أكاديمية كاوست
KAUST ACADEMY

جامعة الملك عبدالله للعلوم والتقنية
King Abdullah University of Science and Technology

Clause by Clause: Retrieval-Augmented Compliance Verification for Construction Documents

Sohaib Bantan[1], **Azzam Alharthi**[1], **Malik Binmahfouz**[1], **Youssef Mohamed**[2]

sohaib.bantan@kaust.edu.sa, azzam.alharthi@kaust.edu.sa, malik.binmahfouz@kaust.edu.sa, youssef.mohamed@kaust.edu.sa

[1] *Department of Computer Science and Engineering, University of Minnesota: Twin Cities, Minneapolis, MN, U.S.A.*

[2] *King Abdullah University of Science and Technology (KAUST)*

Abstract

Reviewing a construction scope-of-work document (which typically contains a drawing, project specifications, & bill-of-quantity) against a large multi-body standards library requires an engineer to hold the entire document and the entire standards corpus in mind simultaneously, a burden that renders manual review slow, error-prone, and difficult to scale. We present a retrieval-grounded pipeline that automates this process: standards and project documents are each extracted into discrete, citable units; for each unit of the project document, the relevant standards clauses are retrieved; and each is classified as inadequate, adequate, or exceeding, with every classification traceable to its source clause. When a retrieved clause does not genuinely govern an element, or the element's own text is too vague to grade against it, the tool withholds a finding rather than forcing a low-confidence guess, positioning it as a screening layer that augments, rather than replaces, engineer judgment. We evaluate the pipeline on a 30-case synthetic benchmark: it classifies compliance correctly 80.0% of the time and correctly identifies both the governing standard and clause 70.0% of the time, with document-level retrieval alone succeeding in 86.7% of cases. This indicates that the pipeline's main remaining weakness is narrowing down to that final clause, rather than finding the right document.

Date: August 17, 2026

Contents

1 Introduction **2**

2 Related Work **2**

3 Methodology **2**

3.1 Standards Ingestion . . . 3

3.2 Document Extraction . . . 7

3.3 Clause Retrieval . . . 8

3.4 Review Engine . . . 10

3.5 Report Generation . . . 11

4 Results **11**

4.1 Evaluation Setup . . . 11

4.2 Classification Results . . . 13

4.3 Retrieval Results . . . 14

4.4 Error Analysis . . . 14

4.5 Analysis . . . 15

5 Discussion **15**

6 Limitations and Future Work **15**

7 Conclusion **16**

References **16**

1 Introduction

PDEC (Planning, Design, Engineering & Construction) is KAUST's in-house owner's-representative consultancy. Part of its role is reviewing construction scope-of-work packages – specifications and bills of quantities (BOQs) – against a clause-level standards library spanning multiple codes and bodies (SBC, NFPA, ADA, KAUST-internal), before a package is approved.

Checking a scope-of-work package by hand against a large, multi-body standards library is slow and easy to get wrong: a reviewer has to find the right clause for every checkable requirement before they can even judge compliance. This project builds a tool that automates that retrieval-and-classification step. Given a project's selected standards and its uploaded documents, it produces a clause-cited, per-item compliance breakdown that an engineer can check and act on.

The system is explicitly a *screening layer* that supports engineer judgment, not a replacement for it – every report it produces says so directly. It currently only reads text-based PDFs (no OCR), does not yet review construction drawings (see Section 6), and runs as a single-team local tool with no authentication. It has so far only been optimized and tested against KAUST's own standards library, not against other institutions' standards.

System summary.. The pipeline moves a project through five stages, described in Section 3: turn a standards PDF into individual, citable clauses; turn a project's specification/BOQ PDFs into review-ready pieces; retrieve the handful of clauses that actually govern each piece, so every later judgment is grounded in a citable source rather than the model's own recall; classify each piece against its retrieved clauses; and render the results into a report an engineer can act on. Section 4 evaluates this pipeline on a synthetic benchmark and reports both classification and retrieval accuracy.

2 Related Work

Automated code compliance checking (ACC) has traditionally been built on Building Information Modeling: hand-encoded rules are run against a structured BIM model, as in Solibri [1] and the review this approach surveys [2]. UpCodes layers real-time AI checks for a fixed set of elements onto the same model-centric pattern [3]. Peng and Liu instead automate rule *extraction*, converting code text into a knowledge graph matched against BIM data [4], and the most recent work applies LLMs directly – Chen et al. pair an LLM with an ontology [5], while Madireddy et al. use LLMs to compile clauses into Revit scripts [6]. All of these assume a BIM model as the object of review and compile each requirement into rule logic or a graph/script ahead of time.

PDEC's review stage has no BIM model to check – only PDF specifications and BOQs – and this project targets that gap: rather than authoring rules per clause, it performs retrieval-augmented, per-item classification at review time, following the RAG paradigm of Lewis et al. [7], but applied ahead of retrieval to first turn heterogeneous standards PDFs into a citable representation that RAG pipelines normally assume already exists. The trade-off is less of the clear-cut auditability that hand-authored rule logic provides.

3 Methodology

The tool is a standard web application (Python backend, SQL database, browser-based frontend). Because parsing standards, extracting documents, and running reviews can each take seconds to minutes, all of this work happens in the background rather than blocking the interface, and any failure is written back to the record so it is visible in the UI rather than lost in a log.

At a high level, the system works in five stages, as illustrated in Figure 1:

1. **Standards ingestion** – a standards PDF (SBC, NFPA, ADA, KAUST-internal, etc.) is broken down into individual, citable clauses.
2. **Document extraction** – a project's specification and BOQ PDFs are split into review-ready pieces.
3. **Retrieval** – before any compliance judgment is made, the standards library is narrowed down to the handful of clauses that actually govern each piece, so the judgment that follows is grounded in a citable source rather than the model's own recall.
4. **Review** – each piece of the reviewed documents is compared against its retrieved clauses and classified.
5. **Report generation** – findings are compiled into a single PDF an engineer can act on.

