# SynchroTail User Guide

## Version 1.4.2

29 September 2026

# Contents

# 1. What SynchroTail does

SynchroTail turns **log lines into Synchro metrics**. It follows a set of log files as they are written and rotated, matches each line against regular expressions you supply, pulls numbers and names out of the matched text, and sends the result to SynchroServer as `SC01 stat` messages.

It is the third way data reaches Synchro, and the one that needs no cooperation from the system being measured:

| Source | Needs |
|---|---|
| SynchroAgent | an agent installed on the host |
| SynchroSocket | an agent installed on the host |
| **SynchroTail** | **only that the application writes a log** |

That distinction is the whole point. A closed third-party system that will never accept an agent, and cannot be instrumented, can still be measured in detail if it writes a log line per event.

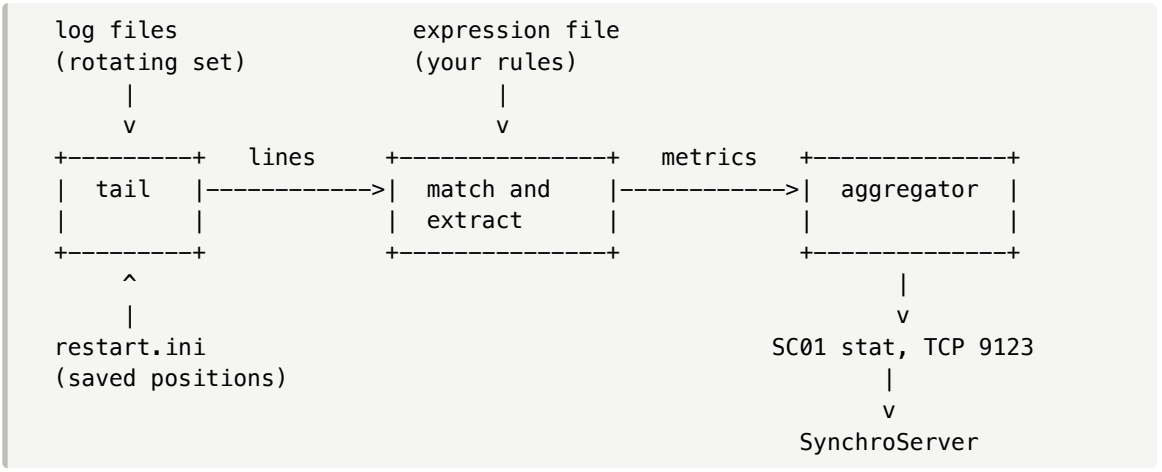
## 1.1 What makes it more than a grep

Two things.

**It understands a rotating set of files.** Not one file, but every file matching a pattern, processed in a defined order, with its position saved so a restart resumes where it stopped rather than replaying or skipping.

**It tracks transactions.** A start line and an end line, correlated by an identifier carried in the text, give you in-flight counts and durations for work the application never reports as a number. Section 6 covers this; it is the capability most log tools lack.

## 2. How it works




A line is read, tested against each expression's include and exclude patterns, and where it matches, an extraction regular expression pulls out capture groups. Those groups are assembled into one or more metrics according to a **match specification**, and handed to the aggregator, which batches them and ships them to SynchroServer.

Input does not have to be files. `input=` accepts `FILE` (the default), `STDIN`, `HTTP` or `STAT`, the last two opening listening ports so other processes can feed lines in. Output does not have to be SynchroServer either: `output=` accepts `STDOUT`, `SOCKET`, `HTTP` and `QUIET`, which is how you test a rule set without writing anything to a live server.

# 3. The expression file

Rules live in a text file named by `expressionfile=`. Lines starting with `#` or `//` are comments, blank lines are ignored, and each remaining line is parsed as comma-separated fields.

A field may be quoted with `"` when it contains a comma — which regular expressions frequently do, for example `^(\d{1,3}),`. **Quote any pattern containing a comma**, or the line will be split in the wrong place and the rule will be silently misread.

