

Decision Management Community Challenge: Benchmark “Medical Services”

Solution using Progress Corticon.js

Summary

In response to the decision service implementation challenge described by DM Community [here](#), we have used Progress Corticon.js to create a JavaScript decision service hosted on AWS as a Lambda function behind an API gateway. It incorporates all of the 16,369 rules specified within the prompt’s decision table, and is easily callable using an importable Postman collection. The included collection has a preconfigured input which mirrors the test cases provided in the prompt.

Public link to Postman collection - <https://www.getpostman.com/collections/0a82a7dbcf72b47aab6>

Walkthrough

The initial decision table, formatted as a CSV file, presents an array of 16,379 ‘rule’ combinations applicable to medical claims. The rules are made up of varying types of data across 11 different attributes: **Plan, Place Of Service, Service Type, Group Size, In Network, Is Covered, Effective Period Start, Effective Period End, Covered in Full, Copay, and Coinsurance.**

To simplify the importation of these rules into Corticon, we start by transposing the data, reformatting the table from having each data attribute underneath a column header specifying the attribute name (figure 1), into a table with each data attribute next to a row header specifying the attribute name (figure 2).

	A	B	C	D	E	F	G	H	I	J	K
1	Plan	Place Of Service	Service Type	Group Size	In Network	Is Covered	Effective Period Start	Effective Period End	Covered in Full	Copay	Coinsurance
2	PL123	Inpatient	dentalAccidental	L	Y	Y	1/1/2015	12/31/2023	N	Copay Minimal	N
3	PL123	Outpatient	dentalAccidental	L	Y	Y	1/1/2015	12/31/2023	N	Copay Minimal	N
4	PL123	Office	dentalAccidental	L	Y	Y	1/1/2015	12/31/2023	N	Copay Minimal	N
5	PL123	Inpatient	dentalAccidental	L	N	Y	1/1/2015	12/31/2023	N	N	Y
6	PL123	Outpatient	dentalAccidental	L	N	Y	1/1/2015	12/31/2023	N	N	Y
7	PL123	Office	dentalAccidental	L	N	Y	1/1/2015	12/31/2023	N	N	Y
8	PL123	Inpatient	acupuncture	L	Y	N	1/1/2015	12/31/2023	N	N	N
9	PL123	Office	acupuncture	L	Y	N	1/1/2015	12/31/2023	N	N	N
10	PL123	Outpatient	acupuncture	L	Y	N	1/1/2015	12/31/2023	N	N	N

Figure 1: Claim Data Table's format prior to transposing (records cropped from bottom of table)

Plan	PL123	PL123	PL123	PL123
Place Of Service	Inpatient	Outpatient	Office	Inpatient
Service Type	dentalAccidental	dentalAccidental	dentalAccidental	dentalAccidental
Group Size	L	L	L	L
In Network	Y	Y	Y	N
Is Covered	Y	Y	Y	Y
Effective Period Start	1/1/2015	1/1/2015	1/1/2015	1/1/2015
Effective Period End	12/31/2023	12/31/2023	12/31/2023	12/31/2023
Covered in Full	N	N	N	N
Copay	Copay Minimal	Copay Minimal	Copay Minimal	N
Coinsurance	N	N	N	Y

Figure 2: Claim data table after transposing (records cropped from right side of table)

Building the Rule Vocabulary in Corticon.js Studio

The first step in building a decision service in Corticon.js is to build a ‘rule vocabulary’ using Corticon.js Studio. Based upon the name of each data attribute (the headers in the leftmost column of figure 2), we have various attributes specific to our primary entity, which we have called an **Encounter**. The vocabulary can be defined from scratch, or automatically generated from a JSON schema. Our completed rule vocabulary is shown in figure 3.

