, $N > 1$, physical servers; each server has number of 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 hypertext transfer protocol (HTTP) request to any server in system.

In Figure 1, Server₁ receives a request from Client₁, makes queries to database, receives results, processes information, stores result in the database and communicates with Server₂; Server₂ further processes information and updates database, sends result to Server₃; Server₃ 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., Server₃ might send result to Server₂). Figure 1 also shows that some processes may require only one server. For example, Client₂ sends request to Server_N, which processes the data, updates database and sends response to the client.

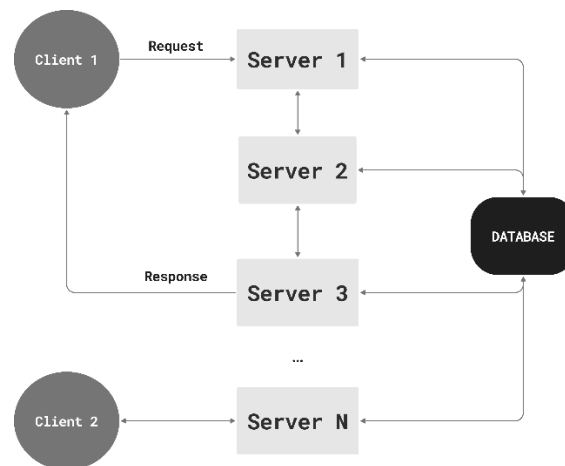


Figure 1. Enterprise system architecture

As a prototype process for implementing and validating purposed ideas we use document processing workflow that is a part of our enterprise information system. The workflow of document processing can be seen in Figure 2. Client request lands at Server₁, where SA₁ executes the request. The results and the request are passed on to Server₂ where two applications SA₂ and SA₃ process it in that order. The output from SA₃ is sent to SA₄ in Server₃ and then from SA₄ to SA₅ in Server₁, which sends out the response to the Client. This particular workflow involves a total of 5 SAs distributed on three servers.

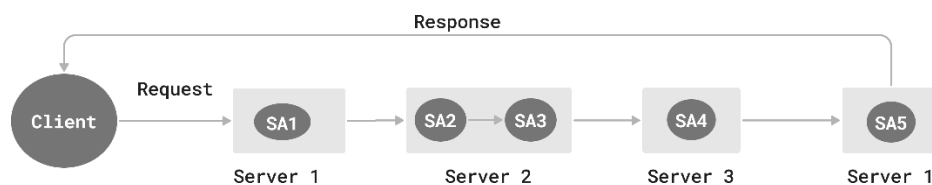


Figure 2. Workflow of document processing

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:

- SA 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 paper we do not focus on database failures. 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, and resource exhaustion [7], [8]. 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 SA={SA₁, SA₂, SA₃, SA₄, SA₅, SA₆}, among these: SA₁ and SA₅ are crashed and unavailable. Note that some SAs send results to each other, particularly SA₂ to SA₃, SA₃ to SA₆, SA₄ to SA₅. Let's look at failed SAs that are painted in red color in Figure 3. As SA₅ failed, SA₄ 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 SA₅ was lost. Same happens with SA₁, 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.

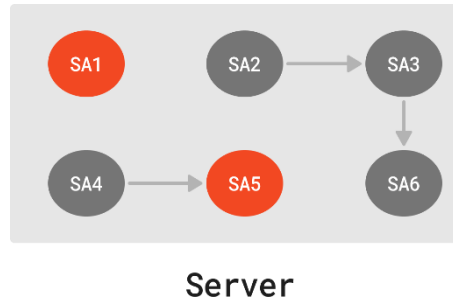


Figure 3. SA(s) failure assumption

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, and power loss. This crash leads to failure of all SA 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 2, we described how our system processes document workflow. Let σ denote 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, $(\sigma=5, n=3)$; $(\sigma=3, n=3)$ and $(\sigma=1, n=1)$ hold for Request₁ and Request₂ respectively. Note that σ can be large and $1 \leq n \leq \text{minimum } \{\sigma, n\}$.

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 $1 - R_{SA}^\sigma$. 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 as described in (1):

$$1 - R_s^n \quad (1)$$

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

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 [9]. To achieve that we utilize checkpointing-recovery algorithm.

The idea of checkpointing is to periodically store the state of a computation called checkpoints in stable storage. Checkpoints are established while SA 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 [10]–[14]. Figure 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 4. Checkpointing during process execution

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 5 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.

