BE SOFTWARE ARCHITECTURE WISE · CHAPTER 0

Start Here: How Software Works

This chapter is the on-ramp to the course. It assumes you have used apps and websites but never programmed, and it explains, from the ground up, what software is, how it represents and processes data, how code is organised, how computers talk across networks, where data is stored, how separate systems work together through APIs, what servers and the cloud are, and how teams build and release software. Experienced engineers can skim it as a shared vocabulary; everyone else should read it before Chapter 1, because every later chapter builds on these ideas.

bewiseapps.com/software-architecture-wise/foundations/ · Version 2026-10-26. Educational information only, not professional advice.

AT A GLANCE

You will learn to

- Explain what happens, step by step, when a program runs, from source code through the operating system to the processor.
- Distinguish the common data structures and use Big-O notation to predict how an operation slows down as data grows.
- Describe how functions, modules, objects and interfaces divide code, and the difference between a library and a framework.
- Trace what happens when someone types a web address, naming DNS, TCP, TLS and HTTP and what each contributes.
- Compare files, relational databases and the main NoSQL families, and explain what a transaction guarantees.
- Explain what an API is, how a REST call over HTTP works and when an asynchronous message is a better fit than a synchronous call.
- Distinguish physical servers, virtual machines, containers and the main cloud service models.
- Follow a single user action through every layer of a real system, and recognise the decisions that make up its architecture.

Before you start

None. Familiarity with using apps and websites is enough.

Time

About 3 hours to read; 45 minutes for the test.

Key terms

Process · Compiler and interpreter · Data structure · Big-O notation · Interface · Framework · Dependency · IP address and port · DNS · HTTP and HTTPS · Latency · Database and transaction · API · Container and cloud

Why start here

Software architecture is about the big decisions in a software system: what its main parts are, how they talk to each other, where data lives and which qualities, such as speed, reliability and security, the design must achieve. Those decisions are made in a vocabulary: processes and threads, indexes and caches, interfaces and dependencies, HTTP and DNS, transactions, APIs, containers and pipelines. Architects use these words precisely, and much of the course depends on knowing what each one really means and what it costs.

This chapter supplies that vocabulary. It is written for someone who has never programmed, and every section opens with a short everyday analogy before explaining the real mechanism. It is also meant to be accurate enough that an experienced engineer reading it would agree

with every statement. Where a topic deserves much more depth, the chapter introduces it and links forward to the later section that treats it properly.

One program on one computer

The first three sections stay inside a single machine. What software is explains the difference between hardware and software, how source code written by people becomes instructions a processor can run (through compilers, interpreters and virtual machines), what the operating system does, and why data in memory disappears on a restart while data on disk survives. It ends by following a nine-line Python program through every step of its life.

Data and algorithms shows how computers represent numbers and text, and introduces the everyday data structures: lists, dictionaries, sets, queues, stacks, trees and graphs. Using Big-O notation, it explains why one design for finding a customer stays fast at a million customers while another collapses, and why indexes and caches exist. Organising code then climbs from functions to modules, objects and interfaces, explains the difference between a library and a framework, warns about the hidden cost of dependencies, and gives a first look at coupling and cohesion, the two ideas that run through the whole of software design.

Many computers working together

The next three sections leave the single machine. How computers talk explains packets, IP addresses and ports, TCP and UDP, how DNS turns names into addresses, and how HTTP requests and responses work, secured by TLS. It also makes a point that surprises many people: latency has a floor set by the speed of light, so a round trip between London and Sydney can never be fast, however much bandwidth you buy.

Storing data covers files and databases: tables, keys and SQL queries, indexes, transactions and the ACID guarantees (illustrated with a bank transfer that must never half-happen), and the main NoSQL families and when each fits. APIs and integration explains how separate systems cooperate: an API as a contract, REST and JSON over HTTP, keys and tokens, webhooks, the difference between waiting for an answer and sending a message through a queue, and why breaking changes to an API hurt everyone who depends on it.

Where software runs and how it is built

Servers and the cloud shows what a server really is, how virtual machines and containers let many workloads share one physical machine, and what cloud providers sell: from renting raw machines to running your code only when an event arrives. It introduces regions, availability zones, content delivery networks and the two ways to handle growth, a bigger machine or more machines. How software gets built turns from technology to practice: requirements, version

control with Git, the testing pyramid, environments, continuous integration and delivery, agile ways of working and DevOps. It closes by following one tap on "Place order" in a food-delivery app through every part of the chapter, from the phone to DNS, HTTPS, a load balancer, services, a database transaction, a payment provider, a queue and a push notification.

The idea to take away

Every system in this chapter is built from parts that hide their details behind an agreed boundary: the operating system behind system calls, a module behind its interface, a web server behind HTTP, a database behind SQL, a payment provider behind its API, a cloud platform behind its services. Boundaries are what let people build large systems without understanding every detail, and they are where the important trade-offs sit: a function call is fast and reliable, while a call across a network is slower and can fail; a dependency saves effort but brings risk; a cache saves time but can serve stale data.

Deciding where those boundaries go, what crosses them and which trade-offs to accept is software architecture. Each section ends with a short note on why its subject matters for architecture, and Chapter 1 begins exactly where this chapter ends.

SECTION 0.1 OF 0.8 · BEGINNER · 15 MIN READ

Programs, code and how a computer runs them

THE IDEA IN BRIEF

A computer is like a restaurant kitchen. The chef (the processor) can only follow written recipes, and follows them exactly, word for word. The worktop (memory) holds what the chef is using right now and is cleared every night; the pantry (storage) keeps things for good. Software is the set of recipes. People write recipes in a language they can read, and a translator turns them into the chef's own very simple language. The kitchen manager (the operating system) decides which recipe gets the chef's attention and keeps the stations from getting in each other's way.

WHAT YOU NEED TO KNOW

- Hardware is the physical machine (CPU, RAM, storage, network card); software is the set of instructions and data that tell it what to do.
- A program is a list of precise instructions; the computer does exactly what it says, so mistakes in the instructions become bugs.
- Programmers write source code in high-level languages, which compilers, interpreters or bytecode virtual machines turn into machine code the CPU can run.
- The operating system shares the CPU, memory, files and network between programs and enforces security; programs ask it for services through system calls.
- A running program is a process with its own memory; threads inside a process share that memory, which is useful but can cause concurrency bugs.
- RAM is fast but loses its contents on restart; anything that must not be lost has to be written to durable storage, usually a database.

Hardware and software

A computer is a machine that follows instructions very quickly. The physical parts you could touch are the hardware; the instructions that tell those parts what to do are the software. A laptop, a phone, the server that runs your bank's website and the chip inside a smart thermostat are all computers built from the same four kinds of part.

The CPU (central processing unit, or processor) carries out instructions. A modern processor has several *cores*, each able to work through its own stream of instructions, and each core runs billions of simple steps per second. RAM (random-access memory, usually just called *memory*) holds the instructions and data the CPU is working on right now. It is fast, but it is volatile:

switch the power off and its contents vanish. **Storage** (a solid-state drive or a spinning hard disk) keeps files when the power is off, but is much slower to reach. A **network card** (wired or Wi-Fi) sends and receives data to and from other computers.

Think of a restaurant kitchen. The chef is the CPU: the only one who actually cooks. The worktop is RAM: everything the chef is using at this moment is laid out there, within arm's reach, but it is cleared at the end of the night. The pantry and freezer are storage: large, permanent and a walk away. The hatch to the dining room is the network card, through which orders arrive and dishes leave. The recipes are the software.

The four parts every computer has

Part	What it does	Kitchen analogy	What happens if it is the bottleneck
CPU (processor)	Executes instructions: arithmetic, comparisons, moving data	The chef	Work queues up; pages and reports become slow even though there is spare memory
RAM (memory)	Holds running programs and the data they are using	The worktop	The system starts shuffling data to disk and slows sharply, or programs are stopped
Storage (SSD or disk)	Keeps files and databases permanently	The pantry and freezer	Saving and loading data becomes slow; a full disk can stop a database
Network card	Sends and receives data to other machines	The hatch to the dining room	Requests wait to be sent or received; users see delays

What a program is

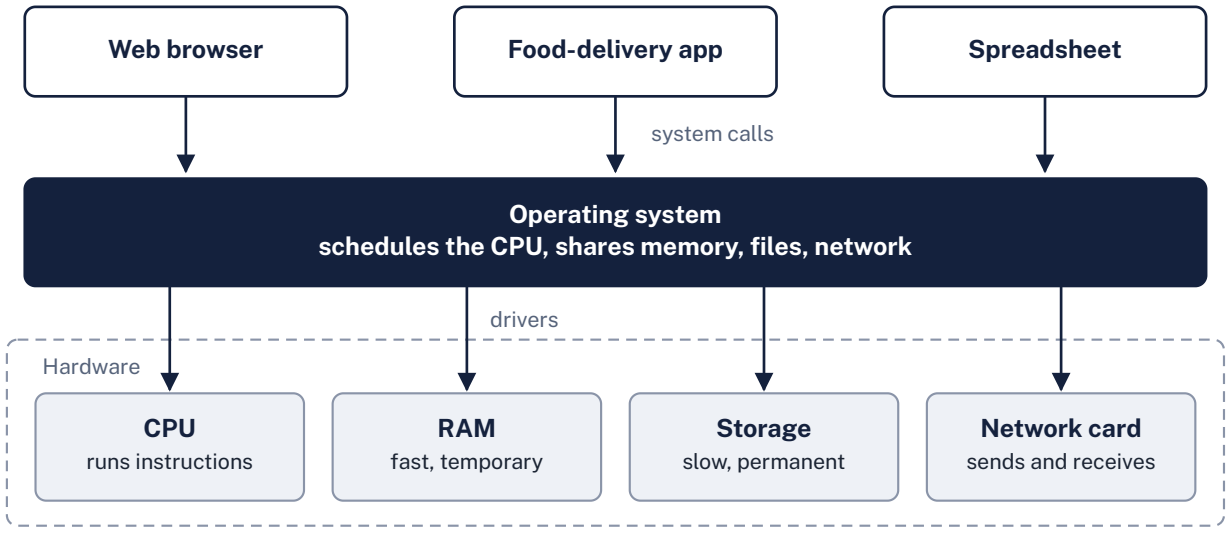
A program is a list of precise instructions that a computer follows to do a job: calculate a bill, display a web page, send a payment. The word that matters is *precise*. A human cook can cope with "add salt to taste"; a computer cannot. It does exactly what it is told, in exactly the order it is told, and nothing more. If an instruction says to divide by a number that turns out to be zero, the computer does not shrug and carry on: the program stops with an error, unless the programmer wrote further instructions saying what to do in that case.

At the lowest level the CPU repeats one cycle for as long as it is switched on: fetch the next instruction from memory, decode what it means, execute it, and move on to the next one. Each instruction is tiny: add two numbers, compare two values, copy a value from one place in

memory to another, jump to a different instruction if a comparison was true. Everything a computer does, from playing a film to training an AI model, is built from billions of these small steps.

Software is the broader term: programs together with the data, configuration and documentation they need. An *application* (or app) is software that does something useful for a person, such as a banking app. *System software*, such as the operating system described below, exists mainly to run other software.

Layers: applications, the operating system and the hardware



Applications never drive the hardware directly: they ask the operating system, which shares the hardware fairly and safely between them.

Source code and programming languages

CPUs understand only machine code: instructions encoded as numbers, specific to a family of processors. Intel and AMD chips in most laptops and servers use one family (x86-64); the chips in phones, recent Macs and many cloud servers use another (Arm). Machine code that runs on one does not run on the other. Writing machine code by hand is slow, error-prone and almost never done today.

Instead, people write source code in a **programming language**: a precisely defined notation that reads partly like English and partly like mathematics. Python, JavaScript, Java, C#, Go, Rust and C are widely used examples. These are called *high-level* languages because they let a programmer say "add up the prices in this basket" in one or two lines instead of dozens of machine instructions, and because they hide details of the particular CPU. Source code is plain text, saved in files, and it is what programmers read, write, review and store in version control (a system that keeps a history of every change, explained in the last section of this chapter).

Languages differ in what they make easy. Python is valued for readability and is dominant in data science and AI work; JavaScript is the language every web browser runs; Java and C# are common in large business systems; Go is popular for network services; C and Rust are used where every microsecond and byte matters, such as operating systems and databases. No language is best at everything, and choosing one is partly a technical decision and partly a question of which skills a team has.

From source code to a running program: compilers, interpreters and virtual machines

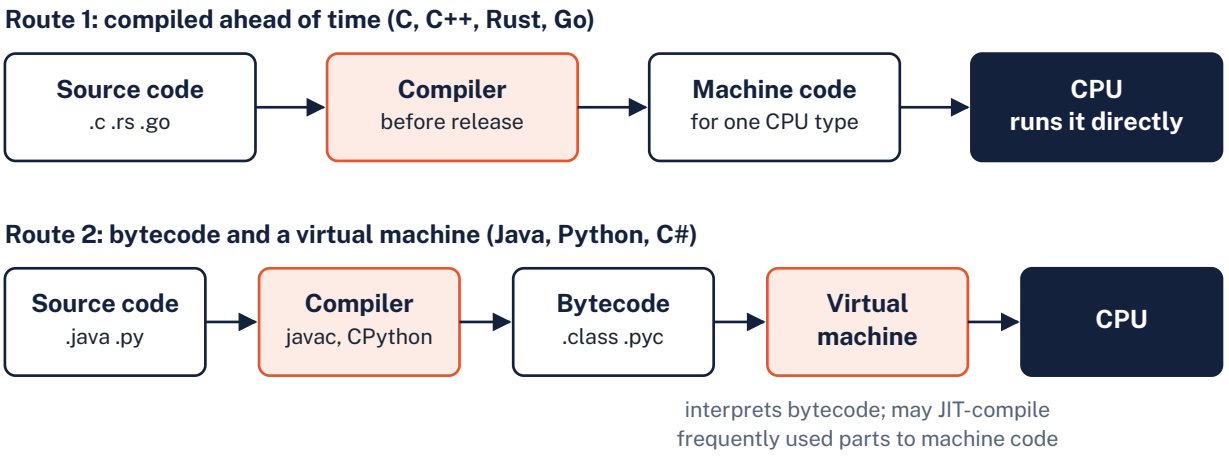
Source code has to be turned into something the CPU can execute. There are three broad approaches, and real languages often mix them.

A compiler is a program that translates the whole of a program's source code into another form before it runs. Languages such as C, C++, Rust and Go are usually compiled *ahead of time* into machine code for a particular CPU and operating system. The result is an *executable* file that starts quickly and runs fast, but you must compile a separate version for each kind of machine you want to support, and the compiler rejects the program if it finds certain kinds of mistake.

An interpreter is a program that reads source code (or something close to it) and carries out its instructions as it goes, one step at a time. Interpretation makes it easy to try things out and to run the same program on any machine that has the interpreter, at the cost of speed.

Many popular languages sit in between. Java source code is compiled by a tool called `javac` into bytecode: compact instructions for an imaginary, idealised computer rather than for a real CPU. The **Java Virtual Machine** (JVM) then runs that bytecode on whatever real machine it is installed on, which is how one Java program runs on Windows, macOS and Linux unchanged. The JVM watches which parts of the program run most often and compiles those parts into machine code while the program is running, a technique called *just-in-time* (JIT) compilation. C# on .NET works in a similar way. The standard Python implementation, CPython, also compiles your `.py` files to bytecode (caching the bytecode of imported modules in `.pyc` files) and then runs that bytecode in its own interpreter loop.

Two routes from source code to the CPU



Compiling ahead of time produces machine code for one kind of CPU; compiling to bytecode lets one file run anywhere a suitable virtual machine is installed.

WATCH OUT

Two meanings of "virtual machine"

The language virtual machines described here (the JVM, the .NET runtime, the CPython interpreter) are programs that run bytecode. They are different from the virtual machines that cloud providers rent out, which imitate a whole computer so that one physical server can be split into several. Both are called virtual machines; the second kind is explained in Servers and the cloud.

Comparing the approaches

Approach	Typical languages	Strengths	Costs
Ahead-of-time compilation to machine code	C, C++, Rust, Go	Fast start-up and execution; one self-contained file to deploy	Build a separate version per CPU and operating system; slower edit-and-try cycle
Bytecode plus a virtual machine with JIT	Java, Kotlin, C#	One build runs on many platforms; JIT can make long-running services very fast	The runtime must be installed; slower start-up and extra memory while the JIT warms up
Bytecode plus an interpreter	Python (CPython), Ruby	Quick to write and try; very portable	Usually slower for heavy computation; speed-critical parts are often written in C and called from Python
JIT-compiled in the browser or server	JavaScript (V8 in Chrome and Node.js)	Runs everywhere a browser runs; fast enough for most web work	Performance can be less predictable than compiled code

The operating system's job

When you switch on a computer, the first substantial program to start is the operating system (OS): Windows, macOS, Linux, Android or iOS. Most servers on the internet run Linux. The operating system has four main jobs.

First, it **shares the CPU**. A laptop may have a few dozen programs running but only a handful of cores, so the OS switches each core between programs many times a second, fast enough that they all appear to run at once. Second, it **manages memory**, giving each program its own region and stopping one program from reading or overwriting another's. Third, it provides **files and devices**: programs ask the OS to open a file, draw on the screen or send data over the network, and the OS, through small programs called *drivers*, deals with the particular hardware. Fourth, it **enforces security**: users, permissions and the rule that an app may not read your photos unless you allow it.

Programs make these requests through *system calls*, a fixed set of services the OS offers. This is the first example in this course of an important architectural idea: a layer that hides messy details behind a stable set of requests. The person writing a food-delivery app does not need to know which brand of network chip the phone has.

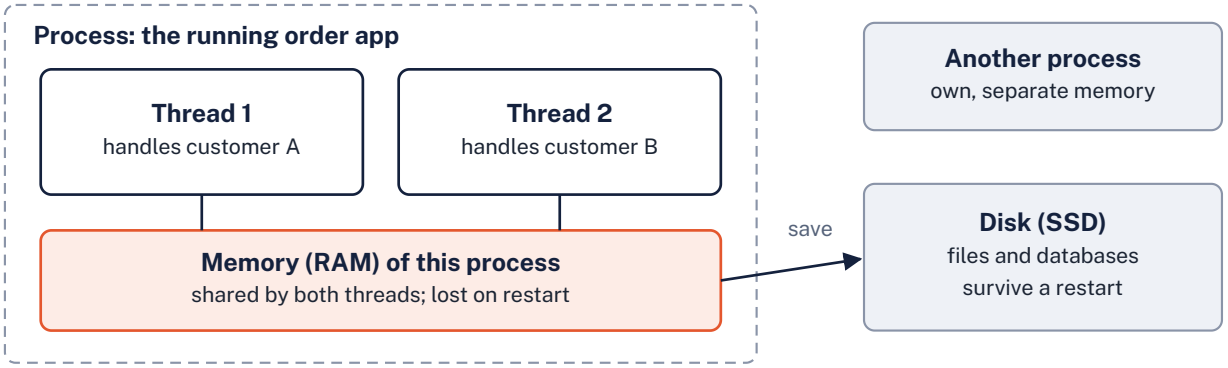
Processes, threads, memory and disk

A program file sitting on disk is just stored instructions. When you start it, the OS creates a process: a running instance of the program with its own private memory, its own identity and its own share of CPU time. Open two browser windows and you may have several processes; start a web server and it may run as one process or many.

Inside a process there can be several threads. A thread is one sequence of instructions being carried out. A process with several threads can do several things at once, for example handle two customers' requests at the same time, and all its threads share the process's memory. That sharing is convenient but risky: if two threads change the same piece of data at the same moment without coordination, the result can be wrong in ways that appear only occasionally. These *concurrency bugs* are among the hardest to find, which is one reason many systems prefer separate processes, or designs where each request is handled independently.

The difference between memory and disk explains a great deal of software design. Data in RAM is quick to reach, roughly a tenth of a microsecond, but it disappears when the process ends, crashes or the machine restarts. Data on disk survives, but reaching it takes far longer: tens of microseconds or more on a solid-state drive, and several milliseconds on a spinning disk. So any information that must not be lost, such as an order or a payment, has to be written to durable storage, usually through a database, before the system tells the customer "done".

A process, its threads, its memory and the disk



Threads in one process share its memory; separate processes do not; only what is saved to disk survives a restart.

```
PRICE_PER_ITEM = 12.50   # pounds
VAT_RATE = 0.20          # 20 per cent

def total_with_vat(quantity):
    subtotal = quantity * PRICE_PER_ITEM
    return subtotal * (1 + VAT_RATE)

quantity = int(input("How many items? "))
print("Total to pay:", round(total_with_vat(quantity), 2))
```

A tiny Python program: work out the price of an order including VAT

WORKED EXAMPLE

Worked example: what happens when the program runs

Suppose the code above is saved as `order.py` and you type `python order.py`, then enter 3.

1. Start	The operating system creates a new process, gives it memory and starts the Python interpreter (itself a compiled program) inside it.
2. Compile	CPython reads `order.py` from disk, checks that it is valid Python and compiles it to bytecode. A missing bracket or colon would be reported here as a *syntax error*, before anything runs.
3. Define	The interpreter runs the bytecode from the top. It stores 12.5 and 0.2 in memory under the names `PRICE_PER_ITEM` and `VAT_RATE`, and records the function `total_with_vat` without running it yet.
4. Input	`input()` asks the OS to read from the keyboard. The process waits, using no CPU, until you type 3 and press Enter. `int()` turns the text "3" into the number 3.
5. Calculate	The function runs: 3 times 12.5 is 37.5, then 37.5 times 1.2 is 45.0. Each step is a handful of bytecode instructions, each of which becomes many machine instructions on the CPU.
6. Output	`print()` asks the OS to show `Total to pay: 45.0` on the screen.
7. Finish	The program reaches its last line, the process ends and the OS reclaims its memory. Nothing was saved to disk, so the result is gone unless you copied it.

Even this nine-line program relies on the operating system for memory, keyboard and screen, and on a language runtime to translate it for the CPU.

If you typed `three` instead of 3, `int()` would fail with a *runtime error* and the program would stop: the code never said what to do with bad input.

The program keeps money in a floating-point number to stay short. Real financial systems store whole pence or use a decimal type, for reasons the next section explains.

Bugs: when the instructions are wrong

A bug is a mistake in software that makes it behave differently from what was intended. Because computers follow instructions exactly, almost every bug is a gap between what the programmer meant and what the programmer actually wrote, or a situation nobody thought about.

It helps to recognise three kinds. A **syntax error** breaks the language's grammar (a missing bracket), so the compiler or interpreter refuses to run the program at all; these are found immediately. A **runtime error** happens while the program runs, such as dividing by zero or trying to read a file that does not exist; the program usually stops with an error message. A **logic error** is the most dangerous: the program runs happily and gives the wrong answer, for example charging VAT twice. Nothing crashes, so only testing, careful review or an unhappy customer will reveal it.

Professional teams expect bugs and build defences: automated tests, code review, gradual releases and monitoring that notices when error rates rise. Many of these defences are architectural, and they appear throughout this course.

WHERE THIS CAME FROM

The moth in the machine

Engineers were calling faults "bugs" long before computers; Thomas Edison used the word in the 1870s. The best-known computing story dates from 9 September 1947, when operators of the Harvard Mark II computer found a moth trapped in one of its relays, taped it into the logbook and wrote "First actual case of bug being found". The joke works precisely because the term was already in use. Grace Hopper, who worked on the machine and later shaped early programming languages, popularised the story.

