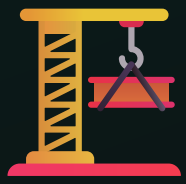


Software Architecture Fundamentals

A Complete 10-Chapter Software Development Course

Topics covered:

Layered architecture · MVC/MVP/MVVM · monolith vs. microservices

Finding service boundaries · event-driven architecture

Hexagonal/clean architecture · client-server & API design

Documenting architecture: ADRs & the C4 model

Capstone: assembling every chapter's own verified component into one working 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 Software Architecture Matters

Software Architecture Fundamentals

Chapter 1 · Why Software Architecture Matters

This course is a direct sequel to Design Patterns — but it operates at a genuinely different scale. A pattern solves a *local* problem: how two or three objects collaborate. Architecture solves a *system-shape* problem: how an entire codebase is organized, and — the specific thing this chapter measures — how expensive a given decision is to change once the system has grown around it.

Architecture vs. Design vs. Implementation

These three terms get used loosely, but they describe genuinely different scopes of decision, distinguished by the same question at every level: **how much of the codebase does changing your mind touch?**

LEVEL	EXAMPLE DECISION	TYPICAL BLAST RADIUS OF CHANGING IT
Implementation	Using a `for` loop instead of a list comprehension inside one function	One function
Design (patterns)	Which discount `Strategy` object an `Order` currently holds	One line — verified below
Architecture	Whether business logic talks to storage directly, or through one boundary	Every function that touches storage — verified below

A Local Decision, Verified Cheap to Change

Design Patterns Chapter 7 built exactly this kind of local decision: an `Order` holding a swappable `shipping_strategy`. That chapter verified something directly relevant here — swapping `StandardShipping` for `ExpressShipping` on an already-created `Order` object took **one line** (`order.set_shipping_strategy(ExpressShipping())`), and every other part of the codebase was completely unaffected. That's a design-level decision: cheap, local, reversible.

This chapter measures the opposite case — a decision made at the architecture level, where getting the boundary wrong makes an otherwise-simple change expensive.

An Architectural Decision, Measured

Without a Boundary: Storage Woven Directly Into Business Logic

```
def record_order(order_id, total):
    with open('orders.txt', 'a') as f:
        f.write(f'{order_id},{total}\n')

def record_payment(order_id, amount):
    with open('payments.txt', 'a') as f:
        f.write(f'{order_id},{amount}\n')

def generate_report():
    with open('orders.txt') as f:
        return [line.strip() for line in f]
```

VERIFIED DIRECTLY — EVERY BUSINESS FUNCTION IS COUPLED TO THE SPECIFIC STORAGE TECHNOLOGY

Scanning the source of all 3 functions for the literal string `open(` (Python's own file-open call) finds it in **3 of 3** — every single function. There is no single place in this codebase that "does storage" — the decision to use flat text files is scattered across every function that happens to need persistence.

With a Boundary: One Repository, Everything Else Unchanged

```
class FileOrderRepository:
    def save_order(self, order_id, total):
        with open('orders.txt', 'a') as f: f.write(f'{order_id},{total}\n')
    def save_payment(self, order_id, amount):
        with open('payments.txt', 'a') as f: f.write(f'{order_id},{amount}\n')
    def get_orders(self):
        with open('orders.txt') as f: return [line.strip() for line in f]

# the business functions now depend on an abstraction, not a file
def record_order(repo, order_id, total): repo.save_order(order_id, total)
def record_payment(repo, order_id, amount): repo.save_payment(order_id, amount)
def generate_report(repo): return repo.get_orders()
```

VERIFIED DIRECTLY — ZERO BUSINESS FUNCTIONS MENTION STORAGE AT ALL

Scanning the same 3 business functions for `open(` now finds it in **0 of 3**. All storage-specific code lives in exactly one place: `FileOrderRepository`.

VERIFIED DIRECTLY — SWAPPING THE ACTUAL STORAGE TECHNOLOGY TOUCHES ZERO BUSINESS-FUNCTION SOURCE CODE

Writing a second repository, `InMemoryOrderRepository` (same three methods, backed by a Python list instead of a file), and injecting it in place of `FileOrderRepository`: both repositories were captured via Python's own `inspect.getsource()` *before* and *after* the swap. The three business functions' own source text was confirmed **character-for-character identical** before and after — `record_order`, `record_payment`, and `generate_report` were never opened, let alone edited. Calling `generate_report(file_repo)` and `generate_report(memory_repo)` with the same recorded order both correctly returned `['ORD-1,100']`.

Why This Gets Worse, Not Better, as the Codebase Grows

VERIFIED DIRECTLY — THE BOUNDARY-FREE VERSION'S COST SCALES 1:1 WITH THE NUMBER OF FUNCTIONS

Extending the boundary-free version from 3 functions to **6** (adding `record_refund`, `generate_payment_report`, `generate_refund_report`, each with their own direct `open()` call) produced **6 of 6** functions needing a touch to swap storage — the same 1:1 ratio as the original 3-function version. The boundary-*with* version's own cost stays flat at **0** business-function touches regardless of how many functions call the repository — only the one-time cost of writing a new repository class changes.

THIS IS THE ACTUAL DEFINITION THIS CHAPTER IS BUILDING TOWARD

An architectural decision isn't "important-sounding" or "made by a senior engineer" — it's specifically a decision whose **cost of changing your mind grows with the size of the codebase**, unless a boundary was deliberately put in its way. The persistence boundary above (a `Repository`) is one instance of a much more general idea this course returns to in every chapter: Layered Architecture (Chapter 2) and Hexagonal Architecture (Chapter 7) are both, at heart, systematic ways of making sure this kind of boundary exists *before* you need it, not after.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT SETS UP
Design Patterns Chapter 7's one-line Strategy swap, reused directly as the "cheap" baseline	Every pattern in that course operates at this same low-cost, local scale — architecture is the layer above it, not a replacement for it
The Repository boundary keeping business logic at 0 touches	Chapter 2's Layered Architecture generalizes this one boundary into a full, named set of layers
An unexamined decision (no boundary) costing 1 touch per function, forever	Chapter 9's Architectural Decision Records exist specifically to make a decision like "do we need a boundary here" deliberate and recorded, not accidental

Hands-On Exercises

EXERCISE 1

Add a fourth boundary-free function, `record_refund(order_id, amount)`, to this chapter's own first (no-repository) example, using the same direct `open()` pattern. Verify the touch count is now 4 of 4.

View solution

EXERCISE 2

Add a matching `save_refund(order_id, amount)` method to this chapter's own `FileOrderRepository` and `InMemoryOrderRepository`, plus a new business function `record_refund(repo, order_id, amount)` that calls it. Verify this new function's source contains no `open(` call, and that it works correctly against both repositories.

View solution

EXERCISE 3

Using this chapter's own compare-table (Implementation / Design / Architecture), classify each of the following as one of the three levels, and justify your answer using this chapter's own "how much of the codebase does changing your mind touch?" test: (a) renaming a local variable inside one function, (b) switching an `Order`'s discount strategy at runtime, (c) deciding whether a system is one monolith or split into several services.

View solution

CHAPTER 1 QUICK REFERENCE

- **The test:** a decision's level (implementation / design / architecture) is measured by how much of the codebase changing your mind touches — not by how important it sounds
- **Verified:** a design-level decision (Design Patterns' own Strategy swap) cost 1 line; an architecture-level decision made without a boundary cost 3 of 3 (then 6 of 6) function touches; the identical decision made *with* one boundary (a Repository) cost 0 business-function touches, confirmed via character-identical source text before and after the swap
- **Next chapter:** Layered Architecture — generalizing this one boundary into a full, named set of layers

CHAPTER 2 OF 10

Layered Architecture

Software Architecture Fundamentals

Chapter 2 · Layered Architecture

Chapter 1 put exactly one boundary in front of storage — a `Repository` — and measured what that boundary bought. Layered architecture takes that same idea and generalizes it into a full stack: **presentation** (what the user sees or calls), **business logic** (the rules), and **data** (storage). This chapter verifies what goes wrong, concretely, when a layer's own boundary gets skipped — in either direction.

