FREE DEV CHEAT SHEET

# THE DEV FIELD BOOK

30 chapters of practical reference for web, robotics, games and AI.

**By Mohamed Nasser - moonasser.site**

No fluff. Every page is something you will actually use.
Copy the code, print the tables, keep it open while you build.
Made for students, self-taught developers and makers.

# Contents

*30 chapters. Jump to what you need.*





*“The best error message is the one that never shows up. Mine writes novels.”*

# 01. HTML Essentials

*The skeleton of every page: structure, semantics and forms.*

## Page shell you should memorise

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1" />
  <title>Page title under 60 characters</title>
  <meta name="description" content="Short summary under 160 characters." />
  <link rel="stylesheet" href="style.css" />
</head>
<body>
  <header>...</header>
  <main>...</main>
  <footer>...</footer>
  <script src="app.js" defer></script>
</body>
</html>
```

## Semantic tags and when to use them

| | |
|---|---|
| **<header>** | Top of a page or of a section: logo, title, nav. |
| **<nav>** | A block of navigation links. One main nav per page. |
| **<main>** | The unique content of this page. Only one per page. |
| **<section>** | A themed group of content that has a heading. |
| **<article>** | Standalone content: a blog post, a product card. |
| **<aside>** | Side content: related links, notes, ads. |
| **<figure>/<figcaption>** | An image or chart plus its caption. |
| **<button> vs <a>** | Button does something. Link goes somewhere. |

## Forms that actually work

```
<form action="/subscribe" method="post">
  <label for="email">Email</label>
  <input id="email" name="email" type="email" required
         autocomplete="email" placeholder="you@example.com" />
  <button type="submit">Subscribe</button>
</form>
```

## Rules of thumb

- Every input needs a <label for> - placeholders are not labels.
- One <h1> per page, then h2, h3 in order. Never skip levels for styling.
- Images always get alt text. Decorative images get alt="".
- Use type=email / tel / number so mobile shows the right keyboard.

*"Do not worry if it does not work. If it did, you would be out of a job."*

– Add loading="lazy" to images below the fold.

# 02. CSS Layout: Flexbox

*One dimension at a time: rows or columns.*

## The core recipe

```
.row {
  display: flex;
  flex-direction: row;   /* row | column */
  justify-content: space-between; /* main axis */
  align-items: center;            /* cross axis */
  gap: 16px;
  flex-wrap: wrap;
}
.grow { flex: 1 1 0; }   /* fill the leftover space */
.fixed { flex: 0 0 240px; }
```

## Property cheat table

| | |
|---|---|
| **justify-content** | flex-start, center, flex-end, space-between, space-around, space-evenly |
| **align-items** | stretch (default), center, flex-start, flex-end, baseline |
| **align-self** | Override align-items for one child only. |
| **flex: 1** | Shorthand for grow 1, shrink 1, basis 0%. |
| **gap** | Space between items. Replaces margin hacks. |
| **order** | Visual reorder only - screen readers keep DOM order. |

## Patterns you will reuse forever

```
/* Perfect centering */
.center { display:flex; align-items:center; justify-content:center; min-height:100svh; }

/* Sticky footer */
body { min-height:100svh; display:flex; flex-direction:column; }
main { flex:1; }

/* Nav: logo left, links right */
.nav { display:flex; justify-content:space-between; align-items:center; }
```

## Traps

– min-width:auto stops flex children from shrinking - set min-width:0.
– Use 100svh not 100vh on mobile, or the browser bar cuts your layout.
– Flexbox is for one axis. Two axes at once means Grid.

*“if (tired) sleep(); else code(); // never reaches else”*

# 03. CSS Layout: Grid

*Two dimensions: rows and columns together.*

## Responsive card grid with no media queries

```
.grid {
  display: grid;
  grid-template-columns: repeat(auto-fit, minmax(260px, 1fr));
  gap: 24px;
}
```

## Named areas for page layouts

```
.page {
  display: grid;
  grid-template-columns: 240px 1fr;
  grid-template-rows: auto 1fr auto;
  grid-template-areas:
    "head head"
    "side main"
    "foot foot";
  min-height: 100svh;
}
.head{grid-area:head} .side{grid-area:side}
.main{grid-area:main} .foot{grid-area:foot}
```

## Grid vocabulary

| | |
|---|---|
| **`1fr`** | One share of the free space. |
| **`minmax(a,b)`** | Never smaller than a, never bigger than b. |
| **`auto-fit vs auto-fill`** | auto-fit collapses empty tracks, auto-fill keeps them. |
| **`grid-column: span 2`** | Make one item two columns wide. |
| **`place-items: center`** | Centers every item on both axes in one line. |
| **`subgrid`** | Child grid inherits the parent tracks - great for card rows. |

## Decision rule

- Content decides the layout -> Flexbox.
- Layout decides where content goes -> Grid.
- Mixing both in one page is normal and correct.

*“AI will not replace developers. It will just co-write the bugs.”*

# 04. Responsive Design & Modern Units

*Design once, fit every screen.*

## Breakpoints that cover the real world

```
/* mobile first: write the phone styles, then add up */
@media (min-width: 640px)  { /* large phone / small tablet */ }
@media (min-width: 768px)  { /* tablet */ }
@media (min-width: 1024px) { /* laptop */ }
@media (min-width: 1280px) { /* desktop */ }
```

## Units

| | |
|---|---|
| **rem** | Relative to root font size. Use for spacing and text. |
| **em** | Relative to the element font size. Good inside components. |
| **svh / dvh / lvh** | Small / dynamic / large viewport height on mobile. |
| **ch** | Width of the 0 character. max-width: 65ch = readable line. |
| **clamp(min, ideal, max)** | Fluid value with hard limits. |
| **%** | Relative to the parent - beware of percentage heights. |

## Fluid type and spacing with no breakpoints

```
h1   { font-size: clamp(2rem, 5vw + 1rem, 4rem); }
.sec { padding-block: clamp(3rem, 8vw, 7rem); }
.prose { max-width: 65ch; }
img, video { max-width: 100%; height: auto; display: block; }
```

## Mobile checklist

- Tap targets at least 44x44 px.
- Never let a page scroll sideways: use overflow-x: clip on body.
- Test at 320px width - the smallest phone still in use.
- Respect prefers-reduced-motion for every animation.

*“My code does not have bugs, it has undocumented surprise features.”*

# 05. Tailwind CSS in Practice

*Utility classes without the mess.*

## Class order that stays readable

```
<!-- layout -> spacing -> size -> colour -> text -> state -->
<div class="flex items-center gap-3 p-4 w-full rounded-xl
            border bg-white text-sm hover:shadow-lg">
```

## Most used utilities

| | |
|---|---|
| **flex / grid** | display. Add items-center, justify-between, gap-4. |
| **p-4 px-6 py-2** | padding. m-* is margin. Scale: 1 = 0.25rem. |
| **w-full max-w-3xl** | width limits. mx-auto centres a block. |
| **text-sm/base/xl** | font size. font-bold, tracking-tight, leading-relaxed. |
| **rounded-xl border** | shape. shadow-md for depth. |
| **hover: focus: sm: dark:** | Variants. Stack them: sm:hover:bg-red-500. |
| **space-y-4** | Vertical rhythm between direct children. |

## Design tokens instead of hard-coded colours

```
/* styles.css */
@theme {
  --color-ink:    #1A1A1A;
  --color-paper:  #FDFBF6;
  --color-brand:  #E63946;
}
/* then in markup */
<button class="bg-brand text-paper">Send</button>
```

## Habits

- Repeating 10 classes three times? Make a component, not a copy.
- Never hard-code hex colours in markup - dark mode will break.
- Use line-clamp-2 / truncate instead of manual text cutting.

*“I told my computer I needed a break. It said: Error 418, I am a teapot.”*

# 06. JavaScript Core

*The language parts you use every single day.*

## Variables, values, truthiness

```
const fixed = 1;      // cannot rebind
let counter = 0;      // can rebind
// falsy: false 0 -0 0n "" null undefined NaN -> everything else is truthy
const name = input ?? "guest";   // only null/undefined fall through
const port = cfg.port || 3000;   // careful: 0 falls through too
user?.profile?.city;             // safe access
```

## Array methods ranked by usefulness

| | |
|---|---|
| **`map`** | Transform each item -> new array of the same length. |
| **`filter`** | Keep the items that pass a test. |
| **`find / findIndex`** | First match, or undefined / -1. |
| **`reduce`** | Fold a list into one value (sum, group, object). |
| **`some / every`** | Boolean: any match / all match. |
| **`sort((a,b)=>a-b)`** | Sorts in place. Copy first with [...arr]. |
| **`flat / flatMap`** | Flatten nested arrays one level. |
| **`includes`** | Simple membership test. |

## Destructuring and spread

```
const { title, tags = [] } = post;
const [first, ...rest] = list;
const next = { ...state, loading: false };
const merged = [...a, ...b];
function draw({ x = 0, y = 0 } = {}) { /* ... */ }
```

## Group a list in one line

```
const byTag = items.reduce((acc, it) => {
  (acc[it.tag] ||= []).push(it);
  return acc;
}, {});
```

*"Works 60% of the time, every time."*

# 07. DOM & Events

*Talking to the page from JavaScript.*

## Select, change, create

```
const el  = document.querySelector("#app");
const all = document.querySelectorAll(".card");

el.textContent = "safe text";       // never innerHTML with user input
el.classList.toggle("open", isOpen);
el.dataset.state = "ready";         // <div data-state="ready">

const li = document.createElement("li");
li.textContent = "New item";
list.append(li);
```

## Event delegation: one listener for a whole list

```
list.addEventListener("click", (e) => {
  const btn = e.target.closest("[data-id]");
  if (!btn) return;
  remove(btn.dataset.id);
});
```

## Useful browser APIs

| | |
|---|---|
| **`IntersectionObserver`** | Run code when an element enters the screen. |
| **`ResizeObserver`** | React to element size changes, not window size. |
| **`AbortController`** | Cancel a fetch or remove many listeners at once. |
| **`localStorage`** | Small strings that survive reload. Not for secrets. |
| **`matchMedia`** | Read a media query from JS, e.g. dark mode. |
| **`requestAnimationFrame`** | Run code once per frame, smooth animation. |

## Rules

- Debounce input handlers, throttle scroll handlers.
- Always remove listeners you add on temporary elements.
- Reading layout (offsetWidth) inside a loop causes thrash - batch reads.

*“There is no cloud, it is just someone else laptop. Be nice to it.”*

# 08. Async JavaScript & Fetch

*Promises, await, and talking to APIs.*

## Fetch with error handling and timeout

```
async function getJSON(url, ms = 8000) {
  const ctrl = new AbortController();
  const timer = setTimeout(() => ctrl.abort(), ms);
  try {
    const res = await fetch(url, { signal: ctrl.signal });
    if (!res.ok) throw new Error(`HTTP ${res.status}`);
    return await res.json();
  } finally {
    clearTimeout(timer);
  }
}
```

## Running things together vs one by one

```
// slow: 3 x latency
const a = await f1(); const b = await f2(); const c = await f3();

// fast: all at once
const [a2, b2, c2] = await Promise.all([f1(), f2(), f3()]);

// never fail the whole batch
const results = await Promise.allSettled(tasks);
```

## Promise helpers

| | |
|---|---|
| **`Promise.all`** | All must succeed. Fails fast on first error. |
| **`Promise.allSettled`** | Always resolves with status per item. |
| **`Promise.race`** | First to settle wins - good for timeouts. |
| **`Promise.any`** | First success wins, ignores failures. |
| **`for await (... of ...)`** | Consume async iterators / streams. |

## Retry with backoff

```
async function retry(fn, tries = 3) {
  for (let i = 0; i < tries; i++) {
    try { return await fn(); }
    catch (e) {
      if (i === tries - 1) throw e;
      await new Promise(r => setTimeout(r, 2 ** i * 300));
    }
  }
}
```

*"TODO: remove this before launch. Launched 2 years ago."*

# 09. React Fundamentals

*Components, props, state and lists.*

## A component is a function that returns UI

```
function Card({ title, children, onOpen }) {
  return (
    <article className="card" onClick={onOpen}>
      <h3>{title}</h3>
      {children}
    </article>
  );
}
```

## State drives the screen

```
const [count, setCount] = useState(0);
setCount(c => c + 1);          // use the updater when it depends on the old value

const [form, setForm] = useState({ name: "", email: "" });
setForm(f => ({ ...f, name: value }));   // never mutate state
```

## Lists, keys and conditions

```
{items.map(it => <Row key={it.id} item={it} />)}
{isLoading ? <Spinner /> : <List items={items} />}
{error && <p role="alert">{error}</p>}
```

## Core rules

| | |
|---|---|
| **Keys** | Stable unique id. Never the array index for dynamic lists. |
| **Props flow down** | A child never edits props - it calls a callback. |
| **Lift state up** | Two siblings need the same data -> move it to the parent. |
| **Derive, do not store** | total = items.length: compute it, do not useState it. |
| **Pure render** | No fetch, no DOM writes, no randomness during render. |

*“It worked on my machine, so ship the machine.”*

# 10. React Hooks & Patterns

*The hooks that matter and how not to abuse them.*

## Hook map

| | |
|---|---|
| **`useState`** | Local value that changes over time. |
| **`useEffect`** | Sync with the outside world: subscriptions, timers. |
| **`useRef`** | Mutable box that survives renders. Also DOM handles. |
| **`useMemo`** | Cache an expensive calculation. |
| **`useCallback`** | Stable function identity for memoised children. |
| **`useReducer`** | Several values that change together by action. |
| **`useContext`** | Read shared data without prop drilling. |
| **`useId`** | Stable unique id for label/input pairs. |

## Effect with cleanup - the shape to memorise

```
useEffect(() => {
  const ctrl = new AbortController();
  fetch(url, { signal: ctrl.signal }).then(/* ... */);
  return () => ctrl.abort();   // runs before the next effect and on unmount
}, [url]);
```

## A tiny custom hook

```
function useDebounced(value, delay = 300) {
  const [v, setV] = useState(value);
  useEffect(() => {
    const id = setTimeout(() => setV(value), delay);
    return () => clearTimeout(id);
  }, [value, delay]);
  return v;
}
```

## Do not

- Do not use an effect to compute state from props - just compute it.
- Do not call hooks in conditions or loops. Top level only.
- Do not fetch in an effect if your framework has a loader - use it.
- Do not wrap everything in useMemo. Measure first.

*“Real programmers count from 0 and apologise later.”*

# 11. TypeScript Survival Kit

*Types that catch bugs before your users do.*

## Everyday syntax

```
type Role = "admin" | "editor" | "viewer";

interface User {
  id: string;
  name: string;
  role: Role;
  bio?: string;              // optional
  readonly createdAt: Date;
}

function greet(u: User): string { return `Hi ${u.name}`; }
const ids: string[] = users.map(u => u.id);
```

## Utility types

| | |
|---|---|
| **`Partial<T>`** | All fields optional - patch objects. |
| **`Required<T>`** | All fields required. |
| **`Pick<T,'a'\|'b'>`** | Keep only some fields. |
| **`Omit<T,'a'>`** | Drop some fields. |
| **`Record<K,V>`** | Dictionary type: Record<string, number>. |
| **`ReturnType<F>`** | The type a function returns. |
| **`as const`** | Freeze a literal so its exact values become the type. |

## Narrowing instead of casting

```
function area(s: Shape) {
  if (s.kind === "circle") return Math.PI * s.r ** 2;  // narrowed
  return s.w * s.h;
}
// avoid `as` - it silences the compiler, it does not check anything
if (typeof x === "string") x.trim();
if (Array.isArray(y)) y.length;
```

## Habits

- Turn on strict mode from day one - retrofitting it is painful.
- Type the boundaries (API responses, props). Let inference do the rest.
- unknown is safe, any is a hole in your type system.

*“Programming is 10% writing code and 90% wondering why it works.”*

# 12. Git: Daily Commands

*Version control you can actually trust.*

## From zero to pushed

```
git init
git add .
git commit -m "feat: first version"
git branch -M main
git remote add origin https://github.com/you/repo.git
git push -u origin main
```

## Command table

| | |
|---|---|
| **git status -sb** | Short view of what changed and which branch. |
| **git switch -c feat/x** | Create and move to a new branch. |
| **git add -p** | Stage hunk by hunk - makes clean commits. |
| **git commit --amend** | Fix the last commit message or content. |
| **git pull --rebase** | Update without a noisy merge commit. |
| **git log --oneline --graph** | Readable history. |
| **git diff --staged** | See exactly what you are about to commit. |
| **git stash / stash pop** | Park work, switch task, bring it back. |

## Undo table - the one you will search for

| | |
|---|---|
| **Unstage a file** | git restore --staged file |
| **Throw away local edits** | git restore file |
| **Undo last commit, keep work** | git reset --soft HEAD~1 |
| **Undo last commit, drop work** | git reset --hard HEAD~1 |
| **Revert a pushed commit** | git revert <sha> |
| **Find a lost commit** | git reflog |

## Commit messages that read well

- feat: add contact form / fix: stop double submit / docs: update readme
- chore, refactor, test, style, perf are the other common prefixes.
- Present tense, under 72 characters, explain why in the body.

*"I have a joke about recursion, but first, I have a joke about recursion."*

# 13. Git: Branching & Collaboration

*Working with other people without pain.*

## Feature branch flow

```
git switch main && git pull
git switch -c feat/contact-form
# ...work, commit...
git push -u origin feat/contact-form
# open a Pull Request, get review, squash merge, delete branch
```

## Fixing a conflict

```
git pull --rebase origin main
# git marks the file:
# <<<<<<< HEAD  (your side)
# =======       (their side)
# >>>>>>> main
# edit the file, keep what is right, remove the markers
git add file
git rebase --continue
```

## Good PR checklist

- One purpose per PR. Small PRs get reviewed, big ones get ignored.
- Describe what changed and how to test it.
- Add a screenshot or short clip for anything visual.
- Self review your own diff before asking anyone else.
- Never commit .env, node_modules, build output or API keys.

## .gitignore starter

