waylay

eBOOK

Big Book of IoT Automation



We are now living in an automation economy and witnessing the shift towards software squared, where automation software is eating up the software that is eating up the world.

AUTOMATION IS THE technology by which a process or procedure is performed with minimum human assistance. In the context of IoT, automation operates across physical devices, services and people.

When faced with the challenge of implementing automation for IoT, one might be tempted to start quickly with hard coded rules in the application.

This e-book consists of three parts:

- **PART I** explains why using a rules engine from the get-go is preferable to hard coding rules when implementing automation logic for IoT applications (chapters one to three).

- **PART II** then defines a seven-point benchmark to help Enterprise Developers and Architects choose the right rules engine technology for their IoT use case (chapters four and five).

- **PART III** takes the most common types of rules engines that can be used in the IoT domain and evaluates them against the seven-point benchmark defined in PART I, giving them a 0-100 score.

This eBook is for Enterprise Architects and Developers who are involved in building, packaging and commercialising IoT solutions and who already understand the need for an automation framework for IoT application development.

TABLE OF CONTENTS

PART 1: WHEN DO YOU NEED A RULES ENGINE IN IOT

LOGIC 6

Learn why computer-expressed logic differs from human expressed logic and how that makes it difficult to translate human user requirements into computer code.

TIME 9

The conditional statements used to translate user requirements into code do not function in a static vacuum. Learn how the time dimension adds complexity.

UNCERTAINTY 11

User requirements are often dependent on changing conditions, and probabilities affect these changes. Conditional statements (rules) must be able to express uncertainty.

THE RULES ENGINE 13

The rules engine is an abstraction tool that helps developers to create software with logic, time & uncertainty without modeling them separately in the code.

PART 2: HOW TO EVALUATE A RULES ENGINE

INTRODUCTION 17

MODELING COMPLEX LOGIC 18

Handling high-order logic (HOL), conditional execution of functions, non-Boolean / nonbinary states, majority voting.

MODELING TIME 19

Dealing with the past, present and future within rules and handling expired information, combinations of synchronous and asynchronous data and forecasting.

MODELING UNCERTAINTY 20

Dealing with noisy data or missing data, modeling the utility function, handling probabilistic reasoning.

EXPLAINABILITY 21

Clarity of the rule's intent, compact representation of logic, simulation and debugging capabilities during design and at runtime.

ADAPTABILITY 22

Flexibility (dealing with changing business and technical requirements) and extensibility (extensions & integrations with third-party applications).

OPERABILITY 23

Capabilities to ensure supporting large volumes (bulk operations, templating, versioning, searchability and rule analytics).

ARCHITECTURAL SCALABILITY 24

Sharding and distributed computing for horizontal scalability.

CONCLUSION 25

PART 3: RULES ENGINES BENCHMARK FOR IOT

INTRODUCTION 29

HOW TO READ THIS GUIDE 30

BENCHMARK CRITERIA 32

FORWARD CHAINING ENGINES 33

An engine using forward chaining (FC) applies a set of rules and facts to deduce conclusions, searching the rules until it finds one where the IF clause is known to be true.

CONDITION ACTION ENGINES 37

Condition-action engines are a sub-class of forward chaining engines that are using only one conditional statement as the IF clause.

FLOW PROCESSING ENGINES 40

Flow based programming (FBP) uses “black-box” functions represented as nodes, that exchange data across predefined connections by message passing.

DECISION TREES / DECISION TABLES 44

Decision trees use a branching method to illustrate every possible outcome of a decision.

STREAM PROCESSING ENGINES 48

Stream processing is the processing of data in motion. In other words, computing on data directly as it is produced or received.

CEP ENGINES 52

CEP engines differ from stream processing engines in the way they handle events and are mostly deployed in edge computing.

FINITE STATE MACHINES 55

A state machine can be used to describe a system in terms of a set of states that it goes through. Business rules engines (BRE) are an example of finite state machines.

BAYESIAN ENGINES 58

Rules engines based on Bayesian networks allow both backward and forward inference (state propagation).

CONCLUSION 63

Part 1 When Do You Need A Rules Engine In Iot

Logic

Part 1
When Do You Need A Rules Engine In Iot

See how formal computer logic is different from human expressed logic and why developers have a difficult time translating user requirements into conditional statements (rules) when designing software.

KNOWING A LANGUAGE means being able to produce an infinite number of sentences never spoken before and to understand sentences never heard before. For us humans, it's natural to say things like **Tom likes football and pancakes.** For non-developers, the mental effort required to translate such statements into computer language might not be that obvious. If we were to literally write the same statement into a computer program, it would mean (for the machine) that **Tom is happy only when watching football while eating pancakes.**

So when writing software, what you are basically doing is translating **user requirements** (human stories described using human language) into **rules** (conditional constructions written using computer language). And while doing that, you need to be aware of the differences between human-spoken logic and computer-spoken logic so that you don't accidentally condemn Tom to only finding happiness when watching football while eating pancakes.

As computer language consists of both Propositional Logic (assumes that the world contains facts) and First Order Logic (assumes that the world contains objects, relations and functions), one may argue that developers are well-equipped and computer language is all that is needed to enable them to write any algorithm and conditional statement (rule) needed to translate a human requirement into code.

We argue that it isn't, primarily because of three major difficulties that stand in the way of developers - the first difficulty is brought about by the complexity of the logic, as we will see below. The second and third difficulties (brought about by time and uncertainty) will be covered in the next two sections.

So now let's look more closely into the process of building software applications using computer logic made up of conditional constructions.

Boolean Algebra - the language of mathematics and machines the equivalent of Propositional Logic), has precise and well defined constructions or "machine words" that make up its vocabulary: the conjunction AND denoted as ", the disjunction OR denoted as #, and the negation NOT denoted as ¬.

So how should we then read this table:

Rule	$\wedge$ (AND) form	$\vee$ (OR) form
Identity	$1 \wedge p = p$	$0 \vee p = p$
Null	$0 \wedge p = 0$	$1 \vee p = 1$
Idempotent	$p \wedge p = p$	$p \vee p = p$
Inverse	$p \wedge \neg p = 0$	$p \vee \neg p = 1$
Commutativity	$p \wedge q = q \wedge p$	$p \vee q = q \vee p$
Associativity	$(p \wedge q) \wedge r = p \wedge (q \wedge r)$	$(p \vee q) \vee r = p \vee (q \vee r)$
Distributivity	$p \vee (q \wedge r) = (p \vee q) \wedge (p \vee r)$	$p \wedge (q \vee r) = (p \wedge q) \vee (p \wedge r)$
Absorption	$p \wedge (p \vee q) = p$	$p \vee (p \wedge q) = p$
De Morgan's Law	$\neg(p \wedge q) = \neg p \vee \neg q$	$\neg(p \vee q) = \neg p \wedge \neg q$

For instance, De Morgan's Law says the following: the negation of "a and b" is equivalent to "not a or not b", while the negation of "a or b" is equivalent to "not a and not b".

Now imagine a software program in which multiple statements using Boolean Algebra are joined together. The longer the conditional statements are, the harder it is to test their validity by reading the code alone. For instance, these statements are equivalent: (a < b || (a >= b && c == d)) , (a < b || c == d). Chaining a couple of these statements together makes it very hard to verify the intended logic.

The difficulty of verifying intended logic can also be measured by a metric called cyclomatic complexity, that Thomas McCabe came up with in 1976. Cyclomatic complexity is a quantitative measure of the number of linearly independent paths through a program's source code. Even though its usefulness as a measure of software quality has been questioned, in general, in order to fully test a module, all execution paths through the module must be exercised. This also implies that a module with higher complexity is more difficult to understand, since the programmer must understand the different pathways and the results of those pathways.

As circuit designers may point out, there are methods such as Boolean simplification and Karnaugh mappings for simplifying digital logic. K-maps can help, but someone reading the code must also understand its intent, something that isn't at all easy with K-Maps.

So what we have seen already is that the complexity of logic already brings about two major hurdles for developers trying to translate user requirements into correct conditional statements (rules) when designing software:

First, humans often use logical statements "incorrectly" when expressing rules using spoken language.

Second, even when these constructions are coded properly, it is still hard for humans to check their intended logic, if conditional statements are too long.

Let us now look at the other two major challenges that stand in the way of developers, one brought about by time and the other by uncertainty.

