UDC 004.9
DOI: 10.15587/1729-4061.2025.325932

CACHE INVALIDATION BASED ON A DECLARATIVE APPROACH FOR SEPARATING BUSINESS LOGIC OF MICROSERVICES FROM CACHE UPDATE RULES

Vitalii Falkevych
Corresponding author
Department of Computer Science*
E-mail: vitaliifalkevich@gmail.com
Andrii Lisniak
PhD
Department of Software Engineering*
*Zaporizhzhia National University
Zhukovskoho str., 66, Zaporizhzhia, Ukraine, 69600

The object of this study is the mechanisms of cache invalidation within a microservice architecture. The research addresses the challenge of reducing inter-service dependences, enhancing system performance, and ensuring data consistency in distributed high-load environments by optimizing cache management strategies. The study's findings include the development of a declarative approach to cache invalidation, which ensures the decoupling of microservice business logic from cache update mechanisms. The proposed solution is based on a centralized cache management strategy utilizing YAML configurations in conjunction with asynchronous message exchange between services.

Optimization of the mechanisms for adding, updating, and deleting cache elements enables efficient management of cached data, warranting the avoidance of residual entries after deletion, updating all lists in which the element appeared, as well as maintaining cache consistency when updating or deleting data.

The proposed approach contributes to increasing the autonomy of microservices, reducing inter-service dependences, and more efficient use of the cache. The features and distinctive characteristics of the results relate to the fact that the proposed approach uses a declarative cache management model, which differs from conventional imperative solutions. This enables flexibility in configuring cache update mechanisms without the need to modify the business logic of microservices.

The practical implications of the study extend to high-load distributed systems where rapid data retrieval, scalability, and resilience to varying workloads are critical. The proposed approach could be applied in the design of effective caching strategies within microservice-based architectures

Keywords: cache invalidation, Redis, Memcached, declarative approach, cache management, performance optimization

Received 10.01.2025
Received in revised form 05.02.2025
Accepted date 24.03.2025
Published date 22.04.2025

How to Cite: *Falkevych, V., Lisniak, A. (2025). Cache invalidation based on a declarative approach for separating business logic of microservices from cache update rules. Eastern-European Journal of Enterprise Technologies, 2 (2 (134)), 68–74. https://doi.org/10.15587/1729-4061.2025.325932*

1. Introduction

Microservice architecture is widely used in the construction of modern highly loaded web systems as it enables their scalability, flexibility, and independence of the deployment of individual services. At the same time, with the growth of data volumes, the number of simultaneous users and the need for operational access to information, new challenges arise related to the performance of systems. One of the key tools for overcoming them is caching, which makes it possible to reduce the load on servers, reduce the frequency of database requests, and speed up the processing of user requests [1].

Despite significant advantages, the use of caching is accompanied by a number of problems, in particular, the need to ensure data relevance, the complexity of synchronizing the cache with changes in the system and the potential possibility of using outdated information. The lack of standardized cache management methods complicates its support and can negatively affect the reliability of the system [2].

There are various approaches to solving these problems, including Spring Cache, GraphQL/Apollo Client, Redis & Cache Invalidation Rules. Spring Cache provides an easy-to-use caching mechanism integrated into the Spring ecosystem, making it possible to cache the results of method calls without significant code changes. It supports automatic caching using annotations, which makes it attractive for rapid integration. However, this approach does not solve the problem of global cache consistency across microservices and requires manual invalidation management, which can become a point of failure in distributed systems. GraphQL/Apollo Client offers normalized client-side caching, which allows data to be automatically refreshed when new queries are made. Apollo Client has cache management mechanisms that make it possible to track relationships between entities, but this approach is mainly focused on the client level and does not solve the problem of cache consistency across services. In addition, to maintain data freshness, one needs to manually configure cache refresh policies, which adds complexity to development.

Redis & Cache Invalidation Rules provide powerful cache management capabilities, such as TTL (Time-To-Live), LRU (Least Recently Used) eviction, and cache change notification mechanisms via Pub/Sub. Redis makes it possible to build complex caching strategies and invalidation mechanisms, which makes it a flexible solution for scalable systems. However, its use requires careful tuning of invalidation rules, in particular to maintain consistency between services. The high speed of Redis is offset by additional overhead for supporting complex invalidation logic and data replication.