GOING DEEPER

Going deeper: you can see the bytecode

Python's standard library includes a module called `dis` that shows the bytecode for any function. Running `dis.dis(total_with_vat)` on the example above lists instructions such as `LOAD_FAST` (fetch a local variable), `LOAD_GLOBAL` (fetch a module-level name), `BINARY_OP` (multiply or add) and `RETURN_VALUE`. The exact instructions change between Python versions, which is why `.pyc` files are tied to the version that produced them.

Why this matters for architecture

Architecture is about the big decisions that are expensive to change, and several of them rest on the ideas in this section. The choice of language and runtime affects start-up time, memory use, which libraries you can use and whom you can hire. The split between memory and disk decides what must be stored durably, and how a system recovers after a crash. Processes and threads are the units that run, fail and scale: a later chapter asks whether a system should be

one process or many cooperating ones. And the operating system's layered design, hiding hardware behind a stable set of requests, is the first instance of a principle you will meet again as information hiding.

The next section looks at what programs actually work on: data, and the structures and algorithms that make working with it fast or slow.

CHECK YOUR UNDERSTANDING

▼ Why does a program's data disappear when it crashes, unless it was saved?

While a program runs its data lives in RAM, which belongs to the process and is cleared when the process ends or the machine restarts. Only data written to storage such as a disk or database survives.

▼ What is the difference between a compiler and an interpreter?

A compiler translates the whole program into another form (machine code or bytecode) before it runs; an interpreter carries out the instructions as it reads them. Many languages, such as Java and Python, compile to bytecode first and then run that bytecode in a virtual machine.

▼ Why can a Java program built on a Windows laptop usually run unchanged on a Linux server?

Java is compiled to bytecode, not to machine code for one CPU. Any machine with a Java Virtual Machine installed can run that bytecode.

▼ Why do programs ask the operating system to read a file or send data, instead of controlling the disk or network card themselves?

The operating system shares hardware safely between programs, enforces permissions and hides the details of each device behind system calls, so applications do not need to know which hardware they are running on.

SOURCES

- The Python Tutorial (Python documentation): https://docs.python.org/3/tutorial/index.html
- Python glossary: bytecode, interpreter and related terms: https://docs.python.org/3/glossary.html
- dis: disassembler for Python bytecode: https://docs.python.org/3/library/dis.html
- The Java Virtual Machine Specification, Java SE 21 Edition: https://docs.oracle.com/javase/specs/jvms/se21/html/index.html

SECTION 0.2 OF 0.8 · BEGINNER · 16 MIN READ

Data, data structures and algorithms, and why speed depends on them

THE IDEA IN BRIEF

Imagine looking for one letter in a post office. If the letters are in a heap, you must check them one by one, and a bigger heap takes longer. If they are sorted into pigeonholes by postcode, you walk straight to the right hole, however many letters there are. Data structures are the ways of arranging data (heaps, pigeonholes, queues, piles), and algorithms are the methods for working with them. Choosing them well decides whether a system stays fast as it grows.

WHAT YOU NEED TO KNOW

- Computers store everything as bits; a byte is eight bits, and data types (integers, floats, booleans, strings) say how to read them.
- Floating-point numbers are approximate, so money should be stored as integers of the smallest unit or with a decimal type.
- Text is stored as Unicode characters encoded as bytes, usually in UTF-8, where one character can take one to four bytes.
- Data structures such as lists, dictionaries, sets, queues, stacks, trees and graphs each make different operations fast.
- Big-O notation describes how an algorithm's work grows with data: O(1), O(log n), O(n), O(n log n), $O(n^2)$.
- Indexes and caches trade extra space (and the work of keeping them current) for faster answers.

Everything is bits

Computers store everything as bits: binary digits that are either 0 or 1. Physically a bit might be a tiny electrical charge, a magnetic region on a disk or a pulse of light in a fibre, but to software it is just one of two values. Eight bits make a byte, which can hold $2^8 = 256$ different patterns, from 00000000 to 11111111.

Sizes build up from there. A kilobyte is about a thousand bytes, a megabyte about a million, a gigabyte about a billion and a terabyte about a trillion. (Strictly, the standard prefixes kilo, mega and giga mean powers of 1,000; memory is often measured in powers of 1,024, for which

the units kibibyte, mebibyte and gibibyte exist. The difference rarely matters for design but explains why a "1 TB" disk shows less in some file managers.) A page of plain text is a few kilobytes, a phone photo a few megabytes, an hour of high-definition video a few gigabytes.

Bits mean nothing on their own. The pattern 01000001 could be the number 65, the letter A, part of a colour in a photo or part of an instruction for the CPU. What gives bits meaning is the agreement about how to read them, which is the idea of a data type.

Data types: numbers, text and true or false

A data type tells a program what kind of value a piece of data is, and therefore what operations make sense. Every programming language has a few basic types, and Python's are typical.

Integers are whole numbers such as 3, 0 or -42, used for counting things: items in a basket, seats on a flight. **Floating-point numbers** (floats) such as 12.5 or 0.2 represent decimals approximately, in binary. Because many decimals have no exact binary form, `0.1 + 0.2` in Python gives `0.30000000000000004`. That is fine for physics and graphics but wrong for money, which is why financial systems store amounts as whole numbers of pence or cents, or use a decimal type (Python has `decimal.Decimal`). **Booleans** have just two values, `True` and `False`, and record yes-or-no facts such as "has this order been paid?". **Strings** are text, such as a customer's name. Real systems add more types on top: dates and times, money, and binary data such as images.

Types matter beyond a single program. When two systems exchange data, they must agree on types as well as on values: is a postcode a number or text (text: some begin with letters, and some numeric codes begin with zero)? Is a date in day-month or month-day order, and in which time zone? Many expensive integration bugs are type disagreements.

Text and UTF-8

Text is stored by giving every character a number and then storing the numbers as bytes. The international standard **Unicode** assigns a number, called a code point, to well over a hundred thousand characters from the world's writing systems, plus symbols and emoji. An *encoding* then decides how those numbers become bytes. Today the dominant encoding, on the web and in most new systems, is UTF-8.

UTF-8 uses a variable number of bytes per character: one byte for the basic English letters, digits and punctuation (so text that uses only these characters is byte-for-byte identical in UTF-8 to the much older ASCII encoding), two bytes for characters such as `é`, three for characters such as `€` and most Chinese, Japanese and Korean characters, and four for emoji

and some rarer scripts. This has a practical consequence: the number of characters in a string and the number of bytes it occupies can differ. A database column limited to 50 bytes may not hold a 50-character name written in Greek.

The classic text bug is garbled characters (for example `CafÃ©` instead of `Café`), which appears when bytes written in one encoding are read as another. The cure is simple to state: use UTF-8 everywhere, and say so explicitly whenever text crosses a boundary, such as a file, a database or a network message.

Data structures: ways of arranging data

Programs rarely deal with one value at a time. A data structure is a way of organising many values in memory so that the operations you need are quick. The right choice can make the difference between a page that loads instantly and one that never loads. Each has an everyday analogy.

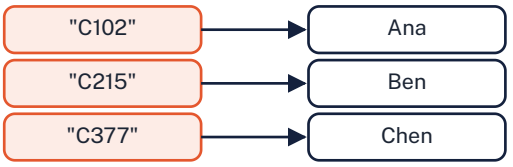
A **list** (called an *array* in many languages) keeps items in order, like a numbered shopping list. Getting the fifth item is instant, because the computer can calculate where it is. Finding an item by its value means checking items one by one. A dictionary (also called a *map* or *hash table*) stores values under unique keys, like a coat check: hand over ticket C377 and you get the right coat at once, without looking at every coat. A **set** is a collection with no duplicates, like a guest list, and answers "is this person on it?" quickly.

A **queue** works like a line at a post office counter: first in, first out. Systems use queues for work waiting to be done, such as orders waiting for a kitchen. A **stack** works like a pile of plates: last in, first out. Your browser's Back button uses one: the page you visited most recently is the first one you return to. A **tree** has a single root with branches beneath it, like a family tree or the folders on your computer; trees are also how databases keep data sorted for fast searching. A **graph** is items (nodes) joined by links (edges) in any pattern, like a road map or a social network, and answers questions such as "what is the shortest route?" or "who are friends of friends?".

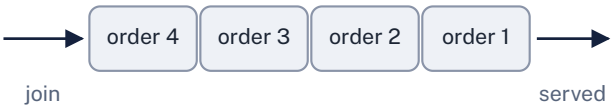
Six common data structures

List (array): items in order, found by position

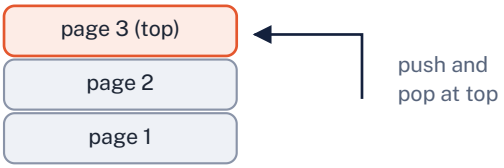
Dictionary (map): value found by key



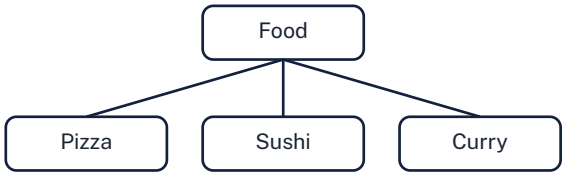
Queue: first in, first out



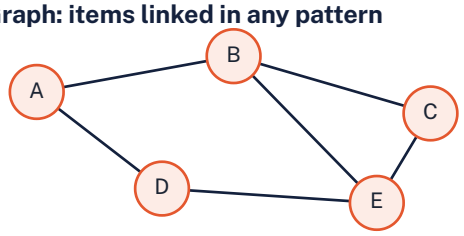
Stack: last in, first out



Tree: one root, each item has children



Graph: items linked in any pattern



Each structure arranges the same kind of data differently, which makes different questions fast.

What each structure is good at

Structure	Fast at	Slow at	Typical use
List / array	Getting the item at a position; adding to the end	Finding an item by value (must scan); inserting near the front	An ordered basket of items; a page of search results
Dictionary / map	Finding, adding or removing by key	Anything that needs the items in sorted order	Customers by ID; settings by name; a cache
Set	Checking membership; removing duplicates	Keeping order (most sets have none)	Which users have already voted; unique tags
Queue	Adding at the back and taking from the front	Reaching items in the middle	Orders waiting for the kitchen; jobs waiting for a worker
Stack	Adding and taking from the top	Reaching items lower down	Undo history; the Back button; tracking function calls
Tree (sorted)	Finding by value and listing a range in order	More complex to keep balanced	Database indexes; folders; menus
Graph	Following relationships	Very large graphs need specialised storage	Routes and maps; social networks; recommendations

```
basket = ["pizza", "cola", "pizza"]           # list: ordered, duplicates allowed
print(basket[0])                               # by position: "pizza"

customers = {"C102": "Ana", "C377": "Chen"}     # dictionary: key to value
print(customers["C377"])                       # by key: "Chen"

voted = {"ana@example.com", "ben@example.com"}   # set: no duplicates
print("ana@example.com" in voted)              # True

from collections import deque
kitchen = deque(["order 1", "order 2"])         # queue
kitchen.append("order 3")                      # joins at the back
print(kitchen.popleft())                       # served first: "order 1"
```

The same structures in Python

Algorithms

An algorithm is a precise, step-by-step method for solving a problem that always finishes: like a recipe, but with no room for interpretation. Sorting a list of names, finding the shortest route between two addresses, checking whether a password matches and deciding which advert to show are all done by algorithms.

There is usually more than one algorithm for the same problem, and they can differ enormously in how much work they do. Consider finding a name in a printed telephone directory. You could start at page one and read every entry until you find it: correct, but slow. Or you could open the book in the middle, see whether the name comes before or after that page, discard the half that cannot contain it and repeat. Because the directory is sorted, each look halves the remaining pages. This second method is called **binary search**, and for a directory of a million names it needs about 20 looks instead of up to a million.

Binary search: halving the problem

Highlighted: the middle item compared at each step. Grey: ruled out. Items left: 16, 7, 3, 1

In a sorted list each comparison rules out half of what remains, so 16 items need at most 4 or 5 looks.

Big-O: how work grows with data

Engineers describe how an algorithm's cost grows using Big-O notation. It ignores the exact time on a particular machine and asks one question: if the amount of data, called *n*, gets ten or a thousand times larger, how much more work is there? It describes the worst case unless someone says otherwise, and it ignores constant factors, which is both its strength (it captures the shape of growth) and its weakness (two O(n) algorithms can still differ by a factor of ten in practice).

O(1), constant time: the work does not depend on n. Looking up a key in a dictionary is O(1) on average. **O(log n)**, logarithmic: the work grows by one step each time n doubles, like binary search. **O(n)**, linear: double the data, double the work, like scanning a list. **O(n log n)**: slightly

worse than linear; good general-purpose sorting algorithms work at this rate. **$O(n^2)$**, quadratic: double the data, four times the work. This typically appears when, for every item, a program loops over every other item, such as comparing each customer with every other customer to find duplicates.

How many basic steps, roughly, as data grows

n (items)	O(1)	O(log n)	O(n)	O(n log n)	$O(n^2)$
10	1	about 3	10	about 33	100
1,000	1	about 10	1,000	about 10,000	1,000,000
1,000,000	1	about 20	1,000,000	about 20,000,000	1,000,000,000,000

Read the last column carefully. At a billion simple steps per second, a million-squared algorithm needs a trillion steps: around a quarter of an hour of pure computation, and in practice far longer. The same job done with an O(n log n) method takes a small fraction of a second. Code that looked fine in testing with 1,000 customers can grind to a halt with a million, and no amount of extra hardware will rescue a quadratic algorithm for long.

WORKED EXAMPLE

Worked example: finding a customer among a million

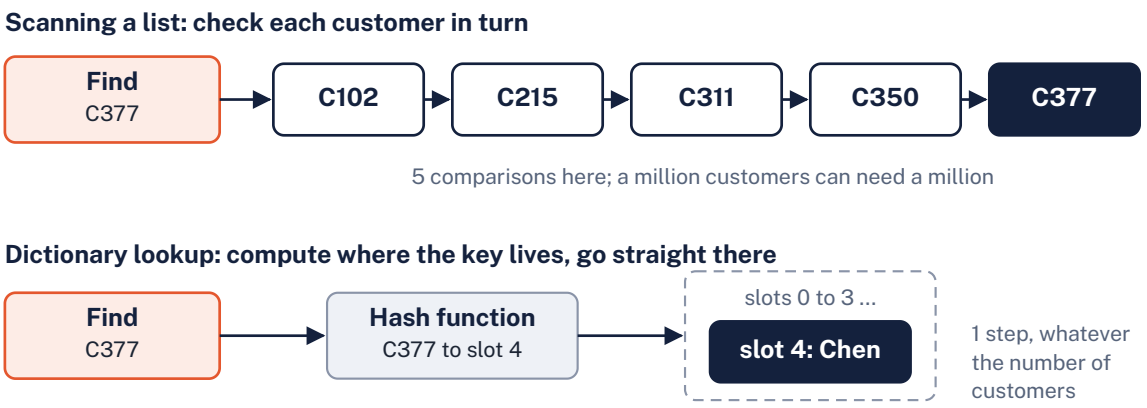
A food-delivery app stores 1,000,000 customers. Each time an order arrives it must find the customer by ID. Compare two designs, assuming each check takes 0.1 microseconds.

Design A: a list	`for c in customers: if c.id == wanted: ...` scans until it finds a match.
Checks per lookup (A)	Up to 1,000,000; on average about 500,000.
Time per lookup (A)	About 500,000 x 0.1 microseconds = 50 milliseconds.
At 200 orders per second (A)	200 x 50 ms = 10 seconds of CPU work every second: the server cannot keep up, so requests queue and time out.
Design B: a dictionary keyed by ID	`customers_by_id[wanted]` computes where the entry is and goes straight there.
Checks per lookup (B)	About 1, whether there are a thousand customers or a million.
Time per lookup (B)	Well under a microsecond.
Cost of Design B	Extra memory for the dictionary, and the work of building it and keeping it up to date whenever a customer is added or changed.

The two designs give identical answers; only the data structure differs, and that changes the outcome from failure to comfortable headroom.

Databases offer the same choice. Without an index, finding a row by a column means scanning the table; with an index it means a few steps through a sorted tree. The section on storing data returns to this.

Scan versus lookup



A scan does work in proportion to the number of items; a dictionary goes directly to the right slot.

Space versus time

The dictionary in the worked example is faster because it uses extra memory to organise the data in advance. This is one of the most common trade-offs in computing: you can often buy speed with space, or save space by spending time.

An index is extra data, kept beside the main data, that answers a particular kind of question quickly, like the index at the back of a book. It speeds up reads, but it takes storage and must be updated on every write, so a table with ten indexes is slow to insert into. A cache keeps a copy of the answer to an expensive question somewhere fast, such as in memory, so the next person to ask gets it immediately. Caches are why a popular product page loads quickly even when the database behind it is busy, but they raise a hard question: when the underlying data changes, how quickly must the copy change too? Showing a stale price for a few seconds may be acceptable; showing a stale bank balance may not.

Compression is the opposite trade: spend CPU time to squeeze data into fewer bytes, which saves storage and network transfer time.

WATCH OUT

Common misconceptions

- **"Big-O tells you how fast code runs."** It tells you how cost grows. For ten items an $O(n^2)$ method may be faster than a clever O(n log n) one; Big-O matters as n becomes large.
- **"Faster hardware fixes slow algorithms."** A machine twice as fast lets a quadratic algorithm handle only about 1.4 times as much data in the same time.
- **"Dictionary lookups are always instant."** They are O(1) on average, but use more memory than a list and give no ordering, so "all customers whose names start with C" still needs a scan or a sorted structure.
- **"Floats are fine for money."** Rounding errors add up across millions of transactions. Use integers of the smallest unit, or a decimal type.

GOING DEEPER

Going deeper: averages, worst cases and hashing

A dictionary achieves O(1) on average by running each key through a *hash function*, which turns the key into a number that picks a slot. Two keys can land in the same slot (a *collision*); implementations handle this, and in a pathological case where many keys collide, lookups degrade towards O(n). This is not only theoretical: attackers have sent web servers request parameters crafted to collide, which is why modern languages, Python included, randomise their string hashing. Database indexes typically use a balanced tree called a B-tree, which gives O(log n) lookups and, unlike a hash table, also supports ranges and sorted order.

Why this matters for architecture

Architects rarely write sorting algorithms, but they constantly make decisions whose consequences are exactly these. Which questions will the system need to answer quickly, and does the data have the indexes to answer them? How large will the data grow in three years, and which parts of the design are linear or worse in that growth? Where is caching worth its complexity, and how stale may cached data be? When data volume grows tenfold, which component breaks first?

These are questions about performance and scalability, which Chapter 2 makes precise, and about caching and database choice, which Chapter 7 treats in depth. The next section turns from data to code, and to how code is organised so that people can understand and change it.

CHECK YOUR UNDERSTANDING

▼ Why does `0.1 + 0.2` not give exactly 0.3 in most languages?

Floating-point numbers are stored in binary, and many decimal fractions, including 0.1 and 0.2, have no exact binary form, so tiny rounding errors appear.

▼ Which data structure suits an "undo" button in a text editor, and why?

A stack: each action is pushed on top, and undo pops the most recent one first (last in, first out).

▼ If an O(n) process takes one second for 100,000 items, roughly how long for 1,000,000?

About ten seconds, because linear work grows in proportion to the data.

▼ What do you pay for the speed a database index gives you?

Extra storage, and extra work on every insert, update and delete to keep the index up to date.

SOURCES

- Python tutorial: data structures: https://docs.python.org/3/tutorial/datastructures.html
- Python wiki: time complexity of built-in operations: https://wiki.python.org/moin/TimeComplexity
- Unicode HOWTO (Python documentation): https://docs.python.org/3/howto/unicode.html
- RFC 3629: UTF-8, a transformation format of ISO 10646: https://www.rfc-editor.org/rfc/rfc3629

SECTION 0.3 OF 0.8 · BEGINNER · 16 MIN READ

Functions, modules, objects, interfaces, libraries and frameworks

THE IDEA IN BRIEF

Think of a large restaurant. Instead of one cook doing everything, there are stations (grill, pastry, salads), each with its own staff, tools and recipes. Orders pass between stations through a hatch with a ticket system, so the grill does not need to know how the pastry chef works. Code is organised the same way: small named jobs (functions) are grouped into stations (modules), and stations deal with each other only through agreed tickets (interfaces). That way one station can change its methods without upsetting the rest of the kitchen.

WHAT YOU NEED TO KNOW

- A function is a named, reusable piece of code with inputs and an output; it lets a rule be written once and changed in one place.
- Modules and packages group related code and separate a small public part from private details that can change freely.
- Classes bundle data with the methods that act on it; encapsulation keeps the data private so the rules are always enforced.
- An interface is a contract between parts; code that depends on an interface rather than an implementation can swap implementations easily.
- You call a library; a framework calls your code. This inversion of control makes frameworks powerful but harder to replace.
- Dependencies, fetched by package managers such as pip and npm, save effort but bring transitive code, security risk and maintenance work.
- Good structure means loose coupling and high cohesion, which keeps change cheap and lets teams work independently.

Why organise code at all

A small script of twenty lines can be read top to bottom in a minute. A real system, such as a food-delivery platform, may have hundreds of thousands or millions of lines written by dozens of people over many years. Nobody can hold that in their head. The only way to work on such a system is to divide it into parts that can be understood, tested and changed one at a time, with clear rules about how the parts may use each other.

Programming languages give us a ladder of building blocks for this, from small to large: functions, modules and packages, classes and objects, and the interfaces between them. Above the code we write ourselves sit libraries and frameworks written by others. This section climbs that ladder. The same ideas, applied at larger scale, are what software architecture is about.

Functions: name a job, then reuse it

A function is a named piece of code that does one job. It takes *inputs* (called parameters or arguments), does some work and usually gives back an *output* (its return value). Think of a coffee machine: you put in beans, water and a choice of drink; it does its work; out comes a coffee. You do not need to know how the pump works to use it.

Functions bring three benefits. **Reuse**: the VAT calculation is written once and called from the checkout, the invoice and the refund screen. **Readability**: `total_with_vat(basket)` says what is happening without showing how. **Change in one place**: when the VAT rate changes, one function changes, not twenty copies of the same arithmetic scattered through the code (copies that would soon drift apart).

A good function does one thing, has a name that says what it does, takes the inputs it needs as parameters rather than reaching out for hidden data, and is short enough to read at a glance. A function that only computes its output from its inputs, without changing anything else, is called a *pure* function; pure functions are especially easy to test, because the same inputs always give the same output.

```
VAT_RATE = 0.20

def total_with_vat(prices_in_pence):
    """Return the total of a basket in pence, including VAT."""
    subtotal = sum(prices_in_pence)
    return round(subtotal * (1 + VAT_RATE))

checkout_total = total_with_vat([1250, 399])   # 1979
invoice_total = total_with_vat([5000])         # 6000
```

One function, reused in several places

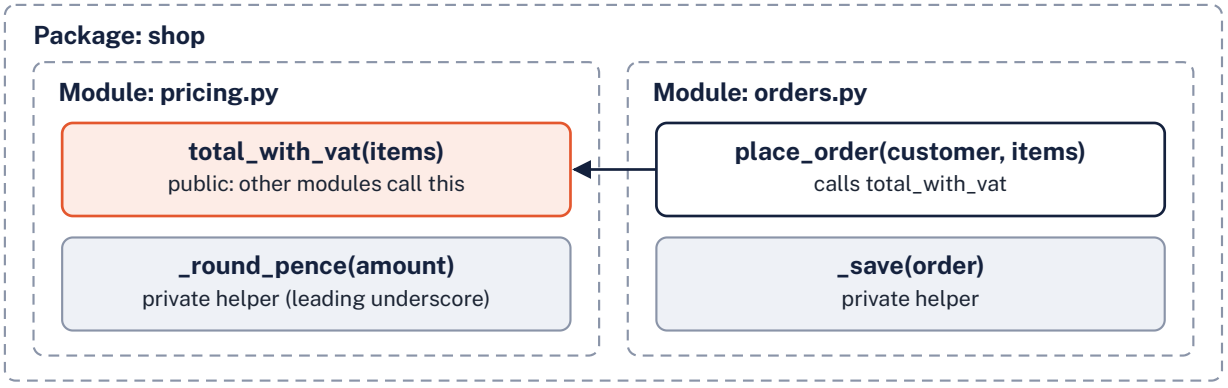
Modules and packages: grouping related code

A module groups related functions and data into a unit with its own name. In Python a module is simply one `.py` file; `pricing.py` might hold everything to do with prices, discounts and tax. Another file uses it by writing `import pricing`. A **package** is a folder of related modules,

so a shop might have a `shop` package containing `pricing`, `orders` and `customers` modules. Other languages use different words (Java has packages, C# has namespaces and assemblies, JavaScript has modules), but the idea is the same.

