

How I build with Claude

Ahmed Alzoubi · Computer engineer

The setup, tools and engineering checklists behind the projects marked with the Claude Code badge on ahmed-alzoubi.pages.dev.

September 2026

Contents

1. How I work
2. Standing instructions
3. Toolkit
4. Appendix A · Software Engineering Master Checklist
5. Appendix B · Embedded Systems, PCB & Electronics Master Checklist

How I work

I'm an embedded engineer. For software I work with **Claude Code**, Anthropic's AI coding agent. It writes most of the code. I decide what gets built, how it should look and behave, and what it has to pass before it ships.

The setup has four parts:

1. **A standing brief.** The same instructions load into every Claude session: which tool to use for which job, what needs my approval first, and which checklist applies. See section 2.
2. **The right tool for each job.** Skills, plugins and MCP servers that give Claude specialist know-how, or direct access to tools like KiCad, a browser or Cloudflare. See section 3.
3. **Engineering checklists.** Before planning, and again before calling anything done, the work is checked against the software or the embedded master checklist, scaled to the project. Items that don't apply are skipped with a stated reason. See appendix A and appendix B.
4. **Project memory.** An Obsidian vault holds decisions, commands, paths and lessons learned. Claude reads the relevant notes before it starts and updates them when it finishes, so the next session picks up where the last one stopped.

Who does what

	Me	Claude Code
Requirements and priorities	Set them	Asks questions, flags gaps
Design direction	Choose, reject, give feedback	Proposes options and drafts
Code	Try the result and send changes. I don't review the code line by line.	Writes it
Checks	Set the bar (the checklists)	Runs them, reports what was skipped and why
Accounts, money, publishing	Approve every time	Waits for approval

Example: this portfolio

- **Planning:** the Impeccable design skill interviewed me about audience and goals, then offered design directions. I turned down the first three and picked "modern and colourful".
- **Building:** plain HTML, CSS and JavaScript generated from one content file, so English and Arabic stay in sync. Ponytail and the Karpathy guidelines kept the code small.
- **Checking:** against the software checklist: security headers and a strict content policy, accessibility, both themes, phones, SEO. A check script fails the build on broken links, missing Arabic text or unsafe inline code. An independent review agent scored the design, and its fixes went in before publishing.
- **Feedback:** friends who are software engineers reviewed it. Their notes (less text, more icons, no flashing colours) became the next round of changes.

Standing instructions

This block is loaded into every session: the Claude app's profile preferences and Claude Code's global `CLAUDE.md`.

```
## Tools: pick the best one for each job
Before starting a task, check which skills, plugins, MCP servers, connectors
and tools are available, and use the ones that fit the job. My current setup
is documented in my Obsidian vault; read the relevant notes there first.

Typical matches, as examples only:
- Visual design, social posts, branding assets → Canva (or Figma for UI)
- Frontend, UI/UX, websites → impeccable (run `init` once per project),
  motion-web for animated/creative sites
- Writing code → ponytail + karpathy-guidelines (simplest code that works)
- Text, website copy, posts → humanizer
- ESP32 / firmware → ESP32-AI-Agent-Skill, esp-expert
- PCB / KiCad → the kicad MCP server
- SEO → searchfit-seo; launch videos → brag / brag-slim

These are not a fixed rulebook. If a task needs none of them, don't force one.
If a better-suited tool exists (installed or not), use it, and install it if
needed, as long as:
- it comes from a trusted, active source (official marketplace, well-known repo);
- skills/plugins from GitHub are scanned first (e.g. NVIDIA SkillSpector) and
  anything that bypasses security is refused;
- you ask me first if it needs an account, API key, token, payment, or changes
  system/app settings (add to config files, never overwrite them);
- you tell me what you installed, where, and why.

## Engineering checklists: use the one that matches what we're building
Before planning, and again before calling the work done, open the one that fits
and check the work against the sections that apply:
- Software (websites, web apps, APIs, backends, dashboards) → Software
  Engineering Master Checklist
- Hardware (embedded systems, firmware, PCBs, electronics, enclosures) →
  Embedded Systems, PCB and Electronics Master Checklist
- Products with both → use both, each for its own part.
Scale the checklist to the project: a small portfolio site doesn't need
multi-region failover. Tell me which items were skipped and why.

## My Obsidian vault = the project memory
- Start of a task: read the notes related to the task.
- End of every task: update the vault with what changed or was learned (tools,
  install steps, commands, decisions, fixes), in the note it belongs to.
- Never store passwords, API keys or tokens in the vault.
```

Toolkit

Skills are instruction packs that teach Claude how to do one kind of task well. **Plugins** bundle skills, commands and hooks for Claude Code. **MCP servers** (connectors) give Claude access to an outside tool or service.

Claude app vs Claude Code

The Claude app (web, desktop, phone) and Claude Code (terminal, or the desktop app's Code tab) keep separate skill lists. Plugins switched on in the claude.ai account sync into Claude Code automatically; skills uploaded as zip files do not.

Tool	Claude app	Claude Code	Notes
Impeccable	✅	✅	`live`, `generate` and `hooks` only in Claude Code
motion-web	✅	✅	Works best in Claude Code, where it can run the site
Humanizer, Karpathy guidelines, Ponytail	✅	✅	
brag-slim / brag	✅ slim	✅ both	Full brag (HyperFrames) is Claude Code only
Engineering plugins (ECC, Graft, Graphify…)	❌	✅	See Plugins
Playwright MCP, 21st.dev MCP	❌	✅	`claude mcp add`
AnythingMCP, Reticle	✅	✅	App: Settings → Connectors
Embedded / PCB tools	mostly ❌	✅	See Embedded & PCB

Installing with **user scope** (`/plugin install` → "Install for you") is a once-per-computer step; after that a plugin works in every project folder.

Skills

Skill	What it does	How I use it
Impeccable	Design director for interfaces: typography, layout, colour, motion, accessibility, polish. 24 commands.	`init` once per project (writes `PRODUCT.md`), then `shape`, `critique`, `audit`, `polish` on a named page.
motion-web	Builds motion-first sites (GSAP, Framer Motion, light 3D) and tests them in a headless browser.	Animated landing pages, replicating motion from a reference video or URL.
Humanizer	Rewrites AI-sounding text so it reads like a person wrote it, without adding facts.	Website copy, About pages, LinkedIn posts.

Skill	What it does	How I use it
Karpathy guidelines	Four rules against over-building: think first, keep it simple, change only what's needed, define "done".	Always on for code.
Ponytail	Pushes for the smallest code that works: reuse, standard library, native platform features first.	Always on; `/ponytail-review` on a change.
brag / brag-slim	Turns a finished project or live site into a short launch video with a caption.	Only once a project is finished and live.

Plugins

Plugin	Purpose
ECC	Full engineering workflow: plan, test, review, remember
deja-vu	Shared memory across Claude Code, Codex and Cursor sessions
Headroom	Context and token saving hooks
Memory-Palace	Persistent memory for agents
Graft	Codebase context with fewer tokens and tool calls
Graphify	Builds a knowledge graph of a codebase
BrowserSkill	Lets agents use a logged-in browser
teamai-cli	Shares AI skills across a team
npx skills	CLI to install skills into Claude Code and other agents
SkillSpector	Security scanner: check a skill before installing it
Agent Skills	The open skills standard (docs)
OmniRoute	One endpoint routing to many AI providers

MCP servers and connectors

Server	Purpose
Playwright MCP	Browser automation and testing: `claude mcp add playwright npx @playwright/mcp@latest`
21st.dev MCP	Search and install React components from the 21st catalog (needs an API key)
AnythingMCP	Turns any REST, GraphQL or SQL API into MCP tools
Reticle	Tests AI-built apps by replaying user flows
Cloudflare	Docs, deployments and account API (used to host this site)
Canva, Figma	Brand assets and UI design (connected per account when needed)

Embedded and PCB tools

Claude Code can already build and flash firmware by itself (`pio run -t upload`, `idf.py build flash monitor`, `arduino-cli`) and read the errors. These add expert knowledge and safety checks on top.

Tool	Type	What it adds
ESP32-AI-Agent-Skill	Skill	Chip selection, GPIO checks that stop you using strapping or flash pins, Arduino and ESP-IDF code, LVGL screens
esp-expert	Skill + MCP	Real checks instead of guesses: pin conflicts, partition tables, crash-log decoding, sdkconfig and PlatformIO analysis. Never flashes without asking.
KiCAD MCP Server	MCP	Claude reads and edits KiCad projects: placement, traces, board outline, DRC, Gerber/BOM export, JLCPCB part search
Konnect	KiCad 10 plugin + MCP	Same author, native KiCad 10 plugin with design-review audits (AGPL; commercial use needs a licence)
kicad-mcp	MCP	Older, read-and-analyse only; kept as a backup

Rules I follow with hardware tools: save or close the board in KiCad before Claude edits it, keep a git commit of the project, and review every change in KiCad myself before ordering boards.

Web design and front-end

Component and animation libraries (installed per project with npm): React Bits, Cult UI, Skiper UI, Motion Primitives, Motion (Framer Motion), Anime.js, React Three Fiber, Shader Gradient.

Online tools: Haikei (SVG backgrounds), Liquid Logo (liquid-metal logos), ezgif (GIF and video compression), Lighthouse (speed, accessibility and SEO audits), Google Search Console and a sitemap at launch.

Installing tools safely

1. Check the repository has real stars and recent activity.
2. Scan skills and plugins from GitHub with SkillSpector before installing.
3. Refuse anything that bypasses security or antivirus.
4. Anything that needs an account, API key, payment or a settings change waits for my approval.

Things to look out for when designing with AI

AI-built sites share the same blind spots. Every website gets checked against this list before launch, where it applies.

Site essentials: privacy, terms, refund and cookie policies (only when data or payments are involved), cookie consent, no unnecessary data, audited third-party SDKs, no dark patterns, hidden fees, fake reviews or unsupported claims, alt text, colour contrast, keyboard navigation, business details, licensed fonts and images, a way to request data deletion.

Interface details: dark mode toggle, site search, scroll-to-top button, mobile menu, loading animations, hover states, scroll progress bar, copy buttons, print stylesheet, sticky header, skip-to-content link, password visibility toggle, form success and error states, confirmation dialogs, last-updated dates, FAQ, contact button.

Security: no session tokens in localStorage, no client-side-only admin checks, 2FA, rate-limited login, password strength checks, API keys and environment variables kept out of git, real authorisation checks, input sanitisation, XSS and SQL-injection protection, database rules, spend caps, safe file uploads, CSRF protection, strict CORS, HTTPS, security headers, secure cookies, debug mode off in production.

Before launch: no horizontal scroll, no broken links or buttons, favicon, page titles and meta descriptions, working footer links, current copyright year, compressed images, clickable logo, phone number and email, no placeholder text, real 404 pages.

Appendix A · Software Engineering Master Checklist

> Compiled with AI research tools from the standards and sources cited below, and used as a working checklist. Scaled to each project: items that do not apply are skipped with a reason.

A professional software build is not judged only by whether it "works." Developers evaluate product requirements, architecture, security, reliability, scalability, performance, usability, accessibility, testing, deployment, observability, cost, compliance, documentation, maintenance, and eventual retirement.

No finite list can cover every technology or regulation, but the guide below is a near-comprehensive map of the standard, advanced, and often-overlooked topics used when building websites, applications, backends, and APIs.

Core quality attributes

Before choosing frameworks or writing code, teams decide what "good" means for the product. These attributes frequently conflict—for example, stronger consistency can reduce availability, more security can add friction, and extreme scalability can increase cost and complexity.

Attribute	Questions developers ask	Typical measurements
Functionality	Does it perform the required business operations correctly?	Acceptance criteria, passed scenarios, defect rate
Security	Can users, attackers, or dependencies misuse it?	Open vulnerabilities, attack surface, patch time
Reliability	Does it continue operating correctly over time?	Availability, error rate, mean time between failures
Resilience	Can it recover from failures and degraded dependencies?	Recovery time, failover success, retry behavior
Performance	Is it sufficiently fast under realistic conditions?	p50/p95/p99 latency, throughput, CPU, memory
Scalability	Can it handle growth without redesigning everything?	Maximum sustainable requests, users, jobs, connections
Usability	Can intended users complete their tasks easily?	Completion rate, abandonment rate, support requests
Accessibility	Can people with disabilities use it?	WCAG checks, keyboard coverage, screen-reader testing
Maintainability	Can developers safely understand and modify it?	Complexity, duplication, build time, change-failure rate
Testability	Can components be tested independently and reliably?	Coverage, test isolation, flaky-test rate
Portability	Can it move between environments or providers?	Platform dependencies, configuration portability

Attribute	Questions developers ask	Typical measurements
Interoperability	Can it exchange information with other systems?	Protocol and schema compatibility
Observability	Can operators understand internal behavior from outputs?	Logs, metrics, traces, profiles, alert coverage
Privacy	Is personal information collected and processed appropriately?	Data inventory, retention, consent and deletion coverage
Compliance	Does it meet contractual, legal, or industry requirements?	Audit findings, control coverage, evidence freshness
Cost efficiency	Is the value justified by development and operating costs?	Cost per user, request, tenant or transaction
Sustainability	Does it avoid unnecessary compute, storage and transfer?	Energy use, idle capacity, data-transfer volume

Lifecycle framework

NIST's Secure Software Development Framework divides secure development into four groups: prepare the organization, protect the software, produce well-secured software, and respond to vulnerabilities.

At the organizational level, NIST CSF 2.0 uses six continuous functions: Govern, Identify, Protect, Detect, Respond, and Recover.

Phase	Work performed	Expected artifacts
Discovery	Understand users, problem, constraints and risks	Product brief, stakeholder map, initial risk register
Requirements	Define functional and non-functional behavior	User stories, acceptance criteria, SLOs, security requirements
Architecture	Define boundaries, data flows and technology choices	Architecture diagrams, threat model, decision records
Design	Design interfaces and user workflows	Wireframes, design system, API specification
Implementation	Write application and infrastructure code	Source code, tests, migrations, configuration
Verification	Test functionality, security and performance	Test reports, scan findings, accessibility results
Release	Package, approve and deploy changes	Versioned artifact, release notes, deployment record
Operation	Monitor, support and improve the service	Dashboards, alerts, runbooks, incident records
Retirement	Migrate users and safely destroy obsolete resources	Deprecation plan, archive, deletion evidence

Product discovery

A technically excellent system can still fail if it solves the wrong problem. Product discovery establishes the target users, their actual workflows, the business objective and the boundaries of the first release.

- Define the problem without assuming the solution.
- Identify users, administrators, operators, support staff and external integrators.
- Document primary workflows and unusual edge cases.
- Separate “must have,” “should have,” “could have” and “not now.”
- Define success metrics before implementation.
- Validate assumptions through interviews, prototypes or controlled experiments.
- Identify competitors and alternative non-software solutions.
- Define budget, deadline, team size and available skills.
- Determine supported countries, languages, currencies and devices.
- Decide whether the product is public, internal, business-to-business or regulated.
- Define ownership for product, engineering, security, operations and data.
- Document assumptions and the evidence supporting them.
- Maintain a risk register with probability, impact, owner and mitigation.
- Establish criteria for cancelling or changing the project.

Requirements engineering

Requirements should describe both what the application does and how well it must do it. Non-functional requirements such as availability, latency, privacy and recoverability are often harder to add after the architecture is complete.

Functional requirements

- User registration, login, logout and account recovery.
- Roles, permissions and approval workflows.
- Data creation, reading, updating, deletion and export.
- Search, filtering, sorting and pagination.
- Notifications through email, SMS, push or in-app channels.
- Billing, subscriptions, refunds and taxation.
- File upload, download, preview and retention.
- Administrative and support capabilities.
- Import, export and integration workflows.
- Offline, synchronization and conflict behavior.
- Audit history and user-visible activity logs.
- Failure behavior, partial completion and retry behavior.

Non-functional requirements

- Expected normal, peak and future traffic.
- Maximum acceptable response times.
- Availability and maintenance-window expectations.
- Data-loss tolerance and recovery-time targets.
- Geographic deployment and data-residency requirements.
- Browser, device and operating-system support.
- Accessibility target, normally expressed through WCAG conformance.
- Security assurance level and threat profile.
- Retention, deletion and archival rules.
- Maximum operating cost or cost per transaction.

- Support hours and incident-response expectations.
- Upgrade, backward-compatibility and deprecation policies.

Architecture decisions

Architecture should follow the product's real constraints instead of fashion. For many new applications, a well-structured modular monolith is simpler to secure, deploy and debug than premature microservices; services can be separated later when ownership, scaling or isolation genuinely requires it.

Architectural choices

- Monolith, modular monolith, microservices or event-driven system.
- Server-rendered, single-page, static or hybrid frontend.
- REST, GraphQL, gRPC, WebSocket, webhooks or asynchronous messaging.
- Relational, document, key-value, graph, time-series or search database.
- Synchronous versus asynchronous processing.
- Cloud, on-premises, edge, serverless or hybrid hosting.
- Single-region, multi-region or active-passive disaster recovery.
- Shared versus isolated infrastructure for multiple tenants.
- Build versus buy for identity, billing, notifications and analytics.
- Managed service versus self-hosted operation.
- Strong, eventual or user-selectable consistency.
- Stateful versus stateless application processes.

Architecture documentation

- Context diagram showing people and external systems.
- Container or deployment diagram.
- Component boundaries and ownership.
- Data-flow diagram, especially across trust boundaries.
- Network-flow and firewall diagram.
- Authentication and authorization flows.
- Database entities and relationships.
- Event and message schemas.
- Failure-mode diagrams.
- Architecture Decision Records explaining important choices.
- Dependency inventory.
- Threat model and abuse-case catalogue.
- Capacity and cost model.

Security foundations

CISA's secure-by-design guidance says security should be a core product objective before development, while secure-by-default products should enable important protections without requiring customers to discover and configure them. OWASP ASVS provides testable requirements for web-application controls and is more useful as an engineering checklist than relying only on a short "top vulnerabilities" list.

