UDC 004.75
DOI: https://doi.org/10.17721/1812-5409.2026/1.30

Anatolii PASHKO[1], DSc (Phys. & Math.), Prof.
ORCID ID: 0000-0001-6944-8477
e-mail: aap2011@ukr.net

Ihor PAPROTSKYI[1], PhD Student
ORCID ID: 0009-0006-1644-2833
e-mail: igorpapr@gmail.com

Andrii HLYBOVETS[2], DSc (Engin.), Prof.
ORCID ID: 0000-0003-4282-481X
e-mail: a.glybovets@ukma.edu.ua

Svitlana TERENCHUK[3], PhD (Phys. & Math.), Prof.
ORCID ID: 0000-0001-6527-4123
e-mail: terenchuksa@ukr.net

[1]Taras Shevchenko National University of Kyiv, Kyiv, Ukraine
[2]National University of Kyiv-Mohyla Academy, Kyiv, Ukraine
[3]Kyiv National University of Construction and Architecture, Kyiv, Ukraine

IMPROVEMENTS TO THE CIRCUIT BREAKER PREDICTION MODEL FOR INCREASING THE FAULT TOLERANCE OF MICROSERVICES

In high-load distributed systems, it is crucial to pay strict attention to the reliability of communication between architectural components, making fault tolerance a fundamental property of microservice architecture. While various fault tolerance patterns exist to improve microservice resilience, and their implementations are widely adopted, none are universal for the range of possible failures in service communication. Therefore, there is a continuous need to search for more effective approaches to achieving fault tolerance in distributed software systems.

This research is building upon our earlier study, which introduced the Circuit Breaker Prediction Model. The main principle of the Prediction model is to calculate the rating of communication between two services, defined as a convex combination of various performance metrics, and the rating value is then used to determine whether the interaction with the target service can be safely re-established after the period of blocking outgoing requests. Compared to standard implementations of this method, it was specifically designed to reduce the latency involved in blocking request transmission between microservices, especially under uncertain load conditions.

The purpose of this paper is to enhance this model and verify its effectiveness when combined with other resilience patterns, particularly Retry and Timeout. The study enhances the flexibility of the Circuit Breaker Prediction Model by describing an adaptable sliding window mechanism that dynamically adjusts the context size based on load conditions at runtime. Also, the algorithm of rating calculation is generalized to enable the usage of any substantial number of performance metrics. Finally, this research conducts simulated experiments with multiple service communication configurations and fault tolerance pattern combinations, using the standard Circuit Breaker implementation as a baseline for comparison. The findings show improved communication performance, mainly due to reduced request transmission delays under the same load conditions. The prospects of applying this model's principles to other fault tolerance patterns have been discussed, with particular emphasis on the potential of using machine-assisted optimization techniques in dynamic decision making.

Keywords: ***microservices architecture, distributed systems, fault tolerance, resilience patterns, high load, circuit breaker, service communication, runtime adaptation.***

AMS 2020 classification: 68M15, 68M14.

Introduction

In software engineering, microservice architecture (MA) is considered a practical approach for managing large-scale enterprise applications. Breaking down the application structure into many functional parts not only allows for the division of various entity domains and the isolation of distinct parts of the business logic but also increases the autonomy of teams in research and development (R&D): this way, they can work in separate code repositories in parallel, making the development process more granular, faster, and isolated.

Recent surveys highlight the popularity of microservices in software development. According to the JetBrains 2023 Developer Ecosystem survey (JetBrains, 2024), 82 % of respondents use microservices in their daily work. Similarly, Stack Overflow's 2023 Developer Survey (Stack Overflow, 2024) found that nearly half of the participants said their companies rely on microservices. These statistics prove that more and more software development companies, teams, and developers are considering this approach suitable for their system design. This makes microservice architecture one of the leading system design trends in the 2020s.

Despite its popularity, the approach can sometimes be challenging in designing, developing, maintaining, securing, ensuring availability, and implementing correct application logic. Because the key feature of microservice architecture is decentralization, components rely heavily on communication, especially through events that may occur at different levels of the entire system. According to (Lewis, & Fowler, 2014) "the microservice preference towards choreography and event collaboration leads to emergent behavior," which may sometimes be negative and result in unforeseen consequences. If not handled properly, even a brief network failure can cause data inconsistencies between the services, potentially leading to

© Pashko Anatolii, Paprotskyi Ihor, Hlybovets Andrii, Terenchuk Svitlana, 2026

incorrect delivery of the intended functionality. To mitigate this risk, the developers must consider ensuring eventual message delivery between services. This is where the concept of "resilience" emerges.

Fault tolerance patterns, also known as resilience patterns, are commonly used to mitigate disruptions, network outages, security breaches, changes in application load, and system failures. They aim to recover from failures and operate effectively, minimizing the impact of incorrect system behavior on the application's availability.

While numerous fault tolerance patterns and techniques exist, and they can be classified into different categories, this research focuses on those that influence direct synchronous communication between services, one of the basic types of data transfer. Among such patterns, experts outline the following: Timeouts, Circuit Breaker (CB), Retries, Fallbacks, Bulkheads, Rate Limiters, and others (Nygard, 2018, pp. 110–146; Resilience4j, 2025). Even though they seem to be simple in their working principles, many software libraries and frameworks may provide different implementations of some of them. Mostly, those algorithms use the canonical logic of a pattern, and most software developers may use the existing solutions out of the box, changing only variable values in the configuration provided by a library. However, sometimes even a small optimization of the algorithm can lead to a meaningful performance increase.

In this article, we focus on the Circuit Breaker method and extend our previous research published in (Hlybovets, & Paprotskyi, 2024), where we proposed the Circuit Breaker Prediction Model as a modification to it. The model aimed at dynamic adaptation to conditions in synchronous communication between two microservices by predicting the performance of the interaction between two microservices and using a set of runtime metrics, reducing the Circuit Breaker's state-transition latency and improving the request success rate.

However, this model can still be enhanced to increase the method's ability to adapt to load changes. Also, the previous approach provided only a preliminary evaluation of the algorithm. Despite the promising results compared to the traditional Circuit Breaker, we want to further validate whether the implementation is more effective in more complex microservice setups. Apart from that, since the method used a particular set of performance metrics for the communication rating calculation, there is a need to generalize the algorithm so that it could work with any number or combination of metrics.

According to the background, **the object of this research** is microservices that operate under high-load conditions.

The goal of the study is to improve and validate the Circuit Breaker Prediction Model in a more practical environment and in conjunction with other resilience patterns, as well as to generalize the algorithm to support varying numbers and types of performance metrics.

In this work, we consider a "more practical environment" to involve different combinations of synchronous microservice-to-microservice communications, the simultaneous use of multiple resilience patterns, and larger workloads that change over time.

To achieve the goals of the research, **the following objectives were defined for this paper**:

● Enhance the model's ability to adapt to workload changes by introducing automatic adjustment of the sliding window size, which provides a more accurate data context for calculating service communication performance.

