Automatic Detection of Data Entry Errors

Tancrède Lepoint Muhammad Numair Mansur Martin Schäf Willem Visser

Amazon Web Services
{tlepoint, numairm, schaef, vissie}@amazon.com

ABSTRACT

Data entry errors pose a significant challenge in workflows involving manual input of information from unstructured sources into structured formats. These errors, often arising from fatigue or oversight, lead to costly corrections and operational inefficiencies. Existing approaches, such as input validation and independent reviews, while valuable, are resource-intensive and fail to address all error types effectively.

This paper introduces DocuBot, a system powered by large language models (LLMs) designed to detect data entry errors early in the process. By comparing natural language inputs with their structured representations, DocuBot identifies inconsistencies, flags potential errors, and, where possible, suggests corrections. When validation is inconclusive, it prioritizes entries for human review. This early detection reduces the reliance on traditional review methods, thereby lowering compliance costs and enhancing data integrity.

We evaluate DocuBot through a series of experiments, demonstrating its high precision in error detection and its capacity to streamline workflows that require human oversight. While not yet suitable for full automation, DocuBot represents a promising step toward minimizing the financial and operational impacts of data entry errors.

ACM Reference Format:
Tancrède Lepoint Muhammad Numair Mansur Martin Schäf Willem Visser . 2025. Automatic Detection of Data Entry Errors. In *Proceedings of The 1st International Workshop on Advancing Static Analysis for Researchers and Industry Practitioners in Software Engineering. (STATIC 2025).* ACM, New York, NY, USA, 10 pages.

1 INTRODUCTION

Human data entry errors are a common source of errors in software. Many information systems rely on humans entering data from an unstructured source, such as a transcript or a legal text, into a machine readable form (e.g., via a graphical user interface). Humans get tired, lose focus, and are prone to making typos. The larger and more complex the amount of entered data becomes, the more likely it is that fatigue and boredom set in and data entry errors occur.

Literature offers many examples of data entry errors, their impact, and the cost of correcting them. A more humorous example is Erich von Wolf entering the iron content of spinach as 35mg instead of 3.5mg [2] in 1870 which was only corrected in 1937 (and gave rise to Popeye the Sailor). More serious examples include NASA's Mars Climate Orbiter where data from tables in pound-second units was mixed with data in newton-second units, which sent the orbiter into a lower orbit and ultimately led to its destruction. Another example is the New York Stock Exchange (NYSE) Decimalization Error in 2001, where a stock was briefly traded at $112 instead of $1.12 because of a decimal point entry error.

These examples demonstrate that data entry errors can be hard to detect, expensive to correct, and often have real economic consequences. Some data entry errors can be prevented with input validation using techniques, such as schema validations, sanity checks on the range of values, or using control sums to ensure that multiple data entries add up. Unfortunately, some data entry errors are still hard to catch with these techniques, as the entered data may be plausible within its domain but still does not match what was written in the original source.

Many jurisdictions have introduced legal requirements to implement controls that can detect data entry errors, like the Sarbanes–Oxley Act in the United States (SOX for short). A typical control for SOX compliance is to require independent reviews by multiple individuals to detect data entry errors.

While necessary, these independent reviews are not perfect, slow down business processes, and are thus expensive. Often, reviewers are scarce, and just as in code reviews [16], the entered data can sit idle for hours or days until it is reviewed. Ideally, we would like to detect as many data-entry errors as possible before data reaches the reviewer to avoid unnecessary iterations and further slow-down of the process.

In this paper, we report on an experiment where we use large language models (LLMs) to reduce the cost of this manual review process. We implement a system called DocuBot, that assists a human when entering data from natural language document into a graphical user interface (GUI). DocuBot takes as input the natural language data and the JSON representation of the data that is entered in the GUI. DocuBot then constructs a series of prompts from the JSON representation and uses the LLM to ask if this data matches what is stated in the natural language data. If the LLM is able to confirm that the data matches, we provide a check mark, if a data entry error is detected, we inform the human (where possible, we provide a suggested fix), and if the LLM is inconclusive, we prioritize the entry for human review.

We demonstrate the functionality of DocuBot on an example in Section 2. In Section 3, we discuss how we generate LLM prompts from structured data and how we process the LLM's response to provide feedback to the user

We conduct a series of experiments in Section 4, and discuss their results in Section 5 to answer the following research questions:

RQ1 Is DocuBot sufficiently precise to detect data entry errors without producing too many false alarms?

STATIC 2025, 29 April, 2025, Ottawa, Canada

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00

RQ2 If DocuBot can detect data entry errors precisely, can we use it to replace the human and automate the data entry process entirely?

2 MOTIVATING EXAMPLE

We illustrate the concept of a pricing agreement and how we validate it using the example in Figure 1. Here, we see an agreement between a vendor and a buyer that sets the price of fruit under certain conditions. The agreement has a period during which it is valid (here, the year 2025), some commitments where the buyer agrees to minimum purchase volumes in terms of quantity and/or price, and a set of specific rates that the vendor offers if the commitment is met.

In an ideal world, the buyer would provide an API or a standardized interface to the vendor to share pricing agreements to eliminate possible data entry errors by removing the human from the loop. However, in practice, this is often not possible because one buyer might want to interact with a variety of vendors that have different representations for their agreements, and forcing vendors to use a predefined API may create an unnecessary cost of entry.