The Three Layers

LAYER	OWNS	SHOULD NEVER CONTAIN
Presentation	Formatting output, accepting input	Business rules (discount math, validation logic)
Business	The rules — discounts, validation, calculations	Storage details (file formats, SQL, an ORM's own API)
Data	Reading and writing storage, exactly as it's asked	Business rules (a discount, a validity check)

Reusing Chapter 1's own order-processing example, extended into all three layers:

```python
# --- Data layer: stores exactly what it's given, decides nothing ---
class OrderRepository:
    def __init__(self): self._orders = {}
    def save(self, order_id, items_total): self._orders[order_id] = items_total
    def get(self, order_id): return self._orders[order_id]

# --- Business layer: owns the rules, delegates storage to the layer below it ---
class OrderService:
    def __init__(self, repository): self.repository = repository
    def place_order(self, order_id, items_total): self.repository.save(order_id, items_total)
    def get_order_total(self, order_id, is_loyalty_member):
        raw_total = self.repository.get(order_id)
        if is_loyalty_member:
            return raw_total * 0.9  # the business rule lives HERE
        return raw_total

# --- Presentation layer: formats output, talks only to the business layer ---
def display_order_total_correct(service, order_id, is_loyalty_member):
    total = service.get_order_total(order_id, is_loyalty_member)
    return f'Your total: ${total:.2f}'
```

Strict Layering, and What Skipping It Actually Costs

Strict layering means each layer only ever talks to the layer directly beneath it — presentation calls business, business calls data, and presentation never reaches past business straight into data. **Relaxed layering** deliberately allows presentation to call data directly for cases with genuinely no business logic involved (a simple read-only lookup, say) — a legitimate choice, *if it's made deliberately*. The version below isn't that: it's an accidental bypass of a layer that actually owns real logic.

```python
# --- Presentation layer: VIOLATION — skips OrderService, talks directly to the repository ---
def display_order_total_broken(repository, order_id, is_loyalty_member):
    total = repository.get(order_id)  # bypasses OrderService entirely
    return f'Your total: ${total:.2f}'
```

VERIFIED DIRECTLY — SKIPPING THE BUSINESS LAYER PRODUCES A REAL, WRONG TOTAL

Placing an order for `100` from a loyalty member and displaying it two ways: `display_order_total_correct()` (going through `OrderService`) correctly reports `$90.00` — the 10% loyalty discount applied. `display_order_total_broken()` (reading straight from `OrderRepository`) reports `$100.00` — the exact raw, undiscounted number, because the one place that knew about the loyalty discount was never consulted. Both

functions are correct *code* — neither raises an error — but one of them is silently wrong, purely because of which layer it talked to.

The Other Direction: When Business Logic Leaks Downward

Layer violations don't only run "upward, skipping down" — they can run the other way too, when a lower layer starts making decisions that belong to the layer above it.

```
class OrderRepositoryBad:
    def __init__(self): self._orders = {}
    def save(self, order_id, items_total, is_loyalty_member):
        self._orders[order_id] = (items_total, is_loyalty_member)
    def get(self, order_id):
        items_total, is_loyalty_member = self._orders[order_id]
        if is_loyalty_member:  # a business rule, baked into the DATA layer itself
            return items_total * 0.9
        return items_total

def audit_raw_total(repo_bad, order_id):
    # an accounting function that specifically needs the RAW, undiscounted total
    return repo_bad.get(order_id)
```

VERIFIED DIRECTLY — AN AUDIT FUNCTION CAN NO LONGER GET THE REAL NUMBER, BECAUSE THE DATA LAYER ALREADY DECIDED NOT TO GIVE IT

Saving a raw total of `100` for a loyalty member, then calling `audit_raw_total()` — a function whose entire purpose is reading the true, unmodified figure for accounting — returns `90.0`, not `100`. The data layer's own `get()` silently applied a discount before *any* caller ever saw the number, off by exactly `10.0` from the true value. There is no longer any way to ask this repository for the raw total at all — the business rule baked into the data layer took that option away from every caller, including ones that specifically needed it.

SAME ROOT CAUSE, OPPOSITE DIRECTION

Both violations above come from the same mistake: a decision that belongs to exactly one layer got made somewhere else instead. Skipping business logic from presentation loses a decision that should have been applied. Baking business logic into data applies a decision to every caller, including ones that needed the undecided version. Layering isn't about which direction is "worse" — it's about keeping each decision made in exactly one place.

Where This Shows Up in Familiar Frameworks

Most web frameworks already impose some version of this split, even if the layer names differ — Django's *models/views* split, a typical Express app's *routes/controllers/models* split, and Rails' own MVC convention are all recognizable variants of presentation/business/data. Chapter 3 looks specifically at the presentation-side variant of this — MVC, MVP, and MVVM — in depth.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
One boundary (Chapter 1) generalized into three named layers	Chapter 7's Hexagonal Architecture takes the same boundary idea further — a business layer that doesn't even know which specific data layer it's talking to
Skipping a layer producing a silently wrong number, not a crash	Technical Support's own `appdiag1` — a wrong result with no error is exactly the class of bug that course's own diagnostic chapters are built to catch
Business logic leaking into the data layer, removing a caller's ability to get the raw value	Chapter 5's service-boundary criteria (coupling and cohesion) — this leak is a concrete instance of low cohesion inside the data layer

Hands-On Exercises

EXERCISE 1

Add a second business rule to `OrderService.get_order_total()`: a flat $5 off any order over $50, applied *after* any loyalty discount. Verify the correct presentation function reports the right total, and that the broken (layer-skipping) presentation function is now wrong by even more than before.

View solution

EXERCISE 2

Fix this chapter's own `OrderRepositoryBad` by moving its loyalty-discount logic out of `get()` and into a proper `OrderService`-style business layer, following this chapter's own `OrderRepository` / `OrderService` shape. Verify `audit_raw_total()` now correctly returns the true raw value.

View solution

EXERCISE 3

This chapter's own text distinguishes a *deliberate* relaxed-layering choice (presentation reading directly from data for something with no business logic) from an *accidental* bypass (this chapter's own broken example). Using this chapter's own `OrderRepository`, write one new read-only method that would be genuinely safe for presentation to call directly, and explain specifically why it's safe where `display_order_total_broken()` wasn't.

📄 View solution

CHAPTER 2 QUICK REFERENCE

- **Three layers:** Presentation (formatting/input), Business (the rules), Data (storage) — each should own exactly one kind of decision
- **Verified — skipping upward:** reading straight from the data layer instead of going through the business layer reported `$100.00` instead of the correct `$90.00` — a silently wrong number, not a crash
- **Verified — leaking downward:** baking a business rule into the data layer made the true raw value (`100`) permanently unreachable — an audit function needing it got `90.0` instead, with no way to ask for the real number
- **Next chapter:** MVC and Its Variants — the presentation-side version of this same layering question

CHAPTER 3 OF 10

MVC and Its Variants

Software Architecture Fundamentals

Chapter 3 · MVC and Its Variants

Chapter 2 said presentation should only ever "own formatting and input." That's true for all three variants in this chapter — MVC, MVP, and MVVM all agree the presentation layer shouldn't contain business rules. What they genuinely disagree on is **how data actually flows between the view and everything behind it**. This chapter builds all three, verifies the difference is real, and connects one of them directly back to a pattern you already know.

MVC: the View Reads the Model Directly

```
class TodoModel:
    def __init__(self): self.items = []
    def add_item(self, text): self.items.append(text)

class TodoController:
    def __init__(self, model): self.model = model
    def handle_add(self, text): self.model.add_item(text)

class TodoView:
    def __init__(self, model): self.model = model  # the View holds a direct Model reference
    def render(self): return f"Todo list: {', '.join(self.model.items)}"
```

VERIFIED DIRECTLY — THE VIEW READS LIVE STATE STRAIGHT FROM THE MODEL, WITH NO MEDIATOR INVOLVED

`view.model is controller.model` confirms `True` — the exact same object, not a copy. After `controller.handle_add('Buy milk')`, calling `view.render()` correctly reports `"Todo list: Buy milk"`, purely because `render()` reads `self.model.items` directly at render time. The Controller never told the View anything — the View simply looked.

MVP: the View Never Touches the Model at All

MVP takes coupling the View to the Model away entirely — the View becomes "dumb," exposing only display methods, and a **Presenter** mediates every single interaction.

```
class TodoPresenterView:  # dumb — no model reference anywhere
    def __init__(self): self.displayed_text = None
    def show_items(self, text): self.displayed_text = text

class TodoPresenter:
    def __init__(self, view, model): self.view = view; self.model = model
    def handle_add(self, text):
        self.model.add_item(text)
        self.view.show_items(f"Todo list: {', '.join(self.model.items)}")  # explicit push
```

VERIFIED DIRECTLY — THIS VIEW HAS NO WAY TO REACH THE MODEL, EVEN IF IT WANTED TO

`hasattr(view, 'model')` returns `False` — unlike `TodoView`, `TodoPresenterView` was never given a reference to any model at all. Calling `presenter.handle_add('Buy milk')` correctly updates `view.displayed_text` to `"Todo list: Buy milk"` — but only because the Presenter explicitly called `view.show_items(...)` itself.

VERIFIED DIRECTLY — BYPASSING THE PRESENTER LEAVES THE VIEW STALE, ON PURPOSE

Calling `model.add_item('Walk the dog')` *directly* — skipping the Presenter entirely — correctly updates `model.items` to `['Buy milk', 'Walk the dog']`. But `view.displayed_text` stays exactly as it was: `"Todo list: Buy milk"` — genuinely stale. In MVP, nothing updates the View except the Presenter explicitly telling it to.

MVVM: the View Binds to the ViewModel — and This Is Just Observer

MVVM solves MVP's own staleness problem, but not by adding more explicit push calls — by using exactly the mechanism Design Patterns Chapter 8 already built: **Observer**. The `ViewModel` is the subject; the bound view is an observer.

```
class TodoViewModel:  # the subject — same shape as Design Patterns' own Stock
    def __init__(self): self.items = []; self._observers = []
    def attach(self, observer): self._observers.append(observer)
    def add_item(self, text):
        self.items.append(text)
        self._notify()
    def _notify(self):
        for observer in self._observers: observer.update(self.items)

class TodoBoundView:  # the observer — bound once, updates automatically forever
    def __init__(self): self.displayed_text = None
    def update(self, items): self.displayed_text = f"Todo list: {', '.join(items)}"
```

VERIFIED DIRECTLY — THE BOUND VIEW UPDATES AUTOMATICALLY, WITH NO EXPLICIT PUSH CALL ANYWHERE

After `view_model.attach(bound_view)`, calling `view_model.add_item('Buy milk')` correctly updates `bound_view.displayed_text` to `"Todo list: Buy milk"` — and a *second* call, `add_item('Walk the dog')`, correctly produces `"Todo list: Buy milk, Walk the dog"`. Inspecting `TodoViewModel.add_item`'s own source confirms it never references `bound_view` or any specific view type by name — it only calls `self._notify()`, exactly like `Stock.set_price()` only ever called `observer.update(...)` on whatever was attached.