● Extend and unify the performance calculation formula introduced in our previous study, and present a generalized version that supports an arbitrary set of performance metrics.

● Validate the model under more practical conditions, including a series of experiments that involve its simultaneous use with other resilience patterns.

It makes sense to briefly mention the main concepts of the traditional Circuit Breaker pattern: it works analogously to a real-world electric fuse. In software programming, this concept is typically implemented as a state machine that wraps the logic of a given function, which makes a call to another service. Depending on the outcome of that function being executed repeatedly, the CB state machine can transition between three states: Closed, Open, and Half-Open (Fowler, 2014).

The flow where one microservice requests another may sound simple and appear easy to manage. However, since microservice architecture often distributes different domains across multiple services, the execution of application logic likely requires multiple cascading requests between them. Given a situation where there is one or more services between the two (as an example, service A makes a request to service B that makes a request to service C…), there is a possibility for some of them to have a failure in communication, which, if not handled properly, may result in cascading failures of the whole call chain. Not only will this break the application logic, but, more importantly, it will also overuse the available resources of all the services in the mentioned call chain.

The Circuit Breaker pattern is used to counteract the situations described above. It rejects the execution of requests that will almost certainly not result in a successful response, thus saving resources and reducing the network load (Fowler, 2014).

In the current study, other resilience patterns, which are often used in conjunction with the Circuit Breaker pattern, will be considered as the methodological basis for experimentation. The patterns of Retry and Timeout. Their role in the research is not isolated but rather complementary: they provide the context in which the proposed Circuit Breaker Prediction Model will be validated, allowing us to test its behavior under combined resilience strategies.

The mentioned mechanisms were described in detail in (Nygard, 2018, pp. 110–146; Resilience4j, 2025).

Since resilience in microservice architecture is a highly multifaceted topic, it has also become the subject of numerous research studies in the field of software engineering.

The study (De Souza Miranda et al., 2024) explores models, methods, and technologies for building resilient microservices. They rely on various strategies to maintain system stability and reliability. The authors examine and present the initial catalog of patterns for developing fault-tolerant microservices. They also provided a review of a wide range of literature and organized a survey in this field of study to determine the difficulties and solutions for them.

The paper (Valdivia et al., 2020) presents a multivocal systematic literature review for microservices, identifying the benefits and trade-offs of different patterns used in microservice architecture development and their association with quality attributes and metrics. The paper identified the metrics that help to measure a system's readiness to meet quality criteria that improve overall performance.

Recent research (Sun, Cui, & Cai, 2024) applied a deep reinforcement learning approach to fault tolerance in microservice environments. They use metrics gathered from a service mesh and cluster in their circuit breaker strategy. The authors aim

to dynamically adjust tripping thresholds and recovery times. This method learns policies from external telemetry data and focuses on environmental observations outside the service. In contrast, our work uses microservices' internal context data for calculating state predictions.

The research (Zhang et al., 2025) proposed a fault-tolerance framework using Spring Cloud and Docker. It utilizes techniques and features like service discovery, load balancing, and failover. The authors focused on fault tolerance at the architectural and orchestration levels, and focused on redundancy and quick service recovery. However, in our study, we focus on the individual pattern's level of optimization in the communication between individual pairs of services.

To sum up, the variety of different approaches in the recent studies shows that microservice resilience can be approached from various angles. It can be observed and improved at the architecture level by analyzing the whole system performance, or it can be optimized by changing the algorithms responsible for controlling service communication. Each perspective has different pros and cons depending on the size and complexity of the target informational system.

There are various implementations of the CB in different programming languages, various libraries, and frameworks. Many of them were mentioned in our previous study (Hlybovets, & Paprotskyi, 2024). The Resilience4j framework was selected for implementing the Prediction Model, and the choice was explained in the mentioned article.

Most frameworks provide the standard Circuit Breaker pattern implementation, but some of them add extra features and configuration variations to improve their customization capabilities. For example, (Resilience4j, 2025) offers highly customizable configuration, the possibility to combine multiple fault tolerance patterns, and advanced metrics (such as "failure rate" and "slow call rate"). The latter framework is used in this research as a base for experiments.

1. Methodology

Circuit Breaker Prediction Model. The main concepts of our model, introduced in (Hlybovets, & Paprotskyi, 2024), are briefly described below.

When using the Circuit Breaker pattern to resolve communication errors between microservices, it is crucial to understand how it can be configured most effectively. The most complex challenge in such a configuration is determining the optimal amount of time for the Circuit Breaker to wait in the Open state until the requests can be permitted.

The period between when a target service started failing to respond and when it became available is not static and may vary across different failure occurrences. As a result, there is almost always a point when the target service has fully recovered, but the Circuit Breaker in the consuming service remains in the Open/Half-open state, blocking requests from being sent. Moreover, the transitions from the Half-open state back to the Open state sometimes make the total delay even longer: after each such transition to the Open state, a cycle of waiting starts again, potentially resulting in the loss of those invaluable seconds of reestablished communication between services, which may cost a lot of business value in an enterprise application.

Since the problem of operational delays is related to the static time-guessing nature of pattern configuration, the Circuit Breaker Prediction Model was previously proposed. The idea of making the pattern to adapt to the communication performance between the two services and predict whether the target service has already recovered forms the expectation of working more efficiently than the original CB model.

Instead of using the canonical three states in the CB, the proposed model has only two: Open and Closed. Upon the target function being called, which is wrapped by the Circuit Breaker, a decision is made on whether to close the "circuit" by calculating the rating value of the communication. Hereinafter, the "rating value" represents the prediction of whether the target microservice is ready to receive requests. If it is higher than the Rating Threshold value configured in the application, the CB mechanism makes a transition to the Closed state, enabling it to make requests to the target service. There were no changes to the Closed state, its conditions for detecting failures, and transitioning to the Open state compared to the canonical CB. (Hlybovets, & Paprotskyi, 2024)

The Circuit Breaker Prediction Model implementation must gather a set of metrics to ensure its operation via the rating value calculation. The sliding window method of data collection is used, thereby taking into account only the last configured number of function invocations.

The particular metrics can vary depending on the future research of that algorithm, but the initial set of them was used in the previous study:

- *Failure rate*. The percentage of failed requests compared to the total number of requests observed in the sliding window. A request is considered failed if its invocation raises an exception from a preconfigured list.
- *Slow call rate*. The ratio of the number of requests that took more time than a configured threshold to the total number of requests recorded in the sliding window.
- *Success call rate*. The ratio of successful (not failed) requests to the total number of requests recorded in the sliding window.
- *Time in the Open state*. The current amount of time CB has been operating in the Open state, starting from the previous transition. It is also essential to check if the time in the Open state has not exceeded the preconfigured threshold, the maximum allowed time in that state, and close the CB if it does.

After some revision, we decided to use an enhanced list of metrics for our experiments in this research. The mentioned list is discussed in Section 1.4.