Governance

- Assign a security owner.

- Define security requirements and release gates.
- Classify systems and information by sensitivity.
- Maintain asset, domain, certificate and dependency inventories.
- Establish secure coding standards for each language.
- Train developers in relevant vulnerability classes.
- Require review for sensitive code and infrastructure changes.
- Define vulnerability severity and remediation deadlines.
- Publish a vulnerability-disclosure policy.
- Provide a monitored security contact such as `security.txt`.
- Maintain incident-response and breach-notification procedures.
- Track exceptions, compensating controls and expiration dates.

Threat modelling

Threat modelling asks how the design could be abused before the code exists. It should cover users, administrators, APIs, databases, third parties, build systems, recovery mechanisms and operational staff—not merely the public login page.

- Identify valuable assets and sensitive actions.
- Draw data flows and trust boundaries.
- List attacker capabilities and motivations.
- Examine spoofing, tampering, repudiation, disclosure, denial of service and privilege escalation.
- Write misuse and abuse cases.
- Identify single points of compromise.
- Analyse insider and supply-chain threats.
- Check account recovery and support-channel attacks.
- Prioritize threats by likelihood and impact.
- Assign a control and owner to every accepted threat.
- Revisit the model after architecture or feature changes.

Identity and access

Authentication proves identity; authorization decides what that identity may do. These must be tested separately because a perfectly authenticated user can still exploit broken authorization.

Authentication controls

- Prefer established identity libraries or managed identity providers.
- Offer phishing-resistant passkeys or FIDO/WebAuthn for sensitive systems.
- Require MFA for administrators and high-risk actions.
- Protect login, registration and recovery endpoints from automation.
- Prevent account enumeration through responses and timing.
- Apply secure password storage and breached-password screening.
- Avoid arbitrary periodic password changes unless compromise is suspected.
- Reauthenticate before security-sensitive operations.
- Notify users about important account changes.
- Provide secure recovery codes and factor-revocation mechanisms.
- Record authentication events without recording credentials.
- Design recovery so that it is not weaker than normal login.

CISA describes FIDO/WebAuthn as the widely available phishing-resistant approach and recommends moving toward it; weaker methods may still be better than having no MFA. The older NIST SP 800-63B was withdrawn in August 2025 and superseded by SP 800-63B-4, so developers should verify that authentication advice refers to the current revision.

Authorization controls

- Deny access by default.
- Apply least privilege.
- Check authorization on the server for every protected operation.
- Never trust a hidden button or disabled frontend control as enforcement.
- Validate ownership of every requested object.
- Separate normal users, support staff and administrators.
- Use role-based or attribute-based access where appropriate.
- Separate read, create, update, delete, export and administrative permissions.
- Prevent privilege escalation through editable role fields.
- Test horizontal access, such as one user reading another user's record.
- Test vertical access, such as a normal user invoking an administrator endpoint.
- Review permissions periodically and immediately remove departed users.
- Use temporary just-in-time privileges for particularly sensitive operations.

Sessions and tokens

- Use unpredictable session identifiers.
- Rotate sessions after login and privilege changes.
- Expire idle and absolute session lifetimes.
- Invalidate sessions on logout, compromise and important credential changes.
- Use `Secure`, `HttpOnly` and appropriate `SameSite` cookie attributes.
- Avoid storing long-lived sensitive tokens in browser-accessible storage.
- Verify token signature, issuer, audience, algorithm, expiry and intended use.
- Keep access tokens short-lived.
- Protect refresh tokens and rotate them where practical.
- Provide session/device visibility and remote revocation.
- Do not include sensitive personal data inside readable token payloads.
- Prevent replay where a protocol requires one-time use.

Input and output safety

Every value from users, browsers, mobile applications, files, webhooks, databases and third parties should be treated as untrusted. OWASP ASVS explicitly covers parameterized database queries, contextual output encoding, command injection, CSV injection and related classes.

- Validate type, size, range, format and allowed values.
- Prefer allowlists over attempts to enumerate malicious inputs.
- Use parameterized database queries.
- Use context-specific encoding for HTML, attributes, JavaScript, URLs and headers.
- Sanitize intentionally accepted HTML with a maintained library.
- Avoid dynamic code execution such as `eval`.
- Avoid constructing shell commands from untrusted values.

- Restrict dangerous URL schemes and outbound destinations.
- Protect against server-side request forgery.
- Validate redirects against permitted destinations.
- Defend against path traversal and unsafe archive extraction.
- Validate serialized objects and avoid unsafe deserialization.
- Place limits on deeply nested JSON and extremely large requests.
- Protect regular expressions against catastrophic backtracking.
- Escape spreadsheet formula characters in CSV exports.
- Restrict template-language expressions.
- Normalize Unicode carefully before security comparisons.
- Avoid relying only on client-side validation.

OWASP additionally recommends controlling dangerous server-side URL destinations, sanitizing scriptable formats such as SVG or templates, and examining regular expressions for denial-of-service behavior.

Browser security

Browser security depends on origin boundaries, cookies and correctly constrained content. OWASP ASVS recommends validating cross-origin requests, protecting sensitive operations from CSRF and hosting separate applications on separate hostnames when origin isolation is needed.

- Use HTTPS everywhere.
- Enable HSTS after confirming HTTPS coverage.
- Create a restrictive Content Security Policy.
- Prevent clickjacking with `frame-ancestors` or equivalent policy.
- Set an appropriate referrer policy.
- Set MIME-sniffing protection.
- Use restrictive CORS rules instead of `*` with credentials.
- Protect state-changing requests against CSRF.
- Keep sensitive operations off GET requests.
- Validate the origin of `postMessage` events.
- Use Subresource Integrity for appropriate external static resources.
- Isolate untrusted content on another origin.
- Sanitize user-controlled HTML, SVG, Markdown and CSS.
- Review third-party scripts because they execute with page privileges.
- Minimize browser-stored sensitive information.
- Do not expose private keys, service secrets or unrestricted API keys in frontend bundles.

Data protection

Data protection starts by collecting less information, not merely encrypting everything collected. Teams should know exactly what information exists, why it exists, where it travels, who can access it and when it will be deleted.

- Maintain a data inventory and classification scheme.
- Minimize collection to what the feature actually requires.
- Define purpose and lawful basis where privacy law applies.
- Encrypt information in transit.
- Encrypt sensitive information at rest.

- Manage encryption keys separately from encrypted data.
- Restrict database and storage access by identity and network.
- Mask sensitive values in interfaces and support tools.
- Redact secrets and personal data from logs.
- Tokenize or pseudonymize sensitive identifiers where appropriate.
- Define retention and automatic deletion.
- Support correction, export and deletion requests when required.
- Protect backups to the same standard as production.
- Securely delete temporary files and obsolete snapshots.
- Avoid copying production data into development.
- Use synthetic or anonymized test data.
- Document international transfers and data residency.
- Record access to sensitive information.
- Separate customer data in multi-tenant systems.
- Test that deletion covers caches, search indexes, replicas and delayed jobs.

Secrets and API keys

An API key does not make an application scale. It identifies a calling application or account; scaling comes from capacity, architecture, caching, database design and traffic management.

API-key controls

- Generate keys with cryptographically secure randomness.
- Give each client or environment a different key.
- Show the complete key only at creation.
- Store only a protected representation when possible.
- Include a non-secret prefix or identifier for lookup and support.
- Scope keys to specific operations and resources.
- Support read-only keys.
- Apply per-key quotas, rate limits and concurrency limits.
- Add expiration where operationally practical.
- Support overlapping keys during rotation.
- Revoke compromised keys immediately.
- Record creation, last use, source and revocation.
- Alert on unusual geography, volume or error patterns.
- Prevent keys from appearing in URLs, analytics and logs.
- Never commit keys to source control.
- Never put privileged service keys in public browser code.
- Keep development, staging and production keys separate.
- Use OAuth, workload identity or mTLS when a plain key cannot provide sufficient identity or delegation.

Secrets management

- Keep secrets in a dedicated secrets manager.
- Grant each service only the secrets it needs.
- Prefer short-lived, automatically issued credentials.
- Rotate database, cloud, signing and third-party credentials.

- Scan repositories and commit history for leaked credentials.
- Prevent secrets from being printed by CI pipelines.
- Restrict access to production secrets.
- Audit secret reads and changes.
- Maintain emergency rotation procedures.
- Assume a secret is compromised once exposed, even if the commit is later deleted.

API engineering

OpenAPI provides a language-independent description of HTTP APIs in JSON or YAML and can support documentation, code generation, testing and infrastructure configuration. A well-designed API treats its contract as a product rather than exposing whatever database structure happens to exist.

API contract

- Define endpoints and resource naming consistently.
- Use HTTP methods according to their semantics.
- Use appropriate status codes.
- Validate request and response schemas.
- Define required, optional, nullable and default fields.
- Use consistent date, time, number and identifier formats.
- Standardize error responses.
- Include machine-readable error codes.
- Provide request or correlation IDs.
- Document authentication and authorization.
- Define pagination, filtering, sorting and searching.
- Define rate limits and quota headers.
- Document retries, timeouts and idempotency.
- Publish examples and a change log.
- Generate and validate an OpenAPI document.
- Test the implementation against the contract.

Compatibility

- Prefer additive changes.
- Do not silently change field meaning.
- Avoid reusing removed fields for another purpose.
- Tolerate unknown response fields in clients.
- Define an explicit deprecation period.
- Measure deprecated-version usage before removal.
- Version only when compatibility cannot be maintained.
- Test old clients against new servers.
- Test new clients against supported old servers.
- Evolve event and database schemas safely.
- Keep consumer-driven contract tests for important integrations.

Reliability

- Set client and server timeouts.

- Retry only suitable failures.
- Use exponential backoff with random jitter.
- Place limits on retry counts and total retry time.
- Use circuit breakers where dependency failure could cascade.
- Apply bulkheads to isolate resource pools.
- Make operations idempotent where practical.
- Use idempotency keys for payment, order or job-creation requests.
- Return cached previous results for valid duplicate operations.
- Validate that a reused key has the same request content.
- Expire stored idempotency records according to a documented policy.
- Use dead-letter queues for repeatedly failing asynchronous messages.
- Design webhooks for duplicates, delays and out-of-order delivery.

HTTP defines PUT, DELETE and safe methods as idempotent, while potentially repeated non-idempotent operations need an application-level strategy. The proposed `Idempotency-Key` pattern lets a client identify retries of one logical operation, but keys must be unique and should not be reused for different payloads.

API abuse protection

- Rate-limit by key, account, IP, route and risk where appropriate.
- Distinguish short bursts from sustained quotas.
- Apply maximum body, page and batch sizes.
- Limit expensive search and GraphQL query complexity.
- Bound concurrency for expensive operations.
- Protect login, registration, password reset and code-verification endpoints.
- Add bot detection only where risk justifies its privacy and usability cost.
- Use a WAF as an additional layer, not as a replacement for secure code.
- Detect scraping, credential stuffing and enumeration.
- Return `429 Too Many Requests` with useful retry guidance.
- Prioritize essential traffic during overload.
- Provide administrative quota overrides with expiration and auditing.

Scalability engineering

Scalability begins with measurement. Teams define expected traffic and then test throughput, latency, resource saturation and failure behavior instead of simply adding more servers.

Capacity parameters

- Requests or operations per second.
- Concurrent users and concurrent connections.
- Background jobs per minute.
- WebSocket or streaming connections.
- Database reads and writes per second.
- Query latency and lock time.
- Queue depth and message age.
- CPU, memory, disk and network utilization.
- Cache hit and eviction rates.
- External API quotas.

- Storage growth per day or customer.
- p50, p90, p95, p99 and worst-case latency.
- Error, timeout and retry rates.
- Cost per request, tenant or workload.
- Headroom for traffic spikes and failover.

Scaling patterns

- Keep application instances stateless where possible.
- Place instances behind a load balancer.
- Scale horizontally by adding instances.
- Scale vertically only when it is the sensible short-term or database option.
- Use autoscaling based on meaningful load and saturation signals.
- Serve static files through a CDN.
- Cache frequently read and safely reusable results.
- Precompute expensive recurring results.
- Compress appropriate text responses.
- Move slow work to background jobs.
- Add backpressure instead of accepting unlimited work.
- Batch suitable database or network operations.
- Pool and limit database connections.
- Optimize algorithms before multiplying infrastructure.
- Partition workloads by tenant, region or key when necessary.
- Apply graceful load shedding during overload.
- Warm critical caches or instances before known events.

Database scaling

- Choose indexes based on real query patterns.
- Inspect query plans.
- Avoid repeated per-item queries.
- Select only needed columns.
- Paginate large result sets.
- Use cursor pagination for large or changing datasets.
- Enforce constraints at the database level.
- Keep transactions short.
- Monitor locks, deadlocks and long-running queries.
- Tune connection pools against actual database capacity.
- Add read replicas for suitable read-heavy workloads.
- Cache repeated reads carefully.
- Partition very large tables.
- Archive cold information.
- Use sharding only when simpler approaches are insufficient.
- Plan how globally unique identifiers are generated.
- Define behavior during replica lag.
- Avoid cross-shard operations where possible.
- Test migrations using production-like volume.

Cache correctness

- Define what can be cached and for how long.
- Choose cache keys carefully.
- Prevent one tenant's data from appearing in another tenant's cache.
- Invalidate or version stale entries.
- Prevent cache stampedes through locking, request coalescing or staggered expiry.
- Add random variation to large groups of expiry times.
- Decide behavior when the cache is unavailable.
- Track hit rate, miss rate, eviction and memory usage.
- Never assume a cache is the only durable copy.
- Consider stale-while-revalidate for suitable non-sensitive content.

Asynchronous systems

- Design messages to tolerate duplicate delivery.
- Include event identifiers and timestamps.
- Make consumers idempotent.
- Define ordering requirements explicitly.
- Version message schemas.
- Monitor queue length and oldest-message age.
- Place poison messages in a dead-letter queue.
- Set retry delay and maximum attempts.
- Preserve enough context for investigation.
- Avoid placing secrets or unnecessary personal data in messages.
- Implement backpressure and admission control.
- Plan replay and reprocessing procedures.
- Test what happens if processing succeeds but acknowledgement fails.

Performance testing

Performance testing should use realistic behavior and data sizes. Grafana k6 supports pass/fail thresholds such as error rate and percentile latency, while Locust can define user behavior in Python and distribute load generation across multiple machines.

Test	Purpose
Baseline	Establish normal single-user or low-load performance
Load	Verify behavior at expected production traffic
Stress	Discover the point at which the system degrades
Spike	Test sudden traffic increases
Soak	Find memory leaks and degradation over hours or days
Scalability	Confirm that added resources increase capacity
Failover	Measure behavior while an instance, zone or dependency fails
Volume	Test very large databases, files or message counts

Test	Purpose
Concurrency	Detect races, locks and shared-resource bottlenecks
Endurance	Test recurring jobs, rotations and long-lived connections

Useful tools include k6, Locust, JMeter, Gatling, Vegeta, wrk and cloud-provider load-testing services. Tests should define pass/fail criteria such as “p95 latency below the chosen target at expected load” rather than producing graphs with no decision attached; k6 thresholds are explicitly designed for automated performance gates.

Reliability engineering

Reliability is engineered through explicit service expectations and controlled failure behavior. Google’s SRE guidance distinguishes service-level indicators, objectives and agreements and recommends using an error budget rather than pretending that 100% reliability is practical.

Reliability concepts

- **SLI:** Measurement such as successful-request ratio or latency.
- **SLO:** Internal target for that measurement.
- **SLA:** External contractual commitment and consequences.
- **Error budget:** Permitted unreliability within the SLO.
- **RTO:** Maximum desired recovery time.
- **RPO:** Maximum acceptable data-loss window.
- **MTTD:** Mean time to detect a problem.
- **MTTR:** Mean time to restore service.
- **Change-failure rate:** Percentage of deployments causing incidents.
- **Availability:** Successful service over an agreed measurement window.

Resilience controls

- Health, readiness and liveness checks.
- Redundant instances across failure zones.
- Dependency timeouts.
- Bounded retries with backoff and jitter.
- Circuit breakers.
- Bulkheads and isolated worker pools.
- Queue-based buffering.
- Backpressure.
- Graceful degradation.
- Load shedding.
- Feature kill switches.
- Automatic or documented failover.
- Tested backups and restoration.
- Chaos and fault-injection tests.
- Dependency outage simulations.
- DNS and certificate-expiry preparation.
- Clock-skew tolerance.
- Graceful process shutdown.

- Protection against cascading failures.
- Capacity reserved for failover.

Testing strategy

No single type of test proves software is correct. A healthy strategy combines fast local checks, component verification, integration testing and a smaller number of realistic end-to-end tests.

Functional testing

- Unit tests for isolated logic.
- Component or service tests.
- Integration tests with databases and dependencies.
- API contract tests.
- End-to-end tests for essential user journeys.
- Smoke tests after deployment.
- Regression tests for fixed defects.
- Acceptance tests tied to requirements.
- Exploratory testing.
- Cross-browser and device testing.
- Installation and upgrade testing.
- Database migration and rollback testing.

Specialized testing

- Property-based testing.
- Fuzz testing.
- Mutation testing to evaluate test effectiveness.
- Snapshot or golden-file testing.
- Visual regression testing.
- Accessibility testing.
- Localization and bidirectional-text testing.
- Time-zone and daylight-saving testing.
- Concurrency and race-condition testing.
- Chaos testing.
- Recovery and backup-restoration testing.
- Compatibility and contract testing.
- Performance and resource-leak testing.
- Security penetration testing.
- User-acceptance testing.

Test quality

- Tests should be deterministic.
- Tests should not depend on execution order.
- Shared test state should be minimized.
- Flaky tests should be fixed or quarantined with an owner.
- Test data should be created and cleaned predictably.
- Mocks should not hide integration failures.