Copyright © 2025 Authors. This is an open access article under the Creative Commons CC BY license

Under modern conditions, there is a need to devise a declarative approach to cache invalidation, which would facilitate a clear separation of microservices business logic from cache update rules. This could reduce the complexity of data synchronization, unify cache management mechanisms, and minimize the risks associated with untimely updating of cached information. From a scientific point of view, such research would contribute to the formalization of approaches to the integration of caching into microservice systems and the development of effective algorithms for its update [2].

From a scientific point of view, such research will contribute to the formalization of approaches to the integration of caching into microservice systems and the development of effective algorithms for its update. From a practical point of view, the results of the research could be used to increase the performance of web systems operating in real time, reduce infrastructure costs, and ensure stable operation of services under high loads. The proposed solutions could be used in various industries, in particular in e-commerce, financial technologies, streaming services, and other areas where fast data processing is critically important [3].

Thus, the problem of data consistency and effective cache management in microservice architecture remains relevant, especially in highly loaded systems. A declarative approach to cache invalidation can help reduce the complexity of caching management and improve the performance of distributed applications.

2. Literature review and problem statement

Effective cache management in distributed systems is a critical aspect of optimizing performance and data consistency. Current research focuses on memory allocation mechanisms, cache invalidation strategies, and adaptive approaches to data updates.

In [4], an analysis of memory allocation mechanisms in Memcached was conducted. The study showed that the use of the slab-based allocation approach, which involves static memory allocation between object classes, can lead to its "calcification". This limits the flexibility of the cache when data access patterns change, which, in turn, reduces memory efficiency and increases the frequency of database accesses. At the same time, the issues of dynamic cache adaptation to changing access conditions remain unresolved. The main reason for this is the limitation of conventional memory allocation algorithms, which do not take into account the change in the popularity of objects over time and require complex mechanisms for dynamic reallocation.

An approach to automatic dependence management between services and the use of the lazy-invalidation mechanism is discussed in [5]. It has been demonstrated that automating cache update processes in microservice systems can significantly reduce data update delays. However, the accuracy of tracking changes in complex call graphs between services remains a problem, due to unpredictable interaction scenarios and the possibility of superimposing independent changes on the same data.

In [6], the scalability and reliability of Redis in the context of cache invalidation were analyzed. The authors showed that Redis provides high query processing speed due to the key-value mechanism, but its cache update is based on an imperative approach, which can complicate integration in complex microservice architectures.

An important aspect of cache management is memory optimization mechanisms. In [7], the use of a segment approach in Redis was investigated, which makes it possible to reduce memory fragmentation and increase overall performance. However, it was found that such a strategy contributes to the accumulation of inactive objects, which can lead to inefficient use of resources. It was found that the main reason for this problem is the limitation of the memory reuse mechanism, which requires additional data collection strategies.

One approach to solving this problem is the use of segment utility assessment strategies to optimize memory cleaning. In [8], a method for caching Memcached on a reconfigured network interface (NIC) using FPGAs was proposed. The study showed that caching the most frequently requested data on the NIC can significantly reduce Memcached latency, minimizing access to the server's main processor. However, the limitation of FPGA hardware resources creates additional difficulties in scaling the system, as managing large amounts of data becomes more difficult. As a solution, the integration of dynamic caching and adaptive synchronization between the FPGA and the server is proposed.

Similar scalability issues are addressed in [9], which analyzes the use of Redis in distributed environments. The study demonstrates that integrating Redis with ZooKeeper provides efficient cluster management and automation of disaster recovery processes. At the same time, it was found that under conditions of high update competition, additional delays arise due to complex coordination between nodes. This indicates the need for further optimization of replication and load balancing algorithms.

Particular attention is paid to the problem of cache invalidation in the microservice architecture of web applications. In [3], a centralized cache management mechanism is analyzed, which makes it possible to reduce update delays and increase data consistency. However, the scalability of such an approach in large distributed systems remains open. The main limitations are the need for high performance of the cache management server and the risk of its overload.

All this suggests that it is advisable to conduct research on the development of a declarative approach to cache invalidation in microservices, which will ensure the separation of microservice business logic from cache update mechanisms. Such an approach should take into account both adaptive memory allocation to reduce caching delays and automatic tracking of dependences between services. It is expected that this will significantly increase the performance of distributed systems, minimize the overhead of cache updates, and improve data consistency between microservices.