Time

Part 1
When Do You Need A Rules Engine In Iot

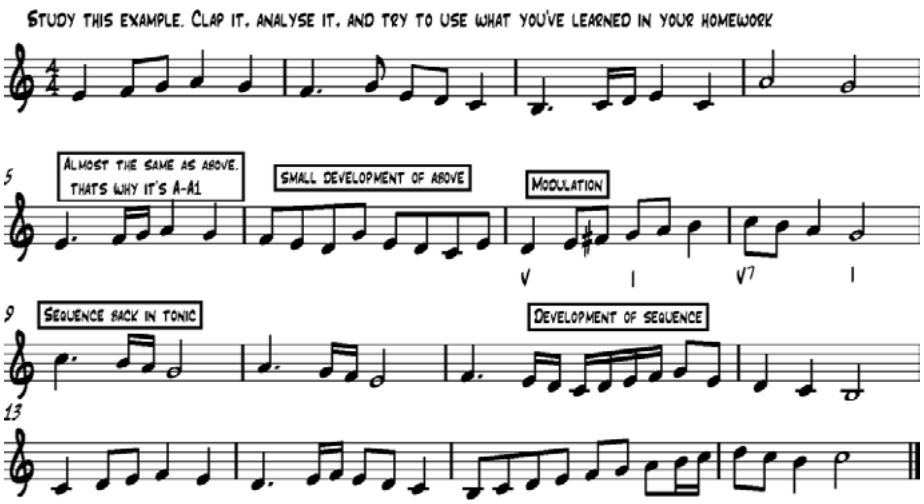
See how the time dimension complicates matters further for developers that are building logic with conditional statements (rules) that change over time.

"TIME IS AN observed phenomenon, by means of which human beings sense and record changes in the environment and in the universe. Time has been called an illusion, a dimension, a smoothflowing continuum, and an expression of separation among events that occur in the same physical location." — whatis.techtarget.com

Imagine that you wake up in the middle of the night to the sound of your dog barking. You then hear footsteps from your kitchen. Just as you're about to call the police, you remember that your friend Tom is sleeping at your place over the weekend and it's probably just him getting something from the kitchen. As you hear the familiar sound of the fridge door and bottles clinking, you are now sure it's him and immediately go back to sleep.

What we can see with this short story is that the order of events in time, in combination with their joined likelihood is what triggers us to take further action. In the previous section, we have introduced formal logic, which would govern the world if the world was a static place. It isn't.

If we were to look for a safe refuge in computer language to help us deal with the time dimension while building logic, similarly to what music notes are doing for music, we would find nothing.



To deal with time in the code, all we have at our disposal is a "CPU clock". We often make use of UML State/Flow diagrams to help us with time, but UML is a "language" for specifying, visualizing, constructing, and documenting the artifacts of the software system. UML helps us communicate what we want to build, it's not a framework for building software.

With the introduction of time, the developer needs not only to understand the different pathways and the results of those pathways (as explained in the first section on inductive reasoning), but she also has to grasp how these pathways change over time.

Uncertainty

Part 1
When Do You Need A Rules Engine In Iot

The uncertainty principle is an important element to building computer logic as probabilities can affect conditional statements (rules) over time.

THERE IS ONE more thing that's important to note about that time when you almost called the police on your thirsty friend. As events were unfolding, you were getting more and more certain that the mysterious person in your home was not an intruder, but your friend Tom. Choosing between calling the police or going back to sleep was guided only by your belief.

This example may sound a little forced, but in reality, more and more software applications require this sort of expression capabilities.

Assisted living systems are one of these, where good-enough rules enable people to live independently and hold on to their sense of dignity while caretakers can still make sure they're safe. In other words, you can either go full CCTV on your grandma or rely on a couple of smart and nonintrusive rules to achieve the same level of safety, with a rules-based approach being easier to maintain and friendlier towards the assisted person.

AI and ML algorithms also give outcomes as probabilities. You will never truly get a definitive "This is a dog." type of answer from them, but rather a "There is a 99.9% probability that this is a dog and a 0.1% probability that it is a cat" type of answer. The same tools give us indications such as "errors of classification".

As more and more applications make use of these tools, we need to somehow express these uncertainties in our applications as well.

The Rules Engine

Part 1
When Do You Need A Rules Engine In Iot

The rules engine is a tool that offers developers a framework to combine logic, time and uncertainty so that they can build software without modeling each of the three manually and separately in the code.

ABSTRACTION IS THE removal of details in order to enhance the visibility of a pattern. A useful abstraction is one that removes things that we don't need to concern ourselves within a given context. In software development, we refer to abstraction as declarative programming.

The goal of a rule engine is to bring the abstraction one level higher than the code and to enable developers to model the world in a declarative way.

Here's how Mundy Follow explains declarative programming:

Declarative Programming is like you telling your friend to paint a landscape. You don't care how they draw it, that's up to them.

Imperative Programming is like Bob Ross telling your friend how to paint a landscape, giving them step by step instructions to achieve the desired result.

Declarative programming is also understood as expressing the logic of a computation without describing its control flow. The control flow is one of three processes that every rule can be broken down into:

• **Information flow** - the gathering of facts.

• **Control flow** - the order in which individual statements, instructions or function calls of an imperative program are executed or evaluated.

• **Decisions** - the outcomes of the reasoning, conclusions which are followed by actions.

The majority of rules engines fit into one of these two categories:

• Rules engines where these three processes happen all at once (we feed the engine with facts, the engine evaluates them, the result is reached and actions are called).

• Rules engines where each process is viewed as a step that is strictly followed by the next one (information enters the engine and at each step certain statements are executed until the outcome is reached, which results in a given action).

As we shall see further on, linear one dimensional thinking in this respect has a big impact on how useful the rule engine actually is.

Part 2
How To Evaluate A Rules Engine

PART 2: HOW TO EVALUATE A RULES ENGINE

INTRODUCTION 17

MODELING COMPLEX LOGIC 18

Handling high-order logic (HOL), conditional execution of functions, non-Boolean / nonbinary states, majority voting.

MODELING TIME 19

Dealing with the past, present and future within rules and handling expired information, combinations of synchronous and asynchronous data and forecasting.

MODELING UNCERTAINTY 20

Dealing with noisy data or missing data, modeling the utility function, handling probabilistic reasoning.

EXPLAINABILITY 21

Clarity of the rule's intent, compact representation of logic, simulation and debugging capabilities during design and at runtime.

ADAPTABILITY 22

Flexibility (dealing with changing business and technical requirements) and extensibility (extensions & integrations with third-party applications).

OPERABILITY 23

Capabilities to ensure supporting large volumes (bulk operations, templating, versioning, searchability and rule analytics).

ARCHITECTURAL SCALABILITY 24

Sharding and distributed computing for horizontal scalability.

CONCLUSION 25

INTRODUCTION

The primary role of a rules engine is to abstract away complexity and enable developers to model the world in a declarative way. There are seven core competencies against which a rule engine could be tested, in order to assess its success.

WHEN EVALUATING ANY new tool, it's good practice to look at how powerful it is (its depth of functionality), how easy it is to use (its level of complexity) and how ready it is to support your future needs (based on your growth trajectory and features you may need).

In the case of a rules engine, when looking at the depth of functionality we need to see if it supports complex logic building, how well it handles the time dimension and how it deals with uncertainty.

Ease of use can be assessed by looking at things like how clear the intent of the rule is, if there is a visual representation of the logic you are building and how easy it is to simulate, test and debug rules.

Evaluating the engine's readiness to grow as your business needs grow can be done by checking how well it responds to changes, how easy it is to extend and integrate with third-party systems, and how well it can scale.

Here are the seven core capabilities that you can look at in order to evaluate a rules engine, explained in detail.

MODELING COMPLEX LOGIC

Real-life application logic is complex, it will inevitably involve more variables than any textbook example. Complex logic is made up of high order logic (HOL) constructions, which is why it's important for the rule engine to support them. To manage that, the rule engine should support the following:

Combining multiple non- binary outcomes of functions (observations) in the rule, beyond Boolean true/false states.

Combining multiple non-binary states is what we do for example when we account for a multitude of factors before we decide whether to go on a city trip over the weekend. We look at the weather (and its range of non-binary states: light rain, heavy storms, snow, cloudy skies, sunny skies) in combination with day of week (seven states), cost of flights and accommodation, etc.

Dealing with majority voting conditions in the rule.

Majority voting refers to taking action based on most of the conditions for that action being met, but not necessarily all. This capability is important for example in healthcare diagnostic systems where some but not all indications may point to a medical issue. Another example where it's used is in "fly by wire" control systems, such as the Boeing 777 fly-by-wire triple redundant computer that replicates the computations in three processors and then performs majority voting to determine the final result.

Handling conditional executions of functions based on the outcomes of previous observations.

An execution based on previous observation is: "If the machine malfunctions, only then check the asset database, otherwise do nothing."

MODELING TIME

Time adds complexity to the rules. In order to deal with the challenges of building time-bound logic, the rule engine should support the following:

Dealing with the past (handling expired or soon-to-expire information)

You often have to use information that is only valid for a fixed period of time or merge data streams that are not fully in sync. This is important in connected home, connected building or industry 4.0 applications. Here are a couple of examples:

- If there is motion in the living room, followed by motion in the sleeping room, then (...).
- If there is motion in the living room but no motion in the sleeping room within the next 5 minutes, then (...).
- Apply this rule only if temperature and humidity data from two different sensors comes no more than 10 seconds apart.
- Check if the state of the machine has changed between two consecutive measurements (and within a window).

