# System Design: The Distributed Cache

### 

### Overview of Distributed Cache

* **Problem Statement**: It outlines a typical scenario where a system comprises clients making requests, hosts servicing these requests, and a database for data storage. The necessity for a cache arises as user requests increase, putting a strain on the database and slowing down service response times.

### Essentials of Distributed Caching

* **Basic Concepts**: A cache is described as a temporary storage that speeds up data retrieval by keeping frequently accessed data in memory, thus reducing database query load and improving performance.
* **Distributed Cache Definition**: A distributed cache involves multiple cache servers working together to store and manage data, ensuring scalability and higher availability than a single cache server could offer.

### Functionalities and Benefits

* **Use Cases**: The document highlights several benefits of using a distributed cache, such as minimizing latency for end-users, pre-calculating expensive database queries, temporarily storing user session data, and maintaining service availability during data store downtimes.
* **System Impact**: It further discusses how distributed caching reduces network costs by serving data from local resources and how it enhances user experience by decreasing perceived delays.

### Designing a Distributed Cache

* **System Layers**: Different caching mechanisms are needed at various system layers (web, application, database) to decouple sensitive data and reduce latency specific to each layer.
* **Technology Utilization**: Various technologies like HTTP cache headers, local caches, and key-value data stores are employed across different layers to optimize data retrieval and system responsiveness.

### Detailed Design Approach

* **Lessons on Design**: The course breaks down the design process into several lessons, starting with background knowledge, moving through high-level and detailed design, and finally evaluating the design against non-functional requirements like scalability and consistency.

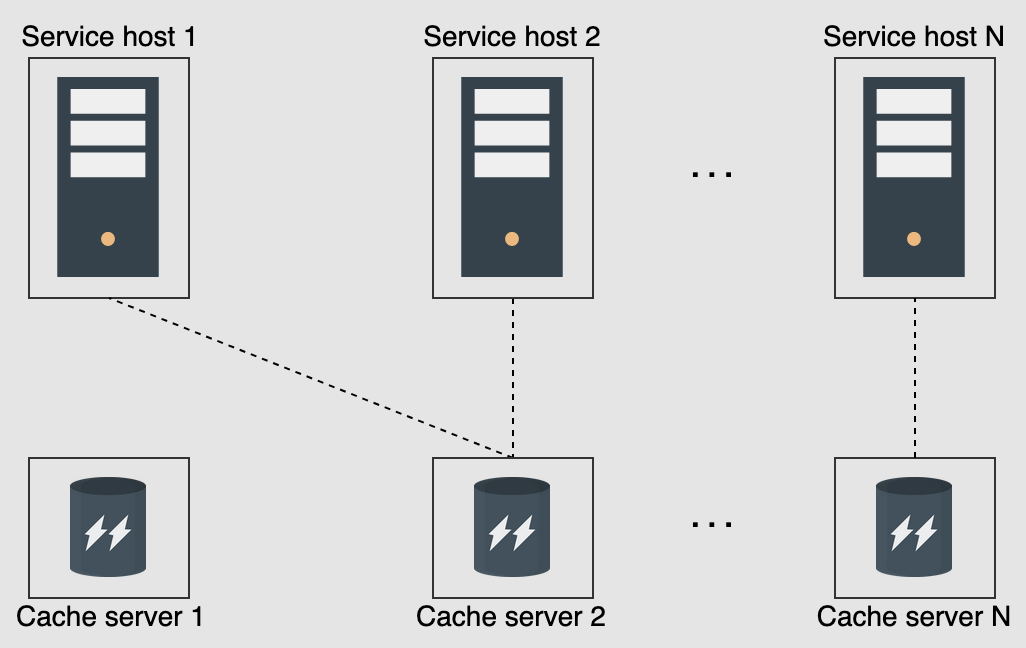
### Comparison and Practical Applications

* **Memcached vs. Redis**: A comparative analysis of these two popular caching solutions is provided, detailing their features, advantages, and potential use cases.

### Broader Applications

* **Designing Major Systems**: The document briefly mentions how distributed cache principles can be applied to the design of major platforms like Google Maps, Uber, and Twitter, suggesting that caching is a critical component in the scalability and efficiency of these large-scale systems.

