INTRODUCTION TO SOFTWARE ENGINEERING · CHAPTER 02

Types of Applications

Different kinds of software face different constraints. This chapter classifies applications and shows how the type of an application drives its architecture, its quality attributes and how it is delivered.

ISO/IEC 25010:2023 | SWEBOK Guide v4.0 | C++20 · HTTP · Mermaid | Case study: ITC Club Hub

AGENDA

What we will cover

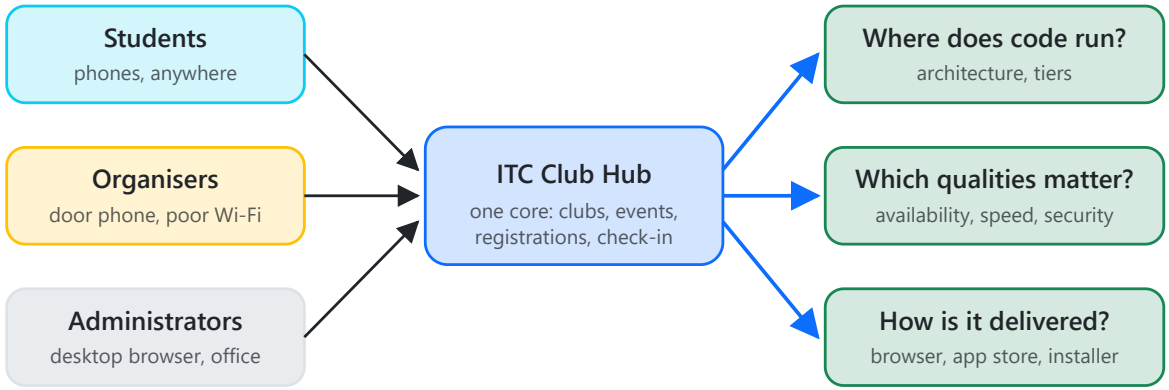
1. System software versus application software
2. Desktop, web (multi-page and single-page) and mobile (native, cross-platform, progressive web app) applications
3. Enterprise systems: ERP, CRM, information systems
4. Embedded, real-time and Internet of Things systems
5. Cloud applications and software as a service
6. Data-intensive and AI-enabled applications, and games
7. Client-server and multi-tier architecture basics
8. Quality attributes by application type: performance, availability, security, usability, portability
9. Platform constraints: browsers, app stores, devices, connectivity
10. Choosing an application type for a problem
11. Chapter Quiz
12. Lab-02

Click any item to jump directly to that topic.

One problem, several kinds of users: why the type of application matters

- In Lab-01 your team wrote the problem statement for **ITC Club Hub** and listed its stakeholders. Before we write requirements (Chapter 05) we must decide *what kind of software* it is.
- Students use phones anywhere on campus; organisers check people in at the door of a room with weak Wi-Fi; administrators prepare reports on a desktop.
- The **type of application** decides three things early: where the code runs (architecture), which qualities matter most (availability, speed, security…), and how the software reaches users (browser, app store, installer).
- Changing the type late is expensive: it is close to a rewrite of the user-facing part.

💡 SWEBOK Guide v4.0 treats this under *Software Architecture* and *Software Quality*: architecture decisions are driven by quality requirements, not only by features.



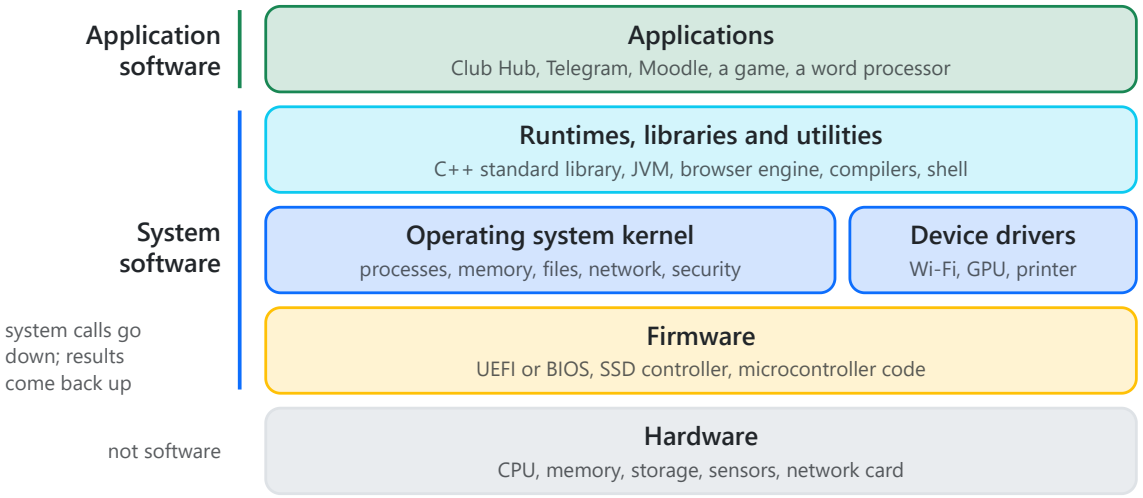
the application type answers all three questions

TOPIC 1

System software versus application software: the software stack

- **System software** runs and manages the computer itself and offers services to other programs: firmware, the operating system kernel, device drivers, runtimes, compilers and utilities.
- **Application software** helps a person or an organisation do a task: chat, learning, banking, games, Club Hub.
- Each layer only talks to the layer below through a defined interface: an application calls the OS through *system calls* (open a file, send a packet); the OS talks to hardware through drivers.
- System software must be fast, small and extremely reliable, because every application above it depends on it. That is why it is mostly written in C, C++ and, more and more, Rust.

Firmware is software stored in a device's non-volatile memory (UEFI in a PC, the code in a washing machine or an ABS unit). It is the first code to run when power is applied.