Instead, we want to make it easy for vendors to share agreements in a more or less arbitrary format, and we have a human analyst that reads the agreement and enters it into our system in a standard JSON format, as shown in Fig 2.

The JSON schema provides generic building blocks for agreements, such as rates and commitments, and provides some basic types that are needed to express the relevant variables in an agreement, such as dates, monetary values, and units.

In practice, such a JSON representation could be generated from a general purpose UI, where analysts can click together the terms they see in the agreement and enter the necessary values. Even this relatively simple illustrative example already has 20 values that need to be entered by the analyst. In practice, these agreements are significantly bigger and can contain hundreds or thousands of values that need to be entered by hand. At this scale, even a low human error rate during data entry can have a significant business impact.

To reduce the risk of human data-entry errors, we built DocuBot; a system where analysts can specify a set of input validation *question templates* for each node type in their JSON schema. When an analyst enters a new agreement, DocuBot automatically instantiates these question templates and prompts an LLM by providing the input agreement and the generated set of questions.

Figure 3 shows how analysts specify a question template for the node types `AgreementPeriod` and `Bonus` from our schema. For each node type, the analyst provides a prompt that can be answered with yes or no by an LLM. The prompt contains placeholders that can reference concrete values in an agreement (via JSON-pointers).

DocuBot first prompts the LLM with the text of the input agreement, and a short list of instructions, to answer questions with *yes* or *no* tags in a specific format to allow DocuBot to parse the LLM response more efficiently. Then the JSON representation provided by the analyst is used to instantiate the question templates. Each template can be instantiated multiple times depending on how often this node type appears in the agreement. DocuBot then prompts each question to the LLM and scans the response for the predefined yes and no tags. If the LLM answers with a *yes*, we assume the data entry was correct. If the answer is *no*, we flag it as a potential data-entry error to the analyst including the full prompt output for debugging. If the answer is inconclusive, we flag this element for human review.

The instantiated question templates and corresponding LLM responses for our example are shown in Fig 4. For the first question, the LLM confirms that the dates of the agreement period match the dates in the original document. One advantage of using an LLM here is that these models tend to be good at comparing values, like dates, even if they are not formatted in the same way.

For the second question, we instantiate the question template for `Bonus` nodes with the relevant values for produces, quantities, and units. Here, the LLM does spot a discrepancy between the JSON data and the original input text: the agreement states that, for every $100 spent on apples, the customer receives 5 lbs of oranges (which is equivalent to $9.95 of oranges). However, the JSON data that was used to populate the question template states that the buyer gets only $5 worth of oranges for the same spent on apples. With a minimum commitment of 2,000 apples this would result in a loss of $99 for the buyer, or approximately 1% of the agreement value.

This example motivates why having using such question templates as a mechanism against data entry errors is interesting.

3 APPROACH

We implement DocuBot for the type of agreements discussed in Section 2. DocuBot takes as input the textual and JSON representation of an agreement and a set of question templates. As output, it produces a list of findings for each instantiated question that results in an unexpected answer by the LLM, together with the relevant node in the JSON document and the LLM's output.

A question template (as shown in Figure 3) consists of a node type in the JSON document on which the question will be instantiated, a format-string representing the prompt, and a list of variables for the format-string that are expressed as JSON-path expressions relative to the matched node. For each question template, we find all nodes in the JSON document that match the node-type in the question template. Then, for each node, we resolve the values of each JSON-path expression in the variables relative to the path of the matched node. Finally, we insert these values in order into the format-string to construct the instantiated question we want to ask to the LLM.

Depending on the node-type and JSON representation of the agreement, each question template may be instantiated multiple times (or not at all), so the number of LLM prompts is not directly correlated with the number of question templates that are provided to DocuBot.

Question template design. DocuBot's ability to detect data-entry errors depends on the design of the question templates. For our experiment, we follow the existing best-practices by providing at least one in-context example [10] and a chain-of-thought (CoT) [19] why this example should or should not be flagged.

Our question template is a super-set of the example shown in Figure 3:

1. Pricing Terms.

Term	Meaning
Agreement Period	January 1, 2025 – December 31, 2025
Vendor IDs	8f7b1a3e-6c2d-4f9d-b8a5-1e9d3f2c0e4b
Specific Rates	The following discounted rates are valid through the Agreement Period: Apples $1.59 per Unit Bag of Oranges $5.97 per 3 lbs 1 kg Cherry Good Bag $12.02 per 1 kg For every $100 purchase of apples, the buyer receives 5 lbs of oranges.
Use Commitment	2000 apples 3000 lbs of oranges
Spend Commitment	$10,000

2. Termination of Prior Agreements. The Prior Agreements is terminated as of the first day of the Discount Term.

3. Price Protection. The vendor will charge the lesser of the agreed rate and the going market rate.

4. ...