Generalizing the algorithm. Despite the model having been proven to work, there is a necessity to generalize the algorithm so that it can work with an arbitrary set of performance metrics.

Depending on its nature, we can logically categorize a specific metric as "positive" or "negative," based on how it influences the communication rating value. For example, the Failure rate and Slow call rate are negative metrics because the higher they are, the lower the CB rating value should be.

Therefore, formula (1) is used to represent the calculation logic for the Circuit Breaker Prediction Model:

$$rating = posM + negM, \quad (1)$$

where $posM$ and $negM$ are defined using (2) and (3):

$$posM = \sum_{i=1}^{a}(c_i * m_{pos^i}); \quad (2)$$

$$negM = \sum_{j=1}^{b}(c_j * (1 - m_{neg^j})), \quad (3)$$

where variable a holds the quantity of positive metrics in (2), variable b holds the quantity of negative metrics in (3), and both refer to the number of metrics gathered by the service in the runtime. The c_i and c_j are the preconfigured weight coefficients for each positive or negative metric, which define the level of influence of the corresponding metric on the whole rating value. The m_{pos^i} and m_{neg^j} are the values of each metric (also positive or negative). The weights of m_{neg^j} are inverted, are inverted, ensuring that all components contribute uniformly to communication quality.

To maintain the correctness of calculations, the following static configuration check, which verifies if the equality (4) is satisfied, must be performed when the CB is loaded into the context of an application:

$$\sum_{i=1}^{a} c_i + \sum_{j=1}^{b} c_j = 1. \quad (4)$$

The check ensures that all coefficient values for the metrics are properly distributed relative to one another, and therefore, the proportion of each metric in the overall rating calculation formula is accurately estimated.

Each metric in the mentioned equations is expressed as a ratio; the values are bound to the range from 0 to 1, so the final rating value is also in the range from 0 to 1:

$$c_i,\ m_{pos^i},\ c_j,\ m_{neg^j}, rating \in [0; 1]. \quad (5)$$

Therefore, the whole rating calculation is derived from a normalized weighted aggregation of metrics, with negatively oriented ones inverted to ensure that higher values uniformly indicate better performance.

Finally, the rating value is compared with the preconfigured rating threshold. If the calculated value is bigger, the CB changes its state to Closed, resuming the communication.

It is also important to briefly describe the core part of the model's behaviour.

The algorithm was formalized in the following pseudocode listing:

```
When in CLOSED state:
  * allow request *
  result ← * execute call and gather metrics *
  if (result is error AND error not in IGNORED_ERRORS_LIST):
    if (request duration OR failure counter exceeds thresholds):
      transitionToOpenState()
      * disallow further requests *
      * update metrics *
  else if (success):
    if (request duration OR failure counter exceeds thresholds):
      transitionToOpenState()
      * disallow further requests *
      * update metrics *
  return

When in OPEN state:
  On invocation attempt:
    communicationRating ← calculateRatingValue(metrics)
    if (communicationRating > RATING_TRANSITION_THRESHOLD):
      transitionToClosedState()
      * allow request *
      * update metrics *
    else:
      * decline request and update metrics *
  return

function calculateRatingValue(metrics):
  for each metric in metrics:
    * compute individual metric's component value *
    if metric is negatively oriented:
      * invert its contribution *
  return sum of all rating components
```

According to what is described in this section, it is obvious that as long as both metric values and a sum of coefficients are normalized, any substantial number of performance metrics can be used in this calculation. Each metric contributes to the rating proportionally to its preconfigured weight, and negatively oriented metrics are inverted to ensure consistent interpretation. Also, each metric's component value can be calculated differently if it makes sense for the algorithm, and if any extra logic is required for the metric component to participate in the communication rating.

The Slow Call Rate metric value can be calculated by inverting the rate value and multiplying by the coefficient (Java code):

```
double slowCallRateMetric = countOfSlowCalls / totalCallsInAWindow;
```

```
double slowCallRating = (1 - slowCallRateMetric) * SLOW_CALL_RATE_COEFFICIENT;
```

However, such metrics as Consecutive Failure Streak can be calculated in the following way:

```
int maxStreak = Math.max(maxStreak, consecutiveFailures);
double failureStreakMetric = Math.min((float) maxStreak / S_MAX, 1.0);
double consecutiveFailureStreakComponent = (1 - failureStreakMetric) * FAILURE_STREAK_COEFFICIENT;
```

In this case, the consecutive failure streak metric is additionally normalized to [0, 1] using a saturation limit S_MAX, because there's a need to ensure that excessively long failure sequences do not disproportionately affect the final communication rating.

From an implementation perspective, such metric calculations can be provided by a fault-tolerance framework at the library level, allowing a software developer to potentially select from a predefined list of metric components available for the Circuit Breaker Prediction Model without the necessity to write their own calculation for it. The important detail is that all the reads and writes to the state of CB and its set of metrics must be implemented thread-safe, ensuring consistency and correctness.

Enhancing the adaptiveness to periodic load changes. It is common for a microservice to experience varying load levels over time, depending on the context within which the software system operates. For example, a Kafka worker instance subscribed to multiple user-related topics in the backend of a public business-related website might experience a higher average load during the working part of the day than at night. In this case, for each event processed by that worker instance, if a synchronous request function (wrapped in the Circuit Breaker instance) is invoked to another microservice through the processor's logic, the potential performance metrics increase in proportion to the number of such request invocations. This should provide a better context for the Circuit Breaker Prediction Model, if it is used in this setup.

Most modern Circuit Breaker implementations work based on a sliding window, which is a continuously updating time or call-based interval that collects recent request outcomes and performance metrics to evaluate the current reliability of service communication. Existing frameworks and libraries only provide the static window size, depending on the sliding window type, and only suggest using a fixed sampling duration; some of them also provide the mechanism of evicting the "bad instances" (e.g., Istio/Envoy – "Outlier Detection" (Istio, 2025)), but also without automatic window size adjustments.

In this study, we propose an enhancement to the Circuit Breaker Prediction Model for dynamically adjusting the window size based on recent load. Since our model analyzes performance metrics and is also based on a count-based sliding window, it makes sense to adjust the context to natural traffic oscillations: the lower the traffic, the less recent requests should influence the prediction about the state of communication between microservices, and vice versa.

To implement this idea, an adaptive sliding window controller was added to the existing model that scales the data window size according to the observed throughput.

First, the current request rate is calculated (6):

$$\hat{\lambda}_t = \frac{n_t}{\Delta}, \tag{6}$$

where n_t is the number of requests for the last interval Δ, and $\hat{\lambda}_t$ is the current request rate.

Then, we apply exponential smoothing to the observed request rate to stabilize short-term noise and produce a smoothed throughput (7):

$$\lambda_t = (1-\gamma)\lambda_{t-1} + \gamma\hat{\lambda}_t, \quad \gamma \in (0,1], \tag{7}$$

where λ_t is the smoothed request rate at the moment t, $\lambda_{t-1}$is the previous smoothed request rate, $\hat{\lambda}_t$ is the current "raw" request rate from (6), and γ is the smoothing factor.