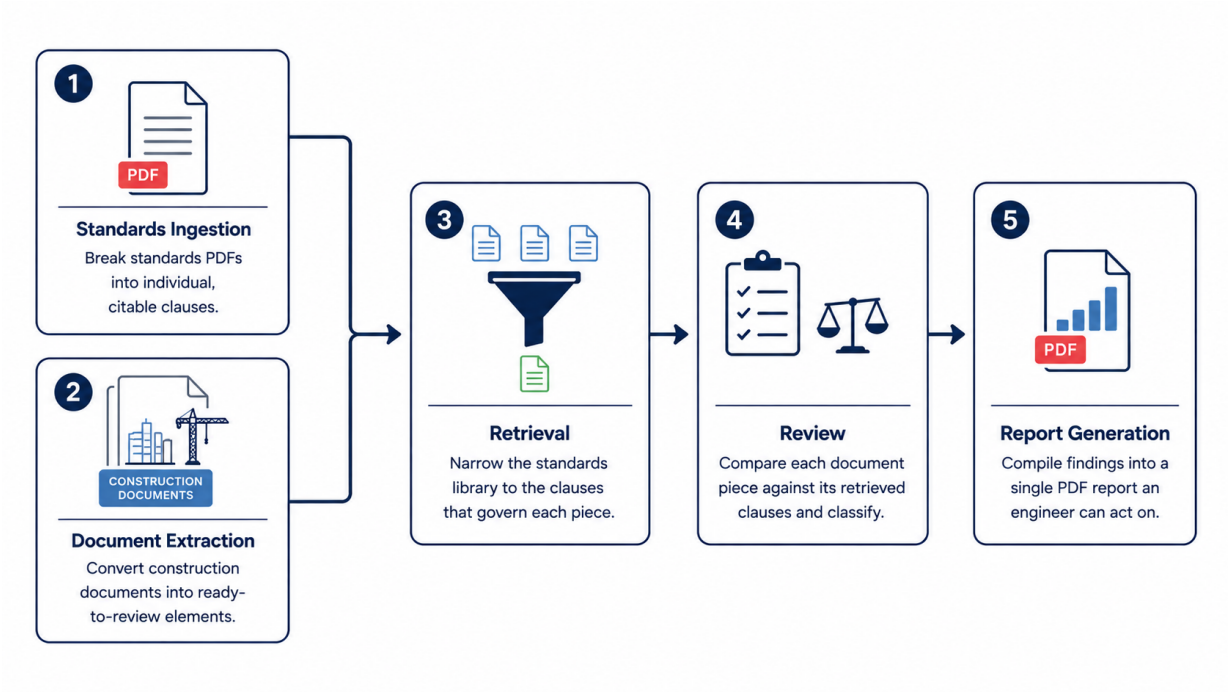


Figure 1 The main pipeline, from a standards PDF and a project's specification/BOQ PDFs through to a clause-cited compliance report.

3.1 Standards Ingestion

Standards ingestion turns a standards PDF into a library of individually citable clauses, arranged as a tree that mirrors the document's real hierarchy (part, section, article, clause) rather than a flat list. The current approach makes no upfront pass over the whole document at all: each page-bounded chunk is handed to the model with one instruction, to work out that chunk's own heading/numbering scheme and extract it top-down, word for word. The response schema places a `structure_reasoning` field before the extracted items, so the model has to state the chunk's own hierarchy before emitting anything.

Motivation. Standards documents vary widely in structure, spanning numbered codes such as SBC and NFPA, KAUST's own CSI-style specification format, and unstructured prose, so a single fixed parsing strategy does not generalize across them. Printed clause numbers are also unreliable within KAUST's standards: numbering restarts under some headings, and sub-items are occasionally printed under the wrong parent heading. Document hierarchy therefore cannot

be recovered from printed numbers alone; it has to be rebuilt from what each item actually says about its own position in the document.

Why a tree, not a flat list. Each extracted item reports its own open ancestor headings by title, and a script-side resolver rebuilds the real parent/child structure by matching those titles against a running stack of currently open headings, rather than by parsing printed numbers. Printed numbers are kept only as display metadata for citation. The resulting tree is not cosmetic: it is what later stages of the pipeline actually operate on. Coarse-to-fine retrieval narrows candidate clauses in two tiers, and its second tier groups clauses by immediate parent heading, so that grouping only exists because clauses carry a real parent reference. Each clause's embedding text is also prefixed with its full ancestor path (e.g. its part and section titles) before its own title and text, so that a short leaf clause is never embedded as context-free text – the surrounding hierarchy is what lets retrieval and the reviewing model both understand what a brief clause is actually about.

Why clause-by-clause. Extraction is instructed to make every requirement-bearing paragraph its own clause, however short, and never to merge sibling paragraphs or collapse a whole section into one clause, since doing so would destroy the granularity a reviewer needs. This granularity is what makes a review finding meaningful evidence rather than a vague pointer: the review engine is required to cite the exact clause a finding was checked against, so an engineer reading a report can go straight to the specific requirement in question instead of re-reading an entire section to find what was actually being judged.

Pipeline. Each standard is first parsed to page-level text and cleaned of repeated headers and footers. For KAUST's three-part specification format, a single model call is given a complete, unfiltered listing of every line in the document and locates the Products and Execution section boundaries by meaning rather than by matching fixed heading text; the General section, which contains no verifiable requirements, is then excluded from extraction entirely rather than extracted and discarded. The remaining pages are split into page-aligned chunks under a fixed character budget, with the whole document capped at an overall character limit; a document beyond that limit is only partially ingested, and this is surfaced to the user rather than left silent. Each chunk is sent to the model once, in document order, together with the ancestor headings still open at the end of the previous chunk and the last few items already extracted, so the model stays consistent with what came before without re-extracting it. The model returns individual items, each carrying its type (heading or clause), its printed number, its title, its verbatim text, its page, and the titles of its open ancestor headings. Document hierarchy is then rebuilt programmatically as described above, and where a chunk's first item is really a continuation of the previous chunk's last clause, its text is merged into that clause rather than creating a duplicate, as formalized in Algorithm 1. The resolved tree is flattened into clause records; a clause whose text exceeds a fixed length is split into ordered, still-verbatim continuation rows rather than being truncated. Each clause is then paired with a short, separately generated summary of its intent, and an embedding built from its ancestor path, title, intent summary, and text.

