

Rules-as-Code: Lessons learned from a Code For Canada Fellowship

This document summarizes some lessons learned by Team Babel while working in the Rules-as-Code (RaC) space. This is based off of our experience in building a policy difference engine (PDE) for the maternity benefits calculation as well as participating in a collaborative Rules-as-Code sprint focusing on Motor Vehicle Operator Hours of Work Regulations.

We will begin with a description of the two projects and how a RaC engine fits into both. This is followed by an in-depth discussion on the properties and requirements of a RaC engine. We will then look at some technologies and architectures we worked with in order to meet this definition of the RaC engine. Throughout these discussions, challenges that we faced and lessons that we learned will be highlighted.

Part I: Project Descriptions

Maternity Benefits PDE

The PDE is a prototype for a data-driven tool that measures the impact of a change to the maternity benefits calculation formula. There are many factors involved in the calculation, but one of the most important starting points is the average weekly income of an applicant. As a simple example, let's suppose we have an applicant with an average weekly income of \$1000. The calculation proceeds as follows:

- Take **55%** of the average weekly income (\$550)
- Compare that value with the maximum weekly amount of **\$595** (550 is less, so keep that value)
- Multiply that by the maximum number of weeks allowed for maternity, which is **15** ($15 \times 550 = 8250$)

While there are many changes that could be made to the maternity benefits calculation, we limited our scope to updating the three values shown above: The percentage of average weekly income, the maximum weekly amount, and the number of weeks. Suppose we changed these values as follows:

- Increase the percentage from 55% to 60%
- Decrease the maximum weekly amount from \$595 to \$500
- Increase the number of weeks from 15 to 16

If we run this calculation again, this applicant ends up with:

- 60% of 1000 = 600
- Compare 600 to max of 500 (use 500, since 600 is greater than 500)
- \$500 x 16 weeks = \$8000

So this proposed change causes the person to lose money (\$250). This is just one example, but we want to know the impact on the whole population. Furthermore, we want to identify any groups that are disproportionately affected. For example, suppose we have the average weekly income of 10000 applicants. We run our proposed change against each of them, and we find that overall, 8000 of them end up gaining money, and 2000 end up losing money. Depending on your policy objectives, this might initially look positive - 80% of applicants gain money. Now suppose we zoom in on some demographic properties of our population, and we find that - for example - 95% of women between the ages of 30-35 who live in Alberta all end up losing money. This would be an example of a proposed change that disproportionately affects a certain group. The goal of the PDE is to show how we can use data to bring the impact of proposed changes such as this to light. The PDE makes no judgment about whether a policy change is good or bad, it simply performs the calculation on the data that is available to it, leaving the interpretation open to the policy experts.

The PDE is composed of 4 key components:

- Rules engine: This is where the rules for the maternity benefit calculation are stored. We give the engine the information about the applicant and the rule specification, and it returns the amount to which the individual is entitled.
- Data: The PDE is a data-driven tool. It requires individual data for running the calculations, including income data and demographic data. Note that care must be taken when working with real data, as privacy/confidentiality is a priority.
- Simulation Engine: This component brings together the rules and the data. The simulation engine loads all of the data (e.g. our 10000 applicants) into the rules engine and gathers all of the results
- User interface: The user of the PDE requires an interface for entering their proposed change, as well as visualizing the results of the simulation


The PDE works as follows:

- The user proposes a change via the user interface, specifying values for the percentage, max weekly amount, and number of weeks
- The user interface will pass this request onto the simulation engine
- The simulation engine will reach into its data storage and load the dataset (each data point representing an applicant)
- The simulation engine then iterates over these individuals and passes their data to the rules engine. It does this twice for each person. It first sends the person with the original rule, so that we can know how much this person is entitled to *before* the change. It then sends the person with the proposed change to the rule, so we can know how much they are entitled to *after* the change.
- The rules engine calculates these entitlement amounts and sends the results back to the simulation engine.
- Once all results have been calculated for each person, the results are sent back to the user interface for visualization and analysis.

Web interface, where a user can propose a change to the maternity benefits entitlement calculation

Test Site

You cannot apply for services or benefits through this test site. Parts of this site may not work and will change.

Government of Canada

Gouvernement du Canada

[Français](#)

[Home](#) [New Simulation](#)

[Canada.ca](#) > [Service Canada Labs](#)

Policy Difference Engine - Form

The Policy Difference Engine (PDE) is a data-driven tool that can help inform policy research and policy design. It measures the impact of a proposed change to a government legislation, regulation and policy. It allows the user to enter in a proposed change, and a simulation of that change is then run against a sample population. The results of the simulation can then be analyzed to drive insights. Policy researchers are one of the main users of PDE.

[▶ Learn more about the Policy Difference Engine](#)

Propose a policy adjustment

These values are extracted from the Maternity Benefits entitlement calculation, found in the EI Act. Change any or all of these values and run a simulation to see the potential impact. Remember, this is a sample population of individuals, and the calculations are approximate.

Maximum Weekly Amount

500

Percentage of Average Income

60

Number of Eligible Weeks

16

Reset Fields

Run Simulation

Contact us

Departments and agencies

Public service and military

News

Treaties, laws and regulations


Government-wide reporting

Prime Minister

How government works

Open government

Social media • Mobile applications • Terms and conditions • Privacy



Simulation Results part 1: Shows overall results of the change

Simulation Results

You've changed some values in the Maternity Benefits calculation. We ran your proposed changes on a population of sample individuals. This page allows you to explore the aggregated results of your simulation to show which groups might be impacted and how. In this way, the PDE promotes a data-driven approach to policy-making.

Your Changes

Parameter	Original Value	Proposed Change
Maximum Weekly Amount	\$595.00	\$500.00
Percentage of Average Income	55%	60%
Number of Eligible Weeks	15	16

Overall Results

Sample Size	1000
Percent Change in Cost	1.6%
Average Entitlement Change	\$104.43
Average Percent Change	1.6%

People Who Gained Money









Percent of Sample that Gained Money	60.5%
Average Entitlement Gain	\$707.12
Average Percent Gain	13.8%

People Who Lost Money









Percent of Sample that Lost Money	39.5%
Average Entitlement Lost	-\$818.68
Average Percent Lost	-9.3%

Simulation results part 2: Shows aggregate results of the change, by various demographic properties

Aggregation By Age

Age 	Sample Size 	Sample Percent 	Average Entitlement Change 	Average Percent Change 	Percent that Gained 	Percent that Lost 	Percent Unchanged 
0 - 20	176	17%	\$8.21	0.1%	52.8%	47.2%	0.0%
20 - 25	213	21%	\$134.89	2.1%	62.4%	37.6%	0.0%
25 - 30	207	20%	\$131.00	2.0%	63.3%	36.7%	0.0%
30 - 35	218	21%	\$114.81	1.7%	60.6%	39.4%	0.0%
35 - 40	186	18%	\$118.84	1.8%	62.4%	37.6%	0.0%
40+	0	0%	\$0.00	0.0%	0.0%	0.0%	0.0%

Aggregation By Province

Province 	Sample Size 	Sample Percent 	Average Entitlement Change 	Average Percent Change 	Percent that Gained 	Percent that Lost 	Percent Unchanged 
Alberta	132	13%	\$193.81	3.0%	67.4%	32.6%	0.0%
British Columbia	116	11%	\$61.38	0.9%	56.0%	44.0%	0.0%
Manitoba	48	4%	\$11.06	0.2%	56.2%	43.8%	0.0%
Newfoundland and Labrador	40	4%	\$61.76	0.9%	55.0%	45.0%	0.0%
Northwest Territories	40	4%	\$48.22	0.7%	55.0%	45.0%	0.0%
Nouveau Brunswick	32	3%	\$73.71	1.2%	62.5%	37.5%	0.0%
Nova Scotia	46	4%	\$196.96	3.1%	65.2%	34.8%	0.0%
Nunavut	29	2%	\$385.77	6.2%	75.9%	24.1%	0.0%
Ontario	317	31%	\$68.59	1.0%	58.7%	41.3%	0.0%
Prince Edward Island	43	4%	\$9.90	0.1%	51.2%	48.8%	0.0%
Quebec	85	8%	\$243.49	3.9%	70.6%	29.4%	0.0%
Saskatchewan	38	3%	\$95.57	1.4%	60.5%	39.5%	0.0%
Yukon	34	3%	-\$68.01	-1.0%	50.0%	50.0%	0.0%

