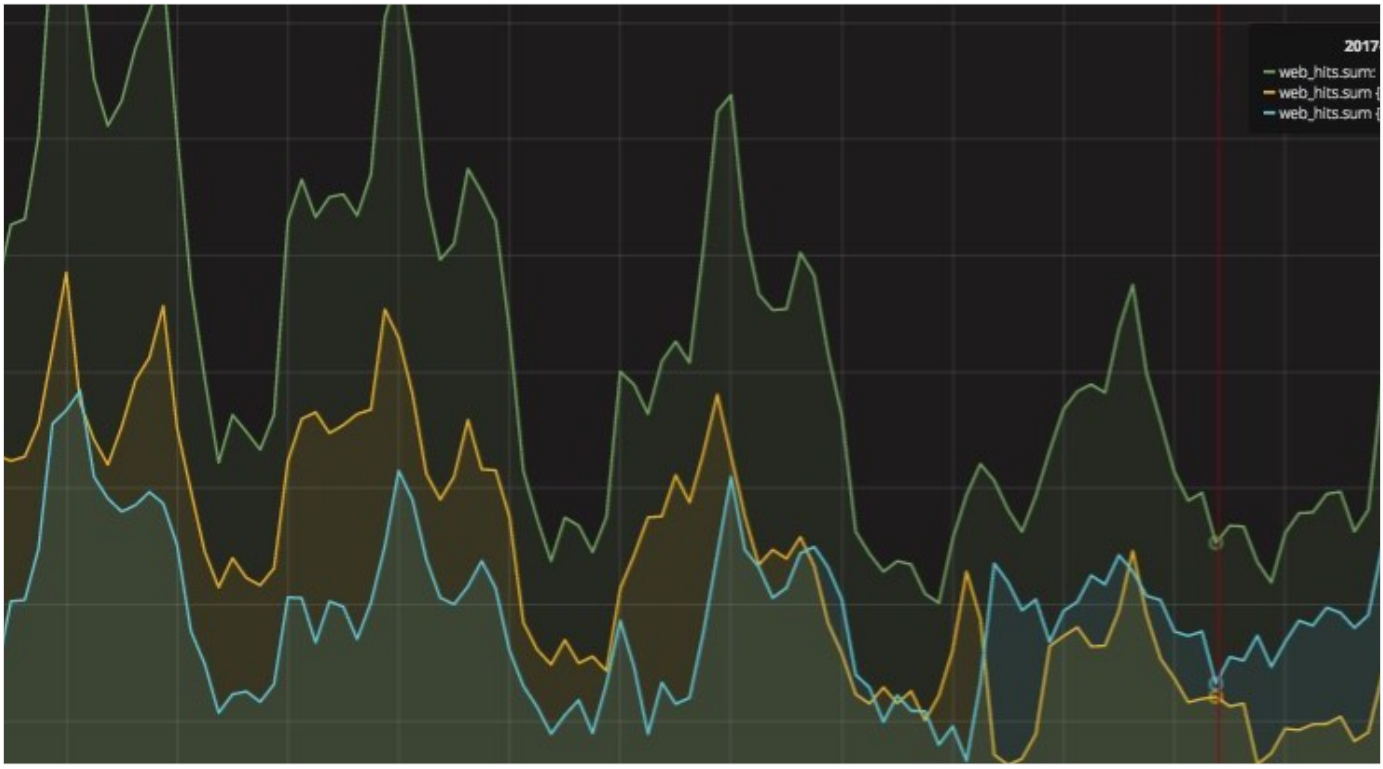
Behind the scenes - Continuous log analytics



Here at Loway, we take data very seriously. QueueMetrics Live, our hosted call-center analytics platform for Asterisk PBXs, is constantly monitored and probed for anomalies in its performance metrics. This way, we are able to find and fix issues - often automatically - before they become apparent to our growing number of users.

The goals

During the years we developed a number of scripts using different technologies (Perl, bash, Ruby) that go search for specific patterns of interest across our logs, categorize them and push them to an InfluxDB database for central anomaly detection. As our platform is built on top of a

good number of servers that are commissioned and decommissioned according to expected occupancy and behavioral patterns, we needed a way to correlate events happening on different layers and understand whether they are caused by anomalies at the customer level or at the server level.

Most of these scripts read data that comes in as HTTP logs, where we go look for specific parameters (type of request, elapsed time, response size, errors....) and we categorize these calls into different "buckets". Categorization rules evolve and change over time (for example: all QueueMetrics instances are monitored by feeding in transactions and checking that results are correct and returned within a specific time frame, but we make a distinction between transactions run for automated monitoring versus transactions started by our customers), while the basic format for all our queries is always reading incremental logs, processing new lines and returning results to InfluxDB.

