Golang: Tracking 1% request timeouts with Sumologic complex log parsing

levelup.gitconnected.com/golang-tracking-1-request-timeouts-with-sumologic-complex-log-parsing-febf9e8ef59e

mourya venkat November 16, 2020

Introduction

mourya venkat

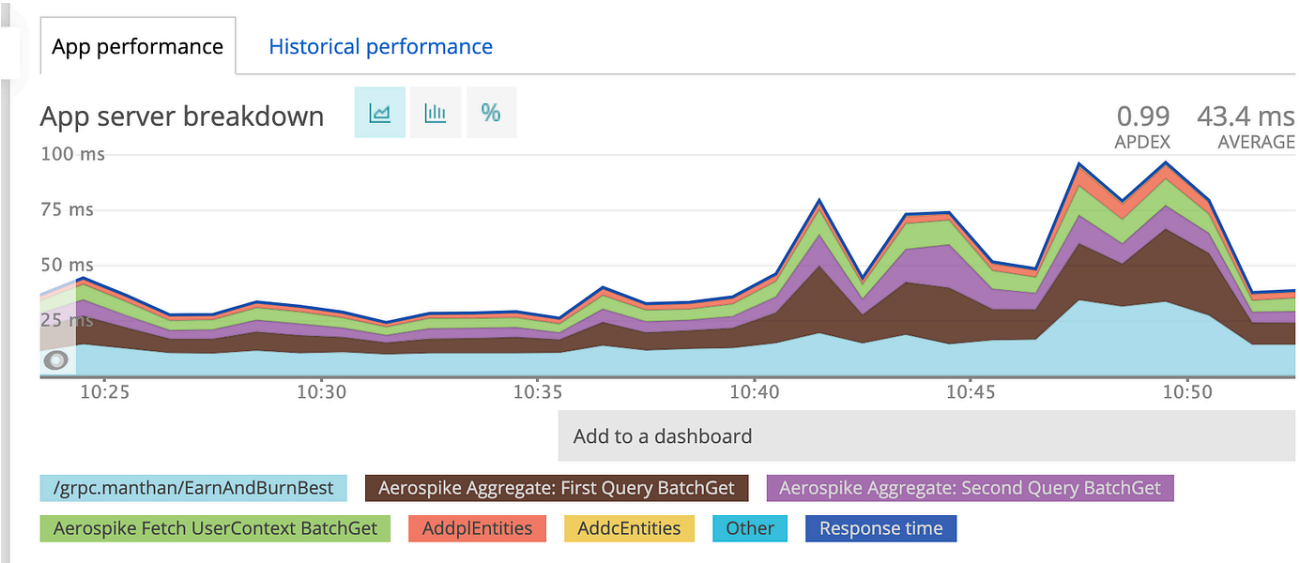
One primary factor that attracts the end-user is how quickly the application responds back to the user request. Applications perform pretty well when the traffic is low but things start failing once the load increases on the application.

Before taking the application to production, there will be rigorous benchmark testing on what's the max load that the application server or database server can handle. Based on the expected load and fluctuations, the service sets an SLA Timeout(Service Level Agreement) saying if processing any request takes more than this SLA time, the upstream service has to cancel the request with a DeadLine exceeded error.