```
node_modules/
dist/
build/
.env
.env.local
.DS_Store
*.log
```

*“Estimated: 2 hours. Actual: 2 hours and one existential crisis.”*

# 14. Terminal & Shell

*Move faster without touching the mouse.*

## Navigation and files

| | |
|---|---|
| **pwd / ls -la / cd -** | Where am I / list all / jump back. |
| **mkdir -p a/b/c** | Create nested folders in one go. |
| **cp -r src dst** | Copy folder. mv renames and moves. |
| **rm -rf folder** | Delete recursively. Check the path twice. |
| **cat / less / head / tail -f** | Read files. tail -f follows a live log. |
| **du -sh * \| sort -h** | What is eating my disk? |

## Search like a pro

```
grep -rn "TODO" src/          # recursive, with line numbers
rg "useState" -l              # ripgrep: much faster, list files
find . -name "*.png" -size +1M
ls -la | grep ".env"
history | grep docker
```

## Pipes, redirects, chains

```
cmd > out.txt      # write (overwrite)
cmd >> out.txt     # append
cmd 2>&1 | tee log.txt   # stdout + stderr to screen AND file
cmd1 && cmd2       # run cmd2 only if cmd1 succeeded
cmd1 || cmd2       # run cmd2 only if cmd1 failed
cmd &              # run in background
```

## Keyboard shortcuts

| | |
|---|---|
| **Ctrl+C / Ctrl+D** | Kill the process / send end of input. |
| **Ctrl+R** | Search your command history. |
| **Ctrl+A / Ctrl+E** | Jump to start / end of the line. |
| **Ctrl+W / Ctrl+U** | Delete previous word / whole line. |
| **!! and !$** | Last command / last argument. sudo !! is a classic. |

*"The definition of insanity: refreshing localhost expecting a different result."*

# 15. VS Code Mastery

*The editor is a tool - learn to play it.*

## Shortcuts (Cmd on macOS = Ctrl elsewhere)

| | |
|---|---|
| **Ctrl+P** | Quick open any file by name. |
| **Ctrl+Shift+P** | Command palette - everything lives here. |
| **Ctrl+D** | Select next occurrence. Ctrl+Shift+L selects all. |
| **Alt+Click** | Add another cursor anywhere. |
| **Alt+Up/Down** | Move the current line. |
| **Shift+Alt+Down** | Duplicate the current line. |
| **Ctrl+/** | Toggle comment. |
| **F2** | Rename symbol across the whole project. |
| **F12 / Alt+F12** | Go to definition / peek definition. |
| **Ctrl+Shift+F** | Search in all files. |
| **Ctrl+`** | Toggle the integrated terminal. |
| **Ctrl+B** | Toggle sidebar for more code space. |

## settings.json worth copying

```
{
  "editor.formatOnSave": true,
  "editor.defaultFormatter": "esbenp.prettier-vscode",
  "editor.tabSize": 2,
  "editor.linkedEditing": true,
  "editor.bracketPairColorization.enabled": true,
  "files.autoSave": "onFocusChange",
  "files.trimTrailingWhitespace": true,
  "emmet.includeLanguages": { "javascript": "javascriptreact" }
}
```

## Extensions that pay for themselves

- Prettier + ESLint: formatting and lint errors while typing.
- GitLens: who wrote this line and why.
- Error Lens: shows errors inline instead of hidden underlines.
- Tailwind CSS IntelliSense: class autocomplete and colour previews.
- Thunder Client / REST Client: test APIs without leaving the editor.

*“I do not always test my code, but when I do, I do it in production.”*

# 16. Debugging & DevTools

*Stop guessing. Read the evidence.*

## A repeatable method

- 1. Reproduce it reliably. A bug you cannot repeat cannot be fixed.
- 2. Read the actual error and the first line of the stack trace.
- 3. Cut the problem in half: remove code until it stops happening.
- 4. Check your assumptions with a log or a breakpoint, not your memory.
- 5. Fix the cause, not the symptom. Then add a test or a guard.

## Console beyond console.log

```
console.table(users);            // arrays of objects as a grid
console.group("checkout"); ... console.groupEnd();
console.time("load"); ... console.timeEnd("load");
console.assert(total > 0, "total must be positive");
console.trace();                 // how did we get here?
debugger;                        // pause here when DevTools is open
```

## DevTools panels

| | |
|---|---|
| **Elements** | Live DOM + computed CSS. Edit styles to test ideas. |
| **Console** | Errors, logs, and a full JS REPL on the page. |
| **Network** | Status, timing, payloads. Throttle to Slow 4G. |
| **Application** | localStorage, cookies, service workers, cache. |
| **Performance** | Record and find the slow frames. |
| **Lighthouse** | Score performance, a11y, SEO, best practices. |

## Common front-end errors decoded

| | |
|---|---|
| **Cannot read properties of undefined** | You used x.y before x existed. Use ?. or guard. |
| **Hydration mismatch** | Server HTML differs from client. Move browser-only reads into an effect. |
| **CORS blocked** | The server must send the header. You cannot fix it in the browser. |
| **Maximum update depth** | setState during render or a bad effect dependency. |



*“It is not a bug, it is a feature request from the universe.”*

# 17. Web Performance

*Fast sites make more money and rank higher.*

## The three metrics you are graded on

| | |
|---|---|
| `LCP < 2.5s` | Largest Contentful Paint: the hero image or headline. |
| `INP < 200ms` | Interaction to Next Paint: how snappy clicks feel. |
| `CLS < 0.1` | Cumulative Layout Shift: things jumping around. |

## Images: the biggest win, always

- Serve WebP or AVIF instead of PNG/JPG - often 50-70% smaller.
- Always set width and height so the layout does not jump.
- loading="lazy" below the fold, fetchpriority="high" on the hero.
- Use srcset so phones do not download desktop-sized files.

## JavaScript and fonts

```
<script src="app.js" defer></script>       <!-- never blocking -->
const Chart = React.lazy(() => import("./Chart"));   // split heavy code

/* font that never blocks text */
@font-face { font-family: Brand; src: url(b.woff2) format("woff2");
             font-display: swap; }
```

## Quick audit routine

- Run Lighthouse in an incognito window, mobile preset.
- Sort the Network tab by size: anything over 200KB needs a reason.
- Count requests before first paint. Fewer is faster.
- Cache static assets for a year with a hashed filename.

# 18. Accessibility (a11y)

*Usable by everyone, including future you.*

## The quick wins

- Use real <button> and <a> elements - they get keyboard support free.
- Every image: meaningful alt, or alt="" if purely decorative.
- Colour contrast at least 4.5:1 for body text, 3:1 for large text.
- Never remove the focus ring. Restyle it instead.
- Label every form field and link errors with aria-describedby.
- Do not rely on colour alone to show state - add an icon or text.

## ARIA you will actually use

| | |
|---|---|
| `aria-label` | Name for an element with no visible text (icon buttons). |

*"My favourite language is coffee, it compiles instantly."*

| | |
|---|---|
| `aria-expanded` | true/false on accordion and menu triggers. |
| `aria-current="page"` | Marks the active nav link. |
| `role="alert"` | Announce an error immediately. |
| `aria-live="polite"` | Announce updates when the user is idle. |
| `sr-only class` | Visible to screen readers, hidden on screen. |

### Keyboard test - takes 60 seconds

- Tab through the whole page. Can you reach every control?
- Is the focus always visible and in a logical order?
- Escape closes modals. Enter and Space activate buttons.
- Focus is trapped inside an open modal and returns when it closes.

### Reduced motion

```
@media (prefers-reduced-motion: reduce) {
  *, *::before, *::after {
    animation-duration: .01ms !important;
    transition-duration: .01ms !important;
  }
}
```

# 19. SEO for Developers

*Being found is part of shipping.*

### On-page basics

| | |
|---|---|
| `<title>` | Unique per page, under 60 chars, keyword near the front. |
| `meta description` | Under 160 chars. It is your ad copy, not a ranking factor. |
| `One <h1>` | Matches the page intent. Then h2/h3 in order. |
| `canonical` | Point duplicates at the one true URL. |
| `og:title / og:image` | Controls how the link looks when shared. |
| `robots.txt + sitemap.xml` | Tell crawlers what exists and what to skip. |
| `alt text` | Image SEO and accessibility in one move. |

### Structured data example

*"A programmer stops at a red light and waits for the callback."*

```
<script type="application/ld+json">
{
  "@context": "https://schema.org",
  "@type": "Person",
  "name": "Your Name",
  "jobTitle": "Web Developer",
  "url": "https://yoursite.com",
  "sameAs": ["https://github.com/you", "https://linkedin.com/in/you"]
}
</script>
```

### Technical rules

- Content must exist in the HTML, not only after JavaScript runs.
- Clean readable URLs: /projects/robot-arm, not /p?id=42.
- Fast mobile pages rank better - performance is an SEO feature.
- Internal links matter: link your pages to each other.
- Fix broken links and redirect old URLs with 301, never 404 silently.

# 20. Python Essentials

*Scripting, automation and data work.*

### Data structures at a glance

| | |
|---|---|
| **`list [1,2,3]`** | Ordered, changeable. append, pop, sort. |
| **`tuple (1,2)`** | Ordered, fixed. Good as dict keys. |
| **`dict {"a":1}`** | Key -> value. get, items, keys, values. |
| **`set {1,2}`** | Unique items. Fast membership, union, intersection. |
| **`f-string`** | f"{name} is {age} years old" |

### Comprehensions and unpacking

```
squares = [n * n for n in range(10) if n % 2 == 0]
by_id    = {u["id"]: u for u in users}
names    = {u["name"] for u in users}          # set, no duplicates
a, b     = b, a                                 # swap
first, *rest = numbers
```

### Files, JSON and errors



*"The app is fast, the internet is slow, blame the internet."*

```
import json, pathlib

data = json.loads(pathlib.Path("in.json").read_text())
pathlib.Path("out.json").write_text(json.dumps(data, indent=2))

try:
    value = int(text)
except ValueError as e:
    print("bad number:", e)
finally:
    print("done")
```

### Useful standard library

- pathlib - paths done right, forget os.path.
- collections.Counter / defaultdict - counting and grouping.
- itertools - chain, groupby, combinations.
- datetime + zoneinfo - dates without a library.
- argparse - turn a script into a real CLI tool.
- venv - one isolated environment per project, always.

# 21. C++ Essentials

*The language behind performance and hardware.*

### Program shape

```
#include <iostream>
#include <vector>
#include <string>
using namespace std;

int main() {
    vector<int> nums = {5, 2, 9};
    for (int n : nums) cout << n << " ";
    string name = "Nasser";
    cout << "\nHello " << name << endl;
    return 0;
}
```

### Core concepts

| | |
|---|---|
| **Pass by reference** | void f(vector<int>& v) - no copy, can modify. |
| **const ref** | const string& s - no copy, read only. Default choice. |
| **Pointer vs reference** | A pointer can be null and rebound, a reference cannot. |
| **new/delete** | Avoid. Use vector, string and smart pointers. |
| **unique_ptr** | Owns memory, frees it automatically. Cannot be copied. |
| **auto** | Let the compiler write the type for you. |
| **struct vs class** | Same thing; struct is public by default. |

*“Merge conflicts: where two good ideas go to fight.”*

### STL you will use

```
#include <algorithm>
sort(v.begin(), v.end());
auto it = find(v.begin(), v.end(), 42);
int total = accumulate(v.begin(), v.end(), 0);
reverse(v.begin(), v.end());

map<string,int> counts;   counts["a"]++;
unordered_map<string,int> fast;   // hash, no ordering
```

### Compile and debug

```
g++ -std=c++17 -Wall -Wextra -g main.cpp -o app
./app
# sanitizers catch memory bugs early:
g++ -fsanitize=address,undefined -g main.cpp -o app
```

# 22. Arduino & Embedded Basics

*Where the code meets the real world.*

### The two functions

```
const int LED = 13;
const int BTN = 2;

void setup() {
  Serial.begin(9600);
  pinMode(LED, OUTPUT);
  pinMode(BTN, INPUT_PULLUP);   // no external resistor needed
}

void loop() {
  if (digitalRead(BTN) == LOW) digitalWrite(LED, HIGH);
  else                         digitalWrite(LED, LOW);
}
```

### Pins and signals

| | |
|---|---|
| **digitalRead/Write** | HIGH or LOW only. Buttons, LEDs, relays. |
| **analogRead(A0)** | 0-1023 on Uno. Potentiometers, LDR, sensors. |
| **analogWrite(pin,0-255)** | PWM on ~ pins. Brightness, motor speed. |
| **INPUT_PULLUP** | Idle reads HIGH, pressed reads LOW. Fewer parts. |
| **millis()** | Non-blocking timing. Prefer it over delay(). |
| **Serial.println** | Your only debugger. Use it constantly. |

### Blink without delay - stop freezing your robot

*“There are 2 hard things in tech: naming things, cache invalidation, and off-by-one errors.”*

```
unsigned long last = 0;
const unsigned long INTERVAL = 500;

void loop() {
  unsigned long now = millis();
  if (now - last >= INTERVAL) {
    last = now;
    state = !state;
    digitalWrite(LED, state);
  }
  readSensors();   // keeps running - delay() would block this
}
```

### Hardware rules

- Never power motors from the Arduino 5V pin - use a driver and its own supply.
- All grounds must be connected together.
- Add a flyback diode across DC motors and relays.
- Debounce buttons in code (ignore changes under ~30ms).

# 23. Robotics: Sensors & Motors

*Read the world, then move in it.*

### Ultrasonic distance (HC-SR04)

```
long readCm(int trig, int echo) {
  digitalWrite(trig, LOW);  delayMicroseconds(2);
  digitalWrite(trig, HIGH); delayMicroseconds(10);
  digitalWrite(trig, LOW);
  long dur = pulseIn(echo, HIGH, 30000);   // timeout 30ms
  return dur == 0 ? -1 : dur * 0.034 / 2;  // -1 means nothing in range
}
```

### Line follower core logic

```
int L = digitalRead(leftIR), R = digitalRead(rightIR);
if (L == LOW && R == HIGH)      turnLeft();
else if (L == HIGH && R == LOW) turnRight();
else if (L == LOW && R == LOW)  stopMotors();   // junction or lifted
else                            forward();
```

### Motor control patterns

*“First rule of CSS: the div is centered when it says it is.”*

```
// L298N style driver
void drive(int in1, int in2, int en, int speed) {
  bool fwd = speed >= 0;
  digitalWrite(in1, fwd ? HIGH : LOW);
  digitalWrite(in2, fwd ? LOW : HIGH);
  analogWrite(en, constrain(abs(speed), 0, 255));
}

// Servo
#include <Servo.h>
Servo s; s.attach(9); s.write(90);   // 0-180 degrees
```

### Sensor picking guide

| | |
|---|---|
| **IR reflectance** | Line following, edge detection. Cheap, needs calibration. |
| **Ultrasonic** | Distance 2-400cm. Fails on soft or angled surfaces. |
| **IMU (MPU6050)** | Tilt and rotation. Needs a filter to be usable. |
| **Encoder** | Real wheel distance and speed. Essential for straight lines. |
| **LDR** | Light level. Simplest analog sensor to learn on. |
| **DHT11/22** | Temperature and humidity. Slow: read once per 2s. |

### Smoothing a noisy reading

```
float smooth = 0;                    // exponential filter
smooth = smooth * 0.8 + newReading * 0.2;
```

# 24. Unity: C# Basics

*Scripts, components and the game loop.*

### Lifecycle order

| | |
|---|---|
| **Awake()** | Once, before anything. Set up references to yourself. |
| **OnEnable()** | Every time the object turns on. Subscribe to events. |
| **Start()** | Once, before the first frame. Talk to other objects. |
| **Update()** | Every frame. Input and non-physics logic. |
| **FixedUpdate()** | Fixed step. All Rigidbody / physics work. |
| **LateUpdate()** | After all Updates. Camera follow lives here. |
| **OnDestroy()** | Clean up, unsubscribe. |

### Player movement done right

*“Refactoring: making the same bug prettier.”*

```
public class Player : MonoBehaviour {
  [SerializeField] float speed = 6f;
  Rigidbody2D rb;  Vector2 input;

  void Awake() { rb = GetComponent<Rigidbody2D>(); }

  void Update() {   // read input every frame
    input.x = Input.GetAxisRaw("Horizontal");
    input.y = Input.GetAxisRaw("Vertical");
  }

  void FixedUpdate() {   // move in the physics step
    rb.linearVelocity = input.normalized * speed;
  }
}
```

### Finding and creating things

```
transform.position += Vector3.up * Time.deltaTime;
var enemy = GameObject.FindWithTag("Enemy");
Instantiate(bulletPrefab, muzzle.position, muzzle.rotation);
Destroy(gameObject, 2f);          // delayed destroy
GetComponent<Animator>().SetBool("running", moving);
```

### Performance habits

- Cache GetComponent in Awake, never call it inside Update.
- Always multiply movement by Time.deltaTime.
- Pool bullets and particles instead of Instantiate/Destroy every shot.
- Use [SerializeField] private instead of public fields.



# 25. Unity: Physics, UI & Scenes

*Collisions, canvases and moving between levels.*

### Collision vs trigger

```
void OnCollisionEnter2D(Collision2D c) {   // solid impact
  if (c.gameObject.CompareTag("Ground")) grounded = true;
}

void OnTriggerEnter2D(Collider2D c) {      // pass-through zone
  if (c.CompareTag("Coin")) { score++; Destroy(c.gameObject); }
}
```

### Physics notes

| | |
|---|---|
| **Rigidbody** | Add it or physics does not apply. Kinematic = you move it. |
| **Is Trigger** | No blocking, only notifications. |
| **Layers + matrix** | Decide which layers can collide at all. |
| **Raycast** | Physics2D.Raycast for ground checks and aiming. |

*“Deleting node_modules is the modern form of meditation.”*

| | |
|---|---|
| **AddForce vs velocity** | Force feels heavy, velocity feels arcade. |
| **Interpolate** | Turn it on to remove jitter on moving bodies. |

### UI and scenes