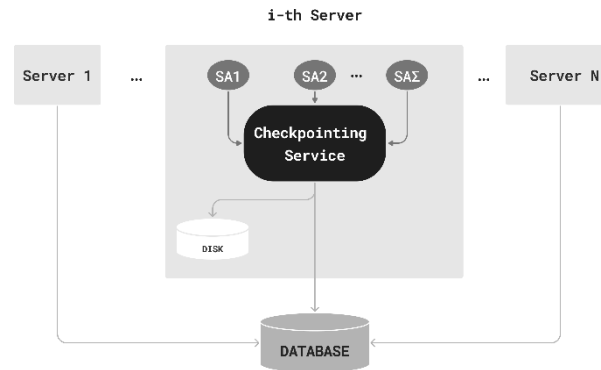


Figure 5. Dual redundant strategy for checkpointing states

To solve the SA crash scenario, we use checkpointing-recovery method described above. When a SA crashes, available SA recovers the crashed one's state using last checkpoint that is stored in the server disk 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 Figure 5.

Figure 6 describes an example of how this solution is applied in a proposed workflow in our system. When Server₂ receives a request from Server₁, it crashes. As soon as it does, its backup server picks up the last checkpoint from remote database and recovers Server₂ with all the SAs (SA₂ and SA₃ are recovered using respective checkpoints). After processing the request, it sends result further to Server₃, 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.

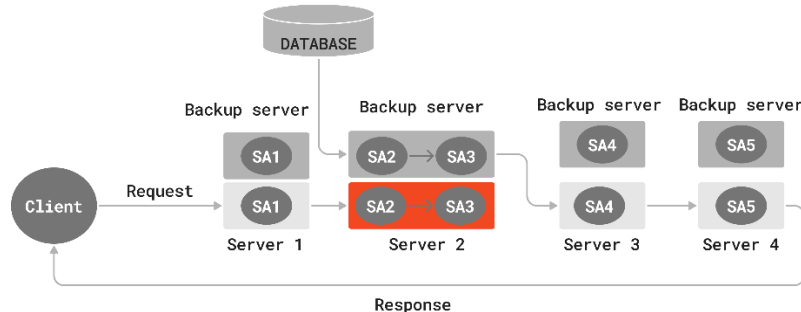


Figure 6. Checkpointing-recovery approach when server crashes

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 SA 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 SA 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 Figure 6.

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.

In this work, the optimal checkpointing interval is calculated using model from recent research [15] done by one of the authors of this work: P. Ezhilchelvan, with modifications and adaptations introduced by the authors to meet the requirements of the proposed approach. Table 1 lists all the necessary parameters needed to calculate optimal checkpointing interval using this model [16]–[19].

Table 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	$1/\mu$
Average back-up interval	$1/\beta$
Breakdown rate	ξ
Average repair (replacement) interval	$1/\eta$
Average service interval in phase k	$1/\nu$
Average interruptable back-up interval	$1/\sigma$

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.

2. METHOD

2.1. Computing parameters for the optimal checkpointing interval

The first step is to calculate the necessary parameters for the checkpointing model Table 1. In Table 2, we make assumptions before calculating other parameters.

Table 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	$1/\mu$	To be calculated
Average back-up interval	$1/\beta$	To be calculated
Breakdown rates	ξ	0.001, 0.01, 0.1
Average repair (replacement) interval	$1/\eta$	0 (we assume the repair is instantaneous)
Average service interval in phase k	$1/\nu$	To be calculated
Average interruptable back-up interval $-1/\sigma$	$1/\sigma$	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 (4)–(13).

Jobs arrive to the system according to a Poisson process with intensity λ . Any back-up operation follows an exponential distribution (4):

$$\sigma = \beta + \xi \quad (4)$$

Any service interval is exponentially distributed:

$$\nu = \mu + \alpha + \xi \quad (5)$$

Calculate expected values (first and second moments of a random variable V related to service interval) using (6) and (7). V is defined as the time interval from the moment of failure (breakdown) to the restart of the service.

$$E(V) = \frac{1}{v} \quad (6)$$

$$E(V^2) = \frac{1}{v^2} \quad (7)$$

Calculate expected time considering backup intervals and breakdowns (8) and (9). Y is the time interval from the start of the attempt to create a checkpoint (checkpoint) to the resumption of the service.

$$E(Y) = \frac{1}{\sigma}(1 + \beta * E[V]) \quad (8)$$

Calculate variance:

$$E(Y^2) = \frac{1}{\sigma^2}(1 + \beta * E[V]) + \frac{\beta}{\sigma}E[V^2] \quad (9)$$

