Distributed Systems & Scalability

A Complete 10-Chapter Software Development Course

Topics covered:

Scaling fundamentals · load balancing · caching strategies

Database scaling · the CAP theorem · message queues

Rate limiting · API gateways & service discovery

Fault tolerance & resilience patterns

Capstone: a URL shortener assembling every chapter's own verified pattern into one system

Exercises: 30 hands-on exercises with worked solutions

Format: A4 · Dark-theme code examples

Philip Osztromok · Generated with Claude

Table of Contents

CHAPTER 1 OF 10

Why Systems Need to Scale

Distributed Systems & Scalability

Chapter 1 · Why Systems Need to Scale

Software Architecture Fundamentals answered "how should this system's own code be organized?" This course answers a genuinely different question: once that system is correctly organized, what happens when real load hits it? A correctly-layered, correctly-bounded architecture (Software Architecture Fundamentals' own `OrderService` / `OrderRepository` / `PricingEngine`) doesn't automatically survive scale — this chapter measures two completely different reasons why.

Reason 1: An Algorithmic Bottleneck, Hiding in Correct-Looking Code

A duplicate-order-ID check is a natural thing to add to `OrderRepository`. The obvious implementation scans every existing order — and that "obvious" choice is the actual bottleneck.

```
def order_id_exists_linear(order_ids_list, order_id): return order_id in order_ids_list # O(n) — scans the list def order_id_exists_set(order_ids_set, order_id): return order_id in order_ids_set # O(1) average case — hash lookup
```

VERIFIED DIRECTLY — THE LINEAR CHECK'S COST GROWS WITH DATA SIZE; THE HASH CHECK'S DOESN'T

Checking for the last-inserted order ID (the worst case for a linear scan) against a growing order list: at **1,000** orders, the linear check took **7.78 microseconds** versus the set check's **0.075 microseconds** — **104×** slower. At **20,000** orders, the linear check grew to **141.88 microseconds** while the set check stayed essentially flat at **0.080 microseconds** — now **1,785×** slower. The linear check's own cost grew roughly in proportion to the order count; the set check's didn't move at all.

THIS IS NOT A SCALING PROBLEM — AND NO AMOUNT OF SCALING FIXES IT

Buying a faster server (vertical scaling) makes every microsecond figure above smaller, but the *growth curve* stays exactly the same shape — the linear check will still eventually outpace any fixed amount of extra speed as the order count keeps growing. Running the check on more machines (horizontal scaling) doesn't help either, unless the order list itself gets split up somehow — which raises its own hard question (Chapter 4's own database-sharding chapter). The actual fix here is Chapter 1's own lesson from Software Architecture Fundamentals, one level deeper: sometimes what "breaks first" isn't the architecture's own shape, it's one specific algorithm hiding inside a correctly-organized component.

Reason 2: A Single Process Can Only Use So Much Hardware

Even a perfectly-written `PricingEngine` runs as ordinary Python code in one process. Genuinely parallelizing it — running several calculations at once, on several CPU cores — needs more than just "a bigger server."

```
def cpu_heavy_pricing_calc(base_price): # genuine CPU-bound work, not a sleep() simulation total = base_price for i in range(200000): total = (total * 1.0000001) % 100000 return round(total, 2)
```

VERIFIED DIRECTLY — PARALLELIZING A SMALL WORKLOAD MADE IT GENUINELY SLOWER

Running **8** pricing calculations sequentially took **0.069s**. Running the same 8 across **4 processes** (Python's own `multiprocessing.Pool`) took **0.207s** — roughly **3× slower**, not faster. Spinning up separate OS processes has real overhead, and for a workload this small, that overhead cost more than the parallelism saved.

VERIFIED DIRECTLY — A GENUINELY LARGE WORKLOAD DOES BENEFIT, BUT SUB-LINEARLY

Running **160** calculations sequentially took **1.327s**. The identical 160 calculations across **2** processes took **0.816s** (**1.63×** faster); across **4** processes, **0.532s** (**2.49×**); across **8** processes, **0.472s** (**2.81×**). Doubling the workers from 2 to 4 improved things — but nowhere near doubled the speedup, and going from 4 to 8 barely moved the number at all. Every result matched the sequential version's own output exactly, confirming the parallel version computed the identical correct answers, just faster.

VERTICAL VS. HORIZONTAL, STATED PRECISELY

Vertical scaling — a faster CPU, more RAM — would have sped up *every* number in both findings above uniformly, small workload included, with zero code changes and zero risk of making anything slower. **Horizontal scaling** — more processes, more machines — only helped once the workload was large enough to be worth splitting up, and even then delivered real but diminishing returns, never the "8 workers = 8× faster" result a naive mental model would predict.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
A 1,785× algorithmic slowdown, invisible until measured at real scale	Chapter 4's Database Scaling — sharding a growing dataset is the production-scale version of fixing exactly this kind of bottleneck
Small-workload parallelism verified genuinely slower, not just "less beneficial"	Chapter 2's Load Balancing — routing overhead has the same shape of cost, worth measuring before assuming it's free
Sub-linear speedup even for a large, genuinely parallelizable workload	Technical Support's own `perfdiag1` — this chapter's own honest ceiling is exactly the kind of number that course's diagnostic chapters teach how to notice in a real production dashboard

Hands-On Exercises

EXERCISE 1

Repeat this chapter's own linear-vs-set benchmark, but check for an order ID that's *first* in the list rather than last. Verify the linear check's timing changes dramatically while the set check's doesn't, and explain why using this chapter's own O(n)/O(1) framing.

View solution

EXERCISE 2

Re-run this chapter's own small-workload multiprocessing benchmark (8 calculations) with only **2** processes instead of 4. Verify whether the "parallelizing makes it slower" finding still holds, and report the actual speedup ratio.

View solution

EXERCISE 3

Using this chapter's own two verified findings, explain why "just add more servers" is not a universal fix for a slow system — name the specific condition each finding shows has to be true before horizontal scaling actually helps.

View solution

CHAPTER 1 QUICK REFERENCE

- **Vertical scaling:** a more powerful single machine — helps uniformly, but has a ceiling and never fixes an algorithmic growth-rate problem
- **Horizontal scaling:** more machines/processes working in parallel — verified: genuinely slower for a small workload (overhead dominates), genuinely faster but sub-linear for a large one (1.63×–2.81× across 2–8 processes)
- **Verified:** a linear duplicate-check grew from 104× to 1,785× slower than an O(1) equivalent as data size grew from 1,000 to 20,000 — a bottleneck no amount of scaling alone fixes
- **Next chapter:** Load Balancing — how requests actually get distributed once there's more than one server to send them to

CHAPTER 2 OF 10

Load Balancing

Distributed Systems & Scalability

Chapter 2 · Load Balancing

Chapter 1 showed horizontal scaling only helps once work can genuinely be split across machines. Load balancing is the part that actually does the splitting — and *how* it splits requests turns out to matter as much as whether it splits them at all.

Round Robin vs. Least Connections, With Unequal Servers

```
class RoundRobinBalancer: def __init__(self, servers): self.servers = servers; self.index = 0 def route(self): server = self.servers[self.index % len(self.servers)] self.index += 1 server.enqueue(); return server class LeastConnectionsBalancer: def __init__(self, servers): self.servers = servers def route(self): server = min(self.servers, key=lambda s: s.queue) server.enqueue(); return server
```

Three servers — two fast (capacity 5 requests/tick), one genuinely slow (capacity 2/tick) — under 12 new requests every tick for 20 ticks:

VERIFIED DIRECTLY — ROUND ROBIN LETS THE SLOW SERVER'S QUEUE GROW UNBOUNDEDLY

Under **Round Robin**, which sends exactly a third of all traffic to the slow server regardless of how backed up it gets, the slow server's queue reached **40** outstanding requests and never drained — it was still growing when the simulation ended. Under **Least Connections**, which always routes to whichever server currently has the fewest queued requests, the slow server's queue never exceeded **3**. Both balancers received the identical total request volume — the only difference was which server each new request landed on.