Algorithm 1 Chunk-wise joining of the clause tree

Require: chunks $C_1, \ldots, C_n$ in document order
Ensure: clause tree ROOTS, each node linked to its parent and children

```
STACK ← ⟨⟩                                   ▷ currently open ancestor nodes, outermost first
ROOTS ← ⟨⟩
for i ← 1 to n do
    items ← EXTRACTCHUNK(C_i, STACK) ▷ one LLM call, given titles in STACK; each item reports its own parentPath
    if items[1] is a same-number, same-ancestors continuation of the node at top of STACK then
        merge items[1]'s text into that node in place; drop items[1]
    end if
    for each item in items do
        pop STACK to at most |item.parentPath| entries, then further to the first title that disagrees with item.parentPath, if any ▷ if STACK is already shallower, no ancestor is synthesized – attach under what remains open
        node ← new node wrapping item, child of top of STACK (or a new root if STACK is empty)
        push node onto STACK
    end for
end for
return ROOTS
```

Example structure. The following is a schematic illustration of the resulting shape, not real extracted document content:

```
PART 2 - PRODUCTS                                (heading)
  2.1 Materials                                  (heading)
    2.1.3  Door hardware corrosion resistance    (clause)
           "Door hardware shall be corrosion-resistant
            stainless steel..."                  (verbatim text)
```

The clause 2.1.3 above is stored with its parent reference pointing to the 2.1 Materials heading, its topic reference pointing to the PART 2 - PRODUCTS heading, and an embedding built from the text `"PART 2 - PRODUCTS > 2.1 Materials > Door hardware corrosion resistance: <intent summary>\n<verbatim clause text>"`.

Storage. There is no separate table for the tree itself. Every row, heading or clause, is stored in a single clauses table, and the tree is represented entirely as self-referencing parent and topic columns on that table: a clause's parent column points at its immediate parent row, and its topic column points at the nearest top-level ancestor (a top-level heading points at itself). Heading rows carry an empty text field and are never embedded, so they cannot be returned as a review finding's evidence, but they remain real rows so that clause rows have something to attach to. Printed clause numbers are stored purely as display metadata for citation, never used to reconstruct structure.

Figure 2 shows this stored tree as an engineer actually sees it: the Standards Library page for a real ingested KAUST KDCS document (KDCS 09 21 16, Gypsum Board Assemblies), expanded down through its PART, section, and sub-section headings to two individual clauses (1.3.1 and 1.3.2), each shown with its own generated intent summary and its full verbatim text.

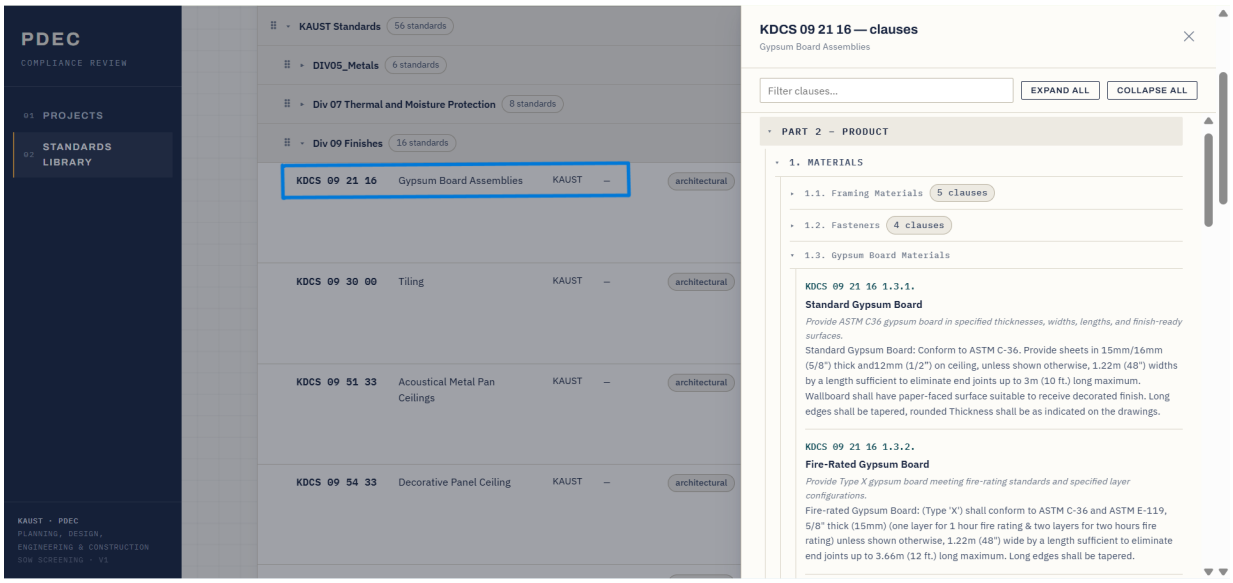


Figure 2 The Standards Library page, showing the ingested clause tree for a real KDCS document (KDCS 09 21 16, Gypsum Board Assemblies) expanded down to two individual, citable clauses.

Cost, model, and timing. Standards ingestion is instrumented end to end: every LLM call records its token usage, and every stage of the pipeline – the part-boundary locator, the chunk-extraction loop, intent summarization, embedding, and the database write – records its own wall-clock time. Table 1 reports three such measurements, taken by running the ingestion pipeline end to end, extraction and database persistence included, against three real KAUST standards PDFs spanning close to a two-order-of-magnitude range in extracted text volume, all on the same ingestion model, `gpt-5.6-luna`.

Table 1 Three end-to-end standards-ingestion runs on `gpt-5.6-luna`, measured directly by the pipeline's own timing and token-usage instrumentation. "File size" is the size of the source PDF on disk. "Extracted text" is the character count actually sent through the chunk-extraction loop, after the Products/Execution boundary locator excludes the General section (Section 3.1). "Tokens" is prompt / completion, summed over every LLM call the run made, including the boundary locator and the intent-summary batches.

