Blueprint: A Constraint-solving Approach For Document Extraction

Andrey Mishchenko
University of Michigan
mishchea@umich.edu

Dominique Danco
University of Amsterdam
dominique.danco@student.uva.nl

Abhilash Jindal
IIT Delhi
ajindal@cse.iitd.ac.in

Adrian Blue
Instabase
adrian.blue@instabase.com

ABSTRACT

Blueprint is a declarative domain-specific language for document extraction. Users describe document layout using spatial, textual, semantic, and numerical fuzzy constraints, and the language runtime extracts the field-value mappings that best satisfy the constraints in a given document.

We used Blueprint to develop several document extraction solutions in a commercial setting. This approach to the extraction problem proved powerful. Concise Blueprint programs were able to generate good accuracy on a broad set of use cases. However, a major goal of our work was to build a system that non-experts, and in particular non-engineers, could use effectively, and we found that writing declarative fuzzy constraint-based extraction programs was not intuitive for many users: a large up-front learning investment was required to be effective, and debugging was often challenging.

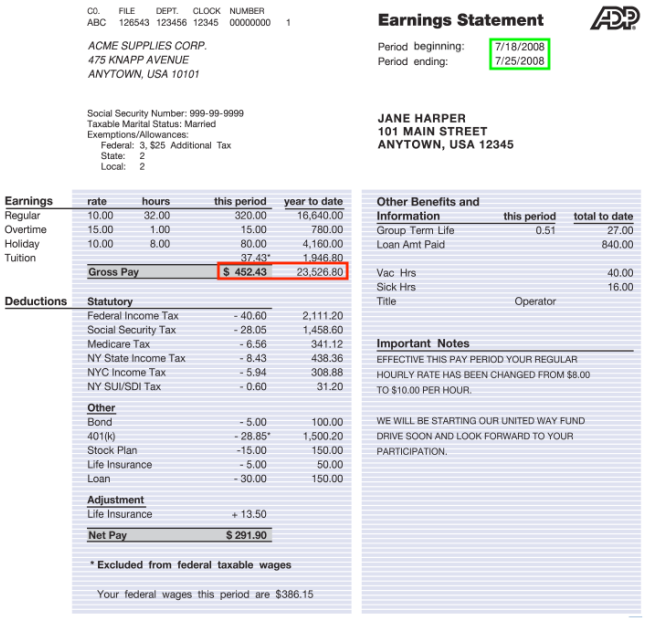
To address these issues, we developed a no-code IDE for Blueprint, called Studio, as well as program synthesis functionality for automatically generating Blueprint programs from training data, which could be created by labeling document samples in our IDE. Overall, the IDE significantly improved the Blueprint development experience and the results users were able to achieve.

In this paper, we discuss the design, implementation, and deployment of Blueprint and Studio. We compare our system with a state-of-the-art deep-learning based extraction tool and show that our system can achieve comparable accuracy results, with comparable development time, for appropriately-chosen use cases, while providing better interpretability and debuggability.

PVLDB Reference Format:
Andrey Mishchenko, Dominique Danco, Abhilash Jindal, and Adrian Blue. Blueprint: A Constraint-solving Approach For Document Extraction. PVLDB, 15(12): 3459 - 3471, 2022.
doi:10.14778/3554821.3554836

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/instabase/blueprint-oss.

Proceedings of the VLDB Endowment, Vol. 15, No. 12 ISSN 2150-8097.
doi:10.14778/3554821.3554836



CO. FILE DEPT. CLOCK NUMBER
ABC 126543 123456 12345 00000000 1

ACME SUPPLIES CORP.
475 KNAPP AVENUE
ANYTOWN, USA 10101

Social Security Number: 999-99-9999
Taxable Marital Status: Married
Exemptions/Allowances:
Federal: 3, $25 Additional Tax
State: 2
Local: 2

Earnings Statement

Period beginning: 7/18/2008
Period ending: 7/25/2008

JANE HARPER
101 MAIN STREET
ANYTOWN, USA 12345

Earnings	rate	hours	this period	year to date
Regular	10.00	32.00	320.00	16,640.00
Overtime	15.00	1.00	15.00	780.00
Holiday	10.00	8.00	80.00	4,160.00
Tuition			37.43*	1,946.80
Gross Pay			**$ 452.43**	23,526.80
Deductions	**Statutory**			
	Federal Income Tax		- 40.60	2,111.20
	Social Security Tax		- 28.05	1,458.60
	Medicare Tax		- 6.56	341.12
	NY State Income Tax		- 8.43	438.36
	NYC Income Tax		- 5.94	308.88
	NY SUI/SDI Tax		- 0.60	31.20
	Other			
	Bond		- 5.00	100.00
	401(k)		- 28.85*	1,500.20
	Stock Plan		-15.00	150.00
	Life Insurance		- 5.00	50.00
	Loan		- 30.00	150.00
	Adjustment			
	Life Insurance		+ 13.50	
	Net Pay		**$ 291.90**	

Other Benefits and Information	this period	total to date
Group Term Life	0.51	27.00
Loan Amt Paid		840.00
Vac. Hrs		40.00
Sick Hrs		16.00
Title	Operator	

Important Notes

EFFECTIVE THIS PAY PERIOD YOUR REGULAR HOURLY RATE HAS BEEN CHANGED FROM $8.00 TO $10.00 PER HOUR.

WE WILL BE STARTING OUR UNITED WAY FUND DRIVE SOON AND LOOK FORWARD TO YOUR PARTICIPATION.

* Excluded from federal taxable wages

Your federal wages this period are $386.15

Figure 1: A sample paystub.

1 INTRODUCTION

Document extraction is the process of retrieving data from documents, including the data's semantics. For us, a *document* is a digital version (PDF, DOCX, TIFF, etc.) of what would traditionally be a normal paper document (a driver's license, a tax form, etc.). Documents may be digitized through scanning, or may be created directly in digital form, e.g., by typing a Word document. For example, given the paystub shown in Figure 1, a bank may wish to extract the gross salary data (highlighted in red) and the pay period begin and end dates (highlighted in green), in the context of processing a loan application.

As businesses move to automate more of their operations, the ingestion of unstructured or semi-structured documents often presents as a major bottleneck, with manual data entry steps performed by humans at high operational cost still common in many critical business processes. In addition, many businesses have valuable data trapped in large stores of documents – financial reports, expenditure receipts, contracts, etc. – for which full manual extraction would be cost-prohibitive.

Document extraction today typically begins with optical character recognition (OCR), which over the last few years has become

commoditized and widely-used in industry. OCR models ingest documents in image form, and return the documents' text fragments in string form, along with bounding rectangles identifying every fragment's origin in the document. After OCR, it is generally necessary to use the document's structure – spatial, textual, semantic, or numerical – to extract the document's full semantics. In this paper, we describe Blueprint, a declarative domain-specific language that works on OCR output and produces extractions.

In the course of this work, a primary focus area for us was building tools for document extraction that non-experts, and ideally non-engineers, could use effectively, and picking abstractions that would enable us to build those tools.

1.1 Outline

We begin by introducing our design principles in Section 2. Our design is informed by challenges we experienced writing extraction programs in a general purpose programming language (GPPL), which we also discuss in some detail.

In Section 3, we compare our approach to related work.

In Section 4, we describe the Blueprint language. The majority of a Blueprint program consists of declarations of *constraints*, essentially statements of fact about a document's spatial layout, logical or numerical relationships among the document's parts, and so on. Blueprint provides a library of constraints which can be used to describe basic document structure (e.g., `left_aligned`, `greater_than`, etc.). These may be combined with logical connectives to express more complex constraints. Blueprint also provides good extensibility: for example, it is easy for users to define custom constraints to be used as part of the language.

In Section 5, we sketch the Blueprint runtime implementation.

In Section 6, we discuss some challenges users encountered while working with Blueprint as a programming language, and steps we took to resolve these challenges. In particular, we built a no-code graphical IDE called Studio, which includes program synthesis functionality and intuitive debugging capabilities.

In Section 7, we provide some user feedback, and discuss lessons we learned developing and deploying Blueprint and Studio.

In Section 8, we evaluate Blueprint using the MIDV-2020 dataset [6]. We first show that the performance and accuracy of automatically-synthesized Blueprint programs is similar to that of hand-written Blueprint programs. Next, we show that on this dataset, Blueprint is able to achieve accuracy results comparable to a state-of-the-art deep-learning-based document extraction system, LayoutLM [34], while remaining interpretable and debuggable.

We discuss future work in Section 9 and conclude in Section 10.

The contributions of this paper are summarized as follows:

- We describe Blueprint, a novel constraint-based document extraction language and runtime.
- We describe Studio, a no-code IDE for synthesizing, editing, testing, and debugging Blueprint programs.
- The source code for Blueprint and Studio is available at [1].

2 BACKGROUND AND MOTIVATION

This work began while the authors were at a company building document extraction tools and solutions for customers. Initially, our solutions were implemented as heuristics-based programs written using a general-purpose programming language (GPPL). This approach was adequate for some use cases, but presented many challenges. We begin this section by criticizing the GPPL approach, and proceed to outline the Blueprint design principles that came out of this experience.

