Journal of Computers Vol. 35 No. 4, August 2024, pp. 259-276
doi: 10.53106/199115992024083504018

Architecture Design of Embedded Software IP Knowledge Base

Zechang Xiong[1], Cheng Chen[2*], Haojie Feng[1], Xiong Xu[3], and Zhenyan Ji[1]

[1] School of Software Engineering, Beijing Jiaotong University, Beijing 100044, China
{23126487, 23126424, zhyji}@bjtu.edu.cn

[2] Transportation and Economics Research Institute, China Academy of Railway Sciences Corporation Limited, Beijing 100081, China
chenchengcars@126.com

[3] Institute of Software, Chinese Academy of Sciences, Beijing 100190, China
xux@ios.ac.cn

Received 19 July 2024; Revised 25 July 2024; Accepted 15 August 2024

Abstract. Effective management and reuse of existing embedded software knowledge is vital for enhancing development efficiency. To better reuse embedded software knowledge, embedded software intellectual property is designed. Embedded software IP refers to a collection of reusable knowledge entities with intellectual property. Software IP knowledge base has been designed and implemented to store and manage software IP efficiently. The architecture design based on the microservice architectural pattern makes the IP knowledge base flexible and extensible. Specifically, considering software IP's characteristics and usage requirements, microservice boundaries has been defined and comprehensive architectural design and program implementation has been conducted. Within the knowledge base, IP design, IP management, IP view, retrieval and recommendation, user management, and file management service of software IP are distributed to multiple service instances. The approach reduces coupling between different modules, balances distributed loads within the system, and enhances the overall capacity of the system. It lays the groundwork for the widespread application of embedded software IP.

Keywords: knowledge base, software architecture, microservice architecture, software IP

1 Introduction

With the widespread application of embedded systems in consumer electronics [1], industrial automation [2], transportation [3], and aerospace [4], the significance of embedded software has become increasingly prominent. Embedded systems are specialized computing systems that perform dedicated functions or tasks within larger systems. They are designed to operate with limited resources and often in real-time environments, making efficient and reliable software crucial. Faced with the continuously expanding application domains and evolving technological demands, one of the challenges in embedded software development is how to improve efficiency, ensure software quality, and reduce development costs simultaneously. In this context, the reuse of existing embedded software knowledge becomes crucial.

Embedded software intellectual property (software IP) refers to reusable software entities protected by intellectual property. These software IPs are rigorously validated components with specific functionalities, clear context-dependent relationships, and well-defined port definitions. Efficient reuse of these knowledge entities is essential for accelerating development processes, enhancing product quality, and lowering costs. However, due to the complexity and diversity of software IP, effectively storing, managing, and reusing these entities poses an urgent challenge.

To address this challenge, this study proposes a microservice-based software IP knowledge base architecture design and implementation. The microservice architecture [5] divides a single application into a set of small, loosely coupled services that coordinate and collaborate to provide comprehensive functionality. This approach allows for independent development, deployment, and scaling of services, enhancing the system's flexibility and maintainability.

Following the principles of microservice, the architecture of this software IP knowledge base is divided into

* Corresponding Author

modules including software IP design, software IP management, software IP view, retrieval and recommendation, user management, and file management. This architecture covers the entire lifecycle of software IP, from creation and storage to application, providing comprehensive foundational support for software IP.

To further support the efficient management and use of software IP, we explore the storage method of software IP. As software IP actually represents a software functional module with input and output ports, the software IP knowledge base employs a graph database to store IP model information. In the graph, input ports, software IP, and output ports correspond to nodes, and the names, types, and lengths of input ports and output ports correspond to the properties of nodes while a relational database stores other descriptive data. This differentiated storage approach, tailored to the data characteristics, efficiently captures relationships between software IPs, providing efficient knowledge retrieval and recommendations based on graphs.

In summary, the designed software IP knowledge base architecture, crafted to align with the attributes of software IP, possesses features such as scalability, high reliability, and flexibility. It provides a platform to enhance the efficiency of managing and using embedded software IP, establishing a robust foundation for innovation and development in the future of embedded software domains.

2 Related Work

The microservice architecture is a method for developing an application as a set of small, autonomous services that work together [6, 7]. Each service runs in its own process and communicates with other services via lightweight mechanisms, often HTTP resource APIs. This approach contrasts with traditional monolithic architectures [8], where all components and services are interwoven into a single application. In recent years, software designed based on microservice architecture has exploded across various domains. The flexibility and scalability offered by microservices have made them a preferred choice for developing complex and large-scale applications [9].

In the realm of Smart Grid, Wang et al. [10] proposed a smart grid post-evaluation platform based on a microservice framework to visually display power generation data, enhancing the system's modularity and maintainability. Similarly, Yin et al. [11] presented a subgrid-oriented privacy-preserving microservice framework based on deep neural network for detecting false data injection attacks, demonstrating the architecture's capability to handle complex security challenges in smart grids. In the e-commerce sector, Asrowardi et al. [12] adopted the Design Science Research Method to explore microservice design for e-commerce services. Wu et al. [13] developed a B2B e-commerce platform using a microservice architecture, showcasing significant improvements in system performance and maintainability. Ivaylo et al. [14] introduced microservice architecture into an intelligent railway control system to improve service reliability. Yu et al. [15] designed an urban rail transit monitoring system based on cloud computing and microservice, further validating the architecture's versatility and effectiveness in complex application environments.

In summary, the widespread application of microservice is evident in diverse fields, demonstrating its versatility and potential to address specific challenges within each domain. The microservice architecture has been proven to have good scalability, stability, and scalability. Therefore, this paper explores the software IP knowledge base based on microservice architecture.

3 Design of Software IP

Software intellectual property [16] is a collective term for reusable software knowledge entities with intellectual property, as displayed in Fig. 1. It refers to rigorously validated software with specific functionality, clear context-dependent relationships, and well-defined port definitions, representing a highly structured and validated form of software. Essentially, software IP is a type of software functional module, with its input and output ports constituting the port [17]. A well-structured software IP should comprehensively and accurately reflect key characteristics throughout its creation and usage. It serves as an abstraction and aggregation of software knowledge for developers. In summary, the representation model of software IP can be defined as the following triad:

$$softwareIP = (KM, FM, IMP). \tag{1}$$

Where KM represents the knowledge model, FM represents the formal model, and IMP represents the implementation. The knowledge model is a structured representation of knowledge within software IP, being the most content-rich part. It is primarily described using informal natural language and graphical models. The formal model employs formal language, while the implementation part mainly includes program code. From an abstract perspective, the knowledge model is the most abstract and covers the entire lifecycle of software IP, making it the most content-rich.

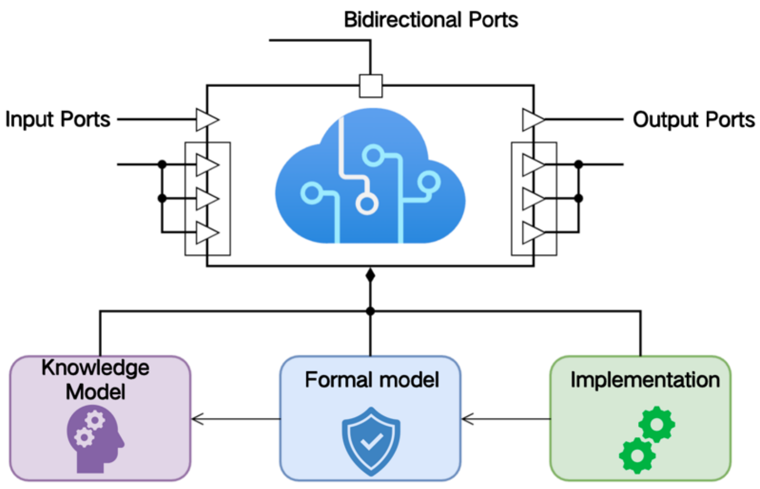


Fig. 1. Model of embedded software IP

The knowledge model assists developers and users in understanding and searching for software IP, enhancing the effectiveness of software IP utilization. As an independently deployable and composable knowledge entity, the most crucial aspect of software IP is its knowledge model. The knowledge model is structured information with instructive significance, covering the entire lifecycle of software IP. The definition of knowledge model is as follows:

$$KM = (Basic, Expert, SoftDev). \tag{2}$$

Where *Basic denotes* basic knowledge, *Expert* signifies domain expert knowledge and *SoftDev* refers to software developing knowledge. The basic knowledge of software IP provides a general overview of the software IP, encapsulating various attribute information inherent to the IP itself, which includes details such as the IP's name, ID, version, copyright ownership, authorship, keywords, and the application domain and category it belongs to. The domain expert knowledge of software IP encompasses the development standards and protocols adhered to by the IP, as well as the conditions for its use and typical scenarios in which it is applied. The software developing knowledge covers the entire lifecycle of software IP development, including documentation and records related to requirements, design, implementation, testing, and maintenance.

