

Compile the specification

AI can reduce the effort of producing code, increasing the relative importance of resolving ambiguity before implementation. Specification-driven development turns business intent into versioned schemas, state, contracts and executable acceptance evidence before agents multiply the wrong assumption.

AUTHORED BY

Daniel Johnson & Gary McLean Hall

AI DRAFTING

Astra, Fable & the models of the moment

EVIDENCE CHECKED

15 September 2026. The date tells you when the cited material was reviewed, not when the future agreed to behave.

INVALIDATED BY

When new evidence materially contradicts a claim or assumption.

EXPIRY DATE

The reader's call. Before "the model knew what we meant" becomes a project control.

The founders own the argument; AI helped draft the prose. The sources and limits are here so you can challenge both.

Contents

01. The new bottleneck
 02. Vibe coding and product engineering
 03. Context is king, but authority matters more
 04. Start with the domain
 05. Write obligations, not feature labels
 06. Compile into an execution graph
 07. Specifications must be executable where possible
 08. The evidence boundary
 09. Change control without freezing the product
 10. A practical compilation sequence
 11. What leaders should ask for
 12. Conclusion
 13. References
-

This is a method paper, combining an engineering proposal with cited external research. The review date records a source and editorial check, not an experimental validation of the method or approval of every inference by the premise authors. Worked examples are constructed cases, not reported customer outcomes.

The new bottleneck

When implementation is expensive, teams learn to tolerate ambiguity until a developer is ready to resolve it. When implementation becomes cheap, that habit becomes dangerous. An agent can turn a loose sentence into a schema, interface, API and test suite before anybody notices that the sentence supported three different interpretations.



Where generation reduces implementation effort, more of the remaining work lies upstream: deciding what the system means, which facts are authoritative, which states are legal and what evidence could disprove completion.

To compile the specification is to transform an aspiration into an execution-ready contract. The output is not one giant requirements document. It is a connected set of artefacts that constrain construction: domain vocabulary, actors, obligations, data ownership, schemas, state transitions, interfaces, quality properties, acceptance examples and release gates.

Ambiguity has a multiplication factor. If one uncertain ownership decision reaches the schema, API, permission model and reports, correcting it requires reconciling four interpretations as well as changing the original sentence. The apparent speed of generation can make this propagation hard to see. Each local artefact looks reasonable because the agent made a plausible choice at the point where the specification fell silent.

A useful compilation process therefore rejects unresolved choices that would constrain downstream work. It does not reject every unknown. Some uncertainties can be isolated behind an interface; others need a research unit or a decision from the domain owner. The output should distinguish these cases. “Unknown but contained” is a legitimate state. “Unknown, therefore silently invented” is not. That distinction lets construction continue where contracts are sound without allowing momentum to settle consequential meaning by accident.

Vibe coding and product engineering

Vibe coding is useful for discovery. Describe an idea, generate something visible, react, and repeat. It is a fast way to learn what a problem might be. It becomes dangerous when the prototype's apparent completeness is mistaken for a product boundary.

The second click is a requirement

Prototype question

Does the submit control work?

Product question

Does repeating a lost-response request create a second business event?

A product has obligations the screen does not reveal. What happens on the second submission? Who may reverse a decision? Can two workers update the same case? Which time zone controls a statutory deadline? How is old data migrated? What happens when an email provider fails after the database commits? Who can show what occurred six months later?

An agent can invent answers to every one of those questions. That is precisely the risk. Unless the organisation decides and records the answer, locally plausible choices accumulate into an accidental architecture.

Specification-driven development separates exploration from commitment. Models may propose alternatives during discovery. The compiled specification records the selected obligation. Implementation remains replaceable; the contract survives.

Discovery should produce information that can change a decision. Before an exploratory build, state which uncertainty it is intended to resolve and what observations would favour the alternatives. An attractive prototype that confirms only that a screen can be generated has answered a narrow question. A prototype that exposes a missing actor, an unusable approval sequence or an impossible data dependency has supplied material for the specification. Preserve that finding even if the prototype code is discarded. The value of exploratory implementation lies in what it teaches about the obligation, not in the pressure its existence creates to ship it.

Context is king, but authority matters more

Giving a model more context is not automatically better. The useful context is the smallest coherent set of authoritative facts.