Figure 1: Example of an agreement with a vendor. Such an agreement contains a period during which it is valid, a list of agreed upon rates, and some minimum use/spend commitments for the agreement period. Agreements can also contain other clauses such as price protection, etc.

```
{
  "Agreement": {
    "AgreementPeriod": {
      "StartDate": "2025/01/01",
      "EndDate": "2025/12/31"
    },
    "VendorIDs": [
      "8f7b1a3e-6c2d-4f9d-b8a5-1e9d3f2c0e4b"
    ],
    "SpecificRates": {
      "Apples": {
        "Rate": 1.59,
        "Unit": "unit"
      },
      "BagOfOranges": {
        "Rate": 5.97,
        "Unit": "lbs",
        "UnitFactor": 3
      },
      "CherryGoodBag": {
        "Rate": 12.02,
        "Unit": "kg"
      },
      "Bonus": {
        "PurchaseInterval": {
          "Product": "Apples",
          "Amount": 100,
          "Unit": "USD"
        },
        "FreeResources": {
          "Product": "Oranges",
          "Amount": 5,
          "Unit": "USD"
        }
      }
    },
    "Commitments": {
      "UseCommitment": {
        "Apples": 2000,
        "Oranges": 3000
      },
      "SpendCommitment": 10000
    },
    "PriceProtection": true
  }
}
```

Figure 2: The JSON representation of the agreement from Fig 2. This representation is typically generated via a UI where an analyst encodes the details of the agreement.

```
[
  {
    "NodeType": "AgreementPeriod",
    "Question": "Does this have an agreement period from %s to %s?",
    "Variables": ["/StartDate", "/EndDate"]
  }, {
    "NodeType": "Bonus",
    "Question": "Does this guarantee a bonus of %s%s of %s for every %s%s of %s?",
    "Variables": [
        "/FreeResources/Unit",
        "/FreeResources/Amount",
        "/FreeResources/Product",
        "/PurchaseInterval/Unit",
        "/PurchaseInterval/Amount",
        "/PurchaseInterval/Product"
    ]
  }
]
```

Figure 3: We now define a set of questions for each node node type in our JSON schema used to store the agreement in Fig 2. The questions consists of the node type they apply to, a format string, and a list of JSON pointers to the values for the format string.

> **DocuBot:** Does this have an agreement period from 2025-01-01 to 2025-12-31?
> **Model:** Yes, the pricing agreement explicitly states that the Agreement Period is from January 1, 2025 to December 31, 2025, which corresponds to the dates 2025-01-01 to 2025-12-31.
> **DocuBot:** Does this pricing agreement guarantee a bonus of $5 of oranges for every $100 spent on Apples?
> **Model:** No, the pricing agreement specifies that the buyer **will receive 5 lbs of oranges** (not $5 worth of oranges) for every $100 spent on apples. The bonus is described in terms of weight (5 lbs), not monetary value. This is an important distinction, as the monetary value of the bonus will depend on the current price of oranges, which is given as $5.97 per 3 lbs.

Figure 4: Simplified example of a conversation between DocuBot and a LLM. The prompts are generated by applying the question template from Fig 2 to the JSON data in Fig 2. We can see how DocuBot sport the data entry error in the `Bonus` entry of the JSON document. In practice, these prompts are more detailed which we will discuss in the following sections.

```
ruleId: example-of-rule-structure
context: >
  An optional context to describe the problem
examples:
  - input: >
      An example agreement
    question: >
      An example question about the agreement
    answer: The expected answer for this question
    chainOfThought:
      - expected thought 1
      - expected thought 2
question: >
  The actual question that we want to ask
nodeType: /Expected/Node
variables:
  - /var1
```

The question template requires at least the a `ruleId`, a `nodeType`, a `question`, and at least one variable. DocuBot then matches each the `nodeType` in the JSON representation of the agreement. For each matched node, it instantiates the format string in the `question` with the values obtained by evaluating the `variables` relative to the matched `nodeType`.

In addition, the question template can include a `context`, which can be used to provide some general context about the task, and a list of `examples`. Each `example` consists of an example input, the question, for that input, the expected output, and an optional chain-of-thought (CoT). The example input typically is an agreement or fragments of an agreement. The question in the example should be an instantiated version of the main `question` in the template, and the expected answer should be what we want to see as an output. The optional chain-of-thought in the example can be used to explain step by step how we want to get from the example question to the expected answer.

DocuBot expects the user to use specific `{{YES}}` and `{{NO}}` tags in the answers of the example questions. These are the same tags that DocuBot uses to parse the LLMs response to the main `question`.

DocuBot is designed to construct a prompt that is adequately formatted for the LLM and wraps examples, chain-of-thought descriptions, the input agreement, and the actual question into the format preferred by the LLM. This way, we can abstract the specifics of the underlying model from the user. Internally, DocuBot relies

on the Amazon Bedrock APIs to interact with the model, and uses different configurations based on the selected model.

We give a full example of one of the question templates used in Section 4 in Figure 5. One of the beautiful things about language models is that things can be messy. It's perfectly fine to only use parts of an agreement in the example, and formatting and indentation does not really matter. The creator of the question template can copy and paste these examples together with relatively low effort, and just iterate the question design and the CoT instructions a few time until it achieves the desired results. The author also does not need to worry about the format of dates, numbers or currencies since the LLM will be able to understand it either way.

We discuss the full list of question templates used in our experiment, and the impact of using examples and chain-of-thought in Section 4.