A module is like a department in an organisation. The finance department has a public front desk where others can make requests ("please raise this invoice"), and internal processes nobody outside needs to see. Modules similarly separate a small **public** part, meant for other modules to use, from **private** details. Python marks private names by convention with a leading underscore, such as `_round_pence`; Java and C# enforce privacy with keywords such as `private`. The value of the split is that the private part can be rewritten freely without breaking anyone else, as long as the public part keeps its promises.

Functions, modules and a package



Functions live in modules; related modules are grouped into a package; only public names are meant to be used from outside

The orders module uses only the public function of the pricing module, never its private helper.

Classes and objects: data with its behaviour

Many languages let you bundle data together with the functions that work on it. A **class** is the template; an **object** is one thing made from it. A `BankAccount` class might say that every account has an owner and a balance, and can accept deposits and withdrawals. Each customer's account is then a separate object made from that class. Functions that belong to a class are called *methods*. (See the glossary entry for class and object.)

The most useful idea here for architecture is encapsulation: an object keeps its data private and lets other code change it only through its methods. If anyone could set `account.balance` directly, nothing would stop code elsewhere from making a balance negative or skipping the audit log. With encapsulation, the `withdraw` method is the only door, and it can enforce the rules every time. A bank teller is the analogy: customers cannot walk into the vault, but they can ask the teller, who checks the rules before handing over money.

There is much more to object-oriented programming (inheritance, polymorphism, design patterns), and practitioners disagree about how much of it is helpful: some languages, such as Go, deliberately leave out inheritance, and many teams favour a more functional style. Encapsulation, however, is valuable in every style.

```
class BankAccount:
    def __init__(self, owner):
        self.owner = owner
        self._balance = 0          # private by convention

    def deposit(self, pence):
        if pence ⩽ 0:
            raise ValueError("Deposit must be positive")
        self._balance += pence

    def withdraw(self, pence):
        if pence > self._balance:
            raise ValueError("Insufficient funds")
        self._balance -= pence

    def balance(self):
        return self._balance

acct = BankAccount("Ana")
acct.deposit(10000)
acct.withdraw(2500)
print(acct.balance())   # 7500
```

Encapsulation: the balance can only change through methods that enforce the rules

Interfaces: contracts between parts

An interface is the agreed set of operations one part of a system offers to others: their names, what inputs they take, what they return and what can go wrong. It is a contract. The caller relies on the contract, not on how the work is done. A power socket is an interface: any appliance with the right plug works, and the appliance does not care whether the electricity comes from wind, gas or nuclear power.

In code, an interface lets you swap one implementation for another. Suppose a shop takes card payments through one payment provider today but may switch providers next year. If the checkout code calls the provider's own functions directly, everywhere, switching means rewriting the checkout. If instead the shop defines its own small `PaymentGateway` interface with one operation, `charge(amount, card_token)`, then writes one adapter per provider, the

checkout depends only on the interface and switching providers means writing one new adapter. The same trick makes testing easy: in tests, a fake gateway that never contacts a real bank can stand in for the real one.

Interfaces are the most important idea in this section for architects. Every boundary in a system, from a function signature to a web API used by thousands of other companies, is an interface. Good ones are small, stable and hide decisions that are likely to change; that principle, called information hiding, was set out by David Parnas in 1972 and is covered in Chapter 3.

```
from typing import Protocol

class PaymentGateway(Protocol):
    def charge(self, amount_pence: int, card_token: str) → bool: ...

class ProviderAGateway:
    def charge(self, amount_pence, card_token):
        ...  # call Provider A's web API
        return True

class FakeGateway:                      # used in tests
    def charge(self, amount_pence, card_token):
        return amount_pence < 100_000   # pretend large payments fail

def checkout(basket_total, card_token, gateway: PaymentGateway):
    return "paid" if gateway.charge(basket_total, card_token) else "declined"
```

An interface (here a Python Protocol) with two interchangeable implementations

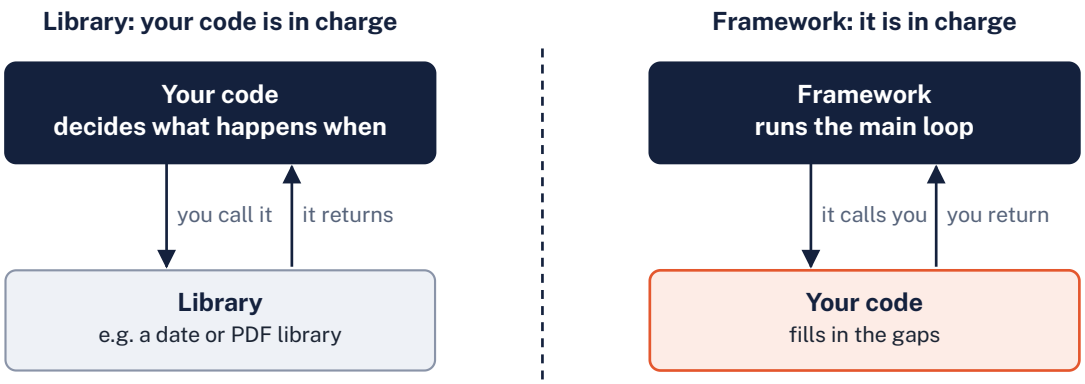
Libraries and frameworks: code written by others

Almost no one writes everything from scratch. A library is reusable code that someone else has written and that your code calls when it needs it: a library for handling dates and time zones, for generating PDFs, for talking to a database. You stay in charge; the library is a tool you pick up and put down.

A framework is different. It provides the overall skeleton of an application and decides when things happen; you fill in the gaps with your own code, and the framework calls that code at the right moments. A web framework such as Django (Python), Spring (Java), Ruby on Rails or Express (JavaScript) receives each incoming web request, works out which of your functions should handle it, calls that function and turns its result into a web page or data. The short way to remember the difference: *you call a library; a framework calls your code*.

That reversal is called inversion of control. Martin Fowler credits Ralph Johnson and Brian Foote with using the term in 1988, and the idea is often summed up as the Hollywood principle: “Don't call us, we'll call you”. A restaurant franchise is a fair analogy: the franchise supplies the building, the ordering system, the opening hours and the rules; the franchisee supplies the cooking in the slots the franchise provides.

Who calls whom



The direction of control is the real difference between a library and a framework.

Library or framework

	Library	Framework
Who controls the flow	Your code	The framework (inversion of control)
Examples	A date library, a PDF generator, a JSON parser	Django, Spring, Ruby on Rails, Angular
Ease of replacing later	Usually moderate: change the places that call it	Usually hard: the framework's structure shapes the whole application
Main benefit	Saves writing one capability yourself	Saves designing the whole structure; consistent conventions across a team
Main risk	One more dependency to maintain	Lock-in to its way of working and its upgrade cycle

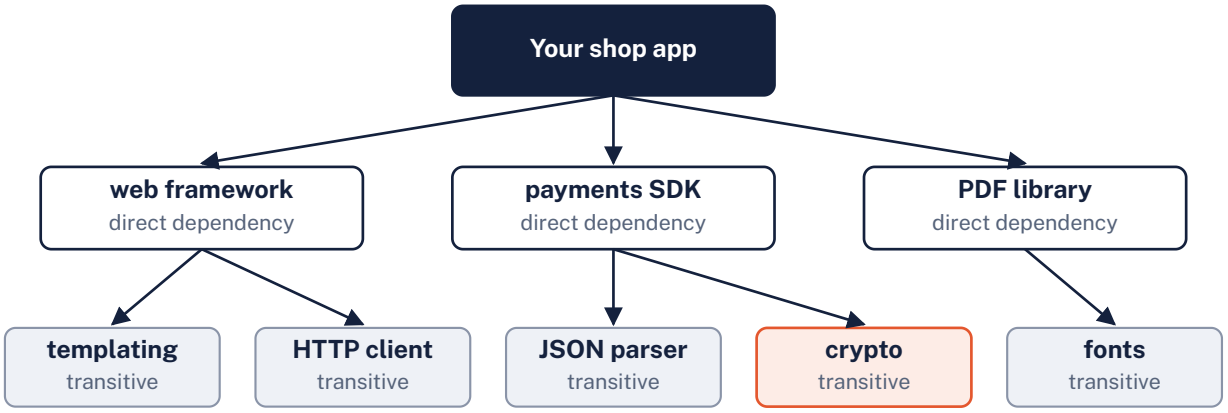
Dependencies and package managers

Every library or framework your code needs is a dependency. Modern projects have many, so they use a package manager to fetch them: `pip` for Python (from the Python Package Index, PyPI) and `npm` for JavaScript (from the npm registry) are the best known, with equivalents such as Maven for Java and NuGet for .NET. A project lists what it needs, often with version

ranges, and the package manager downloads those packages and, crucially, the packages *they* depend on. These indirect, *transitive* dependencies often outnumber the direct ones many times over.

Versions are commonly numbered MAJOR.MINOR.PATCH under a convention called semantic versioning: a change to the major number signals a breaking change, a minor number new features that should not break existing users, and a patch number bug fixes. Projects also keep a *lock file* (such as `package-lock.json` for npm) recording the exact version of every package installed, so that every developer's machine and the production build use the same code.

Direct and transitive dependencies



You chose three packages; you depend on eight. A flaw in any one (for example crypto) is a flaw in your app.

Most of the code your application runs was chosen by someone else's choices.

Dependencies save enormous effort, but each one is code you did not write, running with all the permissions of your own. Two well-known incidents show the risks. In March 2016 a developer removed his packages from npm in a dispute, among them `left-pad`, a tiny function of about eleven lines for padding text; thousands of projects that depended on it, directly or indirectly, suddenly failed to build until npm restored it. In December 2021 a severe security flaw, nicknamed Log4Shell, was disclosed in Log4j 2, a widely used Java logging library; organisations around the world spent weeks finding every system that contained it, often buried several levels deep in their dependencies.

Checklist: before adding a dependency

Questions a careful team asks before relying on someone else's package.

- [ ] **Do we need it?**
  A few lines of our own code may be cheaper over time than a package that must be tracked and updated.
- [ ] **Is it maintained?**
  Look at recent releases, open security issues and how many people maintain it. An abandoned package becomes your problem.
- [ ] **What does it bring with it?**
  Check how many transitive dependencies it adds; each is more code to trust.
- [ ] **Is the licence compatible?**
  Open-source licences differ; some impose obligations when you distribute your software.
- [ ] **Are versions pinned and scanned?**
  Use a lock file and an automated tool that warns when a known vulnerability affects a version you use.
- [ ] **Could we replace it?**
  Wrap important dependencies behind our own interface, so a change of supplier touches one place.

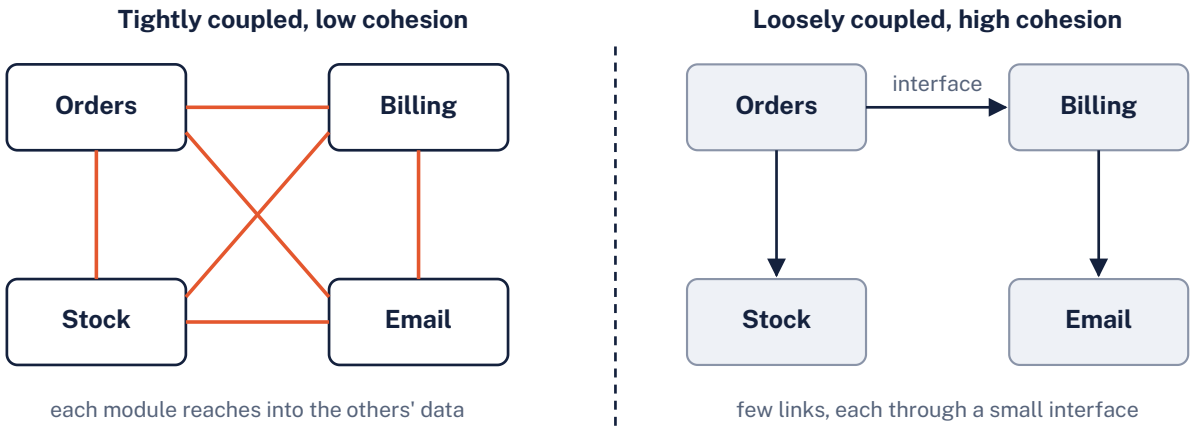
Why structure matters: change and teams

Code is read and changed far more often than it is first written. Most of the lifetime cost of software is spent changing it: fixing bugs, adding features, adapting to new laws and new partners. Structure decides how expensive that is. In a well-organised system, a change to how VAT is calculated touches one module. In a badly organised one, it touches forty files owned by five teams, and something unrelated breaks.

Two ideas, first described by Larry Constantine and colleagues in the structured design movement of the late 1960s and 1970s, capture what "well organised" means. Coupling is how much one part depends on the details of another: in a tightly coupled system, changing one part forces changes in others. Cohesion is how well the contents of one part belong together: a highly cohesive module does one clear job. The goal is **loose coupling and high cohesion**: parts that each do one job well and talk to each other through small, stable interfaces.

Structure also matters for people. If two teams must both edit the same module for most of their work, they will constantly wait for and break each other. If each team owns cohesive modules with clear interfaces, they can work and release independently. Melvin Conway observed in 1968 that systems tend to mirror the communication structure of the organisations that build them, an observation known as Conway's law and explored in Chapter 3.

Coupling at a glance



The same four modules, connected in two ways: every extra link is another path along which a change can spread.

WATCH OUT

Common mistakes

- **Copy and paste instead of a function.** Duplicated logic drifts apart, so the same rule gives different answers in different places.
- **Everything public.** If every detail of a module can be used from outside, every detail becomes something other code depends on, and nothing can change safely.
- **Framework code everywhere.** When business rules are tangled with framework calls, upgrading or replacing the framework means rewriting the business.
- **Unexamined dependencies.** Adding a package for a trivial task brings its whole dependency tree, its bugs and its security risks.

Why this matters for architecture

Architecture is largely the same activity as this section, done at a larger scale and with higher stakes: deciding what the main parts of a system are, what each is responsible for, which interfaces connect them and which way dependencies point. A module inside one program and

a separately deployed service across a network are both parts with interfaces; the questions of coupling, cohesion and information hiding apply to both.

Chapter 3 makes these principles precise, starting with modularity and coupling and cohesion, and Chapter 8 returns to dependencies as a security matter in supply-chain security. Next, the code leaves the single computer: the following section explains how computers talk to each other.

CHECK YOUR UNDERSTANDING

▼ **Why is it better to call one `total_with_vat` function from three places than to write the calculation three times?**

The rule then lives in one place: when the VAT rate changes, one change fixes every caller, and the copies cannot drift apart.

▼ **What does encapsulation stop other code from doing to a bank account object?**

It stops other code changing the balance directly, so every change has to go through methods such as withdraw that enforce the rules.

▼ **A web framework calls your function whenever a request for `/orders` arrives. What is this arrangement called?**

Inversion of control: the framework, not your code, controls the flow and calls your code when needed.

▼ **What is a lock file, such as `package-lock.json`, for?**

It records the exact version of every package installed, including indirect ones, so every developer machine and every production build use exactly the same code.

SOURCES

- Python tutorial: modules and packages: https://docs.python.org/3/tutorial/modules.html
- Python tutorial: classes: https://docs.python.org/3/tutorial/classes.html
- Martin Fowler: Inversion of Control: https://martinfowler.com/bliki/InversionOfControl.html
- Semantic Versioning 2.0.0: https://semver.org/
- npm documentation: package-lock.json: https://docs.npmjs.com/cli/v10/configuring-npm/package-lock-json

SECTION 0.4 OF 0.8 · BEGINNER · 18 MIN READ

Networks, the internet and the web: IP, DNS, HTTP and the client-server model

THE IDEA IN BRIEF

Sending data over the internet is like sending post. Your message is split into parcels (packets), each with a sender's and recipient's address (IP addresses) and a room number (a port), and sorting offices (routers) pass them along. Names like shop.example are looked up in a giant shared address book (DNS) to find the real address. The letters themselves follow a fixed format (HTTP): a request such as "please send me product 42", and a reply with a code saying how it went. HTTPS puts the letter in a sealed, tamper-proof envelope.

WHAT YOU NEED TO KNOW

- Data crosses networks in packets, which routers forward hop by hop using the destination's IP address; a port number identifies the program on that machine.
- TCP provides a reliable, ordered connection after a handshake; UDP sends individual packets with no guarantees but less delay.
- DNS turns names into IP addresses through a hierarchy of root, top-level domain and authoritative servers, with heavy caching controlled by TTLs.
- HTTP is a stateless request-response protocol: a method (GET, POST, PUT, DELETE), a path, headers and a body, answered by a status code (2xx, 3xx, 4xx, 5xx).
- HTTPS is HTTP over TLS, which encrypts traffic, detects tampering and uses certificates signed by trusted authorities to prove the server's identity.
- In the client-server model clients start requests and servers hold shared data and rules; servers must never trust clients blindly.
- Latency (delay) and bandwidth (capacity) are different; latency is limited by the speed of light, so distance and round trips matter.

Networks and packets

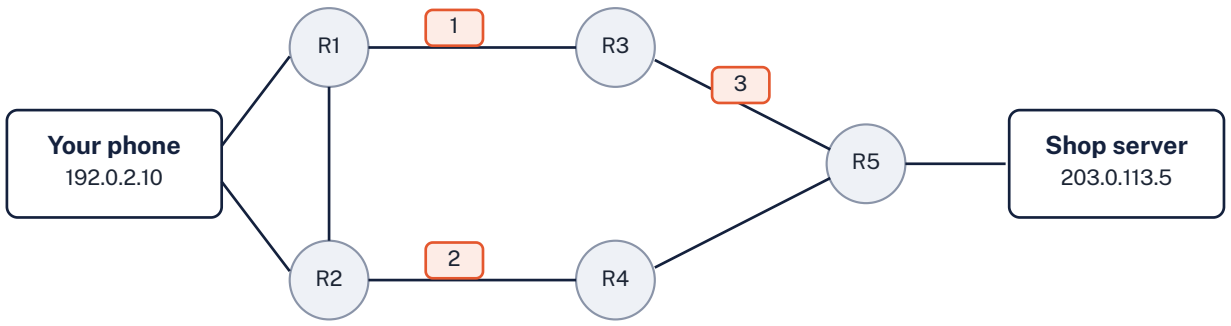
Almost no modern software runs alone. A food-delivery app on your phone talks to the company's servers, which talk to restaurants' tablets, a payment provider, a mapping service and a notification service. This section explains how that talking works, from the bottom up.

A **network** is a set of computers connected so they can exchange data, by cable, Wi-Fi, mobile signal or fibre-optic link. The **internet** is a network of networks: millions of separately owned networks (home broadband, mobile operators, universities, cloud providers) that have agreed to

carry each other's traffic using common rules called *protocols*.

Data does not travel as one continuous stream. It is cut into packets: chunks typically up to about 1,500 bytes, each labelled with where it came from and where it is going, like a parcel with a sender's and recipient's address. Devices called **routers** pass packets from network to network, each choosing a good next hop, much as sorting offices pass parcels on towards their destination. Different packets from the same message may take different routes, arrive out of order or occasionally be lost, and the receiving computer puts them back together. Splitting data into packets lets millions of conversations share the same cables fairly, and lets traffic flow around a broken link.

Packets find their own way



One message is split into numbered packets; each finds its own way across routers and is reassembled at the end

No single wire connects your phone to the server; packets hop between routers owned by many different networks.

Addresses: IP addresses and ports

For a packet to arrive, the destination needs an address. Every device on the internet is reached through an IP address (IP stands for Internet Protocol). The older version, IPv4, writes addresses as four numbers from 0 to 255, such as `192.0.2.10`; it allows about 4.3 billion addresses, which are not enough for every phone, laptop and sensor in the world. The newer IPv6 uses much longer addresses written in hexadecimal (base 16, using the digits 0 to 9 and the letters a to f), such as `2001:db8::1`, and has room for practically unlimited devices. Both are in use side by side. (The example addresses in this section come from ranges reserved for documentation, so they belong to no real machine.)

An IP address gets a packet to the right computer, but a computer runs many programs at once. A port number, from 0 to 65535, says which program should receive it. If the IP address is the street address of an office building, the port is the room number. Well-known services

have standard ports: 443 for HTTPS, 80 for plain HTTP, 53 for DNS, 5432 for the PostgreSQL database. A server program *listens* on a port, waiting for connections; when your browser contacts a website it connects to port 443 at the site's IP address.

TCP and UDP: two ways to deliver

On top of IP sit two main transport protocols, which decide how much care is taken over delivery. TCP (Transmission Control Protocol) gives two programs a reliable, ordered stream of bytes. Before any data is sent, the two sides set up a connection with a three-step *handshake*. Each packet is numbered; the receiver acknowledges what arrived; anything missing is sent again; everything is handed to the program in the right order. TCP also slows down when the network is congested. It is like a recorded-delivery service with tracking: dependable, but with some overhead.

UDP (User Datagram Protocol) simply sends individual packets, called datagrams, with no handshake, no acknowledgements, no resending and no ordering. It is like posting postcards: cheap and quick, but some may not arrive. That is exactly right when a late packet is useless anyway. In a video call, a fragment of sound that arrives half a second late is worse than one that never arrives, so calls, live streams and online games use UDP and cope with small gaps. DNS lookups, which are tiny questions and answers, also traditionally use UDP.

TCP and UDP compared

	TCP	UDP
Connection set-up	Yes: a handshake before data flows	None: just send
Delivery	Guaranteed: lost packets are resent	Best effort: losses are not repaired
Order	Preserved	Not guaranteed
Overhead and delay	Higher	Lower
Typical uses	Web pages and APIs (HTTP/1.1 and HTTP/2), email, file transfer, database connections	Voice and video calls, live streaming, online games, DNS

GOING DEEPER

Going deeper: QUIC and HTTP/3

The newest version of the web's protocol, HTTP/3 (published as RFC 9114 in 2022), does not run over TCP. It runs over QUIC (RFC 9000, 2021), a transport protocol built on top of UDP that re-implements reliable delivery itself and combines connection set-up with encryption set-up, saving round trips. It shows that the line between "reliable TCP" and "unreliable UDP" is about what the transport layer guarantees, and that reliability can be built at a higher layer when that serves the application better.

