

Product Management in the Agentic Era

By Asaf Saar

Chapter 12

The Independent Check

I spent an hour reviewing a pull request on Dofek. I had generated most of the change with an AI coding assistant the night before. The code worked. The tests passed. The continuous integration checks were green.

Reading the change revealed three problems: a credential written to a log in plain text, unsanitized input flowing into a query, and an outdated transitive dependency associated with a known security vulnerability.

None had prevented the visible functionality from working. None had made those checks turn red. I had generated a working change with risks that I had not explicitly accepted.

That hour brought me back to a question I had been asking since I started in quality engineering: what, exactly, have we established about this software?

I took the three findings back to the same assistant and asked it to fix them. It did so in about two minutes. The tests still passed.

Finding the problems had taken much longer than asking for the corrections. I had been the engineer, reviewer, and security function for this project. That arrangement could teach me something; it could not be the operating model for a larger organization.

The connection to my day job is direct. As EVP and Chief Product Officer at Mend.io, I work on application and AI security and the risks introduced by software built with AI. Dofek put those questions into my own hands: a secret exposed in a log, an unsafe input path, and a vulnerable dependency. A working feature and green tests had left all three for me to discover.

Read the green result carefully

The three findings belonged to different parts of the system. Logging exposed a value that should not have been written there. The input path raised a question about what could reach a query. The dependency concern sat in software brought in through another dependency, beyond the code I had just generated.

One successful demonstration could miss all three. A test might exercise the feature without examining the log. It might supply ordinary input without challenging the query boundary. It might never inspect the dependency graph at all.

The review changed what I could honestly say about the pull request. Before reading it, I had working behavior and passing checks. After reading it, I also had three concrete reasons those observations were insufficient for the confidence I wanted to place in the change.

Make each finding specific. Name the value that reaches the log, the input that reaches the query, and the dependency and version at issue. Give the person correcting the problem a place to begin and a result to check.

The next check should be capable of detecting the discovered failure. For a logging problem, that could mean verifying that sensitive values stay out of the relevant output. For an input boundary, it could mean testing the unsafe path and the control meant to prevent it. A dependency finding calls for checking the actual dependency graph and the applicable advisory. The method follows the finding.

Ask the same question of an existing test: what would make it fail? Introduce a known incorrect condition in a controlled environment where appropriate. If the test remains green, the test has taught you something about itself.

Begin with the claim being checked

Replace broad declarations such as good, safe, and ready with the claim the release decision depends on.

Start with a claim. The calculator applies the intended pricing rule. A user cannot retrieve another user's private record. A draft contains only commitments supported by its sources. A deployed service uses the approved configuration. A customer can complete the task without assistance.

Each claim needs an appropriate kind of evidence. A calculation can be compared with independently prepared cases. An access boundary needs tests under different identities and conditions. A sourced draft needs comparison with the underlying records. A deployment claim needs observation of what actually runs. A usability claim needs people attempting the task.

No single review method answers all of these questions. A code review may reveal a likely problem without demonstrating the user outcome. A customer interview may reveal value without establishing the access boundary. A scanner may identify a known vulnerability without proving the product is free of every relevant risk.

The PM helps connect the claim to the product consequence. The specialists help determine what evidence is sufficient for that consequence. Together they can define a release decision that is more concrete than everyone saying they feel comfortable.

Use different methods for different failure modes

Assign each method the work it can check. A model can propose a solution or flag a concern. A schema validator, policy rule, static analyzer, or executable test checks a defined property reproducibly.

Deterministic does not mean complete. A test proves only what its setup and assertions actually exercise. A rule can be wrong or incomplete. A scan can miss a failure outside its model. The value is that the method has a defined basis that can be inspected, repeated, and challenged.

Human review contributes another kind of judgment. A reviewer may notice that the proposed behavior technically meets the brief while violating the purpose. They may recognize a system interaction that the chosen test suite does not cover. They may also be tired, rushed, or unfamiliar with the domain.

Choose the combination of methods according to the product's risks and the decisions the review needs to support.

For a small internal prototype, a focused review and a few representative checks may be proportionate. For a multi-user system acting on sensitive records, the evidence must be broader and the operating controls stronger. The amount and kind of assurance should follow consequence, not the excitement surrounding the build.

A second model is a reviewer with limits

Using a different model for review can be useful. It may notice a mismatch or propose a test the first model missed. Giving it the intended behavior and the artifact without the original conversation can reduce the influence of the builder's explanation.

It does not guarantee independent understanding. Models can share common assumptions, training patterns, and weaknesses. A second model can confidently agree with an incorrect premise. It can also invent a problem that wastes the team's attention.

Treat its findings as claims to investigate. Ask for the relevant evidence, a concrete reproduction where appropriate, and the product consequence. A plausible paragraph about a possible race condition is not the same as a demonstrated failure. A clean review is not proof that no problem exists.

The reviewer should not be asked to justify the builder's work. Give it the authority within the review task to question the candidate and identify missing evidence. If the prompt assumes success and asks only for confirmation, changing the model may accomplish little.

The final assurance should include observed behavior and checks appropriate to the property. A model can help direct that work. It should not be the sole source of both the claim and the evidence for the claim.

Independence has several dimensions

I have argued for keeping generation and verification structurally separate. To apply that argument well, examine the actual dimensions of separation.

