Extraction Pipelines: Reliable Table and Text Extraction with OCR and Visual Context

Caleb Yohannes
Founding Engineer, Novoflow
caleb@novoflow.io, calebyhns@gmail.com

Abstract

Multimodal models that include VLMs (vision-language models) have shown some success on easily digestible input, however when more complex input is given, VLMs have shown to be unreliable. OCR (optical character recognition) offers a solution, through taking prompts and images and input, and turning pixels into machine-readable characters, however, reliability is not guaranteed outside a defined set of conditions. We propose a new way to leverage the highest reliability possible through a multi-stage pipeline engineered to ensure high-confidence results through creating an environment tailored to achieving the highest possible extraction accuracy: we call this an "extraction pipeline".

1. Introduction

Through the emergence of modern artificial intelligence in the last few years, the demand for visual understanding has grown exponentially: a trend that continues to grow across varying industries, however VLMs still struggle with producing reliable results on unfamiliar and out-of-distribution interfaces. With respect to healthcare, many EHR systems that control the software of healthcare providers have antiquated systems, and outdated user interfaces that many Large Language Models aren't familiar with. Hallucinations, that stem from the pretraining process of large-scale vision-language also can manifest across multiple semantic layers, including, but not limited to:

- Misinterpreting elements/object from images (object hallucinations)
- assigning incorrect properties to visible objects (attribute hallucinations)
- falsely describing spatial or semantic relationships (relational hallucinations). [1, 2]

In this work, we propose extraction pipelines that create an environment tailored to maximize success rates and minimize extraction failures. Through these extraction pipelines, we have tested them against the following use cases:

a) timeslot extraction
b) text extraction through calendar columns
c) text extraction through varying grids, and
d) text extraction through individual elements.

We have managed to achieve 99%+ reliability against all EHR (Electronic Health Record) Systems tested. This is a significant improvement from our previous approach with VLMs, which haven't been able to deliver more than 50% reliability through any of the use cases mentioned above.

2. Reliability Limits of OCR-based Extraction

Research on optical character recognition (OCR) has been extensive over the past decades, leading to high character-level accuracy under favorable conditions. However, reliability remains a key bottleneck under unfavorable conditions, particularly for real-world inputs involving degraded text and complex table structures.

2.1 Text Recognition Under Degraded Conditions

OCR is highly dependent on the quality of its input. When the input is severely degraded or partially missing, no model can reliably recover information that does not exist. In such cases, it is essential that the preprocessed data provided to the OCR system is as legible and high quality as possible. While improvements over the original input are inherently limited, they can be achieved through preprocessing techniques such as resolution normalization, cropping, and contrast enhancement. Semantic correction is not recommended within an extraction-pipeline approach, as it attempts to compensate for system errors rather than creating conditions that maximize the system's potential performance.

2.2 Structural Challenges in Table Extraction

In terms of table extraction, the challenge becomes far more unique. Unlike plain text, tables create structural and logical relationships through different elements through techniques such as alignment, grouping, and spatial structures. OCR systems are heavily trained on linearized text output, and do not have enough context nor training on understanding these relationships comfortably. This is especially true in inconsistent layouts, elements that are almost indistinguishable from one another, tables with barely visible or no borders, etc.

Earlier solutions to combat these bottlenecks have seen promising results, however still came with multiple drawbacks.

One of the earlier table extraction systems was Tabfinder (Cesarini et al.), a layout-driven approach that relied heavily on explicit layout structure through ruling lines to infer table hierarchy. Tables were detected by identifying horizontal and vertical ruling lines that satisfied predefined constraints. An MXY tree was then constructed to represent the spatial hierarchy of the table, recursively partitioning the input into row regions along the Y-axis and column regions along the X-axis. A depth-first traversal of this tree was used to recover individual cells and reconstruct the table structure. [3]

Another approach to table extraction is table detection using Hidden Markov Models (HMMs), as proposed by Silva et al. In this method, table structure is formulated as a labeling problem in which visual elements are processed line by line, and an HMM is used to estimate the probability that each element belongs to a table. This probabilistic formulation reduces reliance on rigid geometric constraints and explicitly models uncertainty. [4]

Despite these advantages, both approaches remain highly constrained. In legacy OCR systems, Tabfinder's structure-driven method is prone to failure in the presence of borderless tables, unconventional layouts, or significant visual noise. Similarly, the HMM-based approach suffers from strong dependence on the quality and distribution of its training data and struggles to represent column relationships that require modeling cross-row alignment.

