

What Engineers Need From AI-Assisted Application Analysis

Lavan Nallainathan · 17 October 2026



Cover illustration, not an application screenshot, source extract or record of measured results. The source-linked evidence figures appear inside the paper.

Canonical publication: <https://harten.io/papers/what-engineers-need-from-ai-assisted-application-analysis/>

Published 17 October 2026.

Three source-linked evidence figures and one worked characterisation check accompany this exploratory case analysis. Candidate checks have not been executed.

Technical case analysis based on an exploratory inspection of public PRSD-IWS analysis and a pinned source reference, not a controlled product benchmark.

Abstract

AI-assisted application analysis is useful when it helps somebody answer an engineering question, inspect the supporting material and prepare a change without silently altering the meaning of the existing behaviour. A report can be readable and substantially correct while still leaving important work to its reader: identifying which execution path a rule describes, finding the relevant implementation, deciding whether two observations really conflict, and turning a broad acceptance statement into a test with a definite expected result.

This paper examines six questions using Harten's published PRSD-IWS analysis and targeted source-file inspection. It identifies useful distinctions already preserved in the analysis, including different completion paths and the separation of a financial guarantee's current status from its recorded decision. It also shows how a supposed contradiction can arise from comparing different operations, and how a correct boundary condition becomes more useful when its application area and enforcement layer are explicit. The resulting engineering answer records combine the broader explanation with source locators, change implications and concrete candidate checks. The study does not measure comparative product accuracy or human review-time savings. It establishes a set of observed presentation and reasoning issues that a subsequent controlled comparison can test.

The Engineering Question Behind an Application Report

Imagine an engineer preparing to replace the shipment-completion behaviour in an established service. The documentation says that completion must not be dated before receipt. The statement is understandable, and it sounds like an entirely reasonable rule. Before implementing it, however, the engineer needs to ask where that rule applies. Is it checked when an external user uploads a certificate, when an internal operator records completion, or in both places? Is it a condition in the domain model, a form validator, or a rule enforced only by one caller?

That distinction can change what the engineer builds. A replacement that applies the rule everywhere may introduce a restriction not established for every existing path. A replacement that omits it everywhere may remove a control that the current implementation does contain. Neither choice should be made simply because one sentence is easier to implement than a qualified account of the application.

In the earlier [paper on modernisation cost, duration and confidence](#), we explored how application findings can affect proposed work and delivery assumptions. This paper asks a different question: what must an analysis provide before an engineer can use a finding reliably?

The comparison began with the existing PRSD-IWS material rather than a newly designed demonstration. We wanted to see what the published output already helped a reader understand and where a targeted return to source made the engineering answer more precise. The purpose was not to construct a test that Harten would necessarily win.

What We Inspected and What the Comparison Can Establish

We selected six questions about movement completion, receipt clearing, financial-guarantee state, replacement integrity, partial-rejection validation and the preparation of tests. These were deliberately chosen from issues already visible in our discussion and application material. They are a purposive sample, not a random estimate of codebase-wide correctness.

The first material was Harten's published lifecycle synthesis, selected specification sections and candidate-characterisation chapter, recorded at Harten repository commit `96e92c1ea9cb9457406c1622efdbd02d402159e6`. The second was a targeted source-assisted investigation using GitHub search and file reading at DEFRA PRSD-IWS commit `95f7eeae92f82a8db3c609f14ce2483c6e7cb738`. The investigation used the same AI assistant that had already encountered parts of the Harten material. It was not blinded, and the source-assisted answers are not an independent coding-assistant benchmark. [1–5]

There is also an input-version boundary. The Harten case-study manifest records the analysed archive's hash but does not verify the corresponding upstream commit. We did not recover that archive in this inspection. The reference commit therefore cannot be represented as a proven identical input, and an apparent difference against it is not automatically a historical analysis error. The public package also intentionally excludes its underlying evidence reports and private evidence index. This study assesses the material that was available, not every feature of the Wayfinder workspace. [1]

These limits do not prevent us examining whether a statement supplies the scope needed to use it, whether its reasoning follows, or which additional source observation makes the next engineering action clearer. They do prevent us claiming a percentage accuracy improvement, a reduction in review time or superiority over another vendor. No application tests or proposed changes were executed. The inspection produced six comparison records and seventeen candidate checks, which remain proposals for verification.