Resolve a source conflict

Current contract

Authority for the unit's accepted behaviour.

Old proposal

History to explain a decision; it cannot silently override that contract.

A project normally contains several classes of material:

- source of truth: current law, policy, product decision, schema and contract;
- evidence: observed production behaviour, research, user interviews and measurements;
- explanation: guides and summaries derived from the sources;
- history: superseded proposals and earlier decisions; and
- noise: logs, discussion and examples unrelated to the unit.

An agent needs to know which class it is reading. Otherwise an old proposal can silently outrank a current decision. A summary can override the schema it summarises. Two equally fluent documents can pull the implementation in opposite directions.

Good context engineering therefore begins with provenance and precedence. Every unit identifies its authoritative sources. Contradictions stop the unit. Derived documents point back to originals. Decisions carry dates and owners. A fresh session can reconstruct the obligation without relying on hidden conversational memory.

Authority should be recoverable at the level of an individual obligation. A statement such as "policy takes precedence" is not enough when two policies conflict or an operational exception is approved by a different owner. Record the relevant source, its date or version, the interpretation adopted and the person accountable for that interpretation. An agent can then surface a contradiction instead of privately adjudicating it.

Derived documentation needs an update path too. A schema summary that was accurate last month can become dangerous after a migration. Where practical, generate descriptive views from the authoritative artefact. Where prose must remain hand-authored, identify the dependency so changes trigger review. The goal is not a document for every fact. It is to make visible the cost of keeping an explanation accurate, and to prevent an attractive summary from gaining authority simply because it is easier to read than the source.

Start with the domain

Schemas should not be generated from the first screen. They should follow the domain.

Identity is not a label

Person / account

One represents a human; the other represents access.

Instruction / event

A request to pay is not proof that payment occurred.

Identify the real entities, their identity and their lifespan. Separate a person from an account, a case from a referral, a decision from a note, a payment instruction from a payment event. Decide which domain owns each fact and how other domains refer to it. Record invariants: one active plan per subject, an appeal cannot precede a decision, a ledger entry is immutable, a role grants capabilities rather than page visibility.

This work prevents a common failure in generated systems: a database shaped around interface convenience. Such schemas look efficient until the product meets history, concurrency, reporting or policy change. Then nullable fields become states, strings become identifiers, and duplicated facts disagree.

Schema-first does not mean database-first. It means semantic structure before implementation momentum. The schema may be relational, event-based, typed messages or formal state. Its job is to make meaning explicit.

Consider an illustrative case-management domain. A person may have several referrals, a referral may produce several assessments, and an assessment may inform a decision without being the decision. Combining these into one mutable record makes an initial form easy to build. It makes later questions difficult: which evidence existed when a decision was made, which referral supplied it, and whether a changed assessment should alter the historical decision.

The specification should establish those identities and relationships before choosing tables or services. It should also name who can correct a fact and whether correction changes the old record or creates a new version. These choices affect audit, user workflow and data migration together. A domain model earns its place when it prevents a real ambiguity or illegal state; an elaborate vocabulary with no effect on behaviour is merely another document to maintain.

Write obligations, not feature labels

“Add referrals” is not a specification. A useful obligation identifies actor, trigger, preconditions, transition, output, failure behaviour and evidence.

One commit, two facts

The durable referral and its outstanding delivery obligation belong together. Provider delivery remains a separate fact.

Accepted	One referral identity and immutable submitted version.
Outstanding	One persisted notification obligation, visible for retry.
Not established	That the receiving team actually received the message.

Constructed contract · the local transaction does not guarantee exactly-once email.

For example: an authorised practitioner can submit a referral for a known child when mandatory evidence exists; submission creates an immutable version, starts the applicable clock, records the rule version and emits a notification request; if notification delivery fails, the referral remains submitted and retry is visible.

That statement exposes decisions. Can evidence be amended? Which clock? What does known mean? Is duplicate submission allowed? Does notification failure roll back the transaction? Those questions are the product.

Examples then make boundaries concrete. Include ordinary cases, edge cases and forbidden cases. Pair them with properties that should hold across many inputs. A date-calculation feature needs examples around holidays and leap years plus a property about ordering. An access rule needs allowed cases and attempts by every disallowed role.

