

From MVP Specification to Working Code with Cursor

A practical workflow for turning an approved product definition into software you can trust

AI can write code quickly. The challenge is building the right product.

AI coding tools such as Cursor can accelerate software development, but speed alone does not guarantee a successful MVP. If the product definition is incomplete, contradictory or out of date, an AI tool can implement the wrong solution very efficiently. The result may compile and look convincing while still failing to solve the intended problem.

The way to reduce this risk is to connect product thinking and implementation through a clear, approved specification and a disciplined development workflow. Cursor should help translate decisions into working software—not silently make important product decisions on your behalf.

1. Start with the approved specification—not a coding prompt

Before asking Cursor to implement a feature, establish which document is the authoritative product specification. Confirm its version and approval status, and identify the supporting documents that explain architecture, design, data structures or implementation constraints.

This matters because a project may contain several kinds of documents: an app specification, a technical design, a task plan, experiments, and notes from earlier discussions. They do not necessarily have equal authority. An implementation plan generated by an AI assistant is not automatically the source of truth simply because it is detailed or recent.

Make the hierarchy explicit. The approved specification defines what the product must achieve and what belongs in the MVP. Technical documents explain how the solution is intended to work. The implementation plan breaks the work into manageable steps. If these sources conflict, pause and resolve the conflict rather than letting Cursor guess.

2. Prepare the project before implementation

Give Cursor the context it needs before asking it to change code. Confirm the relevant repository, framework, existing architecture, environment configuration, and current state of the application. Ask Cursor to inspect the project and report what already exists before proposing changes.

Translate the specification into a practical implementation checklist. For each capability, identify the user outcome, the relevant requirements, acceptance criteria, dependencies, and any technical boundaries. Record unresolved questions instead of filling them with assumptions.

This preparation reduces rework. It also helps distinguish a genuinely missing capability from one that is already implemented but needs testing, refinement or documentation.

3. Use Cursor as an implementation partner—not as the product owner

Cursor is most useful when the task is bounded and the expected result is clear. Instead of asking it to “build the whole app,” give it a specific task, the relevant context, constraints, and a definition of done.

A good task explains what should change, what must remain unchanged, which files or components are in scope, and how the result will be verified. Ask Cursor to outline its approach before consequential changes, particularly when a task affects architecture, data persistence, authentication, or multiple parts of the application.

Keep the human decision-maker responsible for product scope, trade-offs, and approval. When requirements are ambiguous, ask for the ambiguity to be surfaced and resolved. Do not allow plausible AI-generated assumptions to become product requirements by accident.

4. Build in small, verifiable increments

Implement the MVP in steps that can be reviewed and tested independently. The exact sequence depends on the application, but a useful pattern is:

- **Foundation:** Confirm the project runs, dependencies are understood, configuration is safe, and the development workflow is repeatable.
- **Core data model:** Implement the entities, relationships, validation rules, and persistence needed for the central user journey.
- **Core user journey:** Build the smallest end-to-end path that delivers the product’s primary value.
- **Supporting capabilities:** Add secondary features only when the core journey is working and verified.
- **Integration and hardening:** Check error handling, security, accessibility, performance, edge cases, and regression risks.

After each increment, review the code changes, run the relevant checks, and compare the result with the specification. Small steps make defects easier to locate and reduce the risk that a large batch of AI-generated changes hides an incorrect assumption.

5. Turn acceptance criteria into the bridge between specification and code

A requirement states what the product needs to do; acceptance criteria make the expected behavior testable. They help both the developer and the AI coding tool distinguish a completed feature from a feature that merely appears to work.

For example, a requirement to create and save an idea might have acceptance criteria such as:

- A user can enter the required fields and submit a valid idea.
- The idea is saved and remains available after the page is refreshed or the user returns.
- Missing or invalid fields produce clear, actionable feedback.
- The interface communicates success or failure without losing entered information unexpectedly.
- Any access-control and data-ownership rules are enforced.

These criteria can guide implementation, manual testing, and automated tests. They also make reviews more objective: instead of asking whether a feature “looks finished,” ask whether every agreed behavior has been demonstrated.

6. Verify the result independently

Do not treat Cursor’s statement that a task is complete as proof that it is complete. Verify the result in the running application and with appropriate technical checks. Depending on the change, this may include:

- Running the build, type checks, linting, and relevant automated tests.
- Testing the actual user journey in the browser or target environment.
- Checking persistence, validation, permissions, and error states.
- Testing important edge cases and confirming that existing features still work.
- Comparing the implementation against the requirement and acceptance criteria.

A successful build is necessary but not sufficient. The application may build while implementing the wrong behavior, losing data, or violating an agreed scope boundary. Product-level verification is therefore just as important as technical verification.

7. Keep changes controlled and traceable

Use version control to make progress reversible. Review diffs before accepting changes, commit coherent increments, and avoid mixing unrelated refactoring with feature work. When a change introduces a new assumption or alters an earlier decision, update the relevant documentation and obtain approval where needed.

Maintain a traceable connection from the original problem to the implemented behavior:

Problem → User → Outcome → Job → Capability → Requirement → Acceptance Criterion → Metric

This chain helps answer an important question during reviews: why does this piece of code exist, and how will we know whether it delivers the intended value? If a feature cannot be connected to the agreed MVP goals, reconsider whether it belongs in the current release.

A repeatable workflow for specification-led development

Step	What to do
Validate	Confirm the authoritative specification, version, and approval status.
Inspect	Ask Cursor to describe the current codebase and relevant existing behavior.
Reconcile	Identify conflicts, missing decisions, dependencies, and scope boundaries.
Plan	Break the next increment into a bounded task with testable acceptance criteria.
Implement	Make the smallest coherent change that advances the agreed user journey.
Verify	Run technical checks and test the actual behavior against the criteria.

Reconcile	Update documentation and resolve any gap between specification and implementation.
Release	Only declare the increment complete when evidence supports that conclusion.

What does “working code” really mean?

Working code is more than code that compiles or a screen that looks complete. For an MVP, it means the agreed core experience works for its intended users, important data is handled correctly, errors are understandable, security and access rules are respected, and the application can be tested and maintained well enough for the next stage of learning.

It also means respecting the MVP boundary. Extra features can increase complexity and delay validation. The goal is not to build everything that might eventually be useful; it is to deliver the smallest credible product that can test the most important assumptions.

How MVPC helps

MVPC is designed to help founders and product builders move from an initial idea toward a clearer, more testable MVP definition. By clarifying the problem, target user, value proposition, product model, core experience, requirements, and MVP boundaries before implementation, the process creates a stronger foundation for technical planning and AI-assisted development.

That foundation does not replace engineering judgment or testing. It helps make both more effective: define what success means before asking AI to build, then verify the result before calling the MVP complete.

The principle to remember

Clarify before implementing. Plan before making consequential changes. Build in small increments. Verify against explicit criteria. Keep the approved specification as the reference point throughout development.

Cursor can help you move faster from specification to working code. A disciplined process helps ensure that you are moving in the right direction.