Aggregation By Education









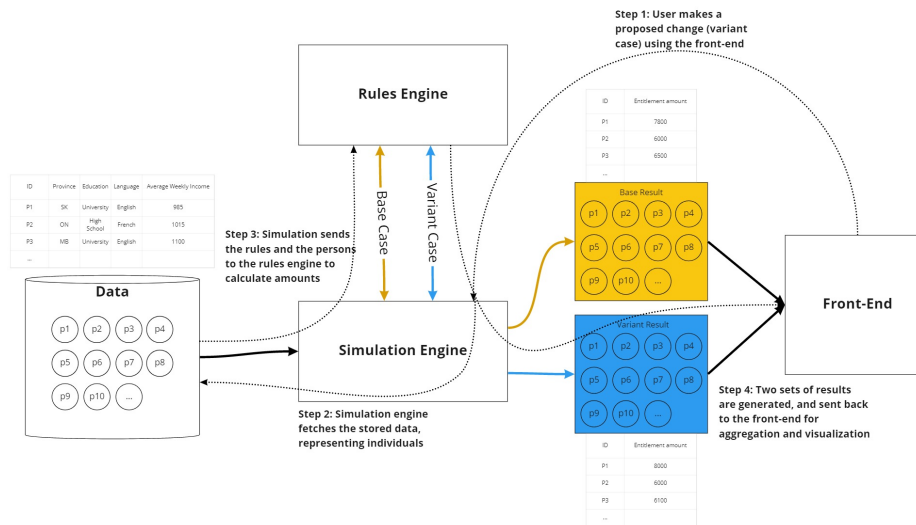
Education 	Sample Size 	Sample Percent 	Average Entitlement Change 	Average Percent Change 	Percent that Gained 	Percent that Lost 	Percent Unchanged 
Apprenticeship	240	24%	\$107.74	1.6%	60.8%	39.2%	0.0%
College	185	18%	\$131.01	2.0%	62.7%	37.3%	0.0%
Other	49	4%	\$108.82	1.7%	63.3%	36.7%	0.0%
Primary - Elementary	161	16%	-\$19.54	-0.3%	52.8%	47.2%	0.0%
Secondary - High School	161	16%	\$184.36	2.9%	65.2%	34.8%	0.0%
University	204	20%	\$110.12	1.7%	59.8%	40.2%	0.0%

Diagram of how the 4 main components of the PDE interact with each other



The PDE heavily relies on a precise and accurate encoding of the maternity benefit calculation rules. If the rules are not encoded properly, then the results are essentially meaningless, and even *detrimental* if taken out of context. The rules in the PDE have been coded with the help of maternity benefits policy experts, but they are still incomplete. We emphasize that the PDE is in prototype stage and only serves to show what could be possible if we have a reliably encoded rule set as well as access to real maternity benefit application data.

Motor Vehicle Regulations RaC Sprint

The Motor Vehicle Operator Hours of Work Regulations (MVOHWR) RaC sprint was a collaborative project between ESDC, the Canada School of Public Service, the Department of Justice, and the Community of Federal Regulators. The goal of this project was to measure the impact of a potential change to the Motor Vehicle Operators Hours of Work Regulations (https://laws-lois.justice.gc.ca/eng/regulations/C.R.C.,_c._990/index.html). This rule specifies how much overtime hours motor vehicle operators are entitled to. There are some basic steps for calculating this value, but the complexity arises when these operators work under multiple classifications - for example, someone can work as a city vehicle operator and a highway vehicle operator in the same week. These classifications have different rules for how overtime is calculated, so when someone works both, the calculation becomes more complex.

After some initial discussion on the goals of the sprint, we largely ended up using the same model that we used for the Maternity Benefits PDE: The rules are encoded in the rules engine, it is powered by relevant data, a simulation engine joins the data with the rules engine, and a user interface allows someone to propose a change and visualize results. We see the same 4 components that we saw with the PDE, with different details:

- Rules engine: This system encodes the rules for calculating the number of overtime hours that a motor vehicle operator is entitled to, given the work schedule
- Data: The data in this case is represented by weekly schedules of vehicle operators
- Simulation engine: This performs the same conceptual function as it did for the PDE. It retrieves the data and sends it to the rules engine
- User interface: This is where the user can enter a potential schedule, propose changes, have the request sent to the simulation engine, where the simulation is run. The user interface can then request the results of the proposed change so that they can be displayed to the user.

User interface: The user can enter a mock schedule, which will serve as the single data point for the simulation. Below they enter the proposed change to the rule. When the user clicks “Run”, the schedule will be sent to the rules engine twice. Once with the ‘Existing regulation’ and once with the ‘Possible change’. Each will result in a different number of overtime hours.

Canada

Home

Test Prototype

Documentation

Policy Difference Engine: Motor Vehicle Operator - Hours of Work Regulations

Schedule of hours of work

Enter a sample schedule of hours of work that you'd like to test. Please mark any holidays. This simulation assumes no hours were worked on a holiday.

Day	CMVO Hours	HMVO Hours	Other Hours	Holiday	Totals
Mon	<input type="text" value="8"/>	<input type="text" value="4"/>	<input type="text" value="0"/>	<input type="checkbox"/>	12
Tue	<input type="text" value="4"/>	<input type="text" value="6"/>	<input type="text" value="2"/>	<input type="checkbox"/>	12
Wed	<input type="text" value="8"/>	<input type="text" value="0"/>	<input type="text" value="4"/>	<input type="checkbox"/>	12
Thu	<input type="text" value="4"/>	<input type="text" value="4"/>	<input type="text" value="4"/>	<input type="checkbox"/>	12
Fri	<input type="text" value="4"/>	<input type="text" value="3"/>	<input type="text" value="2"/>	<input type="checkbox"/>	9
Sat	<input type="text" value="0"/>	<input type="text" value="0"/>	<input type="text" value="5"/>	<input type="checkbox"/>	5
Sun	<input type="text" value="0"/>	<input type="text" value="0"/>	<input type="text" value="0"/>	<input type="checkbox"/>	0
Total	28	17	17		62

Simulation Cases

The first column are key parts of the existing regulation. The second column is the existing rules. The third column is where you can enter the possible changes you'd like to test.

Rule Property	Existing Regulation	Possible Change
Standard CMVO Daily Hours	<input type="text" value="9"/>	<input type="text" value="8"/>
Standard CMVO Weekly Hours	<input type="text" value="45"/>	<input type="text" value="40"/>
Standard Other Daily Hours	<input type="text" value="8"/>	<input type="text" value="8"/>
Standard Other Weekly Hours	<input type="text" value="40"/>	<input type="text" value="40"/>
Standard Highway Weekly Hours	<input type="text" value="60"/>	<input type="text" value="55"/>
Standard Highway Reduction Hours	<input type="text" value="10"/>	<input type="text" value="10"/>

Run

2021 © Employment and Social Development Canada

[Back to top](#)

Results page: Here we see how the proposed change can result in a different number of entitled overtime hours for the motor vehicle operator

Simulation Results

	Existing Regulation	Possible Change
Total Hours OT	3	7

Schedule of hours of work

Day	CMVO Hours	HMVO Hours	Other Hours	Holiday	Totals
Mon	8	4	0	False	12
Tue	4	6	2	False	12
Wed	8	0	4	False	12
Thu	4	4	4	False	12
Fri	4	3	2	False	9
Sat	0	0	5	False	5
Sun	0	0	0	False	0
Totals	28	17	17		62

Simulation Cases

Rule Property	Existing Regulation	Possible Change
Standard CMVO Daily Hours	9	8
Standard CMVO Weekly Hours	45	40
Standard Other Daily Hours	8	8
Standard Other Weekly Hours	40	40
Standard Highway Weekly Hours	60	55
Standard Highway Reduction Hours	10	10
Total Hours OT	3	7

[Modify Simulation](#)

Comparisons

A key difference between these two systems is that the maternity benefits PDE uses a large set of data, which the simulation engine will send to the rules engine. In the Motor Vehicle system, we only use a *single* data point, which is the schedule that is input by the user. The system is flexible in that it *could* support simulating many data points, but we did not have access to real data for this project, so we simply made the data another piece of user input.