3. The aim and objectives of the study

The purpose of our research is to devise a declarative approach to cache invalidation in the context of microservice architecture, focusing on separating the business logic of microservices from cache update rules. This will allow for effective management of caching processes, optimize interaction between microservices, and ensure the relevance and accuracy of data in highly loaded systems.

To achieve this goal, the following tasks must be solved:

– to determine mechanisms for dividing responsibility between microservices to ensure autonomy and reduce inter-service dependences in caching processes;

– to build a conceptual model of a declarative approach to cache invalidation that defines the structure and mecha-

nisms for separating the business logic of microservices from cache update processes;

– to optimize mechanisms for adding, updating, and deleting cache elements to reduce delays and speed up data retrieval.

4. The study materials and methods

The object of our study is the mechanisms of cache invalidation in microservice architecture.

The main research hypothesis assumes that the use of a declarative approach to cache invalidation, which involves the separation of microservice business logic and cache update rules, makes it possible to increase system performance and ensure data relevance without excessive load on services.

The following assumptions are accepted in the study:

– cache invalidation is a critical aspect of the operation of highly loaded microservice systems, which affects the performance and accuracy of data;

– the use of an asynchronous messaging mechanism reduces delays in cache updating and increases the system's resilience to loads;

– the division of responsibility between microservices and the cache makes it possible to reduce inter-service dependences and increase the scalability of the architecture.

To implement the cache invalidation mechanism, a technology stack based on TypeScript and NestJS was used. NestJS provides built-in support for microservices architecture, and communication between services is done through the @nestjs/microservices library, which allows for event-driven interaction between components. This provides efficient and asynchronous handling of cache invalidation events, which is critical for scalable solutions.

Redis is used as a message broker for asynchronous exchange between microservices. It acts as a Publish/Subscribe (Pub/Sub) mechanism that provides reliable message delivery without the need for direct calls between services. This approach allows microservices to remain loosely coupled and handle cache invalidation events independently of each other, which helps increase system performance and stability.

For the practical implementation of the proposed approach, the main structural components of the system are defined:

– client sends requests to the API gateway;

– API gateway routes requests to the corresponding services, manages caching and data updates;

– business logic service is the main service that contains business logic and processes requests;

– caching service is responsible for managing the cache and performing data invalidation;

– cache storage is a storage where cached data is stored.

An architecture diagram of an application with a cache invalidation system has been developed, which illustrates the interaction of these components and cache update mechanisms.

5. Results of the study of cache invalidation based on the declarative approach

5. 1 Separation of responsibilities between microservices for autonomous cache management and minimizing inter-service dependences

Effective cache management in a microservice architecture requires a clear division of responsibilities between services, which makes it possible to minimize inter-service dependences and ensure their autonomy. One of the key aspects of this approach is the isolation of separate components for cache management, its updating and invalidation without interfering with the business logic of microservices [10].

Fig. 1 proposes an architectural model of a cache invalidation system that enables data consistency between different services in a distributed environment. This architecture includes five key components:

– client;

– API gateway;

– business logic service;

– caching service;

– cache storage.

The client acts as the initial point of interaction with the system, sending requests to retrieve or update data. These requests are directed to the API gateway, which acts as an intermediary responsible for routing incoming requests to the appropriate backend services. In addition to forwarding requests, the API gateway also receives responses from services and manages cache update schemes. This enables that after processing the request, all necessary changes to the cache are made to maintain data integrity.

The business logic service implements the core functionality of the application by processing requests, interacting with databases or external APIs, and generating responses with the necessary data. In addition, in the case of data updates, this service generates a cache invalidation scheme – a structured set of instructions that determines which cached records should be updated or deleted. This scheme, along with the corresponding data, is passed to the API gateway, which routes it to the caching service for execution.

