A Formal Analysis of the EDGAR EPS Parser: Architectural Patterns, Heuristic Robustness, and Strategic Evolution

Karan Vora

Trexquant Investment Interview

October 2025

Abstract

This report presents a formal analysis of the EDGAR EPS Parser, a standard-library-only Python system for extracting quarterly Earnings Per Share from SEC filings. We deconstruct its multi-modal architecture, comprising a targeted tabular search followed by a tiered narrative fallback. A detailed critique of its novel columnar selection heuristic reveals a significant over-weighting of recency signals relative to periodicity cues. We confirm and analyze a critical scoring anomaly concerning Basic vs. Diluted EPS, tracing its origin to the absence of a unified scoring philosophy across different extraction paths.

Section 1: Architectural Assessment of a Heuristic-Based Extraction System

An evaluation of the parser's high-level design reveals a sophisticated and deliberate architecture. The system's philosophy strikes a balance between the need for precision and the practical challenges of parsing heterogeneous financial documents, although these choices introduce specific constraints and potential failure points that warrant examination.

1.1 The Precision-Recall Cascade: A Sound Architectural Pattern

The system's end-to-end flow is structured as a **precision-recall cascade**. This is an established and highly effective pattern for information extraction tasks where the cost of a false positive is high. The parser first attempts extraction from structured tabular data (`extract_from_table`), which is the highest-precision source. If this fails, it sequentially falls back to a series of narrative extraction methods, ordered from most to least constrained: `extract_from_narrative_strict`, `extract_from_text_eps_token`, and finally the general `extract_from_text`. This architecture correctly assumes that a value found in a well-defined table row labeled "Earnings per share" is more likely to be correct than a number found in proximity to the phrase in unstructured text. By prioritizing high-precision, narrow-scope methods and only engaging broader, higher-recall methods upon failure, the system maximizes its chances of finding an accurate value while minimizing the risk of introducing errors from ambiguous contexts.

However, this cascaded design, while effective, introduces a critical dependency chain that can lead to cascading failures. The entire tabular extraction path is gated by the `pick_best_quarter_column` function. This function's determination of the "best" column is a "winner-take-all" decision; if it incorrectly identifies an annual column as the target or fails to identify any column, the subsequent, highly nuanced logic for scanning the EPS band is rendered ineffective. The system is not designed to recover from an error at this stage. It cannot, for example, reconsider other columns if the "best" one yields no plausible candidates. This makes the overall system accuracy critically dependent on the performance of this single, upstream heuristic. An initial misjudgment forces a premature and complete fallback to the lower-precision narrative methods, bypassing potentially correct data that exists within the tables.

1.2 The "Standard Library Only" Constraint: A Deliberate Trade-Off

A core design principle of the parser is its exclusive reliance on Python's standard library, eschewing common external dependencies like `BeautifulSoup` or `lxml`. This constraint offers significant practical advantages, including maximum portability, zero dependency management overhead, and a reduced attack surface, making the tool easy to deploy in a wide variety of environments. To achieve this, the system implements a custom `TableParser` built upon the native `html.parser.HTMLParser`.

This decision, however, imposes an implicit limit on the system's robustness. The custom `TableParser` does not process complex table structures involving `rowspan` and `colspan` attributes. This is not a trivial omission but a direct consequence of the `stdlib-only` constraint. Robustly parsing modern HTML, especially the kind of structurally complex tables often generated by financial reporting software, is a problem that has been addressed over decades of development in dedicated libraries. By forgoing these tools, the system implicitly accepts that it will perform exceptionally well on the large corpus of simple, well-formed tables but will fail silently and unpredictably on the long tail of filings that use more advanced layouts. This architectural choice means that the most

significant future gains in parsing accuracy may not come from further refinement of scoring heuristics, but rather from a strategic decision to relax this constraint and integrate a more powerful and comprehensive HTML parsing engine.

Section 2: A Formal Analysis of the Columnar Selection Heuristic

The introduction of the `pick_best_quarter_column` function is a major architectural enhancement in v3, designed to focus the parser's attention on the most relevant data columns. A formal analysis of its internal scoring model, however, reveals a critical imbalance that can undermine its objective.