2.1 Procedural Extraction Logic Using a GPPL

We begin by defining two key terms, *field* and *extraction*. In document processing, a *field* is the name of a piece of data that the user wishes to extract. For example, when working with passports, some examples of fields might be `'issue_date'` and `'last_name'`. An *extraction* for some sample document is a field-to-value mapping. For example, an extraction for a passport might look like `{'issue_date': "Jan 2, 2003", 'last_name': "Smith", ...}`.

A *document class* is a collection of documents having some shared characteristic. Examples of document classes are "US passports", "global identity documents" (which includes US passports), and "restaurant receipts". The goal of an extraction program is to produce extractions for some document class. For example, a passport extraction program should accept an image of a passport as input, and return an extraction. For us, the extraction process always starts with OCR, so all the extraction programs we discuss accept OCR output – words and their bounding boxes – as input.

Extraction programs written in GPPLs, even for relatively simple document types, can be surprisingly complex. This is because they require a lot of branching logic, both low-level (e.g., regex tolerances) and high-level (e.g., choosing between two distinct layouts for part of a document, or combining layouts), and fallback error handling. The level of complexity can be such that writing production-quality extraction programs in an industrial setting using a GPPL requires relatively highly-skilled and expensive software engineers. We walk through some specific challenges.

Field extraction order. Extractions generally involve multiple interrelated fields in a document. Often there are multiple reasonable-looking choices of value for each of the fields individually, and in order to get the right overall extraction we need to leverage the interrelationships among the fields. The complete search space is combinatorial in size and too big to search exhaustively in a reasonable amount of time.

Therefore, GPPL extraction programs generally build up an extraction incrementally, adding one field at a time and backtracking when they encounter an error condition. The order in which fields are extracted is important for extraction quality and runtime [30, 31], and in some cases there is no natural or obvious order in which to proceed. Expressing this multiple-options-and-backtracking logic in GPPL code becomes quite verbose and tedious. If developers do not design their programs from the beginning to gracefully support error fall-back between distant parts of their programs, it can be difficult to add this capability later. We discuss this further in Section 5.2.

Rank-ordering extractions by quality. It often becomes important to have a way of comparing the quality of two candidate extractions. This comes up especially when building an extraction program for

a class of document which comes in one of several layouts. The natural thing to do is to design a heuristic scoring system to evaluate the quality of an extraction numerically, and indeed we observed developers doing this in the course of writing extraction programs using a GPPL, multiple times. This requires up-front design to integrate well into a large, complex extraction program, and can also be tedious.

We saw that developers were solving the field-ordering and extraction-rank-ordering subproblems again and again, and decided that an extraction framework having the right set of abstractions could lift those burdens from developers.

Amenability to automation. An extraction program automates the process of extracting data from a particular kind of document, but the holy grail of extraction automation is automating the generation of the extraction programs themselves. This seems natural in the context of the document extraction problem space: human labelers can generate training data by annotating sample documents, which can then be used to generate extraction programs, or to train machine learning models. Unfortunately, due to the nature of procedural programming, automatically generating GPPL extraction programs from training data is hard.

2.2 Blueprint Language Design Principles

Declarative interface. We decided that it was important for the user interface to be declarative rather than procedural. In other words, rather than asking users to say something like, "find some label `A`, *and then* find some value `B` in relation to `A` by following a procedure `C`," we ask user to write expressions of the form, "the document contains a label `A` and a value `B`, and the relationship between them is `C`."

A declarative interface simplifies several of our other goals, such as a unified scoring system, reduced verbosity, and amenability to automation. In Blueprint, the bulk of user code is the specification of *constraints*, which are essentially declarations of facts about the documents under consideration. For example, a user might say something like `left_aligned('date_of_birth', 'first_name')` to indicate that in their document class, the date of birth and the first name fields happen to be left-aligned.

Choice of scale. We needed to decide whether we would ask our users to think at the level of pixels, individual characters, whole words, paragraphs, or some other scale. In the context of our system, it made the most sense to operate at the word/sentence scale. We accepted that this would mean a loss of flexibility at the margins, where it might sometimes be useful to work at a lower level (e.g., to correct character-level OCR errors), and decided that errors of that type could best be dealt with in pre- or post-processing.

Built-in fuzziness and scoring. The job of an extraction program is to discriminate among different candidate extractions for a particular document and pick the best one. Procedural programs do this implicitly. We made this explicit in our system via scoring and ranking of candidate extractions. Given a Blueprint program consisting essentially of a collection of constraints which are fuzzy by design, for each candidate extraction (a possible output), we compute a so-called *extraction score* which represents the degree to which the extraction satisfies the constraints specified in the program. The fuzziness is particularly important because real-life documents have noise. For example, the Tesseract OCR engine, when run on 100 upright images of Azerbaijani passports in the MIDV-2020 dataset [6], reads the "gender" labels in these passports in five different ways: `"Cinsi/Sex"` (the correct reading), `"(insi/5ex"`, `"(insi/$ex"`, `"Cinsi /5ex"`, and `"Cinsi/5ex"`. In our extraction program, we will want to include the constraint `text_equals("Cinsi/Sex")('gender_label')`, but if this constraint is not checked allowing some fuzziness or error tolerance, it will result in extraction failing on many samples.

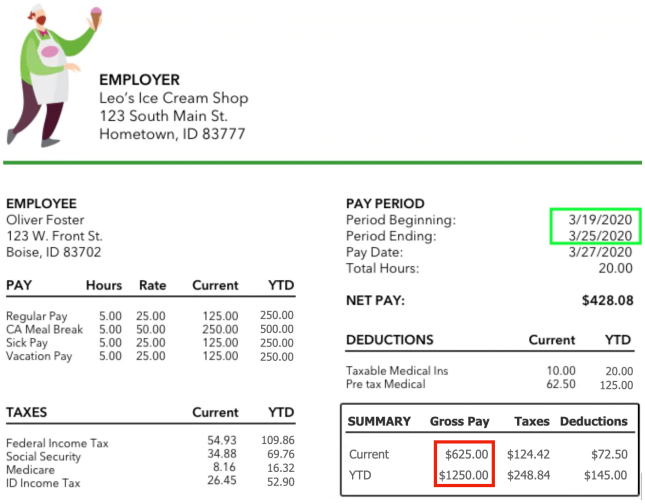
Explainability and debuggability. We wanted it to be the case that when an extraction program fails to generate the correct result for some document, it is straightforward to inspect the output to figure out why. This "output" can include the internal state of the runtime, with the caveat that this internal state should be interpretable, or easily made interpretable through tooling. For a counterexample, one can inspect the internal state of a deep-learning model during debugging, but this internal state can be very difficult to interpret – we wanted to avoid this situation.

Amenability to automation. We wanted our system to be usable by non-experts, ideally non-engineers, so we must auto-generate as much extraction logic as possible from labeled training data. We designed our system with this in mind.

3 RELATED WORK

Rule-based extraction. Many important business problems have been solved using rule-based approaches [5, 8, 13, 30, 31]. In the document-processing space, various extraction systems [14, 15, 24, 26, 32] were built before the advent of modern OCR engines. These systems worked at the pixel level, required extensive manual effort to build and maintain, and failed to generalize to unexpected scenarios, for example to documents with colorful backgrounds. To the best of our knowledge, Blueprint is the first rule-based document extraction system that works at level of OCR output, i.e., at the word-and-sentence level. Blueprint's layout-oriented rules also operate at this level of abstraction (as opposed to the pixel level), facilitating extraction logic that maps more directly to document structure as perceived by humans. Finally, Blueprint programs can be synthesized, reducing the required manual labor. Previous research [3] has shown that auto-generating information extraction rules can significantly reduce manual effort. This was true in our experience as well and led to drastic improvements in usability.

Similar to Blueprint, many rule-based systems have used the page structure, as perceived by humans, such as for extracting information from webpages [7, 12, 20, 21]. Some of these systems [7, 20, 25, 29] work on the webpage's DOM trees that are not available in document images. More generally, these systems do not deal with noise originating from OCR engines and have differing usage goals than Blueprint. Hence, these systems typically employ strict rules whereas Blueprint is built using fuzzy rules. Further, these systems usually only interface with expert engineers writing extraction logic in the context of a search engine or a knowledge base trying to understand the webpage. Whereas, we purposefully designed a no-code IDE to support interactively synthesizing, editing, testing, and debugging programs for use by non-experts.



EMPLOYER
Leo's Ice Cream Shop
123 South Main St.
Hometown, ID 83777

EMPLOYEE
Oliver Foster
123 W. Front St.
Boise, ID 83702

PAY PERIOD
Period Beginning: 3/19/2020
Period Ending: 3/25/2020
Pay Date: 3/27/2020
Total Hours: 20.00

NET PAY: $428.08

PAY	Hours	Rate	Current	YTD
Regular Pay	5.00	25.00	125.00	250.00
CA Meal Break	5.00	50.00	250.00	500.00
Sick Pay	5.00	25.00	125.00	250.00
Vacation Pay	5.00	25.00	125.00	250.00

DEDUCTIONS	Current	YTD
Taxable Medical Ins	10.00	20.00
Pre tax Medical	62.50	125.00

TAXES	Current	YTD
Federal Income Tax	54.93	109.86
Social Security	34.88	69.76
Medicare	8.16	16.32
ID Income Tax	26.45	52.90

SUMMARY	Gross Pay	Taxes	Deductions
Current	$625.00	$124.42	$72.50
YTD	$1250.00	$248.84	$145.00

Figure 2: Another sample paystub with a different layout.