Security Consideration. Before we instantiate the question templates, we ensure that DocuBot only has access to the minimal amount of domain specific information necessary. We implement a pseudonymization algorithm (e.g., [3–5]) that consistently replaces agreement specific values by tokens of the same format. For example, the UUID of the Vendor ID in our example from Section 2 is replaced by another UUID in the textual representation and the JSON document.

4 EVALUATION

To evaluate DocuBot, we design 3 question templates; one for each type of term shown in our examples in Section 2:

Period Checks if the dates `AgreementPeriod` match the agreement period stated in the text. This rule uses 2 variables (for start and end date), and it provides 3 examples with chain-of-thought. The full version of Period is shown in Figure 5.

Commitment Ensure that each `Commitment` value, unit, and period match the textual presentation. In this rule we use 4 variables, the unit and value of the commitment, and its start and end date, and we provide 3 examples with chain-of-thought.

Rates Checks if the rates in `SpecificRates` and their respective units match what's in the agreement. Further, if the rates specify a validity period, the rule also ensures that the dates match. In this rule use 3 variables, the rate, and the start and end date of its validity. We provide 4 examples with chain-of-thought.

For space reasons, we only include the template for Period in Figure 5. The templates range from 69 lines (in yaml format) for the Period template to 109 lines for the Rates template.

We try not to over-fit our question templates to the agreements used in our experiments by using a different set of agreements for testing when design the templates. However, a certain bias has to be assumed.

4.1 Dataset

To evaluate if DocuBot is suitable to detect data entry errors, we obtained a set of 50 expired agreements from an Amazon internal team that interacts with external vendors. We chose expired agreements because we know they that were ingested correctly. And we chose them because we don't experiment on live/active/production data. For each agreement, we download the textual representation and the corresponding JSON file.

On these 50 agreements, our question template Period matches exactly 50 nodes (since each agreement has exactly one Agreement Period node). Commitment matches 86 times since agreements can make more than one commitment, and Rates matches 32 times, since not all agreements specify a rate.

That is, between our 50 agreements, DocuBot will match on $50 + 86 + 32 = 168$ nodes, and thus send 168 requests to the LLM to check if the values under this node match the values in the textual representation of the agreement.

Ground Truth. To ensure that this set of agreements is a suitable ground truth, we perform a manual inspection of each text/JSON pair with a domain expert where we validate that the entered JSON data indeed matches the text in the agreement. That is, we expect DocuBot to report no findings for any of the three question templates on this data set.

Mutated Data Set. To evaluate to ability of DocuBot to detect data entry errors with our 3 question templates, we create mutated versions of our ground truth. For each question template, we identify the matching node types in each JSON agreement. For each matched node, we choose one of the variables in the template at random, and identify the value in the JSON agreement that is referenced by the variable and mutate it. Date values get mutated by adding a random temporal offset (e.g., +1 day or −3 years). Currency values get mutated by either adding an offset to the numeric value, or by replacing the unit of measurement (when available).

Using the ground truth, and the mutated ground truth, we can now measure the precision. The mutation of the ground truth provides us with a fully labeled set of JSON agreements for each question template to measure:

- TP true positives, where DocuBot reports a data-entry error for a mutated value.
- FN false negative, where DocuBot fails to report a data-entry error on a mutated value.

Further we use the ground truth to measure:

- TN true negatives, where DocuBot does not report a data-entry error on the ground truth. I.e., DocuBot matches on a node type, and the LLM confirms that the values in the JSON representation match those in the textual representation.
- FP false positives, where DocuBot reports a data-entry error on the ground truth.

4.2 Results

We run DocuBot with our three question templates on the 50 agreements and their mutated versions. As LLM, we use a Claude Haiku 3.5 model via Amazon Bedrock in its standard configuration. To control for potential non-determinism in the response of the LLM, we run each query 3 times, and we distinguish between unanimous decisions where all 3 prompts yield the same answer, and *majority* decisions where 2 out of 3 prompts yield the same answer. We did not encounter any cases of inconclusive responses.

We only count LLM responses if they can be parsed by DocuBot. That is, if the response contains a predefined tag, `{{YES}}`, we count it as a *negative* because the the value in the question template matches the textual input, and if the response contains a `{{NO}}`,

```
ruleId: PERIOD
context: >
  I want you to check if an agreement specifies a given Agreement Period. You need to
  ensure that the start and end that of the discount term match exactly the dates that
  I will give to you.

examples:
  - input: >
      Agreement Period    April  1,  2023  – March  31,  2026
      Vendor ID           1234-5678-1234
      Apples              $100
      Upfront Payment     Contract Year 1:  $5
    question: >
      Does this contract define an agreement period that starts on the date 2023/04/01
      and ends on the date 2026/03/31?
    answer: >
      {{YES}}, the agreement defines an agreement period from April  1,  2023 to March  31,  2026,
      which matches the
      provided start date of 2023/04/01 and the provided end date 2026/03/31.
    chainOfThought:
      - The start of the agreement period April 1, 2023 matches exactly the expected start date of 2023/04/01
      - The end of the agreement period March 31, 2026 matches exactly the expected end date of 2026/03/31
      - Therefore the agreement period matches the expected dates and the answer is {{YES}}

  - input: >
      Agreement Period      January  1,  2020  – December  31,  2020
      Vendor IDs:           1234-5678-1234
      General Discount      7.5%
      Spend Commitment      $5,500
    question: >
      Does this contract define a discount term that starts on the date 2020/01/01 and ends on the date 2020/11/31?
    answer: >
      {{NO}}, the agreement defines a agreement period from January  1,  2020  – December  31,  2020,
      which does not match the
      provided end date of 2020/11/31.
    chainOfThought:
      - The start of the agreement period January  1,  2020 matches exactly the expected start date of 2020/01/01
      - The end of the agreement period December  31,  2020 does not match the expected end date of 2020/11/31
      - Therefore the agreement period does not match the expected dates and the answer is {{NO}}

prompt: >
  Does this contract define an agreement period that starts on the date %s and ends on the date %s?
match:
  - $..AgreementPeriod
variables:
  - /StartDate
  - /EndDate
```