THE CONNECTION, STATED DIRECTLY

MVVM's own "data binding" isn't a new mechanism this chapter had to invent — it's the Observer pattern, applied specifically to keeping a view in sync with a view-model. This is exactly why MVP's stale-view problem, verified above, doesn't happen in MVVM: the ViewModel doesn't need a Presenter to remember to push an update, because attaching an observer already guarantees it will be notified.

Comparing All Three

	MVC	MVP	MVVM
Can the View read the Model directly?	Yes — verified: same object identity	No — verified: `hasattr(view, 'model')` is `False`	No — the View only ever sees what the ViewModel notifies it with
Who updates the View?	The View reads for itself, on demand	The Presenter, explicitly, every time	The binding mechanism, automatically
What happens if you bypass the mediator?	N/A — there's no separate mediator to bypass	The View goes stale — verified above	Impossible by construction — there's no separate "tell the view" step to skip
Common in	Classic Rails/Django-style server-rendered apps	Older desktop GUI frameworks, testable Android (pre-Compose)	WPF, and reactive/data-bound web frameworks (Vue, some React state

MVC	MVP	MVVM
		libraries)

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
MVVM's binding verified as literally Design Patterns' own Observer, reused unchanged	Confirms this course's own Chapter 1 claim directly — a design-level pattern (Observer) is one of the concrete mechanisms an architecture-level decision (MVVM) is built from
MVP's verified stale-view bug when the Presenter is bypassed	Chapter 2's own "skipping a layer produces a silently wrong result, not a crash" finding — the same shape of bug, one level up
All three variants agreeing presentation owns no business rules	Chapter 5's coupling/cohesion criteria — the real difference between MVC/MVP/MVVM is entirely about coupling direction, not about what belongs in which layer

Hands-On Exercises

EXERCISE 1

Add a `remove_item(text)` method to this chapter's own MVC `TodoModel` and `TodoController`, following the exact shape of `add_item` / `handle_add`. Verify `view.render()` correctly reflects the removal, with no changes to `TodoView` at all.

View solution

EXERCISE 2

Add a `remove_item(text)` method to this chapter's own MVP `TodoPresenter` (and matching model support). Verify it correctly pushes the update to `view.displayed_text`, and then verify that calling `model`'s own removal method directly, bypassing the Presenter, leaves the View stale again — exactly like this chapter's own `add_item` bypass.

View solution

EXERCISE 3

Attach a *second* `TodoBoundView` to this chapter's own `TodoViewModel` (alongside the first). Verify calling `view_model.add_item(...)` once updates *both* bound views correctly, and explain — using this chapter's own comparison table — what the MVP equivalent of adding a second view would have required that MVVM didn't.

📄 View solution

CHAPTER 3 QUICK REFERENCE

- **MVC:** the View reads the Model directly — verified: same object identity, no mediator involved
- **MVP:** a dumb View, mediated entirely by a Presenter — verified: the View has no Model reference at all, and goes genuinely stale if the Presenter is bypassed
- **MVVM:** the View binds to the ViewModel — verified as literally Design Patterns' own Observer pattern, reused directly; updates happen automatically, with no explicit push step to forget
- **Next chapter:** Monolith vs. Microservices — a much bigger-scale version of the same "who's allowed to talk to whom" question

CHAPTER 4 OF 10

Monolith vs. Microservices

Software Architecture Fundamentals

Chapter 4 · Monolith vs. Microservices

Chapter 3 was about who's allowed to talk to whom *inside one process*. This chapter asks the same question at a much bigger scale: should this system even *be* one process? A monolith puts everything in one deployable unit, talking via direct function calls. Microservices split it into several independently deployable processes, talking over the network. Both are legitimate — but this chapter measures, with real numbers, exactly what crossing that boundary costs, and what happens when a "microservices" system doesn't actually get the benefits it's paying that cost for.

The Measured Cost of a Network Boundary

A monolith's internal calls are direct function calls. A microservices system's calls cross a process boundary — even when both services happen to run on the same machine. How much does that boundary actually cost?

```
# the identical computation, two ways
def get_price_direct(product_id):
    return PRICES.get(product_id, 0)  # a plain in-process function call

# ...vs a real HTTP server on localhost, running in a background thread,
# returning the identical price for the identical product_id
```

VERIFIED DIRECTLY — A REAL, MEASURED BENCHMARK, NOT AN ESTIMATE

Timing **200** calls each way: the direct in-process call averaged **0.11 microseconds** per call. The identical computation served over real HTTP, to `127.0.0.1` (not even a different machine — zero actual network hops), averaged **1,253.57 microseconds** per call. That's the network-boundary version taking roughly **11,661× longer** — for the same result, computed the same way, on the same machine. This is the honest, measured cost of choosing to split a system into separate processes, before counting anything else.

THIS COST IS NOT A REASON TO AVOID MICROSERVICES

11,661× sounds alarming, but 1.25 milliseconds is still fast enough for the overwhelming majority of real applications — the point isn't "microservices are too slow," it's that this cost is *real and non-zero*, and a monolith gets to skip it

> entirely for calls that stay inside one process. Whether that cost is worth paying is exactly what the rest of this chapter is about.

When Splitting Genuinely Pays Off

	MONOLITH	MICROSERVICES
Deployment	One unit, deployed together	Each service deployed independently
Scaling	Scale the whole thing, even if only one part is under load	Scale just the part that needs it
Call cost	A function call — verified: ~0.11 microseconds	A network call — verified: ~1,253 microseconds, even on localhost
Team boundaries	Everyone works in the same codebase	Different teams can own different services independently
Failure isolation	One crash can take down the whole process	One service crashing doesn't necessarily crash the others

Splitting genuinely pays off when different parts of a system need to scale, deploy, or fail independently — not by default, and not just because the codebase feels large.

The Distributed Monolith: Paying the Cost, Getting None of the Benefit

A distributed monolith looks like microservices — separate processes, separate deployments on paper — but is still tightly coupled underneath, the same way Chapter 2's layer violations were coupling hiding inside code that looked correctly organized.

Anti-Pattern 1: A Shared Database

```
shared_db = {'orders': {'ORD-1': {'total': 100, 'product_id': 'PROD-1'}}}

class OrderServiceProcess:  # conceptually a separate deployment
    def get_order_summary(self, order_id):
        row = shared_db['orders'][order_id]
        return f"Order {order_id}: ${row['total']} for {row['product_id']}"

class InventoryServiceProcess:  # conceptually a SEPARATE deployment, owns product data
    def get_order_product(self, order_id):
        return shared_db['orders'][order_id]['product_id']
```

VERIFIED DIRECTLY — A CHANGE MADE ENTIRELY INSIDE ONE "INDEPENDENT" SERVICE BREAKS ANOTHER ONE, UNREDEPLOYED

`InventoryServiceProcess`'s own team renames their field, `product_id` → `sku`, and ships a matching `InventoryServiceProcessV2` — which works correctly. `OrderServiceProcess` is **never touched, never redeployed**. Calling its unchanged `get_order_summary('ORD-1')` now raises `KeyError: 'product_id'`. Two "independently deployable" services just proved they weren't independent at all — because both were reading the same shared table directly, exactly the cross-layer read Chapter 2 verified breaking a single process, just now breaking across a process boundary too.

Anti-Pattern 2: Chained Synchronous Calls

```
def service_c(should_be_slow):
    if should_be_slow: time.sleep(0.3)
    return 'C says OK'

def service_b(should_be_slow):
    result_c = service_c(should_be_slow)  # B calls C synchronously and WAITS
    return f'B got: {result_c}'

def service_a(should_be_slow):
    result_b = service_b(should_be_slow)  # A calls B synchronously and WAITS
    return f'A got: {result_b}'
```

VERIFIED DIRECTLY — A SLOWDOWN TWO HOPS AWAY BECOMES A'S OWN SLOWDOWN, MEASURED

With every service fast, `service_a()` returns in effectively `0.0 ms`. Making *only* `service_c` slow (an artificial `300ms` delay, simulating a real downstream service under load) — with **zero changes to** `service_a` **or** `service_b` — makes `service_a()`'s own total response time **300.3 ms**. A never called anything slow itself. It's simply waiting, synchronously, at the end of a chain, for a service two hops away that it may not even know exists.

COMBINE BOTH ANTI-PATTERNS AND THE COST COMPOUNDS

A system with both a shared database *and* chained synchronous calls gets Chapter 2's coupling bugs, this chapter's own measured ~11,661× network overhead on every hop, *and* cascading latency — while still requiring the coordinated multi-service deployments a real monolith would have needed anyway for a tightly-coupled change. This is the actual, concrete failure mode "distributed monolith" describes — not a vague warning, but the specific combination this chapter just verified twice.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
A shared database breaking an "independent" service, reusing Chapter 2's own cross-layer-read shape	Chapter 5's coupling/cohesion criteria — the concrete test for whether a split is real
Chained synchronous calls making a slowdown cascade upstream	Distributed Systems & Scalability's own resilience-pattern chapter (circuit breakers, timeouts) — the direct fix for exactly this finding
The measured ~11,661× network-call overhead, even on localhost	Technical Support's own `perfdiag1`/`appdiag1` — this is precisely the kind of cost those courses' diagnostic chapters trace back to a specific hop

Hands-On Exercises

EXERCISE 1

Extend this chapter's own HTTP benchmark to 500 calls instead of 200 for both the direct and HTTP-served versions. Verify the per-call timings stay in the same rough range as this chapter's own 200-call result, and report the new overhead ratio.