Deep-learning based document extraction. There are many researchers tackling document extraction by building sophisticated deep-learning models [22, 23, 34]. LayoutLM [34] extends BERT [11] by adding bounding box position and image embeddings, enabling extraction models to learn document layout. Majumder, et al., [23] apply representation learning to learn the type representations of fields to be extracted, e.g., learning that `invoice_date` is often a date (a formal class or type), and to learn neighborhood relationships.

Deep-learning models are tricky to understand, debug, and maintain in industry settings, an observation also made in past research [9]. When a model is observed generating incorrect output, it remains very hard to understand why the incorrect output was generated, and even harder to fix the error. Further, deep-learning models often require powerful GPUs to train and run, and have large memory footprints. Blueprint programs are inherently explainable and comparatively cheap to run.

However, deep-learning models can be very effective for extraction from *long-tail* document types: classes of document where the format varies widely, making rule- or template-based extraction difficult. US paystubs are one example of a long-tail document type. Figures 1 and 2 show two sample paystubs. These two documents contain the same kind of information – employer name, monthly salary, etc. – but their layouts are very different. Using Studio, our approach to creating an extraction program that works on both of these samples will be to annotate these layouts separately. Blueprint will test each given layout against each document sample, and simply use whichever layout works best in each case. This works well for classes of document having tens or even sometimes hundreds of layouts, but breaks down when the number of layouts grows beyond a certain point.

Synthesizing constraint programs. A major advantage constraint programs have over ML models is interpretability. However, as discussed in previous research [4, 17], when constraint programs are synthesized, interpretability often vanishes because the number of generated constraints may be very large and as a result, hard for humans to reason about.

```
run(input_doc,
  extract(
    is_dollar_amount('period_gross_pay'),
    is_dollar_amount('ytd_gross_pay'),
    text_equals("Gross Pay")('gross_pay_label'),
    text_equals("Current")('period_label'),
    text_equals("YTD")('ytd_label'),
    right_aligned('gross_pay_label', 'period_gross_pay',
                  'ytd_gross_pay'),
    bottom_aligned('period_label', 'period_gross_pay'),
    bottom_aligned('ytd_label', 'ytd_gross_pay')))
```

Listing 1: An example Blueprint program.

```
{ 'gross_pay_label':  "Gross Pay",
  'period_gross_pay': "$625.00",
  'period_label':     "Current",
  'ytd_gross_pay':    "$1250.00",
  'ytd_label':        "YTD" }
```

Listing 2: An example Blueprint extraction. Note that extraction values technically are not just strings, but also include bounding-box information.

Blueprint programs can be automatically synthesized, see Section 6.1. In alignment with the previous research, we found that indeed, synthesized Blueprint programs tend to be larger than hand-written programs by a factor of about 5. However, in our deployments, we observed that operators are rarely interested in fully understanding extraction programs. Interpretability is important primarily during debugging, where it is enough to determine which particular constraints failed on a particular document, and how they should be adjusted to fix the program. Our system allows operators to do this easily and is described in Section 6.

Disjunctive programs. Recent research [18] improves deep-learning based document extraction by verifying model outputs with a synthesized disjunctive program. A disjunctive program consists of a set of procedural programs, and a set of constraints for the desired output [28]. Blueprint provides a more-fleshed-out DSL for expressing complex document extraction logic, and contains a larger library of constraints, such as spatial and algebraic constraints, not present in [18].

4 THE BLUEPRINT LANGUAGE

Blueprint is a DSL, inner to Python 3, for document extraction. When writing Blueprint programs, users describe their documents by stating facts about them. These facts may be very general, but often describe spatial, textual, semantic, or numerical relationships among pieces of the document.

Suppose that given the document shown in Figure 2, we wish to extract the `'period_gross_pay'` and the `'ytd_gross_pay'`, highlighted by a red box for illustration. Listing 1 shows the body of a possible Blueprint program solving this task. Running this program on this document results in the extraction shown in Listing 2.

More formally, a Blueprint program consists of a collection of constraints joined by logical connectives and other composition mechanisms. The goal of a Blueprint program is, given an input document, to produce an *extraction* for that document, satisfying the constraints and other logic of the program. An *extraction* is a dictionary or mapping from the *fields* of the program that produced it to *entities* in the input document.

Table 1: Examples of Blueprint constraints. Each entry is a single constraint.

Example	Meaning
Textual, semantic, and numerical constraints	
`text_equals("Gross Pay")('gross_pay_label')`	The text of the `'gross_pay_label'` is roughly equal to `"Gross Pay"`, scored by edit distance.
`is_date('pay_date')`	The `'pay_date'` is a date. Scored based on similarity to hard-coded patterns like `"JAN-01-01"`, `"01-JAN-01"`, `"01-01-01"`, `"January 1, 2001"`.
`is_dollar_amount('gross_pay')`	The `'gross_pay'` is a dollar amount. Scored based on multiple heuristics: whether the candidate's text consists of only digits and `"$"`, `"."`, etc.
`sum_is_approximately(15, [1, 2, -1])('regular', 'overtime', 'gross')`	Checks an approximate weighted sum: informally, 15 ≈ `'regular' + 2 * 'overtime' - 'gross'`. Scored based on the difference between the target and the observed sum.
`sum_is_at_least(200, [1, 1.5])('wages', 'overtime')`	Checks an inequality with a weighted sum: informally, `200 <= 1 * 'wages' + 1.5 * 'overtime'`. Scores 1 if the inequality is satisfied, else 0. This is a pass-fail constraint.
Spatial constraints	
`left_aligned('date_of_birth', 'id_number')`	The `'date_of_birth'` and `'id_number'` are left-aligned. Scored based on on the distance between the x coordinates of the left sides of the candidates' bounding boxes.
`top_down('period_label', 'period_pay')`	The `'period_label'` appears above the `'period_pay'` in the document, as it might in a tabular layout, for example. Scored based on the y coordinates of the bottom edge of the bounding box of `'period_label'` and the top edge of the bounding box of `'period_pay'`.
`nothing_between_horizontally('pay_period_label', 'period_begin_date')`	The `'pay_period_label'` and `'period_begin_date'` do not have anything occluding the space between them, looking in the horizontal direction. For example, a dense block of text placed between them would result in a constraint score near 0.
`is_in_page_region((0.5, 1), (0, 0.5))('pay_date_label')`	The `'pay_date_label'` should appear in the upper-right quadrant of the page. Scored based on the portion of the candidate's bounding box which lies in the specified region.
`is_entire_phrase('gross_pay_label')`	The `'gross_pay_label'` consists of all of the text in a cluster. For example, an entire multiword table column header should satisfy this constraint, but a single word picked out of the same header should not. Clustering is based on variance in word height, distance between words, etc.
Logical connectives and modifiers	
`any_holds(text_equals("Gross pay"), text_equals("Total pay"))('gross_pay_label')`	Disjunction: the text of the `'gross_pay_label'` matches either `"Gross pay"` or `"Total pay"`. The score is the maximum of the scores of the subconstraints.
`all_hold(text_equals("Current"), is_entire_phrase)('current_label')`	Conjunction: the text of the `'current_label'` matches `"Current"`, and it is not part of a larger cluster of text. The score is the product of the scores of the subconstraints.
`negate(is_in_page_region((0, 0.5), (0.5, 1)))('date_of_birth')`	The `'date_of_birth'` must *not* be in the lower-left-hand quadrant of the page. The score is 1 minus the subconstraint score.
`any_holds(text_equals("Mailing address"), penalize(text_equals("Address")))('mailing_address_label')`	The `'mailing_address_label'` may have text equal to either `"Mailing address"` or `"Address"`, but matching with `"Address"` is penalized. The `penalize` modifier linearly scales the score of its subconstraint $[0,1] \mapsto [0,0.7]$. Another modifier `non_fatal` linearly scales the score of its subconstraint $[0,1] \mapsto [0.4,1]$.

For example, the fields of the program in Listing 1 are `'period_gross_pay'`, `'ytd_gross_pay'`, `'gross_pay_label'`, `'period_label'`, and `'ytd_label'`. If `E` is the extraction in Listing 2, then `E('period_gross_pay') = "$625.00"`, etc.

An *entity* is defined to be a piece of the document that can appear as a value in an extraction. We discuss entities further in Section 5.1. For now, we should think of the entities as (1) the individual words in the document, and (2) clusters of words, e.g., `"PAY PERIOD"`. We call $\{E(f) \mid f \in \text{Fields}(E)\}$ the *extracted values* of `E`.

The main primitives of the Blueprint language are fields, entities, extractions, predicates, constraints, and nodes. We have just described what fields, entities, and extractions are, and will describe predicates, constraints and nodes in the rest of this section. Throughout, it will be helpful to refer to Table 1 for examples.

4.1 Constraints

Blueprint programs consist mostly of *constraints*. A *constraint* is a declaration of fact about a document, and consists of a *predicate*, which is the verb portion, and one or more fields that the predicate applies to. For example, the constraint `is_date('date_of_birth')` applies the predicate `is_date` to the field `'date_of_birth'`.

The syntax to define a constraint is `predicate_name(field1, field2, ...)`. Some predicates, e.g., `text_equals` and `is_in_page_region`, accept args – for these, the syntax is `predicate_name(arg1, arg2, ...)(field1, field2, ...)`.