We can compare these two systems on the complexity of the encoded rules themselves. Both cases involved taking a written government rule and attempting to convert it to a specification that machines can understand, i.e. computer code. This is an inherently difficult process since the written rules were not made with machine encoding in mind. The written rules may contain ambiguities and assumptions that might be reasonable for humans to make, but cannot be made by computers, since computers deal only with very precise instructions. These instructions need to be fundamentally expressed using math and logic. The conversion process is very much a manual one at this point and for the foreseeable future. There has been a small amount of work done on extracting logical rules from human-written rules, but we are far away from automated capabilities. The complexity of the written rules themselves differed greatly, which also had a direct impact on the complexity involved in the translation process. The maternity benefits rules are very much tied up with the rest of the Employment Insurance Act, making it difficult to disentangle the pieces required for a PDE. This is an ongoing challenge. The motor vehicle regulation, however, is a relatively short document which is not heavily tangled up with other government rules. This makes it easier to draw clear lines around the specific written language that must be encoded.

As mentioned, without an accurate encoding of the rules, the results of these simulation systems are meaningless. This accuracy becomes a key feature for a simulation engine. For the maternity benefits, we sought access to policy experts, but found this to be one of the main challenges of the fellowship. We had to connect many dots and follow many leads trying to find the right experts, and even when we did, we found that they were very busy, and it was (understandably) difficult to get continuous buy-in from them. With the Motor vehicles project, on the other hand, there was a lot of preparation and stakeholder engagement organized by higher authorities long before we had actually started working on the project. The policy experts were involved from the beginning and were specifically assigned to collaborate on the project. This made it a much more collaborative effort between policy and technical experts, which led to a

much higher certainty on the precision of the encoded rules. The policy experts were working alongside the technical experts, resolving ambiguities, stepping through test scenarios, answering questions for clarification. This process helped us to build a more accurate system, and it also helped the policy experts themselves gain a new perspective on the rules that they are so familiar with. It is very important to have buy-in from policy experts on projects like this. Since this is a relatively new experiment, there isn't a culture of close collaboration between tech and policy in this particular area. Therefore, this collaboration requires planning and oversight from higher levels of the organizations involved.

Part II: Rules as Code

Now that we have some context behind the projects we worked on, we can have a more in-depth look at the notion of Rules as Code. We've established that these systems require some form of coded rules in order to function properly. Speaking generally, this is nothing novel. Every software application requires "coded rules", and every stakeholder wants those coded rules to fulfill their business requirements. So what are the differences between these traditional systems and systems that involve "government" rules, such as policy, regulation, and legislation. Over the course of the fellowship, we've identified several characteristics that deserve emphasis when discussing these rule systems in a government context.

The first characteristic has already been mentioned - and that is the **collaborative** nature of the requirements between technical and policy experts. If you give the written rules to ten different development teams, you might end up with ten different versions of the rule, none of which may be correct. By having policy experts in on the process from the beginning, we can ensure that the rule is as accurate as possible.

Another important characteristic is **reusability**. Many software systems strive to be reusable. A company may want a reusable API that could be used as the backend for both a web application and a mobile application. In the government context, this reusability is even more important, because these rules affect many different people, and there may be many different ways that people could interact with them. For maternity benefits, an applicant wants to know how much they are entitled to. They may have preferences for how they would like to get this information. For example they may prefer to go through a call centre where a service agent is

using a computer program to determine the amount, or they may prefer to use an online calculator. Businesses may want to make use of an HR tool that calculates the overtime employees are entitled to to ensure that they are being compliant. The point here is that there are many different use cases for a particular government rule. It would be a waste of time to have a different development team rewrite this code for each new application. It's much more efficient to have a single system that is reusable, that all of these applications can make use of.

Building on this idea of reusability is the idea of **transparency**. Since these are the rules that determine eligibility and entitlement amounts for many people, these should be open for viewing to the public. This can help build trust in the digital systems as well as subject the system to scrutiny so that it can be improved.

A question that arises with these rules is “how do we know when they are complete?” At what point do we say the rules are fully encoded and we feel confident having them incorporated into larger production-level systems? This should again be a very collaborative effort between tech and policy. Policy experts are crucial in making this determination, and a good way to measure the accuracy of the system is through **testability**. Test-driven development is a well-established practice in software development and it bears emphasis in this context. If we can come up with a rigorous set of test scenarios together with the policy experts, and our code passes all of the tests, then we can say the code is, in a sense, complete. Note that tests won't magically be able to find bugs in the code and they aren't foolproof, but they are a very important line of defense for ensuring integrity and completeness of the code. This is another area where access to data becomes very important. Real data can be used to come up with and validate test cases.

This list of properties may not be exhaustive, but they are the ones that we've determined are important to emphasize when dealing with government rules, and we've been using the presence of these properties to distinguish between traditional “coded rules” and “rules as code”.

At this point, we've identified the need for a rules engine in these systems, and we've emphasized some characteristics that this engine should have. The key challenge is to translate our written rules into such a system. We've mentioned that the maternity benefits process was significantly more difficult than the motor vehicle process, due to both the size of the rule to be

encoded and the engagement from policy experts. We will now go discuss the different strategies that we took for the translation process.

Maternity Benefits PDE

In order to make this a collaborative effort between tech and policy, we initially sought to set up ongoing meetings with maternity policy experts so that we could iteratively improve our encoding of the maternity benefits rule. This involved both asking around to our close contacts to see if they could put us in touch with anyone, and also the less traditional route of simply looking up maternity policy experts online and attempting to reach out to them. Both of these had mixed levels of success and occasionally even led to the same people. We spoke with many people who gave us small pieces of a larger picture. The difficulty at the beginning was compounded with the fact that we were still trying to concretely define our problem space. So we initially had a few different goals for these early conversations, such as understanding the current approach, discovering if there were any systems/teams in place that already existed in the problem space, and grasping the intricacies of the EI eligibility and entitlement calculations. Over the course of this, we learned that there are certain communication channels that are in place within these government organizations, and if you do not go through these channels, it's unlikely you will get in touch with the right people. Unfortunately these channels can take quite a bit of time to get you in touch with someone, and then you may get passed along to someone else. Sometimes we would hit complete dead ends where we might learn of a potential expert, but never hear back from them.

While we searched for these experts, we made use of the close contacts that we did have access to that had varying levels of experience working within the EI/maternity benefits system. As we clarified our requirements and made contacts with many different experts, we started to identify the exact people that we wanted to form close relationships with. We found that it was often helpful to have a demonstration of our product ready to go, to expedite the communication process. The demo gave a very clear picture of what we were doing, and any issues the experts saw could be brought to light in a short amount of time, compared to a situation where we would simply try to *explain* the nature of our project without a visual demo.

Once we finally did get in touch with a handful of experts on the maternity benefits calculation, we found that they were understandably very busy, and were not able to devote repeating blocks of time to help us translate the rules. To make the most use of the time that we *did* have

with the experts, we read lots of documentation in advance, and set up our own test scenarios with some prepared questions. When we had access to the policy experts, we would walk through these scenarios, and have the experts point out any inconsistencies, and clarify the questions we had prepared. This process was fairly effective, but we still lacked the high level of engagement from policy experts that was needed in order to have a high degree of confidence in the encoded rules.

Placing this in the context of our RaC characteristics, we can say that this was *partially* collaborative. The system is exposed as an API and therefore reusable. The code is open-source and available on github for transparency, and there are tests built around the system, although again, these tests were not as collaborative as we would like. This all highlights the importance of the aspect of collaboration. If we don't have heavy collaboration with policy experts, then we cannot have confidence in the reliability of the rules.

