ChatGPT

Parsing & JSON-Generation Scripts

- **Input-type inference:** The code's `classify_input_type` (QuestionParser) only returns `"email"`, `"phone"`, `"ssn"`, `"zip"`, or `"initials"` (or `"text"` if none match) [1]. However, the Modento schema requires `control.input_type` to be one of `name|email|phone|number|ssn|zip|initials`. In practice this means many generic fields default to `"text"`, which is *not* an allowed `input_type`. For example, a "First Name" or "Age" field would be classified as `"text"`, violating the schema. **Fix:** Enhance the logic to detect and assign `"name"` or `"number"` where appropriate – e.g. treat fields matching `/name/i` as `input_type: name`, fields with numeric context (digits or words like "age", "number") as `input_type: number`. Alternatively, omit `input_type` when it's not one of the allowed values or map `"text"` → `"name"` for name fields. After parsing, the template-merge step (using the dental dictionary) may correct some fields' `input_type` (e.g. merging "first_name" templates enforces `input_type: name`), but any fields not matched in the dictionary will retain `input_type: "text"`, which must be fixed for schema compliance [1].

- **Unique keys (global):** The code enforces unique keys via `dedupe_keys` (top-level) and `TemplateCatalog._dedupe_keys_dicts` [2] [3]. These append suffixes to duplicate keys, and specifically set the signature key to `"signature"` [4] [5]. This satisfies the schema rule that *"Every question's 'key' must be globally unique, including nested questions under a multiradio"*. **Issue:** Nested questions inside a `multiradio` have their own `key` fields, but the current deduplication only runs on the top-level question list. We must verify that nested `control.questions` (if any) also get unique keys. If needed, extend the deduplication to include those nested items. (By schema, *"All nested question keys are globally unique"* [6].)

- **Single signature field:** The code enforces exactly one signature question and forces its key to `"signature"` [7] [8]. Specifically, if no signature is found a default is added, and if multiple exist extras are removed. At output, it checks `len(sig)==1 and key=="signature"`, else emits an error [9] [10]. This aligns with the schema rule *"Only one signature control in the entire form… its key must be exactly 'signature'"* [11]. No change needed here – the enforcement is correct.

- **Compound-field splitting:** The parsing code includes multiple routines (e.g. `split_compound_field_with_slashes`, `split_label_with_subfields`, `split_short_label_underscore_pattern`) to break multi-part lines into separate fields [12] [13] [14]. In particular, "Apt/Unit/Suite____" will split into separate fields [15] [16], and patterns like "Phone: Mobile Home Work" are also handled [17] [18]. This satisfies the requirement to split compound fields correctly. (No violation – but verify that all known compound patterns on your forms are covered by these routines.)

- **Dropdowns, radios, extras – option values:** The code's `fill_missing_option_values` ensures every option has a non-empty `"value"` [19]. It defaults blank values by slugifying the name (or using True/False for Yes/No) [20]. Validation also checks for any remaining empty values and reports warnings [21]. This meets the schema rule *“Every option must have a non-null, non-empty ‘value’”* [6]. The **only gap** is with `control.extra`: when the code adds an extra input (e.g. for “If yes, please explain”), it sets `{"type":"Input","hint":...}` but *does not include the* `"value"` *or* `"optional"` *keys*. The schema requires extras of type *Input* to include `value:true` and `optional:true` (see example in the guide) [22]. **Fix:** In those places, include `"value": true` (and `"optional": true` if needed) in the extra. For example: `control["extra"] = {"type":"Input","value":true,"hint":..., "optional":true}`. This ensures all radio extras follow the schema’s expected format [22].

- **Enforcing control object:** The parser always includes `"control"` (possibly empty) on each output question [23]. The schema requires a `control` object (even empty) for every question. The code’s `ensure_control_present` initializes missing controls (and default fields for each type) [24], so no questions lack a control. (E.g. input fields default to `input_type: "text"`) [25]. Just be sure no downstream code removes or nullifies the `control` field.

- **Key format validation:** The code’s `validate_form` warns if a key is not snake_case lowercase [26], matching the guide’s note that keys must be lowercase with only letters, digits, underscores [26]. This is correct. Ensure keys like `"q"` or ones starting with digits are avoided by the initial slugification.