ROUND ROBIN ISN'T BROKEN — IT'S ANSWERING A DIFFERENT QUESTION

Round Robin guarantees each server gets an equal *share of requests*. Least Connections guarantees each server gets a roughly equal *amount of outstanding work*. Those are the same thing only when every server processes requests at the same speed — Chapter 1's own horizontal-scaling findings already showed that assumption doesn't always hold, even for identical hardware under different load.

Health Checks: Routing Around a Server That's Actually Down

VERIFIED DIRECTLY — A HEALTH-BLIND BALANCER KEPT FAILING REQUESTS AFTER A SERVER WENT DOWN; A HEALTH-CHECKING ONE DIDN'T

Simulating server `B` going down partway through 30 requests: a plain Round Robin balancer with **no health checking** kept routing roughly a third of all remaining requests to the dead server, producing **7 failed requests** out of 30. The identical scenario, routed through a balancer that filters to only currently-healthy servers before choosing one, produced `0` **failed requests** — every request was automatically redirected to `A` or `C` instead.

A LIGHTER-WEIGHT COUSIN OF DESIGN PATTERNS' OWN STATE

This chapter modeled server health as a simple `healthy` boolean, checked with a plain `if`. Design Patterns Chapter 8 modeled a genuinely richer set of behaviors — an order's status — as full state *objects*, each owning its own transition rules. If a real health-check system needed more than "route or don't" (say, a "draining" state that finishes existing connections but accepts no new ones), the boolean would stop being enough, and reaching for that same State pattern would be the natural next step — not a different idea, just a heavier tool for a genuinely more complex version of the same problem.

Sticky Sessions: Solving Statelessness's Problem, Creating a New One

Software Architecture Fundamentals Chapter 8 verified a stateful design losing data on a server restart, and a stateless one surviving it. **Sticky sessions** are the load-balancer-level alternative: pin a client to the same server for their whole session, so that server *can* hold state in memory safely. What does pinning cost?

```
def sticky_route(session_id): h = int(hashlib.md5(session_id.encode()).hexdigest(), 16) return
SERVERS[h % len(SERVERS)] # pinned for the session's whole lifetime
```

VERIFIED DIRECTLY — STICKY SESSIONS PRODUCED A GENUINELY UNEVEN SPLIT FROM NOTHING BUT HASH LUCK

Hash-pinning **30** distinct client sessions across 3 identical servers, then sending 5 requests per client (**150** total): the resulting split was **A: 60, B: 65, C: 25** — a spread of **40** between the busiest and quietest server, despite every server being identical and every client sending exactly the same amount of traffic. Routing the identical 150 requests through **Least Connections** instead — no stickiness, no pinning — produced a perfectly even **A: 50, B: 50, C: 50**, a spread of `0`.

THE REAL TRADEOFF, STATED PRECISELY

Sticky sessions buy back the option of stateful, in-memory design that Software Architecture Fundamentals Chapter 8 verified breaking under a stateless assumption — but they do it by giving up the load balancer's own ability to route around uneven load, verified above producing a 40-request spread from just 30 clients. This is exactly the kind of tradeoff Chapter 8 itself flagged: neither option is free, and which one to accept depends on what the system actually needs more.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
An unbounded queue under Round Robin with unequal server speeds	Chapter 1's own horizontal-scaling findings — unequal processing capacity is exactly the condition that makes a "fair share of requests" different from "fair share of work"
Zero failures with health checking vs. 7 without	Chapter 9's own Fault Tolerance & Resilience Patterns — health checking is the first, simplest resilience pattern this course covers
Sticky sessions' verified 40-request spread from pure hash luck	Software Architecture Fundamentals Chapter 8's own statelessness finding — the two chapters verify opposite sides of the identical tradeoff

Hands-On Exercises

EXERCISE 1

Re-run this chapter's own Round Robin vs. Least Connections simulation with *two* slow servers (capacity 2/tick) and only one fast server (capacity 5/tick). Verify whether Least Connections still keeps queues bounded, and report the new queue depths for both balancers.

View solution

EXERCISE 2

Using this chapter's own health-checking simulation, make *two* of the three servers go down at different points (B at request 10, C at request 20). Verify the health-checking balancer still produces zero failures, and report how many requests land on the one server left standing.

View solution

EXERCISE 3

Using this chapter's own two verified findings (unequal-speed Round Robin, and sticky sessions), explain what specifically these two scenarios have in common: why does "give every option an equal share" produce a worse outcome than "route based on current state" in both cases?

View solution

CHAPTER 2 QUICK REFERENCE

- **Round Robin:** equal share of requests — verified: let a slow server's queue grow to `40` and keep climbing, when server speeds genuinely differ
- **Least Connections:** routes to whichever server has the least outstanding work — verified: kept the same slow server's queue capped at `3`
- **Health checks, verified:** `7` failed requests without them, `0` with them, for the identical mid-run server failure
- **Sticky sessions, verified:** a `40`-request spread from pure hash luck across 3 identical servers, versus `0` spread for the same traffic under Least Connections — the direct tradeoff against Software Architecture Fundamentals Chapter 8's own statelessness finding
- **Next chapter:** Caching Strategies — cache-aside, write-through, write-behind, and why invalidation is the genuinely hard part

CHAPTER 3 OF 10

Caching Strategies

Distributed Systems & Scalability

Chapter 3 · Caching Strategies

Caching trades correctness risk for speed — every strategy in this chapter is a different way of drawing that trade. The famous line about cache invalidation being one of the two hard problems in computer science isn't a joke about difficulty in the abstract; this chapter builds a real, verified case of exactly what goes wrong.

Cache-Aside: Check the Cache First, Populate on Miss

```
class CacheAsideRepository: def __init__(self, database): self.database = database; self.cache = {} def get(self, key): if key in self.cache: return self.cache[key] value = self.database.get(key) self.cache[key] = value return value
```

VERIFIED DIRECTLY — A REAL, MEASURED SPEEDUP ON EVERY READ AFTER THE FIRST

Against a simulated database with real I/O latency, the first read of `PROD-1` (a genuine cache miss) took **10.37ms** and incremented `db.query_count` to **1**. Twenty subsequent reads of the identical key averaged **0.0005ms** each — `db.query_count` never moved past `1`. Cache-hit reads were roughly **19,566×** faster than the original cache-miss read.

Write-Through vs. Write-Behind: Where the Real Tradeoff Lives

```
class WriteThroughCache: def set(self, key, value): self.cache[key] = value self.database.set(key, value) # synchronous, immediate class WriteBehindCache: def set(self, key, value): self.cache[key] = value self.pending_writes[key] = value # NOT written to the database yet def flush(self): for key, value in self.pending_writes.items(): self.database.set(key, value) self.pending_writes.clear()
```

VERIFIED DIRECTLY — WRITE-BEHIND IS DRAMATICALLY FASTER, AND THAT SPEED IS EXACTLY WHAT MAKES IT RISKY

Twenty writes through **write-through** took **206.47ms** total (each one blocking on the real database write). The identical twenty writes through **write-behind**, before `flush()` ever runs, took **0.02ms** total — roughly **12,145×** faster, because nothing has touched the database yet.

VERIFIED DIRECTLY — A SIMULATED CRASH BEFORE FLUSH() LOSES THE WRITE ENTIRELY

Calling `write_behind.set('PROD-1', 100)` and then checking `database.get('PROD-1')` *before* `flush()` runs returns `None` — the write genuinely never reached the database. `cache.cache['PROD-1']` correctly shows `100` the whole time. Simulating a crash at exactly this point (the process ends, `flush()` never gets called) confirms the write is **gone** — not delayed, gone. Only once `flush()` actually runs does `database.get('PROD-1')` correctly return `100`.

THE TRADEOFF, STATED PRECISELY

Write-through pays the full write cost every time, in exchange for a guarantee: the moment `set()` returns, the database and cache genuinely agree. Write-behind defers that cost — verified **12,145×** faster — but during the deferral window, the only copy of the truth lives in memory, verified vanishing entirely on a simulated crash. This is the identical shape of tradeoff Distributed Systems & Scalability's own future CAP Theorem chapter formalizes: speed and immediate durability aren't both free at the same time.