- Coverage should identify gaps, not become the only quality target.
- Critical authorization and payment logic deserves direct negative testing.
- Production defects should normally result in regression tests.
- Test duration and failure rate should be monitored.

Vulnerability testing

Security testing should combine design review, human code review, static analysis, dependency analysis, dynamic testing and authorized manual penetration testing. Automated scanners find known patterns, but they cannot reliably understand every business rule or prove the absence of vulnerabilities.

Technique	What it examines	Example tools
SAST	Source code without running the application	Semgrep, CodeQL, SonarQube
SCA	Dependencies, versions and known vulnerabilities	Dependabot, Renovate, Snyk, OSV-Scanner
Secret scanning	Exposed passwords, tokens and private keys	Gitleaks, TruffleHog, GitHub secret scanning
DAST	A running web application or API	ZAP, Burp Suite
IAST	Runtime behavior from inside an instrumented application	Contrast, commercial IAST platforms
Container scanning	OS packages, dependencies and image configuration	Trivy, Grype, Docker Scout
IaC scanning	Terraform, Kubernetes and cloud misconfigurations	Checkov, tfsec, Trivy
Cloud posture	Deployed cloud permissions and resources	Prowler, ScoutSuite, cloud-native services
Fuzzing	Unexpected, malformed or random inputs	AFL++, libFuzzer, Jazzer, Schemathesis
Network scanning	Hosts, ports and known service issues	Nmap, Nessus, OpenVAS
TLS testing	Protocols, certificates and cipher configuration	testssl.sh, SSL Labs
Dependency inventory	Components and transitive relationships	Syft, CycloneDX, SPDX tools
Manual testing	Logic, chaining and context-dependent weaknesses	Burp Community/Professional, browser tools

Semgrep performs static analysis and can operate in editors, pre-commit hooks and CI/CD, while CodeQL treats code as queryable data for finding vulnerability variants. Trivy can examine repositories, container images, Kubernetes and infrastructure code for vulnerabilities, misconfiguration, secrets and software inventories.

ZAP is a free open-source web scanner with automation options, while Burp Scanner performs crawling and dynamic auditing of a running application. Active security testing should only be performed on systems you own or are explicitly authorized to test, preferably in a safe environment where destructive requests cannot affect real users.

Supply-chain security

Modern products include framework packages, build actions, container images, operating-system libraries and external services. An inventory is necessary because a vulnerability in a transitive component may affect the product even though the team never imported it directly.

- Pin dependency versions and verify lock files.
- Review new dependencies before adoption.
- Consider maintenance health, ownership, release history and license.
- Remove unused dependencies.
- Automate vulnerability and update notifications.
- Separate routine updates from emergency security patches.
- Generate an SBOM for releases.
- Use SPDX or CycloneDX formats.
- Sign release artifacts.
- Record build provenance.
- Protect package publishing credentials.
- Prevent dependency-confusion attacks.
- Use trusted registries and namespace controls.
- Pin CI actions or plugins to trusted versions or immutable references.
- Use isolated, disposable build workers.
- Keep signing keys outside user-controlled build steps.
- Verify artifacts before deployment.
- Preserve source, build logs and release metadata.
- Establish an emergency dependency-patching process.

CISA guidance identifies SPDX and CycloneDX as widely used machine-readable SBOM formats. SLSA provides progressively stronger supply-chain controls around provenance, signed hosted builds and hardened build environments.

Code quality

Readable, predictable code is a security and reliability control because unclear code is harder to review and change safely. Teams automate basic rules so reviewers can focus on correctness, design and risk.

- Consistent formatting.
- Linting.
- Type checking.
- Compiler warnings treated seriously.
- Small, focused modules.
- Clear names.
- Explicit error handling.
- Limited duplication.
- Controlled complexity.
- No dead code.
- Comments explaining decisions rather than restating syntax.
- Stable public interfaces.
- Dependency injection where it improves testability.
- Centralized security-sensitive primitives.

- Code ownership for critical areas.
- Pull-request templates.
- Peer review.
- Small changes that can be understood and reverted.
- Commit and release conventions.
- Technical-debt tracking.

Version control

The repository is part of the production system because it contains code, configuration, infrastructure and history. Access, review and automation rules should reflect that importance.

- Protect main and release branches.
- Require pull requests.
- Require passing automated checks.
- Require review for sensitive components.
- Use code-owner rules.
- Prevent direct force pushes.
- Sign commits or releases where assurance requires it.
- Keep secrets out of all branches and history.
- Use meaningful commit messages.
- Tag releases.
- Maintain a change log.
- Define branching and release strategy.
- Archive rather than silently delete important repositories.
- Back up critical repositories.
- Remove access promptly when team membership changes.
- Audit administrative repository changes.

CI/CD pipelines

Continuous integration verifies changes repeatedly; continuous delivery or deployment moves approved artifacts toward production. GitHub Actions, for example, can automate build, test and deployment workflows in response to repository events.

Pipeline stages

1. Install locked dependencies.
2. Verify formatting and linting.
3. Compile or type-check.
4. Run unit tests.
5. Run SAST and secret scanning.
6. Scan dependencies and licenses.
7. Build the release artifact once.
8. Generate an SBOM and provenance.
9. Scan the container or package.
10. Run integration and contract tests.
11. Deploy to an isolated test environment.

12. Run smoke, DAST and selected performance tests.
13. Require approval for sensitive production changes.
14. Deploy the same tested artifact.
15. Verify health and important user journeys.
16. Roll back or disable the feature when verification fails.

Pipeline protection

- Use least-privilege pipeline identities.
- Prefer short-lived workload identities over stored cloud keys.
- Separate untrusted pull-request jobs from secret-bearing jobs.
- Do not expose secrets to forked contributions.
- Pin external actions.
- Isolate build runners.
- Protect deployment environments.
- Record who approved and deployed each release.
- Prevent unauthorized pipeline-file changes.
- Retain logs and evidence according to risk.
- Scan build logs for accidental secret exposure.
- Avoid rebuilding differently for each environment.
- Test pipeline disaster recovery.

Infrastructure and deployment

Infrastructure as Code makes environments reviewable and repeatable. Terraform, for example, represents infrastructure in declarative, versioned configuration and provides a plan before applying changes.

Infrastructure practices

- Define infrastructure as code.
- Review infrastructure changes through pull requests.
- Separate development, testing and production accounts or projects.
- Use reusable, versioned modules.
- Encrypt remote state and restrict access.
- Detect configuration drift.
- Validate plans before applying.
- Avoid manual production changes.
- Use private networking where appropriate.
- Restrict inbound and outbound traffic.
- Apply least privilege to cloud identities.
- Patch hosts, runtimes and managed-service versions.
- Set resource limits.
- Tag resources with owner, environment and cost centre.
- Enable audit logs.
- Automate certificate issuance and renewal.
- Protect DNS administration.
- Periodically remove unused resources and permissions.

Containers and orchestration

Containers package code and dependencies so the application runs consistently across environments. They improve packaging but do not automatically provide security or orchestration.

- Use small trusted base images.
- Pin image versions or digests.
- Run as a non-root user.
- Use read-only filesystems where possible.
- Remove unnecessary tools and packages.
- Do not bake secrets into image layers.
- Scan images before release.
- Sign and verify images.
- Set CPU and memory requests and limits.
- Drop unnecessary Linux capabilities.
- Restrict network communication.
- Use readiness and liveness checks correctly.
- Gracefully handle termination signals.
- Avoid privileged containers.
- Apply orchestration admission policies.
- Update base images regularly.

Release strategies

Strategy	Purpose
Rolling	Gradually replace old instances
Blue-green	Keep old and new environments, then switch traffic
Canary	Send a small percentage of traffic to the new release
Feature flag	Separate deployment from feature activation
Shadow	Send copied traffic without affecting user responses
Ring deployment	Release progressively to increasingly broad user groups
A/B test	Compare user or business outcomes between variants
Emergency rollback	Restore the previous known-good artifact

Database changes must remain compatible while old and new application versions overlap. Destructive schema changes should generally follow an expand–migrate–contract sequence rather than assuming every instance changes simultaneously.

Observability

Observability combines telemetry so operators can determine what happened and why. OpenTelemetry correlates logs, metrics and traces across services through shared context, while profiles can identify which code consumes resources.

Logs

- Use structured logs.

- Include timestamps, service, environment and severity.
- Include request, trace and correlation identifiers.
- Record important state transitions.
- Redact passwords, tokens and personal data.
- Avoid logging complete request bodies by default.
- Protect logs from modification and unauthorized access.
- Define retention and deletion.
- Sample high-volume low-value events.
- Ensure logging failure cannot crash the application.
- Synchronize system clocks.
- Make security audit logs distinguishable from debugging logs.

Metrics

- Request rate.
- Success and error rate.
- Latency distributions.
- Active users and sessions.
- Queue depth and job age.
- CPU, memory, disk and network.
- Database query and connection-pool metrics.
- Cache hit rate.
- Dependency latency and failure rate.
- Authentication failures.
- Rate-limit activity.
- Business outcomes such as completed orders.
- Cost and quota consumption.
- Deployment and version markers.

Prometheus stores labelled time-series data and uses PromQL for queries and alerting; Alertmanager handles notification routing and silencing.

Tracing

- Trace requests across service boundaries.
- Propagate trace context through queues and jobs.
- Record major dependency calls.
- Avoid putting sensitive values in span attributes.
- Sample strategically to control cost.
- Retain error and high-latency traces preferentially.
- Link traces to logs.
- Show deployment version and region.
- Monitor instrumentation overhead.
- Use traces to identify distributed latency rather than guessing.

Alerting

- Alert on user-visible symptoms and imminent exhaustion.
- Align alerts to SLOs.

- Avoid paging for every warning.
- Include severity, owner, dashboard and runbook.
- Deduplicate related alerts.
- Define escalation and acknowledgement.
- Test notification paths.
- Review noisy or ignored alerts.
- Track time to detect and acknowledge.
- Ensure each page has a realistic human action.

Incident response

Incidents are expected operational events, not proof that planning failed. The goal is rapid detection, limited impact, safe recovery and learning without hiding mistakes.

- Define incident severity levels.
- Establish an incident commander.
- Maintain contact and escalation lists.
- Provide internal and external communication templates.
- Record a timeline.
- Preserve relevant evidence.
- Rotate or revoke compromised credentials.
- Provide feature and traffic kill switches.
- Know how to roll back releases.
- Maintain service-specific runbooks.
- Communicate status without unsupported claims.
- Meet contractual and legal notification requirements.
- Conduct blameless post-incident reviews.
- Assign corrective actions, owners and deadlines.
- Test the response plan through tabletop exercises.
- Track recurring root causes.
- Update tests and controls after incidents.

Backup and disaster recovery

A successful backup job does not prove that restoration works. Teams should perform restoration exercises and measure them against documented RPO and RTO targets.

- Define which systems require backup.
- Back up databases, object storage, configuration and critical keys.
- Encrypt backups.
- Separate backup credentials and locations from production.
- Use immutable or offline copies for high-risk systems.
- Define retention and legal-hold behavior.
- Monitor backup completion and integrity.
- Test complete and partial restoration.
- Test restoration into a clean environment.
- Document dependency restoration order.
- Plan for region, account and provider failure.

- Test DNS and traffic failover.
- Keep emergency documentation accessible if normal identity systems fail.
- Record actual restoration time and data loss.
- Delete expired backups securely.

Frontend quality

Frontend engineering covers more than visual appearance. It includes network efficiency, state management, browser compatibility, accessibility, resilience and protection of user information.

Performance

- Minimize initial JavaScript.
- Split code by route or feature.
- Remove unused code and styles.
- Optimize and responsively size images.
- Use modern formats where supported.
- Lazy-load non-critical content.
- Preload only genuinely critical resources.
- Avoid layout shifts.
- Reserve dimensions for images and advertisements.
- Optimize fonts and provide fallbacks.
- Cache versioned static assets.
- Minimize third-party scripts.
- Avoid long main-thread tasks.
- Use virtualization for very long lists.
- Test on slower devices and networks.
- Measure real-user performance, not only laboratory scores.

Lighthouse audits web pages for performance, accessibility and basic SEO, and can run in DevTools, from the command line or as a Node module.

Accessibility

- Use semantic HTML.
- Support complete keyboard navigation.
- Provide visible focus indicators.
- Associate labels and instructions with controls.
- Provide text alternatives for meaningful images.
- Use sufficient colour contrast.
- Do not communicate information through colour alone.
- Support zoom and reflow.
- Provide captions and transcripts for media.
- Respect reduced-motion preferences.
- Announce dynamic changes appropriately.
- Give useful validation and error messages.
- Avoid inaccessible custom controls where native controls work.
- Test with screen readers and keyboard-only interaction.
- Ensure modals manage focus correctly.

- Set document language and meaningful page titles.
- Test accessibility manually; automated tools cannot detect everything.

Useful tools include axe, Pa11y, Lighthouse, WAVE, Accessibility Insights and screen readers such as NVDA, VoiceOver and TalkBack.

User experience

- Clear information architecture.
- Consistent design system.
- Responsive layouts.
- Useful loading, empty, error and success states.
- Prevention of accidental destructive actions.
- Undo where practical.
- Progress indication for long operations.
- Saved state for interrupted workflows.
- Helpful forms with correct input types.
- Confirmation only where consequences justify interruption.
- Mobile touch targets and safe areas.
- Reduced dependence on hover.
- Clear privacy and permission explanations.
- Consistent terminology.
- User testing with representative users.

SEO and sharing

- Unique page titles and descriptions.
- Semantic heading hierarchy.
- Indexable server output where required.
- Canonical URLs.
- Sitemap and robots configuration.
- Structured data where appropriate.
- Correct redirects.
- Useful `404` and removed-content behavior.
- Open Graph and social preview metadata.
- Stable, descriptive URLs.
- Avoid duplicate or thin generated pages.
- Monitor crawl and indexing errors.
- Separate SEO needs from authenticated application pages.

Database and data integrity

Application validation improves the user experience, but the database should enforce essential invariants. Without database constraints, races or alternative code paths can create invalid states despite perfect form validation.

- Primary and foreign keys.
- Uniqueness constraints.
- Nullability and check constraints.

- Appropriate transaction isolation.
- Idempotent writes where retries occur.
- Optimistic or pessimistic concurrency where needed.
- Audit fields and immutable event history where justified.
- Correct monetary types and rounding.
- UTC storage with clear local-time conversion.
- Explicit character encoding and collation.
- Migration versioning.
- Forward and backward-compatible migrations.
- Backup before dangerous migrations.
- Online migration strategy for large tables.
- Referential cleanup and retention jobs.
- Data-quality checks.
- Reconciliation between internal and external systems.
- Duplicate detection and merge rules.
- Import validation and quarantine.

Multi-tenancy

Multi-tenant systems create a particularly serious risk of cross-customer information exposure. Tenant isolation must exist in authorization, queries, caches, jobs, storage paths, logs, exports and administrative tools.

- Include tenant identity in every relevant authorization decision.
- Ensure every tenant-owned database query is scoped.
- Add tenant context to cache keys.
- Isolate files and object-storage prefixes.
- Prevent tenant identity from being supplied without verification.
- Test cross-tenant access automatically.
- Separate especially sensitive tenants physically if required.
- Apply per-tenant quotas and rate limits.
- Attribute cost and usage to tenants.
- Support tenant-specific retention and residency.
- Protect support impersonation.
- Audit administrative access to tenant data.
- Design safe tenant deletion and export.
- Avoid customer-controlled identifiers as the only isolation boundary.

Payments and high-risk workflows

High-risk operations require stronger correctness, auditing and replay protection than ordinary content browsing. Financial operations should be designed as state machines rather than a collection of loosely connected Boolean fields.

- Use specialist payment providers instead of storing card information unnecessarily.
- Verify webhook signatures.
- Handle duplicate and out-of-order events.
- Use idempotency keys.

- Reconcile provider and internal records.
- Keep an immutable transaction ledger where appropriate.
- Separate authorization, capture, refund and dispute states.
- Avoid floating-point types for money.
- Record currency with every amount.
- Make refunds and administrative adjustments auditable.
- Require stronger authorization for high-value actions.
- Detect fraud without silently locking legitimate users indefinitely.
- Test partial failure between internal and external systems.
- Never trust a browser redirect as final proof of payment.

Privacy and compliance

Compliance frameworks vary by product, country and industry, so developers first identify which obligations actually apply. Common examples include privacy regulation, payment requirements, healthcare rules, contractual controls, SOC 2, ISO 27001 and government-specific standards.

- Record processing purposes and data categories.
- Capture valid consent where required.
- Support consent withdrawal.
- Publish accurate privacy information.
- Define data-controller and processor responsibilities.
- Review third-party processors.
- Sign appropriate processing agreements.
- Define retention by data category.
- Support access, correction, portability and deletion rights.
- Conduct privacy-impact assessments for risky processing.
- Minimize tracking and advertising identifiers.
- Restrict employee access to personal data.
- Document cross-border transfers.
- Protect children’s data where relevant.
- Maintain audit evidence continuously.
- Do not claim certification or compliance without evidence.
- Obtain qualified legal and compliance advice for regulated products.

Documentation

Documentation is part of the system because it determines whether developers and operators can use it safely. Documentation should be versioned and tested where possible.

- README with purpose and local setup.
- Architecture overview.
- Environment and configuration reference.
- API documentation and examples.
- Authentication and authorization model.
- Database schema and migration procedures.
- Deployment and rollback instructions.
- Monitoring and alert descriptions.

- Incident runbooks.
- Backup and restoration procedures.
- Security assumptions and threat model.
- Dependency and licensing information.
- Release notes and change log.
- Troubleshooting guide.
- Ownership and escalation contacts.
- Known limitations.
- Deprecation policy.
- Disaster-recovery plan.
- New-developer onboarding.
- User and administrator documentation.

Team practices

Good engineering also depends on communication and decision quality. Process should create useful feedback and accountability without turning every small change into bureaucracy.