# Background of Distributed Cache



### Key Concepts and Components

* **Writing Policies**: Discusses the strategies for writing data to cache and databases, such as write-through, write-back, and write-around policies, each with its implications for performance and consistency.
* **Eviction Policies**: Outlines methods for managing cache data due to limited storage capacity, including Least Recently Used (LRU), Most Recently Used (MRU), Least Frequently Used (LFU), and Most Frequently Used (MFU) strategies.
* **Cache Invalidation**: Addresses the removal of stale or outdated entries from the cache, with a focus on using metadata like time-to-live (TTL) values for active and passive expiration of data.
* **Storage Mechanism**: Examines considerations for distributed storage, including the allocation of cache entries across multiple servers and the selection of appropriate data structures for efficient storage and retrieval.

### Advanced Techniques and Structures

* **Hash Functions**: Explains the use of hash functions to identify appropriate cache servers for storing and retrieving data and to manage internal data structure within each cache server.
* **Sharding in Cache Clusters**: Introduces sharding to distribute data across multiple cache servers to prevent single points of failure and manage high load scenarios.
* **Dedicated Cache Servers**: Discusses the benefits of using dedicated servers for caching separate from application servers, such as scalability and specialized hardware utilization.
* **Co-located Cache**: Contrasts with dedicated servers by embedding cache functionality within the same hardware as service applications, highlighting trade-offs in terms of cost and risk of simultaneous service failures.

### Practical Implementations and Client Interaction

* **Cache Client**: Describes the client-side component that interacts with cache servers, performing computations to determine data storage and retrieval paths and handling communication using standard protocols like TCP or UDP.

### Conclusion and Integration

* **System Integration**: Emphasizes the importance of integrating caching strategies into the overall system design to enhance performance and reliability. The document suggests considering cache placement at various points within the system to optimize data access speeds and system responsiveness.

# High-level Design of a Distributed Cache

### Requirements

* **Functional Requirements**: The system must allow users to insert and retrieve data based on a unique key. This is fundamental to the operation of a distributed cache.
* **Non-functional Requirements**: Focuses on performance (speed of operations), scalability (handling more requests with no bottlenecks), high availability (resilience to failures), consistency (uniform data across all cache servers), and affordability (cost-effective hardware utilization).