DNS: the internet's address book

People remember names such as `shop.example`; networks route by IP addresses. The Domain Name System (DNS) translates one into the other. It is a worldwide, distributed directory: no single computer holds every name. Instead, responsibility is split by the dots in the name, read from right to left. The **root** servers know who is responsible for each top-level domain such as `.com` or `.uk`; those top-level domain servers know which **authoritative** name servers are responsible for each domain registered under them; and the authoritative servers hold the actual records, such as the IP address of `www.shop.example`. The root zone is served by 13 named server identities, run by 12 independent organisations through well over a thousand server instances around the world.

Your device does not usually walk this chain itself. It asks a **recursive resolver**, typically run by your internet provider or a public service, which does the legwork and remembers (caches) answers for a period set by the domain owner, called the TTL (time to live). Because answers are cached at many levels, most lookups are answered in a few milliseconds without reaching the root at all. The same caching explains why a change to a DNS record can take minutes or hours to be seen everywhere.

DNS resolution step by step (nothing cached)

1. **Ask the resolver**
   Your device asks its configured recursive resolver: what is the IP address of `www.shop.example`?

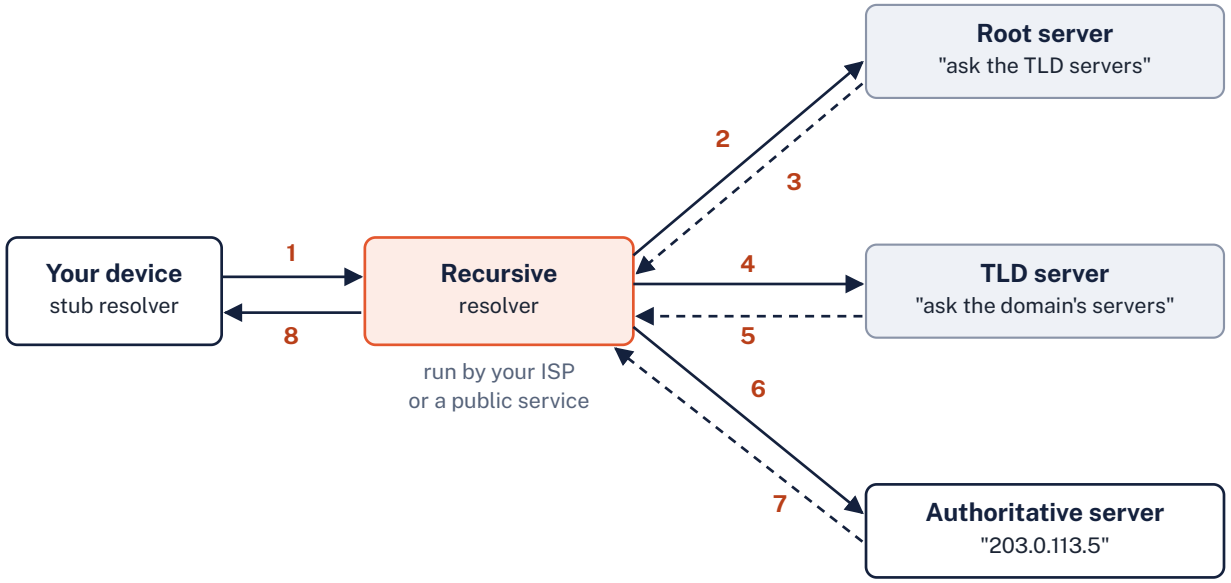
2. **Ask a root server**
   The resolver asks a root server, which does not know the answer but replies with the servers responsible for the top-level domain.

3 **Ask the top-level domain server**
The resolver asks one of those servers, which replies with the authoritative name servers for `shop.example`.

4 **Ask the authoritative server**
The resolver asks the authoritative server, which replies with the record: for example `203.0.113.5`, valid for 300 seconds.

5 **Answer and remember**
The resolver returns the address to your device, and both keep it in their caches until the TTL runs out, so the next lookup is almost instant.

Who answers a DNS question



Solid arrows are questions, dashed arrows are answers; caching means most real lookups skip steps 2 to 7.

HTTP: requests and responses

Once a program can reach another, the two need a shared language for what they want. On the web that language is HTTP (Hypertext Transfer Protocol). It follows a simple pattern: the client sends a **request**; the server sends back one **response**. Each request names a **method** (what to do), a **path** (to what), some **headers** (extra information as name-value pairs) and sometimes a **body** (data being sent). Each response carries a status code, headers and usually a body: a web page, an image or data in a format such as JSON (a simple text format for structured data, explained in the section on APIs).

The most common methods are `GET` (fetch something, without changing it), `POST` (submit something new, such as placing an order), `PUT` (replace something at a known address, such as updating a delivery address) and `DELETE` (remove something). GET, PUT and DELETE are defined to be *idempotent*: sending the same request twice has the same effect as sending it once, which makes them safe to retry after a network failure. POST is not, which is why a careless retry can place an order twice. Headers carry things such as the content type (`Content-Type: application/json`), the language the user prefers, cookies (small pieces of data a site asks the browser to store and send back with later requests) and credentials, and caching instructions.

HTTP is *stateless*: each request is understood on its own, and the server is not required to remember earlier ones. Logged-in sessions are built on top, usually by the client sending a cookie or token with every request. Statelessness is one reason the web scales so well, because any of many identical servers can handle any request.

```
POST /api/orders HTTP/1.1
Host: shop.example
Content-Type: application/json
Authorization: Bearer eyJhbGciOi...

{"product_id": 42, "quantity": 2}

HTTP/1.1 201 Created
Content-Type: application/json
Location: /api/orders/9917

{"order_id": 9917, "status": "received", "total_pence": 3000}
```

A real-looking HTTP exchange: placing an order, then the server's reply

Status codes by family

Family	Meaning	Common examples
2xx	Success	200 OK; 201 Created (something new was made); 204 No Content
3xx	Redirection: look elsewhere	301 Moved Permanently; 302 Found (temporary move); 304 Not Modified (your cached copy is still good)
4xx	Client error: the request was wrong	400 Bad Request; 401 Unauthorized (not logged in); 403 Forbidden (logged in but not allowed); 404 Not Found; 429 Too Many Requests
5xx	Server error: the request was fine, the server failed	500 Internal Server Error; 502 Bad Gateway; 503 Service Unavailable; 504 Gateway Timeout

The split between 4xx and 5xx matters in practice. A 4xx says "do not simply retry; fix the request". A 5xx says "the problem is on our side"; retrying later may work, and the operations team should be alerted when 5xx rates rise.

HTTPS and TLS: privacy and identity

Plain HTTP sends everything as readable text, so anyone on the path, such as someone on the same café Wi-Fi, could read or alter it. HTTPS is HTTP sent through a protected channel created by **TLS** (Transport Layer Security; the current version, TLS 1.3, was published as RFC 8446 in 2018). TLS provides three things: *confidentiality* (eavesdroppers see only scrambled data), *integrity* (tampering is detected) and *authentication* (you are really talking to `shop.example`, not an impostor).

Authentication rests on **certificates**. A site's certificate is a digital document saying, in effect, "this public key belongs to `shop.example`", signed by a **certificate authority** (CA) that your browser or phone already trusts. It works like a passport: issued by an authority both sides trust, checked against the holder, and valid only until its expiry date. During the TLS handshake the server presents its certificate, the client checks the signature, the name and the dates, and the two sides agree on fresh secret keys for encrypting the rest of the conversation. If any check fails, the browser shows a warning.

TLS protects data *in transit* between two points. It does not protect data once it is stored, and it does not prove that the site is honest, only that it is who it claims to be. Today HTTPS is the norm for all web traffic, not just payments, and certificates are often issued free and renewed automatically.

The client-server model

Most of the systems in this course follow the client-server model. A **client** (a browser, a mobile app, another program) starts the conversation by sending a request. A **server** waits for requests and answers them. The server holds what must be shared and trusted: the product catalogue, everyone's orders, the rules about prices and refunds. The client handles the user's screen and input.

In a restaurant, diners are clients and the kitchen is the server: diners initiate orders; the kitchen holds the ingredients and the expertise and serves many diners at once. One consequence is that the server must never trust the client blindly. A phone app can be modified by its user, so prices, permissions and stock levels must be checked on the server, however carefully the app also checks them.

A server is often also a client: the shop's web server is a client of its database and of the payment provider. Real systems are chains and webs of such relationships, as the sequence diagram below shows.

Latency and bandwidth

Two numbers describe how fast a network feels, and they are often confused. Bandwidth is how much data a connection can carry per second, such as 100 megabits per second: the width of the pipe. Latency is how long a piece of data takes to get from one end to the other, usually measured as a **round-trip time** (RTT) in milliseconds: the length of the pipe. A motorway analogy helps: adding lanes (bandwidth) lets more cars through per hour but does not make any one car's journey shorter (latency).

Latency has a floor set by physics. Light in optical fibre travels at roughly 200,000 km per second, about two-thirds of its speed in a vacuum. London and Sydney are about 17,000 km apart in a straight line around the globe, so even a perfect fibre along that line would need about 85 ms one way and 170 ms for a round trip. Real cables do not follow the shortest path and routers add delay, so measured round trips are typically around 250 ms. No upgrade can remove that. If loading a page from Sydney needs ten sequential round trips to a server in London, the user waits two and a half seconds, however fast their broadband is.

WORKED EXAMPLE

Worked example: a London server, a Sydney customer

A shop hosts everything in London. A customer in Sydney opens a product page over a new HTTPS connection. Assume a round trip of 250 ms and ignore server processing time.

DNS lookup (not cached)	Resolver may need several queries; say 1 round trip to nearby servers: about 50 ms
TCP handshake	1 round trip to London: 250 ms
TLS 1.3 handshake	1 round trip: 250 ms
Request the HTML page	1 round trip: 250 ms
Request CSS and scripts the page needs	At least 1 more round trip: 250 ms
Total before anything useful appears	About 1,050 ms, with bandwidth playing almost no part
Same customer, content served from Sydney	Round trip of perhaps 10 ms: the same five steps take roughly 50 ms

For small pages, latency, not bandwidth, dominates. Upgrading the customer's broadband from 50 to 500 megabits per second would barely change the result.

Architects reduce latency by cutting round trips (reusing connections, combining requests) and by moving content closer to users, for example with content delivery networks, covered in Servers and the cloud.

DECISION GUIDE

Decision guide: is a slow network experience about latency or bandwidth?

A rough first diagnosis when users complain that something is slow over the network.

Q1 **Are small requests (a short API call, a tiny page) slow too?**

- **Yes** Latency is the likely cause. Check the distance to the server and how many sequential round trips the page or app makes.
- **No** Go to the next question.

Q2 **Are only large transfers (videos, big downloads, large reports) slow?**

- **Yes** Bandwidth is the likely limit, on the user's connection or the server's. Consider compression, smaller files or a content delivery network.
- **No** Go to the next question.

Q3 **Is the delay the same from every location?**

- **Yes** The network may not be the problem at all: look at server processing, database queries and algorithms (see the data and algorithms section).
- **No** Location-dependent delay points to latency and routing: serve users from closer to where they are.

What happens when you type a web address

All of these pieces come together every time you visit a website. Here is the sequence when you type `https://shop.example/products/42` and press Enter, simplified but accurate.

From Enter to page

1. **Parse the address**
   The browser splits the URL into the scheme (`https`, so port 443 and TLS), the host name (`shop.example`) and the path (`/products/42`).

2. **Find the IP address**
   The browser checks its own cache, then the operating system's, then asks the recursive resolver, which consults DNS as described above and returns, say, `203.0.113.5`.

3. **Open a TCP connection**
   The browser and server exchange the three handshake messages to set up a reliable connection to port 443.

4 **Set up TLS**
The server presents its certificate; the browser checks it is valid for `shop.example` and signed by a trusted authority; both sides agree on encryption keys.

5 **Send the HTTP request**
The browser sends `GET /products/42` with headers such as the accepted languages and any cookies for the site.

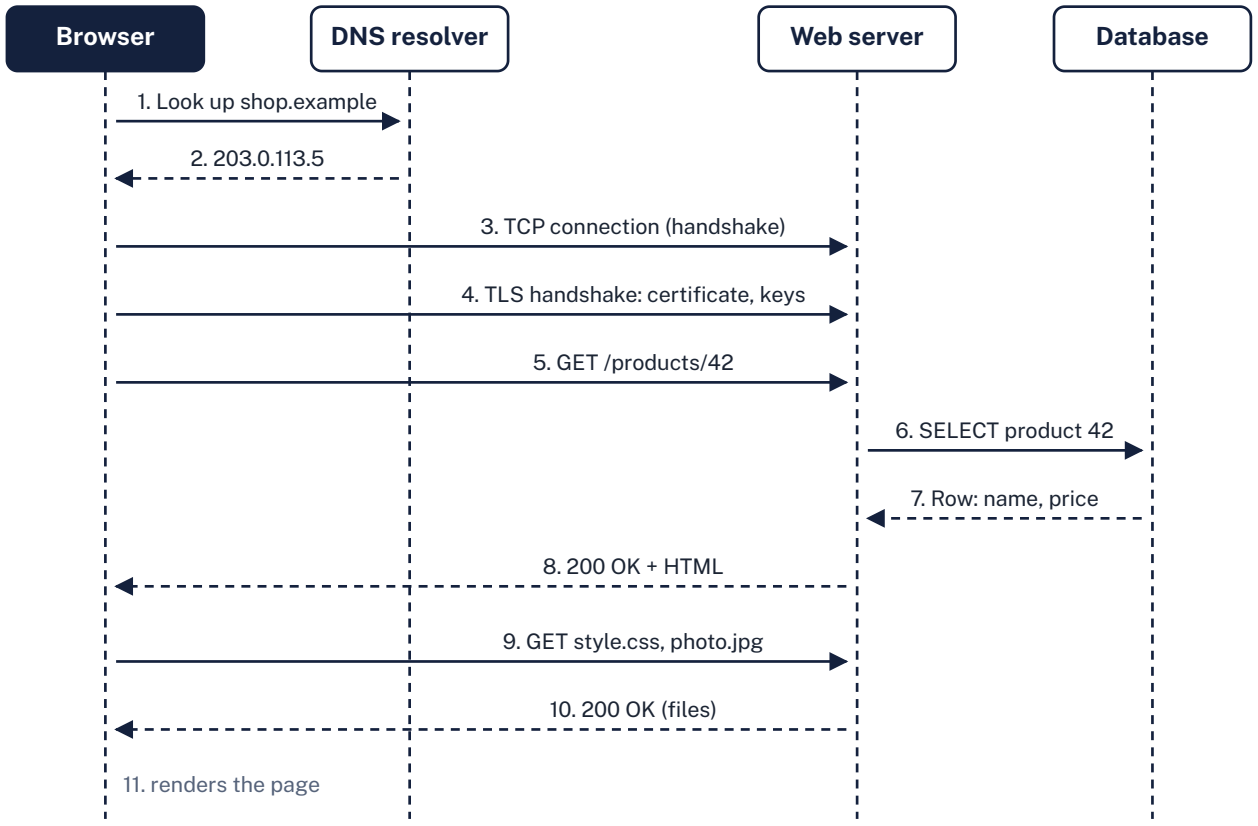
6 **The server does its work**
Server software receives the request, runs the shop's code, which queries a database for product 42, and builds an HTML page.

7 **Receive the response**
The server replies `200 OK` with the HTML in the body and headers such as `Content-Type: text/html` and caching rules.

8 **Fetch the rest**
The HTML refers to stylesheets, scripts and images; the browser requests those too, often many at once over the same connection.

9 **Render**
The browser lays out the page and draws it on screen; scripts may then make further requests, for example to fetch reviews.

Sequence diagram: loading a product page



Time runs downwards; solid arrows are requests, dashed arrows are responses.

WATCH OUT

Common misconceptions

- **"The internet is fast, so distance does not matter."** Bandwidth has grown enormously; the speed of light has not. Every round trip to a distant server costs real time.
- **"HTTPS means the site is safe."** It means the connection is private and the site is who it claims to be. A scam site can have a valid certificate.
- **"A 404 means the server is down."** A 404 is the server working correctly and saying the thing asked for does not exist. A server that is down usually gives no answer, or a 5xx from something in front of it.
- **"Changing DNS is instant."** Cached answers live until their TTL expires, so changes spread gradually.

Why this matters for architecture

The moment a system is split across more than one computer, every interaction between its parts becomes a network call, with latency, the chance of failure and the need for security. A function call inside a program takes nanoseconds and never gets lost; a call across a network takes milliseconds, sometimes far more, and sometimes never returns. Many of the hardest problems in architecture follow from that difference, which is why Chapter 6 opens with the fallacies of distributed computing, such as the belief that latency is zero.

The protocols here also reappear as building blocks: HTTP underpins the web APIs in the next part of this chapter and the communication styles of Chapter 6, while TLS and certificates return in secure communication and zero trust. Placing servers close to users is part of multi-region and edge architecture.

CHECK YOUR UNDERSTANDING

▼ Why does a packet need a port number as well as an IP address?

The IP address gets it to the right computer, but that computer runs many programs; the port says which program, such as the web server listening on 443, should receive it.

▼ What does a 3xx status code tell the client?

That what it asked for is elsewhere or unchanged: for example 301 means it has moved permanently, and 304 means the client's cached copy is still valid.

▼ What is the difference between a 401 and a 503 response?

401 is a client error: the request lacks valid credentials. 503 is a server error: the service is temporarily unable to handle the request.

▼ Why can no network upgrade bring a London to Sydney round trip down to 10 ms?

Light in fibre covers about 200,000 km per second, so the roughly 34,000 km round trip takes at least about 170 ms.

SOURCES

- MDN: An overview of HTTP: https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides/Overview
- MDN: HTTP response status codes: https://developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Status
- RFC 9110: HTTP Semantics: https://www.rfc-editor.org/rfc/rfc9110

- RFC 8446: The Transport Layer Security (TLS) Protocol Version 1.3: https://www.rfc-editor.org/rfc/rfc8446
- Root Server Technical Operations Association: https://root-servers.org/

SECTION 0.5 OF 0.8 · BEGINNER · 18 MIN READ

Files and databases: relational and non-relational storage, queries and transactions

THE IDEA IN BRIEF

A database is like a library with a careful librarian. Instead of every program stuffing papers into its own drawer, all the data goes to one place that keeps a catalogue (an index), checks that every record follows the rules, makes sure two people cannot change the same thing at once, and never loses a change it has promised to keep. Most databases store data in tables, like spreadsheets with strict rules, and you ask them questions in a language called SQL.

WHAT YOU NEED TO KNOW

- Files suit whole documents such as images and logs; databases suit structured data that many users read and change at once, because the DBMS handles concurrency, rules, fast lookup and crash recovery.
- Relational databases store data in tables of rows and columns; a primary key identifies each row and a foreign key links a row to a row in another table.
- SQL is declarative: you state the result you want, such as a filtered SELECT or a JOIN across tables, and the database's query planner decides how to get it.
- An index lets the database find rows in O(log n) steps instead of scanning the whole table, at the cost of extra storage and slower writes.
- A transaction makes several changes succeed or fail together; ACID (atomicity, consistency, isolation, durability) describes its guarantees, as in a bank transfer that can never half-happen.
- NoSQL families (document, key-value, wide-column and graph) fit particular data shapes and access patterns; many systems use a relational database plus one or more of these.
- Backups are only proven by restoring them, replicas do not protect against mistakes, and managed cloud databases take over operations but not schema and query design.

Why programs need somewhere to keep data

A running program keeps its working data in memory (RAM), which is fast but forgets everything when the program stops or the machine restarts (see what software is). Anything that must survive a restart, such as a customer's account, an order or a hospital appointment,

has to be written to storage that keeps its contents without power: a disk or solid-state drive. Data that survives in this way is called *persistent*, and deciding how to persist it is one of the first and longest-lived decisions in any system.

The simplest way to persist data is a file: a named sequence of bytes that the operating system stores on disk. Files are excellent for things that are naturally whole documents: images, PDFs, videos, configuration, logs, exported reports. Systems often keep them in cloud *object storage* (a service that stores files by name) and record only a reference elsewhere.

Files become painful when many users read and change small pieces of structured data at once. Imagine a shop keeping all its orders in one spreadsheet file. Two customers check out at the same moment: whose change wins? The server crashes halfway through saving: is the file half old, half new? A support agent wants one customer's orders: must the program read the whole file? Someone types a customer number that does not exist: who stops them? Every program using the file would have to solve each problem, identically.

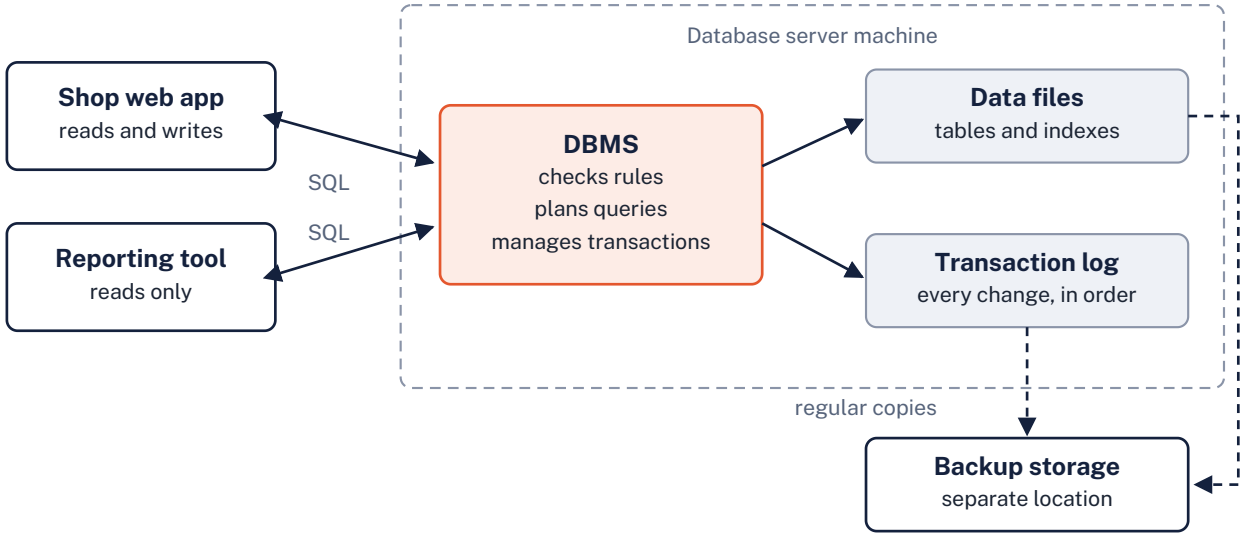
What a database is

A database is an organised collection of data, and the program that looks after it is a database management system (DBMS), such as PostgreSQL, MySQL, Microsoft SQL Server, Oracle Database or SQLite. People often say "the database" for both. The DBMS solves the file problems once, for everyone: it lets many users work at the same time without trampling each other, finds data quickly, refuses data that breaks the rules, and recovers to a correct state after a crash.

The analogy is a library. A heap of papers on a desk works if you are its only user. A library has a catalogue, rules about where things go, a record of who borrowed what, and a librarian who will not let two people take the same copy. Applications do not touch the shelves; they ask the librarian.

In most business systems the DBMS runs as its own server process, often on its own machine, and applications talk to it over the network (see how computers talk). SQLite is a well-known exception: it is a library that runs inside the application and stores the whole database in a single file, which is why it is found in phones, browsers and small tools.

Where the data actually lives



Applications never write the database's files directly: they send requests to the DBMS, which writes data files and a log, and a separate process copies both to backup storage.