The smoothed rate is then used to compute the target window size at the moment t (8):

$$W_{target,t} = clip(\alpha\lambda_t, W_{min}, W_{max}), \tag{8}$$

where $W_{target,t}$ is the target sliding window size at the moment t, λ_t is the smoothed request rate at the moment t, α is the scaling coefficient that controls the proportional growth of the window size relative to the request rate, W_{min}, W_{max} are the lower and upper bounds for the allowable window size, preventing excessive shrinking or expansion, $clip$ is the bounding function that limits the resulting value within the specified range $[W_{min}, W_{max}]$.

To avoid oscillations due to minor traffic variations, the window size is updated only when its relative change exceeds the hysteresis threshold (9):

$$W_t = W_{target,t} \ \ if \ \frac{|W_{target,t} - W_{t-1}|}{W_{t-1}} > \varepsilon; \quad otherwise \ W_t = W_{t-1}, \tag{9}$$

where W_t is the final window size applied at the current update step, $W_{target,t}$ is the newly computed target window size before stabilization, W_{t-1} is the sliding window size from the previous step (current value), and ε is the hysteresis threshold that defines the minimum relative change required to trigger an update.

The recalculation process is performed periodically (e.g., every couple of seconds, configured according to the application's needs) and runs as a separate thread from the main Circuit Breaker flows, or is initiated by a background scheduler via a dedicated timer. The window size and request count variables should be accessed and updated in a thread-safe way. The initial window size value W is meant to be configured at application startup.

As a result, the proposed algorithm conceptually operates as illustrated in Fig. 1:

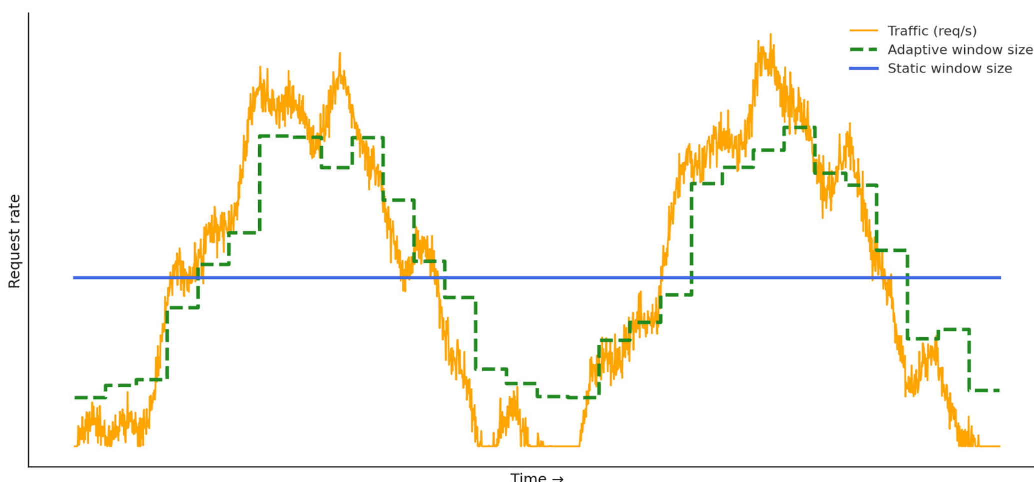

Fig. 1. Conceptual illustration of adaptive sliding window behavior compared to a static configuration under periodic load changes

The adaptive sliding window adjusts its size according to the current request rate, and uses exponential smoothing and a hysteresis threshold to avoid frequent value changes. In turn, the static window size remains insensitive to the load changes and may capture either too little or too much data, so the adaptive window size should provide a more balanced representation of system activity metrics.

We integrated this mechanism into the Circuit Breaker Prediction Model by default and used it in all scenarios in the experimental study.

Experiments. Unlike the earlier study that used a simplified service communication simulation, this work assesses the Circuit Breaker Prediction Model under more realistic and diverse experimental conditions.

In modern microservice systems, developers frequently apply multiple fault tolerance mechanisms together within the same communication flow. They typically combine Circuit Breaker with Retry, Timeout, and other resilience patterns. The experiments in this study were designed according to this idea, and the goal was to validate the CB Prediction Model under these combined conditions in different setups and in a controlled environment to allow measuring and observability.

The configuration of each resilience pattern was selected through extensive empirical analysis. Parameter values were chosen after a series of preliminary runs aimed at identifying stable and distinguishable behaviors between the compared CB implementations of traditional and the Prediction models. Although no formal optimization techniques were used during the preliminary tuning process, the configuration proved suitable for a comparison between the canonical and predictive versions of the Circuit Breaker.

To ensure repeatability, we used identical configuration values for all experiments, making the observed performance differences attributable to the model design itself rather than to random variation in settings.

In comparison to the previous study, we decided to enhance the set of performance metrics used for rating calculation in our experiments. To avoid redundancy, only one of the complementary success and failure metrics was kept, as the Success Call Rate and Failure Rate (with the inverted value) have the same contextual meaning when analyzing the state of communication between services. Additionally, a couple of new metrics were introduced to expand the behavior.

The Permitted Call Rate is updated after each invocation attempt and reflects the recent availability trend of the protected service.

The Consecutive Failure Streak metric accumulates failed requests until a successful one occurs, after which the counter resets to zero. Its value is then normalized to remain within a fixed [0, 1] interval. To calculate it, we added a small addition to the pattern's base code, tracking consecutive failures through a simple internal counter.

The influence coefficients were slightly adjusted compared to previous work to reflect the updated metrics while maintaining the model's overall balance (see the previous set of metrics in Section 1.1). The main idea was to preserve the weight of reliability- and time-related indicators, but also allow some space to the new ones that capture how the Circuit Breaker behaves and how short-term instability appears.

The coefficients used in the experiments are shown in Tab. 1.

Table 1

Influence coefficients of metrics used for service communication rating calculation

Metric	Orientation	Coefficient	Description
Success call rate	+	0.3	The ratio of successfully completed requests to the total number of requests within the sliding window.
Slow call rate	-	0.15	The ratio of requests exceeding the preconfigured response time threshold. Adds the context of request processing speed.
Permitted call rate	+	0.2	The ratio of requests permitted by the Circuit Breaker to the total number of attempted requests. Shows communication invocation availability.
Consecutive failure streak	-	0.1	The normalized length of the current sequence of consecutive failed requests. Captures short-term spikes of instability.
Time in the Open state	-	0.25	The current duration of the Circuit Breaker operating in the Open state. Shows the context of the current waiting time denying the requests.

Since the goal of the new CB implementation is to reduce the potential time spent in the Open state while the target service has already recovered from the outage and thus increase the percentage of successful invocations under load, the experiments were designed as a simulation where one microservice (utilizing the CB) makes a large number of requests to another service. The target service, in turn, tends to fail with a random probability and then recover after a random time within a fixed interval. The response times are generated randomly, limited by a fixed timeframe.

