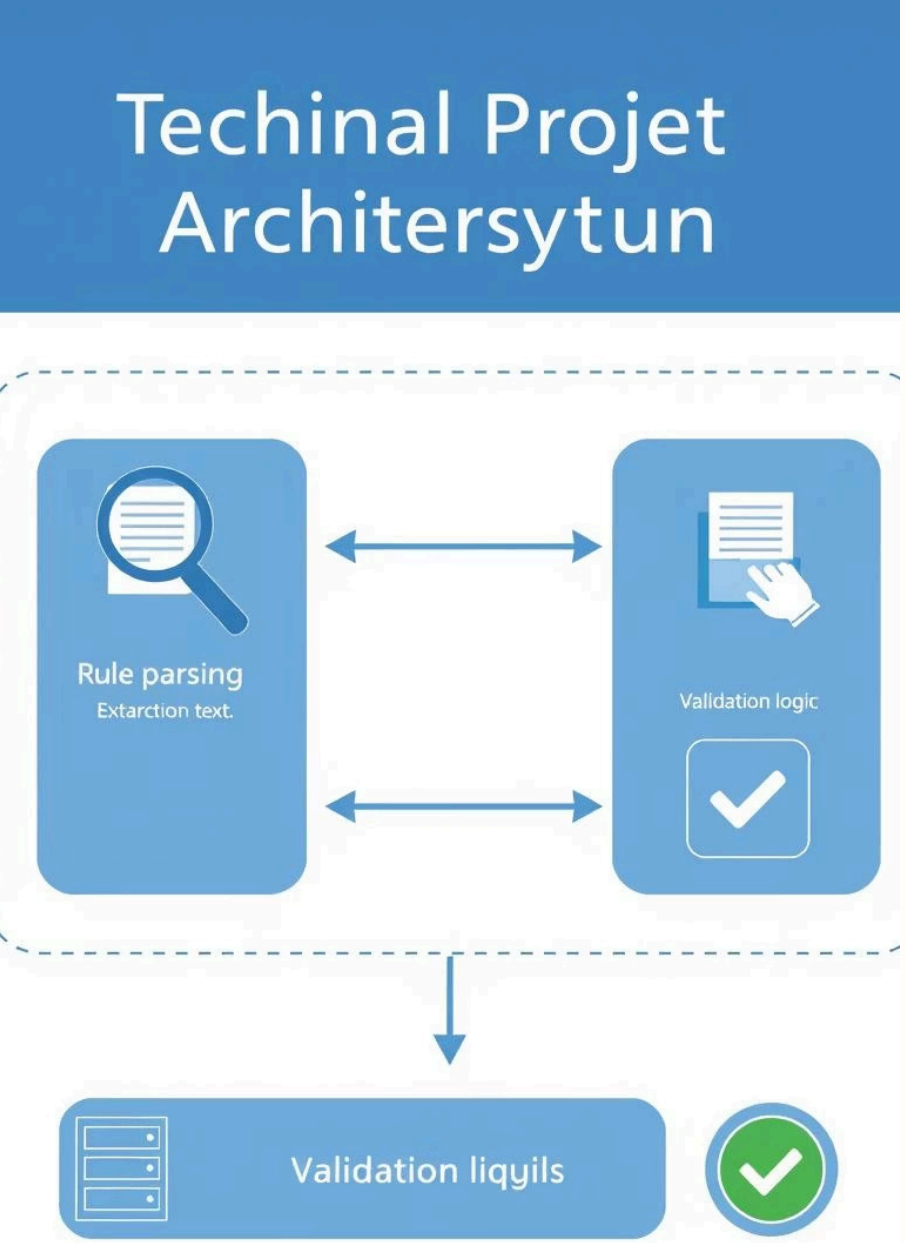


Detailed Project Explanation

This presentation provides a comprehensive overview of our project's architecture, focusing on three key modules: Rule Parsing, Document Extraction, and Validation Logic. Each module plays a critical role in transforming natural language rules into structured validation processes for document analysis.

by Shady Sakr

Rule Parsing

</>

Define JSON Schema

- category: one of money, date, time, text.

- condition: operator such as equals, greater_than, before_date.

- value: the expected threshold or text.