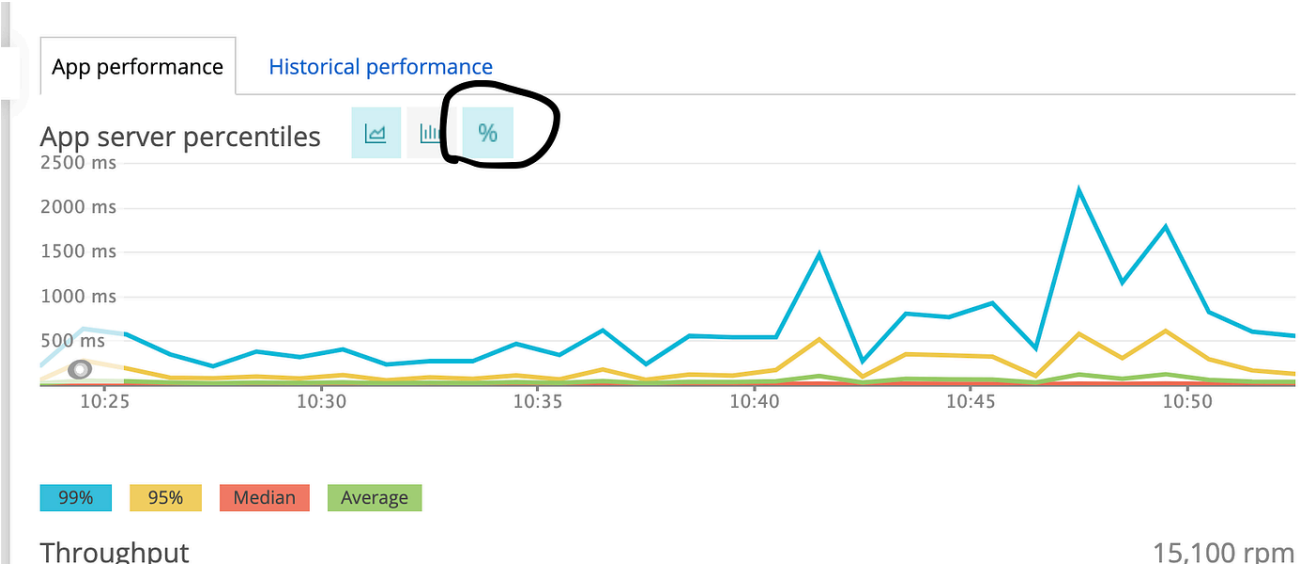
Problem Statement

The manual benchmarking is just a preliminary check to estimate the capacity of the system but we developers can't solely depend on the benchmark result. Hence to monitor real-time traffic information, we started using application monitoring tools like **NewRelic.**

NewRelic is the best in the market in terms of real-time application monitoring, insights, alerts, etc. But what I felt is it lacked in one thing.



The above picture depicts a GRPC Service with the average time taken for each sub-component. We identity the pain points of a service and start a transaction for each of the pain points so we can track them.



NewRelic also provides us with %(percentile) performance. Based on this, we can say the Median is below 50 ms, 95 % of requests are getting served in less than 500 ms and 4% of the requests are taking almost 2 seconds. Assume my service SLA is 250 ms. If we observe carefully, there are a few major things that are missing.

Working at a scale of almost 150K RPM, it's pivotal for us to track all these metrics in order to track the exact pain point and identify what went wrong.

Solution

That’s when we built a custom tracker module which calculates the time taken by each component and logs it to stdout.

Let’s get into details of code. My tracker has few struct fields.

- trackerEnabled -> This says if I have to track the request or not. Can be toggled dynamically at run time
- trackerName -> This resembles the API or GRPC Service Name for which we are tracking.
- trackerStartTime -> This states the time at which the request started.
- componentExecTrace -> This is a slice that contains the sub-component name and its execution time.
- slaTimeOut -> If the total processing time exceeds this, the component execution time will be logged.

Functionalities

- StartTransaction -> Starts a new transaction for sub-component
- EndTransaction -> Ends the started transaction
- LogComponentExecutionTrace -> Checks the diff b/w current time and requests start time in milliseconds and if the diff is greater than SLA Timeout, it logs the individual component execution trace.
- AddCanonicalLog -> If any specific information needs to be logged along with the same component trace, we can add it as part of canonical logs.

At the start of each API or service, we initialize the tracker with trackerName & slaTimeOut. (SLA timeout is made non-mandatory so, if we need to log all requests component execution trace we don't need any extra dev)

Usage

We initialize a tracker object on every API request and create a context out of it. As the context is part of each function as the first parameter, we don't need to pass this tracker as a separate argument to all sub-functions. If any of subfunction needs to get the tracker object, all they need to do is:

```
trackerObject = tracker.FromContext(parentContext)
```

This is how a sample log looks like.

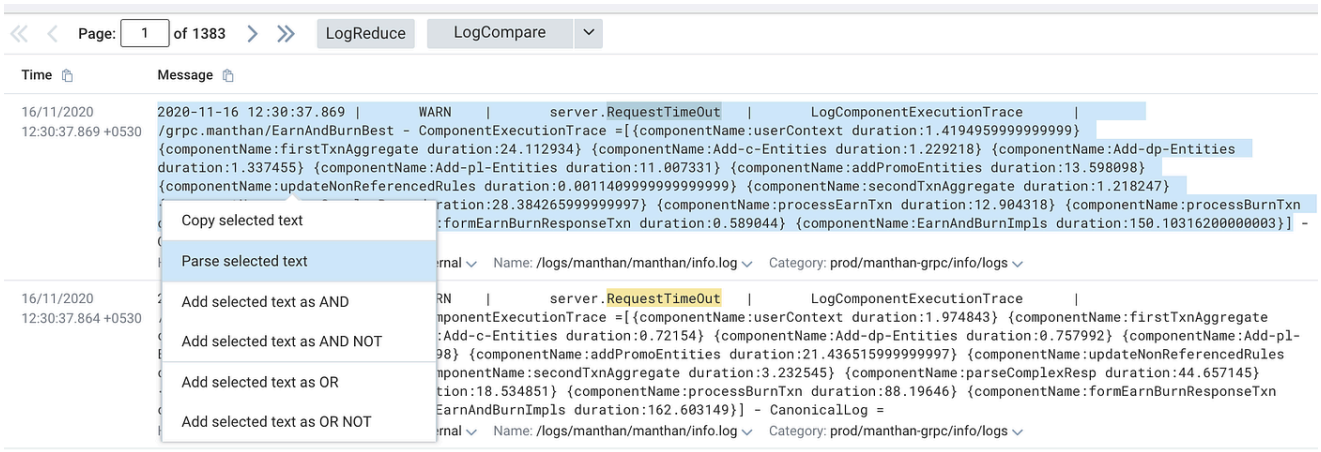
```
2020-11-16 12:30:37.869 |       WARN    |       server.RequestTimeOut   |
LogComponentExecutionTrace      |       /grpc.manthan/EarnAndBurnBest -
ComponentExecutionTrace =[{componentName:userContext duration:1.4194959999999999}
{componentName:firstTxnAggregate duration:24.112934} {componentName:Add-c-Entities
duration:1.229218} {componentName:Add-dp-Entities duration:1.337455}
{componentName:Add-pl-Entities duration:11.007331} {componentName:addPromoEntities
duration:13.598098} {componentName:updateNonReferencedRules
duration:0.0011409999999999999} {componentName:secondTxnAggregate duration:1.218247}
{componentName:parseComplexResp duration:28.384265999999997}
{componentName:processEarnTxn duration:12.904318} {componentName:processBurnTxn
duration:83.656057} {componentName:formEarnBurnResponseTxn duration:0.589044}
{componentName:EarnAndBurnImpls duration:150.10316200000003}]
```