Motor Vehicle RaC Sprint

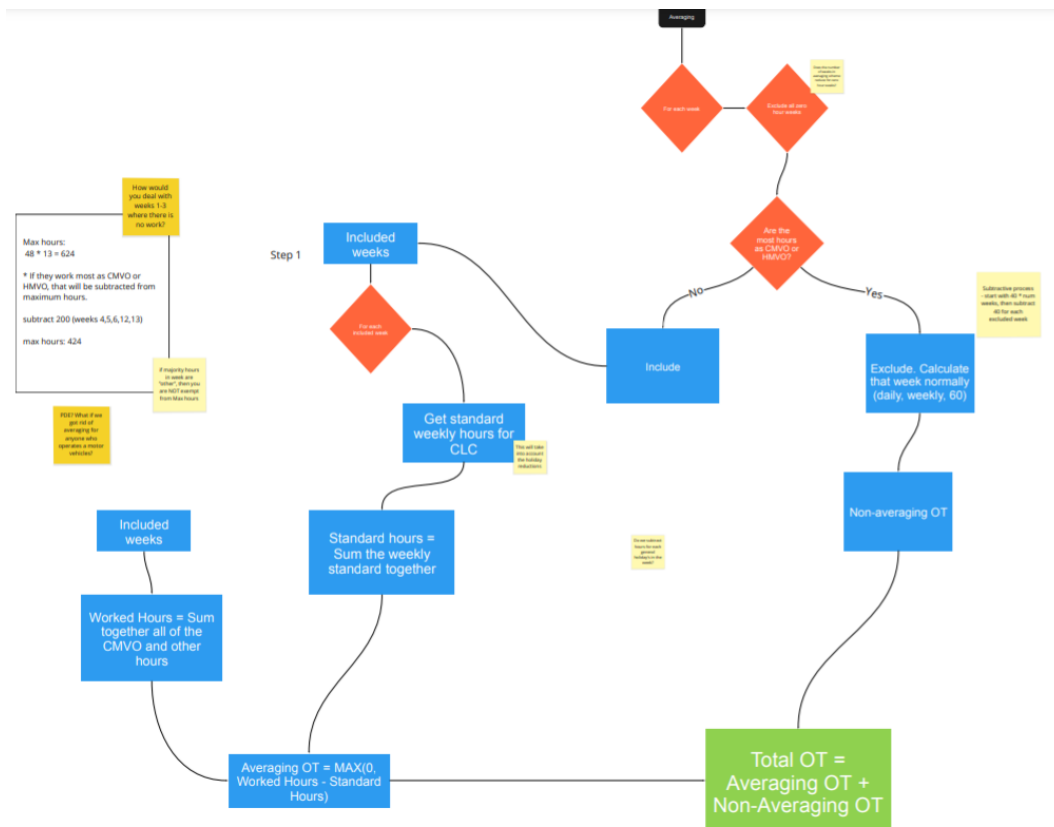
The translation process for the Motor Vehicle Regulation sprint was a very different experience than for maternity benefits. This was because the sprint had stakeholders across many important organizations and we had direct engagement from the policy experts. The sprint itself was 3 weeks and it was a very involved and collaborative process. We had a very large shared virtual whiteboard (Miro) where we organized all of our activities and ideas, and we would spend hours every day working on this translation process.

We began by having everyone get a clear understanding of the regulations. This involved multiple thorough readings, both as a team and individually. After a few reads, we would mark down any areas that were confusing or ambiguous. Regulations contain a lot of legalese, so this was much needed. The policy experts would then clarify these areas. Once we had a general shared understanding, we began the process of mapping out some flows and diagrams of how the overtime calculation proceeded. This involved capturing the various entities and relationships involved between those entities. Flowcharts were a very useful way of helping to understand how the rules worked and for pointing out any extra complexities. We reduced the rules to as simple language as possible, with the policy experts ensuring that any simplification of the language didn't lose the original meaning of the document.

Once we had built some flowcharts, we created several example scenarios of motor vehicle operator schedules. We used examples of traditional schedules, and also some intentionally complex cases to really test the flexibility of the rules. Once we established these test scenarios, we could use our flowcharts to walk through each example, to ensure all of the appropriate logic was captured. This resulted in many refinements to the flowcharts to make them as precise as possible. Occasionally we would run into some ambiguity that the policy experts had not actually encountered before. For the most part, these ambiguities hadn't been resolved in the written rules due to the fact that they rarely came up in the real world, but as we've mentioned, encoding written rules to machine language necessitates the resolution of ambiguities.

Once we felt fairly comfortable with the test cases and flowcharts, we were able to begin the actual coding and automated testing process. Refined flowcharts translate fairly seamlessly to code, so this was largely a smooth process. We would code the rules, and write the coded tests that corresponded to the tests that we fleshed out on the whiteboard, and we would verify it again with the policy experts. We would get some feedback on the code and design some improvements, and then iterate on it. The result was a precise rules engine that was built collaboratively between technical and policy experts, giving a high degree of confidence in the accuracy of the system. In addition to being collaborative and testable, we also exposed the engine as a reusable API, and the code was made available on github for transparency.

Screenshots of flowcharts and examples from our virtual whiteboard session:



Scenario 1

	BO	CMVO	HMVO	Other	Total
Monday		8	4	12	
Tuesday		4	8	12	
Wednesday			10	10	
Thursday			10	10	
Friday		6	6	12	
Saturday			8	8	
Sunday					
Total		0	28	36	64

Scenario 2

	BO	CMVO	HMVO	Other	Total
Monday		5		4	12
Tuesday		4		8	12
Wednesday		10			10
Thursday			10		10
Friday		6		6	12
Saturday			5		5
Sunday					
Total		28	5	28	61

Regulations do not specify how to break ties. Always a winner

Daily:
 - M: 0 OT
 - T: HMVO is excluded - 0 OT
 - W: 0 OT
 - R: 10 - 8 = 2 OT
 - F:
 -- Tie!
 -- a) HMVO wins - 0 OT
 -- b) Other wins - 0 OT
 - S: 0 OT
 - Total: 2 OT

Weekly:
 - inspector: 36 < 40 => 0 OT
 - regs: 36 < 40 => 0 OT

60:
 - 64 > 60 => 4 OT

MAX = 4 OT (HMVO rate)

Daily:
 - M: 12 - 9 = 3 OT (CMVO)
 - T: 12 - 8 = 4 OT (other)
 - W: 10 - 9 = 1 (CMVO)
 - R: 10 - 8 = 2 (other)
 - F: tie
 -- CMVO wins:
 -- 12 - 9 = 3 (CMVO)
 -- Other wins:
 -- 12 - 8 = 4 (other)
 - S: 0 OT
 - Total: 13/14

Weekly:
 - CMVO wins: 45
 - 56 > 45 => 11 OT
 - Other wins: 40
 - 56 > 40 => 16 OT

60:
 - 61 > 60 => 1 OT

MAX =
 - CMVO: 13
 - Other: 16

Scenario 7

	BO	CMVO	HMVO	Other	Total
Monday		4	4	8	16
Tuesday		4	6	4	14
Wednesday		6	4	4	14
Thursday		5	5	5	15
Friday			4		4
Saturday			4		4
Sunday					
Total		19	23	19	61

Scenario 8

	BO	CMVO	HMVO	Other	Total
Monday		8	4		12
Tuesday		8	4		12
Wednesday			2	8	10
Thursday	-	-	-	-	-
Friday		8	4		12
Saturday			6	4	10
Sunday					
Total		24	16	16	56

Scenario 9

	BO	CMVO	HMVO	Other	Total
Monday		8	4		12
Tuesday		8		4	12
Wednesday			2	8	10
Thursday	-	6	5	1	12
Friday		6	4		12
Saturday			6	4	10
Sunday					
Total		24	16	16	56

Daily:
 - M: 10 - 8 = 2
 - T: HMVO wins - 0
 - W: 10 - 9 = 1
 - R:
 -- HMVO wins - 0
 -- CMVO wins - 1
 -- other wins - 2
 - S: 0
 - Total: 3 - 5

Weekly:
 - regulation: skip to 60 (HMVO wins)
 - inspector: exclude HMVO
 -- tiebreak - either way, 0

60:
 - 61 - 60 = 1 OT

MAX = 3 - 5

Daily:
 - M: 0
 - T: 12 - 9 = 3
 - W: 0
 - R: * assume no work on holiday
 - F: 0
 - S: 0
 - Total: 3

Weekly:
 - CMVO wins (45)
 - holiday: -> 45 - 9 = 36
 - 40 > 36 = 4 OT (CMVO)

60:
 - literal reading:
 -- 56 < 60 => 0
 - if we drop to 50
 - 56 > 50 => 6 OT
 ** in this case, no difference

MAX = 6 (HMVO)

Could have situation where employee works on holiday. Not considered for OT

Same problem would occur in 8(4)

regulation doesn't specify to drop on a week with general holiday: instead AT 60, according to 7(2)

PDE candidate? Measure what happens in both cases?