Formal methods involve the comprehensive and systematic use of mathematical languages, techniques, and tools to specify, develop, and verify software systems precisely [18]. The formal model describing software IP using formal methods can significantly reduce the cognitive gap between creators and users, aiding in a correct understanding of the meaning of software IP. The definition of formal model is as follows:

$$FM = (Con, Inv, NonFun). \tag{3}$$

Where *Con* stands for the contract, *Inv* represents invariant and *NonFun* signifies non-functional constraints. From a formal perspective, the contract of software IP can be characterized by logical formulas that define the relationship between its inputs and outputs, encapsulated within the form of a contract. Non-functional constraints encompass various aspects such as operational environment constraints, resource constraints, and performance constraints. Operational environment constraints include requirements for the processor, compiler, and operating system. Resource constraints pertain to the memory space occupied by the IP. Performance constraints involve the response time and execution time of the IP during its operation. These constraints can be articulated through first-order logical formulas. During the execution of software IP, it may involve state variables, and the invariant of the software IP describes the constraints that the values of these state variables consistently meet throughout the dynamic operation of the IP.

$$IMP = (Interface, Entity). \tag{4}$$

The implementation of software IP consists of two parts: interface and entity. The interface describes the externally visible part of software IP, with each software IP offering only one interface composed of input ports, output ports, and input-output ports. Taking C language as an example, the interface can be declared in the .h file of software IP code, and external environments can only access the interface through the .h file without knowledge of the implementation details. The entity of software IP exists in the form of .c, representing the concrete implementation of the software IP's functionality.

The knowledge model, formal model, and implementation constitute the main components of the software IP. In addition, each software IP entity also includes some explanatory documents. Specifically, the directory structure of a software IP is illustrated in Fig. 2 and primarily includes datasheet, knowledge structure, formal model, implementation, and product.

The datasheet is presented in document form and provides a comprehensive overview of the software IP. It encompasses essential details such as functionality, performance metrics, port descriptions, usage examples, and simulation test data. This document serves as a reference guide for developers and users, offering crucial information needed to effectively utilize the software IP.

The knowledge model, formal model, and implementation involve XML files, contracts, source code, etc. The knowledge model captures the abstract representation of the software IP, the formal model provides a precise specification, and the implementation details the actual code.

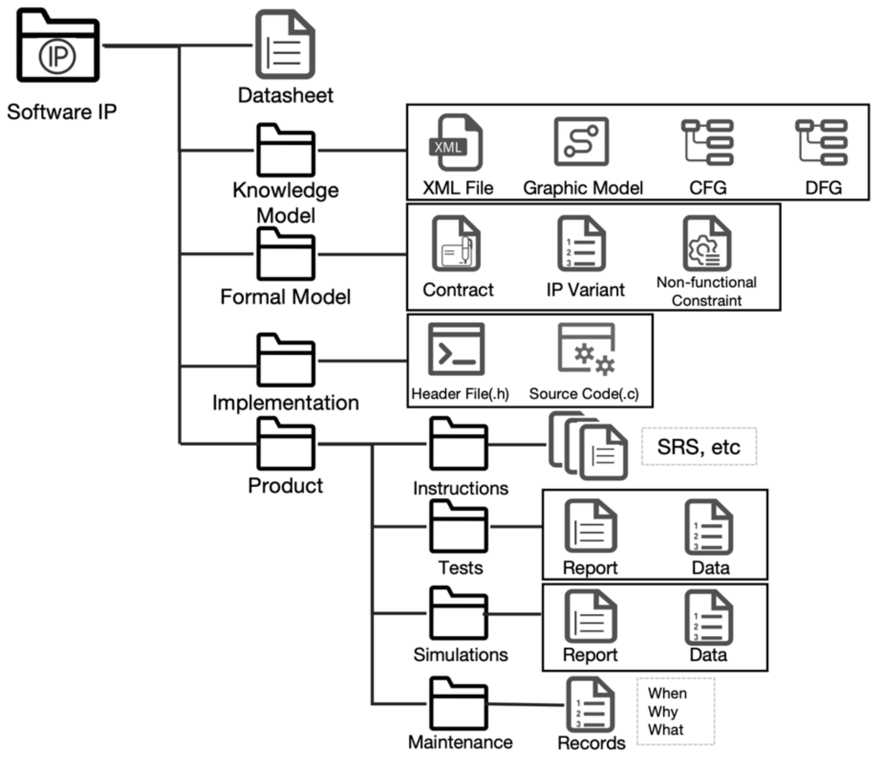


Fig. 2. Composition of software IP

The product is structured into instructions, tests, simulations, and maintenance. Instructional materials provide informative documents for reference, guiding users on how to use the software IP effectively. Testing is represented by reports and test cases within the management view, ensuring the software IP meets its quality standards. Simulations encompass both simulation reports and simulation data, providing insights into the software IP's behavior under different conditions. Maintenance records are integrated into the maintenance section of the management view, ensuring that the software IP can be effectively updated and managed over its lifecycle.

4 Architecture Design of Software IP Knowledge Base

4.1 Overall Architecture

The complexity and diversity of embedded software necessitate robust solutions for effective reuse and management [19]. The emergence of embedded software IP provides a possible solution to this challenge. To facilitate a more comprehensive and convenient utilization of software IP, we have developed the software IP knowledge base. It assists users in managing, constructing, and utilizing software IP throughout its lifecycle—from creation and storage to application.

The overall architecture of the embedded software IP knowledge base, as illustrated in Fig. 3, adopts the microservices architecture. This approach decomposes the system into multiple independent microservices, each focusing on specific functional modules. This division enhances system maintainability and scalability. By decoupling services, independent development, testing, and deployment of each module are possible, significantly improving agility and reliability.

The platform follows a Browser/Server architecture [20], requiring users to access the knowledge base through a web browser. Upon sending a request, it undergoes identity verification by the API Gateway on the server side. Once authentication is successful, the system distributes the request to the appropriate service through load-balancing mechanisms. The system consists of six main microservices: IP design service, IP management service, IP view service, recommendation and retrieval service, user management service, and file management service.

Due to the complexity of the software IP components, including IP models, descriptive text, files, etc. We utilize diversified storage methods to better accommodate the diversity and complexity of software IPs, providing comprehensive data support. Descriptive information about IPs is stored in the relational database MySQL, and a graph database Neo4j is used to store structural information. A distributed file system HDFS is employed for storing relevant files. Middleware is utilized to optimize database services, incorporating Redis for hot-cold data separation, enhancing query efficiency, ElasticSearch for optimizing indexes and Kafka for message queue.

Furthermore, infrastructure services are built for the knowledge base, including Registry Center, Configuration, Log Center, and Monitoring Center. The Registry Center facilitates the registration and discovery of microservices, enabling efficient service communication. Configuration is responsible for managing and dynamically configuring the configuration information of microservices. Log Center stores and manages logs generated by the microservice system, aiding in tracking issues and analyzing system behavior. The Monitoring Center monitors the operational status and performance metrics of the microservice system.

This architecture divides the knowledge base into independent microservices, enhancing system maintainability. It employs diverse storage methods to meet the complex storage requirements of software IPs. Middleware [21] is introduced to optimize service response times, and infrastructure services ensure the stability, scalability, and maintainability of the knowledge base. Through these comprehensive optimization measures, the knowledge base is ensured to meet the storage and usage requirements of software IPs, operating robustly in a complex and dynamic environment.

