

Redis A Visual Guide

Understand Data Structures Caching and Reliable Operations from the Ground Up

Reiner Braun

Sample Book — Chapters 1 to 2

Introduction

This book is designed for readers who are taking their first steps with Redis, including systems engineers who want to build the right foundation from the start.

Rather than asking you to memorize more than you need, the book focuses on the essential concepts that will support your future study and day-to-day work. More advanced topics are included only when they are likely to become useful to many readers, and even then they are introduced only far enough to explain their purpose, context, and overall role.

Processes, relationships, and structures that can be difficult to picture from text alone are supported by diagrams. By moving between explanations, visuals, and examples, you can build a clearer and more practical understanding of the subject.

Product names, service names, logos, icons, and other identifying marks mentioned in this book are trademarks or registered trademarks of their respective owners. This book is not an official publication, certified training resource, or endorsed learning material of any company or organization.

The Big Picture

Redis is a networked, in-memory data structure server. Its main advantage is not simply that memory is fast. Redis combines low-latency access with server-side operations on useful structures such as strings, hashes, lists, sets, sorted sets, and streams. A sound Redis design therefore starts with four questions: **what data is stored, how it is accessed, how long it should live, and what may happen when the Redis service is unavailable or loses data.**

The guide first builds the mental model needed to answer those questions. Chapters 1 through 8 explain where Redis fits in an architecture, how to work with an isolated Redis instance, how keys and expiration behave, how the main data structures differ, and how command cost follows from the shape of the data.

Chapters 9 through 17 extend that foundation into application and operational practice. They explain connection management, retries, caching, concurrency, memory control, persistence, high availability, horizontal scaling, security, monitoring, and incident response. **A reliable Redis system is defined as much by its client behavior and operating policy as by its data model.**

The recurring principle is simple: **Redis should fail in a way the surrounding system has already planned for.** Engineers should be able to state what happens after an ambiguous timeout, which data may become stale or be lost, how much memory and recovery time are acceptable, who may reach Redis, and which signals reveal a developing problem.

The command examples use `redis-cli`, the standard Redis command-line client. Unless a block says otherwise, commands are entered after opening an interactive `redis-cli` session; the leading `>` represents the prompt and is not part of the command. A few application examples use Python and the `redis` client library because client-side behavior cannot be demonstrated fully from the CLI. Those snippets are **one concrete environment example**; production applications should use a maintained client for their language and follow that client's connection, retry, and topology guidance.

How to Use This Book

- 1. Read the chapters in order.** The sequence builds the required background step by step, beginning with the Redis operating model and data structures before moving into application correctness and production operations.
 - 2. Review “What You’ll Be Able to Do” first.** At the beginning of each chapter, use this section to identify the practical capabilities and concepts you should gain from the chapter.
 - 3. Use the diagrams to see relationships and processes.** The visuals clarify boundaries, flows, state changes, responsibilities, and structures that are harder to reconstruct from prose alone.
 - 4. Run the examples whenever possible.** Use an isolated learning environment and type the commands or code yourself so that replies, failures, and state changes become concrete.
 - 5. Confirm the fundamentals at the end of each chapter.** Read the chapter summary and answer the review questions without looking back before checking any uncertain point in the text.
 - 6. Treat advanced-topic overviews as signposts.** You do not need to memorize them now. Remember what problem each topic solves so that you can recognize when deeper study may be needed later.
-

Table of Contents

[Introduction](#)

[How to Use This Book](#)

[Part I — Build the Mental Model](#)

[Chapter 1 — Redis in the System Architecture](#)

- [1.1 Redis as a Data Structure Server](#)
- [1.2 Why Redis Is Fast](#)
- [1.3 A Typical Request Path](#)
- [1.4 Common Roles in Real Projects](#)
- [1.5 Cache or System of Record](#)
- [1.6 Where Redis Is a Poor Fit](#)
- [1.7 Deployment Shapes at a Glance](#)
- [1.8 Four Questions That Follow Every Redis Design](#)
- [1.9 Correcting the Most Important Misconceptions](#)

[Chapter 2 — A Safe First Redis Session](#)

- [2.1 The Pieces of a Redis Session](#)
- [2.2 Start an Isolated Docker Lab](#)
- [2.3 Verify Connectivity](#)
- [2.4 A Minimal Command Sequence](#)
- [2.5 Reading CLI Replies](#)
- [2.6 Observe the Server, Not Just the Data](#)
- [2.7 A First Restart Experiment](#)
- [2.8 Graceful Shutdown in the Lab](#)
- [2.9 Security Boundaries for a Learning Environment](#)
- [2.10 The Practical Lesson](#)

Part I — Build the Mental Model

Chapter 1 — Redis in the System Architecture

What You'll Be Able to Do

- Explain Redis as a remote data structure server rather than an in-process dictionary.
- Distinguish latency, throughput, and durability when evaluating a Redis design.
- Identify common Redis roles, deployment shapes, and the failure questions that accompany them.

The most useful mental model is that Redis is a remote service that owns named data structures and performs operations on them. It is not merely a dictionary inside the application process, and it is not automatically a replacement for a relational database.

1.1 Redis as a Data Structure Server

A **data structure** is an organized form of data designed for particular operations, such as a list for ordered items or a set for unique membership. In Redis, each data structure is stored under a **key**, which is a byte sequence used as its name.

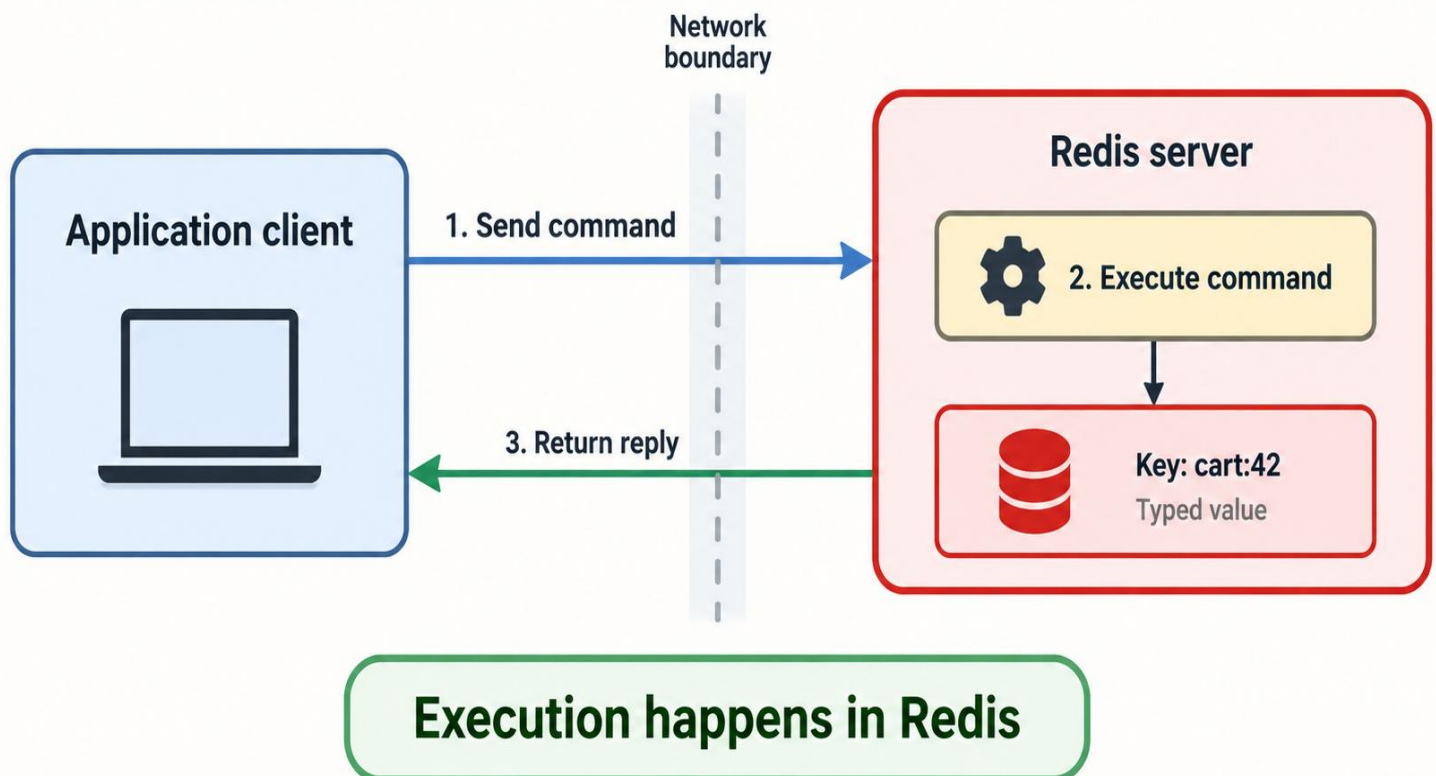
An application normally interacts with Redis in four steps:

1. The application opens or reuses a network connection to a Redis endpoint.
2. It sends a command such as `GET product:1842` or `SADD project:7:members user:42`.
3. Redis finds the key and executes the requested operation on the value.
4. Redis returns a reply to the client.

The important distinction is that **the operation runs inside Redis**, not in the application. For example, `INCR page:home:views` asks Redis to increment a counter as one command. The application does not need to read the old value, calculate a new value, and send the result back.

Chapter 1: Redis in the System Architecture

1-1 Command Execution Boundary



This is similar to asking a bank teller to transfer money rather than taking the ledger away, editing it yourself, and returning it. Keeping the operation at the service that owns the data reduces network exchanges and provides a clear execution boundary.