2.3 Implications for Reliability

These limitations demonstrate that OCR-based extraction is far more than a simple intrinsic limitation, but rather a limitation on many external factors, such as context, input quality, and structure. While models continue to improve, and error rates continue to drop, the broader challenges of creating an environment for OCR to flourish, through spatial context, structural analysis and preprocessing. In practice, factors lead to extraction systems that appear reliable at first glance, but exhibit growing unpredictability due to increasing diversity in input data and rising complexity of the content to be interpreted.

These drawbacks demonstrate the necessity for extraction pipelines that treat OCR output as ground truth, while curating an environment created through iterations of better structural context, predictable components, and configurations built through suites of validation mechanisms, to ensure the most reliable results achievable.

3. Design Goals and Constraints

The objective of extraction pipelines is to achieve the highest possible results of table and text extraction through the control of 3 key dimensions:

- Algorithmic Programming
- Prompting
- OCR Models

The success of these models is defined not only by accuracy, but also reliability, interpretability, and deployability.

3.1 Reliability Across Different EHR Formats

Formats of EHR systems vary significantly across clinics. Each visual element can be interpreted differently by OCR, resulting in outputs of varying quality.

One of the largest constraints with OCR is its sensitivity towards structure. OCR's strength in interpreting structure lies in its ability to recognize colors, unique elements, and consistent formatting and alignment. The degree to which these visual signals are present also significantly influences OCR output quality.

Below is an example of an EHR format that OCR struggles to interpret.

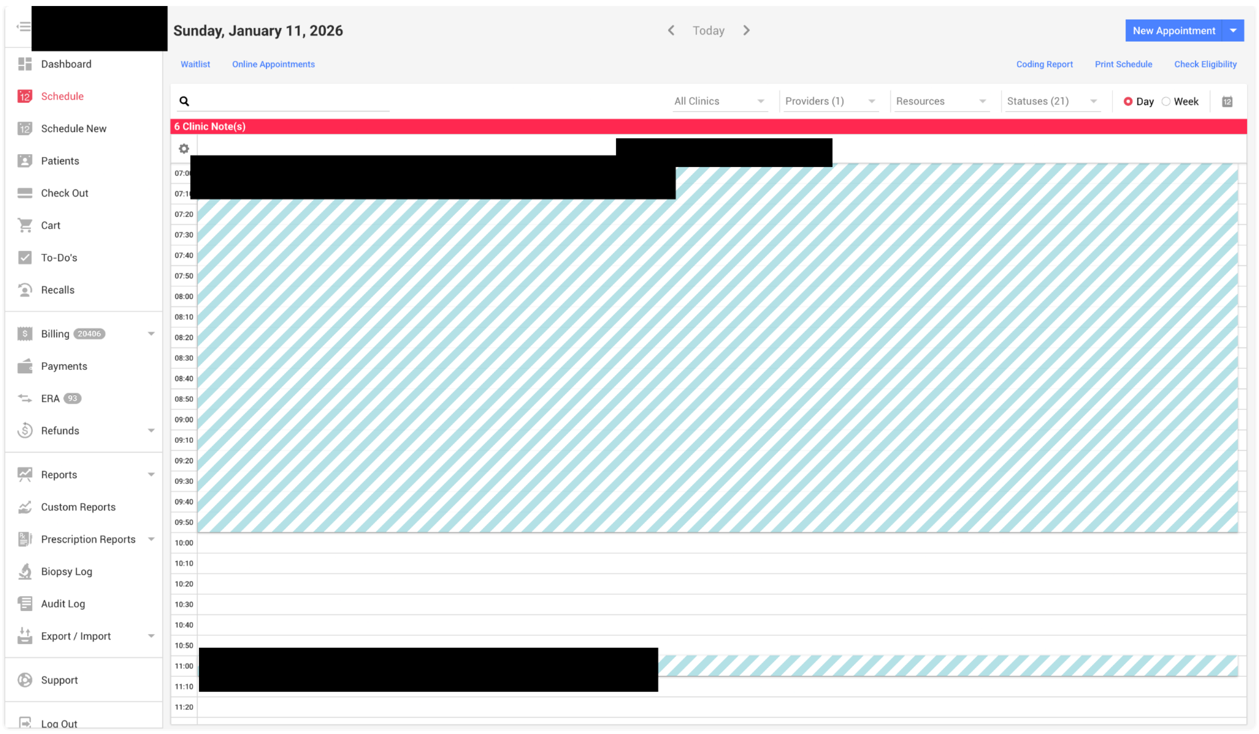
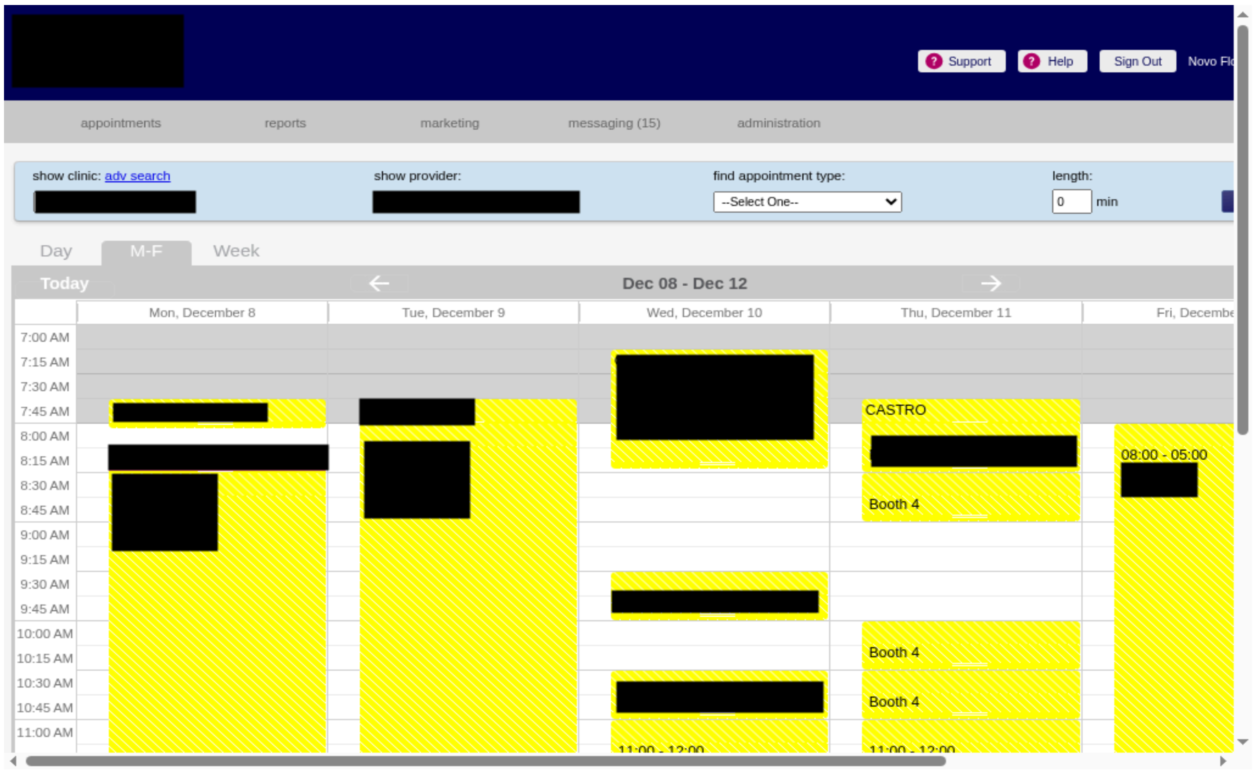