Relational databases: tables, keys and relationships

The most widely used kind of database is the relational database, based on a model published by Edgar F. Codd of IBM in 1970. Data lives in **tables**. Each table describes one kind of thing (customers, orders, products). Each **row** is one instance of that thing (one customer), and each **column** is one fact about it (name, email address, date joined), with a declared type such as integer, text, date or decimal.

Each table has a primary key: a column (or combination of columns) whose value is unique for every row and never empty, such as `customer_id`. A foreign key is a column in one table that holds the primary key of a row in another table. An order has a `customer_id` column, which says which customer placed it. Declaring it as a foreign key tells the DBMS to refuse an order for customer 999 if there is no customer 999: this is called *referential integrity*.

The full set of tables, columns, types, keys and rules is the schema. The schema is a contract: every program that uses the database can rely on an order always having a valid customer and a price that is a number. Changing a schema later, when millions of rows and several programs depend on it, is possible but takes care, which is why schema evolution has its own section in Chapter 7.

Relational designs try to store each fact once. The customer's email address lives in one row of the customers table, not copied into every order. This is called *normalisation*. It means an update happens in one place and cannot leave contradictory copies behind. The price for this

tidiness is that answering a question often means combining tables, which is what joins are for.

Asking questions with SQL

SQL (Structured Query Language, pronounced "S-Q-L" or "sequel") is the language used to work with relational databases. It was developed at IBM in the 1970s and became an ANSI and ISO standard in the late 1980s; every major relational DBMS speaks it, each with small dialect differences. SQL is *declarative*: you describe the result you want, not the steps to find it. The DBMS's *query planner* chooses the steps, for example whether to read a whole table or use an index.

A join combines rows from two or more tables by matching columns, usually a foreign key against the primary key it refers to. The second query below lists each line of a customer's orders together with the product's name, drawing on three tables at once.

```
-- One customer's orders since 1 October, newest first
SELECT order_id, status, placed_at
FROM orders
WHERE customer_id = 42
  AND placed_at ⩾ '2026-10-01'
ORDER BY placed_at DESC;

-- Each order line with the product's name (a join)
SELECT o.order_id, p.name, i.quantity, i.unit_price
FROM orders AS o
JOIN order_items AS i ON i.order_id = o.order_id
JOIN products AS p ON p.product_id = i.product_id
WHERE o.customer_id = 42;
```

Two queries against the shop schema shown later in this section: a filtered SELECT and a three-table JOIN.

`INSERT`, `UPDATE` and `DELETE` change data, and `CREATE TABLE` defines the schema. Application code often sends SQL through an *object-relational mapper* (ORM), a library that turns rows into objects. ORMs save typing but hide the SQL actually sent; a classic performance problem is an ORM quietly issuing one query per row instead of one join.

Indexes: the book's index, not the whole book

Without help, the only way for the database to find every order for customer 42 is to read every row in the orders table and check each one: a full table scan, which is O(n) in the notation from data and algorithms. A database index is an extra, sorted data structure that maps values of a column to the rows that contain them, just as the index at the back of a book maps words to page numbers. Most relational databases use a *B-tree* by default, a wide, shallow, always-balanced tree that finds a value in O(log n) steps.

Primary keys are indexed automatically. Other indexes are added deliberately, for columns that queries filter, join or sort on: `CREATE INDEX idx_orders_customer ON orders (customer_id);`. Indexes are not free. Every insert, update or delete must also update each index on the table, and each index takes disk space and memory. A table with fifteen indexes reads quickly and writes slowly. Choosing indexes is a trade-off guided by the queries the system actually runs, which is an early example of the space-for-time trade-off that runs through all of architecture.

WORKED EXAMPLE

Worked example: finding one customer's orders

A shop has 10 million rows in its orders table. The customer service screen asks for all orders for one customer, about 20 rows.

Without an index	Full table scan: the database reads and checks all 10,000,000 rows to find the 20 that match.
With a B-tree index on customer_id	Each index page holds hundreds of keys, so the tree is only three or four levels deep. The database reads a handful of index pages, then fetches the 20 matching rows.
Growth	If the table grows to 100 million rows, the scan does ten times the work; the index lookup adds at most one more level.
Cost paid elsewhere	Every new order now writes to the table and to the index. Disk use grows by the size of the index.

This is O(n) against O(log n) from Section 0.2: often the difference between a response in seconds and one in milliseconds. Many slow-query complaints in young systems are fixed by the right index, not bigger hardware.

Transactions and ACID: the bank transfer

Moving 100 pounds from Alice's account to Bob's needs two changes: subtract 100 from Alice, add 100 to Bob. If the server crashes after the first change and before the second, 100 pounds has vanished. If two transfers from Alice run at once, both might see a balance of 150 and both might succeed, leaving her overdrawn. A transaction groups several operations into one unit of work that either completes entirely (*commit*) or has no effect at all (*rollback*).

The guarantees a transaction gives are summarised as ACID, an acronym coined by Theo Härder and Andreas Reuter in 1983, building on earlier work by Jim Gray:

Property	What it promises	In the bank transfer
Atomicity	All the changes in the transaction happen, or none do.	Alice is never debited without Bob being credited.
Consistency	A transaction moves the database from one valid state to another; rules such as keys and checks are never left broken.	A rule that balances may not go below zero rejects a transfer Alice cannot afford.
Isolation	Transactions running at the same time do not see each other's half-finished work; the result is as if they ran one after another (to a degree set by the isolation level).	Two simultaneous transfers from Alice cannot both rely on the same old balance.
Durability	Once the database confirms a commit, the change survives a crash or power cut.	When the app says "Transfer complete", the money has really moved, because the change is already in the log on disk.

```
import sqlite3

conn = sqlite3.connect("bank.db")
# The accounts table was created with CHECK (balance ⩾ 0)

def transfer(from_id, to_id, amount):
    try:
        with conn:  # one transaction: commit on success, rollback on error
            debit = conn.execute(
                "UPDATE accounts SET balance = balance - ? WHERE id = ?",
                (amount, from_id))
            credit = conn.execute(
                "UPDATE accounts SET balance = balance + ? WHERE id = ?",
                (amount, to_id))
            if debit.rowcount ≠ 1 or credit.rowcount ≠ 1:
                raise ValueError("No such account")  # forces a rollback
        return "Transfer complete"
    except (sqlite3.Error, ValueError):
        return "Transfer refused: no account was changed"
```

A transfer using Python's built-in sqlite3 module. The with block commits if both updates succeed and rolls back if either fails or an account does not exist.

Note the check on `rowcount` (the number of rows each update changed): an update that matches no account is not an error to the database, so the code must treat it as one itself. Two further details matter beyond this example. The `?` placeholders pass values separately from the SQL text, which is the standard defence against *SQL injection*, an attack in which user input

is smuggled into a query as code (Chapter 8 returns to this). And the durability promise is kept by a *write-ahead log*: the DBMS records each change in an append-only log on disk before confirming the commit, so after a crash it can replay or undo work to reach a correct state.

Isolation is the subtlest property. Full isolation (called *serializable*) is safe but limits how much work can run in parallel, so databases offer weaker *isolation levels* that trade some protection for speed, and many use a weaker level by default. Which anomalies each level allows is covered in transactions and isolation. For now, the key idea is that a single database gives strong, well-understood guarantees within one transaction, and those guarantees are much harder to obtain once data is spread across several databases or services.

NoSQL: other shapes of data

Relational databases are not the only option. NoSQL is an umbrella name, popularised at a 2009 meetup in San Francisco, for databases that do not use the relational model. They grew up to handle very large volumes spread across many machines, flexible data shapes, and access patterns that tables serve awkwardly. The name says what they are not, so it covers quite different families.

The four main NoSQL families

Family	How it stores data	Good fit	Examples
Document database	Each record is a self-contained document, usually JSON-like, which can nest lists and sub-objects. Documents in one collection need not share exactly the same fields.	Data read and written as a whole: a product catalogue with varied attributes, a user profile, content for a web page.	MongoDB, Couchbase, Firestore
Key-value store	A giant dictionary: a unique key maps to a value the database does not look inside. Lookup is by key only.	Very fast simple lookups: sessions, shopping baskets, caches, feature flags.	Redis, Amazon DynamoDB, etcd
Wide-column store	Rows identified by a key, each with many columns, grouped and spread (partitioned) across many machines. Tables are designed around the queries that will be run.	Huge volumes of writes, such as sensor readings, event logs and messaging history, across many servers and data centres.	Apache Cassandra, Apache HBase, Google Bigtable
Graph database	Nodes (things) and edges (relationships), both with properties, as in the graphs of Section 0.2.	Questions about connections: friends of friends, fraud rings, route planning, recommendations.	Neo4j, Amazon Neptune

Practitioners disagree about the default. One camp says to start with a relational database unless there is a clear reason not to: SQL is widely known, transactions and joins are powerful, and modern relational systems also store JSON documents well. The other camp says to choose the store that matches the access pattern from the start, because a document or key-value store can be simpler and scale more cheaply for the right workload. Both agree that many real systems use more than one kind, each where it fits, which is called *polyglot persistence*. Chapter 7's section on choosing a database gives a structured way to decide.

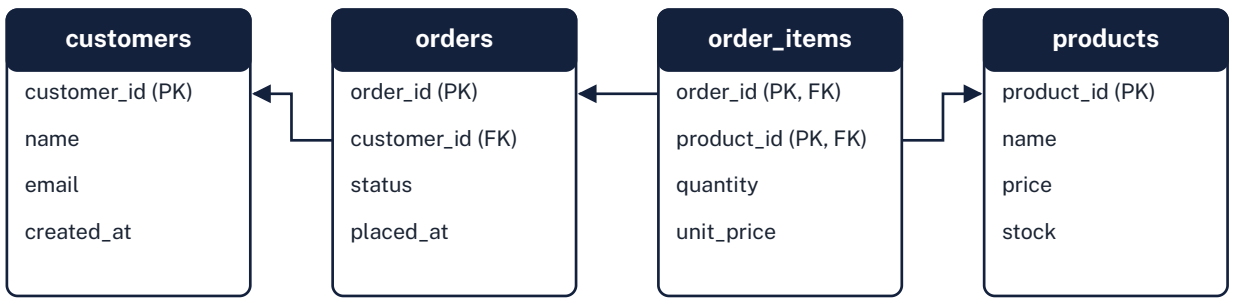
WATCH OUT

"Schemaless" does not mean "no schema"

A document database will accept a customer document with `email` today and `emailAddress` tomorrow. The data still has a shape; it is now enforced, or not, by every program that reads it instead of by the database. Teams that skip schema thinking in NoSQL stores usually rediscover it later, in the form of code full of special cases for old documents.

Worked example: an online shop's schema

Consider a small online shop. It needs to know who its customers are, what it sells, and who ordered what. A first attempt might put everything in one table with a row per order and columns `product1`, `product2`, `product3`. That breaks as soon as someone orders four things, and it copies the customer's name and address into every order.



Each arrow runs from a foreign key (FK) to the primary key (PK) it refers to.

A normalised shop schema: each fact is stored once, and foreign keys record the relationships.

WORKED EXAMPLE

Worked example: why the schema has four tables

The schema above answers the shop's questions without copying facts.

customers	One row per person. Changing an email address changes one row.
orders	One row per checkout. `customer_id` is a foreign key, so every order belongs to a real customer. One customer has many orders: a *one-to-many* relationship.
products	One row per item for sale, with the current price and stock level.
order_items	An order contains many products and a product appears in many orders: a *many-to-many* relationship. Relational databases represent it with a linking table, one row per product per order.
Deliberate copy: unit_price	The price is copied into `order_items` when the order is placed. This is not a normalisation mistake: the order must record what the customer actually paid, even after the catalogue price changes next week.
Placing an order	Inserting the order, inserting its items and reducing each product's stock happen in one transaction, so a crash can never leave an order without items or stock that was never reduced.

Good schema design starts from the questions the business asks and the facts it must never lose, not from the screens of the first version of the app.

The same shop in a document database might store each order as one document with its items nested inside, which suits reading a whole order at once but makes questions such as "total sales per product" harder.

Backups, recovery and managed databases

Disks fail, people run the wrong `DELETE`, software has bugs and attackers encrypt data for ransom. A backup is a copy of the data kept somewhere separate, from which the database can be restored. Three professional habits matter. A backup that has never been restored is a hope, not a backup, so teams rehearse and time restores. Copies are kept in more than one place (a common rule of thumb is three copies, on two kinds of media, one off-site). And a *replica*, a second database that continuously copies the first, protects against a machine failing but not against mistakes: a bad `DELETE` reaches the replica within milliseconds. Many databases can combine a periodic full backup with the saved transaction log to restore to any moment, called *point-in-time recovery*.

Running a database well (patching, tuning, replicating, backing up and restoring at three in the morning) is skilled, never-ending work, so cloud providers sell a managed database: Amazon RDS, Azure SQL Database and Google Cloud SQL are examples. The provider handles the machines, software updates, automated backups and failover to a standby copy; the customer still owns the schema, the queries, the indexes, access permissions and the cost. Managed services are covered more broadly in servers and the cloud.

Why this matters for architecture

Data usually outlives the code around it: applications are rewritten every few years, while customer records and medical histories are kept for decades. Decisions about where data lives, which kind of database holds it, which program owns it and which guarantees it needs are therefore some of the most expensive to reverse. Several big architectural questions grow directly from this section: whether each service should own its own database (data ownership), which database fits which job (choosing a database), and how to keep data correct when one business action touches several databases, where a single ACID transaction is no longer available (sagas). When an architect asks "where is the source of truth for this?", this section is the reason.

CHECK YOUR UNDERSTANDING

▼ Why is a spreadsheet file a poor place to keep an online shop's orders once many customers are buying at once?

Simultaneous writers can overwrite each other, a crash mid-save can corrupt the file, finding one customer's orders means reading everything, and nothing enforces rules such as every order having a real customer. A DBMS solves all four.

▼ In the shop schema, what makes customer_id in the orders table a foreign key, and what does that buy you?

It holds the primary key of a row in the customers table and is declared as a reference to it. The database then refuses orders for customers that do not exist.

▼ Why not simply index every column of every table?

Each index must be updated on every insert, update and delete, and takes space. Too many indexes make writes slow and storage expensive, so indexes are chosen to match the queries actually run.

▼ Which ACID property is broken if a crash leaves Alice debited but Bob not credited?

Atomicity: the two changes should have happened together or not at all.

▼ **A team says "we have a replica in another data centre, so we do not need backups". What have they missed?**

A replica copies mistakes and corruption as faithfully as good data, so an accidental delete or a ransomware attack reaches it within moments. Backups, ideally with point-in-time recovery, protect against that.

SOURCES

- PostgreSQL documentation: Transactions (tutorial): https://www.postgresql.org/docs/current/tutorial-transactions.html
- PostgreSQL documentation: Indexes, introduction: https://www.postgresql.org/docs/current/indexes-intro.html
- PostgreSQL documentation: Joins between tables: https://www.postgresql.org/docs/current/tutorial-join.html
- Martin Fowler: NosqlDefinition: https://martinfowler.com/bliki/NosqlDefinition.html

SECTION 0.6 OF 0.8 · BEGINNER · 17 MIN READ

APIs, JSON and how separate systems work together

THE IDEA IN BRIEF

An API is like a restaurant menu and its waiter. The menu tells you exactly what you can order and what you will get; the waiter takes your order to the kitchen and brings back the dish. You never go into the kitchen, and the kitchen can change how it cooks without you noticing, as long as the dishes on the menu stay the same. Software uses APIs in the same way: one program asks another for something in an agreed format and gets an agreed answer back.

WHAT YOU NEED TO KNOW

- An API is a contract between pieces of software: what can be requested, with what inputs, and what comes back, while the implementation behind it stays hidden and free to change.
- In-process APIs are function calls inside one program; web APIs cross a network, so they are slower and can fail in ways a function call cannot.
- REST APIs expose resources at URLs, act on them with HTTP methods (GET, POST, PUT, PATCH, DELETE), report outcomes with status codes and usually exchange JSON.
- API keys identify a calling application; access tokens are short-lived credentials, often on behalf of a user; both must travel only over HTTPS and stay out of source code.
- A webhook is the provider calling the client's URL when an event happens; receivers must verify signatures and handle duplicates.
- Synchronous calls wait for an answer; asynchronous messages go through a queue so that sender and receiver need not be available at the same time.
- Breaking changes hurt clients, so APIs are versioned, and rate limits (HTTP 429) protect providers; every third-party API is a dependency to isolate and plan around.

What an API is

An API (application programming interface) is the defined way one piece of software asks another to do something. It lists what can be requested, what must be supplied, what comes back and what can go wrong. The word that matters is *defined*: an API is a contract. The provider promises to keep it working; the consumer promises to use only what it describes.

A restaurant is the usual analogy. The menu lists what you can order and what each dish contains; the waiter carries your order to the kitchen and brings the result back. You do not walk into the kitchen, and you do not need to know whether the chef uses gas or induction. The

kitchen can change its equipment, staff and suppliers at will, as long as the dishes on the menu still arrive as described. The menu is the API, the kitchen is the implementation, and the separation between them is what lets both sides change independently.

This is the same idea as the interfaces in organising code, applied at every scale. It is also why architects care so much about APIs: an API is where one team's work meets another's, and once others depend on it, it is expensive to change.

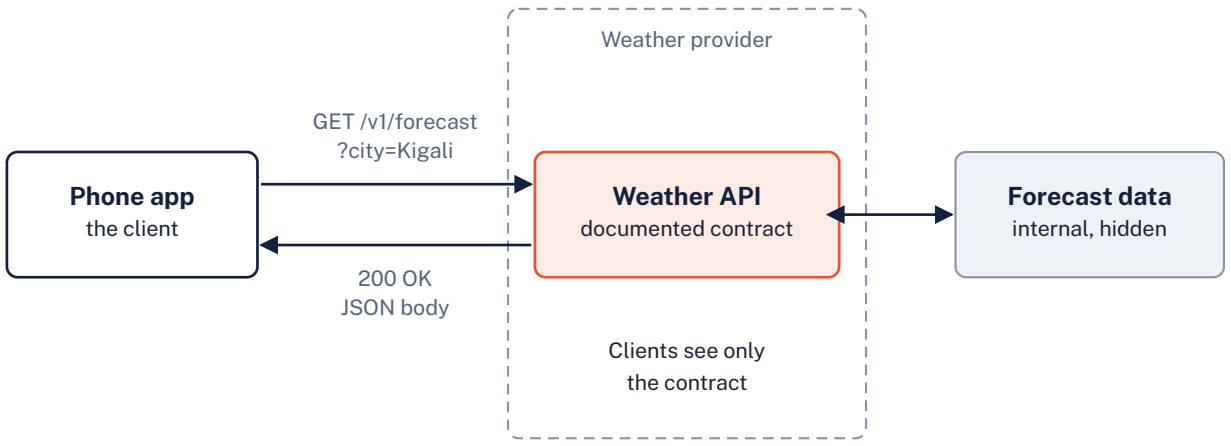
In-process APIs and web APIs

Some APIs are called inside one running program. When Python code calls `json.loads(text)`, it uses the API of the standard library's `json` module: a function call that takes nanoseconds and fails only if the input is bad. A library's public functions and classes are its in-process API.

A web API is called over a network, usually with HTTP (see how computers talk). The caller and the provider can be on different machines, written in different languages and owned by different organisations. A mobile banking app talks to the bank's web API; the bank's servers talk to a card network's API; a shop talks to a payment provider's API.

The difference is not cosmetic. A network call is thousands to millions of times slower than a function call, and it can fail in ways a function call cannot: the network drops the request, the server is overloaded, the answer arrives after the caller has given up, or the request succeeded but the response was lost, so the caller cannot tell whether the payment went through. Much of distributed-systems design, covered in Chapter 6, is about living with these failures.

A web API call



The client sees only the API's contract; how the provider stores and computes forecasts stays hidden behind it.

REST: resources, URLs and verbs

Most web APIs today follow the style called REST (Representational State Transfer), described by Roy Fielding in his 2000 doctoral dissertation. Fielding defined REST as a set of constraints for large networked systems such as the web; in everyday use, "a REST API" means something simpler: an HTTP API organised around **resources**.

A **resource** is a thing the API exposes: a customer, an order, a forecast. Each has a URL, such as `/orders/1042`. A collection of them also has a URL, such as `/orders`. The client acts on resources with the standard HTTP methods, reads the result's status code, and exchanges data in the body, usually as JSON. The same few verbs therefore work for every resource, which makes well-designed REST APIs predictable.

One resource, the standard verbs

Request	Meaning	Typical success status
`GET /orders?customer=42`	List orders, filtered by a query parameter. Reads never change anything.	200 OK
`GET /orders/1042`	Fetch one order.	200 OK (or 404 Not Found if there is no such order)
`POST /orders`	Create a new order from the JSON in the request body.	201 Created, with the new order's URL
`PUT /orders/1042`	Replace order 1042 with the representation sent.	200 OK or 204 No Content
`PATCH /orders/1042`	Change some fields only, such as the delivery address.	200 OK
`DELETE /orders/1042`	Cancel or remove the order.	204 No Content

Status codes carry meaning a client program can act on. A **4xx** code says the client did something wrong: 400 for a malformed request, 401 when credentials are missing or invalid, 403 when the caller is known but not allowed, 404 when the resource does not exist, 429 when it has sent too many requests. A **5xx** code says the server failed: 500 for an unexpected error, 503 when it is temporarily unavailable. The distinction matters because retrying a 4xx request unchanged will fail again, while retrying a 503 a little later may succeed.

Data travels as JSON (JavaScript Object Notation), a text format standardised as RFC 8259. It has only a few building blocks: objects (name-value pairs inside braces), arrays (lists inside brackets), strings in double quotes, numbers, `true`, `false` and `null`. Despite the name, every mainstream language can read and write it, which is why it became the common

language of web APIs. Other styles exist and suit different needs: GraphQL lets the client specify exactly which fields it wants, and gRPC uses a compact binary format for fast service-to-service calls; Chapter 6 compares them.

WORKED EXAMPLE

Worked example: calling a weather API

A travel app wants tomorrow's forecast for Kigali. The provider (here a made-up api.example.com) documents an *endpoint* (a URL that accepts a particular kind of request) and issues the app an API key.

1. Read the contract	The documentation says: `GET /v1/forecast` with query parameters `city` and `days`; send the key in an `Authorization` header; responses are JSON; limit 60 requests per minute.
2. Send the request	The app sends the HTTP request shown below, over HTTPS so the key and data are encrypted in transit.
3. Check the status	200 means the body holds a forecast. 401 means the key is wrong; 429 means slow down; 503 means try again later.
4. Parse the JSON	The app reads `days[0].max_c` and `days[0].rain_chance` and ignores fields it does not need.
5. Handle failure	If no answer arrives within, say, 3 seconds, the app shows the last cached forecast rather than a blank screen.

Every step depends on the contract, not on how the provider computes forecasts. The provider can move to new servers or a new weather model tomorrow without the app changing.