- Clear ownership.
- Defined review expectations.
- Small, reversible changes.
- Shared definition of done.
- Regular backlog refinement.
- Architecture and security review for risky work.
- Pairing or mobbing for difficult changes.
- Retrospectives with tracked improvements.
- Rotation of operational knowledge.
- On-call compensation and sustainable schedules.
- Learning time and secure-development training.
- Documented decisions.
- Psychological safety for reporting errors.
- Dependency and technical-debt maintenance time.
- Succession planning for critical components.
- Avoiding single-person knowledge silos.

Cost engineering

Cost should be measured alongside performance and reliability. A design that scales technically but costs more than the business value is not sustainable.

- Estimate development and operating costs.
- Track cost by service, environment and tenant.
- Tag cloud resources.
- Set budgets and anomaly alerts.
- Remove idle development resources.
- Right-size compute and databases.
- Apply storage lifecycle policies.
- Monitor data-transfer charges.

- Control log and trace volume.
- Cache expensive repeated operations.
- Batch suitable workloads.
- Use reserved or committed capacity only for predictable usage.
- Measure cost per useful business transaction.
- Include third-party API, email, SMS and AI-model costs.
- Consider engineering labour before self-hosting.
- Conduct load tests before purchasing excess capacity.
- Include disaster-recovery infrastructure in estimates.

Niche edge cases

These issues appear small until they cause production failures or security incidents.

- Leap years and daylight-saving transitions.
- Time zones with non-hour offsets.
- Leap seconds and unsynchronized clocks.
- Unicode normalization and visually similar characters.
- Right-to-left and mixed-direction text.
- Extremely long names and addresses.
- Users without surnames.
- Currency rounding and zero-decimal currencies.
- Large integer precision in JavaScript.
- Floating-point comparison.
- Duplicate clicks and network retries.
- Browser back-button behavior.
- Expired tabs and stale forms.
- Upload filenames containing paths or control characters.
- Zip bombs and archive traversal.
- Image metadata containing sensitive location information.
- CSV formula injection.
- Catastrophic regular-expression backtracking.
- Partial database or message-processing success.
- Out-of-order webhook delivery.
- Email case handling and internationalized domains.
- DNS failure and delayed propagation.
- Certificate expiry.
- Clock skew affecting tokens.
- Exhausted file descriptors, ports or thread pools.
- Disk-full behavior.
- Log system failure.
- Queue poisoning.
- Cache failure and cache stampede.
- Very slow clients.
- Abandoned uploads.
- Search-index lag.

- Replica lag.
- Account merge and identity-provider changes.
- User deletion while jobs remain queued.
- Third-party API behavior changing without notice.
- External service returning valid but unexpected data.
- Feature flags remaining permanently.
- Rollback after an irreversible database migration.
- Deployment while long-running jobs use old code.
- Browser extensions modifying page behavior.
- Ad blockers blocking badly named application resources.
- Corporate proxies modifying traffic.
- IPv6-only clients.
- Shared IP addresses causing unfair rate limiting.

Advanced practices

These techniques become valuable as risk, scale or organizational complexity grows.

- Formal threat modelling workshops.
- Architecture fitness functions.
- Policy as code.
- Admission control for infrastructure.
- Signed commits, artifacts and attestations.
- Reproducible or hermetic builds.
- Runtime application protection.
- eBPF-based runtime visibility.
- Continuous profiling.
- Chaos engineering.
- Game days and disaster simulations.
- Error-budget-based release policy.
- Progressive delivery.
- Automated canary analysis.
- Multi-region traffic engineering.
- Cell-based architecture limiting blast radius.
- Per-tenant encryption keys.
- Confidential computing.
- Hardware-backed keys.
- Zero-trust service identity.
- Mutual TLS between services.
- Database activity monitoring.
- Data-loss prevention.
- Automated evidence collection.
- Privacy-preserving analytics.
- Differential privacy or secure computation for specialized use cases.
- Property-based and model-based testing.
- Mutation testing.

- Continuous fuzzing.
- Digital twins or production-traffic replay with sanitized data.

AI-enabled applications

Applications using language or generative models inherit normal software risks and introduce model-specific ones. Model output must be treated as untrusted input, especially when it can call tools, execute code, query private information or generate HTML.

- Protect against prompt injection.
- Separate trusted instructions from untrusted content.
- Restrict model tools through explicit allowlists.
- Apply authorization independently of the model.
- Minimize information placed into prompts.
- Prevent cross-user retrieval leakage.
- Validate and sanitize generated output.
- Require confirmation for destructive actions.
- Sandbox generated code.
- Set token, time and cost limits.
- Log model and prompt versions appropriately.
- Evaluate hallucination and unsafe-output rates.
- Test indirect prompt injection in retrieved documents.
- Prevent model output from silently changing security policy.
- Provide fallbacks for model or provider outages.
- Review data-retention terms of model providers.
- Monitor quality drift after model changes.
- Keep deterministic validation around financial, safety or access decisions.

Recommended starter toolchain

For a student, individual developer or small team, this provides strong coverage without immediately requiring enterprise products.

Area	Practical starting choice
Repository	GitHub or GitLab with protected main branch
Planning	GitHub Issues, Linear, Jira or a simple project board
Formatting/linting	Language-native formatter and linter
Type checking	TypeScript, mypy, compiler checks or equivalent
Unit testing	Framework native to the selected language
Browser testing	Playwright
API testing	Bruno, Postman/Newman or REST Assured
API specification	OpenAPI with Swagger UI or Redoc
CI/CD	GitHub Actions or GitLab CI

Area	Practical starting choice
SAST	Semgrep or CodeQL
Dependency updates	Dependabot or Renovate
Secret scanning	Gitleaks
Container/IaC scanning	Trivy
DAST	ZAP
Manual web security	Burp Suite Community
Load testing	k6 or Locust
Frontend audit	Lighthouse and axe
Containers	Docker
Infrastructure as Code	Terraform or OpenTofu
Metrics	Prometheus
Dashboards	Grafana
Telemetry	OpenTelemetry
Error tracking	Sentry or equivalent
Logs	Loki, Elasticsearch/OpenSearch or a managed platform
Secrets	Cloud secrets manager, Vault or platform-managed secrets
SBOM	Syft, Trivy or CycloneDX tooling

Minimum pipeline

For a portfolio, API or IoT dashboard project, a sensible minimum sequence is:

1. Protect the main branch and require pull requests.
2. Run formatter, linter, type checker and unit tests.
3. Run Semgrep or CodeQL.
4. Run dependency and secret scanning.
5. Build a container or deployable artifact.
6. Scan it with Trivy.
7. Deploy automatically to a test environment.
8. Run smoke and API tests.
9. Run ZAP baseline scanning against the test environment.
10. Run a small k6 test with explicit latency and error thresholds.
11. Require manual approval before production.
12. Deploy using a reversible strategy.
13. Run production smoke tests.
14. Send errors, metrics and structured logs to monitoring.
15. Maintain backups, a restoration test and a basic incident runbook.

Release checklist

A production release should not require every advanced technique above, but these questions should have clear answers:

- Is the requirement and acceptance criterion clear?
- Has another person reviewed the change?
- Do automated tests pass?
- Were authentication and authorization tested negatively?
- Were user inputs and outputs reviewed?
- Are migrations backward-compatible?
- Are secrets outside the repository and artifact?
- Are dependencies and images scanned?
- Is the API contract still compatible?
- Is expected load known and tested?
- Are logs, metrics and alerts available?
- Is sensitive information absent from telemetry?
- Can the change be rolled back or disabled?
- Is the backup recent and restorable?
- Is documentation updated?
- Is there an owner for production problems?
- Will users be informed of breaking or visible changes?
- Has operating cost changed unexpectedly?
- Are temporary permissions and feature flags scheduled for removal?

The most important lesson is not to adopt every tool immediately. Start with explicit requirements, secure defaults, version control, automated tests, dependency and secret scanning, reproducible deployment, observability, backup restoration and a rollback plan; then add complexity only when the product's risk and scale justify it.

Reference links

- NIST Secure Software Development Framework SP 800-218
- NIST Cybersecurity Framework 2.0
- NIST Digital Identity Guidelines SP 800-63B-4
- CISA More Than a Password
- CISA Secure by Design
- OWASP Application Security Verification Standard
- OpenAPI Specification
- HTTP Semantics RFC 9110
- Google Site Reliability Engineering
- OpenTelemetry Documentation
- Prometheus Documentation
- SLSA Supply-chain Levels
- OWASP ZAP
- PortSwigger Burp Suite
- Semgrep
- CodeQL

- Trivy
- Grafana k6
- Locust
- Chrome Lighthouse
- GitHub Actions Documentation
- Terraform Documentation
- Docker Containers Overview

Appendix B · Embedded Systems, PCB & Electronics Master Checklist

> Compiled with AI research tools from the standards and sources cited below, and used as a working checklist. Scaled to each project: items that do not apply are skipped with a reason.

Overview

A professional embedded product is not merely firmware running on a microcontroller or a PCB that powers on. It is a coordinated electrical, mechanical, firmware, radio, manufacturing, test, safety, security, regulatory, supply-chain and lifecycle system. The design process therefore begins with measurable requirements and hazards, continues through architecture, component selection, schematic capture, PCB layout, firmware, prototype bring-up and verification, and ends only after production support and responsible retirement.

This guide is a broad engineering map rather than a substitute for the product-specific standards, component datasheets, calculations, laboratory testing and design reviews required for a commercial product. IEC 61508, for example, treats functional safety as a lifecycle concern spanning sensors, logic, processors and actuators rather than as a final test added to a completed design.[1]

Quality attributes

Attribute	Main engineering question	Typical evidence
Functional correctness	Does the product perform every required function and reject invalid states?	Requirements traceability, verification tests
Electrical safety	Can normal use or a single fault cause shock, fire, excessive heat or hazardous energy?	Hazard analysis, insulation design, safety tests
Functional safety	Can an electronic or software malfunction create an unacceptable hazard?	Safety analysis, diagnostics, safe-state tests
Reliability	Will it continue operating for its intended lifetime and mission profile?	Derating, reliability analysis, life testing
Signal integrity	Do digital signals arrive with valid timing and voltage margins?	Simulation, eye diagrams, oscilloscope measurements
Power integrity	Do supply rails remain within limits during static and transient loading?	PDN analysis, ripple and transient measurements
EMC	Does it avoid excessive emissions and tolerate expected disturbances?	Pre-compliance and accredited EMC reports
Thermal performance	Do junctions, batteries, connectors and enclosures stay within limits?	Thermal model, thermocouple and infrared tests
Real-time behavior	Are deadlines met under worst-case execution and interrupt loading?	Timing analysis, trace captures, stress tests
Cybersecurity	Can unauthorized parties read, modify, control or permanently disable it?	Threat model, secure boot, penetration tests

Attribute	Main engineering question	Typical evidence
Manufacturability	Can the board be fabricated and assembled consistently at target yield?	DFM/DFA review, process capability, pilot build
Testability	Can faults be detected and isolated economically?	Test points, boundary scan, fixtures, diagnostics
Serviceability	Can it be updated, repaired, calibrated and diagnosed in the field?	Update mechanism, logs, service procedures
Compliance	Does it meet every market and industry rule that applies?	Technical file, declarations and certifications
Cost	Is total unit and lifecycle cost acceptable?	Costed BOM, yield, test time, warranty forecast
Sustainability	Can material, energy, packaging and electronic waste be reduced?	Power budgets, repairability and end-of-life plan

Development lifecycle

Phase	Main activities	Outputs
Product discovery	Identify users, environment, hazards, volumes, cost and regulations	Product brief, use cases, constraints
Requirements	Convert expectations into measurable limits	System, hardware, firmware and test requirements
Feasibility	Prove uncertain circuits, radios, sensors, algorithms and suppliers	Bench experiments, simulations, risk register
Architecture	Partition power, analog, digital, RF, firmware and mechanics	Block diagrams, interface control documents
Detailed design	Schematic, PCB, firmware, enclosure and test design	Design database, source code, drawings
Prototype	Build controlled engineering samples	Prototype records and as-built configuration
Bring-up	Energize safely and validate subsystems in sequence	Bring-up log, fault list, measurements
Verification	Prove that implementation meets specified requirements	Test procedures, reports and traceability
Validation	Prove that the complete product meets user needs in its real environment	Field trials, compliance and qualification data
Industrialization	Optimize assembly, programming, test and supply chain	Manufacturing package, fixtures, pilot run
Production	Control process, quality, serialization and changes	Yield data, inspection and release records

Phase	Main activities	Outputs
Operation	Monitor failures, deploy updates and manage vulnerabilities	Service data, updates, corrective actions
Retirement	Stop production, revoke services and dispose safely	Last-time-buy, migration and recycling plan

Requirements engineering

Hardware errors are expensive because a schematic or layout change can require new boards, assembly and compliance testing. Requirements should therefore be numeric, testable, versioned and connected to a verification method.

Product context

- Intended users, installers, maintainers and adversaries.
- Indoor, outdoor, automotive, industrial, marine, aerospace, medical or consumer environment.
- Expected annual volume and product lifetime.
- Target selling price, maximum BOM cost and assembly cost.
- Required markets, languages, labeling and certification.
- Repairable, disposable, field-upgradable or sealed product strategy.
- Operating, storage and transport conditions.
- Normal use, foreseeable misuse and abuse.
- Required availability and consequences of failure.
- Intellectual-property and export-control constraints.
- Manufacturing location, approved vendors and supply-chain risks.

Electrical requirements

- Input voltage nominal, minimum, maximum, reverse polarity and transient profile.
- Maximum steady-state, startup, sleep and fault current.
- Power-source impedance and cable resistance.
- Battery chemistry, cell count, capacity, charge rate and discharge rate.
- Required regulated rails, accuracy, noise and sequencing.
- Digital logic levels and tolerances.
- Sensor input ranges, bandwidth, resolution, noise and accuracy.
- Output voltage, current, load type and protection.
- Isolation voltage and insulation class where applicable.
- Connector pin current, insertion cycles and hot-plug behavior.
- Grounding, shielding and chassis connection requirements.
- Surge, ESD, EFT, radiated RF and conducted RF environment.
- Radio bands, transmit power, bandwidth, sensitivity and antenna restrictions.

Firmware requirements

- Startup time and boot behavior.
- Control-loop and communication deadlines.
- Worst-case interrupt latency and jitter.
- RAM, flash, external storage and stack budgets.

- Sleep states, wake sources and resume time.
- Supported protocols, packet sizes and error behavior.
- Data retention, event logging and timestamp accuracy.
- Local and remote update behavior.
- Recovery after interrupted update or unexpected reset.
- Authentication, authorization and credential lifecycle.
- Diagnostic and manufacturing modes.
- Safe state after detected faults.
- Supported hardware revisions and compatibility policy.

Mechanical requirements

- Board outline, thickness, mounting points and keep-outs.
- Connector positions and mating clearances.
- Enclosure material, sealing and ventilation.
- Weight and center of mass.
- Shock, vibration and drop profile.
- Ingress-protection target.
- User touch temperature and surface temperature.
- Cable bend radius, strain relief and pull force.
- Assembly and disassembly sequence.
- Service access and tamper resistance.
- Antenna clearance and radome material.

Verification language

Avoid statements such as “low power,” “fast,” “robust” or “long range.” Replace them with conditions and pass criteria, such as maximum sleep current at a stated voltage and temperature, packet delivery ratio at a defined distance and antenna orientation, or recovery within a specified time after supply interruption.

Each requirement should have:

- Unique identifier.
- Rationale and source.
- Owner.
- Priority.
- Applicable operating mode and environment.
- Numeric tolerance or pass/fail criterion.
- Verification method: inspection, analysis, simulation, demonstration or test.
- Links to implementation and test evidence.
- Revision history.

System architecture

Partitioning

- Separate high-energy power, noisy switching, digital processing, precision analog, RF and user-accessible domains.
- Define each subsystem’s power rail, ground reference, reset, clock and communications.
- Decide which safety and security functions require independent hardware.

- Identify trust boundaries and isolation barriers.
- Distinguish hard real-time control from background processing.
- Determine what must operate when the main processor fails.
- Identify shared resources and single points of failure.
- Decide whether redundancy must be diverse, independent or merely duplicated.
- Define startup, normal, degraded, maintenance and shutdown states.
- Design observability and test access into the architecture.

Architecture artifacts

- System context diagram.
- Functional block diagram.
- Power tree and power-state diagram.
- Clock tree.
- Reset and boot chain.
- Interface control document.
- Pin-multiplexing plan.
- Memory map.
- Data-flow and trust-boundary diagram.
- State-machine definitions.
- Timing and scheduling model.
- Fault-tree and failure-mode analysis.
- Thermal path diagram.
- Grounding and shielding strategy.
- Programming, calibration and test architecture.
- Manufacturing and field-update flow.

Technology choices

- MCU, MPU, DSP, FPGA, SoC or mixed architecture.
- Bare metal, cooperative scheduler, RTOS or embedded Linux.
- Internal versus external flash, RAM and nonvolatile memory.
- Wired versus wireless communication.
- Standard module versus custom radio design.
- Linear regulator versus switched converter.
- Galvanic isolation versus common ground.
- Two-layer, four-layer, multilayer, flex, rigid-flex or metal-core PCB.
- Commercial, industrial, automotive, aerospace or radiation-tolerant parts.
- In-house firmware stack versus vendor middleware.
- Custom bootloader versus maintained secure update framework.