1.2 Why Redis Is Fast

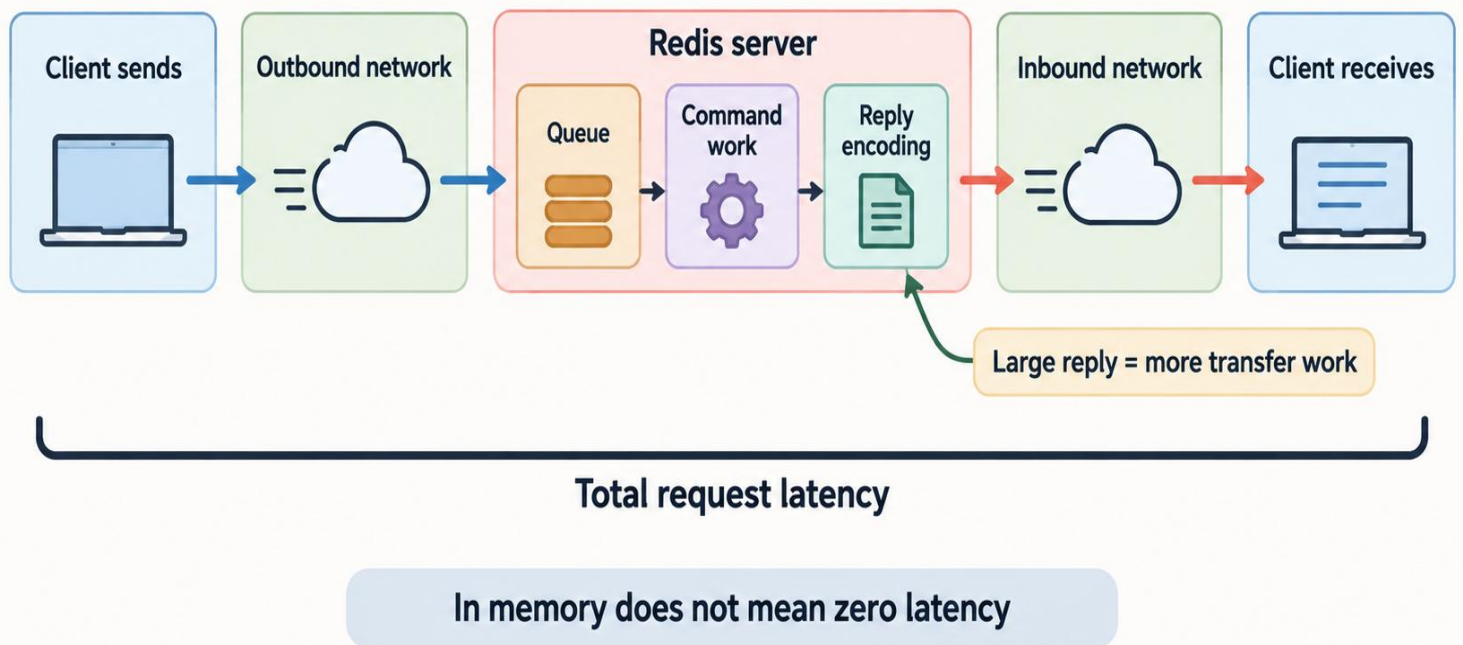
Redis is designed for low-latency access. **Latency** is the elapsed time between starting an operation and receiving its result. Several design choices contribute to Redis performance:

- The active dataset is normally served from memory, avoiding a storage read for each command.
- Redis implements common operations directly, such as incrementing a counter or adding a member to a set.
- Its request path is compact compared with a general-purpose query engine.
- Most command logic on one Redis server follows a serialized execution path, which avoids many locking and coordination costs; Redis can still use other threads for network I/O and background work.
- Redis uses efficient internal encodings and background work where appropriate.

Memory access alone does not guarantee low application latency. A request still crosses the network, waits behind other work, executes a command, and returns a response. A large response, an expensive command, an overloaded host, or many round trips can make an in-memory service slow.

Chapter 1: Redis in the System Architecture

1-2 Where Request Latency Comes From



002

Figure 1-2: End-to-End Sources of Redis Request Latency

Latency, Throughput, and Durability

These three properties answer different questions.

Property	Question it answers	Example
Latency	How long does one operation take?	A cache lookup completes in 1 millisecond.
Throughput	How many operations can be completed in a period?	A service completes 80,000 commands per second.
Durability	How much acknowledged data survives a failure?	A restart loses at most a stated interval of recent writes.

Throughput is the amount of work completed per unit of time. Low latency often helps throughput, but the two are not identical. One request may be fast while the service still has insufficient capacity for a large number of concurrent requests.

Durability is the ability to retain accepted data across process, host, or storage failures. Redis can persist data, but durability depends on the selected persistence and replication design. A fast acknowledgement is not proof that the write has reached durable storage or another node.

1.3 A Typical Request Path

Consider a product API backed by a relational database. The database is the authoritative source of product names, prices, and inventory. Product descriptions are read far more often than they are changed, so the application keeps recently requested descriptions in Redis.

A simplified read proceeds as follows:

1. The API receives a request for product `1842`.
2. It asks Redis for `catalog:product:1842`.
3. If Redis returns a value, the API uses it and avoids a database query.
4. If Redis reports that the key does not exist, the API reads the database.
5. The API stores the result in Redis for a limited time and returns it to the caller.

Redis acts as a **cache**, which is a faster temporary copy of data whose authoritative version exists elsewhere. The database remains the **source of truth**, meaning the system whose data is treated as authoritative when copies disagree.

Chapter 1: Redis in the System Architecture

1-3 Product Cache Hit and Miss

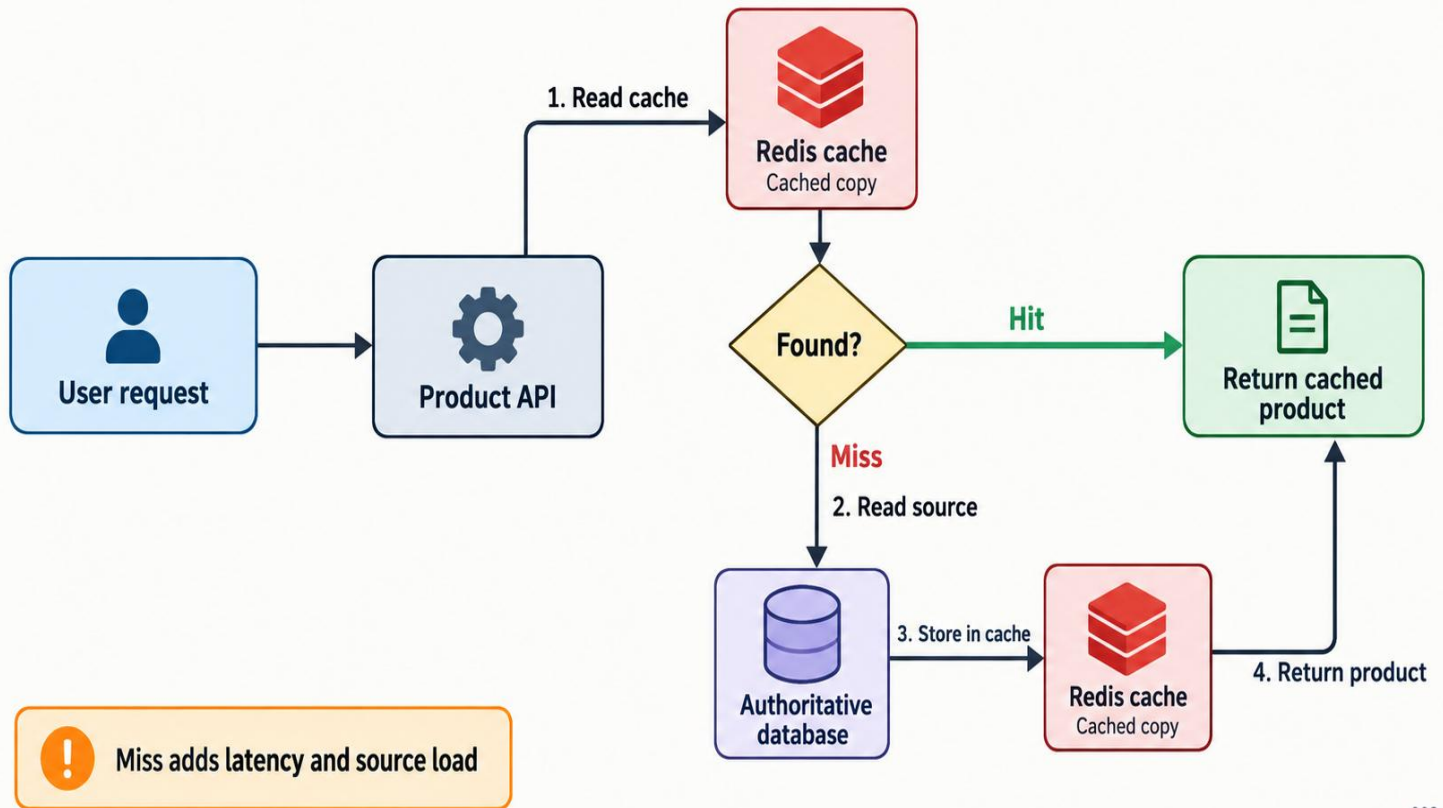


Figure 1-3: Product API Cache Hit and Miss Paths

The same Redis instance could also hold a page-view counter or a login session. Those roles have different failure requirements. Losing a cached product description may only cause an extra database query. Losing active sessions may sign users out. Losing an order event may create a business incident. **The Redis technology can be the same while the acceptable risk is completely different.**

1.4 Common Roles in Real Projects

Redis is frequently used for the following roles:

- **Cache:** a temporary copy that reduces repeated work or source-system load.
- **Session store:** server-side state associated with a logged-in user or client.
- **Counter:** an integer updated by commands such as `INCR` or `HINCRBY`.
- **Rate limiter:** a mechanism that counts or records requests to enforce a policy over time.
- **Leaderboard:** a ranking represented naturally by a sorted set.
- **Short-lived state store:** temporary tokens, verification state, idempotency records, or workflow markers.
- **Queue:** ordered work items consumed by workers, often using lists for simple cases.
- **Event stream:** retained ordered entries processed by one or more consumers.
- **Transient notification channel:** live messages sent through Pub/Sub to currently connected subscribers.