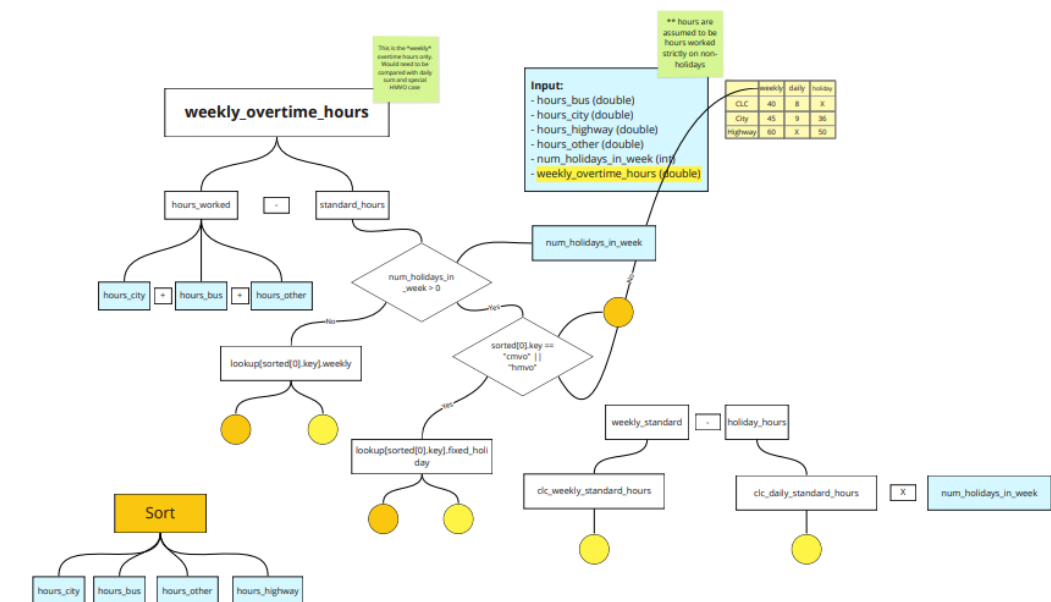
Daily:
 - M: 0
 - T: 12 - 9 = 3
 - W: 0
 - R: * Don't count holiday hours
 - F: 0
 - S: 0
 - Total: 3

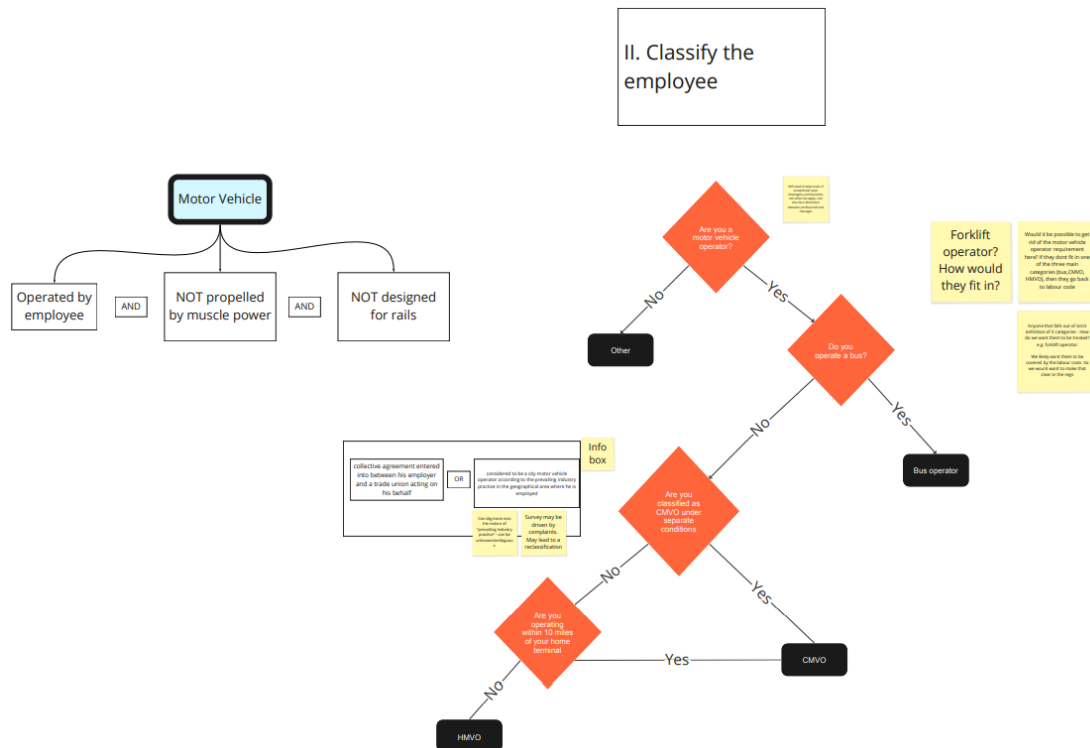
Weekly:
 - CMVO wins (45)
 - holiday: -> 45 - 9 = 36
 - 40 > 36 = 4 OT (CMVO)

60:
 - literal reading:
 -- 56 < 60 => 0
 - if we drop to 50
 - 56 > 50 => 6 OT
 ** in this case, no difference

MAX = 6

* Same results but Thursday holiday is not counted. Need to ensure key are submitted





We found some of these approaches to be very effective for coming up with a precise encoding of the rules. Establishing a basic flowchart of the rules in question and then iteratively refining it as the team collaboratively walks through the examples was very productive, and made it very easy to actually write the code and the corresponding tests. Once you have a refined logical flowchart, it is straightforward to convert the steps into computer code, and the tests that you walked through together then become your acceptance tests for the system, since they have been validated by the policy experts.

Part III: Technologies

Now that we've discussed the process for translating the written rules to RaC, we will explore some technologies that were used in the development of our two RaC systems.

OpenFisca and The Rules API

OpenFisca (<https://openfisca.org/en/>) is a software that provides a framework for encoding certain types of rules - particularly government rules that involve some sort of entitlement or eligibility system. If we zoom out on a generic government benefit that involves monetary entitlement, some common concepts begin to emerge. Most benefits will involve some sort of eligibility test. You are eligible or you are not. There may be many factors involved in this determination, such as age, location, employment status, etc, but the idea of an eligibility test is common. The "target" of such a benefit may initially seem like it is always an individual person (as it is with maternity benefits and motor vehicle operator overtime hours), but this isn't always the case. Instead it may be a different entity, such as a family, or a couple, or an organization. If the entity is deemed to be eligible for a benefit, then we must next determine the amount that the entity is entitled to. This will once again involve various factors, depending on the actual benefit, but again, the idea of the entitlement calculation is common. These are the types of scenarios that OpenFisca is well-suited to handle.

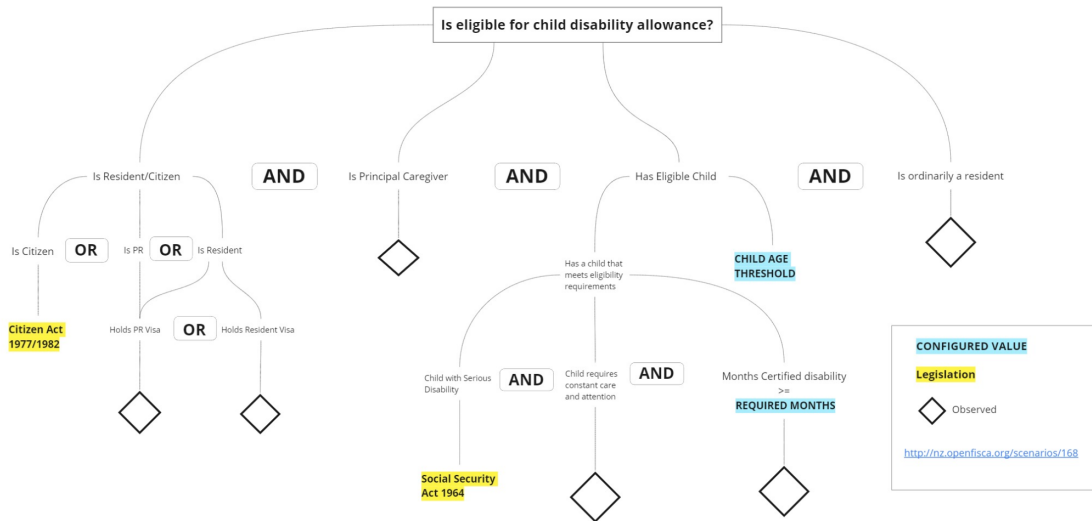
We can map these generic descriptions onto our two systems:

- Maternity benefits: If someone is applying for maternity benefits, we must first ensure that the applicant is eligible. Some of the eligibility criteria includes the expected due date of the baby, as well as the number of insurable hours that the applicant has worked in the past year. If they are determined to be eligible, then we must calculate the entitlement amount. This depends on factors such as the income that they made in the past year as well as the unemployment rate in their region of Canada. Note that for the PDE, we only focused on the entitlement amount, not the eligibility. These can be thought of as two distinct rules, which would require lots of input from policy experts to get right. To keep our project manageable, we scaled it down to focus only on the *entitlement* calculation, and ignored the *eligibility* calculation.
- Motor Vehicle Operator Overtime hours: The eligibility criteria for getting overtime hours is specified in the regulations, and includes information about the nature of the applicant's work. To be eligible they must be operating a vehicle that meets the definition

of a motor vehicle (e.g. cannot run on rails, cannot be muscle-powered). If they are deemed eligible to receive overtime hours, then we must calculate the amount of overtime hours they are entitled to. Note that in this case, it is not a monetary amount, but it is still a quantity, so the system still works just fine. This calculation depends on the number of hours they worked during the week under the different vehicle classifications (city vehicle, highway vehicle, bus, etc).