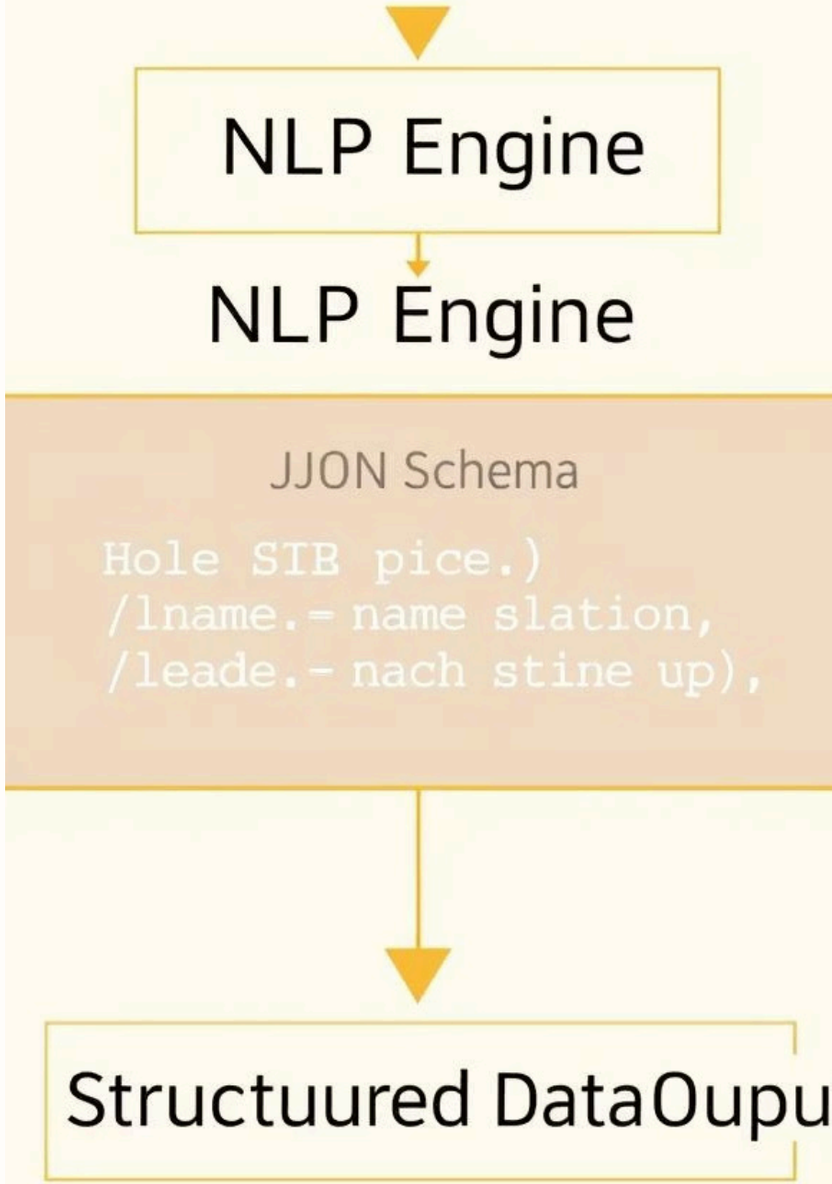
Prepare LLM Prompt

We craft a prompt template that instructs the LLM to return ONLY the JSON object. It includes strict rules for formats (e.g., YYYY-MM-DD for dates).

ƒ(x)

Implement parse_rule() Function

```
def parse_rule(rule_text):

# Build prompt

rule_prompt = f"...Rule to convert: '{rule_text}'..."

# Call Ollama CLI

response = run_ollama_command(rule_prompt)

# Extract JSON

return extract_json_from_response(response)
```



Rule Parsing Example

Example Prompt

"""

Convert this rule to JSON with EXACTLY these fields: category, condition, value.

RULE: "Invoice date must be after 2025-01-01"

