LOGZILLA DOCUMENTATION

Lua Rules

Lua rules for advanced LogZilla event parsing and enrichment using the event object API, key-value helpers, and conditional field extraction

Data Transforms · Generated June 11, 2026 · logzilla.ai/docs/data-transforms/lua-rules

Lua rules

Lua rules provide advanced parsing and enrichment capabilities for complex data transformation scenarios. They execute custom logic to extract fields, apply conditional processing, and enrich events with contextual information.

When to use Lua rules

- Vendor-specific formats that require multiple extractions.
- Conditional logic to interpret message variants.
- Field mappings, lookups, or normalization across sources.
- Message reformatting to create consistent, readable content.
- Context enrichment that adds durable, searchable tags.

Event object API

Lua rules operate on an `event` object with these key properties:

- `event.message` - The original log message text
- `event.host` - Source hostname or IP address
- `event.program` - Program or service name
- `event.severity` - Numeric severity level (0-7)
- `event.facility` - Syslog facility code
- `event.user_tags` - Table for custom searchable tags
- `event.extra_fields` - Table for additional structured data

Available helper functions

LogZilla provides helper functions for common parsing tasks:

- `get_kv_parser(sep, delim, quote)` - Creates key-value parser
- `starts_with(str, prefix)` - Check string prefix
- `ends_with(str, suffix)` - Check string suffix
- `get_port_name(port)` - Convert port number to service name
- `push_event(event)` - Generate extra event from current processing context

Format examples

Lua rules are stored as `.lua` files in the rule directory. The following example shows a basic structure for parsing key-value pairs:

```
-- Parse key-value pairs from messages like: srcip="192.168.1.10" dstip="8.8.8.8" action="accept"
local kv_parser = get_kv_parser(' ', '=', '"')

function process(event)
    -- Check if this looks like a key-value format
    if string.find(event.message, '="') then
        -- Set program name for identification
        event.program = "CustomApp"

        -- Parse all key-value pairs from the message
        local kvpairs = kv_parser:match(event.message)

        -- Convert parsed pairs to searchable user tags
        for key, value in pairs(kvpairs) do
            if key == "srcip" then
                event.user_tags["SrcIP"] = value
            elseif key == "dstip" then
                event.user_tags["DstIP"] = value
            elseif key == "action" then
                event.user_tags["Action"] = value
            end
        end
    end
end
```

For conditional tagging based on message content:

```
function process(event)
    -- Only process events from specific sources
    if event.program == "MyApp" then
        -- Add severity tags based on message content
        if string.find(event.message, "ERROR") then
            event.user_tags["Severity"] = "High"
            event.user_tags["Alert"] = "Error Detected"
        elseif string.find(event.message, "WARN") then
            event.user_tags["Severity"] = "Medium"
            event.user_tags["Alert"] = "Warning"
        end

        -- Extract and tag IP addresses
        local ip_pattern = "(%d+%.%d+%.%d+%.%d+)"
        local ip = string.match(event.message, ip_pattern)
        if ip then
            event.user_tags["IP_Address"] = ip
        end
```

```
    end
end
```

How Lua rules work

- The parsing engine evaluates installed rules in a defined order.
- Each rule inspects the event and decides whether to act.
- Rules can enrich events by writing into structured fields and tags.
- Rules can perform controlled rewrites for consistency and clarity.
- Rules can skip events that do not meet criteria without overhead.
- Rules can generate new events if needed for complex scenarios.

The result is a normalized event with useful fields for search, dashboards, and triggers.

Common enrichment patterns

- Add normalized tags, such as device role, location, or application.
- Extract identifiers (for example, user, session, interface) into dedicated fields.
- Convert vendor codes into readable labels.
- Reconstruct terse messages into human-readable summaries when needed.

Creating new events from rules

Since version 6.39, Lua rules can create additional events derived from the original event or created completely from scratch. This capability enables complex processing scenarios such as splitting message into multiple events.

The `push_event(new_event)` helper function creates new events. The `new_event` object passed to this function will be processed and indexed as any other event.

The `Event:new()` constructor creates new, empty events. All fields must be populated with appropriate values:

```
local new_event = Event:new()

new_event:set_timestamp_now() -- this updates event timestamp to current ts
new_event.message = "Sample log message"
new_event.host = "localhost"
new_event.program = "MyApp"
new_event.user_tags["CustomTag"] = "Value"

push_event(new_event)
```

Cloning existing events and modifying specific fields provides a more convenient approach. The `clone()` method supports this pattern:

```
local cloned_event = event:clone()

cloned_event.message = "Modified log message"

push_event(cloned_event)
```

Generated events are not subject to rule processing to prevent infinite loops.

The event passed as an argument to the `process()` function will be processed and indexed as usual after the rule completes. Rules that intend to create new events and skip the original should return `ProcessResult.DELETE`. Modifying the original event is more efficient than creating a new one in most cases.

Authoring guidelines

- Keep rule intent focused; split distinct concerns into separate rules.
- Prefer stable, low-cardinality tags for dashboards and filters.
- Use consistent field names across apps to improve reuse.
- Gate rules to the intended sources using dedicated inputs where applicable; see Syslog pipeline customization (https://www.logzilla.ai/docs/administration/syslog-pipeline-customization).

Managing rules via CLI

Refer to the CLI section for full syntax and options: Command Line Tools —- Data Commands (https://www.logzilla.ai/docs/command-line-tools/data-commands).

Typical operations include:

```
# List rules and review status
logzilla rules list

# Validate or test a rule file before enabling
logzilla rules validate /path/to/rule.yaml
logzilla rules test --path /path/to/rule.yaml

# Enable, disable, or reload rules after changes
logzilla rules enable "My Lua Rule"
logzilla rules disable "My Lua Rule"
logzilla rules reload

# Review recent runtime errors for rules
logzilla rules errors
```

Performance and reliability

- Narrow rule scope early to minimize unnecessary processing.
- Avoid creating many high-cardinality tags; prefer normalized keys.
- Monitor parser metrics and rule errors during rollouts.
- Validate and test rules prior to enabling in production.

```
# Parser overview and throughput
logzilla events parser-stats

# Field and tag insights
logzilla events values --scope fields --limit 50
logzilla events values --scope tags --limit 50
```

Safety and governance

- Treat rules as configuration-as-code; review changes and track history.
- Favor readability and stable behavior over clever parsing tricks.
- Ensure sensitive data is not emitted as tags or message content.

Related reading

- Overview and decision guide (https://www.logzilla.ai/docs/data-transforms/transforming-event-streams)
- Rewrite rules (https://www.logzilla.ai/docs/data-transforms/rewrite-rules)
- LogZilla Apps (https://www.logzilla.ai/docs/data-transforms/creating-custom-apps)
- Syslog pipeline customization (https://www.logzilla.ai/docs/administration/syslog-pipeline-customization)
- Command Line Tools —- Data Commands (https://www.logzilla.ai/docs/command-line-tools/data-commands)