Dealing with the present (combining asynchronous and synchronous information)

You often times need to combine asynchronous data flows (streaming IoT device data) with synchronous information (polling cloud service API endpoints) at the time a rule is executed.

A simple user requirement such as: "Send an SMS alert whenever the freezer temp is above 4 degrees" translates into this rule: "When the freezer's temperature is above 4 degrees, check the asset database (over API) to find the location of the warehouse where the freezer is. Then check the weather at that location (to verify if it is not too warm outside). Then create a ticket in the CRM database, before sending SMS to the person that operates the building in which the freezer is located".

Dealing with the future (forecasting for prediction and anomaly detection)

Anomaly detection is typically derived from time series data and it is usually formulated in two different ways:

- Finding outliers (values that are highly above or below the average / rolling average) or finding the time window in which the standard deviation is higher than the expected value.

- Data points that differ too much from expected values - which are mostly statistically derived. In this case, anomaly detection and forecast rely on the capability of the rules engine to find a good "fitting algorithm" for the observed measurements.

MODELING UNCERTAINTY

Uncertainty is unavoidable, and the rule engine should have a mechanism for accounting for it in the way it builds logic.

NOISY SENSOR DATA or even missing data is common in IoT applications where we often deal with wireless sensors which fully depend on the battery lifespan, intermittent network connectivity or with network outages making API endpoints unreachable.

Modeling the utility function relies on the engine's capability of dealing with uncertainty. As we rank and define our preferences among alternative uncertain outcomes, we need rules where for the same outcome of an observation, different actions can be taken.

For even more advanced use cases, the rule engine should enable probabilistic reasoning, supporting logic building based on the likelihood of different outcomes for one given sensory output. Here are some IoT-specific examples:

- Avoid the situation where rules and actions are triggered on data which is too old: only use weather information in the rule if the weather API call hasn't failed for the past 10 minutes. (In Belgium, we don't trust weather forecasts that are longer than 10 minutes).

- Only send an SMS to the police if the security system believes with over 80% certainty that there is an intruder in the house. If certainty is over 50%, turn on the lights in the living room. If it is between 30-50%, send the SMS to the homeowner. That decision can further depend on time-of-day or day-of-week (utility).

You will recognize uncertainty and probabilistic reasoning as concepts regularly dealt with under the general umbrella of AI technologies. We include only these concepts in this category as they serve automation developers to model the world in a declarative way.

Arguably, other AI technologies, such as swarm intelligence algorithms or reinforced learning tools may also lead to actions (and be perceived as rule-generators) but they do not enable declarative modeling. Reinforcement learning is the training of machine learning models to make a sequence of decisions on their own, while swarm intelligence is composition of many individuals agents that coordinate using decentralized control and self-organization based on some very simple rules.

Other AI technologies still, such as supervised and unsupervised machine learning algorithms, are out of the scope of automation but are very useful as inputs for the decision engine.

EXPLAINABILITY

Users require a high level of understanding and transparency into decisions with inherent risk. It's thus important on one hand that rules are easily understood by users that were not directly involved in writing them, and on the other hand, that there's a diagnostic mechanism available.

NEW DEVELOPERS MUST be able to understand a rules engine implementation without having prior knowledge.

The rules engine should include explanation capabilities that make it easy for its users to understand why rules fired (or not) and identify and correct errors. In other words, being able to build and run complex logic is important, but the engine's internal complexity should not come in the way of its users being able to easily test, simulate and debug that complexity.

- **The intent of the rule should be easily understandable** by all users, developers and business owners alike.

- **The representation of the logic should be compact.** Visual representations such as state graphs are useful even when you know how to code because they let you structure your logic in a way that's not easy to visualize in code.

- **Simulation and debugging capabilities should be available,** to provide additional insights both at the time of logic creation and at runtime:

    - During design time - verify the intended logic by testing rules against data logs or simulating logic statements to verify outcomes.
    - At runtime - reconstruct decisions made by the rule engine based on the rules logs and the states of observations.

OPERABILITY

When deploying applications with many thousands or possibly millions of rules running in parallel, the engine should effectively manage the large volumes by supporting:

• **Templating** - so that you can apply the same rule to multiple of devices, or to similar use cases.

• **Versioning** - of both templates and running rules, for snapshotting and rollbacks.

• **Searchability** - so that you can easily search rules by name, API in use, type of device and other filters.

• **Rules analytics** - so that you understand which of your rules triggered the most, what are the most common actions taken etc.

• **Bulk upgrades** - so that you can perform lifecycle mngt across groups of rules, useful for updates or end-of-life.

ARCHITECTURAL SCALABILITY

Although horizontal scalability is more of an implementation effort than a capability of a particular type of rules engine, to enable easy sharding, the rules engine should provide a good initial framework and abstractions for distributed computing.

SHARDING IS A well known concept in database design and refers to a component that can be horizontally partitioned, which enables linear scaling - deploying N times the same component leads to N times improved performances. Some engines are harder to shard, either due to their stateful nature or due to the difficulties to distribute rules computation, so this is an important final aspect to take into consideration when evaluating alternatives.

CONCLUSION

IoT application development involves working across multiple dimensions, combining streaming data with data at rest to connect physical products with people and business processes.

BUILDING LOGIC BY configuring rules straight into your code is sub-optimal. Using a rules engine may alleviate the problems, if the engine meets a number of core capabilities that respond to the specific requirements of IoT.

If you are curious to see how the most common types of rules engines nowadays perform against the seven major criteria that we've described here, you should check out our rules engines benchmarking results. We have analysed forward-chaining engines, flow engines, state machines, decision trees, stream processors and cep engines and rated each against the criteria to see how well they perform.

Have a look at the results in Part 3 of this eBook.

Part 3
Rules Engine Benchmark For Iot

PART 3: RULES ENGINES BENCHMARK FOR IOT

INTRODUCTION 29

HOW TO READ THIS GUIDE 30

BENCHMARK CRITERIA 32

FORWARD CHAINING ENGINES 33
An engine using forward chaining (FC) applies a set of rules and facts to deduce conclusions, searching the rules until it finds one where the IF clause is known to be true.

CONDITION ACTION ENGINES 37
Condition-action engines are a sub-class of Forward Chaining engines that are using only one conditional statement as the IF clause.

FLOW PROCESSING ENGINES 40
Flow based programming (FBP) uses “black-box” functions represented as nodes, that exchange data across predefined connections by message passing.

DECISION TREES / DECISION TABLES 44
Decision trees use a branching method to illustrate every possible outcome of a decision.

STREAM PROCESSING ENGINES 48
Stream processing is the processing of data in motion. In other words, computing on data directly as it is produced or received.

CEP ENGINES 52
CEP engines differ from stream processing engines in the way they handle events and are mostly deployed in edge computing.

FINITE STATE MACHINES 55
A state machine can be used to describe a system in terms of a set of states that it goes through. Business rules engines (BRE) are an example of finite state machines.

BAYESIAN ENGINES 58
Rules engines based on Bayesian networks allow both backward and forward inference (state propagation).

CONCLUSION 63

INTRODUCTION

IoT application design is quite different from typical IT solutions, in that it bridges the physical world of Operations Technology (OT) with sensors, actuators and communication devices, and the digital world of Information Technology (IT) with data, analytics and workflows.

IN AN ENTERPRISE environment, IoT is very complex, not only because IoT deployments in large organisations will almost certainly need to quickly scale to thousands and then hundreds of thousands of devices (or sensors) and more, but also because the solution needs to work across all other enterprise systems and comply with specific enterprise software requirements.

This bridging of two worlds has important and unique consequences over how business logic and business rules are built within the IoT application. PART II of this eBook is discussing different rules engine technologies that can be used in the IoT domain. The term rules engine is used quite loosely, to refer to automation technologies in general, not just typical business rules engines (BREs).

This guide is not a vendor comparison sheet. It considers automation technologies, not specific implementations of these technologies.

Short references to specific tools are made throughout the paper, such as Node-RED being referenced when discussing flow processing engines or ifttt.com being referenced when discussing condition-action engines. These references are only made to show the extent to which a specific implementation can differ from its generic technology category.

This guide is based on our team's experience of over 20 years in software automation technologies and maps the capabilities of the most common rules engine technologies to the requirements of IoT.

HOW TO READ THIS GUIDE

The main body of PART 3 is made up of detailed explanations for the scores that each technology has received.

THE GUIDE IS not meant to be read linearly, but rather should serve as a reference to check out the scores once you have identified which of the eight rules engines is the one that best suits your needs.

Here is how to do that:

- Think of one specific IoT use case and its key requirements.