To simulate load fluctuations, we designed the workload generation process to follow a precomputed trace of request intensities over a fixed period of experiment runtime. The load was computed using the intensity function, which was defined as a smooth oscillating signal with additive noise, representing natural load variations. We configured the duration of one full oscillation cycle to be 20 min, while the total duration of each experimental run was 30 min, resulting in 1.5 cycles per run.

Since the workload trace was precomputed, it determined the number of requests to be sent at each discrete tick.

For each time tick Δt, the expected request rate $\lambda(t)$ was calculated, and the final workload trace can be represented as (10):

$$\lambda(t) = \lambda_{min} + A(1 + sin(\frac{2\pi t}{T})) + \eta(t), \quad \eta(t) = Poisson(\mu_n), \tag{10}$$

where λ_{min} is the baseline request rate, A is the load amplitude, T is the period of oscillation, and $\eta(t)$ is a Poisson-distributed random noise with a mean value μ_n.

Each computed value $\lambda(t)$ was rounded to the nearest integer to obtain the number of requests issued during the time interval Δt, and these requests were distributed across a fixed-size worker pool of 16 threads. The purpose of the mentioned worker pool was to act like independent users that constantly initiate a varying number of invocations of the source service (where the CB is configured). The same workload trace was reused across all runs to ensure consistency and comparability between experiments.

For this study, the following values were used for the parameters mentioned above: λ_{min} = 1000 requests/s, A = 800 requests/s, T = 1200 s (20 min), Δt = 0.5 s, μ_n = 200.

During a 30-minute run, this workload generated approximately:

- 3.5 million requests in total (220 thousand requests per worker);
- an average intensity of around 120 requests/s per worker;
- a minimal intensity of around 420 requests/s;
- a maximum intensity of around 3100 requests/s.

Such a configuration provided a stable yet sufficiently dynamic load for observing how both Circuit Breaker implementations process the requests.

To model the behavior of an external dependency, a simulated target service was implemented, where we defined internal pseudo-randomized logic. The only purpose of this service was to simulate realistic failure and recovery processing instead of performing a real business logic, so each API endpoint invocation was followed by responding with a delaying thread execution, instead of the real downstream call.

The target service was switching between three operational states:

- UP – the service is healthy, and returns a successful response after a pseudo-random delay (taken from a uniform distribution between T_{min} and T_{max} values).
- DEGRADED – the service is partially healthy, meaning it returns either a successful response after a delay, or fails with a preconfigured probability.
- DOWN – the service is not operational. All requests return a failed response after a maximum delay.

Transitions between the mentioned states were implemented by a semi-random schedule, controlled by a timer that recalculated the next transition interval based on semi-random values within a fixed range. The idea behind this design was to replicate the temporal instability and simulate a real-world microservice, where the service's availability may be dependent on load spikes, resource exhaustion, or network issues.

The main configuration parameter values used in the process described above (and fixed for the experiment) are presented in Tab. 2.

Table 2

State configuration parameter values used in the experiments

Parameter \ State	UP	DEGRADED	DOWN
Response time, ms	50–400	200–2500	3000
Failure probability	0.01	0.15	1.0
Probability of switching to the DOWN state	0.05	0.2	-
Probability of switching to the DEGRADED state	Load-dependent: • Base degradation probability: 0.04 • Overload coefficient (sensitivity): 0.3 • Overload threshold: $0.7\lambda_{max}$, where λ_{max} is the current maximum request rate	-	0.3
Probability of switching to the UP state	-	0.4	0.6
Random delay bounds before switching back to UP state, s	-	1-10	1-10

The interval between state recalculations was set to 10 s.

We used the same pseudo-random seeds in our experiments to ensure the controlled and deterministic behavior during the testing of two CB implementations. This made the pseudo-random parts of the environment's behavior controllable, allowing performance differences to be attributed solely to the use of different Circuit Breaker models.

We also used the following Resilience4j configuration parameters of the traditional model of Circuit Breaker:

- Fixed sliding window size: 1000 requests.
- Slow-call duration threshold: 3.75 s.
- Time to wait in the Open state: 4 s.
- Failure rate threshold: 50 %.
- Slow-call rate threshold: 50 %.
- Permitted calls in Half-Open state: 20.

The default Circuit Breaker was used as a base for implementation by following the instructions documented in (Resilience4j, 2025). The CB Prediction Model was implemented via overriding the basic CB classes of Resilience4j components. Because the library already monitors the metrics selected earlier, it was very useful for this implementation. For more specific implementation details, refer to our previous work (Hlybovets, & Paprotskyi, 2024).

For the CB Prediction Model, we decided to use the following adapting window configuration described in detail in Section 1.3: smoothing factor (7) γ = 0.2, recomputation cycle = 30 seconds, the target count-based sliding window $W_{target,t}$ is computed (8) with the scaling coefficient α = 0.6, and bounds W_{min}= 300 and W_{max}= 2600. To prevent oscillations, the window is only changed when the relative deviation from the current value exceeds 20% upward or 8% downward. The starting window size W_0 = 1000 requests, so the traditional and adaptive versions of CB start with the same window size.

Circuit Breaker. Given the setup specified above, the simplest way to compare the difference in pattern accuracy between the proposed model and the regular implementation is to run the simulation only using the Circuit Breaker.

Each request from the previously mentioned workers is targeted at the source service's API, which processes it by calling the invocation function wrapped in a particular Circuit Breaker implementation. In this case, either the default implementation from the Resilience4j library or our enhanced implementation of the Circuit Breaker Prediction Model was used. The invocation function, in turn, makes another synchronous request to the API of the target service. The effectiveness of this communication is measured by calculating the success rate statistics.

See Fig. 2 for a schematic representation of this setup in our experiments.

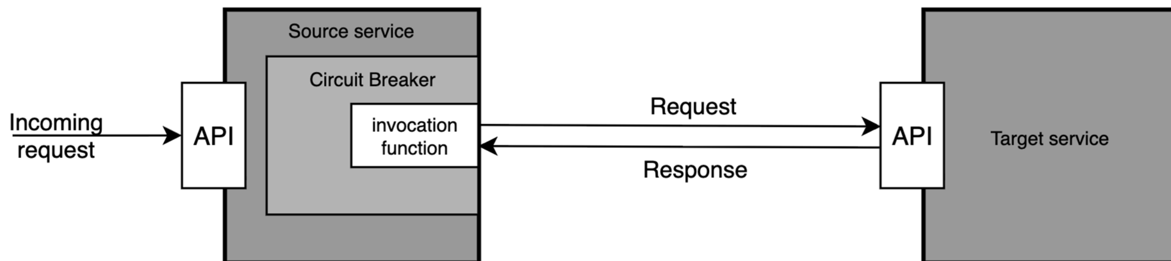

Fig. 2. The simplest Circuit Breaker usage in the experiment configuration

The results of applying the environment described in Section 1.4 to this and all the subsequent experimental configurations listed later in this section are shown in Section 2.