Specification independence asks whether the expected behavior comes from a source other than the candidate being checked. Method independence asks whether the checker can detect a different class of error. Evidence independence asks whether observations are collected in a way the generator cannot simply rewrite to claim success. Organizational independence asks whether the people responsible for checking can report an unwelcome result.

These dimensions can reinforce one another. They can also be uneven. A separate vendor may run a checker that shares the same weak assumptions. An internal team may operate an independent method with a strong escalation path. The arrangement needs to be evaluated rather than inferred from the logo.

Different vendors can help diversify methods, incentives, and failure modes. They can also add integration cost and operational complexity. The objective is not to maximize vendor count. It is to preserve a credible challenge to the system producing the work.

When evaluating a product that both generates and reviews, ask how those functions are separated. Which specification does the checker use? What evidence does it observe? Can generation alter the tests or disable the control? Who receives the findings? What happens when the two functions disagree?

The check needs access to the real system

A reviewer who sees only the final artifact can miss the route by which it was produced and the environment in which it will operate.

For an agent, relevant context can include tools, permissions, loaded instructions, data sources, configuration, and the actions actually taken. A clean summary may conceal an unauthorized read. Correct code may run with a configuration different from the one reviewed. A dependency may fetch mutable content after the artifact inspection has finished.

Give reviewers the view needed to assess the claim, with access limited to that task. Where records contain secrets or customer information, design a way to inspect the relevant behavior without distributing unrelated sensitive material.

Logs and records should describe actions and effects in a way that can be correlated with the task. They should also be designed with privacy and retention in mind. Collecting everything indefinitely is not a substitute for deciding what evidence is necessary.

The PM can make this requirement part of the product from the start. If a user or administrator must later understand why a record changed, the system needs to preserve the relevant evidence when the change occurs. An explanation generated afterward cannot reliably reconstruct facts that were never recorded.

The right to report is part of the control

Independent technical evidence can still fail organizationally if the people receiving it have no practical ability to act.

A security reviewer may identify a serious concern while the product team controls whether it reaches leadership. A quality engineer may be told to record the issue but not delay the release. A PM may recognize that a bet has weak demand while being evaluated primarily on delivery against the original roadmap.

These arrangements create reasons to look away. They do not require bad intent. A person under pressure can reinterpret an uncomfortable finding as something to address later. Repeated across an organization, that behavior weakens the check.

Give the verification function an escalation path that the delivery owner cannot quietly suppress. Define which findings require a decision, who can accept a residual risk, and what record must remain. The arrangement should support reasoned disagreement rather than turning every concern into an unlimited veto.

Leadership also needs to examine its own response. If reporting a problem reliably produces blame while concealing it preserves status, the organization will receive less accurate information. A stated commitment to transparency cannot overcome that practical lesson.

The circuit-breaker mode of product management belongs in this broader system. It helps surface a product concern and reach a legitimate decision. It should work alongside independent specialist authority, not absorb or override it.

Verify fixes through their effects

A generated fix is a new candidate. Check its behavior before closing the finding.

Return to the property that failed. Does the corrected implementation now behave as required? Did the change create a different problem? Does the deployed system actually contain the correction? Is any consequence of the original issue still unresolved?

A logging issue such as the one I found on Dofek raises a further remediation question. Removing sensitive content from future logs addresses one part of the problem. Existing records, access, and any exposed credential may require separate attention by the appropriate owners. Declaring the issue fixed because a line of code changed would hide those obligations.

The same applies to a product misunderstanding. Changing interface wording may help, but the team should observe whether users now interpret the result correctly. A revised draft that sounds clearer to its author is not sufficient evidence of improved comprehension.

Before closing a finding, record the correction, the checks, and any consequence still requiring attention. Keep that history available to whoever makes the next release decision.

Make disagreement useful

Independent review will produce disagreement. A checker may be wrong. A requirement may be ambiguous. A team may reasonably accept a bounded risk that another participant would prefer to eliminate.

The process should make those differences inspectable. State the claim, the evidence, the consequence, and the decision owner. If a finding is rejected, explain why. If a risk is accepted, identify the scope and the conditions that would reopen it. If the specification changes, update the source of expected behavior rather than leaving the old requirement in place.

This discipline prevents two opposite failures. One is treating every finding as absolute truth because it came from a reviewer. The other is treating review as advisory theater that never changes the plan. Useful independence lies in a credible, evidence-based challenge that can affect the outcome.

A product organization can make that challenge easier by defining acceptance before the last moment. When the bar is invented at release time, disagreement becomes entangled with sunk effort and public commitment. Earlier clarity makes later evidence less personal.

Let the finding change the next build

The Dofek review gave me three problems I could act on. I want the same specificity from every incident review: a finding that changes how we build or check the next version.

If users misunderstand a status, change the experience and the acceptance criteria. If a permission change exposes a gap, preserve the case in the boundary tests. If a model update changes behavior on a known task, keep that task in the evaluation set. Asking people to be more careful leaves the mechanism that produced the problem largely untouched.

Positive results deserve the same attention. A successful pilot may depend on unusually clean data, a narrow audience, or a reviewer who quietly repairs the output. Understand those conditions before expanding it.

The bottleneck moved for me during that hour with the Dofek pull request. Producing the change had become easier. Establishing what I could trust about it still demanded work. The opportunity for product and quality leadership is to make that work deliberate, repeatable, and connected to the promise we are asking someone else to depend on.