The referral example also needs a transaction boundary. Submission and notification are not the same fact. The specification might require the durable submission and notification request to be recorded together, with delivery attempted afterwards. It must then define duplicate requests, retry identity and what the practitioner sees while delivery is pending. A second click should not create a second legal event merely because the first response was lost.

Acceptance examples can expose this before implementation. Submit once, repeat the same request, interrupt the delivery worker, and resume it. Observe both the response and durable records. Try an unauthorised actor and missing evidence. Ask whether an amendment after submission changes the original version. These are not extra features

discovered by testing; they are concrete expressions of the original obligation. If the domain owner gives different answers to two equivalent examples, the specification needs reconciliation before more code is produced.

Worked contract: a referral whose notification fails

Take the brief “practitioners submit referrals and the receiving team is notified”. Three reasonable implementations follow. One treats submission as complete only after email succeeds. Another commits the referral immediately and ignores failed delivery. A third records submission and an outstanding delivery obligation together. The difference determines what happens when the practitioner closes the browser, repeats the action or asks whether a deadline has begun. Choosing a button label does not resolve it.

For this constructed contract, the domain owner chooses the third interpretation. Submission is an accepted business event; notification is a separately tracked consequence. A referral has a stable identity and version. A submission request has an identity scoped to the submitting organisation. Reusing that identity with the same payload returns the existing submission; reusing it with different evidence is rejected as a conflict. This rule makes a transport retry distinguishable from a request to amend evidence.

Interruption or attempt	Required durable result	Required user meaning
Validation rejects missing evidence	No submission event	Draft remains editable; missing evidence is named
Commit succeeds; browser loses response	One accepted submission and one delivery obligation	Retrying recovers the existing receipt
Notification provider is unavailable	Submission persists; delivery remains pending	Submitted, with notification still outstanding
Worker retries after an uncertain provider response	Delivery follows the provider's documented duplicate-handling contract	No unsupported claim that the recipient received it
Practitioner changes evidence after submission	An explicit amendment or a new version under the chosen policy	Original submission remains distinguishable

The fourth row is deliberately less comforting than “exactly once”. A local transaction cannot establish what an external mail service did after a connection broke. The specification must name whether the provider supports an idempotency key, whether

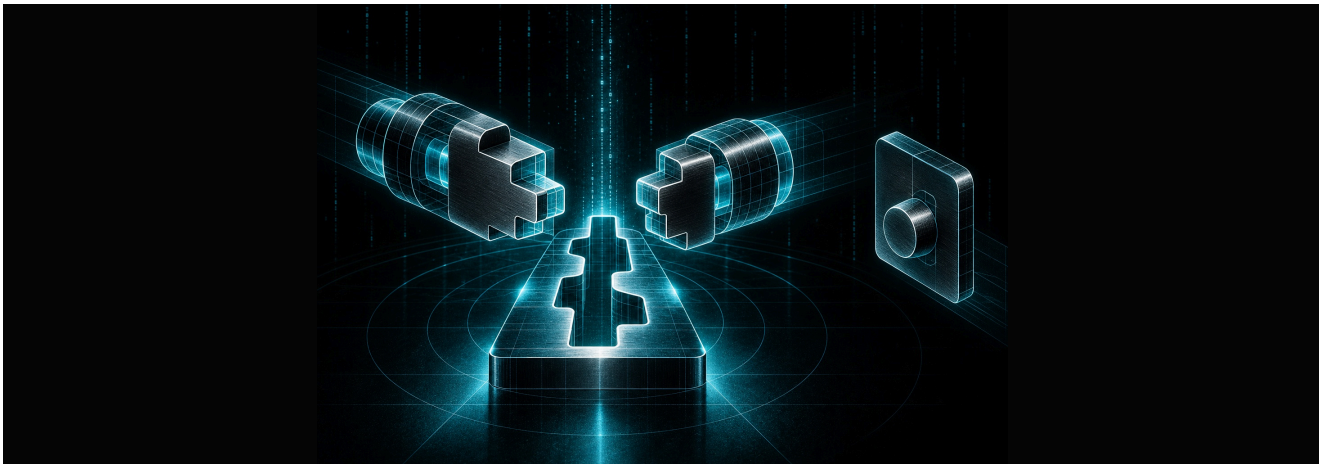
delivery can be reconciled, and whether occasional duplicate notification is an accepted limitation. It can still require exactly one local submission event. Conflating those two guarantees would put an impossible promise into the acceptance suite.