4.2 Design of Data Storage

Considering the diverse composition of software IP information, which includes storage in .docx format for datasheets, IP descriptions, and descriptions of IP structures, we employ three types of databases for collaborative storage and utilize three middleware solutions for acceleration. The diversity in databases ensures the knowledge base effectively stores and manages various forms of software IP-related data. The three databases used in this system, as shown in Fig. 3 include the relational database MySQL, the non-relational database Neo4j, and the distributed file system HDFS.

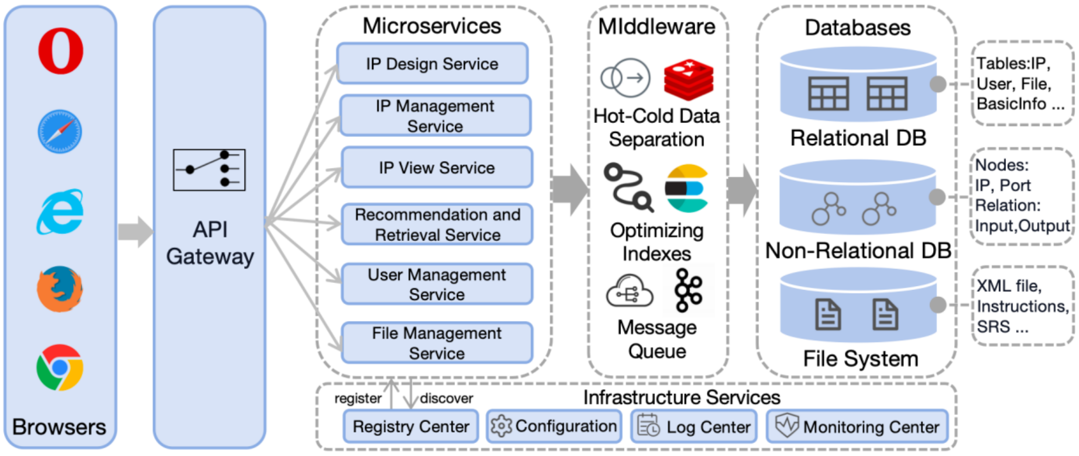


Fig. 3. Embedded software IP knowledge base software architecture

MySQL is responsible for storing structured textual data, and its entity-relationship model is illustrated in Fig. 4. The main entities include IP, User, File, Port, BasicView, DesignView, Valinfo, DevInfo, and ManageInfo. Each IP has BasicInfo, DesignInfo, ValiInfo, DevInfo, and ManageInfo. Each IP entity encompasses:

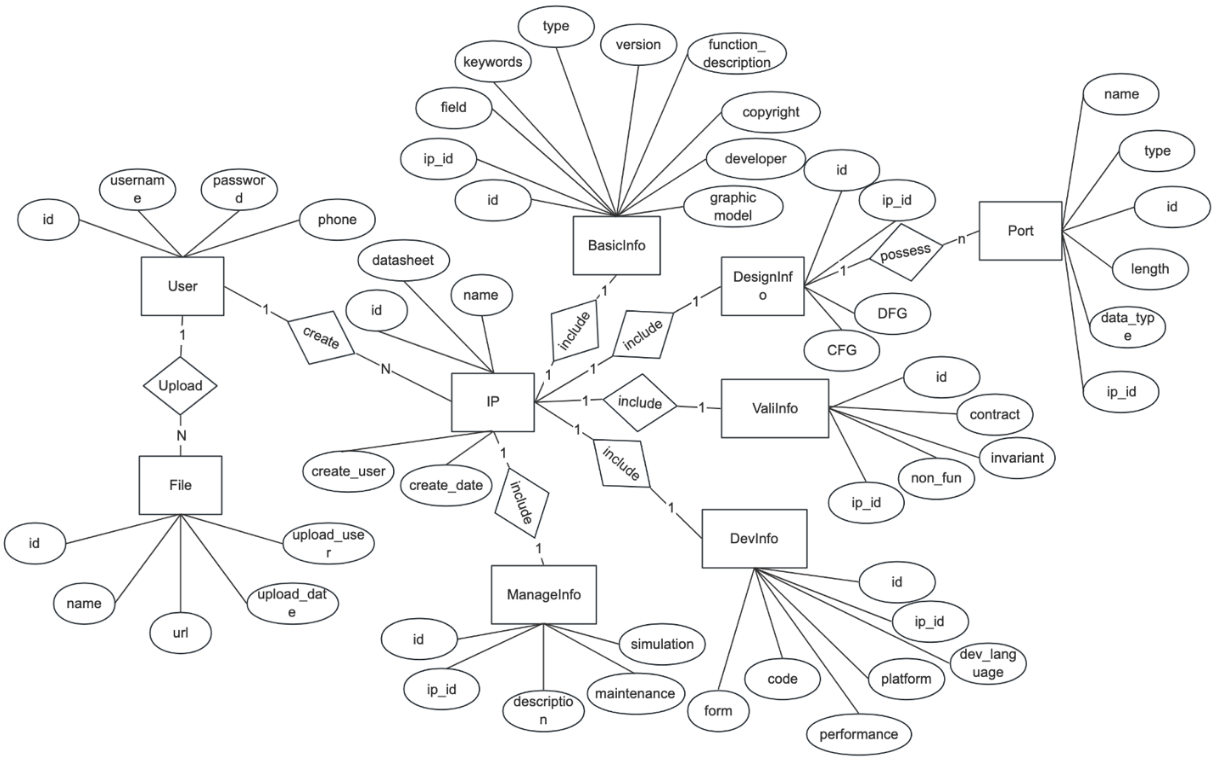


Fig. 4. Entity relationship diagram

1) BasicInfo: Attributes like id, ipid, and field.

2) DesignInfo: Includes DFG, CDG, and Port information, with each DesignInfo containing multiple Ports, each having attributes such as data_type, type, and length.

3) ValiInfo: Comprises Contract and Invariant details.

4) DevInfo: Covers development language, platform, and performance.

5) ManageInfo: Includes simulation and maintenance details.

User information consists of attributes like username, password, and each user, during IP creation or modification, can upload or delete N files with properties like file URLs, upload times, and more.

Neo4j graph database is used to store the structural information of software IP models, with the graph composed of:

$$G = \{(V, E, A)\}. \tag{5}$$

Where G represents a directed graph, V represents nodes, A expresses attributes of nodes, E represents relationships.

$$V = \{I, IP, O\}. \tag{6}$$

Nodes include input port nodes (I), software IP nodes (IP), and output port nodes (O).

$$E = \{(I_i, IP_j), (IP_k, O_j)\} \middle| I_i \in I, IP_j, IP_k \in IP, O_l \in O. \tag{7}$$

Relationships indicate connections from input to IP for input relationships and from IP to output for output relationships.

$$A = \{Attr_P, Attr_{IP}\}. \tag{8}$$

Attributes of nodes include software IP attributes (name, ID, and functional description) and input/output port attributes (names, types, meanings).

The distributed file system HDFS provides storage and retrieval for software IP-related files, including XML files, datasheets, source code, etc. Collaborative storage through three types of databases enables the system to leverage the strengths of each database when handling various types of software IP data, enhancing overall system performance.

The knowledge base also employs Redis as a caching database to store hot data and improve system performance, including recent user query data and user tokens. ElasticSearch is used to optimize indexing, supporting users in real-time full-text searches and reindexing descriptive text of software IPs, such as keywords, application scenarios, and fields. Kafka is employed reliable queue that holds messages until they are consumed. It ensures no data loss with features like replication and persistent storage.

The practice of collaborative storage across multiple databases greatly enhances the system’s querying capabilities, allowing for a more rapid response to user demands and significantly improving the data response speed. However, this collaborative storage presents challenges regarding data consistency. For instance, partial descriptive information of software IP is stored in both Neo4j and MySQL. When data in MySQL is updated, the corresponding changes must be synchronized with Neo4j.