Step 5 is the step beginners forget: any call across a network can be slow or fail, so the caller must decide what to do when it does.

```
GET /v1/forecast?city=Kigali&days=2 HTTP/1.1
Host: api.example.com
Authorization: Bearer sk_test_4f9a...
Accept: application/json

HTTP/1.1 200 OK
Content-Type: application/json

{
  "city": "Kigali",
  "units": "metric",
  "days": [
    {"date": "2026-10-23", "min_c": 16, "max_c": 27, "rain_chance": 0.6},
    {"date": "2026-10-24", "min_c": 16, "max_c": 26, "rain_chance": 0.4}
  ]
}
```

The request and response from the worked example.

```
import os
import requests

API_KEY = os.environ["WEATHER_API_KEY"]   # never hard-code secrets

resp = requests.get(
    "https://api.example.com/v1/forecast",
    params={"city": "Kigali", "days": 2},
    headers={"Authorization": f"Bearer {API_KEY}"},
    timeout=3,                # give up after 3 seconds
)
resp.raise_for_status()       # raises an error for 4xx and 5xx
tomorrow = resp.json()["days"][0]
print(tomorrow["max_c"], "degrees, rain chance", tomorrow["rain_chance"])
```

The same call from Python with the widely used requests library, including a timeout and status check.

Who is calling? API keys and tokens

Most APIs need to know who is calling, to bill them, limit them and decide what they may see. Two ideas are worth separating from the start. *Authentication* answers "who are you?"; *authorisation* answers "what are you allowed to do?"

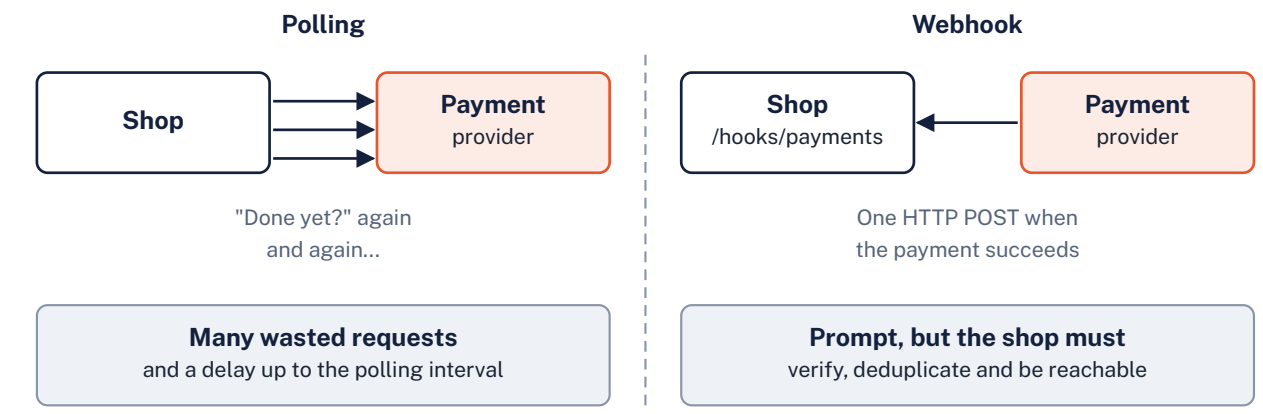
An API key is a long secret string that identifies the calling application, sent with each request, typically in a header. It is simple, and it identifies a program rather than a person. An access token is a short-lived credential issued after someone signs in, often on behalf of a particular user and with particular permissions ("read this user's calendar, but not their email"); it is

usually sent as `Authorization: Bearer <token>`. Tokens expire, so a stolen one is useful for a limited time. The standard protocols behind them, OAuth 2.0 and OpenID Connect, are covered in identity and access.

Whichever is used, the rules are the same: send credentials only over HTTPS, keep them out of source code and out of mobile apps where anyone can extract them, store them in a secrets manager or environment settings, and be able to revoke and replace them quickly.

Webhooks: the API calls you back

Sometimes the client needs to know when something happens on the provider's side: a payment succeeds, a parcel is delivered, a document is signed. The client could *poll*, asking "has it happened yet?" every few seconds, which wastes requests and still notices late. A webhook reverses the direction: the client registers a URL of its own, and the provider sends an HTTP POST to that URL when the event occurs. The term was coined by Jeff Lindsay in 2007.



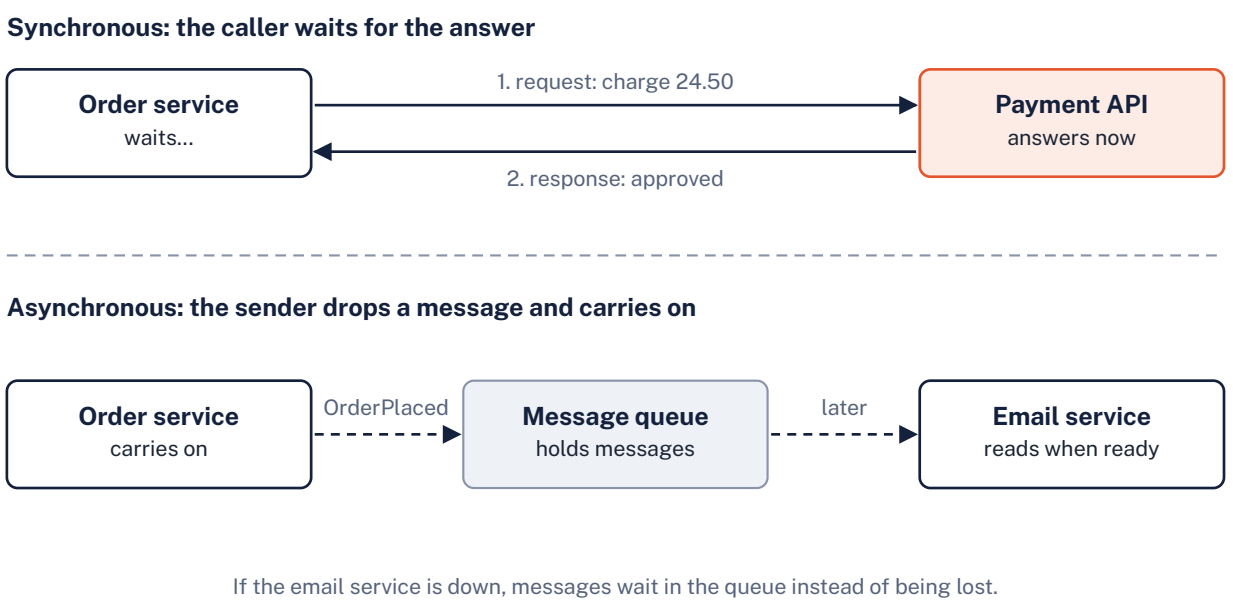
Polling asks repeatedly; a webhook waits to be told, at the cost of new responsibilities for the receiver.

Webhooks bring responsibilities. The receiving endpoint is on the public internet, so it must check that each call really comes from the provider, usually by verifying a cryptographic signature in a header. Providers retry when the receiver does not answer, so the same event can arrive twice and the receiver must recognise duplicates (processing a duplicate must not ship the order twice). And events can arrive out of order.

Waiting for an answer, or not: synchronous and asynchronous

Everything so far has been synchronous: the caller sends a request and waits for the response before continuing, like a phone call. It is simple to reason about, and it is the right choice when the caller genuinely needs the answer now, such as "was this card payment approved?". Its weakness is that the caller is only as available and as fast as everything it is waiting on.

The alternative is asynchronous messaging, more like leaving a letter in a post box. The sender puts a message into a message queue and carries on. The receiver takes messages from the queue when it is ready. If the receiver is slow or down for an hour, messages wait instead of failing. When the shop confirms an order, sending the confirmation email does not need to hold up the customer's screen; a message saying `OrderPlaced` can go into a queue for the email service to handle a moment later.



A synchronous call ties the caller to the callee's availability; a queue decouples them in time.

Asynchrony is not free. The sender does not learn at once whether the work succeeded, the system becomes harder to trace and test, and messages may be delivered more than once. Deciding which interactions should be synchronous and which asynchronous is one of the central decisions in distributed systems, treated properly in communication and APIs and queues and back pressure.

DECISION GUIDE

Decision guide: call now or send a message?

A first-pass guide for one interaction between two parts of a system. Chapter 6 refines it.

Q1 **Does the caller need the result before it can answer its own user (for example, was the card approved)?**

- **Yes** Use a synchronous call, with a timeout and a plan for when the other side is slow or down.
- **No** Go to the next question.

Q2 **Can the work happen a few seconds or minutes later without anyone noticing (sending an email, updating a report)?**

- **Yes** Send an asynchronous message through a queue, so a slow or failed receiver does not hold up the caller.
- **No** Go to the next question.

Q3 **Is the work slow (seconds to hours), such as generating a large export?**

- **Yes** Accept the request at once (for example with 202 Accepted), do the work from a queue, and tell the client when it is done by a webhook or a status URL it can check.
- **No** A simple synchronous call is usually fine; keep it fast and set a timeout.

Living with other people's APIs: SDKs, versions and limits

Many providers publish an SDK (software development kit): a library for each popular language that wraps the HTTP calls in ordinary functions, so a developer writes `client.refunds.create(payment_id)` instead of building requests by hand. SDKs save time and handle details such as retries and signatures, but they are another dependency to keep up to date.

Once an API has users, changing it becomes delicate. Adding a new optional field to a response is usually safe. Renaming a field, removing one, changing its type or meaning, or making an optional input compulsory is a breaking change: every client that relied on the old behaviour breaks, often in production and often for a provider who cannot see or update those clients (think of mobile apps that users never update). Providers therefore version their APIs, for example `/v1/` and `/v2/` in the URL or a version header, keep the old version running for a published period, and announce *deprecations* (notices that a feature will be withdrawn) well in

advance. Hyrum Wright's observation, known as Hyrum's law, adds a warning: with enough users, someone will depend on every observable behaviour of an API, including ones the documentation never promised.

Providers also protect themselves with a rate limit: a cap such as 100 requests per minute per key. Requests above the limit are refused with status 429 Too Many Requests, often with a `Retry-After` header saying when to try again. Clients should respect it, spread their requests, cache answers that do not change often, and back off when told to.

Using third-party APIs: build or buy

Modern applications are assembled as much as written. A food-delivery app might use a payment provider's API to take card payments, a mapping API for addresses and routes, an email or SMS API for receipts, and a push-notification service for "your rider is nearby". Each saves months of specialist work and brings expertise (card security, global map data, email deliverability) that few teams could match.

Each also brings a dependency the team does not control: the provider's outages become your outages, its price changes become your costs, its data-handling becomes part of your compliance story, and leaving it may mean rewriting code. Professionals manage this deliberately. They isolate each provider behind a small interface of their own, so that only one part of the code knows the provider's API; set timeouts and decide what happens when the provider is down; watch the provider's status page and change log; and keep a note of what it would take to switch.

Why this matters for architecture

Architecture is largely about the parts of a system and the contracts between them, and APIs are those contracts made concrete. An API that is easy to use correctly and hard to break lets teams work in parallel and change their internals freely; a leaky or unstable one couples everyone to everyone. The choice between synchronous calls and asynchronous messages shapes how failures spread. Chapter 3 explains the principles behind good interfaces (information hiding and abstraction), and Chapter 6 shows how API and messaging choices make or break distributed systems.

CHECK YOUR UNDERSTANDING

▼ In the restaurant analogy, what corresponds to the API and what to the implementation?

The menu (and the waiter's way of taking orders) is the API: the agreed list of what can be requested and what comes back. The kitchen is the implementation, free to change as long as the menu's promises are kept.

▼ What is the difference between a 404 and a 503 response, and which is worth retrying later?

404 Not Found says the requested resource does not exist, a client-side problem that retrying will not fix. 503 Service Unavailable says the server is temporarily unable to respond, so retrying after a delay may succeed.

▼ Why must a webhook receiver be ready to see the same event twice?

Providers retry when they do not get a timely success response, even if the first delivery was actually processed, so duplicates are normal. The receiver must recognise an event it has already handled and not act on it again.

▼ Is adding a new optional field to a JSON response normally a breaking change?

Usually not, because well-behaved clients ignore fields they do not recognise. Removing or renaming a field, or changing its type, is breaking.

SOURCES

- Roy Fielding (2000), Architectural Styles and the Design of Network-based Software Architectures, Chapter 5: REST: https://ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm
- RFC 8259: The JavaScript Object Notation (JSON) Data Interchange Format: https://www.rfc-editor.org/rfc/rfc8259
- MDN Web Docs: 429 Too Many Requests: https://developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Status/429
- Stripe documentation: Receive events in your webhook endpoint: https://docs.stripe.com/webhooks
- Hyrum's Law: https://www.hyrumslaw.com/

SECTION 0.7 OF 0.8 · BEGINNER · 18 MIN READ

Servers, operating systems, virtual machines, containers and cloud services

THE IDEA IN BRIEF

A server is just a computer whose job is to answer requests, kept in a large building full of other servers called a data centre. Instead of buying their own, most organisations now rent computing from cloud providers, much as homes buy electricity from the grid rather than running a generator. You can rent a whole virtual computer, a ready-made platform for your code, or finished software, and pay only for what you use. The more you rent ready-made, the less you have to look after, and the less you control.

WHAT YOU NEED TO KNOW

- A server is any computer, or program, that waits for requests and answers them; most internet servers run Linux and live in data centres built for power, cooling and security.
- A hypervisor lets one physical machine run several virtual machines, each with its own guest operating system, isolated from the others.
- A container packages an application with its libraries and settings and runs as an isolated process sharing the host's kernel, so it is lighter and faster to start than a VM but less strongly isolated.
- Cloud computing (NIST SP 800-145) means on-demand, self-service, pay-as-you-go access to pooled, elastic resources over the network.
- IaaS, PaaS, serverless (FaaS) and SaaS hand progressively more of the stack to the provider, trading control for less operational work; customers always remain responsible for their data and configuration.
- Regions are isolated geographic areas; availability zones are separate data-centre groups within a region; CDNs serve static content from edge locations near users.
- Scaling up means a bigger machine; scaling out means more machines behind a load balancer, which scales further and survives failures only if the software is designed for it.

A server is just a computer that serves

A server is a computer whose job is to wait for requests and answer them: web pages, API calls, database queries, files. The word describes a role, not a special kind of machine. A laptop running a small web application is a server while it answers requests; a rack-mounted machine

in a data centre is a server because that is all it ever does. The same word is also used for the program doing the serving (a "web server" such as nginx), so context matters.

Server hardware has the same parts as any computer (see what software is): processors, memory, storage and network cards. It differs in emphasis. Servers usually have many processor cores, large amounts of memory, redundant power supplies and disks so that one failed part does not stop the machine, remote management so that nobody has to be in the room, and no screen or keyboard at all. They are built to run for years without being switched off.

Most servers on the internet run Linux or another Unix-like operating system; Windows Server is common inside organisations that rely on Microsoft products. A server operating system does the same jobs as the one on your laptop (running processes, managing memory and files, talking to the network) but is usually administered through a text command line over a remote connection, and is configured so that it can be rebuilt automatically rather than by hand.

Data centres: where the servers live

Servers are kept in a data centre: a building designed around their needs. Thousands of machines sit in racks, cooled by air or liquid because they turn almost all of their electricity into heat. Power arrives over more than one supply, backed by batteries for the first seconds of an outage and by diesel generators after that. Network connections come from several providers. Doors need badges, and often fingerprints.

An organisation can run its own data centre or rent rack space in someone else's (called *colocation*), and before about 2006 that was nearly the only choice. Running servers yourself (*on-premises*) means buying machines months before you need them, guessing how many, paying for them whether they are busy or idle, and employing people to replace failed disks at night. That cost and inflexibility is the problem the cloud set out to solve.

Virtual machines: many computers inside one

A typical server sits mostly idle if it runs one application. A virtual machine (VM) lets one physical machine pretend to be several. Each VM believes it is a complete computer with its own processors, memory, disk and network card, and runs its own operating system, called the *guest*. A layer of software called the hypervisor shares out the real hardware among the VMs and keeps them apart, so that a crash or attack in one does not reach the others. Some hypervisors run directly on the hardware (VMware ESXi, Microsoft Hyper-V, KVM, which is part of Linux); others run as an application on a normal operating system (VirtualBox).

Virtualisation changed the economics of servers. One machine can be packed with VMs from different teams or customers. A VM is a set of files, so it can be copied, backed up, moved to another machine and created in minutes instead of being ordered and delivered. Note that this is a different use of "virtual machine" from the language VMs such as the Java Virtual Machine mentioned in what software is: those run programs, these imitate whole computers.

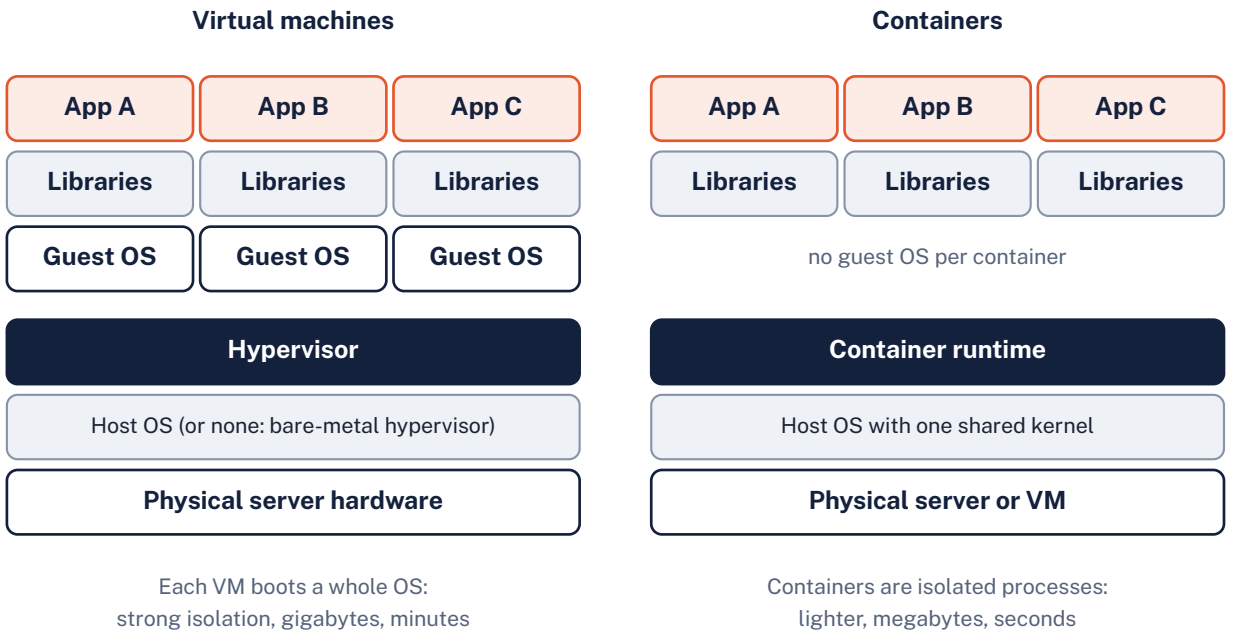
Containers: shipping the application with everything it needs

Every developer has heard "it works on my machine". The program depends on a particular language version, particular libraries and particular settings, and the server has slightly different ones. A container solves this by packaging an application together with everything it needs to run (its runtime, libraries, configuration) into a standard unit that runs the same way on a laptop, a test server or the cloud. The name comes from shipping containers: standard boxes that any crane, ship or lorry can handle without knowing what is inside.

The package is a container image: a read-only, versioned file built from a short recipe (with Docker, a `Dockerfile`). A running copy of an image is a container. Docker, released in 2013, made this practical for everyday developers, and its image format is now an open standard maintained by the Open Container Initiative, so other tools can build and run the same images.

The key difference from a VM is what is shared. A VM imitates hardware and runs a whole guest operating system. A container is an isolated process: it shares the host's operating-system *kernel* (the core of the operating system, which controls the hardware, memory and processes) with other containers, and the kernel's isolation features give it its own view of files, processes and network. Containers therefore start in seconds or less and take megabytes rather than gigabytes, so dozens can run where a few VMs would fit. The price is weaker isolation, because every container depends on the one shared kernel; that is why cloud providers commonly run customers' containers inside VMs, combining both.

Virtual machines and containers compared



VMs virtualise the hardware and each carries an operating system; containers virtualise the operating system and share its kernel.

```
# Start from an image that already has Python installed
FROM python:3.12-slim
WORKDIR /app
# Bake the libraries into the image
COPY requirements.txt .
RUN pip install -r requirements.txt
# Add the application code
COPY . .
EXPOSE 8000
CMD ["python", "-m", "uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8000"]
```

A complete Dockerfile for a small Python web API. Running docker build turns this recipe into an image.

Running one container is easy. Running hundreds across dozens of machines, restarting the ones that crash, adding copies when traffic rises and rolling out new versions without downtime is not. That job is done by an orchestrator, of which Kubernetes is the best known; it is covered in containers and orchestration.

The cloud: renting computing by the minute

Cloud computing means renting computing resources over the internet, on demand, from a provider that runs huge data centres: Amazon Web Services (AWS), Microsoft Azure and Google Cloud are the largest, with many others. The US National Institute of Standards and

Technology (NIST) published the definition most people cite, Special Publication 800-145, in 2011. It lists five essential characteristics: on-demand self-service (you create resources yourself, through a website or an API, without phoning anyone), broad network access, resource pooling (many customers share the provider's hardware), rapid elasticity (capacity grows and shrinks quickly) and measured service (you pay for what you use).

The analogy is electricity. Factories once ran their own generators; now they plug into the grid and pay per kilowatt-hour. With the cloud, a start-up can rent a hundred servers for an afternoon to test a launch, and stop paying when it hands them back. That is pay-as-you-go pricing. It turns a large up-front purchase into a running cost, and it rewards systems that can switch capacity off when it is not needed. It also creates a new risk: costs that grow quietly, because nothing stops anyone creating resources and forgetting them.

IaaS, PaaS, SaaS and serverless: how much do you manage?

Cloud services differ mainly in how much of the stack the provider manages for you. NIST defined three service models; a fourth, serverless, became popular after AWS launched its Lambda service in 2014. A popular way to remember them is pizza: you can make it at home, buy a ready-made one to bake, have one delivered, or eat out. Each step hands more of the work to someone else, and takes away some of your choices.

Cloud service models compared

Model	You manage	The provider manages	Examples	Pizza analogy
On-premises	Everything: building, hardware, OS, runtime, application, data	Nothing	Your own servers in your own data centre	Made at home from scratch
IaaS (infrastructure as a service)	Operating system, runtime, application, data	Hardware, virtualisation, network, building	Amazon EC2, Azure Virtual Machines, Google Compute Engine	Take-and-bake: you cook it in your own oven
PaaS (platform as a service)	Application code and data	Everything underneath, including OS patching and scaling	Heroku, Azure App Service, Google App Engine	Delivery: you supply the table and drinks
Serverless / FaaS (function as a service)	Individual functions, triggered by events	All servers, scaling to zero when idle, billing per call	AWS Lambda, Azure Functions, Google Cloud Run functions	Pizza by the slice, paid per bite
SaaS (software as a service)	Your data, your users and their settings	The entire application	Microsoft 365, Gmail, Salesforce	Eating at a restaurant

"Serverless" does not mean there are no servers; it means you never see or manage them. You upload a function, say which events trigger it (an HTTP request, a file arriving, a message on a queue), and the provider runs as many copies as needed and charges per execution and per millisecond. It is cheap for spiky or low traffic and needs almost no operations work. Its costs show up elsewhere: the first call after a quiet spell can be slow while the provider starts the function (a *cold start*), there are limits on how long a function may run, and costs can exceed a plain server's for steady heavy traffic. Chapter 6 discusses serverless architectures in depth.

Between these models sit the managed services: databases, message queues, caches, object storage, search engines and many more, which the provider installs, runs, patches, backs up and scales. Using a managed database instead of installing one on a VM moves a large amount of operational work to the provider. The trade-off, discussed in FinOps and lock-in, is that the more provider-specific services a system uses, the harder it is to move.