The execution contract now has testable seams. A storage gate checks that submission and delivery obligation survive or fail together. A request gate repeats identical and conflicting payloads. A worker gate interrupts delivery before and after the simulated provider accepts it. A presentation gate distinguishes saved draft, accepted submission and confirmed delivery. A recovery gate restarts the worker from persisted state. The domain owner reviews the meaning of each state; engineering demonstrates that the implementation preserves it.

This is also a useful procurement exhibit. A buyer can ask what evidence they receive if a referral appears lost, which receipt proves acceptance and who owns a failed notification. A supplier can price a concrete operating obligation. Both can identify a proposed change without relying on what “submit” seemed to imply during a demonstration.

Compile into an execution graph

A large specification cannot be handed to an agent as one prompt. It must become an ordered graph of units.



Some artefacts establish contracts used by later work: identity, domain types, schemas, interfaces and deployment foundations. Other units can proceed independently once those contracts hold. Dependencies should be explicit, not inferred from file names.

Each unit carries:

1. the obligation it fulfils;
2. the authoritative sources it may rely on;
3. the artefacts it may change;

4. the invariants it must preserve;
5. the depth signature—the failure dimensions that must be tested for its particular risks;
and
6. the gates whose results determine its technical acceptance.

This is compilation in the practical sense: ambiguous prose is rejected or resolved; valid intent becomes a structure an execution system can schedule. The graph also reveals missing foundations before implementation begins.

A graph is useful only if its edges describe real dependencies. Suppose a shared identity contract feeds a referral service and an audit export. The two consumers can be built in parallel after the identity semantics are accepted. If the export assumes a mutable display name is an identifier while the service uses an immutable key, file-level independence has hidden a semantic conflict. The contract must specify identity, not just a field named `id`.

Units also need exit states other than success. A worker may discover missing authority, a dependency that was not ready or a gate that cannot observe the claimed behaviour. Returning that finding is useful work, but it is not implementation acceptance. The plan should show whether the unit is blocked, requires a contract change or has completed its original obligation. Otherwise an apparently finished graph can contain nodes whose meaning changed during execution without any corresponding change to their consumers.

Specifications must be executable where possible

Prose remains necessary for purpose, nuance and human judgement. Claims that can be made executable should be.

Make traceability usable both ways

From an obligation

Find the particular challenge and evidence that support it.

From a failed gate

Find the endangered obligation and the owner who can settle ambiguity.

A schema can reject malformed state. A type system can exclude illegal combinations. A contract test can enforce an interface. A state-machine test can explore transitions. A property test can generate cases. A performance budget can fail a build. A deployment rehearsal can prove that migrations and rollback actually run.

NIST's Secure Software Development Framework includes tracking security requirements, risks and design decisions and integrating verification into the lifecycle. Our proposed generalisation is that important non-functional properties need an artefact and an

enforcement point. [NIST SSDF 1.1](#).

Executable does not mean complete. A test encodes somebody's understanding. It can be wrong or shallow. The link back to the obligation and source is therefore essential.

Traceability lets a reviewer ask whether the gate tests the right thing, not only whether it is green.

Traceability should work in both directions. From an obligation, a reviewer should find the gates and examples that support it. From a failed test, the team should identify which obligation is at risk and who can resolve an ambiguity. A list of requirements linked indiscriminately to a large test suite supplies neither route. The connection must be specific enough to guide action.

Not every valuable requirement can become a binary automated check. "A caseworker can understand why the deadline changed" may require a structured usability exercise as well as assertions about displayed fields. The specification can still define participants, scenarios, evidence to retain and conditions requiring revision. Executability is a spectrum of disciplined observation, not an excuse to discard claims that need human evaluation. The failure to automate a judgement does not justify leaving its acceptance criteria entirely implicit.

The evidence boundary

Recent papers describe specification-driven development as a response to AI-native engineering and warn of a "specification paradox": as code generation improves, the correctness and governance of specifications matter more. These are useful emerging frames, not settled industrial proof. Much of the published evidence remains early, tool-specific or based on benchmark tasks.