2.1 Deconstruction of the Heuristic Scoring Model

The function employs a linear weighted model to score each potential data column within a table's header. This model synthesizes three primary signals to identify the most recent quarterly data column:

- **Periodicity Cues:** A strong positive weight (+12.0) is assigned to columns whose headers contain explicit quarterly phrases (e.g., "three months ended"), while a significant negative weight (-8.0) is applied for annual phrases (e.g., "year ended").
- **Recency Boost:** A score bonus is calculated based on the most recent year found in the column header, using the formula (max(years)-1990)/2.0.
- **Positional Bias:** A small penalty is applied based on the column's position ($-0.15 * j$), creating a bias toward leftmost columns as a tie-breaker.

This model correctly identifies the key attributes a human analyst would use. However, the relative magnitudes of these weights create a vulnerability where one signal can systematically overpower another, leading to predictable errors.

2.2 Critical Vulnerability: The Dominance of the Recency Heuristic

The scoring model is imbalanced in a way that allows the recency boost to dominate the explicit periodicity cues. This can cause the parser to select the wrong column, thereby subverting its primary goal of finding *quarterly* EPS.

Consider a filing from the year 2024. The recency bonus for a column header containing "2024" is calculated as (2024-1990)/2.0=17.0. The score for a perfect, unambiguous quarterly signal ("Three months ended") is a fixed +12.0. This means that a column header containing only the year "2024" receives a higher score from the recency component alone than a column header with an explicit quarterly phrase but no year.

This imbalance systematically prioritizes recency over periodicity. For example, the parser might be faced with two columns:

- Column A Header: "Fiscal Year 2024 Projections"
- Column B Header: "Three Months Ended December 31, 2023"

The system would score them as follows (approximating positional bias as negligible):

- Score(A) $\approx$ (Annual Penalty) + (Recency Bonus for 2024) $= -8.0 + 17.0 = 9.0$
- Score(B) $\approx$ Quarterly Bonus + Recency Bonus for 2023 $= 12.0 + \frac{2023-1990}{2.0} = 12.0 + 16.5 = 28.5$

In this case, the logic holds. However, let's consider a different, plausible scenario:

- Column A Header: "2024"
- Column B Header: "Three Months Ended"

The scores would be:

- Score(A) $\approx$ (Recency Bonus for 2024) = 17.0
- Score(B) $\approx$ (Quarterly Bonus) = 12.0

Here, the column that is only marked with a recent year is incorrectly preferred over the column explicitly marked as quarterly. The system's primary objective—to find quarterly data—is subordinated to its secondary objective of finding recent data. This over-weighting of the year represents a key vulnerability in the column selection logic.

2.3 Recommendation: Evolving from Heuristics to Deterministic Ranking

To mitigate this vulnerability, the selection logic should be evolved from a weighted heuristic model to a more deterministic, rule-based algorithm that mirrors the system's primary objective. The suggestion to use full dates for ranking is sound and can be formalized into the following procedure:

- **Date Extraction:** For each column, parse the header text to extract all possible `(Month, Day, Year)` tuples. This can be achieved with a regular expression that maps month names (e.g., "January", "Jan") to their numeric equivalents.
- **Canonical Representation:** Convert each extracted date tuple into a canonical integer format, such as `YYYYMMDD`, which allows for direct and unambiguous comparison.
- **Periodicity Filtering:** First, filter the set of all columns to include only those whose headers contain a strong quarterly signal (e.g., matching a `QUARTERLY_RE)`. This enforces the primary objective.
- **Deterministic Selection:** From this filtered subset of quarterly columns, select the one associated with the maximum `YYYYMMDD` value.
- **Tie-Breaking:** The existing positional bias ($-0.15 * j$) should be retained, but only as a final tie-breaker in the rare event that two distinct quarterly columns share the exact same latest date.

This deterministic approach is not only more accurate but also more robust. Financial reporting language can evolve, but the structure of dates is stable. A date-based system is therefore more resilient to changes in disclosure phrasing than a model based on matching specific text strings, representing a strategic investment in the system's long-term maintainability.

2.4 Table 1: Heuristic Scoring Component Analysis

The following table provides a detailed breakdown of the scoring components in both the column picker and the per-candidate header scoring function, highlighting their intended purpose and potential failure modes.

Component	Function/Path	Value/Weight	Purpose	Analysis & Potential Failure Mode
Quarterly Signal	pick_best_quarter_ column	+12.0	Identify quarterly data columns.	**Under-weighted**. Can be overpowered by the recency bonus, leading to incorrect period selection.
Annual Penalty	pick_best_quarter_ column	−8.0	Avoid annual data columns.	Appears reasonably balanced against the quarterly signal but is also vulnerable to the recency bonus.
Recency Bonus	pick_best_quarter_ column	$\frac{year - 1990}{2.0}$	Prefer columns with more recent years.	**Over-weighted**. Recent-year bonus can exceed the quarterly signal, biasing toward recency over periodicity.

