



adaptTo()

Reimagining AEM Forms Rule Authoring with Coding Agents

Navneet Agarwal, Ravisanker E.

About Us



Navneet Agarwal

Software Engineer, Adobe
AEM Forms



Ravisanker E.

Software Engineer, Adobe
AEM Forms



Agenda

- Complexity of Enterprise Forms
- The Business Rule Authoring
- Our First Attempts to Generate Rules using Agents
- Rethinking the Grammar
- Building for Reliability
- Results, Demo & Takeaways

The Premise

An enterprise form is really an application

A single customer journey - a loan application, a health-plan enrollment - is not simple data capture. The complexity shows up across four dimensions :

1

Business Logic

2

Integration & Orchestration

3

Form State & Lifecycle

4

Compliance & Experience

Where The Effort Goes

Business rules are the real work

In AEM Forms, the business rules that drive dynamic behavior, calculations, state, and API calls account for the overwhelming majority of authoring effort on a real journey.



Patient Intake Form

FULL NAME *

DATE OF BIRTH *

dd/MM/YYYY



CONTACT NUMBER *

+1234567890

DO YOU HAVE ANY KNOWN ALLERGIES?

☐ Yes ☐ No

Submit



Visual Rule Editor

1

No-code visual authoring

Add dynamic form logic without scripting.

2

Predefined rule types

When, Show, Set Value Of, Validate, and more.

3

Manage & reuse rules

Reorder, copy-paste, enable/disable, nest AND/OR.

4

Drag and drop building

Drop in fields, functions, operators, events.

Visual Representation of the rule

Allergy Details - Visibility

? Help

CancelDone

Form Objects and Functions

+ Create API Integration

SHOW ▾

allergy_details

WHEN

do_you_have_any_known_allergies

is equal to

Yes

+ Add Condition

Delete

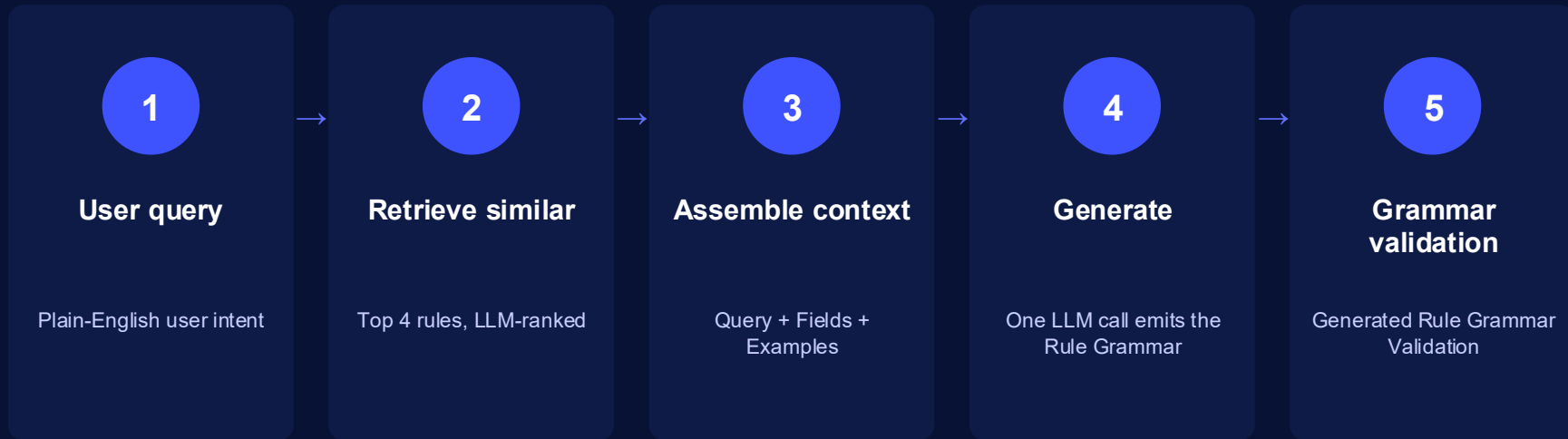
ELSE

Hide ▾

Grammar Representation of the Rule

```
{
  "fd:visible": [{
    "nodeName": "ROOT",
    "items": [{
      "nodeName": "STATEMENT",
      "choice": {
        "nodeName": "SHOW_EXPRESSION",
        "items": [
          { "nodeName": "AFCOMPONENT",
            "value": { "id": "$form.main_panel.allergy_details" } },
          { "nodeName": "When", "value": null },
          { "nodeName": "CONDITIONORALWAYS",
            "choice": {
              "nodeName": "COMPARISON_EXPRESSION",
              "items": [
                { "nodeName": "EXPRESSION",
                  "choice": { "nodeName": "COMPONENT",
                    "value": { "id": "$form.main_panel.has_allergies" } } },
                { "nodeName": "OPERATOR",
                  "choice": { "nodeName": "EQUALS_TO", "value": null } },
                { "nodeName": "EXPRESSION",
                  "choice": { "nodeName": "STRING_LITERAL", "value": "yes" } }
              ]
            }
          },
          { "nodeName": "Else", "value": null },
          { "nodeName": "DONOTHING_OR_HIDE",
            "choice": { "nodeName": "Hide", "value": null } }
        ]
      }
    ]
  }
],
  "enabled": true, "isValid": true}
```