Figure 5: Abbreviated version of question template Period. For space reasons, we omit one of the example in the actual template.

	Ground Truth Data				Mutated Data				Scores		
	TN	majority TN	FP	majority FP	TP	majority TP	FN	majority FN	Precision	Recall	F1
Period	50	0	0	0	49	1	0	0	1	1	1
Commitment	86	0	0	0	12	73	1	0	1	0.98	0.98
Rates	30	2	0	0	29	3	0	0	1	1	1
Total	166	2	0	0	90	77	1	0	1	0.99	0.99

Table 1: Results of running DocuBot for our 3 question templates over rounds. In total, DocuBot runs 168 LLM queries in each round. For the categories true/false positive/negative, we distinguish between unanimous decisions, where all 3 rounds yield the same result and majority decisions, where 2 out of 3 runs yield the same result.

we count it as *positive*, if both or neither are found in the response, we count it as inconclusive.

On the ground truth we measure the true negatives (TN) and the false positives (FP) and on the mutated ground truth we measure true positives (TP) and false negatives (FN). That is, if all LLM responses can be parsed we expect the sum of the true negatives and false positives to add up to the total number of queries of 168, and we expect the same for the sum of true positives and false negatives on the mutated data.

The results are shown in Table 1. On the ground truth data, we receive no false positives, and only 2 majority true negatives (where 1 out of 3 runs had a dissenting opinion) for our 168 prompts.

On the mutated data set we only receive 1 false negative of the 168 expected findings. For Period and Rates most decisions are also made unanimous, where the LLM produces the same result in all three runs. For Commitment, however, only 14% of decisions are made unanimously and 86% are majority decisions.

In summary, DocuBot has a perfect precision score of 1 on our data set and a recall of 0.99 as well as a F1 score of 0.99.

Impact of CoT and examples. We also want to understand the impact that providing examples and the associated chain-of-thought reasoning has on the usability of the question templates. To that end, we re-run the experiment above, but we omit the CoT, or the CoT and examples from the prompts respectively. Figure 2 shows the results of this experiment.

Omitting examples or the CoT reasoning for examples does not have an impact on DocuBot's precision. No false positives are introduced by omitting these additional instructions and only 2 out of 168 prompts yield a result that cannot be mapped to a yes/no answer (this means that model rambles about unrelated things without emitting the specific yes/no tag that we demand in the instructions).

However, we see a drop in DocuBot's recall from 0.99 to 0.75 if we omit the CoT reasoning and a further drop to 0.58 if we omit the examples entirely. Period, which is arguably the simplest question template loses 10% recall if no examples are provided. For Rates, we see a 7% drop (1 prompt yields a false negative, and 3 more prompts yield an inconclusive answer). The biggest drop in recall is for Commitment which drops to 0.28 (not that, for computing recall, we also consider cases where the LLM does not provide a decision as false negatives).

Impact of the selected model. We further want to investigate the effect of using a different LLM with DocuBot on our data set. To that end, we repeat our first experiment of running our 3 question template on the 50 agreements and their mutated versions for 3 rounds. This time, we use Meta's Llama 3.2 3B Instruct and the Llama 3.2 11B Instruct models. As in the first experiments we include the examples and CoT in the prompt.

Results of running DocuBot with these models are shown in Table 3. For the 11B parameter Llama model, we get a precision of 0.94, recall of 0.85 and an F1 score of 0.89. And as expected, we get slightly lower results for the 3B parameter model with a precision of 0.77, recall of 0.76 and an F1 score of 0.76.

4.3 Extraction of JSON with DocuBot

Finally, we want to know if DocuBot can be used to extract the JSON representation from an agreement directly. If we can extract the JSON instead of relying on human ingestion, we can further shorten the related business process.

For this experiment, we modify our question templates and tell the model to generate the JSON directly. In the examples, we show a pair of textual agreement and the resulting JSON term(s) from the ground truth. That is, in this configuration, instead of producing yes/no answers, DocuBot produces a (potentially empty) list of terms per question template for each agreement in our data set.

We evaluate the generated JSON against our ground truth as follows: if DocuBot generates a JSON term for a question template that exists in the JSON from the ground truth, we count it as a True Positive. If the ground truth contains a JSON term that was not generated by DocuBot, we count it as a false negative. That is, true positives and false negatives combined add up to the 168 terms that we used in the previous experiments. If DocuBot generates a term that is not found in the ground truth, we count it as a false positive, and if it does not generate a term for a question template and the ground truth does not contain this term either, we count it as true negative. Note that, if DocuBot creates a term with an unexpected value it can be double counted as false positive (because it does not match the ground truth), and as false negative (because the related term in the ground truth does not match it precisely).