These are patterns, not automatic features. For example, storing a counter is straightforward, but a correct rate limiter also needs a defined time window, expiration, concurrency behavior, and failure policy.

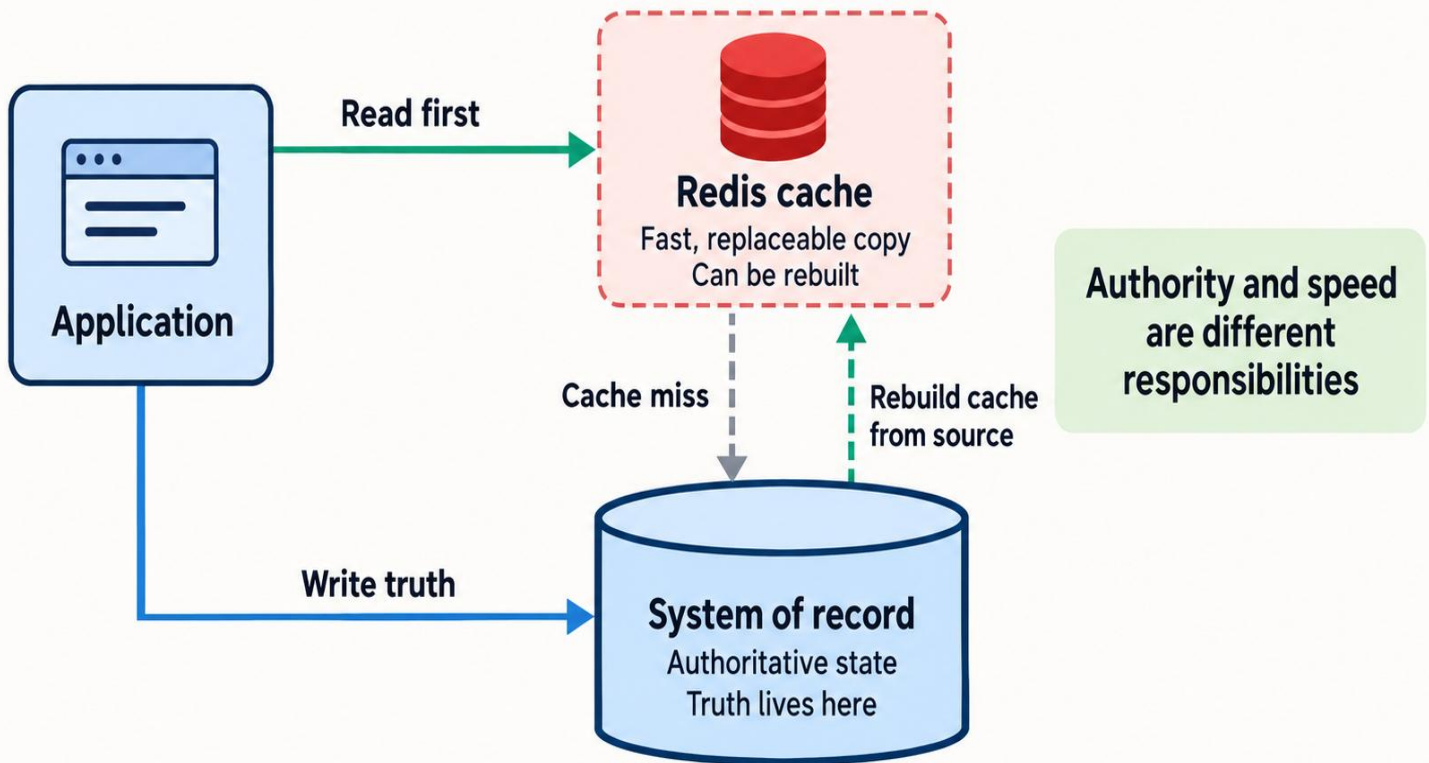
1.5 Cache or System of Record

A **system of record** is a system that is responsible for the authoritative version of data. Redis can be used as a system of record in some designs, but doing so changes the engineering requirements.

When Redis is only a cache, the application can usually reconstruct lost data from another source. The main concerns are performance, freshness, and source-system load during cache misses.

Chapter 1: Redis in the System Architecture

1-5 Cache or System of Record



004

Figure 1-4: Cache Copy versus System of Record

When Redis holds the only accepted copy of data, the design must answer harder questions:

- What amount of recent data loss is acceptable?
- How will the service recover after a restart or host loss?
- How will backups be created and restored?
- What consistency is expected during failover?
- How will operators detect persistence or replication failure?

A useful rule is: **the harder the data is to reconstruct, the more Redis must be treated like a primary database rather than an optional accelerator.**

1.6 Where Redis Is a Poor Fit

Redis is not the best tool for every workload. It is usually a poor fit when the main requirement is one of the following:

- Complex relational joins and constraints across many entities.
- Broad ad hoc queries over fields that were not planned as access paths.
- Very large objects or datasets without a realistic memory and tiering strategy.
- Long-term analytical scans over most of the dataset.

- Workflows that depend on broad, multi-record ACID transactions.
- Message retention, routing, replay, or cross-region guarantees beyond what the Redis deployment can safely provide.

ACID is a group of transaction properties—atomicity, consistency, isolation, and durability—commonly associated with relational databases. Redis offers atomic commands and other coordination tools, but it does not behave like a relational transaction engine across arbitrary records and queries.

Choosing Redis because it benchmarks faster than a database is not enough. The correct question is whether the required access pattern matches Redis's data structures and failure model.

1.7 Deployment Shapes at a Glance

A **deployment topology** is the arrangement of Redis processes and the way clients reach them. A **shard** is one partition of a larger dataset, usually owned by a particular primary node. A **primary** accepts writes, while a **replica** maintains a copy of a primary's data. **Failover** is the process of switching service to another node after a failure.

Shape	Main purpose	Important limitation
Standalone	One Redis server for development or workloads that accept a single-node failure domain	The server is a single point of availability and capacity.
Primary with replicas	Keep copies of the primary's data and optionally serve selected reads	Replication is normally asynchronous, so replicas may lag.
Sentinel-managed deployment	Detect failure and promote a replica in a non-sharded topology	Failover is not the same as zero interruption or zero data loss.
Redis Cluster	Partition keys across multiple primary nodes for horizontal scale	Clients and key design must be cluster-aware.
Managed service	Delegate part of deployment, patching, backups, or failover to a provider	Exact behavior, limits, commands, and guarantees depend on the service.