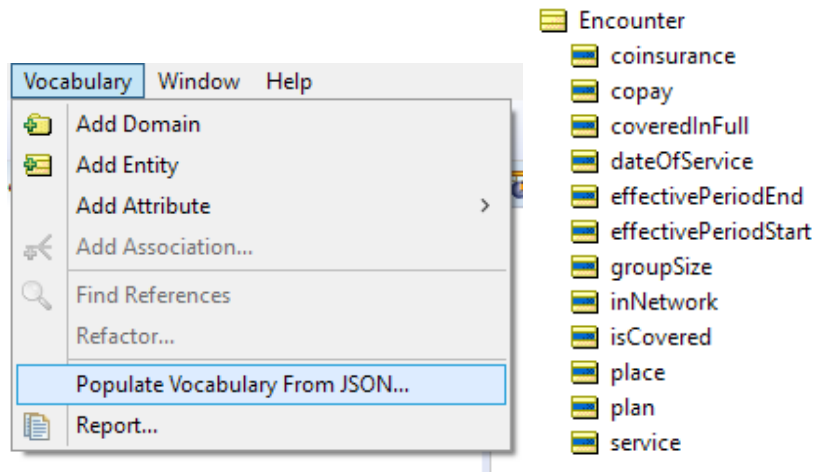


Figure 3: Creating the rule vocabulary

Building the Rule Sheets in Corticon.js Studio

Corticon.js Studio, the rule authoring interface for Corticon.js, is built for use by business analysts, and as such, Corticon's rule sheets are easy to grasp for users comfortable with excel spreadsheets. The rule vocabulary, built in the previous step, will now effectively serve as a bucket of available business terms from which the end user will drag and drop attributes of the entity (the encounter) onto a decision table called a Rule Sheet. As shown in figure 4, each column of the rule sheet serves as a discrete rule, each with any number of conditions which when met produce any number of actions.

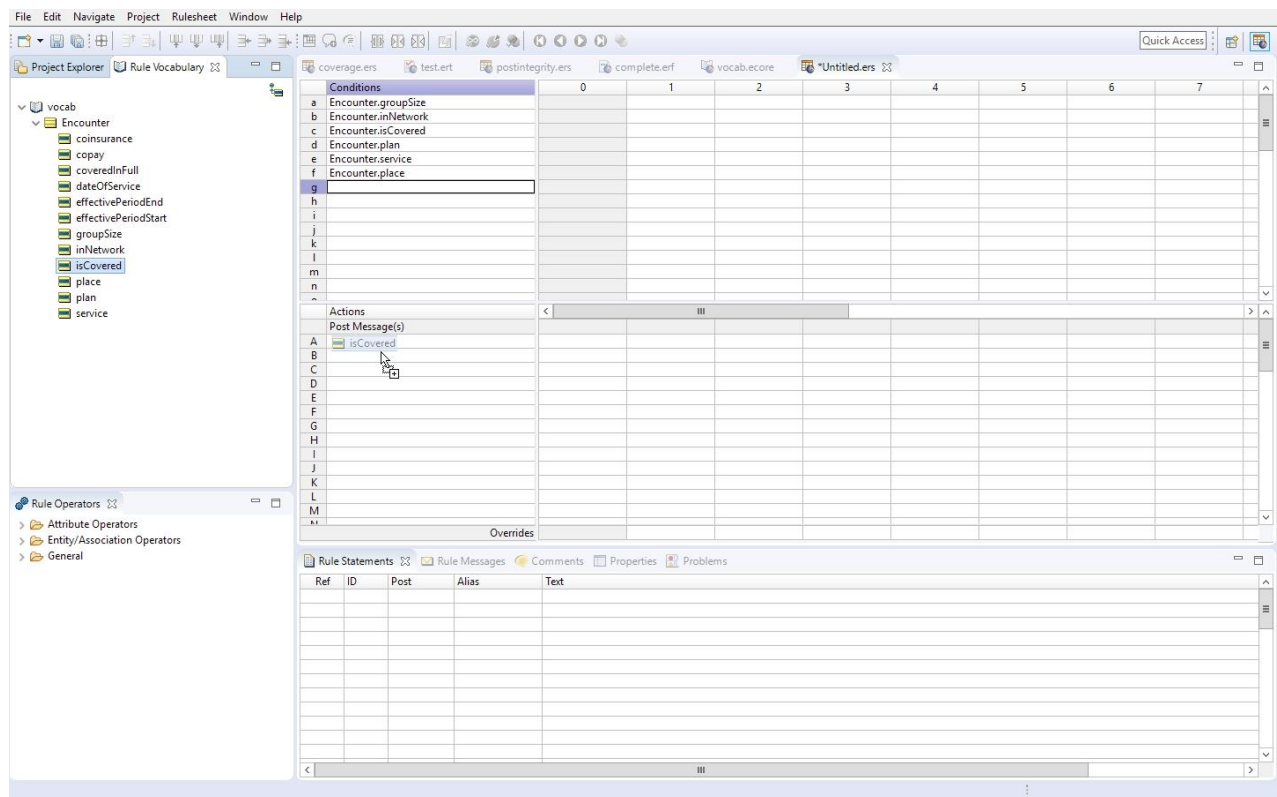


Figure 4: Defining rules in a rule sheet

Once our Rule Sheet's parameters are organized into 'condition' attributes and 'action' attributes, we can copy and paste the rules within the CSV decision table into the appropriate row (figure 5).