Constraint scoring. We define a notion of *constraint score*, which depends on (1) a constraint and (2) an extraction, and measures how-well the extraction satisfies the constraint. A constraint score is a number between 0 and 1, and how exactly it is calculated depends on the predicate used to define the constraint – see Table 1.

As an example, if `C = bottom_aligned('last_name', 'first_name')`, and `E` is an extraction with values for `'last_name'` and `'first_name'`, then ConstraintScore(`C`, `E`) is a number in $[0,1]$, which is computed by examining the bottom edges of the bounding boxes of `E('first_name')` and `E('last_name')` and measuring their alignment.

Constraint composability. Blueprint ships with a library of basic predicates. It also provides the logical connectives `all_hold` and

any_holds to form conjunctions and disjunctions of predicates, respectively. In addition, we provide the modifiers non_fatal and penalize for finer-grained control over constraint scoring. Again, refer to Table 1 for examples.

4.2 Simple Fixed-layout Extraction

For a document type with just a single, fixed layout, a Blueprint solution can consist of essentially just a list of constraints describing the document layout. Listing 1 is such a program. In this section we deconstruct this program in slightly more detail. The only pieces of this program, besides the constraints, are (1) the extract node on line 2, and (2) the run(input_doc, ...) call on line 1.

extract *nodes.* We can think of an extract node as essentially just a list of constraints. It is called a *node* because as we discuss in the next section, Blueprint allows users to model hierarchical extraction logic in a tree, and extract is one of the node types of a Blueprint extraction tree.

When an extract node is given an input document, the node returns a collection of extractions for that document which satisfy the node's constraints – a collection because there may be more than one extraction that works, or zero. The extractions' fields will be the same as the union of the fields of the extract node's constraints. For example, compare the fields in Listings 1 and 2. If an extract node cannot find an extraction having *all* of these fields, and satisfying *all* of the constraints (except those wrapped in non_fatal), it will return a null extraction. It will never return "partial" extractions.

Extraction scoring. We need a way of comparing different extractions on the metric of how-well they satisfy the constraint logic of a Blueprint program. To do this, we designed a method of scoring extractions. We wanted a scoring method that has the properties that (1) a partial or total failure of a constraint is reflected in the score of every field participating in that constraint, and that (2) scoring weighs the fields additively. We found that the following heuristic approach worked well in practice:

Suppose that N is an extract node, that E is an extraction produced by N, and that f is a field in E. Then

$$\text{FieldScore}(f, E, N) = \prod_{\substack{C \in \text{Constraints}(N), \\ \text{where } f \in \text{Fields}(C)}} \text{ConstraintScore}(C, E), \text{ and}$$

$$\text{ExtractionScore}(E, N) = \operatorname*{Average}_{f \in \text{Fields}(E)} \text{FieldScore}(f, E, N).$$

The run *function.* Lines 2–11 of Listing 1 define an extract node. The run call on line 1 generates entities for this input_doc (see Section 5.1), passes the entities to this extract node, and returns the highest-scoring extraction that is returned by the node. Every Blueprint program looks like run(input_doc, node). We discuss execution more in Section 5.

4.3 Hierarchical and Variable Document Layouts

It can be useful to extract different portions of a document separately and combine the results, or to express several alternative approaches for extracting a particular part of a document due to variations in format. For example, perhaps we wish to extract both the green- and red-highlighted portions of the sample in Figure 2 – the layouts of these two parts of the document are unrelated, so we would like to express their extraction logics separately if we can, for reasons of modularity and separation of concerns.

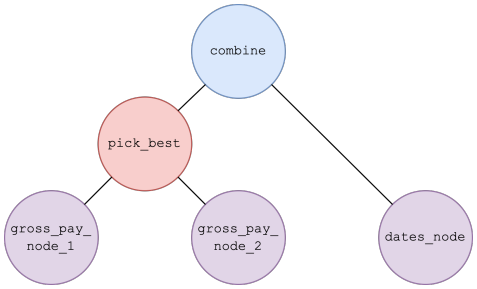


Figure 3: Blueprint extraction tree.

To support these capabilities, Blueprint allows users to model extraction logic in a tree. The three main node types of a Blueprint extraction tree are (1) extract, for expressing local, self-contained document layout or structure, (2) pick_best, for handling variability in document layouts, and (3) combine, which in effect allows users to express extraction logic for different parts of their document separately. We have just described extract nodes in Section 4.2; we now move on to the other two node types.

Variable document layouts. The pick_best node type is used to express that there are multiple, different ways that part of a document could be extracted, depending on layout format or other factors. For example, suppose that we define gross_pay_node_2 = extract(⟨constraints from Listing 1 which correspond to Figure 2⟩), and gross_pay_node_1 to be another extract node consisting of the constraints for extracting the red-highlighted portion of the paystub in Figure 1, which has the same fields but in a different format. Then pick_best(gross_pay_node_1, gross_pay_node_2) will be a Blueprint node for extracting gross pay values which will work on both of our paystub samples.

More formally, the node pick_best(N1, N2, ...) multiplexes the extractions returned by N1, those returned by N2, etc., into a single collection, sorted roughly by extraction score. Typically the Ni will have the same or largely-overlapping sets of fields.

Hierarchical layouts. The combine node type is used to combine extraction logic for two different parts of a document. For example, suppose we define dates_node to be an extract node which has the appropriate constraints to extract the dates highlighted in green in Figures 1 and 2 (noting that the labels and layout are the same in both documents). Then we may put all of the above building blocks together into an extraction program that extracts both the red- and green-highlighted portions of the two sample paystubs as shown in Listing 3. The resulting tree structure can be seen in Figure 3.

In general combine(N1, N2, ...) will be a node which returns extractions having all of the fields of all the Ni. It is required that the Ni do not have any fields in common.

4.4 Extensibility

Custom predicates. In addition to building predicates via composition using any_holds and all_hold, it is straightforward to

```
# Defining a custom constraint
next_to_each_other = all_hold(
  bottom_aligned, left_to_right,
  nothing_between_horizontally)

# Constraints to extract red portions of Figures 1 and 2
gross_pay_node_1 = extract(...)
gross_pay_node_2 = extract(...)

# Constraints to extract period beginning/ending
dates_node = extract(
  text_equals("Period beginning:")('period_begin_label'),
  text_equals("Period ending:")('period_end_label'),
  is_date('period_begin_date'),
  is_date('period_end_date'),
  next_to_each_other('period_begin_label',
                     'period_begin_date'),
  next_to_each_other('period_end_label',
                     'period_end_date'))

extraction_result = run(input_doc,
  pick_best(
    # Note we can use the dates_node repeatedly
    combine(gross_pay_node_1, dates_node),
    combine(gross_pay_node_2, dates_node)))

# Alternate version
extraction_result = run(input_doc,
  combine(
    pick_best(gross_pay_node_1, gross_pay_node_2),
    dates_node))

# An advantage of the first `extraction_result` version
# is that it would be easier to add a third format that
# does *not* use the same layout for the period begin
# and end dates.
```

Listing 3: Blueprint program for extracting the gross pay and date information from paystubs. The `combine` and `pick_best` node types are used to form a hierarchical extraction tree.

define custom predicates. An arity-n predicate is defined by a `score` function which takes n entities as input and returns a number between 0 and 1. We refer readers to the source code [1] for examples.

Custom node types. The interface of a Blueprint node is: the node will be passed an input document, and should return a collection of extractions, with extraction scores. In order to write custom node types, we need only to implement this interface. An example of a useful custom node type that would be straightforward to implement is one which returns extractions generated using a deep learning model. These deep-learning-generated extractions could be integrated with Blueprint-generated extractions using `combine` nodes, or users could create extraction programs which use either deep-learning or constraint-based logic for all or part of a document depending on which performs better at runtime, using `pick_best`.

5 EXECUTION

The goal of a Blueprint program is, given OCR words coming from an input document, produce an extraction for that document. We break this process into two steps: (1) entity generation, where we discretize the search space by identifying the *entities* in the document which we may wish to return as extracted values, and (2) exploring the search space of possible extractions to find one that satisfies our program's constraints and other logic.

5.1 Entity Generation – Discretizing the Search Space

An *entity* is a data type which represents something in the document which we may want to extract. In all of our examples, entities consist of a string of text (without formatting information) and a bounding box representing the entity's location on the page, plus some additional information, such as whether the entity has been identified as a date. In principle [30], entities could include font or color information, and then this information could be used in constraints, or we could include non-textual entities such as dividing lines or images, but this is not done in the current implementation.

The top-level input to a Blueprint program is the output of an OCR engine, which consists of words and their bounding box information. Each of these words is taken to be an entity. The Blueprint language itself does not give the user any control over entity generation. However, by default the Blueprint runtime generates further entities from the OCR words, for example by forming entities from clusters of nearby words, such as `"Total Hours:"`, using spatial heuristics. Note that entities are not pairwise-disjoint: the phrase `"Total Hours:"` and its constituent words `"Total"` and `"Hours:"` may all appear as entities simultaneously. The configuration parameters of this process are tweakable by the user.

Performing entity generation up-front discretizes our search space for the constraint logic step. The only thing which may appear as a value in a Blueprint extraction is an entity. However, the more entities we include in the entity pool, the larger the search space, which can negatively impact runtime. We have found that optimal entity generation varies from use case to use case, but Blueprint's current default configuration is generally a good starting point.