- **Witness/header filtering:** The code explicitly drops any field whose title matches a “witness” regex [27] and also runs `scrub_headers_footers` on the raw text [28]. This complies with the rule *“Witness fields are not allowed”* and header/footer text should be filtered. (E.g. the loop `if not WITNESS_RE.search(q.title)` [27].) No further action needed unless you find orphaned address lines slipping through; if so, consider extending the skip patterns at [55†L18-L22].

Generated-JSON Output Checks

- **Schema conformance:** Use the Modento guide’s rules. For example, validate each JSON field has the required keys (`"key"`, `"type"`, `"control"`) and that `"control"` is an object. The scripts ensure this by construction. You should spot-check outputs: e.g. every question should list `"type": "input" | "date" | "radio" | "dropdown" | "multiradio" | "signature" | "terms" | ...` matching the schema types. Validate with the provided `validate_output.py` or similar.

- **Field types and order:** Confirm each question’s `"type"` is correct for its content. For instance, dates (labels with “Date”, “DOB”) should have `type: "date"` and a `control.input_type` of “past” or “any”. The code correctly sets date fields to type `"date"` (often with `input_type="any"` by default) [29] [30]. Fields like state or country should use `type: "states"` or just `"input"` with `input_type:"text"` if not matching special types. Emails and phones should still be `type:"input"` with `input_type:"email"` or `"phone"` [1]. Field ordering (Patient Info → Insurance → Referral → Medical History → Consents → Signature) isn’t enforced by schema but is a logical recommendation. Ensure the output groups sections in this order (postprocessing infers sections and merges templates [31], but if order is off, you may need to reorder fields in code or report it as a fix).

- **No disallowed fields:** Check that the JSON has **no witness-related entries** (the code filters these) [32]. Ensure header/footer text did not become a question (the scrubbing in parsing should prevent this) [28]. Also verify there are no "repeat" fields (the semantic dedupe removes exact duplicates of simple input questions) [33].

- **Nested questions (multiradio):** If any fields became `"type": "multiradio"`, verify that each has a `"control.questions"` array of well-formed question objects. The schema requires nested keys within `control.questions` to be unique [6] and of proper format. The parsing code does build nested questions (see archive example), but review them for uniqueness and completeness. If you find a nested question missing a `"control"`, add it (similar to top-level).

- **Required fields:** Ensure every question has a non-null `"control"` object. The code ensures one exists, but check that fields like signature have `"control": {}` and terms fields have the required `agree_text` or `html_text` set (see schema for terms [34]). For example, a `type:"signature"` should have `"control": {}` [35].

- **Extras format:** As noted above, any `control.extra` for a field of `type:"radio"` must follow the schema. The guide's examples show `extra` objects with a `value` and (usually) an `optional` flag [22]. Our code should be updated to include those keys, or remove incomplete extras if unnecessary.

- **Option values:** Every `"options"` array for `"radio"` or `"dropdown"` must have `{"name": string, "value": non-empty}` entries [36]. The script's fill logic handles this, but double-check outputs for empty values or nulls. Also ensure no duplicate option `name`s, which would be confusing to users (the validator flags duplicate names [37]).

Overall, most compliance rules are already addressed in code (and explicitly warned by the `validate_output.py` script). The key issues to fix are the ones where the code's defaults violate the guide's checklists: namely using `input_type: "text"` by default, and incomplete `extra` fields. Adjust those and rerun validation to ensure zero schema errors or warnings [19].

---

[1] question_parser.py
https://github.com/rontavious999/pdf-docx-to-json-docling-v1/blob/d61fa1039b593ac68127367f35f9dfca88a10779/docling_text_to_modento/modules/question_parser.py

[2] [4] [7] [8] [9] [10] [12] [13] [14] [15] [16] [17] [18] [19] [20] [21] [23] [24] [25] [26] [27] [28] [29] [30] [31] [32] [33] core.py
https://github.com/rontavious999/pdf-docx-to-json-docling-v1/blob/d61fa1039b593ac68127367f35f9dfca88a10779/docling_text_to_modento/core.py

[3] [5] template_catalog.py
https://github.com/rontavious999/pdf-docx-to-json-docling-v1/blob/d61fa1039b593ac68127367f35f9dfca88a10779/docling_text_to_modento/modules/template_catalog.py

[6] [11] [22] [34] [35] [36] Modento_Forms_Schema_Guide (1).txt
file://file_000000000608622fa7684d572cf5bd36

37 validate_output.py
https://github.com/rontavious999/pdf-docx-to-json-docling-v1/blob/d61fa1039b593ac68127367f35f9dfca88a10779/validate_output.py