Figure 1: Screenshot of EHR system that is hard to understand via pure OCR. Note: Sensitive

```
information has been redacted out to maintain anonymity of our clients and their patients.
This will be a continuing occurrence.
```

Here is an example of an EHR system that OCR interprets well.



```
Figure 2: Screenshot of EHR system that is easy to understand via pure OCR.
```

In the first photo, the colors are much less distinguishable. Timeslots are also only half covered in light blue, with diagonal stripes across the slot, which makes it hard to interpret, as OCR treats color as high availability.

There is some structure to the calendar region; there aren't any distinguishable shapes surrounding the exterior, such as lines or rectangular borders. The text on the screenshot is also quite small, making it more challenging for OCR to reliably extract it.

For the second screenshot, colors are much more visually distinguishable. Timeslots are marked in vibrant yellow, covering the entire area for which the slot is filled. The yellow color is also much more distinguishable from other elements, making it much easier for OCR to interpret.

While there isn't much structure on the sides of the calendar, it covers the entire screen, making it very visible to OCR. The top part of the calendar also has a gray rectangular shape, with more legible text and elements.

3.2 Minimal Dependence on Model Training

Model training, especially in this domain, is very undesirable. There is too little publicly available data on EHR systems to train on, and the data varies greatly from the system you are working with. Retraining should always be considered a last resort and used only when no other viable options remain.

As the advancement of artificial intelligence continues, the need for adaptation to new types of systems, documents, and formats will only increase. The extraction pipeline is built to adapt without requiring retraining across system types. This is done through custom-made logic, configurations, and constraints rather than model updates, reducing development time to mere hours.