5.2 Searching the Space of Possible Extractions

Next, we pass the generated entity pool to the root node of our extraction tree, which is in turn responsible for producing a collection of extractions for this document. We return the highest-scoring extraction as our final result.

It remains only to describe how the different node types produce their extractions. We have already sketched the implementation of `pick_best` in Section 4.3. The implementation of `combine` has a similar spirit, combining extractions coming from its children to produce a new collection of output extractions. The implementation of `extract` is non-trivial.

An `extract` node `N` having fields `f1, ..., fn` is responsible for producing extractions of the form `E = {f1: e1, ..., fn: en}`, where the `ei` are entities, such that `E` satisfies all constraints in `N`. The search space is discrete but far too large to search exhaustively.

In our implementation, we build `E` iteratively by successively adding values for the required fields one by one. After each addition, we verify that every constraint `C` in `N` which we can check (those `C` for which we have assigned values for every field in `C`) has non-zero score. If we assign a value `e` to a field `f`, and this causes some constraint involving `f` to fail, then we back out this assignment for `f` and continue the search. Note that we may revisit the assignment `f: e` at a later point in the search, since locally it may be the correct assignment, and fails only in conjunction with some earlier assignment for another field, which may be changed later. When we arrive at an `E` which has an assignment for every required field,

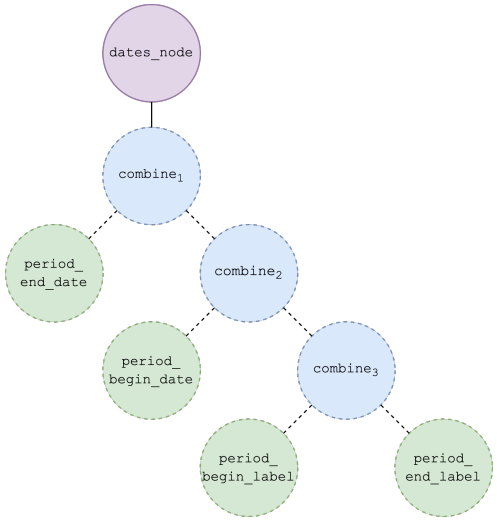


Figure 4: An `extract` node's implementation in terms of other Blueprint primitives.

we save it in our list of extractions to return, and then back out one or more assignments and continue the search.

The main implementation challenge is that the order in which we assign values to the fields has a very large impact on runtime. For example, referring back to Listing 1, if we assign values to `'period_gross_pay'` and `'ytd_gross_pay'` first, then we will end up looping through every partial extraction of the form `{'period_gross_pay': V1, 'ytd_gross_pay': V2}` where `V1` and `V2` are dollar values, searching essentially at random for the values of `V1` and `V2` for which we end up being able to find assignments for the labels satisfying the layout constraints.

The solution in this case is to assign values to the label fields first. Then when we reach the step of assigning a value to `'period_gross_pay'`, we will be able to discard most candidates immediately because they will cause one of the layout constraints to fail. Our solution in the general case is to use a combination of heuristics to determine a field extraction order at runtime. Such heuristics include the number of entities which satisfy the arity-1 constraints in which a field participates (consider the number of entities which satisfy `text_equals("Gross Pay")` vs. `is_dollar_amount`), and the number of outstanding constraints that assigning a value for a field would immediately allow us to check. We do this at runtime because these heuristics can be document-dependent.

We implement `extract` nodes using the other Blueprint language primitives. Specifically, we transform every `extract` node into a Blueprint extraction subtree, as shown in Figure 4. In the figure, the green nodes are so-called *leaf nodes*, which produce single-field extractions. The choice of field extraction order corresponds to the order in which leaf nodes appear in the tree, and again, this is computed at runtime using heuristics on the constraints and the document. The constraints from our `extract` node are distributed throughout this tree, with each constraint pushed down to the lowest node where all of the constraint's fields are present. For example, the constraint `left_aligned('period_begin_label', 'period_end_label')` will be checked at `combine3`, and `bottom_aligned('period_end_label', 'period_end_date')` will be checked at `combine1`. Combining the Blueprint primitives in this way to implement `extract` effectively achieves the implementation sketched above.

We may do further optimizations, such as by collecting and employing data set statistics [31], but we found it to be unnecessary. This is because Blueprint programs could finish reasonably quickly on its target use cases and does not need to maintain an execution context, such as for keeping statistics, across extractions.

6 STUDIO: NO-CODE IDE FOR BLUEPRINT

Our experience building Blueprint extractions in a commercial setting, as well as training engineers to do the same, revealed several challenges. First, we learned that writing declarative fuzzy constraint-based programs, especially from scratch, is difficult and unintuitive for many developers. This is true, in particular, for developers trained in Python and other GPPLs, who are used to thinking procedurally. Also, debugging Blueprint programs often proved challenging, for reasons discussed below.

To address these issues, we built a no-code GUI application called Studio. Studio is written in React and runs in the browser. In a similar spirit as other annotator GUIs [10, 19], Studio has five main views, as shown in Figure 5:

1. A list of documents that a user has loaded into their project, and extraction accuracy metrics for recent trial runs. Accuracy and runtime metrics for different versions of the user's extraction program can be compared side-by-side.
2. A view showing OCR results, generated entities, target values, and extracted values rendered directly on the document. This section can also be used to annotate target values for a sample document, which can in turn be used for automatic constraint synthesis, and to compute extraction accuracy.
3. The high-level structure of the extraction tree. The rows in this view correspond to Blueprint nodes. These nodes can have user-defined names for readability during development, with their node types visible alongside in muted text. (Some primitives are named differently in the GUI compared to the language version of Blueprint: a *model* is an extraction program, a *pattern* node is an `extract` node, and a *merge* node is a `combine` node.)
4. When inspecting a pattern/extract node, this region allows the user to specify the fields they wish to extract, to specify types (date, word, phrase, etc.) for each field, to inspect and edit target values, and to inspect extracted values ("Model results") after running their extraction program.
5. The set of constraints in the current node. From here the user may add, edit, and delete constraints. In addition, if target values are available, here the user is shown how-well their constraints are satisfied *by the target values*. This is very useful for debugging synthesized programs, as discussed later.

6.1 Program Synthesis

Studio provides program synthesis functionality to make it easy to create Blueprint programs. Users load their document samples, label their target values by clicking in the graphical document view,

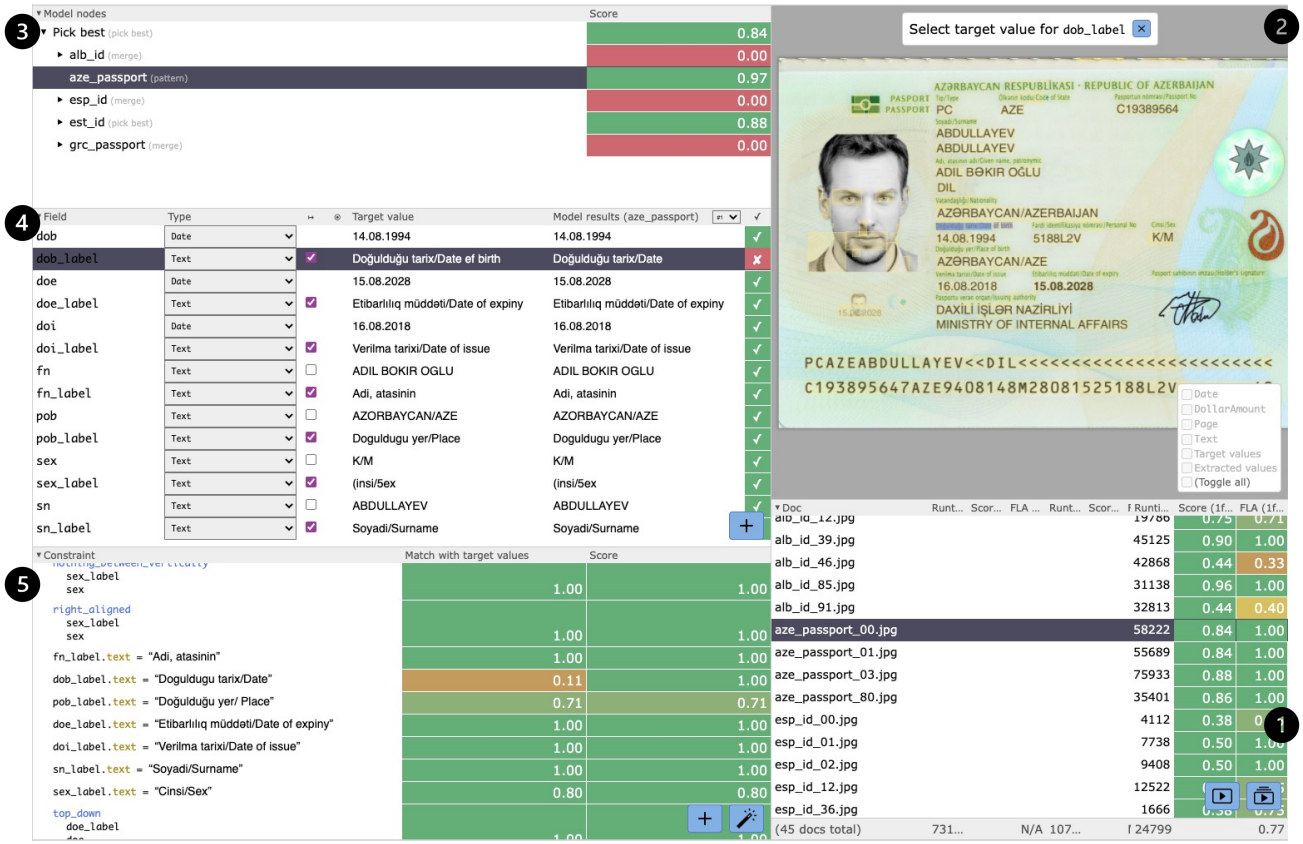


Figure 5: Studio, a no-code IDE for interactive development of Blueprint extractions.

run synthesis from the GUI to generate a Blueprint program, and then may immediately run their synthesized program against all of their samples for testing, debugging, and further development.

