Requirements Before, During, or On the Fly?

Three ways engineers specify, what each costs, and how RAG-SVA keeps each one written down and checked

Ben Cohen · Dr. Kais Chibani · systemverilog.us · Ben_Cohen@SystemVerilog.us · November 2026

Abstract

Requirements reach code in one of three ways. They are written in full before design starts (*requirements-first: waterfall, V-model*). They are written in slices while the code is written (*iterative and agile: just-in-time or emergent requirements*). Or they are never written as a separate document at all and live in the engineer's head while code is written on the fly (*code-and-fix, exploratory or prototype-first work*). Each approach has real advantages, and each has a characteristic failure. This paper names the approaches, compares them, explains why hardware weighs the trade-offs differently from software, and shows how the Cohen-Chibani RAG-SVA ruleset turns requirements from any of the three into executable, line-item requirements: a clarification gate that refuses to guess, assertions and assumptions with stated intent, covers, and a read-back specification that commercial tools can trace.

1. The question

In software in particular, requirements are often written on the fly, or while the code is written. In other organizations they are written, reviewed and frozen before a single line of code exists. Both camps can point to projects that succeeded and to projects that failed. The useful question is not which camp is right, but what each approach buys, what it risks, and what can reduce the risk.

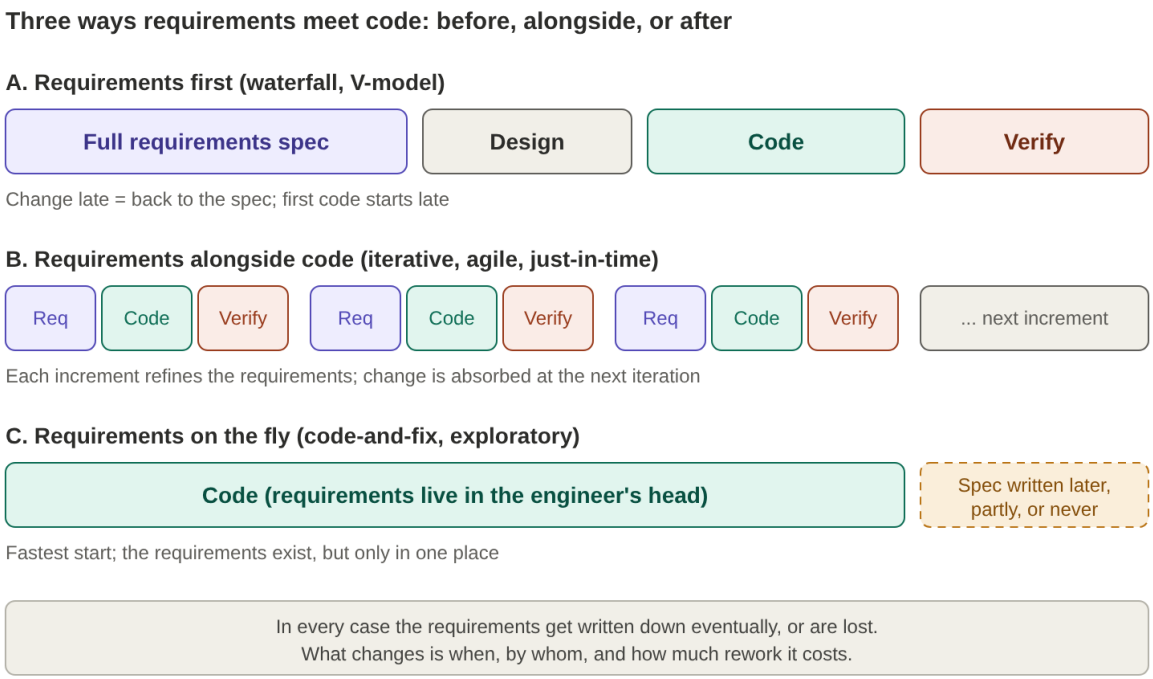


Figure 1. Three ways requirements meet code: before, alongside, or after.