Worked example: where C++ is used, and where other languages win

Domain	Typical languages	Why that choice
Embedded and firmware	C, C++, Rust	No OS or a tiny one, kilobytes of RAM, direct access to hardware registers, predictable timing.
Games and game engines	C++ (Unreal Engine), C# for Unity scripts	A frame must be ready every 16.7 ms; no garbage-collector pauses; control over memory layout.
Systems software	C, C++, Rust	Browsers, databases, compilers and OS components need speed and control of every byte.
Performance-critical libraries	C++ core, Python bindings	Machine-learning and image libraries run C++ underneath while users write Python on top.
Web front ends	JavaScript, TypeScript	The browser runs JavaScript; C++ only through WebAssembly for special cases.
Enterprise back ends	Java, C#, Go, Python	Big frameworks, fast development, memory safety through garbage collection.
Mobile apps	Kotlin, Swift, Dart	The platform SDKs and UI toolkits are built for these languages.
Data analysis and AI	Python, SQL, R	Productivity and libraries matter more than raw speed of your own code.

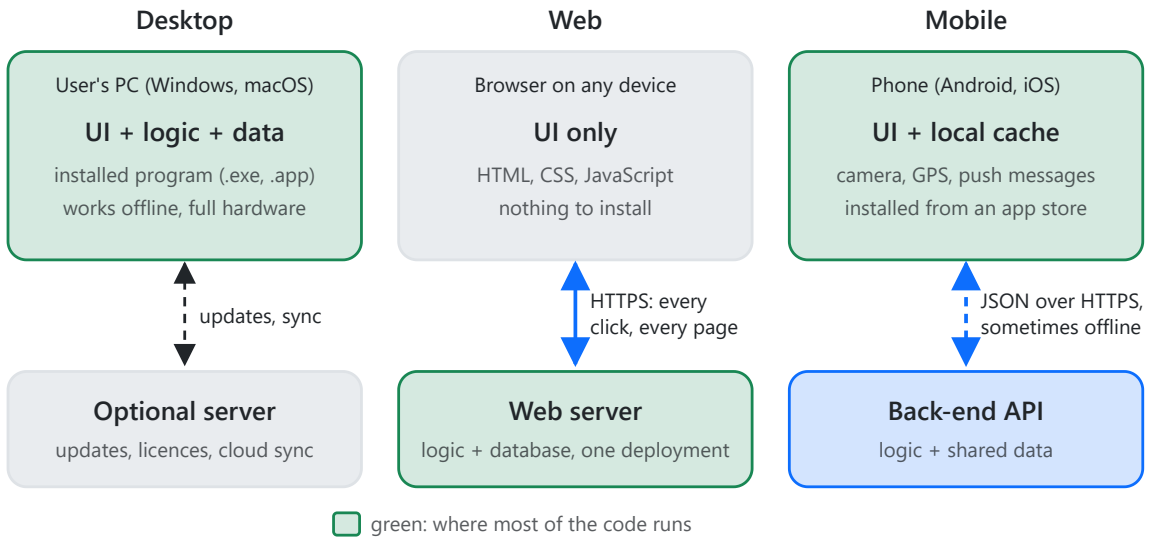
- C++ is chosen when you need **zero-overhead abstractions**, **predictable timing** and **direct hardware access**.
- The price: memory safety is the programmer's job (use RAII, `std::vector`, `std::string`, smart pointers, no raw `new`/`delete`), builds are slower, and there are fewer ready-made web and mobile frameworks.
- Most real products mix languages: a C++ engine with a scripting language on top, or a Java back end calling a C++ library.

💡 **Club Hub in this course:** we implement the *core* (domain model and business rules) in C++20 with a command-line front end, and *model* the web and mobile front ends. A production team would probably pick a web stack for the front ends; the Software Engineering course builds one in Java.

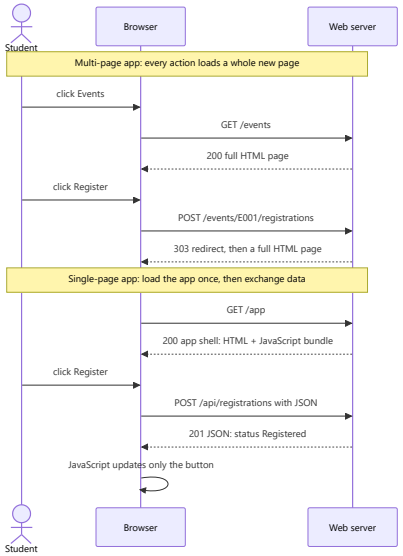
⚠ "Which language is best?" has no answer without an application type and its quality attributes.

Desktop, web and mobile applications: where does the code run?

- **Desktop application**: installed on the user's computer; interface, logic and often the data all run locally. Works offline and uses the full hardware. Updates need an installer or an update service. Examples: VS Code, a spreadsheet, most PC games.
- **Web application**: runs in a browser; nothing to install. The server holds the logic and the shared data, so one deployment updates every user at once. Needs a network. Examples: Moodle, web mail.
- **Mobile application**: installed on a phone, usually from an app store. Uses the camera, GPS and push notifications, keeps a local cache, and talks to a back-end API over a network that comes and goes. Examples: Telegram, a banking app.
- Many products are all three at once, sharing one back end: Telegram has desktop, web and mobile clients.



Web applications: multi-page (MPA) versus single-page (SPA)



	MPA	SPA
Server returns	HTML pages	JSON data (after the first load)
First load	fast, small	slower: JavaScript bundle
Next clicks	full reload	instant, partial update
Search engines	easy to index	needs server-side rendering
Offline	no	possible (see PWA)
Complexity	mostly on the server	much logic in the browser
Examples	news sites, Moodle pages	web mail, Google Maps

💡 An SPA and a mobile app can call the **same JSON API**. That is one reason teams separate the back end from every front end.

