DEVELOPER GUIDE

Understanding Custom Rulesets

For developers writing their own rulesets · Team plan and up

A developer's guide to writing rules of your own — what a ruleset can express, what it deliberately cannot, and how to get from an integration manual to a file that survives an audit.

As of 16 August 2026. Format version **1.0**. The authoritative JSON Schema, `ruleset-1.0.schema.json`, ships with every installation and is available on request from **support@logiq-vda5050.com**.

What a custom ruleset is

No fleet fails commissioning on VDA5050 alone. Every vehicle vendor's integration guide is full of sentences that are binding on the site and invisible to the standard — "*pickFromBin requires a binId of the form BIN-0000*", "*visualization messages must not arrive faster than every 200 ms*". Today those sentences live in a PDF that nobody checks against the actual traffic.

A custom ruleset carries those sentences into the analyzer as a JSON file. Once uploaded, its checks run in the same machinery that evaluates the standard: same run, same report, same evidence trail from a finding down to the exact message and field that caused it. Findings from your rules carry the **CR** family badge and are attributed to a vehicle and, if you say so, to the side that caused them.

Two properties of the design are not negotiable, and everything else in this guide follows from them:

1. **A ruleset is data, never code.** The file is declarative JSON against a published schema, interpreted by a closed set of seven rule types. Nothing in it is ever executed. Every report carries an integrity statement saying that only signed software runs on an installation — an uploaded file that could execute would make that statement false.
2. **A custom rule never moves the VDA5050 verdict.** The report shows two verdicts side by side: the standard alone, and the standard plus your rules. The first is computed before a single custom finding exists. Anything else would let a vendor improve their conformance by uploading a friendlier file — the opposite of what this product is for.

What rulesets can do — and what they cannot

A ruleset judges **one message at a time** against constants you wrote down, plus two per-vehicle properties over the message stream (spacing and monotonicity). That covers a surprising share of every integration manual. It does not cover everything, on purpose.

Within reach:

- A field's presence, type, exact value, allowed set, pattern, numeric range or length — per message, at any path, including inside filtered array items (`actions[?actionType=pickFromBin]`).
- A whole message or subtree against an embedded JSON Schema (draft 2020-12), for shapes too rich for a single predicate.
- How many items an array 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).
- The monotonicity of a numeric field, per vehicle (a counter that must be gapless, a value that must never decrease).
- A value that must not appear at all, optionally only under a condition (*while charging,* `driving` *must not be true*).
- A message-level gate (`when`) so a rule only speaks in the situation the manual describes.

Out of reach, deliberately:

- **Comparing two fields to each other.** Predicates compare against constants in the file, never against another value from the message ("*velocity must stay below the declared maxSpeed*" is not expressible).
- **Arithmetic and derived values.** No sums, no rates, no differences.
- **Correlation across message kinds.** "*Every order must be acknowledged in a state message*" is the job of the built-in coherence checks — a ruleset never joins two messages.
- **State machines.** Beyond monotonicity there is no memory between messages; action-lifecycle checking is built in, not writable.
- **Anything environmental.** No wall clock, no network, no lookups, no randomness. The same file over the same capture yields the same findings, every time, on every installation.
- **Touching the standard's verdict, score or severity weights.** Custom findings are counted in the Custom Rules verdict and nowhere else.

If a sentence in your manual does not fit these seven shapes, the answer is not an eighth rule type — every additional type is a step toward a programming language, and property 1 forbids that. Write to support@logiq-vda5050.com instead; if the sentence generalises, it becomes a built-in check that everyone gets.