2. The names for these processes

Approach	Common names	Origin and notes
A. Requirements first	Waterfall; V-model; Big Design Up Front (BDUF); requirements-driven or plan-driven development	Usually traced to W. W. Royce (1970), who drew the sequential phases but warned against running them strictly once. The V-model pairs each specification level with a verification level. Requirements practice is codified in ISO/IEC/IEEE 29148.
B. Requirements alongside code	Iterative and incremental development; Agile (Scrum, XP); just-in-time requirements; emergent requirements; progressive elaboration; rolling-wave planning; Spiral model	Spiral: B. Boehm (1988), risk-driven iterations. Agile Manifesto (2001): working software and responding to change are valued over comprehensive documentation and following a plan. Progressive elaboration and rolling-wave planning are project-management (PMI) terms.
C. Requirements on the fly	Code-and-fix; cowboy coding; exploratory programming; prototype-first; "the code is the spec"	Boehm described code-and-fix as the oldest model: write code, then fix it. In exploratory work this is often the honest choice: the requirement is not known until a prototype shows it.
Executable requirements (a bridge)	Test-Driven Development (TDD); Behavior-Driven Development (BDD); specification by example; assertion-based verification	TDD (K. Beck) writes a failing test before the code. BDD (D. North) writes scenarios in a structured Given/When/Then form. In hardware, assertions and assumptions play this role: the requirement is written in a form a tool can check.

The phrase most often used for approach C is **emergent requirements**: requirements that emerge as the work proceeds rather than being stated up front. When the emergence is unmanaged, people call it code-and-fix or cowboy coding; when it is managed in short, reviewed increments, it becomes approach B.

3. Approach A: requirements first

How it works

A requirements specification is written, reviewed and baselined. Design, implementation and verification follow, and each verification level traces back to a specification level. Changes go through a formal change process.

Advantages

- One shared, reviewed statement of intent before money is spent on implementation.
- Traceability from each requirement to design, test and sign-off; this is what standards such as DO-254 (airborne hardware), ISO 26262 (automotive) and IEC 61508 expect.
- The verification plan can be written early, in parallel with the design.
- Contracts, schedules and estimates have a defined scope.
- Knowledge survives turnover: the intent is on paper, not in one person's memory.

Disadvantages

- Slow to first working code.
- A wrong or incomplete specification is implemented faithfully; the error surfaces late, when it is most expensive to fix.
- The specification is written before the team has learned what the hard parts are; the hardest requirements are often the ones nobody thought to write.
- Documents drift: the specification and the implementation diverge unless someone keeps them aligned.
- Prose is ambiguous. "The output shall be valid within 70 ns" can be read several ways, and nothing checks the reading.

Where it fits

Safety-critical and regulated products, interfaces between teams or companies (a bus protocol, a datasheet), and silicon, where a mask set makes late change very costly.

4. Approach B: requirements alongside code

How it works

A coarse scope and a backlog are set first. Each short iteration refines a slice of requirements (*user stories, acceptance criteria, scenarios*), implements it, and verifies it. The requirements for later slices are written when the team knows enough to write them well.

Advantages

- Change is expected and absorbed at the next iteration.
- Feedback from working code and from users corrects the requirements early.
- Requirements are written when knowledge is highest, so they tend to be more accurate.
- Risk is reduced in steps; the Spiral model makes risk the explicit driver of each loop.

Disadvantages

- Without discipline, acceptance criteria stay informal and the requirements end up scattered across tickets, chat and code comments.
- Traceability for audits must be reconstructed; it is not a natural by-product.
- Cross-cutting requirements (*safety invariants, timing budgets, reset behavior, interface contracts*) do not fit neatly into one story and can be missed.
- Architecture decisions made in early increments can be expensive to undo later.

Where it fits

Software products with users who can give fast feedback; firmware and testbench development; hardware blocks whose interfaces are fixed but whose internals are still being explored.

5. Approach C: requirements on the fly

How it works

The engineer writes code directly from an understanding of the problem. Requirements exist, but implicitly: in the engineer's head, in the code, and perhaps in a few comments. A specification, if one is ever written, is written afterwards and describes what was built.