This shows that both of our situations are well-suited to a system such as OpenFisca. They are very different rules, so they may not appear to have much in common, but OpenFisca captures these high-level concepts such as eligibility, entities, and entitlement. Note the criteria in the examples given are all expressible as numbers or simple logic. The goal of the RaC translation process is to parse this logic out of the written rule. Some of the strategies we took to this end will be described in the next sections. The end result will either be that the logic is successfully parsed out, or an ambiguity will be encountered. The positive side-effect of this process is that ambiguities are brought to light and can be ironed out.

Another concrete example is the New Zealand eligibility for a child disability benefit. This diagram was generated early on in our research to help visualize how a written eligibility rule might translate into machine code. We began by breaking down the key requirements for eligibility. Each of those requirements can then be broken down into sub-requirements, and so on. At some point, these sub-requirements must break down into something concrete. This could be an observable fact (individual holds a Permanent Resident Visa), some quantity measurement (age verification), or a definition stated from another government rule.



Once the rules have been mapped out and encoded into OpenFisca (this is done in the Python language), the project can be exposed as a Web API, so that other applications are able to make use of it. In this way, OpenFisca can act as our rules engine. The Web API functions by exposing a single endpoint: “POST /calculate”. The client that is calling the API adds in properties to reflect known values that are used in the calculation, and passes in the unknown values (such as the entitlement amount) as a null value. OpenFisca processes the request by calculating and filling in the null values where it can and passing it back to the client.

POST Request being made to OpenFisca for the Maternity Benefits calculation, followed by the generated response. Note that the desired value (maternity_benefits_entitlement_amount) in the request is marked as null. In the response, it is filled in with the calculated result.

POST

{{host}}/calculate

Params

Authorization

Headers (11)

Body

Pre-request Script

Tests

Settings

none

form-data

x-www-form-urlencoded

raw

binary

GraphQL

JSON

```

1  {
2    "persons": [
3      "test_person": {
4        "maternity_benefits__entitlement_amount": {"2020-08": null},
5        "maternity_benefits__average_income": {"2020-08": 1000},
6        "maternity_benefits__max_weekly_amount": {"2020-08": 595},
7        "maternity_benefits__percentage": {"2020-08": 55},
8        "maternity_benefits__number_of_weeks": {"2020-08": 10}
9      }
10   ]
11 }
12
  
```

```

1  {
2    "persons": {
3      "test_person": {
4        "maternity_benefits__entitlement_amount": {"2020-08": 8250 },
5        "maternity_benefits__average_income": {"2020-08": 1000},
6        "maternity_benefits__max_weekly_amount": {"2020-08": 595},
7        "maternity_benefits__percentage": {"2020-08": 55},
8        "maternity_benefits__number_of_weeks": {"2020-08": 10}
9      }
10   }
11 }
12

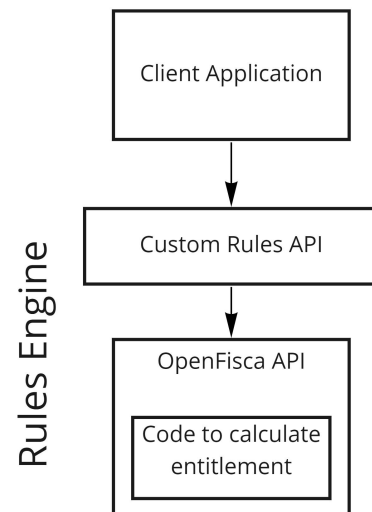
```

This means that any application that makes use of OpenFisca must adhere to this type of API interaction. For many benefits and eligibility scenarios, this would not be an issue. Our first major use of it was with the Motor Vehicle sprint, where we encountered a drawback with it.

In order to calculate the number of hours that a motor vehicle operator is entitled to, the calculation requires looking at the employee's entire weekly schedule. We need to know how many hours they worked in each category (bus driver, highway operator, city operator) on each day of the week. We can say that the data point that is fed into the rules engine is a weekly schedule, and that this weekly schedule is a collection of different values. At the time of writing, OpenFisca is not built to operate with complex inputs such as this. Rather, it is suited for simpler inputs, such as age, income, date of birth, etc. Since we required the input to be a complex object, we managed to come up with some workarounds that worked within OpenFisca's system. This workaround ended up involving two calls to the OpenFisca API for each overtime calculation. Ideally, any application that is talking to the rules engine would only need to make a single call to the rules engine for the entitlement determination. By having two calculations, we are forcing these client applications to handle some of the rules logic on their end, which is not ideal for designing decoupled and reusable systems. After a series of discussions, we decided that this was acceptable for our use case, since we were only doing a 3-week sprint. We acknowledged that if this was to scale up, this issue would need to be addressed.

We decided to address this issue as we built out our maternity benefits PDE. We didn't necessarily encounter the same roadblock as we did for the motor vehicle sprint, but the general problem was that we were **relying on a specification of a third party**. If we were to use

OpenFisca as our application-facing Rules Engine, then we could potentially run into more rules that needed special treatment and would require more workarounds. We did not want to introduce this dependency into the system, so we designed to mitigate that risk early on, after learning about it from the Motor Vehicle sprint. We still wanted to make use of OpenFisca, but we needed a way to define our own contract. This involved building out our own web API layer that sat in front of the OpenFisca web API. This layer would handle any of the extra logic required for these potential workarounds, making the complexity invisible to the client application. So the client application would talk to our own rules API, which defined a clear contract (and would be able to accommodate complex inputs such as a weekly schedule). This rules API would in turn build out the requests for the OpenFisca call, make the call, and send the results back to the client application. So the idea is that it is a thin layer that handles the extra workaround logic required to work with OpenFisca and it makes for a more architecturally sound development experience.

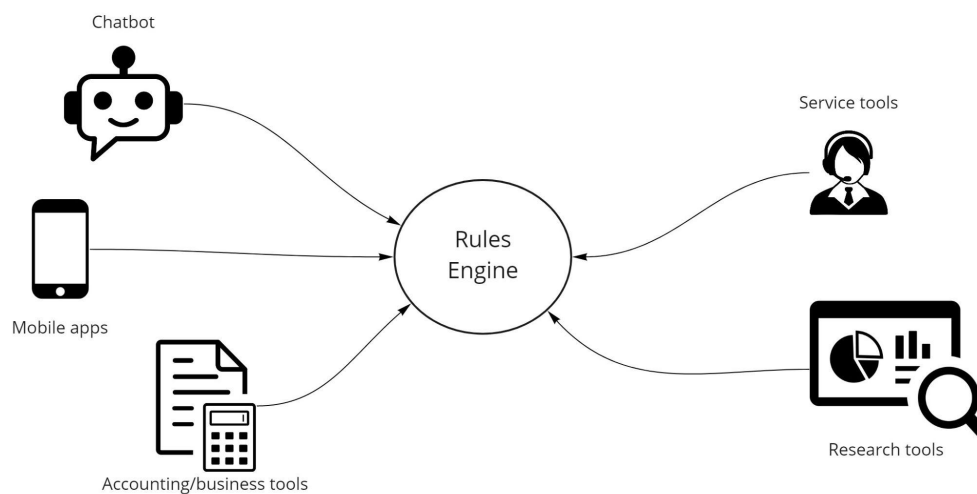


So at this point our “rules engine” has evolved from simply being the exposed OpenFisca API to being a custom web API that can talk to OpenFisca in the background. The main benefit here is that we are not reliant on the limitations of a third party. An added benefit is that if we discover yet another third party rules system that meets some of our needs, we can easily integrate it behind the scenes of our Rules API, and the experience is seamless to the client applications.