CMSIS gives Arm microcontroller projects standardized processor, peripheral, driver and RTOS interfaces, which can reduce vendor-specific coupling and improve reuse. Zephyr is one example of a small-footprint, multi-architecture RTOS ecosystem for constrained connected devices, while FreeRTOS is commonly configured with fixed-priority preemptive scheduling and equal-priority time slicing.[2][3][4]

Risk management

Engineering risk register

Track each risk with likelihood, impact, detection method, mitigation, contingency, owner and review date. Typical risks include:

- New component not available at production volume.
- Undocumented silicon erratum.
- RF range dependent on enclosure orientation.
- Sensor drift beyond calibration range.
- Converter unstable with selected capacitor.
- Thermal limit exceeded inside sealed enclosure.
- Flash endurance insufficient for logging rate.
- External API or cloud service discontinued.
- Safety certification requiring redesign.
- Contract manufacturer unable to inspect hidden joints.
- Test fixture throughput too low.
- Programming key exposure in production.
- Single-source connector or custom mechanical part.

FMEA and fault analysis

For each function, component or interface, ask:

- How can it fail: open, short, stuck, drift, intermittent, corrupt or delayed?
- What causes the failure?
- What local and system effects follow?
- Can the failure be detected before harm?
- What is the severity, occurrence and detectability?
- Is there a safe state?
- Can another fault defeat the protection?
- Does the diagnostic itself fail safely?
- Is the failure latent until another failure occurs?
- What proof or test closes the action?

Use system FMEA, design FMEA, process FMEA, fault-tree analysis, hazard and operability analysis, worst-case circuit analysis and common-cause analysis according to product risk. IEC 61508 defines four Safety Integrity Levels and requires hazards and risks to be considered early in the lifecycle.[1]

Component selection

Selection parameters

- Electrical ratings across tolerance and temperature.
- Absolute maximum versus recommended operating conditions.
- Package, pitch, thermal pad and assembly capability.
- Temperature grade.
- Qualification grade and manufacturer quality system.
- Lifecycle status and longevity commitment.

- Manufacturer, distributor and country-of-origin risk.
- Lead time, minimum order quantity and price breaks.
- Pin-compatible or functionally compatible alternatives.
- Datasheet completeness, errata and application support.
- Evaluation board and reference design availability.
- Development tools, programmer and debugger support.
- Software drivers and license.
- Security functions and update support.
- Calibration, accuracy and long-term drift.
- Moisture-sensitivity level.
- ESD sensitivity.
- RoHS, REACH, conflict-mineral and material declarations.
- Counterfeit risk and authorized supply channels.

BOM governance

- Use manufacturer part numbers rather than vague descriptions.
- Record approved alternates with explicit engineering review.
- Attach lifecycle, compliance and qualification evidence.
- Separate do-not-substitute parts from flexible commodities.
- Track manufacturer change notices and end-of-life notices.
- Version the BOM with the schematic, PCB, firmware and assembly drawing.
- Record fitted and not-fitted variants.
- Define approved distributors.
- Review cost at prototype, pilot and volume quantities.
- Maintain last-time-buy and redesign plans for critical parts.
- Validate alternates electrically, thermally, mechanically and in firmware.

Derating

Derating keeps normal stresses below component maxima to reduce degradation and catastrophic failure. It can apply to voltage, current, power, junction temperature, ripple current, electric field, contact load and mechanical stress; the correct limits depend on the product standard and mission rather than one universal percentage. NASA reliability guidance explicitly connects reduced electrical, thermal and mechanical stress with lower failure probability.[56]

Review:

- Capacitor voltage, ripple current and temperature.
- Resistor continuous and pulse power.
- MOSFET voltage, current, avalanche energy and safe operating area.
- Diode repetitive and surge current.
- Inductor saturation, RMS current and core temperature.
- Connector pin current and temperature rise.
- Relay contact load and switching life.
- Semiconductor junction temperature.
- PCB trace and via current with acceptable rise.
- Insulation working and transient voltage.
- Battery charge/discharge current and temperature.

Schematic design

Schematic quality

- Organize hierarchically by functional block.
- Show signal flow consistently.
- Name rails and signals clearly.
- Label active-low signals unambiguously.
- Show connector pin numbers and mating orientation.
- Mark differential polarity.
- Mark controlled-impedance, matched and sensitive nets.
- Identify default states and pull resistors.
- Include test points and programming interface.
- Place explanatory notes near unusual circuits.
- Include revision, sheet references and design ownership.
- Avoid hidden power pins.
- Mark intentional no-connect pins.
- Verify every symbol against the current datasheet.
- Verify each symbol-to-footprint pin mapping independently.
- Record component tolerances and voltage ratings.

KiCad's ERC can find common connection problems such as unconnected pins, two power outputs tied together, unpowered power inputs, annotation errors and net-label mistakes, but it cannot prove that the circuit will function. A clean ERC is therefore necessary but not sufficient.[7]

Datasheet review

For every active component, inspect:

- Pin descriptions and alternate functions.
- Recommended operating conditions.
- Absolute maximum ratings.
- Power-up and power-down sequence.
- Decoupling and bulk-capacitance requirements.
- Reset timing and threshold behavior.
- Clock requirements.
- Boot-strap pins.
- Unused-pin recommendations.
- Thermal data.
- Layout guidance.
- Package drawing and pin-one convention.
- Reference circuits.
- Known errata.
- Peripheral electrical characteristics.
- Input leakage and clamp-current limits.
- Output drive and edge-rate options.

Power architecture

Power budget

Create a mode-by-mode budget for startup, transmit, receive, computation, charging, actuator operation, sleep and fault states. Include nominal, realistic worst case and transient peak values.

- Source voltage tolerance.
- Cable and connector drop.
- Reverse polarity.
- Cold-crank, load dump or hot-plug transients where relevant.
- Inrush current.
- Brownout threshold.
- Regulator dropout and headroom.
- Efficiency versus load and input voltage.
- Quiescent current in each state.
- Load-transient response.
- Rail sequencing.
- Power-good timing.
- Hold-up energy.
- Thermal dissipation.
- Fault isolation.
- Battery runtime and aging.

Input protection

Depending on source and environment, consider:

- Fuse, resettable fuse or electronic fuse.
- Reverse-polarity MOSFET or diode.
- Transient-voltage suppressor.
- Surge stopper.
- Current limiter.
- Undervoltage and overvoltage lockout.
- Inrush or soft-start control.
- Common-mode and differential filtering.
- Connector-first-mate ground.
- Arc and contact-bounce effects.
- Crowbar protection where suitable.
- Protection coordination so downstream parts survive until the fuse opens.

Regulators and converters

- Verify input, output and switching frequency ranges.
- Check compensation and loop stability.
- Check allowed output-capacitance range, ESR and bias derating.
- Calculate inductor peak, RMS and saturation current.
- Calculate diode or synchronous-switch stress.
- Verify minimum on/off time.
- Analyze discontinuous, pulse-skipping and light-load modes.

- Check startup into pre-biased output.
- Check reverse current and back-powering.
- Account for switch-node ringing and snubbers.
- Minimize hot-loop area.
- Provide thermal copper and vias.
- Validate transient response on hardware.
- Measure ripple with a low-inductance probing method.

Decoupling and PDN

Decoupling is a current-loop problem, not just a BOM quantity. TI guidance emphasizes short, solid connections and locating decoupling so the current path reaches the capacitor before the device supply pin.[89]

- Place high-frequency capacitors near each supply pin or pin group.
- Minimize the loop formed by supply pin, capacitor and ground return.
- Use short, wide connections and nearby ground vias.
- Add local bulk capacitance for slower load changes.
- Account for ceramic-capacitor DC-bias loss.
- Avoid resonances created by arbitrary capacitor combinations.
- Model target impedance for demanding processors or FPGAs.
- Consider package and via inductance.
- Measure rail impedance, ripple and transient droop.
- Separate especially noisy rails when filtering is justified.
- Do not insert ferrite beads without checking DC current, impedance curve and possible resonance.

Batteries and charging

Battery design topics

- Cell chemistry and approved cell supplier.
- Series/parallel configuration.
- Capacity at actual discharge rate and temperature.
- Cycle life and calendar aging.
- Charge profile and termination.
- Cell balancing.
- Overcharge, over-discharge, overcurrent and short-circuit protection.
- Temperature sensing and charge inhibition.
- Fuel gauging and state-of-charge learning.
- Cell matching.
- Pack fuse and secondary protection.
- Swelling, venting and thermal-runaway path.
- User replacement and reverse insertion.
- Storage and shipping state of charge.
- Deep-discharge recovery policy.
- Charger and load power-path operation.
- Connector spark and inrush.
- Mechanical crush and puncture protection.

Lithium cells and batteries generally must satisfy UN Manual subsection 38.3 before transport; its test family covers altitude, temperature cycling, vibration, shock, external short circuit and battery-type-dependent impact, overcharge and forced-discharge tests. A battery product may also require separate cell, pack, charger, end-product and regional safety evaluations.[10][11]

Analog and sensor design

Signal-chain budget

- Input range and common-mode range.
- Sensor sensitivity and offset.
- Gain and gain error.
- ADC full-scale range and reference accuracy.
- Resolution versus effective number of bits.
- Noise density and integrated noise bandwidth.
- Offset drift and gain drift.
- Linearity.
- Source impedance and ADC acquisition time.
- Anti-alias filtering.
- Input protection leakage and capacitance.
- Settling time.
- Calibration uncertainty.
- Quantization and rounding.
- Ground and supply rejection.
- Crosstalk from digital and power circuits.

Op-amp checks

- Input common-mode range.
- Output swing at required load.
- Gain-bandwidth and phase margin.
- Slew rate.
- Input bias and offset current.
- Noise.
- Rail-to-rail behavior.
- Capacitive-load stability.
- Input overvoltage and phase reversal.
- Power-up behavior.
- Input/output ESD and fault protection.

ADC and DAC checks

- Reference source, noise, drive and decoupling.
- Sampling capacitor settling.
- Input driver stability.
- Clock jitter for higher-frequency conversion.
- Differential/common-mode requirements.
- Digital output timing.
- Calibration and self-test.

- Missing-code and monotonicity requirements.
- Ground return placement.
- Reconstruction filtering for DAC outputs.

Sensor concerns

- Cross-axis and cross-sensitivity.
- Warm-up time.
- Mounting stress.
- Board bending.
- Self-heating.
- Airflow and thermal gradients.
- Magnetic materials near magnetometers.
- Vibration rectification.
- Acoustic coupling.
- Sensor poisoning and contamination.
- Drift, hysteresis and aging.
- Factory and field calibration.
- Traceability to a reference instrument.
- Plausibility checks and redundant sensing.

Clocks, reset and supervision

Clock design

- Frequency accuracy over tolerance, temperature and aging.
- Crystal load capacitance and drive level.
- Oscillator startup margin.
- Trace symmetry and short crystal loop.
- Keep noisy and high-current signals away.
- Jitter and phase-noise requirement.
- Clock-domain crossing.
- Spread-spectrum implications.
- Backup clock and failure detection.
- Real-time-clock drift and calibration.
- Timestamp continuity across reset and power loss.

Reset design

- Guaranteed reset at slow power ramps.
- Brownout detection.
- Power-good supervision.
- Minimum reset pulse.
- External reset filtering.
- Reset-source logging.
- Independent watchdog.
- Debug behavior under watchdog.
- Safe GPIO state during reset and boot.

- Peripheral reset sequencing.
- Recovery from clock failure.
- Reset-loop prevention.

Digital interfaces

General interface review

- Voltage levels and absolute maxima.
- Input thresholds and noise margin.
- Pull-up/down values.
- Rise and fall times.
- Source and sink current.
- Drive strength and slew-rate settings.
- Termination.
- Cable length and impedance.
- Hot-plug and back-power paths.
- ESD protection capacitance.
- Ground-reference difference.
- Connector pin order.
- Fault behavior when one side is unpowered.
- Timeout and recovery behavior.

UART

- Voltage standard: logic UART, RS-232 or RS-485.
- Baud tolerance and clock error.
- Framing, parity and stop bits.
- Buffering and overrun handling.
- Flow control.
- Line termination and biasing for RS-485.
- Half-duplex direction timing.
- Ground reference and isolation.
- Break detection and bootloader exposure.

I2C and SMBus

- Total bus capacitance.
- Pull-up calculation for rise time and sink current.
- Address collisions.
- Level translation.
- Clock stretching.
- Multi-master behavior.
- Stuck-low recovery.
- Hot-plug behavior.
- Bus segmentation or buffering.
- Connector exposure and ESD.
- Repeated-start and timing compatibility.

SPI

- Mode and sampling edge.
- Maximum clock at actual trace/cable length.
- Chip-select setup, hold and idle state.
- MISO contention.
- Series damping resistors.
- Separate chip selects and boot states.
- DMA and transfer framing.
- CRC or application-level integrity when needed.

CAN and LIN

- Correct transceiver voltage and standby control.
- Bus termination and split termination.
- Stub length.
- Common-mode range.
- ESD, surge and automotive transients.
- Isolation when ground difference requires it.
- Bit-timing and sample point.
- Error counters and bus-off recovery.
- Wake-up behavior.
- Connector and shielding strategy.

USB

- Correct connector and USB-C configuration resistors.
- VBUS detection and power role.
- Inrush and current limits.
- Differential impedance and routing.
- ESD devices with suitable capacitance.
- Return path and connector shielding.
- Crystal/clock accuracy.
- Device descriptors and unique identifiers.
- Power Delivery policy if used.
- Test fixtures and compliance pattern support.

USB-IF certification uses electrical, protocol and interoperability test specifications and assigns a Test ID to qualifying products; official analysis tools include SigTest and USBET for relevant electrical measurements.[1][2][3]

Ethernet

- PHY and magnetics compatibility.
- Controlled differential impedance.
- Pair length and intra-pair skew.
- Reference-plane continuity.
- Bob Smith/chassis termination where required by the design.
- Center-tap supply filtering.
- Isolation and creepage.

- ESD and surge at the connector.
- Link LEDs and strap-pin states.
- MAC address provisioning.
- Cable category and PoE requirements.

PCB stack-up

The stack-up should be agreed with the fabricator before final routing because impedance, dielectric thickness, copper weight, via aspect ratio and manufacturability depend on it.

Stack-up parameters

- Layer count and symmetry.
- Finished board thickness and tolerance.
- Core and prepreg materials.
- Glass transition temperature.
- Loss tangent and dielectric constant at relevant frequency.
- Copper weight and finished plating.
- Signal-to-reference-plane spacing.
- Controlled-impedance geometries.
- Plane pairing for power integrity.
- Warp and bow risk.
- Sequential lamination for HDI.
- Blind, buried, microvia and via-in-pad requirements.
- Surface finish.
- Solder mask type and color.
- Silkscreen process.
- IPC performance class and acceptance criteria.

Layer planning

- Place high-speed signals next to continuous reference planes.
- Give critical signals predictable return paths.
- Avoid routing over plane splits or voids.
- Keep stack-up symmetric to reduce warpage.
- Route adjacent signal layers orthogonally when useful, but rely on spacing and references rather than this alone.
- Reserve plane continuity under clocks, buses and differential pairs.
- Use ground stitching where return currents change layers.
- Avoid broadside coupling between long parallel traces on adjacent layers.
- Plan power distribution before placement becomes fixed.

TI high-speed guidance stresses continuous reference planes, minimum return-loop area and avoiding signal routes over gaps in their reference plane. Analog Devices similarly emphasizes controlling the actual return-current path; high-frequency return current tends to follow the low-impedance path beneath the signal trace.[149]

PCB placement

Placement often determines whether routing, EMC and thermal performance will succeed.

Placement sequence

1. Fix board outline, mounting holes, keep-outs and enclosure constraints.
2. Place connectors, switches, LEDs, antennas and user interfaces.
3. Place protection components directly at external entry points.
4. Place power conversion according to the converter's critical loops.
5. Place the MCU/SoC, memories, oscillators and decoupling.
6. Place analog and sensor chains in signal-flow order.
7. Place RF components and matching network.
8. Place termination and bias components at the electrically correct endpoint.
9. Place test points, programming headers and fiducials.
10. Review assembly access, rework and mechanical collision.

Placement checks

- Functional blocks are compact and recognizable.
- High-current loops are short.
- Sensitive analog is separated from switching nodes, clocks and antennas.
- Crystal components are close to oscillator pins.
- Decoupling capacitors have minimal loops.
- Connector protection is before internal circuitry.
- Hot components have spreading area and airflow.
- Temperature sensors are located according to what they must measure.
- Magnetometers are far from inductors, motors and ferromagnetic hardware.
- Antenna keep-out includes copper, components, battery, display and enclosure effects.
- Large or heavy components have mechanical support.
- Polarized and pin-one marks are visible after assembly.
- Components are oriented consistently where practical.
- Rework tools can reach likely service parts.
- Probe access exists for critical nodes.

PCB routing

General routing

- Route critical signals first.
- Keep current paths short and loops small.
- Keep traces over continuous references.
- Change layers only when needed.
- Add nearby return vias when a signal changes reference layer.
- Use trace widths based on current, acceptable temperature rise and fabrication limits.
- Avoid neck-downs at pads and vias that become current bottlenecks.
- Use teardrops or pad-entry improvements where fabrication/reliability requires them.
- Avoid acid traps and awkward slivers according to fabricator capability.
- Keep copper away from board edges per manufacturing and safety requirements.
- Remove unintended antennas such as long stubs.
- Route clocks and fast edges away from connectors and sensitive analog.
- Do not route under noisy inductors or switch nodes unless the reference design explicitly permits it.

Differential pairs

- Select target differential and common-mode impedance.
- Keep pair geometry consistent.
- Match within protocol needs, not arbitrary perfection.
- Minimize intra-pair skew.
- Avoid asymmetrical via transitions.
- Keep pair spacing from aggressors.
- Maintain reference-plane continuity.
- Avoid stubs.
- Place common-mode chokes only when justified and according to vendor routing guidance.
- Validate connector, package and ESD-device discontinuities.

Length matching

- Derive tolerance from timing budget and propagation delay.
- Match at the receiver-relevant point.
- Match within byte lanes or clock groups as required by the interface.
- Avoid excessive serpentine coupling.
- Keep meander spacing sufficient to reduce self-coupling.
- Include package and connector delays where important.
- Do not length-match slow, asynchronous signals without reason.