```
using UnityEngine.UI;
using UnityEngine.SceneManagement;

public Text scoreText;
void UpdateScore(int s) { scoreText.text = $"Score: {s}"; }

SceneManager.LoadScene("Level2");
Time.timeScale = 0f;   // pause  (1f to resume)
PlayerPrefs.SetInt("best", score); PlayerPrefs.Save();
```

### Project hygiene

- Folders: Scripts, Prefabs, Scenes, Art, Audio, Materials.
- Everything repeated becomes a prefab - edit once, update everywhere.
- Use ScriptableObjects for weapon/enemy stats instead of hard-coded numbers.
- Build and test on the target device early, not the night before the deadline.

# 26. Game Design Loop

*Why some games feel good and others do not.*

### The core loop

- Every game is: challenge -> action -> feedback -> reward -> harder challenge.
- If the loop is not fun in 30 seconds with grey boxes, art will not save it.
- Prototype the verb first (jump, shoot, stack), then build content around it.

### Game feel checklist

| | |
|---|---|
| **Acceleration** | Never instant top speed. Ramp up and down. |
| **Coyote time** | Allow a jump ~0.1s after leaving the ledge. |
| **Input buffering** | Register a jump pressed just before landing. |
| **Screen shake** | Small and short. Big shake feels cheap. |
| **Hit stop** | Freeze 2-4 frames on impact - huge perceived weight. |
| **Audio** | A sound for every action. Silence feels broken. |
| **Particles** | Dust on landing, sparks on hit. Cheap, huge payoff. |

### Difficulty and progression

- Teach one mechanic per level, in a safe space, then test it.
- Introduce -> practise -> combine -> subvert.
- Player death should always feel like the player's fault.

*"The rubber duck reviewed my PR and approved it."*

– Show progress: a bar, a number, a new colour. People need proof.

### Playtesting

– Watch someone play in silence. Do not explain anything.
– Where they hesitate is a design bug, not a player problem.
– Ask what they were trying to do, never whether they liked it.

# 27. Databases & SQL

*Store data so you can ask it questions later.*

### Queries you need

```
SELECT id, name, created_at
FROM users
WHERE role = 'admin' AND created_at > '2026-01-01'
ORDER BY created_at DESC
LIMIT 20;

INSERT INTO posts (title, user_id) VALUES ('Hello', 3);
UPDATE posts SET title = 'New' WHERE id = 7;
DELETE FROM posts WHERE id = 7;
```

### Joins and grouping

```
SELECT u.name, COUNT(p.id) AS posts
FROM users u
LEFT JOIN posts p ON p.user_id = u.id
GROUP BY u.name
HAVING COUNT(p.id) > 2
ORDER BY posts DESC;
```

### Concepts

| | |
|---|---|
| **Primary key** | Unique id for every row. Usually uuid or bigint. |
| **Foreign key** | A column pointing at another table's key. |
| **Index** | Makes WHERE and JOIN fast. Costs write speed. |
| **INNER vs LEFT JOIN** | Only matches vs keep the left side always. |
| **Transaction** | BEGIN ... COMMIT: all of it, or none of it. |
| **Migration** | A versioned SQL file that changes the schema. |
| **N+1 query** | A query inside a loop. Join or batch it instead. |

### Rules

– Never build SQL by string concatenation - use parameters, always.
– Add an index on any column you filter or join on often.
– Store dates in UTC. Convert only when displaying.
– Soft delete (deleted_at) when the data has value later.

*“Ship it. Fix it. Tweet about it.”*

# 28. APIs, HTTP & Security

*Talking to servers safely.*

## HTTP status codes

| | |
|---|---|
| **200 / 201 / 204** | OK / Created / Success with no body. |
| **301 / 302** | Moved permanently / temporarily. |
| **400** | Bad request - the client sent something invalid. |
| **401 / 403** | Not logged in / logged in but not allowed. |
| **404 / 409** | Not found / conflict (duplicate). |
| **422 / 429** | Validation failed / too many requests. |
| **500 / 503** | Server crashed / temporarily unavailable. |

## REST shape

```
GET    /api/posts          list
GET    /api/posts/:id      one
POST   /api/posts          create        -> 201
PATCH  /api/posts/:id      partial edit
DELETE /api/posts/:id      remove        -> 204

// auth header
Authorization: Bearer <token>
```

## Security rules you cannot skip

- Secrets live in environment variables on the server. Never in the client bundle.
- Validate every input on the server. Client validation is only UX.
- Escape output / use textContent to stop XSS.
- Hash passwords with bcrypt or argon2. Never store or email a raw password.
- Rate limit login and any endpoint that sends email or costs money.
- Verify webhook signatures before trusting the payload.
- HTTPS everywhere. Cookies: httpOnly, secure, sameSite.
- Authorisation on the server: never trust a role sent from the browser.

*"Weeks of coding can save you hours of planning."*

# 29. AI Tools & Prompting

*Working with models instead of against them.*

## Anatomy of a prompt that works

- Role: who the model should act as.
- Task: one clear objective, not five.
- Context: the code, data or constraints it needs.
- Format: exactly how you want the answer (JSON, table, 5 bullets).
- Examples: one or two show more than a paragraph of description.

## Vocabulary

| | |
|---|---|
| **Token** | A chunk of text. Cost and limits are counted in tokens. |
| **Context window** | How much it can read at once. Old text falls out. |
| **Temperature** | 0 = predictable, 1 = creative. Code likes low. |
| **System prompt** | Persistent instructions that outrank the chat. |
| **Embedding** | Text turned into numbers so you can search by meaning. |
| **RAG** | Retrieve your own documents, then ask the model about them. |
| **Fine-tuning** | Train on your examples. Usually not needed - try RAG first. |
| **Hallucination** | Confident and wrong. Always verify facts and APIs. |

## Calling a model from code

```
const res = await fetch(API_URL, {
  method: "POST",
  headers: { "Content-Type": "application/json",
             Authorization: `Bearer ${process.env.API_KEY}` },
  body: JSON.stringify({
    model: MODEL,
    messages: [
      { role: "system", content: "You are a concise code reviewer." },
      { role: "user", content: userInput }
    ],
    temperature: 0.2
  })
});
```

## Using AI without getting worse

- Read every line before you accept it. If you cannot explain it, do not ship it.
- Ask for the reasoning, then the code. You learn more.
- Never paste secrets, private keys or client data into a prompt.
- AI is fastest at boilerplate and slowest at your specific domain rules.

# 30. Ship It: Workflow & Portfolio

*Turning practice into a career.*

## Project workflow that finishes things

- 1. Define the one problem and the one user. Write it in a sentence.
- 2. List features, then cut half. Version 1 is the smallest useful thing.
- 3. Sketch the screens on paper before opening the editor.
- 4. Build the ugly working version first. Style it after it works.
- 5. Test on a real phone and with a real person.
- 6. Deploy, write the README, then iterate.

## Deployment checklist

| | |
|---|---|
| **Env vars** | Set on the host, not in the repo. .env is gitignored. |
| **Build locally** | The production build must pass before you push. |
| **404 + error page** | Every app needs a friendly failure screen. |
| **Favicon + OG image** | First impression when the link is shared. |
| **Analytics** | You cannot improve what you do not measure. |
| **Custom domain + HTTPS** | Looks professional, costs almost nothing. |
| **Backups** | Anything in a database needs a backup plan. |

## README template

```
# Project name
One line about what it does and who it is for.

## Demo
Live: https://...    Video: https://...

## Features
- ...

## Tech
React, TypeScript, Tailwind, Postgres

## Run locally
npm install && npm run dev

## What I learned
Two or three honest sentences.
```

## Portfolio rules

- Three finished projects beat twelve half-built ones.
- Every project: a live link, the code, a screenshot, and the problem it solves.
- Explain your decisions, not just your stack. That is what hiring managers read.

*“Documentation is a love letter you write to your future confused self.”*

– Keep a changelog of what you learn each week. In a year it is a resume.
– Publish something publicly every month. Consistency beats talent.



*“Naming a variable temp is a promise you will never keep.”*

# PART II
# Shortcuts & AI

Ten extra chapters: every keyboard shortcut worth memorising, and a full working guide to the AI tools that actually save you hours.





*“Every CSS bug is one !important away from becoming three CSS bugs.”*

# 31. VS Code Shortcuts (Win / Mac)

*The keys that separate slow typing from real speed. Mac: Ctrl = Cmd unless stated.*

## Move around the code

| | |
|---|---|
| `Ctrl+P` | Go to any file by name. The single most used shortcut. |
| `Ctrl+Shift+P` | Command palette - run any command without the mouse. |
| `Ctrl+G` | Go to line number. |
| `Ctrl+Shift+O` | Jump to a symbol (function, class) inside the file. |
| `Ctrl+T` | Search a symbol across the whole project. |
| `F12 / Alt+F12` | Go to definition / peek definition inline. |
| `Shift+F12` | Find every place this thing is used. |
| `Ctrl+- / Ctrl+Shift+-` | Jump back / forward through your navigation history. |
| `Ctrl+Tab` | Cycle open editors, most recent first. |

## Edit like a pro

| | |
|---|---|
| `Alt+Up / Alt+Down` | Move the current line (or selection) up or down. |
| `Shift+Alt+Up / Down` | Duplicate the line above or below. |
| `Ctrl+Shift+K` | Delete the whole line. |
| `Ctrl+Enter` | Open a new line below without splitting the current one. |
| `Ctrl+D` | Select next occurrence of the word - press again for more. |
| `Ctrl+Shift+L` | Select every occurrence at once. |
| `Alt+Click` | Add another cursor anywhere. |
| `Ctrl+Alt+Up / Down` | Add a cursor on the line above / below. |
| `Ctrl+/` | Toggle line comment. Shift+Alt+A for a block comment. |
| `Shift+Alt+F` | Format the whole document with Prettier. |
| `F2` | Rename a symbol everywhere in the project safely. |
| `Ctrl+Space` | Force IntelliSense suggestions. |
| `Ctrl+.` | Quick fix: auto import, add missing prop, extract function. |

*“99 little bugs in the code... 127 little bugs in the code.”*

# 31. VS Code Shortcuts (continued)

*Panels, search, multi-file work and the settings that make them stick.*

## Search and replace

| | |
|---|---|
| **Ctrl+F / Ctrl+H** | Find / replace in the current file. |
| **Ctrl+Shift+F** | Search the whole project. |
| **Ctrl+Shift+H** | Replace across the whole project - preview before applying. |
| **Alt+R** | Toggle regex mode inside the find box. |
| **Alt+Enter** | Select every match of the current search at once. |

## Windows, panels and terminal

| | |
|---|---|
| **Ctrl+B** | Show / hide the sidebar - more room for code. |
| **Ctrl+`** | Toggle the integrated terminal. |
| **Ctrl+Shift+`** | Open a second terminal. |
| **Ctrl+** | Split the editor into two columns. |
| **Ctrl+1 / 2 / 3** | Focus editor group 1, 2 or 3. |
| **Ctrl+W / Ctrl+K W** | Close current editor / close all editors. |
| **Ctrl+Shift+T** | Reopen the editor you just closed by mistake. |
| **Ctrl+K Z** | Zen mode: only code on screen. Esc Esc to exit. |
| **Ctrl+Shift+E / G / D / X** | Explorer / Source control / Debug / Extensions. |
| **Ctrl+K Ctrl+S** | Open the keyboard shortcuts editor itself. |

## Debug keys

| | |
|---|---|
| **F5 / Shift+F5** | Start or continue debugging / stop. |
| **F9** | Toggle a breakpoint on this line. |
| **F10 / F11** | Step over / step into the function. |
| **Shift+F11** | Step out of the current function. |

## Make the editor faster than your brain

– Turn on **Format on Save**: settings, search "format on save", tick it. You never think about spacing again.
– Enable **Auto Save: onFocusChange** so the browser reload always shows real code.
– Learn only 5 new shortcuts a week. Print this page and stick it next to the screen.

*"console.log('here') is a legitimate debugger. Fight me."*

# 32. Browser & DevTools Shortcuts

*You spend half your day in the browser. Stop using the mouse there.*

## Browser navigation

| | |
|---|---|
| **Ctrl+T / Ctrl+W** | New tab / close tab. |
| **Ctrl+Shift+T** | Reopen the tab you just closed. |
| **Ctrl+1..8 / Ctrl+9** | Jump to tab number / jump to last tab. |
| **Ctrl+L** | Focus the address bar and select it. |
| **Ctrl+Shift+N** | Incognito window - a clean session with no cache or cookies. |
| **Ctrl+Shift+R** | Hard reload: ignores cache. Fixes "my CSS did not update". |
| **Ctrl+F / Ctrl+P** | Find on page / print (also how you export a page as PDF). |

## DevTools

| | |
|---|---|
| **F12 / Ctrl+Shift+I** | Open DevTools. |
| **Ctrl+Shift+C** | Inspect element: click anything on the page to select it. |
| **Ctrl+Shift+M** | Device toolbar - test mobile sizes instantly. |
| **Ctrl+Shift+J** | Jump straight to the Console. |
| **Esc** | Show the console as a drawer under any other panel. |
| **Ctrl+Shift+P in DevTools** | Command menu: "screenshot", "coverage", "disable cache". |
| **Ctrl+P in DevTools** | Open any source file the page loaded. |
| **Arrow keys in Elements** | Expand / collapse DOM nodes without clicking. |
| **Up / Down in Styles** | Nudge any CSS number by 1. Shift+arrow = 10, Alt+arrow = 0.1. |

## Console commands worth memorising

```
$0                    // the element currently selected in Elements
$('.card')            // querySelector shortcut
$$('.card')           // querySelectorAll -> real array
copy($0)              // copy the element HTML to your clipboard
console.table(users)  // arrays of objects as a readable table
console.time('t'); console.timeEnd('t')   // measure how long code takes
monitorEvents($0,'click')                 // log every click on that element
```

## Fast checks before you ship

– Network tab: sort by size. Any image over 300 KB needs compressing.
– Lighthouse tab: run the mobile audit; fix anything under 90 in Performance or SEO.
– Toggle **Rendering > Emulate prefers-color-scheme** to test dark mode in one click.

*"Git push --force is a lifestyle, not a command."*

# 33. Terminal, Git & OS Shortcuts

*Keys that work in bash, zsh and most terminals, plus the system keys people forget.*

## Terminal line editing

| | |
|---|---|
| **Ctrl+A / Ctrl+E** | Jump to start / end of the line. |
| **Ctrl+U / Ctrl+K** | Delete everything before / after the cursor. |
| **Ctrl+W** | Delete the previous word. |
| **Alt+B / Alt+F** | Move back / forward one word. |
| **Ctrl+R** | Reverse search your command history - type a few letters. |
| **Ctrl+L** | Clear the screen (keeps history). |
| **Ctrl+C / Ctrl+D** | Kill the running command / exit the shell. |
| **Ctrl+Z then bg / fg** | Pause a job, send it to background, bring it back. |
| **!! / !$** | Repeat last command / reuse its last argument. |

## Git aliases that save hundreds of keystrokes

```
git config --global alias.s  "status -sb"
git config --global alias.lg "log --oneline --graph --decorate -20"
git config --global alias.cm "commit -m"
git config --global alias.undo "reset --soft HEAD~1"
# usage:  git s    git lg    git cm "fix nav"    git undo
```

## Git emergencies

| | |
|---|---|
| **git restore** | Throw away uncommitted changes in one file. |
| **git reset --soft HEAD~1** | Undo last commit, keep the changes staged. |
| **git stash / stash pop** | Park your work, switch branch, bring it back. |
| **git reflog** | The time machine: every HEAD you ever had. Recovers "lost" commits. |
| **git commit --amend** | Fix the last commit message or add a forgotten file. |
| **git cherry-pick** | Copy one commit onto the current branch. |

## Operating system keys

| | |
|---|---|
| **Win+V / Cmd+Shift+V** | Clipboard history - paste something you copied 5 items ago. |
| **Win+Shift+S / Cmd+Shift+4** | Screenshot a region. Cmd+Shift+5 records video on Mac. |
| **Win+Left / Right** | Snap the window to half the screen. |
| **Alt+Tab / Cmd+Tab** | Switch apps. Hold and press again to go further back. |
| **Win+D / Cmd+F3** | Show the desktop instantly. |
| **Win+. / Cmd+Ctrl+Space** | Emoji picker inside any text field. |

*“Dark mode exists because bugs look scarier in the light.”*

# 34. Unity, Figma & Editing Shortcuts

*Game engine, design tool and video editor - the three other windows you live in.*

## Unity editor

| | |
|---|---|
| `Q W E R T Y` | Hand, Move, Rotate, Scale, Rect, Transform tools. |
| `F / Shift+F` | Frame the selected object / lock the camera to follow it. |
| `Ctrl+P / Ctrl+Shift+P` | Play / pause the game. Ctrl+Alt+P steps one frame. |
| `Ctrl+D / Ctrl+Shift+N` | Duplicate object / create empty GameObject. |
| `Alt+Click arrow` | Expand the whole hierarchy branch. |
| `Ctrl+Shift+F` | Snap the selected object to the scene view camera. |
| `V (hold)` | Vertex snap - drag a corner exactly onto another corner. |
| `Ctrl+6` | Open the Animation window. |

## Figma

| | |
|---|---|
| `R / O / T / F` | Rectangle, Ellipse, Text, Frame. |
| `Ctrl+G / Ctrl+Alt+G` | Group / wrap the selection in a frame. |
| `Shift+A` | Auto layout - the single most important Figma feature. |
| `Ctrl+D` | Duplicate. After a move, Ctrl+D repeats the exact offset. |
| `Alt+Drag` | Duplicate by dragging. Hold Shift too to keep it aligned. |
| `Ctrl+Alt+C / V` | Copy properties / paste properties onto another layer. |
| `Alt+Hover` | Measure the distance between two layers. |
| `Ctrl+Shift+K` | Place an image inside the selected shape. |

## Premiere & CapCut (video)

| | |
|---|---|
| `Space / L J K` | Play-pause / fast forward, rewind, stop. |
| `I / O` | Mark in point / out point on a clip. |
| `C / V` | Razor (cut) tool / selection tool. |
| `Ctrl+K` | Cut the clip exactly at the playhead. |
| `Shift+Delete` | Ripple delete - removes the gap too. |
| `Ctrl+M` | Export the sequence. |

## Photoshop quick keys

| | |
|---|---|
| `Ctrl+J` | Duplicate the current layer. |
| `Ctrl+Alt+G` | Clipping mask - the clean way to mask a photo into a shape. |
| `[ / ]` | Smaller / bigger brush. Shift+[ or ] changes hardness. |
| `Ctrl+T` | Free transform. Hold Shift for proportional resizing. |

*“Stack Overflow is the real senior developer on this project.”*

# 35. Build Your Own Shortcuts

*The real speed jump: shortcuts and snippets you define yourself.*

## Custom VS Code snippet (30 seconds to add)

Command palette > **Snippets: Configure Snippets** > pick the language, then:

```
{
  "React component": {
    "prefix": "rfc",
    "body": [
      "export function ${1:Name}() {",
      "  return (",
      "    <div className=\"$2\">$0</div>",
      "  );",
      "}"
    ],
    "description": "Function component"
  }
}
// type rfc + Tab -> full component, cursor already in the right place
```

## Rebind a key you actually want

```
// keybindings.json  (Ctrl+K Ctrl+S -> keyboard shortcuts -> {} icon)
[
  { "key": "ctrl+alt+t", "command": "workbench.action.terminal.new" },
  { "key": "ctrl+alt+r", "command": "workbench.action.reloadWindow" },
  { "key": "alt+s",      "command": "editor.action.formatDocument" }
]
```

## Shell aliases (add to ~/.bashrc or ~/.zshrc)

```
alias ..='cd ..'
alias ll='ls -lah'
alias dev='bun run dev'
alias gs='git status -sb'
alias serve='python3 -m http.server 5500'
mkcd() { mkdir -p "$1" && cd "$1"; }      # make a folder and enter it
port() { lsof -ti:"$1" | xargs kill -9; } # port 3000 -> kill whatever blocks it
```

## Rules for remembering shortcuts

– Pick **three** shortcuts, force yourself to use them for a week, then add three more.
– Every time you reach for the mouse twice for the same action, search the command palette for its key.
– Keep one text file called *keys.md* in your project and write the ones you keep forgetting.

*“I am not procrastinating, I am waiting for the build.”*

# 36. AI Tools Directory

*What each tool is genuinely best at, and the free way in. Pick one per job, not ten.*

## Chat and reasoning

| | |
|---|---|
| **ChatGPT** | All-round: explaining, drafting, code review, images, voice. Strong free tier. |
| **Claude** | Long documents and big code files, careful writing, refactors that keep your style. |
| **Gemini** | Free with a Google account, huge context, best when your data is in Docs or Drive. |
| **DeepSeek / Qwen** | Free strong coding and maths models; good fallback when limits hit. |
| **Perplexity** | Search with sources - use it when the answer must be current and citable. |

## Coding

| | |
|---|---|
| **GitHub Copilot** | Inline autocomplete inside VS Code. Free for verified students. |
| **Cursor / Windsurf** | Editors that change many files at once from one instruction. |
| **Lovable** | Describe an app in chat, get a working full-stack web app with a database. |
| **v0** | Turn a screenshot or a sentence into React + Tailwind UI code. |
| **Codeium / Continue** | Free autocomplete alternatives; Continue can run fully offline. |

## Images, video and audio

| | |
|---|---|
| **Midjourney / Ideogram** | Posters and thumbnails; Ideogram is the one that spells text right. |
| **Nano Banana / Flux** | Edit an existing photo with a sentence, keep the same face or product. |
| **Runway / Kling / Veo** | Turn a still image into a few seconds of moving video. |
| **ElevenLabs** | Voice-over that does not sound robotic; also dubbing into Arabic. |
| **Whisper** | Free speech to text, runs on your own machine, great for subtitles. |
| **Suno** | Background music for reels when you need something copyright-free. |

## Study, docs and productivity

| | |
|---|---|
| **NotebookLM** | Upload your PDFs and get answers grounded only in them, with citations. |
| **Napkin / Excalidraw** | Turn text into a diagram you can actually put in a report. |
| **Gamma** | First draft of a presentation in one prompt, then you fix the wording. |
| **Otter / Fireflies** | Meeting notes and action items without typing. |

## How to choose without wasting money

– Free tiers cover almost everything a student needs. Pay only when a tool saves you hours *every week*.
– Never paste passwords, API keys, exam material or client data into a chatbot.
– Tools change monthly. Learn the **skill** of prompting - it transfers to whatever wins next year.

*“Commit message: fixed stuff. Nobody remembers what stuff.”*

# 37. Prompting That Works

*Prompting is just clear specification. Here are patterns you can reuse today.*

## The five-part prompt

| | |
|---|---|
| **Role** | "You are a senior React reviewer." Sets vocabulary and depth. |
| **Task** | One sentence, one goal. Split anything bigger into separate prompts. |
| **Context** | Paste the real code, the real error, the real audience. No summaries. |
| **Constraints** | Language, length, libraries allowed, what to never do. |
| **Output format** | "Return only a table" / "only the changed file" / "JSON with keys x,y". |

## Copy these templates

```
DEBUG
Here is the error, the file and what I already tried.
Give me: (1) the most likely cause, (2) the exact fix, (3) how to verify it.
Do not rewrite unrelated code.

LEARN
Explain <topic> at three levels: one sentence, one paragraph, one worked example.
Then give me 5 questions to test myself, answers at the end.

REVIEW
Review this code for bugs, security and readability.
Output a table: line | issue | severity | fix. No compliments.

BUILD
Goal: <what the user should be able to do>.
Stack: <fixed>. Output: full files only, no placeholders, no TODOs.
Ask me questions first if anything is ambiguous.
```

## Techniques that measurably help

| | |
|---|---|
| **Give examples** | Show 2-3 input/output pairs. The single biggest quality jump. |
| **Ask for a plan first** | "Plan, then wait for my OK." Stops 300 wrong lines of code. |
| **Force the format** | Ask for JSON, a table or bullets and you get something usable directly. |
| **Iterate, do not restart** | "Keep everything, change only the header." Small diffs beat rewrites. |
| **Ask for trade-offs** | "Give 3 options with pros, cons and when to choose each." |
| **Make it self-critique** | "List 3 weaknesses in your answer, then fix them." |

## Traps

– It invents APIs, functions and citations with total confidence. Run the code before you trust it.
– Vague prompt = generic answer. If the reply is boring, your prompt was boring.
– Long chats drift. Start a fresh chat when the topic changes.

*“Semicolons: the tiny commas of regret.”*

# 38. AI in Your Code Editor

*How to use an AI pair programmer without it destroying your project.*

## Workflow that keeps you in control

| | |
|---|---|
| **1. Commit first** | Clean git status before any AI edit, so a bad result is one command from undone. |
| **2. Small tasks** | One feature, one file group, one prompt. Never "build the whole app" twice. |
| **3. Read the diff** | Review every changed line. If you cannot explain it, do not keep it. |
| **4. Run it** | Build, click the feature, check the console. Green output is not proof. |
| **5. Commit again** | Working state saved with a real message before the next prompt. |

## Prompts that work inside an editor

```
"Refactor this component into two: a presentational one and a container.
 Keep the same props API and the same class names."

"Add TypeScript types to this file. Do not change any runtime behaviour."

"Write vitest tests for this function: happy path, empty input, and the
 error branch. Use only the existing test helpers."

"This is slow with 5000 rows. Explain why before writing any code."
```

## Good tasks vs bad tasks for AI

| | |
|---|---|
| **Great** | Boilerplate, tests, regex, SQL, format conversion, docs, renaming, explaining errors. |
| **Fine with review** | New components, refactors, algorithms you can verify quickly. |
| **Risky** | Auth, payments, security rules, database migrations - anything touching money or user data. |
| **Never** | Pasting straight to production, or letting it invent secrets and keys. |

## Keep a project rules file

```
# AGENTS.md  (or .cursorrules)
Stack: React 19 + TanStack Start + Tailwind v4. Bun, not npm.
Style: no default exports, colours only from CSS tokens, no inline hex.
Testing: vitest. Every new util needs a test.
Never: install a new UI library, touch generated files, or use any.
```

*“The best error message is the one that never shows up. Mine writes novels.”*

# 39. Calling AI APIs

*The moment you put AI inside your own app. Same shape for almost every provider.*

## Minimal chat call (JavaScript, OpenAI-compatible)

```
const res = await fetch("https://api.provider.com/v1/chat/completions", {
  method: "POST",
  headers: {
    "Content-Type": "application/json",
    Authorization: `Bearer ${process.env.API_KEY}`,   // server side only!
  },
  body: JSON.stringify({
    model: "gpt-5-mini",
    messages: [
      { role: "system", content: "You reply in one short paragraph." },
      { role: "user", content: userText },
    ],
    temperature: 0.7,
  }),
});
const data = await res.json();
const answer = data.choices[0].message.content;
```

## Streaming so the user sees words appear

```
body: JSON.stringify({ model, messages, stream: true })

const reader = res.body.getReader();
const decoder = new TextDecoder();
while (true) {
  const { done, value } = await reader.read();
  if (done) break;
  for (const line of decoder.decode(value).split("\n")) {
    if (!line.startsWith("data: ")) continue;
    if (line === "data: [DONE]") break;
    const delta = JSON.parse(line.slice(6)).choices[0].delta.content;
    if (delta) setText((t) => t + delta);
  }
}
```

## Force structured output you can trust

```
messages: [{ role: "user", content: "Extract name, email, budget from: " + text }],
response_format: { type: "json_object" }

// then ALWAYS validate before using it:
const parsed = Schema.parse(JSON.parse(data.choices[0].message.content));
```

## Production rules

| | |
|---|---|
| **Key on the server** | A key in frontend code is a stolen key. Call it from a server function. |
| **Rate limit** | Per user and per IP, or one script drains your credits overnight. |
| **Cache** | Same question, same answer: cache by a hash of the prompt. |
| **Cap tokens** | max_tokens plus a max input length keeps cost predictable. |

*“Do not worry if it does not work. If it did, you would be out of a job.”*

| | |
|---|---|
| **Handle 429 / 5xx** | Retry with exponential backoff, then a friendly error message. |
| **Log usage** | Store tokens per request so you can see the cost of each feature. |

# 40. AI for Study, Content & Robots

*Where AI gives a student the biggest unfair advantage.*

## Study system that actually raises grades

| | |
|---|---|
| **Feynman loop** | "I will explain X to you. Correct me and mark what I got wrong." |
| **Active recall** | Ask for 20 questions from your notes, answers on a separate page. |
| **Spaced plan** | "Turn this syllabus into a 6-week plan, 1 hour a day, with review days." |
| **Past papers** | Paste a question, ask for the method - never just the final answer. |
| **SAT / exams** | Ask it to grade your written answer against the official rubric with a score. |

## Content and design

| | |
|---|---|
| **Hooks** | "30 first lines for a video about X, each under 8 words, no clickbait." |
| **Repurpose** | One long script becomes a reel, a carousel, a thread and a caption set. |
| **Thumbnails** | Generate 4 concepts, then rebuild the best one yourself so it stays on brand. |
| **Subtitles** | Whisper for the transcript, then ask AI to clean punctuation and translate. |

## Robotics and hardware

| | |
|---|---|
| **Datasheets** | Upload the PDF, ask for pinout, voltage limits and the wiring as a text diagram. |
| **Code translation** | "Rewrite this Arduino sketch for ESP32, keep the same pin names." |
| **Tuning** | Describe the oscillation, ask for the PID direction - then test on the real robot. |
| **Vision** | Run a small model on-device for line following or object detection. |

## A prompt pack you can steal

```
TEACH ME:  Teach <topic> as if I am 15 but smart. One analogy, one text
           diagram, one real example from code I would actually write.

FIX MY ESSAY: Improve clarity only. Keep my voice.
              Show a table: original | improved | why.

PROJECT IDEA: Give 5 project ideas using <my skills>, ordered by how
              impressive they look in a portfolio vs how long they take.

INTERVIEW:   Act as an interviewer for a junior <role>. One question at a
             time, wait for my answer, then rate it out of 10 with feedback.
```

## The rule that keeps you honest

– AI is a faster draft, never the final word. You verify, you test, you sign it.
– If you cannot rebuild it without AI, you have not learned it yet - redo it once by hand.

# PART III
# Depth: Engineering & Career

Twenty chapters that go past syntax: how computers actually run your code, how to choose a data structure, how to design a system that survives users, how to test and secure it, and how to turn all of that into money and a career while you are still in school.





*“AI will not replace developers. It will just co-write the bugs.”*

# 41. How a Computer Runs Your Code

*Source text to electricity. Knowing this makes every performance bug obvious.*

## The journey of one line of code

| | |
|---|---|
| **1. Source** | Text you wrote. Means nothing to the CPU. |
| **2. Parse** | Compiler builds an AST - a tree of what your code means. |
| **3. Compile / JIT** | C++ compiles ahead of time. JS is parsed, interpreted, then hot functions get JIT-compiled to machine code. |
| **4. Link** | Your code plus libraries become one runnable binary. |
| **5. Load** | OS maps the binary into memory, creates a process. |
| **6. Execute** | CPU fetches, decodes, executes instructions - billions per second. |

## Memory is not one thing

| | | |
|---|---|---|
| **Registers** | ~1 cycle | A few dozen slots inside the CPU. Fastest thing that exists. |
| **L1 / L2 / L3 cache** | 1-40 cycles | Small copies of recent memory. Why looping an array beats a linked list. |
| **RAM** | ~200 cycles | Your variables, heap objects, textures. 100x slower than cache. |
| **SSD** | ~100000 cycles | Files, databases. Never touch it in a game loop. |
| **Network** | millions | Another computer. Always async, always may fail. |

**The one lesson:** speed today is mostly about memory layout, not instruction count. An array of 1000 numbers beats a linked list of 1000 numbers even though both are O(n), because the array is one cache-friendly block.

## Stack vs heap

```
void demo() {
    int  a = 5;                 // stack: fast, freed automatically at }
    int* b = new int[1000];     // heap: slower, YOU must delete[] b
    std::vector<int> c(1000);   // heap data, stack owner - freed for you
}                               // a gone, c gone, b LEAKED
```

– Stack: automatic, tiny (~1-8 MB), dies when the function returns. Recursion too deep = stack overflow.

– Heap: manual or garbage-collected, big, fragmentable. Every leak lives here.

– In JS/Python/C# the GC frees the heap for you - but a reference you forgot to drop is still a leak.

## Why your app freezes

| | |
|---|---|
| **Blocking the main thread** | One 200 ms loop = a dropped frame and a frozen UI. Move it to a worker or split it. |
| **Garbage collection spike** | Allocating inside a loop makes the GC run mid-frame. Reuse objects. |
| **Layout thrash** | Reading a DOM size after writing a style forces the browser to recompute layout. Batch reads then writes. |
| **Sync disk / network** | Any await-less blocking IO. Always async. |

*“My code does not have bugs, it has undocumented surprise features.”*

# 42. Big-O Without the Maths Degree

*How the cost of your code grows when the data grows. That is the whole idea.*

## Read this table once, keep it forever

| | | |
|---|---|---|
| `O(1)` | constant | Array index, hash map get, push. Size does not matter. |
| `O(log n)` | halving | Binary search, balanced tree, DB index lookup. 1M items = 20 steps. |
| `O(n)` | one pass | Loop, filter, sum, max. The honest default. |
| `O(n log n)` | good sort | sort(), merge sort, quicksort average. Best possible comparison sort. |
| `O(n^2)` | nested loop | Loop inside a loop over the same data. Fine at 100, dead at 100000. |
| `O(2^n)` | explosion | Naive recursion over every subset. Only for tiny n. |

## What the numbers feel like

| | |
|---|---|
| `n = 1000` | O(n)=1 thousand steps. O(n^2)=1 million (still fast). O(2^n)=never finishes. |
| `n = 1000000` | O(n)=instant. O(n log n)=20 million, fine. O(n^2)=10^12, your laptop dies. |
| `Rule of thumb` | Modern CPU does roughly 10^8 simple steps per second in a scripting language, 10^9 in C++. |

## Spot the hidden O(n^2)

```
// LOOKS like one loop. It is not.
const dupes = items.filter(x => others.includes(x.id));   // includes = O(n) inside a loop

// FIX: build a Set once, then every lookup is O(1)
const ids = new Set(others);                              // O(n)
const dupes2 = items.filter(x => ids.has(x.id));          // O(n) total