Similarly, calculate expected service time and second moment of the service time (10) and (11). X is a random variable that represents the full time from the beginning of the service of the task to its completion.

$$E(X) = \frac{1}{\mu} * \frac{\alpha + \sigma}{\sigma} * \frac{v + \beta}{v} \quad (10)$$

$$E[X^2] = 2 * E[X]^2 + \frac{1}{\mu}(1 + \alpha * E[Y^2] + \beta * E[V^2]) \quad (11)$$

The system is considered to be stable if the following is true [13]:

$$\lambda * E[X] < 1 \quad (12)$$

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 (13):

$$L = \lambda * E[X] + \frac{\lambda^2 * E[X^2]}{2(1 - \lambda * E[X])} \quad (13)$$

This (13) 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.

2.2. Performance evaluation

To evaluate performance of the method, the function of generating documents was chosen as a test process [20]–[25]. 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 cumulative distribution function (CDF) displayed in Figure 7.

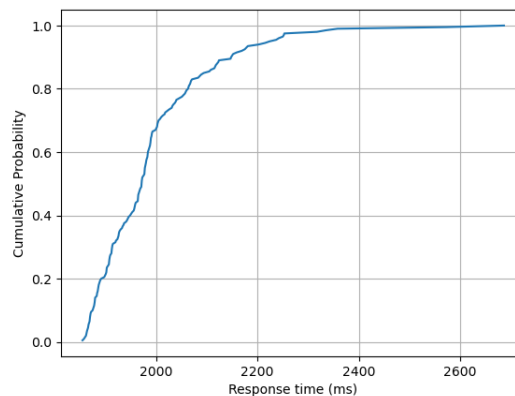


Figure 7. CDF of service times (without checkpointing and crash-free)

The area under the CDF was computed using the Trapezoidal rule (14), resulting in a value of 695.82.

$$Area_{CDF} = \sum_{i=1}^{N-1} \frac{(h_i + h_{i+1})}{2} * (x_{i+1} - x_i) \quad (14)$$

Mean service time (μ) value is calculated using total area and area under the curve (15):

$$\mu_{CDF} = Total\ area_{CDF} - Area\ under_{CDF} \quad (15)$$

The total area represents the highest response time, which amounts to 2686 milliseconds. Using (3), 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 (14). The mean CDF of backup interval is calculated as 0.125 seconds. Next, (4) and (11) were calculated, giving us all the necessary parameters for finding optimal checkpointing interval.

In Figure 8, the average number of jobs present in the system (13) is calculated for different checkpointing values ($1/\alpha$) 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 .

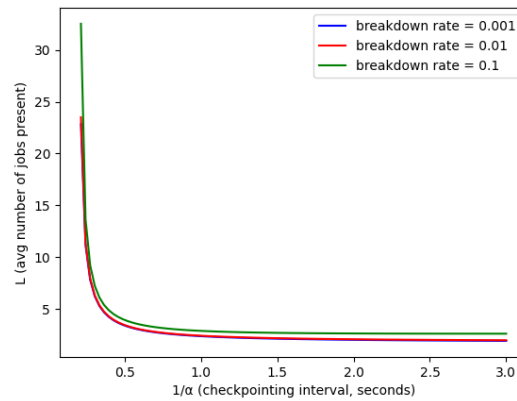


Figure 8. Finding optimal checkpointing interval, $1/\alpha$

2.3. Algorithm description

The description of the proposed algorithm for finding optimal checkpointing interval can be viewed in the Algorithm 1 pseudocode or in Figure 9 in the form of block scheme.

Algorithm 1. Optimal checkpoint interval selection based on minimum system load

```

Generate a set of candidate intervals: Intervals  $\leftarrow$  Linspace(I_min, I_max, N)
Initialize minimal load: Minimal_Load  $\leftarrow$  + $\infty$ 
Initialize optimal interval: Optimal_Interval  $\leftarrow$  null
Initialize empty list of loads: Loads  $\leftarrow$  []
For each I in Intervals do:
    Compute checkpointing rate:  $\alpha \leftarrow 1 / I$ 
    Compute system load:  $L \leftarrow$  Calculate_Load( $\lambda$ ,  $\mu$ ,  $\alpha$ , avg_backup_interval,  $\beta$ )
    Append load to list: Append L to Loads
    If L is defined AND  $L < \text{Minimal\_Load}$  then:
        Update minimal load: Minimal_Load  $\leftarrow$  L
        Update optimal interval: Optimal_Interval  $\leftarrow$  I
    End If
End For
Return Optimal_Interval

```

2.4. System failure experiments

We conducted experiments involving the failure of the primary server to assess the recovery capabilities of an available server in recovering the SAs. The experiment was iterated 200 times, maintaining a consistent arrival rate of 0.3. To save the state of the running process, a script was written and implemented.