Cache Invalidation: The Genuinely Hard Part

Reusing Software Architecture Fundamentals' own `PricingEngine` (Ch.7): what happens when a cached price outlives the data it was calculated from?

VERIFIED DIRECTLY — A CUSTOMER SEES A REAL, WRONG PRICE AFTER THE UNDERLYING DATA GENUINELY CHANGED

With `PROD-1` well-stocked (`50` units), `CachedPricingService.get_price()` correctly caches and returns `100` (no scarcity pricing). Stock then genuinely drops to `5` — a real change that should trigger scarcity pricing. Querying the *identical* cache key again, without invalidating anything, still returns the stale `100` — while the correct price, computed fresh, is now `115.0`. The customer would be quoted the wrong price, with no error, no warning, and nothing in the code path that looks broken.

VERIFIED DIRECTLY — EXPLICIT INVALIDATION IS WHAT ACTUALLY FIXES IT

Calling `cached_service.invalidate('PROD-1')` — clearing every cached entry for that product — and querying again correctly returns `115.0`, matching the true current price exactly.

WHY THIS IS THE "HARD" PART, SPECIFICALLY

Caching itself (Cache-Aside above) was mechanical — check, miss, populate, return. Knowing *when* a cached value has become wrong requires knowing about every possible change to the data it depends on, from every part of the system that could make one — here, specifically, that a stock update needs to trigger a price-cache invalidation, a connection that isn't visible anywhere in the caching code itself. Missing even one such trigger reproduces the exact bug just verified: a technically-working cache silently serving wrong answers.

CDN Basics

A Content Delivery Network applies cache-aside's own idea geographically: static content (images, scripts, stylesheets) gets cached at servers physically close to each visitor, instead of every request crossing however much distance separates the visitor from the origin server. Software Architecture Fundamentals Chapter 4 measured a real, non-zero cost for crossing a network boundary even on `localhost` — a CDN exists specifically to shrink that same kind of cost when the distance is a real geographic one, not a loopback address.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
Write-behind's verified 12,145× speed gain, traded for a verified real data-loss window	Chapter 5's own CAP Theorem & Consistency Models — the same speed-vs-guarantee trade, formalized
A real, verified stale-price bug from a missing invalidation trigger	Software Architecture Fundamentals Chapter 6's own eventual-consistency finding — both are the same underlying risk (data that's technically present but no longer true) surfacing in different layers
Cache-aside's own 19,566× read speedup	Software Architecture Fundamentals Chapter 7's own ~18,111× testability speedup — a strikingly similar-shaped number, from the identical underlying idea: avoid real, slow I/O whenever it's safe to

Hands-On Exercises

EXERCISE 1

Extend this chapter's own `CacheAsideRepository` with a second key, `'PROD-2'`, and verify that a cache hit on `'PROD-1'` doesn't accidentally also count as a hit for `'PROD-2'` — confirm the first read of `'PROD-2'` still triggers a genuine database query even after `'PROD-1'` is fully cached.

View solution

EXERCISE 2

Using this chapter's own `WriteBehindCache`, write two values for two different keys before ever calling `flush()`, then simulate a crash. Verify both writes are lost from the database, and verify calling `flush()` afterward (simulating a recovery that never happened) can't bring back data that was never in `pending_writes` to begin with.

View solution

EXERCISE 3

This chapter's own `CachedPricingService.invalidate()` requires something else in the system to remember to call it whenever stock changes. Using this chapter's own verified stale-price bug, explain specifically what would have to change elsewhere in a real system (not in the cache itself) to make that invalidation call actually happen reliably.

View solution

CHAPTER 3 QUICK REFERENCE

- **Cache-aside:** check cache, miss falls through to the database and populates the cache — verified: ~19,566× faster on repeated reads, with `db.query_count` never incrementing past the first miss
- **Write-through:** writes hit cache and database together, synchronously — always consistent, always pays the full write cost
- **Write-behind:** writes hit cache immediately, database later — verified ~12,145× faster, verified genuinely losing data on a simulated crash before `flush()`
- **Invalidation, verified as the hard part:** a real, wrong price (`100` instead of the correct `115.0`) was served with zero errors until the cache was explicitly told to forget — the caching mechanism itself was never the problem
- **Next chapter:** Database Scaling — replication, read replicas, and sharding

CHAPTER 4 OF 10

Database Scaling

Distributed Systems & Scalability

Chapter 4 · Database Scaling

A database can scale reads by copying itself (replication) or scale everything by splitting itself (sharding). Both genuinely work — and both cost something specific and measurable, not just "some consistency" in the abstract.

Read Replicas: Real Load Relief, and a Real Staleness Window

```
class PrimaryDatabase: def __init__(self): self.data = {}; self.write_log = [] def write(self, key, value): self.data[key] = value self.write_log.append((key, value)) def replicate_to(self, replica): for key, value in self.write_log: replica.data[key] = value self.write_log.clear()
```

VERIFIED DIRECTLY — A REPLICA GENUINELY DOESN'T HAVE A WRITE UNTIL REPLICATION ACTUALLY RUNS

Writing `{'price': 100}` to the primary makes it immediately available from `primary.data`. Reading the identical key from the replica *before* `replicate_to()` runs returns `None` — genuinely missing, not just slow. Only after `replicate_to()` actually runs does the replica correctly return `{'price': 100}`.

VERIFIED DIRECTLY — REPLICAS REDUCE PER-SERVER LOAD, NOT THE TIME ANY SINGLE QUERY TAKES

Running 30 reads entirely against one primary took the same total wall-clock time as spreading the identical 30 reads across 3 replicas — each individual read still costs what it costs. What genuinely changed: the primary alone would have handled all **30** reads; with 3 replicas sharing the work, each server handled only **10** — **3×** less load per server. This is Chapter 1's own "capacity, not speed" distinction, applied to reads specifically.

WHY THIS MATTERS FOR WHAT YOU CAN SAFELY READ FROM A REPLICA

Any read that needs the absolute latest write — checking a payment that was just submitted, confirming an order that was just placed — can genuinely fail if it's routed to a replica during the staleness window just verified. Reads that can tolerate being slightly out of date (a product catalog, a dashboard) are exactly what read replicas are for.

Sharding: Splitting Data, and What a Bad Key Does to It

```
def shard_by_user_id(user_id): return user_id % NUM_SHARDS # GOOD: evenly-distributed integers def shard_by_country(country): # BAD: real-world signup data is rarely even h = int(hashlib.md5(country.encode()).hexdigest(), 16) return h % NUM_SHARDS
```

VERIFIED DIRECTLY — A WELL-CHOSEN KEY SPREADS 1,000 RECORDS PERFECTLY EVENLY

Sharding 1,000 sequential user IDs across 4 shards by `user_id % 4` produced exactly **250 records per shard** — a spread of `0` between the busiest and quietest shard.

VERIFIED DIRECTLY — A PLAUSIBLE, REALISTIC KEY PRODUCED A SEVERE HOTSPOT, AND TWO ENTIRE SHARDS SAT EMPTY

Sharding the same 1,000 users by their own signup country — a realistic distribution where 80% of users share one primary market — produced **863 records on one shard, 137 on another, and `0` on each of the remaining two**. One shard alone carried **86.3%** of all data. With only 4 distinct country values feeding the hash function, two of the four available shards never received a single record at all — sharding infrastructure was fully deployed, but most of it was doing nothing.

WHY THIS CONNECTS DIRECTLY TO CHAPTER 2'S OWN LOAD-BALANCING FINDINGS

An overloaded shard is the database-layer version of Chapter 2's own overloaded server under Round Robin — the fix isn't "add more shards," it's choosing a key that actually reflects how the data is genuinely distributed, the same way Least Connections' fix wasn't "add more servers," it was routing based on real, current load.

