AI-ASSISTED BUILD CHEAT SHEET

Building Software with AI Without Creating Chaos

Move quickly while learning the product, then deliberately shift into a maintainable build mode.

Core idea: Prototype to learn. Freeze to build. Give AI bounded jobs. Turn every meaningful failure into permanent coverage. The danger is not AI speed itself - it is letting fast changes outrun architecture, ownership, and tests.

The phase change that matters

1
Prototype

Goal: discover the product and interaction model.

AI: explore, vary, challenge, generate disposable code.

Exit when: you can explain what the product is and which behaviors matter.

2
Transition

Goal: convert learning into a build contract.

AI: audit, identify invariants, map ownership and source-of-truth paths.

Exit when: golden behaviors and architecture boundaries are explicit.

3
Real build

Goal: implement reliably without losing product lessons.

AI: bounded slices, tests, QA, conflict removal.

Exit when: automated coverage and deployed smoke checks pass.

Before AI writes production code

SOURCE OF TRUTH Exact frontend, backend, schema, migration, and deploy paths.

BEHAVIORAL CONTRACT What must always remain true regardless of model output.

GOLDEN BEHAVIORS Representative user journeys with expected outcomes written first.

OWNERSHIP MAP Which layer owns routing, validation, persistence, rendering, and authority.

PROVIDER SPIKE Real schema support, latency, cost, and failure modes.

SMOKE GATE The few deployed journeys that prove the product is basically usable.

Repository clarity is part of AI coding quality. If two files look equally authoritative, an AI coding assistant can confidently change the wrong one. Clean paths and explicit ownership reduce both human and model mistakes.

Keep the architecture boring

Useful complexity

Product rules, authority boundaries, meaningful state transitions, and user behavior learned through use.

Avoidable complexity

Duplicate runtime paths, patch-on-patch styling, stale configs, hidden fallbacks, unclear routing, multiple owners for the same behavior.

AI-ASSISTED BUILD CHEAT SHEET

Building Software with AI Without Creating Chaos

Use AI differently depending on the job, and debug the whole system before reaching for a prompt change.

Use AI as four different jobs

ARCHITECT
Find the owning layer, invariants, dependencies, and simplest design **before** changing code.

BUILDER
Implement one bounded slice. Preserve the contract. Prefer removing conflicts to adding overrides.

QA
Assume the builder missed something. Reproduce, probe adjacent paths, and identify permanent coverage.

PRODUCT
Use it like a real user. Separate product confusion from implementation defects and model failures.

The repair loop

Reproduce → **Failing test** → **Find owning layer** → **Remove conflict** → **Fix** → **Full suite + deployed check**

> **A bug that mattered once should not depend on someone remembering to manually check it forever.**

Three State lessons worth keeping

Symptom	Actual cause	Reusable lesson
Wrong factual answer	Wrong frontend execution path; model was not the core failure.	Verify routing, retrieval, context, and authority before prompt-engineering.
Broken deployed modal	Conflicting CSS/JS visibility mechanisms and accumulated overrides.	Do not patch a symptom with another mechanism. Pick one owner.
AI calls felt slow	Provider inference dominated latency.	Instrument first. Optimize the actual bottleneck; use interaction design and real streaming when latency remains.

AI-specific build rules

- ✓ Models make semantic judgments; software should own mechanical facts when possible.
- ✓ Adapt provider output to the product contract, not the reverse.
- ✓ Schema-valid is not the same as semantically correct.
- ✓ Capability does not imply authority: recommend, validate, authorize, then mutate.
- ✓ Use real product usage to discover useful rework.
- ✓ Repeated implementation debt means the process or ownership model needs to change.

Move fast while discovering the product. Then make implementation boring: one source of truth, bounded AI jobs, explicit phase gates, permanent regression coverage, and deployed-browser verification.