Chapter 1: Redis in the System Architecture

1-7 Standalone and Primary-Replica

Not sharded

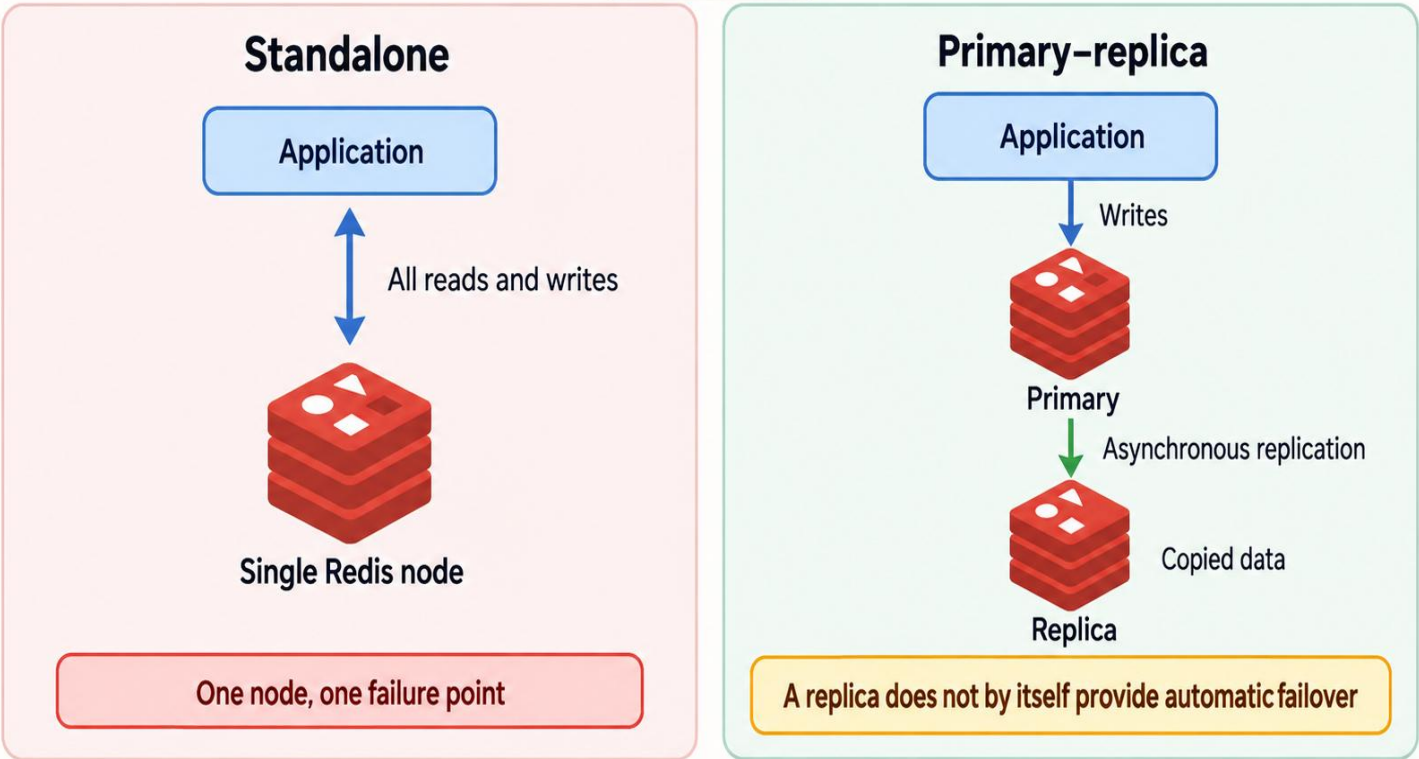
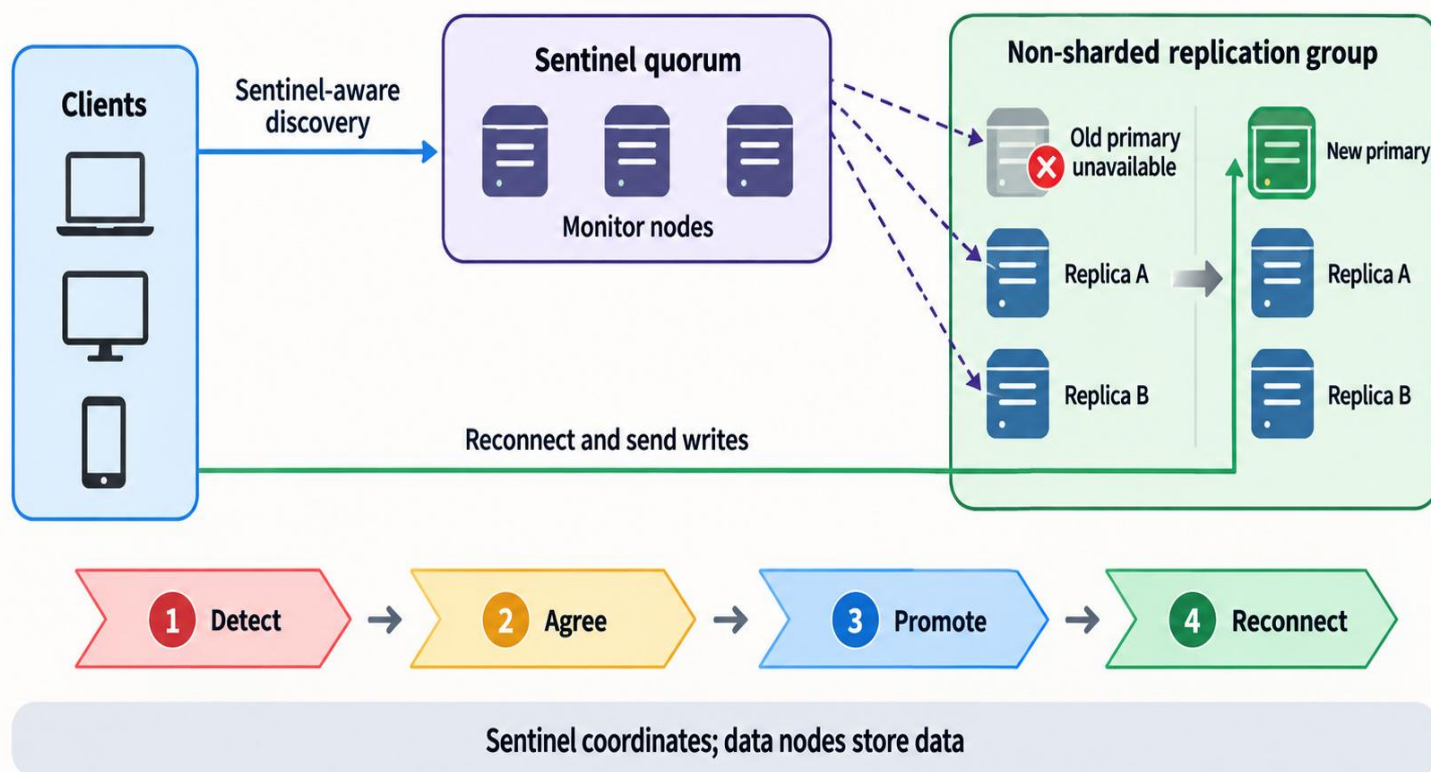


Figure 1-5: Standalone and Primary-Replica Deployment Shapes

Chapter 1: Redis in the System Architecture

1-7 Sentinel-Managed Failover

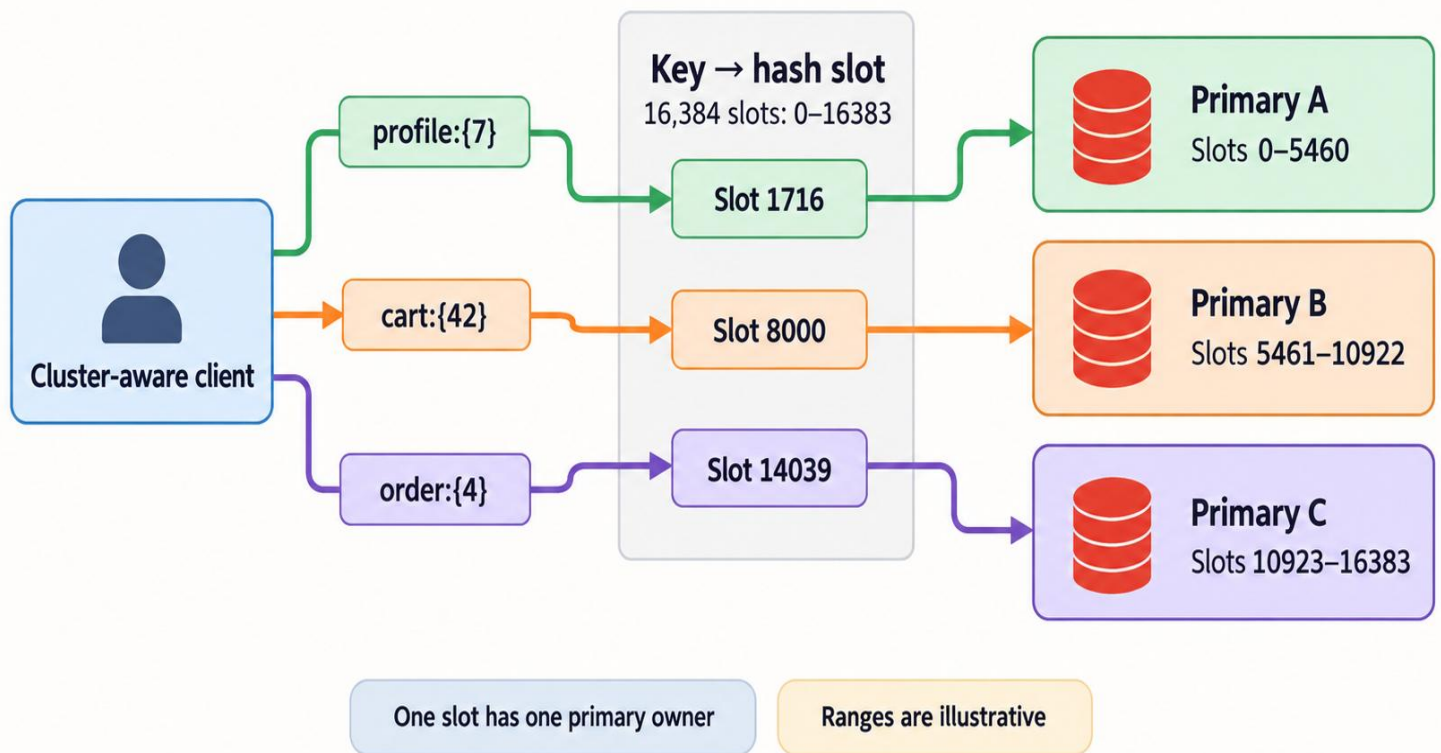


006

Figure 1-6: Sentinel-Managed Non-Sharded Failover Shape

Chapter 1: Redis in the System Architecture

1-7 Redis Cluster Sharding



007

Figure 1-7: Redis Cluster Sharding Shape

These shapes solve different problems. Replication creates copies. Sentinel coordinates failover for a non-sharded deployment. Cluster partitions data. A managed service may implement one or more of these ideas behind a provider-specific interface.