Crosstalk

- Increase spacing.
- Reduce parallel run length.
- Use a close reference plane.
- Reduce driver edge rate where possible.
- Route sensitive signals away from clocks and switching nodes.
- Add ground shielding only with a correct return connection.
- Use layer transitions strategically.
- Simulate buses whose margins are tight.

Grounding and return paths

“Ground” is not an equipotential point at high frequency or high current. The layout must control where current flows.

- Use continuous ground planes for most digital and mixed-signal boards.
- Partition components and routing by function even when using one plane.
- Keep noisy return currents out of sensitive analog paths.
- Avoid cutting a plane without a deliberate return-current strategy.
- Never route a high-speed trace across a plane split.
- Connect cable shield and chassis according to frequency, safety and enclosure strategy.
- Use short, wide chassis connections and stitching near external connectors when appropriate.
- Separate high-current motor or converter returns from measurement returns.
- Join grounds at a deliberate point when isolation or architecture requires separate domains.
- Verify isolated-domain capacitance and common-mode current.

Ground-plane advice is context dependent: some converter and mixed-signal reference designs use carefully controlled islands or star connections, while many systems are better served by one continuous plane with strict placement and routing. The unifying rule is to understand the return currents rather than splitting planes by label alone.[15][14]

Signal integrity

When to treat a signal as high speed

Edge rate, not only clock frequency, determines whether transmission-line effects matter. Analyze traces whose propagation delay is significant relative to rise/fall time, especially clocks, memories, USB, Ethernet, fast SPI, display links and long cables.

SI parameters

- Driver rise/fall time and output impedance.
- Trace characteristic impedance.
- Flight time.
- Receiver thresholds and hysteresis.
- Reflection coefficient.
- Overshoot and undershoot.
- Ringing and settling time.
- Setup and hold margin.
- Jitter.
- Crosstalk.
- Via, package and connector discontinuities.
- Termination topology.

SI tools

- IBIS model simulation.
- SPICE simulation for simpler networks.
- Field solver for impedance and coupling.
- Time-domain reflectometry.
- High-bandwidth oscilloscope and suitable probe.
- Logic analyzer for protocol-correlated timing.
- Eye-diagram and compliance software.

Termination options

- Source-series termination.
- Parallel termination.
- Thevenin termination.
- AC termination.
- Differential termination.
- On-die termination.

Choose from topology, edge rate, line impedance, driver strength, DC power and receiver location. Place termination at the electrically correct end and provide footprints for tuning during prototype work.

Power integrity

PI concerns

- DC voltage drop.
- Plane and trace resistance.
- Via current and bottlenecks.
- Regulator control-loop response.
- Decoupling anti-resonance.
- Package inductance.
- Simultaneous switching noise.
- Ground bounce.
- Rail-to-rail coupling.
- Core-rail transient demand.
- FPGA/SoC power sequencing.

PI verification

- DC drop simulation.
- PDN impedance simulation.
- Rail ripple measurement.
- Load-step response.
- Near-pin probing.
- Current-probe measurement.
- Thermal observation of current paths.
- Margin testing at low/high input and temperature.
- Worst-case activity patterns.

EMC and ESD

EMC has two sides: the product must limit emissions and remain functional under expected disturbances. The IEC 61000 family covers terminology, environments, limits, measurement techniques, immunity tests, installation and mitigation guidance.[16]

Emission control

- Minimize current-loop area.
- Slow unnecessary digital edges.
- Keep clocks and switch nodes compact.
- Maintain continuous return planes.
- Filter external cables at entry points.
- Prevent common-mode current on cables.
- Shield noisy or sensitive sections where justified.
- Stitch ground around board edges and connectors when appropriate.
- Choose switching frequency with system bands and harmonics in mind.
- Avoid routing fast signals toward board edges.
- Control enclosure seams and apertures.
- Measure both conducted and radiated emissions early.

Immunity controls

- Place ESD protection close to connectors.
- Route surge current away from sensitive circuitry.
- Use short paths to chassis or the intended return.
- Select TVS standoff, clamp voltage, surge rating and capacitance correctly.
- Add common-mode or differential filtering based on coupling mode.
- Protect reset, interrupt and high-impedance lines.
- Debounce and validate external inputs in firmware.
- Use watchdog and brownout recovery.
- Protect communication transceivers against expected common-mode stress.
- Define acceptable behavior during and after each immunity test.

Common tests

- Electrostatic discharge.
- Electrical fast transient/burst.
- Surge.
- Conducted RF immunity.
- Radiated RF immunity.
- Voltage dips and interruptions.
- Magnetic field immunity.
- Conducted emissions.
- Radiated emissions.
- Harmonics and flicker where applicable.

System-level ESD testing such as IEC 61000-4-2 is distinct from component HBM/CDM ratings; a chip's ESD rating does not prove the finished product will survive field discharge. JEDEC also maintains manufacturing and handling standards for ESD-sensitive devices.[17][18]

Pre-compliance tools

- Spectrum analyzer.
- Near-field electric and magnetic probes.
- LISN for conducted-emission work.
- Current probe.
- TEM/GTEM cell where available.
- ESD simulator.
- EFT/burst and surge generators.
- RF signal generator and amplifier.
- Oscilloscope with appropriate isolation and probes.
- DIY current-injection and close-field experiments for diagnosis, followed by proper laboratory testing.

RF and antenna design

RF requirements

- Region and frequency band.
- Channel plan and bandwidth.
- Output power.

- Receiver sensitivity.
- Modulation and data rate.
- Link margin.
- Duty-cycle or listen-before-talk restrictions.
- Antenna type, gain, polarization and efficiency.
- Coexistence with other radios.
- Spurious, harmonic and occupied-bandwidth limits.
- Human RF-exposure requirements.
- Certification strategy: module or custom radio.

Link budget

Include:

- Transmitter conducted power.
- Feed and matching loss.
- Transmit antenna gain.
- Path loss.
- Obstruction, orientation, body and enclosure losses.
- Receive antenna gain and feed loss.
- Receiver sensitivity at chosen data rate.
- Fade margin.
- Interference margin.
- Regulatory power and antenna constraints.

RF layout

- Follow the transceiver and front-end reference design closely.
- Use the intended stack-up and controlled impedance.
- Keep matching components close and ordered correctly.
- Keep RF path short.
- Use solid reference ground and dense local stitching.
- Isolate RF from clocks, converters, display cables and high-current loops.
- Keep copper and components out of the antenna keep-out.
- Provide a conducted RF test connector or removable link when feasible.
- Avoid vias in critical RF paths unless modeled.
- Tune the antenna in the final enclosure with battery, display and cables installed.
- Validate every product variant because enclosure or PCB changes can detune the antenna.

RF measurements

- Vector network analyzer for impedance and return loss.
- Spectrum analyzer for output spectrum and spurious emissions.
- Power meter.
- Signal generator for receiver tests.
- Network analyzer fixtures and calibration standards.
- Anechoic or reverberation chamber for radiated performance.
- Packet-error-rate and sensitivity testing.
- Coexistence and blocking tests.

- Temperature and supply-corner measurements.

In the United States, RF devices generally require the applicable FCC authorization before marketing or import, and certification testing must use the required recognized laboratory path. EU radio products fall under the Radio Equipment Directive and associated cybersecurity and market-access requirements, so regulatory applicability must be checked against the planned sale date and product category.[19][20][21][22]

Isolation and high voltage

Insulation concepts

- Functional insulation.
- Basic insulation.
- Supplementary insulation.
- Double insulation.
- Reinforced insulation.
- Working voltage.
- Rated impulse voltage.
- Overvoltage category.
- Pollution degree.
- Material group and comparative tracking index.
- Mains transient and altitude.
- Accessible, SELV/PELV and hazardous circuits.

Clearance is the shortest path through air; creepage is the path along an insulating surface. Their required values depend on factors such as working or impulse voltage, pollution, material group, insulation class and altitude, so a generic “millimetres per volt” rule is unsafe.[23][24]

High-voltage design checks

- Apply the actual product safety standard.
- Calculate creepage and clearance at all component leads, pads, slots and board edges.
- Include manufacturing tolerances and contamination.
- Consider altitude correction.
- Use certified isolation components where required.
- Check transformer construction and hipot rating.
- Keep copper pours and test points out of barriers.
- Use slots only when accepted by the relevant standard and manufacturer.
- Mark hazardous zones and access restrictions.
- Design discharge paths for capacitors.
- Verify touch current and leakage.
- Analyze single faults.
- Perform dielectric withstand and insulation-resistance tests safely.
- Maintain isolation in test fixtures and programming equipment.

Thermal design

Thermal model

- Ambient minimum and maximum.

- Internal enclosure air temperature.
- Power dissipation by component and mode.
- Junction-to-case, junction-to-board and junction-to-ambient paths.
- Copper spreading.
- Thermal-via performance.
- Interface material and heatsink resistance.
- Airflow and orientation.
- Solar loading.
- Nearby heat sources.
- Transient versus steady-state operation.
- Component power derating with temperature.

Thermal layout

- Spread high-power components where appropriate.
- Keep heat away from precision references, oscillators and temperature-sensitive sensors.
- Use recommended thermal-pad and via patterns.
- Avoid solder voiding risk through unsuitable via-in-pad construction.
- Provide copper area and heatsink mounting.
- Consider enclosure conduction paths.
- Keep electrolytic capacitors away from heat.
- Check connector and trace temperature rise.
- Account for conformal coating and potting.
- Model sealed enclosures pessimistically.

Thermal testing

- Thermocouples attached appropriately.
- Infrared camera with emissivity correction.
- On-die sensors with known accuracy limitations.
- Maximum-load soak.
- Low/high input voltage.
- Blocked ventilation.
- Hot ambient and solar load.
- Fault and overload cases.
- Startup and repeated duty cycles.
- Battery charging and radio-transmit peaks.

Mechanical integration

- Confirm PCB outline from controlled CAD data.
- Verify mounting-hole size, plating and keep-outs.
- Define screw torque and washers.
- Avoid board bending under connectors or buttons.
- Support heavy inductors, transformers, batteries and connectors.
- Include strain relief.
- Check enclosure tolerances and tolerance stack-up.
- Verify connector mating and unmating forces.

- Account for cable routing and minimum bend radius.
- Prevent sharp edges and wire abrasion.
- Consider vibration modes and resonances.
- Keep antennas away from metal, battery and users as required.
- Provide pressure equalization for sealed outdoor enclosures where necessary.
- Choose coating, gasket and potting compounds for chemical and temperature compatibility.
- Check plastics for flammability and UV stability when relevant.
- Consider galvanic corrosion between metals.
- Design drainage and condensation management.

Environmental design

IEC 60068 supplies general environmental test methods and severities intended to assess performance under transport, storage and operational conditions.[25]

Mission profile

Define:

- Operating and non-operating temperature.
- Rate of temperature change.
- Humidity and condensation.
- Altitude and pressure.
- Dust, water and salt exposure.
- Corrosive gases and industrial chemicals.
- Sunlight and UV.
- Vibration spectrum and duration.
- Mechanical shock and drop.
- Acoustic vibration.
- Radiation environment.
- Biological growth and insects.
- User sweat, oils and cleaning agents.
- Storage duration.
- Transport packaging and handling.

Protection techniques

- Conformal coating.
- Potting or encapsulation.
- Gaskets and seals.
- Vent membranes.
- Corrosion-resistant finishes.
- Stainless or coated hardware.
- Connector boots and seals.
- Desiccants for packaging, not as a substitute for correct enclosure design.
- Drainage and orientation.
- Shock mounts.
- Thread locking.
- Cable retention.

- UV-resistant materials.

Coatings and potting can improve environmental robustness but may alter RF tuning, capacitance, thermal performance, reworkability and stress, so they must be included in validation samples.

Design for manufacture

Fabrication DFM

- Minimum trace and space.
- Minimum drill and finished hole.
- Annular ring.
- Via aspect ratio.
- Copper-to-edge clearance.
- Solder-mask dam.
- Legend width and clearance.
- Controlled-impedance tolerance.
- Copper balance.
- Stack-up symmetry.
- Rout, V-score and panel rails.
- Surface finish.
- Via tenting, plugging, filling or capping.
- HDI lamination count.
- Board warp and bow.
- Coupon and impedance-test requirements.

Assembly DFA

- Package pitch and assembler capability.
- Component-to-component spacing.
- Courtyard and tool clearance.
- Reflow direction for large packages.
- Shadowing for wave or selective solder.
- Thermal relief and solderability.
- Paste-aperture design.
- Thermal-pad paste segmentation.
- Via-in-pad treatment.
- Tombstoning risk.
- Bottom-terminated component inspection.
- BGA X-ray requirement.
- Double-sided reflow and heavy-component risk.
- Hand-solder access.
- Fiducials.
- Pick-and-place nozzle clearance.
- Consistent polarity orientation.
- Panelization and depanelization stress.
- Wash and no-clean process compatibility.

Manufacturing standards

IPC standards cover many stages of electronic product development and provide a shared language for quality and process expectations. J-STD-001 defines materials, methods, process control and acceptance requirements for soldered electronic assemblies. The exact IPC design, fabrication, assembly and acceptance documents and product class should be explicitly placed on drawings and purchase orders rather than described vaguely as "IPC compliant."[26][27]

Moisture-sensitive devices

J-STD-033 defines handling, packing, shipping and floor-life controls for moisture/reflow-sensitive devices to prevent moisture-related damage during solder reflow.[28][29]

- Record MSL from component labels and datasheets.
- Track cumulative floor exposure.
- Use moisture-barrier bags, desiccant and humidity indicator cards.
- Store sensitive devices in controlled dry cabinets.
- Bake only according to approved temperature/time limits.
- Avoid baking unsuitable tape, trays or components.
- Reset records after controlled bake.
- Train receiving, kitting, assembly and rework staff.

ESD control in manufacturing

- Define ESD-protected areas.
- Ground operators through monitored wrist straps or approved footwear/floor systems.
- Use grounded dissipative work surfaces.
- Control insulators and static-generating materials.
- Ionize where grounding cannot neutralize charge.
- Use ESD-safe containers and packaging.
- Test grounding and monitoring equipment.
- Train staff.
- Audit the program.
- Protect boards through receiving, assembly, test, repair and shipping.

JEDEC JESD625 establishes minimum controls for handling ESD-sensitive electronic devices.[30][17]

Design for test

Test strategy

Choose coverage according to volume, risk and fault economics:

- Incoming inspection.
- Automated optical inspection.
- Solder-paste inspection.
- X-ray inspection.
- Bare-board flying probe or electrical test.
- In-circuit test.
- Boundary scan/JTAG.
- Flying-probe assembled-board test.
- Functional test.

- Programming and provisioning.
- Calibration.
- Burn-in.
- Environmental stress screening.
- Final product test.
- Sample audit and destructive analysis.

Test-point design

- Ground references near measured signals.
- Power rails.
- Reset and boot pins.
- Programming/debug signals.
- Important analog nodes.
- Communication buses.
- Factory-mode entry.
- Current-measurement links.
- RF conducted-test point if appropriate.
- Isolation-safe access.
- Adequate pad diameter, spacing and keep-out for fixture probes.
- No solder mask over probe contact.
- Access from a consistent side where possible.

Production test principles

- Test the likely defect mechanisms, not every mathematical possibility.
- Separate fixture self-test from device test.
- Use known-good reference units.
- Record raw measurements and limits, not only pass/fail.
- Serialize units and link results to PCB, firmware and calibration revisions.
- Control instrument calibration.
- Prevent test software changes without review.
- Use guard bands thoughtfully.
- Measure false-fail and escape rates.
- Minimize test time without hiding marginal products.
- Design safe retest and rework policies.
- Secure programming keys and credentials.

Fabrication outputs

A PCB release package commonly includes:

- Gerber or IPC-2581/ODB++ fabrication data.
- NC drill and route files.
- Stack-up and impedance table.
- Fabrication drawing.
- Assembly drawings for each side.
- Pick-and-place coordinates.
- BOM with approved manufacturer parts.

- Paste layers and stencil notes.
- Panelization requirements.
- Test-point file.
- Netlist for fabrication verification.
- 3D model or STEP assembly.
- Programming files and checksums.
- Firmware version and configuration.
- Calibration and test procedures.
- Special process notes.
- Compliance and material requirements.
- Approved deviation process.

Before release:

- Run ERC and DRC.
- Resolve every warning explicitly.
- Compare schematic and PCB netlists.
- Verify board outline and dimensions.
- Inspect every footprint against the datasheet.
- Review pin one, polarity and connector orientation.
- View all Gerbers independently.
- Inspect drill pairs and plated/non-plated holes.
- Check solder-mask and paste openings.
- Check copper-to-edge and mounting holes.
- Review 3D fit with enclosure.
- Conduct DFM/DFA review with the actual manufacturer.
- Freeze and archive the complete release revision.

KiCad strongly recommends fixing DRC violations before fabrication output; DRC can identify shorts, insufficient clearances, disconnected tracks and schematic-layout mismatches, but human review remains necessary.[7]

Firmware architecture

Layering

- Boot ROM and immutable trust anchor.
- Bootloader and update manager.
- Board support package.
- Hardware abstraction.
- Peripheral drivers.
- RTOS or scheduler.
- Middleware and protocol stacks.
- Services such as storage, time, logging and diagnostics.
- Application logic.
- Safety and security monitor.
- Manufacturing and calibration functions.

Keep application logic separate from direct register manipulation where that improves portability and testability. Define clear ownership of peripherals and shared state.

State machines

Model explicit states for:

- Boot and self-test.
- Provisioning.
- Normal operation.
- Sleep and wake.
- Communication connection.
- Charging.
- Fault and degraded operation.
- Firmware update.
- Factory test.
- Calibration.
- Safe shutdown.

For each transition define trigger, guard, action, timeout, failure behavior and persistence across reset.