Worked example: the same idea as a console program and as an HTTP exchange

Console program: one process, one user, output is text for a human.

```cpp
// list_events.cpp: the whole application runs in one process
#include <iostream>
#include <string>
#include <vector>

struct EventRow {
    std::string title;
    std::string start;
    int seatsLeft;
};

int main() {
    const std::vector<EventRow> events{
        {"Robotics Workshop", "2026-10-14 14:00", 12},
        {"Debate Night", "2026-10-16 18:00", 0},
    };
    for (const auto& e : events) {
        std::cout << e.start << "  " << e.title << "  ("
                  << e.seatsLeft << " seats left)\n";
    }
}
```

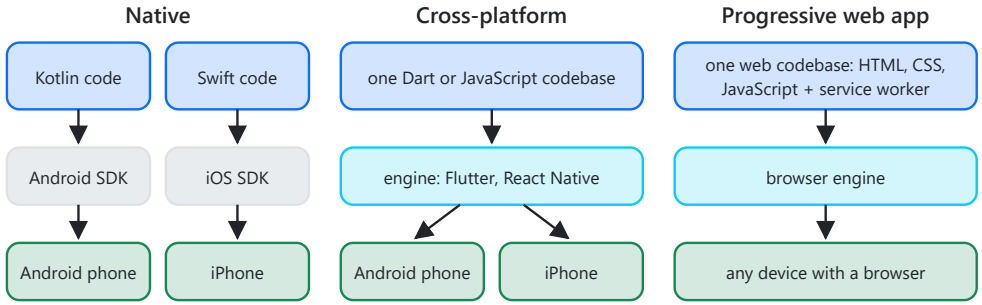
```
$ ./list_events
2026-10-14 14:00  Robotics Workshop  (12 seats left)
2026-10-16 18:00  Debate Night  (0 seats left)
```

Web or mobile: a client sends a request over the network; the server answers with data.

```
GET /api/events?from=2026-10-01 HTTP/1.1
Host: clubhub.example.edu
Accept: application/json
User-Agent: Mozilla/5.0 (Linux; Android 15; Mobile)

HTTP/1.1 200 OK
Content-Type: application/json
Cache-Control: max-age=60

[
  {"id": "E001", "title": "Robotics Workshop",
   "start": "2026-10-14T14:00+07:00", "seatsLeft": 12},
  {"id": "E002", "title": "Debate Night",
   "start": "2026-10-16T18:00+07:00", "seatsLeft": 0}
]
```

- The loop moved to the **server**; the output became **data** (JSON), and each client decides how to display it.
- The network adds new concerns: latency, failures and retries, many users at once, HTTPS, authentication, versioning of the API.
- Status codes carry meaning: `200` OK, `201` created, `400` bad input, `404` not found, `503` server unavailable.

Mobile applications: native, cross-platform or progressive web app



cost and reach improve to the right; device access and platform feel improve to the left

	Native	Cross-platform	PWA
Languages	Kotlin (Android), Swift (iOS)	Dart (Flutter), JS/TS (React Native), C# (.NET MAUI)	HTML, CSS, JS/TS
Distribution	app stores, review	app stores, review	a URL, "Add to home screen"
Device access	everything	most, via plugins	camera, location, Web Push; less on iOS
Offline	local database	local database	service worker cache + IndexedDB
Updates	store release, users update	store release	instant on next load
Team skills	two specialists	one framework	web developers

💡 A **progressive web app** is a web app with a *manifest* (name, icon: installable) and a *service worker* (a script that intercepts network requests, so the app can answer from a cache when offline). It is served over HTTPS.

Worked example: Club Hub as a native app, a PWA or a responsive web app

Criterion (weight)	Native apps (Android + iOS)	PWA	Responsive web app
Cost for a 5-person team, one semester (3)	3 front ends: 2 apps + web admin. **1**	1 codebase + offline work. **4**	1 codebase. **5**
Reach: every student, no install (3)	only those who install. **3**	a link in the club chat, installable. **5**	a link, any browser. **5**
Offline check-in at the door (2)	local database. **5**	service worker + IndexedDB queue. **4**	needs a connection. **1**
Event reminders (1)	push notifications. **5**	Web Push; on iOS only when installed. **3**	e-mail only. **2**
QR scanning with the camera (1)	full camera API. **5**	camera in the browser. **4**	camera in the browser, online only. **3**
Weighted total (max 50)	3 + 9 + 10 + 5 + 5 = **32**	12 + 15 + 8 + 3 + 4 = **42**	15 + 15 + 2 + 2 + 3 = **37**

Scores 1 (poor) to 5 (excellent), multiplied by the weight. Weights come from the stakeholders: here reach and cost matter most.

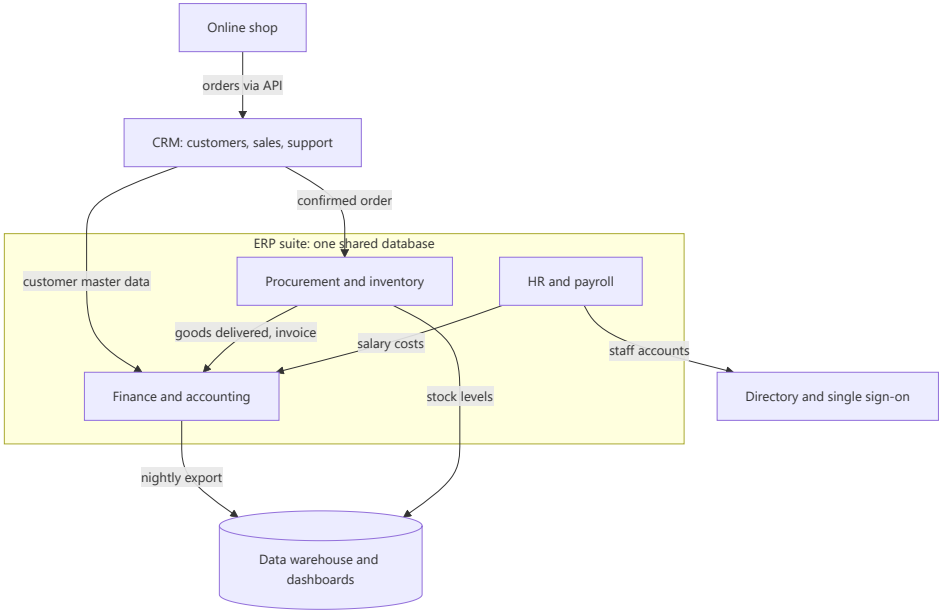
- The PWA wins *with these weights*. If the club office required reminders that always arrive on iPhones, native would gain; if the venues had good Wi-Fi, the plain responsive web app would be enough.
- One application does not have to be one type: administrators can use the **same web app** on a desktop browser with a wider layout.
- Whatever the front end, the **core rules** (capacity, no double registration, 24-hour cancellation) live in one place behind an API. In this course that core is your C++ code.