### API Design

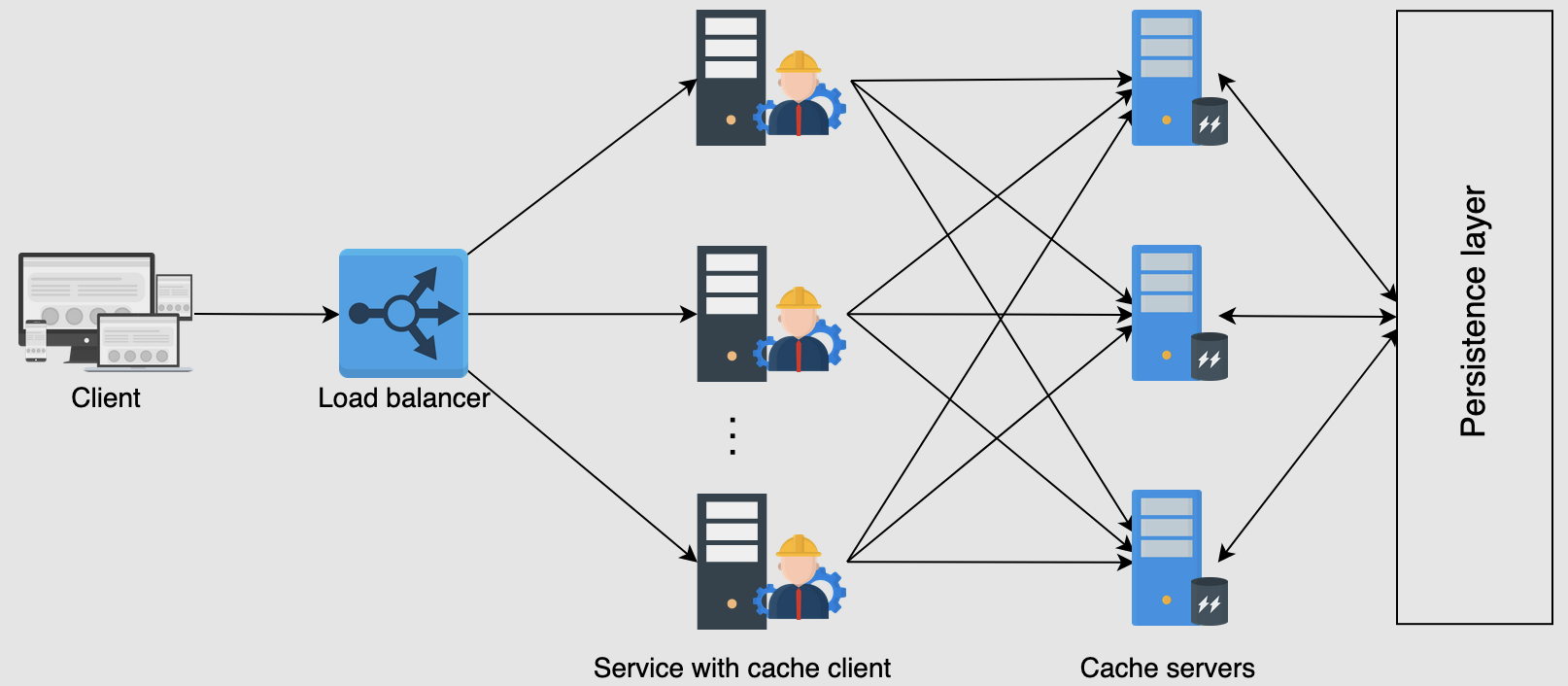
* **Insertion and Retrieval**: Simple APIs are proposed for inserting (insert(key, value)) and retrieving (retrieve(key)) data. These operations are designed to be intuitive and efficient, mirroring the simplicity of key-value stores.

### Design Considerations

* **Storage Hardware**: Discussion about whether to use specialized or commodity hardware, balancing cost against performance and capacity.
* **Data Structures**: The use of hash tables for fast access and linked lists to manage eviction processes is suggested. The possibility of storing various data types (strings, hash maps, etc.) is also considered.

### High-Level Design Components

* **Cache Client**: Describes the software component that interacts with cache servers to perform data insertions and retrievals. It ensures consistency and effective load distribution among cache servers.
* **Cache Servers**: These are responsible for storing the actual data and handling operations requested by cache clients. The servers must be robust to handle failures without disrupting the cache's availability.



### Writing and Eviction Policies

* **Writing Policy**: The design must decide whether to use write-through, write-back, or write-around caching, with each choice affecting the data consistency and operation latency.
* **Eviction Policy**: Policies like Least Recently Used (LRU) are considered to manage the limited space in the cache by determining which data to retain and which to discard.

### System Architecture

* The document outlines an architecture where cache clients and servers interact seamlessly, using hashing algorithms to distribute and retrieve data efficiently. The system is designed to be fault-tolerant, ensuring that cache availability is maintained even when individual components fail.

### Conclusion

The high-level design aims to provide a blueprint for implementing a distributed cache that meets the specified functional and non-functional requirements. The design focuses on practical and efficient data management strategies to optimize performance and cost.

# Detailed Design of a Monitoring System

### Problem Identification and Solutions

* **Identifying Limitations**: The document begins by addressing limitations in the previous high-level design, notably the single point of failure (SPOF) and the inability of cache clients to automatically recognize changes in the cache server landscape, such as server additions or failures.
* **Dynamic Server List Management**: Solutions to maintain an updated list of cache servers include:
  + **Configuration File**: Each service host could maintain a configuration file updated through DevOps tools, although manual updates are a drawback.
  + **Centralized Configuration**: Storing configuration files centrally to ease updates, still requiring manual changes.
  + **Configuration Service**: An automated service that monitors cache server health and updates cache clients dynamically, presenting the most robust solution.

### Improving Availability

* **Replica Nodes**: To combat cache server failures, the design proposes using primary and secondary (replica) nodes within cache shards, enabling high availability and consistency. This setup helps manage hot shards (data that is frequently accessed) by distributing the load across multiple nodes.