Continued on next page

Component	Function/Path	Value/Weight	Purpose	Analysis & Potential Failure Mode
Positional Bias	pick_best_quarter_column	$-0.15 \times col\_idx$	Tie-break to the leftmost column.	Effective minor tie-breaker; no major failure modes observed.
Quarterly Signal	header_context_score	+8.0	Boost candidates from quarterly columns.	Consistent with the column picker's goal.
Annual Penalty	header_context_score	−3.0	Penalize candidates from annual columns.	Weaker penalty than in the column picker, which is logical as this is a secondary check.

Section 3: Examination of Candidate Generation and Scoring within Tabular Contexts

The core of the tabular extraction logic resides in how the system identifies and scores potential EPS values. This process involves a clever "EPS Band" heuristic and a series of guard clauses that encode valuable domain knowledge, but it also contains a significant logical inconsistency.

3.1 The "EPS Band" and Fixed-Window Rigidity

The primary table-based extraction path operates on the concept of an "EPS Band." Upon finding a row containing the phrase "earnings per share" at index `idx`, the system defines a search window of `rows[idx: min(len(rows), idx+4)]`. This fixed, 4-row window is intended to capture the header row itself plus the subsequent "Basic" and "Diluted" data rows. This is an efficient heuristic that correctly models a very common data presentation pattern.

However, the rigidity of this fixed window makes the system brittle and highly sensitive to trivial formatting variations in the source HTML. For example, if a reporting template includes a single blank or decorative row for spacing between the "earnings per share" header and the "Basic" data row, the `idx+4` window may be too small to capture all the necessary data, potentially missing the "Diluted" value. Similarly, some financial tables place the "Basic" and "Diluted" rows above the summary "per share" line. The current forward-looking window (`idx` to `idx+4`) is completely blind to this alternative but valid layout. This brittleness means that minor, semantically meaningless formatting choices can cause the entire high-precision "EPS Band" logic to fail, forcing the system to fall back to less reliable methods when the correct data was readily available. A more resilient implementation might use a dynamic window that expands or shifts based on the content of adjacent rows.

3.2 The Basic vs. Diluted Scoring Anomaly: A Symptom of Architectural Decoupling

A critical logical flaw exists in the scoring of "Basic" versus "Diluted" EPS candidates. The system's preference between these two metrics is contradictory across its different extraction paths.

- **EPS Band Path:** Assigns a score of `+4.0` for "Basic" and **+1.0** for "Diluted", strongly preferring Basic EPS. This is in direct contradiction to an inline code comment which states: `"Prefer Diluted slightly over Basic (sample set expectation)"`.
- **Fallback Path:** When searching for "Basic" or "Diluted" rows outside of an EPS band, the system assigns `+2.0` for "Basic" and `+3.0` for "Diluted", correctly implementing a preference for Diluted EPS.

This inconsistency is more than a simple implementation bug; it reveals a fundamental contradiction in the system's objective function. The parser lacks a unified "scoring philosophy." The conflicting weights suggest that the two code paths were developed or modified in isolation, each with its own local, unstated assumptions about the relative importance of Basic vs. Diluted EPS. As a result, the system's output can be unpredictable. For the exact

same set of underlying data, it might return Basic EPS or Diluted EPS depending on trivial formatting differences in the table that cause one extraction path to be triggered over the other.

This anomaly is a symptom of a deeper architectural problem: business logic (the desired preference for a specific type of EPS) is tightly coupled with and fragmented across disparate implementation paths. This makes the system's behavior difficult to reason about and maintain. The immediate tactical fix is to unify these scores. The more strategic, architectural solution is to decouple the scoring logic entirely from the extraction flow.

3.3 The Role of Guard Clauses: Encoding Domain Knowledge

The parser effectively uses several guard clauses to encode domain-specific knowledge and improve precision. The hard cap on magnitude (`HARD_ABS_CAP = 20.0`) correctly filters out most non-EPS per-share metrics like Net Asset Value or Book Value, which often have much larger values.

Another particularly insightful rule is the guard: `if abs(val) >= 5.0 and not (BASIC_RE.search(label) or DILUTED_RE.search(label)): continue`. This clause encodes the expert knowledge that while EPS can sometimes exceed $5.00, such values are uncommon and, if legitimate, are almost always explicitly labeled as "Basic" or "Diluted". An unlabeled number of that magnitude within an EPS band is highly suspect and likely represents a different metric (e.g., share count in thousands).