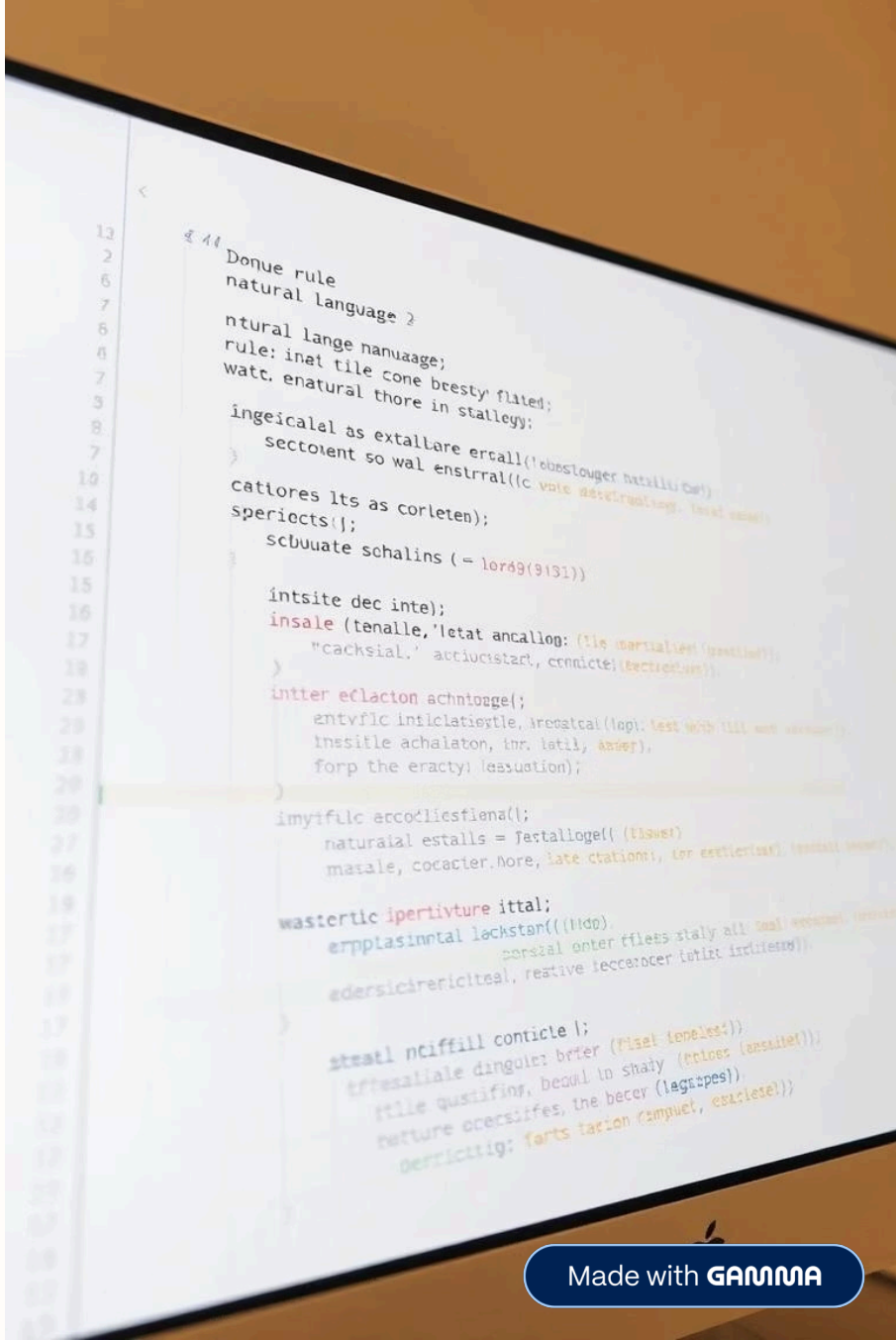
"""

Expected Output

The LLM will process this prompt and return a structured JSON object that contains the three required fields with appropriate values extracted from the natural language rule.

Benefits

This approach allows users to write rules in natural language while ensuring the system receives consistently structured data for validation processing.



Document Extraction

PDF Text Extraction

```
with pdfplumber.open(BytesIO(file_bytes)) as pdf:

text = []

for page in pdf.pages[:10]:

page_text = page.extract_text()

if page_text:

text.append(page_text.strip())

return '\n\n'.join(text)
```