// Same trap: array.indexOf, array.find, "SELECT ... " inside a for loop (N+1 query)
```

## Space complexity counts too

– A Set that turns O(n^2) time into O(n) costs O(n) memory. Almost always worth it.
– Streaming beats loading: process a 2 GB log line by line in O(1) memory instead of reading it all.
– Recursion uses O(depth) stack. Deep recursion on user input is a crash waiting to happen.

## How to actually reason about your own code

| | |
|---|---|
| `Count the loops` | Nested over the same n means multiply. Sequential means add, and you keep the biggest. |
| `Look inside calls` | sort() is n log n. includes/indexOf is n. Object spread in a loop is n^2. |
| `Drop constants` | O(3n + 500) is O(n). Constants matter in practice, not in the notation. |
| `Measure after` | Big-O predicts growth, not speed. Always confirm with a real timer on real data. |

*“I told my computer I needed a break. It said: Error 418, I am a teapot.”*

# 43. Data Structures: Pick the Right One

*Half of all performance problems are the wrong container.*

## The decision table

| | | |
|---|---|---|
| **Array / List** | index O(1), search O(n) | Ordered data you mostly read and append. Default choice. |
| **Hash map / dict** | get/set O(1) | Look something up by a key. Counting, caching, dedupe, joins. |
| **Set** | has O(1) | Membership and uniqueness. Visited nodes, tags, ids. |
| **Stack (LIFO)** | push/pop O(1) | Undo, DFS, matching brackets, call history. |
| **Queue (FIFO)** | push/shift O(1) | BFS, job queues, event loops, printer order. |
| **Priority queue** | O(log n) | Always take the smallest/biggest next: Dijkstra, A*, schedulers. |
| **Linked list** | insert O(1) | Only when you constantly splice in the middle and hold the node. |
| **Tree / BST** | O(log n) | Sorted data with range queries. Databases use B-trees for this. |
| **Graph** | depends | Anything with relationships: maps, social, dependency, robot paths. |
| **Trie** | O(word) | Autocomplete and prefix search over many words. |

## Counting, grouping, deduping - the three you will write weekly

```
// count
const counts = {};
for (const w of words) counts[w] = (counts[w] || 0) + 1;

// group by a key
const byUser = new Map();
for (const t of tasks) {
  if (!byUser.has(t.userId)) byUser.set(t.userId, []);
  byUser.get(t.userId).push(t);
}

// dedupe, keeping order
const seen = new Set();
const unique = items.filter(i => !seen.has(i.id) && seen.add(i.id));
```

## Python equivalents worth memorising

```
from collections import Counter, defaultdict, deque
import heapq

Counter(words).most_common(5)          # top 5 words
g = defaultdict(list); g[u].append(v)  # adjacency list, no key checks
q = deque([start]); q.popleft()        # O(1) queue for BFS
heapq.heappush(pq, (cost, node))       # priority queue
```

## A rule that saves hours

If your code searches a list inside a loop, you needed a map. If it keeps calling sort, you needed a heap. If it keeps checking "did I already do this", you needed a set.

*“Works 60% of the time, every time.”*

# 44. Algorithms You Will Actually Use

*Not competition tricks - the eight patterns that solve real product problems.*

## The pattern list

| | | |
|---|---|---|
| **Two pointers** | sorted arrays | Pairs that sum to X, removing duplicates, merging two lists. |
| **Sliding window** | streams | Longest streak, moving average, rate limiting, sensor smoothing. |
| **Binary search** | sorted / monotonic | Find a value, or find the smallest setting that still works. |
| **BFS** | shortest hops | Nearest node, friend degrees, grid pathing with equal cost. |
| **DFS + backtracking** | all options | Permutations, sudoku, maze filling, dependency cycles. |
| **Dijkstra / A*** | weighted map | Robot and game pathfinding with real distances and costs. |
| **Greedy** | local best | Interval scheduling, coin change with nice coins, Huffman. |
| **Dynamic programming** | overlapping subproblems | Edit distance, knapsack, best path in a grid. |

## BFS: the one you will use most

```
function shortestPath(graph, start, goal) {
  const q = [start], prev = new Map([[start, null]]);
  while (q.length) {
    const node = q.shift();
    if (node === goal) break;
    for (const next of graph[node] ?? []) {
      if (prev.has(next)) continue;          // already reached, ignore
      prev.set(next, node);
      q.push(next);
    }
  }
  if (!prev.has(goal)) return null;          // unreachable
  const path = []; for (let n = goal; n !== null; n = prev.get(n)) path.push(n);
  return path.reverse();
}
```

## Binary search on the answer - the underrated trick

```
# "What is the smallest number of workers that finishes in time?"
lo, hi = 1, 1000
while lo < hi:
    mid = (lo + hi) // 2
    if finishes_in_time(mid): hi = mid      # works, try smaller
    else:                     lo = mid + 1  # too few, need more
answer = lo
```

## Dynamic programming in two sentences

If the same subproblem is computed more than once, store it. That is memoisation (top-down) or a table (bottom-up), and it turns exponential into polynomial.

*“There is no cloud, it is just someone else laptop. Be nice to it.”*

```
memo = {}
def ways(n):                        # climb stairs, 1 or 2 steps
    if n < 0: return 0
    if n == 0: return 1
    if n in memo: return memo[n]
    memo[n] = ways(n-1) + ways(n-2)
    return memo[n]
```

# 45. Clean Code & Refactoring

*Code is read far more than it is written - mostly by future you at 2 a.m.*

## Naming rules that survive review

| | |
|---|---|
| **Say what, not how** | activeUsers beats filteredArray2. getPrice beats doStuff. |
| **Booleans read as claims** | isLoading, hasAccess, canEdit - never flag or status2. |
| **No mystery numbers** | MAX_RETRIES = 3 instead of a naked 3 sitting in an if. |
| **Same word, same meaning** | Pick fetch OR get OR load for the whole codebase and never mix. |
| **Length matches scope** | i in a 3-line loop is fine. i as a module-level variable is a crime. |

## Smells and the fix

| | | |
|---|---|---|
| **Long function** | does 4 things | Split by the comments you had to write. Each becomes a function. |
| **Boolean argument** | render(true) | Split into two clearly named functions. |
| **Deep nesting** | 5 levels of if | Guard clauses: return early on the invalid cases first. |
| **Duplicated logic** | copy-paste x3 | Extract only after the third copy - two is a coincidence. |
| **God component** | 600 lines JSX | Pull out sub-components and move data logic into a hook. |
| **Comment explaining what** | // add 1 to i | Delete it. Comments explain WHY, code explains what. |

## Guard clauses beat pyramids

```
// before
function save(user) {
  if (user) { if (user.email) { if (!user.banned) { db.save(user); } } }
}
// after
function save(user) {
  if (!user) return;
  if (!user.email) throw new Error("email required");
  if (user.banned) return;
  db.save(user);
}
```

## How to refactor without breaking everything

– Commit first. A clean git state is your undo button.

*“TODO: remove this before launch. Launched 2 years ago.”*

– Write a test that passes on the current behaviour, then refactor until it still passes.
– One transformation at a time: rename, then extract, then move. Never all three in one commit.
– Never refactor and add a feature in the same commit - if it breaks you cannot tell which half did it.

# 46. Software Architecture Patterns

*How to arrange files and layers so the project still moves at month six.*

## Layers, in one diagram

```
UI  (components, screens)      knows about ->  hooks/state
State / hooks                  knows about ->  services
Services (api, business rules) knows about ->  data access
Data access (db, fetch)        knows about ->  nothing above it

Arrows point ONE way. If data access imports a component, the design is broken.
```

## Patterns you will meet

| | | |
|---|---|---|
| **MVC / MVVM** | classic apps | Model = data, View = screen, Controller/ViewModel = the glue. |
| **Layered** | most web apps | UI, service, data. Simple and boring, which is good. |
| **Feature folders** | growing apps | Group by feature (auth/, orders/), not by type (components/, utils/). |
| **Repository** | swappable data | One module owns all DB access, so the rest never writes SQL. |
| **Event driven** | decoupling | Publish 'order.paid'; the email and stock code react on their own. |
| **Microservices** | large teams | Separate deployables. Buys team independence, costs enormous ops. |
| **Monolith first** | you, now | One app until it genuinely hurts. Splitting early is the classic mistake. |

## State management: choose by scope

| | |
|---|---|
| **Local useState** | Belongs to one component. 80% of state. Start here always. |
| **Lifted / props** | Two siblings need it. Move it to the closest common parent. |
| **Context** | Rarely changing global things: theme, current user, locale. |
| **Server cache** | Anything from an API. Use React Query - it is cache, not state. |
| **Global store** | Genuinely app-wide, frequently changing client state. The last resort. |

## The dependency rule in practice

– Business rules should be testable without a browser or a database.
– Keep framework code at the edges - a pure function is portable forever.
– If a file imports from five folders, it is doing five jobs.

*“It worked on my machine, so ship the machine.”*

# 47. Designing a Database That Scales

*Schema decisions are the hardest to reverse. Spend the hour up front.*

## Normalise, then denormalise on purpose

| | |
|---|---|
| **1NF** | No arrays or CSV in a column. One value per cell. |
| **2NF** | Every column depends on the whole primary key. |
| **3NF** | No column depends on another non-key column (store city_id, not city name everywhere). |
| **Then break it** | Copy a value on purpose (a cached comment_count) only when reads hurt - and keep it updated in a trigger. |

## Relationships

```
-- one to many: put the foreign key on the MANY side
create table posts (
  id uuid primary key default gen_random_uuid(),
  author_id uuid not null references users(id) on delete cascade,
  title text not null,
  created_at timestamptz not null default now()
);

-- many to many: a join table with a composite unique
create table post_tags (
  post_id uuid references posts(id) on delete cascade,
  tag_id  uuid references tags(id)  on delete cascade,
  primary key (post_id, tag_id)
);
```

## Indexes: the 10x button

| | | |
|---|---|---|
| **B-tree** | default | Equality and ranges: where status = 'x', order by created_at. |
| **Composite** | (a, b) | Only helps if your filter uses a, or a and b - never b alone. |
| **Partial** | where active | Small, fast index over the rows you actually query. |
| **GIN** | json / text search | Search inside JSONB or full text. |
| **Unique** | constraint | Correctness first, speed as a bonus. |

– Every foreign key you filter on deserves an index - Postgres does not create it for you.

– Indexes slow writes and use disk. Ten indexes on a hot table is a bug, not caution.

– Read the plan before guessing: `EXPLAIN ANALYZE select ...` and look for 'Seq Scan' on a big table.

## Kill the N+1 query

```
-- BAD: 1 query for posts, then 1 per post for the author  (101 round trips)
-- GOOD: one join
select p.*, u.name as author
from posts p join users u on u.id = p.author_id
order by p.created_at desc
limit 20;
```

## Rules that prevent 3 a.m. incidents

*“Real programmers count from 0 and apologise later.”*

| | |
|---|---|
| **Never store money as float** | Use integer cents or numeric(12,2). 0.1 + 0.2 is not 0.3. |
| **Always timestamptz in UTC** | Convert to local time in the UI, never in the table. |
| **Soft delete carefully** | deleted_at is fine, but then EVERY query must filter it. Use a view. |
| **Migrations only** | No manual clicking in a prod database. The schema lives in git. |
| **Backups you restored** | An untested backup is a rumour. |

# 48. System Design Interview Kit

*The same six steps answer 'design Twitter', 'design a URL shortener', or your own app.*

## The six steps

| | |
|---|---|
| **1. Requirements** | Functional (what it does) and non-functional (how many users, how fast, how consistent). |
| **2. Estimate** | Users x actions x size. 1M users x 10 posts/day x 1 KB = 10 GB/day. |
| **3. API** | List the endpoints first - it forces the data model out into the open. |
| **4. Data model** | Tables, keys, what is indexed, what is cached, what is stored as a blob. |
| **5. Diagram** | Client, CDN, load balancer, app servers, cache, DB, queue, storage. |
| **6. Bottleneck** | Name the first thing that breaks and how you scale it. This is the actual test. |

## The building blocks and what each solves

| | | |
|---|---|---|
| **Load balancer** | traffic | Spreads requests, removes dead servers, ends TLS. |
| **CDN** | latency | Serves images and JS from a city near the user. |
| **Cache (Redis)** | read load | Hot keys in memory. Always set a TTL and plan for a miss. |
| **Queue** | spikes | Accept now, process later: email, video encode, reports. |
| **Read replica** | read scale | Copies of the DB for reads. Slightly stale - fine for feeds. |
| **Sharding** | write scale | Split rows across databases by a key. The last thing you do. |
| **Object storage** | big files | S3/R2 for uploads. Never store images as DB bytes. |

## Worked example: a URL shortener

```
POST /links {url}   ->  {short: "aX9k2"}      GET /aX9k2 -> 302 redirect

id: base62 of an auto counter (short, no collisions) or 7 random chars + unique index
table links(code text primary key, url text, owner uuid, created_at, clicks bigint)

Read:write is about 100:1  ->  cache code->url in Redis, TTL 24h
Clicks: do NOT update the row per hit. Push to a queue, aggregate every minute.
Scale path: CDN edge redirect -> Redis -> DB replica -> shard by code prefix
```

## Trade-offs to say out loud

– CAP: during a network split you choose consistency or availability. A bank picks C, a feed picks A.

*"Programming is 10% writing code and 90% wondering why it works."*

– Strong vs eventual consistency: a like count can be 2 seconds late, a wallet balance cannot.
– Latency vs cost: every cache layer is money and a new source of stale bugs.

# 49. Testing: The Whole Pyramid

*Tests are not homework - they are how you change code without fear.*

## The pyramid

| | | |
|---|---|---|
| **Unit** | many, ms | One pure function. No network, no DB. Where 70% of your tests live. |
| **Integration** | some | Several modules together, real DB in a test container. |
| **E2E** | few, slow | A real browser doing the money path: sign up, buy, log out. |
| **Manual / exploratory** | always | You, trying to break it. Never fully replaceable. |

## Anatomy of a good test

```
import { describe, it, expect } from "vitest";
import { priceWithVat } from "./pricing";

describe("priceWithVat", () => {
  it("adds 14% and rounds to the nearest piastre", () => {
    expect(priceWithVat(10000)).toBe(11400);      // arrange, act, assert
  });
  it("throws on a negative price", () => {
    expect(() => priceWithVat(-1)).toThrow();     // the edge case IS the test
  });
});
```

## What to test, ranked

| | |
|---|---|
| **Money and auth** | Anything that charges, grants access, or deletes. Test first, always. |
| **Pure logic** | Formulas, parsers, validators, state machines. Cheapest tests, highest value. |
| **Bug regressions** | Every bug you fix gets a test so it can never come back. |
| **Happy path E2E** | One test that proves the product works end to end. |
| **Not: styling** | Do not snapshot-test CSS. It breaks daily and proves nothing. |

## Rules

– A test that needs the internet is not a unit test - mock the boundary, not your own code.
– Test behaviour, not implementation. Renaming a private function should not break a test.
– Flaky test = broken test. Fix it or delete it; a red build people ignore is worse than none.
– Coverage is a hint, not a goal. 100% coverage of getters proves nothing.

*“I have a joke about recursion, but first, I have a joke about recursion.”*

# 50. Security: Attacks and Defences

*You do not need to be a hacker. You need to stop the ten attacks everyone uses.*

## The top ten, and the actual fix

| | | |
|---|---|---|
| **SQL injection** | user text in SQL | Parameterised queries only. Never string-concatenate input. |
| **XSS** | user HTML rendered | Escape by default; never innerHTML raw input; set a CSP. |
| **CSRF** | other site posts as you | SameSite=Lax cookies + a CSRF token on state-changing routes. |
| **Broken access control** | you can edit /users/7 | Check ownership on the SERVER for every single request. |
| **Secrets in the client** | API key in JS | Any key in frontend code is public. Call it from the server. |
| **Weak auth** | plain passwords | bcrypt/argon2, rate limit logins, offer 2FA, expire sessions. |
| **IDOR** | guessable ids | UUIDs plus an ownership check - ids alone are never authorisation. |
| **Dependency holes** | old packages | npm audit, dependabot, and actually merge the updates. |
| **Over-sharing errors** | stack trace to user | Log the detail, return a generic message. |
| **No rate limit** | brute force / cost | Limit per IP and per user on login, signup, and AI endpoints. |

## Do it right, in code

```
// parameterised - the driver escapes it, you never do
const { rows } = await db.query("select * from users where email = $1", [email]);

// ownership check on the server, every time
const note = await db.note.find(id);
if (note.ownerId !== session.userId) return new Response("Not found", { status: 404 });

// hashing
const hash = await bcrypt.hash(password, 12);
const ok   = await bcrypt.compare(attempt, hash);
```

## Row Level Security mindset

If your database supports RLS, treat the database as the last line of defence: a policy that says "a user only sees rows where user_id = auth.uid()" survives even a bug in your API code.

## Personal security hygiene

| | |
|---|---|
| **.env is never committed** | One leaked key on GitHub is scraped within minutes. |
| **Rotate what leaked** | Deleting the commit does not delete the key from history or bots. |
| **Least privilege** | A service key that can only read is a service key that cannot destroy. |
| **HTTPS everywhere** | Free with any modern host. No excuse in 2026. |

*"Estimated: 2 hours. Actual: 2 hours and one existential crisis."*

# 51. Concurrency Without Fear

*Doing several things at once, and the four bugs that come with it.*

## Three different words

| | | |
|---|---|---|
| **Concurrency** | interleaving | Tasks take turns. One core is enough. JS async is this. |
| **Parallelism** | at the same time | Multiple cores really running together. Workers, threads. |
| **Async IO** | waiting smartly | The CPU does other work while the network answers. |

## JavaScript: the event loop in five lines

```
console.log("1");
setTimeout(() => console.log("4"), 0);     // macrotask, runs last
Promise.resolve().then(() => console.log("3"));  // microtask, before timers
console.log("2");
// output: 1 2 3 4   -- sync code first, then microtasks, then timers
```

## Patterns that matter

```
// run in parallel, not one by one   (3s -> 1s)
const [user, posts, tags] = await Promise.all([getUser(), getPosts(), getTags()]);

// tolerate partial failure
const results = await Promise.allSettled(jobs.map(run));

// timeout anything
const ctrl = new AbortController();
setTimeout(() => ctrl.abort(), 5000);
await fetch(url, { signal: ctrl.signal });