The script sets optimal checkpointing interval that periodically writes current state of the process to a file as a Javascript object notation (JSON) object.

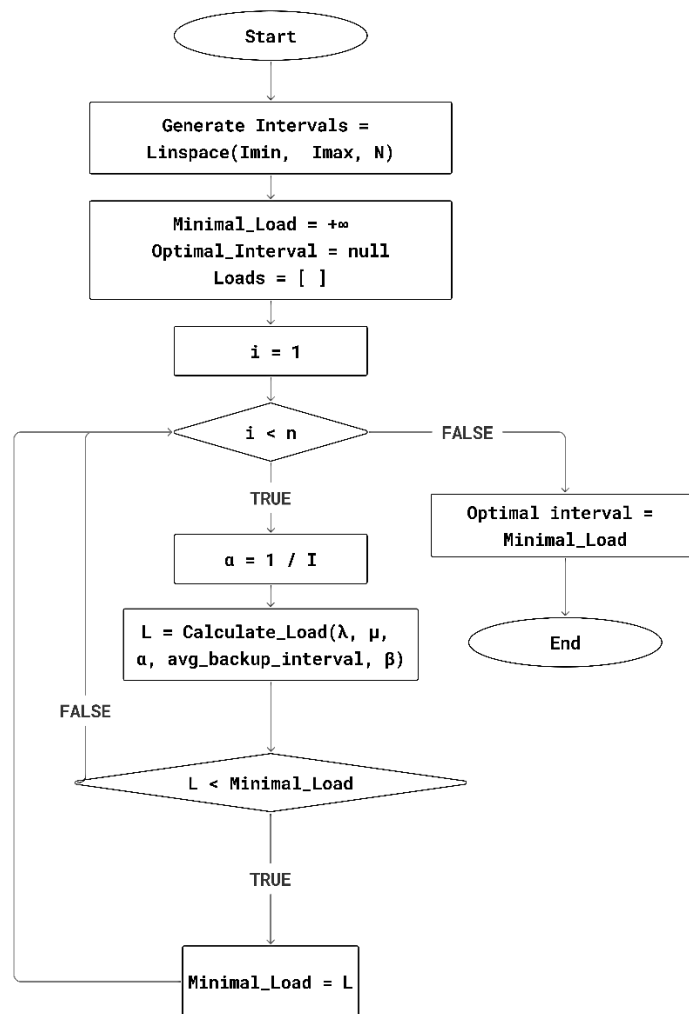


Figure 9. Finding optimal checkpointing interval block scheme

The script will automatically terminate the process at a random moment in the process execution, simulating a failure. Thus, we will be able to compare and evaluate the effectiveness of our implementation. The contents of the checkpoint can be seen in Figure 10. Checkpointing states save the PDF file's base64-encoded result. Upon function restart, it verifies saved states by matching the document's "id" with folders in the "checkpoints" folder. Then it searches for the last checkpoint, if it exists, the process is restored using this data.

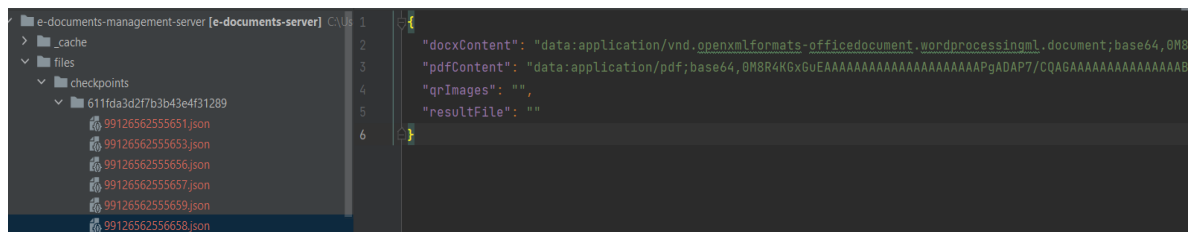


Figure 10. Checkpointing data

2.5. System architecture diagram

Figure 11 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.