Circuit Breaker and Retry. The combination of the CB and Retry patterns was analyzed and recreated, since these patterns are often used together. The Resilience4j framework already provides a highly customizable implementation of the Retry pattern, which meets the needs of this research. It is important to mention that the Retry pattern context was wrapped by the Circuit Breaker, thus providing the following invocation flow: request, then a sequence of retries if a failure occurs, then the logic of the Circuit Breaker in case the number of retries is exceeded.

The schematic representation of this process is shown in Fig. 3.

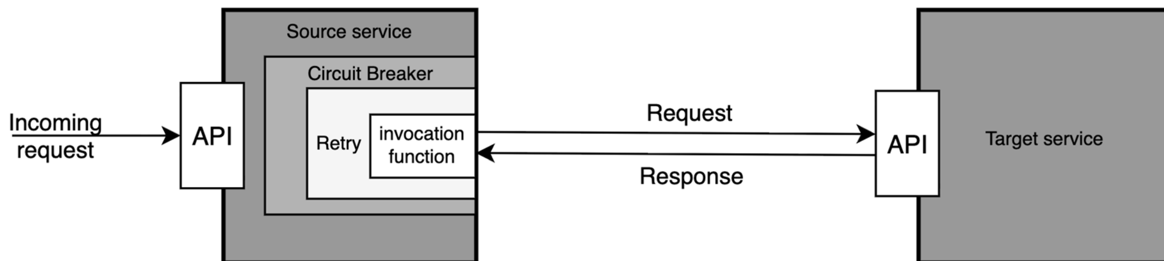

Fig. 3. The combination of Circuit Breaker and Retry in the experiment configuration

This way, most small failures, such as network blips, should be covered by the logic of the Retry mechanism, leaving the Circuit Breaker with the duty of handling more significant outages and issues.

A simple configuration with two retries and an exponential backoff was established, resulting in the following delay order between attempts: 0.5 seconds and 1 second.

Circuit Breaker, Retry, and Timeout. It is also a common use case when the Timeout is used together with the already mentioned patterns. In Resilience4j, this pattern's implementation is called a TimeLimiter, and its configuration is simple: a single duration is set, after which the request is considered failed. We put a Timeout to wrap the invocation function first (see Fig. 4), so the invocation flow results in the following: if a request takes too long to receive the response, the Timeout breaks this function's invocation, and the first retry is executed; in case this happens more times, and after all the retries were executed, the Circuit Breaker's logic considers this function's work as failed.

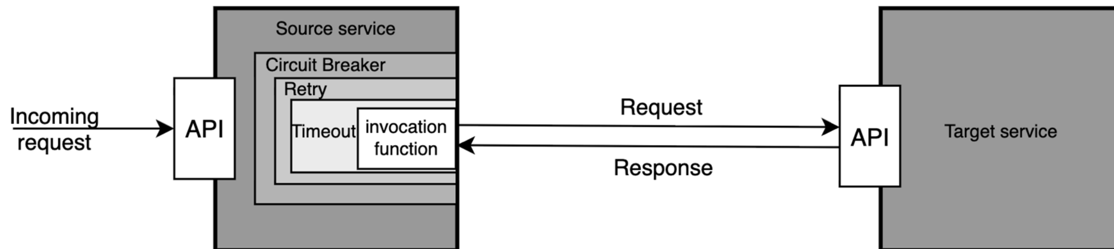

Fig. 4. The combination of Circuit Breaker, Retry, and Timeout in the experiment configuration

In our setup, we use a 2-second timeout. Since we previously configured the two retries with delays of 0.5 seconds and 1 second, this resulted in a worst-case response time of up to 7.5 seconds. Half of this duration (3.5 seconds) is already considered a slow request by both Circuit Breaker models.

Three Services: Circuit Breaker, Retry, and Timeout. Since one of the CB's main goals is to reduce the probability of cascading failures, the impact of using the different implementations should also be analyzed in a more complex setup.

This experiment involves adding an additional service to the call chain and configuring the Circuit Breaker accordingly (Fig. 5).

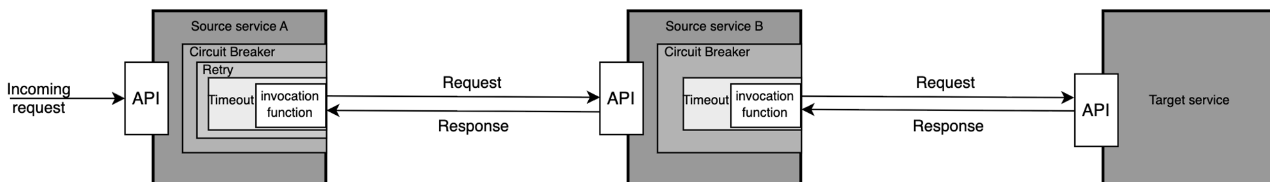

Fig. 5. Chaining the calls – adding an additional service with Circuit Breaker and Timeout in the middle of the flow

Service A sends a request to Service B, which in turn sends a request to the Target service synchronously.

In this two-hop configuration, retries were applied only at the edge (Service A). Service B used a Circuit Breaker with a per-attempt TimeLimiter but no retries. This choice avoids retry-storm amplification and better reflects production guidance for multi-hop RPC chains. We also had to increase the Timeout duration on Service A to 2.25 s (consequently, the worst-case time budget was increased to 8.25 s) and the slow call threshold on Service A to 4.0 s. All other parameters remained identical to the single-hop experiments to maintain comparability.

The percentage of successful requests in this setup only counts those that successfully invoke all three services in the chain and return a response, thereby processing the entire end-to-end flow.

2. Results

While running all the previously mentioned scenarios, we collected the following primary metrics to compare the traditional and predictive models of the Circuit Breaker: request success rate, end-to-end (e2e) 95th percentile (p95) latency, and the fraction of time in the unhealthy state (Half-Open + Open for the traditional model, and only Open for the Prediction model). To count the success rate, we used lock-free accumulators. Latency statistics and the CB state-based unhealthy time were accumulated from internal transition events using aggregated counters. All the aggregations were performed in memory to avoid the I/O overhead and any additional logging during the experiment runs.

Each experiment scenario was run in four iterations. The first was run using the traditional implementation of CB. The other three were run using the Circuit Breaker Prediction model, but with a different threshold value for the calculated rating, which triggered the state change from Open to Closed. We decided to use the following values for that parameter (based on empirical research): 0.60, 0.65, and 0.70.

In Tab. 3, we present the results gathered from the first experimental setup described in Section 1.4.1.

Table 3

Performance results: Circuit Breaker only configuration

Model	Rating Threshold	Success rate (%)	E2e p95 latency (ms)	Unhealthy time (%)
Traditional CB	–	53.4	2520	27.6
Prediction CB	0.6	60.8	2388	21.9
	0.65	59.7	2426	22.7
	0.70	55.1	2483	25.8

Under the basic configuration, both Circuit Breaker versions showed similar trends in stability and latency during load oscillations. However, the predictive model consistently maintained a higher success rate and shorter recovery times. Compared with the traditional CB implementation, it increased the overall success rate by approximately 7–9 percent, reduced the 95th-percentile end-to-end latency by around 6 %, and decreased the total unhealthy time by almost 5 %. These

improvements indicate that the adaptive window mechanism provided a more responsive adjustment to traffic variations, maintaining better throughput even under unstable service conditions.