To address this issue, we have implemented a multi-database collaborative model based on the Kafka message queue to ensure eventual consistency of data. The system adopts an eventual consistency principle, which allows for temporary data inconsistencies immediately after a change occurs but guarantees that data will eventually converge to a consistent state in the absence of further updates.

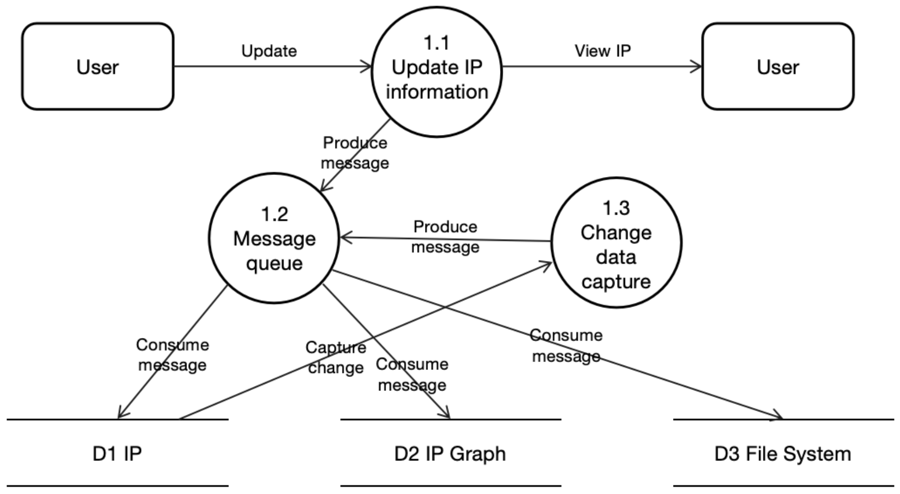


Fig. 5. Multi-database collaborative process

The multi-database collaborative process of IP update service is shown on Fig. 5. The content updated by user will be treated as a message entering the message queue and consumed by the IP table in MySQL. We utilize Change Data Capture (CDC) tools to monitor database change operations and synchronize these changes to Kafka's message queues. When a database change occurs, the CDC tool captures these changes and publishes messages to specific Kafka Topics. Each consumer subscribes to the relevant Kafka Topics receives the change information and applies the changes to its local data storage. During the synchronization process, if a conflict is detected, it is resolved using timestamps and version control. After each consumer has applied the data changes, the system performs data verification to ensure that the changes have been correctly applied and that data consistency is maintained. If an error occurs during the synchronization process, the system executes a failure retry mechanism. If the retries fail multiple times, a rollback operation is performed to ensure data consistency.

We model the data synchronization state of a multi-database system as a Markov chain [22]. The state set $S = \{s_0, s_1, ...s_n\}$ represents different synchronization levels, with s_0 as the initial state and s_n as the fully synchronized state. The transition probability from state s_i to state s_i is denoted as P_{ij}, forming the transition probability matrix P:

$$P = \begin{bmatrix} P_{00} & \cdots & P_{0n} \\ \vdots & \ddots & \vdots \\ P_{n0} & \cdots & P_{nn} \end{bmatrix}. \quad \textbf{(9)}$$

To demonstrate the system's long-term reliability, we find the steady-state probability distribution π, which satisfies:

$$\pi P = \pi. \quad \textbf{(10)}$$

$$\sum_{i=0}^{n} \pi_i = 1. \quad \textbf{(11)}$$

For a simplified example with three states (initial s_0, partial s_1, and fully synchronized s_2), we determine the transition probabilities as follows:

$$P = \begin{bmatrix} 0.6 & 0.3 & 0.1 \\ 0.2 & 0.5 & 0.3 \\ 0.0 & 0.05 & 0.95 \end{bmatrix}. \quad \textbf{(12)}$$

We solve for π from

$$\begin{aligned} \pi_0 &= 0.6\pi_0 + 0.2\pi_1 \\ \pi_1 &= 0.3\pi_0 + 0.5\pi_1 + 0.05\pi_2 \\ \pi_2 &= 0.1\pi_0 + 0.3\pi_1 + 0.95\pi_2 \\ \pi_0 &+ \pi_1 + \pi_2 = 1. \end{aligned} \tag{13}$$

Solving these, we might get: $\pi = (0.033, 0.017, 0.95)$. it indicates a long-term probability of approximately 95% for the system to reach full synchronization. Thus, the model demonstrates the long-term reliability of multi-database collaboration, showing a high probability of achieving full synchronization over time. This method can be extended to more complex systems by adjusting state definitions and transition probabilities accordingly.

Through this collaborative model, the system can effectively leverage the advantages of various databases and improve the system's performance and response speed through caching mechanisms. While maintaining high performance, it effectively manages data consistency, ensuring the reliability of the system and the accuracy of the data. This comprehensively supports the data needs of software IP.

5 Design and Implementation of Microservice

Based on the idea of microservice architecture, we divide the software IP knowledge base into six services, defined as follows:

$$Knowledge\ Base = Design, Manage, View, R\ \&\ R, User, File. \tag{14}$$

Where Design, Manage, and View represent the design, management, and display services of software IP, R&R represents the retrieval and recommendation service based on the knowledge graph, User represents the user management service of knowledge base, and File represents the file management service of the knowledge base. These application services support the knowledge base together. To construct a stable and scalable software IP knowledge base, we selected Spring Cloud as our microservices framework and MyBatisPlus for database connections. Given the high access volumes for the software IP Design Service, Management Service, View Service, and Recommendation and Retrieval Service, multiple instances of these services are deployed. In contrast, the User Management and File Management Services, which are less frequently accessed, have only one instance each.

In terms of microservices components, we have adopted Eureka for service registration and discovery. Eureka Server manages the registration information of each service, enabling automatic service discovery and registration. For configuration management, we use Spring Cloud Config to centrally manage configuration information. It provides configuration files to each microservice through the configuration server, allowing the system to modify configuration information without redeploying services, thereby enhancing system maintainability. The Log Center utilizes Logcenter as a solution for centralized collection, storage, and analysis of log information generated by applications. Logcenter can be used to manage logs from all microservices, facilitating troubleshooting, performance monitoring, and security auditing. For system monitoring, Spring Boot Admin is used as a monitoring tool, providing a web interface for monitoring the operational status and performance metrics of the system. To ensure the security of the system, we have adopted an API Gateway as the entry point for the microservices system. We use Kafka as the message queue between microservices, offering a highly reliable and high-throughput messaging mechanism that helps us achieve asynchronous communication and decoupling between microservices. Finally, we employ Docker for containerized deployment to achieve a lightweight and portable deployment solution. By packaging each microservice into an independent Docker container, we can quickly and reliably deploy the system, which greatly simplifies the system's operational and maintenance tasks.

5.1 Software IP Design Service

The Software IP design service is a crucial service of software IP. Users complete the design of software IP by filling in form content and uploading files on the frontend interface, shown in Fig. 6. The design form of the software IP case is divided into five parts: Datasheet, Knowledge Model, Formal Model, Implementation, and Product.

1) Datasheet: Consists of the user-submitted software IP's DOCX file, containing information such as the software IP name, interface description, and functional introduction.
2) Knowledge Model: Requires users to fill in basic knowledge, domain expert knowledge, and software development process knowledge.
3) Formal Model: Users upload a description file using formal methods for contracts, non-public constraints, and IP invariants.
4) Implementation: Users upload source code.
5) Product: Users upload documentation, testing, simulation, and maintenance information.