	Conditions	0	1	2	3	4	5	
a	Encounter.place		'Inpatient'	'Outpatient'	'Office'	'Inpatient'	'Outpatient'	'Ot
b	Encounter.service		'dentalAccidental'	'dentalAccidental'	'dentalAccidental'	'dentalAccidental'	'dentalAccident...	'dentalA
c	Encounter.groupSize		'L'	'L'	'L'	'L'	'L'	
d	Encounter.inNetwork		'Y'	'Y'	'Y'	'N'	'N'	
e	Encounter.isCovered		'Y'	'Y'	'Y'	'Y'	'Y'	
f	Encounter.plan		'PL123'	'PL123'	'PL123'	'PL123'	'PL123'	'Pl
g								
h								
i								
j								
k								
l								
m								
n								
o								
	Actions	<						>
	Post Message(s)							
A	Encounter.coveredInFull		'N'	'N'	'N'	'N'	'N'	
B	Encounter.copay		'Copay Minimal'	'Copay Minimal'	'Copay Minimal'	'N'	'N'	
C	Encounter.coinsurance		'N'	'N'	'N'	'Y'	'Y'	
D								

Figure 5: Rule sheet after pasting in values from the decision table in Excel

When building decision services with Corticon.js, the decision's rules can be authored in just one Rule Sheet, or any number of Rule Sheets chained together into a 'Rule Flow'. For this scenario, we have broken out the date specific rules (date of service, coverage period start, coverage period end) into a separate rule sheet which precedes the previously created Rule Sheet in our Rule Flow. Our Rule Flow is shown in figure 6.

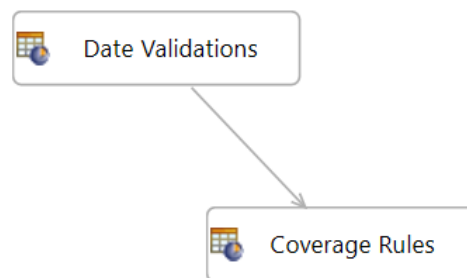


Figure 6 Rule Flow which will be compiled into a JavaScript decision service

In this date validation rule sheet, shown in figure 7, we're assigning the coverage period dates based upon the encounter's plan (columns 1 and 2). In columns 3 and 4, we're defining actions for when the date of service took place within the coverage start and end dates, and for when it does not take place during the coverage start and end dates.

Conditions		0	1	2	3	4	5	
a	Encounter.plan		{'PL012', 'PL059', 'PL100', 'PL123', 'PL455', 'PL499', 'PL555', 'PL666', 'PL720', 'PL733'}	{'PL222', 'PL511', 'PL534', 'PL622', 'PL711', 'PL788', 'PL823', 'PL888', 'PL898', 'PL911'}	-	-		
b	Encounter.dateOfService in(Encounter.effectivePeriodStart.. Encounter.effectivePeriodEnd)		-	-	T	F		
c								
Actions		<	III					
Post Message(s)			✉	✉	✉	✉		
A	Encounter.effectivePeriodStart		'Jan 1, 2015'	'Jan 1, 2016'				
B	Encounter.effectivePeriodEnd		'Dec 31, 2023'	'Dec 31, 2023'				
C	Encounter.isCovered				'Y'	'N'		
D								
E								
F								
G								
H								
I								
J								
K								
Overrides								

Rule Statements

✕

✉ Rule Messages

✕

💬 Comments

✕

📄 Properties

✕

🚫 Problems

✕

Ref	ID	Post	Alias	Text
{1, 2}		Info	Encounter	Plan name {Encounter.plan} has an effective period start date of {Encounter.effectivePeriodStart} and end date of {Encounter.effectivePeriodEnd}
3		Info	Encounter	Encounter took place during coverage period
4		Violation	Encounter	Encounter did not take place during coverage period

Figure 7: Date validation rule sheet

Additionally, we are defining rule statements for each rule that we have created as part of this writeup, though we have removed them from the final submission as they were beyond the scope of the challenge. These rule statements will be sent back to the calling application as natural language descriptions of the rules that have fired. Before testing our Rule Sheets or Rule Flows with some sample data, it's worth seeing if we can optimize these rules at all to improve processing performance and validate the logical integrity of the rules. Corticon can automatically perform tasks like this at a click of button.