Using this system, extraction results are idempotent: they return the same results when given the same inputs. Debugging, visual representation, and validation testing are also heavily used alongside the extraction pipeline to ensure idempotency and the highest possible accuracy. It is also important to observe system behaviour and results as changes to the inputs or pipelines are made, as this can affect results.

4. System Overview

Here is a diagram of our extraction pipeline:

```
Input: Screenshot + ehr_type
        ↓
┌─────────────────────────────────────────────────────────────┐
│  Phase 1: Router                                            │
│  Routes to correct pipeline based on ehr_type               │
└──────────────────────────────┬──────────────────────────────┘
        ↓
┌─────────────────────────────────────────────────────────────┐
│  Phase 2: Detect Table                                      │
│  Find boundaries of the table area using Moondream3         │
└──────────────────────────────┬──────────────────────────────┘
        ↓
┌─────────────────────────────────────────────────────────────┐
│  Phase 3: Reconstruct Table                                 │
│  OCR permanent data to create non-overlapping column regions │
└──────────────────────────────┬──────────────────────────────┘
        ↓
```

`Output: JSON with clickable coordinates`

It is important to note that at each step in each phase of the extraction pipeline, we have implemented a variety of debugging tools and visualization tests to understand what OCR is returning and why. OCR is very sensitive to a wide range of elements, formats, and structures in the input, and can be unpredictable at times.

A key factor in improving our approach was the use of labeled bounding boxes to visualize OCR detections and better understand model behavior. This enables direct inspection of OCR detections, substantially simplifying the iterative development and refinement process.

4.1 Pipeline Routing

Phase 1 is where the EHR gets routed to the proper extraction pipeline. Every pipeline built is tailored not only to a specific EHR system but also to a specific format within that system. EHRs can have varying formats within the same system, such as one format for a waitlist table and another one for a calendar table.

All our agents live on their own projects, and within these projects, we specify which pipeline to use in their .json files via ehr_type. The output is then processed by the application, which contains the extraction pipelines, and routed to the appropriate pipeline. We call this application "service platform".

```
MOCK_EHR_CONFIGS = {
    "doctorpro_timeslots": MockDoctorProConfig,
    "medischedule_waitlist": MockMediScheduleConfig,
    "healthbook_outbound": MockHealthBookConfig,
    "healthbook_outbound_conversational": MockHealthBookConfig,
    "clinicflow": MockClinicFlowConfig,
    "carescheduler": MockCareSchedulerConfig,
    "patienthub": MockPatientHubConfig,
}
```

All EHRs are first mapped to their respective configuration files and associated with an extraction pipeline. The EHR type is then defined from the main application. All calls made to the service platform by projects pass configurations and are read to identify any associated extraction pipelines.

```
"Configuration": {
    "ehrType": "cid",
# Other configurations
}
```

4.2 Table Area Detection

The first part of allowing OCR to understand the contents of the input is to establish the region you are trying to extract data from. This can be done in many ways, but the recommended approach is to first try to identify the region via prompting.

Some systems are very easy to identify and can be identified with a simple prompt. Here is an example of a prompt that we used for the EHR system below:

```
@property
def moondream_prompt(self) -> str:
    return "only grid calendar table"
```

Other EHR systems are not as simple. Some require a far more detailed prompt, and some can't be identified with only a prompt. The screenshot of the EHR system shown in Figure 1 is an example of a calendar table that could not be identified solely by prompting and required pre-processing steps, including cropping the original photo to show only the calendar region.

```
# Extract date from top center-left area before cropping
date_region_x1 = int(width * 0.21)  # Start at 21% from left
date_region_x2 = int(width * 0.50)  # End at 50% from left
date_region_y1 = 0                  # Very top
date_region_y2 = height // 10       # Top 10% of image

date_region = image.crop((date_region_x1, date_region_y1,
date_region_x2, date_region_y2))
```

4.3 Reconstruct Table

Table reconstruction is the most important part of the extraction pipeline, as it recreates the table structure in a format that OCR understands. Data from tables cannot be reliably extracted via OCR unless the table structure is understood. In

EHR systems, this can be achieved by leveraging persistent data fields to define grid regions and establish expected data patterns.