## 3.1 The five-field form

```
<exprID>,<include regex>,<exclude regex>,<match specification>,<extract regex>
```

| Field | Meaning |
|---|---|
| `exprID` | a name for the rule; also groups continuation lines |
| `include regex` | the line must match this to be considered; empty means all lines |
| `exclude regex` | the line is skipped if it matches this; empty means exclude nothing |
| `match specification` | what to build from the captures — section 4 |
| `extract regex` | the pattern whose capture groups feed the specification |

Fewer fields are accepted — three and four both construct a rule — but the five-field form is the one to write, because it is the only one where every part is explicit.

The include and exclude patterns exist for **speed**, not just selection. They are cheap tests run before the expensive extraction regex, so a narrow include pattern on a high-volume log is the difference between keeping up and falling behind.

The extract regex is applied with `matches()`, not `find()`, so **it must match the line from end to end**. A pattern that would find its target comfortably in the middle of a line never fires; anchor it, and pad with `.*` on either side if the rest of the line is uninteresting.

## 3.2 The two-field continuation form

```
<exprID>,<match specification>
```

A line with two fields **adds another metric to an existing rule**, reusing the same capture groups. One log line often carries several numbers, and this is how you emit all of them without matching the line more than once:

```
Exp-1,,,Date{MMMddyyyy}[$1$2$3]_Time{HHmmss}
[$4$5$6]_Component[%7]_Type[%y]_Metric[Total]_Target[%8]_Value[$9],^(...) (..
Exp-1,Date{MMMddyyyy}[$1$2$3]_Time{HHmmss}
[$4$5$6]_Component[%7]_Type[%y]_Metric[Current]_Target[%8]_Value[$A]
Exp-1,Date{MMMddyyyy}[$1$2$3]_Time{HHmmss}
[$4$5$6]_Component[%7]_Type[%y]_Metric[AvgRate]_Target[%8]_Value[$B]
```

One match, three metrics, differing only in `Metric[]` and which group supplies the value. **The continuation line must come after the rule that defines the ID**; if it does not, SynchroTail logs `invalid expression line` and prints the expected format.

# 4. The match specification

This is the heart of the tool and the part that cannot be guessed. A specification is a list of tokens joined by underscores:

```
Date{MMMddyyyy}[$1$2$3]_Time{HHmmss}
[$4$5$6]_Component[%7]_Type[%y]_Metric[Total]_Target[%8]_Value[$9]
```

Read it as: build the date from groups 1, 2 and 3 parsed as `MMMddyyyy`; the time from groups 4, 5 and 6 as `HHmmss`; the component from group 7 cleaned; the type from the default set on the command line; a literal metric name `Total`; the target from group 8 cleaned; and the value from group 9 as-is.

## 4.1 The tokens

| Token | Purpose |
|---|---|
| `Date[...]` | the date part of the timestamp |
| `Date{fmt}[...]` | the same, parsed with `fmt` |
| `Date[]` | use the server's date instead of the log's |
| `Time[...]`, `Time{fmt}[...]`, `Time[]` | the time part, likewise |
| `TimeUS[...]` | a microsecond-resolution time |
| `Component[...]` | the Synchro COMPONENT |
| `Type[...]` | the Synchro TYPE |
| `Metric[...]` | the Synchro METRIC |
| `Target[...]` | the Synchro TARGET |
| `Value[...]` | the number being measured |
| `TRID[...]` | transaction identifier — section 6 |
| `TR-START`, `TR-INTER`, `TR-END` | this line marks that point in a transaction |
| `TTL[ms]` | how long an open transaction may live; `TTL[]` means 5000 |
| `MaxOpen[n]` | how many transactions may be open at once; `MaxOpen[]` means 100 |
| `DTIME` | emit the delta since the previous value rather than the value |
| `BREAK` | stop evaluating further expressions once this one matches |
| `MICROBURST{Nms}`, `MICROBURST{Nus}` | sub-second bucketing at the given slice |

`fmt` is a Java `SimpleDateFormat` pattern, so `MMMddyyyy`, `HHmmss`, `yyyy-MM-dd` all behave as they do elsewhere in Java. `Date{SOLA}` selects a built-in format rather than a pattern.

Note the asymmetry in the empty forms: **`Date[]` and `Time[]` mean "use the server clock"**, while `Component[]`, `Metric[]`, `Target[]` and `Value[]` mean "take this from the next positional group". They look alike and do not behave alike.