### Cache Server Internals

* **Data Structures and Mechanisms**:
  + **Hash Map**: Used within cache servers for quick data location.
  + **Doubly Linked List**: Supports efficient data eviction based on access frequency, implementing Least Recently Used (LRU) eviction policies.
* **Eviction Policy**: Tailored to application needs, with LRU highlighted for scenarios like social media platforms where recent interactions are prioritized.

### Detailed Design Proposition

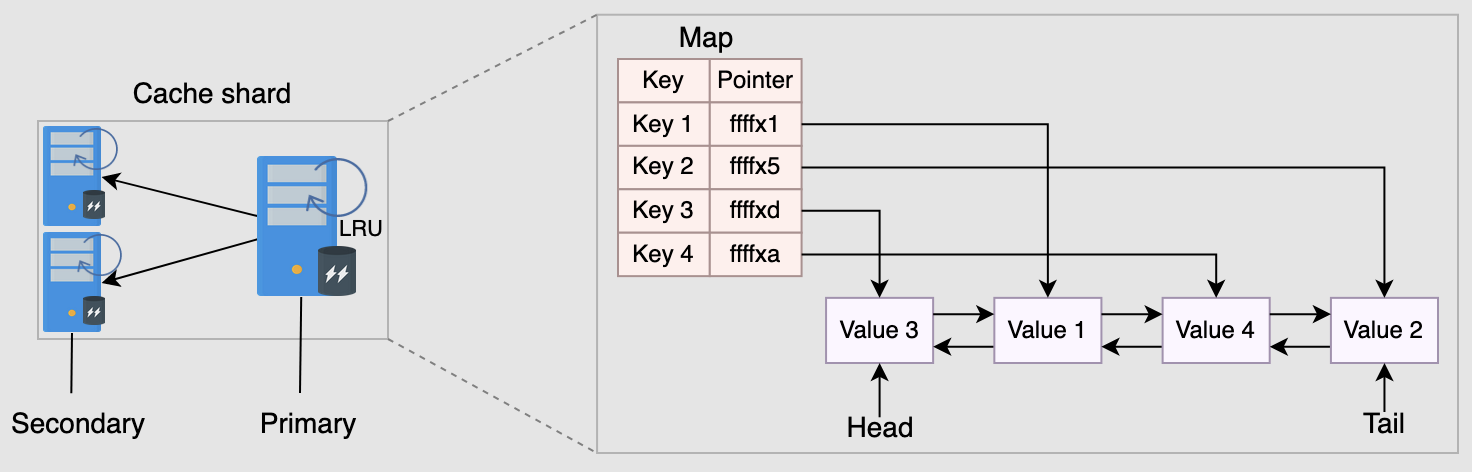
* **Consistent Hashing**: To distribute data evenly across cache servers, consistent hashing is employed, minimizing data overload on any single server.
* **Primary and Replica Servers**: Each shard has a primary server and one or more replicas, ensuring data redundancy and system resilience.
* **Configuration Service**: Ensures all cache clients have an up-to-date and consistent view of the server landscape, improving system reliability and response to server availability changes.
* **Monitoring Services**: These are suggested for ongoing assessment of cache performance and health, aiding in proactive system management.

### Design Reflections and Enhancements

* **System Scalability and Reliability**: The design focuses on scalability by efficiently managing data distribution and server load balancing. Reliability is addressed through redundancy and real-time configuration updates.
* **Operational Complexity**: While the proposed solutions enhance performance and reliability, they also increase the system's operational complexity. This complexity is justified by the significant improvements in system robustness and user response times.

### Conclusion

The detailed design elaborates on the necessary infrastructure and strategies to effectively implement a distributed cache that can handle large-scale deployments. It addresses critical architectural considerations like server failure, system updates, and load balancing, ensuring the cache system is both robust and efficient.