The results of this experiment using Claude Haiku are shown in Table 4. For Period we obtain the best results with a high recall of 0.94 and a precision of 0.78 and for Rates we have the lowest precision and recall. These results are expected, as Period only has two variables that need to be identified and every agreement has exactly one period, but Commitment and Rates can occur multiple times (or not at all) in the same agreement.

In the following we discuss the conclusion we draw from these experiments.

	Ground Truth Data				Mutated Data				Score					
	example no CoT		no example no CoT		example no CoT		no example no CoT		example no CoT			no example no CoT		
	TN	FP	TN	FP	TP	FN	TP	FN	P	R	F1	P	R	F1
Period	50	0	50	0	50	0	45	5	1	1	1	1	0.9	0.95
Commitment	86	0	86	0	24	7	24	1	1	0.28	0.43	1	0.28	0.43
Rates	30	0	32	0	28	1	30	0	1	0.87	0.93	1	0.93	0.96
Total	166	0	168	0	102	8	99	6	1	0.6	0.75	1	0.58	0.73

Table 2: The results of running experiments Table 1 without the CoT instructions or without the examples and the CoT instructions on our data set, together with precision (P), recall (R), and F1 score. For computing the scores we count cases where the LLM did not produce a response as false negatives.

	Llama 3.2 11B Instruct								Llama 3.2 3B Instruct							
	Ground Truth Data				Mutated Data				Ground Truth Data				Mutated Data			
	TN	mTN	FP	mFP	TP	mTP	FN	mFN	TN	mTN	FP	mFP	TP	mTP	FN	mFN
Period	49	1	0	0	47	1	0	2	26	5	15	4	37	6	3	4
Commitment	12	59	0	0	3	50	0	19	3	67	0	11	12	40	0	13
Rates	15	7	0	9	27	6	0	0	12	10	1	7	28	5	0	0
Total	76	67	0	9	77	67	0	21	41	82	16	22	77	51	3	17

Table 3: Re-run of the experiment from Table 1 using Meta's Llama 3.2 11B Instruct and Llama 3.2 3B Instruct models. As before, we aggregate the results over 3 runs of the LLM, and we distinguish between unanimous decisions and majority decisions. In total, 168 prompts are sent to the models and we only count true/false positives or negatives if the model gives an answer that can be parsed.

	TP	FN	TN	FP	Precision	Recall	F1
Period	47	3	0	13	0.78	0.94	0.86
Commitment	60	26	19	18	0.76	0.70	0.72
Rates	13	19	26	22	0.37	0.41	0.39
Total	120	48	45	53	0.69	0.71	0.70

Table 4: Results of using DocuBot generating the JSON representation from the textual representation on our data set. Here, a TP means that DocuBot correctly generated the expected JSON, and FN means the JSON ground truth contained a term but DocuBot did not generate it. So TP and FN add up to the total number of 168 from our previous experiments. TN means that it did not generate a term on an agreement where none was expected, and FP means it did create a term that did not have a corresponding term in our JSON ground truth.

5 DISCUSSION

We discuss our research questions from the beginning of this paper using the results from our experiments. Then we discuss generalizability and threats to validity and some directions for future work.

5.1 RQ1: Data-entry error detection

The main research question that we set out to answer is, if a system like DocuBot can effectively detect data entry errors and thus help shorten the human peer-review process.

In our target configuration, using Claude Haiku and question templates with examples and CoT, and a best-of-three prompting approach in Table 1, DocuBot delivers almost perfect scores on our data set. Our experiments show that DocuBot has a reliably high precision, even if we remove details from the prompt in Table 2, or switch to smaller models in Table 3. As expected, recall (i.e., it's ability to detect data-entry errors) suffers if we use a smaller model or omit examples or chain-of-thought reasoning.

Our main insight is that the DocuBot approach consistently has a high precision across our experiments. This makes it easy to integrate this approach into existing processes. Since DocuBot rarely produces false positives, it can be integrated into a human review process without causing distraction. Question templates with low recall (i.e., it fails to detect data-entry errors), can be gradually improved to provide incremental value to the human reviewers.

Comparing the results of Claude Haiku, Llama 11B, and Llama 3B in Table 3 further suggest that model size tends to affect recall more than precision, and that larger models perform better on the same question templates.

More importantly, we are not aware of a comparable alternative to a language model for detecting this type of data entry error. The only other automation that we could identify is to use techniques like controlled natural language to best-effort parse the textual representation of the agreement. Compared to the low engineering effort of writing a DocuBot question template (by just creating

a yaml file), we do not believe that there is a more cost effective technique to detect data entry errors.

Hence we answer **RQ1** with **yes**; in our setting, DocuBot effectively detects data entry errors that would require iterations in our human review process, while have almost no false positives which would slow down the review process. Thus, DocuBot has a net-positive impact on our processes.

5.2 RQ2: Agreement Generation

Our results of using DocuBot to generate JSON agreements in Table 4 do not reach the same levels of precision and recall as our results when detecting data-entry errors for RQ1. We see high numbers of False Positives and False Negatives across all question templates.