API Gateway layer

As we built out our maternity benefits PDE, we began to ideate on how the systems involved, particularly the rules engine, might scale out to accommodate further rules. Our system was focused on a single rule - the maternity benefits entitlement calculation. One of the stated goals of the rules engine is to be reusable, so we started to think about other systems that might make use of this rules engine. In addition to the policy difference engine, what other applications would use a tool that calculates maternity benefits entitlement? Some examples include:

- A tool for service desk agents: If a potential applicant calls in and wants to know how much they might be entitled to, the agent could use such an application to give them an estimate
- An online calculator: If the applicant wants to go online to find their potential entitlement, then that web application could make use of it
- A chatbot or virtual assistant: If someone wants to find out what benefits and entitlements they are eligible for, then this application may use the rule in the backend
- HR applications: If a company wants to know how much an employee might be entitled to so that they can customize a benefits matching program, then they might be able to make use of it as well.
- Another potential tool that we came across is a benefits finder. This tool would allow someone to discover what benefits they might be eligible for given a set of personal information, and maternity benefits could be one of these. However, our PDE tool only focused on the entitlement calculation, rather than eligibility, and these are different calculations that make use of different data. So while our RaC system would not be directly used by the benefits finder, a related RaC system would definitely be valuable.



Sidebar: There is a very important point worth mentioning here. The most obvious tool that would make use of a rules engine would be the tool that *actually calculates how much an applicant* is entitled to. When an official maternity application is filed, there is an existing coded rules system that performs this calculation. We did not get a chance to look at this system, but we were informed that it is a very complex set of different processes and systems, sometimes involving human input (e.g. in the edge cases). In an ideal world, this system itself and the

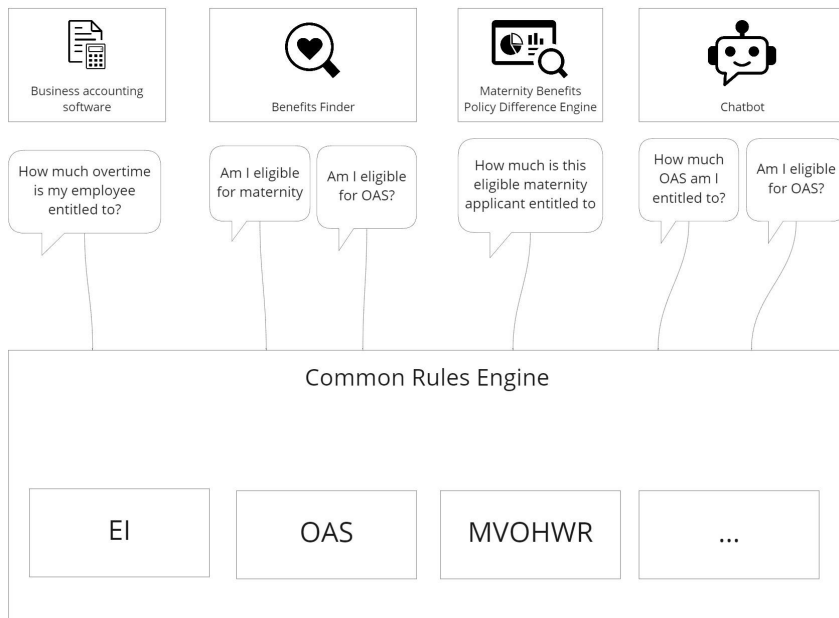
system we are proposing would be one and the same. There would be a single source of truth that is the rules engine, exposed as an API and usable by any of the applications listed above. We have been led to believe that this is very far from reality, but part of our goal is to show the value of Rules as Code systems to build momentum behind the approach.

In absence of having access to the real maternity benefits calculation code, we are ideating on how to build out a reusable RaC system that captures the exact same rules. Suppose we do this for many different government rules. We've done it for the MVOHWR and we are working on it for maternity benefits, but there are many other government rules that we could apply the same treatment to, such as CPP, OAS, regular EI, special cases of EI, etc. The question arises of how we organize these different systems. Do we have them all part of the same project? Do we just keep them completely decoupled so they aren't related at all? Or is there something in between?

To illustrate this problem, let's consider a client application such as a benefits finder. This application is going to need to have access to many different coded government rules that determine eligibility for different benefits. Suppose we got a bunch of different policy experts to help us code all of these different rules with appropriate tests. We will look at different scenarios for how to organize this code.

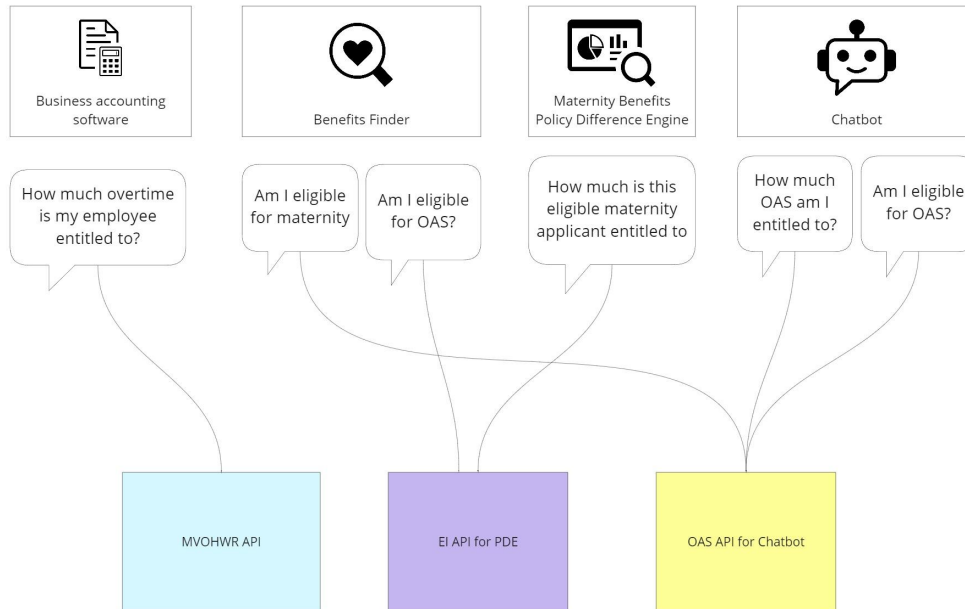
Let's suppose first that we put all of the code into a single project, such as a web API. This may make it smoother for the client application to interact with, since it must only speak with one application. But it should be immediately clear that this will be very unsustainable. These rules can be very complex and may require lots of code. If they are all in the same project, then we have a monolith. If we need to tweak *one* of the rules slightly, then we need to deploy the entire project, which contains all of the rules. This is one of the usual arguments against monolithic architectures for any sort of software system. It's too much code in one system. It slows down development and one error can bring down the entire system. Furthermore, it may enforce a single set of technologies (programming language) to be used for the entire system.

Diagram illustrating the monolithic RaC architecture approach. All of the code is stored in a single system, the “Common Rules Engine”



This motivates the idea of microservices, where the code is logically separated into different components which can be developed and deployed individually. We may have one API for maternity benefits, another API for OAS, another for Motor vehicle regulations, etc. This decoupled approach promotes quicker and more agile development and allows for development teams to use whichever language they prefer, as long as the functionality is exposed as an API. This all sounds good, but now the benefits finder client application has a bit of a usability problem. It is now responsible for knowing which API is responsible for what, and ensuring it has the right connection specifications for all them. This problem is not insurmountable, but it makes for a rougher development experience.

Diagram showing a microservices approach to the RaC architecture



So on one hand, we can make a monolith that is easy for a client application to use, yet is unsustainable for development. Or we can take a decoupled microservices approach, which makes the development process smoother, but makes for a rougher integration experience for the users of the RaC systems. We are looking for something in between. We want to create a simple and seamless user experience for the client application developers, but we want the RaC systems to be easy to work with. One solution is to use a “gateway” layer. This layer has many synonyms, which, depending on the context, are used to mean something similar - gateway, management layer, orchestration, facade, etc. We will stick with the term gateway for now.