After adding the Retry pattern usage to the given setup (Section 1.4.2), the following results were obtained (Tab. 4).

Table 4

Performance results: Circuit Breaker and Retry configuration

Model	Rating Threshold	Success rate (%)	E2e p95 latency (ms)	Unhealthy time (%)
Traditional CB	–	74.2	2260	12.8
Prediction CB	0.6	86.1	2142	10.1
	0.65	84.7	2170	10.6
	0.70	79.8	2229	12.1

After adding the retries, as shown in the table, overall service availability increased notably, with the success rate increasing from roughly 74 % to over 86% in the best-performing configuration of the predictive model. In parallel, the unhealthy time dropped from 12.8 % to nearly 10 %, suggesting that the model allowed faster reopening of the circuit once the target service resumed stable operation.

Adding the Timeout pattern to the previous configuration (Section 1.4.3) provided us with the following results (Tab. 5):

Table 5

Performance results: Circuit Breaker, Timeout, and Retry configuration

Model	Rating Threshold	Success rate (%)	E2e p95 latency (ms)	Unhealthy time (%)
Traditional CB	–	83.5	1764	8.8
Prediction CB	0.6	94.6	1639	6.9
	0.65	93.8	1660	7.1
	0.70	90.1	1712	7.9

As expected, the addition of the Timeout pattern reduced the total response time, so the difference between the CB Prediction Model and the traditional CB was almost 7 % in the p95 latency and about a 10–11 % in the success rate.

Finally, introducing the scenario discussed in Section 1.4.4, presented in Tab. 6:

Table 6

Performance results: two-hop chain – Service A to Service B to Target service, Retries only at A

Model	Rating Threshold (A / B)	Success rate (%)	E2e p95 latency (ms)	Unhealthy time (%) at A / B
Traditional CB	–	69.8	3010	19.8 / 17.5
Prediction CB	0.60 / 0.60	81.6	2790	15.7 / 13.2
	0.65 / 0.65	80.2	2831	16.2 / 13.6
	0.70 / 0.70	76.4	2905	17.4 / 14.9

In the two-service chain setup, overall latency and error accumulation increased as expected, but the general pattern remained consistent. It improved the end-to-end success rate by approximately 11-12%, reduced the 95th-percentile latency by about 8%, and shortened the unhealthy periods at both Service A and Service B by up to 3-4%. These results demonstrate that the approach continues to provide value in chained communication scenarios, where slower recovery recognition in any upstream service can lead to additional failures.

In all experiment configurations, different scenarios with varying Rating Threshold values showed that overall success depends on the correctly selected rating limit for transitioning from Open to Closed states of CB. However, even manual tuning of that parameter was intuitive: the higher the threshold, the more strict we want our CB to be towards communication instabilities.

Discussion and conclusions

The conducted study expanded the earlier work on the Circuit Breaker Prediction Model presented in (Hlybovets, & Paprotskyi, 2024) by improving it and validating its performance under more realistic and dynamic conditions.

The model was extended with an adaptive sliding window controller that scales the data window size according to the observed throughput. This increased the model's ability to adapt to workload fluctuations by providing a more accurate data context for calculating service communication performance.

The main concept of this model's calculation algorithm was generalized in the form of communication rating calculation formulas, allowing the use of any substantial number of performance metrics within the algorithm. The pseudocode instructions listing was also included to ensure reproducibility.

Through a series of simulation-based experiments combining the model with other fault-tolerance patterns, it was shown that adaptive estimation of communication performance can improve the CB recovery speed and reduce unnecessary downtime. Across all experiment configurations, the predictive Circuit Breaker implementation demonstrated an improvement in request success rate ranging from 7 % to 12 %, a reduction of 5–8 % in the 95th-percentile end-to-end latency, and a decrease of 4–6 % in total unhealthy operation time compared with the traditional model. The configurations involving two microservices demonstrated that integrating the prediction-driven rating of service communication calculation (1) increases the successful requests rate and reduces request latency, especially when combined with other stabilizing patterns like the Retry and the Timeout. When testing the three-service configuration, the experiment results remained positively stable: the percentage of successful requests remained better in the prediction-based Circuit Breaker, proving its better resilience capabilities in withstanding cascading failures.

As a result of the above-mentioned, we consider the study's goal to have been achieved.

The process of preparing the experiments revealed the role of tuning the parameters and thresholds to achieve optimal fault tolerance performance in this method. The tasks of adjusting thresholds and identifying perfect correlations between various performance metrics are complex for the human mind. It should be possible to do it more effectively by using automated data-driven analytical approaches and various optimization techniques.

The conclusions align with the study (Souza, Neves, & Kimura, 2024), which analyzed industry practices in research papers and surveyed microservice practitioners. It highlighted standard practices, gaps, supporting technologies, and the evolution of fault tolerance. The study confirms the benefits of continuously improving resilience patterns and methods, and emphasizes the growing integration of AI-based approaches into industry resilience strategies.

This points toward the future of prediction-based fault tolerance, where machine-assisted analysis may help reveal complex dependencies among runtime metrics and improve adaptive decision-making.

Authors' contribution: Anatolii Pashko – conceptualization, formal analysis, methodology, supervision, validation, writing – review & editing; Ihor Paprotskyi – conceptualization, formal analysis, investigation, methodology, software, validation, visualization, writing – original draft, writing – review & editing; Andrii Hlybovets – conceptualization, methodology, writing – review & editing; Svitlana Terenchuk – conceptualization, methodology, validation, visualization, writing – original draft.

Sources of funding. Funding is partially provided by Taras Shevchenko National University of Kyiv.