Anatomy of the file

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

```
{
  "ruleset": "1.0",
  "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; the analyzer records name, version and the file's SHA-256 in every report that used it.

Field	Required	Meaning
`ruleset`	yes	Format version, always `"1.0"`.
`name`	yes	The ruleset's name, up to 120 characters. Shown wherever the report names its rules.
`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.
`source`	no	The document the rules were taken from — the report reader will want to look things up.
`target_versions`	yes	Which VDA5050 versions the rules apply to: `"2.1.0"`, `"3.0"` or both. The ruleset is only offered where the run's target version matches.
`checks`	yes	1 to 200 checks.

Unknown keys are rejected everywhere in the file — a typo like `"severty"` fails the upload instead of being silently ignored.

Every check carries its own paperwork before it carries a rule:

Field	Required	Meaning
`id`	yes	`^[A-Z][A-Z0-9_]{2,63}$`, unique within the file, and not colliding with a 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` — 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 caused a violation: `master_control` or `vehicle`. Say it when the manual says it — attribution is what ends commissioning arguments.
`group`	no	Groups checks under a heading in the selection UI and the report.
`spec_ref`	no	Chapter and verse in the vendor document, e.g. `"Rev. C, 4.2.1"`. Becomes the finding's recommendation if you give none.
`recommendation`	no	What to do about a violation, up to 500 characters.

The path language

Rules point into the message payload with a dot path. Three segment forms exist, and only these three:

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

Segments join with dots: `nodes[].actions[?actionType=pickFromBin].actionParameters[?key=binId].value`. Paths are relative to the payload — MQTT envelopes are unwrapped before evaluation, so never write a `payload.` prefix. When a path names several values (through `[]` or a filter), the rule judges **each of them**, and each violation's finding shows the concrete location — `nodes[2].actions[0]…` — in the same notation every built-in finding uses.

There are no wildcards, no numeric indexes and no "search anywhere" operator. A path that steps through a missing field simply names nothing — which is why existence is its own predicate with its own semantics, below.

The seven rule types

`field` — a predicate on a value

The workhorse: find every value the path names, test each against the `must` block. Available predicate keys, checked in this order:

Key	Passes when the value…
`exists`	is present (see below — existence is judged at the parent).
`is`	has the type: `string`, `number`, `integer`, `boolean`, `object`, `array`. Booleans never count as numbers.
`equals`	equals the constant exactly.
`in`	is one of the listed constants.
`matches`	is a string containing the regular expression — anchor with `^…$` when you mean the whole value.
`min` / `max`	is a number in the range. A non-number fails.
`min_length` / `max_length`	is a string or list within the length bounds.

Several keys combine as *and*; the finding names the first broken promise in words: `is 'BIN-12', which does not match /^BIN-[0-9]{4}$/`.

Existence has one subtlety. `"must": { "exists": true }` must end in a plain field name, and the *parent* decides: a missing container is not a missing field. If `batteryState` is absent entirely, a rule on `batteryState.acmeCellTemp` stays silent — a different rule on `batteryState` itself is the one that should speak. This keeps every finding pointing at its own cause instead of echoing another rule's.

A `when` condition (next section) gates the whole rule per message.

`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 }
    }
  }
}
```

Every schema violation becomes its own finding with the inner path appended. The embedded schema is itself validated on upload — a broken schema fails the upload, not the analysis.