WATCH OUT

The shared responsibility model

Moving to the cloud does not move all responsibility. Providers state plainly that they secure the cloud itself (buildings, hardware, the virtualisation layer) and the customer secures what they put in it: their data, their access permissions, their configuration and, for IaaS, the operating system. Many well-publicised cloud data leaks were not provider failures but customer-misconfigured storage left open to the internet.

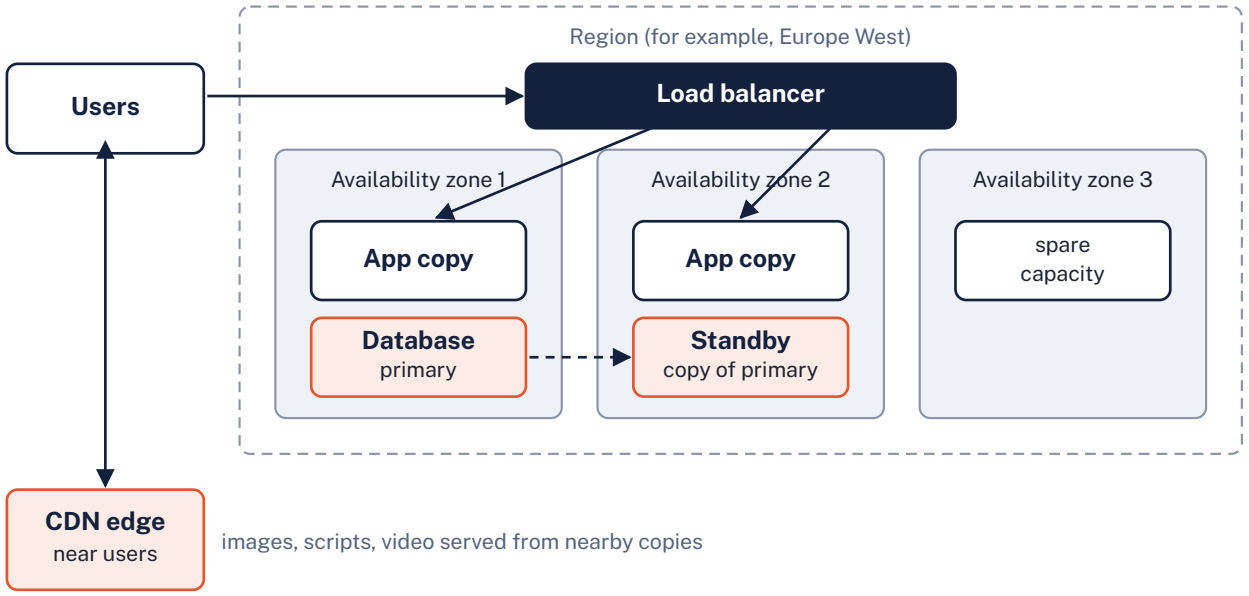
Regions, availability zones and content delivery networks

Cloud providers group their data centres by geography. A region is a geographic area (such as Ireland, Frankfurt or Virginia) with its own separate set of data centres, isolated from other regions. Each region is divided into several availability zones: physically separate groups of one or more data centres, each with independent power, cooling and networking, typically kilometres apart but close enough for fast connections between them. A fire, flood or power failure should take out at most one zone.

This gives architects a menu of protection. Running copies of an application in two or three zones of one region survives the loss of a data centre at little extra latency; running in two regions survives the loss of a whole region but makes keeping data in step much harder. Regions also matter for law and latency: data protection rules may require personal data to stay in a particular jurisdiction, and users are served faster from a region near them, because no network can beat the speed of light (see how computers talk).

A content delivery network (CDN) pushes this further for content that does not change per user, such as images, style sheets, scripts and video. It keeps copies on servers in hundreds of locations, called edge locations, so that a user in Nairobi fetches the shop's product photos from a nearby city rather than from a data centre in Europe. The page loads faster, and the shop's own servers handle far fewer requests. Many CDNs also absorb attack traffic and can run small pieces of code at the edge.

Inside a region



Copies of the application in two availability zones keep the service running if one zone fails, while a CDN serves static content from locations close to users.

Scaling up and scaling out

When a system needs to handle more work there are two basic moves. Scaling up (vertical scaling) means a bigger machine: more cores, more memory. It needs no change to the software, which is its great advantage, but there is a largest machine you can buy, it gets disproportionately expensive near the top, and one machine is still one point of failure. Scaling out (horizontal scaling) means more machines sharing the work, usually behind a load balancer that spreads requests across them and stops sending to any that fail. It can grow far further and survive machine failures, but only if the software is designed for it: for example, a server must not keep a user's session only in its own memory, or save uploaded files only on its own disk, if the user's next request might go to a different server.

The cloud makes scaling out attractive because adding a machine is an API call, and *autoscaling* can add and remove copies automatically as load changes. Databases are usually the hardest part to scale out, which is why Chapter 7 spends a section on replication and partitioning.

WORKED EXAMPLE

Worked example: hosting a ticket-booking site

A small theatre chain sells tickets online. Traffic is light most of the week, but when a popular show goes on sale, ten times as many people arrive within minutes.

One rented VM (IaaS)	Simple and cheap. The team patches the OS. On sale day the single machine is overwhelmed, and if it fails the site is down.
Bigger VM	Scaling up buys headroom with no code change, but the theatre pays for the big machine all week to cover a few busy hours.
App on a PaaS, two zones, autoscaling	Copies of the app run in two availability zones behind a load balancer. On sale day the platform adds copies automatically and removes them afterwards. Requires the app to keep no session data in local memory.
Managed database	The provider runs a primary in one zone and a standby in another, with automated backups. The team still designs the schema and indexes (see storing data).
CDN for images and scripts	Seating plans and show photos are served from edge locations, so the app servers spend their effort on bookings.

The final design costs more to set up than one VM but less to run than a permanently oversized machine, and it survives both a traffic spike and the loss of a data centre.

Every step traded control for convenience: the team chose which work to keep and which to hand to the provider.

Why this matters for architecture

Where software runs is now an architectural decision rather than an afterthought. The choice between VMs, containers, platforms and functions shapes cost, how quickly the team can release, how the system scales and what happens when a data centre fails. Regions and zones turn availability targets into concrete deployment plans; managed services trade operational work for dependence on a provider. Chapter 9 develops scaling strategies, and Chapter 10 covers cloud service and deployment models, containers and infrastructure in professional depth.

CHECK YOUR UNDERSTANDING

▼ Why do containers start faster and use less space than virtual machines?

A container does not boot its own operating system: it is an isolated process sharing the host's kernel and carries only the application and its libraries. A VM carries and boots a full guest operating system.

▼ A team rents a VM in the cloud and installs its own database on it. Who is responsible for patching the operating system?

The team. With IaaS the provider manages the hardware and virtualisation, and the customer manages the operating system and everything above it.

▼ What failure does running in two availability zones protect against, and what does it not protect against?

It protects against the loss of one data centre or zone (power, fire, network). It does not protect against the loss of the whole region, or against software bugs and bad data, which affect every copy.

▼ Why might an application that works fine on one server misbehave when scaled out behind a load balancer?

It may keep state such as user sessions in its own memory, so when the load balancer sends the next request to another copy, that copy does not know the user.

SOURCES

- NIST SP 800-145: The NIST Definition of Cloud Computing (2011): https://csrc.nist.gov/pubs/sp/800/145/final
- Docker: What is a container?: https://www.docker.com/resources/what-container/
- AWS documentation: Regions and Availability Zones: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-regions-availability-zones.html
- Microsoft Learn: What are availability zones?: https://learn.microsoft.com/en-us/azure/reliability/availability-zones-overview
- MDN Web Docs: CDN (glossary): https://developer.mozilla.org/en-US/docs/Glossary/CDN

SECTION 0.8 OF 0.8 · BEGINNER · 20 MIN READ

Requirements, version control, testing, delivery and the journey of one request

THE IDEA IN BRIEF

Building software is a team sport with a careful routine. The team writes down what users need as short stories, keeps every change to the code in a shared history so nothing is lost, has a colleague check each change, and runs automatic tests that act like a safety net. Changes then travel through practice copies of the system before reaching the real one that customers use. At the end of the section, one tap of a Place order button is followed through every part of a food-delivery system.

WHAT YOU NEED TO KNOW

- Requirements describe what a system must do and how well; user stories (As a..., I want..., so that...) with acceptance criteria are a common lightweight way to capture functional needs.
- Git records every change as commits; branches isolate work in progress, and pull requests bring code review and automated checks before a merge into main.
- The testing pyramid recommends many fast unit tests, fewer integration tests and few slow end-to-end tests.
- Changes move through development, test and staging environments before production; the closer staging is to production, the more problems it catches.
- Continuous integration verifies every change automatically; continuous delivery keeps software always releasable; continuous deployment releases every passing change automatically.
- Waterfall, Scrum and Kanban organise work differently; DevOps is a culture of shared responsibility for building and running software.
- A single request in a real system crosses DNS, HTTPS, a load balancer, services, a database transaction, third-party APIs and queues, and deciding how those parts are arranged is software architecture.

From an idea to a requirement

Software starts with a need: a restaurant wants to take orders online, a hospital wants patients to book appointments themselves. Turning that need into something a team can build is the work of requirements. Some requirements describe what the system must do ("a customer can

pay by card"); others describe how well it must do it ("checkout completes within two seconds for 95% of customers"). Chapter 2 shows that the second kind, the quality attributes, drive most architectural decisions.

Many teams capture functional needs as user stories: short descriptions written from the user's point of view. Kent Beck introduced them in Extreme Programming in the late 1990s, and a team at the London company Connextra devised the template most people now use in 2001:

```
As a hungry customer,
I want to reorder a meal I have ordered before,
so that I can eat sooner without searching the menu again.

Acceptance criteria
- My last 10 orders appear on the home screen.
- Tapping "Reorder" fills the basket with the same items.
- Items that are no longer available are flagged, not silently dropped.
```

A user story with acceptance criteria. The criteria are what testers, and automated tests, will check.

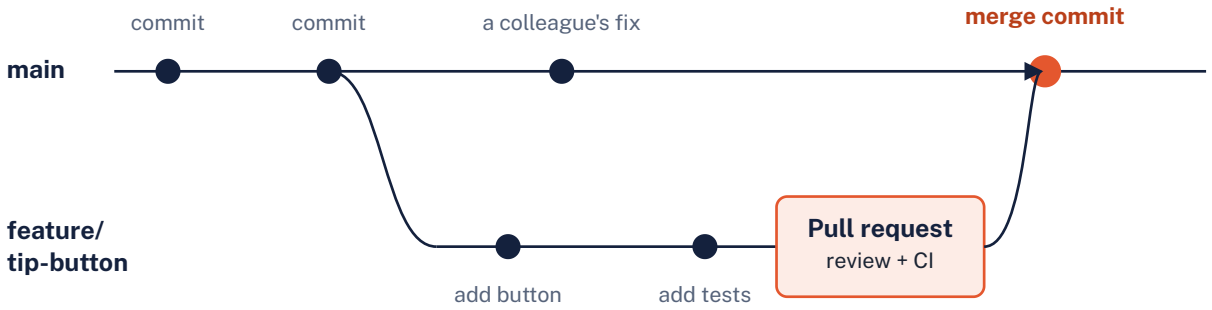
A story is deliberately small and incomplete. Ron Jeffries described it as a card, a conversation and a confirmation: the card is a reminder, the details are worked out in conversation between the people who want the feature and the people building it, and the acceptance criteria confirm when it is done. Practitioners disagree about how much should be written down in advance. Teams in regulated industries such as medical devices and aviation must keep formal, traceable requirements; product teams often prefer lighter stories refined just before work starts. Both are legitimate in their context.

Version control with Git

Source code is the team's most valuable shared document, and many people change it every day. Version control records every change, who made it and why, lets people work in parallel without overwriting each other, and allows any earlier state to be restored. Almost every team now uses Git, created by Linus Torvalds in 2005 for the development of the Linux kernel, usually together with a hosting service such as GitHub, GitLab or Bitbucket.

A few ideas cover most daily use. A commit is a saved snapshot of a set of changes, with a message explaining it ("Add tip button to checkout"). A branch is a separate line of work: a developer creates a branch from `main`, makes commits there without disturbing anyone, and when finished *merges* the branch back, combining the two histories. If two people changed the same lines, Git reports a *merge conflict* for a human to resolve.

On hosting services, merging usually goes through a pull request (GitLab calls it a merge request): a web page showing the proposed change, where automated checks run and colleagues carry out code review, reading the change, asking questions and suggesting improvements before approving it. Review catches defects, but its larger benefits are spreading knowledge (at least two people understand every change) and keeping the design consistent. Architects often take part in reviews of changes that affect structure or interfaces.



The branch keeps work in progress away from main until it has been reviewed and tested.

Branch, commit, review, merge: the everyday cycle of version control.

```
git switch -c feature/tip-button     # create a branch and move onto it
# ... edit files ...
git add checkout.py test_checkout.py
git commit -m "Add tip button to checkout"
git push -u origin feature/tip-button  # upload; then open a pull request
# after review and green checks, the branch is merged into main
```

The everyday Git cycle from the command line.

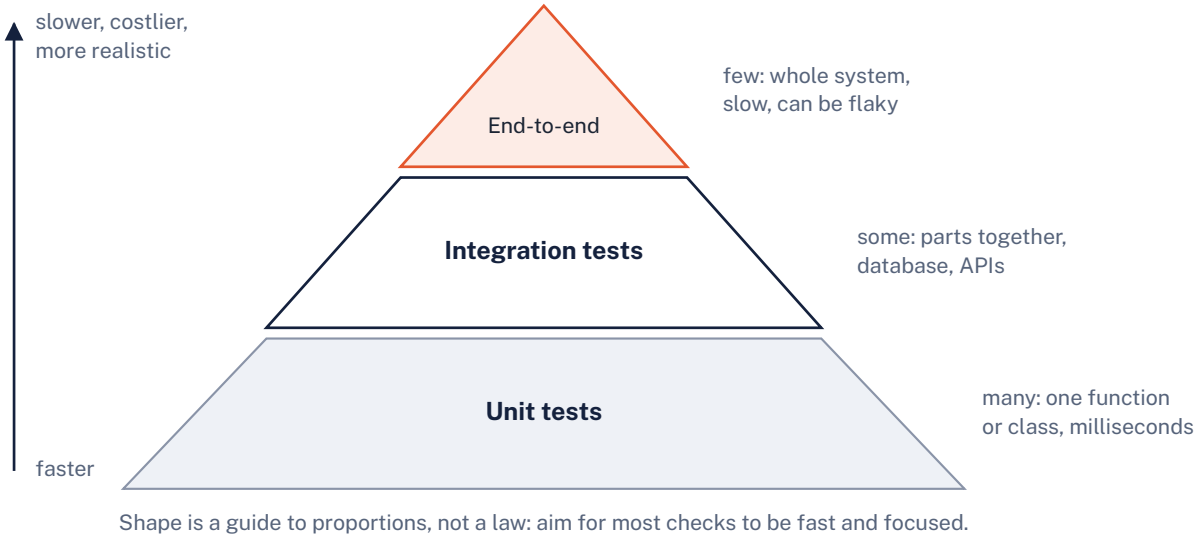
Teams differ on branching. Some keep long-lived branches per feature or release; others practise *trunk-based development*, merging small changes into `main` at least daily and hiding unfinished features behind switches called feature flags. Research by the DORA programme associates short-lived branches with better delivery performance, but the right choice also depends on how a team releases.

Testing: the pyramid

Every change risks breaking something that used to work, which is called a *regression*. Automated tests are programs that check other programs, so they can be run on every change in minutes instead of by people over days. They come in layers. A unit test checks one small piece, such as a function, in isolation, and runs in milliseconds. An integration test checks that

parts work together: the order code with a real database, or the shop with a payment provider's test environment. An end-to-end test drives the whole system as a user would, for example tapping through the app from menu to order confirmation.

Mike Cohn's testing pyramid, published in his 2009 book *Succeeding with Agile*, suggests the proportions: many unit tests, fewer integration tests, and few end-to-end tests. The higher layers are more realistic but slower, more expensive to maintain and more prone to *flaky* failures that come and go for reasons unrelated to the change. Some practitioners argue for a different shape, such as more integration tests for systems that are mostly glue between services; the principle all agree on is to catch most problems with fast, reliable, focused tests.



Many fast unit tests, fewer integration tests and a few end-to-end tests give quick feedback without losing realism.

```
def delivery_fee(basket_total, distance_km):
    if basket_total ⩾ 25:
        return 0.0                      # free delivery on larger orders
    return round(1.50 + 0.40 * distance_km, 2)

def test_free_delivery_over_threshold():
    assert delivery_fee(30.00, 5) == 0.0

def test_fee_grows_with_distance():
    assert delivery_fee(10.00, 5) == 3.50
```

Two unit tests written with pytest for a delivery-fee function. They run in milliseconds and pin down the rules.

Environments: development to production

Code passes through several environments on its way to customers, each a separate copy of the system with its own servers, data and settings. The aim is to find problems where they are cheap and harmless, before they reach production, the live environment real users depend on.

Environment	Who uses it	Data	Purpose
Development	One developer, on a laptop or a personal cloud space	Small, made-up	Write and try code quickly.
Test (or integration)	The team and the automated pipeline	Made-up or anonymised	Run automated and exploratory tests on the combined changes.
Staging (or pre-production)	Testers, product owners, sometimes selected customers	Production-like in shape and volume, never raw personal data without safeguards	Rehearse the release on a copy as close to production as possible.
Production	Real customers	Real	Run the business. Changes arrive only through the pipeline.

The more environments differ from production, the more problems slip through: a test database with a hundred rows will not reveal the missing index that makes a query take forty seconds against ten million. Keeping environments alike is easier when they are built from the same container images and the same scripted configuration, a theme of infrastructure as code.

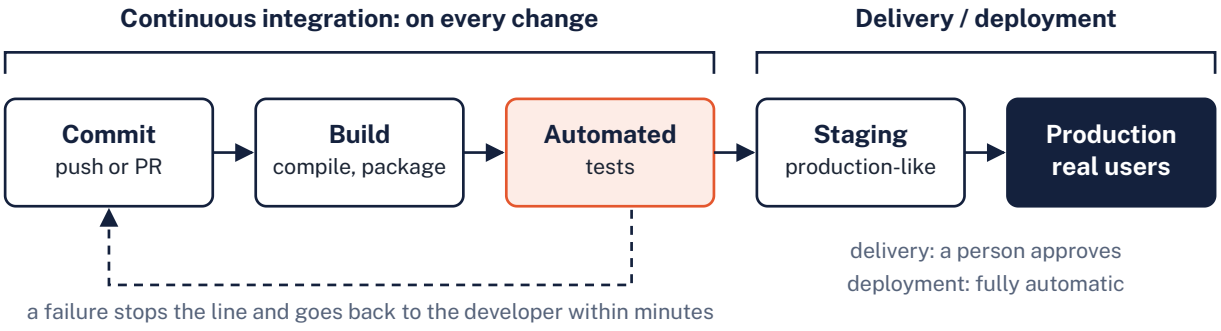
Continuous integration, delivery and deployment

Continuous integration (CI) is the practice of merging everyone's work into the shared main line frequently, at least daily, with an automated build and test run checking every change. It grew out of Extreme Programming in the 1990s. Its point is that integration problems are found within minutes, while the change is fresh in the author's mind, instead of in a painful "integration phase" weeks later. A team practising CI treats a failing build as an emergency: fixing it comes before any new work.

Continuous delivery (CD) extends CI so that the software is always releasable: every change that passes the pipeline could be put into production at the press of a button, and releasing becomes a routine business decision rather than a risky event. The approach was set out by Jez

Humble and David Farley in their 2010 book *Continuous Delivery*. Continuous deployment goes one step further and releases every passing change automatically, with no human approval; teams that do this may deploy many times a day.

The automation that carries a change from commit to production is a pipeline. Because each step is automated and identical every time, releases stop depending on one person remembering a checklist at midnight.



Continuous integration checks every change automatically; delivery and deployment carry the passing build to production.

Checklist: before a change reaches production

What a mature team's process checks, mostly automatically, for every change.

- [ ] **The change is linked to a story or issue**
  Anyone can later find out why it was made.
- [ ] **It is small**
  Small changes are easier to review, test, deploy and, if necessary, undo.
- [ ] **A colleague has reviewed it**
  Through a pull request, with questions answered and comments resolved.
- [ ] **Automated tests pass at every level**
  Unit and integration tests in CI, and the end-to-end checks that cover the affected journey.
- [ ] **It ran in staging**
  The same build, configuration mechanism and deployment steps as production.
- [ ] **There is a way back**
  A rollback or a feature flag can switch it off quickly if something goes wrong.
- [ ] **Someone will notice if it breaks**
  Monitoring and alerts cover what the change touches (Chapter 9, observability).

Ways of working: waterfall, Scrum and Kanban

Teams also choose how to organise the work itself. In a sequential approach, often called *waterfall*, the project moves through phases in order: requirements, design, implementation, testing, deployment. Each phase finishes before the next begins. The name is usually traced to Winston Royce's 1970 paper, although Royce did not use the word and in fact warned that a single pass through the phases was risky. Sequential plans still suit work where requirements are fixed and change is very costly, such as some regulated or hardware-dependent projects. Their weakness for most software is that users see nothing until late, so misunderstandings surface when they are most expensive to fix.

Agile approaches, named in the Agile Manifesto of 2001, deliver working software in short cycles and adjust plans as they learn. Scrum is the most widely used framework: a small team works in fixed cycles called sprints, of one month or less, each starting with planning and ending with a review of the work and a retrospective on how the team worked. The 2020 Scrum Guide defines three accountabilities: the Product Owner (deciding what is most valuable), the Scrum Master (helping the team use Scrum effectively) and the Developers (doing the work).

Kanban, adapted for software from Toyota's production system, has no fixed cycles: work items flow across a visible board (to do, doing, review, done), and the team limits how many items may be in progress at once so that work is finished before more is started.