⚠ A weighted matrix supports a decision; it does not make it. Always check the winner against the one criterion that would be a deal-breaker (here: check-in must work without Wi-Fi).

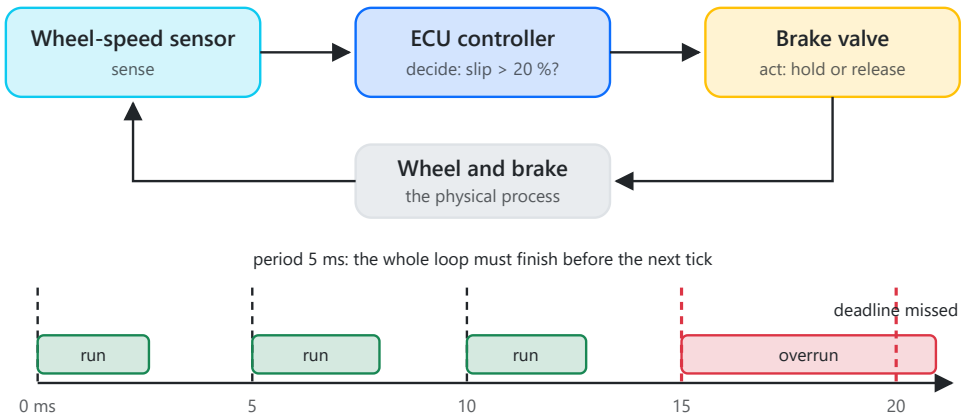
Enterprise systems: ERP, CRM and information systems

- An **information system** is people, processes, data and software that together support an organisation, for example a university's student information system.
- **ERP** (enterprise resource planning) is a suite of integrated modules (finance, HR and payroll, procurement, inventory) that share *one database*, so a purchase immediately appears in the accounts. Examples: SAP S/4HANA, Odoo.
- **CRM** (customer relationship management) tracks leads, customers, sales and support tickets. Example: Salesforce.
- Typical traits: hundreds of users with roles and permissions, audit trails, a life of 10 to 20 years, and more *integration* and *configuration* work than new code.
- Dominant qualities: security, data integrity, interoperability, maintainability.

🏢 Club Hub is small, but it already has the enterprise shape: it reads the **ITC student directory** and sends mail through an **e-mail service**. You will draw these in the Lab-02 context diagram.



Embedded and real-time systems: sense, decide, act before the deadline



ⓘ In a **real-time system** correctness depends on the result *and* on the time it is delivered. A braking decision that arrives late is a wrong decision.

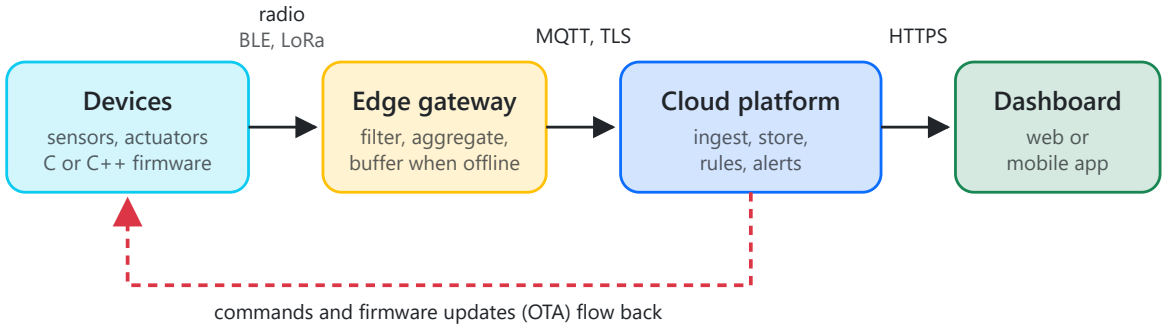
```cpp
#include <chrono>
#include <thread>

using namespace std::chrono_literals;
using Clock = std::chrono::steady_clock;

// Sense, decide, act: once every 5 ms, forever
void controlLoop(const WheelSensor& sensor, BrakeValve& valve,
                 FaultLog& log) {
    constexpr auto period = 5ms;
    auto deadline = Clock::now() + period;
    for (;;) {
        const double slip = sensor.slipRatio();
        // wheel about to lock: release pressure briefly
        valve.set(slip > 0.20 ? Valve::Release : Valve::Hold);
        if (Clock::now() > deadline) {
            // a late answer is a wrong answer
            log.deadlineMissed();
        }
        std::this_thread::sleep_until(deadline);
        deadline += period;
    }
}
```

- `steady_clock` never jumps (unlike the wall clock), so it is the right clock for periods and deadlines.
- An **embedded system** is a computer inside a larger device with one dedicated job, little memory and often no screen. Real ECUs run this loop on a real-time OS or bare metal, not on `std::thread`; the idea is the same.
- No heap allocation, no exceptions and no unbounded loops inside the period: timing must be predictable.