References

De Souza Miranda, F., Santos, D. S. D., Vilela, R. F., Assunção, W. K. G., Santos, R. C. D., & Pinto, V. H. S. C. (2024). A proposed catalog of development patterns for fault-tolerant microservices. In I. Machado, J. Maldonado, T. Conte, E. Canedo, J. Marques, B. B. de França, P. Matsubara, D. Viana, S. Soares, G. Santos, L. Rocha, B. Gadelha, R. Santos, A. Oran, & A. G. Seca Neto (Eds.), *SBQS 2024: XXIII Brazilian Symposium on Software Quality* (pp. 406–416). Association for Computing Machinery. https://doi.org/10.1145/3701625.3701678

Zhang, P., Xiang, L., Song, Z., & Yang, Y. (2025). Adaptive load balancing and fault-tolerant microservices architecture for high-availability web systems using docker and spring cloud. *Discover Applied Sciences*, 7(7). https://doi.org/10.1007/s42452-025-07320-7

Fowler, M. (2014). Circuit Breaker. martinfowler.com. https://martinfowler.com/bliki/CircuitBreaker.html

Sun, X., Cui, B., & Cai, Z. (2024). Deep Q-learning based circuit breaking method for micro-services in cloud native systems. In Y. Sun, T. Lu, T. Wang, H. Fan, D. Liu, & B. Du (Eds.), *Communications in computer and information science: Vol. 2012. Computer supported cooperative work and social computing: ChineseCSCW 2023* (pp. 348–362). Springer. https://doi.org/10.1007/978-981-99-9637-7_26

Souza, V. J. S., Neves, V. O., & Kimura, B. Y. L. (2024). Dependable Microservices in the Kubernetes era: A Practitioners survey. *Journal of Internet Services and Applications,* 15(1), 561–583. https://doi.org/10.5753/jisa.2024.4000

Istio. (2025). *Documentation*. https://istio.io/latest/docs

JetBrains. (2024). *General Development Trends –The State of Developer Ecosystem in 2023 Infographic. JetBrains: Developer Tools for Professionals and Teams*. https://www.jetbrains.com/lp/devecosystem-2023/development/#mcrsrvc_design_approaches

Hlybovets, A., & Paprotskyi, I. (2024). Increasing the fault tolerance in microservice architecture. Cybernetics and Systems Analysis, 60(3), 480–488. https://doi.org/10.1007/s10559-024-00689-0

Resilience4j. (2025). Introduction. https://resilience4j.readme.io/docs/getting-started

Lewis, J., & Fowler, M. (2014). Microservices. martinfowler.com. https://martinfowler.com/articles/microservices.html

Valdivia, J. A., Lora-González, A., Limón, X., Cortes-Verdin, K., & Ocharán-Hernández, J. (2020). Patterns Related to Microservice Architecture: a Multivocal Literature Review. *Programming and Computer Software,* 46(8), 594–608. https://doi.org/10.1134/s0361768820080253

Nygard, M. T. (2018). *Release it!: Design and Deploy Production-Ready Software* (2nd ed.). The Pragmatic Programmers LLC.

Stack Overflow. (2024). Stack Overflow Developer Survey 2023. https://survey.stackoverflow.co/2023/#section-developer-experience-developer-experience-processes-tools-and-programs-within-an-organization

Отримано редакцією журналу / Received: 20.10.25
Прорецензовано / Revised: 18.03.26
Схвалено до друку / Accepted: 16.04.26
Опубліковано / Published: 05.06.26

Анатолій ПАШКО[1], д-р фіз.-мат. наук, проф.
ORCID ID: 0000-0001-6944-8477
e-mail: aap2011@ukr.net

Ігор ПАПРОЦЬКИЙ[1], асп.
ORCID ID: 0009-0006-1644-2833
e-mail: igorpapr@gmail.com

Андрій ГЛИБОВЕЦЬ[2], д-р техн. наук, проф.
ORCID ID: 0000-0003-4282-481X
e-mail: a.glybovets@ukma.edu.ua

Світлана ТЕРЕНЧУК[3], канд. фіз.-мат. наук, проф.
ORCID ID: 0000-0001-6527-4123
e-mail: terenchuksa@ukr.net

[1]Київський національний університет імені Тараса Шевченка, Київ, Україна
[2]Національний університет "Києво-Могилянська академія", Київ, Україна
[3]Київський національний університет будівництва і архітектури, Київ, Україна

УДОСКОНАЛЕННЯ ПРЕДИКТИВНОЇ МОДЕЛІ ПАТЕРНУ CIRCUIT BREAKER ДЛЯ ПІДВИЩЕННЯ ВІДМОВОСТІЙКОСТІ МІКРОСЕРВІСІВ

У високонавантажених розподілених системах надзвичайно важливо приділяти особливу увагу надійності комунікації між архітектурними компонентами, що робить відмовостійкість однією з базових властивостей мікросервісної архітектури. Попри наявність різноманітних патернів підвищення відмовостійкості, спрямованих на поліпшення стійкості мікросервісів, жоден із них не є

універсальним для всього спектра можливих відмов у сервісній взаємодії. З огляду на це зберігається постійна потреба в пошуку ефективніших підходів до досягнення відмовостійкості в розподілених програмних системах.

Спираючись на наше попереднє дослідження, у якому було представлено предиктивну модель Circuit Breaker, пропонована робота детальніше досліджує її застосування для оцінювання ефективності в поєднанні з іншими поширеними патернами відмовостійкості. Центральним принципом предиктивної моделі є обчислення рейтингу комунікації між двома сервісами, визначеного як опукла комбінація різних метрик ефективності. Отримане значення рейтингу використовується для визначення того, чи може взаємодія із цільовим сервісом бути безпечно відновлена після періоду блокування. Подібно до традиційного патерна Circuit Breaker, запропонована модель спрямована на зменшення затримок під час передавання запитів в умовах невизначеного навантаження. Порівняно зі стандартними імплементаціями цього методу модель розроблено з метою скорочення часу блокування викликів між мікросервісами.

Метою статті є вдосконалення моделі Circuit Breaker та перевірка її ефективності в поєднанні з іншими патернами відмовостійкості, зокрема Retry та Timeout. У дослідженні розширено гнучкість Circuit Breaker Prediction Model шляхом опису адаптивного механізму ковзного вікна для метрик, що динамічно змінює розмір контексту залежно від навантаження під час роботи застосунку. Також алгоритм обчислення рейтингу було узагальнено, щоб забезпечити можливість використання довільної кількості метрик ефективності. У межах роботи проведено серію симульованих експериментів із різними конфігураціями комунікації між сервісами та комбінаціями патернів відмовостійкості, використовуючи стандартну реалізацію Circuit Breaker як базову для порівняння.

Отримані результати показують підвищення продуктивності комунікації насамперед шляхом зменшення затримок у передаванні запитів за однакових умов навантаження. Також розглянуто перспективи застосування принципів цієї моделі до інших патернів відмовостійкості, зокрема з акцентом на можливості використання різноманітних методів оптимізації для підтримки динамічного ухвалення рішень.

К л ю ч о в і   с л о в а : *мікросервісна архітектура, розподілені системи, відмовостійкість, патерни відмовостійкості, високе навантаження, Circuit Breaker, взаємодія сервісів, адаптація під час виконання.*

Один з авторів (А. Пашко) є членом редколегії видання, тому він не брав участі у рецензуванні та прийнятті рішення щодо публікації цієї статті.

Автори заявляють про відсутність конфлікту інтересів. Спонсори не брали участі в розробленні дослідження; у зборі, аналізі чи інтерпретації даних; у написанні рукопису; в рішенні про публікацію результатів.

One of the authors (A. Pashko) is a member of the journal editorial board, therefore did not take part in the peer-review process or in the decision to publish this article.

The authors declare no conflicts of interest. The funders had no role in the design of the study; in the collection, analyses or interpretation of data; in the writing of the manuscript; in the decision to publish the results.