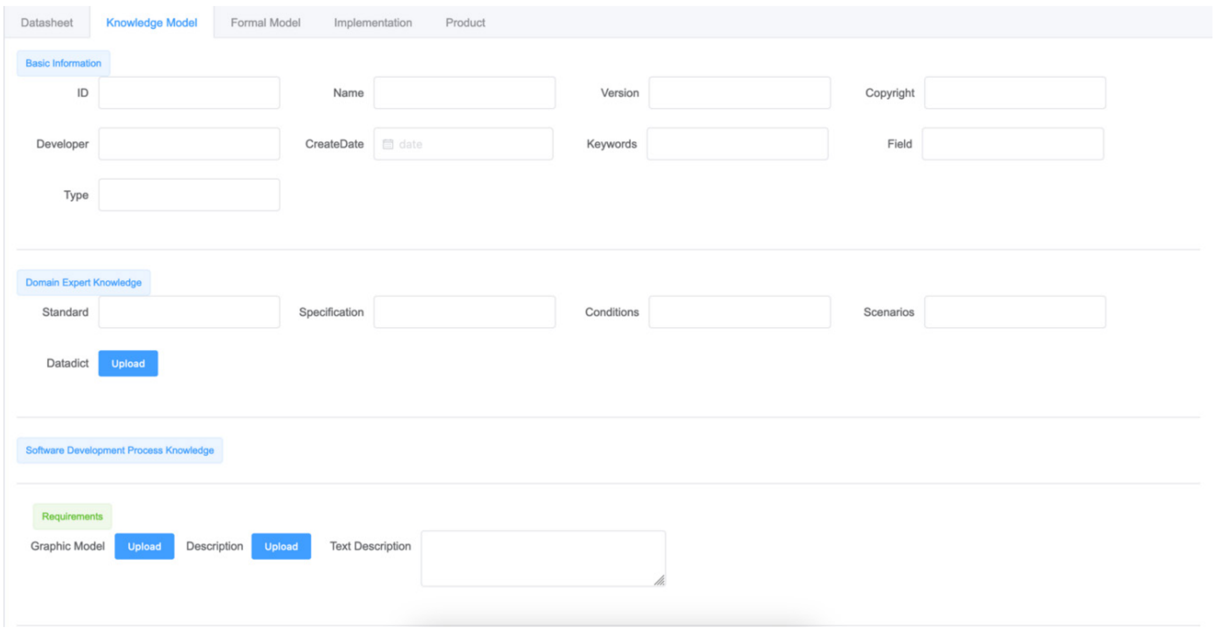


Fig. 6. Page of IP design

After filling out the design forms, the backend verifies the content of each form against the standard. Once verification passes, the backend saves all the information of the IP to the database and writes all information into an XML document, generating the corresponding directory structure. Through the semi-structured XML document, we can extract the text content or convert it to a certain degree of structured data, making it more flexible for downstream uses.

5.2 Software IP Management Service

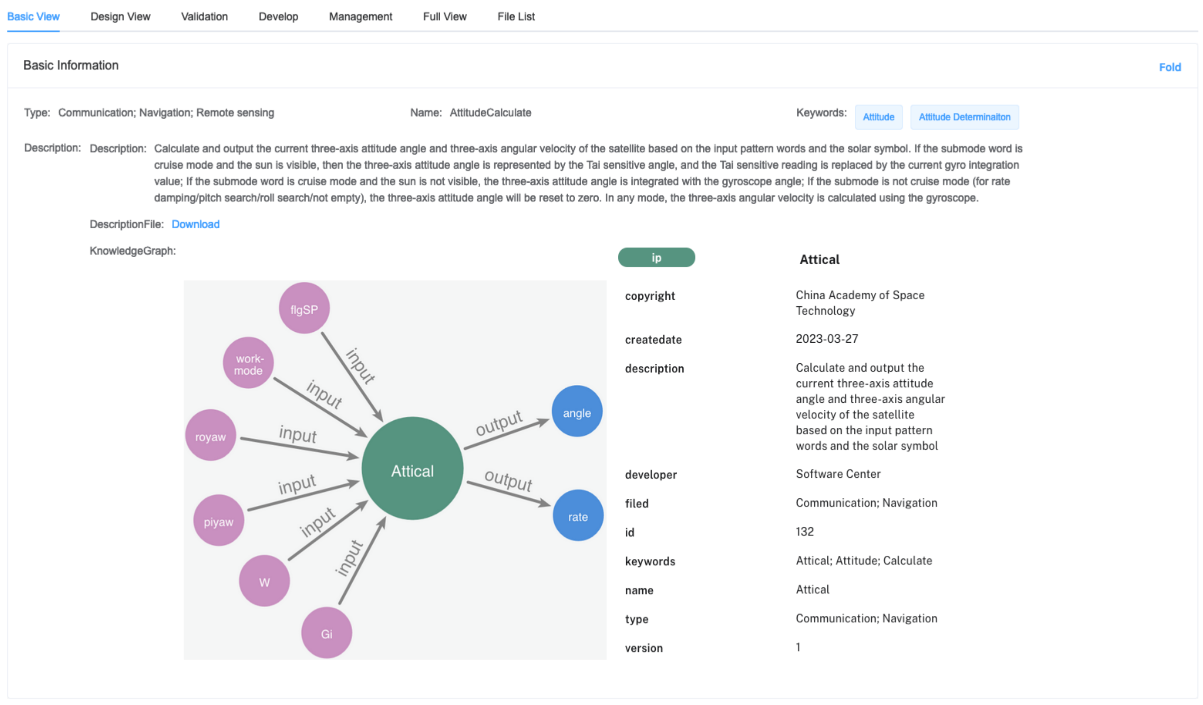
The software IP management service is the core service of the system, responsible for storing and managing the model information and descriptive information of software IP. The software IP knowledge base adopts the Neo4j graph database to store the model information of software IP and MySQL to store the descriptive information. In the software IP model, input ports, software IP, and output ports correspond to nodes in the graph. The properties of input ports and output ports, such as names, types, lengths, and meanings, correspond to the attributes of the nodes. The identification part of the knowledge model corresponds to the attributes of the software IP. The software IP management service implements functions such as creating, storing, modifying, and deleting software IPs. By combining graph databases and relational databases, efficient management of software IP models and descriptive content is achieved.

5.3 Software IP View Service

Software IP view service displays a complete software IP through various views to meet user viewing and analysis needs in different scenarios. The IP view is divided into six types of view. The Full View is editable, while the other views are read-only. The information contained in each view is as follows:

1) Basic View includes basic information about the software IP, such as type, name, and functional description.
2) Design View includes both basic and design information on the software IP. Design information comprises port details, parameter information, and flowcharts.
3) Verification View encompasses basic, design, and verification information of the software IP. Verification information includes contracts, non-functional constraints, and invariants.
4) Development View comprises basic, design, and development information of the software IP. Development information includes programming language, platform environment, performance characteristics, representation form, and source code.

(5) Management View covers management information of the software IP, including basic information, instructions, testing, simulation, and maintenance details.
(6) The Full View contains all information when creating an IP, with a structure like the IP design tool. In the Full View, users can modify IP information.



Basic Information

Type: Communication; Navigation; Remote sensing
Name: AttitudeCalculate
Keywords: Attitude; Attitude Determinaiton

Description: Description: Calculate and output the current three-axis attitude angle and three-axis angular velocity of the satellite based on the input pattern words and the solar symbol. If the submode word is cruise mode and the sun is visible, then the three-axis attitude angle is represented by the Tai sensitive angle, and the Tai sensitive reading is replaced by the current gyro integration value; If the submode word is cruise mode and the sun is not visible, the three-axis attitude angle is integrated with the gyroscope angle; If the submode is not cruise mode (for rate damping/pitch search/roll search/not empty), the three-axis attitude angle will be reset to zero. In any mode, the three-axis angular velocity is calculated using the gyroscope.

DescriptionFile: Download

KnowledgeGraph:

	Attical
copyright	China Academy of Space Technology
createdate	2023-03-27
description	Calculate and output the current three-axis attitude angle and three-axis angular velocity of the satellite based on the input pattern words and the solar symbol
developer	Software Center
filed	Communication; Navigation
id	132
keywords	Attical; Attitude; Calculate
name	Attical
type	Communication; Navigation
version	1

Fig. 7. IP basic view

Different views can be utilized in various downstream application scenarios. The availability of multiple views allows better adaptation to downstream tasks. The view service presents the structure and attributes of software IPs through a graphical interface, providing an intuitive user experience. In Fig. 7, using the example of the "Attitude Calculation" IP, its basic view is displayed. The basic presenting the fundamental elements that a software IP as a knowledge entity should encompass. Other views follow a similar format to the basic view, providing relevant information displays.

