DEVELOPER GUIDE

Understanding Custom Rulesets

For developers writing their own rulesets · Project plan and up

A guide to writing checks of your own: what a ruleset can express, what it deliberately can't and how an integration manual becomes a file that holds up in an audit.

As of 25 October 2026. Format version **1.2**; files in 1.0 and 1.1 are still read. The authoritative JSON Schemas (`ruleset-1.0.schema.json` to `ruleset-1.2.schema.json`) ship with every installation. You can also get them from **support@logiq-fleets.com**. The format documents ruleset-format-1.1.md and ruleset-format-1.2.md describe each addition in detail.

What a custom ruleset is

No fleet fails commissioning on VDA5050 alone. Every vehicle manufacturer's integration guide is full of sentences that bind the site and that the standard doesn't see: *"pickFromBin requires a binId of the form BIN-0000"*, *"visualization messages must not arrive faster than every 200 ms"*. Those sentences usually sit in a PDF that nobody holds against the traffic.

A custom ruleset brings them into the Analyzer as a JSON file. Its checks run in the same machinery that judges the standard. Same run, same report, same trail from a finding down to the message and field that caused it. Findings from your rules carry the **CR** badge and name the vehicle. If you say which side caused a violation, they name that side as well.

Two properties carry the design. The rest of this guide follows from them.

A ruleset is data, never code. The file is declarative JSON against a published schema, read by a closed set of rule types. Nothing in it is ever executed. The application's seal states that only software we signed runs on an installation. An uploaded file that could execute would make that statement false.

A custom rule never moves the VDA5050 verdict. The report shows the standard's verdict and the *Custom Rules Verdict* as separate cards. The first is settled before a single custom finding exists. Otherwise a manufacturer could raise its conformance by uploading a friendlier file.

What a ruleset can and cannot do

Most rules judge **one message at a time** against values written in the file. Some follow a vehicle's stream: the spacing of messages, the progression of a number, the moves of a state field. One rule type links a message to the answer that has to follow it. Catalogues list the vendor's actions, errors and fields of its own and the Analyzer derives fixed checks from them. That covers a large share of every integration manual. It's deliberately not everything.

Within reach:

- A field's presence, type, exact value, allowed set, pattern, numeric range or length, at any path, including inside filtered list items (`actions[?actionType=pickFromBin]`).
- A whole message or subtree against an embedded JSON Schema (draft 2020-12), for shapes one predicate can't describe.
- How many items a list holds (`nodes[]` at most 64).
- The spacing between messages of one kind, per vehicle (`state` at least every 1,000 ms, `visualization` no faster than 200 ms).
- How a number progresses, per vehicle or per a field you name: rising, never falling, without gaps or without repeats.
- A value that must not appear, either anywhere or only in a given situation (*while charging,* `driving` *must not be true*).
- The moves a state field may make, per entity (`WAITING` → `RUNNING` , never `FINISHED` → `RUNNING`).
- An answer within a deadline: *after a stateRequest, a state within one second. Once pick reports FINISHED, the loads change within two seconds.*
- The vendor's tables of actions, errors and extension fields, with their parameters, places, levels and durations.
- A condition on the message (`when`), so a rule only speaks in the situation the manual describes.
- Deadlines and limits as named parameters with a default. A requirement file can set them for its runs.

Out of reach, on purpose:

- **Comparing two fields with each other.** A predicate compares with a value in the file, never with another value from the message. *"velocity stays below the declared maxSpeed"* can't be written.
- **Arithmetic.** No sums, rates or differences.
- **Joining messages beyond a trigger and its answer.** A `response` rule links two messages of one vehicle. Wider questions such as *"every order is acknowledged in a state"* belong to the built-in coherence checks.
- **Anything from outside the capture.** No clock, no network, no lookups, no randomness. The same file over the same capture gives the same findings on every installation.
- **The standard's verdict, score or weights.** Custom findings count in the Custom Rules Verdict and nowhere else.

If a sentence of your manual fits none of the rule types, write to support@logiq-fleets.com. New types come with a new format version and stay declarative. A rule type that could compute would be a step towards a programming language. The first property rules that out. A sentence that holds for every vehicle may also become a built-in check that everyone gets.