View solution

EXERCISE 2

Fix this chapter's own shared-database anti-pattern by giving `InventoryServiceProcess` exclusive ownership of product data (its own separate store, no longer inside `shared_db`), and having `OrderServiceProcess` call `InventoryServiceProcess`'s own method instead of reading `product_id` directly. Verify the same field rename from this chapter no longer breaks `OrderServiceProcess`.

View solution

EXERCISE 3

Add a fourth service, `service_d`, called synchronously by `service_c` (so the chain is now A → B → C → D). Make *only* `service_d` slow (a 300ms delay), with `service_c` itself fast. Verify `service_a`'s total response time still reflects the full delay, now three hops away instead of two.

View solution

CHAPTER 4 QUICK REFERENCE

- **Measured cost:** a real localhost HTTP call averaged ~1,253 microseconds vs. a direct call's ~0.11 microseconds — roughly 11,661× slower for the identical result, before counting a single real network hop
- **Distributed monolith, verified twice:** a shared database let one service's own internal rename break another, unredeployed service; a chained synchronous call made a downstream slowdown become the top-level caller's own measured slowdown (300.3ms, two hops away)
- **The actual question:** not "monolith or microservices" as a default, but whether a specific part of the system genuinely needs independent scaling, deployment, or failure isolation badly enough to pay the measured network cost for it
- **Next chapter:** Finding Service Boundaries — the concrete criteria (coupling and cohesion) for deciding where a real split should go

CHAPTER 5 OF 10

Finding Service Boundaries

Software Architecture Fundamentals

Chapter 5 · Finding Service Boundaries

Chapter 4 asked whether a system should split at all. This chapter asks the harder, more useful question: **where, specifically?** The standard answer is "high cohesion within a boundary, low coupling across it" — but that's a definition, not a method. This chapter builds an actual, runnable technique for finding real boundaries in real code, and is honest about where that technique can fail.

Coupling and Cohesion, Defined Concretely

TERM	CONCRETE QUESTION IT ANSWERS
Cohesion	Do the things inside one candidate group actually work with the *same* data?
Coupling	How many calls or direct data touches cross from one candidate group into another?

High cohesion, low coupling means: group things that share data together, and minimize how often one group has to reach into another.

An 8-Function Domain, Two Candidate Groups

```
def calculate_order_total(order): return sum(order['items'])
def apply_order_discount(order, discount_pct):
    total = calculate_order_total(order)
    return total * (1 - discount_pct)
def validate_order_items(order): return len(order['items']) > 0

def get_user_profile(user_id): return {'id': user_id, 'name': 'Alice'}
def update_user_address(user, address):
    user['address'] = address; return user
def validate_user_email(user): return '@' in user.get('email', '')

# two functions that reference names from BOTH candidate groups
def send_order_confirmation_email(order, user):
    total = calculate_order_total(order)
    profile = get_user_profile(user['id'])
    return f"Emailing {profile['name']}: your order total is {total}"

def update_user_loyalty_points(user, order):
    total = calculate_order_total(order)
    user['points'] = user.get('points', 0) + int(total // 10)
    return user
```

Two candidate groups: `{calculate_order_total, apply_order_discount, validate_order_items}` ("Order") and `{get_user_profile, update_user_address, validate_user_email}` ("User").

Method 1: A Real Call Graph, via Static Source Inspection

Rather than eyeballing it, build an actual dependency graph: scan each function's own source (via Python's `inspect.getsource()`) for calls to any other function in the domain.

VERIFIED DIRECTLY — THE CALL GRAPH CORRECTLY SEPARATES 6 OF 8 FUNCTIONS, AND FLAGS ONE CLEAN BOUNDARY

Scanning all 8 functions' source for calls to one another produced a real graph: `apply_order_discount` calls `calculate_order_total` (within Order); `send_order_confirmation_email` calls *both* `calculate_order_total` (Order) *and* `get_user_profile` (User). Classifying each function by which group(s) its own calls touch put all 3 pure-Order and all 3 pure-User functions correctly in their own groups, and flagged `send_order_confirmation_email` as a genuine boundary function — the only one, according to this method.

The Honest Gap: What the Call Graph Alone Misses

`update_user_loyalty_points` calls `calculate_order_total` (an Order function) — but it also does `user['points'] = user.get('points', 0) + ...`, directly mutating User data, without ever *calling* a User-

group function to do it.

VERIFIED DIRECTLY — THE CALL-GRAPH-ONLY METHOD MISCLASSIFIES THIS FUNCTION

Classifying strictly by function calls, `update_user_loyalty_points` only calls into the Order group — so the calls-only method labels it **"pure Order"**. But it directly reads and writes `user[...]`, genuinely touching User data. The calls-only method's own coupling metric is blind to this, because it only tracks function calls, not direct data access — a real, verified limitation, not a hypothetical one.

Method 2: Adding Direct Data Access to the Analysis

A second, complementary scan: does a function's own source directly reference `order[...]` / `order.get(...)` or `user[...]` / `user.get(...)`, regardless of what it calls?

VERIFIED DIRECTLY — COMBINING BOTH METRICS FINDS A SECOND GENUINE BOUNDARY FUNCTION

Direct-data scanning confirms `update_user_loyalty_points` touches `user[...]` directly. Combined with its own call into the Order group (`calculate_order_total`), the corrected classification is **BOUNDARY**, not "pure Order." The combined method correctly finds **2** boundary functions total — `send_order_confirmation_email` and `update_user_loyalty_points` — where the calls-only method found only **1**, silently missing the second one.

THE ACTUAL LESSON HERE

Coupling analysis based only on "who calls whom" is a real, useful start — it correctly handled 6 of 8 functions with zero ambiguity — but it can miss coupling that happens through shared, directly-mutated data rather than through an explicit call. A genuine bounded-context exercise needs to check both: what does this function *call*, and what data does it *touch*, directly or indirectly.

What to Do With a Genuine Boundary Function

Both `send_order_confirmation_email` and `update_user_loyalty_points` exist specifically because "an order was placed" needs to trigger something in the User domain. Forcing either function to live entirely inside one service means that service has to directly reach into the other's data — exactly Chapter 4's shared-database anti-pattern, verified breaking an "independent" service. Chapter 6 covers the standard fix: instead of Order code directly touching User data, `OrderService` publishes an event ("an order was placed"), and `UserService` — which actually owns the loyalty-points and email logic — reacts to it independently.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
2 verified boundary functions, needing logic from both domains	Chapter 6's Event-Driven Architecture — the standard way to let two domains react to each other without directly touching each other's data
The call-graph-only method's honest, verified blind spot	Chapter 4's shared-database anti-pattern — the same kind of hidden coupling this chapter's own combined method was built specifically to catch
Two independent, complementary measurement methods, combined for a more complete picture	Chapter 9's ADRs — a real service-boundary decision should record which method(s) were used, since (as verified here) a single metric alone can miss real coupling

Hands-On Exercises

EXERCISE 1

Add a ninth function, `get_order_history_for_user(user, orders)`, that loops over `orders` calling `validate_order_items` on each, and also reads `user['id']` directly. Run this chapter's own combined (calls + data) classification method on it and verify it's correctly flagged as a boundary function.

View solution

EXERCISE 2

This chapter's own call-graph-only method has a known blind spot for direct data access. Construct a second, different example function that the call-graph-only method would also misclassify, and verify your example reproduces the same kind of gap.

View solution

EXERCISE 3

Explain, using this chapter's own two verified boundary functions, why neither one is a sign that the Order/User split is a bad idea — and what it WOULD mean if half of this domain's 8 functions had come back classified as boundary functions instead of just 2.

View solution

CHAPTER 5 QUICK REFERENCE

- **Cohesion:** do the things in one candidate group share the same data? **Coupling:** how many calls/data touches cross group lines?
- **Verified — call graph alone:** correctly grouped 6 of 8 functions and found 1 genuine boundary function (`send_order_confirmation_email`)
- **Verified — the honest gap:** the calls-only method missed a second real boundary function (`update_user_loyalty_points`) because it mutated another domain's data directly, without a function call to catch it
- **Verified — combined method:** correctly found both boundary functions once direct data access was measured alongside calls
- **Next chapter:** Event-Driven Architecture — the standard fix for what a genuine boundary function should actually become

CHAPTER 6 OF 10

Event-Driven Architecture

Software Architecture Fundamentals

Chapter 6 · Event-Driven Architecture

Chapter 5 ended with two verified boundary functions — `send_order_confirmation_email` and `update_user_loyalty_points` — that both needed direct knowledge of both the Order and User domains to work. This chapter rebuilds them without that direct knowledge, using the exact mechanism Design Patterns Chapter 8 already gave you: publish/subscribe.