First Attempt ; Retrieve examples, Generate the rule



Step 1 · User query

1 Query

2 Retrieve

3 Assemble

4 Generate

5 Validation

User Intent

“When the user selects yes in ‘do you have any known allergies’ then we should show the allergy details field”

Trigger Field

has_allergies

radio-group · “Do you have any known allergies?”

enum: [yes, no]

Target Field

allergy_details

text-input · “Allergy Details”

shown only when the trigger fires

Step 2 · Retrieve similar

1 Query

2 Retrieve

3 Assemble

4 Generate

5 Validation

1 WHEN field_0 changed AND field_0 == '<string>' → Show field_1

2 WHEN field_0 changed AND field_0 == '<string>' → Show field_1 · ELSE Hide field_1

3 WHEN field_0 changed AND field_0 == '<string>' → Show field_1, Show field_2

4 WHEN field_0 changed AND (field_0 == '<string>' OR ...) → Show id_1 · ELSE Hide + Reset

Each match arrives with its full **Rule** Grammar — a structural template to adapt, not text to copy.

Step 3 · Assemble context

1 Query

›

2 Retrieve

›

3 Assemble

›

4 Generate

›

5 Validation

System Message

- **Task**
- **Instructions**
- **Output schema**
- **Rule grammar description**
- **Examples** – the 4 retrieved rules, injected here

User Message

- **Form Definition** — 7 fields (names, paths, ids, enums)
- **Custom Functions**
- **Query** — the plain-English intent

Step 4 · Generate

1 Query

2 Retrieve

3 Assemble

4 Generate

5 Validation

```
...
[{"nodeName": "STATEMENT",
  "choice": {"nodeName": "EVENT_SCRIPTS", "items": [

    { "nodeName": "EVENT_CONDITION", "choice": {
      "nodeName": "EVENT_AND_COMPARISON", "items": [
        { "nodeName": "COMPONENT", "value": "$form....has_allergies" },
        { "nodeName": "OPERATOR", "choice": "EQUALS_TO" },
        { "nodeName": "STRING_LITERAL", "value": "yes" } ]}},
    ...
    { "nodeName": "Then" },
    { "nodeName": "SHOW_STATEMENT", "value": "$form....allergy_details" },
    ...
    { "nodeName": "Else" },
    { "nodeName": "HIDE_STATEMENT", "value": "$form....allergy_details" }
  ]
}]
...
```

Step 5 · Validation

1 Query



2 Retrieve



3 Assemble



4 Generate



5 Validation

What It Checks

- Valid AST Syntax for Rule Grammar
- Did not refer any non-existing external functions
- Malformed output is caught here



Validation Pass

Everything had to fit in one window, and it didn't.

The Input Squeeze

- Verbose grammar + every form field + several full example rules
- The prompt was huge before the query even mattered
- More context or examples = better quality, but less room to work

The Output Limit

- The model had to emit the entire deeply-nested AST, often hundreds of lines long.
- Large rules hit the completion-token limit and truncated mid-generation
- We were trading input richness against output completeness but were turning out to be bottlenecks

Identifying the pattern

When condition (AND / OR ...)

Then action (action ...)

Else action *optional*

What This Approach Targets

- ✓ **When ... Then...**
- ✓ **When ... Then ... Else ...**

Not Yet Covered

- ✗ Set Value of \$field to 'a'
- ✗ custom functions, math

In this Form's Terms

WHEN has_allergies = 'yes'

THEN Show allergy_details

Creating the catalog

Conditions

is changed · is equal to · has selected ·
is empty · contains · starts with · is not
empty ...