Hard, firm and soft real-time, and the Internet of Things

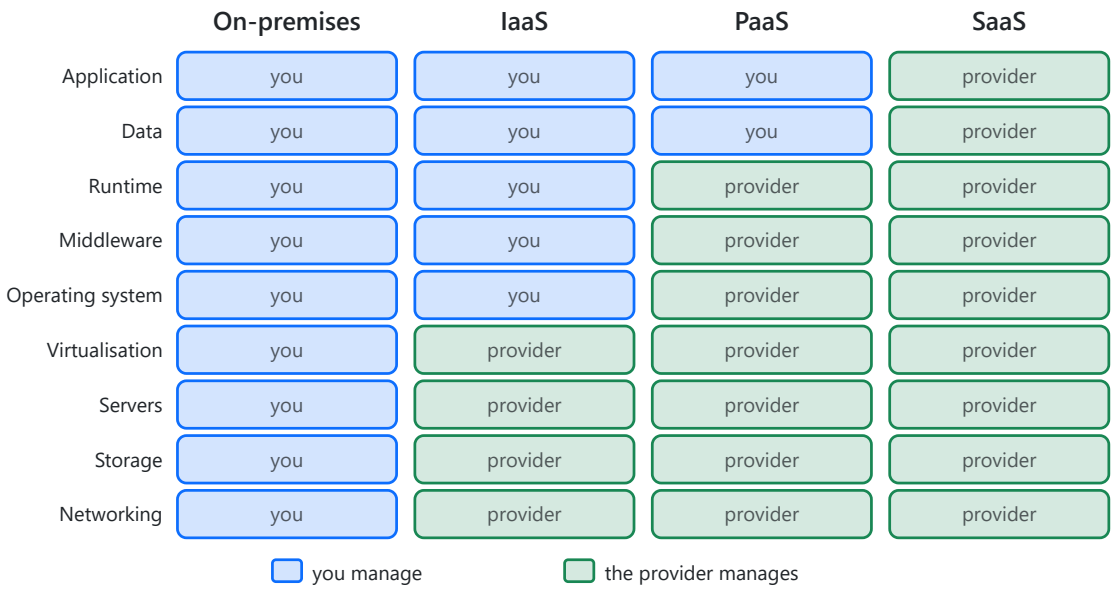
Kind	A missed deadline means	Example
Hard real-time	system failure, possibly harm	ABS brakes, airbag, pacemaker
Firm real-time	the late result is useless, but no harm	a sensor sample, a video frame in a live call
Soft real-time	quality drops gradually	music streaming buffer, a game at 50 instead of 60 fps

- The **Internet of Things** (IoT) connects embedded devices to the network so that their data reaches cloud services and apps: smart meters, room sensors, parking sensors.
- New concerns appear: battery life, unreliable radio links, remote firmware updates (OTA, over the air) and security of thousands of devices that nobody patches by hand.
- Dominant qualities: safety, reliability, timing, energy efficiency, security.



one IoT product is really three applications: embedded firmware, a cloud back end and a user app

Cloud applications and software as a service: who manages what?



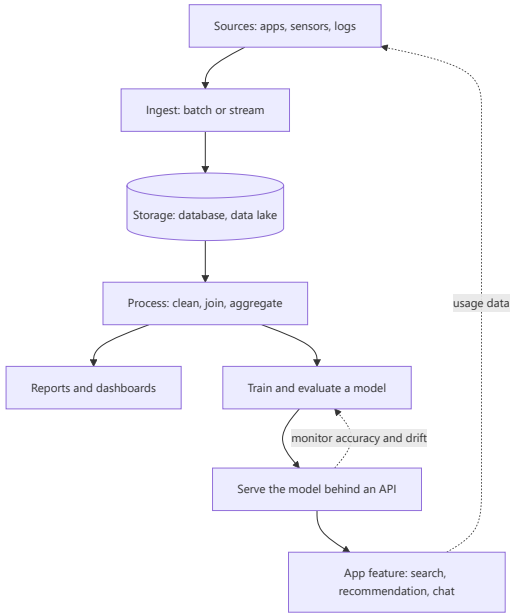
Model	You rent	Examples
IaaS infrastructure	virtual machines, disks, networks	AWS EC2, Azure Virtual Machines, a VPS
PaaS platform	a place to run your code	Azure App Service, Google App Engine, Heroku
SaaS software	a finished application	Gmail, Microsoft 365, Zoom, Salesforce

- A **cloud application** runs on rented, elastic infrastructure: capacity grows and shrinks with demand, and you pay for what you use.
- **SaaS** products are *multi-tenant* (one installation serves many customer organisations, each isolated), paid by subscription and updated continuously.
- Short calculation (illustrative prices): a small VM at 0.02 USD per hour runs 730 hours a month: 0.02 × 730 = **14.60 USD per month**, with no hardware to buy.

⚠ In every model **you** still own your data, user accounts and configuration. A leaked password is never the provider's fault.

TOPIC 6

Data-intensive and AI-enabled applications



- A **data-intensive** application is limited by the amount, speed or variety of its data rather than by CPU: a bank ledger, a video platform's view counts, a university's exam results over 20 years.
- An **AI-enabled** application has behaviour *learned from data* instead of written as rules. Its output is probabilistic, so it needs evaluation data, monitoring after release and a fallback when the model is wrong.
- Dominant qualities: scalability, data quality, privacy, explainability, cost of computing.

A possible Club Hub extension: "recommend events based on the clubs a student attended". It needs attendance history, which is personal data: Chapter 03 asks whether you may use it.

Games: a soft real-time application built around a loop

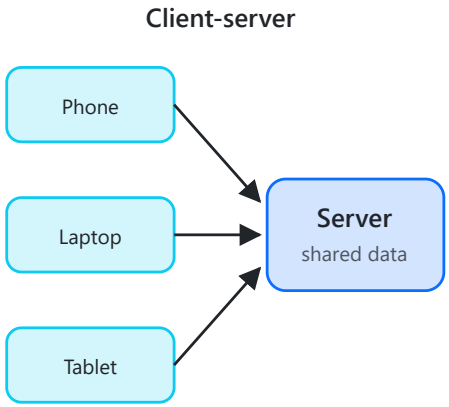
```cpp
#include <chrono>

using namespace std::chrono_literals;
using Clock = std::chrono::steady_clock;

// Classic game loop: fixed-step simulation, render as often as possible
void run(Game& game) {
    constexpr std::chrono::milliseconds step = 16ms;
    auto previous = Clock::now();
    Clock::duration lag{};
    while (game.running()) {
        const auto now = Clock::now();
        lag += now - previous;
        previous = now;
        game.processInput();
        while (lag >= step) {
            // physics and rules advance in 16 ms steps
            game.update(step);
            lag -= step;
        }
        // drawing must also fit in the frame budget
        game.render();
    }
}
```