Memory planning

- Flash partitions.
- Bootloader and application slots.
- Update scratch or secondary image.
- Configuration and calibration storage.
- Crash log.
- Filesystem.
- Stack and heap budgets.
- DMA buffers.
- Alignment and cache requirements.
- Memory-protection regions.
- External-memory integrity and encryption.
- Reserve for future updates.

Real-time design

FreeRTOS's default fixed-priority preemptive scheduler runs the highest-priority ready task, so a high-priority task that never blocks can starve lower-priority work.[3]

Scheduling parameters

- Task period.
- Deadline.
- Worst-case execution time.
- Priority.
- Stack size.
- Blocking time.
- Jitter.

- Shared resources.
- Interrupt source and rate.
- Queue depth.
- CPU utilization.
- Watchdog participation.

RTOS hazards

- Priority inversion.
- Deadlock.
- Livelock.
- Starvation.
- Race conditions.
- Unbounded blocking.
- Excessive interrupt duration.
- Stack overflow.
- Heap fragmentation.
- Lost wakeups.
- Queue overflow.
- Timer wraparound.
- Reentrancy problems.
- Incorrect interrupt-safe API use.

Mutex priority inheritance can reduce some priority-inversion effects but does not eliminate the underlying design problem.[31]

Interrupt design

- Keep handlers short and bounded.
- Acknowledge the source correctly.
- Move slow processing into tasks.
- Avoid blocking and unsafe APIs.
- Protect shared data with suitable atomic or synchronization methods.
- Budget nested interrupt latency.
- Validate peripheral flag-clearing sequence.
- Handle interrupt storms and stuck sources.
- Trace maximum handler duration.

Embedded coding practices

General practices

- Enable compiler warnings and treat important warnings as errors.
- Use static analysis.
- Define integer widths explicitly for protocols and registers.
- Avoid undefined and implementation-defined behavior.
- Check return values.
- Validate lengths before copying or parsing.
- Avoid unsafe string operations.

- Bound loops and retries where real-time behavior requires it.
- Make ownership and lifetime explicit.
- Keep interrupt-shared variables and memory ordering correct.
- Use assertions in development and controlled fault handling in production.
- Centralize hardware-register definitions.
- Document concurrency assumptions.
- Avoid dynamic allocation after startup in deterministic systems unless rigorously controlled.
- Make time comparisons safe across counter wraparound.
- Include unit suffixes in names or types.
- Review endian conversion and packed structures.
- Do not map network data directly onto compiler-dependent structs.

For safety- and security-related C, MISRA C is a widely adopted rule set intended to reduce ambiguous and undefined language behavior and improve analyzability.[32]

Numerical correctness

- Fixed-point scaling and overflow.
- Signed/unsigned conversion.
- Integer promotion.
- Saturation versus wraparound.
- Floating-point precision and NaN behavior.
- Division by zero.
- Sensor unit conversion.
- Calibration coefficient range.
- Accumulated integration error.
- Timestamp rollover.
- Atomic access to multiword values.

Firmware security

Threat model

Consider:

- Physical access to UART, SWD, JTAG, boot pins and flash.
- Malicious or counterfeit firmware.
- Compromised update server.
- Network attacker.
- Stolen device credentials.
- Extracted flash contents.
- Debug-port abuse.
- Fault injection and voltage/clock glitching.
- Side-channel leakage.
- Rollback to vulnerable firmware.
- Malicious peripheral, USB host or radio peer.
- Factory insider and contract manufacturer.
- Cloud-account compromise.
- Decommissioned devices retaining secrets.

Root of trust and boot

- Immutable first-stage verification.
- Protected public-key hash or root key.
- Signed image verification before execution.
- Verified chain for every boot stage.
- Secure boot failure policy.
- Measured boot or attestation where needed.
- Debug authentication or permanent disable policy.
- Protected security configuration and lifecycle state.
- Device-unique identity.
- Key revocation and rotation strategy.

NIST firmware-resiliency guidance organizes protection around preventing unauthorized change, detecting corruption and recovering securely.[33][34]

Secure updates

- Cryptographically authenticate images.
- Protect image integrity and metadata.
- Confirm device and hardware compatibility.
- Enforce version and anti-rollback policy.
- Support power-loss-safe updates.
- Use A/B slots, swap, scratch or recovery partition appropriately.
- Verify the image before activation and after reboot.
- Preserve or migrate configuration safely.
- Rate-limit repeated failures.
- Provide controlled local recovery.
- Log update result without exposing secrets.
- Maintain signing-key separation and offline protection.
- Support signing-key rotation.
- Define end-of-support policy.

NIST specifically recommends update roots or chains of trust, authenticated updates and controls against unauthorized rollback to older vulnerable firmware.[34]

Secrets and provisioning

- Generate unique device credentials.
- Never ship universal production passwords or keys.
- Use true hardware random generation where available and validated.
- Keep private keys inside secure elements or protected hardware when risk warrants it.
- Separate development, manufacturing and production trust roots.
- Secure key injection at the factory.
- Audit provisioning events.
- Avoid printing secrets in logs or labels.
- Protect backup and escrow material.
- Revoke lost or compromised devices.
- Zeroize sensitive data during decommissioning.
- Plan certificate expiration and renewal.

Runtime hardening

- Memory protection unit or memory management unit.
- Execute-never regions.
- Stack protection.
- Control-flow protection where available.
- Privileged/unprivileged separation.
- Input length and format validation.
- Authenticated and encrypted communications.
- Rate limits and lockout resistant to denial of service.
- Minimal enabled services.
- Secure defaults.
- Watchdog and fault recovery.
- Security-event logging.
- Safe handling of malformed packets and files.

Firmware assessment tools

- Compiler sanitizers on host-test builds.
- Static analysis: clang-tidy, cppcheck, Coverity, CodeQL or suitable commercial tools.
- MISRA checkers.
- Dependency and license scanning.
- Binary analysis with Ghidra or equivalent.
- Firmware extraction with binwalk and filesystem tools.
- Debug hardware such as J-Link, ST-Link, OpenOCD and pyOCD.
- Protocol fuzzers and custom property-based tests.
- Hardware interface tools for UART, SPI, I2C, CAN, USB and RF.
- Fault-injection tools for high-assurance research.

OWASP's firmware methodology covers reconnaissance, firmware acquisition, extraction, static analysis, emulation, dynamic/runtime testing and exploitation; these activities should only be used on systems owned by or explicitly authorized for the tester.[35][36]

Communications and protocols

Protocol design

- Explicit frame boundary.
- Version field.
- Message type.
- Length field.
- Sequence number.
- Integrity check or authenticated encryption.
- Source/destination identity.
- Replay protection.
- Timeout.
- Acknowledgement and retry semantics.
- Duplicate handling.
- Flow control.

- Maximum sizes.
- Error codes.
- Backward compatibility.
- Byte order.
- Unit and scaling definition.
- Reserved-field behavior.

Reliability

- Assume packets can be lost, duplicated, delayed, corrupted or reordered.
- Bound retransmission count and time.
- Add random backoff where multiple devices contend.
- Make commands idempotent when possible.
- Separate transport acknowledgement from application completion.
- Persist only what must survive resets.
- Handle partial writes.
- Recover from peer reset independently.
- Design for clock drift and temporary disconnection.
- Provide congestion and queue-overflow behavior.

Wireless coexistence

- Frequency and channel planning.
- Antenna isolation.
- Shared-antenna switching.
- Transmit scheduling.
- Receiver blocking and desense.
- Harmonics and intermodulation.
- Shared-clock interference.
- Wi-Fi/Bluetooth coexistence control.
- LTE/cellular transmit effects on GNSS and sensors.
- LoRa duty cycle and regional parameters.
- Retries and battery impact.

Low-power design

Power-state planning

- Active.
- Processing burst.
- Radio transmit/receive.
- Idle.
- Light sleep.
- Deep sleep.
- Shipping mode.
- Charger-only mode.
- Fault mode.

For each state document:

- Enabled rails and peripherals.
- Clock source.
- RAM retention.
- GPIO levels.
- Wake sources.
- Entry/exit latency.
- Expected current.
- Maximum residency or duty cycle.

Common leakage paths

- Regulator quiescent current.
- Divider resistors.
- Pull-ups.
- LEDs.
- Sensors not truly powered down.
- Back-powering through GPIO or communication pins.
- USB and debugger interfaces.
- Battery monitor.
- Level shifters.
- External flash standby current.
- Floating inputs.
- Charger reverse leakage.

Measurement

- Use a current analyzer or source-measure unit with enough dynamic range.
- Capture transition and radio peaks, not only average current.
- Measure all modes at voltage and temperature corners.
- Remove debugger and indicator effects.
- Integrate charge over realistic duty cycles.
- Include battery self-discharge and regulator loss.
- Verify wakeup and failed-connection scenarios.

Nonvolatile storage

- Endurance per location.
- Total expected write count.
- Data retention versus temperature.
- Power-loss behavior.
- Atomic update mechanism.
- Wear leveling.
- Error detection/correction.
- Versioned data format.
- Default recovery.
- Migration between firmware versions.
- Factory reset policy.
- Secure erase limitations.

- Bad-block handling.
- Filesystem corruption recovery.
- Log rotation.
- Unique identifier and calibration protection.

Avoid repeatedly writing a counter or log header to one flash page without an endurance calculation. Test power removal during every stage of an update or filesystem operation.

Watchdogs and diagnostics

Watchdog design

- Prefer an independent clock source where possible.
- Use a timeout based on measured worst-case timing.
- Feed only after all critical tasks report health.
- Do not feed blindly from a timer interrupt.
- Record reset cause.
- Detect reset loops.
- Escalate repeated failures to recovery mode.
- Test watchdog operation deliberately.
- Consider window watchdogs for too-early as well as too-late servicing.

Self-test and diagnostics

- Power-on self-test.
- RAM and flash integrity tests where required.
- Clock monitor.
- Supply and temperature monitoring.
- Sensor plausibility.
- Communication loopback.
- Actuator feedback.
- Stack high-water marks.
- Heap and resource statistics.
- Brownout and watchdog counters.
- Persistent crash record.
- Firmware and hardware revision report.
- Manufacturing-test results.
- Calibration validity.
- Safe, rate-limited logs.

Firmware testing

Test levels

- Pure host unit tests.
- Driver tests with mocks or fakes.
- Hardware abstraction contract tests.
- Simulation and emulation.
- Hardware-in-the-loop tests.

- Board-level integration tests.
- System tests.
- Long-duration soak tests.
- Timing and stress tests.
- Fault-injection tests.
- Power-cycle and brownout tests.
- Update and rollback tests.
- Security and fuzz tests.
- Environmental chamber tests with firmware running.

Test cases often missed

- Counter and timestamp wraparound.
- Empty and maximum-length packet.
- Duplicate command.
- Out-of-order message.
- Interrupted flash write.
- Reset during bootloader swap.
- Full queue, filesystem or log storage.
- Sensor disconnected, shorted or saturated.
- Communication peer absent or noisy.
- Clock missing or out of tolerance.
- Low battery during radio transmission.
- Rapid connect/disconnect.
- Debugger not attached.
- Compiler optimization differences.
- Temperature-corner oscillator behavior.
- Repeated sleep/wake cycles.
- Factory-reset interruption.
- Invalid calibration record.

Laboratory tools

Core bench

- Current-limited adjustable power supply.
- Digital multimeter.
- Oscilloscope with suitable bandwidth and probes.
- Logic analyzer.
- Soldering and hot-air rework equipment.
- Microscope.
- ESD-safe workstation.
- Bench cables, probes and breakout boards.
- Electronic load.
- Frequency counter or oscilloscope timing reference.
- Programmer/debugger.

Advanced tools

- Source-measure unit.
- Power analyzer.
- Differential and current probes.
- Spectrum analyzer.
- Vector network analyzer.
- RF signal generator.
- Thermal camera.
- Environmental chamber.
- Vibration table.
- Hipot and insulation tester.
- LCR meter and impedance analyzer.
- TDR.
- Protocol analyzers for USB, Ethernet, CAN and others.
- X-ray inspection.
- Boundary-scan equipment.
- EMC near-field probes, LISN and pre-compliance setup.

Measurement discipline

- State the probe, bandwidth limit and coupling.
- Minimize probe ground-loop inductance.
- Use differential probes for non-ground-referenced nodes.
- Respect instrument input limits and safety category.
- Calibrate probes and compensate them.
- Do not attach an earth-grounded oscilloscope clip to a live floating high-voltage node.
- Record setup photographs and firmware revision.
- Save raw data, not only screenshots.
- Distinguish measurement uncertainty from product tolerance.
- Reproduce failures before changing the design.

Prototype bring-up

Before power

- Inspect assembly under magnification.
- Verify component orientation and polarity.
- Check for solder bridges and missing parts.
- Compare assembled variant with BOM.
- Measure resistance from each power rail to ground.
- Check continuity across fuses, links and critical paths.
- Verify no short across isolation barrier.
- Inspect regulator feedback values.
- Confirm supply polarity and expected current limit.
- Review first-power sequence with another engineer.

First power

1. Use a current-limited supply.
2. Start at the safest valid input and conservative current limit.
3. Monitor input current.
4. Check each rail in sequence.
5. Verify reset and clock.
6. Check component temperature.
7. Confirm debugger connection.
8. Run minimal firmware before full application.
9. Test one peripheral at a time.
10. Save measured values in a bring-up table.

Bring-up order

- Power rails.
- Reset and clocks.
- Debug interface.
- Minimal GPIO.
- Console or diagnostic channel.
- External memory.
- Sensors and analog chain.
- Wired communication.
- Radio.
- Actuators and high-current loads.
- Sleep and wake.
- Bootloader and update.
- Fault and recovery behavior.

Debug method

- Reproduce consistently.
- Reduce to the smallest failing subsystem.
- Compare failing and working units.
- Form a testable hypothesis.
- Measure before modifying.
- Change one variable at a time.
- Record result.
- Distinguish schematic, layout, component, assembly, firmware and test-equipment causes.
- Convert the final cause into a design rule or regression test.

Validation and qualification

Corner testing

- Minimum/nominal/maximum input voltage.
- Minimum/room/maximum temperature.
- Component tolerance corners.
- Maximum processing and radio load.

- Minimum and maximum external loads.
- Slow and fast supply ramps.
- Different cables, antennas and accessories.
- Fresh and aged batteries.
- Worst enclosure and airflow configuration.

Reliability testing

- Temperature cycling.
- Thermal shock.
- High-temperature operating life.
- High-temperature/humidity exposure.
- Vibration.
- Mechanical shock.
- Drop.
- Connector insertion cycles.
- Button and relay life.
- Cable flex.
- Power cycling.
- Battery cycling.
- Long-duration operation.
- Corrosion and salt mist where relevant.
- Highly accelerated testing when correctly planned.

Failure analysis

- Preserve failed units and configuration.
- Record exact test history.
- Photograph and X-ray before destructive work.
- Compare electrical signatures.
- Localize heat and emissions.
- Inspect solder joints and vias.
- Cross-section if necessary.
- Analyze firmware logs and crash dumps.
- Determine root cause, not merely the failed component.
- Check whether the failure affects other batches or revisions.
- Implement corrective and preventive action.

Safety

Hazard categories

- Electric shock.
- Fire and smoke.
- Excessive temperature.
- Battery thermal runaway.
- Moving mechanisms and pinch points.
- Unexpected actuator motion.

- Optical or laser radiation.
- RF exposure.
- Acoustic energy.
- Pressure or stored mechanical energy.
- Chemical exposure.
- Loss of essential function.

Safety design

- Identify hazards before architecture freeze.
- Define safe states.
- Use independent protection for severe hazards.
- Detect open and short sensor faults.
- Design single-fault tolerance when required.
- Apply fuses, thermal cutoffs and current limits correctly.
- Avoid relying only on normal application software for critical protection.
- Use plausibility and cross-checking.
- Make dangerous overrides controlled and logged.
- Validate startup and shutdown states.
- Analyze protection-component failure.
- Define maintenance and calibration intervals.
- Preserve safety during firmware update.

Sector standards

The applicable standard depends on product and market. Examples include:

- IEC 61508 for generic functional safety.[1]
- ISO 26262 for road-vehicle E/E functional safety.[37][38]
- IEC 62304 for medical-device software.
- IEC 60601 family for medical electrical equipment.
- IEC 62368-1 for audio/video, information and communication equipment safety.
- IEC 61010-1 for measurement, control and laboratory equipment.
- IEC 60335 family for household appliances.
- DO-254 and DO-178C in civil airborne electronic hardware and software contexts.
- EN 50128/EN 50716 and related railway standards.
- UL and CSA product-specific safety standards.

Do not select a standard from a generic list alone; obtain a formal applicability review for the actual product, energy sources, users, environment and target countries.

Regulatory planning

Market-access topics

- Product safety.
- EMC emissions and immunity.
- Radio spectrum and RF exposure.
- Environmental substances and material declarations.
- Battery transport and safety.

- Energy efficiency.
- Telecom or network approval.
- Automotive type approval.
- Medical-device regulation.
- Cybersecurity requirements.
- Labeling, language and user instructions.
- Importer and authorized representative obligations.
- Technical-file retention.

Compliance strategy

- Identify markets before design freeze.
- Create a standards matrix.
- Choose the exact product-family standards.
- Confirm editions and transition dates.
- Use certified modules and components only where their approvals genuinely reduce end-product work.
- Preserve certificates and conditions of use.
- Build pre-compliance samples early.
- Reserve board footprints for filters and tuning.
- Test the final enclosure and cables.
- Control all certified hardware and firmware changes.
- Determine whether changes require partial or full retest.
- Maintain declarations, labels, manuals and evidence.

Production control

Pilot build

- Use production-intent suppliers, tooling and test fixtures.
- Record every deviation.
- Measure first-pass yield.
- Classify defects by process and design cause.
- Review solder profile.
- Check MSL and ESD control.
- Validate programming and provisioning throughput.
- Validate calibration repeatability.
- Audit serialization and traceability.
- Confirm rework limits.
- Run sample environmental and functional tests.
- Update work instructions from operator feedback.