- Choose which of the seven criteria for evaluating a rules engine are most relevant to your use case's requirements. E.g. One use case may not require modeling high order logic, while for another use case the time dimension may be completely irrelevant.

- Check out the table below to find the technology that scores highest for your chosen criteria.

- Once you find the rules engines that best suit your IoT use case, go to that section for a full explanation of the scores received.

Table

Rules Engine	Complex Logic	Modeling Time	Uncertainty	Explainability	Adaptability	Operability	Scalability
Forward Chaining	40%	0%	0%	20%	20%	20%	20%
Condition-Action (IFTTT)	0%	0%	0%	100%	80%	80%	80%
Flow Based Processing	20%	0%	0%	40%	60%	20%	80%
Decision trees	40%	0%	0%	60%	0%	0%	0%
Stream Processing	20%	40%	0%	20%	0%	20%	100%
Complex Event Processing	20%	40%	0%	20%	20%	20%	20%
Finite State Machines	20%	0%	0%	40%	40%	40%	40%
Waylay Engine	100%	100%	100%	100%	100%	100%	80%

BENCHMARK CRITERIA

TECHNOLOGY-SPECIFIC CRITERIA

1. Modeling complex logic
- Combining multiple non-binary outcomes of functions (observations) in the rule.
- Dealing with majority voting conditions in the rule.
- Handling conditional executions of functions based on outcomes of previous observations.

2. Modeling time
- Dealing with the past (handling expired or soon-to-expire information).
- Dealing with the present (combining asynchronous and synchronous information).
- Dealing with the future (outliers, time windows, fitting algorithms - forecasting for prediction and anomaly detection).

3. Modeling uncertainty
- Handling noisy data or missing data.
- Handling the utility function.
- Handling probabilistic reasoning (building logic based on the likelihoods of different outcomes for one given sensory output).

IMPLEMENTATION-SPECIFIC CRITERIA

4. Explainability
- The intent of the rule should be clear to all users, developers and business owners alike.
- Compact representation of logic.
- Simulation and debugging capabilities, during design time and at runtime.

5. Adaptability
- Flexibility (supporting changes, both technical and commercial).
- Extensibility (integrating with external systems).

6. Operability
- Templating to apply the same rule to multiple devices, or to similar use cases.
- Versioning of both templates and running rules, for snapshotting and rollbacks.
- Searchability to easily search rules by name, API in use, type of device, and other filters.
- Rules analytics to understand most triggered rules, most common actions taken, etc.
- Bulk upgrades for lifecycle mngmt across groups of rules, useful for updates or end-of-life.

7. Architectural Scalability (sharding and distributed computing)

Forward Chaining Engines

Part 3
Rules Engine Benchmark For Iot

An inference engine using forward chaining (FC) applies a set of rules and facts to deduce conclusions, searching the rules

UNTIL IT FINDS one where the IF clause is known to be true. The process of matching new or existing facts against rules is called pattern matching, which FC inference engines perform through various algorithms, such as Linear, Rete, Treat, Leaps etc.

When a condition is found to be TRUE, the engine executes the THEN clause, which results in new information being added to its dataset. In other words, the engine starts with a number of facts and applies rules to derive all possible conclusions from those facts. This is where the name "forward chaining" comes from - the fact that the inference engine starts with the data and reasons its way forward to the answer, as opposed to backward chaining, which works the other way around.

In backward chaining, the system works from conclusions backwards towards the facts, an approach called goal driven. Compared to forward chaining, few data are asked, but many rules are searched. We have made a conscious choice not to consider backward-chaining rules in this benchmark as they are not suited for dynamic situations and are mostly only used as expert systems in decision making.

1. Modeling complex logic (40/100)

- Combining multiple non-binary outcomes of functions (observations) in the rule.
- Dealing with majority voting conditions in the rule.
- Handling conditional executions of functions based on outcomes of previous observations combining multiple non-binary outcomes of functions (observations) in the rule is not possible, since conditions are applied on Boolean (true/false) outcomes.

FC engines "collapse" on the majority voting requirement almost immediately, since they search inference rules until they find one or multiple where the IF clause is known to be true. This means that several, potentially contradicting rules may fire at the same time and the engine needs to deal with conflict resolutions to decide which one to execute. Adding majority voting into this mix is too much to handle.

Conditional executions of functions based on the outcomes of previous observations is not easy, as FC rules engines expect all data to be present at the moment rules are evaluated. We still give them a score of 40 out of 100 as they provide a good framework for expressing conditional (Boolean) logic.

2. Modeling time (0/100)

- Dealing with the past (handling expired or soon-to-expire information).
- Dealing with the present (combining asynchronous and synchronous information).
- Dealing with the future (outliers, time windows, fitting algorithms - forecasting for prediction and anomaly detection).

FC engines are incapable of expressing the time dimension in a rule - all rules modeled by forward chaining feel as if they are frozen in time.

3. Modeling uncertainty (0/100)

- Handling noisy data or missing data.
- Handling the utility function.
- Handling probabilistic reasoning (building logic based on the likelihoods of different outcomes for one given sensory output).

FC engines are incapable of expressing uncertainty or utility functions within a rule.

4. Explainability (20/100)

- The intent of the rule should be clear to all users, developers and business owners alike.
- Compact representation of logic.
- Simulation and debugging capabilities.
- during design time.
- at runtime.

For simple problems, FC engines provide us with an easy way to design rules. In fact, there is nothing easier to grasp than "if this then that" type of rules! However, adding more conditional statements into a rule leads to very complex analyses, which hinder the understanding of the rule's intent.

Moreover, the actual conditions of the rules, which often include thresholds and other Boolean expressions, are written and buried somewhere in the code, and are as such difficult to expose to the outside observer. As a work-around, during the design phase, rules are often represented as graphs with conditional outcomes modeled as labeled "arrows". However, these graphs are nowhere to be seen or inspected once the rule is implemented.

Simulation, debugging and decision tracking (why has the rule fired at runtime) is not a trivial task, since the data determines which rules' paths are selected and used. Moreover, as described earlier, the conflict resolution requires a prior selection of the conflict resolution strategy, which is not part of the rule but often a configuration parameter of the rules engine.

5. Adaptability (20/100)

- Flexibility (supporting changes, both technical and commercial).
- Extensibility (integrating with external systems).

Changing rules is possible but always problematic, as conflict resolution needs to be re-evaluated every time a condition in the rule changes.

Adding third-party API services to forward chaining engines is not a straightforward task and it is often accomplished directly in the code, leading to the API endpoints being directly coupled at the rule level. Since thresholds and other conditions are also often defined in the code, it is difficult to reuse the same API services across multiple instantiated rules.

6. Operability (20/100)

- Templating to apply the same rule to multiple of devices, or to similar use cases.
- Versioning of both templates and running rules, for snapshotting and rollbacks.
- Searchability to easily search rules by name, API in use, type of device and other filters.
- Rules analytics to understand most triggered rules, most common actions taken etc..
- Bulk upgrades for lifecycle mngt across groups of rules, useful for updates or end-of-life.

Applying the same rule across many devices is possible as long as thresholds and other conditions do not change across devices. Anything more complicated is extremely difficult to achieve with FC, since many inputs to the rules are buried deeply inside the code.

7. Architecture Scalability (sharding and distributed computing) (20/100)

Forward chaining rules are stateless, which means that you can easily run multiple rules in parallel, but you can not distribute the load to different processes while executing one instance of a rule.

Condition-action Engines

Part 3
Rules Engine Benchmark For Iot

Although we can argue that Condition-Action based (CA) rules engines belong to the group of Forward Chaining engines, we have decided to evaluate them as a separate category, since many of the general FC comments don't apply to them.

THE REASON FOR this is that Condition-Action rules engines don't allow multiple conditions, which makes them on one hand very limited in their logic expression capabilities and on the other hand - much more scalable. Condition-Action rules engines (if this - then that) are sometimes extended with the ELSE statement (if this - then that - else - that).

1. Modeling complex logic (0/100)

- Combining multiple non-binary outcomes of functions (observations) in the rule.
- Dealing with majority voting conditions in the rule.
- Handling conditional executions of functions based on outcomes of previous observation.

Unlike FC engines, CA engines can not model any complex logic (combining multiple non-binary outcomes, majority voting, conditional executions). They are meant to be used for very simple use cases.

2. Modeling time (0/100)

- Dealing with the past (handling expired or soon-to-expire information).
- Dealing with the present (combining asynchronous and synchronous information).
- Dealing with the future (outliers, time windows, fitting algorithms - forecasting for prediction and anomaly detection).

One way these engines work around the time dimension problem is that they often rely on external triggers for determining which rule to execute. That is to say, the IF condition of a rule becomes the WHEN condition (e.g. When weather is bad, send an alarm; When I come home, turn on the lights). This is often referred to as a trigger in these tools, and even though we may argue that this is not something that is part of the rules engine per se (because it needs to be coded somewhere else), it is still obvious how the time dimension could to be introduced into the rules engine.