Standard	File size	Extracted text	LLM calls	Tokens (P / C)	Wall time
Solar Hot Water Tanks (3 pp)	117 KB	5,084 chars	4	7,269 / 4,716	36.5 s
Cathodic Protection (13 pp)	186 KB	23,328 chars	8	23,467 / 14,720	87.1 s
Commissioning of HVAC (61 pp)	714 KB	32,184 chars	6	28,005 / 12,119	69.8 s

Table 2 translates those same token counts into dollar cost, at the list price of `gpt-5.6-luna`: $0.20 per million input (prompt) tokens and $1.20 per million output (completion) tokens.

Table 2 Ingestion cost per run, derived from the token counts in Table 1 at $0.20 / $1.20 per million input / output tokens.

Standard	Input cost	Output cost	Total cost
Solar Hot Water Tanks	$0.0015	$0.0057	$0.0071
Cathodic Protection	$0.0047	$0.0177	$0.0224
Commissioning of HVAC	$0.0056	$0.0145	$0.0201

What stands out across all three runs is the asymmetry between cost and time: ingesting a real standard costs a fraction of a cent – at most $0.0224 among the three runs above – while taking well over a minute of wall-clock time in every case, and over five minutes for the largest of the three. In isolation, that wall time looks like a real cost. It is not, because standards ingestion is not something the tool repeats per project or per review: each standard is parsed once, and the resulting clause library is stored and reused by every subsequent review that draws on it, for as long as that standard remains active in the system. The one-time wait therefore does not fall on the engineers using the tool day to day; it is paid once, up front, by whoever is building out the standards library, and every review run afterward draws on already-ingested clauses at essentially no additional latency or cost. This is what makes the ingestion cost profile acceptable despite its per-document wall time: negligible dollar cost times a one-time-per-standard time cost is a cost the team absorbs once, not a recurring tax on every compliance check that follows.

3.2 Document Extraction

Drawings are currently stored but not yet extracted or reviewed; this is a known, deliberate gap discussed further in Section 6. Scanned/image-only PDFs are also rejected up front, since the system has no OCR capability. Two different strategies were investigated for extracting specifications and bills of quantities from a scope-of-work document: a regex/pdfplumber-based mechanical splitter assisted by narrow, opt-in LLM calls, and a purely LLM-driven approach that reads the whole document through a fixed set of compliance-relevant questions. Both are described in turn below.

LLM Query-Based Extraction. Rather than treating specifications and bills of quantities as structurally distinct documents requiring separate extraction pipelines, this approach reads the entire scope-of-work document in a single, uniform pass. The document is divided into page-bounded chunks sized to the language model's context window, and each chunk is extracted independently. Instead of splitting on structural markers such as section headings or table boundaries, the model is guided by a fixed set of compliance-relevant lens questions, covering dimensions and clearances, fire and life-safety provisions, accessibility, structural design, MEP capacities, material performance properties, and testing or certification requirements, and extracts every passage relevant to at least one of these categories as a distinct, independently checkable item, copied verbatim from the source text. Embedded bills of quantities are not extracted through a separate mechanism: any detected table is spliced back into the page text as a formatted table before extraction, so a table row is extracted as a self-contained item, preserving its item number, description, quantity, and unit, through the same pass used for prose requirements. This trades the structural guarantee of a mechanical splitter, where every line of source text is accounted for by construction, for the ability to identify and isolate individual checkable requirements regardless of where or how they appear in the document.

Regex + LLM Extraction. As part of continued development of the document extraction pipeline, a narrow, opt-in LLM classifier was added to the page classification step to address a structural gap that the regex/pdfplumber-based approach alone could not resolve. This addition preserves the original regex-only behavior by default, verified to produce byte-identical output when disabled.

The page classification step previously distinguished only *spec* and *BOQ* content, causing administrative pages (submittal-tracking cover sheets, transmittal forms) to be misclassified as spec content. A third category, *admin*, was introduced via a rule-based decision tree operating on page-level signals (heading-pattern density, colon-density, BOQ table-shape), escalating to a single LLM call only when signals are absent or conflict. A demotion rule further routes pages with

competing signals (e.g., high colon-density alongside genuine numbered-section structure) to the LLM rather than trusting a heuristic that cannot disambiguate the two cases. Pages classified as admin are excluded from spec/BOQ chunking and routed to a separate output, improving downstream chunk quality.

Status. Both extraction approaches were developed and evaluated, and both produced good extractions. LLM Query-Based Extraction produced more diverse results, since the model reads and interprets the document directly rather than following a fixed set of structural rules; Regex + LLM Extraction was cheaper and more mechanical, since only the classification step calls the model and the extraction itself is deterministic pattern matching. The method integrated into the main pipeline is LLM Query-Based Extraction, chosen for its flexibility: because it does not depend on a document's structure matching a fixed set of patterns, it has a better chance of working correctly on visually different input documents, and in practice its LLM usage cost was very low. Regex + LLM Extraction is kept in the codebase as a reference path rather than removed outright.

3.3 Clause Retrieval

The review engine's compliance judgments are grounded in a retrieval-augmented generation (RAG) design: rather than asking the model to classify a piece of a project document against the standards library from memory or from the full corpus at once, the system first retrieves the handful of clauses actually relevant to that piece and grounds the model's judgment in only those, with every classification traceable back to the clause it was retrieved against. Before any compliance judgment can be made, the system must therefore narrow the standards library down to the handful of clauses relevant to the piece of the project document under review. Exposing the model to the entire library is neither affordable nor effective, so retrieval is the stage that decides which clauses the review engine ever sees – and, in the RAG sense, what grounding the model's classification is allowed to rest on.

3.3.1 Motivation

The retrieval process for our pipeline is motivated by how an engineer would manually check whether an element in a construction document complies. Given a requirement such as "wires shall be 10 mm in diameter" and a shelf of candidate standards covering metal fabrication, fire protection, and electrical work, an engineer would not read all three cover to cover. They would first discard the standards that are obviously the wrong subject entirely (the metal and fire codes) and keep only the one that governs wiring. Within that surviving standard, they would then skim its section headings and skip whole sections that are irrelevant to the query (television panels, light fixtures, air conditioning), stopping only on the section that actually covers wiring. Only at that point would they read line by line to find the specific requirement. This two-stage narrowing, first by standard, then by section, before ever reading individual requirement text, is what a human does by instinct, and it is the process retrieval is built to reproduce mechanically. This is named the **Triage Funnel**: a fast, cheap triage pass discards whole standards and whole sections before any individual clause is examined closely, mirroring how an engineer discards the obviously irrelevant before reading anything line by line.