Suppose that `E` is a target extraction – a field-to-entity dictionary containing the user's desired extraction results for a particular document. The goal of the synthesis engine is to generate a Blueprint program which will successfully produce `E` as its extraction result for this document. The synthesis engine we provide only generates `extract` nodes and has a simple implementation. We begin by heuristically generating a large set of candidate constraints, using the fields of `E` and a subset of predicates which tend to be effective at capturing document structure, for example `left_aligned`, `right_aligned`, `top_down`, `left_to_right`, `nothing_between_vertically`, `nothing_between_horizontally`. For every candidate constraint `C`, we compute ConstraintScore(`C`, `E`), and include `C` in our synthesized `extract` node if this score is above a threshold. We also perform some additional heuristic checks, for example to attempt to identify and prune constraints which hold only incidentally, or are superfluous. These heuristics are specific to each constraint in Blueprint; we skip their details due to the page limit.

A typical user workflow is to (1) create a `pick_best` node, (2) identify the distinct formats in their collection of documents, (3) for each format, label an example document, and (4) synthesize an `extract` node for that format as a child of the `pick_best` node. Users found this workflow intuitive and straightforward.

6.2 Debugging

If a Blueprint program is failing to produce the expected results on a particular document, usually the cause is one of the following:

There is no entity in the entity pool corresponding to one of the target values. To help debug this, Studio shows all of a document's entities visually overlaid on the document. In addition, if a user has annotated their target values, Studio shows an alert if any target value does not have a matching entity in the entity pool, in which case there is no possible constraint logic that will result in a successful extraction for that field.

Some constraint(s) are not satisfied by the target extraction. This means that if we scored every constraint against the user's target extraction, at least one of the constraints would fail. This can happen if, for example, the tolerances on a constraint are set too low. Another way this could happen is that synthesis could add a constraint which happens to hold on the sample document used for synthesis, but which does not hold in general. A typical example would be two left-aligned text values being "accidentally" also identified as being right-aligned, because the lengths of their texts happen to be the same in the sample document.

This situation is very hard to debug without special tools, because Blueprint does not return partial extractions, and there is no notion at inference time of "the constraint that failed" – all that the inference engine tells us is that it failed to find an extraction *simultaneously* satisfying *all* of the constraints.

Studio makes this easy to debug. Given a target extraction and a set of constraints, we can identify the constraint causing inference failure by scoring each constraint against the target extraction. The constraint(s) causing inference failure will score 0. The solution is generally either to increase tolerances, or to otherwise modify or delete the faulty constraints; deletion would be the appropriate solution in the accidental-right-alignment example above.

The set of constraints has multiple solutions. It can happen that there are multiple extractions for some document which satisfy all of the constraints in an `extract` node. In this case, an incorrect extraction could be returned because it scores higher than the desired extraction. The solution in this case is usually to add additional constraints to rule out the undesired result while keeping the desired one. Studio indicates when a node returned multiple extractions for a document, and allows users to page through them.

7 DEPLOYMENT EXPERIENCE

Our test users for Blueprint and Studio were predominantly sales and support engineers. These users write code as part of their job, but generally do not design, build, and maintain complex software systems. Our users generally had prior experience writing extractions using a GPPL (usually Python).

User 1 built Blueprint extractions for several customers. They write: "Blueprint significantly lowers the technical barrier for entry to writing document understanding solutions. Once you understand how a Blueprint template gets constructed, writing your solution becomes a problem of solving a logic puzzle rather than constructing a piece of software from scratch. This reframing of document understanding makes scaling and changing solutions much easier."

User 2 is a sales engineer, is an expert in building document understanding solutions, experimented with using Blueprint for various customer trials during sales, and oversaw the development of a large (1000-line), long-tail Blueprint solution, written by hand without Studio. They write: "The mental model you need to solve an extraction problem using raw Python required not only skill in being able to articulate your intent as a developer in Python code, but also the ability to hold the mental model in your head about all of the steps and processes that you need to do... it was just such a big barrier to entry. [...] Through Studio, with the ability to annotate and auto-generate constraints, the Blueprint language then became about tuning, like removing constraints, adding constraints, which is a much easier thing to get your head around. And the experience of going through and annotating the documents was way more intuitive. [...] Starting from a blank slate and trying to write a Blueprint program was very intimidating. The Blueprint language was actually much harder to wrap your head around than writing the raw Python, but Studio abstracted away a lot of that pain to the point where you could just annotate and then refine, and for some reason, once you click auto-generate, and you see all the rules generated, you haven't had to think about how to generate them, but now that you see them it makes sense."

User 2's feedback is very representative of the general feedback we got. Users often struggled initially to wrap their heads around the main Blueprint abstractions, but found the Studio development experience intuitive and productive, and ultimately it was the technical design of the Blueprint language (constraint matching, scoring, etc.) that made it possible for us to build program synthesis and the other interactive development functionality in Studio.

8 EVALUATION

In this section, we first compare synthesized Blueprint programs with hand-written Blueprint programs, and then compare Blueprint with a state-of-the-art deep-learning based document extraction tool, LayoutLM [34]. All experiments are done on a server with an Intel Xeon Gold 6330 processor and an Nvidia A40 GPU. Each experiment is run serially, with no other workloads running concurrently on the server. Execution time is "user time" and memory usage is "maximum resident set size", as reported by `time -v`. Blueprint does not use the GPU.

Blueprint and LayoutLM both operate on Tesseract [33] OCR output, and we used Tesseract as our OCR engine. When checking extraction accuracy, we ignore errors stemming from OCR. For example, in one sample, the true text value for a particular field is `"BUŞ"`, but OCR reads this as `"BUS"` – if either tool obtains `"BUS"`, we treat that as a correct extraction.

8.1 Comparing Synthesized and Hand-written Blueprint Programs

Experimental setup. For this comparison, we used 100 Azerbaijani passport images from the MIDV-2020 [6] dataset. We develop an extraction program for the fields `'first_name'`, `'surname'`, `'date_of_birth'`, `'date_of_issue'`, `'date_of_expiry'`, `'gender'`, and `'place_of_birth'`.

Summary of results. Table 2 summarizes our findings. Writing a Blueprint program for this dataset by hand is straightforward: the program consists of only 18 constraints, and could be written by an experienced Blueprint user in just a few minutes. The resource usage is also very modest: extraction for a single document finishes in just over a second and uses only 30 MB of memory.

We also created a synthesized Blueprint program for the same dataset using Studio. This took just a couple of minutes. The synthesized program had similar memory usage and ran in an average of 3.2 seconds per document. In line with previous results [4, 17] on fuzzy constraint program synthesis, our synthesized program contained more constraints: 42 compared to 18 in the handwritten program. The main reason for this is that a human author will tend to include the minimal set of constraints required to capture the documents' structure, whereas the synthesis engine tends to include many spatial constraints that a human may consider extraneous – these extra constraints do hold, and so they don't interfere with successful extraction, but they are not critical parts of the structure of the document and could as well be left out. When the synthesized program was run on the full 100-sample dataset, some documents did not extract properly due to OCR errors, but after a small amount of error tolerance tuning using Studio's debugging facilities, both the hand-written and synthesized programs were able to extract all the fields correctly.

8.2 Comparing Blueprint with LayoutLM

Experimental setup. We have applied Blueprint and Studio in commercial settings to build extractions for paystubs, bills of lading,

Table 2: Running Blueprint and LayoutLM extractions against the MIDV-2020 dataset. HBP = hand-written Blueprint program, SBP = synthesized Blueprint program.

Approach	Total constraint count	Average runtime per doc	Average memory usage	Field-level accuracy
Single-layout extraction (100 Azerbaijani passports)				
HBP	18	1.1 s	30 MB	100%
SBP	42	3.2 s	28 MB	100%
Multi-layout extraction (500 ID cards from 5 different nations)				
SBP	99	3.4 s	32 MB	94.4%
LayoutLM	-	16.5 s	4.2 GB	83.1%

bank checks, etc. However, we did not find open datasets for these use cases. On the receipt images in the SROIE [16] dataset used in the original LayoutLM paper, we did not attain accuracy results using Blueprint which were competitive with those of LayoutLM. This is because receipts contain less structure and fewer labels and anchors than the other document classes mentioned above, and present with a very long tail of layouts. Blueprint works best for datasets with fairly-consistent layout, where fields also tend to have good labels or anchors. For these reasons, Blueprint also does not work well on natural language use cases.