What the benchmark supplies

Repository issue

Existing architecture and recorded acceptance context.

New service

Domain meaning and unstated exceptions still have to be discovered.

SWE-bench evaluates models on real repository issues rather than isolated synthesis. Its task boundary also clarifies a limit: resolving a recorded issue within an existing test environment is different from discovering what a new public service must mean. [Jimenez et al., SWE-bench](#).

Research on generating tests from requirements shows promise, but generated tests inherit omissions and ambiguity in the source document. An agent can increase coverage of a stated requirement. It cannot know that an unstated safeguarding exception matters unless evidence or a domain expert supplies it.

The studies should be read at their actual scale. Bhatia and colleagues used requirements from five projects and asked the original developer teams to assess generated test designs. They report useful additional cases alongside inapplicable or redundant suggestions. This supports using generation to propose challenges for review; it does not establish that the generated suite completely specifies an unfamiliar service. [Bhatia et al., 2024](#).

Similarly, the 2026 specification-governance and specification-paradox papers provide frameworks and arguments about where engineering difficulty moves. They are relevant conceptual support, but their existence is not an independent evaluation of the compilation process proposed here. The distinction matters commercially: an attractive research frame cannot supply a promised delivery multiplier. [Alenezi, 2026](#); [Sirqueira and Facioli, 2026](#).

The defensible claim is therefore structural: explicit, versioned and testable specifications reduce the room in which generated implementation must guess. The magnitude of productivity or quality improvement must be measured per organisation.

The argument is also vulnerable to specification theatre. A team can create a large set of mutually consistent artefacts around the wrong problem. Formal schemas and green generated tests may then increase confidence while making it harder to question the premise. Compilation must therefore include an upstream challenge: whose work is represented, which counterexamples were considered, and what observation would make the domain owner change the obligation?

This is why benchmark performance and product discovery must remain separate evidence categories. A repository issue supplies substantial pre-existing context, including architecture and some acceptance expectations. A new service may lack all of those. Success at implementing a supplied contract does not show that a process can discover a legitimate contract. The proposed discipline addresses that gap structurally; an organisation still needs to measure omissions, rework and operational outcomes to know whether its particular practice is effective.

Change control without freezing the product

A specification is not a fixed monument. Products learn. Law, policy and user needs change. The advantage of an explicit contract is that change becomes visible.

Classify the change before pricing it

Defect

A documented exception was implemented incorrectly.

Changed obligation

The owner introduces or changes an exception; revise affected contracts.

A proposed change should identify the affected obligations, schemas, interfaces, migrations, tests and operating procedures. The execution graph shows blast radius. Versioned decisions preserve why the earlier rule existed. Compatibility and data migration become part of the change rather than later surprises.

This also sharpens fixed-price work. Discovery inside an agreed obligation remains supplier execution risk. A decision to alter the obligation becomes an explicit scope change. Without a compiled specification, every difficulty can be mislabelled as discovery and every new requirement can be mislabelled as completion.

For an illustrative rule change, imagine extending an appeal period while preserving decisions already made. The change affects more than a date function. It needs an effective date, a rule for cases already in progress, amended acceptance examples, historical reproduction and possibly a migration or notification. The graph makes those dependencies available for review before the first edit creates an inconsistent half-change.

Change control should preserve the distinction between new knowledge and a new obligation. Discovering that the implementation mishandles a documented exception is a defect. Discovering that the exception was never decided is a specification gap. Deciding to introduce an exception is a change of scope. Those labels affect accountability and commercial treatment, but their primary purpose is accurate reasoning. A record of the decision and affected artefacts is more useful than an argument about whether a sentence was technically implied in a sales conversation.

A practical compilation sequence

The sequence is iterative but ordered:

The hostile-read exercise

Input to reviewer

Only the versioned specification and connected artefacts.

Failure signal

Routine cases require the original author's verbal explanation.

1. Define the business outcome and non-negotiable constraints.

2. Map actors, domains, vocabulary and ownership.
3. Record current behaviour and external obligations.
4. Specify states, rules, data and interfaces.
5. Add quality properties: security, accessibility, performance, resilience, cost and operability.
6. Write examples, counterexamples and properties.
7. Build traceability from obligations to gates.
8. Compile the work into dependent units.
9. Run a contradiction and omission review before construction.
10. Treat every accepted change as a versioned update to both product and specification.