3.3.2 The Triage Funnel Retrieval Strategy

The search is narrowed in tiers: first to the most relevant standards, then to the most relevant sections (clauses grouped by their parent heading) within those standards, and only the clauses inside the surviving sections are then scored individually by cosine similarity. Every tier is

scored from the already-stored embeddings, so the narrowing itself costs no additional model calls beyond embedding the query once. A group's score at the standard and section tiers is taken as the maximum clause similarity within it rather than an average, so a section whose single relevant clause sits among many unrelated siblings is still surfaced rather than diluted away. The surviving clauses are handed to the review engine (Section 3.4) as the candidate set for classification. Figure 3 lays the two side by side: the human narrowing process from the motivation above on top, and the Triage Funnel's own three tiers, discarding standards, then sections, then scoring individual clauses, directly underneath it.

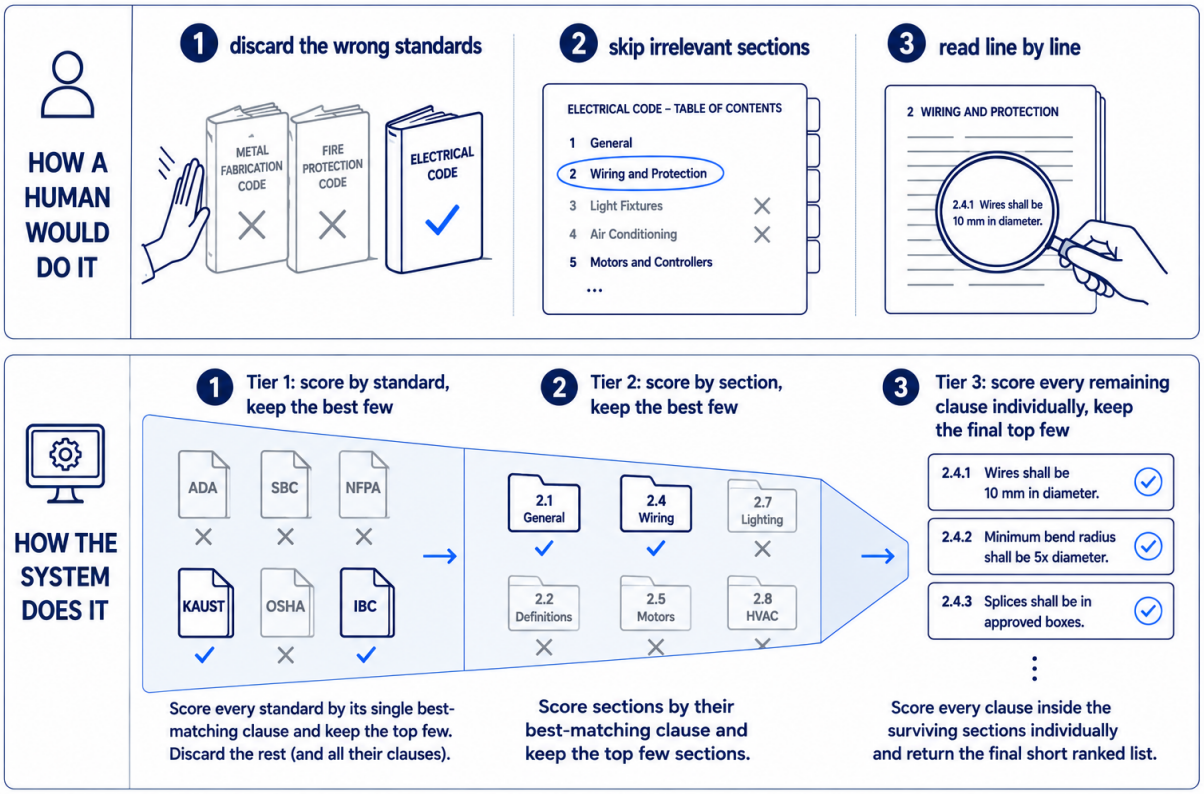


Figure 3 The Triage Funnel, mirrored against the human process that motivates it. Top: an engineer discards the wrong standards, skips irrelevant sections, then reads line by line. Bottom: the same narrowing done mechanically in three tiers, scoring by standard, then by section, then by individual clause.

Setup, once per review. Before any chunk is routed, the project's active standards are loaded once into a flat, database-free snapshot: every embedded clause the project has not excluded, each carrying its own vector alongside its standard, title, text, and parent reference. That flat list is then grouped, once, into two sets of buckets that mirror the two tiers below: one bucket per standard, and one bucket per section (clauses sharing the same immediate parent heading). Each bucket precomputes and stores its own stacked matrix of clause vectors and their norms, so that scoring an entire bucket against a query later is a single matrix operation rather than a loop over its clauses. This setup work depends only on the project's standards, never on the query, so it happens exactly once per review rather than once per chunk, and the same prebuilt buckets are reused unchanged by every chunk and every concurrent review worker.

Per chunk, three tiers. For each chunk under review, its own content (the same text that will later be shown to the review model) is embedded into a single query vector. Tier one scores every

standard bucket against that query, where a bucket's score is the single highest-scoring clause inside it, and keeps only the top few standards (three by default) – the rest, and every clause inside them, are dropped before any individual clause is ever scored. Tier two repeats the same best-of-bucket scoring, but only over the section buckets belonging to the standards that survived tier one, and keeps only the top few sections (eight by default). Tier three is the only point where individual clauses are scored: every clause inside the surviving sections, typically a few dozen rather than the full library, is scored individually against the query and cut down to the final top-k (eight by default). That final ranked list of clauses and their similarity scores is what the review engine receives as its candidate set for one chunk. In shape, retrieval therefore takes one embedding vector in; passes through an intermediate pool of clauses that survived the standard and section cuts but have not yet been individually scored; and returns a short, ranked list of (clause, score) pairs out.