Advantages

- Fastest start; no process overhead.
- Ideal for exploration and prototypes, where the real requirement is not yet known.
- A single expert can move very quickly on a small, well-understood problem.

Disadvantages

- The intent is not reviewable: nobody can check a requirement that was never written.
- Knowledge leaves with the author.
- Tests, when they exist, check what the code does, not what it was supposed to do.
- Integration exposes mismatched assumptions between blocks, late.
- No evidence for compliance, and little basis for reuse.

Where it fits

Spikes, proofs of concept, throw-away experiments and one-person tools. The risk starts when a prototype quietly becomes the product.

6. The trade-offs side by side

Trade-offs at a glance

	A. Requirements first	B. Alongside code	C. On the fly
Time to first working code	Slow	Medium	Fast
Absorbing changing requirements	Poor	Good	Good
Shared understanding across a team	Strong	Good	Weak
Traceability and compliance evidence	Strong	Medium	Weak
Verification plan ready early	Strong	Medium	Weak
Risk of building the wrong thing	Low*	Low	High
Risk of late, costly rework	High if spec wrong	Low	High at integration
Knowledge survives the author	Strong	Medium	Weak

favorable depends unfavorable

* Low only if the up-front requirements are right; a wrong spec is built faithfully. Qualitative assessment, not measured data.

Figure 2. Qualitative comparison. The ratings are the authors' assessment, not measured data.

No column is favorable on every row. Requirements-first buys shared understanding, traceability and early verification planning at the price of speed and flexibility. On-the-fly work buys speed at the price of everything that depends on the intent being written down. The iterative approach sits between them, and how close it comes to either end depends on how carefully each increment's requirements are written.

7. Why hardware weighs the trade-offs differently

- **Cost of a late change.** Software can be patched after release; a bug in silicon may need a new mask set. The cost of building the wrong thing is much higher, which pushes hardware toward approach A for anything that reaches tape-out.
- **Concurrency and timing.** Hardware requirements are about cycles, ordering and simultaneous events. Prose is weakest exactly there.
- **Interfaces are contracts.** A bus protocol or a memory datasheet is a requirement that two independent parties must read the same way.
- **Verification is most of the effort.** The verification plan is derived from the requirements; missing requirements become missing checks.
- **Formal verification needs assumptions.** A formal tool proves an assertion only under stated assumptions about the environment. An assumption that was never written is a requirement that was never stated.

At the same time, hardware teams do write requirements on the fly: the RTL designer writes a block from a micro-architecture sketch, and the verification engineer writes checks from a reading of the RTL. That is approach C inside an approach A project.

8. The hidden requirement

Every approach produces requirements; the difference is where they end up. In approach A they are in a document that may drift from the code. In approach B they are spread across increments. In approach C they are in the code and in someone's head. The common failure is the **implicit requirement**: a decision that was made, is relied on, and is written nowhere a reviewer or a tool can see it. Examples from memory and interface work:

- Is the data bus driven while an address is changing, or only after it has been stable for t_AA?
- Is a write terminated by the first of two control signals going high, or by a specific one?
- Is the environment allowed to change the address during a write cycle? If not, that is an assumption, and it must be stated.

The aim, then, is not to choose one process for every project, but to make sure that whichever process is used, each requirement becomes explicit, unambiguous and checkable soon after it is decided.

9. How the RAG-SVA rules help

RAG-SVA is the Cohen-Chibani ruleset for generating SystemVerilog Assertions with an AI assistant. It was designed for assertion generation, but several of its rules act directly on the requirements problem. They work the same way whether the input is a full specification, the requirements for one increment, or intent stated by an engineer while coding.