We saw an opportunity for Clojure as a way to do this task, as it seemed well suited to ETL-style data categorization and transformation, and being very terse means that the code that has to be maintained ended up being smaller than the Ruby equivalent.

Implementation

Is it fast enough?

As Clojure runs on the JVM, it has performance characteristics that match the JVM's - so it is very efficient in working on large homogeneous batches, but we feared its startup times would pose an excessive load especially on busy (and not especially powerful) web proxies.

So we run some tests by assembling together all the libraries we needed and initializing them, to measure the "cost of Clojure":

```
$ time java -jar target/l2i-standalone.jar


real    0m1.986s
user    0m3.976s
sys     0m0.213s
```

This ends up being acceptable, because those jobs are called by a cron at given intevals; as these are “dry runs” in which no data is actually processed, this is a bottom-line fixed cost for adoption.

Log processing in Clojure

Once we saw that the Clojure + JVM were an acceptable environment that we could run on our production servers without excessive load, we started analyzing logs with Clojure. And this experience was very good.

```
(:require [clash.core :as clash])


(defrecord LogRecord

    [ip date time method uri referrer status useragent bytes])


(def log-pattern #"^(\S+) \S+ \S+ \[(.....):(.....).+?\] .(\S+) (\S+) \S+ (\d+) (\d+) \"(\S+)\" \"(.+?)\"")

(defn log-parser [line]

    (if-let [[_ i d t m u s b r ua] (re-find log-pattern line)]

        (->LogRecord i d t (keyword m) u r (keyword s) ua (parse-int b))))


(defn load-log-verbose

    [file]

    (clash/transform-lines-verbose file log-parser :max 10000000))
```

This is basically all the logic you need to parse file and store them into a common parsed structure. We created a LogRecord to store data, a parser that uses a regexp to find data in logs, and we use the excellent (and quite performant) library Clash to parse them very efficiently in-memory. Clash then gives us the results of parsing and a list of lines where parsing failed, that we store for further inspection.

As we only want to read incremental lines, we save a “brainfile” that stores the file we are parsing, a SHA of its first line and the current offset. If the file remains the same (that it, the first

line is not changed) we read from the current offset onwards and write a new brainfile; if it has changed, we read it from the beginning.

This required very little code: using Cheshire for reading/writing JSON and Clojure Spec we can easily handle the case where the file is non existent and return a “dummy” file instead of an exception. We can also check (as a post-condition) that what we read was actually the right kind of file.

```
(defn load-brain-file
    [filename]
    {:post [(spec/valid? ::brainfile %)]}
    (try
        (let [contents (slurp filename)]
            (json/parse-string contents (fn [k] (keyword k))))
    (catch Exception e {:sha    ""
                                    :offset 0})))


(defn save-brain-file
    [brainfile filename]
    (let [offset  (file-size filename)
                sha     (generate-file-signature filename)
                brain   {:sha       sha
                                :offset     offset
                                :filename   filename
                                :lastupdate (str (java.util.Date.))}]
        (json/generate-stream brain
                                        (clojure.java.io/writer brainfile))))
```

Business logic

Business logic - for each of the cases supported - ends up being a single, stateless function that maps over a sequence of LogRecords and returns a set of measurements to be added to InfluxDB. It is usually made up of a function (in our case below named `downloads-which`) that returns a hash of attributes or *nil* if we don't care about this specific record; then nils are stripped out and we count unique hashes.

```
(defn downloads-parse
    [logrecords]
    (let [items (map downloads-which logrecords)
                items-present (filter some? items)]
        (frequencies items-present)))
```

To run the loader, we pack everything into a single Uberjar and parse its command-line parameters. By using `clojure.tools.cli` creating a well-behaved command line tool with help and validation took us very little time.

As we have multiple distinct pipelines that measure different elements (e.g. instance activity, download links, data uploads, etc) and a server might have data for more than one pipeline, to avoid paying the price of starting the loader multiple times we create a "multibrain" file that is a JSON representation of an array of command line parameters that we want invoked in sequence. We validate our JSON input using Spec and then share most of the code with our command-line parser.

Final notes

The resulting script is packaged as an Uberjar; the resulting size is acceptable (~7M) though we had to be careful with transitive dependencies (e.g. the innocent looking URL-parsing library Cemerik made it 3x as big because it imports a lot of other libraries we did not need, and so we preferred apache.http.client using Java interop instead).

In the end, we run our loader with a simple cron job and a customized "multibrain" file for each server. When updating the loader, we deploy a new JAR file and it just works.

Developing analytics with a REPL proved to be a productive experience, because you can immediately see the results of what you are doing, so it is relatively easy to see the details and test different approaches. When then you find something, you can create a test so you are sure the behavior won't change by mistake with further interactions.

Java interop was also very important in getting things done, as it gave us a huge pool of libraries to choose from and made it possible to do low-level file access when needed.