3.3.3 Worked Example

The following walks through plausible numbers for one chunk, to make the tier-by-tier narrowing concrete; the figures are illustrative, not a measurement from a real run. Suppose a chunk extracted from a specification reads "door clear width [accessibility]: all accessible doors shall provide a minimum clear opening width of 800 mm," and the project's active standards are ADA, SBC-Architectural, NFPA-101, KAUST-Internal, and OSHA. Tier one scores each standard by its single best-matching clause and might return ADA 0.81, SBC-Architectural 0.77, KAUST-Internal 0.69, NFPA-101 0.44, and OSHA 0.21; keeping the top three drops NFPA-101 and OSHA entirely; none of their clauses are scored again. Tier two scores the sections within the three surviving standards and might keep, among others, ADA's door-clearance section, SBC-Architectural's means-of-egress and doors sections, and KAUST-Internal's accessibility section – eight sections in total, out of several dozen candidates. Tier three then individually scores every clause inside those eight sections, perhaps fifty to sixty clauses combined, and returns the top eight as the chunk's final candidate set.

3.4 Review Engine

Each piece of a reviewed document is classified only against the clauses Section 3.3's Triage Funnel surfaces for it, keeping every review call small, cheap, and focused on what is actually relevant.

For each piece of reviewed text and its candidate clauses, the model classifies compliance into one of three outcomes: *inadequate*, *adequate*, or *exceeding*. A finding requires both a clause that genuinely governs the piece under review and text specific enough to grade against it; when either condition is not met, including a requirement that is genuinely vague or open-ended, the model emits no finding at all rather than a placeholder category. Silence is therefore a normal, expected outcome for a piece of reviewed text, not a fallback result. An earlier revision instead routed exactly this case into a fourth *manual review* category, but that category grew to roughly two thirds of all persisted findings in practice, the model using it as a catch-all for "this clause doesn't really apply" rather than the genuine-vagueness case it was meant for, and was removed rather than tuned further. The model is also asked to cite the exact clause, locate the finding within the document, flag any mismatch between the standard the text refers to and the standard the project is actually governed by, and assign a confidence level (high, medium, or low) reflecting how much interpretation the finding required; confidence is recorded for display only and never changes which category a finding falls into, or whether one is made at all. Each finding also carries a set of tags describing the nature of the requirement (measurement, material, implementation, testing, performance, documentation), a third classification axis alongside category and discipline, with

zero or several tags assigned per finding as applicable. Which documents a review covers is a snapshot chosen at review-start time rather than implicitly "the whole project," and each review run is kept as a separate record rather than overwriting past results.

3.5 Report Generation

The output is a single self-contained PDF an engineer can use without needing the tool itself: a scannable summary up top, findings grouped by category with citations and details, a dedicated section for low-confidence items, and a standing disclaimer that the report is a screening aid rather than a substitute for professional review.

4 Results

4.1 Evaluation Setup

4.1.1 Synthetic Evaluation Set

No engineer-labeled compliance corpus exists for this tool, so a synthetic evaluation set was constructed instead: 30 test cases were LLM-generated against 14 real KAUST design-standard sections spanning five CSI divisions (05 – structural, 09 – finishes, 22 – plumbing, 23 – HVAC, 26 – electrical). Each case pairs a short SOW-style element description with a known governing clause and a known correct verdict, which makes it possible to score both *what the tool concluded* and *what it cited* against ground truth, rather than only the former.

The set is deliberately balanced across the three verdict classes – 10 Inadequate, 10 Adequate, 10 Exceeding – so that accuracy is not inflated by a majority class, and is dominated by measurement-style cases (25 of 30: a numeric field value checked against a clause threshold, e.g. a duct gauge, a clearance, an air-balancing tolerance) with a smaller set of execution-style cases (5 of 30: a procedural or workmanship requirement rather than a single number). Table 3 gives the breakdown by division.

Table 3 Evaluation set composition by CSI division.

Div.	Standards covered	Standards	Cases
05	Structural Steel Framing, Aluminum Space Frame	2	2
09	Gypsum Board, Painting, Resilient/Cork/Wood/Access Flooring, Tiling	5	11
22	Plumbing Piping, Hangers/Supports, Specialties, Equipment	4	7
23	HVAC Duct and Casing, Testing/Adjusting/Balancing	2	5
26	Indoor/Exterior Lighting, Panelboards	3	5
Total		**14**	**30**

Two axes are scored independently for every case, matching the two things the review pipeline actually does (Sections 3.1 and 3.3 onward): **classification accuracy** – whether the tool assigned the correct Inadequate / Adequate / Exceeding label – and **retrieval accuracy** – whether the tool cited the correct standard document and clause/subsection. These are not the same signal: a case can retrieve the exactly correct clause and still misjudge it, or misjudge *because* it retrieved the wrong clause and graded the element against an unrelated threshold. Reporting them separately identifies which stage of the pipeline a failure belongs to. Retrieval accuracy is further split into *document-level* (the correct standard was cited, regardless of clause) and *full* (the correct standard *and* the correct clause/subsection were both cited), since these two failure sizes have

different causes: a wrong standard is a coarse routing failure, a right-standard-wrong-clause is a fine-grained selection failure inside an already-correct document.

4.1.2 Run Configuration

The evaluated run (**R37**) used the pipeline's default configuration, unchanged from the values in Section 3.1 onward. Table 4 lists every setting relevant to the review call and the retrieval it depends on. Ingestion and review use independently configurable models (Section 3.1); only the review model is exercised by this evaluation, since the standards library was ingested once, ahead of time, and reused unchanged across runs.

Table 4 Run configuration and hyperparameters for R37.