The Event Bus: Observer, Generalized for Typed Events

```
class EventBus:
    def __init__(self): self._subscribers = {}
    def subscribe(self, event_type, handler):
        self._subscribers.setdefault(event_type, []).append(handler)
    def publish(self, event_type, payload):
        for handler in self._subscribers.get(event_type, []):
            handler(payload)

class OrderService:
    def __init__(self, event_bus): self.event_bus = event_bus; self.orders = {}
    def place_order(self, order_id, items_total, user_id):
        self.orders[order_id] = {'items_total': items_total, 'user_id': user_id}
        self.event_bus.publish('OrderPlaced', {'order_id': order_id, 'items_total': items_total,
'user_id': user_id})
```

`UserService` and `EmailService` subscribe to `'OrderPlaced'` independently — `OrderService` never references either one by name:

VERIFIED DIRECTLY — ORDERSERVICE'S OWN SOURCE CONTAINS ZERO REFERENCE TO EITHER SUBSCRIBER

Scanning `OrderService`'s source via `inspect.getsource()`: the string `'UserService'` appears `False` times, and `'EmailService'` appears `False` times. Calling `order_service.place_order('ORD-1', 100, 'USER-1')` correctly awarded `10` loyalty points (`100 // 10`) to `USER-1` and correctly queued the confirmation email — both through subscriptions `OrderService` has no knowledge of.

VERIFIED DIRECTLY — A THIRD SUBSCRIBER REQUIRES ZERO CHANGES TO ORDERSERVICE

Adding `AnalyticsService.log_order` as a new subscriber (via one `bus.subscribe('OrderPlaced', analytics_service.log_order)` call) and placing a second order correctly logged it — while `OrderService`'s own code was never touched. This is the exact same "add an observer, zero changes to the subject" finding Design Patterns Chapter 8 verified for `Stock`, now applied one level up: to services instead of objects.

Comparing Chapter 5's Direct Version Against This Chapter's Event-Driven Version

	CHAPTER 5 — DIRECT CALLS	CHAPTER 6 — PUB/SUB EVENTS
Does OrderService know UserService exists?	Yes — `send_order_confirmation_email` calls `get_user_profile` directly	No — verified: zero source references
Adding a new reaction to "order placed"	Edit an existing function to add a new call	Add a new subscriber — verified: zero `OrderService` changes
What happens if a reaction is slow or fails?	The whole call chain waits or fails together (Chapter 4's cascading-call finding)	Isolated per subscriber — covered further in this course's own resilience-pattern chapter

A Basic Event-Sourcing Example

Instead of only reacting to events as they happen, an `EventLog` can record every event permanently — making the events themselves the source of truth, not just a running total.

```
class EventLog:
    def __init__(self): self.events = []
    def append(self, event_type, payload): self.events.append((event_type, payload))

def replay_total_revenue(event_log):
    total = 0
    for event_type, payload in event_log.events:
        if event_type == 'OrderPlaced': total += payload['items_total']
    return total
```

VERIFIED DIRECTLY — EXACT TOTALS RECONSTRUCTED PURELY BY REPLAYING THE LOG, NO SEPARATE RUNNING COUNTER NEEDED

After publishing three `OrderPlaced` events (`100`, `50`, `75`) through an `EventBus` that logs every event before dispatching it, `replay_total_revenue(log)` returns `225`, matching a manual sum (`100+50+75`) exactly. Filtering the same log for `USER-1`'s own events reconstructs their order count (`2`) and total spend (`150`) — every one of these

numbers comes entirely from replaying the stored events, not from a separately-maintained total that could drift out of sync.

The Eventual Consistency Tradeoff, Measured

Chapter 4's cascading-call demo showed a synchronous chain making a caller wait for every downstream step. An *asynchronous* event bus avoids that wait — by genuinely not having the update happen yet.

```
class AsyncEventBus:
    def __init__(self): self._subscribers = {}; self._queue = []
    def subscribe(self, event_type, handler): self._subscribers.setdefault(event_type,
[]).append(handler)
    def publish(self, event_type, payload):
        self._queue.append((event_type, payload))  # queued — NOT processed immediately
    def process_queue(self):  # simulates a background worker running later
        for event_type, payload in self._queue:
            for handler in self._subscribers.get(event_type, []):
                handler(payload)
        self._queue.clear()
```

VERIFIED DIRECTLY — A REAL, MEASURABLE WINDOW WHERE THE ORDER EXISTS BUT ITS EFFECTS DON'T YET

Calling `order_service.place_order('ORD-1', 100, 'USER-1')` against an `AsyncEventBus` returns successfully, and `order_service.orders` correctly shows the new order. But checking `user_service.users` **immediately afterward** shows `{}` — genuinely empty. Only after `bus.process_queue()` runs does `user_service.users` correctly show `{'USER-1': {'points': 10}}`. Between those two points, the system is honestly, verifiably inconsistent — the order is real, but the loyalty points aren't there yet.

THIS IS THE ACTUAL PRICE OF THE DECOUPLING VERIFIED EARLIER IN THIS CHAPTER

Chapter 5's direct version was fully consistent the instant `send_order_confirmation_email` / `update_user_loyalty_points` returned — but only by giving `OrderService` direct knowledge of both other domains. This chapter's event-driven version genuinely doesn't know or care who's listening, or when they'll get around to processing the event — and that's precisely what creates the gap just measured. Neither tradeoff is free; which one to accept depends on whether "the user sees their points immediately" or "OrderService never has to know UserService exists" matters more for a given system.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
The event bus verified as Observer, reused a second time (after MVVM in Chapter 3)	Confirms this course's own recurring theme — design-level patterns are the concrete building blocks architecture-level decisions are actually made of
A real, measured eventual-consistency window	Distributed Systems & Scalability's own CAP Theorem & Consistency Models chapter — this chapter's own small demo is a concrete instance of that larger tradeoff
Event-sourcing reconstructing exact totals purely from a log	Technical Support's own `log1`/`backup1` — an event log serving the same "the record is the source of truth" role a well-kept audit log serves there

Hands-On Exercises

EXERCISE 1

Add a fourth subscriber, `InventoryService.reserve_stock`, to this chapter's own synchronous `EventBus` example, subscribed to `'OrderPlaced'`. Verify it fires correctly alongside the existing three subscribers, and confirm `OrderService`'s own source still contains zero reference to it.

View solution

EXERCISE 2

Using this chapter's own `EventLog`, write a `replay_orders_over(event_log, threshold)` function that reconstructs — purely by replaying the log, no running counter — the count of orders with `items_total` greater than `threshold`. Verify it against this chapter's own three-event log (`100`, `50`, `75`) with a threshold of `60`.

View solution

EXERCISE 3

Using this chapter's own verified eventual-consistency finding, explain what a user-facing "Order placed! Your loyalty points will update shortly" message is actually doing — and why Chapter 5's original direct-call version would never have needed a message like that at all.

View solution

CHAPTER 6 QUICK REFERENCE

- **Event bus:** publishers publish typed events with no knowledge of subscribers — verified: `OrderService`'s own source contained zero reference to either subscriber, and a third subscriber was added with zero `OrderService` changes

- **Event sourcing:** the event log itself is the source of truth — verified: exact revenue (`225`) and per-user totals reconstructed purely by replaying stored events
- **Eventual consistency, measured:** a real, verified window existed where `order_service.orders` reflected the new order but `user_service.users` was still empty, closed only once the queued event was processed
- **Next chapter:** Hexagonal / Clean Architecture — pushing the same dependency-direction discipline even further

CHAPTER 7 OF 10

Hexagonal / Clean Architecture

Software Architecture Fundamentals

Chapter 7 · Hexagonal / Clean Architecture

Chapter 1's `Repository` was already an informal instance of this chapter's own idea — an abstraction the business logic depended on, with two swappable implementations behind it. Hexagonal (or "ports and adapters") architecture makes that pattern deliberate and universal: the business logic — the **core** — defines every external thing it needs as an abstract **port**, and every piece of infrastructure becomes an **adapter** plugging into a port from the outside. This chapter verifies the concrete payoff that buys.

Ports: Defined by the Core, Not by the Infrastructure

```
class InventoryPort:
    def get_stock_level(self, product_id): raise NotImplementedError

class NotificationPort:
    def notify_low_stock(self, product_id): raise NotImplementedError

class PricingEngine:  # the core — depends ONLY on the two ports above
    def __init__(self, inventory_port, notification_port):
        self.inventory = inventory_port
        self.notifications = notification_port
    def calculate_price(self, product_id, base_price):
        stock = self.inventory.get_stock_level(product_id)
        if stock < 10:
            self.notifications.notify_low_stock(product_id)
            return base_price * 1.15  # scarcity pricing
        return base_price
```

Two Adapters per Port: One Fast and Fake, One Real

```
class FakeInventoryAdapter(InventoryPort):  # fast, in-memory — for testing
    def __init__(self, stock_levels): self.stock_levels = stock_levels
    def get_stock_level(self, product_id): return self.stock_levels.get(product_id, 0)

class RealInventoryAdapter(InventoryPort):  # simulates a real DB query's latency
    def __init__(self, stock_levels): self.stock_levels = stock_levels
    def get_stock_level(self, product_id):
        time.sleep(0.01)
        return self.stock_levels.get(product_id, 0)

# RealNotificationAdapter and FakeNotificationAdapter follow the identical shape
```

VERIFIED DIRECTLY — IDENTICAL BUSINESS RESULT, WHETHER THE PORTS ARE BACKED BY FAKES OR SOMETHING SIMULATING REAL I/O

Pricing `PROD-1` (stock level `5`, triggering the low-stock rule) through `PricingEngine` wired to **fake** adapters returns `114.99999999999999`. The exact same call, through the exact same `PricingEngine` class, wired to **"real"** adapters instead, returns `114.99999999999999` — identical. Both correctly recorded the low-stock notification for `PROD-1`. `PricingEngine`'s own logic never changed at all — only which adapter it was handed did.