Different Completion Paths Need Different Explanations

The published states and acceptance material already distinguishes internal from external completion. It records an internal check against the receipt date and a check rejecting a date later than the current UTC date. It does not assert that the corresponding external path enforces those same conditions. That is a useful result: the analysis preserves a difference an implementer could otherwise overlook. [2, 4]

The two completion methods do not contain the same checks

The same business action has two method-level paths. The highlighted conditions apply to internal completion.

EXTERNAL COMPLETION METHOD

Movement.Complete

Selected, non-contiguous statements from lines 320–326.

```
L322 Guard.ArgumentNotDefaultValue(() => completedDate,  
completedDate);  
  
L323 Guard.ArgumentNotDefaultValue(() => fileId, fileId);  
  
L325 stateMachine.Fire(completedTrigger, completedDate, fileId,  
createdBy);
```

What these lines do: reject default date or file-identifier values, then fire the external completion trigger.

No receipt-date or future-date comparison appears in this method.

[Read the complete method](#)

INTERNAL COMPLETION METHOD

Movement.CompleteInternally

Selected, non-contiguous statements from lines 333–344.

```
L335 if (completedDate < Receipt.Date)  
  
L339 if (completedDate > SystemTime.UtcNow.Date)  
  
L343 stateMachine.Fire(internallyCompletedTrigger, completedDate,  
createdBy);
```

What these conditions do: reject a completion date before receipt or after the current UTC date. Each rejection raises `InvalidOperationException` before the trigger fires.

[Read the complete method](#)

Consequence for a replacement: do not infer one universal rule from either method. The external controller, handler and validation path still need investigation before deciding the required future behaviour.

Source: DEFRA PRSD-IWS, commit 95f7eeae92f8. [Movement.cs](#)

Verbatim statements; indentation removed and long display lines may wrap. Omitted lines are not shown. Annotations are Harten interpretation; source excerpts are separately identified.

The pinned source makes the distinction inspectable. In `Movement.Complete`, the method checks that the completion date and file identifier are not their default values before firing the completion trigger. `CompleteInternally` instead reads `Receipt.Date`, rejects an earlier completion date, rejects a future date and then fires its internal completion trigger. The inspected completed-receipt constructors assign the supplied values; they do not add the missing receipt-date comparison to the external method. [6, 7]

An engineer can now identify a specific characterisation test: construct an eligible movement with a receipt dated on day 10, call internal completion with day 9, and check the exception raised by the date-order guard. An external endpoint test needs more work because its controller or handler may impose an additional check before reaching the inspected method. The analysis should say that, rather than turning the absence of a local guard into a claim about the whole endpoint.

The source-assisted answer is not valuable because it contains more technical names. It is valuable because those names identify the boundary of the claim and the place where the engineer can check it. The broader explanation tells the reader why two paths matter; the source reference makes the explanation actionable.