We perform our evaluation using five ID card types (Azerbaijani passports, Albanian IDs, Spanish IDs, Estonian IDs, and Greek passports) in the open MIDV-2020 [6] dataset. Both Blueprint and LayoutLM are given the same OCR outputs. A few ID card types have colorful backgrounds and labels, which can interfere with OCR. To mitigate this, we preprocess some of our samples to remove a color channel. In particular, for the `alb_id` samples we use only the blue and green channels, and for the `esp_id` samples we use only the blue and red channels. We trained a LayoutLM model for performing extraction across the five ID card types using 20 training samples per card type, and built a Blueprint program to do the same extraction. In both cases, a single model or program is responsible for extracting all five document types or layouts – there is no preliminary classification step. The Blueprint program consists of a `pick_best` node with five child nodes, one per layout, synthesized using Studio.

Summary of results. Our results are summarized in Table 2.

We first observe that predictably, the five-layout synthesized Blueprint program is larger (has more constraints) than the single-layout programs, but this does not significantly affect runtime or memory usage in this case. This is because adding constraints to a Blueprint program tends to improve rather than hurt its runtime, because every extra constraint shrinks the search space the program needs to examine.

Next, we examine accuracy. This experiment extracts 2900 fields in total, from 500 images. The synthesized Blueprint program extracted 2738 fields correctly (94.4%), whereas LayoutLM extracted 2409 fields correctly (83.1%)[1]. We see that on this dataset, a synthesized Blueprint program outperformed LayoutLM on extraction accuracy. We do not make the claim that Blueprint is more accurate than LayoutLM. For example, we would expect the LayoutLM

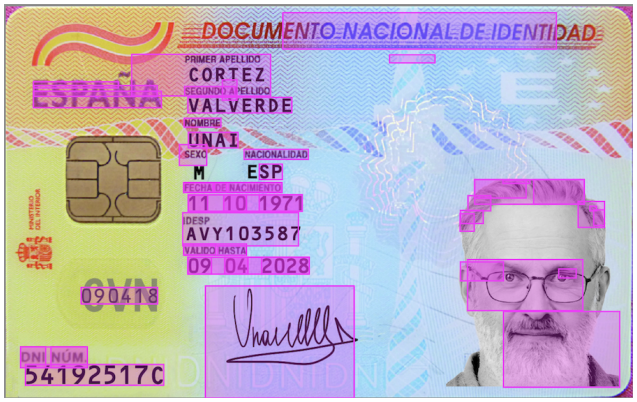


Figure 6: OCR errors on esp_id/12.jpg from MIDV-2020 dataset. All OCR words and generated Blueprint entities are outlined with a magenta box.

model to perform better on a document sample having an unseen-but-similar layout, for example a French passport, where we would expect the Blueprint program to return nothing at all, since Blueprint generally does not return partial matches.

Training the LayoutLM model takes 75 seconds using the GPU, and over 10 minutes on a CPU. Synthesizing the Blueprint program does not take significant compute time. Developer time – training the LayoutLM model versus synthesizing and tweaking the Blueprint program – is comparable on this use case, although for long-tail use cases with many layouts, LayoutLM training could require less developer effort.

Finally, we consider resource usage. As shown in Table 2, on this dataset, Blueprint compares favorably to LayoutLM regarding resource usage, taking 5x less runtime and using 150x less memory[2]. We make the disclaimer that in general, LayoutLM is an ML model and will at least have consistent, predictable runtime, whereas the runtime of a Blueprint program will depend on how the program is written, as well as on the input documents. Blueprint runtime can be very high in bad cases and can be difficult for users to predict or reason about, which was a problem we encountered in user testing.

Overall, this evaluation shows that for appropriately-chosen use cases, a heuristic constraint-based approach such as Blueprint can be comparably effective to a deep-learning approach in terms of accuracy and development effort, and can have lighter resource requirements. This is consistent with our experience running Blueprint programs in a commercial setting.

Detailed failure-mode analysis. There are three main failure modes for Blueprint on this data set.

First, the OCR engine sometimes completely fails to recognize some text in the document sample. For example, in Figure 6, every recognized OCR word and generated entity is shown outlined with a magenta box. We see that here, the OCR engine completely failed to recognize the letter `"M"` (representing gender), located in the upper-left quadrant of the image.

Second, the OCR engine sometimes returns severely incorrect results: completely-wrong text, a bad bounding box, or both. Blueprint constraints are fuzzy and have some error tolerance, but in some

[1]We do not claim that LayoutLM accuracy could not be improved through model tuning. Our LayoutLM experimental setup is based on the scripts described in [2].

[2]We also do not claim that LayoutLM resource requirements could not be reduced through approaches such as distillation and quantization [27].

cases the OCR errors are too severe. For example, in Figure 6, we see that the bounding box around the surname "CORTEZ" extends too far to the left, all the way to the Ñ in ESPAÑA. The synthesized Blueprint node for this layout has a `left_aligned('surname', 'first_name')` constraint, which fails due to this bad bounding box, causing extraction to fail on this sample.

Third, because of the above errors in OCR output, our heuristic algorithm for generating multi-word cluster entities sometimes fails to create the expected entities. For example, in some documents, we observed that the day, month, and year portions of a date have widely-varying bounding box heights as reported by OCR, even though the word heights on the page are all the same. Our heuristic clustering algorithm prefers words in a cluster to have the same height (as a proxy for font size), so clustering could fail in such cases. The only things that can appear as values in a Blueprint extraction are entities, so if the clustering algorithm fails to cluster a day-month-year triple into a single entity, there is no way that the corresponding date field will be extracted correctly.

In many cases, LayoutLM is able to recover from such OCR errors better than Blueprint, and successfully return a value. However, this is sometimes of dubious utility. For example, for a certain date in sample `est_id/41.jpg`, the correct value is `"21.03.2020"`, but OCR returns `"21-0020!"`. The LayoutLM model *was* able to extract this value, but it is hard to see how this could be useful downstream, given how different the OCR output is from the true value on the document. More examples of this scenario, where LayoutLM extracted the value having the correct provenance despite bad OCR:

- `(file_name, actual_text, extracted_text)`
- `alb_id/05.jpg, "14-11-2018", "AA2018"`
- `alb_id/07.jpg, "12-07-2016", "apie(2016"`
- `alb_id/12.jpg, "03-10-2015", "PeeAS-2015"`
- `est_id/99.jpg, "27.05.2022", "27:GEEEO2"`

We observed failure modes in LayoutLM that Blueprint does not have. LayoutLM treats extraction as a classification problem on the words of a document: every field we are trying to extract is treated as a class, and every word in a document is classified either as one of the fields, or as "none". Then the extracted value for a field is defined to be the set of words in that field's class. As a consequence, LayoutLM sometimes produces partial extractions, or extracts "multiple values" for a given field. For instance, in `grc_passport/87.jpg`, LayoutLM extracts the `'issue_date'` as `"03 15"` instead of `"03 Sep 15"`. Generally, in the `grc_passport` samples, LayoutLM misses the month portion of an extracted date value in 30 out of 100 images. LayoutLM also extracted multiple values for at least one field in 182 out of the 500 document samples. Some of these multi-value extractions can easily be fixed in post-processing: for example, in `alb_id/79.jpg` both `"Vlore,ALB"` and `"nr.no."` are extracted as the `'birth_place'`, and the correct value is clearly `"Vlore,ALB"`. However, in some cases, finding the one correct value in a multi-value extraction for a given field may be non-trivial: for example, in `aze_passport/23.jpg`, both `"06.03.2013"` and `"05.03.2023"` are extracted as the `'expiry_date'`. Also because LayoutLM treats extraction as a classification task, it sometimes provides extracted values for fields which are completely absent in a sample: for example, in `est_id/86.jpg`, LayoutLM extracts the `'issue_date'` as `"04.01.2005"`, but `est_id` samples do not have any issue date. We count partial and multi-value extractions as correct, because they may be fixable in post-processing.

9 FUTURE WORK

9.1 Better Synthesis for Long-tail Documents

Blueprint's program synthesis functionality is simple and very effective for working with document types which have a small-to-medium number of fairly-fixed layouts. However, the synthesis engine could be much better overall. For example, right now, if the dataset consists of documents having five distinct layouts (such as MIDV-2020), we require the user to recognize this fact, to create a `pick_best` node, and to populate it with synthesized `extract` nodes, one per layout. All of this could be automated to some degree.

Users have been successful building Blueprint programs for document classes having as many as 200 layouts (US driver's licenses, for example). Each layout is quick to build and straightforward to tune and debug, so this approach is not as onerous as it might sound. However, with better synthesis, which could heuristically group documents by layout and automatically generate constraints and the extraction tree structure with a single click, the development experience of this approach could become quite serviceable.

9.2 Integration with Deep Learning

As mentioned in Section 4, arbitrary parts of a Blueprint extraction tree could be replaced with deep-learning extraction models generating extractions for various parts of a document – for example, natural language text, for which we have not found Blueprint particularly well-suited. Similarly, we could implement predicates whose score functions invoke ML models, and we could use ML models in the entity generation step for tasks such as identifying person names or other semantic types.

10 CONCLUSION

We described the design and implementation of Blueprint, a declarative domain-specific language for document extraction, which provides building blocks for describing spatial, textual, semantic, and numerical relationships in documents. We further described Studio, a no-code IDE for Blueprint which provides helpful debugging and automatic program synthesis functionality.