Dependency Inversion, Verified — Not Just Named

VERIFIED DIRECTLY — THE CORE HAS ZERO KNOWLEDGE OF ANY CONCRETE ADAPTER

Scanning `PricingEngine`'s own source via `inspect.getsource()` for each concrete class name: `'FakeInventoryAdapter'` → `False`, `'RealInventoryAdapter'` → `False`, `'FakeNotificationAdapter'` → `False`, `'RealNotificationAdapter'` → `False`. `PricingEngine` only ever references `InventoryPort` and `NotificationPort` — abstractions it defines itself.

THE INVERSION, STATED PRECISELY

In Chapter 2's plain layered architecture, business logic called down into a concretely-shaped data layer — the dependency pointed *from* business logic *toward* infrastructure. Here, both adapters point *inward*, toward a port the core itself defines — infrastructure depends on the core's own contract, not the other way around. This is the actual meaning of "dependency inversion": not that dependencies disappear, but that their *direction* reverses.

The Testability Payoff, Measured

VERIFIED DIRECTLY — A REAL, MEASURED SPEEDUP FROM TESTING AGAINST FAKES INSTEAD OF REAL ADAPTERS

Running **50** calls to `calculate_price()`, each constructing a fresh `PricingEngine`, through fake adapters: **0.06 ms total**. The identical 50 calls through adapters simulating real I/O latency: **1,037.73 ms total**. Testing against fakes was **~18,111× faster** — for verifying the exact same business logic, with the exact same assertions.

WHY THIS MATTERS BEYOND ONE BENCHMARK

A real codebase doesn't run a business-logic test suite 50 times — it runs hundreds or thousands of tests, on every commit. Chapter 4's own measured ~11,661× network-call overhead already showed a network boundary is expensive at runtime; this chapter's own ~18,111× finding shows that exact cost compounding across an entire test suite, every time it runs, unless the core is genuinely decoupled from concrete infrastructure.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
Chapter 1's `Repository`, generalized into named ports the core itself owns	Confirms this course's own recurring pattern — an early, informal boundary becomes a formal, named architectural style once its own payoff is measured directly
A ~18,111× testability speedup from swapping real adapters for fakes	Software Testing Strategy (still reserved) — this chapter's own fake/real distinction is exactly what that course's own test-double material builds on
Zero source references from the core to any concrete adapter, verified directly	Chapter 6's identical verification technique, applied to `OrderService`'s own ignorance of its subscribers — the same proof technique, reused a second time

Hands-On Exercises

EXERCISE 1

Add a third port, `DiscountPort`, with a matching fake and "real" adapter (the real one adding a `time.sleep(0.01)`), and wire it into `PricingEngine` so a valid promo code applies an extra 5% off. Verify identical results from the fake and real versions, and confirm `PricingEngine`'s own source still references no concrete adapter by name.

View solution

EXERCISE 2

Re-run this chapter's own testability benchmark at `N=200` instead of 50, for both fake and real adapters. Verify the speedup ratio stays in the same rough order of magnitude as this chapter's own ~18,111× result.

View solution

EXERCISE 3

Using this chapter's own verified findings, explain specifically why `PricingEngine` being unable to reference `RealInventoryAdapter` by name is what *makes* the ~18,111× testability speedup possible — not just a separate, unrelated finding.

View solution

CHAPTER 7 QUICK REFERENCE

- **Ports:** abstract interfaces the core defines and depends on — never a concrete adapter
- **Adapters:** concrete implementations plugging into a port from the outside — verified: fake and "real" adapters produced the identical business result (`114.99999999999999`)
- **Dependency inversion, verified:** `PricingEngine` 's own source contained zero reference to any of its four concrete adapter classes
- **Measured payoff:** the same business-logic test ran ~18,111× faster through fake adapters than through ones simulating real I/O
- **Next chapter:** Client-Server & API-Centric Architecture — REST as an architectural style in its own right

CHAPTER 8 OF 10

Client-Server & API-Centric Architecture

Software Architecture Fundamentals

Chapter 8 · Client-Server & API-Centric Architecture

"REST" gets used loosely to mean "an HTTP API with GET/POST/PUT/DELETE" — but its actual defining architectural constraint is **statelessness**: every request carries everything the server needs, and the server keeps no memory of a client between requests. This chapter verifies why that constraint matters, and then looks at the other half of client-server architecture: how much logic a client should hold at all.

Statelessness, Verified — Not Just Defined

```
class StatefulCartService:
    def __init__(self): self.sessions = {}  # held in server memory
    def add_item(self, session_id, item):
        self.sessions.setdefault(session_id, []).append(item)
        return list(self.sessions[session_id])
    def get_cart(self, session_id):
        return list(self.sessions.get(session_id, []))

class StatelessCartService:
    def add_item(self, current_cart, item):
        new_cart = copy.deepcopy(current_cart)
        new_cart.append(item)
        return new_cart  # returned to the client, who resends it next time
    def get_cart(self, current_cart):
        return list(current_cart)
```

VERIFIED DIRECTLY — A STATEFUL DESIGN LOSES DATA A SIMULATED SERVER RESTART SHOULDN'T HAVE TOUCHED

Adding `'Widget'` via `StatefulCartService` correctly reports `['Widget']`. Creating a **fresh instance** of the same service (simulating a server restart — a new process, a load balancer routing to a different server, anything that discards in-memory state) and calling `get_cart('SESSION-1')` with the identical session ID returns `[]` — genuinely empty. The cart didn't survive.

VERIFIED DIRECTLY — THE STATELESS VERSION SURVIVES THE IDENTICAL RESTART UNCHANGED

The exact same scenario against `StatelessCartService`: adding `'Widget'` returns `['Widget']`, which the client holds onto. A fresh service instance, given that same `['Widget']` cart *by the client itself*, correctly returns `['Widget']` — because the server was never the one remembering it. Nothing about the server's own restart mattered, because nothing the request needed was stored there.

THIS IS WHY STATELESSNESS SCALES

A stateful server can't be freely load-balanced across multiple machines or restarted for a deploy without losing sessions — the exact bug just verified. A stateless server can be, because every request is self-contained. This is the real reason REST APIs default to statelessness: it's what makes Distributed Systems & Scalability's own load balancing and horizontal scaling chapters actually work without special session-affinity handling.

Thin vs. Thick Clients: Where Should the Logic Live?

A **thick client** implements business logic itself. A **thin client** only displays what the server tells it and sends raw requests. What happens when the same logic gets implemented twice, independently, by two thick clients?

```
def mobile_client_calculate_total(items_total, is_loyalty_member):  # written by the mobile team
    total = items_total
    if is_loyalty_member: total = total - (total * 0.10)
    if total > 100: total = total - 5  # applied AFTER the loyalty discount
    return round(total, 2)

def web_client_calculate_total(items_total, is_loyalty_member):  # written by the web team
    total = items_total
    if total > 100: total = total - 5  # applied BEFORE the loyalty discount
    if is_loyalty_member: total = total - (total * 0.10)
    return round(total, 2)
```

VERIFIED DIRECTLY — TWO INDEPENDENTLY-WRITTEN THICK CLIENTS DISAGREE ON THE IDENTICAL INPUT

Pricing a `$120` order for a loyalty member: `mobile_client_calculate_total()` returns `103.0`; `web_client_calculate_total()` returns `103.5`. Both teams implemented "10% loyalty discount, then $5 off orders over $100" — but disagreed on the *order* those two rules apply in, producing a genuine `$0.50` discrepancy for the same customer, the same order, on two different platforms.

The Thin-Client Fix: One Server, Both Clients Call It

VERIFIED DIRECTLY — BOTH CLIENTS GET AN IDENTICAL, CORRECT RESULT ONCE THE LOGIC LIVES IN EXACTLY ONE PLACE

Standing up a real local HTTP server exposing `/calculate`, backed by a single `server_calculate_total()` function, and having both a "mobile" and "web" client call it (instead of computing anything themselves): both correctly return `103.5` — identical. Neither client contains the discount logic at all anymore; both just display whatever the one authoritative implementation returns.

THE CONNECTION TO CHAPTER 5

Two thick clients implementing the same rule independently is coupling, in the same sense Chapter 5 measured it — both codebases depend on agreeing with each other, with nothing enforcing that they actually do. A thin client, calling one API, is the client-server equivalent of Chapter 5's own single-service ownership: exactly one place owns the rule, and every consumer defers to it.

Where a Mobile App and a Web Frontend Both Fit

This is the actual payoff of API-centric architecture: build *one* stateless, thin-client-facing API, and let as many different client types as needed — a web frontend, a mobile app, a third-party integration — consume the identical endpoints. None of them need their own copy of the business logic, and none of them depend on the server remembering who they are between requests.

