Detailed Project Explanation

1. Rule Parsing

The rule parsing module converts user-friendly natural language rules into a structured JSON schema that the validation engine can process. Each rule is defined with three fields: **category**, **condition**, and **value**.

Step 1: 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.

Step 2: 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).

```
"""
Convert this rule to JSON with EXACTLY these fields: category, condition, value.
RULE: "Invoice date must be after 2025-01-01"
"""
```

Step 3: 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)
```

2. Document Extraction

This module handles both PDF text extraction and OCR on images, then leverages the LLM to structure the extracted text into JSON.

Step 1: 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)
```

Step 2: OCR on Images

```
image = Image.open(BytesIO(file_bytes))
text = pytesseract.image_to_string(image)
return text
```

Step 3: 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.

```
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)
```

3. Validation Logic

The validation engine compares extracted data with rules and reports PASS/FAIL. Detailed results include actual vs expected values.

Step 1: 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}
```

Step 2: 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]
```