// limit how many run at once (do not open 500 sockets)
async function pool(items, n, fn) {
  const it = items[Symbol.iterator]();
  await Promise.all(Array.from({ length: n }, async () => {
    for (const item of it) await fn(item);
  }));
}
```

## The four classic bugs

| | |
|---|---|
| **Race condition** | Two writes, last one wins randomly. Fix: a transaction, or a version/updated_at check. |
| **Deadlock** | A waits for B, B waits for A. Fix: always take locks in the same order. |
| **Starvation** | One task never gets a turn. Fix: fair queue or priorities with ageing. |
| **Stale closure** | An async callback uses an old value of state. Fix: refs, or functional updates. |

## The safest concurrency is no shared state

Pass messages instead of sharing memory: web workers postMessage, queues between services, immutable updates in React. Nothing to lock means nothing to corrupt.

*“The definition of insanity: refreshing localhost expecting a different result.”*

# 52. Docker & Environments

*'It works on my machine' is a bug report about your setup, not your code.*

## The mental model

| | |
|---|---|
| **Image** | A frozen filesystem + command. Built once, identical everywhere. |
| **Container** | A running copy of an image. Disposable by design. |
| **Volume** | The only place data survives a container restart. |
| **Layer** | Each Dockerfile line is a cached layer. Order them cheap-to-expensive. |

## A Dockerfile you can reuse

```
FROM node:22-alpine AS build
WORKDIR /app
COPY package*.json ./          # copy manifests FIRST so installs stay cached
RUN npm ci
COPY . .
RUN npm run build

FROM node:22-alpine
WORKDIR /app
ENV NODE_ENV=production
COPY --from=build /app/dist ./dist
COPY --from=build /app/node_modules ./node_modules
EXPOSE 3000
CMD ["node", "dist/server.js"]
```

## Compose for local dev

```
services:
  app:
    build: .
    ports: ["3000:3000"]
    env_file: .env
    depends_on: [db]
  db:
    image: postgres:16
    environment: { POSTGRES_PASSWORD: dev }
    volumes: ["pgdata:/var/lib/postgresql/data"]
volumes: { pgdata: {} }
```

## Commands you will use daily

| | |
|---|---|
| **docker compose up -d** | Start everything in the background. |
| **docker compose logs -f app** | Follow one service's logs. |
| **docker exec -it app sh** | Get a shell inside a running container. |
| **docker compose down -v** | Stop and delete volumes - a clean slate. |
| **docker system prune -af** | Reclaim tens of gigabytes of dead images. |

## Environments and config

*“I do not always test my code, but when I do, I do it in production.”*

– Same image in dev, staging and prod. Only the environment variables change.
– Config comes from the environment, secrets from a secret store - never from the repo.
– A .env.example committed with empty values documents what the app needs.

# 53. CI/CD and Shipping Safely

*Small releases, automatic checks, and a way back. That is the whole discipline.*

## A pipeline that catches real problems

```
name: ci
on: { push: { branches: [main] }, pull_request: {} }
jobs:
  check:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: oven-sh/setup-bun@v2
      - run: bun install --frozen-lockfile
      - run: bun run lint
      - run: bunx tsc --noEmit
      - run: bun run test
      - run: bun run build        # a broken build must never reach main
```

## Release strategies

| | | |
|---|---|---|
| **Rolling** | default | Replace instances a few at a time. Simple, brief mixed versions. |
| **Blue / green** | safe | Two full environments, flip the router. Instant rollback. |
| **Canary** | risky change | 5% of traffic first, watch errors, then widen. |
| **Feature flag** | best | Ship the code dark, turn it on for yourself, then for everyone. |

## Migrations without downtime

| | |
|---|---|
| **Expand** | Add the new nullable column. Old code ignores it. |
| **Backfill** | Fill it in batches, not one giant UPDATE that locks the table. |
| **Migrate** | Deploy code that writes both old and new. |
| **Contract** | Once nothing reads the old column, drop it - in a later release. |

## Git workflow that scales from 1 to 10 people

– main is always deployable. Branch per change, small PRs, merge daily.
– Conventional commits (feat:, fix:, chore:) give you a free changelog.
– Tag releases: v1.4.0. Knowing exactly what is in production is half of debugging.
– Rollback plan before deploy: previous image tag ready, migration reversible.

# 54. Observability: Logs, Metrics, Traces

*In production you cannot attach a debugger. You only have what you recorded.*

## The three pillars

| | | |
|---|---|---|
| **Logs** | what happened | Discrete events with context. Structured JSON, not sentences. |
| **Metrics** | how much | Numbers over time: requests/s, p95 latency, error rate, queue depth. |
| **Traces** | where the time went | One request followed across services with timing per step. |

## Log like an engineer

```
// bad
console.log("error!", err);

// good - structured, searchable, no secrets
logger.error({
  event: "checkout_failed",
  userId: user.id,
  orderId: order.id,
  amount: order.total,
  reason: err.code,
  requestId,               // the same id on every log of this request
}, "checkout failed");
```

| | |
|---|---|
| **debug** | Loud detail, off in production. |
| **info** | Business events: user signed up, order paid. |
| **warn** | Recovered from something odd - retry succeeded, cache missed badly. |
| **error** | A request failed. Someone should be able to act on it. |
| **Never log** | Passwords, tokens, card numbers, full personal data. |

## The four golden signals to alert on

| | |
|---|---|
| **Latency** | p95 and p99, not the average. The average hides every angry user. |
| **Traffic** | Requests per second - context for everything else. |
| **Errors** | Rate, not count. 50 errors in 5M requests is fine; in 500 it is on fire. |
| **Saturation** | CPU, memory, connection pool, queue depth. The thing that runs out first. |

## Alerting rules that keep you sane

– Alert on symptoms users feel (checkout error rate), not on causes (CPU at 80%).
– Every alert must have an action. If nobody does anything, delete the alert.
– Track a request id from the browser through every server hop - it turns a mystery into a search.

*“My favourite language is coffee, it compiles instantly.”*

# 55. Math You Need for AI

*Less than you fear, but you cannot skip it. Here is exactly what earns its place.*

## Linear algebra: data is a matrix

| | | |
|---|---|---|
| **Vector** | [1, 3, 5] | One example, or one word embedding. Direction and length. |
| **Matrix** | rows x cols | A whole dataset, or the weights of one layer. |
| **Dot product** | a.b | Similarity. Big when two vectors point the same way. All of search. |
| **Matmul** | X @ W | One neural layer in one operation. Shapes must line up: (n,d)@(d,k)=(n,k). |
| **Norm** | ||v|| | Length. Normalise before comparing so scale does not lie to you. |
| **Transpose** | X.T | Flip rows and columns - usually to fix a shape error. |

## Cosine similarity - the formula behind every 'related items'

```
import numpy as np
def cosine(a, b):
    return float(a @ b / (np.linalg.norm(a) * np.linalg.norm(b)))
# 1.0 = same direction, 0 = unrelated, -1 = opposite
# This one line powers semantic search, recommendations and RAG.
```

## Calculus: only the gradient

– A derivative answers: if I nudge this weight up, does the error go up or down, and how fast?
– Gradient descent: `w = w - lr * dError/dw`, repeated. That is training.
– Learning rate too big = it explodes; too small = it never arrives. Start at 1e-3.
– The chain rule is how the error travels backwards through layers - that is backpropagation.

## Probability and statistics

| | |
|---|---|
| **Mean / median** | Median when there are outliers. Salaries and latencies are never normal. |
| **Std deviation** | Spread. 'Accuracy 90% +/- 8%' is a very different claim from '90% +/- 0.5%'. |
| **Distribution** | Normal, uniform, long tail. Web traffic is long tail, always. |
| **Bayes** | P(A\|B) = P(B\|A)P(A)/P(B). Why a 99% accurate test for a rare disease still misleads. |
| **Sampling bias** | Your model learns your data's prejudices. Check who is missing from the set. |

*"A programmer stops at a red light and waits for the callback."*

# 56. Machine Learning From Zero

*What is really happening, in the order you should learn it.*

## The three families

| | | |
|---|---|---|
| **Supervised** | labelled data | Predict a label or number: spam, price, disease. 90% of real work. |
| **Unsupervised** | no labels | Find structure: clustering customers, anomaly detection, embeddings. |
| **Reinforcement** | reward | Learn by trying: game agents, robot control. Powerful, data hungry. |

## The full pipeline (this is the job)

| | |
|---|---|
| **1. Frame it** | What exactly do you predict, and what decision changes because of it? |
| **2. Data** | Collect, clean, label. This is 70% of the time and nobody warns you. |
| **3. Split** | Train / validation / test. NEVER look at test until the very end. |
| **4. Baseline** | Predict the average, or a 5-line rule. Beat that before touching a neural net. |
| **5. Model** | Start simple: linear, then trees/boosting, then deep learning if needed. |
| **6. Evaluate** | Right metric for the problem, not just accuracy. |
| **7. Deploy + watch** | Real data drifts. A model that was 92% is 78% a year later. |

## Metrics: accuracy lies

| | | |
|---|---|---|
| **Accuracy** | balanced classes | 99% useless if 99% of emails are not spam. |
| **Precision** | of the flagged, how many were right | Use when a false alarm is expensive. |
| **Recall** | of the real ones, how many did I catch | Use when a miss is dangerous - disease, fraud. |
| **F1** | both | Harmonic mean when you need balance. |
| **MAE / RMSE** | numbers | Average error size. RMSE punishes big misses harder. |

## A first real model in 12 lines

```
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import classification_report

X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.2, stratify=y, random_state=42)
model = RandomForestClassifier(n_estimators=300, random_state=42).fit(X_tr, y_tr)
print(classification_report(y_te, model.predict(X_te)))
for name, score in sorted(zip(X.columns, model.feature_importances_), key=lambda p: -p[1])[
:10]:
    print(f"{name:22s} {score:.3f}")     # what the model actually leans on
```

## Overfitting, said simply

Your model memorised the answers instead of learning the pattern: near-perfect on training data, poor on new data. Fixes: more data, fewer parameters, regularisation, dropout, early stopping, cross-validation.

*"The app is fast, the internet is slow, blame the internet."*

# 57. Computer Vision for Robots

*Turning a camera into a sensor your robot can act on.*

## The pipeline

| | |
|---|---|
| **1. Capture** | Fix resolution and FPS. 640x480 at 30 fps beats 4K at 4 fps for control. |
| **2. Pre-process** | Blur to kill noise, convert to HSV for colour, threshold to a mask. |
| **3. Detect** | Contours, edges, ArUco markers, or a small neural net. |
| **4. Measure** | Centre, area, angle - numbers the controller can use. |
| **5. Decide** | Feed the error into PID or a state machine. Vision without control is a demo. |

## Line following in OpenCV

```
import cv2, numpy as np
cap = cv2.VideoCapture(0)
while True:
    ok, frame = cap.read()
    if not ok: break
    roi  = frame[300:480, :]                          # only look at the floor ahead
    gray = cv2.cvtColor(roi, cv2.COLOR_BGR2GRAY)
    blur = cv2.GaussianBlur(gray, (7, 7), 0)
    _, mask = cv2.threshold(blur, 0, 255, cv2.THRESH_BINARY_INV | cv2.THRESH_OTSU)
    M = cv2.moments(mask)
    if M["m00"] > 0:
        cx    = int(M["m10"] / M["m00"])              # centre of the black line
        error = cx - roi.shape[1] // 2                # + means line is right of centre
        steer(error)                                  # PID lives here
```

## Colour tracking: use HSV, never RGB

```
hsv  = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)
mask = cv2.inRange(hsv, (0, 120, 70), (10, 255, 255))     # red, low hue band
mask = cv2.morphologyEx(mask, cv2.MORPH_OPEN, np.ones((5,5), np.uint8))
cnts, _ = cv2.findContours(mask, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
if cnts:
    x, y, w, h = cv2.boundingRect(max(cnts, key=cv2.contourArea))
```

## Practical truths nobody tells beginners

| | |
|---|---|
| **Lighting is the algorithm** | Half of vision failures are a lamp. Add a hood or your own LED. |
| **Calibrate thresholds live** | Build a trackbar UI once; you will reuse it in every project. |
| **Crop before you compute** | A region of interest is the cheapest 5x speed-up. |
| **Measure your FPS** | A control loop under 10 Hz will oscillate no matter how good the vision is. |
| **Edge models** | YOLO-nano / MobileNet on a Pi 5 or Jetson. Quantise to int8 for 2-4x speed. |

*"Merge conflicts: where two good ideas go to fight."*

# 58. 3D Math & Shaders for Games

*The maths that makes movement feel right and pixels look expensive.*

## Vectors are the language

| | | |
|---|---|---|
| **`a - b`** | direction | From b to a. Normalise it to get a pure direction. |
| **`dot(a,b)`** | how aligned | >0 in front, 0 perpendicular, <0 behind. Field of view checks. |
| **`cross(a,b)`** | perpendicular | Surface normals, and which side you turned. |
| **`magnitude`** | distance | Compare sqrMagnitude instead - skipping the square root is free speed. |
| **`lerp(a,b,t)`** | blend | Smooth movement, colour fades, camera follow. |
| **`Quaternion`** | rotation | Use them. Euler angles gimbal-lock and fight you at 90 degrees. |

## Frame-rate independent movement

```
// WRONG - faster on a 144 Hz monitor
transform.position += velocity;

// RIGHT
transform.position += velocity * Time.deltaTime;

// smooth follow that behaves at any frame rate
cam.position = Vector3.Lerp(cam.position, target.position,
                            1f - Mathf.Exp(-speed * Time.deltaTime));
```

## Game feel: the cheap tricks that read as polish

| | |
|---|---|
| **`Coyote time`** | Let the jump work ~0.1s after leaving the ledge. Players never notice, they just feel good. |
| **`Input buffer`** | Remember a jump pressed 0.1s before landing and fire it on land. |
| **`Screen shake`** | Tiny, short, and scaled to impact. 0.15s is plenty. |
| **`Squash and stretch`** | Scale on land and launch. Two lines, enormous payoff. |
| **`Hit stop`** | Freeze 40 ms on impact. The single strongest 'weight' trick. |

## Your first shader (HLSL idea, any engine)

```
// fragment: a rim light that makes objects pop off the background
float rim = 1.0 - saturate(dot(normalize(viewDir), normalize(normal)));
rim = pow(rim, 3.0);                        // tighten the edge
float3 col = baseColor + rimColor * rim;    // add, do not multiply, for a glow
```

## Performance rules in a renderer

- Draw calls kill frame rate: batch static meshes, atlas your textures, share materials.
- Overdraw is invisible cost: transparent particles stacked ten deep will melt a phone.
- Do work per object, not per pixel, whenever the maths allows it.
- Profile before optimising. In games the bottleneck is almost never where it feels.

*“There are 2 hard things in tech: naming things, cache invalidation, and off-by-one errors.”*

# 59. Freelancing, Pricing & Clients

*How to turn the skills in this book into income while still at school.*

## Pricing models

| | | |
|---|---|---|
| **Hourly** | unclear scope | Safe for you, scary for the client. Good for maintenance and consulting. |
| **Fixed price** | clear scope | Higher profit if you scope well. Requires a written spec. |
| **Value based** | business impact | Charge a slice of what it earns or saves. The most profitable, needs trust. |
| **Retainer** | ongoing | A monthly fee for updates and support. Predictable income - aim for this. |

## Never skip these five contract lines

| | |
|---|---|
| **Deliverables** | Exactly what pages, features and files they receive. |
| **Revisions** | Two rounds included; extra rounds billed at X per hour. |
| **Payment** | 40% upfront, 30% at review, 30% before final delivery. |
| **Timeline** | Your dates depend on their feedback within N days. |
| **Ownership** | Code and design transfer on final payment, not before. |

## Scope creep, handled politely

```
"Happy to add that. It is outside the agreed scope, so here are two options:

 1. Add it now  - +3 days, +X EGP, new delivery date <date>.
 2. Phase two   - we ship what we agreed on <date>, then start this next week.

Which do you prefer?"