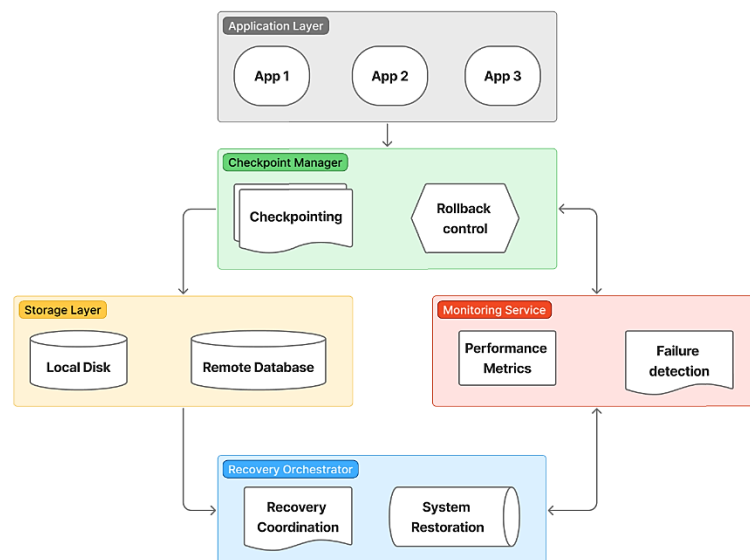


Figure 11. Architecture diagram

2.6. 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.

2.6.1. Nodes

The system consists of a set of physical servers: $N = \{S_1, S_2, \dots, S_n\}$, $n > 1$, where each server represents a computational node capable of processing client requests. Each node hosts multiple 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.

2.6.2. Processes

Processes in the system are modeled as SAs executing tasks within enterprise workflows. Let $P = \{SA_1, SA_2, \dots, SA_\sigma\}$ 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.

2.6.3. Failure model

The system considers two primary types of failures:

- **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.
- **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.

2.6.4. Recovery model

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: i) SA failure, another available SA retrieves the latest checkpoint from the local disk and resumes execution and ii) 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.

2.6.5. Consistency model

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:

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

2.7. Experimental design and evaluation framework

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 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 $\lambda=0.3$ 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)

2.7.1. Evaluation metrics

To evaluate the effectiveness of the proposed checkpointing and recovery mechanisms, a set of performance and reliability metrics is considered. The following metrics are used in the evaluation:

- Recovery time (RT)

RT is defined as the time required to restore a failed process or server to a consistent operational state after failure detection (16):

$$RT = t_{\text{recovery_complete}} - t_{\text{failure}} \quad (16)$$

This metric captures the responsiveness of the recovery mechanism.

- Mean time to recovery (MTTR)

MTTR represents the average RT across all failure events as seen in (17):

$$MTTR = \frac{1}{N_f} \sum_{i=1}^{N_f} RT_i \quad (17)$$

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 (18):

$$D_{\text{throughput}} = \frac{T_{\text{baseline}} - T_{\text{checkpoint}}}{T_{\text{baseline}}} * 100\% \quad (18)$$

where:

T_{baseline} : throughput without checkpointing

$T_{\text{checkpoint}}$: 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 (19):

$$O_{storage} = \frac{S_{checkpoints}}{S_{total}} * 100\% \quad (19)$$

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 (20):

$$L_{checkpoint} = t_{checkpoint_end} - t_{checkpoint_start} \quad (20)$$

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 (21):

$$Availability = \frac{T_{uptime}}{T_{uptime} + T_{downtime}} * 100\% \quad (21)$$

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

2.8. Benchmark workloads

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.

2.8.1. Synthetic enterprise workloads

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 SA σ
- Number of servers n
- 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.

2.8.2. Microservice transaction simulation

To reflect modern enterprise architectures, the system is evaluated under a microservice-based transaction model. In this setup, each SA is treated as an independent microservice that communicates with others via request-response interactions.

A transaction consists of a chain of dependent service calls: $SA_1 \rightarrow SA_2 \rightarrow \dots \rightarrow SA_k$. Failures may occur at any stage, requiring partial or full rollback. This benchmark evaluates:

- Recovery efficiency in interdependent workflows
- Impact of checkpointing on latency propagation
- Ability to maintain consistency across distributed services

This scenario closely resembles real-world cloud-native systems and highlights the effectiveness of the proposed recovery orchestrator.

2.8.3. Distributed service orchestration workloads

This benchmark models complex enterprise workflows involving orchestration across multiple servers. In our case we use the document-processing pipeline described in Figure 2.

The workload evaluates:

- System behavior under multi-node coordination
- Recovery from both SA-level and server-level failures
- Scalability when increasing the number of nodes n and processes σ

This benchmark is particularly important for validating the system under realistic enterprise orchestration scenarios with high interdependence.