The positional form also **skips the cleaning and truncation** described in section 4.2.1: it returns the capture group untouched. `Component[]` can emit characters that `Component[$1]` would have rewritten, so prefer the bracketed form for anything that becomes part of a key.

## 4.2 `$` and `%` are not the same

This is the single most important detail in the language, and the easiest to get wrong because the two look interchangeable in a working rule.

| Form | Gives you |
|---|---|
| `$1` | capture group 1 as written |
| `%1` | capture group 1 **with every character outside `A-Z a-z 0-9 _` deleted** |

`%` does not replace invalid characters, it **removes** them. A group containing `order-42/A` becomes `order42A`. Two different sources can therefore collapse onto one target name without any warning.

Use `%` for COMPONENT, TYPE, METRIC and TARGET, where the text becomes part of a metric key and stray punctuation would corrupt it. Use `$` for `Value[]`, where you want the number as written, and for date and time groups, where the format pattern must see the original characters.

### 4.2.1 Both forms are cleaned afterwards

`$` is not "raw". After substitution the **whole** assembled field passes through a final clean that rewrites every character outside `A-Z a-z 0-9 @ .` – to a hyphen, and then truncates the result to **30 characters**.

The two forms therefore differ in how they treat punctuation, not in whether it survives:

| Captured | `Target[$1]` | `Target[%1]` |
|---|---|---|
| `order-42/A` | `order-42-A` | `order42A` |
| `TIME_WAIT` | `TIME-WAIT` | `TIME-WAIT` |

Two consequences worth keeping. An underscore **cannot** reach a metric key by either route, so the key-splitting hazard that affects other collectors does not arise here. And a field longer than 30 characters is silently shortened — another way for two distinct sources to collapse onto one name.

## 4.3 Group numbers above nine

Capture groups past 9 continue **into the alphabet**:

| Reference | Capture group |
|---|---|
| `$0` | the whole match |
| `$1` … `$9` | 1 … 9 |
| `$A` | 10 |
| `$B` | 11 |
| `$C` … `$Z` | 12 … 35 |

So `Value[$F]` is group 15. The same applies to `%A` … `%Z`.

This is worth pausing on, because a rule using `$A` reads as though it might be a literal `A`, and a regex with ten or more groups is common in log parsing. When counting groups, remember that **every** opening parenthesis that is not `(?:` counts, including ones you added only for alternation.

## 4.4 Defaults from the command line

Four shortcuts insert the values given by the `component=`, `type=`, `metric=` and `target=` options:

| Reference | Inserts |
|---|---|
| `$c` or `%c` | the `component=` option |
| `$y` or `%y` | the `type=` option |
| `$m` or `%m` | the `metric=` option |
| `$t` or `%t` | the `target=` option |

`Type[%y]` throughout a rule set, with `type=` given on the command line, lets one expression file serve several deployments — the same pattern as `SynchroType` in SynchroSocket.

## 4.5 Computed values

A token containing parentheses is evaluated as a **JavaScript expression**, after positional substitution:

```
Value[($9 * 1000)]
Value[(bytesNew - bytesOld)]
```

Variables you have defined in earlier tokens are in scope. A variable named `xOld` holds the previous value of `x`, which is how rates and deltas are written; if the `Old` value does not exist yet — on the first matching line — the result is skipped rather than reported as a spurious figure.

> **Portability warning.** This uses the JVM's built-in JavaScript engine (Nashorn), which was removed in **Java 15**. SynchroTail builds for Java 8 and runs correctly on Java 8 and 11. On a newer JVM the engine is absent, and because the failure is caught internally, computed values would fail **quietly** rather than stopping the process. Run SynchroTail on Java 8 or 11, or verify every computed rule before moving to a newer JVM.