1.8 Four Questions That Follow Every Redis Design

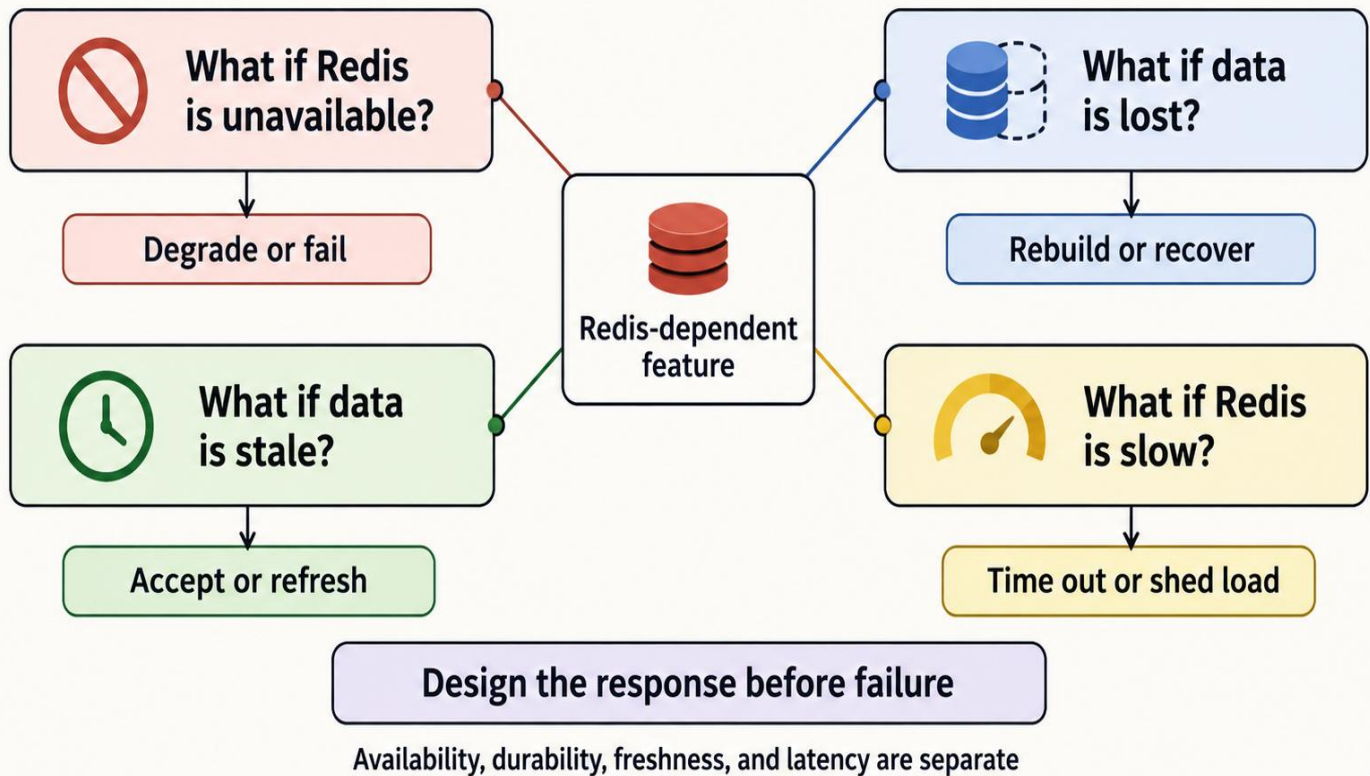
Before approving a Redis use case, ask the following:

- 1. What happens when memory fills?** Writes may fail, keys may be evicted, or the process may be terminated by its environment, depending on configuration.
- 2. What happens when the process restarts?** Data may be reconstructed from persistence, partially recovered, or lost.
- 3. What happens when a node fails?** Clients may see errors, reconnect to another node, read stale data, or lose recent acknowledged writes.
- 4. What happens when a client retries?** A timed-out command may already have executed, so repeating a non-idempotent write may apply it twice.

An **idempotent operation** produces the same intended result when repeated with the same input. Reading a value is normally idempotent; charging a payment twice is not.

Chapter 1: Redis in the System Architecture

1-8 Four Failure Questions



008

Figure 1-8: Four Failure Questions around Every Redis Design

These questions are not reasons to avoid Redis. They are the basic conditions for using any networked stateful service responsibly.

1.9 Correcting the Most Important Misconceptions

“Redis is only a cache.” Redis is a data structure server used for caches, counters, sessions, queues, streams, indexes, and other workloads. Whether a particular use is safe depends on its data-loss and availability requirements.

“In-memory data can never be persisted.” Redis supports persistence, but persistence has selectable trade-offs. It must be configured, monitored, and tested rather than assumed.

“Redis can replace any database because it is faster.” Redis has a different query model, transaction model, memory profile, and operational behavior. Speed does not make two systems interchangeable.

“High availability guarantees zero data loss.” High availability aims to restore service after failure. It does not automatically make replication synchronous or eliminate failover gaps.

Working Principle

Use Redis when its native operations fit the access pattern and when the system explicitly accepts its memory, durability, consistency, and failure trade-offs.

Chapter Summary

Redis is a networked service that performs operations on named data structures. Its speed is useful only when its role, source of truth, durability expectations, topology, and failure behavior are explicit.

Review Questions

1. Where does a Redis command execute, and why does that boundary matter?
 2. How does using Redis as a cache differ from using it as a system of record?
 3. Which four operational questions should follow every Redis design?
-

Chapter 2 — A Safe First Redis Session

What You'll Be Able to Do

- Identify the Redis server, client, endpoint, configuration, and data involved in a session.
- Run a small isolated lab, interpret common replies, and observe basic server state.
- Restart and shut down a lab safely while keeping development exposure separate from production practice.

The first practical goal is to understand the boundary between the Redis server and a client without exposing a database to an untrusted network. An isolated local instance is sufficient for learning commands; it is not a production deployment model.

2.1 The Pieces of a Redis Session

A Redis session involves several distinct components:

- The **Redis server process** owns the data and executes commands.
- A **TCP endpoint** combines a network address and port through which clients connect.
- `redis-cli` is a command-line client that sends commands and displays replies.
- A **configuration** controls server behavior such as networking, persistence, memory limits, and security.

Chapter 2: A Safe First Redis Session

2-1 Components of a Redis Session

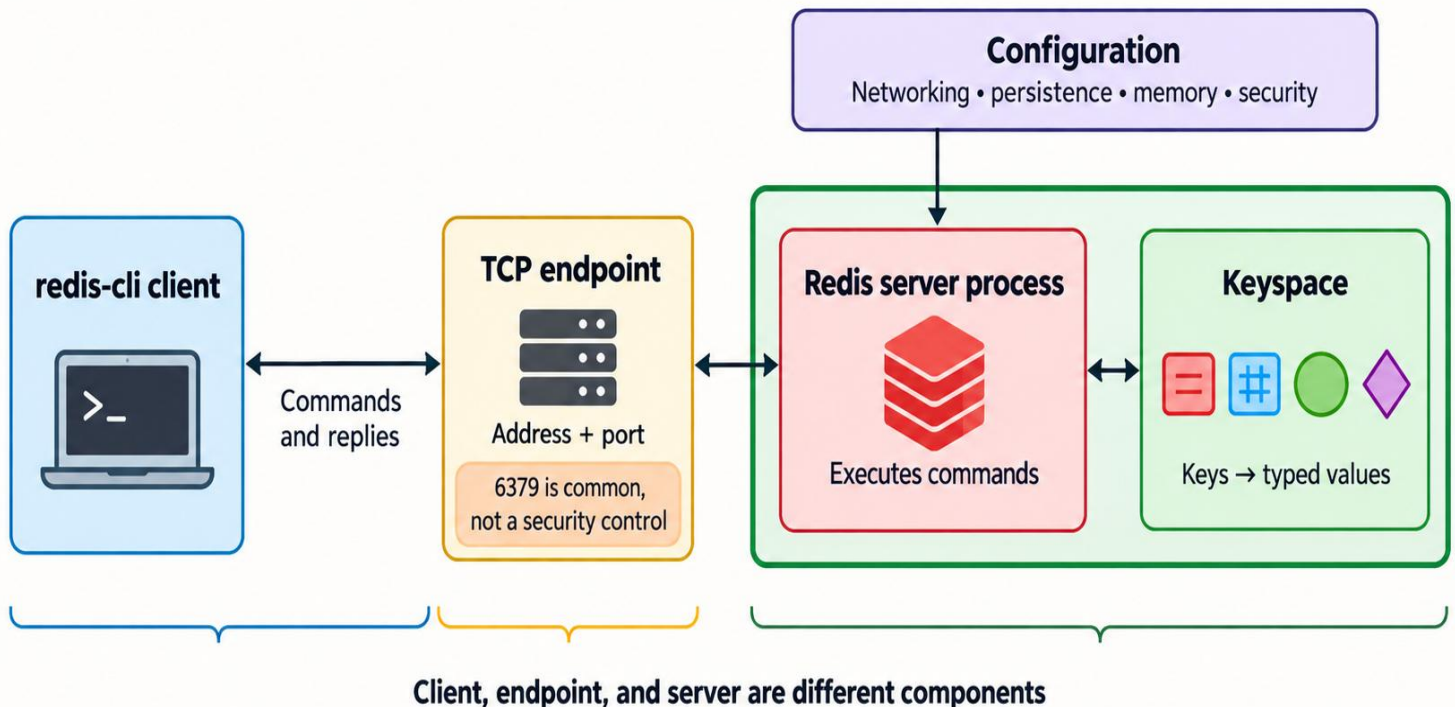


Figure 2-1: Components of a Redis Session

The commonly used Redis TCP port is 6379, but a port number is not a security control. A service that listens on an untrusted interface without appropriate authentication and network restrictions can be attacked even when it is intended only for development.

2.2 Start an Isolated Docker Lab

The following is **one local example** for a machine with Docker installed. Native packages and managed services use different startup procedures.

Prerequisites: Docker is installed, no other service is using local port 6379, and the machine is not treating this container as a production data store.