# Evaluation of a Distributed Cache's Design

### High Performance

* **Consistent Hashing**: Utilized to ensure that finding a key has a time complexity of O(log(N)), enhancing the efficiency of data retrieval across cache shards.
* **Hash Tables**: Employed within cache servers for constant-time data location, crucial for quick data access.
* **LRU Eviction Policy**: Implemented using a doubly linked list, allowing constant-time updates and access, optimizing cache entry management.
* **Communication Protocols**: TCP and UDP are used for fast data transmission between cache clients and servers.
* **Replicas**: Addition of replicas helps distribute the request load, reducing performance penalties on single machines and improving response times.
* **RAM Usage**: Data operations (addition, retrieval, serving) are performed directly from RAM, significantly reducing latency.

### Scalability

* **Sharding**: The system supports dynamic sharding based on changing server loads and requirements, allowing flexibility in managing data distribution.
* **Consistent Hashing and Rehashing**: Minimizes the need for extensive rehashing when new servers are added, facilitating easier scaling.
* **Replication and Hotkeys**: Strategies such as further sharding within hotkeys and dynamic replication are suggested to manage load on frequently accessed keys.

### High Availability

* **Redundant Servers**: The design includes multiple replicas for each shard to ensure availability even if one server fails.
* **Leader-Follower Architecture**: Used to manage shard replicas efficiently, though the evaluation notes that placing replicas in the same data center limits the system’s ability to handle complete data center failures.

### Consistency

* **Write Strategies**: The system can perform writes in synchronous or asynchronous modes, with a preference for asynchronous to enhance performance.
* **Inconsistencies**: Potential inconsistencies due to asynchronous replication are acknowledged, with measures suggested to minimize impact, such as delaying service from recovered servers until they are fully updated.

### Affordability

* **Cost-Effectiveness**: The system is designed to be affordable, utilizing commodity hardware which makes it economically viable for large scale deployments.

### Summary

* The evaluation confirms that the distributed cache design achieves a high level of performance, scalability, and availability, with trade-offs in consistency for performance gains. It also highlights the system's cost-effectiveness and its capability to handle large-scale operations efficiently.