5.4 Retrieval and Recommendation Service

The retrieval and recommendation service provides graph-based search and recommendation services. Graph structures are inherently suitable for expressing and querying data with complex relationships. Using a graph database to store the model information of software IP provides better support for search and recommendation functionality, outperforming traditional keyword matching methods.

Search is achieved by describing query patterns using Cypher, specifying conditions for nodes and relationships, and their topological structures. Fig. 8 illustrates the sequence diagram for the search functionality. When the user inputs a search keyword and clicks the search button, the frontend sends a URL request. Upon receiving the request, the IPSearchController layer on the server invokes IPSearchService. Initially, the service assembles a query statement based on the user's search content. Subsequently, it checks whether the data is already present in the Redis cache by using the query statement as a key. If there is no cache hit, the service performs a query on Neo4j, writes the retrieved data into the Redis cache, and simultaneously returns the data to the frontend for display.

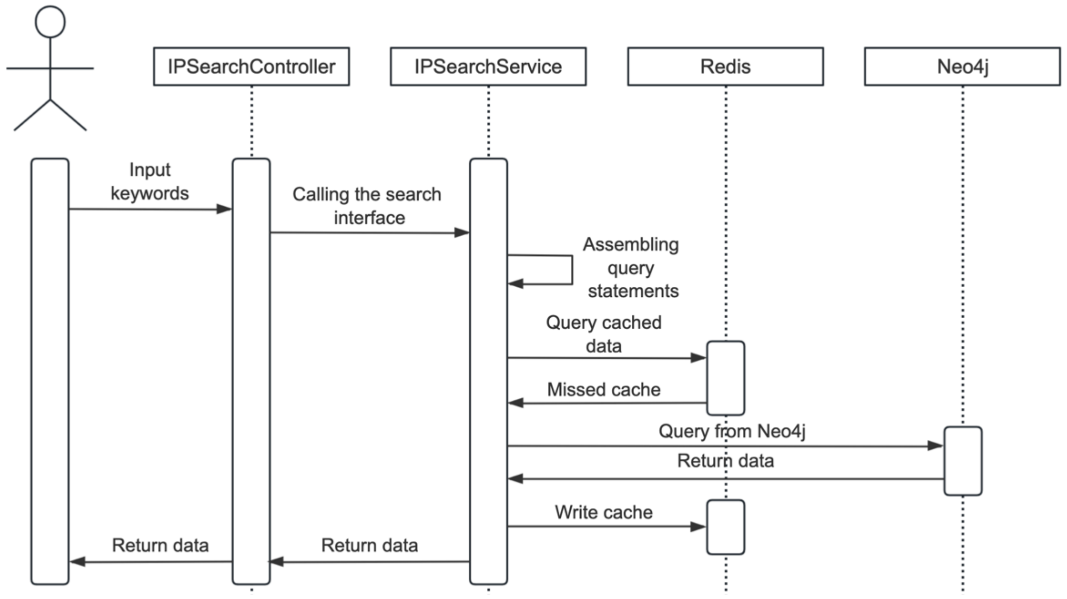


Fig. 8. Sequence diagram of search IP search service

The recommendation algorithm initially utilizes Cypher to query user history behavior and extract relevant software IP graph structure information. It then utilizes the BERT [23] model to extract semantic features from both the user's historical behavior and the information of software IPs. The BERT model transforms textual data into high-dimensional semantic representations, capturing the semantic information between user behaviors and software IPs. Subsequently, the system constructs a collaborative filtering model [24] based on user preferences and the similarity between software IPs. This model utilizes the user-item matrix and similarity information to predict the degree of user preference for uninteracted IPs by learning the common preferences among users and the associations between software IPs. By integrating the semantic features extracted by the BERT model and the user preferences and item associations learned by the collaborative filtering model, the system can provide users with more personalized and accurate recommendations.

5.5 User Management Service

The user management service is responsible for handling various aspects related to user accounts, authentication, and authorization within the software IP knowledge base. Users are categorized into administrators and regular users. Due to the privacy concerns associated with software IP, the new user registration is banned. Only administrators are permitted to add new users to the system, generate random passwords for them, and manage their roles and permissions. During the login process, user identities are verified, and upon successful validation, time-limited tokens are issued. Access policies are enforced based on user roles, managing user sessions, and permissions. Additionally, security protocols will be implemented to safeguard user information, and user activities will be monitored and logged to ensure security.

5.6 File Management Service

The file management service is implemented based on HDFS and is primarily responsible for the management and storage of file resources in the software IP knowledge base. The services provided include: uploading, downloading, storing, and retrieving files, viewing the historical version, etc. It allows users to upload new file resources to the knowledge base and supports the download of stored files, including XML files, datasheets, source code, and other software IP-related files. The service supports file version management, ensuring the tracking and maintenance of different versions of the same file. This helps users understand the evolution of files and revert to previous versions when necessary. The implementation includes periodic backup of files to prevent loss or damage and supports file recovery from backups when needed. The service records the operational history of files, including upload, download, modification, etc., for audit and monitoring of file usage.

6 Evaluation

6.1 Functional Testing

Functional testing is essential to ensure that each component of the software IP knowledge base performs its intended functions correctly. The primary focus is on verifying the correctness of the implemented features and the seamless interaction between different microservices. This system comprises six services with over fifty API and rich functionalities. During testing, we will focus only on the main API, and will not present the testing of other functionalities here. The following functional tests were conducted:

Table 1 outlines the functional test cases for the IP Design Service. It includes details on how each API was tested, the expected outcomes, and the actual results. The tested API including Add IP, IP Form Validation, Obtain IP Graph Structure. In the context of the Add IP API, the functionality was evaluated by populating the IP design form with all necessary information. This form submission was intended to create a new IP record within the system. The expected result was that the IP would be successfully generated and stored, which aligns with the observed outcome where the IP was indeed created as anticipated. The IP Form Validation API was tested by entering a range of inputs into the IP design form, encompassing both valid and invalid data. The test results confirmed that the API correctly validated the inputs and provided the expected validation messages, thereby ensuring data integrity.

Table 1. IP design service test cases

API	Implementation steps	Expected results	Actual results
Add IP	Filling the IP design form	Create IP successfully	Same as expected results
IP form validation	Fill in the IP design form with correct and incorrect inputs	Correctly output the validation results for each input	Same as expected results

Table 2 describes the functional test cases for the IP Management Service, detailing the API actions, expected results, and actual outcomes. The tested API including the CRUD of software IP. The IP Download API was tested by initiating the download of a ZIP file that encompasses the entire IP package. The test confirmed that the download process worked as intended. The Search IP API was evaluated by entering various keywords associated with IP names to retrieve relevant information. The test results showed that the API accurately performed the search and provided information on all items meeting the specified criteria, as expected. For the Delete IP API, the test involved passing the ID of a specific IP item to be removed. The expected functionality is that the item would undergo pseudo-deletion within the database, meaning that it would be marked as deleted but not physically removed. The results indicated that the pseudo-deletion process was executed successfully, aligning with the anticipated outcome. In testing the Modify IP API, the process involved submitting updated information for an existing IP item. The actual results demonstrated that the API effectively updated the IP information as required, meeting the expected results. The Recommend IP API was assessed by generating a recommendation list based on user preferences. The test confirmed that the recommendation system functioned correctly, producing a list of IPs that aligned with user specifications.

Table 2. IP management service test cases

API	Implementation steps	Expected results	Actual results
IP Download	Download the Zip file for a whole IP	Download successfully	Same as expected results
Search IP	Enter the keywords of the IP name	Search for information on all items that meet the criteria	Same as expected results
Delete IP	Pass the ID of the deleted item for item deletion	Pseudo-deletion of items in databases successful	Same as expected results
Modify IP	Pass in the modified item information for modification	Modified successfully	Same as expected results
Recommend IP	Generate IP recommendation list based on users	Return a list of IPs related to user	Same as expected results

Table 3 outlines the functional test cases for the View Service, including API actions, expected results, and actual outcomes. The Get IP View API was assessed by navigating to the Basic view page of the IP system. The test verified that accessing this view successfully returned the expected IP details, confirming that the API operates as intended. The Obtain Node of IP Graph Characteristics API was evaluated by invoking it to retrieve detailed characteristics related to specific nodes within the IP graph. This API should return a list of attributes associated with the selected node. The test results indicated that the API accurately provided the list of characteristics, meeting the expected outcome. For the Obtain IP Graph Structure API, the test involved invoking the interface to retrieve and present the graph structure of the current IP. The anticipated outcome was a clear and accurate graphical representation of the IP structure, which was successfully achieved according to the test results.