Anatomy of the file

A ruleset is a single JSON object. This one is complete and passes upload:

```
{
  "ruleset": "1.2",
  "name": "Acme R-Series",
  "version": "1.2.0",
  "date": "2026-08-09",
  "vendor": "Acme Robotics",
  "source": "Acme VDA5050 Integration Guide, Rev. C",
  "target_versions": ["2.1.0"],
  "checks": [
    {
      "id": "ACME_PICK_BIN_ID",
      "group": "Actions",
      "label": "pickFromBin carries a binId",
      "description": "Acme vehicles reject pickFromBin without a binId of the form BIN-0000.",
      "spec_ref": "Integration Guide Rev. C, 4.2.1",
      "severity": "Major",
      "addressee": "master_control",
      "applies_to": { "kind": "order" },
      "rule": {
        "type": "field",
        "path": "nodes[].actions[?actionType=pickFromBin].actionParameters[?key=binId].value",
        "must": { "is": "string", "matches": "^BIN-[0-9]{4}$" }
      }
    }
  ]
}
```

The header identifies the file. Every report that used it records its name, version and SHA-256.

Field	Required	Meaning
`ruleset`	yes	Format version: `"1.0"`, `"1.1"` or `"1.2"`. The version decides what the file may contain.
`name`	yes	The ruleset's name, up to 120 characters.
`version`	yes	Your version string, up to 40 characters. Version it like software: the report freezes it.
`date`	yes	`YYYY-MM-DD`, a real calendar date.
`vendor`	no	Whose rules these are, up to 120 characters.
`source`	no	The document the rules come from, up to 300 characters. Whoever reads the report will want to look things up.
`target_versions`	yes	The VDA5050 versions the rules are written for: any of `"1.1.0"`, `"2.0.0"`, `"2.1.0"` and `"3.0"`.
`checks`	yes in 1.0	Up to 200 checks. From 1.1 a file may hold catalogues only.
`catalogues`	no, from 1.1	Tables of the vendor's actions, errors and extension fields.
`parameters`	no, from 1.2	Named numbers with a default, used in place of a deadline or limit.

Unknown keys are refused everywhere in the file. A typo such as `"severty"` fails the upload. It isn't ignored.

Each check carries its own description before it carries a rule:

Field	Required	Meaning
`id`	yes	`^[A-Z][A-Z0-9_]{2,63}$` , unique in the file and different from every built-in check id. Use a vendor prefix: `ACME_PICK_BIN_ID` .
`label`	yes	One line, up to 160 characters: the finding's headline.
`description`	yes	What the rule means and why it exists, up to 1,000 characters. Write it for the commissioning engineer who sees the finding.
`severity`	yes	`Critical` , `Major` , `Minor` or `Info` , from 1.2 also `Warning` . Counted in the Custom Rules Verdict.
`applies_to.kind`	yes	The message kind the rule reads: `order` , `state` , `instantActions` , `connection` , `visualization` or `factsheet` .
`addressee`	no	Who causes a violation: `master_control` or `vehicle` . Give it when the manual does. Attribution is what ends commissioning arguments.
`group`	no	A heading for the check in the selection and the report, up to 80 characters.
`spec_ref`	no	Chapter and verse in the manufacturer's document, e.g. `"Rev. C, 4.2.1"` . It becomes the finding's recommendation when you give none.
`recommendation`	no	What to do about a violation, up to 500 characters.

The path language

A rule points into the message payload with a dot path built from these segments:

Segment	Meaning
`name`	Go into the object field `name` .
`name[]`	Take every item of the list `name` .
`name[?field=value]`	Take the items of `name` whose `field` equals `value` , compared as text: `[?headerId=7]` matches the number 7.

Dots join the segments: `nodes[].actions[?actionType=pickFromBin].actionParameters[?key=binId].value` . Paths start at the payload. MQTT envelopes are unwrapped first, so a path never begins with `payload.` . Where a path names several values, the rule judges each of them. Every finding shows the concrete place, `nodes[2].actions[0]…` , in the notation of the built-in findings.

There are no wildcards, no numeric indexes and no search across the whole message. A path through a missing field names nothing. That's why existence is a predicate of its own, with its own rule.

The rule types

`field` : a predicate on a value

The workhorse. The rule finds every value the path names and tests each one against the `must` block:

Key	Passes when the value …
`exists`	is present. Judged at the parent, see below.
`is`	has the type `string` , `number` , `integer` , `boolean` , `object` or `array` . A boolean never counts as a number.
`equals`	equals the value given.
`in`	is one of the listed values.
`matches`	is a string containing the regular expression. Anchor it with `^…$` when you mean the whole value.
`min` / `max`	is a number in the range. Anything else fails.
`min_length` / `max_length`	is a string or list within the length bounds.