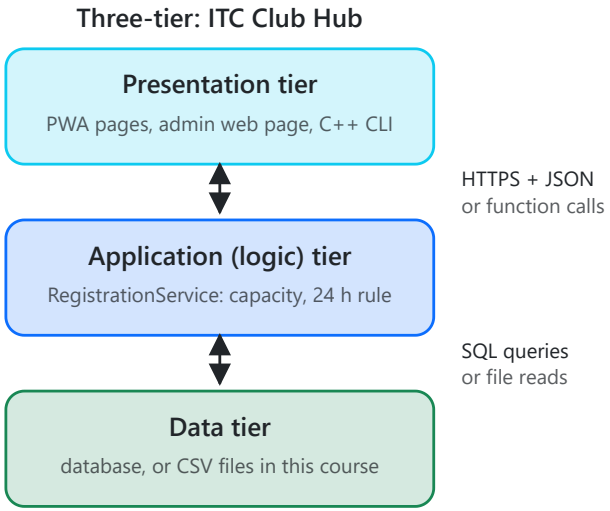
Target	Frame budget = 1000 ms / fps
30 fps (console, older phones)	1000 / 30 = 33.3 ms
60 fps (typical)	1000 / 60 = **16.7 ms**
144 fps (gaming monitor)	1000 / 144 = 6.9 ms

- Input, game rules, physics, audio and drawing must all fit in the budget, every frame. Missing it causes stutter, not a crash: **soft real-time**.
- Engines such as Unreal Engine are written in C++; gameplay is often scripted in a higher-level language on top.
- Dominant qualities: performance, usability (fun), portability across PC, consoles and phones; for online games also availability and cheat-resistant security.

Client-server and multi-tier architecture basics

- **Client-server**: many clients send requests; one shared server answers. Clients start the conversation; the server owns the shared data. Web, mobile and most enterprise systems work this way.
- **Three-tier** splits the server side further:
  - **presentation**: shows data and collects input (browser page, mobile screen, CLI);
  - **application (logic)**: business rules and workflows;
  - **data**: stores and retrieves data (database, files).
- Each tier talks only to its neighbour. You can replace the presentation (add a mobile app) or the data store (CSV to a database) without touching the rules.
- *Tier* usually means a separately deployed part (another machine or process); *layer* means a logical part of the code. A C++ program can have three layers in one process.





TOPIC 7

Three layers in C++: keep the core free of input and output

```cpp
// includes and the Event struct omitted
namespace clubhub {

// ---- data layer: an interface; CSV today, a database later
class EventRepository {
public:
    virtual ~EventRepository() = default;
    virtual std::vector<Event> findAll() const = 0;
};

// ---- logic layer: business rules, no console, no files
class EventService {
public:
    explicit EventService(const EventRepository& repo) : repo_{repo} {}
    std::vector<Event> upcoming(TimePoint now) const {
        std::vector<Event> result;
        for (const auto& e : repo_.findAll()) {
            if (e.start > now) result.push_back(e);
        }
        return result;
    }
private:
    const EventRepository& repo_;
};

} // namespace clubhub

// ---- presentation layer: today a console, later a web or mobile UI
void printUpcoming(const clubhub::EventService& service,
                   clubhub::TimePoint now) {
    for (const auto& e : service.upcoming(now)) {
        std::cout << e.title << '\n';
    }
}
```

- `EventService` never prints and never opens a file. The same class can serve the CLI, an HTTP handler that returns JSON, or a unit test.
- `EventRepository` is an *interface* (a class with pure virtual functions). A CSV-backed version and a database-backed version can be swapped without changing the rules.
- The presentation function only formats what the service returns.
- Folder convention for Club Hub: `include/clubhub/` for the core headers, `src/` for their implementation, `src/cli/` for the console front end.

Test for a clean core: could you delete `src/cli/` and still compile and test every business rule? In Lab-02 Task 5 you build exactly this split.

Quality attributes by application type

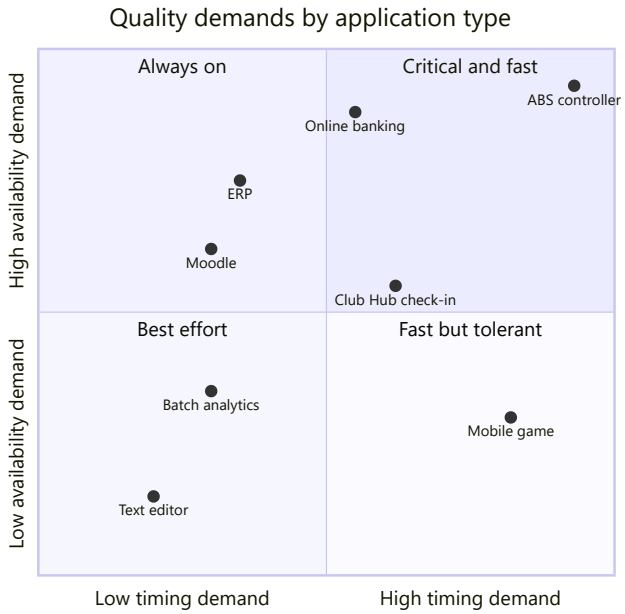
Application type	Performance	Availability	Security	Usability	Portability
Desktop tool	M	L	M	H	M
Public web app	M	H	H	H	H
Mobile app	M	M	H	H	H
Enterprise (ERP, CRM)	M	H	H	M	L
Embedded real-time	H	H	M	L	L
Cloud SaaS	M	H	H	H	M
Game	H	M	M	H	H
AI-enabled	M	M	H	M	L

H = usually decisive, M = matters, L = rarely the driver. Typical values; a specific product can differ (an online bank's mobile app has H availability).