Two True Observations Can Be Mistaken for a Contradiction

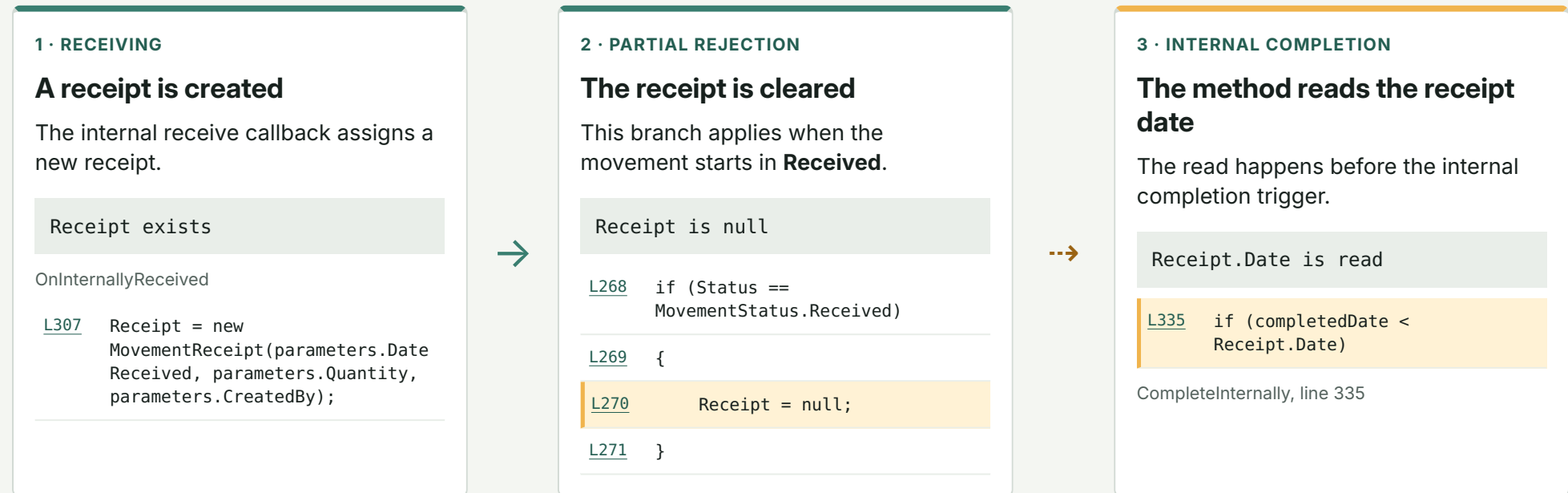
The published business-rules material identifies a more subtle problem around receipt data. It contrasts an account in which rejection or partial rejection clears a receipt with an account in which receive methods do not clear it, and calls the contrast unresolved. Those descriptions concern different operations. A receipt method can create a receipt while a later rejection method removes it. The two statements do not, by themselves, contradict each other. [3]

This is not a reason to remove uncertainty from the analysis indiscriminately. It is a reason to identify exactly which claims are being compared. Before treating two observations as conflicting, we need to establish that they concern the same object, operation, starting conditions, execution path and version. Otherwise the report can create an investigation task that exists only because its descriptions lost their scope.

The pinned `Movement` source illustrates the distinction. `OnReceived` and `OnInternallyReceived` create receipt data. `PartialReject` clears `Receipt` when the movement's starting status is `Received`. The state configuration permits an internally completed transition from `PartiallyRejected` to `Completed`. Yet `CompleteInternally` reads `Receipt.Date` before firing that transition. [6]

Receipt creation and later receipt clearing are different operations

Both observations can be true. The unresolved question concerns the route from partial rejection into internal completion.



Solid connection: receiving and later clearing refer to different operations in the inspected class, not contradictory statements.

Dashed connection: caller sequence not yet verified. Does another operation restore the receipt, or does a caller prevent this route?

Next investigation: trace the caller and persistence sequence, then test a representative partially rejected movement. The source configuration permits internal completion from PartiallyRejected, but this diagram does not establish a production failure.

Source: DEFRA PRSD-IWS, commit 95f7eeae92f8. [Movement.cs](#)

The sequence is an explanatory reconstruction from source, not a runtime trace. No candidate check was executed. Annotations are Harten interpretation; source excerpts are separately identified.

The useful engineering question is therefore whether the actual caller and persistence sequence can reach internal completion with a missing receipt. Another operation might restore it, or a higher-level invariant might prevent the sequence. We have not established either outcome. The next action is to inspect that path and build a representative test, not to assert a production failure.