Actions

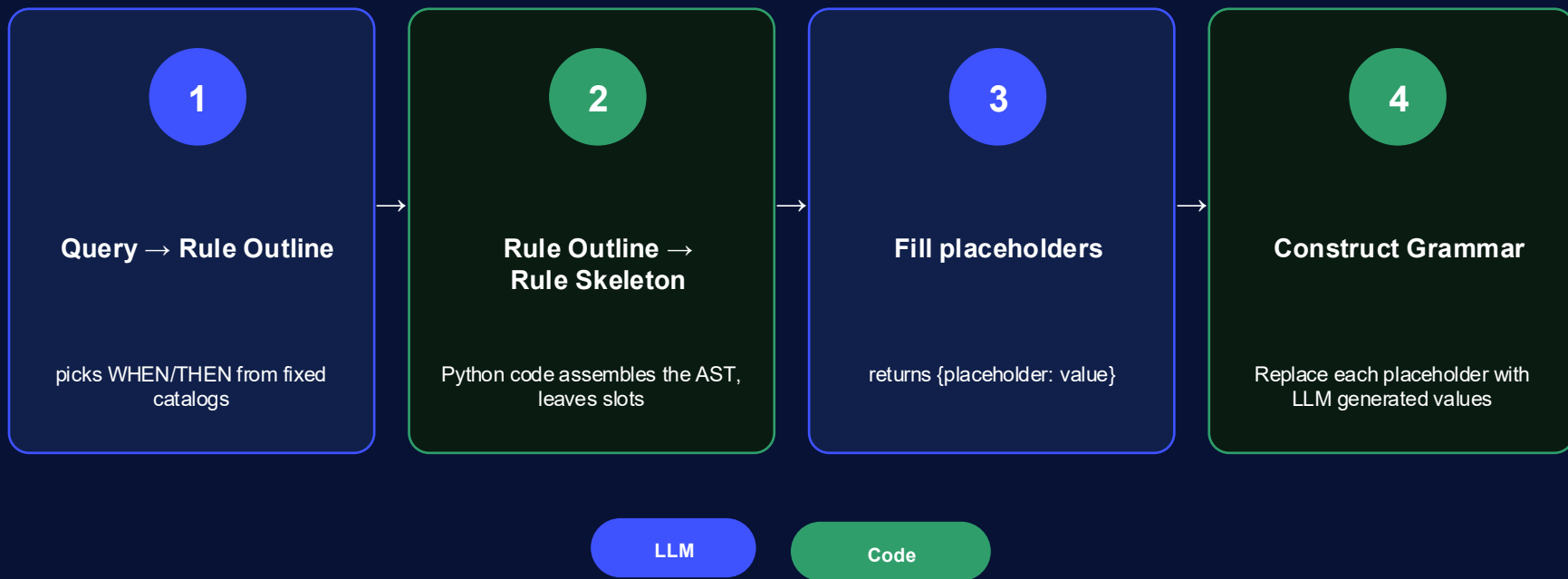
Show · Hide · Enable · Disable · Set value
Submit · Reset · Set focus · Clear value ·
...

Each Catalog Item Contains

- Index
- Label – English Representation of condition/action
- Grammar Block – The AST fragment of that action/condition

Second attempt: Let the LLM fill-up placeholders; let code build the grammar

Same rule — **show Allergy Details when the patient answers 'yes'** — but the LLM never writes grammar.



Step 1 · Query → Rule Outline

1 Outline
LLM

2 Skeleton
code

3 Fill
LLM

4 Construct Grammar
code

Context Passed To LLM

- The query
- **CONDITION catalog** — the WHEN menu
- **ACTION catalog** — the THEN / ELSE menu
- a few worked examples

It Generates — indices only

["WHEN", 9, "THEN", 15]

= condition **#9** (is equal to)

+ action **#15** (Show)

It writes no JSON — it just points at menu items.

Step 2 · Rule Outline → Rule Skeleton

1 Summary
LLM

2 Skeleton
code

3 Fill
LLM

4 Construct Grammar
code

Assembled Skeleton

```
{ "nodeName": "ROOT", "items": [{ "STATEMENT":  
  "EVENT_SCRIPTS", "items": [  
    EVENT_CONDITION → AND_COMPARISON:  
      COMPONENT.id    <placeholder_1>  
      COMPONENT.name  <placeholder_2>  
      OPERATOR        EQUALS_TO  
      STRING_LITERAL  <placeholder_3>  
    Then → SHOW_STATEMENT → AFCOMPONENT:  
      id <placeholder_4>  name <placeholder_7>  
      displayName <placeholder_5>  
      displayPath <placeholder_6>  
      parent <placeholder_8>  
  ]}] , "eventName": "Value Commit" }
```

No LLM Here

- Blocks #9 + #15 are pre-authored AST fragments
- Python stitches them into ROOT → STATEMENT → ...
- Real values unknown → **8 slots** left as placeholders

Step 3 · Fill placeholders

1 Summary
LLM

2 Skeleton
code

3 Fill
LLM

4 Construct Grammar
code

Context Passed To LLM

- The query
- Instructions
- Output schema
- All the form fields — names, ids, fd:path, enums
- The Step-2 skeleton — 8 `<placeholder>` slots
- A few worked examples