3. Modeling uncertainty (0/100)

- Handling noisy data or missing data.
- Handling the utility function.
- Handling probabilistic reasoning (building logic based on the likelihoods of different outcomes for one given sensory output).

CA rules engines are not capable of expressing uncertainty or utility functions within a rule.

4. Explainability (100/100)

- The intent of the rule should be clear to all users, developers and business owners alike.
- Compact representation of logic.
- Simulation and debugging capabilities.
    - during design time
    - at runtime

Just like forward chaining engines, CA engines like IFTTT provide us with an easy way of designing rules for simple problems. There is nothing easier to grasp than if this then that rules!

5. Adaptability (80/100)

- Flexibility (supporting changes, both technical and commercial).
- Extensibility (integrating with external systems).

CA engines are both flexible and extensible. Adding third-party API services is rather simple, as the API extensions require minimal abstractions (if and act part). However, due to the nature of CA engines, there are limitations w.r.t. the usage of API services within rules.

Most of the time, CA engines use API services as triggered actions, not as inputs, as there is a single conditional input slot available, which in IoT use cases is usually taken by the device data. A typical example would be, for instance, to call an external API service (SMS, email, support ticket etc) if a device temperature registers above 25.

When CA engines do model the data of an API service as an input (what the ifttt.com CA engine for example calls a trigger, e.g. “if it is going to rain”), then we cannot combine this API service input with device data, as the single input slot is taken, and we can only use the device in this case as the actuator (e.g. “turn on lights”).

6. Operability (80/100)

- Templating to apply the same rule to multiple of devices, or to similar use cases.
- Versioning of both templates and running rules, for snapshotting and rollbacks.
- Searchability to easily search rules by name, API in use, type of device and other filters.
- Rules analytics to understand most triggered rules, most common actions taken etc.
- Bulk upgrades for lifecycle mngt across groups of rules, useful for updates or end-of-life.

Applying the same rule across many devices is possible as long as thresholds and all the other conditions do not change. Templating, versioning and searchability are rather easy to achieve with such rules engines but bulk upgrades are more difficult, as conditions and thresholds are often global variables and hard to change per instance of the running rule.

7. Architecture scalability (sharding and distributed computing) (80/100)

CA rules are stateless and very simple, so it is very easy to scale these rules engines. However, they do not get the maximum score for this category, as scalability is truly achieved by only one rules engine category, namely the stream processing engines.

Flow Processing Engines

Part 3
Rules Engine Benchmark For Iot

Flow based programming (FBP) is a programming paradigm that defines applications as networks of "black box" processes.

THESE PROCESSES, A.KA. functions, are represented as nodes that exchange data across predefined connections by message passing. The nodes can be reconnected endlessly to form different applications without having to change their associated functions.

FBP is thus naturally "component-oriented." Some of the benefits of FBP are:

- Change of connection wiring without rewriting components.
- Inherently concurrent - suited for the multi-core CPU world.

Yahoo! Pipes and Node-RED are two examples of rules engines built using the FBP paradigm. FBP has become even more popular with the introduction of "serverless" computing, where cloud applications can be built by chaining functions.

IBM's OpenWhisk is an example of flow based programing by chaining cloud functions (which IBM calls actions). Another serverless orchestration approach, based on finite state machine rules engines, such as AWS step functions, is discussed later.

1. Modeling complex logic (15/100)

- Combining multiple non-binary outcomes of functions (observations) in the rule.
- Dealing with majority voting conditions in the rule.
- Handling conditional executions of functions based on outcomes of previous observations.

FBP has no notion of states and state transitions. Combining multiple non-binary outcomes of functions (observations) in the rule is still possible, but must be coded in every function where it is applied. That also implies that you have to branch at every function where you need to model a multiple-choice outcome. This leads to extremely busy flow graphs that are hard to follow, especially since logic is expressed both in the functions themselves and in their "connectors" - path executions. These connectors somehow suggest not only the information flow but also the decisions that are being taken.

Similar to decision trees (which are discussed further on), such an approach for modelling suffers from an exponential growth of the number of nodes, as the complexity of the logic increases. What makes the matter even worse is that, unlike in decision trees, we cannot track the function outcomes as states. There is no better illustration of this drawback than to look at a slightly more complex flow being implemented using node-RED, and count the number of nodes and connectors. It is not unusual to have simple use cases designed by node-RED with 30 or 40 nodes and connectors, which can hardly even fit on one screen.

Majority voting in flow engines is possible only if we introduce the concept of merging the outputs of different nodes into a separate merge node. Even so, it's still problematic, as it requires to code majority rules within the function of that merge node.

Conditional executions of functions based on outcomes come out of the box.

2. Modeling time (10/100)

- Dealing with the past (handling expired or soon-to-expire information).
- Dealing with the present (combining asynchronous and synchronous information).
- Dealing with the future (outliers, time windows, fitting algorithms - forecasting for prediction and anomaly detection).

Flow engines can barely deal with any aspect of the time dimension, since FBP is by design a stateless rules engine. In some limited use cases (which can hardly scale) you can merge streams within a time window.

3. Modeling uncertainty (0/100)

- Handling noisy data or missing data.
- Handling the utility function.
- Handling probabilistic reasoning (building logic based on the likelihoods of different outcomes for one given sensory output).

FBP rules are not capable of expressing uncertainty or utility functions within a rule.

4. Explainability (35/100)

- The intent of the rule should be clear to all users, developers and business owners alike.
- Compact representation of logic.
- Simulation and debugging capabilities.
    * during design time
    * at runtime

For simple use cases, a flow based data stream representation feels natural, at least from the perspective of the information flow. But any attempt to create complex logic using FPB makes validating the intended logic very difficult.

Having said that, understanding which decisions are taken by looking at the flow graph is a very difficult task. The main reason for this is that the logic representation is not compact and the validation of the rules often requires streaming test data, followed by the validation of the function logs across all pipelines.

The logic is split between the flow pathways (as data travels between processing nodes) and the payload processing in each node, which might lead to different paths being taken after that processing node. Hence debugging and rules validation becomes a very tedious and error prone process. Moreover, we are never sure that all corner cases (the outputs as decisions from different inputs) are covered by a particular rule expressed using FBP - it looks almost as FBP based rules validation is an NP hard problem.

5. Adaptability (60/100)

- Flexibility (supporting changes, both technical and commercial).
- Extensibility (integrating with external systems).

FBP engines have reusable black box nodes (functions). However, a partial update of any particular rule is nevertheless difficult and risky because this usually implies major changes to the graph and revalidation of the rules.. In a way, the main reason for this is that for most rules engines, and for FBP in particular, there is a high correlation between explainability and flexibility. Flow based rules engines are easy to extend with third-party services and extensibility is achieved in an elegant way.

6. Operability (20/100)

- Templating to apply the same rule to multiple of devices, or to similar use cases.
- Versioning of both templates and running rules, for snapshotting and rollbacks.
- Searchability to easily search rules by name, API in use, type of device and other filters.
- Rules analytics to understand most triggered rules, most common actions taken etc.
- Bulk upgrades for lifecycle mngt across groups of rules, useful for updates or end-of-life.

Templating is very difficult to achieve, since special care needs to be taken when handling payload transformations that happen as payloads are passed between different processing nodes. Also, thresholds and branching logic are part of the same payloads processing flow, making it very hard to abstract this logic. It's for this same reason that bulk upgrades are error prone and risky.

.

7. Architecture scalability (sharding and distributed computing) (75/100)

FBP engines are inherently concurrent since they have to distribute functional computations. They are also stateless, which means that the rules engine only needs to keep track of the current execution and further actions that need to be executed. On the other hand, if merging multiple outputs of different nodes is required in one rule, or when decision branching is introduced with different path executions, the rules engine will need to keep the snapshot (scope) of the rules execution somewhere.

Side note on flow engines: Node-RED

Operability (0/100)

Node-RED suffers from bigger operability issues than FBPs. The main reason is that its authors have chosen to let different protocol streams come directly into nodes as input data events. This was done deliberately in order to simplify protocol termination and to allow payload normalization being performed within node-RED. But it's a decision that acts as a double-edged sword.

On the one hand, it means that protocol-dependent data streams can be implemented by any third-party and immediately used within the node-RED environment. It's why node-RED is today very popular in the maker community and why it is the de-facto tool in the gateways of many industrial vendors. As protocol transformation and payload normalization are very important in IoT deployments, node-RED can be very valuable for edge deployments.

On the other hand, that same decision makes templating an order of magnitude more difficult: protocol transformation and payload normalization need to be part of the node-RED template, together with threshold definitions and branching.

Architecture Scalability (sharding and distributed computing) (75/100)

Though a good fit for edge deployments, an off-the-shelf Node-RED instance is not scalable for the cloud. Some vendors provide cloud solutions with sharding implemented on top of node-RED and by externalizing the protocol termination in a separate component. However, when taking such an approach they could as well switch back to the more generic FPB engines.