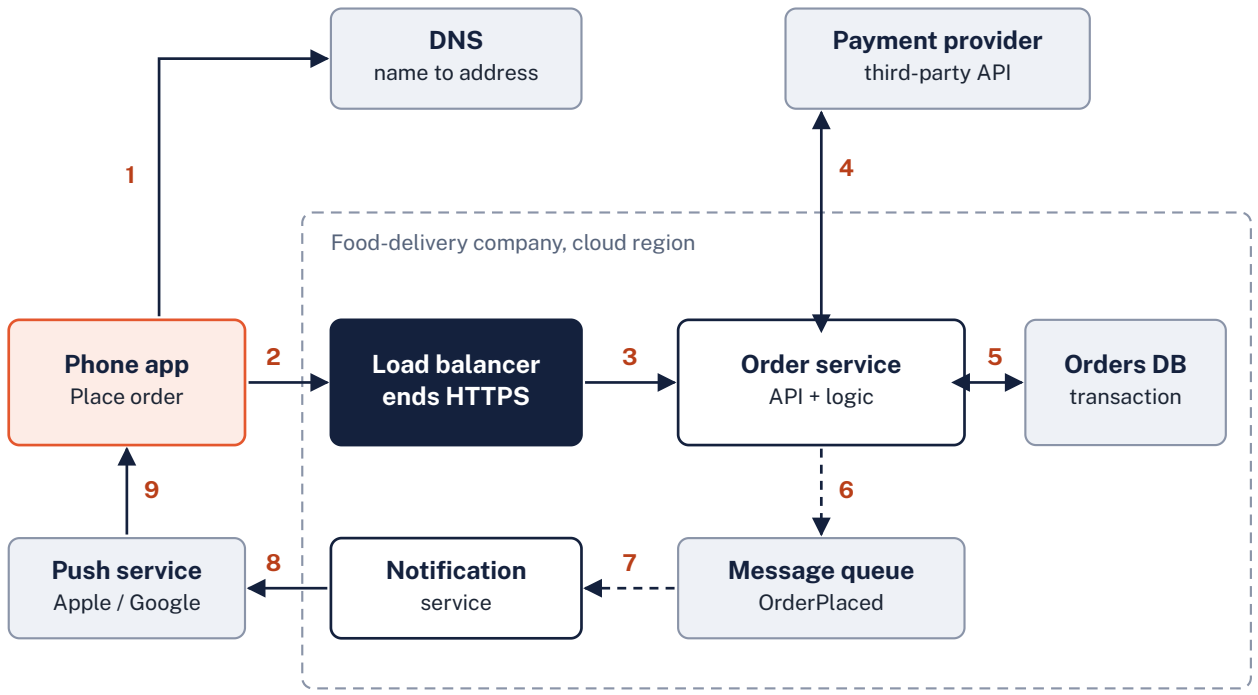
None of these is right for every team, and many organisations mix them: Scrum for product teams, Kanban for support and operations, a more sequential plan around a fixed regulatory deadline. Calling a process "agile" does not make it so; what matters is whether working software reaches real users often enough to learn from them.

DevOps: one team's responsibility from code to customer

Traditionally, developers wrote software and a separate operations team ran it, with conflicting incentives: developers were rewarded for change, operations for stability. DevOps is the movement, named at the first DevOpsDays conference in Ghent in 2009, that brings the two together. It is primarily a culture of shared responsibility ("you build it, you run it"), supported by practices such as CI/CD, infrastructure as code, monitoring and blameless review of incidents. It is not a job title or a tool, although both are sold under the name. The DORA research programme is best known for measuring delivery performance with four key metrics: how often a team deploys, how long a change takes to reach production, how often changes fail, and how quickly service is restored after a failure.

Worked example: the journey of one tap

Everything in this chapter comes together in a single moment. A customer has filled a basket in a food-delivery app and taps **Place order**. Within a second or two the screen says "Order confirmed", and twenty minutes later the phone buzzes: "Your rider is on the way". Here is what happens, step by step, with the concept behind each step and where it was introduced.



One tap crosses the phone, the internet, the company's cloud region and two outside providers; the numbers match the steps below.

Step by step

1. **0. The app builds the request**
   The app is software on the phone (Section 0.1). It gathers the basket from its in-memory data structures (Section 0.2) and turns them into a JSON body: restaurant, items, delivery address, and a unique order key generated on the phone.

2. **1. DNS**
   The app needs the address of `api.fooddelivery.example`. The phone asks a DNS resolver, which returns an IP address, often from a cache (Section 0.4).

3. **2. HTTPS**
   The app opens a TCP connection and a TLS session, checking the server's certificate, then sends `POST /v1/orders` with an access token proving which customer is signed in (Sections 0.4 and 0.6).

4. **3. Load balancer**
   The request reaches a load balancer in the company's cloud region. It decrypts HTTPS and forwards the request to one of several healthy copies of the order service, running in containers across availability zones (Section 0.7).

5

4. API and payment

The order service is reached through its REST API. Its code, organised into modules (Section 0.3), validates the basket, then calls the payment provider's web API to authorise the card, waiting synchronously with a timeout, because the customer must not be told "confirmed" until the payment is approved (Section 0.6). It sends the order key as an idempotency key, so that if the call is retried the card is charged only once.

6

5. Database transaction

In one ACID transaction the service inserts the order and its items and records the payment reference. Either all of it is saved or none of it is; once committed it survives a crash (Section 0.5). An index on the customer's ID will make their order history fast to load.

7

6. Queue

The service puts an `OrderPlaced` message on a message queue and immediately replies `201 Created`. The app shows "Order confirmed". Everything else happens asynchronously (Section 0.6).

8

7. Other services react

The restaurant's tablet service, the rider-dispatch service and the notification service each read the message when ready. If one is briefly down, its messages wait in the queue.

9

8. Push notification

The notification service reacts to `OrderPlaced`, and later to a similar message when a rider is assigned, by calling the push service run by Apple or Google, another third-party API, which delivers the notification to the customer's phone.

10

9. And behind it all

Every piece of code on this path was written from a user story, reviewed in a pull request, tested at several levels and released through a CI/CD pipeline, via staging, to production (this section).

WORKED EXAMPLE

Worked example: what the journey teaches

Ask "what if?" at each step and the architectural questions appear.

What if the payment call times out?	Did the charge happen or not? The idempotency key lets the order service safely ask again without charging twice.
What if the database commit fails after payment?	Payment and database are separate systems, so no single ACID transaction covers both. The design needs a way to refund or retry, a problem Chapter 6 solves with sagas.
What if the notification service is down?	Nothing visible breaks: orders are still taken, and messages wait in the queue. That is the reason for making this step asynchronous.
What if dinner-time traffic triples?	The load balancer spreads requests over more copies of the order service, added by autoscaling; the database is now the likely bottleneck.
What if a release breaks checkout?	Tests in the pipeline should catch it; if not, monitoring raises an alert and the team rolls back to the previous version.

None of these questions is about how to write a function. They are about how the parts are arranged and what each part promises the others.

Why this matters for architecture

The journey of one tap uses a dozen components, three organisations and both synchronous and asynchronous communication, and it still had to be designed: someone decided that payment is synchronous and notifications are not, that orders live in a relational database, that the service runs in containers across two zones, that the payment provider is isolated behind one module. Deciding how these parts are arranged, which qualities each arrangement achieves and what it costs, and recording why, is software architecture. The way software is built matters to architecture too: an architecture that cannot be tested in pieces, or released without coordinating ten teams, slows every change. Chapter 1 begins with what software architecture is.

CHECK YOUR UNDERSTANDING

▼ What is the difference between continuous delivery and continuous deployment?

With continuous delivery every passing change can be released at the press of a button, but a person decides when. With continuous deployment every passing change is released automatically.

▼ Why does the testing pyramid put unit tests at the base?

They are fast, cheap and reliable, so a team can run thousands on every change and get feedback in minutes. Slower, more fragile end-to-end tests are kept for the few journeys that need them.

▼ In the journey of one tap, why is the payment call synchronous but the notification asynchronous?

The app must not say "confirmed" until the payment is approved, so it waits for that answer. The notification can be sent a little later, so it goes through a queue and its failure cannot block ordering.

▼ What are the two main benefits of a pull request beyond catching bugs?

It spreads knowledge, because at least one other person understands each change, and it keeps the code's design consistent, because reviewers can question structural choices.

SOURCES

- Martin Fowler: Continuous Integration: https://martinfowler.com/articles/continuousIntegration.html
- Martin Fowler: Continuous Delivery: https://martinfowler.com/bliki/ContinuousDelivery.html
- Ham Vocke: The Practical Test Pyramid (martinfowler.com): https://martinfowler.com/articles/practical-test-pyramid.html
- The 2020 Scrum Guide: https://scrumguides.org/scrum-guide.html
- Pro Git (Scott Chacon and Ben Straub), free online book: https://git-scm.com/book/en/v2

Further reading

- *Code: The Hidden Language of Computer Hardware and Software (2nd edition)*, Charles Petzold (2022), Microsoft Press. Builds a computer up from switches and codes, in readable steps; the best non-technical explanation of how hardware runs software.
  https://www.microsoftpressstore.com/store/code-the-hidden-language-of-computer-hardware-and-software-9780137909100
- *Grokking Algorithms (2nd edition)*, Aditya Y. Bhargava (2024), Manning. An illustrated, gentle introduction to data structures, algorithms and Big-O notation, ideal after the data and algorithms section.
  https://www.manning.com/books/grokking-algorithms-second-edition
- *The Python Tutorial*, Python Software Foundation, docs.python.org. The official, free introduction to the language used in this chapter's code examples, including functions, modules and classes.
  https://docs.python.org/3/tutorial/index.html
- *Learn web development*, MDN Web Docs contributors, Mozilla. Free, carefully maintained guides to how the web works, HTTP, HTML and JavaScript, written for beginners.
  https://developer.mozilla.org/en-US/docs/Learn_web_development
- *Computer Networking: A Top-Down Approach (9th edition)*, James F. Kurose and Keith W. Ross (2025), Pearson. The standard textbook on networking, starting from the web and working down to packets; read selectively once the basics are clear.
- *Pro Git (2nd edition)*, Scott Chacon and Ben Straub (2014), Apress (free online at git-scm.com). The free, official book on Git: commits, branches, merges and collaboration, as introduced in the last section.
  https://git-scm.com/book/en/v2
- *The NIST Definition of Cloud Computing (SP 800-145)*, Peter Mell and Timothy Grance (2011), National Institute of Standards and Technology. A short, widely cited definition of cloud computing and its service and deployment models.
  https://csrc.nist.gov/pubs/sp/800/145/final
- *Designing Data-Intensive Applications (2nd edition)*, Martin Kleppmann and Chris Riccomini (2026), O'Reilly Media. The best next step once this chapter is comfortable: how databases, networks and distributed systems really behave, and why it matters.

Glossary

Access token
A short-lived credential, issued after a user or application signs in, that is sent with each API request to prove what the caller may do.

ACID
Atomicity, consistency, isolation, durability: the guarantees a database transaction provides so that data stays correct despite crashes and simultaneous users.

Agile
A family of approaches to software development, set out in the 2001 Agile Manifesto, that deliver working software in short cycles and adapt plans based on feedback.

Algorithm
A precise, step-by-step method for solving a problem, such as sorting a list or finding the shortest route, that always finishes.

API (application programming interface)
A defined set of operations, inputs and outputs through which one piece of software uses another, without needing to know how it works inside.

API contract
The documented promise an API makes: which requests it accepts, what each returns, which errors can occur and what will not change without notice.

API key
A long secret string that identifies the calling application to an API. Anyone holding the key can act as that application, so it must be kept secret.

API versioning
Publishing changes to an API under a new version identifier, such as /v2/, while keeping the old version working for a period so that clients can move over in their own time.

Asynchronous messaging
Communication in which the sender puts a message somewhere for the receiver to process later, and carries on without waiting for a reply.

Availability zone
One of several physically separate groups of data centres within a cloud region, each with independent power, cooling and networking, close enough for fast connections.

Backup
A copy of data kept separately from the original so that it can be restored after loss, corruption or deletion.

Bandwidth
How much data a connection can carry per second, such as 100 megabits per second. High bandwidth does not reduce latency.

Big-O notation
A way of describing how the work an algorithm does grows as the amount of data grows, for example O(n) for linear growth or O(1) for constant time.

Bit
A binary digit, 0 or 1: the smallest unit of information a computer stores.

Branch
A separate line of development in version control, used to work on a change without affecting the main line until it is ready to merge.

Breaking change
A change to an API that makes existing, correct client code stop working, such as renaming or removing a field.

Bug
A mistake in software that makes it behave differently from what was intended, from a crash to a silently wrong answer.

Byte
A group of eight bits, able to hold 256 different values. Sizes of files and memory are counted in bytes (kilobytes, megabytes, gigabytes).

Bytecode
Compact instructions for a virtual machine rather than a real CPU. Java and Python compile source code to bytecode, which their runtimes then execute on any supported machine.

Cache
A copy of data kept somewhere faster to reach, so repeated requests avoid slow work. It trades extra memory, and the risk of out-of-date data, for speed.

CI/CD pipeline
The automated sequence of steps (build, test, deploy to staging, deploy to production) that every change passes through on its way to users.

Class and object
A class is a template that bundles data with the functions (methods) that work on it; an object is one instance made from that template, such as one customer's account.

Client-server model
An arrangement in which clients (such as browsers or mobile apps) send requests and servers, which hold shared data and logic, respond.

Cloud computing
Renting computing resources (servers, storage, databases and other services) over the internet, on demand, and paying for what you use.

Code review
Examination of a proposed code change by colleagues before it is merged, to find defects, share knowledge and keep the design consistent.

Cohesion
How closely the contents of one part of a system belong together. High cohesion means a module does one clear job.

Commit
A saved snapshot of changes in version control, with an author, a timestamp and a message explaining the change.

Compiler
A program that translates source code into another form, such as machine code or bytecode, before the program runs.

Container
A packaged application with its libraries and settings, run as an isolated process that shares the host's operating-system kernel. Lighter and faster to start than a virtual machine.

Container image
A read-only, versioned package containing an application and everything it needs to run, from which identical containers can be started anywhere.

Content delivery network (CDN)
A network of servers in many locations that keeps copies of content such as images, scripts and video close to users, so that it loads faster and the origin server does less work.

Continuous delivery
Keeping software always in a releasable state, so that any version that passes the automated pipeline can be put into production at the press of a button.

Continuous deployment
Automatically releasing every change that passes the pipeline to production, with no manual approval step.

Continuous integration (CI)
The practice of merging every developer's changes into the shared main line frequently, at least daily, with an automated build and tests verifying each change.

Coupling
How much one part of a system depends on the details of another. Loose coupling means a change in one part rarely forces changes elsewhere.

CPU (processor)
The central processing unit: the chip that carries out a program's instructions, one simple step at a time, billions of times a second. Modern CPUs have several cores that work in parallel.

Data centre
A building designed to house many servers with reliable power, cooling, physical security and network connections.

Data structure
A way of organising data in memory, such as a list, dictionary, set, queue, stack, tree or graph, chosen to make the operations you need fast.

Data type
The kind of value a piece of data is, such as an integer, a decimal number, text or true/false, which decides what operations make sense on it.

Database
An organised collection of data stored so that it can be searched, updated and kept consistent, usually managed by a database management system.

Database index
An extra data structure, usually a B-tree, that lets the database find rows by a column's value without reading the whole table. It speeds up reads but slows writes and uses space.

Database management system (DBMS)
The software that stores a database and controls access to it: it answers queries, enforces rules, handles many users at once and recovers after crashes. PostgreSQL and MySQL are examples.

Dependency
Any piece of software that your code needs in order to work, such as a library, a framework or another service. If it changes or breaks, your code may break too.

DevOps
A culture and set of practices in which development and operations work as one, sharing responsibility for building, releasing and running software, supported by automation and measurement.

Dictionary (map)
A data structure that stores values under unique keys, so a value can be found directly by its key, usually in roughly constant time. Also called a map or hash table.

DNS
Domain Name System: the internet's directory service, which turns names such as www.example.com into IP addresses.

Docker
A widely used set of tools, released in 2013, for building container images and running containers. Its image format became an open standard.

Document database
A NoSQL database that stores each record as a self-contained document, usually JSON-like, which can hold nested data and need not share a fixed schema.

Encapsulation
Keeping an object's or module's internal data and details private, so other code can use it only through the operations it chooses to offer.

End-to-end test
An automated test that drives the whole system as a user would, for example by tapping through an app, to check a complete journey.

Environment
A separate, complete copy of the system and its settings used for one purpose, such as development, testing, staging or production.

File
A named sequence of bytes stored on a disk and managed by the operating system, such as a photo, a spreadsheet or a log.

Foreign key
A column in one table that holds the primary key of a row in another table, linking the two. The database can refuse values that point to nothing.

Framework
Reusable code that provides the overall structure of an application and calls your code at the right moments, such as Django or Spring.

Function
A named, reusable piece of code that takes inputs (parameters), does some work and usually returns an output.

Git
The most widely used version control system, created by Linus Torvalds in 2005. Every developer has a full copy of the project's history.

Graph database
A database that stores items as nodes and the relationships between them as edges, so that following chains of connections is fast.

Hardware
The physical parts of a computer that you could touch: the processor, memory, storage, network card, screen and so on.

HTTP
Hypertext Transfer Protocol: the request-and-response language of the web, in which a client sends a method such as GET to a URL and a server replies with a status code and content.

HTTP status code
A three-digit number in every HTTP response saying how the request went: 2xx success, 3xx redirection, 4xx a problem with the request, 5xx a problem on the server.

Hypervisor
Software that creates and runs virtual machines, sharing one physical computer's processors, memory and devices among them while keeping them isolated.

IaaS (infrastructure as a service)
A cloud model in which you rent virtual machines, storage and networks, and manage the operating system and everything above it yourself.

Index
An extra data structure, kept alongside data, that lets a system find items by a chosen field without scanning everything, like the index at the back of a book.

Integration test
An automated test that checks that several parts work correctly together, such as code with a real database or another service's API.

Interface
The agreed set of operations a piece of software offers to others, with their inputs and outputs: a contract that hides how the work is done.

Interpreter
A program that reads code and carries out its instructions as it goes, rather than translating the whole program in advance.

Inversion of control
The arrangement in which a framework, rather than your own code, controls the flow of the program and calls your code when needed.

IP address
The numeric address that identifies a device on a network so packets can be delivered to it, such as 192.0.2.10 (IPv4) or 2001:db8::1 (IPv6).

Join
An SQL operation that combines rows from two or more tables by matching related columns, such as orders with the customers who placed them.

JSON (JavaScript Object Notation)
A text format for structured data made of objects (name-value pairs in braces), arrays (lists in brackets), strings, numbers, true, false and null. Standardised in RFC 8259.

Kanban
A way of managing work as a continuous flow on a visual board, limiting how many items may be in progress at once.

Key-value store
A database that stores values under unique keys and retrieves them only by key, like a very large, very fast dictionary.

Latency
The delay before data arrives or a response comes back, usually measured in milliseconds. Distance, network hops and slow processing all add to it.

Library
Reusable code written by someone else that your code calls when it needs it, such as a library for dates or for sending email.

Linux
A free, open-source operating system kernel, used with other software as the operating system on most servers, cloud machines and containers, as well as Android phones.

Load balancer
A component that receives incoming requests and spreads them across several servers, skipping any that are unhealthy.

Machine code
Instructions encoded as numbers that a particular family of CPU can execute directly. Machine code for one family (for example x86-64) does not run on another (for example Arm).

Managed database
A database service run by a cloud provider, which handles installation, patching, backups and failover while the customer manages the schema, queries and access.

Managed service
A cloud service, such as a database, queue or storage service, that the provider installs, operates, patches and scales, so that customers use it without running it.

Message queue
Software that stores messages from senders until receivers are ready to process them, so the two need not be running or fast at the same time.

Module
A unit of code, often a single file, that groups related functions and data under one name and decides what it makes available to other code. A package is a group of modules.

NoSQL
An umbrella term for databases that do not use the relational table model, such as document, key-value, wide-column and graph databases.

Operating system
The software that manages a computer's hardware and shares it between programs: CPU time, memory, files, network and security. Examples are Linux, Windows, macOS, Android and iOS.

PaaS (platform as a service)
A cloud model in which you supply your application code or container and the provider runs, patches and scales the platform underneath it.

Package manager
A tool such as pip (Python) or npm (JavaScript) that downloads libraries and the libraries they depend on, at the versions a project specifies.

Packet
A small chunk of data sent across a network with addressing information attached. Large messages are split into many packets and reassembled at the other end.

Pay-as-you-go
A pricing model in which you are charged for resources only while you use them, for example per second of server time or per gigabyte stored.

Port
A number from 0 to 65535 that identifies which program on a device should receive network traffic, such as 443 for HTTPS.

Primary key
A column, or set of columns, whose value uniquely identifies each row in a table, such as customer_id.

Process
A running instance of a program, with its own private memory and share of CPU time, created and managed by the operating system.

Production
The live environment that real users and real data depend on.

Program
A list of precise instructions that a computer follows to do a job. A running program is called a process.

Pull request
A request to merge a branch into the main line, on a service such as GitHub, where colleagues review the change and automated checks run before it is accepted.

RAM (memory)
Random-access memory: fast, temporary storage holding the programs and data in use right now. Its contents are lost when the power goes off or the program ends.

Rate limit
A cap on how many requests a client may make in a period, such as 100 per minute. Requests above it are refused, typically with HTTP status 429.

Region
A geographic area, such as Western Europe or East US, where a cloud provider operates a separate group of data centres. Regions are isolated from each other.

Relational database
A database that stores data in tables of rows and columns and links tables through keys. It is queried with SQL.

Requirement
A statement of something a system must do, or a quality it must have, agreed with the people who need it.

REST (Representational State Transfer)
An architectural style for web APIs, described by Roy Fielding in 2000, in which clients act on resources identified by URLs using standard HTTP methods.

SaaS (software as a service)
Complete software used over the internet, usually through a browser or app, with the provider running everything.

Scaling out (horizontal scaling)
Handling more load by adding more machines and spreading the work across them, usually behind a load balancer.

Scaling up (vertical scaling)
Handling more load by moving to a bigger machine with more processors or memory.

Schema
The declared structure of a database: its tables, columns, data types, keys and rules.

Scrum
An agile framework in which a small team works in fixed cycles called sprints, of one month or less, with defined events, accountabilities and artefacts.

SDK (software development kit)
A library, usually provided by an API's owner, that wraps the API's HTTP calls in ordinary functions for a particular programming language.

Server
A computer, or a program running on one, that waits for requests from other computers (clients) and answers them.

Serverless and FaaS (function as a service)
A cloud model in which you upload individual functions that run only when an event triggers them; the provider handles all servers and you pay per execution.

Software
Programs together with the data and configuration they need; the instructions that tell hardware what to do.

Source code
The human-readable text of a program, written in a programming language such as Python or Java, which is then compiled or interpreted so a computer can run it.

SQL (Structured Query Language)
The standard language for defining, querying and changing data in relational databases. You describe the result you want and the database works out how to get it.

Synchronous call
A request in which the caller waits for the response before continuing, as in an ordinary HTTP API call.

TCP
Transmission Control Protocol: delivers a reliable, ordered stream of data between two programs, resending lost packets. Most web traffic has traditionally used it.

Testing pyramid
A guide, from Mike Cohn, to the proportions of automated tests: many fast unit tests, fewer integration tests and only a few slow end-to-end tests.

Thread
One sequence of instructions being executed inside a process. Threads in the same process share its memory, so they can work at the same time but must coordinate.

TLS and HTTPS

Transport Layer Security encrypts data between two programs and, through certificates, proves the server's identity. HTTPS is HTTP carried over TLS.

Transaction

A group of database operations treated as one unit: either all of them take effect (commit) or none do (rollback).

UDP

User Datagram Protocol: sends individual packets with no guarantee of delivery or order. Faster and simpler than TCP; used for video calls, games and DNS.

Unit test

An automated test that checks one small piece of code, such as a function, in isolation, and runs in milliseconds.

User story

A short description of a feature from the point of view of the person who wants it, often written as: As a [role], I want [capability], so that [benefit].

UTF-8

The dominant way of encoding text as bytes. It can represent every Unicode character, using one byte for basic English letters and up to four bytes for others.

Version control

A system that records every change to a set of files, who made it and why, so that people can work in parallel, review changes and return to any earlier version.

Virtual machine (VM)

A software imitation of a complete computer, with its own operating system, running alongside other VMs on one physical machine under a hypervisor.

Web API

An API offered over a network, usually using HTTP, so that programs on different machines, written in different languages, can work together.

Webhook

An HTTP request that a provider sends to a URL registered by its customer when an event happens, such as a payment succeeding. It reverses the usual direction of an API call.

Wide-column store

A NoSQL database that groups data into rows identified by a key, each with many columns, spread across many machines. Designed for very large write volumes.