First, we suspected that there could be some redundancies in these rules, meaning they could be expressed in fewer discrete rules than the quantity in the excel decision table. In the rule sheet where we pasted in the values from the excel table, which resulted in in 16,369 columns of rules, we click the 'compress rules' button. This compresses that number substantially down to a total of 1,010 columns of rules.

Compression does not alter the Rulesheet's logic; it simply affects the way the rules appear in the Rulesheet – the number of columns, values sets in the columns, etc. Compression also streamlines rule execution by ensuring that no rules are processed more than necessary. A screenshot of this rule sheet post-compression is shown in figure 8.

Conditions	0	1	2	3	4	5
a Encounter.groupSize		'L'	'L'	'L'	'L'	'L'
b Encounter.inNetwork		'Y'	'N'	'Y'	'N'	'N'
c Encounter.isCovered		'Y'	'Y'	'N'	'N'	'N'
d Encounter.plan		'PL123'	{'PL100', 'PL123'}	'PL123'	'PL123'	'PL123'
e Encounter.service		{'adultImmunizations', 'dentalAccidental'}	'dentalAccidental'	{'acupuncture', 'childbirthClasses', 'christianScienceServices', 'clinicalTrials', 'contraceptiveDevices', 'diabeticDrugs', 'electiveAbortion', 'gymReimbursement', 'routineFootCare', 'transgenderSurgery', 'visionContactLens', 'visionExam', 'visionFrame', 'visionLens', 'weightManagementProgram'}	{'acupuncture', 'childbirthClasses', 'christianScienceServices', 'clinicalTrials', 'contraceptiveDevices', 'diabeticDrugs', 'electiveAbortion', 'gymReimbursement', 'routineFootCare', 'transgenderSurgery', 'visionContactLens', 'visionExam', 'visionFrame', 'visionLens', 'weightManagementProgram'}	{'adultImmunizations', 'childImmunizations', 'clinicalTrials', 'clinicServices', 'contrastAgents', 'diabeticEducati...
f Encounter.place		{'Inpatient', 'Office', 'Outpatient'}	{'Inpatient', 'Office', 'Outpatient'}	{'Inpatient', 'Office', 'Outpatient'}	'Inpatient'	{'Office', 'Outpatient'}
g						
h						
i						
j						
k						
l						
Actions	< >					
Post Message(s)						
A Encounter.coveredInFull		'N'	'N'	'N'	'N'	'N'
B Encounter.copay		'Copay Minimal'	'N'	'N'	'N'	'N'
Overrides						