Practically almost all EHR systems follow the same calendar format: date headers horizontally across the top of the calendar, and time headers across the leftmost column. The detected cells for these headers are then converted into cell objects.

```
def _calculate_cell_colors(self, cropped_image: Image.Image,
child_cells: List[List[Cell]]) -> None:

# Convert detected cells to Cell objects
time_cells = [Cell(cell["coordinate"]) for cell in
header_column_cells]  # Left column (times)
date_cells = [Cell(cell["coordinate"]) for cell in header_row_cells]
# Top row (dates)

ROWS = len(time_cells)  # Number of time slots
COLS = len(date_cells)  # Number of date column
```

From here, individual rows within the calendar must be identified. Lines between rows can be quite tricky for OCR to read, and it is common to see bleeding between lines or inconsistent row sizing. To combat this, we have a variety of methods to ensure the highest possible accuracy.

The most reliable method we have seen is to use permanent data, such as date and time headers, as anchors to create bounding boxes between rows. OCR's strongest ability is to extract text and its position. We specifically use the separation between time headers and date columns to reconstruct each cell within the grid.

```
time_cells = [
    {"coordinate": [10, 100, 80, 130]},  # 8:00 AM detected at Y=100
    {"coordinate": [10, 147, 80, 177]},  # 8:15 AM detected at Y=147
    {"coordinate": [10, 189, 80, 219]},  # 8:30 AM detected at Y=189
]
```

The spacing between time headers isn't always consistent; however, alignment between cells and their corresponding time headers is. Using this, we can confidently determine where each cell will be.

```
# Use ACTUAL detected positions (not calculated spacing)
```

```
for i in range(ROWS):
    row_top = time_cells[i].bbox[1]      # Y=100 (actual detected position)
    row_bottom = time_cells[i+1].bbox[1]  # Y=147 (next actual position)

    # This row spans Y=100 to Y=147 (47px tall)
```

The main flaw with this method is that it struggles to establish bounding boxes between date columns and time headers when timeslots are filled with appointments. This is where we recommend synthetically inserting cells using the average spacing calculated from all coordinates. This is computed by summing the pairwise differences between consecutive coordinates and dividing by the total number of coordinates minus one.

Synthetic cell insertion can still cause bleeding issues between empty cell boundaries, so as a third strategy, we recommend percentage-based empty cell detection. We detect the RGB colors of all cells, and if they meet the minimum required color threshold, we consider it an empty slot.

Note: all text within triple quotation marks in functions are prompts.

```
def is_cell_empty(self, r, g, b, white_percentage):
    """
    Check if cell is empty using pixel percentage, not just average color.
    """
    # Method 1: Percentage-based (handles bleeding)
    if white_percentage >= 70.0:
        return True  # ≥70% white pixels = empty
```

Here, we set the minimum white percentage for each cell to 70%. If it meets this minimum, it passes the test and returns an empty white cell.

For certain EHRs, empty cells might not be perfectly white, but rather light gray, or another color. For this, we recommend calculating the average color.

```
    # Method 2: Average color (fallback)
    if r > 240 and g > 240 and b > 240:
        return True  # Pure white average

    return False
```

Here, 240 represents light gray. We calculate the average percentage of each color for each character in the RGB color space, and if it exceeds light gray, we identify it as an empty cell.

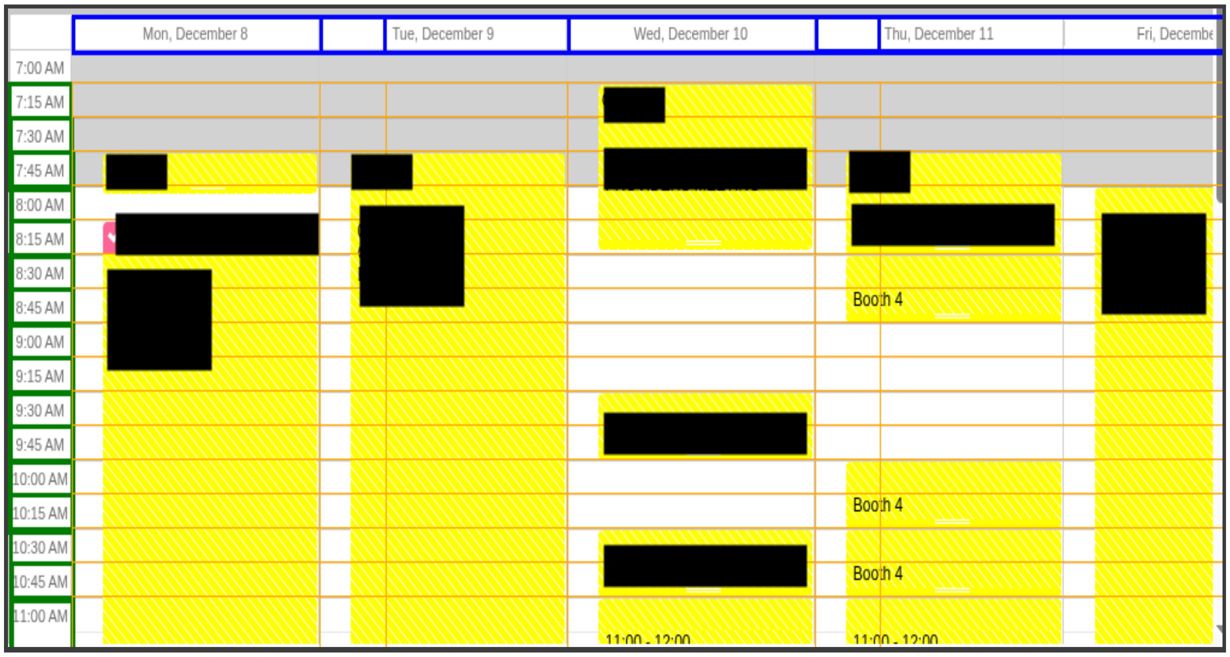
After implementing all these strategies, we have successfully created accurate bounding boxes and accurately detected all empty cells:

Pipeline type: Standard Empty Slots

Empty slots detected: 11

Grid size: {'rows': 16, 'cols': 4}

Mon, December 8 at 8am, Wed, December 10 at 8:30am , Wed, December 10 at 8:45am, Wed, December 10 at 9am, Thu, December 11 at 9am, Wed, December 10 at 9:15am, Thu, December 11 at 9:15am, Thu, December 11 at 9:30am, Thu, December 11 at 9:45am, Wed, December 10 at 10am



Now that our input has gone through our entire extraction pipeline, we configure the pipeline to return a variety of data based on our use case, such as validating the existence of requested data, available timeslots, and the x and y coordinates of certain elements for grounding models to perform clicking and typing actions.

5. Conditioning OCR With Visual Context

With regard to extraction pipelines, OCR is not treated as ground truth, but rather a foundation upon which further methods can be built. Through visual and spatial cues, constraints, and configurations are established and validated to achieve the highest possible percentage of reliable outputs.

We group all these cues in 3 categories: color, positional, and structural. Color cues include any distinguishable color visually represented on the input. Positional refers to the position in which specific parts of structures and elements are. Structural refers to the form and distinguishable features of objects in the inputs.

It is important to note that the highest reliability is achieved through these cues being used in conjunction with one another. OCR can achieve high reliability, but its outputs can be inconsistent at times.

5.1 Conditioning Mechanisms

With regard to color, we have discussed previous RGB methods for properly extracting color, such as the percentage-based and average color methods; however, it is important to note that many EHR systems don't always display the most distinguishable colors. There are many factors that affect OCR's understanding of color, such as how much of the color is present, how different the color is from its surrounding areas, and the color's shades. The largest bottleneck in color is the amount of color present. OCR returns somewhat accurate percentages of color within a defined area; these outputs can be inconsistent and cause pipeline failures if exactness is required.

Position has been found to be the easiest of the 3 cues to understand by OCR, mainly because it is very easy to define where in the input elements it occurs. This usually gets tricky when the item you are trying to extract via OCR is not easily distinguishable or cannot be easily distinguished from the prompt. Some ways to combat these bottlenecks include improving the prompting as much as possible and describing distinguishable features surrounding the item, along with their positions relative to the item. If the extraction pipeline is set up to understand coordinates given by grounding models, and the item for which you are trying to extract with OCR does not move positions, you can also pass in coordinates to the extraction pipeline, so it automatically will extract whatever is within these coordinates. Implementation of this is discussed in Section 7.

Structure is the hardest cue for OCR to understand, as it has not been heavily trained to recognize it, nor is it strong at interpreting non-color or non-text items. Sections 4.2 and 4.3 discuss solutions to improving OCR's structural understanding. It is important to note that you should also combine multiple cues with a structured approach to ensure the highest reliability.

5.2 Adaptation Across Different Systems

When adapting across different systems, a foundational understanding of the layout must be passed to the extraction pipeline to ensure the highest reliability. Some questions that need to be answered include the following:

- Which features of the system must I provide for OCR to understand?
- What can it already understand with little to no adjustments?
- For features that can't be easily understood with OCR, what distinguishable features does it have? How do I set this up in OCR?

Prompting should always be the first option for improving OCR understanding, as it offers the greatest control and flexibility to improve reliability.