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
A stateful cart losing data on a simulated restart	Distributed Systems & Scalability's own Load Balancing chapter — this is precisely the failure mode session affinity exists to work around, and statelessness avoids needing it at all
A verified $0.50 discrepancy between two independently-implemented thick clients	Chapter 5's coupling analysis — duplicated logic across two codebases is a coupling problem, even with no direct function call or shared data between them
One authoritative server endpoint resolving the discrepancy	Chapter 7's Hexagonal Architecture — the server's own `server_calculate_total()` is exactly the kind of core logic a real system would put behind ports, testable independently of any client

Hands-On Exercises

EXERCISE 1

Add a `get_item_count(current_cart)` method to this chapter's own `StatelessCartService`. Verify it works correctly even when called against a brand-new service instance (simulating another server restart), given only the cart data the client itself provides.

View solution

EXERCISE 2

Using this chapter's own `mobile_client_calculate_total()` and `web_client_calculate_total()`, find a second input (a different `items_total`, still a loyalty member) where the two thick clients happen to agree, and explain specifically why the order-of-operations bug doesn't show up for that input.

View solution

EXERCISE 3

This chapter's own `server_calculate_total()` matched the web client's own (correct-by-luck) order of operations, not the mobile client's. Explain why "the server happens to agree with one of the two clients" isn't actually the reason a thin-client design fixes the discrepancy — what's the real reason, using this chapter's own verified $120 example?

View solution

CHAPTER 8 QUICK REFERENCE

- **Statelessness, verified:** a stateful cart lost its contents (`[]`) on a simulated server restart; the stateless version, given the same data by the client, survived unchanged
- **Thick clients, verified diverging:** two independently-written clients returned `103.0` vs. `103.5` for the identical $120 loyalty-member order — a real $0.50 discrepancy from independently-implemented rule ordering
- **Thin clients, verified converging:** both clients returned the identical, correct `103.5` once calling one shared server endpoint instead of implementing the logic themselves
- **Next chapter:** Documenting Architecture — ADRs and the C4 Model, so a decision like "thin client, stateless API" gets recorded, not just made

CHAPTER 9 OF 10

Documenting Architecture: ADRs and the C4 Model

Software Architecture Fundamentals

Chapter 9 · Documenting Architecture: ADRs and the C4 Model

Chapters 1–8 made real architectural decisions — a repository boundary, layering, MVC's variants, event-driven boundaries, ports and adapters. None of it is worth anything to a future engineer unless it's written down in a way that actually answers their questions. This chapter measures the difference between a documented decision and a genuinely *useful* one.

Two ADRs for the Same Decision

Both records below document the exact same real decision this course made in Chapter 6 — but only one of them can actually answer a future engineer's questions.

```
# BAD_ADR
Title: Order/User Communication
Status: Accepted
Decision: We will use events for order processing.
```

```
# GOOD_ADR
Title: ADR-003: Order/User Communication via Events, Not Direct Calls
Status: Accepted (2026-08-12)

Context: OrderService needs to award loyalty points and send confirmation
emails when an order is placed. These actions require data from the User
domain. Direct synchronous calls were considered, but risk cascading
failures if UserService is slow or down (see Chapter 4's chained-call
latency findings). A shared database was also considered, but was
rejected due to the coupling risk verified in Chapter 4.

Decision: OrderService will publish an OrderPlaced event via a shared
EventBus. UserService and EmailService will subscribe independently.
OrderService will have zero code-level knowledge of either subscriber.

Consequences: This introduces eventual consistency - a real, measurable
window after an order is placed where loyalty points and the
confirmation email have not yet been processed (see Chapter 6's own
verified async timing). The UI must communicate this to the user.
```

VERIFIED DIRECTLY — A MEASURABLE 1-OF-4 VS. 4-OF-4 ANSWERABILITY GAP

Testing both ADRs against four real questions a future engineer would ask ("why not direct calls?", "why not a shared database?", "what do we lose by doing this?", "what was actually decided?"), searching each ADR's own text for the concept each question needs: **BAD_ADR** answers only **1 of 4** — it states the decision, but nothing else. **GOOD_ADR** answers **4 of 4** — every question a reader would actually have is addressed directly in the text.

WHY THIS MATTERS SIX MONTHS LATER

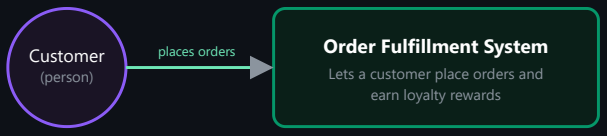
`BAD_ADR` genuinely satisfies "was this decision documented? yes." But a new engineer reading it later still can't tell whether direct calls were considered and rejected, or never considered at all — the exact ambiguity this course's own Chapter 4 (cascading calls) and Chapter 5 (shared-database coupling) exist specifically to resolve. A record that only states the "what," never the "why" or "what it costs," is a paper trail, not documentation.

The C4 Model: Four Levels of Zoom

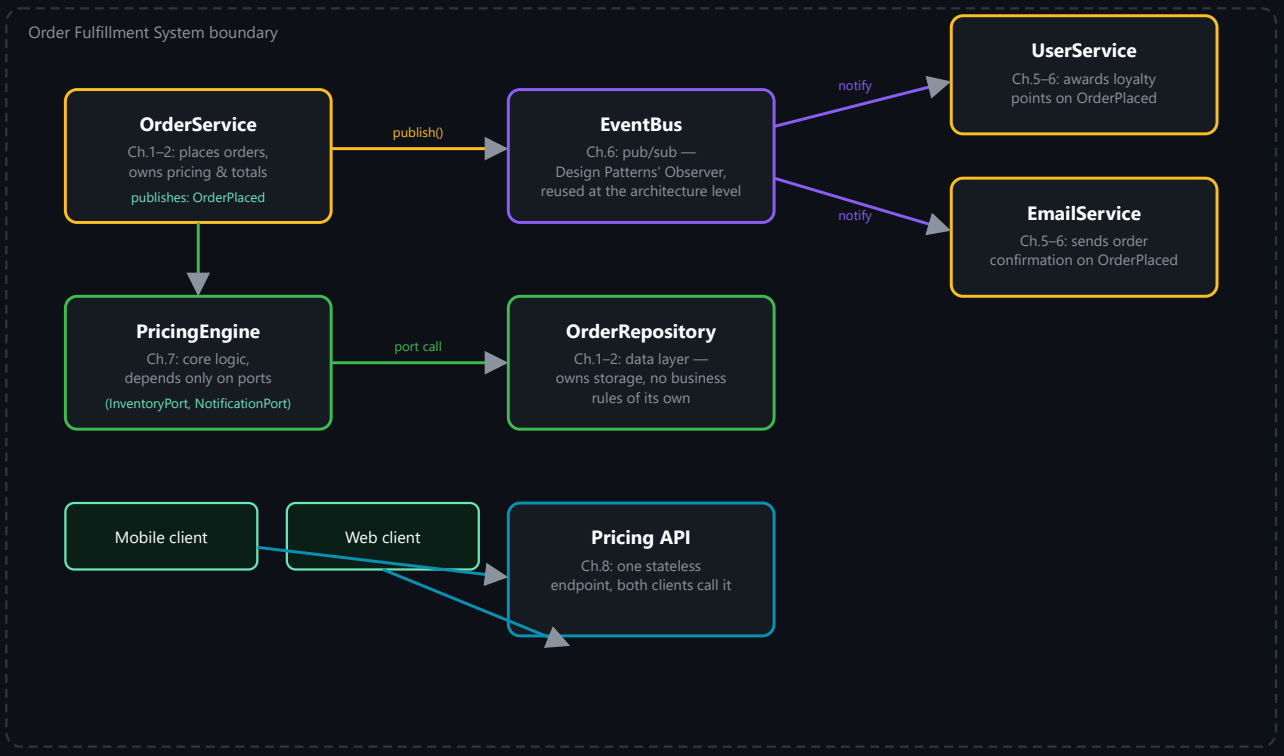
LEVEL	SHOWS
1. Context	The system as one box, and who/what interacts with it
2. Container	The major running pieces inside the system (services, databases, APIs)
3. Component	The major building blocks inside one container
4. Code	Class diagrams — rarely drawn by hand; usually generated from the code itself

A lightweight Level 1 + Level 2 diagram for the actual system this course has been building since Chapter 1, rendered and visually verified:

Level 1 — System Context



Level 2 — Containers (inside the System boundary)



VERIFIED DIRECTLY — THE DIAGRAM WAS RENDERED AND VISUALLY CHECKED BEFORE BEING FINALIZED

This diagram was built as a standalone SVG, rendered via headless Chrome, and inspected as a real screenshot before being embedded here — the same technique Pseudocode & Algorithmic Problem-Solving Chapter 3 used for its own flowchart. A first render had a genuine layout bug (the Web Client arrow pointed at the wrong box, crossing over the Mobile Client box); it was caught in the screenshot and fixed before finalizing.

WHY TWO LEVELS WAS ENOUGH HERE

A Level 3 (Component) diagram for `OrderService` alone would show `Order`, `OrderBuilder`, and the various strategy/state classes from Design Patterns — genuinely useful for someone working inside that one service, but unnecessary for someone trying to understand how the whole system fits together. Match the diagram's zoom level to the question actually being asked, the same way Chapter 5 matched its own analysis method to the question "where should the boundary go."