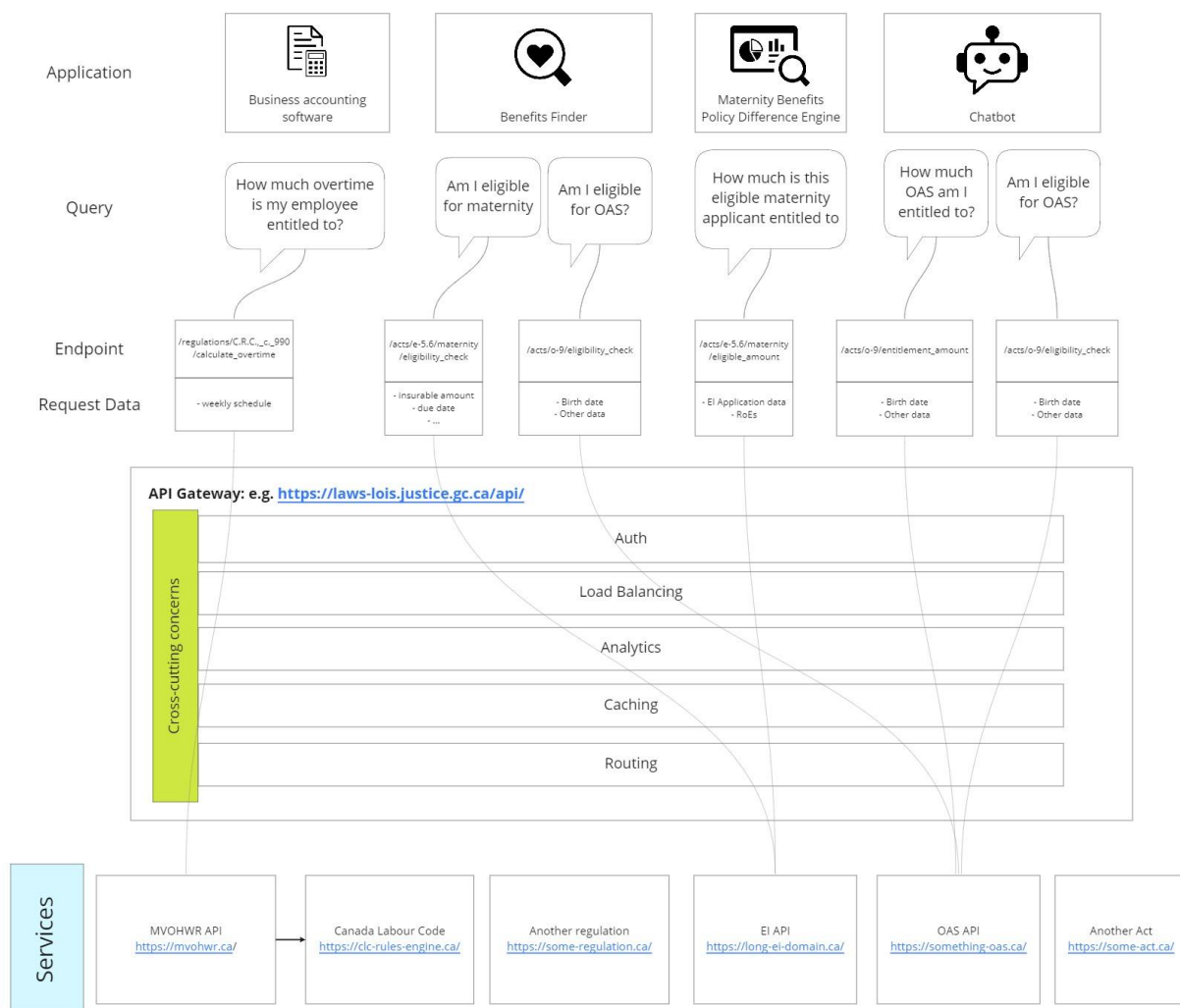
A gateway creates a common entry point into our different APIs. The gateway sits in front of all the APIs so that the client application doesn’t talk directly with the API, but instead communicates with the gateway. Depending on the request given, the gateway will route the request to the appropriate API. So if the benefits finder is checking if someone is eligible for OAS and for CPP, then instead of making these two calls to separate APIs (these are just example calls):

- https://oas-api.net/eligibility_check
- https://cpp-api.net/eligibility_check

It can instead make the calls to the same API:

- <https://canada-rules.net/oas/eligibility>
- <https://canada-rules.net/cpp/eligibility>

The difference may not seem like a big win, but there are many other added benefits to having this gateway layer. If every call to the rules engine is going through this gateway, then we can add any cross-cutting functionality to this layer, rather than duplicating it in each API. For example, suppose we want to add some logging, error-handling, or authentication on some of these calls. Rather than building it into every single API, we can add this functionality *once* to the gateway layer, and it will be triggered on every call.



The gateway approach, while largely theoretical in the government RaC space, is a common architecture pattern in software development, and there are many resources that support such an architecture. As a proof of concept, we ran a small but successful experiment with Azure API management. We put a set of mock rule APIs behind the management layer gateway, then set up a client application to talk to the gateway rather than the APIs directly, and we were able to establish a successful line of communication. Azure API management also comes with other bonuses, such as an automatically generated developer portal, where the Open API specs for the various APIs behind the gateway can be easily displayed to developers of client applications. There may be many other technologies that meet the needs for a gateway design, but Azure API management was the one we tried a demo with, and for our purposes, it was successful.

Conclusion

This document covers in detail the lessons learned throughout two different experiences we had in working on RaC systems - the Maternity Benefits PDE and the Motor Vehicle RaC Sprint. The Maternity benefits PDE was the main focus of our 10-month fellowship, and the RaC sprint was a 3-week sprint that involved collaboration with many other stakeholder organizations. From a RaC perspective, the Motor vehicle sprint was much more successful in establishing a reliable rules engine. This was primarily due to 2 factors:

- The chosen rule (MVOHWR) was much smaller in scope and self-contained (i.e. it was not mixed in with other large pieces of legislation, although we *did* occasionally need to make reference to the Canada Labour Code). This made the problem space smaller and more manageable. Maternity Benefits are mixed into the much larger and more complex EI legislation, making it difficult to build precise requirements around.
- More importantly, we had engagement and buy-in from other stakeholders, particularly the policy experts. They were dedicated to working collaboratively on this project for 3 weeks. This gave us the opportunity to get quick feedback and refinement on our understanding of the rules. The sprint itself was only 3-weeks, but that glosses over the extremely valuable work that was put in before the sprint started in identifying and getting buy-in and commitment from the key stakeholders.

As we developed the Rules as code requirements, we found several strategies that were most effective:

- We found that many of the policy experts were understandably unfamiliar with the processes involved in software development. Since this is a collaborative and iterative process, it is important to establish at the very minimum a shared vocabulary. We created a short presentation to bring all stakeholders up to speed on some important concepts, such as source control, security, APIs, the agile process, etc.
- Work iteratively and have a demo to show stakeholders. Having a demo of the working RaC engine is a great way to quickly communicate your progress and allows policy experts to more easily identify any areas that need improvement.
- Iteratively building flowcharts representing your rules and stepping through examples to refine those charts was a productive way of comprehending the rules. It also make it easier to start coding the rules, since concrete flowchart logic can easily translate into machine code. And you can use the tests you've walked through as acceptance tests for the system

Once we had established a productive process for refining the core requirements of a rules engine, we iteratively wrote the code and sought feedback from the policy experts. We explored various technologies to meet the needs we defined for a RaC engine, such as OpenFisca and a custom rules layer. We explored how this architecture might scale up to incorporate more government rules without becoming unstable.

- OpenFisca is well-suited to handle benefits and eligibility scenarios. It comes with built-in testing and API functionality so that rules can be quickly coded and tested. We did encounter some limitations to the types of calculations it is able to handle, which made it architecturally unsound to use as the main entry point into the Rules engine.
- To remedy this, we built a thin custom layer on top of OpenFisca that could handle the extra logic required for the more complex scenarios. This allows us to take advantage of the strength of OpenFisca, while still maintaining control of our own API contracts, as well as creating a flexible system that could accommodate other third-party RaC tools.
- Another layer that can be added on top of this system is an API gateway layer. If Canada's RaC system were to expand to encompass many more rules, then we need a sustainable way of scaling out to more rules engines. An API Gateway avoids a monolithic architecture that is cumbersome to change, it creates a simple interface for client applications that want to use the various rules engines, and it is able to handle any

cross-cutting concerns for the various rules engines (logging, error-handling, authentication, load-balancing, etc).