Several keys must all hold. They're tested in the order of the table and the finding names the first one broken, in words: `is 'BIN-12', which does not match /^BIN-[0-9]{4}$/` .

Existence is judged at the parent. `"must": { "exists": true }` has to end in a plain field name. A missing container isn't a missing field. If `batteryState` is absent, a rule on `batteryState.acmeCellTemp` stays silent. The rule on `batteryState` itself is the one that speaks. Each finding then points at its own cause and doesn't repeat another rule's.

`schema` : an embedded JSON Schema

When the shape is too rich for one predicate, embed a JSON Schema (draft 2020-12) and apply it to the whole payload or to a `path` :

```
{
  "type": "schema",
  "path": "acmeDiagnostics",
  "schema": {
    "type": "object",
    "required": ["frameCounter", "motorTemp"],
    "properties": {
      "frameCounter": { "type": "integer", "minimum": 0 },
      "motorTemp": { "type": "number", "maximum": 90 }
    }
  }
}
```

Each violation becomes a finding of its own, with the inner path appended. The upload checks the embedded schema. A broken schema fails the upload, never the analysis.

`cardinality` : how many

Counts the items the path names in each message and holds the count against `min` , `max` or both:

```
{ "type": "cardinality", "path": "nodes[]", "max": 64 }
```

"Acme vehicles buffer at most 64 nodes per order" is one line. The finding reads `78 present, at most 64 allowed` .

`interval` : spacing between messages, per vehicle

Measures the time between consecutive messages of the check's kind, per vehicle, by the messages' own timestamps:

```
{ "type": "interval", "min_ms": 200 }
```

Give `min_ms` , `max_ms` or both. There's no `path` . The analysis-wide timing tolerance only ever **widens** the allowed band, as it does for the built-in interval checks. It can excuse a borderline case and never creates a violation. A stream that's off throughout is one fact, not a thousand findings. The rule reports at most one finding per vehicle for intervals too short and one for intervals too long, with the count and the worst value: `AGV-3: 41 intervals below 200 ms, worst 12 ms` .

`sequence` : how a number progresses

Follows one numeric field through a vehicle's stream:

```
{ "type": "sequence", "path": "acmeDiagnostics.frameCounter", "mode": "gapless" }
```

Mode	Each value must be …
`increasing`	greater than the one before.
`non_decreasing`	at least the one before.
`gapless`	exactly the one before plus 1.
`no_gap`	at most the one before plus 1. A repeat or a drop is allowed, a jump forward isn't.
`distinct`	a value not seen before in the stream, however far back.

`group_by` narrows the stream further. With `"group_by": "orderId"` the rule follows `orderUpdateId` per order rather than per vehicle. A message in which the path names no number is skipped and the rule keeps its memory. The finding names both messages: `is 17 after 19 (message 204), expected 20` .

`forbidden` : a value that must not appear

The reverse of `field` : the rule fires when the path names something. On its own it forbids presence. With `equals` , `in` or `matches` it forbids only those values:

```
{
  "id": "ACME_NO_SEMIAUTOMATIC",
  "label": "SEMIAUTOMATIC is not implemented",
  "description": "Acme R-Series firmware has no SEMIAUTOMATIC mode; a vehicle reporting it is
misconfigured.",
  "severity": "Critical",
  "addressee": "vehicle",
  "applies_to": { "kind": "state" },
  "rule": { "type": "forbidden", "path": "operatingMode", "equals": "SEMIAUTOMATIC" }
}
```

`transition` : the moves a state field may make

The rule follows each entity, named by `id_field` , through the listed `collections` of one vehicle's stream. Whenever its `value_field` changes, the move has to be one of the `allowed` `[from, to]` pairs:

```
{
  "type": "transition",
  "collections": ["actionStates"],
  "id_field": "actionId",
  "value_field": "actionStatus",
  "states": ["WAITING", "RUNNING", "FINISHED", "FAILED"],
  "allowed": [["WAITING", "RUNNING"], ["RUNNING", "FINISHED"], ["RUNNING", "FAILED"]]
}
```

`states` limits which moves are judged. A change into a value outside that list is left to an enumeration check. A wrong value is then reported once, as a wrong value, and not again as a wrong move. Values are compared regardless of case. VDA5050 1.1 prints its action states in lower case and only 2.0 made upper case binding. How a value is spelled is the enumeration check's business.