The application analysis did something useful by identifying a possible relationship between partial rejection and completion. The improvement is to separate that risk from a supposed conflict between different operations. Keeping everything under an undifferentiated label of unknown would make it harder for the team to decide what to do next.

A Lifecycle Distinction Matters When Another Component Uses It

The financial-guarantee lifecycle provides a clear example of something the published analysis already does well. It explicitly distinguishes the guarantee's current status from its recorded decision. A guarantee can become Released while retaining an Approved decision and the original decision date. It also distinguishes the separate release date. [8]

The reference implementation supports that account. Approval sets the decision to Approved. Release changes the lifecycle status and records the release date. The release callback only changes the decision and its date when the previous decision was None, so the earlier approval survives the ordinary approved-then-released flow. [9]

The next source observation makes the consequence clearer. The assessment view displays the recorded decision separately from the current status. Its display logic explicitly retains Released as the current status when the decision is Approved. The distinction is therefore not merely an incidental property of a domain class; this inspected consumer uses it to present different information to the user. [10]

Releasing a guarantee does not erase its approval decision

One guarantee, two meanings: what was decided earlier and its current lifecycle status. This shows the ordinary approved-then-released flow.

Information	After approval	After release
Current status	Approved	Released
Recorded decision	Approved	Approved, retained
Decision date	Original approval date	Original approval date, retained
Release date	Not yet recorded	Release date recorded

The release callback preserves an existing decision

FinancialGuarantee.OnReleased · lines 209–215

```

L209 ReleasedDate = releasedDate;
L210
L211 if (Decision == FinancialGuaranteeDecision.None)
L212 {
L213     Decision = FinancialGuaranteeDecision.Released;
L214     DecisionDate = releasedDate;
L215 }
```

An Approved decision is not None. This conditional branch therefore does not overwrite the earlier decision or its date.

The assessment template uses both values

Exact expressions from Index.cshtml · not an application screenshot.

```

L31 @EA.Prsd.Core.Helpers.EnumHelper.GetDisplayName(Model.Decision)
L35 @EA.Prsd.Core.Helpers.EnumHelper.GetDisplayName(statusToUse)
```

These expressions occupy separate table rows for the decision and current status. The display logic on lines 13–16 retains Released when the decision is Approved.

[Inspect the actual template](#)

Consequence for a replacement: preserve both meanings and their relevant dates, or approve an explicit alternative mapping. A single simplified status value could discard information used by this consumer.

Source: DEFRA PRSD-IWS, commit 95f7eeae92f8. [FinancialGuarantee.cs](#) | [Index.cshtml](#)

The before/after table is a source-derived explanation, not measured application data. Annotations are Harten interpretation; source excerpts are separately identified.

Suppose a replacement design proposes to merge the decision and status into one simplified field. The team now has a specific question to answer: how will that representation preserve the meanings and dates still needed by the service? The existing implementation does not prohibit redesign, but it supplies evidence that a redesign must address. A migration test should include an approved-then-released record, not only a newly approved one.

The output retains a distinction with a concrete consequence for implementation and migration. The additional source link identifies a consumer that makes the consequence easier to inspect.

A Validation Rule Needs an Application Area and an Enforcement Layer

A similar issue appears in partial-rejection quantities. The published material says that the rejected quantity must be strictly less than the actual quantity in the cited domain path. It correctly preserves the equality boundary rather than simply repeating wording that says a rejected quantity must not exceed the actual quantity. [3, 4]

The pinned import service, `PartialRejectionImportMovement.PartailReject`, checks `rejectedQuantity >= actualQuantity`. With equal units and quantities of 10 and 10, that condition is true and the service throws. The accompanying error message refers to a quantity being more than the actual shipment quantity, but the executable condition also covers equality. A test based only on the message could therefore encode the wrong boundary. [11]