3. RESULTS AND DISCUSSION

This section describes the results of the implementation and experiments of proposed methods.

3.1. Performance evaluation results

Table 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. Performance evaluation results

Metric	Baseline system	Proposed system
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)

3.2. Scalability analysis

To evaluate scalability, the system was tested by increasing:

- Number of servers n
- Number of SAs σ
- Arrival rate λ

Results indicate that:

- The system maintains near-linear scalability due to distributed processing and backup server integration
- RT remains stable even as σ 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.

3.3. 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 12 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 13 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.

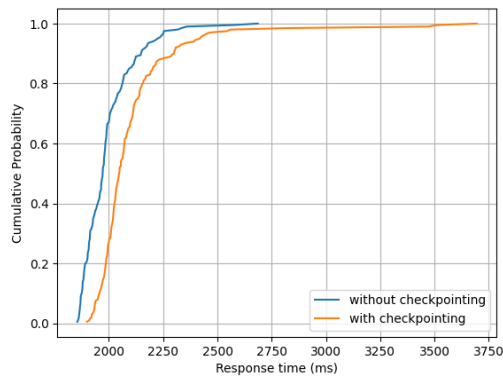


Figure 12. CDF of service rate in a crash-free experiment

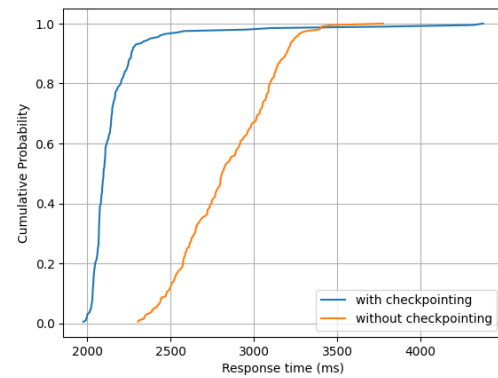


Figure 13. CDF of service rate with during crashes

3.4. Bottleneck analysis

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.

3.5. Overhead vs. recovery trade-off

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

- Minimizes RT
- Keeps throughput degradation below 3%
- Maintains high availability (92%)

4. CONCLUSION

Extensive research was conducted to enhance the fault tolerance and scalability of enterprise applications, leading to the implementation of rollback-recovery method with optimal checkpointing interval. This comprehensive approach ensures the reliability and scalability of enterprise applications, contributing to a resilient and scalable digital ecosystem. Paper addresses two failure types: SA crash and physical server crash. Authors provide dual-redundant strategy and backup servers to tackle the issues.

The system's resiliency is tested using implementations. Recovering a process from the last saved checkpoint demonstrated system resiliency by implementing a process recovery, highlighting the role of the last saved checkpoint in protecting against failures. Implementations with crash and crash-free scenarios in environments both with checkpointing and without checkpointing provide proof of the robustness of our resiliency approach, revealing its potential to ensure uninterrupted operation even in the face of unexpected challenges.

Despite these contributions, the dual-storage mechanism introduces extra overhead and resource consumption, particularly under high checkpointing frequency. The approach also assumes stable network conditions for remote data synchronization, which may not always be guaranteed. Future work will focus on adaptive checkpointing mechanisms that dynamically adjust intervals based on system load and failure likelihood. Further research may also explore the integration of machine learning (ML) techniques for predictive fault management, as well as validation in large-scale cloud environments to improve scalability and real-world applicability.

FUNDING INFORMATION

Authors state no funding involved.

AUTHOR CONTRIBUTIONS STATEMENT

This journal uses the Contributor Roles Taxonomy (CRediT) to recognize individual author contributions, reduce authorship disputes, and facilitate collaboration.

Name of Author	C	M	So	Va	Fo	I	R	D	O	E	Vi	Su	P	Fu
Gulnar Tultayevna Balakayeva	✓	✓		✓	✓	✓	✓	✓	✓	✓		✓	✓	✓
Paul Ezhilchelvan	✓	✓		✓	✓	✓		✓	✓	✓		✓		✓
Mukhit Zhanuzakov		✓	✓			✓		✓	✓	✓	✓		✓	✓

C : Conceptualization

M : Methodology

So : Software

Va : Validation

Fo : Formal analysis

I : Investigation

R : Resources

D : Data Curation

O : Writing - Original Draft

E : Writing - Review & Editing

Vi : Visualization

Su : Supervision

P : Project administration

Fu : Funding acquisition

CONFLICT OF INTEREST STATEMENT

Authors state no conflict of interest.