The interaction between these rules creates distinct magnitude zones. For unlabeled candidates, values in the range $$, which the system currently accepts. The threshold of 5.0 is a "magic number," likely derived from intuition or a small sample set. This threshold is a prime candidate for empirical tuning. By implementing logging for every candidate that is rejected (or would be accepted) by this guard, it would be possible to collect data to statistically validate or refine this threshold, transforming a heuristic guess into a data-driven parameter.

3.4 Table 2: Basic vs. Diluted Scoring Unification Proposal

To resolve the scoring anomaly, a single, unified scoring model should be adopted across all extraction paths. The following table illustrates the current contradictory state and proposes a consistent model.

Extraction Path	**Current Basic Score**	**Current Diluted Score**	**Implied Preference**	**Proposed Unified (Basic)**	**Proposed Unified (Diluted)**	**Rationale**
EPS Band	+4.0	+1.0	Strongly Prefers **Basic**	+2.0	+2.5	Unify scoring across all paths. Slightly prefer *Diluted* as the more conservative, comprehensive GAAP measure for complex capital structures; the small delta preserves a weak preference while keeping both signals strong.
Fallback	+2.0	+3.0	Prefers **Diluted**	+2.0	+2.5	
Narrative (General)	+2.0	+1.0	Prefers **Basic**	+2.0	+2.5	

Section 4: A Critique of the Tiered Narrative Extraction Strategy

When tabular extraction fails, the system falls back to a three-pass narrative extraction process. This tiered approach is logical, but its implementation relies on heuristics that have their own distinct failure modes and inconsistencies.

4.1 Windowing Strategy: Fixed Characters vs. Semantic Boundaries

The narrative functions (`extract_from_narrative_strict`, `extract_from_text`, etc.) define their search context using fixed-character windows (e.g., `text[m.end(): m.end() + 120]`). This method is computationally simple and provides a basic proxy for textual locality.

However, this approach is brittle because character offsets are ignorant of linguistic and document structure. A fixed-character window does not respect semantic boundaries like sentences, paragraphs, or even the remnants of table cells that have been flattened into a single text string. A simple line break, a long sentence, or a stray piece of HTML boilerplate can easily push the target EPS value just outside the 120-character window, resulting in a false negative. Conversely, a number from a completely unrelated adjacent sentence that happens to fall within the character window can be incorrectly associated with an EPS phrase, creating a false positive. The reliability of narrative extraction could be substantially improved by shifting from character-based windows to semantic ones. A more robust, standard-library-compatible approach would be to split the text by sentence terminators (`.`, `?`, `!`) and define the search context as "the sentence containing the match, plus the N preceding and succeeding sentences."

4.2 Proximity Logic: A Robust Implementation

The general narrative function, `extract_from_text`, demonstrates a particularly well-designed and robust heuristic for identifying the correct numeric token. Its logic is to search for the "first number after, else nearest number before" the matched EPS phrase. This correctly models the two most common linguistic patterns: "EPS was $X.XX" and "$X.XX was the EPS". The implementation is subtle and effective: by searching the `after` text first, it prioritizes the most frequent structure. If that fails, it finds all numeric matches in the `before` text and selects the last one (`prev[-1]`), which correctly corresponds to the number closest to the search phrase. This demonstrates a mature understanding of the problem domain and avoids common pitfalls of naive proximity searches.

4.3 Scoring Incoherence Across Narrative Tiers

The narrative extraction module suffers from the same architectural flaw as the table parser: a lack of a unified scoring model. Each of the three narrative functions implements its own miniature, inconsistent, and contradictory scoring logic for Basic vs. Diluted EPS:

- `extract_from_narrative_strict`: Diluted `+0.75`, Basic `+0.25` (Prefers Diluted)
- `extract_from_text_eps_token`: Basic `+0.5`, Diluted `+0.25` (Prefers Basic)
- `extract_from_text`: Basic `+2.0`, Diluted `+1.0` (Prefers Basic)

This fragmentation means that the final narrative result is entirely path-dependent. The returned value depends on which function happens to find a candidate first, not on a holistic evaluation of all potential candidates found across the text. This again points to the need for a single, centralized `score_candidate` function that can be called from any extraction context—tabular or narrative—to provide a consistent and predictable evaluation based on a unified set of rules.