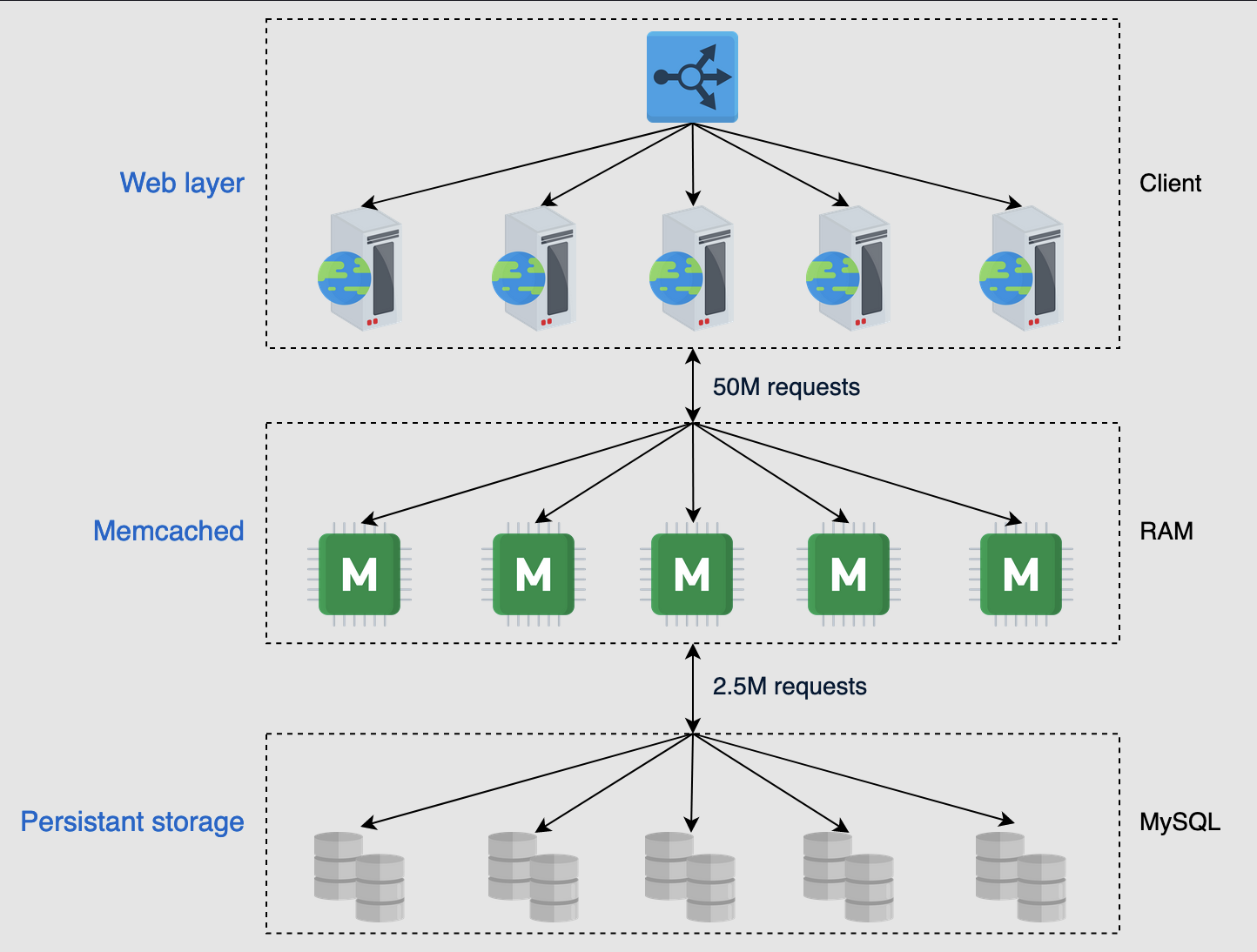
# Memcached versus Redis

### Introduction

* **Context**: Both Memcached and Redis are used for implementing distributed caches that are scalable, performant, and robust. These systems achieve sub-millisecond latency and follow the client-server model.

