waylay.io connect reason act

WHITEPAPER

How To Choose A Rules Engine

Seven Things To Look At When Automating for IoT

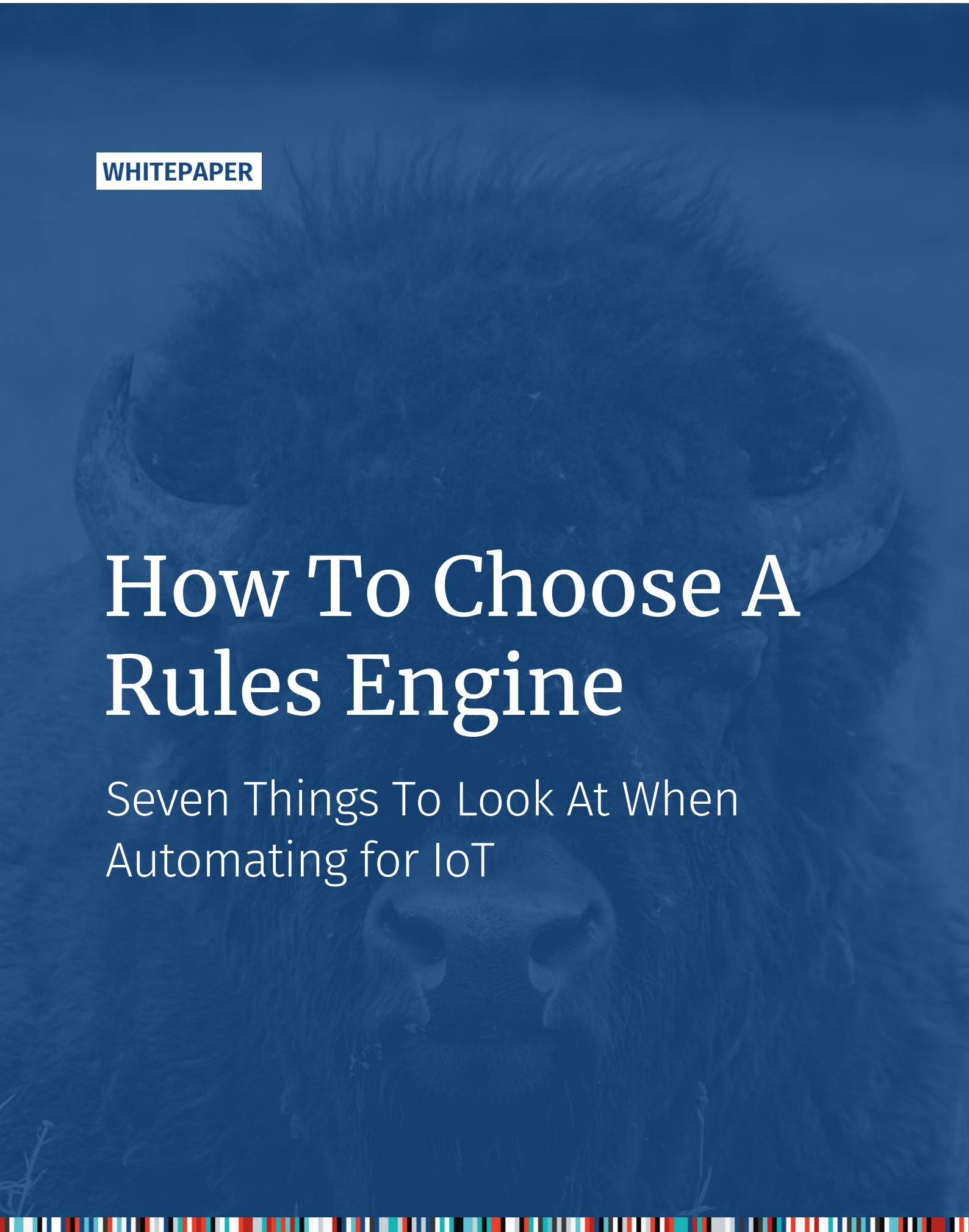


TABLE OF CONTENTS

Introduction 1

1. Logic 2

In this section, we introduce formal logic and see how computer expressed logic is different from human expressed logic and why developers have a difficult time translating user requirements into conditional statements (rules) when designing software.