`response` : an answer within a deadline (from 1.1)

A message (the **trigger**) requires a second message (the **answer**) from the same vehicle within a deadline. Where needed, a field links the two:

```
{
  "type": "response",
  "trigger": { "kind": "order", "path": "nodes[].actions[?actionType=liftUp]" },
  "expect": { "kind": "state", "path": "actionStates[?actionStatus=RUNNING]" },
  "correlate": [ { "trigger": "actionId", "expect": "actionId" } ],
  "within_ms": 5000,
  "from": "released"
}
```

`trigger` and `expect` each name a message kind and optionally a `path` and a `when`. Every element under the trigger's path is a trigger of its own. `not_before_ms` sets an earliest moment for the answer. For actions in orders, `from` starts the clock when the order is sent (`sent` , the default), when the node is released (`released`) or when the vehicle reaches it (`reached`). The clock is the capture's arrival time, because controller and vehicle clocks differ. `clock: "message"` reads the timestamps instead, for a guide that means them.

No answer in time is one finding at the trigger, with the measured time or "no state answered". If the capture ends before the deadline, the rule says "not judged" as Info and counts neither a pass nor a failure. An order update that resends its stitching node isn't a second trigger.

Format 1.2 adds two keys for sentences like *"once pick has finished, the load shows in the state"*:

- `"on": "becomes"` in the trigger: an element is a trigger only in the message in which it starts to match. Later states that still report FINISHED aren't new triggers.
- `"changed": "loads"` in `expect` : a message answers only when the value at that path differs from the vehicle's last one before the trigger. The trigger's own message may already carry the change.

Catalogues: actions, errors and extension fields (from 1.1)

The section `catalogues` sits next to `checks` . It holds tables and the Analyzer derives fixed checks from them. Their ids are built from `id_prefix` , for example `ACME_ACTION_PARAMETER` . A check is only derived where the catalogue says something it can judge.

Catalogue	What it lists	Derived checks
`actions`	action type, places (`NODE` , `EDGE` , `INSTANT`), blocking types, parameters with `required` and `must` , `max_duration_ms` , `failed_needs_result`	unknown action (under `strict` only), parameter, place, blocking type, duration, result, comparison with the factsheet (Info)
`errors`	error type, levels, required references, a pattern for the description, `max_duration_ms` , a condition on the state (`only_when`)	unknown error (under `strict` only), level, references, description, duration, context
`extensions`	fields of the vendor's own, per message kind, with `must`	value, unknown field (under `strict` only)

An action is judged once per order, however often updates resend its node. An error is judged once per episode. The standard's predefined actions such as `pick` and `startCharging` are judged by the Analyzer already. Where they're in the catalogue, its entries apply in addition. `severity` sets the default severity per aspect.

From 1.2, `judge` in the action catalogue decides which side it judges. `vehicle` keeps only duration and result, which the vehicle reports. `controller` keeps only what orders and instant actions send. `both` is the default. On a test bench the bench sends the orders, so a vehicle manufacturer usually wants `vehicle` . `"factsheet": false` drops the comparison with the factsheet.

Parameters (from 1.2)

A file declares its deadlines and limits once, with a default, and uses them by name:

```
"parameters": {
  "charge_start_ms": { "default": 60000, "unit": "ms", "assumed": true,
                       "description": "How soon charging starts. Not in the manual." }
}
```

`{ "param": "charge_start_ms" }` then stands in place of the number: in `min` / `max` of a predicate or a cardinality, in `min_ms` / `max_ms` of an interval, in `within_ms` / `not_before_ms` of a response and in `max_duration_ms` of the catalogues. `assumed: true` says the value isn't the vendor's. The report says so for as long as nobody has set it.

A requirement file that names the ruleset may set the values for its runs: `{ "name": "Acme R-Series", "parameters": { "charge_start_ms": 45000 } }`. A value for a parameter the ruleset doesn't declare refuses the run. If two requirement files in one run set the same value differently, the run is refused too, with both files named. The ruleset's fingerprint doesn't change: a pin by SHA-256 still holds.