Table 3. IP view service test cases

API	Implementation steps	Expected results	Actual results
Get IP View	Open the Basic view page	Return IP View	Same as expected results
Obtain Node of IP Graph Characteristics	Call the interface to show node in detail	Return a list of characteristics related to a node	Same as expected results
Obtain IP Graph Structure	Call the interface to show IP graph in basic view	Obtain the graph structure of the current IP and visually display it	Same as expected results

The test results in Table 4 indicate that the back-end successfully retrieves the necessary parameters and files for computation based on the message content sent by the front-end. The system effectively writes the correct results into the database and storage while handling errors or messages requiring waiting in an appropriate manner. This aligns with the intended design goals and fulfills the system requirements.

Table 4. User service test cases

API	Implementation steps	Expected results	Actual results
Login	Enter the username and password	Successfully logged in and return a token	Same as expected results
Logout	Click the log out button	Pop up system	Same as expected results
Super administrator modifies user permissions	Select a new permission level for the user	Prompt for successful change, change user permissions	Same as expected results
Limited time for administrators to modify tokens	Enter the new limited time	Prompt for successful change, change token validity period	Same as expected results

The Login API was examined by providing valid credentials, including a username and password. The test results indicated that the API performed as expected by successfully logging the user in and returning the appropriate authentication token. For the Logout API, the test involved clicking the logout button to end the user session. The results confirmed that the API correctly displayed the logout prompt and successfully terminated the session, aligning with the anticipated outcome. The Super Administrator Modifies User Permissions API was tested by selecting a new permission level for a user. The test verified that the system accurately processed the permission change request and provided a prompt confirming the successful modification. The outcome was consistent with the expected functionality. Testing the Limited Time for Administrators to Modify Tokens API involved entering a new validity period for the tokens used by administrators. The results demonstrated that the API effectively adjusted the token validity period and returned a confirmation of the successful change, meeting the expected results.

6.2 Performance Test

To substantiate the efficacy of our software IP knowledge base architecture, we conducted a series of performance tests. Performance testing assesses the system's responsiveness, throughput, scalability, and reliability under various loads and conditions. Apache JMeter was used to simulate different user scenarios and measure the system's performance. The knowledge base is evaluated by Apache JMeter.

We conducted stress evaluations on two scenarios. The first scenario involved using the GET method to request a list of IPs, while the second scenario involved using the POST method to upload a newly designed IP. The tests were conducted using Apache JMeter, as depicted in Table 5, with the results summarized in Table 6.

Table 5. Apache JMeter parameters

Performance/Scenario	Get IP list (Get)	Design IP (Post)
Threads (users)	1000	200
Ramp-up period (seconds)	1	5
Loop count	2	5

Table 6. Evaluation results

Performance/Scenario	Get IP list (Get)	Design IP (Post)
Sample	2000	1000
Average	74	101
Median	20	122
Min	13	14
Max	257	246
Error %	0.00%	0.05%
Throughput	84.8	91.0/sec
Received KB/s	437.87	31.44
Sent KB/s	19.45	2419.81

For the Get IP List request, the system demonstrated excellent responsiveness, with an average response time of 74 ms and a median response time of 20 ms. The error rate was 0.00%, indicating that all requests were successfully processed. The throughput was 84.8 requests per second, highlighting the system's ability to handle high concurrent loads efficiently. Fig. 9 shows the delay value when requesting an IP list using the GET method. It lists delay samples for 2000 operations under different sequence operations. This can help evaluate the response time of the system under different operating loads, thereby understanding system performance. Fig. 10 shows the cumulative distribution function values of GET requests. It provides the relative frequency of requests under different delay values, and through PDF values, we can understand the probability of requests completing within a specific delay range. This distribution can help us understand the consistency and reliability of system performance.

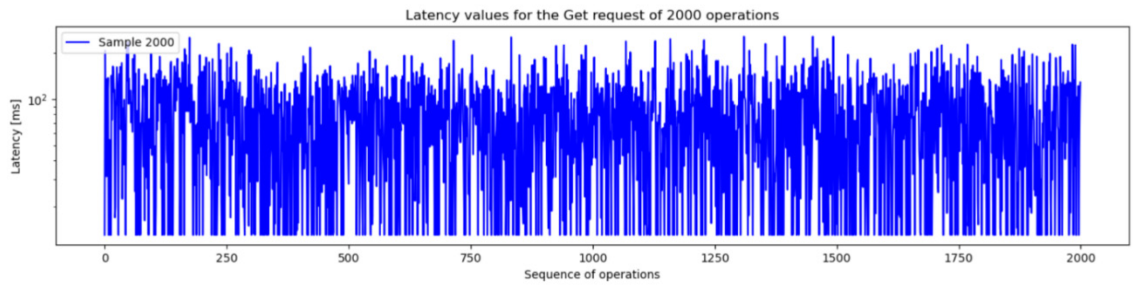


Fig. 9. Latency values for the Get request

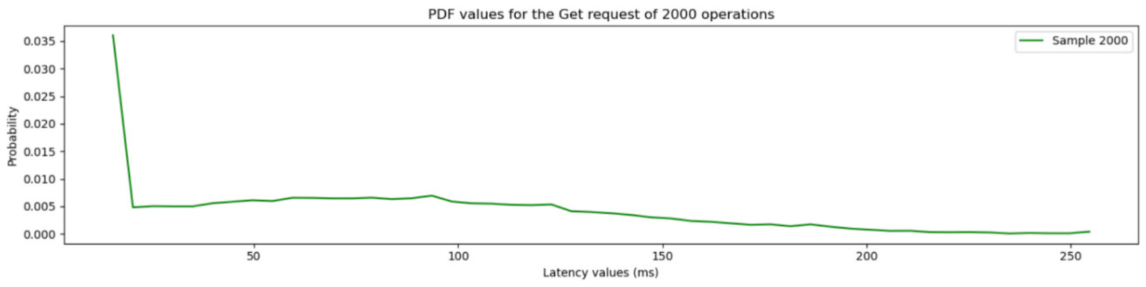


Fig. 10. PDF values for the Get request

For the Design IP request, the average response time for submitting new software IP designs was 507 ms, with a median response time of 59 ms. The error rate was 0.05%, indicating a very low failure rate in handling POST requests. The throughput was 91.0 requests per second, demonstrating the system's capability to manage frequent write operations effectively. Fig. 11 shows the latency values over a sequence of 1000 operations. The x-axis represents the sequence of operations, while the y-axis shows the latency in milliseconds (ms) on a logarithmic scale. The latency values exhibit significant variation at the beginning, with a notable peak, followed by a more stable region with occasional spikes. This indicates that the system experiences varying latency, with some operations taking significantly longer than others. Fig. 12 represents the Probability Density Function (PDF) of latency values for a sample of 1000 operations. The x-axis denotes the latency values in milliseconds (ms), while the y-axis represents the probability. The distribution shows that most latency values are concentrated around a certain range, with the highest probability density occurring at lower latency values. The probability decreases as the latency values increase, indicating that higher latency values are less frequent.

The data presented in Table 5 highlight the system's performance metrics, including response times and throughput, which are essential benchmarks for the reliability of the architecture. The successful completion of these stress tests with satisfactory results reinforces our confidence in the architecture's design.