Setting	Value
Review / compliance-judgment model	`gpt-4.1-mini`
Standards-ingestion model	`gpt-5.6-luna`
Embedding model	`text-embedding-3-small`
Retrieval top-k (clauses per chunk)	8
Triage Funnel tier 1 – standards kept	3
Triage Funnel tier 2 – sections kept	8
Response format	Structured Outputs (typed schema)

Retrieval narrows in two tiers before any clause is scored against a document chunk: the project's active standards are ranked by maximum cosine similarity and the top 3 survive; that survivor set's sections (clauses grouped by immediate parent heading) are then ranked and the top 8 survive; only clauses inside those surviving sections are scored for the final top-8 list handed to the review call. Both cuts reuse embeddings already loaded once per review, so this narrowing adds no additional API calls beyond the retrieval a flat top-k search would already require. Each finding is produced by one `chat.completions.parse` call per document chunk against a Pydantic Structured Outputs schema, which is what allows every citation to be checked mechanically against the ground-truth clause labels in Table 3's standards rather than free-text-matched. The reviewed run is one immutable record among several; re-running review on the same project always produces a new record rather than overwriting the prior one, so R37 is reproducible as a specific, citable evaluation instance rather than a floating "current" number.

4.1.3 Extraction-Robustness Evaluation

A separate, reference-free evaluation framework was developed to assess extraction quality without requiring manually labeled reference chunks, comparing extracted output directly against the source document across six categories: coverage (spec and BOQ), block integrity, intrachunk cohesion, Document Contextual Coherence (DCC), chunk repetition, and BOQ row sanity. Flagged cases are grouped by root cause to surface when multiple checks stem from the same underlying issue. The framework was validated by manually reviewing flagged cases against source documents, which surfaced and corrected several measurement errors, including an initial coverage calculation that excluded filtered header/footer lines from both the extracted and reference text, inflating reported coverage; the corrected calculation compares against the full, unfiltered document text. This framework measures extraction robustness only – a different stage of the pipeline from the classification/retrieval evaluation above, and one with no notion of a "correct verdict" to score against.

4.2 Classification Results

Table 5 summarizes R37 against the 30-case set. Overall classification accuracy was 80.0% (24/30); retrieval was correct at the document level in 86.7% of cases (26/30) but correct at the full standard-*and*-clause level in only 70.0% (21/30) – a 16.7 percentage point gap that is entirely fine-grained clause selection inside an already-correctly-identified standard, discussed in Section 4.3. No duplicate findings were produced.

Table 5 R37 headline metrics.

Metric	Value
Classification accuracy	24/30 = 80.0%
Document-level retrieval accuracy (correct standard, any clause)	26/30 = 86.7%
Full retrieval accuracy (correct standard *and* clause)	21/30 = 70.0%

Figure 4 shows the classification confusion matrix. Off-diagonal mass concentrates one step away from the diagonal – Inadequate is never confused with Exceeding or vice versa; every misclassification involves the Adequate class as either the true or predicted label – consistent with Adequate sitting in the middle of an ordinal severity scale rather than the three classes being interchangeably confusable.

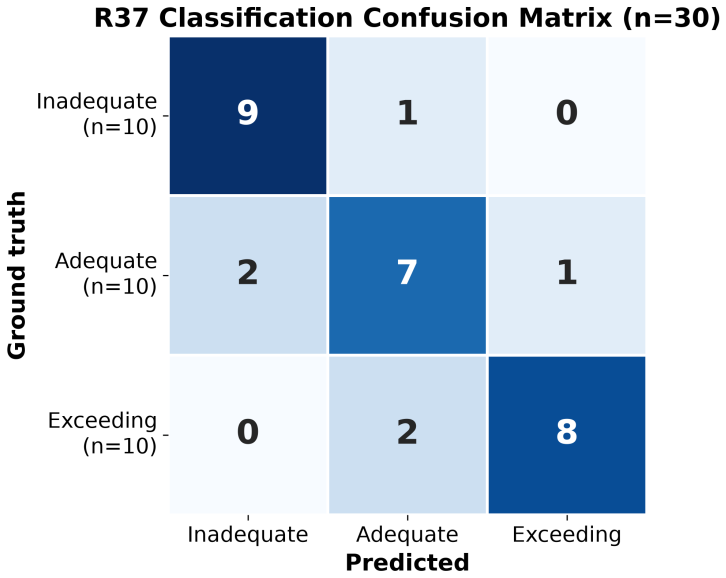


Figure 4 R37 classification confusion matrix (ground truth vs. predicted).

Table 6 breaks classification performance down by class. Inadequate and Exceeding are both detected reliably (F1 85.7% and 84.2%); Adequate is the weakest class on every metric (70.0% precision, 70.0% recall) because, as the middle class on an ordinal scale, it absorbs errors pulled from both neighbors.

Table 6 Per-class classification metrics for R37.

Class	TP	FP	FN	Support	Precision	Recall	F1
Inadequate	9	2	1	10	81.8%	90.0%	85.7%
Adequate	7	3	3	10	70.0%	70.0%	70.0%
Exceeding	8	1	2	10	88.9%	80.0%	84.2%
Macro avg	–	–	–	30	**80.2%**	**80.0%**	**80.0%**
Overall accuracy	–	–	–	30	–	–	**80.0%**

4.3 Retrieval Results

Table 7 breaks retrieval accuracy down by CSI division. Division 22 (plumbing) and division 05 (structural) are perfect on both axes; division 09 (finishes) is the clear outlier, at 72.7% document-level and only 45.5% full retrieval accuracy, and alone accounts for 6 of the 9 total retrieval failures despite being only 37% of the case set.

Table 7 R37 retrieval accuracy by CSI division.

Div.	Cases	Doc-level correct	Full correct	Doc acc.	Full acc.
05	2	2	2	100%	100%
09	11	8	5	72.7%	45.5%
22	7	7	7	100%	100%
23	5	5	3	100%	60.0%
26	5	4	4	80.0%	80.0%
Total	**30**	**26**	**21**	**86.7%**	**70.0%**

The 9 retrieval failures split evenly by type: 4 cross-document (the wrong standard was cited entirely – e.g. Resilient Flooring mis-cited as Cork Flooring, Wood Flooring mis-cited as Access Flooring) and 5 same-document-wrong-subsection (the correct standard was cited but the wrong clause within it – typically an adjacent subsection, e.g. clause 8.5.4 cited as 8.5.3).

4.4 Error Analysis

Two representative cases illustrate the classification errors that occur even when retrieval is correct – i.e. the model cited the right clause but still misjudged the field value against it.