The Cost Sharding Adds: Cross-Shard Queries

VERIFIED DIRECTLY — AN AGGREGATE QUERY ACROSS ALL USERS COSTS 4× THE SHARD-TOUCHES OF A SINGLE-SHARD QUERY

Computing total revenue for users on *one* shard touched exactly **1** shard. Computing total revenue across *all* 1,000 users touched all **4** shards, one query each, before the results could even be combined — correctly summing to `100,000`, matching a manual check (`1,000 × $100`) exactly. A question that was one query against an unsharded database becomes `N` queries plus a combine step, where `N` is however many shards exist.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
A verified replica staleness window	Chapter 3's own cache-invalidation bug and Software Architecture Fundamentals Chapter 6's own eventual-consistency finding — the identical shape of risk, at the database layer this time
A dramatic, realistic sharding hotspot (86.3% of data on one	Chapter 2's own Round Robin finding — an "equal treatment" rule (hash the key, mod by shard count) producing a badly uneven outcome

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
shard)	
Cross-shard queries costing N shard-touches instead of 1	Chapter 5's own CAP Theorem chapter — sharding is a concrete instance of trading single-query simplicity for horizontal capacity

Hands-On Exercises

EXERCISE 1

Using this chapter's own `PrimaryDatabase` / `ReadReplica`, write TWO keys to the primary before calling `replicate_to()`. Verify both are missing from the replica beforehand, and both are correctly present afterward.

View solution

EXERCISE 2

Using this chapter's own sharding setup, try `shard_by_country` with a LESS skewed distribution — say, 30/25/25/20 percent across the same 4 countries instead of 80/7/7/6. Verify whether the hotspot shrinks, and report the new spread compared to this chapter's own 726-record spread.

View solution

EXERCISE 3

Using this chapter's own two verified findings (replica staleness and the sharding hotspot), explain why "add read replicas" and "shard the database" solve genuinely different scaling problems — which one would have helped Chapter 1's own O(n) duplicate-check bottleneck, and why?

View solution

CHAPTER 4 QUICK REFERENCE

- **Read replicas:** verified reducing per-server load 3× (30 reads → 10 per server across 3 replicas) with zero change to any single query's own speed, at the cost of a real, verified staleness window before replication runs
- **Sharding, verified:** a good key (user_id) spread 1,000 records perfectly evenly; a plausible but poor key (signup country) put 86.3% of records on one shard and left two of four shards completely empty
- **Cross-shard queries, verified:** an aggregate across all data cost 4× the shard-touches of a single-shard query

- **Next chapter:** The CAP Theorem & Consistency Models — formalizing the exact tradeoffs this chapter and Chapter 3 both verified concretely

CHAPTER 5 OF 10

The CAP Theorem & Consistency Models

Distributed Systems & Scalability

Chapter 5 · The CAP Theorem & Consistency Models

"Pick two of Consistency, Availability, and Partition tolerance" is the version of CAP most people hear — and it's misleading. Partition tolerance isn't optional in a real distributed system; network partitions happen whether you design for them or not. The actual theorem is narrower and sharper: **when a partition is actually happening, a system must choose between Consistency and Availability** — and, this chapter verifies, that choice doesn't matter at all until a partition actually occurs.

No Partition: Both Choices Look Identical

```
class CPSystem: # sacrifices Availability during a partition, keeps Consistency def write(self, node, key, value): if self.partitioned and node is self.node_b: raise ConnectionError('Write rejected: partitioned, cannot guarantee consistency') node.data[key] = value if not self.partitioned: self.node_a.data[key] = value; self.node_b.data[key] = value class APSystem: # sacrifices Consistency during a partition, keeps Availability def write(self, node, key, value): node.data[key] = value # always succeeds, regardless of partition if not self.partitioned: self.node_a.data[key] = value; self.node_b.data[key] = value
```

VERIFIED DIRECTLY — WITH NO PARTITION, BOTH SYSTEMS ARE EQUALLY CONSISTENT AND EQUALLY AVAILABLE

Writing `stock=50` to either `CPSystem` or `APSystem` while `partitioned` is `False` produces the identical result both times: `node_a.data == node_b.data` is `True` for both. No request was ever rejected in either system. There is no observable difference between "a system designed to prioritize consistency" and "a system designed to prioritize availability" — until something actually goes wrong.

A Real Partition: The Tradeoff Becomes Real

VERIFIED DIRECTLY — CP REJECTS THE WRITE; THE SYSTEM STAYS CONSISTENT BUT GENUINELY UNAVAILABLE

Setting `partitioned = True` and attempting to write `stock=30` to `node_b` (the isolated node) through `CPSystem` correctly raises `ConnectionError: Write rejected: partitioned, cannot guarantee consistency`. `node_b.data` stays exactly as it was — no write happened, no risk of disagreement, and a real customer request genuinely failed.

VERIFIED DIRECTLY — AP ACCEPTS BOTH WRITES; THE SYSTEM STAYS AVAILABLE BUT GENUINELY DIVERGES

The identical partition, but through `APSystem`: writing `stock=45` to `node_a` (representing a real sale processed on that side) and, separately, `stock=42` to `node_b` (a *different* real sale, processed on the isolated side) both **succeed**. `node_a.data` now shows `{'stock': 45}`; `node_b.data` shows `{'stock': 42}` — **genuinely disagreeing**, confirmed directly (`node_a.data != node_b.data` → `True`).

The Cost of Reconciling an AP System's Own Conflict

Once the partition heals, *something* has to decide which of the two conflicting values is "correct." A common, simple strategy: last-write-wins, by timestamp.

VERIFIED DIRECTLY — LAST-WRITE-WINS DOESN'T RECOVER THE TRUTH, IT DISCARDS IT

`node_b`'s write (`stock=42`) happened microseconds after `node_a`'s (`stock=45`), so last-write-wins correctly picks `node_b`'s value — the reconciled stock becomes `42`. But both writes represented *real* sales that genuinely happened: if both should have counted, the true combined stock is `50 - 5 - 8 = 37`. The reconciled value (`42`) is off from the true combined value (`37`) by a real, unrecoverable **5 units** — `node_a`'s own sale wasn't merged with `node_b`'s, it was silently overwritten and lost.

THIS IS WHAT "EVENTUAL CONSISTENCY" ACTUALLY COSTS

"Eventually consistent" doesn't mean "eventually correct" — it means the system eventually agrees on *some* single value, with no guarantee that value reflects everything that genuinely happened during the partition. Chapter 3's own cache staleness and Chapter 4's own replication lag both showed *temporary* disagreement that later resolved cleanly. This chapter's own AP finding is a sharper case: the disagreement resolves, but the resolution can permanently lose real information.

Strong vs. Eventual Consistency, Named Precisely

MODEL	GUARANTEE	ALREADY VERIFIED IN THIS COURSE AS
Strong consistency	Every read reflects the most recent write, always	`CPSystem`'s own behavior — verified here rejecting availability to keep this guarantee
Eventual consistency	Reads may be temporarily stale, but converge given enough time	Chapter 3's cache-invalidation bug; Chapter 4's replica staleness window; Software Architecture Fundamentals Chapter 6's own event-processing gap

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
No observable difference between CP and AP without a partition	Chapter 9's own resilience-pattern chapter — designing for a partition that hasn't happened yet is exactly the discipline that chapter covers
AP's own verified information-loss on reconciliation	Chapter 4's own sharding chapter — a cross-shard write conflict is a structurally identical problem, one layer over
Write-behind's own verified crash-loss finding, revisited here as a genuine AP-style tradeoff	Chapter 3's own caching chapter — write-behind is, in CAP terms, an availability-favoring choice made at the cache layer specifically

Hands-On Exercises

EXERCISE 1

Using this chapter's own `CPSystem`, verify that writes to `node_a` (the non-isolated node) still succeed correctly *during* a partition, even though writes to `node_b` are rejected. Explain what this confirms about which specific node a CP system sacrifices availability for.

View solution

EXERCISE 2

Using this chapter's own reconciliation scenario, implement a different strategy — "highest value wins" instead of last-write-wins — and verify what final stock value and what discrepancy from the true combined total (37) it produces.

View solution

EXERCISE 3

Using this chapter's own verified findings, explain why a real production system's own choice of CP or AP would most plausibly be decided per-operation (some writes CP, others AP) rather than as one single, system-wide setting — use this chapter's own stock-tracking scenario alongside a login/authentication scenario to make the contrast concrete.