2. Time 4

In this section, we introduce the time dimension and see how it complicates matters further for developers that are building logic with conditional statements (rules) that change over time.

3. Uncertainty 5

In this section, we introduce the uncertainty principle as an important element to building computer logic and we see how probabilities can affect conditional statements (rules) over time.

4. The Rules Engine 6

In this section we introduce the rules engine as the 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.

5. How to Evaluate a Rules Engine 7

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 key points against which a rule engine could be tested, in order to assess its success.

1. Modeling complex logic 7
2. Modeling time 8
3. Modeling uncertainty 9
4. Explainability 10
5. Adaptability 10
6. Operability 11
7. Architectural Scalability 11

Conclusion 12

Introduction

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 guide explains why using a rules engine from the get-go is preferable to hard coding rules (chapters one to three). It 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).

This guide is the first part of a two-part evaluation of automation technologies in the IoT domain. The second part: A Guide to Rules Engines, evaluates seven classes of rules engines against the benchmark defined here.

1. Logic

In this section, we introduce formal logic and see how computer expressed 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 $\wedge$, the disjunction OR denoted as $\vee$, and the negation NOT denoted as $\neg$.

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:

1. First, humans often use logical statements "incorrectly" when expressing rules using spoken language
2. 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.

2. Time

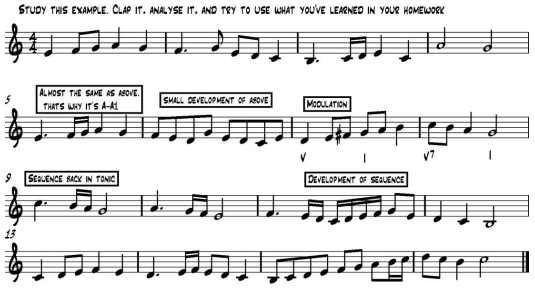
In this section, we introduce the time dimension and see how it complicates matters further for developers that are building logic with conditional statements (rules) that change over time.

"*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 smooth-flowing 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.

3. Uncertainty

In this section, we introduce the uncertainty principle as an important element to building computer logic and we see how 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 non-intrusive rules to achieve the same level of safety, with a rules-based 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.

4. The Rules Engine

In this section we introduce the rules engine as the 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 with in 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:

1. Information flow - the gathering of facts
2. Control flow - the order in which individual statements, instructions or function calls of an imperative program are executed or evaluated
3. 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.

5. How to Evaluate a Rules Engine

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.

1. 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:

a) **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.

b) 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.

c) 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."

2. Modeling time

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

a) 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)

b) 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*".

c) **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.

3. 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 dependent 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 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 very useful as inputs for the decision engine.

4. 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.

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

5. Adaptability

a) Flexibility

Things inevitably change, both on the business side (changing customer requirements for a particular family of devices) and on the technical side (API changes) and the rules engine should be flexible enough to support change with the least friction possible. This means that changing and updating rules should be easy and performing these changes at runtime should be possible with no service interruption or downtime.

Some rules engines are very sensitive to any change to the initial inputs or intended logic, requiring the complete rule to be rewritten and retested again. In rules-based decision trees, for example, any small change in the training data can lead to a big change in the structure of the optimal decision tree.

b) Extensibility

In order to account for future growth, the rule engine should be capable to support extensions and integration with external systems, such as third-party API services. Some rules engines are inherently easier to extend due to their “black box approach” modeling, such as flow engines that chain actions (functions) that are agnostic to the rules to which they are applied, making these actions easy to reuse.

6. 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

7. 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 engine 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 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.

Learn more

We help enterprise developers build IoT applications by providing an automation PaaS to develop apps for cloud and the edge.

Get in touch now to learn more
info@waylay.io
+32 9 311 55 66
www.waylay.io

waylay.io connec reason act