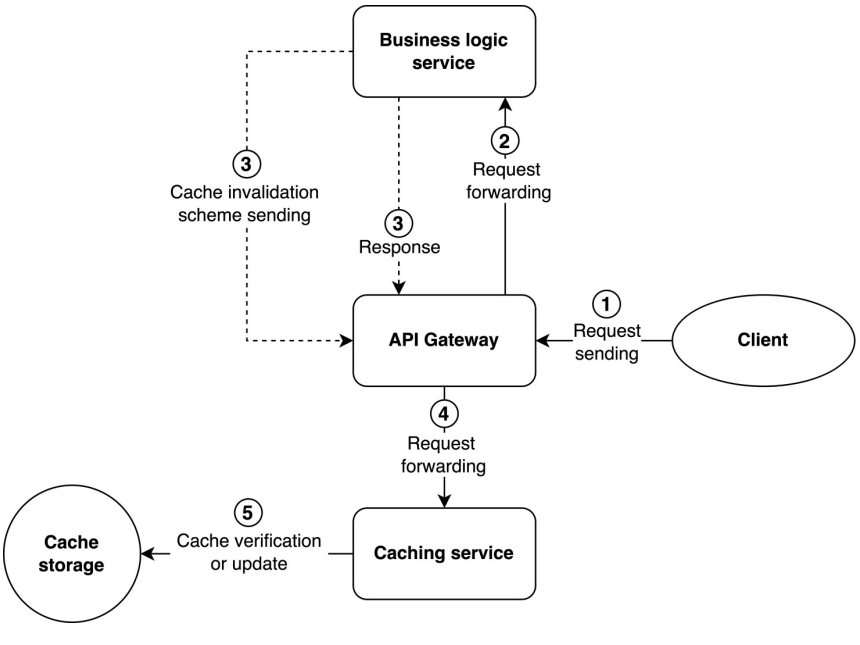

Fig. 1. Application architecture with cache invalidation system

The caching service is a specialized component responsible for managing cached data and implementing invalidation operations based on the received instructions. It acts as an abstraction between the business logic and the main caching mechanism, ensuring consistency between cached and current data. The invalidation scheme received from the business logic service is processed in the Caching Service, which determines the need to update or delete specific entries in the cache.

At the lowest level of the architecture is the cache store, which is the physical storage of cached data. Its implementation can be based on technologies such as Redis, Memcached, or other caching systems in RAM. The cache store is used to quickly retrieve stored data, as well as to update or delete entries according to requests from the caching service [11].

Thus, the distribution of responsibility between microservices allows for autonomous cache management, reducing inter-service dependences, and increasing system scalability.

5. 2. Conceptual model of a declarative approach to cache invalidation

To separate the business logic of microservices from the cache update mechanisms, a conceptual model of a declarative approach to cache invalidation is proposed, which is based on the use of external configuration files in YAML format. These files contain instructions for the cache processing service, which makes it possible to clearly define the invalidation rules without interfering with the business logic of microservices.

Fig. 2 shows a scheme for registering a business logic service in the API gateway under the identifier "BUSINESS_SERVICE". Redis is used for messaging, the connection parameters are defined by the environment variables "HOST_REDIS" and "PORT_REDIS". The API gateway routes client requests to microservices and also passes requests to the caching service to execute invalidation instructions according to the configuration.

```
@Module({
 imports: [
  ClientsModule.register([
   {name: 'BUSINESS_SERVICE',
    transport: Transport.REDIS,
    options: {
     host: HOST_REDIS,
     port: PORT_REDIS,
    }}])],
 controllers: [GatewayController]})
```

Fig. 2. Registering a business logic service in the API Gateway module

To implement the declarative approach, a YAML configuration is used that contains cache invalidation instructions (Fig. 3).

When processing requests in business logic microservices, YAML file instructions are used. For example, when receiving a list of items (Fig. 4), the cache is checked first. If the data is missing, the microservice performs a query to the database and passes the results along with the YAML instructions to the caching service.