View solution

CHAPTER 5 QUICK REFERENCE

- **CAP, precisely:** the Consistency-vs-Availability tradeoff only exists *during* an actual network partition — verified: CP and AP behaved identically with no partition, both perfectly consistent and perfectly available
- **During a partition, verified:** CP rejected a write outright (unavailable, still consistent); AP accepted two conflicting writes (available, now genuinely inconsistent — `45` vs. `42`)

- **Reconciliation cost, verified:** last-write-wins produced `42` against a true combined value of `37` — a real, permanent 5-unit loss of information, not just a delay
- **Next chapter:** Message Queues & Asynchronous Processing — the mechanism most real systems actually use to manage exactly this kind of tradeoff deliberately

CHAPTER 6 OF 10

Message Queues & Asynchronous Processing

Distributed Systems & Scalability

Chapter 6 · Message Queues & Asynchronous Processing

Software Architecture Fundamentals Chapter 6 built an `EventBus` that dispatched to subscribers the instant `publish()` was called — decoupling *who* knows about whom, but not *when* the work happens. A message queue decouples both. This chapter verifies what that second kind of decoupling actually buys, and what it costs.

Real Buffering: Decoupling Timing, Not Just Knowledge

```
class MessageQueue: def __init__(self): self.messages = [] def publish(self, message): self.messages.append(message) def consume_all(self, handler): while self.messages: message = self.messages.pop(0) handler(message)
```

VERIFIED DIRECTLY — PUBLISHING WORKS WITH NO CONSUMER RUNNING AT ALL

Publishing **5** messages with no call to `consume_all()` anywhere yet leaves all 5 correctly sitting in `queue.messages`, genuinely unprocessed. Starting the consumer afterward correctly processes all 5, in the exact order they were published, and drains the queue to empty. Compare this to Software Architecture Fundamentals' own `EventBus.publish()`, which had no way to "publish" without a subscriber already attached to react immediately — this queue lets production and consumption happen on *completely independent* schedules.

At-Least-Once Delivery: Why Idempotency Isn't Optional

A message that's processed but never acknowledged — because the consumer crashed in between — gets redelivered. What happens depends entirely on whether the handler can safely run twice.

VERIFIED DIRECTLY — A NON-IDEMPOTENT HANDLER PROCESSES THE SAME MESSAGE TWICE, AND THE BUG IS REAL

Publishing one message worth `10` points, then simulating a crash *after* processing but *before* acknowledging it: the handler already ran once (`points = 10`), but the message stays in the queue since it was never acked. Redelivery correctly re-triggers the handler — and `points` becomes `20`, not the correct `10`. The same real award was granted twice for one real event.

VERIFIED DIRECTLY — AN IDEMPOTENT HANDLER PROCESSES THE IDENTICAL REDELIVERY SAFELY

The identical crash-then-redeliver scenario, but the handler checks a `processed_ids` set before doing anything: `points` correctly stays at `10` after both the crashed first attempt and the successful redelivery. The message was still delivered twice — at-least-once delivery didn't change — but the handler's own idempotency absorbed the duplicate safely.

"AT-LEAST-ONCE" IS A PROMISE ABOUT DELIVERY, NOT ABOUT CORRECTNESS

A message queue guaranteeing at-least-once delivery is explicitly promising a message *might* arrive more than once — it is not promising your handler will behave correctly when that happens. The `20`-instead-of-`10` bug just verified is not a queue malfunction; the queue did exactly what "at-least-once" means. The correctness burden sits entirely with the consumer.

Dead-Letter Queues: What Happens When a Message Can't Be Processed

VERIFIED DIRECTLY — AN ALWAYS-FAILING MESSAGE GETS RETRIED, THEN MOVED ASIDE, NOT RETRIED FOREVER

A message that raises an exception *every* time it's processed, against a queue configured with `max_retries=3`: the first two attempts return `'requeued'`, the third returns `'dead-lettered'` — the message moves into `dead_letter_queue`, carrying the actual error message (`'malformed payload: missing required field'`), and stops consuming further processing attempts entirely.

VERIFIED DIRECTLY — A TRANSIENT FAILURE GETS A REAL CHANCE TO RECOVER, AND NEVER GETS DEAD-LETTERED

A handler that fails on its first two attempts but succeeds on the third: the outcomes were `['requeued', 'requeued', 'success']` — `dead_letter_queue` stayed empty. The retry mechanism gave the same message multiple genuine chances, and a transient problem (the kind Chapter 9's own resilience-pattern chapter covers in depth) resolved itself without ever needing manual intervention.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
Genuine timing decoupling, verified against Software Architecture Fundamentals' synchronous `EventBus`	Software Architecture Fundamentals Chapter 6's own eventual-consistency window — a message queue is the deliberate, controlled version of that same gap
A real, verified double-processing bug from non-idempotent handling	Chapter 5's own AP reconciliation finding — both are "the same real event counted twice," at different layers
Dead-lettering only after genuine retries, verified not triggering on a recoverable transient failure	Chapter 9's own Fault Tolerance & Resilience Patterns — retry-with-backoff and dead-letter queues are close cousins, covered together there

Hands-On Exercises

EXERCISE 1

Using this chapter's own `MessageQueue`, publish 3 messages, consume just 1 of them (call `consume_all` won't work here — write a small loop that pops and handles only 1), then publish 2 more messages before finally consuming the rest. Verify the final processing order is correct.

View solution

EXERCISE 2

Using this chapter's own `AtLeastOnceQueue` and idempotent handler pattern, simulate the message crashing *twice* before finally being acknowledged successfully (three total delivery attempts). Verify `points` still correctly ends at `10`, not `30`.

View solution

EXERCISE 3

Using this chapter's own two verified findings (the double-processing bug and the dead-letter queue), explain why a message ending up in the dead-letter queue is *not* a case at-least-once delivery's own idempotency requirement can help with — what's the actual difference in what's failing?

View solution

CHAPTER 6 QUICK REFERENCE

- **Message queues:** buffer messages between producer and consumer — verified: 5 messages queued and stayed unprocessed with zero consumers running, then processed correctly, in order, once one started
- **At-least-once, verified:** a non-idempotent handler double-counted a redelivered message (`10 → 20`); an idempotent one, tracking processed IDs, correctly stayed at `10`
- **Dead-letter queues, verified:** an always-failing message was retried twice then moved aside on the third attempt; a transient failure recovered on its own within the same retry budget and was never dead-lettered
- **Next chapter:** Rate Limiting & Throttling — protecting a system's own downstream services from being overwhelmed in the first place

CHAPTER 7 OF 10

Rate Limiting & Throttling

Distributed Systems & Scalability

Chapter 7 · Rate Limiting & Throttling

Chapter 1 measured what happens when load exceeds what a system can handle. Rate limiting is the deliberate decision to reject some requests *before* that happens, protecting whatever sits downstream. Three algorithms answer "how do I count requests fairly" in genuinely different ways — and one classic implementation has a real, verified bug most people don't expect.

Token Bucket: Allows a Burst, Then Throttles

```
class TokenBucket: def __init__(self, capacity, refill_rate_per_sec): self.capacity = capacity; self.tokens = capacity # starts FULL self.refill_rate = refill_rate_per_sec; self.last_refill = time.time() def allow_request(self): now = time.time() self.tokens = min(self.capacity, self.tokens + (now - self.last_refill) * self.refill_rate) self.last_refill = now if self.tokens >= 1: self.tokens -= 1; return True return False
```

VERIFIED DIRECTLY — EXACTLY THE BUCKET'S OWN CAPACITY GETS THROUGH IMMEDIATELY, THEN IT STOPS

Against a bucket with capacity `10` (refilling at `2`/sec): 11 requests fired instantly all return the exact expected pattern — the first **10** all succeed, and the **11th** is correctly rejected, since the bucket started full and every request drains one token. Waiting `0.6s` (refilling roughly `1.2` tokens) makes the next request correctly succeed again.