Rule Statements

Rule Messages

Comments

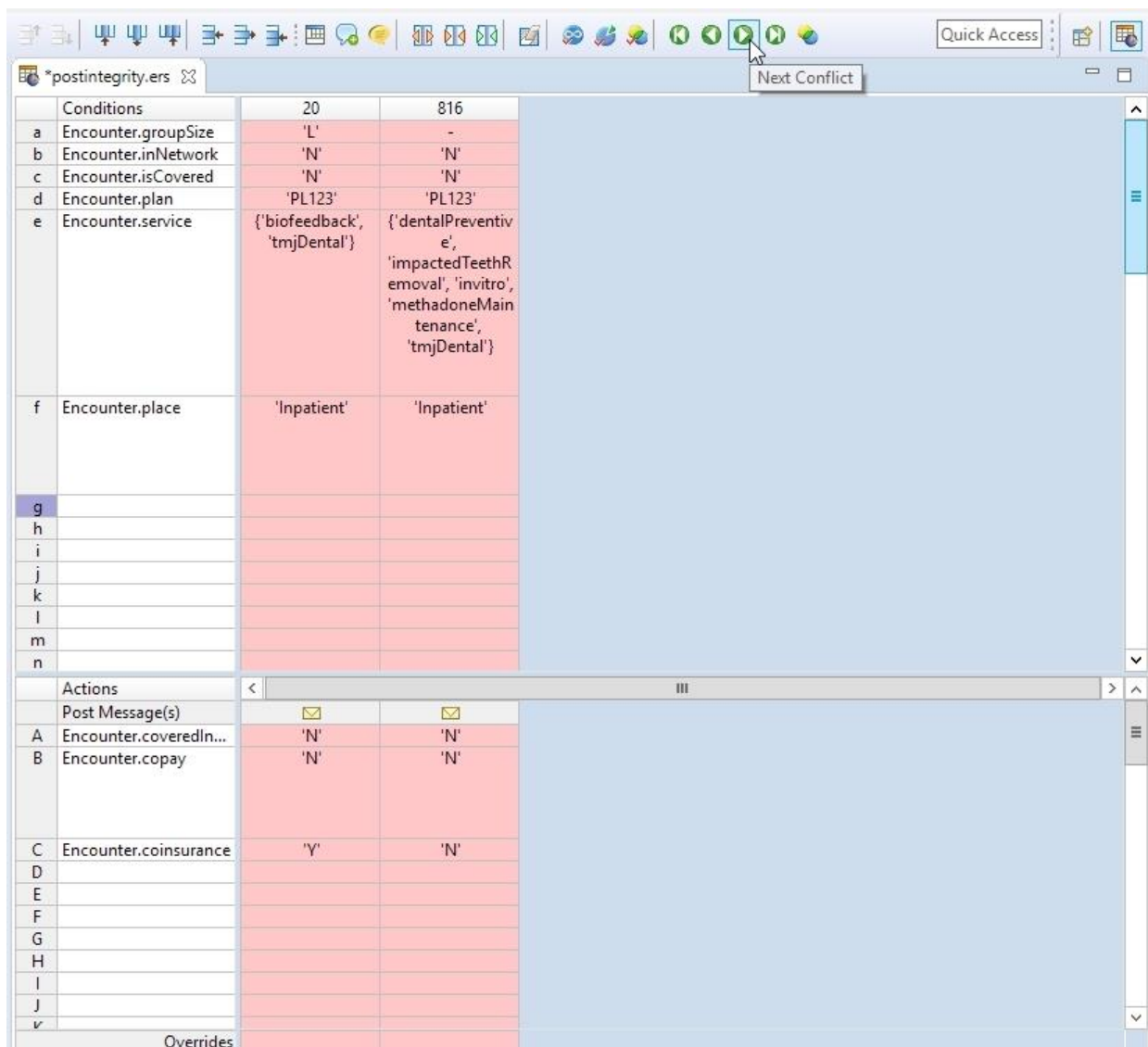
Properties

Problems

Ref	ID	Post	Alias	Text
1:1010		Info	Encounter	Under the [[Encounter.plan]] plan for [[Encounter.place]] encounters for [[Encounter.service]] service with group size [[Encounter.groupSize]] with in network value of [[Encounter.inNetwork]] and is covered value of [[Encounter.isCovered]]--covered in full flag set to [[Encounter.coveredInFull]], copay set to [[Encounter.copay]], and coinsurance set to [[Encounter.coinsurance]]

Figure 8: Rule Sheet after compressing down from 16,369 rules to 1,010 rules

Corticon provides various other logical checks—one for logical loops, one for rule conflicts (overlapping conditions producing different actions), and rule completeness (unaccounted-for scenarios). For example, the rule completeness check on this same rule sheet reveals scenarios (figure 9) which are plausible based upon the rules we have thus far. In a real world scenario, we might want to define actions for each of these plausible sets of conditions.



Conditions	20	816
a Encounter.groupSize	'L'	-
b Encounter.inNetwork	'N'	'N'
c Encounter.isCovered	'N'	'N'
d Encounter.plan	'PL123'	'PL123'
e Encounter.service	{'biofeedback', 'tmjDental'}	{'dentalPreventive', 'impactedTeethRemoval', 'in vitro', 'methadoneMaintenance', 'tmjDental'}
f Encounter.place	'Inpatient'	'Inpatient'
g		
h		
i		
j		
k		
l		
m		
n		

Actions	20	816
Post Message(s)		
A Encounter.coveredIn...	'N'	'N'
B Encounter.copay	'N'	'N'
C Encounter.coinsurance	'Y'	'N'
D		
E		
F		
G		
H		
I		
J		
Overrides		

Figure 11: Using the conflict filter, Corticon has isolated the two rules that are in conflict with one another

At any point when we want to test the behavior of an individual rule sheet or entire rule flow, we can use a Rule Test to run test data against the rules. As with the process to define the rule sheet conditions and actions, we drag and drop from our rule vocabulary onto the 'input' column of the rule test, and then enter some test data into the applicable attributes. For this exercise, we've replicated the pre and post-test data provided by the challenge prompt. Shown in figure 10, we've entered the unprocessed test data into the 'input' column, and expected completed test results into the 'expected' column.