DATA AVAILABILITY

Derived data supporting the findings of this study are available from the corresponding author MZ on request.

REFERENCES

- [1] A. Ledmi, H. Bendjenna, and S. M. Hemam, "Fault tolerance in distributed systems: A survey," in *2018 3rd International Conference on Pattern Analysis and Intelligent Systems (PAIS)*, Oct. 2018, pp. 1–5, doi: 10.1109/PAIS.2018.8598484.
- [2] R. Garg and P. Kumar, "A review of checkpointing fault tolerance techniques in distributed mobile systems," *International Journal on Computer Science and Engineering*, vol. 2, no. 4, pp. 1052–1063, 2010.
- [3] M. Fayyaz and T. Vladimirova, "Survey and future directions of fault-tolerant distributed computing on board spacecraft," *Advances in Space Research*, vol. 58, no. 11, pp. 2352–2375, Dec. 2016, doi: 10.1016/j.asr.2016.08.017.
- [4] P. Kumari and P. Kaur, "Checkpointing algorithms for fault-tolerant execution of large-scale distributed applications in cloud," *Wireless Personal Communications*, vol. 117, no. 3, pp. 1853–1877, Apr. 2021, doi: 10.1007/s11277-020-07949-0.
- [5] S. Chinnathambi, A. Santhanam, J. Rajarathinam, and M. Senthikumar, "Scheduling and checkpointing optimization algorithm for Byzantine fault tolerance in cloud clusters," *Cluster Computing*, vol. 22, no. S6, pp. 14637–14650, Nov. 2019, doi: 10.1007/s10586-018-2375-9.
- [6] I. Dimitriou, "A retrieval queue for modeling fault-tolerant systems with checkpointing and rollback recovery," *Computers & Industrial Engineering*, vol. 79, pp. 156–167, Jan. 2015, doi: 10.1016/j.cie.2014.10.018.
- [7] Q. Wang, Y. Liu, T. Xu, X. Sun, and J. Li, "Design of software fault tolerant system for autonomous underwater vehicles," in *2021 IEEE 4th Advanced Information Management, Communication, Electronic and Automation Control Conference (IMCEC)*, Jun. 2021, pp. 1456–1460, doi: 10.1109/IMCEC51613.2021.9482037.
- [8] Y. Liu *et al.*, "Rethinking software misconfigurations in the real world: An empirical study and literature analysis," *arXiv*, Jul. 2026, doi: 10.1007/s10664-026-10892-6.
- [9] S. Isukapalli and S. N. Srirama, "A systematic survey on fault-tolerant solutions for distributed data analytics: Taxonomy, comparison, and future directions," *Computer Science Review*, vol. 53, Aug. 2024, doi: 10.1016/j.cosrev.2024.100660.
- [10] A. A. Amin and K. M. Hasan, "A review of fault tolerant control systems: Advancements and applications," *Measurement*, vol. 143, pp. 58–68, Sep. 2019, doi: 10.1016/j.measurement.2019.04.083.
- [11] V. Mohammadian, N. J. Navimipour, M. Hosseinzadeh, and A. Darwesh, "Fault-tolerant load balancing in cloud computing: A systematic literature review," *IEEE Access*, vol. 10, pp. 12714–12731, 2022, doi: 10.1109/ACCESS.2021.3139730.
- [12] M. Saadoon, S. H. Ab. Hamid, H. Sofian, H. H. M. Altarturi, Z. H. Azizul, and N. Nasuha, "Fault tolerance in big data storage and processing systems: A review on challenges and solutions," *Ain Shams Engineering Journal*, vol. 13, no. 2, Mar. 2022, doi: 10.1016/j.asej.2021.06.024.
- [13] W. Zhong *et al.*, "Byzantine fault-tolerant consensus algorithms: A survey," *Electronics*, vol. 12, no. 18, Sep. 2023, doi: 10.3390/electronics12183801.
- [14] A. Rullo, E. Serra, and J. Lobo, "Redundancy as a measure of fault-tolerance for the Internet of Things: A review," in *Policy-Based Autonomic Data Governance*, 2019, pp. 202–226, doi: 10.1007/978-3-030-17277-0_11.
- [15] S. Zhou, K.-J. Lin, J. Na, C.-C. Chuang, and C.-S. Shih, "Supporting service adaptation in fault tolerant Internet of Things," in *2015 IEEE 8th International Conference on Service-Oriented Computing and Applications (SOCA)*, Oct. 2015, pp. 65–72, doi: 10.1109/SOCA.2015.38.
- [16] M. A. Solouki, S. Angizi, and M. Violante, "Dependability in embedded systems: A survey of fault tolerance methods and software-based mitigation techniques," *IEEE Access*, vol. 12, pp. 180939–180967, 2024, doi: 10.1109/ACCESS.2024.3509633.
- [17] P. Ezhilchelvan and I. Mitrani, "Checkpointing models for tasks of different types," *ACM Transactions on Modeling and Performance Evaluation of Computing Systems*, vol. 9, no. 3, pp. 1–23, Sep. 2024, doi: 10.1145/3658667.
- [18] J. F. Shortle, J. M. Thompson, D. Gross, and C. M. Harris, *Fundamentals of queueing theory*. Wiley, 2018, doi: 10.1002/9781119453765.
- [19] 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 Generation Computer Systems*, vol. 161, pp. 315–328, Dec. 2024, doi: 10.1016/j.future.2024.07.022.
- [20] B. M. Oki and B. H. Liskov, "Viewstamped replication: a general primary copy," in *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing - PODC '88*, 1988, pp. 8–17, doi: 10.1145/62546.62549.
- [21] A. Razaque *et al.*, "An anti-corruption system for academic achievement verification in kazakhstani higher education using