LLM Generates

{ placeholder: value }

```
1 → $form.main_panel.has_allergies
3 → yes
4 → $form.main_panel.allergy_details
5 → Allergy Details
7 → allergy_details
... 8 slots in all
```

Step 4 · Resolve path

1 Outline
LLM

2 Skeleton
code

3 Fill
LLM

4 Construct Grammar
code

What It Does

- Replaces all the placeholders in the skeleton with the LLM generated values
- Valid AST Syntax for Rule Grammar
- Save the created Rule



Validation Pass

It raised the floor, but not the ceiling.

What It Fixed

- Output truncation errors are gone, because code now assembles the structure.
- The LLM does only what it's reliable at figuring out the next token
- A classifier routes clean WHEN/THEN queries here automatically
- Accuracy improved by 15%

What it didn't cover

- WHEN...THEN and WHEN...THEN...ELSE only
- No SET VALUE, no custom functions, no math
- Anything out of scope fell back to RAG, hitting the same wall as before.
- Still, ~50% of the customer corpus was reachable

Is the grammar the culprit here?

One visibility rule in v1 `fd:visible` — and almost none of it is the rule.

Current grammar *green = the rule* · white = scaffolding

```
{ "fd:visible": [{
  "nodeName": "ROOT", "items": [{
    "nodeName": "STATEMENT", "choice": {
      "nodeName": "SHOW_EXPRESSION", "items": [
        { "nodeName": "AFCOMPONENT",
          "value": { "id": "...allergy_details" } },
        { "nodeName": "When", "value": null },
        { "nodeName": "CONDITIONORALWAYS", "choice": {
          "nodeName": "COMPARISON_EXPRESSION", "items": [
            { "nodeName": "EXPRESSION", "choice": {
              "nodeName": "COMPONENT",
              "value": { "id": "...has_allergies" } }},
            { "nodeName": "OPERATOR", "choice": {
              "nodeName": "EQUALS_TO", "value": null }},
            { "nodeName": "EXPRESSION", "choice": {
              "nodeName": "STRING_LITERAL",
              "value": "yes" } } ] }},
          { "nodeName": "Else", "value": null },
          { "nodeName": "DONOTHING_OR_HIDE",
            "choice": { "nodeName": "Hide" } } ] }
      } } ],
  "enabled": true, "isValid": true }
```

1

Built for UI

EditorUI coupled, and difficult for agents to generate or understand.

2

Mostly dead weight

Verbose representation involving deep nesting.

3

The fix

Optimise for Agents

The new grammar, written to be generated.

Optimized grammar:

```
"fd:rules": {  
  "fd:prop-visible": [{  
    "condition": "has_allergies = 'yes'",  
    "expression": "true()",  
    "else": [{ "expression": "false()" }],  
  }]  
}
```

WHAT EACH PART SAYS

- **fd:prop-visible** — property to update
- **condition** — when the rule triggers
- **expression** — value while true → visible
- **action** — in case of events

- smaller AST and reduced UI coupling
- clearer intent and fewer generation failure points
- easier validation loop
- lower token consumption

Grammar Generation Workflow



Deterministic Grammar Validation Phases

1

Structure

AST conforms to the grammar's production rules grammar only, no form data. An instant syntax check.

2

Scope

Component & function references resolve against the form; bad refs return closest-match alternatives.

3

Context

Rule type matches its storage key.
See the example output.

MACHINE-READABLE ERRORS

```
{
  "valid": false,
  "errors": [
    {
      "code": "CONTEXT_STATEMENT_MISMATCH",
      "path": "fd:click[0]",
      "message": "Event envelope must not
        contain \"expression\";
        use \"actions\" instead."
    }
  ],
  "warnings": []
}
```

→ *valid: false* · a Context (check 3) failure
code = stable key · **path** pins fd:click[0].
message names the fix: use "actions".



adaptTo()

Demo

Accuracy on real customer rules

Measured on **anonymized, customer-authored rules**. Rules that used custom functions were excluded from the dataset.

Templating + RAG

2nd attempt



60%

Coding-first grammar

the new grammar



95%

+35 points — errors fall from 40% to 5%.

Three Layers of Confidence

01 VALIDATORS

Is the generated output valid?

Grammar correctness

02 ESLINT RULES

Does it follow Product guidance?

Enforce product best practices

03 UNIT + E2E

Does the use case work?

Functional correctness

Takeaways

- 1 **Design representations around the model's strengths.**
- 2 **Agents still need deterministic engineering guardrails.**

Thank You

Open for questions