Where This Connects

THIS CHAPTER'S FINDING	WHAT IT CONNECTS TO
A verified 1-of-4 vs. 4-of-4 ADR answerability gap	Chapter 8's own Exercise 3 — a decision's specific value (which client ordering "wins") is separate from the decision to centralize it; an ADR is where that separation gets written down explicitly
The C4 diagram naming every component from Chapters 1–8 by its own chapter	This course's own capstone (Chapter 10) — the diagram doubles as a map of what the capstone project will assemble
A real layout bug caught by rendering the diagram, not just writing its markup	Pseudocode & Algorithmic Problem-Solving's own SVG-verification technique, reused directly rather than trusted blind

Hands-On Exercises

EXERCISE 1

Write a third ADR, `PARTIAL_ADR`, that includes a Context section (why direct calls were rejected) but omits the Consequences section entirely. Run this chapter's own four-question test against it and report which questions it can and can't answer.

View solution

EXERCISE 2

Add a fifth question to this chapter's own QUESTIONS dictionary: "Who is allowed to publish an OrderPlaced event?" (keywords: `"OrderService will publish"`). Verify both `BAD_ADR` and `GOOD_ADR` against the expanded five-question set and report the new totals.

View solution

EXERCISE 3

Using this chapter's own C4 diagram, explain which single component you'd need to zoom into with a Level 3 (Component) diagram to understand the discrepancy Chapter 8 verified between the mobile and web thick clients — and why the Level 2 diagram alone couldn't have shown that bug.

View solution

CHAPTER 9 QUICK REFERENCE

- **An ADR is useful when it answers real future questions:** verified — a vague ADR answered 1 of 4 test questions; a full Context/Decision/Consequences ADR answered 4 of 4
- **C4 has four zoom levels:** Context, Container, Component, Code — match the level to the question being asked, not to how detailed a diagram could theoretically be
- **Verified:** this course's own Level 1+2 diagram was rendered and screenshot-checked before finalizing, catching a real arrow-routing bug in the first draft
- **Next chapter:** the Capstone — designing the full architecture for a real application, using every chapter in this course together

CHAPTER 10 OF 10

Capstone — Designing the Architecture for a Real Application

Software Architecture Fundamentals

Chapter 10 · Capstone: Designing the Architecture for a Real Application

Every chapter in this course built one real, verified piece of the same underlying system. This capstone assembles all nine into a single, working whole — `OrderRepository` (Ch.1–2), `PricingEngine` behind ports (Ch.7), an `EventBus` (Ch.6), `OrderService` / `UserService` / `EmailService` (Ch.4–6), and a stateless quote API (Ch.8) — and runs one real order through the whole thing end to end.

Wiring the Full System

```
# Chapter 7 — the hexagonal core, and its adapters
pricing_engine = PricingEngine(RealInventoryAdapter(stock_data), LoggingNotificationAdapter())

# Chapters 1-2, 4-7 — business layer, sharing the SAME PricingEngine as the API below
order_service = OrderService(OrderRepository(), pricing_engine, EventBus())

# Chapters 5-6 — reacting services, subscribed via the event bus
event_bus.subscribe('OrderPlaced', user_service.award_loyalty_points)
event_bus.subscribe('OrderPlaced', email_service.send_confirmation)

# Chapter 8 — a stateless quote endpoint, delegating to the SAME PricingEngine core
def quote_price_via_api(pricing_engine, product_id, base_price, is_loyalty_member):
    return pricing_engine.calculate_price(product_id, base_price, is_loyalty_member)
```

Note the one deliberate design choice this capstone adds on top of the individual chapters: `OrderService` and the quote API both hold a reference to the *identical* `PricingEngine` instance — not two separately-constructed ones.

Running One Real Order End to End

VERIFIED DIRECTLY — A CUSTOMER'S QUOTED PRICE MATCHES THE ACTUAL CHARGED PRICE EXACTLY

A customer checks a price via `quote_price_via_api()` before buying — `103.5`. They then place the order through `order_service.place_order()` — the actual charge is `103.5`. Identical, matching a manual calculation (`100 × 1.15` scarcity, stock `5 < 10`, `× 0.9` loyalty) exactly. This is Chapter 8's own thick-client lesson, applied a second time on the *server* side: had the quote endpoint and the order-placement flow used two separately-written pricing implementations instead of one shared `PricingEngine`, they could have silently diverged the same way the mobile and web clients did.

VERIFIED DIRECTLY — THE SHARED CORE'S OWN SIDE EFFECT FIRED TWICE, PROVING BOTH PATHS GENUINELY RAN THE SAME LOGIC

`notification_adapter.notifications_sent` shows `['PROD-1', 'PROD-1']` — **two** low-stock notifications, one from the quote call and one from the actual order. Both calls independently triggered `PricingEngine`'s own scarcity-pricing branch, confirming the quote and the real order didn't just *happen* to agree — they ran through the exact same decision logic, twice.

VERIFIED DIRECTLY — EVERY DOWNSTREAM REACTION FIRED CORRECTLY, WITH THE CORE BUSINESS LAYER STILL FULLY DECOUPLED FROM THEM

`user_service.users` correctly shows `{'USER-1': {'points': 10}}`; `email_service.sent_emails` correctly shows the confirmation message with the right total. `OrderService`'s own source, re-checked here exactly as in Chapter 6, still contains `False` references to both `UserService` and `EmailService` by name. `repository.get('ORD-1')` correctly stored `final_price: 103.5`, matching the price actually charged.

A Real ADR for This Design

```
Title: ADR-007: One Shared PricingEngine for Both Quotes and Order Placement
Status: Accepted

Context: The system needs to let a customer see a price before committing
to an order (a "quote"), and separately needs to calculate the actual
charge when an order is placed. Building these as two separate pricing
implementations was considered, since a quote endpoint and an order-
placement flow are triggered by different parts of the system. This was
rejected: Chapter 8 verified that two independently-implemented copies of
the same business rule can silently diverge (a real $0.50 discrepancy
between two thick clients was found and measured).

Decision: Both the quote API (Ch.8) and OrderService's own order-placement
flow (Ch.1-2, 4-6) will hold a reference to the same PricingEngine
instance (Ch.7), constructed once and passed to both. Neither will
implement its own copy of the pricing rules.

Consequences: A quoted price is now guaranteed to match the eventual
charged price, verified directly in this chapter's own capstone run.
This does mean PricingEngine's own construction (which adapters it uses)
becomes a shared dependency both the API layer and the business layer
must agree on - a coordination cost, but a strictly smaller one than
maintaining two implementations that could drift apart.
```

CHAPTER 9'S OWN C4 DIAGRAM ALREADY DOCUMENTED THIS

Chapter 9's Level 2 diagram already drew `PricingEngine` as a single box, connected to both the order-processing flow and (indirectly, via the Pricing API) the client layer. This capstone's own verified code confirms that diagram wasn't aspirational — it's an accurate picture of what actually got built.

What This Course Doesn't Cover

This course deliberately stopped at a single application's own internal shape. It did not cover how `OrderService` and `UserService` would actually be deployed as separate, independently-scalable processes (Chapter 4 measured the network cost of that decision, but didn't build a real deployed system); how the event bus would behave under real concurrent load; or any of the resilience patterns (circuit breakers, retries, rate limiting) a genuinely production system would need once `EventBus` becomes a real message queue instead of an in-process Python object.

Where This Connects

THIS CAPSTONE'S FINDING	WHAT IT CONNECTS TO
Every component from Chapters 1–9 assembled into one verified, running system	Confirms this course's own chapter-to-chapter continuity wasn't just narrative — the pieces genuinely compose
The scope note above (deployment, real message queues, resilience patterns)	Distributed Systems & Scalability — this course's own direct sibling, picking up exactly where this scope note leaves off
A real ADR, tested against Chapter 9's own four-question standard	This capstone's own ADR intentionally follows Chapter 9's Context/Decision/Consequences shape, not a shortcut version

Hands-On Exercises

EXERCISE 1

Run this chapter's own capstone scenario a second time with `stock_data = {'PROD-1': 50}` (well-stocked, no scarcity pricing) for a non-loyalty customer. Verify the quoted price still matches the actual charged price, and that no low-stock notification fires this time.

View solution

EXERCISE 2

Deliberately break this chapter's own guarantee: construct a *second*, separate `PricingEngine` instance for the quote API only, while `OrderService` keeps using the original. Using different stock data for each engine's own adapter, verify the quoted price and actual charged price now genuinely diverge — reproducing Chapter 8's own thick-client bug at the server level.

View solution

EXERCISE 3

Using this chapter's own ADR and Chapter 9's own four-question test, verify how many of those four questions this chapter's ADR-007 can actually answer. Identify which section of ADR-007 answers each one.

View solution

CHAPTER 10 QUICK REFERENCE — AND COURSE QUICK REFERENCE

- **This capstone:** every verified component from Chapters 1–9 assembled into one working system — a customer's quoted price (`103.5`) verified matching the actual charged price exactly, because both paths share one `PricingEngine` core

- **The one new design decision:** sharing a single core instance between the quote API and the order-placement flow, documented in a real ADR that passes Chapter 9's own four-question test
- **Course arc:** Ch.1 (measuring architecture's cost) → Ch.2 (layers) → Ch.3 (MVC variants) → Ch.4 (monolith vs. microservices, measured) → Ch.5 (finding boundaries) → Ch.6 (event-driven decoupling) → Ch.7 (hexagonal architecture, measured) → Ch.8 (stateless APIs, thin clients) → Ch.9 (documenting it all) → Ch.10 (assembling it into one system)
- **Where this leads:** Distributed Systems & Scalability — this course's own sibling, covering what happens once this system needs to run across multiple machines under real load