Similarly, when creating a new element, the corresponding cache invalidation instruction is transmitted (Fig. 5).

```
title: 'cache invalidate instructions'
 service: 'Business'
 operations:
  get:
 actions:
  - operation: "add"
    target: "list"
    key: "item_get"
    action_type: "set"
  - operation: "add"
    target: "slug"
    key: "item_get_one"
    action_type: "set"
   getOne:
 actions:
  - operation: "add"
    target: "slug"
    key: "item_get_one"
    action_type: "set"
   create:
 actions:
  - operation: "add"
    target: "slug"
    key: "item_get_one"
    action_type: "update"
```

Fig. 3. Cache refresh configuration for a business logic service

```
@MessagePattern({ cmd: BUSINESS_EVENTS.ITEMS_GET })
 async getItems({ payload }: PayloadDto<GetItemsDto>):
Promise<Item[]> {
  const cacheConfig =
 this.businessService.getBusinessConfig();
  const dataFromCache = await firstValueFrom(
   this.cacheServiceClient.send(
    {cmd: CACHE_EVENTS.CACHE_CHECK},
    {key: BUSINESS_EVENTS.ITEMS_GET,payload}));
  if (dataFromCache) return dataFromCache;
  const items = await this.businessService.getItems();
  this.cacheServiceClient
   .send({cmd: CACHE_EVENTS.CACHE_RUN_INSTRUCTIONS},
    {cmd: INSTRUCTION_OPERATIONS.GET,
     instructions: cacheConfig,
     payload,
     data: items})
  return items;}
```

Fig. 4. Method for processing a request to obtain a list of items

```
@MessagePattern({ cmd: BUSINESS_EVENTS.ITEM_CREATE })
 async createItem({payload }: PayloadDto<Item>):
Promise<SuccessWithoutDataResponseDto> {
  const cacheConfig =  this.businessService.getBusinessConfig();
  const newItem = await
 this.businessService.createItem(payload);
  this.cacheServiceClient.send(
   {cmd: CACHE_EVENTS.CACHE_RUN_INSTRUCTIONS},
   {cmd: INSTRUCTION_OPERATIONS.CREATE,
    instructions: cacheConfig, data: newItem});
  return successEmptyResponse;}
```

Fig. 5. Method for creating a new element

The caching service processes the YAML file instructions through the "runInstructions" method, which performs the corresponding invalidation operations (Fig. 6).

```
async runInstructions(data: { cmd: INSTRUCTION_OPERATI
  instructions: any;
  data: any;
  payload?: Record<string, string | number>;
}): Promise<void> {
  data.instructions?.operations?.[data?.cmd]?.actions?.forEach(
    (operation) => {
      const args = {
        key: operation.key,
        filters: data.payload,
        data: data?.data,
        actionType: operation.action_type};
      switch (operation) {
        case operation.operation === OPERATION.ADD:
          if (operation.target === TARGET.LIST)
            return this.setItemsToCache(args);
          if (operation.target === TARGET.SLUG) {
            if (this.isObject(args.data))  this.setItemToCache(args);
            else if (Array.isArray(args.data)) {
              return (async () => {
                for (const item of args.data) {
                  await this.setItemToCache(item); }})();}}
        break;
        case operation.operation === OPERATION.REMOVE:
          if (operation.target === TARGET.SLUG) {
            if (this.isObject(args.data)) return this.removeItemFrom
            else if (Array.isArray(args.data)) {
              return (async () => {
                for (const item of args.data) {
                  await this.removeItemFromCache(item); }})();}}}}`
```

Fig. 6. Processing cache update instructions

Thus, the proposed conceptual model of the declarative approach allows for a clear separation of business logic from cache update mechanisms, providing declarative management of invalidation instructions through configuration files. This increases flexibility and simplifies the maintenance of the microservice architecture.

5. 3. Optimization of mechanisms for adding, updating, and deleting cache elements to reduce delays and accelerate data retrieval

Fig. 7 shows the implementation of the algorithm for adding or updating an array of elements in the cache. The algorithm allows for efficient management of cache entries using unique keys and provides fast access to data.

The main principle of the algorithm is to create a unique key for accessing cached records. This ensures that there are no conflicts between different data sets. Additionally, a backlink mechanism is implemented for fast navigation and updating of elements in complex structures.

To manage individual elements in the cache, the "setItemToCache" algorithm is used, which provides efficient updating and the ability to quickly delete related records (Fig. 8).

The algorithm above creates:

– a primary key for the list containing the element;

– a unique key for the element itself;

– a backlink that stores all contexts in which the element is used.