However, we also see 120 out of 168 (71%) terms where DocuBot is able to create a correct term. That is, even our current version could provide some cost savings by creating a first draft of the agreement for the human analyst.

As with all LLM backed tools, it is difficult to say how much of this result is due to poor prompt engineering on the side of DocuBot, how much of it could be improved with a more advanced/specialized model, and how much of it could be improved by making adjustments to the JSON representation of agreements.

In it's current shape, we have to answer **RQ2** with **no** for DocuBot. We still are confident that agreement generation can speed up our processes by generating drafts, but, at this point, a human review will still always be required.

5.3 Validity Considerations

We cannot claim that a system like DocuBot, which relies heavily on a large language model, will generalize to other domains of documents or for other representations of structured data. In our experiments, we try to control for some of this by running multiple iterations of the same query and by comparing more than one model.

Our experiments give us confidence that DocuBot's LLM-backed approach to data-entry error detection works for our domain, and that similar question templates can be written for other domains. However, we are also confident that we could design a JSON representation of the data that would be adversarial to our approach. E.g., if we store the agreement in a tabular format with rows containing the commitment and credit for each calendar week (e.g. calendar week 32 in 2024), our precision and recall might suffer, if the LLM needs to convert from a specific calendar week to the date ranges in the text for every prompt.

Similarly, there is always the threat to validity based on the size of the data set (here 50) or the number of models used (here 3). Since we are not claiming that DocuBot works for all domains, or for any model, we see diminishing returns in further increasing the data-set or the number of models.

Performance. We do not discuss the performance of DocuBot. In our experiments, we did not experience any response time of the LLM slower than 5 seconds. The main limiting factor in our experiments is the availability of compute. Depending on the employed model, we are limited to 200 requests per minute, which means that we have to run some queries sequentially for larger batch processes. However, compared to the hours or days of delay while waiting for a human reviewer to become available, compute time is not a limiting factor in our setting.

5.4 Future work

Ultimately, we would like to use a system like DocuBot to fully automate the data entry process. Our experiments in Table 4 show that DocuBot is indeed able to generate over 70% of terms correctly, which is promising, but still far from a full automation. While we believe that this number can improve through more powerful models or better prompt engineering, we believe that DocuBot needs to be extended with additional reasoning capabilities to ensure that generated JSON is reasonably correct before presenting the generated output to a human. First, we want to investigate if DocuBot's data-entry error detection can be combined with the generation capabilities to create a counter-example guided refinement during generations. Further, we want to experiment with tracking which parts of the textual representation are used during generation, to get a notion of coverage during generation, and to experiment with visual markers to accelerate human reviews.

6 RELATED WORK

Our work intersects with several research areas: data entry error detection, the application of LLMs to software engineering tasks, and automated document understanding.

Data entry error detection. Traditional approaches to detecting data entry errors have focused on statistical methods and rule-based validation. Redman [14] provides a comprehensive overview of data quality challenges, including data entry errors, estimating that up to 5% of data entry fields contain errors. Early detection systems relied primarily on range checks, format validation, and cross-field consistency rules [11, 13]. More sophisticated approaches incorporate machine learning techniques to identify anomalous entries [15], though these typically require substantial training data and careful feature engineering.

LLMs in Software Engineering. Recent years have seen growing interest in applying LLMs to software engineering tasks. Chen et al. [7] demonstrated that LLMs can effectively understand and generate code, while Dvivedi et al. [8] showed their potential for documentation tasks. Particularly relevant to our work is research by Wang et al. [18] on using LLMs for consistency checking between code and documentation.

The use of chain-of-thought prompting [19] and few-shot learning [6] has been shown to improve LLM performance on complex reasoning tasks. Our work builds on these findings, applying similar techniques to the specific challenge of data entry validation. Unlike previous work that focuses on code-documentation consistency, we address the broader challenge of validating structured data against natural language sources.

Document Understanding and Information Extraction. Automated extraction of structured information from documents has been extensively studied. Traditional approaches employ techniques like named entity recognition and relation extraction [1, 20], while more recent work leverages deep learning models [17, 21]. However, these approaches typically require extensive training data and are sensitive to document format variations.

Several commercial systems attempt to automate the extraction of information from contracts and legal documents [9, 12]. These systems often combine optical character recognition (OCR), natural language processing (NLP), and domain-specific rules to process and extract relevant data efficiently. While they can achieve high accuracy for well-structured documents, they struggle with variations in format and language, leading to high maintenance costs as document formats evolve [22].

Our approach differs from previous work in several key aspects. First, we focus on validation rather than extraction, though our results suggest potential for extraction applications. Second, we leverage LLMs' natural language understanding capabilities without requiring extensive training data or rigid rules. Finally, our system is designed to integrate with existing human review processes rather than attempting full automation.

7 CONCLUSION

In conclusion, DocuBot demonstrates the potential of leveraging large language models (LLMs) to enhance data-entry workflows by identifying errors early in the process. While the system is not a replacement for human oversight, its primary value lies in minimizing costly iterations by preemptively addressing issues before they reach human reviewers. This capability has the potential to reduce operational inefficiencies and streamline compliance processes significantly.

Our findings show that DocuBot maintains high precision across various configurations, ensuring that human auditors are not overwhelmed with false positives. Additionally, its recall can be further optimized through targeted improvements, such as better question templates and more advanced models, which enable incremental adoption without disruptive process changes.