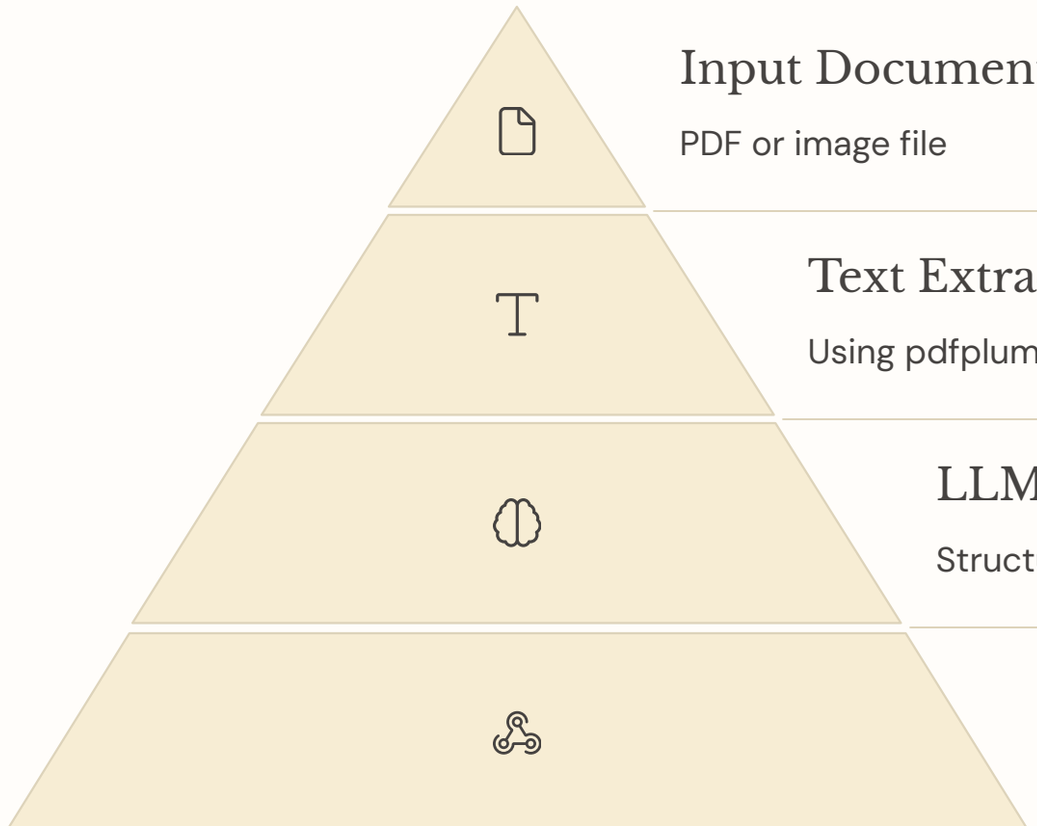
OCR on Images

```
image = Image.open(BytesIO(file_bytes))

text = pytesseract.image_to_string(image)

return text
```

Structured Data Extraction

We create a data_prompt similar to rule parsing, instructing the LLM to output JSON with keys money, date, and time, following strict formatting rules.

Document Extraction Implementation



Input Document

PDF or image file

Text Extraction

Using pdfplumber or pytesseract

LLM Processing

Structured extraction with prompts

JSON Output

Formatted data ready for validation

```
def extract_data_from_text(document_text):

data_prompt = f"...Extract JSON from document_text snippet..."

response = run_ollama_command(data_prompt)

return extract_json_from_response(response)
```

Validation Logic

Rule Input

Structured rule JSON from parsing module

Document Data

Extracted structured data from documents

Results

PASS/FAIL with actual vs expected values

Comparison

Category-specific validation logic

Validation Function Implementation

validate_rule() Function

```
def validate_rule(rule, data):

category = rule['category']

actual = data.get(category)

expected = rule['value']

# Route to comparison helper based on category

if category == 'money':

result = _compare_numbers(rule['condition'], float(actual), float(expected))

elif category in ['date', 'time']:

result = _compare_dates(rule['condition'], actual, expected)

else:

result = _compare_text(rule['condition'], actual, expected)

return {'status': 'PASS' if result else 'FAIL', 'actual': actual, 'expected': expected}
```

Comparison Helpers

```
# Numeric

def _compare_numbers(cond, a, e):

if cond == 'greater_than': return a > e

if cond == 'less_than': return a < e

# Date/Time

def _compare_dates(cond, a, e):

a_dt = parser.parse(a)

e_dt = parser.parse(e)

return {'before_date': a_dt < e_dt, 'after_date': a_dt > e_dt}[cond]

# Text

def _compare_text(cond, a, e):

a, e = a.lower(), e.lower()

return {'contains': e in a, 'equals': a == e}[cond]
```

Complete System Integration

3

Key Modules

Rule Parsing, Document Extraction, and Validation Logic working together

2

File Types

PDF and image processing capabilities for comprehensive document analysis

4

Data Categories

Money, date, time, and text validation for complete document coverage

The complete system integrates all three modules to provide a powerful document validation framework. Users can define rules in natural language, which are parsed into structured JSON. The system extracts relevant data from documents using appropriate techniques based on file type. Finally, the validation engine compares the extracted data against the rules, providing detailed results that include both the actual and expected values for each validation check.