This makes it easy to track where a record has been used and to ensure consistent data updates. For example, if a record is included in several categories or lists, the system can quickly update or delete them.

```
generateKey(baseKey, filters) {
  if (!filters || Object.keys(filters).length === 0) return baseKey;
  const sortedFilters = Object.entries(filters)
    .sort(([a], [b]) => a.localeCompare(b))
    .map(([key, value]) => `${key}=${value}`)
    .join(':');
  return `${baseKey}:${sortedFilters}`;}

async setItemsToCache({
    key, filters = {}, data, actionType
  }: ISetItemsToCache): Promise<void> {
    const listKey = this.generateKey(key, filters);
    const list = (await this.getCacheByKey(listKey)) || [];
    switch (actionType) {
      case ACTION_TYPE.SET:
        await this.setCacheByKey(listKey, data);
        break;
      case ACTION_TYPE.UPDATE:
        await this.setCacheByKey(listKey, [...list, ...data]);}}
```

Fig. 7. Algorithm for adding/updating an array of elements in the cache

```
async setItemToCache({ key, filters = {}, data, actionType }) {
  const { slug } = data;
  const listKey = this.generateKey(key, filters);
  const slugKey = `${key}:${slug}`;
  const referencesKey = `${key}:references:${slug}`;
  const references = [listKey, slugKey];
  switch (actionType) {
    case ACTION_TYPE.SET:
      await this.setCacheByKey(slugKey, data);
      await this.setCacheByKey(referencesKey, references);
    break;
    case ACTION_TYPE.UPDATE:
      const existingData = await this.getCacheByKey(slugKey);
      const updatedData = { ...existingData, ...data };
      await this.setCacheByKey(slugKey, updatedData);
      await this.setCacheByKey(referencesKey, references);
    break;
      default:
        throw new HttpException(
          exceptions.UNSUPPORTED_ACTION_TYPE,
          HttpStatus.UNPROCESSABLE,
        );}}
```

Fig. 8. Cache item addition/update algorithm

```
async removeItemFromCache({ key, data }) {
 const { slug } = data;
 const slugKey = `${key}:${slug}`;
 const referencesKey = `${key}:references:${slug}`;
 const relatedKeys = await this.getCacheByKey(referencesKey)) || [];
  for (const key of relatedKeys) {
   const list = await this.getCacheByKey(key);
    if (Array.isArray(list)) {
     const updatedList = list.filter((book) => book.slug !== slug);
     if (updatedList.length === 0) await this.removeCacheByKey(key);
     else await this.setCacheByKey(key, updatedList);
     }}
     await this.removeCacheByKey(slugKey); }
```

Fig. 9. Algorithm for deleting/updating a single item in the cache

Similarly, the "removeItemFromCache" algorithm guarantees correct deletion of elements along with their links (Fig. 9).

This approach enables:

– avoiding residual records after deletion;

– updating all lists in which the element appeared;

– maintaining cache consistency when updating or deleting data.

The proposed caching optimization significantly improves data processing efficiency. The back-referenced structure allows for fast retrieval and update of elements in complex structures and avoids duplicate or out-of-sync records. This is especially important in high-load systems where caching plays a key role in data performance and availability.

6. Discussion of results related to the study of cache invalidation based on the declarative approach

The results of our study on the separation of responsibility between microservices confirm that the use of clearly defined areas of responsibility for each microservice makes it possible to reduce the level of inter-service dependences and increase autonomy. As shown in Fig. 1, the general architecture of the cache invalidation system provides a clear separation between the business logic of microservices and the mechanisms for updating the cache. The concept is proposed in which the cache update is initiated directly by the service that owns the relevant business data, avoiding the need for global coordination between services.

The results of our study on the development of a conceptual model of the declarative approach to cache invalidation are represented in a series of program listings and diagrams.

Registering a business logic service in the API gateway (Fig. 2) provides a centralized entry point for interaction with cached data, simplifying access management to services and their integration. The cache refresh configuration for a business logic service (Fig. 3) defines caching rules, making it possible to separate business logic from cache management aspects.

The algorithm for processing a request for a list of elements (Fig. 4) implements efficient caching for fast data access, while the algorithm for creating a new element (Fig. 5) is responsible for updating the cache when adding new records. Processing cache update instructions (Fig. 6) ensures correct and consistent changes to cached data in accordance with defined policies, which prevents inconsistent states in the cache.

Optimization of the mechanisms for adding, updating, and deleting cache elements is aimed at reducing delays and accelerating data retrieval. The use of unique keys allows for quick identification of records and enables high cache performance.