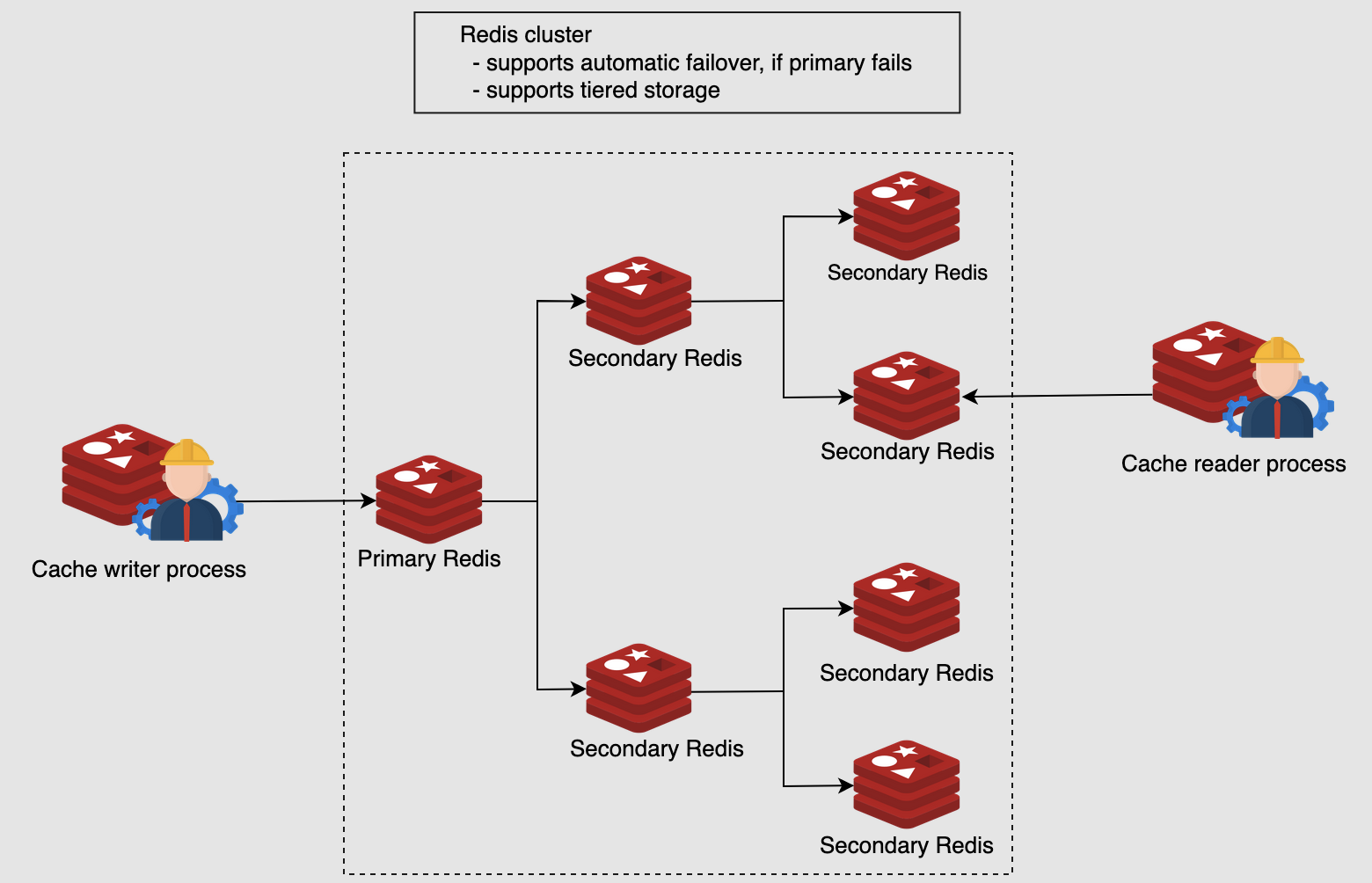
### Memcached

* **History and Usage**: Introduced in 2003, Memcached is a high-speed key-value store used primarily for caching purposes. It stores data as key-value pairs where both keys and values are strings.
* **Architecture**: Memcached employs a shared-nothing architecture, meaning servers operate independently without synchronization or data sharing, which enhances its performance and scalability.
* **Performance**: Known for high throughput and low latency, Memcached can handle millions of keys per second on high-end systems.
* **Example Use Case**: Facebook uses Memcached extensively for its caching layer, placed between its MySQL database and web servers, leveraging roughly 28 Terabytes of RAM across more than 800 servers.



### Redis

* **Features**: Unlike Memcached, Redis supports various data structures such as strings, lists, sets, sorted sets, and hashes. It can act as a cache, database, and message broker.
* **Persistence and Replication**: Offers built-in replication mechanisms, automatic failover, and different levels of data persistence.
* **Complexity and Utility**: Redis provides a more feature-rich environment but at the cost of additional complexity compared to Memcached.
* **Cluster Support**: Redis supports automatic sharding with primary and secondary nodes within its cluster management, enhancing availability and scalability.



### Key Differences

* **Data Handling**: Memcached is simpler, focusing on string-based key-value pairs, while Redis offers support for complex data types and operations directly on these structures.
* **Persistence**: Redis supports data persistence directly through snapshots and append-only files, whereas Memcached does not natively support persistence.
* **Use Cases**: Redis is suited for more complex scenarios that require detailed data manipulation and persistence, while Memcached is optimal for straightforward caching with high-speed requirements.

### Pipelining in Redis

* **Latency Optimization**: Redis uses pipelining to reduce latency and increase throughput. This technique allows multiple commands to be sent together without waiting for individual responses, minimizing round-trip times.

### Comparative Analysis

* **Simplicity vs. Feature Richness**: Memcached offers simplicity and is easier to manage for caching purposes. Redis, while more complex, provides more extensive features and better handles diverse data types and durability requirements.
* **Performance**: Both solutions offer low latency and high throughput, but Redis has the added advantage of built-in data manipulation features.
* **Scalability**: Memcached scales well horizontally with a simple model, whereas Redis provides more sophisticated clustering options.

### Conclusion

* **Application Suitability**: The choice between Memcached and Redis depends on the specific requirements of the application, such as the need for complex data types, persistence, and the scale of deployment. Redis tends to be more versatile, supporting a wider range of use cases, while Memcached excels in scenarios that require raw speed and simplicity.