There is a second distinction. The similarly named export-side `PartialRejectionMovement.PartailReject` checks the date but does not contain the same quantity comparison in the inspected method. It would be incorrect to infer that the export endpoint accepts equality: an earlier form or request validator could reject it. But it would also be incorrect to extend the import service's local guard to every export path without checking. [12]

The source-assisted record therefore names the application area, method and layer next to the condition. It also records that the import service compares the supplied decimal quantities without visible unit conversion. The reviewer still needs to establish where compatible units are normalised. A source-level decimal comparison is not, on its own, a complete account of the business quantity rule. [11]

This example does not show that the published statement is necessarily wrong. It shows that a generally correct statement can leave an implementer with avoidable work if the supporting path is not identified where the rule is used. The path belongs with the claim, not only somewhere in a large supporting report.

Some Questions Should Remain Open After Source Inspection

The approval flow also illustrates a limit that both the published analysis and direct source review should retain. The service looks up a currently approved guarantee, approves the new one, and changes the previous guarantee differently according to whether its reference matches the new reference. The collection lookup uses `SingleOrDefault` over guarantees with `Approved` status. [13, 14]

These files show a sequential replacement procedure and an assumption about the collection. They do not establish database uniqueness, transaction isolation or the result of concurrent approval requests. The published lifecycle material identifies those gaps, and the targeted source inspection provides no basis for declaring them resolved. [8]

An answer that confidently converts the lookup into a universal one-approved-guarantee invariant would be less useful than the qualified analysis. The next work needs database and transaction evidence, other mutation paths and concurrency tests. More confident wording would not remove that work.

For comparison purposes, this is not a win for either approach. It is a case where the available material sets a boundary on what we can conclude. A good evaluation must include questions of this kind, rather than rewarding every system for producing a definite answer.

A Candidate Test Needs a Definite Expected Observation

The existing documentation includes named candidate tests and `Given/When/Then` structures. It also states that no successful execution is being claimed. Those are useful foundations. Some cases, however, still need substantial interpretation before they could become checks an engineer or agent can execute. [5]

For example, the movement-completion candidate allows the result to be `Completed` or an appropriate partial/rejection state. That formulation leaves the evaluator to decide what appropriate means. The inspected completion methods have specific inputs and transition behaviour, so a more useful handoff separates those operations from quantity and rejection tests rather than combining them into one broad scenario. [5–7]

The revised worked records identify the object and entry point, starting state, input values, expected observation, source basis and execution status. Where the source does not supply a complete expected result, the record is labelled as an investigation. A test asking what the external endpoint does with an early completion date is not the same as an approved requirement instructing the replacement to reject it.

Worked Check: Internal Completion Before Receipt

This makes the existing candidate **Q1-T1** concrete; it does not add a newly executed test.

Field	Proposed check
Starting condition	A valid Movement fixture in Received state, with a non-null Receipt dated 10 January 2026. Freeze the current UTC date at 20 January 2026. Use a valid creator identifier and otherwise valid fixture data.
Action	Call <code>Movement.CompleteInternally</code> with a completion date of 9 January 2026.
Expected observation	The date-order guard raises <code>InvalidOperationException</code> before the internal completion trigger is fired.
Why this follows	9 January is earlier than the recorded receipt date of 10 January, so <code>completedDate < Receipt.Date</code> is true.
Source basis	<code>Movement.cs</code> , lines 333–344, at the pinned reference commit.
Execution status	Candidate characterisation check. Not yet executed.
Boundary	This is a method-level expectation. It does not establish external-endpoint behaviour, persistence atomicity, or an approved future business rule.

The dates are chosen test inputs, not production observations. Freezing the clock keeps the future-date check from becoming an accidental second cause of failure. [6]

Seventeen candidate checks were prepared from the first five comparison questions. They include method-level date and identity guards, the approved-then-released guarantee, reference-dependent replacement and equality at the partial-rejection boundary. They have not been executed. Their contribution at this stage is greater specificity and clearer evidence boundaries, not a measured reduction in effort.