## 5. A worked rule

Suppose the application writes:

```
Sep 29 14:22:07 gateway-01 orders processed=1450 queued=38 avgUs=812
```

and you want three metrics per line, timestamped from the log rather than the clock.

```
Exp-Orders,orders processed,,Date{MMMddyyyy}[$1$2$3]_Time{HHmmss}
[$4$5$6]_Component[%7]_Type[%y]_Metric[Processed]_Target[orders]_Value[$9],^(\w{3})
+(\d+) +(\d+):(\d+):(\d+) +(\S+) +orders processed=(\d+) queued=(\d+)
avgUs=(\d+)
Exp-Orders,Date{MMMddyyyy}[$1$2$3]_Time{HHmmss}
[$4$5$6]_Component[%7]_Type[%y]_Metric[Queued]_Target[orders]_Value[$A]
Exp-Orders,Date{MMMddyyyy}[$1$2$3]_Time{HHmmss}
[$4$5$6]_Component[%7]_Type[%y]_Metric[AvgUSec]_Target[orders]_Value[$B]
```

Points to notice:

- The include pattern `orders processed` is a cheap literal test that rejects every other line before the full regex runs.
- The year is absent from the log, so `Date{MMMddyyyy}[$1$2$3]` composes month and day from the log with a year the parser supplies.
- `Component[%7]` cleans the host name; `Value[$9]` does not clean the number.
- `$A` and `$B` are groups 10 and 11 — the third and fourth numbers — reached from continuation lines that never re-match the line.

# 6. Transactions

A transaction is a unit of work that spans **two or more log lines**: a request logged on arrival and again on completion. SynchroTail correlates them and reports what the application never printed — how many are in flight, and how long they took.

You need three things in your rules:

**An identifier.** `TRID[...]` names the value that links the lines — a request ID, a session, an order number. Two lines carrying the same `TRID` are the same transaction.

**A position.** `TR-START` on the rule matching the opening line, `TR-END` on the closing one, and `TR-INTER` for any intermediate stage you want visibility of.

**Bounds.** `TTL[ms]` discards a transaction whose end never arrives, and `MaxOpen[n]` caps how many may be open at once. Defaults are `TTL[]` = 5000 ms and `MaxOpen[]` = 100.

```
Exp-ReqStart,REQ BEGIN,,TRID[$2]_TR-
START_TTL[30000]_MaxOpen[5000]_Component[%1]_Type[%y]_Metric[Open]_Target[requests],^(\S+)
REQ BEGIN id=(\S+)
Exp-ReqEnd,REQ END,,TRID[$2]_TR-
END_Component[%1]_Type[%y]_Metric[Duration]_Target[requests],^(\S+) REQ END
id=(\S+)
```

**Set the bounds deliberately.** They are not tuning knobs, they are the memory limit. Every open transaction is retained until its end line arrives or its TTL expires, so a `TTL` longer than the slowest realistic transaction combined with a generous `MaxOpen` is how this becomes a memory leak on a busy system. Size `TTL` to slightly more than your worst acceptable response time, not to "just in case".

`DTIME` is the related simpler tool: it reports the change since the previous value rather than the value itself, which turns a monotonically increasing counter in a log into a rate without any transaction state.

# 7. Options

Options are given as `name=value` arguments on the command line, in any order. Names are case-insensitive.

## 7.1 Choosing files

| Option | Default | Meaning |
|---|---|---|
| `basedir` | — | directory to read from |
| `filepattern` | — | regular expression selecting files within it |
| `sortby` | `filename` | `filename`, `time`, or `single` for one fixed file |
| `firstfile` | — | begin at this file rather than the first |
| `firstfileoffset` | — | begin at this byte offset within it |
| `input` | `FILE` | `FILE`, `STDIN`, `HTTP`, `STAT` |

`sortby` decides the order a rotated set is replayed in. `time` follows modification time, `filename` follows lexical order — which is correct only if the rotation scheme zero-pads or uses sortable dates. Choose deliberately: getting this wrong replays a day's logs out of order, and the timestamps in the metrics will be right while their arrival order is wrong.

## 7.2 Following and finishing

| Option | Default | Meaning |
|---|---|---|
| `eoftimeout` | `-1` | seconds to wait at end of file before giving up; `-1` waits forever |
| `filetimeout` | `-1` | seconds to wait for the next file |
| `notifytimeout` | `1` | seconds between checks for new data |
| `peeknextfiledelay` | `1` | seconds before looking for a successor file |
| `stoptaildelay` | `1` | seconds before stopping a tail |
| `monitortime` | `60` | seconds between internal monitoring reports |

| Option | Default | Meaning |
|---|---|---|
| `ztail` | `no` | follow compressed files |

`eoftimeout=-1` is the difference between a **live monitor** and a **batch run**. Left at the default the process follows the log indefinitely; set to a small number it processes what exists and exits, which is what you want when reprocessing history.