- A **quality attribute** is a measurable property of the system (how fast, how often up, how secure, how easy), as opposed to a function (what it does).
- **Performance**: response time and throughput. **Availability**: share of time the system is usable. **Security**: confidentiality, integrity, who may do what. **Usability**: how easily users reach their goal. **Portability**: how easily it runs on other platforms.
- Qualities trade off: offline caching helps availability but makes security and consistency harder.

ISO/IEC 25010:2023 is the standard quality model. Its 2023 edition names usability *interaction capability*, names portability *flexibility*, and adds *safety*. Chapter 09 uses it for quality assurance.

Two demands side by side: timing and availability



- Placing products on two axes makes a discussion concrete: "is our check-in really as critical as online banking?"
- **Club Hub check-in** sits in the middle: a queue at the door is annoying, not dangerous, but it happens at a known peak moment (the start of an event).
- Moodle needs high availability during exam weeks but little timing precision; a mobile game needs smooth frames but may be offline for maintenance.
- Positions are judgements, not measurements. The next slide turns a judgement into a **testable** statement.

Worked example: quality attribute scenarios for Club Hub

Bass, Clements and Kazman, Software Architecture in Practice

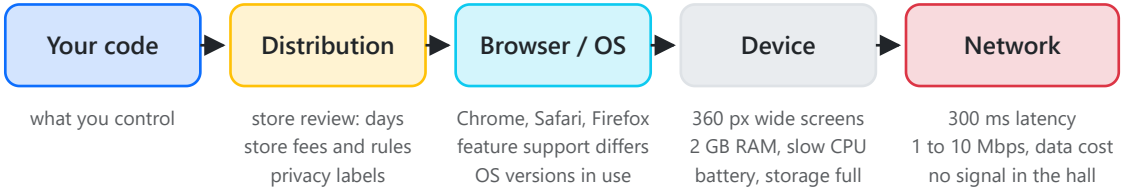
```
QAS-01  Availability of registration
Source       hosting fault (crash, restart of the VM)
Stimulus     the Club Hub API process stops unexpectedly
Environment  semester, Friday 18:00, registration peak
Artifact     Club Hub API and its data store
Response     failure detected, process restarted automatically;
             clients show "try again in a minute";
             no confirmed registration is lost
Measure      service back within 5 minutes; monthly
             availability >= 99.5 % (30 d x 24 h x 0.5 %
             = 3.6 h downtime allowed); 0 lost records
```

```
QAS-02  Performance of check-in at the door
Source       organiser at the entrance
Stimulus     scans a student's QR code to check in
Environment  peak: 200 students arrive in 10 minutes,
             one door, venue Wi-Fi at 1 Mbps
Artifact     check-in (RegistrationService::checkIn)
Response     status becomes CheckedIn, green tick shown,
             duplicate scans are rejected with the reason
Measure      95 % of check-ins confirmed within 1 s;
             throughput >= 20 per minute per door
             (200 / 10 min = 20/min, one every 3 s)
```

⚠ "The system shall be fast and always available" is not a requirement: it has no measure and cannot be tested. Every scenario needs a number.

TOPIC 9

Platform constraints: browsers, app stores, devices, connectivity



everything to the right of your code is outside your control, so it becomes a constraint

Constraint	What it means for design
Browsers	Test on at least two engines (Chromium and Safari's WebKit); check a feature is supported before using it; storage can be evicted.
App stores	Review before every release, store fees on sales, rules on payments and privacy; a critical fix can wait days.
Devices	Design for the smallest, slowest phone your users own; touch targets, not mouse hover; many OS versions at once.
Connectivity	Assume slow, expensive and sometimes absent: small pages, caching, retries, an offline mode where it matters.

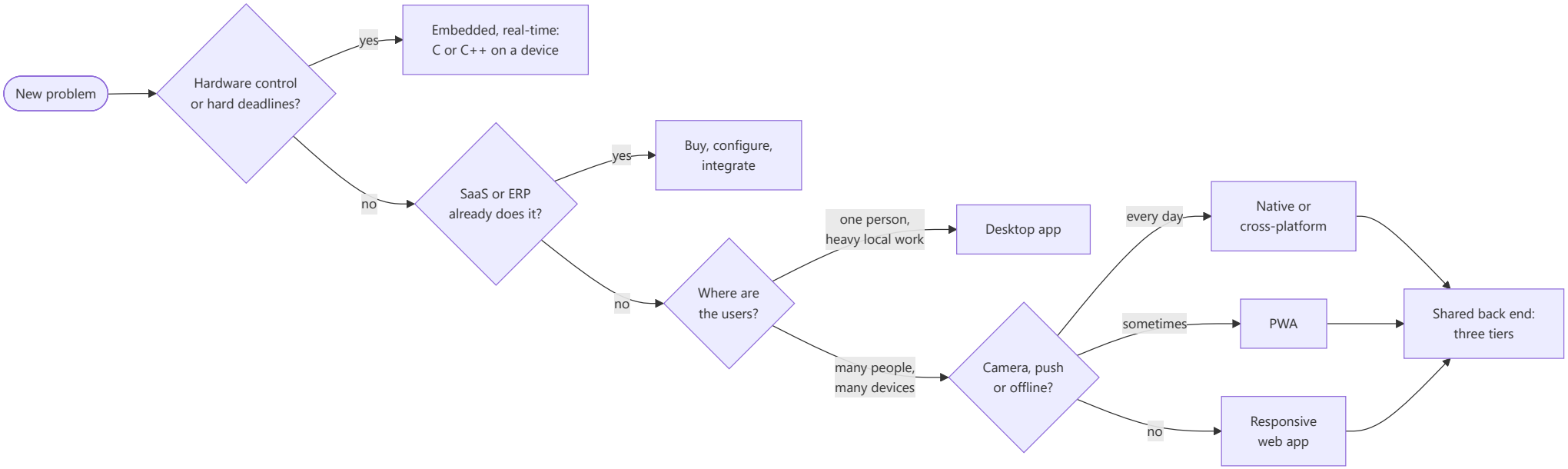
Short calculation: page weight

Time ≈ size in bits / bandwidth.

A 3 MB page on a 2 Mbps link: 3 × 8 / 2 = **12 s**. Most users leave before that.