-- You never said no. You made the cost visible. This one script saves projects.
```

## Finding the first five clients

- Build 3 real things and publish them. A portfolio beats a CV at every age.
- Solve a visible problem for a local business (school, gym, clinic) for a small fee, then ask for a referral.
- Post the work publicly - a build thread on TikTok or LinkedIn is free marketing that compounds.
- Referrals are 80% of steady freelance work: always ask 'do you know someone else who needs this?'

## Red flags to walk away from

| | |
|---|---|
| **'Work free for exposure'** | Exposure does not pay for hosting. |
| **No upfront payment** | A client who will not pay 40% will not pay 100%. |
| **'Just a small change' x10** | No written scope means unlimited work at a fixed price. |
| **Vague decision maker** | If you cannot reach whoever approves it, nothing ever gets approved. |

*“First rule of CSS: the div is centered when it says it is.”*

# 60. Learning Fast & Building a Career

*The compounding part. Skill is a schedule, not a talent.*

## How to learn a hard thing (the loop)

| | |
|---|---|
| **1. Narrow it** | 'Learn AI' fails. 'Train a model that classifies my 500 photos' finishes. |
| **2. Skim then build** | One overview video, then build immediately. Tutorials without building are entertainment. |
| **3. Struggle first** | Try before you look it up. The failed attempt is what makes the answer stick. |
| **4. Explain it** | Write a short post or teach a friend. Gaps in your understanding appear instantly. |
| **5. Space it** | Revisit after 1 day, 1 week, 1 month. Spacing beats hours. |
| **6. Ship it** | A finished small thing teaches more than an unfinished ambitious thing. |

## Deep work beats long work

– Two 90-minute focused blocks beat eight distracted hours. Phone in another room, notifications off.
– Start the session by writing the one sentence you want true when it ends.
– Stop mid-task, not at the end - the unfinished line pulls you back tomorrow.

## Build a portfolio that gets replies

| | | |
|---|---|---|
| **Three projects** | not ten | Depth reads as ability; a list of clones reads as a tutorial history. |
| **Problem first** | in the README | What it solves, a GIF of it working, then the stack. |
| **Live link** | always | A deployed URL is worth more than the source. Include both. |
| **Your role** | be honest | 'I built the pathfinding and the API' beats a vague 'team project'. |
| **Write-up** | 300 words | The hard bug and how you solved it. This is what hiring people read. |

## A 12-month plan you can actually run

| | |
|---|---|
| **Months 1-3** | Foundations: one language deeply, git, the terminal, one finished project. |
| **Months 4-6** | Depth: data structures, a real database, tests. Second project, deployed. |
| **Months 7-9** | Specialise: web, robotics, or games. Contribute to one open source repo. |
| **Months 10-12** | Prove it: a portfolio site, three write-ups, one paid or competition project. |

## The five habits that outperform talent

– Finish things. An unfinished project teaches you the easy 20% only.
– Read other people's code weekly - it is the fastest way to see what good looks like.
– Keep a bug journal: symptom, cause, fix. You will re-meet 80% of your bugs.
– Teach publicly. Explaining is the highest-return study method that also builds your name.
– Be the person who replies, delivers on the date, and says what went wrong early. That reputation outlives every framework in this book.

*"Refactoring: making the same bug prettier."*

```
End of Part III.  40 -> 60 chapters.
Built by Mohamed Nasser  |  moonasser.site  |  keep it, print it, use it.
```



*“Deleting node_modules is the modern form of meditation.”*

# PART IV
# The Projects: Case Studies

Eleven real projects I designed and built - four of them live on the internet right now, the rest in active development or built for a competition. For each one: the problem it solves, who it is for, how it is put together, the hardest part, and what I would do differently. Read this part as a template. Every project you build should be describable this way in ten lines.





*"The rubber duck reviewed my PR and approved it."*

# 61. Mousa SAT

*mousasat.com - a full SAT learning platform, not a page of PDFs.*


**ONE OF MY PROJECTS**
**Mousa SAT** - designed, built and shipped by Mohamed Nasser.

| | |
|---|---|
| `Status` | Live |
| `Type` | Education platform (web app) |
| `Stack` | React + Vite front end, component library, auth and database backend, AI helpers |

## The problem

Most SAT prep in Egypt is a folder of PDFs and a WhatsApp group. Students cannot see progress, cannot practise under timed conditions, and teachers cannot tell who is struggling until it is too late.

## Who it is for

High-school students preparing for the SAT, and the teacher or centre running the course with them.

## How it is built

– Course pages hold structured lessons instead of loose files, so a student always knows what comes next.

– A mock exam engine runs timed, sectioned tests that behave like the real thing - timer, review flags, and a score at the end.

– A student dashboard turns raw attempts into progress: accuracy per topic, weak areas first.

– AI features explain a wrong answer in plain language instead of only marking it red.

– Everything is one deployable web app - no installs, works on the phone a student already owns.

## The hardest part

The scoring and analytics layer. Storing an answer is easy; turning thousands of answers into one honest sentence like 'your algebra is fine, your reading pace is not' is where the real design work lives.

## What is next

Deeper question tagging so the dashboard can recommend the next exercise automatically, and offline-friendly practice.

## What it taught me

Build the data model before the screens. Every dashboard I wanted later was only possible because attempts were stored per question, not per exam.



# 62. RealSatPrep

*realsatprep.com - digital SAT practice with a serious question bank.*

| | |
|---|---|
|  | **ONE OF MY PROJECTS**<br>**RealSatPrep** - designed, built and shipped by Mohamed Nasser. |

| | |
|---|---|
| `Status` | Live |
| `Type` | Education platform (web app) |
| `Stack` | React front end, database-backed question bank, analytics layer |

## The problem

The SAT went digital, but most practice material did not. Students train on paper and then meet a different interface on exam day.

## Who it is for

Students taking the Digital SAT who want repetition with feedback rather than one long test a month.

## How it is built

– A question bank organised by section, topic and difficulty, so practice can be narrow on purpose.
– Practice mode for learning (instant explanation) and mock mode for pressure (timed, no help).
– Analytics that compare attempts over time, so improvement is visible instead of felt.
– An admin path for adding questions without touching code.

## The hardest part

Keeping the question bank clean. One badly tagged question quietly poisons every statistic built on top of it, so validation had to be stricter than the UI.

## What is next

Adaptive practice sets that pick the next question from your own error pattern.

## What it taught me

Content is a feature. A beautiful app with 40 questions loses to a plain app with 2,000 good ones.



# 63. Sideredu

*sideredu.com - your next step in learning.*


**ONE OF MY PROJECTS**
**Sideredu** - designed, built and shipped by Mohamed Nasser.

| | |
|---|---|
| **Status** | Live |
| **Type** | Education platform |
| **Stack** | Vite + React, Tailwind, modular course structure |

## The problem

Learners collect scattered courses from five different sites and finish none of them. There is no single place that holds the path from beginner to something finished.

## Who it is for

Students who want a guided route through a subject rather than a search results page.

## How it is built

– Courses are modelled as ordered steps, so a learner always has exactly one next action.
– Reusable lesson components mean adding a new course is content work, not development work.
– A layout that reads well on a phone first, because that is where most of the studying actually happens.

## The hardest part

Deciding what to leave out. The first version tried to be a marketplace, a course player and a community at once; cutting it down to the learning path is what made it usable.

## What is next

Progress tracking per learner and certificates at the end of a path.

## What it taught me

Scope is a design tool. The feature I deleted saved the product.



# 64. MOO Teams

*mooteams.com - the studio site for MOO Studio.*


**ONE OF MY PROJECTS**
**MOO Teams** - designed, built and shipped by Mohamed Nasser.

| | |
|---|---|
| **Status** | Live |
| **Type** | Brand and studio website |
| **Stack** | React, motion, custom design system |

## The problem

A creative studio is judged in three seconds. A generic template says nothing about the people behind it.

## Who it is for

Clients looking for design, editing and development work, and future collaborators.

## How it is built

– A custom visual language - typography, colour and motion chosen for this brand only.
– Work is shown as outcomes, not as a gallery of screenshots with no context.
– Fast loading and clean structure so the site itself is a demo of the quality on sale.

## The hardest part

Motion discipline. Animation that impresses on the first visit and annoys on the third is a bug, not a feature.

## What is next

Case-study pages with real numbers behind each project.

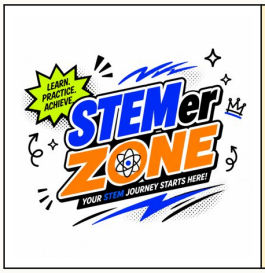
## What it taught me

For a brand site, restraint reads as confidence. Distinctive beats decorated.



# 65. STEMer Zone Platform

*A bigger alternative to the STEM learning sites we had to use.*




**ONE OF MY PROJECTS**
**STEMer Zone Platform** - designed, built and shipped by Mohamed Nasser.

| | |
|---|---|
| `Status` | In development |
| `Type` | Education platform |
| `Stack` | React front end, database backend, role-based accounts |

## The problem

STEM students in Egypt jump between a site for notes, a group for announcements, and a drive folder for past papers. Nothing is connected and nothing is searchable.

## Who it is for

STEM school students, and the seniors and teachers who want to publish resources to them.

## How it is built

– One library for notes, past papers and guides, searchable by subject and year.
– Accounts with roles, so a contributor can publish and a student can only read.
– Structure designed so the archive keeps its value as each year graduates.

## The hardest part

Permissions. The moment more than one person can publish, you need to answer who can edit, who can delete and who can undo - before launch, not after.

## What is next

Contribution workflow with review, so quality stays high as the library grows.

## What it taught me

Multi-user features are mostly rules, not screens. Decide the rules first.



# 66. VisionTrack & Eagle Eye

*Two computer-vision projects: one to find people, one to track objects.*



**ONE OF MY PROJECTS**

**VisionTrack & Eagle Eye** - designed, built and shipped by Mohamed Nasser.

| | |
|---|---|
| **Status** | VisionTrack: demo \| Eagle Eye: ITC 2026 Egypt Edition |
| **Type** | AI / computer vision |
| **Stack** | Python, OpenCV-style pipeline, detection and tracking models |

## The problem

VisionTrack: families searching for a missing person rely on shared photos and luck. Eagle Eye: detection systems that see an object in one frame lose it in the next.

## Who it is for

VisionTrack: search teams and volunteers. Eagle Eye: the ITC competition brief for smart detection and precision tracking.

## How it is built

– A frame pipeline: capture, pre-process, detect, then match detections across frames into a single tracked identity.

– Detection answers 'is it there'; tracking answers 'is it the same one'. They are separate problems and separate code.

– Confidence thresholds tuned per use case - a missed person is a worse error than a false alarm.

– Evaluation on recorded clips, so a change can be proven better instead of feeling better.



## The hardest part

Real footage. Models behave beautifully on clean datasets and fall apart on shaky phone video in bad light, which is exactly the footage that matters.

## What is next

VisionTrack: move from demo to a privacy-safe pilot. Eagle Eye: tighten tracking stability at distance.

## What it taught me

In AI projects, the dataset and the evaluation are the project. The model is the easy part.

*“Every CSS bug is one !important away from becoming three CSS bugs.”*

# 67. DroneRoute AI

*AI-powered routing and navigation for smart aerial missions - ITC 2026 Egypt Edition.*


**ONE OF MY PROJECTS**
**DroneRoute AI** - designed, built and shipped by Mohamed Nasser.

| | |
|---|---|
| **Status** | In development (competition project) |
| **Type** | AI / robotics |
| **Stack** | Python, pathfinding and optimisation, telemetry data |

## The problem

A drone mission is not a straight line. Battery, wind, no-fly zones and multiple stops turn 'go there' into a real optimisation problem, and most hobby setups ignore all of it.

## Who it is for

Aerial mission planning: survey, delivery and inspection scenarios in the competition brief.

## How it is built

– The map becomes a graph: waypoints as nodes, legs as edges weighted by distance, energy and risk.
– Pathfinding chooses the route; the cost function is where the intelligence actually lives.
– Constraints such as battery reserve and restricted zones are hard limits, not preferences.
– Re-planning mid-mission when conditions change, instead of following a plan that is no longer true.

## The hardest part

Writing an honest cost function. The shortest route and the safest route are rarely the same one, and the weights between them are a design decision, not a formula.

## What is next

Live telemetry feedback so the route updates from real battery drain rather than an estimate.

## What it taught me

Chapters 43 and 44 of this book stopped being theory here. Graphs and A* are not interview trivia; they are the product.



*“99 little bugs in the code... 127 little bugs in the code.”*

# 68. MedAtlas

*Find the medicine you need, at a pharmacy that actually has it.*

| | |
|---|---|
| MedAtlas | **ONE OF MY PROJECTS**<br>**MedAtlas** - designed, built and shipped by Mohamed Nasser. |

| | |
|---|---|
| `Status` | In development |
| `Type` | Maps + data platform |
| `Stack` | React front end, geolocation and mapping, database with pharmacy stock |

## The problem

You get a prescription, then call six pharmacies. Availability data exists in each pharmacy's head and nowhere else.

## Who it is for

Patients and families searching for a specific medicine nearby, especially in an emergency.

## How it is built

– Search by medicine, not by shop - the question the user actually has.
– Geolocation ranks results by real travel distance, not by who paid for a listing.
– Map view for orientation, list view for deciding, one tap to call or route there.
– Data model built so a pharmacy can update stock quickly, because stale data is worse than no data.

## The hardest part

Trust. A map is only useful if the availability is current, so the whole design pushes toward fast, low-effort updates from the pharmacy side.

## What is next

Pharmacy accounts for self-service stock updates and alerts when a medicine comes back in stock.

## What it taught me

The hardest part of a data product is not the app, it is convincing the data to stay fresh.



# 69. Step Education

*Step by step to your future.*


**ONE OF MY PROJECTS**
**Step Education** - designed, built and shipped by Mohamed Nasser.

| | |
|---|---|
| `Status` | In development |
| `Type` | Education platform |
| `Stack` | React, structured curriculum data, student accounts |

## The problem

Students are told to 'learn to code' or 'get into engineering' with no visible staircase between where they are and where that leads.

## Who it is for

Students who know the goal but not the order of the steps.

## How it is built

– Each track is an explicit ordered path with a checkpoint at the end of every step.
– Progress is stored per student, so returning after two weeks resumes instead of restarts.
– Content and platform are separated, so new tracks do not need new code.

## The hardest part

Sequencing. Deciding what genuinely must come before what, instead of copying a syllabus, took longer than building the pages.

## What is next

Mentor feedback on checkpoints, and a certificate at the end of a track.

## What it taught me

Teaching forces clarity. You cannot order steps you do not truly understand.



# 70. Voyager

*NASA Space Apps hackathon - a hub for space news, observing guides and learning.*


**ONE OF MY PROJECTS**
**Voyager** - designed, built and shipped by Mohamed Nasser.

| | |
|---|---|
| **Status** | Hackathon project |
| **Type** | Web app / community platform |
| **Stack** | React + Vite, external data sources, content-driven pages |

## The problem

Space content is either a press release you cannot read or a forum thread you cannot verify. A curious beginner has nowhere friendly to start.

## Who it is for

Students, hobby astronomers and anyone who looked up and had a question.

## How it is built

– News, observing guides and learning resources in one place with a consistent, readable layout.
– Guides written for a first-time observer: what to look at tonight, from where, with what.
– Built inside a hackathon window, so scope was fixed on day one and defended after that.

## The hardest part

Time. A hackathon is a scope exercise disguised as a coding exercise - the finished small idea beats the ambitious half-built one.

## What is next

More guides, and a simple community layer for sharing observations.

## What it taught me

A deadline is a design constraint. Decide the one thing that must work before you write any code.



*“Dark mode exists because bugs look scarier in the light.”*

# 71. What All of Them Taught Me

*The patterns that repeated across every project in this part.*

## The seven rules I now start with

| | |
|---|---|
| **Model first** | Screens change weekly; the data model is expensive to change. Design it before the UI. |
| **One user, one job** | Every screen should answer one question. If it answers three, it answers none. |
| **Ship the spine** | Get the smallest end-to-end path working - input, storage, output - then widen it. |
| **Fresh beats fancy** | In any data product, stale correct-looking data destroys trust faster than an ugly layout. |
| **Measure it** | 'Feels faster' is not a result. Record the number before and after. |
| **Write it down** | The README you write at the start is the spec; the one you write at the end is the portfolio. |
| **Finish** | A finished small project has taught me more than every abandoned ambitious one combined. |

## How I scope a new idea in 20 minutes

| | | |
|---|---|---|
| **1. One sentence** | the problem | If it needs a paragraph, the idea is still two ideas. |
| **2. One user** | be specific | 'Students' is not a user. 'A Grade 11 student two months before the SAT' is. |
| **3. Three screens** | maximum | Anything past three screens is version two. |
| **4. The spine** | end to end | Name the single path that must work, and build only that first. |
| **5. Done means** | written down | Define the sentence that will be true when you stop. |

## The mistakes I stopped making

– Designing the admin panel before anyone was using the product.

– Adding a second feature before the first one had a real user.

– Choosing a new framework because the project was boring, not because it was better.

– Waiting for the design to be perfect before deploying - a live ugly version teaches more than a perfect local one.

– Not writing down why a decision was made, and then re-arguing it with myself two months later.