Decision Trees/ Decision Tables

Part 3
Rules Engine Benchmark For Iot

A popular way of capturing the complexity of conditional rules is by using decision trees, which are graphs that use a branching method to illustrate every possible outcome of a decision.

THERE ARE SEVERAL products on the market that offer rules engines based on decision trees/ tables. Drools, mostly known for its rules engine based on forward chaining, also has an extension to integrate with decision tables, using an excel sheet in combination with snippets of embedded code to accommodate any additional logic or required thresholds.

1. Modeling complex logic (30/100)

- Combining multiple non-binary outcomes of functions (observations) in the rule.
- Dealing with majority voting conditions in the rule.
- Handling conditional executions of functions based on outcomes of previous observations.

Decision trees are useful when the number of states per each variable is limited (such as binary YES/NO states) but can become overwhelming when the number of states increases. This is because the depth of the tree grows linearly with the number of variables, but the number of branches grows exponentially with the number of states. For instance, with 6 Boolean variables, there are 2^2^6 = 2^64 = 18,446,744,073,709,551,616 distinct decision trees (in literature, often referred to as the "hypothesis space for decision trees" problem).

Majority voting is not possible, unless we branch even further, where multiple distinct outcomes are also part of the tree structure. Conditional executions should come out of the box. As the name suggests, decision trees are all about conditional executions. Having said that, decision trees are never implemented as such in an IoT context. In expert systems, where decisions are outcomes of Q&A scenarios, logic would follow conditional execution, as new data (questions) is served to the decision tree engine. On the other hand, in an IoT context, we feed rules engines with data, and expect decisions to come back as a result. In that case, we talk about decision tables, which means we feed data into the decision tables and results (decisions) come back at once. More about this not so subtle difference between tables and trees can be found here. We still give decision trees a score of 30 out of 100, since interpretability is what makes them very attractive in use cases where this capability is essential (such as healthcare, among others).

2. Modeling time (0/100)

- Dealing with the past (handling expired or soon-to-expire information).
- Dealing with the present (combining asynchronous and synchronous information).
- Dealing with the future (outliers, time windows, fitting algorithms - forecasting for prediction and anomaly detection).

We can not model the time dimension with decision trees, unless we include time information as nodes within the tree (e.g. weekend, day of week, time of day etc). Even so, we can not use time as a means of expressing change in our underlying observations, so none of these are possible: dealing with the past (handling expired or soon-to-expire information), dealing with the present (combining asynchronous and synchronous information), dealing with the future (forecasting for prediction and anomaly detection).

3. Modeling uncertainty (10/100)

- Handling noisy data or missing data.
- Handling the utility function.
- Handling probabilistic reasoning (building logic based on the likelihoods of different outcomes for one given sensory output).

Decision trees use a white box model. Important insights can be generated based on domain experts describing a situation and their preferences for outcomes. But decision trees are unstable, meaning that a small change in the data can lead to a big change in the structure of the optimal decision tree. They are also often relatively inaccurate. Calculations can get very complex, particularly if many values are uncertain and/or if many outcomes are linked. Decision trees cannot model uncertainty and utility functions, unless, just like with time information, we add these within the tree as decision nodes, which complicates decision tables even further.

4. Explainability (70/100)

- The intent of the rule should be clear to all users, developers and business owners alike.
- Compact representation of logic.
- Simulation and debugging capabilities.
    * during design time
    * at runtime

Decision trees are easy to understand and interpret. People are able to understand decision tree models after just a brief explanation. Still, decisions cannot be seen or inspected once the rule is instantiated and are only represented as labeled “arrows” in the graph during the design phase. When implemented as decision tables, the explainability drops further as each row in the table is a rule with each column in that row being either a condition or action for that rule. That results in the total sequence not being clear - no overall picture is given by decision tables.

5. Adaptability (0/100)

- Flexibility (supporting changes, both technical and commercial).
- Extensibility (integrating with external systems).

Decision trees are mostly used for graphical knowledge representation. It is extremely hard to build a rules engine with decision trees and even harder to build applications on top of it. They are hard to extend with any third-party systems. Also, any small change in the training data can lead to a big change in the structure of the optimal decision tree.

6. Operability (0/100)

- Templating to apply the same rule to multiple of devices, or to similar use cases.
- Versioning of both templates and running rules, for snapshotting and rollbacks.
- Searchability to easily search rules by name, API in use, type of device and other filters.
- Rules analytics to understand most triggered rules, most common actions taken etc.
- Bulk upgrades for lifecycle mngmt across groups of rules, useful for updates or end-of-life.

Applying the same decision tree rule across multiple devices is close to impossible, as most of the decision trees implement rules by mixing logic residing in decision tables with actions defined separately in code, making it extremely difficult to manage the complete process.

7. Architecture scalability (sharding and distributed computing) (10/100)

Decision tree rules are stateless, which means that, in theory, it should be easy to run multiple rules in parallel. However, you cannot, within one instance of a rule, distribute the load to different processes while executing that one particular rule. The fact that the depth of the tree grows linearly with the number of variables but the number of branches grows exponentially with the number of states makes decision trees hard if not impossible to scale. Calculations can get very complex, particularly if many values are uncertain and/or if many outcomes are linked.

Stream Processing Engine

Part 3
Rules Engine Benchmark For Iot

Stream processing is the processing of data in motion- in other words, computing on data directly as it is produced or received (as opposed to map-reduce databases such as Hadoop, which process data at rest).

BEFORE STREAM PROCESSING emerged as a standard for processing continuous datasets, these streams of data were often stored in a database, a file system, or some other form of mass storage. Applications would then query the stored data or compute over the data as needed. One notable downside of this approach (broadly referred to as batch processing) is the latency between the creation of data and the use of data for analysis or action.

In most stream processing engines users have to write code to create operators, wire them up in a graph and run them. Then the engine runs the graph in parallel. Examples of stream processing engines are Apache Storm, Flink, Samza etc.

Upon receiving an event from a data stream, a stream processing application reacts to the event immediately. The application might trigger an action, update an aggregate, or “remember” the event for future use.

Stream processing computations can also handle multiple data streams jointly, and each computation over the event data stream may produce other event data streams.

Stream processing engines have a narrow usage in IoT - for runtime processing of IoT data streams. They are not designed as a generic rules engine and e.g. cannot actuate back on devices directly.

Stream processing rules engines are typically used for applications such as algorithmic trading, market data analytics, network monitoring, surveillance, e-fraud detection and prevention, clickstream analytics and real-time compliance (anti-money laundering).

1. Modeling complex logic (25/100)

- Combining multiple non-binary outcomes of functions (observations) in the rule.
- Dealing with majority voting conditions in the rule.
- Handling conditional executions of functions based on outcomes of previous observations.

No high order logic constructions (combining multiple non-binary outcomes, majority voting, conditional executions) are possible with stream rules engines. However, developers can run StreamSQL on top of the datastreams, where simple thresholds together with aggregation across all streams or certain stream subsets can bring great value for some use cases.

2. Modeling time (30/100)

- Dealing with the past (handling expired or soon-to-expire information).
- Dealing with the present (combining asynchronous and synchronous information).
- Dealing with the future (outliers, time windows, fitting algorithms - forecasting for prediction and anomaly detection).

Stream processing engines cannot cope with synchronous and asynchronous events in the same rule. This means that we can't intercept the stream data and at the same moment call an external API service, while executing the rule. Stream processing engines are designed to focus on the high throughput stream execution, which would, for any API call that has a big round-trip delay for a given event, simply break the processing pipeline.

Still, stream processing engines have a very powerful query language - StreamSQL. StreamSQL queries over streams are generally "continuous", executing for long periods of time and returning incremental results. These operations include: Selecting from a stream, Stream-Relation Joins, Union and Merge and Windowing and Aggregation operations.

3. Modeling uncertainty (0/100)

- Handling noisy data or missing data.
- Handling the utility function.
- Handling probabilistic reasoning (building logic based on the likelihoods of different outcomes for one given sensory output).

Stream processing rules engines are not capable of expressing uncertainty or utility functions within a rule.

4. Explainability (15/100)

- The intent of the rule should be clear to all users, developers and business owners alike.
- Compact representation of logic.
- Simulation and debugging capabilities.
    * during design time
    * at runtime

Unless you are a developer and familiar with Stream SQL, it is impossible as a user to understand the behaviour of any particular rule. We can argue the same for any typical SQL-based solution, hence we give it an overall score of 20 out of 100.

5. Adaptability (10/100)

- Flexibility (supporting changes, both technical and commercial).
- Extensibility (integrating with external systems).

API extensions and overall flexibility are weak points of these rules engines. Stream processing engines are data processing pipelines, not meant to be directly integrated with third-party API systems.