The algorithm for adding or updating an array of elements in the cache (Fig. 7) implements a batch change mechanism, which minimizes the number of read-write operations, thereby improving overall system performance. Thanks to this approach, large amounts of data can be quickly updated without significant load on the caching system.

To manage individual items in the cache, the "setItemToCache" algorithm is used, which provides efficient updating of individual records and allows for quick deletion of related data (Fig. 8). This makes it easy to track where a record has been used and ensures consistent updates. For example, if a record is included in multiple categories or lists, the system automatically updates or deletes all related cached copies, preventing the use of outdated data [12].

Similarly, the "removeItemFromCache" algorithm enables correct removal of items along with their links (Fig. 9). This mechanism prevents situations when records that are no longer used remain in the cache but occupy memory and can lead to incorrect data display.

Comparison of our results with previous studies demonstrates significant advantages of the proposed approach. For example, unlike the slab-based allocation approach [4], which turned out to be less flexible due to memory calcification, the proposed dynamic caching makes it possible to reduce fragmentation and increase resource efficiency. Similarly, the lazy-invalidation mechanism [5] reduces update delays, but has limitations in complex service interaction graphs. The approach proposed in this study uses declarative cache update mechanisms to eliminate these shortcomings.

The Redis study [6] demonstrated high query processing speed but noted the difficulty of integrating imperative cache updates into microservice architectures. The proposed approach solves this problem through declarative cache invalidation, which simplifies management and improves scalability. In addition, the problems of accumulation of inactive objects in Redis, identified in [7], were taken into account in the system design (Fig. 8), which makes it possible to avoid inefficient use of resources.

Unlike FPGA caching [8], which has scalability limitations, our approach (Fig. 9) is based on backlinks, which makes it possible to maintain cache consistency and flexibly adapt the system to changing conditions. The proposed declarative caching rules minimize coordination risks with high update competition, which was noted in [9].

Thus, the proposed solutions directly resolve the identified problems by eliminating inter-service dependences, simplifying cache invalidation, and increasing performance. The cache invalidation architecture minimizes inter-service connections (Fig. 1), the conceptual model of the declarative approach to cache invalidation provides a clear separation of business logic and caching mechanisms (Fig. 2–6), and the use of backlinks in the presented algorithms "setItemToCache" and "removeItemFromCache" (Fig. 7–9) guarantees data consistency and prevents incorrect cached states.

A feature of the proposed solutions is the construction of a conceptual model of a declarative approach to cache invalidation, which defines the structure and mechanisms for separating the business logic of microservices from cache update processes. The proposed model allows us to formalize and automate cache management, which helps minimize the overhead of updating it and improve data consistency between microservices.

The main limitations of the study are related to the peculiarities of caching implementation in microservice architecture. The proposed methods are effective under conditions of stable data structure but may require adaptation in the case of dynamic changes in the model or high variability of requests [13]. Also, the cache invalidation system depends on the correct configuration of return links, which requires careful tuning. To confirm the results, it is worth conducting additional testing on real distributed systems with different loads. In further studies, it is advisable to consider optimization for real scenarios and mechanisms for automated cache dependence management.

7. Conclusions

1. Mechanisms for dividing responsibility between microservices have been defined to ensure their autonomy and

reduce inter-service dependences in caching processes. A special feature is that the separation of cache update processes from the business logic of microservices is provided. This is achieved by implementing a separate cache service in combination with a centralized caching management mechanism at the API gateway level. This approach makes it possible to automate cache updates, improve data consistency in a distributed environment, increase system scalability, and simplify its maintenance.

2. A conceptual model of a declarative approach to cache invalidation has been built, which defines the structure and mechanisms for separating microservices business logic from cache update processes. The proposed model provides flexibility in caching management and makes it possible to easily adapt to changes in the data structure, while maintaining the autonomy of microservices. It makes it possible to effectively separate business logic and caching mechanisms without the need for changes in the microservices code when updating cache rules.