Purpose: Start a Redis 8.10 container that listens only on the local loopback interface. Persistence is disabled so that restart behavior is easy to observe.

```
docker run --name redis-lab -d \
  -p 127.0.0.1:6379:6379 \
  redis:8.10 \
  redis-server --save "" --appendonly no
```

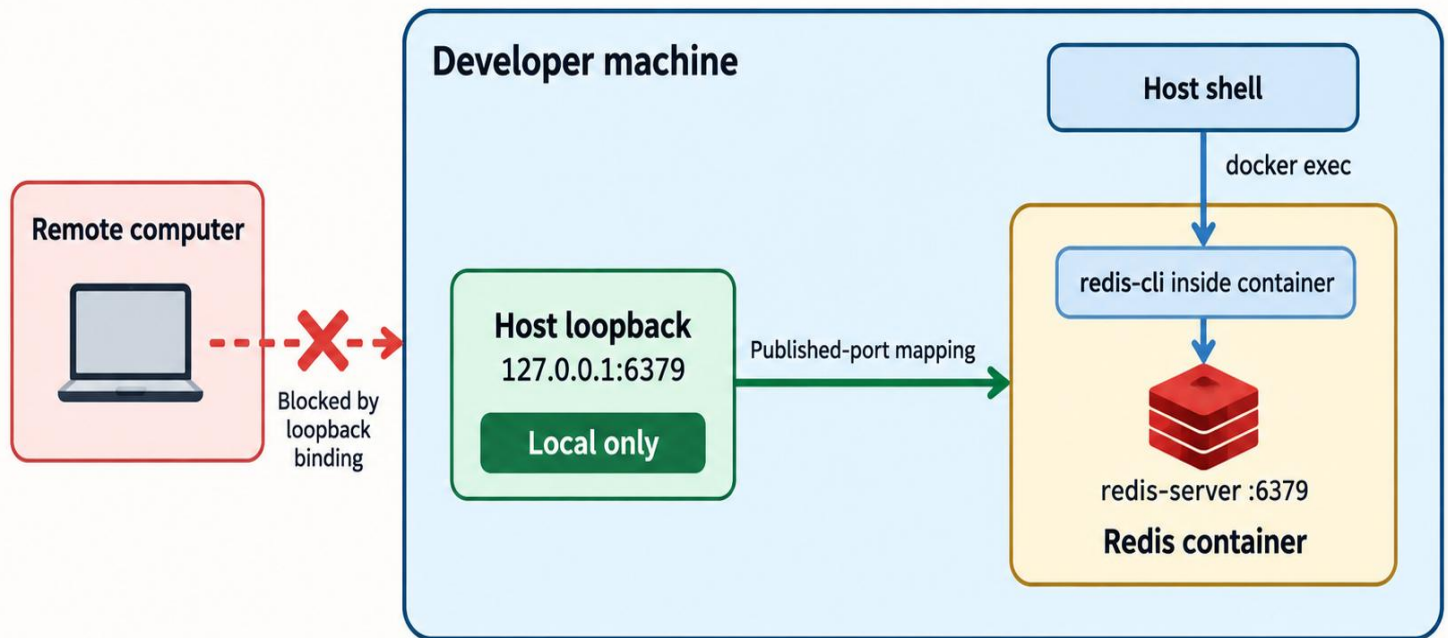
Expected result: Docker prints a container identifier. The container named `redis-lab` runs in the background, and Redis is reachable only through `127.0.0.1:6379` on the host. The **loopback interface** is a network path that reaches only the local machine.

Binding the published port to `127.0.0.1` matters. Publishing `6379:6379` without the address may expose the port on additional host interfaces, depending on the environment.

Chapter 2: A Safe First Redis Session

2-2 Isolated Docker Lab Network

One local Docker lab example



Loopback limits reachability; it is not complete production security

010

Figure 2-2: Isolated Docker Lab Network Path

2.3 Verify Connectivity

Prerequisite: The `redis-lab` container is running.

Purpose: Send the simplest health check from a CLI inside the container.

```
docker exec redis-lab redis-cli PING
```

Expected result:

```
PONG
```

PING proves that the client reached a Redis server and received a reply. **It does not prove that the connection is encrypted, access is properly restricted, persistence is healthy, or the instance is safe for production use.**

Open an interactive client for the remaining exercises:

Prerequisite: The container is running.

Purpose: Start a command session connected to the local Redis server.

```
docker exec -it redis-lab redis-cli
```

Expected result: A prompt similar to `127.0.0.1:6379>` appears.

2.4 A Minimal Command Sequence

The following commands introduce the basic request-and-reply model.

Prerequisite: An interactive `redis-cli` session is open against the lab instance.

Purpose: Store a value, read it, create a counter, inspect a key's type, and delete a key.

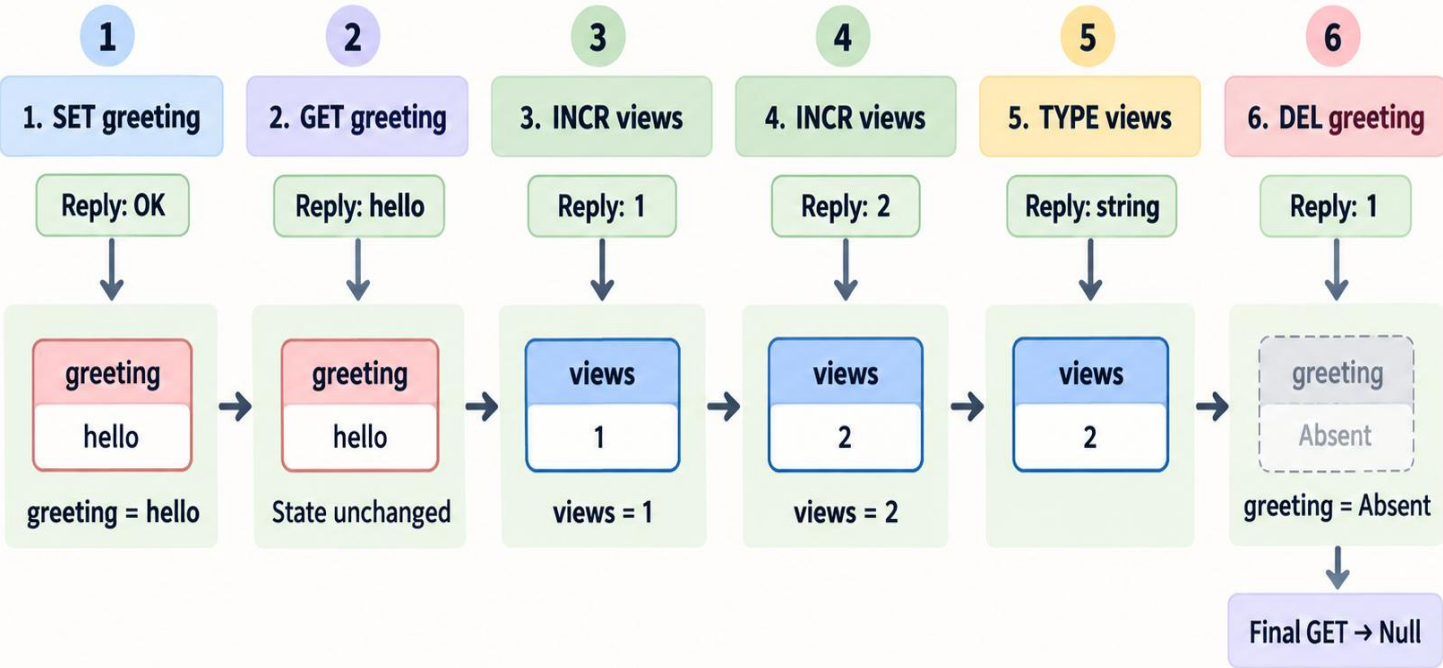
```
> SET training:greeting "hello"
OK
> GET training:greeting
"hello"
> INCR training:page:home:views
(integer) 1
> INCR training:page:home:views
(integer) 2
> TYPE training:page:home:views
string
> DEL training:greeting
(integer) 1
> GET training:greeting
(nil)
```

Expected result: SET returns OK; GET returns the stored bytes; successive INCR commands return 1 and 2; TYPE reports string; DEL reports that one key was removed; and the final GET returns a null reply.

A counter is still stored under the Redis **string** data type. Redis does not expose a separate top-level integer type. Numeric commands interpret compatible string values as numbers.

Chapter 2: A Safe First Redis Session

2-4 First Key State Changes



Commands change or inspect server-owned state

011

Figure 2-3: Key State Changes in the First Command Sequence

2.5 Reading CLI Replies

The CLI formats several reply categories differently.

Display	Meaning	Typical example
OK or PONG	A simple status string	Successful SET or PING
"hello"	A string value	GET or a field read
(integer) 2	An integer reply	INCR, DEL, or a count
A numbered sequence	An array or collection reply	MGET, LRANGE, or HGETALL
(nil)	No value was returned	A missing key or timed-out blocking read
(error) ...	The command could not be completed	Wrong type, invalid arguments, or permission denial

To see a type error safely, use a disposable lab key.

Prerequisite: The interactive CLI is connected to the non-production lab.

Purpose: Observe how Redis reports an operation that is incompatible with a value.

```
> SET training:not-a-number "hello"
OK
> INCR training:not-a-number
(error) ERR value is not an integer or out of range
```

Expected result: The value remains unchanged, and Redis returns an error explaining that it cannot be treated as an integer.

Redis commands are type-aware. A list command applied to a hash, or a numeric command applied to non-numeric bytes, returns an error rather than silently converting the value.

2.6 Observe the Server, Not Just the Data

`INFO` returns operational information grouped into sections. A beginner does not need to memorize every metric, but should learn that the server can report its own state.

Prerequisite: The lab instance is running.

Purpose: Inspect server identity and basic memory information without requesting the full report.

```
> INFO server
> INFO memory
```

Expected result: Each command returns lines in `name:value` form. The server section includes version and process details; the memory section includes measurements such as used memory.

The exact values vary by host and Redis version. The purpose is to distinguish **application data commands** from **operational observation commands**.

Container logs provide another view.

Prerequisite: Run this from the host shell, not inside `redis-cli`.

Purpose: Read the Redis process startup and runtime log messages emitted by the container.