Input		Output	Expected
Encounter [1]	<ul style="list-style-type: none"> coinsurance [null] copay [null] coveredInFull [null] dateOfService [2019-03-25] effectivePeriodEnd effectivePeriodStart groupSize [L] inNetwork [Y] isCovered [Y] place [Inpatient] plan [PL123] service [adultImmunizations] 		<ul style="list-style-type: none"> coinsurance [N] copay [Copay Minimal] coveredInFull [N] dateOfService [2019-03-25] effectivePeriodEnd [2023-12-31] effectivePeriodStart [2015-01-01] groupSize [L] inNetwork [Y] isCovered [Y] place [Inpatient] plan [PL123] service [adultImmunizations]
Encounter [2]	<ul style="list-style-type: none"> coinsurance [null] copay [null] coveredInFull [null] dateOfService [2018-03-25] effectivePeriodEnd effectivePeriodStart groupSize [L] inNetwork [N] isCovered [Y] place [Office] plan [PL123] service [allergyTesting] 		<ul style="list-style-type: none"> coinsurance [Y] copay [N] coveredInFull [N] dateOfService [2018-03-25] effectivePeriodEnd [2023-12-31] effectivePeriodStart [2015-01-01] groupSize [L] inNetwork [N] isCovered [Y] place [Office] plan [PL123] service [allergyTesting]
Encounter [3]	<ul style="list-style-type: none"> coinsurance [null] copay [null] coveredInFull [null] dateOfService [2014-03-25] effectivePeriodEnd effectivePeriodStart groupSize [L] inNetwork [Y] isCovered [Y] place [Inpatient] 		<ul style="list-style-type: none"> coinsurance [null] copay [null] coveredInFull [null] dateOfService [2014-03-25] effectivePeriodEnd [2023-12-31] effectivePeriodStart [2015-01-01] groupSize [L] inNetwork [Y] isCovered [N] place [Inpatient]

Figure 12: Rule Test Pre-Execution

When he hit the run button, as shown in figure 12, Corticon.js will simulate the behavior of the rules at runtime, and highlight whether any of the output did not align with the expected values. We also can see that Corticon has posted rule messages—what we call the ‘rule statements’ when they are returned from a decision service’s execution—for the rules that have fired, and only the rules that have fired. When we defined the rule statements, we used curly brackets where we wanted to use the actual value of each entity’s attribute (e.g. *Under the [{Encounter.plan}] plan for [{Encounter.place}]*). When returned as posted rule messages, these have been populated with the actual value applicable to that record.

Note that in the API endpoint used in our submission, we have removed the rule messages as they were beyond the scope of the challenge—contact the Progress Corticon team to see the version of the API that sends back rule messages.