Process control

- Approved vendor list.
- Incoming inspection.
- Lot traceability.
- Controlled solder paste and storage.
- Reflow profile verification.

- Stencil maintenance.
- AOI/X-ray criteria.
- ESD audits.
- Tool calibration.
- Firmware checksum control.
- Golden-unit management.
- Statistical process control where useful.
- Nonconformance and material-review process.
- Corrective and preventive action.
- Engineering change control.

Traceability

Link each serial number to:

- PCB fabrication lot.
- Assembly lot and date.
- Critical component lots.
- Hardware revision.
- Firmware and bootloader version.
- Provisioned identity or certificate reference.
- Calibration data.
- Production test results.
- Rework history.
- Shipment and warranty history.

Documentation

Engineering documents

- Product and system requirements.
- Hardware requirements.
- Firmware requirements.
- Architecture description.
- Interface control document.
- Schematics.
- PCB database and manufacturing package.
- BOM and approved parts list.
- Calculations and simulation reports.
- Threat model.
- Hazard analysis and FMEA.
- Verification plan and report.
- Compliance matrix.
- Bring-up procedure.
- Programming and provisioning procedure.
- Calibration procedure.
- Production-test specification.
- Service manual.

- Known errata.
- Release notes.

Configuration management

Treat these as one controlled product configuration:

- Schematic revision.
- PCB revision.
- BOM revision.
- Assembly variant.
- Bootloader.
- Application firmware.
- FPGA image.
- Radio configuration.
- Factory test software.
- Calibration algorithm.
- Enclosure and cable revision.
- Manufacturing instructions.

Use version control for firmware, scripts, text-based design files and documentation. Tag released configurations, archive exact compiler and dependency versions, and make release binaries reproducible where possible.

Change management

Every engineering change should answer:

- Why is the change needed?
- Which requirements and risks are affected?
- Is the replacement truly equivalent?
- Does pinout, timing, leakage, RF, thermal or firmware behavior change?
- Must PCB, BOM, software or test limits change?
- Does it affect safety, security or compliance?
- Does it require new calibration?
- Are existing units and spare parts compatible?
- Which tests must be repeated?
- How will old and new inventory be separated?
- What is the rollback plan?

Minor-looking changes such as capacitor dielectric, PCB material, antenna supplier, enclosure pigment, crystal, bootloader or conformal coating can affect EMC, RF, timing, safety or reliability.

Field operation

Maintainability

- Remote diagnostics.
- Safe firmware updates.
- Hardware and firmware version reporting.
- Reset-cause counters.

- Health and battery metrics.
- Calibration status.
- Replaceable modules where justified.
- Service connectors protected from misuse.
- Spare-parts and repair policy.
- Repair-versus-replace economics.
- User-visible fault codes.
- Privacy-conscious logs.

Vulnerability management

- Publish a security contact.
- Maintain an inventory of firmware and third-party components.
- Monitor vendor advisories.
- Triage reports.
- Reproduce and score impact.
- Develop, sign and test fixes.
- Deploy in controlled stages.
- Revoke compromised keys or devices.
- Communicate support duration.
- Preserve recovery for failed updates.

End of life

- Last-time-buy decision.
- Replacement-product migration.
- Final firmware and security update.
- Cloud service shutdown plan.
- Certificate and credential revocation.
- User data export and deletion.
- Battery removal and recycling.
- Spare parts and service duration.
- Documentation archive.
- Regulatory and warranty retention.

Niche failure modes

- Ceramic capacitor loses most nominal capacitance under DC bias.
- MLCC cracks from board flex.
- Electrolytic capacitor life collapses near a heat source.
- Inductor saturates during startup although average current is safe.
- MOSFET exceeds safe operating area during linear startup.
- Connector pin carries more current than the connector-level average suggests.
- Grounded oscilloscope probe creates a short.
- Debugger powers an otherwise unpowered MCU through I/O.
- GPIO back-powers a sensor through protection diodes.
- Reset pin floats during slow ramp.
- Boot strap is changed by an attached peripheral.

- Crystal fails to start at cold temperature.
- Watchdog is serviced by a healthy interrupt while the application is deadlocked.
- Flash log wears one sector early.
- DMA writes to a buffer after it is reused.
- Cache coherency corrupts DMA data.
- Compiler optimization reveals a race hidden in debug builds.
- Timer arithmetic fails at wraparound.
- Interrupt storm starves normal tasks.
- Equal-priority tasks behave differently after timing changes.
- Queue fills during a network outage and exhausts memory.
- Radio transmission browns out the MCU.
- Antenna tuning changes after the enclosure, coating or battery is installed.
- USB ESD diode capacitance closes the eye diagram.
- Common-mode choke resonates and worsens emissions.
- Plane split interrupts return current.
- Stitching via is too far from a layer transition.
- Long test-point stub creates reflection.
- Via-in-pad wicks solder unless filled correctly.
- Thermal pad floats the package during reflow.
- Copper imbalance warps the PCB.
- Panel tabs crack nearby MLCCs during depanelization.
- Moisture-sensitive package popcorning causes latent failure.
- No-clean flux residue becomes conductive in humidity.
- Conformal coating traps contamination.
- Potting compound stresses an oscillator or sensor.
- Magnetometer is disturbed by steel screws.
- Temperature sensor measures a regulator rather than ambient air.
- Barometric sensor vent is blocked by adhesive.
- Pressure equalization membrane changes acoustic response.
- Cable shield is connected in a way that injects ESD into logic ground.
- TVS clamps above the protected IC’s survival limit.
- Fuse does not open before a trace overheats.
- Reverse polarity protection fails with an external programming cable attached.
- Hot plugging causes regulator overshoot.
- Battery protector and charger enter a recovery deadlock.
- Fuel gauge is inaccurate after a cell supplier change.
- I2C bus remains stuck after a slave resets mid-byte.
- RS-485 nodes use incompatible bias and termination.
- CAN transceiver standby pin floats during boot.
- SPI MISO drivers contend during reset.
- UART bootloader remains exposed in production.
- JTAG is disabled before a recoverable manufacturing test path exists.
- Anti-rollback prevents legitimate factory recovery.
- Universal device key compromises the entire fleet.
- Entropy is weak at first boot.

- Certificate expires after devices are deployed offline.
- Production test records pass the wrong serial number.
- Golden unit silently drifts.
- Test fixture pogo resistance creates false failures.
- Firmware update succeeds but configuration migration fails.
- Old application boots with a new incompatible bootloader.
- Rollback firmware cannot interpret the new storage format.
- Brownout corrupts calibration during factory programming.
- EMC fix reduces radio range or increases thermal load.
- A second-source part has different startup, leakage or pin behavior despite the same headline specification.

Tool map

Task	Useful tool categories and examples
Schematic and PCB	KiCad, Altium Designer, OrCAD/Allegro, Siemens Xpedition
Circuit simulation	LTspice, PSpice, TINA-TI, ngspice, SIMetrix
SI/PI	Keysight ADS, Ansys SIwave/HFSS, Cadence Sigrity, HyperLynx, vendor IBIS tools
RF	ADS, HFSS, CST, Sonnet, QUCS-S, NanoVNA/VNA for practical work
Firmware IDE/build	CMake, Ninja, Make, vendor IDEs, VS Code, PlatformIO, west/Zephyr tools
Debug	GDB, OpenOCD, pyOCD, J-Link, ST-Link, SWO, SEGGER SystemView, Tracealyzer
Static analysis	compiler warnings, clang-tidy, cppcheck, CodeQL, Coverity, MISRA tools
Unit test	Ceedling/Unity/CMock, GoogleTest, Catch2, pytest for host/tooling
HIL automation	Python, pytest, Robot Framework, lab instrument APIs
Protocol analysis	Wireshark, Saleae tools, CAN analyzers, USB protocol analyzers
Firmware security	binwalk, Ghidra, FACT, fuzzers, OWASP IoT/FSTM methods
Manufacturing	DFM viewers, Valor/NPI tools, CAM review, AOI, X-ray, flying probe
Documentation	Git, issue tracker, requirements tool, Markdown/Sphinx/Doxygen, CAD PLM

Tools do not replace engineering judgment. A simulator is only as good as its models and boundary conditions; ERC/DRC can detect encoded rule violations but cannot know whether the architecture or selected rule is correct.

Practical project workflow

For an STM32, ESP32, LoRa, sensor, telemetry or CubeSat-style student/prototype project, a disciplined but affordable workflow is:

1. Write measurable power, interface, RF, environmental and firmware requirements.
2. Draw the block diagram, power tree, interface table and state diagram.
3. Identify the top ten electrical, supply-chain, RF and firmware risks.
4. Prototype uncertain analog, power and RF blocks separately.

5. Build a complete worst-case power budget and pin-multiplexing table.
6. Select parts with lifecycle and alternates in mind.
7. Create verified symbols and footprints from current datasheets.
8. Complete and peer-review the schematic.
9. Run ERC and justify every warning.
10. Agree stack-up and capabilities with the selected fabricator.
11. Place by function and current loop before routing.
12. Route clocks, power, analog, RF and high-speed interfaces with controlled returns.
13. Add test points, current links and programming access.
14. Run DRC, 3D mechanical review and manufacturer DFM/DFA.
15. Review all manufacturing outputs independently.
16. Bring up with current limiting and a written sequence.
17. Validate rails, reset, clocks and low-level drivers before the complete application.
18. Add automated host tests and hardware regression tests.
19. Measure sleep/transmit current, thermal behavior, SI and RF range.
20. Perform ESD/EMC pre-compliance before design freeze.
21. Test firmware updates, power interruption and recovery.
22. Run environmental and long-duration tests appropriate to the mission.
23. Freeze a complete, traceable release package.
24. Use pilot production to discover DFM, fixture and process problems.
25. Track field failures and feed them back into requirements and tests.

Design-review gates

Requirements review

- Every requirement is measurable.
- Normal, abnormal and environmental conditions are defined.
- Regulatory scope is known.
- Safety and security risks have owners.
- Cost, volume and lifecycle targets are realistic.

Schematic review

- Every pin is accounted for.
- Power, reset, clock and boot circuits match datasheets.
- Ratings and tolerances cover worst case.
- Protection and isolation are justified.
- Test and programming access exists.
- Symbols and footprints were checked independently.
- ERC findings are resolved.

Layout review

- Stack-up is manufacturer-approved.
- Critical loops and return paths are controlled.
- Decoupling and converter placement follow current paths.
- Analog, RF and switching regions are managed.

- Thermal and mechanical constraints are satisfied.
- Creepage, clearance and edge distances are correct.
- Testability, assembly and rework are practical.
- DRC and manufacturing reviews pass.

Firmware review

- State and fault behavior are explicit.
- Timing, memory and power budgets are measured.
- Concurrency and interrupt design are understood.
- Inputs are bounded and validated.
- Watchdog detects complete-system health.
- Update and recovery survive interruption.
- Debug and credentials are production-safe.
- Tests cover boundaries, failures and rollback.

Production-release review

- Verification and validation evidence is complete.
- Compliance tests pass.
- Pilot yield and defects are acceptable.
- Test limits and fixtures are controlled.
- Firmware and keys are provisioned securely.
- Documentation matches the released configuration.
- Backup suppliers and EOL plans exist.
- Field update, support and incident plans are ready.

Final release checklist

- Requirements, architecture and risks are approved.
- BOM is costed, available and lifecycle-reviewed.
- Parts operate within recommended limits with suitable margin.
- Power, timing, memory, thermal and RF budgets close.
- Schematic and layout have independent peer review.
- ERC, DRC and DFM/DFA findings are resolved.
- Footprints, polarity and connectors are physically verified.
- SI, PI and EMC risks are measured or simulated appropriately.
- Safety insulation and single-fault behavior are verified.
- Firmware has static analysis, automated tests and controlled builds.
- Secure boot, update, recovery and credential provisioning are tested when applicable.
- Prototype corner, fault, power-cycle and soak tests pass.
- Environmental and compliance testing match the actual use case.
- Manufacturing inspection, programming, calibration and functional test are validated.
- Serial traceability works end to end.
- Documentation identifies exact hardware, firmware and manufacturing revisions.
- Rollback, service, vulnerability and end-of-life plans exist.

The central engineering principle is to design the complete lifecycle rather than only the circuit or code. Strong embedded products make requirements measurable, control current and return paths, anticipate

physical and software faults, reserve observability and test access, verify behavior at environmental and tolerance corners, and preserve traceability from the first architecture decision through the last field update.

References

1. Safety and functional safety - IEC - The IEC 61508 series provides functional safety standards for the lifecycle of electrical, electroni...
2. Common Microcontroller Software Interface Standard (CMSIS) - Arm - CMSIS enables consistent device support and simple software interfaces to the processor. CMSIS is pu...
3. FreeRTOS scheduling (single-core, AMP and SMP) - By default, FreeRTOS uses a fixed-priority preemptive scheduling policy, with round-robin time-slici...
4. GitHub - zephyrproject-rtos/zephyr: Primary Git Repository for the ... - Primary Git Repository for the Zephyr Project. Zephyr is a new generation, scalable, optimized, secu...
5. [PDF] Preferred Reliability Practices - EEE Parts Derating - NASA - Derating lowers the probability of failures occurring during assembly, test, and flight. Decreasing ...
6. [PDF] Assurance of Electronic Parts for Aerospace System Reliability - Derating is when a component is designed to operate component. Typically, derating reduces the degra...
7. Getting Started in KiCad | master | English | Documentation - KiCad's Electrical Rules Checker (ERC) cannot make sure that the design in the schematic will work, ...
8. [PDF] TPS65950 Layout Guide - Texas Instruments - The design model is a proven design. The guide is organized as a design process: • Definition of the...
9. [PDF] High Speed Layout Guidelines (Rev. A) - Texas Instruments - Decoupling capacitors between the power pin and ground pin of the device ensure low AC impedance to ...
10. UN 38.3 Testing for Lithium Batteries - Intertek - Intertek offers comprehensive UN 38.3 Testing solutions for the transportation of lithium and lithiu...
11. [PDF] - 394 - 38.3 Lithium metal and lithium ion batteries 38.3.1 ... - UNECE - 38.3.1. Purpose. This section presents the procedures to be followed for the classification of lithi...
12. Compliance | USB-IF - Compliance USB-IF Compliance Program Please review the USB-IF Compliance Updates webpage frequently ...
13. Compliance Tools | USB-IF - SigTest is the official tool for USB transmitter voltage, LFPS, and Signal Quality electrical compli...
14. Successful PCB Grounding with Mixed-Signal Chips - Analog Devices - A basic tutorial on proper printed circuit board (PCB) layout for grounding of mixed-signal designs ...
15. [PDF] Grounding in mixed-signal systems demystified, Part 2 - So for any PCB layout, the important points are to use a single ground plane, partition it into anal...
16. Basic EMC publications - IEC - EMC publications specify the general conditions and rules necessary for achieving electromagnetic co...
17. ESD: Electrostatic Discharge - JEDEC - JEDEC has taken a leadership role in developing standards for ESD since the early 1980s, including s...
18. Part 6: ESD Standards - ESD Association - This standard establishes performance limits for bags intended to protect electronic parts and produ...

19. Equipment Authorization | Federal Communications Commission - Radio Frequency (RF) devices are required to be properly authorized under 47 CFR part 2 prior to bei...
20. EN - Europese Commissie - The RED Delegated Regulation on cybersecurity rendered applicable, as of 1 August 2025, the essentia...
21. Equipment Authorization – RF Device | Federal Communications ... - The FCC regulates radio frequency (RF) devices contained in electronic-electrical products that are ...
22. Delegated regulation - 2022/30 - EN - EUR-Lex - As stated in recital 13 of Directive 2014/53/EU, the protection of personal data and privacy of user...
23. Clearance and Creepage Distance in PCB Design - To avoid such failures, clearance and creepage distances must be calculated based on the actual work...
24. [PDF] Demystifying clearance and creepage distance for high-voltage end ... - Definitions of creepage distance (CPG), clearance distance (CLR) and other relevant terms. • Differe...
25. IEC 60068-1:2013 - IEC 60068-1:2013 includes a series of methods for environmental testing along with their appropriate...
26. IPC Standards | electronics.org - IPC standards help ensure superior quality, reliability and consistency in electronics manufacturing...
27. IPC J-STD-001J - 1.1 Scope This standard describes materials, methods and acceptance criteria for producing soldered ...
28. JOINT IPC/JEDEC STANDARD FOR HANDLING, PACKING, SHIPPING, AND ... - The purpose of this document is to provide manufacturers and users with standardized methods for han...
29. Handling, Packing, Shipping and Use of Moisture, Reflow, and ... - This document describes the standardized levels of floor-life exposure for moisture/reflow sensitive...
30. [PDF] JEDEC STANDARD - This standard establishes the minimum requirements for Electrostatic Discharge (ESD) control methods...
31. FreeRTOS mutexes - FreeRTOS implements a basic priority inheritance mechanism which was designed to optimize both space...
32. MISRA C - Guidelines for the use of the C language in ... - MISRA C is a widely adopted set of guidelines (aka “standard”) for developing software in the C lang...
33. SP 800-193, Platform Firmware Resiliency Guidelines | CSRC - This document provides technical guidelines and recommendations supporting resiliency of platform fi...
34. Platform Firmware Resiliency Guidelines - NIST - This document provides technical guidelines and recommendations supporting resiliency of platform fi...
35. OWASP Firmware Security Testing Methodology - GitHub - The Firmware Security Testing Methodology (FSTM) is composed of nine stages tailored to enable secur...
36. OWASP Internet of Things - The Firmware Security Testing Methodology (FSTM) is composed of nine stages tailored to enable secur...
37. ISO 26262 | Automotive functional safety - TÜV SÜD - ISO 26262 is the international standard for automotive functional safety. It defines a framework for...
38. Functional safety for road vehicles - ISO 26262 - DNV - ISO 26262 is a multipart standard defining requirements and providing guidelines for achieving funct...