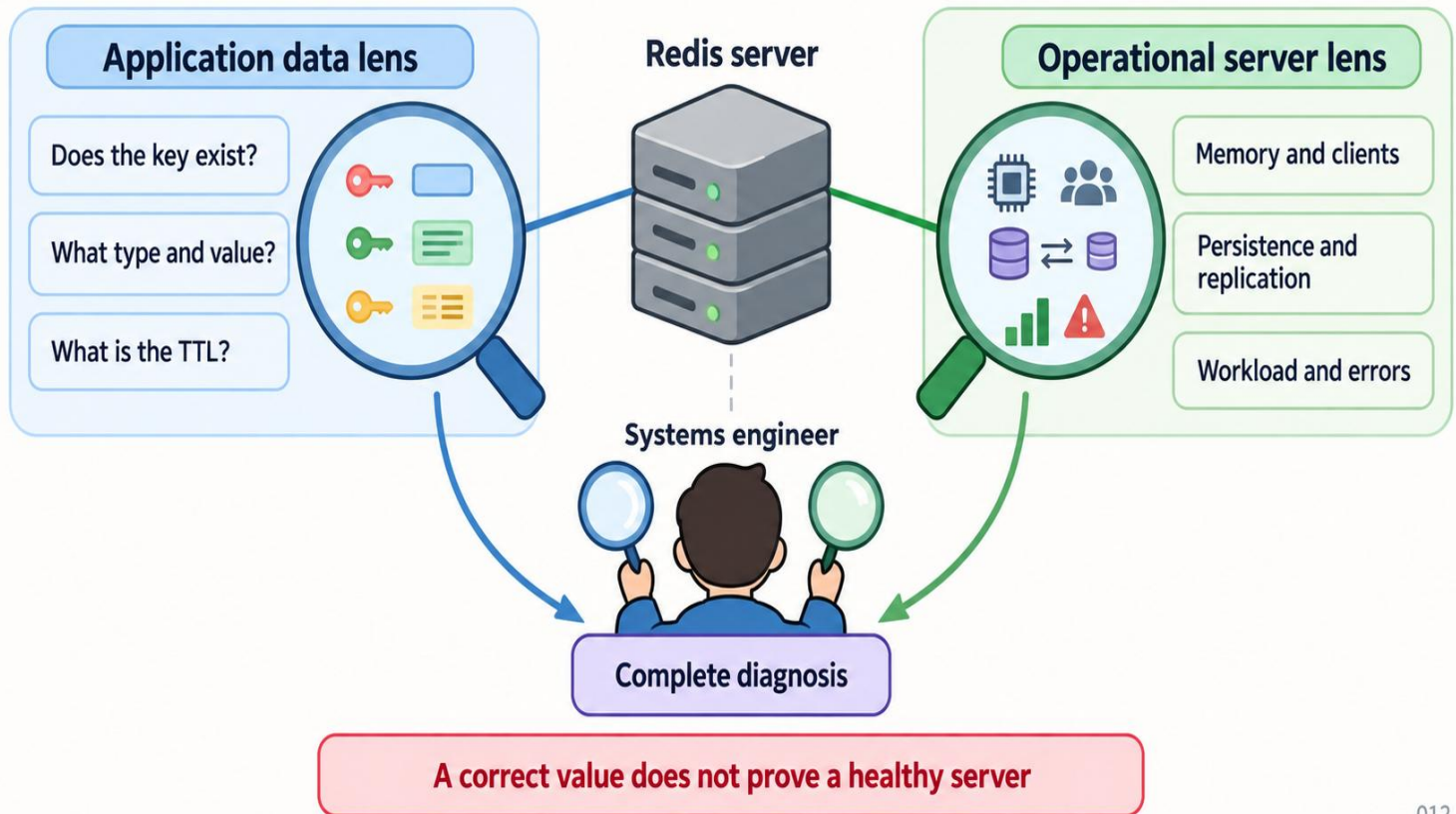
```
docker logs redis-lab
```

Expected result: The output includes Redis startup information and a message indicating that the server is ready to accept connections.

A healthy PING and clean logs are useful signals, but they do not replace monitoring of latency, memory, persistence, replication, and errors in a real environment.

Chapter 2: A Safe First Redis Session

2-6 Data View and Server View



012

Figure 2-4: Application Data View versus Operational Server View

2.7 A First Restart Experiment

The lab was started with snapshot and append-only persistence disabled. The following experiment demonstrates that **memory contents disappear when the process restarts if no persistence mechanism reconstructs them**.

Prerequisites: The `redis-lab` container is running with `--save "" --appendonly no`.

Purpose: Create a page-view counter, restart the Redis process through Docker, and test whether the key returns.

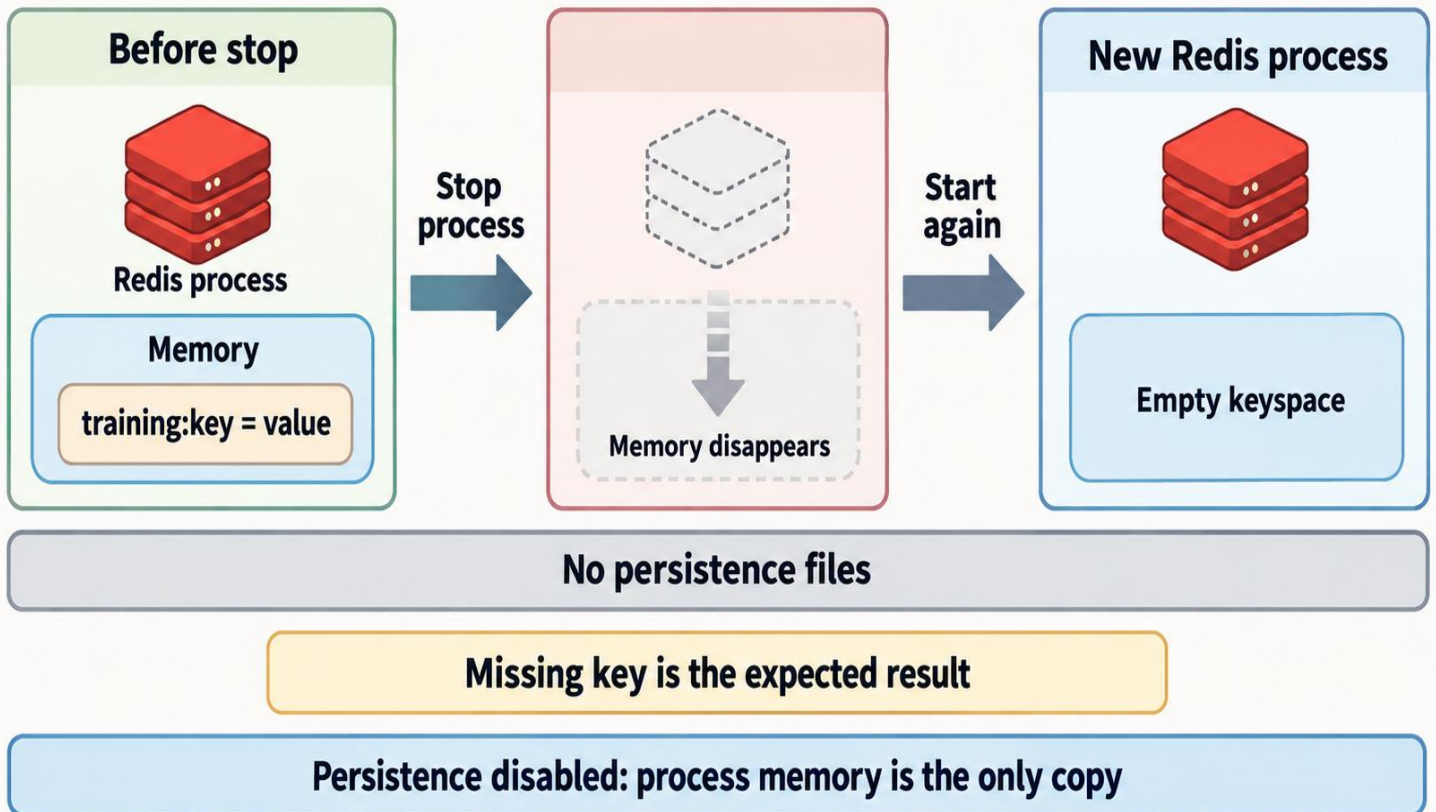
```
docker exec redis-lab redis-cli SET training:restart-test 1
docker restart redis-lab
docker exec redis-lab redis-cli GET training:restart-test
```

Expected result: The `SET` returns OK; Docker restarts the container; and `GET` returns a null reply because the instance was configured not to persist data.

This result does not mean Redis is always volatile. It means **this particular server configuration had no reconstruction source**.

Chapter 2: A Safe First Redis Session

2-7 Restart without Persistence



013

Figure 2-5: Restart without Persistence

Contrast with a Persisted Lab

The next example enables append-only persistence and mounts a Docker volume. It is still a learning setup, not a production prescription.

Prerequisites: Docker is installed, the existing lab can be removed, and losing its test data is acceptable.