Checkpointing for scalable crash-tolerance in the context of enterprise ... (Gulnar Balakayeva)

- blockchain and artificial intelligence,” *Frontiers in Blockchain*, vol. 8, Dec. 2025, doi: 10.3389/fbloc.2025.1683522.
- [22] A. Verbitski *et al.*, “Amazon Aurora: Design considerations for high throughput cloud-native relational databases,” in *Proceedings of the 2017 ACM International Conference on Management of Data*, May 2017, pp. 1041–1052, doi: 10.1145/3035918.3056101.
- [23] D. T. Dang and D. Hwang, “Consensus-based methods for distributed systems, blockchain, and voting: a survey,” *Journal of Information and Telecommunication*, vol. 9, no. 2, pp. 237–260, Apr. 2025, doi: 10.1080/24751839.2024.2416728.
- [24] K. Sampath, K. Devi, T. V. Ambuli, and S. Venkatesan, “AI-powered employee performance evaluation systems in HR management,” in *2024 7th International Conference on Circuit Power and Computing Technologies (ICCPCT)*, Aug. 2024, pp. 703–708, doi: 10.1109/ICCPCT61902.2024.10673159.
- [25] A. Altybay, D. Darkenbayev, and N. Mekebayev, “Parallel numerical simulation of the 2D acoustic wave equation,” *International Journal of Electrical and Computer Engineering (IJECE)*, vol. 14, no. 6, pp. 6519–6525, Dec. 2024, doi: 10.11591/ijece.v14i6.pp6519-6525.

BIOGRAPHIES OF AUTHORS



Gulnar Tultayevna Balakayeva    was born in Almaty, Kazakhstan. She received her specialist degree in applied physics from the Leningrad Polytechnic Institute named after M.I. Kalinin (now Peter the Great St. Petersburg Polytechnic University), St. Petersburg, Russia, in 1972. Her major field of study was applied physics and mechanics. She began her professional career as a research associate at the Almaty Institute of Power Engineering and Communications. She then joined Kazakh State University (KazGU), Almaty, Kazakhstan, where she held positions as a research associate, senior lecturer, and associate professor. In 1986, she received the candidate of sciences degree, and in 1999, the doctor of sciences degree. She can be contacted at email: gulnardsa@gmail.com.



Paul Ezhilchelvan    received Ph.D. degree in computer science in 1989 from Newcastle University. He received the Bachelor of Engineering degree in 1981 from the University of Madras, India, and the Master of Engineering degree in 1983 from the Indian Institute of Science, Bangalore. He joined the School of Computing Science of the University of Newcastle upon Tyne in 1983 where he is currently a reader. His research interests are in distributed algorithms, consensus and multicast protocols, concurrency control protocols in databases, crash-tolerant stream processing, performance evaluation, and stochastic modelling. He can be contacted at email: paul.ezhilchelvan@newcastle.ac.uk.



Mukhit Zhanuzakov    Ph.D. student at al-Farabi Kazakh National University. He received Master degree in Technical Sciences in 2022 at al-Farabi Kazakh National University, Almaty, Kazakhstan. His research interests are software development, algorithms and data structures, ML, and software testing. He can be contacted at email: zhanmuha01@gmail.com.