The Consequence for AI-Assisted Engineering

When a human engineer reads a broad sentence, they may recognise that they need to search for its implementation before relying on it. A software agent also needs a way to recognise that boundary. Giving an agent a long report without clear execution paths or distinguishing observations from approved requirements can leave it to supply the missing decisions itself.

The examples suggest a practical division of work. Application-wide analysis explains the capabilities, relationships and questions worth investigating. A source-linked engineering answer identifies the exact observation relevant to the change. A person or authorised process decides which behaviour the future system should require. Candidate tests then express how the chosen requirement or existing behaviour will be checked.

That division does not require every paragraph to contain a large schema or every missing detail to become a new approval gate. It requires enough information at the handoff to prevent a specific misunderstanding. In these examples that means naming the internal or external completion method, keeping approval history separate from release status, and checking whether two supposedly conflicting descriptions concern the same operation.

We applied that approach to the worked answer records in this comparison. We have not claimed a corresponding generator release, changed the existing public case study, or demonstrated that independent users complete tasks faster. Those are different claims and require their own evidence.

What a Controlled Comparison Should Measure Next

A controlled comparison needs a common, verified input snapshot and questions fixed before the outputs are generated. It should include ordinary rules, cross-component relationships and questions that cannot be answered from the available material. The complete authorised evidence experience should be tested separately from the deliberately narrower public case-study presentation.

Independent reviewers can then compare source investigation, analysis-assisted investigation and a combined approach. They should assess whether the answer is correct within its stated scope, whether the supporting source can be inspected, whether consequential conditions were omitted and whether a concrete change or verification task can be prepared. Active review time should include checking and correcting the output, not just reading its first answer. Reviewer disagreements and learning effects need to be recorded rather than hidden inside an average.

This initial inspection does not tell us which product wins that test. It does show why the test should examine more than the presence of a report, a graph or a plausible explanation. The useful output is the one from which a team can determine what the application does, what remains uncertain, which decision is needed and what must be checked before the change is accepted.

References and Source Notes

All source inspection refers to DEFRA PRSD-IWS commit

95f7eeae92f82a8db3c609f14ce2483c6e7cb738. Harten output locators and Git blob identities are retained in the comparison's evidence register at Harten baseline commit

96e92c1ea9cb9457406c1622efdbd02d402159e6. The original analysed archive identity has not been mapped to the reference commit.

1. Harten, public PRSD-IWS case-study manifest, `content/case-studies/prsd-iws/manifest.json`. Original source archive SHA-256: 3452e285a66ecd243d07e3f1df3ba1379a859b6078f24425b6e159f591283e85. The manifest states that the analysed upstream commit is not verified and lists excluded evidence material.
2. Harten, [States and Transitions](#).
3. Harten, [Business Rules](#).
4. Harten, [Acceptance Criteria](#).
5. Harten, [Candidate Characterisation Tests](#), particularly CT-13-005, CT-13-006 and CT-13-008.
6. DEFRA, [Movement.cs](#), `CreateStateMachine`, `PartialReject`, receipt callbacks, `Complete` and `CompleteInternally`.
7. DEFRA, [MovementCompletedReceipt.cs](#).
8. Harten, [Financial-Guarantee Lifecycle Synthesis](#).
9. DEFRA, [FinancialGuarantee.cs](#), `OnApproved`, `Release`, `OnReleased` and state configuration.
10. DEFRA, [Financial-guarantee assessment view](#), lines 1–90 inspected.
11. DEFRA, [PartialRejectionImportMovement.cs](#).
12. DEFRA, [PartialRejectionMovement.cs](#).
13. DEFRA, [FinancialGuaranteeApproval.cs](#).
14. DEFRA, [FinancialGuaranteeCollection.cs](#).

Attribution: Contains public sector information licensed under the Open Government Licence v3.0. Source software: DEFRA/Environment Agency, PRSD-IWS. The work is independent Harten analysis, not a departmental commission or endorsement. The case-study observations are not claims about the current production service.