Purpose: Start a new local instance whose data directory survives container restart, then observe a normal restart.

```
docker rm -f redis-lab
docker volume create redis-lab-data

docker run --name redis-lab -d \
  -p 127.0.0.1:6379:6379 \
  -v redis-lab-data:/data \
  redis:8.10 \
  redis-server --appendonly yes

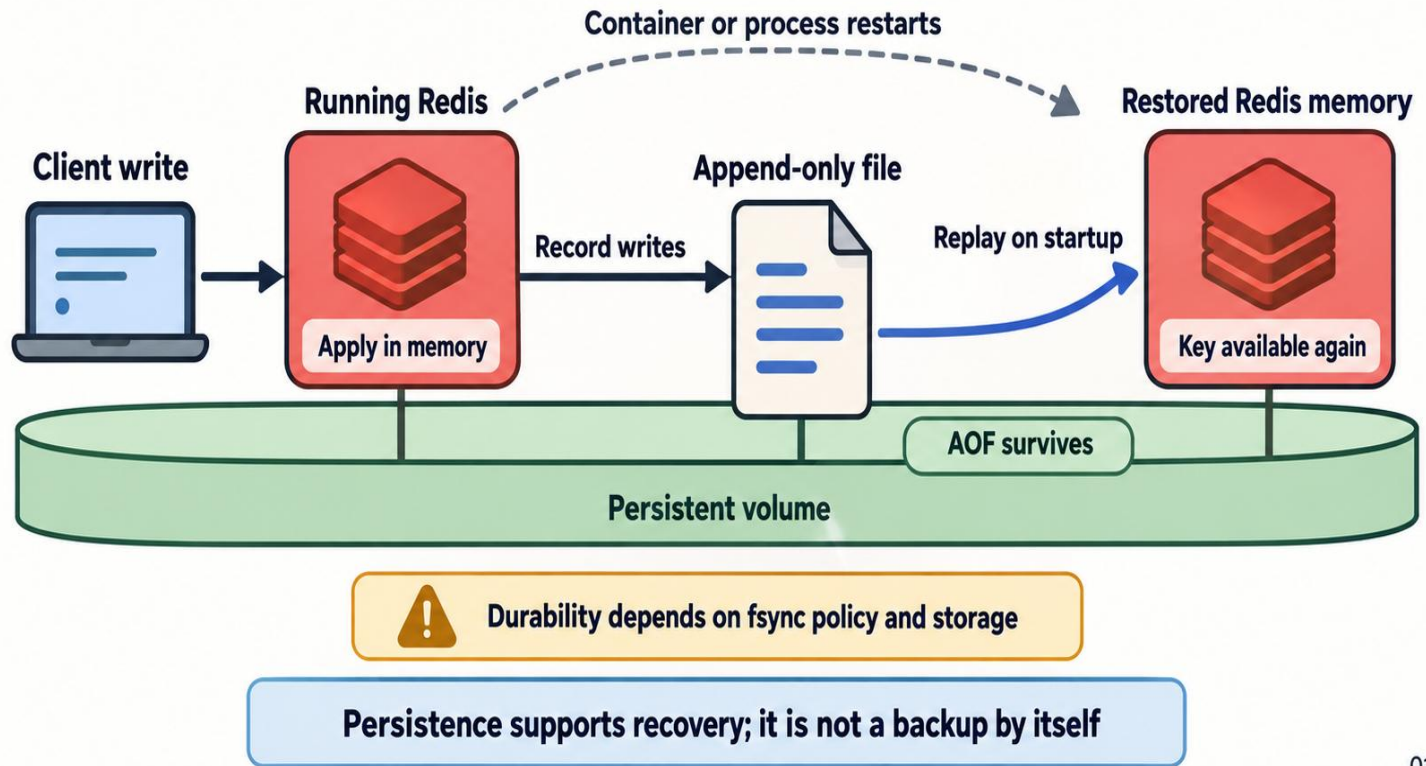
docker exec redis-lab redis-cli SET training:persisted "yes"
docker restart redis-lab
docker exec redis-lab redis-cli GET training:persisted
```

Expected result: After the normal restart, `GET` returns `"yes"`.

An **append-only file**, often abbreviated **AOF**, records write operations so Redis can reconstruct data after restart. This small exercise proves only that the test value returned after a normal restart. It does not establish a complete durability, backup, or disaster-recovery guarantee.

Chapter 2: A Safe First Redis Session

2-7 Restart with AOF and Volume



014

Figure 2-6: Restart with AOF and a Persistent Volume

2.8 Graceful Shutdown in the Lab

A **graceful shutdown** gives the server an opportunity to complete its configured shutdown actions instead of being terminated abruptly.

Prerequisite: The persisted lab is running and no other work depends on it.

Purpose: Ask Redis to shut itself down, then start the same container again.

```
docker exec redis-lab redis-cli SHUTDOWN
docker start redis-lab
docker exec redis-lab redis-cli PING
```

Expected result: The first command closes the Redis server and stops the container; `docker start` starts it again; and `PING` returns `PONG`.

Do not run shutdown commands against shared or production systems without an approved operational procedure.

2.9 Security Boundaries for a Learning Environment

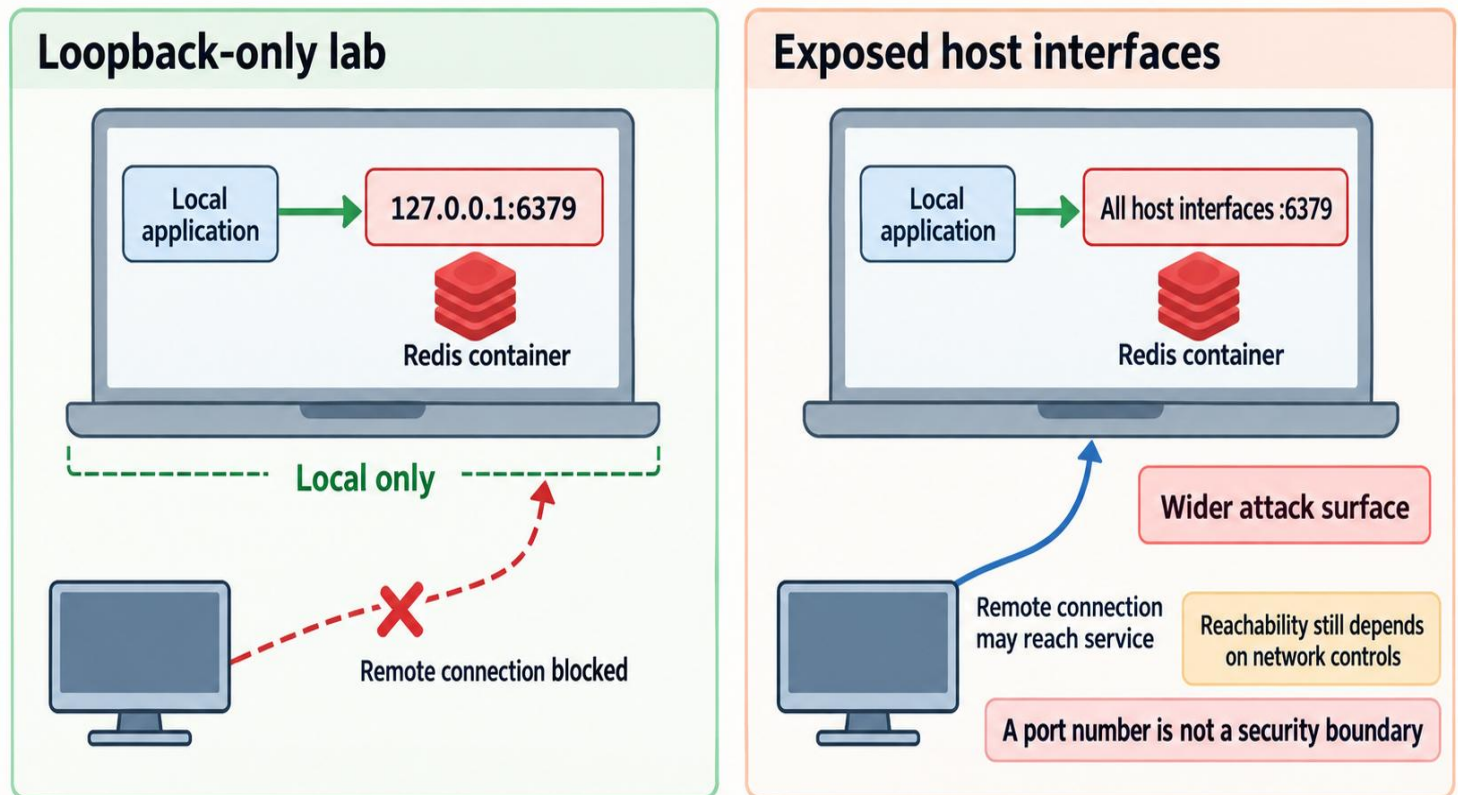
A safe lab follows a few simple rules:

- Publish the port only on loopback unless remote access is intentionally secured.
- Use disposable test data.
- Do not reuse production passwords, certificates, or connection strings.

- Do not treat a successful connection as proof of secure configuration.
- Remove the lab when it is no longer needed.

Chapter 2: A Safe First Redis Session

2-9 Loopback versus Exposed Port



015

Figure 2-7: Loopback-Only Lab versus Exposed Redis Port

To remove the container and its optional test volume:

Prerequisite: No needed test data remains in the lab.

Purpose: Delete the local Redis container and the named Docker volume used in the persistence example.

```
docker rm -f redis-lab
docker volume rm redis-lab-data
```

Expected result: Docker prints the removed container and volume names. If the container or volume has already been removed, Docker reports that it cannot find it.

2.10 The Practical Lesson

The first session establishes four facts:

1. Redis is a separate server reached through a client connection.
2. Commands operate on typed values stored under keys.
3. Replies communicate values, counts, missing data, and errors.
4. Restart behavior comes from configuration and storage, not from the word “Redis” alone.

A development CLI session is a learning and diagnostic tool, not a substitute for secure deployment design.

Chapter Summary

A safe first session establishes the client-server boundary and builds confidence with observable commands. A successful command proves connectivity and behavior in that lab; it does not prove production security, durability, or readiness.

Review Questions

1. What does a successful PING confirm, and what does it not confirm?
 2. How do a null reply and an error reply differ?
 3. Why should a learning instance remain isolated from untrusted networks?
-