3. Optimization of the mechanisms for adding, updating, and deleting cache elements provides effective management of cached data. The proposed approach warrants the avoidance of residual records after deletion, updating all lists in which the element appeared, as well as maintaining cache consistency when updating or deleting data. Creating a unique key for accessing cached records makes it possible to avoid conflicts between different data sets. The implementation of the backlink mechanism provides fast navigation and updating of elements in complex structures, preserving the contexts of element use. This approach makes it possible to work effectively with the cache in distributed environments and supports the stability of data update processes.

Conflicts of interest

The authors declare that they have no conflicts of interest in relation to the current study, including financial, personal, authorship, or any other, that could affect the study, as well as the results reported in this paper.

Funding

The study was conducted without financial support.

Data availability

The data will be provided upon reasonable request.

Use of artificial intelligence

The authors confirm that they did not use artificial intelligence technologies when creating the current work.

References

1. Falkevych, V., Lisniak, A. (2024) Client state management using backend for frontend pattern architecture in B2B segment. Artificial Intelligence, 29 (2), 49–60. https://doi.org/10.15407/jai2024.02.049
2. Falkevych, V. G., Lisniak, A. O. (2023). Methodology of cache invalidation in microservices architecture of the web applications. Scientific Notes of Taurida National V.I. Vernadsky University. Series: Technical Sciences, 1, 131–135. https://doi.org/10.32782/2663-5941/2023.1/20
3. Faridi, M. T., Singh, K., Soni, K., Negi, S. (2023). Memcached vs Redis Caching Optimization Comparison using Machine Learning. 2023 2nd International Conference on Automation, Computing and Renewable Systems (ICACRS), 1153–1159. https://doi.org/10.1109/icacrs58579.2023.10404339
4. Carra, D., Michiardi, P. (2014). Memory partitioning in Memcached: An experimental performance analysis. 2014 IEEE International Conference on Communications (ICC), 1154–1159. https://doi.org/10.1109/icc.2014.6883477
5. Zhang, H., Kallas, K., Pavlatos, S., Alur, R., Angel, S., Liu, V. (2024). MuCache: A General Framework for Caching in Microservice Graphs. Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24), 221–238. Available at: https://www.usenix.org/system/files/nsdi24-zhang-haoran.pdf
6. Chen, S., Tang, X., Wang, H., Zhao, H., Guo, M. (2016). Towards Scalable and Reliable In-Memory Storage System: A Case Study with Redis. 2016 IEEE Trustcom/BigDataSE/ISPA. https://doi.org/10.1109/trustcom.2016.0255
7. Liu, Q. (2019). A High Performance Memory Key-Value Database Based on Redis. Journal of Computers, 14 (3), 170–183. https://doi.org/10.17706/jcp.14.3.170-183
8. Fukuda, E. S., Inoue, H., Takenaka, T., Dahoo Kim, Sadahisa, T., Asai, T., Motomura, M. (2014). Caching memcached at reconfigurable network interface. 2014 24th International Conference on Field Programmable Logic and Applications (FPL), 1–6. https://doi.org/10.1109/fpl.2014.6927487
9. Ji, Z., Ganchev, I., O'Droma, M., Ding, T. (2014). A Distributed Redis Framework for Use in the UCWW. 2014 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery, 241–244. https://doi.org/10.1109/cyberc.2014.50
10. Yang, J., Yue, Y., Vinayak, R. (2021). Segcache: a memory-efficient and scalable in-memory key-value cache for small objects. 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21), 503–518. Available at: https://www.usenix.org/conference/nsdi21/presentation/yang-juncheng
11. Rahul, Singh, K., Vrinda, Dipanshu (2024). Review Paper on Machine Learning Based Memcached Cluster Auto Scaling. Emerging Trends in IoT and Computing Technologies, 330–337. https://doi.org/10.1201/9781003535423-55
12. Almeida, D., Lopes, M., Saraiva, L., Abbasi, M., Martins, P., Silva, J., Váz, P. (2023). Performance Comparison of Redis, Memcached, MySQL, and PostgreSQL: A Study on Key-Value and Relational Databases. 2023 Second International Conference On Smart Technologies For Smart Nation (SmartTechCon), 902–907. https://doi.org/10.1109/smarttechcon57526.2023.10391649
13. Chopade, R., Pachghare, V. (2021). A data recovery technique for Redis using internal dictionary structure. Forensic Science International: Digital Investigation, 38, 301218. https://doi.org/10.1016/j.fsidi.2021.301218