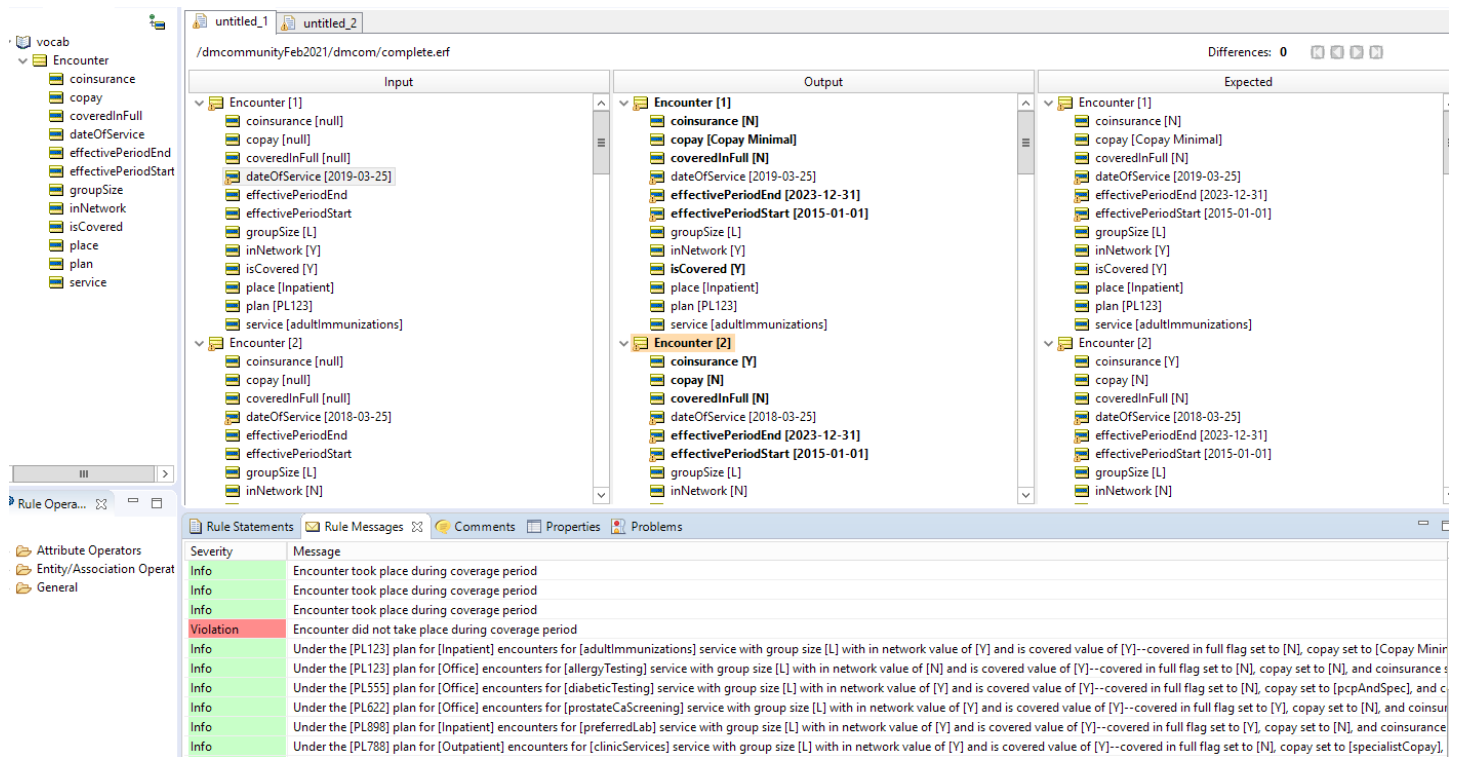


Figure 13: Rule Test Post execution

Now that we're ready to transform these rules into a cloud microservice, we return to our ruleflow, and select our deployment target (figure 14). The output will be a zip file that we can upload to AWS console as a Lambda function (figure 15).