We also compared Blueprint with the state-of-the-art deep-learning-based document extraction system LayoutLM [34] on an open dataset MIDV-2020 [6]. The comparison shows that Blueprint can achieve comparable extraction accuracy results with comparable development time requirements on appropriately-chosen datasets, while remaining interpretable and debuggable.

We have applied Blueprint extensively in commercial settings on various document types, such as bank checks, bills of lading, paystubs, etc. We described some real-life user experiences of building document extraction solutions using Blueprint and Studio. The source code of Blueprint and Studio is available at [1].

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their helpful comments. We thank all the software developers, sales engineers, and other users at Instabase, who contributed to the development of Blueprint.

REFERENCES

[1] 2021 (accessed 12-July-2022). Blueprint source code on GitHub. https://github.com/instabase/blueprint-oss.

[2] 2021 (accessed 12-July-2022). LayoutLM using the SROIE dataset. https://www.kaggle.com/urbikn/layoutlm-using-the-sroie-dataset/notebook.

[3] James Stuart Aitken. 2002. Learning Information Extraction Rules: An Inductive Logic Programming Approach. In *Proceedings of the 15th European Conference on Artificial Intelligence* (Lyon, France) *(ECAI'02)*. IOS Press, NLD, 355–359.

[4] José M. Alonso, Luis Magdalena, and Gil González-Rodríguez. 2009. Looking for a good fuzzy system interpretability index: An experimental approach. *International Journal of Approximate Reasoning* 51, 1 (Dec 2009), 115–134. https://doi.org/10.1016/j.ijar.2009.09.004

[5] Douglas E Appelt and Boyan Onyshkevych. 1998. *The common pattern specification language.* Technical Report. Sri international, Menlo Park CA, AI center.

[6] K.B. Bulatov, E.V. Emelianova, D.V. Tropin, N.S. Skoryukina, Y.S. Chernyshova, A.V. Sheshkus, S.A. Usilin, Z. Ming, J.-C. Burie, M.M. Luqman, and V.V. Arlazarov. 2022. MIDV-2020: a comprehensive benchmark dataset for identity document analysis. *Computer Optics* 46, 2 (Apr 2022), 252–270. https://doi.org/10.18287/2412-6179-co-1006

[7] Michael J Cafarella, Alon Halevy, Daisy Zhe Wang, Eugene Wu, and Yang Zhang. 2008. Webtables: exploring the power of tables on the web. *Proceedings of the VLDB Endowment* 1, 1 (2008), 538–549.

[8] Laura Chiticariu, Marina Danilevsky, Yunyao Li, Frederick Reiss, and Huaiyu Zhu. 2018. SystemT: Declarative Text Understanding for Enterprise. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 3 (Industry Papers)*. Association for Computational Linguistics, 76–83. https://doi.org/10.18653/v1/N18-3010

[9] Laura Chiticariu, Yunyao Li, and Frederick R. Reiss. 2013. Rule-Based Information Extraction is Dead! Long Live Rule-Based Information Extraction Systems!. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 827–832. https://aclanthology.org/D13-1079

[10] Hamish Cunningham, Diana Maynard, Kalina Bontcheva, and Valentin Tablan. 2002. GATE: an architecture for development of robust HLT applications. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*. 168–175.

[11] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *arXiv:1810.04805 [cs]* (May 2019). http://arxiv.org/abs/1810.04805

[12] Emilio Ferrara, Pasquale De Meo, Giacomo Fiumara, and Robert Baumgartner. 2014. Web data extraction, applications and techniques: A survey. *Knowledge-based systems* 70 (2014), 301–323.

[13] David Ferrucci and Adam Lally. 2004. UIMA: an architectural approach to unstructured information processing in the corporate research environment. *Natural Language Engineering* 10, 3-4 (2004), 327–348.

[14] Jaekyu Ha, R.M. Haralick, and I.T. Phillips. 1995. Document page decomposition by the bounding-box project. In *Proceedings of 3rd International Conference on Document Analysis and Recognition*, Vol. 2. 1119–1122 vol.2. https://doi.org/10.1109/ICDAR.1995.602115

[15] Jaekyu Ha, R. M. Haralick, and I. T. Phillips. 1995. Recursive X-Y cut using bounding boxes of connected components. In *Proceedings of the Third International Conference on Document Analysis and Recognition (Volume 2) - Volume 2 (ICDAR '95)*. 952.

[16] Zheng Huang, Kai Chen, Jianhua He, Xiang Bai, Dimosthenis Karatzas, Shijian Lu, and CV Jawahar. 2019. ICDAR2019 competition on scanned receipt ocr and information extraction. In *2019 International Conference on Document Analysis and Recognition (ICDAR)*. IEEE, 1516–1520.

[17] Eyke Hüllermeier. 2015. Does machine learning need fuzzy logic? *Fuzzy Sets and Systems* 281 (Dec 2015), 292–299. https://doi.org/10.1016/j.fss.2015.09.001

[18] Arun Iyer, Manohar Jonnalagedda, Suresh Parthasarathy, Arjun Radhakrishna, and Sriram K. Rajamani. 2019. Synthesis and machine learning for heterogeneous extraction. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 301–315. https://doi.org/10.1145/3314221.3322485

[19] Sanjeet Khaitan, Ganesh Ramakrishnan, Sachindra Joshi, and Anup Chalamalla. 2008. RAD: A Scalable Framework for Annotator Development. In *2008 IEEE 24th International Conference on Data Engineering*. 1624–1627. https://doi.org/10.1109/ICDE.2008.4497637

[20] Nicholas Kushmerick, Daniel S Weld, and Robert Doorenbos. 1997. Wrapper Induction for Information Extraction. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence, IJCAI, 1997*. 729–737.

[21] Alberto HF Laender, Berthier A Ribeiro-Neto, Altigran S Da Silva, and Juliana S Teixeira. 2002. A brief survey of web data extraction tools. *ACM Sigmod Record* 31, 2 (2002), 84–93.

[22] Xiaojing Liu, Feiyu Gao, Qiong Zhang, and Huasha Zhao. 2019. Graph Convolution for Multimodal Information Extraction from Visually Rich Documents. *arXiv:1903.11279 [cs]* (Mar 2019). http://arxiv.org/abs/1903.11279 paper-2019-naacl-liu-gcn.md.

[23] Bodhisattwa Majumder, Navneet Potti, Sandeep Tata, James B. Wendt, Qi Zhao, and Marc Najork. 2020. Representation Learning for Information Extraction from Form-like Documents. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics (ACL 2020)*. 6495–6504.

[24] Song Mao, Azriel Rosenfeld, and Tapas Kanungo. 2003. Document structure analysis algorithms: a literature survey. In *Document Recognition and Retrieval X*, Tapas Kanungo, Elisa H. Barney Smith, Jianying Hu, and Paul B. Kantor (Eds.), Vol. 5010. International Society for Optics and Photonics, SPIE, 197 – 207. https://doi.org/10.1117/12.476326

[25] Ion Muslea, Steve Minton, and Craig Knoblock. 1999. A hierarchical approach to wrapper induction. In *Proceedings of the third annual conference on Autonomous Agents - AGENTS '99*. ACM Press, 190–197. https://doi.org/10.1145/301136.301191

[26] L. O'Gorman. 1993. The document spectrum for page layout analysis. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 15, 11 (Nov 1993), 1162–1173. https://doi.org/10.1109/34.244677

[27] Antonio Polino, Razvan Pascanu, and Dan Alistarh. 2018. Model compression via distillation and quantization. *arXiv preprint arXiv:1802.05668* (2018).

[28] Mohammad Raza and Sumit Gulwani. 2018. Disjunctive program synthesis: A robust approach to programming by example. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 32. 1403–1412.

[29] Arnaud Sahuguet and Fabien Azavant. 2001. Building Intelligent Web Applications Using Lightweight Wrappers. *Data Knowl. Eng.* 36, 3 (mar 2001), 283–316. https://doi.org/10.1016/S0169-023X(00)00051-3

[30] Sunita Sarawagi. 2008. Information Extraction. *Found. Trends Databases* 1, 3 (mar 2008), 261–377. https://doi.org/10.1561/1900000003

[31] Warren Shen, AnHai Doan, Jeffrey F. Naughton, and Raghu Ramakrishnan. 2007. Declarative Information Extraction Using Datalog with Embedded Extraction Predicates. In *Proceedings of the 33rd International Conference on Very Large Data Bases* (Vienna, Austria) *(VLDB '07)*. VLDB Endowment, 1033–1044.

[32] A. Simon, J.-C. Pret, and A.P. Johnson. 1997. A fast algorithm for bottom-up document layout analysis. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 19, 3 (Mar 1997), 273–277. https://doi.org/10.1109/34.584106

[33] Ray Smith. 2007. An overview of the Tesseract OCR engine. In *Ninth international conference on document analysis and recognition (ICDAR 2007)*, Vol. 2. IEEE, 629–633.

[34] Yiheng Xu, Minghao Li, Lei Cui, Shaohan Huang, Furu Wei, and Ming Zhou. 2020. LayoutLM: Pre-training of Text and Layout for Document Image Understanding. *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (Aug 2020), 1192–1200. https://doi.org/10.1145/3394486.3403172