Trimmed to 600 KB (compressed images, no unused libraries): 0.6 × 8 / 2 = **2.4 s**.

Choosing an application type for a problem



1. **Users and context**: who, where, on which devices, how often? (Your Lab-01 stakeholder list.)
2. **Hard constraints**: hardware control, deadlines, offline, regulations, budget, team skills, semester length.
3. **Buy or build**: an existing SaaS may already solve 80 % of the problem.
4. **Dominant quality attributes**: write the top three as scenarios with measures.
5. **Compare two or three options** with a weighted matrix and check deal-breakers.
6. **Record the decision**: context, options, decision, consequences. Revisit it when the context changes.

> A product often combines types: an IoT device + cloud + mobile app, or a PWA for students + the same web app on desktop for admins.

Worked example: an architecture decision record for a campus shuttle tracker

```
ADR-001  Application types for the ITC shuttle tracker
Status   Accepted, 2026-10-02        Deciders: whole team, client

Context  Students want to see where the campus shuttle is and when it
         arrives. Buses have no computer today. Most students have
         Android phones; mobile data is limited; the budget is small.
Options  A. GPS module on each bus (embedded) + native Android app
         B. GPS module on each bus + PWA + small cloud back end
         C. Drivers share location from their own phones + web page
Decision B. The GPS unit (C++ firmware) sends a position every 10 s
         over mobile data; students open a PWA from a QR code at
         each stop; no install and no store review needed.
Consequences
  + one web codebase for Android and iPhone; updates are instant
  + bus position does not depend on the driver's phone
  - firmware must be updated over the air (OTA), a new skill
  - no background push on iPhones unless the PWA is installed
Rejected A: two front ends later for iPhone, store releases.
         C: privacy of drivers' phones, stops when the battery dies.
```

- An **architecture decision record** (ADR) is a short text file kept in the repository, one per important decision, numbered and never deleted (a later ADR can supersede it).
- The valuable part is the *context* and the *rejected options*: a new team member learns *why*, not just *what*.
- Notice the combination of types: **embedded** firmware, a **cloud** back end, a **PWA** front end.
- Consequences list both benefits (+) and costs (-). A decision with no costs has not been thought through.

In Lab-02 Task 2 your team writes the same kind of record for ITC Club Hub, for three user groups at once.

Best practices and common mistakes

1 Start from users and context

Decide the type from who uses the system, where, on which devices and with what connection. Not from the technology the team wants to try.

2 Write qualities with numbers

Use six-part scenarios with a response measure. "Fast" and "secure" cannot be tested; "95 % within 1 s at 20 check-ins per minute" can.

3 Keep rules in the core

Business rules belong in the logic tier, not in the page or the app screen. Then a second front end costs a front end, not a second copy of the rules.

✕ Mistake: "we need an app"

Building two native apps by default, for users who would have opened a link once a week. Compare native, cross-platform, PWA and web first.

✕ Mistake: testing on your own laptop only

Fast Wi-Fi and a large screen hide the real constraints. Test on a cheap phone, a slow network and at least two browsers.

✕ Mistake: undocumented decisions

Six months later nobody remembers why the team chose a PWA. An ADR with rejected options takes 20 minutes and saves weeks of re-arguing.

Chapter quiz 10 questions

1. Which of these is *system* software?

- Telegram
- The driver for a laptop Wi-Fi card
- Moodle
- A spreadsheet file with exam marks

2. In a single-page application, after the first load, what does the server usually return when a student clicks Register?

- A complete new HTML page
- The whole JavaScript bundle again
- JSON data that the JavaScript uses to update part of the page
- Nothing: SPAs never contact the server

3. Which technology lets a progressive web app keep working when the venue Wi-Fi drops?

- An app store licence
- A service worker with a cache and IndexedDB
- Cookies
- A larger JavaScript bundle

4. A suite in which finance, HR and procurement modules share one database is called:

- CRM
- ERP
- IDE
- IoT

5. A function runs `sensor.slipRatio()`, sets a brake valve, and calls `log.deadlineMissed()` when `Clock::now() > deadline`, every 5 ms. What kind of application is it?

- A soft real-time game
- A batch data pipeline
- A hard real-time embedded controller
- A multi-page web application

6. In the PaaS model, what does *your team* still manage?

- The physical servers
- Operating system patches
- The virtualisation layer
- The application code and its data

7. In a three-tier Club Hub, where must the rule "capacity cannot be exceeded" be enforced?

- In the application (logic) tier
- Only in the mobile screen
- Only in the browser JavaScript
- In the network router

8. In a quality attribute scenario, "95 % of check-ins confirmed within 1 s" is the:

- Stimulus
- Environment
- Response measure
- Source

9. Club Hub must be available 99.5 % of a 30-day month. How much downtime is allowed?

- 36 minutes
- 3.6 hours
- 7.2 hours
- 5 hours

10. A 3 MB page is downloaded over a 2 Mbps mobile link. Roughly how long does the transfer take?

- 1.5 s
- 6 s
- 12 s
- 24 s

WRAP-UP

Summary

System software (firmware, OS, drivers, runtimes) runs the machine; application software serves a user's task.	Enterprise systems (ERP, CRM) integrate finance, HR and customers around shared data; integration and a long life dominate.	Client-server and three-tier split presentation, logic and data; keep business rules in the core, never only in the UI.
Desktop, web and mobile applications differ in where the code runs and how it is delivered and updated.	In embedded real-time systems a late answer is a wrong answer; C and C++ dominate there, in games and in performance libraries.	Quality attributes depend on the type; write them as six-part scenarios with a measurable response.
An SPA loads once and exchanges JSON; an MPA reloads pages. A PWA adds a manifest and a service worker: installable, works offline.	IaaS, PaaS and SaaS differ in which layers of the stack you manage and which the provider manages.	Choose a type from users, devices, connectivity, qualities and constraints, and record the decision with its rejected options.

Open Lab-02: 5 tasks + 1 challenge

Next chapter: 03 · Software Engineering Ethics