The output should be portable. Another capable builder should be able to implement it. A customer should not have to retain the original conversation or model vendor to understand what was agreed.

The first compilation pass should finish with a hostile read. Give a fresh reviewer only the proposed artefacts and ask them to derive representative behaviour, identify unresolved terms and find a case the gates cannot adjudicate. This tests portability directly. If the author must explain hidden assumptions verbally, those assumptions have not yet been compiled.

After construction, repeat the exercise against the delivered product and handover. The specification is not valuable because it preceded coding; it is valuable because it remains a usable account of the system. Retire obsolete decisions as history, update changed contracts and preserve the evidence for past releases. A smaller connected specification that can still be used is stronger than a comprehensive-looking document abandoned when implementation became interesting.

What leaders should ask for

Before approving an AI-assisted build, ask:

Test whether compilation earns its cost

Keep comparable

Domain novelty, obligation and acceptance depth.

Count together

Preparation, conflicting interpretations, integration rework and maintenance.

- Where is the authoritative specification?
- Which parts are proposals and which are decisions?
- Can we trace every critical obligation to executable evidence?
- What stops two agents interpreting the same rule differently?
- Which schemas and interfaces are fixed before parallel work begins?

- How are security, accessibility, operations and migration represented?
- What evidence distinguishes a scope change from an implementation problem?

If the answer is a long prompt or a demo, the product has not yet been specified.

Measuring whether compilation earns its cost

The relevant comparison includes discovery and subsequent rework. Counting implementation hours alone penalises early clarification while concealing the same decision being made repeatedly during coding. Record when a consequential ambiguity is found, which artefacts it affects, the time needed to settle it and whether another worker independently chose a conflicting interpretation. Follow the accepted unit into integration and operation, where omissions become visible.

An evaluation should compare work with similar domain novelty and consequences. A familiar export and a new eligibility service are poor controls for each other. Preserve the original agreed obligation and acceptance depth so improved speed cannot be explained by quietly accepting less. Include specification maintenance after a real rule change: a process that saves construction time but creates an unmaintainable web of documents may not be a net improvement.

Three observations would weaken the proposal. Reviewers might discover that most compiled artefacts duplicate one another without preventing conflicts. Builders might need the original conversational author to resolve routine cases despite the claimed portability. Or serious omissions might appear just as late while preparation effort increases. Those are reasons to revise the process and its claims. The method's value lies in decisions made clearer and earlier, not in demonstrating loyalty to a specification format.

Conclusion

AI does not remove the need for requirements engineering. It removes the excuse for leaving it implicit.

When implementation can multiply quickly, ambiguity multiplies with it. Compile the specification first: define the domain, record authority, make obligations testable, order the work and give every consequential claim a gate. Then let models do what they are good at—generate, transform and explore—inside a product boundary the organisation can still own.

For the referral example, the end product is not a stack of approved prose. An operator should be able to identify the rule governing a submission, inspect the accepted event, see whether notification is pending and explain what a retry would change. A builder should be able to amend the notification provider without silently changing when a referral becomes submitted. A reviewer should be able to challenge both behaviours from the same versioned contract. If any of those people has to ask the model that drafted the code what the system means, the specification has not crossed the handover boundary.

Keep that contract alive after release. When policy changes, identify the affected rule, state transition, interface, fixture and operating instruction together; record what applied to earlier decisions instead of overwriting their meaning. Run the old and new acceptance examples, retain the result and have the responsible person accept the changed obligation. This is the useful measure of a compiled specification: not how imposing it looked before coding, but whether an organisation can change its product without rediscovering its own decisions.

References

- NIST, [Secure Software Development Framework 1.1](#), 2022.
- Alenezi, [Specification-Driven Development as the Foundation of AI-Native Enterprise Software Engineering](#), 2026.
- Sirqueira and Facioli, [The Specification Paradox](#), 2026.
- Bhatia et al., [System Test Case Design from Requirements Specifications](#), 2024.
- Jimenez et al., [SWE-bench](#), 2023.