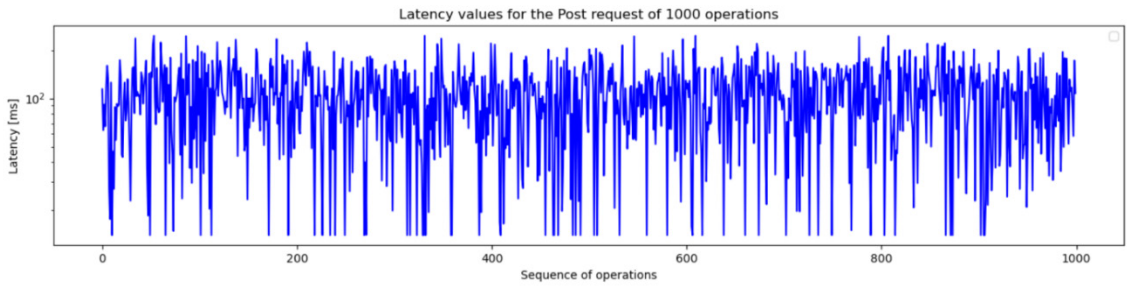


Fig. 11. Latency values for the Post request

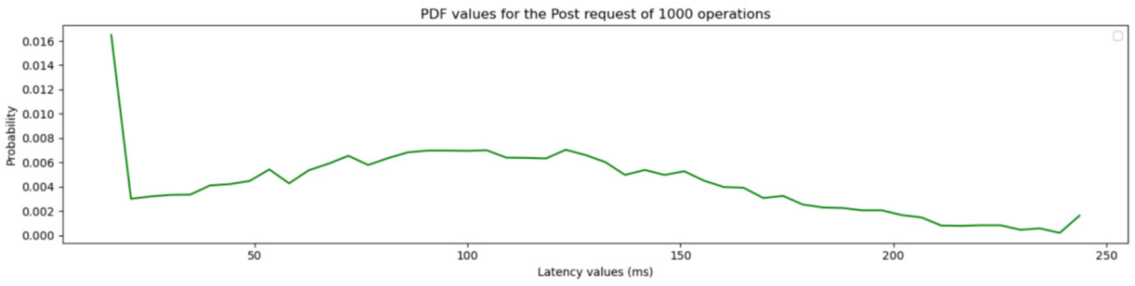


Fig. 12. PDF values for the Post request

7 Conclusion

This work presents the design and implementation of an embedded software IP knowledge base based on microservice architecture. Initially, we explored the concept and structure of software IP. Subsequently, we designed a knowledge base system suitable for storing software IP, considering its characteristics. By adopting a microservice architecture, the system decouples and independently deploys various functional modules, enhancing system flexibility and maintainability. By incorporating graph databases, relational databases, and distributed file systems, the system efficiently stores and retrieves both structured and unstructured data of software IP. Additionally, leveraging middleware such as Redis caching, ElasticSearch full-text search, and Kafka message queues further improves the system's response speed and throughput. Performance testing results indicate that the proposed knowledge base architecture offers significant advantages in terms of data consistency, retrieval efficiency, and system scalability. The proposed microservice-based embedded software IP knowledge base provides a solid platform for improving the efficiency of embedded software development and knowledge reuse. Future research will further optimize the retrieval and recommendation algorithms of the knowledge base and explore performance enhancements and functional expansions in more application scenarios.

8 Acknowledgement

This research was supported by the National Natural Science Foundation of China under Grant (No. 62192732).

References

[1] I. McLoughlin, Reverse engineering of embedded consumer electronic systems, in: Proc. 2011 IEEE 15th International Symposium on Consumer Electronics (ISCE), 2011.

[2] A. Malinowski, H. Yu, Comparison of embedded system design for industrial applications, IEEE transactions on industrial informatics 7(2)(2011) 244-254.
[3] J.C. Castellanos, F. Fruett, Embedded system to evaluate the passenger comfort in public transportation based on dynamical vehicle behavior with user's feedback, Measurement 47(2014) 442-451.
[4] J. Craveiro, J. Rufino, C. Almeida, R. Covelo, P. Venda, Embedded Linux in a partitioned architecture for aerospace applications, in: Proc. IEEE/ACS International Conference on Computer Systems and Applications, 2009.
[5] J. Thönes, Microservices, IEEE software 32(1)(2015) 113-116.
[6] N. Alshuqayran, N. Ali, R. Evans, A systematic mapping study in microservice architecture, in: Proc. 2016 IEEE 9th international conference on service-oriented computing and applications, 2016.
[7] T. Cerny, M.J. Donahoo, M. Trnka, Contextual understanding of microservice architecture: current and future directions, ACM SIGAPP Applied Computing Review 17(4)(2018) 29-45.
[8] G. Blinowski, A. Ojdowska, A. Przybyłek, Monolithic vs. Microservice Architecture: A Performance and Scalability Evaluation, IEEE Access 10(2022) 20357-20374.
[9] Z.-L. Wang, S.-Y. Zhu, J.-G. Li, W. Jiang, K.K. Ramakrishnan, Y.-F, Zheng, M. Yan, X.-H, Zhang, A.X. Liu, Deepscaling: microservices autoscaling for stable cpu utilization in large scale cloud systems, in: Proc. the 13th Symposium on Cloud Computing, 2022.
[10] J. Wang, R.-Q. Ouyang, W. Wen, X. Wan, W. Wang, A. Tolba, X.-G. Zhang, A post-evaluation system for smart grids based on microservice framework and big data analysis, Electronics 12(7)(2023) 1647-1665.
[11] X.-F. Yin, Y.-M. Zhu, J.-L. Hu, A subgrid-oriented privacy-preserving microservice framework based on deep neural network for false data injection attack detection in smart grids, IEEE Transactions on Industrial Informatics 18(3)(2021) 1957-1967.
[12] I. Asrowardi, S.D. Putra, E. Subyantoro, Designing microservice architectures for scalability and reliability in e-commerce, in: Proc. 2nd International Conference on Applied Science and Technology, 2020.
[13] M. Wu, X.-Q. Ding, R.-C. Hou, Design and implementation of B2B E-commerce platform based on microservices architecture, in: Proc. 2nd International Conference on Computer Science and Software Engineering, 2019.
[14] I. Atanasov, V. Vatakov, E. Pencheva. A Microservices-Based Approach to Designing an Intelligent Railway Control System Architecture 15(8)(2023) 1566-1588.
[15] S.-W. Yu, H.-L. Chang, H.-J, Wang, Design of cloud computing and microservice-based urban rail transit integrated supervisory control system plus, Urban Rail Transit 6(4)(2020) 187-204.
[16] M.-F. Yang, B. Gu, Z.-H. Duan, Z. Jin, N.-J. Zhan, Y.-W. Dong, C. Tian, G. Li, X.-G. Dong, X.-F. Li, Intelligent program synthesis framework and key scientific problems for embedded software, Chinese Space Science and Technology 42(4)(2022) 1-7.
[17] I. Crnkovic, S. Sentilles, A. Vulgarakis, M.R. Chaudron, A classification framework for software component models, IEEE Transactions on Software Engineering 37(5)(2010) 593-615.
[18] C.A.R. Hoare, Unified theories of programming, Mathematical methods in program development 158(1997) 313-367.
[19] D. Flemström, D. Sundmark, W. Afzal, Vertical test reuse for embedded systems: A systematic mapping study, in: Proc. 41st Euromicro Conference on Software Engineering and Advanced Applications, 2015.
[20] M.-S. Jin, C.-L. Qiu, J. Lim, The designment of student information management system based on B/S architecture, in: Proc. 2nd International Conference on Consumer Electronics, Communications and Networks (CECNet), 2012.
[21] W. Emmerich, Software engineering and middleware: a roadmap, in: Proc. the Conference on the Future of Software Engineering, 2000.
[22] X. Cheng, Z. Jin, H.-L. Yang, Optimal insurance strategies: A hybrid deep learning Markov chain approximation approach, ASTIN Bulletin 50(2)(2020) 449-477.
[23] J. Devlin, M.-W. Chang, K. Lee, K. Toutanova, BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding, in: Proc. 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, 2019.
[24] L.-X. Zou, L. Xia, Y.-L. Gu, X.-Y. Zhao, W.-D. Liu, J.-X. Huang, D.-W. Yin, Neural interactive collaborative filtering, in: Proc. the 43rd international ACM SIGIR conference on research and development in information retrieval, 2020.