Leaky Bucket: Smooths a Burst Into a Steady Output Rate

```
class LeakyBucket: def __init__(self, capacity): self.capacity = capacity; self.queue = [] def add_request(self, request_id): if len(self.queue) >= self.capacity: return False self.queue.append(request_id); return True def leak(self): # releases exactly one, at whatever pace this is called return self.queue.pop(0) if self.queue else None
```

VERIFIED DIRECTLY — THE BURST GETS QUEUED AND RATE-LIMITED ON THE WAY OUT, NOT REJECTED OUTRIGHT

Submitting **20** requests instantly to a leaky bucket with capacity **15**: **15** are accepted (queued), **5** are rejected outright (the bucket was already full). Calling `leak()` fifteen times released every accepted request in the *exact order it was*

originally submitted — `['req-0', 'req-1', ..., 'req-14']`, confirmed matching the submission order exactly. Downstream code calling `leak()` only ever sees one request at a time, however bursty the input was.

THE GENUINE DIFFERENCE BETWEEN THE TWO

Token bucket lets an entire allowed burst reach the downstream system *immediately* — verified: all 10 of the first requests succeeded in one instant. Leaky bucket accepts a burst but releases it downstream at a controlled, steady pace — verified: 15 requests queued instantly, released one call to `leak()` at a time. Choose token bucket when occasional bursts are genuinely fine; choose leaky bucket when downstream specifically needs a smooth, predictable rate — for instance, exactly the kind of steady processing rate a queue consumer (Chapter 6) or a database write path (Chapter 4) benefits from.

Fixed Window vs. Sliding Window: A Real Boundary Bug

A fixed window counts requests within calendar-aligned buckets (e.g., minute 0–60, 60–120) and resets the count at each boundary. What happens to traffic that straddles that boundary?

VERIFIED DIRECTLY — A FIXED WINDOW LETS EXACTLY DOUBLE THE INTENDED RATE THROUGH, RIGHT AT THE BOUNDARY

With a limit of **10 requests per 60 seconds**: sending 10 requests between `t=55` and `t=59.5` (the tail end of one window) correctly allows all **10**. Sending 10 *more* requests between `t=60.5` and `t=65` — barely **5.5 to 10.5 seconds later**, but now in the "next" window — **also** allows all **10**. Total: **20** requests genuinely allowed within a single 10-second span, against a limit meant to cap traffic at 10 per full 60 seconds.

VERIFIED DIRECTLY — A SLIDING WINDOW, GIVEN THE IDENTICAL TRAFFIC, CORRECTLY CLOSES THE GAP

Running the exact same request timing against a sliding window (which counts requests within the trailing 60 seconds from *right now*, not a fixed calendar bucket): the first batch of 10 is allowed identically. The second batch — arriving only 5.5–10.5 seconds later, well within the same rolling 60-second window as the first batch — is correctly rejected: `0` **of 10** allowed. Total allowed across the identical 10-second span: `10`, matching the intended limit exactly.

WHY THIS BUG IS EASY TO MISS

A fixed window limiter passes every simple, isolated test — send 10 requests, the 11th within the same window correctly fails. The bug only appears at exactly the boundary, and only when traffic is deliberately (or accidentally) clustered around it — which is exactly the shape of traffic a real abuser, or a genuinely bursty legitimate client, produces.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
A verified, doubled-rate fixed-window bug	Chapter 1's own O(n) bottleneck finding — both are bugs invisible in a simple test, only surfacing under a specific, real traffic shape
Leaky bucket's own steady, one-at-a-time release rate	Chapter 6's own message queue — `leak()` is structurally the same operation as a queue consumer processing one message at a time
Rate limiting protecting a downstream system before it's overwhelmed	Chapter 9's own Fault Tolerance & Resilience Patterns — rate limiting is a preventative resilience pattern, applied before a failure rather than reacting to one

Hands-On Exercises

EXERCISE 1

Using this chapter's own `TokenBucket`, verify what happens with a capacity of `5` and a much higher refill rate (`10`/sec) — send 5 immediate requests, then wait 0.3s and send 5 more. Report how many of the second batch succeed and explain why using this chapter's own refill formula.

View solution

EXERCISE 2

Using this chapter's own `LeakyBucket`, submit 10 requests, leak out 3, then submit 8 more before leaking the rest. Verify the bucket correctly rejects any requests that would exceed its own capacity at the moment they're submitted, and verify the final leak order is still fully FIFO across both submission batches.

View solution

EXERCISE 3

Using this chapter's own fixed-window and sliding-window results, construct a traffic pattern that does *not* straddle a window boundary (e.g., all 20 requests sent between `t=10` and `t=20`, well inside one window). Verify both limiters now agree on how many requests are allowed, and explain why the boundary bug specifically requires boundary-straddling traffic to appear at all.

View solution

CHAPTER 7 QUICK REFERENCE

- **Token bucket:** allows a burst up to capacity, then throttles — verified: exactly 10 of 11 immediate requests succeeded against a 10-token bucket

- **Leaky bucket:** queues a burst, releases it downstream at a steady rate — verified: 15 of 20 burst requests accepted, released one at a time in exact FIFO order
- **Fixed window, verified buggy:** let 20 requests through in a 10-second span against a 10-per-60-second limit, purely from boundary timing
- **Sliding window, verified correct:** capped the identical traffic at exactly 10 in the same 10-second span
- **Next chapter:** API Gateways & Service Discovery — where rate limiting and routing typically get enforced in a real microservices system

CHAPTER 8 OF 10

API Gateways & Service Discovery

Distributed Systems & Scalability

Chapter 8 · API Gateways & Service Discovery

Software Architecture Fundamentals Chapter 4 measured a real, non-zero cost for every network call. This chapter shows the two standard fixes for a microservices system that's split into enough pieces for that cost to actually hurt: an **API Gateway** that reduces how many round trips a client pays for, and a **service registry** that lets services find each other without hardcoding addresses that will eventually go stale.

API Gateway: Aggregating Three Calls Into One Round Trip

```
class ApiGateway: def __init__(self, order_service, inventory_service, review_service): self.order_service = order_service self.inventory_service = inventory_service self.review_service = review_service def get_order_page(self, order_id, product_id): order = self.order_service.get_order_summary(order_id) stock = self.inventory_service.get_stock_info(product_id) reviews = self.review_service.get_reviews(product_id) return {'order': order, 'stock': stock, 'reviews': reviews}
```

VERIFIED DIRECTLY — THE CLIENT IS GENUINELY FASTER, NOT JUST MAKING FEWER CALLS IN THE ABSTRACT

A client on a slow external connection (simulated at `50ms` per round trip) calling three services directly — order, inventory, reviews — pays that `50ms` cost **three separate times**: **166.9ms** total. The identical client, calling one `ApiGateway` endpoint instead, pays the `50ms` external cost **once**, with the gateway making the same three calls internally over a fast internal network (`5ms` each): **66.0ms** total — **2.53× faster**. Both approaches return *identical* data.

WHY THIS ISN'T JUST "FEWER FUNCTION CALLS"

The speedup here comes specifically from moving the multi-call fan-out from a slow, external network hop to a fast, internal one — the gateway still makes three calls, it just makes them from somewhere the network cost is small. A gateway that fanned out over the *same* slow network the client was on wouldn't help at all; this is the same "where does the cost actually live" question Software Architecture Fundamentals Chapter 4 first measured.

Service Discovery: Finding a Service That Doesn't Stay in One Place

```
class ServiceRegistry: def __init__(self): self.services = {} def register(self, name, address): self.services[name] = address def discover(self, name): return self.services.get(name)
```

VERIFIED DIRECTLY — A HARDCODED ADDRESS SILENTLY BREAKS AFTER A REAL REDEPLOY; A DISCOVERY-BASED LOOKUP DOESN'T

`InventoryService` registers at `10.0.0.5:8080`. A **hardcoded** client captures that address once, at startup. The service then genuinely redeploys — a real scaling or restart event — and re-registers at a *new* address, `10.0.0.9:8080`; only that new address is actually listening now. The hardcoded client's own call, using its stale captured address, correctly returns `False` — it would silently fail against a service that no longer exists there. A client using **service discovery** — looking up the current address fresh, at the moment of the call — correctly finds `10.0.0.9:8080` and returns `True`.