Conditions: `when`

`field` and `forbidden` rules take an optional `when` block, as do the trigger and the answer of a `response`. The rule then only speaks in messages where the manual's situation holds:

```
{
  "type": "forbidden",
  "when": { "path": "batteryState.charging", "equals": true },
  "path": "driving",
  "equals": true
}
```

While charging, the vehicle must not report driving. A `when` holds a `path` and at most one test. Without a test it asks for something to be there, `exists: false` asks for nothing to be there. The other tests are `equals`, `in` and `matches`. Mind the difference: `when` selects messages, the filter `[?field=value]` selects items of a list. Whatever belongs together inside one list item goes into the path.

Uploading a ruleset

Rulesets live under **Custom Rulesets** in the Analyzer menu. The entry appears when the plan carries the feature. Superusers and administrators may upload, because a ruleset changes what every member's runs are judged with.

The upload checks that the file is readable and understood. It has to be valid against the schema of its format version. Every path must parse, every regular expression compile and every embedded schema be valid. Every id must be unique and none may belong to a built-in check. A parameter used must be declared. A file that passes won't surprise an analysis later. Whether its rules are *wise* is settled elsewhere, see below.

The upload also refuses a file that's already stored. A different file under a name and version already present is refused as well, because a report names what it was measured against. Raise the version or delete the old file.

Right after the upload the page reads the file against the standard (next section). The list of rulesets shows the result in the *Standard* column. A ruleset can be filed under a project. Opening one shows every check with its explanation, severity and source reference, the file itself formatted for reading, its fingerprint and a download.

Propose from a capture on the rulesets page drafts a file from a clean capture of the vehicle: the actions sent with their places, blocking types, parameters and durations, the errors reported and the answer times to `stateRequest` and `factsheetRequest`. Durations and answer times stand at twice the longest seen, rounded up. Nothing is `strict`. The draft is downloaded, corrected and uploaded like any other file. Nothing is stored before that.

Where a rule contradicts the standard

A plant standard is *meant* to be stricter than VDA5050. "Only AUTOMATIC runs here" narrows five allowed operating modes to one. Every message that satisfies it satisfies the standard too. That's the point of the file and it's never reported.

What's worth knowing is the other case: a rule **no conforming message can ever satisfy**. A fleet held to it can't be conformant. Almost nobody means that. It's a typo, a field the standard renamed between versions or a rule written against a different specification.

The ruleset page therefore carries a section *Against the standard*. Both sides are data. A rule is a predicate over a path and the standard is the JSON Schema in the conformance pack, so the question is decided and not guessed. For one path the rule accepts one set of values and the standard another:

	Meaning	Reported
No value in common	no conforming message can pass the rule	**contradicts**
The rule's set lies inside the standard's	the plant standard narrows it	never
The rule also admits what the standard rejects	the two verdicts can disagree	as a note

Decided today:

- a value or pattern outside an enumeration;
- a type the standard defines differently;
- a range outside the standard's;
- a mandatory field forbidden or demanded absent;
- a mandatory list demanded empty;
- every allowed value of a field forbidden at once;
- an action transition the standard doesn't permit.

Judged per version, because the versions differ. A rule on `safetyState.eStop` is right for 2.1.0 and names nothing in 3.0, which renamed the field to `activeEmergencyStop`. A ruleset carried over unchanged goes silent there. The section says so.

Some cases are left out on purpose. A rule requiring a field of your own is no contradiction: every `additionalProperties` in the packaged schemas is `true`, so the standard permits extra fields. A rule with a `when` isn't judged. A condition that never holds makes the rule empty, not contradictory, and only the traffic can tell which. Timing isn't judged at all. VDA5050 names no message rate, so *state every 200 ms* contradicts nothing.

A file written for two versions often carries rules in pairs, one for each spelling of a renamed field. That pattern is recognised and stays silent. A path that **no** version of the file knows is reported, because it's a typo and the rule never fires anywhere.

None of this refuses the file and none of it touches a verdict. Whether a deviation is deliberate is your decision. The finding is a property of the file, which is why it stands on the ruleset page and not in every report.

Choosing rulesets for a run

The Analyze page and the capture profile offer the rulesets that fit:

- **The capture's own project.** A capture filed under a project is offered the rulesets of that project, as with layouts and factsheets. A capture outside any project is offered the rulesets outside any project.
- **The run's version.** A ruleset judges the versions it names. If a capture holds two versions, each ruleset judges the one it was written for and is left out of the other report. The report summary names what was left out.
- **One check at a time.** Each check of a ticked ruleset can be switched off. The report lists it as *switched off*, never as missing.