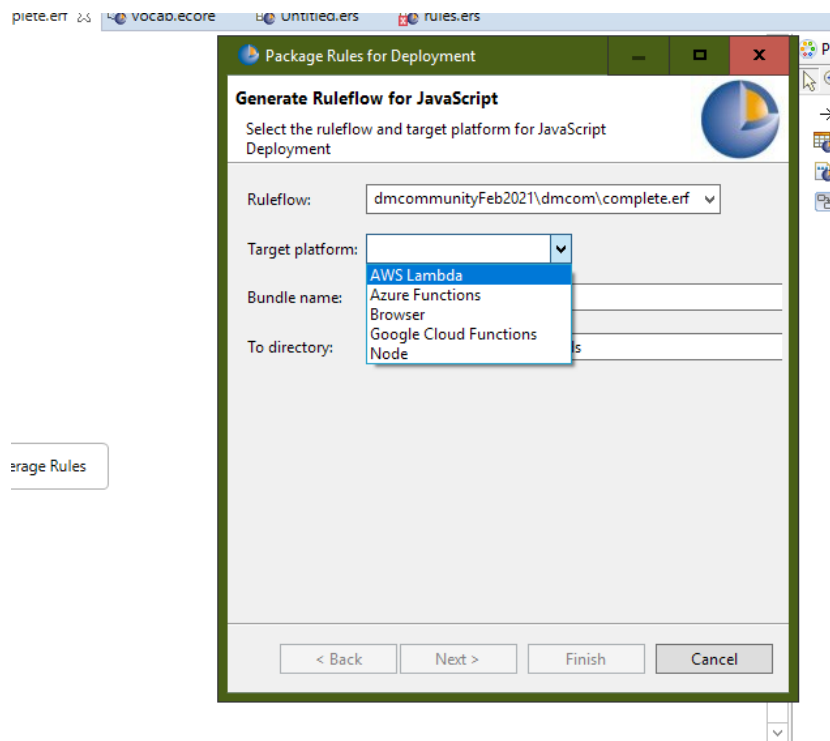


Figure 14: compiling rules into a JavaScript bundle preconfigured for AWS Lambda

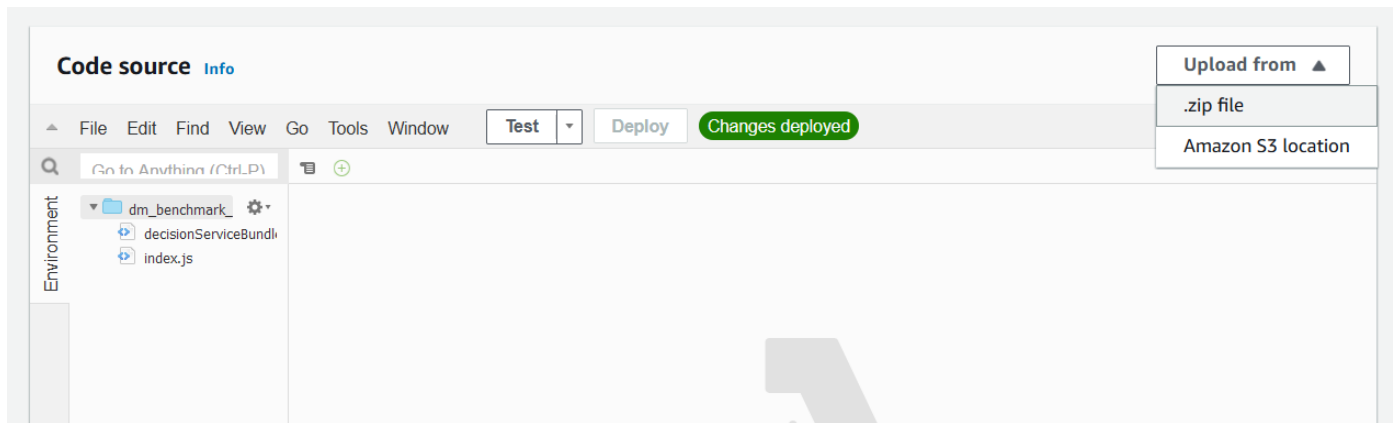


Figure 15: Uploading zip file containing Corticon.js JavaScript bundle to AWS console

Now to test the decision service, we can use the existing the ‘inputs’ from the rule test that we ran in Corticon.js Studio—no need to do the same exercise twice. Figure 16 shows how when we just right click in the ‘input’ column, we can choose to export that input data as JSON directly to our clipboard, which can then be pasted into the AWS console tester or Postman.

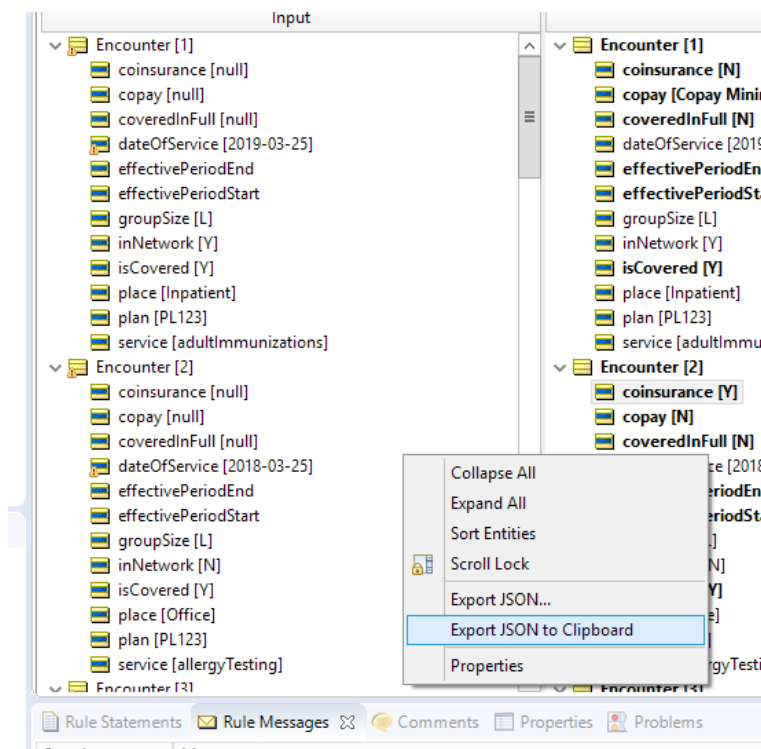


Figure 16: Exporting the JSON input to our clipboard, which can then be pasted into the AWS console tester, Postman, or any other API testing tool

Testing with Postman

Using the below link, you can import a Postman collection pre-configured with JSON test data that mirrors the provided test cases. This data is ready to be sent to the URL endpoint of our deployed solution, which is populated in the URL field. Note that we have not set up caching at any level, so the first request will take longer than all subsequent requests.

The response from the Corticon.js decision service is based off of the rules, so input data may be changed to validate all combinations of rules have been implemented. As part of the response, we also append the time it takes to execute just the decision service function without the time added from transit to and from the AWS server. This value is shown in the output field `decisionServiceExecDurationInMs`.

Public link to Postman collection - <https://www.getpostman.com/collections/0a82a7dbcfc72b47aab6>