6. Operability (10/100)

- Templating to apply the same rule to multiple of devices, or to similar use cases.
- Versioning of both templates and running rules, for snapshotting and rollbacks.
- Searchability to easily search rules by name, API in use, type of device and other filters.
- Rules analytics to understand most triggered rules, most common actions taken etc.
- Bulk upgrades for lifecycle mngt across groups of rules, useful for updates or end-of-life.

In many IoT stream processing use cases, stream processing is used for global threshold crossing (e.g. send an alarm if temperature of any event is above a threshold) or aggregations (e.g. average temperature in a given region) but any more complicated calculation or per device threshold crossing is extremely hard to achieve. This is why templating, updating rules per device or version updates are very difficult.

7. Architecture Scalability (sharding and distributed computing) (100/100)

When it comes to real-time large-volume data processing capabilities, nothing can beat stream processing engines.

CEP ENGINES

Part 3
Rules Engine Benchmark For Iot

Although part (and predecessors) of stream processing engines, Complex Event Processing engines deal with events in a slightly different (and better) way than their bigger and younger siblings.

WE SEE CEP ENGINES being deployed in edge computing nowadays, where locality, low latency and low hardware footprint are important. CEPs are a good fit whenever a low footprint is required, but don't scale well since all event processing happens in-memory.

WSO2, Litmus Automation and Foghorn are examples of vendors offering CEP rules engines for edge computing.

1. Modeling complex logic (35/100)

- Combining multiple non-binary outcomes of functions (observations) in the rule.
- Dealing with majority voting conditions in the rule.
- Handling conditional executions of functions based on outcomes of previous observations.

Arguably, high order logic constructions (Combining multiple non-binary outcomes, Majority voting, Conditional executions) are possible, but with a lot of difficulty and coding effort, since CEP engines are not designed with these features in mind.

2. Modeling time (45/100)

- Dealing with the past (handling expired or soon-to-expire information).
- Dealing with the present (combining asynchronous and synchronous information).
- Dealing with the future (outliers, time windows, fitting algorithms - forecasting for prediction and anomaly detection).

Often CEP engines have built-in operators such as time windows and temporal event sequences integrated into their query language. CEP engines, like Stream Processing engines, can't cope with async and sync events in the rule. They also have difficulty dealing with the "past" - meaning invalidating events after a given period of time. Compared to Stream Processing engines however, they often have better capabilities for pattern matching, which enables better anomaly detection at runtime, hence we give them a better score, as this is one of the stronger points of CEP engines.

3. Modeling uncertainty (0/100)

- Handling noisy data or missing data.
- Handling the utility function.
- Handling probabilistic reasoning (building logic based on the likelihoods of different outcomes for one given sensory output).

CEP engines are not capable of expressing uncertainty or utility functions within a rule.

4. Explainability (15/100)

- The intent of the rule should be clear to all users, developers and business owners alike.
- Compact representation of logic.
- Simulation and debugging capabilities.
    * during design time
    * at runtime

CEP engines rules are hard to reason about, since all logic is buried somewhere deeply in the code.

5. Adaptability (10/100)

- Flexibility (supporting changes, both technical and commercial).
- Extensibility (integrating with external systems).

Flexibility is a weak point of these rules engines but, compared to stream processing engines, it ranks better for extensibility since one can still imagine better API integration capability, mostly in the actionable part (send SMS if something goes wrong).

6. Operability (10/100)

- Templating to apply the same rule to multiple of devices, or to similar use cases.
- Versioning of both templates and running rules, for snapshotting and rollbacks.
- Searchability to easily search rules by name, API in use, type of device and other filters.
- Rules analytics to understand most triggered rules, most common actions taken etc.
- Bulk upgrades for lifecycle mngt across groups of rules, useful for updates or end-of-life.

Similar to Stream Processing engines, in anything beyond simple use cases, operability is extremely hard to achieve since templating, updating rules per device or version updates are very difficult.

7. Architecture Scalability (sharding and distributed computing) (100/100)

CEPs are a good fit when a low footprint is required but suffer from scalability issues because of the lack of distributed computing capabilities and because they process all data in-memory.

Finite State Machine

Part 3
Rules Engine Benchmark For Iot

of a set of states that the system goes through. A state isa description of the status of a system that is waiting toexecute a transition. A transition is a set of actions to be executed when a condition is fulfilled or when an event is received.

1. Modeling complex logic (10/100)

- Combining multiple non-binary outcomes of functions (observations) in the rule.
- Dealing with majority voting conditions in the rule.
- Handling conditional executions of functions based on outcomes of previous observations.

Finite state machines are modeling simple relations, aka transitions from one state to the other, and are mostly used to model business processes. Combining multiple non-binary outcomes and majority voting is not possible with FSM engines. Conditional executions is all they can do (conditions are defined for each transition).

2. Modeling time (0/100)

- Dealing with the past (handling expired or soon-to-expire information).
- Dealing with the present (combining asynchronous and synchronous information).
- Dealing with the future (outliers, time windows, fitting algorithms - forecasting for prediction and anomaly detection).

FSM engines are not capable of expressing time in the rule, unless we talk about state transitions based on time of day / week / month.

3. Modeling uncertainty (0/100)

- Handling noisy data or missing data.
- Handling the utility function.
- Handling probabilistic reasoning (building logic based on the likelihoods of different outcomes for one given sensory output).

FSM engines are not capable of expressing uncertainty or utility functions within a rule.

4. Explainability (50/100)

- The intent of the rule should be clear to all users, developers and business owners alike
- Compact representation of logic.
- Simulation and debugging capabilities.
- during design time.
- at runtime

The concept of FSM is easy to grasp by different types of users. The main selling argument of BREs (Business Rules Engines) is that BRE software allows non-programmers to implement business logic in a business process management (BPM) system.

One thing often overlooked with FSM is that states imply transitions, that is to say, the only purpose of having something modeled as a state is to navigate a particular decision flow.

A direct result of that is that FSM lacks readability as rules become more complex, or when a particular corner case needs to be modeled as a state. Since FSM is capable of executing only one transition at a time, when a user tries to introduce events that might happen under certain conditions, she needs to add a new state. When the number of states becomes too large, the readability of the state machine drops significantly.

5. Adaptability (40/100)

- Flexibility (supporting changes, both technical and commercial).
- Extensibility (integrating with external systems).

Adding third-party API services is quite easy as the API extension requires minimal abstractions (conditional outcomes on any given input that resolves in one of the available set of states). Flexibility is not a forte however, because it is very difficult to change rules once they are implemented.

6. Operability (45/100)

- Templating to apply the same rule to multiple of devices, or to similar use cases.
- Versioning of both templates and running rules, for snapshotting and rollbacks.
- Searchability to easily search rules by name, API in use, type of device and other filters.
- Rules analytics to understand most triggered rules, most common actions taken etc.
- Bulk upgrades for lifecycle mngt across groups of rules, useful for updates or end-of-life.

Applying the same rule across many devices is possible as long as thresholds and all other conditions do not change. Templating and searchability is quite easy to achieve with such rules but versioning and performing bulk upgrades is harder, as conditions and thresholds are often global variables and hard to change per instance of the running rule.

7. Architecture Scalability (sharding and distributed computing) (50/100)

Like FBP engines, FSM engines can distribute functional computations (state executions). On the other hand, within one rule, all executions are sequential. FSM are not stateless, which means that the rules engine needs to keep track of the current rules executions and apply transitions after every function call to delegate to the next node.

Bayesian Engines

Part 3
Rules Engine Benchmark For Iot

The Waylay IoT rules engine is an inference engine based on Bayesian Networks (BN). It allows both backward and forward inference (state propagation) which enables both push (data streams) and pull modes (API synchronous calls) to be treated as first class citizens.

The Waylay IoT rules engine provides three important abstractions on top of off-the-shelf BNs:

- The Waylay IoT rules engine models rules using the Smart Agent concept that consists of Sensors, Logic and Actuators. It decouples logic from sensing and actuation. As a result, sensors and actuators can easily be reused across rules.

- The Waylay IoT rules engine models joint relations of variables (sensors) through simplified Conditional Probability Tables (CPT) and allows very simple compact logic representation, further enhanced by using the DAG model.

- The Waylay IoT rules engine models the information, control and decision flows independently. This empowers the rules designer to have full control over the rules execution.

The Waylay rules engine has the following key characteristics:

• Unlike flow rules engines, there is no left-right input/output logic. The information flow happens – in all directions, all the time.

• Unlike flow rules engines, the Waylay rules engine does not need "injector nodes" or "split/merge" input/outputs nodes in order to deal with multiple possible outcomes.

• Unlike forward chaining algorithms or decision trees, the Waylay rules engine does not model logic by branching all possible outcomes.

• When modeling conditions of multiple variables with multiple states, the Waylay rules engine doesn't suffer from an exponential explosion of the graph size like decision trees.