```
End of Part IV.  61 -> 71 chapters.
Built by Mohamed Nasser  |  moonasser.site  |  the projects are real, go break them.
```

*“Stack Overflow is the real senior developer on this project.”*

# PART V
# Paths, Specialisations & AI Tools

Part V is the map. Every path in tech, what the job actually looks like day to day, what you must learn in what order, how long it realistically takes, and how to tell early whether it fits you. Then the second half: the AI tools that changed how all of this work is done, and how to use them without letting them use you.





*“I am not procrastinating, I am waiting for the build.”*

# 72. How to Choose a Path

*Stop collecting tutorials. Pick a direction for 90 days and judge it by evidence.*

## The four questions

| | |
|---|---|
| **What do you enjoy?** | Not the job title - the actual daily activity. Pixel-perfect layouts, data puzzles, hardware, or maths? |
| **What is the feedback loop?** | Frontend gives you a result in seconds. ML gives you one in hours. Pick the loop your patience can survive. |
| **What is the market?** | Look at real job posts in your country, count how many mention each skill. Reality beats YouTube trends. |
| **What can you ship alone?** | The best first path is the one where one person can finish something real. |

## The 90-day test

| | | |
|---|---|---|
| **Days 1-14** | fundamentals | One course, one language, no switching. Finish it. |
| **Days 15-45** | clone something | Rebuild an existing product badly. Badly is fine, finished is not optional. |
| **Days 46-75** | build original | Your own small idea, end to end, deployed publicly. |
| **Days 76-90** | judge | Did you open the editor voluntarily on a bad day? That is the whole test. |

## The honest truth about switching

Switching paths in your first two years costs you almost nothing - the fundamentals transfer. Switching every three weeks costs you everything, because you never reach the part where it gets interesting. Commit in 90-day blocks, not forever.

## What every path shares

– Git, the terminal and reading error messages carefully.
– One language you know deeply enough to debug at 2am.
– How the internet works: DNS, HTTP, requests, responses, status codes.
– Writing: issues, READMEs and commit messages that a stranger can follow.
– Finishing. The rarest skill in this entire book.

*“Commit message: fixed stuff. Nobody remembers what stuff.”*

# 73. Frontend Development

*Everything the user actually touches.*

## What the job actually is

You turn designs and requirements into interfaces that work on every screen, load fast, stay accessible and do not break when data is missing. Most of your time is state, edge cases and layout - not choosing colours.

## The learning order

| | |
|---|---|
| **HTML + CSS** | Semantics, flexbox, grid, responsive layout. Build three real pages from screenshots before touching a framework. |
| **JavaScript** | The language itself: arrays, objects, async, fetch, modules. Do not skip to React early. |
| **One framework** | React is the safest bet. Components, props, state, effects, lists, forms. |
| **Routing + data** | Client routing, fetching, loading and error states, caching. |
| **Styling system** | Tailwind or CSS modules, plus design tokens so themes are not hardcoded. |
| **Quality** | Accessibility, Lighthouse, images, bundle size, keyboard navigation. |

## Tools you will live in

VS Code, Chrome DevTools, Figma, Vite, npm, Git, and a deploy target you can push to in one command.

## Realistic timeline

6-10 months of consistent work to be genuinely hireable as a junior, if you finish three real deployed projects rather than twelve tutorials.

## It fits you if

– You notice when spacing is wrong on someone else's website.
– You like fast feedback - save the file, see the result.
– You enjoy making something feel smooth rather than just work.

## Skip this path if

You dislike design details and only want to think in data structures and algorithms - backend or ML will suit you better.

## Your first real project

Rebuild a product you use daily - one page, real data from a public API, deployed, responsive, keyboard accessible, and fast on 3G.



*“Semicolons: the tiny commas of regret.”*

# 74. Backend Development

*The part nobody sees until it breaks.*

## What the job actually is

You design data models, write APIs, handle authentication, keep data consistent, and make sure the system survives more users than it was built for. Correctness and security matter more than looks.

## The learning order

| | |
|---|---|
| **One language** | JavaScript/TypeScript, Python, C# or Go. Pick one, learn it properly. |
| **HTTP + REST** | Methods, status codes, headers, idempotency, versioning. |
| **Databases** | SQL first: schemas, joins, indexes, transactions. NoSQL later, when you can explain why. |
| **Auth** | Sessions vs tokens, hashing, roles, and row-level access rules. |
| **Reliability** | Validation, error handling, logging, retries, background jobs. |
| **Deployment** | Environments, secrets, migrations, monitoring, backups. |

## Tools you will live in

A terminal, a database client, Postman or similar, Docker eventually, and logs you actually read.

## Realistic timeline

8-12 months to junior level. Backend rewards depth, so it starts slower and ages better.

## It fits you if

– You would rather design the data model than the screen.
– You enjoy tracing a problem through layers until you find the real cause.
– You care about what happens when two users click at the same millisecond.

## Skip this path if

You need visual feedback to stay motivated - pair backend with frontend work so you can see your API in use.

## Your first real project

Build an API with accounts, permissions and one real business rule - then try to break your own security, and write down what you found.



*"The best error message is the one that never shows up. Mine writes novels."*

# 75. Full-Stack & the T-Shape

*Wide enough to build alone, deep enough to be trusted.*

## What full-stack really means

Full-stack does not mean expert at everything. It means you can carry a feature from idea to production without waiting for anyone. The professional version of this is the T-shape: broad working knowledge across the stack, and one vertical bar of genuine depth.

## Building your T

| | | |
|---|---|---|
| **The bar** | one deep skill | The thing people call you for. Choose it after a year of exposure, not on day one. |
| **The top** | broad literacy | Enough of every layer to talk to specialists and unblock yourself. |
| **The glue** | product sense | Knowing which features are worth building at all. |
| **The risk** | shallow everywhere | If nothing is deep, you are replaceable by the next tool release. |

## A realistic full-stack stack in 2026

| | |
|---|---|
| **Frontend** | React with a typed router and a data-fetching layer, Tailwind for styling. |
| **Backend** | Server functions or a small API, plus Postgres with row-level security. |
| **Auth** | A managed provider - never hand-rolled password storage. |
| **AI layer** | One gateway for text, images and embeddings, called only from the server. |
| **Deploy** | Push to main, preview URL, one-click production. |

## How to grow into it

– Start on the side you enjoy, then force yourself to build the other half of one project.
– Own one feature end to end every month - schema, API, UI, deploy, monitoring.
– Read other people's pull requests in the layer you are weakest in.
– Write the README as if you will be hit by a bus - future you is a different developer.

# 76. Mobile Development

*A computer in a pocket with a battery and a bad network.*

## What the job actually is

You build apps that must survive interruptions, offline moments, small screens and app-store review. Constraints are the job: memory, battery, permissions, and platform rules you do not control.

## The learning order

| | |
|---|---|
| **Choose native or cross** | Swift/Kotlin for platform depth, React Native or Flutter for one codebase. |
| **UI + navigation** | Stacks, tabs, gestures, safe areas, and platform conventions. |
| **State + storage** | Local database, caching, offline-first sync. |
| **Device APIs** | Camera, location, notifications, permissions, background tasks. |
| **Release** | Signing, store listings, review rules, staged rollout, crash reporting. |

*“Do not worry if it does not work. If it did, you would be out of a job.”*

### Tools you will live in

Xcode or Android Studio, a real device (simulators lie about performance), and a crash-reporting service from day one.

### Realistic timeline

8-12 months, with extra time for the release process, which is a skill of its own.

### It fits you if

– You care about feel: animation, gesture, response under the thumb.
– You like working inside hard constraints instead of resenting them.
– You want your work installed on people's home screens.

### Skip this path if

You want the fastest possible iteration loop - the build, sign and review cycle will frustrate you.

### Your first real project

Ship one small app to a store, end to end. The store submission teaches more than the code.

## 77. Game Development

*Systems, feel and maths, disguised as fun.*

### What the job actually is

You build interactive systems: input, physics, state machines, animation, audio, UI, and the loop that ties them together. Most of the work is tuning until something feels right, then optimising until it runs.

### The learning order

| | |
|---|---|
| **`Engine basics`** | Unity with C#: scenes, prefabs, components, the update loop. |
| **`Maths you need`** | Vectors, transforms, interpolation, delta time, basic trigonometry. |
| **`Core systems`** | Input, collision, state machines, save/load, scene flow. |
| **`Polish`** | Animation curves, particles, audio cues, camera shake, game feel. |
| **`Ship it`** | Builds, platform targets, performance profiling, a public playable link. |

### Tools you will live in

Unity, Rider or VS Code, a profiler, free asset packs, and a notebook full of ideas you will not build yet.

### Realistic timeline

12+ months to job level. Games are the most competitive path and pay least at entry - but the skills transfer everywhere.

### It fits you if

– You lose hours tuning a jump until it feels correct.
– You enjoy maths when it visibly moves something on screen.
– You are comfortable with long projects that show nothing for weeks.

### Skip this path if

You need steady income quickly. Build games as a portfolio and passion track, and earn from web or app work while you do.

### Your first real project

One complete tiny game: one mechanic, three levels, a menu, sound, and a public build link. Complete beats ambitious.

## 78. Robotics & Embedded

*Where the code meets a physical world that does not care about your assumptions.*

### What the job actually is

You write firmware and control logic for hardware: sensors, motors, timing, power. Debugging involves a multimeter as often as a debugger, and physics gets a vote on every design decision.

### The learning order

| | |
|---|---|
| **C/C++** | Pointers, memory, structs, bit operations - no garbage collector to save you. |
| **Electronics** | Voltage, current, resistors, motor drivers, reading a datasheet. |
| **Microcontrollers** | Arduino or ESP32 first, then STM32 or a real-time OS. |
| **Sensors + control** | Filtering noisy input, PID control, calibration. |
| **Systems** | Serial protocols, ROS basics, simulation before hardware, then real hardware. |

### Tools you will live in

A soldering iron, a multimeter, a logic analyser eventually, PlatformIO or Arduino IDE, and a lot of spare parts.

### Realistic timeline

12-18 months, and it is cheaper to start via competitions and university labs than by buying everything yourself.

### It fits you if

– You want your code to move something in the real world.
– You are patient with problems that are mechanical, not logical.
– You enjoy competitions and building under a hard deadline.

### Skip this path if

You dislike hardware costs, waiting for parts, and bugs that turn out to be a loose wire.

### Your first real project

An autonomous line-follower or obstacle-avoider, then rebuild it once with clean, modular, documented code.



# 79. Cybersecurity

*Think like the person trying to break what you built.*

## What the job actually is

You find weaknesses before attackers do, or you defend systems while they are being attacked. Either side demands deep understanding of how systems actually work, not just how they are supposed to work.

## The learning order

| | |
|---|---|
| **Fundamentals first** | Networking, operating systems, and at least one programming language. There are no shortcuts here. |
| **Web security** | The OWASP top ten, injection, auth flaws, access control, hands-on in a lab. |
| **Tooling** | Burp Suite, nmap, Wireshark, a Linux environment you configured yourself. |
| **Practice legally** | CTFs and intentionally vulnerable labs. Never a system you do not own. |
| **Defence side** | Logging, monitoring, incident response, hardening, least privilege. |

## Tools you will live in

A Linux VM, a lab environment, CTF platforms, and detailed notes - security careers are built on written write-ups.

## Realistic timeline

12-24 months, and most roles expect development or systems experience first. It is rarely a genuine first job.

## It fits you if

– You enjoy reading documentation for the edge case nobody intended.
– You are ethical by instinct and comfortable with strict rules.
– You like puzzles with no guarantee of a solution.

## Skip this path if

You want to build features rather than break them - security still makes you a better builder, so learn the basics anyway.

## Your first real project

Take one of your own apps, attack it properly, and publish a write-up of every flaw you found and how you fixed it.



*“My code does not have bugs, it has undocumented surprise features.”*

# 80. AI & Machine Learning

*Two very different jobs sharing one buzzword.*

## What the job actually is

There is the researcher/ML engineer who trains and evaluates models, and the AI application engineer who builds products on top of existing models. The second is far more accessible and is where most 2026 jobs actually are.

## The learning order

| | |
|---|---|
| **`Python`** | Fluently. Plus numpy and pandas for anything data-shaped. |
| **`The maths`** | Linear algebra, probability, statistics - needed for training, less for application work. |
| **`Classic ML`** | Train, validate, test, overfitting, metrics that are not accuracy. |
| **`Deep learning`** | PyTorch, neural network basics, transfer learning on a small dataset. |
| **`Applied AI`** | APIs, prompting, structured output, embeddings, retrieval, evaluation. |
| **`Production`** | Cost, latency, caching, fallbacks, and measuring quality over time. |

## Tools you will live in

Python, notebooks, PyTorch, a vector store, an AI gateway, and an evaluation set you wrote yourself.

## Realistic timeline

Applied AI: 6-9 months if you already build software. Research-level ML: years, usually with a degree behind it.

## It fits you if

– You like ambiguous problems with no single correct answer.
– You are comfortable with results that are statistical rather than exact.
– You enjoy measurement and are honest when numbers disagree with you.

## Skip this path if

You need deterministic behaviour and hate uncertainty - stay on the systems side and consume AI as a service.

## Your first real project

Build a retrieval-based assistant over documents you own, with a written evaluation set of 30 questions and measured accuracy before and after each change.



*“I told my computer I needed a break. It said: Error 418, I am a teapot.”*

# 81. The AI Tool Landscape

*What exists, what each category is genuinely good at, and what it costs you.*

## The categories

| | |
|---|---|
| **Chat assistants** | General reasoning, explaining, planning, drafting. Your rubber duck that answers back. |
| **In-editor completion** | Line and block suggestions while you type. Highest daily time saving, lowest risk. |
| **Agentic builders** | Take a task, edit multiple files, run commands. Powerful, and needs review discipline. |
| **Image generation** | Concepts, placeholder art, icons, marketing visuals, mood boards. |
| **Video & audio** | Editing assistance, subtitles, voice-over, background removal, upscaling. |
| **Voice models** | Speech to text for notes and interviews, text to speech for narration. |
| **Embeddings + search** | Semantic search over your own documents, the backbone of every real AI feature. |
| **Research tools** | Summarising papers and documentation, always verified against the original. |

## Choosing between them

| | | |
|---|---|---|
| **Task is fuzzy** | chat assistant | Explaining, planning, comparing approaches. |
| **Task is typing** | editor completion | Boilerplate, tests, repetitive transformations. |
| **Task is multi-file** | agentic tool | Only with a clean git state and a diff review. |
| **Task is knowledge** | embeddings | When the answer lives in your own documents. |
| **Task is visual** | image/video model | Drafts and placeholders, rarely final client work. |

## Cost thinking

– Every AI call has a price. Cache anything you would ask twice.

– Cheap fast models handle classification and extraction; save expensive models for reasoning.

– Set a hard spending limit before you launch anything public, not after.

– Measure quality with your own test set - a more expensive model is not automatically better for your task.

*“Works 60% of the time, every time.”*

# 82. Prompting Like an Engineer

*A prompt is a specification. Vague specs produce vague software.*

## The five-part prompt

| | |
|---|---|
| **Role** | Who the model should be and what standards apply. |
| **Context** | The real code, data or constraints - paste them, do not describe them. |
| **Task** | One clear objective, phrased as a verb. |
| **Format** | Exactly what shape you want back: JSON schema, table, diff, bullet list. |
| **Constraints** | What not to do, what to preserve, and what counts as done. |

## Techniques that actually change output quality

| | | |
|---|---|---|
| **Give examples** | few-shot | Two good examples beat two paragraphs of description. |
| **Ask for a plan** | then execute | Review the plan before letting it write code. |
| **Force structure** | schema output | Structured output turns text into data you can trust. |
| **Split the task** | chain steps | Extract, then transform, then format - not all at once. |
| **Ask for doubt** | list unknowns | Requesting assumptions surfaces hallucinations early. |

## Prompt patterns worth memorising

```
Review:   "Review this diff for correctness, security and edge cases.
           List issues by severity. Do not rewrite the code."

Explain:  "Explain this function line by line to someone who knows
           JavaScript but not this codebase. End with the 3 riskiest lines."

Debug:    "Here is the error, the stack trace and the file.
           Give 3 likely causes ranked by probability, and how to test each."

Extract:  "Return JSON matching this schema. No prose. If a field is
           unknown, use null - never invent a value."
```

## Anti-patterns

– "Make it better" - better by which measure? Name the metric.
– Pasting an error with no code, then arguing with the guesses that follow.
– Letting a long chat accumulate contradictions; restart clean instead.
– Accepting the first answer when the second prompt would have doubled the quality.

*“There is no cloud, it is just someone else laptop. Be nice to it.”*

# 83. AI in Your Daily Workflow

*Where it earns its place in a real working day.*

## A realistic day

| | | |
|---|---|---|
| **Planning** | chat | Turn a vague request into a task list and name the unknowns. |
| **Scaffolding** | agent/editor | Generate the boring 70%: types, forms, CRUD, tests. |
| **Debugging** | chat | Paste trace plus code, ask for ranked causes, verify each yourself. |
| **Review** | chat | Second opinion on your own diff before a human sees it. |
| **Writing** | chat | README, commit messages, release notes, client emails. |
| **Learning** | chat | Explain an unfamiliar library, then read the real docs to confirm. |
| **Media** | image/video | Thumbnails, placeholders, subtitles, quick cuts. |

## Rules that keep you improving instead of decaying

– Never merge code you cannot explain aloud, line by line.
– Write the hard part yourself first; ask afterwards for a critique.
– Once a week, solve something with the tools closed. That is your skill check.
– Keep secrets, client data and private code out of any tool you do not control.
– Treat every generated fact as a claim to verify, not an answer to trust.

## Using AI inside your own products

| | |
|---|---|
| **Server only** | API keys never touch browser code. Every model call goes through your server. |
| **Validate output** | Parse against a schema. Never render raw model output as trusted content. |
| **Handle failure** | Rate limits, credit limits and outages need visible, honest UI states. |
| **Cache** | Identical inputs should not be paid for twice. |
| **Evaluate** | Keep a fixed test set and re-run it whenever you change model or prompt. |

# 84. Where AI Fails You

*The failure modes that cost real time, told honestly.*

## The six failures

| | |
|---|---|
| **Confident invention** | Fabricated functions, options and citations - delivered in the same tone as truth. |
| **Stale knowledge** | Yesterday's API. Always check the version in the actual documentation. |
| **Missing context** | It cannot see your other files, your data shape, or last month's decision. |
| **Plausible wrong architecture** | Code that runs today and becomes unmaintainable in two months. |
| **Silent security holes** | Missing access checks, unvalidated input, secrets in the wrong place. |
| **Skill decay** | The slow one. If you never struggle, you never build the intuition. |

*“TODO: remove this before launch. Launched 2 years ago.”*

### Guardrails

– Clean git state before any agentic run, and read the diff like a stranger wrote it.
– Tests around anything AI touches that involves money, auth or user data.
– Verify every API against official documentation, not against the model's memory.
– Ask a second time in a fresh session - inconsistency between answers is a red flag.
– If you cannot review it, you are not qualified to ship it yet. Learn that part first.

### The honest summary

AI is the best junior collaborator you will ever have and the worst senior engineer. It types faster than you, knows more surface facts than you, and has no judgement, no memory of your product, and no accountability. Judgement stays your job - and judgement is the only part of this career that never gets automated.

# 85. A 12-Month Plan

*One year, four quarters, no wasted motion.*

### The plan

| | | |
|---|---|---|
| **Q1** | fundamentals | One language, Git, terminal, HTTP. Finish one course fully. Build two small things. |
| **Q2** | specialise | Pick a path from this part. Clone a real product. Deploy it publicly. |
| **Q3** | build original | Your own project with real users, however few. Add AI where it genuinely helps. |
| **Q4** | prove it | Portfolio, write-ups, open source contributions, interviews or freelance work. |

### Weekly rhythm that survives school or a job

| | |
|---|---|
| **10-15 hours** | Enough to progress. Consistency beats intensity every single time. |
| **60% building** | Projects, not tutorials. Tutorials are the warm-up, not the workout. |
| **20% fundamentals** | The theory that makes the next project faster. |
| **10% reading** | Other people's code, docs and post-mortems. |
| **10% writing** | Publish what you learned. It compounds harder than any other habit. |

### How to know it is working

– You debug problems you have never seen before without panicking.
– You read documentation before searching for a tutorial.
– You can explain a past decision and what you would change now.
– You have shipped something a stranger has used.
– You are bored by problems that stopped you six months ago.

```
End of Part V.  72 -> 85 chapters.
Pick one path. Give it 90 days. Ship something public.
Mohamed Nasser  |  moonasser.site
```

*"It worked on my machine, so ship the machine."*