Most systems, especially modern ones, do not require manual tweaking; however, if tweaking must be done, understand that certain constraints will be imposed as part of the process. This is especially evident in structural understanding, as OCR is very sensitive to the position in which the structure has been found. Inputs with the same structure in different positions are likely to produce very different results.

5.3 Adaptation Hierarchy

Adaptations across different extraction pipelines follow a 3-tier hierarchy:

- Tier 1 - Zero-touch adaptation
- Tier 2 — Light Configuration
- Tier 3 — Targeted Constraints

The idea of the hierarchy is to develop the extraction pipelines in order: first attempting to adapt the pipeline only through prompting, then through minor configuration/code changes, and finally use-case–specific configurations defined through strict constraints and system context, validated across multiple cues using extensive test cases.

The first tier of adaptation relies only on prompting and model reasoning. There are little to no code changes, no configuration changes, or system-specific rules. This tier enables rapid development of extraction pipelines and generalizes across systems with similar OCR readability and extraction logic, making it the most scalable and easily maintainable tier.

When prompting alone is not enough, custom configurations are needed. This includes minor configuration changes and validating reliability across a few general files. Structure usually does not need to be adapted.

Only when general mechanisms fail are targeted constraints introduced. This corresponds to domain-specific structure encoding, including leveraging the identification of distinguishable visual features, imposing tight constraints on multiple cues, adapting structural cues, and using multiple custom-built test suites to ensure reliability.

6. Application Integration via Tool-based Integration

6.1. Extraction as a Tool Call

The extraction pipeline process via tool calls starts with defining and calling the tool from the system prompt to be used during the agent's process.

```
<tools>
- extract_empty_timeslots: Extract empty time slots from the current
</tools>

**Execution Loop:**
1. Call `extract_empty_timeslots`
```

All extraction pipelines are configured to be invoked at any stage of the AI agent's task execution, through tool calls defined in the agent's project configuration file. All tool calls are also mapped to a file that redirects them to their corresponding functions.

```
Project file

       {
          "name": "extract_empty_timeslots",
          "description": "Extract empty time slots from the current
calendar position",
          "category": "browser"
        },
```

Mapped tools file

```
'extract_empty_timeslots': ServiceTools.extractEmptyTimeslots,
```

The corresponding function then gets run, which calls the service platform with the screenshot taken at the time the tool was invoked.

```
static async extractEmptyTimeslots(params: any): Promise<OperationResult> {
  const { caller_id } = params;

  const workingHours = session.projectConfig?.officeWorkingHours;

  const screenshot = await session.page.screenshot({ type: 'png' });
  const base64Screenshot = screenshot.toString('base64');

  const ehrType = session.projectConfig?.ehrType || 'cid';

  const extractionResult = await processTimeslotScreenshot(
    base64Screenshot,
    ehrType
  );

  const processedTimeslots: Record<string, string[]> = {};
  const processedDates: string[] = [];
  let totalSlots = 0;

  for (const [date, extractedSlots] of
Object.entries(extractionResult.timeslotsByDate)) {
    processedDates.push(date);

    const validatedSlots = validateTimeslots(extractedSlots);
    processedTimeslots[date] = validatedSlots;
    totalSlots += validatedSlots.length;
  }

  const datesSummary =
    processedDates.length === 1
      ? processedDates[0]
      : `${processedDates.length} dates (${processedDates.join(', ')})`;
  return {
    success: true,
    data: {
      message: `Extracted ${totalSlots} time slots across ${datesSummary}.
${extractionResult.message || ''}`,
      dates: processedDates,
      totalSlots,
      timeslotsByDate: processedTimeslots,
    },
  };
}
```

The corresponding function then gets run, which calls the service platform with the screenshot taken at the time the tool was invoked.

Multiple processes are running concurrently while this function is being called. First, we extract the .png screenshot taken by Playwright and convert it to base64 format for ingestion by OCR. We also pass the ehrType from the project's configuration to ensure it is routed to the proper extraction pipeline. We also construct variables to validate and parse the returned data.