Future work will focus on enhancing DocuBot's automating data extraction tasks and reasoning capabilities and combining detection with generation to create a more robust, self-improving system. With continued refinement, we believe systems like DocuBot can become indispensable tools for reducing the costs and complexities of workflows that require human reviews.

REFERENCES

[1] Sakher Khalil Alqaaidi, Elika Bozorgi, Afsaneh Shams, and Krzysztof J. Kochut. 2024. A Few-Shot Learning-Focused Survey on Recent Named Entity Recognition and Relation Classification Models. In *DATA*. SCITEPRESS, 450–459.

[2] S. Arbesman. 2012. *The Half-Life of Facts: Why Everything We Know Has an Expiration Date.* Penguin Publishing Group. https://books.google.cg/books?id=nXho71uvKBsC

[3] Mihir Bellare, Phillip Rogaway, and Terence Spies. 2010. *Addendum to "The FFX Mode of Operation for Format-Preserving Encryption".* Technical Report. National Institute of Standards and Technology. https://csrc.nist.gov/CSRC/media/Projects/Block-Cipher-Techniques/documents/BCM/Proposed-Modes/ffx/ffx-addendum.pdf

[4] Mihir Bellare, Phillip Rogaway, and Terence Spies. 2010. *The FFX Mode of Operation for Format-Preserving Encryption.* Technical Report. National Institute of Standards and Technology. https://csrc.nist.gov/CSRC/media/Projects/Block-Cipher-Techniques/documents/BCM/Proposed-Modes/ffx/ffx-spec.pdf

[5] John Black and Philip Rogaway. 2002. Ciphers with Arbitrary Domains. In *Proceedings RSA-CT*. 114–130. https://www.cs.ucdavis.edu/~rogaway/papers/subset.pdf

[6] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. In *NeurIPS*.

[7] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. *CoRR* (2021).

[8] Shubhang Shekhar Dvivedi, Vyshnav Vijay, Sai Leela Rahul Pujari, Shoumik Lodh, and Dhruv Kumar. 2024. A Comparative Analysis of Large Language Models for Code Documentation Generation. In *AIware*. ACM.

[9] Kudra. [n. d.]. Kudra. https://kudra.ai/.

[10] Jiachang Liu, Dinghan Shen, Yizhe Zhang, Bill Dolan, Lawrence Carin, and Weizhu Chen. 2021. What Makes Good In-Context Examples for GPT-3? arXiv:2101.06804 [cs.CL] https://arxiv.org/abs/2101.06804

[11] Jonathan I. Maletic and Andrian Marcus. 2010. Data Cleansing: A Prelude to Knowledge Discovery. In *Data Mining and Knowledge Discovery Handbook, 2nd ed.* Springer, 19–32.

[12] Nanonets. [n. d.]. Nanonets. https://nanonets.com//.

[13] Erhard Rahm and Hong Hai Do. 2000. Data Cleaning: Problems and Current Approaches. *IEEE Data Eng. Bull.* 23, 4 (2000), 3–13.

[14] Thomas C. Redman. 1998. The Impact of Poor Data Quality on the Typical Enterprise. *Commun. ACM* 41, 2 (1998), 79–82.

[15] Arkaprabha Sau, Santanu Phadikar, and Ishita Bhakta. 2024. Efficient detection of data entry errors in large-scale public health surveys: an unsupervised machine learning approach. *Discover Public Health* 21, 1 (2024), 129.

[16] Qianhua Shan, David Sukhdeo, Qianying Huang, Seth Rogers, Lawrence Chen, Elise Paradis, Peter C. Rigby, and Nachiappan Nagappan. 2022. Using nudges to accelerate code reviews at scale *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 472–482. https://doi.org/10.1145/3540250.3549104

[17] Hongbin Sun, Zhanghui Kuang, Xiaoyu Yue, Chenhao Lin, and Wayne Zhang. 2021. Spatial Dual-Modality Graph Reasoning for Key Information Extraction. *CoRR* abs/2103.14470 (2021).

[18] Yanlin Wang, Tianyue Jiang, Mingwei Liu, Jiachi Chen, and Zibin Zheng. 2024. Beyond Functional Correctness: Investigating Coding Style Inconsistencies in Large Language Models. *CoRR* abs/2407.00456 (2024).

[19] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems* 35 (2022), 24824–24837.

[20] Vikas Yadav and Steven Bethard. 2018. A Survey on Recent Advances in Named Entity Recognition from Deep Learning models. In *COLING*. Association for Computational Linguistics, 2145–2158.

[21] Wenwen Yu, Ning Lu, Xianbiao Qi, Ping Gong, and Rong Xiao. 2020. PICK: Processing Key Information Extraction from Documents using Improved Graph Learning-Convolutional Networks. In *ICPR*. IEEE, 4363–4370.

[22] Qintong Zhang, Victor Shea-Jay Huang, Bin Wang, Junyuan Zhang, Zhengren Wang, Hao Liang, Shawn Wang, Matthieu Lin, Conghui He, and Wentao Zhang. 2024. Document Parsing Unveiled: Techniques, Challenges, and Prospects for Structured Information Extraction. *CoRR* abs/2410.21169 (2024).