## 7.3 Throughput

| Option | Default | Meaning |
|---|---|---|
| `lps` | see below | lines per second ceiling |
| `mbps` | — | megabytes per second ceiling; sets the buffer sizes to match |
| `buffersize` | 1048576 | read buffer, bytes |
| `bufferpersecond` | `10` | buffer reads per second |
| `cachebuffercount` | `3` | buffers held in the cache |

> **A trap worth knowing.** If you supply an `expressionfile` and do **not** supply `lps`, SynchroTail forces `lps=100` and logs a warning. One hundred lines per second is far below what a busy log produces, so a rule set that works perfectly in testing can silently fall behind in production. Always set `lps` explicitly. Production configurations use values such as `lps=100000000`, which is effectively unlimited.

## 7.4 Identity and destination

| Option | Default | Meaning |
|---|---|---|
| `synchroserver` | — | SynchroServer host |
| `serverport` | `9123` | SynchroServer port |
| `component` | `SynchroTail` | value inserted by `$c` / `%c` |
| `type`, `metric`, `target` | — | values inserted by `$y`, `$m`, `$t` |
| `synchrotailid` | `SynchroTail` | identifies this instance |
| `output` | `STDOUT` | `STDOUT`, `SOCKET`, `HTTP`, `QUIET` |
| `outputsocketaddress`, `outputsocketport` | — | for `output=SOCKET` |
| `outputhttpaddress`, `outputhttpport` | — | for `output=HTTP` |
| `inputhttpaddress`, `inputhttpport` | — | for `input=HTTP` |

| Option | Default | Meaning |
|---|---|---|
| `inputstataddress`, `inputstatport` | — | for `input=STAT` |

## 7.5 Restart, logging and the rest

| Option | Default | Meaning |
|---|---|---|
| `restartfilename` | `restart.ini` | where file positions are saved |
| `restartallowoverlap` | `yes` | permit re-reading a little on restart |
| `expressionfile` | — | the rule file |
| `encoding` | `utf-8` | character set of the logs |
| `logfile` | `STDERR` | SynchroTail's own log |
| `verboselevel` | `ERROR` | `ERROR`, `WARN`, `INFO`, `DEBUG`, `TRACE`, `QUIET` |
| `delimiter` | newline | line separator, for logs that do not use one |
| `responsetimelimit` | — | threshold used by the aggregator |
| `nofailstarttime`, `nofailstoptime` | — | window in which missing files are not an error |
| `scriptpath`, `scriptdelay` | — | external script run alongside |
| `filelineprefix`, `rtclasses` | — | see the source; not covered here |

`restartallowoverlap=yes` re-reads a small amount on restart rather than risk skipping. That means a restart may **duplicate** a few metrics rather than lose them — the right trade for monitoring, but worth knowing when a chart shows a brief doubled value after a restart.

`nofailstarttime` and `nofailstoptime` describe a window — typically the overnight gap when an application is down — during which the absence of a log file is expected rather than an error worth reporting.

# 8. Running it

The application is a single jar with `ca.synchro.tail.Startup` as its main class, so `java -jar SynchroTail.jar` is all that is needed. It requires **Java 8 or 11** (see the Nashorn warning in section 4.5).

On startup it logs the version it was built with:

```
INFO: SynchroTail Version: 142.0000
```

That number is not a typo for 1.4.2 — it is the same version in the numeric form the product has always used. The value is published as a `VERSION` metric, so it has to be a number rather than a string, and `142` is how 1.4.2 is encoded. Do not put a dotted version such as `1.4.1` in the jar manifest: the parser reads the text before the first `-` and after the first `.`, and a second dot makes it throw at startup.