The processTimeslotScreenshot() function is what takes the raw data served by the extraction pipeline and filters it by consecutive 30-minute timeslots, as structured in the scheduler:

```
export async function processTimeslotScreenshot(
  screenshot: string,
  workingHours?: OfficeWorkingHours,
  accessToken?: string,
  ehrType?: string
): Promise<TimeslotExtractionResult> {
  const serviceUrl = process.env.NEXT_PUBLIC_SERVICE_PLATFORM_URL;

  const response = await callEmptySlotsEndpoint(
    screenshot,
    ehrType
  );

  const consecutive30MinSlots = filterConsecutive30MinSlots(
    response.empty_slots,
    ehrType
  );
  const timeslotsByDate: Record<string, string[]> = {};

  for (const slot of consecutive30MinSlots) {
    const normalizedTime = normalizeTimeFormat(slot.time);

    if (!timeslotsByDate[slot.day]) {
      timeslotsByDate[slot.day] = [];
    }

    timeslotsByDate[slot.day].push(normalizedTime);
  }

  return {
    success: true,
    timeslotsByDate,
    message: '',
  };
```

Note: The code snippets shown above are simplified for clarity. The actual implementation includes additional parameters, such as accessToken for service-platform authentication, workingHours for filtering slots outside office hours, and error-handling logic.

At this point, the data is passed to extractEmptyTimeslots(), added to the agent's context for understanding, and the process continues.

6.2 Benefits of Tool-Based Integration

Allowing for tool-based integration has multiple advantages. First, it enables maximum observability throughout the extraction pipeline. Error handling attached to each component also allows failures to be localized, identified, and isolated, making them easier to debug and iterate on.

In addition, component isolation enables independent iteration and development without introducing system-wide regressions. Clean separation between application and extraction logic follows best coding practices, improving maintainability and long-term extensibility.

7. Continuous Iteration

Due to the vast number of edge cases in our specific use cases, it is far too time-consuming to find, analyze, and test all possible edge cases for varying inputs. The input's scroll state, changes in content during updates, and resolution are all components of the input that could affect the extraction pipeline's results.

Through our observability systems, we have been able to track all actions, including errors, when calling our extraction pipelines. Rather than only adjusting configurations in response to new document types, the system should be designed so that failures actively improve future performance. This includes adding more edge cases, creating new filters to remove data from the extraction process, and feeding back errors into configurations.

Through the iteration process, the goal should be to optimize, rather than lengthen. The adaptation hierarchy tiers are used not only to describe the state of the extraction pipeline but also to indicate the preferred level of adaptation maturity. Higher tiers are always better, as manual configurations rely less on OCR and increase the pipeline's sensitivity to the input.

8. Limitations and Tradeoffs

While we have recieved 99% reliability across all extraction pipelines, this high percentage cannot be achieved without certain tradeoffs. Firstly, while extraction pipelines significantly increase the reliability of results from OCR, they are not 100% reliable. Proper error handling and fallback mechanisms should be put in place should the extraction pipeline fail.

Extraction pipelines also take more time to process than VLMs. While the increase in time is small (a mere few seconds more per call), this can be critical depending on the use case. The goal for the extraction pipelines is to achieve high latency, not to reduce processing time.

Certain systems that are very complex and hard to understand, even with prompting, will require significant manual configuration. High levels of manual configuration can also pose problems in the future as more configurations and edge cases are added. This makes the extraction pipeline highly sensitive to input, reducing flexibility and adaptability and increasing the difficulty and time required for further development.

9. Conclusion

In this work, I presented the extraction pipeline, a proven tool that significantly improves reliability for table and text extraction with OCR using visual context.

By creating a new environment tailored for OCR to be understood and evaluated, the accuracy of data can be significantly improved compared to traditional OCR models, allowing for production-level table and text extraction across vast numbers of systems.

We continue to monitor the improvement of VLMs and OCR models to evolve extraction pipelines, enabling them to be extended further and applied to an increasing number of specific use cases.

To view code examples of how we run these pipelines on applications, click here.

References

[1] A. Rohrbach, L. A. Hendricks, K. Burns, T. Darrell, and K. Saenko, "Object hallucination in image captioning," in *Proceedings of EMNLP*, E. Riloff, D. Chiang, J. Hockenmaier, and J. Tsujii, Eds. Association for Computational Linguistics, 2018, pp. 4035–4045. https://doi.org/10.18653/v1/d18-1437

[2] J. Wang, Y. Wang, G. Xu, J. Zhang, Y. Gu, H. Jia, J. Wang, H. Xu, M. Yan, J. Zhang, and J. Sang, "Amber: An llm-free multi-dimensional benchmark for llms hallucination evaluation," 2024. https://arxiv.org/abs/2311.07397

[3] F. Cesarini, S. Marinai, L. Sarti, and G. Soda, "Trainable Table Location in Document Images," in 16th International Conference on Pattern Recognition, ICPR 2002. IEEE Computer Society, 2002, pp. 236–240.

[4] A. C. e Silva, "Learning Rich Hidden Markov Models in Document Analysis: Table Location," in ICDAR 2009: 10th International Conference on Document Analysis and Recognition. IEEE Computer Society, 2009, pp. 843–847.