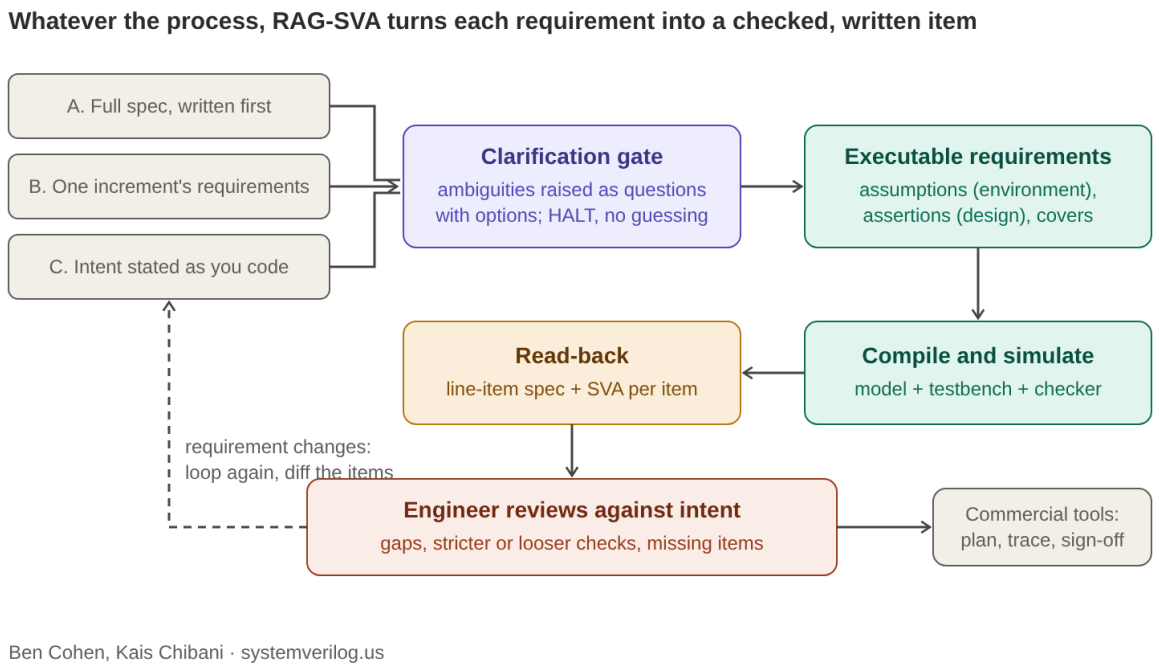


Figure 3. Requirements from any process enter the same loop and leave as written, checked line items.

RAG-SVA rule	What it does for requirements
Clarification gate (RQM0, HALT on ambiguity)	No code is generated while a requirement is ambiguous. The ambiguity is raised as a question with options. This is the step that converts an implicit requirement into an explicit decision, at the moment it matters. Zero tolerance: the assistant does not guess.
INTENTION and the PORT-INTENT table	Each signal is classified as environment (assumed) or design (asserted). This separates what the design promises from what it relies on, a distinction prose specifications often blur.
Assumptions with DOC9 headers	Every assumption carries a header stating why it exists and where it comes from. Assumptions are requirements on the environment, so they are documented and reviewable rather than hidden in a formal setup script.
Assertions as executable requirements	Each requirement becomes a labeled property (`ap_` for assertions, `mp_` for assumptions, `cp_` for covers) that a simulator or a formal tool checks. A requirement that can be checked cannot silently drift from the implementation.
Covers	Every assumption has a hit cover, and assertions have trigger and full covers, so a requirement that is never exercised (a vacuous pass) is visible.
Restricted vocabulary	Banned or restricted operators (multiple implications, `until`, `within`, `first_match`, open-ended `##[0:$]`; `throughout` only for fixed-sequence stability) keep each property simple enough to read back as one requirement.
TOOL EXECUTION rule	Generated files are compiled and simulated with real tools, and the report separates what was compiled, simulated, evaluated by an SVA engine, and covered. Requirements are not claimed as verified on the strength of a script.
Read-back	The checker is translated back into a line-item requirements specification with the SVA for each item. Engineers review the read-back against their intent, and the line items feed commercial planning and traceability tools.

Applied to each approach

Approach A. The clarification gate acts as an automated review of the specification: every ambiguous sentence becomes a question before verification code exists. The read-back gives a second, independent statement of the requirements; differences between it and the original specification are exactly the places where the specification can be read two ways.