Batch: process what exists and exit, printing to the console rather than sending anywhere.

```
java -jar SynchroTail.jar \
      basedir=/var/log/app filepattern='app\.log\..*' sortby=time \
      expressionfile=Expression.conf lps=100000000 \
      eoftimeout=5 filetimeout=5 output=STDOUT
```

Live: follow the log set and report to SynchroServer.

```
java -jar SynchroTail.jar \
      basedir=/var/log/app filepattern='app\.log.*' sortby=filename \
      expressionfile=Expression.conf lps=100000000 \
      synchroserver=live5.synchro-tech.com serverport=9123 \
      component=gateway-01 type=APP \
      restartfilename=/var/lib/synchrotail/restart.ini \
      logfile=/var/log/synchrotail/tail.log verboselevel=ERROR
```

## 8.1 Developing a rule set

Work in this order. Each step removes a class of error before the next can hide it.

1. **`output=QUIET verboselevel=TRACE`** — confirms the file is found, the expression file parses and lines are being read. Nothing is sent.
2. **`output=STDOUT`** on a copied sample of the log — shows the metrics your rules actually produce. Check the target names especially: this is where `%` silently removing characters becomes visible.

3. **`eoftimeout=5`** against real files — confirms behaviour on the true format and volume, then exits.
4. Only then point it at SynchroServer, with `lps` set explicitly.

Keep the sample log from step 2. When a rule set stops working after an application upgrade changes a log format, having the old sample makes the difference obvious in seconds.

# 9. Operating it

## 9.1 Restart behaviour

Positions are written to the restart file, and a shutdown hook saves them on exit. The file is a complete `name=value` dump of the options in force, plus the two that record how far processing reached:

```
firstfile=app3
firstfileoffset=4096
```

**Restarting does not resume by itself.** Nothing reads that file back — it is written, never loaded. Start SynchroTail again with the same arguments and it re-reads the log set from the beginning, double-counting every metric in it.

To resume, feed the restart file back as arguments. It is already in `name=value` form, one per line, so that is:

```
java -jar SynchroTail.jar $(cat restart.ini)
```

Any launcher expected to survive a restart must do this, and it is worth testing that it does before relying on it. A hard kill may lose the most recent positions, in which case `restartallowoverlap` decides whether the gap is re-read.

## 9.2 Troubleshooting

| Symptom | Likely cause |
|---|---|
| `No valid expression found in expression file` | file missing, or every line failed to parse |
| `invalid expression line` | wrong field count, or a continuation line before its rule |
| Rule never fires | include pattern too narrow, or an unquoted comma split the line |
| Targets missing characters | `%` deleting them; use `$` or clean in the regex |
| Metrics stop under load | `lps` left at the forced default of 100 |
| Computed values silently absent | Nashorn missing — running on Java 15 or newer |
| Values doubled briefly after restart | `restartallowoverlap=yes`, working as intended |

| Symptom | Likely cause |
|---|---|
| Memory growth over hours | transactions opened and never closed; check `TTL` and `MaxOpen` |

Raise `verboselevel` to `TRACE` to see each expression as it is loaded and matched. It is extremely verbose; use it on a sample, not a live high-volume feed.

## 9.3 Tests

The project carries a JUnit suite, run with `mvn test`:

| Area | Covers |
|---|---|
| Match specification | `$` and `%`, group letters past `$9`, the cleaning and 30-character truncation, literals, positional forms, dates and times, transaction tokens, `BREAK` |
| Expression file | field counts, quoting, continuation lines, comments, rejection of an orphan continuation |
| Options | the version encoding, including that a dotted version throws |
| File discovery | selection by pattern, recursion, empty directories |
| End to end | file ordering, `firstfile`, a pattern matching nothing, a file appearing mid-run, and the restart behaviour in 9.1 |

The end-to-end cases run SynchroTail as a child process rather than inside the test JVM, because the aggregator calls `System.exit` in more than thirty places and any one of them would take the whole test run down with it.

These replace `run-tests`, which ran a similar set of scenarios and compared the results to nothing.

## 9.4 Known limitations

**Nashorn dependency.** Computed values need Java 8 or 11, as described in section 4.5. The suite deliberately does not cover computed values: the result would depend on the JVM running the build rather than on the code.

**The expression language is unforgiving.** There is no validation pass. A rule that parses but references a capture group the regex does not have fails at match time, not at load time.

**MICROBURST is untested.** Its parsed result is not reachable from outside the class, so the suite does not cover it.