WHY THIS CONNECTS DIRECTLY TO CHAPTER 1'S OWN HORIZONTAL SCALING

Chapter 1 verified genuine scaling means adding or replacing running instances — exactly the kind of event that changes a service's own network address. A hardcoded address isn't just fragile in theory; it's specifically incompatible with the scaling behavior this course has spent seven chapters establishing as necessary. Service discovery is what makes horizontal scaling and service discovery compatible with each other at all.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
A verified 2.53× client-side speedup from aggregating three calls into one	Software Architecture Fundamentals Chapter 4's own ~11,661× network-call overhead finding — this chapter's gateway is a direct, concrete application of minimizing exactly that cost
A hardcoded address silently breaking after a real redeploy	Chapter 2's own health-checking finding — both are examples of a system correctly routing around a change, versus one that doesn't notice at all
Service discovery as the missing piece behind horizontal scaling	Chapter 1's own scaling findings — this chapter closes the gap between "you can add more instances" and "the rest of the system can actually find them"

Hands-On Exercises

EXERCISE 1

Add a fourth service, `ShippingService`, to this chapter's own `ApiGateway` (with the same simulated `5ms` internal latency), and measure the new total time with and without the gateway. Verify the gap between the two approaches widens as more services are aggregated.

View solution

EXERCISE 2

Using this chapter's own `ServiceRegistry`, simulate a *second* redeploy (a third address) happening after the first. Verify a discovery-based client still correctly finds the newest address, and verify the hardcoded client (still holding its original, very first captured address) remains broken exactly as before.

View solution

EXERCISE 3

Using this chapter's own two verified findings, explain why an API gateway that fans out to three services using hardcoded addresses would eventually break in the exact way this chapter's own hardcoded client did — and why a real gateway needs service discovery internally, not just aggregation.

View solution

CHAPTER 8 QUICK REFERENCE

- **API Gateway:** aggregates multiple backend calls into one client-facing round trip — verified: `2.53×` faster from the client's own perspective (`166.9ms` → `66.0ms`), identical results
- **Service registry:** lets services be found by name instead of hardcoded address — verified: a hardcoded client silently broke after a real redeploy; a discovery-based client correctly kept working
- **The connection to Chapter 1:** service discovery is what makes horizontal scaling's own instance churn survivable for the rest of the system
- **Next chapter:** Fault Tolerance & Resilience Patterns — circuit breakers, retries with backoff, and graceful degradation

CHAPTER 9 OF 10

Fault Tolerance & Resilience Patterns

Distributed Systems & Scalability

Chapter 9 · Fault Tolerance & Resilience Patterns

Software Architecture Fundamentals Chapter 4 measured a downstream slowdown cascading straight up through a chain of synchronous calls. Technical Support's own `perfdiag1` / `incident1` courses spend whole chapters diagnosing exactly this kind of failure after the fact. This chapter builds the four patterns that stop it from reaching that point at all — and verifies what each one actually buys.

Circuit Breakers: Fail Fast Instead of Paying the Full Cost Every Time

```
class CircuitBreaker: def call(self, func): if self.state == 'open': if time.time() - self.opened_at >= self.reset_timeout: self.state = 'half-open' else: raise RuntimeError('Circuit open - failing fast') try: result = func() if self.state == 'half-open': self.state = 'closed'; self.failure_count = 0 return result except Exception: self.failure_count += 1 if self.failure_count >= self.failure_threshold: self.state = 'open'; self.opened_at = time.time() raise
```

VERIFIED DIRECTLY — THE BREAKER SAVES REAL, MEASURED TIME ONCE IT OPENS

Against a downstream call that always fails after a simulated `0.1s` timeout: 10 calls with **no** circuit breaker took **1,003.5ms** total — every single call paid the full timeout cost. The identical 10 calls through a breaker (threshold `3`) took **301.1ms** — only the first **3** calls paid the full `0.1s` cost; the remaining **7** failed *instantly* once the circuit opened. **3.3×** less total time wasted, for the identical, genuinely-failed outcome.

Retries With Backoff: Giving a Struggling Service Room to Recover

VERIFIED DIRECTLY — A NAIVE RETRY IS A BURST; EXPONENTIAL BACKOFF GENUINELY SPREADS THE LOAD

Five retry attempts against a service that always fails: a **naive** immediate retry (no delay between attempts) completed all 5 attempts within **0.01ms** — effectively a single instantaneous burst hitting the struggling service five times in a row. The identical 5 attempts with **exponential backoff** (`0.05 × 2ⁿ` seconds between tries) spread across **751.16ms** — roughly **117,000×** more real time between the first and last attempt, for the same 5 total tries.

WHY "JUST RETRY" CAN MAKE AN INCIDENT WORSE

A struggling service under real load doesn't need five more requests arriving in the same millisecond — that's Chapter 1's own overload finding, self-inflicted. Backoff doesn't reduce how many times a client tries; it changes *when* those tries land, giving whatever's struggling downstream genuine breathing room between attempts instead of compounding the problem that caused the failure in the first place.

Bulkheads: Isolating Resources So One Failure Doesn't Starve Everything

VERIFIED DIRECTLY — A SHARED RESOURCE POOL LETS A NON-CRITICAL DEPENDENCY BLOCK A CRITICAL ONE

A shared connection pool (capacity `5`): a slow, non-critical recommendations service acquires all `5` connections and holds them. A critical checkout request, using the *same* pool, then tries to acquire a connection and correctly gets `False` — blocked, purely because an unrelated, lower-priority feature exhausted a resource checkout also needed.

VERIFIED DIRECTLY — ISOLATED POOLS KEEP THE CRITICAL PATH AVAILABLE

The identical scenario, but with recommendations and checkout each given their *own* pool (`3` and `2` connections respectively): recommendations still exhausts its own pool completely (a 4th request correctly returns `False`) — but checkout's own separate pool is untouched, and its request correctly returns `True`. The critical path stayed available specifically because it was never sharing a resource with the failing one.

Graceful Degradation: A Usable Response, Not a Failed One

VERIFIED DIRECTLY — A RESILIENT GATEWAY RETURNS A USABLE PARTIAL RESPONSE WHEN ONE DEPENDENCY IS DOWN

Reusing Chapter 8's own aggregating gateway, with `ReviewService` genuinely failing: a **resilient** version, catching that specific failure and substituting a fallback (`{'reviews': [], 'note': 'reviews temporarily unavailable'}`), correctly returns the full order and stock data *plus* the fallback — a real, usable response. A **brittle** version, with no such handling, correctly fails the *entire* request — order and stock data included, even though both of those were computed successfully before the review call ever failed.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
A circuit breaker saving 3.3× total time on a genuinely failing call chain	Software Architecture Fundamentals Chapter 4's own cascading-call finding — this is the direct, concrete fix for the exact failure that chapter measured
A verified retry-storm vs. spread-out backoff, in real measured milliseconds	Chapter 7's own rate-limiting chapter — both are ways of controlling request density against a downstream system, one from the caller's side, one from the receiver's

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
A brittle gateway discarding two successful results because a third call failed	Technical Support's own `incident1`/`perfdiag1` — this exact pattern (one failed dependency taking down an otherwise-healthy response) is precisely the class of ticket those courses' own diagnostic chapters are built to trace back to its cause

Hands-On Exercises

EXERCISE 1

Using this chapter's own `CircuitBreaker`, lower `failure_threshold` to `1` and re-run the 10-call scenario. Verify the total time drops further than this chapter's own `301.1ms` result, and explain the tradeoff a threshold of 1 introduces that a threshold of 3 avoids.

View solution

EXERCISE 2

Using this chapter's own bulkhead scenario, give recommendations a pool of `4` instead of `3` (checkout stays at `2`, for a combined total of `6` — one more than the original shared pool's own capacity of `5`). Verify checkout still succeeds regardless of how large recommendations' own pool is.

View solution

EXERCISE 3

Using this chapter's own four verified patterns, explain which ONE of them would have been the most direct fix for Software Architecture Fundamentals Chapter 4's own original cascading-call scenario (service A waiting 300ms because service C was slow), and why the other three, while genuinely useful in general, wouldn't have addressed that specific measured problem.