• The Waylay rules engine is a state propagation engine similar to FSM, but unlike FSM, it allows multiple state transitions to happen at the same time.

• Similar to forward chaining, the Waylay rules engine allows modeling multiple conditions, but the decision process is not guided by pattern matching of all conditions.

• The Waylay rules engine can model likelihoods.

• Unlike any of the other technologies discussed here, the Waylay rules engine models the information flow, control flow and decision flow independently.

1. Modeling complex logic (100/100)

- Combining multiple non-binary outcomes of functions (observations) in the rule.
- Dealing with majority voting conditions in the rule.
- Handling conditional executions of functions based on outcomes of previous observations.

The Waylay rules engine combines multiple non-binary outcomes of functions (observations) in a rule, beyond Boolean true/false states.

The combination of variables is simplified by means of aggregation nodes, that also provide a compact representation of logic. The relation between variables and their states is expressed via conditional probability tables (CPT). Conditional dependencies are expressed using gates with "simplified" CPTs (with zeros and ones only). The Waylay rules engine defines three types of gates: AND, OR and GENERAL. Event though the first two (AND, OR) resemble Boolean Logic there are two important differences: All gates can be attached to a "non-binary" sensor (a sensor having more than two states) Not all sensors need to be observed in order to have the gate state with posterior probability 1.

The GENERAL gate allows modeling both a combination of multiple sensor outcomes and majority voting at the same time.

More about this feature can be found here

The Waylay rules engine handles conditional executions of functions based on the outcomes of previous observations by decoupling the information from the control flow. For instance, one can create a rule in which the execution of certain sensors depends on the outcome of others. Conditional execution of functions can also be triggered by the state transitions of the attached sensor. An example of such a rule can be found here.

2. Modeling time (100/100)

- Dealing with the past (handling expired or soon-to-expire information).
- Dealing with the present (combining asynchronous and synchronous information).
- Dealing with the future (outliers, time windows, fitting algorithms - forecasting for prediction and anomaly detection).

The Waylay rules engine handles the past (expired or soon-to-expire information) via the concept of eviction time. The eviction time defines the time after which a sensor goes back to its priors. For example, if a sensor has N states, the system will assume by default that the sensor is with a probability of 1/N in each of the N states, after the eviction time. If the eviction time is not defined, the sensor will never go back to its priors.

Eviction time is useful when dealing with broken sensors (e.g. due to flat batteries), intermittent connectivity or non-responsive APIs. It allows you to specify the period of time during which you can still rely on information from previous observations. It also provides an elegant way of merging streams from different sensors, such in the case of motion sensors, where we can imagine a rule needing to trigger an action only if we have motion registered from multiple sensors within the same time window.

The Waylay rules engine also has an embedded CEP engine (called a Formula node), which takes as input raw sensor measurements (not just sensor states). The CEP engine applies complex statistical formulas, aggregation in time or on number of samples or search for a particular pattern (state changes).

Dealing with the present (combining asynchronous and synchronous information) comes out of the box, there is no extra effort needed from developers in order to use this feature.

With most rules engines, API integration is done on the actuation side (for example - send as SMS when a certain threshold is reached). In the Waylay rules engine, APIs can easily be used as inputs in the rules as well, for example, when combining weather data from an API call with stream data from a sensor. This is an important feature for use cases where the locality of information is very important.

When it comes to anomaly detection and predictions, the Waylay rules engine comes with a specific module called Time Series Analytics that provides advanced capabilities such as anomaly detection and prediction on the data stored in the Waylay Time Series Database. These capabilities are natively exposed in the Waylay rules engine, via the concept of sensors. The rule designer can make use of TSA and implement rules such as:
Send an SMS if the predicted energy consumption for the next 2 weeks is above a threshold.
Create a Zendesk ticket or send an email if the sensor battery will be at 20% in 2 weeks time.
Create an alarm if an anomaly detected.

It is important to mention that the fitting algorithm, anomaly definition and detection (whether we assume that the anomaly is an outlier, or something that doesn't follow the expected behavior, and whether we are concerned with every anomaly or we search for consecutive anomalies) and prediction intervals can all be separately modeled.

3. Modeling uncertainty (100/100)

- Handling noisy data or missing data.
- Handling the utility function.
- Handling probabilistic reasoning (building logic based on the likelihoods of different outcomes for one given sensory output).

The Waylay rules engine allows probabilistic reasoning by assigning actuators to fire when a node or a gate (relation between multiple nodes) is in a given state with a given probability. Moreover, you can as well associate different actuators to different outcome likelihoods for any node in the graph.

4. Explainability (100/100)

- The intent of the rule should be clear to all users, developers and business owners alike.
- Compact representation of logic.
- Simulation and debugging capabilities.
    * during design time
    * at runtime

The Waylay rules engine provides compact representation of logic. Combining two variables (which we refer to as sensors) is done via aggregation nodes. Waylay is also a state propagation engine, which allows an easy interpretation of the rules by following the changes of the states. It also allows easy debugging and simulation, even without data inputs, simply by following the state changes of any given sensor at design time.

Modeling rules using the Smart Agent concept (consisting of Sensors, Logic and Actuators) makes it easy to reuse the building blocks: the Sensors and Actuators. The Waylay rules engine provides a sandbox execution environment in which end users can easily create new sensors and actuators based on external APIs. Once created, these sensors and actuators can be easily shared between different rules.

6. Operability (100/100)

- Templating to apply the same rule to multiple devices, or to similar use cases.
- Versioning of both templates and running rules, for snapshotting and rollbacks.
- Searchability to easily search rules by name, API in use, type of device and other filters
- Rules analytics to understand most triggered rules, most common actions taken etc.
- Bulk upgrades for lifecycle mngt across groups of rules, useful for updates or end-of-life.

Sensors and actuators are versioned. When updating a (sensor or actuator) plug, a new version will be stored in the cloud. These new versions can then be reapplied to the running rules with zero downtime.

Templates are generic rules that have not yet been associated to a particular device or instance. All templates can be stored and shared using JSON representation, while all operations are exposed over APIs. The Waylay rules engine also comes with a rich provisioning API model, which models device relations as well as rules inheritance. That way, the same template can be instantiated many times as tasks, by associating device specific parameters to a specific template.

This mechanism is operationally very efficient in the sense that templates only need to be developed once, but can then be instantiated many times. As an example, assume you generate a template for an appliance and in the field, you have 100k appliances deployed: then you would have one template and 100k tasks running on the Waylay rules engine.

The admin console of the Waylay rules engine provides an inventory of all tasks currently running, as well as lifecycle management on a per task level: create, delete, start, stop, debug. In addition, an actuator log provides a historical overview of the actuators that have been triggered.

7. Architecture Scalability (sharding and distributed computing) (80/100)

The Waylay rules engine has three components: the inference engine (which controls the information, control and decision flows) the sandbox, where external APIs (sensors and actuators) get executed in stateless fashion, similar to lambda (cloud functions) architecture
the time series analytics engine.

All these components can be independently sharded, allowing horizontal scaling.

CONCLUSION

Rules engines are powerful automation SW tools that come in various shapes and flavours.

DIFFERENT TYPES OF engines were built to address different problems and some have overlapping functionality. Hence, it can be difficult to figure out which type of rules engine best suits the needs of your IoT use case. In order to assist you in the evaluation and decision process, we have defined a benchmark composed of seven core rules engine capabilities: modeling complex logic, modeling time, modeling uncertainty, explainability, adaptability, operability and scalability.

This ebook has evaluated and scored the seven most common types of rules engine technologies and our very own Waylay rules engine against this benchmark.

The benchmark results show that existing rules engine technologies have major shortcomings in one or multiple of the dimensions, which in the end may force developers to go back to code anyway.

The benchmark score card also shows how Waylay's rules engine empowers developers and integrators to implement the most demanding and complex automation use cases.

About the author

Veselin Pizurica is CTO and co-founder of Waylay. He designed and implemented products in various domains, including Cloud Computing, Semantic Web, Artificial Intelligence, IoT/M2M, Signal and Image Processing, Spherical Lattice Coding, Pattern Recognition, Home Networking, Mobile Management, MPLS Routing, XDSL Troubleshooting and Fiber Technologies.

Veselin holds 12 patents in Artificial Intelligence, DSL, Home Networks and Cloud Computing

About Waylay

Waylay, headquartered in Ghent, Belgium, operates globally with expanding teams in Europe, North America, and Asia. Our low-code solution revolutionizes automation, workflow, and complex rule integration within diverse software ecosystems. We serve diverse industries, including finance, telecom, manufacturing, connected assets, smart buildings, and logistics.

Automation Made Simple | www.waylay.io

waylay

Blekersdijk 33a, 9000 Gent, Belgium
www.waylay.io | info@waylay.io

© Copyright 2023 Waylay. All rights reserved.
Waylay is a registered trademark of Waylay NV.