Approach B. Each increment's requirements go through the same loop. The assertions and assumptions accumulate into a regression suite of executable requirements, and the read-back regenerates the line-item specification at each step, so traceability is produced continuously rather than reconstructed at the end. When a requirement changes, the loop is run again and the two read-backs are compared item by item.

Approach C. This is where the rules help most. An engineer coding on the fly can state intent in a sentence; the gate asks the questions the engineer has not yet answered; the result is an assertion that checks the intent and a read-back line item that records it. The requirement is still written on the fly, but it is now written, reviewed and checked.

Evidence from the CY62256N SRAM

In the CY62256N-70 work (see *Can You Trust AI-Generated Assertions?* on systemverilog.us) the input was a vendor datasheet, a requirements-first document. Wall-clock timing: the checker was ready 14 minutes after the clarification answers (22 minutes after the datasheet was uploaded); model, testbench and revision brought the total to 36 minutes. The read-back specification took under 8 minutes, and a second checker generated only from that read-back took under 10 minutes. The clarification questions themselves exposed datasheet statements that could be read more than one way. Those numbers come from one design and one session; they show what the loop can do, not a general productivity figure. *As the French say "C'est trop Cool !!"*

10. A recommended practice: enough up front

For hardware, and for software with hard interfaces, the authors recommend a hybrid:

1. Write up front what is expensive to change: interfaces, protocols, timing budgets, reset and power behavior, safety invariants. These are the cross-cutting requirements that iterations tend to miss.
2. Let block-level and implementation details emerge in increments, as in approach B.
3. Allow on-the-fly work for exploration, but do not let a requirement remain implicit once code depends on it.
4. Run every requirement, whatever its source, through the RAG-SVA loop: clarify, classify intent, write the assumption or assertion with a cover, compile and simulate, read back.
5. Keep the read-back line items as the living specification and trace them in the team's commercial tools.

The process used to *discover* requirements can then be chosen for the project, while the process used to *record and check* them stays the same.

11. Conclusion

Requirements-first, iterative and on-the-fly development are each sensible in the right setting, and each fails in a predictable way: the first by being wrong early and found late, the second by scattering requirements, the third by never writing them down. The common remedy is to make each requirement explicit and checkable as soon as it is decided. The RAG-SVA rules do this mechanically: they refuse to guess, separate assumptions from obligations, attach covers, run real tools, and read the result back as a line-item specification. That makes the choice of process a question of how fast a team wants to learn, not of how much intent it is willing to lose.

References

- W. W. Royce, "Managing the Development of Large Software Systems," Proc. IEEE WESCON, 1970.
- B. W. Boehm, "A Spiral Model of Software Development and Enhancement," IEEE Computer, vol. 21, no. 5, 1988.
- B. W. Boehm, *Software Engineering Economics*, Prentice Hall, 1981.
- K. Beck et al., *Manifesto for Agile Software Development*, 2001, agilemanifesto.org.
- K. Beck, *Test-Driven Development: By Example*, Addison-Wesley, 2002.
- D. North, "Introducing BDD," Better Software, 2006.
- ISO/IEC/IEEE 29148, *Systems and software engineering: Life cycle processes, Requirements engineering*.
- RTCA DO-254, *Design Assurance Guidance for Airborne Electronic Hardware*; ISO 26262, *Road vehicles: Functional safety*.
- Project Management Institute, *A Guide to the Project Management Body of Knowledge (PMBOK Guide)*: progressive elaboration, rolling-wave planning.
- B. Cohen, Dr. K. Chibani, *Knowledge-Driven AI for SystemVerilog Assertions* (RAG-SVA), Cohen Publishing, 2026, systemverilog.us.
- B. Cohen, Dr. K. Chibani, *Can You Trust AI-Generated Assertions? The CY62256N SRAM*, systemverilog.us/Trusting_AI_Generated_SVA_CY62256N.pdf.
- Related papers: systemverilog.us/#papers.

RAG-SVA: personal/academic use per book purchase. Commercial use requires a Commercial License from Cohen Publishing (systemverilog.us).