A capture profile keeps a ruleset whatever version it names and applies it at analysis time, because a capture judges every version it hears.

What the report shows

- The **Custom Rules Verdict** stands as its own card next to the standard's, with its own severity counts and the rulesets used. Below each ruleset, every parameter has a row with its value and where the value came from: the default, an assumption or the requirement file that set it.
- The table **Custom Ruleset Checks** lists every check with its result, its findings, *Switched off* or *No matching messages*. A check whose situation never occurred has no data and never counts as a pass.
- Findings carry the CR badge, your severity, the vehicle and your addressee. They can be dispositioned and waived like built-in findings.
- A miswritten rule can't flood a report. After 500 findings per check the last finding says so: *… and 1,250 further occurrences beyond the first 500 were not listed individually.*
- Every report keeps a frozen copy of the rulesets it was judged with. The audit block records name, version and SHA-256 of each, plus the parameter values. Replacing or deleting the upload later never changes what an existing report says it was measured with. In the signed Report Package the copy travels as `references/rulesets.json`, covered by the manifest and the signature.

Sharing a ruleset with partners

A fleet controller vendor can share a ruleset with the vehicle manufacturers it has approved as partners. The right to share comes with the Vendor Programme: a plan of its own, an option on Team and included in Unlimited. It's the same right that shares requirement files. A partner finds shared rulesets on the Analyze page under *Rulesets shared with you*, grouped by vendor.

A requirement file can demand a ruleset by name and pin it by SHA-256. The manufacturer is then judged against exactly that file, with the parameter values the requirement file sets. No longer sharing it takes the ruleset off the partner's list and nothing else. A report already made keeps its verdict and its frozen copy.

Writing rules that hold up against real traffic

Start from the manual, not from the traffic. Every check is a sentence from the manufacturer's document and `spec_ref` names chapter and verse. A rule that can't be cited is an opinion.

Write the description for the reader of the finding. That's a commissioning engineer at 2 a.m. who needs to know what the rule means, why it exists and what to do. The `label` is the headline, `recommendation` the way out.

Choose severities the way the manufacturer would.

- `Critical` : the vehicle stops or rejects the order.
- `Major` : the order degrades or the behaviour is out of specification.
- `Minor` : it works but leaves the declared values.
- `Warning` : worth saying out loud, yet no breach of the standard.
- `Info` : worth seeing, not worth an argument.

Attribute where you can. `addressee` turns *"there is a finding"* into *"the fleet controller sent this"*. That's the difference between a debate and a fix.

Make an assumption visible. Where the manual gives no number, declare a parameter with `assumed: true` . The report then shows the value as an assumption until the vendor confirms one.

Anchor your regular expressions. `matches` searches inside the string, so `"BIN-[0-9]{4}"` also accepts `XBIN-12345` . Write `^BIN-[0-9]{4}$` when you mean the whole value.

Test every rule against a real capture. A rule fails silently in two ways. It **never fires**, because a path is mistyped, and the report reads as if all were in order. Or it **always fires**, because a predicate is too sharp. The customer then drowns in findings and switches the ruleset off. Analyse a capture with the new ruleset and read the table *Custom Ruleset Checks*. Before the file goes into an acceptance, look twice at a check with *No matching messages* where you expected traffic and at one with hundreds of findings.

Writing a ruleset from a manufacturer's integration guide is a service we provide. We write the file, run it against a sample capture, explain both warning signs and hand it over. Ask us with the document at hand.

Limits

Limit	Value
Plans	Project, Monitoring and Vendor Programme: 3 rulesets. Team and Unlimited: any number. Trial and Testbench: none
Who may upload and delete	Superusers and administrators
Checks per file	200
Entries per catalogue	200
Parameters per file	30
File size	2 MB
Findings per check and run	500, the cut stated in the report
Open expectations per `response` rule	10,000 at a time, the cut stated in the report
Check id	`^[A-Z][A-Z0-9_]{2,63}$`, vendor prefix recommended
Visibility	The whole enterprise; a ruleset filed under a project only its members

Questions about custom rulesets: **support@logiq-fleets.com** LogistiXpert · logistixpert.com