View solution

CHAPTER 9 QUICK REFERENCE

- **Circuit breaker:** stops calling a service that's genuinely failing — verified: `3.3×` less total time wasted, `7` of `10` calls failing instantly instead of paying a full timeout
- **Retries with backoff:** spaces out retries instead of bursting them — verified: `751ms` spread vs. a naive retry's `0.01ms` burst for the same 5 attempts
- **Bulkheads:** isolate resource pools per dependency — verified: a shared pool let one feature block another; separate pools kept the critical path available

- **Graceful degradation:** return a usable partial response instead of failing everything — verified: a resilient gateway kept two working results when a third dependency failed; a brittle one discarded all three
- **Next chapter:** Capstone — designing a real system at scale, applying every chapter in this course together

CHAPTER 10 OF 10

Capstone — Designing a System at Scale

Distributed Systems & Scalability

Chapter 10 · Capstone: Designing a System at Scale

One continuous worked project: a URL shortener, assembling every verified pattern from Chapters 1–9 into a single running system — sharded storage, a cache-aside redirect path, an async click-tracking queue, rate-limited creation, and a circuit breaker protecting the critical path from a broken analytics pipeline. Every number below comes from actually running the assembled system, not from restating each chapter's own earlier findings.

Assembling the System

```
class UrlShortener: def __init__(self): self.store = ShardedUrlStore(NUM_SHARDS) # Chapter 4 self.redirect_service = CacheAsideRedirectService(self.store) # Chapter 3 self.creation_limiter = TokenBucket(capacity=5, refill_rate_per_sec=1) # Chapter 7 self.click_queue = ClickEventQueue() # Chapter 6 self.click_counts = {} def create_short_url(self, long_url): if not self.creation_limiter.allow_request(): raise RuntimeError('Rate limit exceeded') code = hashlib.md5(long_url.encode()).hexdigest()[:6] self.store.save(code, long_url) return code def redirect(self, short_code): long_url = self.redirect_service.resolve(short_code) self.click_queue.publish(short_code) # fire-and-forget, never blocks the redirect return long_url
```

Running the System End to End

VERIFIED DIRECTLY — A SHORT CODE IS CREATED AND CORRECTLY STORED IN THE RIGHT SHARD

`create_short_url('https://example.com/a-very-long-article-url')` returns short code `'c69915'`, and `store.get('c69915')` correctly returns the original long URL — the sharded lookup (Chapter 4's own hash-based routing) found the exact same shard it was written to.

VERIFIED DIRECTLY — CACHE-ASIDE REDUCES 21 REDIRECTS TO 1 REAL STORE LOOKUP

Redirecting the same short code **21 times** (1 initial + 20 repeats) leaves `redirect_service.store_lookups` at exactly `1` — every redirect after the first was served entirely from cache, matching Chapter 3's own verified cache-aside shape at a different scale.

VERIFIED DIRECTLY — CLICK TRACKING NEVER BLOCKS A REDIRECT, AND STILL COUNTS EVERY CLICK CORRECTLY

Immediately after those 21 redirects, `click_queue.events` correctly holds **21** pending events — none of them processed yet, because `redirect()` only ever calls `publish()`, never a handler directly. Processing the queue afterward correctly produces `click_counts == {'c69915': 21}` — every single click accounted for, on its own schedule, exactly matching Chapter 6's own decoupling finding.

VERIFIED DIRECTLY — RATE LIMITING CORRECTLY THROTTLES A BURST OF CREATION REQUESTS

8 rapid calls to `create_short_url()` against a token bucket with capacity `5`: the first **5** succeed, the remaining **3** correctly raise `RuntimeError('Rate limit exceeded')` — `['created', 'created', 'created', 'created', 'created', 'rate-limited', 'rate-limited', 'rate-limited']`, matching Chapter 7's own token-bucket burst-then-throttle shape exactly.

VERIFIED DIRECTLY — REDIRECTS KEEP WORKING EVEN WHEN THE ENTIRE ANALYTICS PIPELINE IS BROKEN

Wrapping click processing in a circuit breaker (threshold `2`) and feeding it a handler that *always* raises `ConnectionError`: processing 10 queued clicks produces `['failed-full-cost', 'failed-full-cost', 'failed-fast', 'failed-fast', ...]` — the circuit opens after 2 real failures, exactly matching Chapter 9's own verified breaker behavior. Meanwhile, calling `redirect()` **10 more times** during this same broken-analytics period: every single one succeeds correctly, and `redirect_service.store_lookups` stays at `1` — completely untouched. The core redirect functionality never even knows the analytics pipeline exists.

The Rest of the System, Reasoned About Rather Than Re-Verified

Three more chapters shape this same system's real deployment, without needing fresh code to demonstrate here — each one's own mechanism was already verified in its own chapter, applied directly:

CHAPTER	HOW IT APPLIES TO THIS EXACT SYSTEM
Chapter 2 — Load Balancing	Multiple `UrlShortener` instances behind Least Connections, exactly as verified — redirect traffic vastly outweighs creation traffic in a real URL shortener, so read-heavy load balancing matters most here
Chapter 5 — CAP Theorem	Short-code creation should be CP: two clients racing to create a code for the same long URL must never produce two different codes for it, the identical risk Chapter 5 verified for AP-style writes. Redirects, by contrast, can safely be AP — a slightly stale cache entry (Chapter 3) is a tolerable cost for availability, not a hash collision risk
Chapter 8 — API Gateway & Service Discovery	A gateway would front `create_short_url()` and `redirect()` as separate routes, using discovery to find whichever shard/replica currently holds a given short code — exactly the registry pattern Chapter 8 verified surviving a redeploy

What This Course Doesn't Cover

This course stayed at the level of verifiable, single-process simulations of each pattern's own core mechanism — it did not cover actually deploying multiple real processes or machines, configuring a real load balancer or

message broker, or the operational work of running any of this in production. Software Architecture Fundamentals Chapter 10's own scope note named this course as the next step after getting a system's own shape right; this course, in turn, hands off to actually operating one — Technical Support's own `perfdiag1` / `incident1` / `netdiag1` courses cover diagnosing a system like this one once it's live and something goes wrong.

Hands-On Exercises

EXERCISE 1

Create two more short URLs through this chapter's own `UrlShortener`, then redirect all three short codes a mixed number of times (e.g., 5, 3, and 10 redirects respectively). Verify `store_lookups` is exactly `3` (one genuine miss per distinct code), not 1 and not 18.

View solution

EXERCISE 2

Using this chapter's own rate-limited `create_short_url()`, wait long enough for the token bucket to refill 2 tokens (at `1`/sec, this chapter's own configured rate), then attempt 2 more creations. Verify both succeed, confirming the limiter recovers correctly rather than staying permanently exhausted.

View solution

EXERCISE 3

Using this chapter's own verified resilience finding, explain specifically WHY `redirect()`'s own code never needed a try/except around anything analytics-related, when Chapter 9's own graceful-degradation example needed an explicit try/except around a failing dependency. What's structurally different about how this capstone wired click tracking?

View solution

CHAPTER 10 QUICK REFERENCE — AND COURSE QUICK REFERENCE

- **This capstone:** a URL shortener assembling Chapters 3, 4, 6, 7, and 9 into one running system — verified: 1 real store lookup across 21 redirects, 21 click events fully decoupled and correctly counted, 5 of 8 rapid creations allowed, and redirects proven immune to a fully broken, circuit-breaker-wrapped analytics pipeline
- **Course arc:** Ch.1 (why systems need to scale, measured) → Ch.2 (load balancing) → Ch.3 (caching) → Ch.4 (database scaling) → Ch.5 (CAP theorem) → Ch.6 (message queues) → Ch.7 (rate limiting) → Ch.8 (API gateways & discovery) → Ch.9 (resilience patterns) → Ch.10 (assembling it all)

- **Where this leads:** Technical Support's own diagnostic courses (`perfdiag1` , `incident1` , `netdiag1`) — this course builds the system; those courses diagnose it once it's live