`cardinality` — how many

Counts how many items the path names in each message and holds the count against `min`, `max` or both (at least one is required):

```
{ "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

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

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

`min_ms`, `max_ms` or both; no `path`. The analysis-wide timing tolerance is applied with the same discipline the built-in interval checks use: it only ever **widens** the allowed band, so a tolerance can excuse a borderline case but never create a violation. A stream that is systematically off is one fact, not a thousand findings — the check reports **one finding per vehicle and direction**, with the count and the worst value: `AGV-3: 41 intervals below 200 ms, worst 12 ms`.

`sequence` — monotonicity, per vehicle

Watches one numeric field across the stream of a vehicle:

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

Mode	Each value must be…
`increasing`	strictly greater than the previous one.
`non_decreasing`	greater than or equal to the previous one.
`gapless`	exactly the previous one plus 1.

Messages where the path names no number are skipped without resetting its memory. The finding names both messages: `is 17 after 19 (message 204), expected 20`.

`forbidden` — a value that must not appear

The inverse of `field`: the rule fires when the path names something. Bare, it forbids presence; with `equals`, `in` or `matches` it forbids only the named 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` — allowed moves of a state value

A field that moves along a set of allowed pairs, per entity, per vehicle. The rule follows each entity — named by `id_field` — through the listed `collections` across one vehicle's stream; whenever its `value_field` changes, the move must 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` narrows what counts as a move worth judging: a change into a value outside that set is left to a separate enum check, so a bad value is reported once — as a bad value, not twice as a bad value and a bad move. Values are compared case-blind, because VDA5050 1.1 prints its action statuses in lower case and only 2.0 made upper case the rule; how a value is spelled is the enum check's business, not this one's.

Conditions: `when`

`field` and `forbidden` rules take an optional `when` block that gates the rule at message level — the rule only speaks 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 `path` plus one of: nothing (something exists at the path), `exists: false` (nothing does), or `equals` / `in` / `matches` (any value at the path satisfies the predicate). Note the direction: `when` selects messages, the `[?field=value]` filter selects array items — correlation *within* one array element belongs in the path, not in `when`.

What happens at analysis time

- A ruleset is offered for a run when its `target_versions` contains the run's target version. Selection happens on the Analyze page and on capture profiles; each check can be switched on or off, and a switched-off check appears in the report as *switched off* — never silently missing.
- A check whose message kind never occurs in the capture is reported as **No data** — not as a pass nobody earned.
- Findings carry `type: "custom"`, the CR badge, your severity, the vehicle, your addressee, and a description of the form `label: detail`. They can be dispositioned and waived exactly like built-in findings.
- A miswritten rule cannot flood a report: after 500 findings per check the cut is announced in the last finding — `… and 1,250 further occurrences were not listed individually` — never applied silently.
- The report shows **two verdicts side by side**: VDA5050 alone, and Custom Rules. A frozen copy of every ruleset used is stored next to the report files, and the audit block records name, version and SHA-256 of each. Replacing or deleting the upload later never changes what an existing report says it was measured with. In the signed report package, the frozen rulesets travel under the manifest and the signature.

Where a rule contradicts the standard

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

What is worth knowing is the other case: a rule **no conforming message can ever satisfy**. A fleet held to it cannot be conformant, and almost nobody means that — it is 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, the standard is the JSON Schema in the conformance pack — so the question is decided rather than guessed. For one path the rule accepts a set of values and the standard accepts another:

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

Decided today: a value or pattern outside an enumeration, a type the standard defines differently, a numeric range outside the standard's, a mandatory field forbidden or demanded absent, a mandatory collection demanded empty, every allowed value of a field forbidden at once, and an action transition the standard does not 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, and that is exactly what the section says.

Three things are deliberately **not** reported. A rule requiring a field of your own is not a contradiction: every `additionalProperties` in the packaged schemas is `true`, so the standard permits extra fields. A rule with a `when` is not judged, because a condition that never holds makes it vacuous rather than contradictory, and deciding which needs the traffic. And timing is not judged at all — VDA5050 names no message rate, so *state every 200 ms* cannot contradict it.

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

None of this touches a verdict. It is a property of the file, established when the file is read, which is why it stands on the ruleset page and not in every report.

Writing rules that survive contact with real traffic

Start from the manual, not from the traffic. Every check should be a sentence from the vendor document, with `spec_ref` naming chapter and verse. If you cannot cite it, it is an opinion, not a rule.

Write the description for the reader of the finding, a commissioning engineer at 2 a.m.: what the rule means, why it exists, what to do. The `label` is the headline; `recommendation` is the way out.

Choose severities like the vendor would. `Critical` : the vehicle stops or rejects the order outright. `Major` : the order degrades or the behaviour is out of spec. `Minor` : works, but outside the declared values. `Info` : worth seeing, not worth arguing about.

Attribute when you can. `addressee` is what turns "*there is a finding*" into "*the master control sent this*" — the difference between a debate and a fix.

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

Test every rule against a real capture — this step is not optional. A generated or hand-written rule fails silently in two ways: it **never fires** (a mistyped path — the report then reads as "all in order"), or it **always fires** (a predicate too sharp — the customer drowns in findings and switches the ruleset off). The lint tool runs a ruleset dry against a sample capture, counts hits per check and flags both ends of the table:

```
# Generate: translate the integration guide against the schema
python3 tools/make_ruleset.py acme_guide.md \
    --name "Acme R-Series" --vendor "Acme Robotics" --version 1.0.0

# Test it against a real capture
python3 tools/lint_ruleset.py acme.ruleset.json --sample capture.ndjson
```

A file goes to production only after both warning signs have been looked at and explained.

Upload, rights and limits

Uploading (sidebar → Rulesets) checks exactly one thing: that the file is readable and understood — schema-valid, every path parseable, every regular expression compiling, every embedded schema valid, every id unique and not colliding with a built-in check. A file that passes cannot surprise an analysis later. Whether its rules are *wise* is your business, and the lint step above is where that is settled.

Limit	Value
Plans	From **Team** (5 rulesets); unlimited on **Enterprise** and on-premises
Who may upload	Superusers — a ruleset changes what every member's runs are judged with
Checks per file	200
File size	2 MB
Findings per check and run	500, cut announced in the report
Check id	`^[A-Z][A-Z0-9_]{2,63}$`, vendor prefix recommended
Scope	Not project-bound: a ruleset describes a vendor, and is offered wherever the target version matches

Questions about custom rulesets: **support@logiq-vda5050.com** LogistiXpert · www.logistixpert.de