SYN-10 (pressure test, Inadequate misread as Adequate). *Required:* the clause states no drop in pressure is permitted. *Found:* the recorded test showed a 15 kPa pressure drop. *What went wrong:* a 15 kPa drop against a zero-tolerance requirement should have failed the check, but the model excused it and marked the element Adequate instead of Inadequate.

SYN-16 (air-balancing tolerance, Adequate misread as Exceeding). *Required:* the clause defines a symmetric ± tolerance band around a target value as the compliant range. *Found:* the reported value fell inside that band. *What went wrong:* a value inside a compliance band is, by definition, compliant – but the model read "inside the band" as over-performance and marked the element Exceeding instead of Adequate.

Both cases retrieved the correct clause, so the error is not a citation mistake but a misreading of what the clause's own threshold means: an absolute limit treated as negotiable (SYN-10), and a two-sided tolerance band treated as a one-sided lower bound (SYN-16).

4.5 Analysis

The main takeaway from R37 is that **retrieval, not grading, is the pipeline's current bottleneck** – and specifically the fine-grained step of picking the right clause *inside an already-correctly-identified standard*, not the coarser step of finding the right standard. Full retrieval accuracy (70.0%) is the lowest of the three headline numbers, well below document-level retrieval accuracy (86.7%), and half of all classification errors (3 of 6) are a direct downstream consequence of a bad citation rather than a bad judgment on a correctly retrieved clause. This concentrates almost entirely in division 09 (finishes), whose five near-identical flooring and finish standards account for 6 of the 9 retrieval failures and are the only division below 80% accuracy on either axis, while divisions with clearly distinct standards (e.g. division 22, plumbing) retrieve perfectly – so the practical fix to prioritize is discriminating between closely related standards, not retrieval in general. This is a single run ($n = 30$) against a synthetic set, so it should be read as a diagnostic pointing at a concrete failure mode rather than a generalization claim about accuracy on arbitrary real-world documents.

5 Discussion

The R37 evaluation (Section 4) points to one main takeaway: finding the right *document* is largely solved, but narrowing further to the exact right *clause* within it is where the pipeline still loses the most accuracy, and that gap is what drives most of the remaining classification mistakes. Grading itself is comparatively reliable, and the few errors that do occur are less about doing the math wrong and more about misreading what a clause's own limit is actually asking for – a nuance worth an engineer's second look rather than a sign of a broken system.

This weakness is also not uniform: it shows up far more where several standards cover very similar ground than where standards are clearly distinct from one another, so citations deserve more scrutiny in exactly those overlapping areas. That, more than grading logic, is the clearest target for future work.

These results come from a single small evaluation and should be read as an early, diagnostic signal rather than a claim about accuracy on arbitrary real projects. As throughout this report, the tool remains a screening layer meant to support engineer judgment, not replace it.

6 Limitations and Future Work

- **Evaluation so far is synthetic, not a real project.** R37 (Section 4) measures accuracy on a 30-case synthetic benchmark built from real standard clauses, not on an uploaded real-world SOW package. *Next step:* run a bigger, more formal evaluation against an actual real SOW document – real specifications and BOQ from a real project, ideally with engineer-verified ground truth – to check whether the failure patterns found on the synthetic set (Section 5) still hold once real documents, with all their inconsistency and noise, are involved.
- **Fine-grained clause retrieval is the main accuracy gap.** R37 shows the tool reliably finds the right standard but is weaker at narrowing down to the exact right clause inside it, especially where several standards cover very similar ground (Section 5). This points at the Triage Funnel's tier two, the section-level narrowing step (Section 3.3.2), and the individual clause scoring that follows it, as the concrete place most of the remaining accuracy is lost. *Next step:* improve how tier two discriminates between closely related sections, rather than only between whole standards.
- **Drawings are not yet part of the pipeline.** Construction drawings are currently stored but not extracted or checked – only specifications and BOQs are reviewed. *Next step:* add a

drawing-reading extractor; the pipeline already reserves a slot for this so the review engine itself would not need to change.

7 Conclusion

This project implements an end-to-end pipeline – standards ingestion, document extraction, clause-level review, and report generation – for screening construction scope-of-work packages against a clause-level standards library, built as a support tool for PDEC's review process rather than a replacement for it. The pipeline runs end-to-end today, and Section 4 gives it a first formal accuracy evaluation: on a 30-case synthetic benchmark, the tool classifies compliance correctly 80.0% of the time, and correctly identifies both the governing standard and clause 70.0% of the time.

The main takeaway is that document-level retrieval is largely solved, but narrowing further to the exact right clause is still the weakest link in the pipeline, and it is what drives most of the remaining classification mistakes – grading itself is comparatively reliable. This gives a clear, concrete target for the next round of work, rather than a vague sense that "accuracy should improve": sharpen retrieval where standards closely resemble one another, and the rest of the pipeline should follow.

References

[1] *Solibri Model Checker*. https://www.solibri.com/. Accessed: 2026-07-16.

[2] Aimi Sara Ismail, Kherun Nita Ali, and Noorminshah A. Iahad. "A Review on BIM-based Automated Code Compliance Checking System". In: *2017 International Conference on Research and Innovation in Information Systems (ICRIIS)*. IEEE. 2017, pp. 1–6. DOI: 10.1109/ICRIIS.2017.8002486.

[3] *UpCodes AI: real-time code checking*. https://up.codes/ai. Accessed: 2026-07-16.

[4] Jun Peng and Xin Liu. "Automated Code Compliance Checking Research Based on BIM and Knowledge Graph". In: *Scientific Reports* 13 (2023), p. 7065. DOI: 10.1038/s41598-023-34342-1.

[5] Nanjiang Chen et al. "Automated Building Information Modeling Compliance Check through a Large Language Model Combined with Deep Learning and Ontology". In: *Buildings* 14.7 (2024), p. 1983. DOI: 10.3390/buildings14071983.

[6] Soumya Madireddy et al. "Large Language Model-Driven Code Compliance Checking in Building Information Modeling". In: *Electronics* 14.11 (2025), p. 2146. DOI: 10.3390/electronics14112146.

[7] Patrick Lewis et al. "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks". In: *Advances in Neural Information Processing Systems*. Vol. 33. 2020, pp. 9459–9474.