Now it's time for us to parse the log, build a dashboard out of it and try figuring out which component is taking more time.

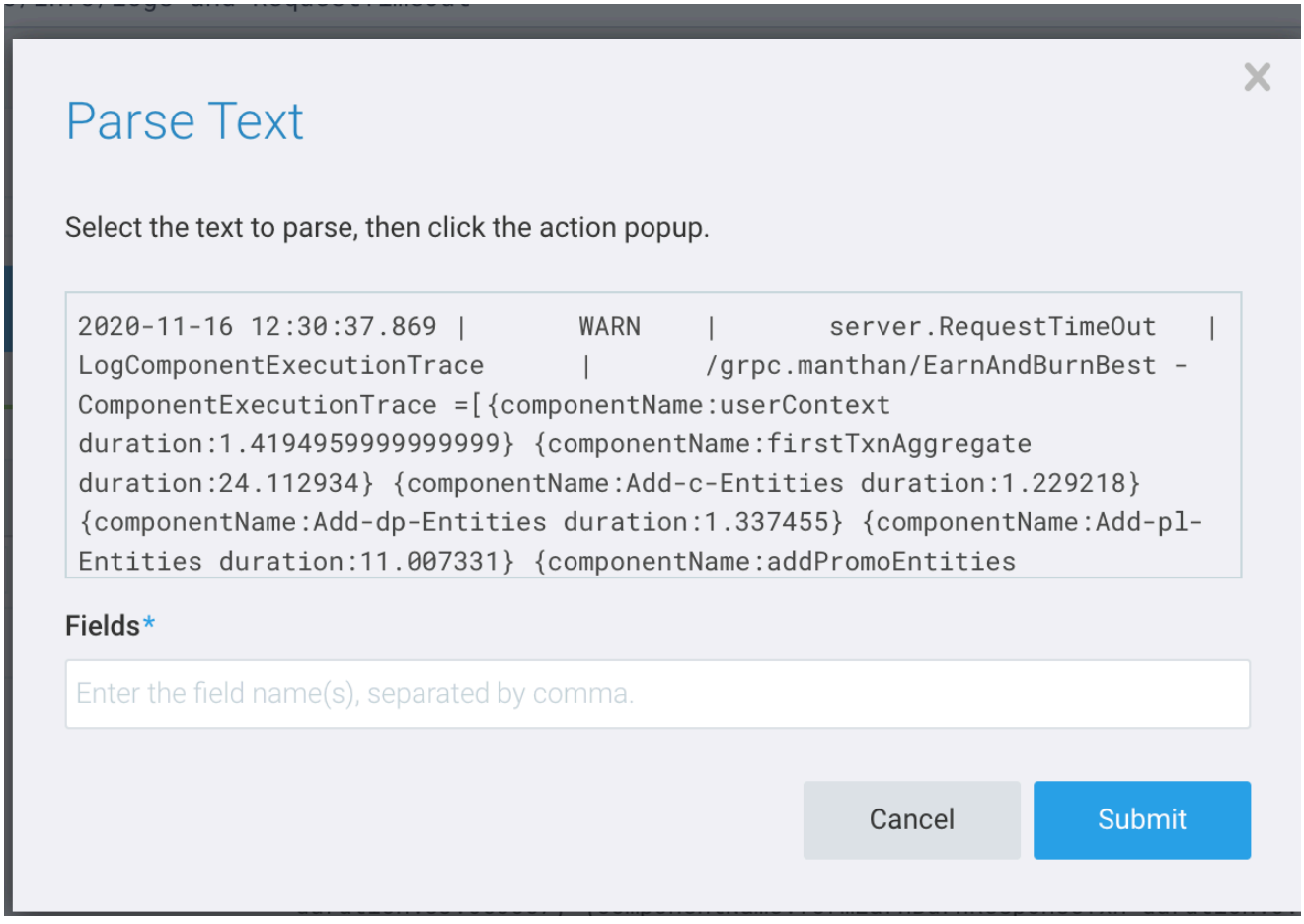
The log aggregator we use is **Sumologic**. It has two kinds of logging.

- Rest Client -> The application has to make a network call to the Sumologic rest-client on every Log Request.
- Background Listener -> Here we log it directly to stdout and the Sumologic listener keeps listening to all the changes and with the help of a cursor, it tracks all these logs and keeps pushing them to its server.

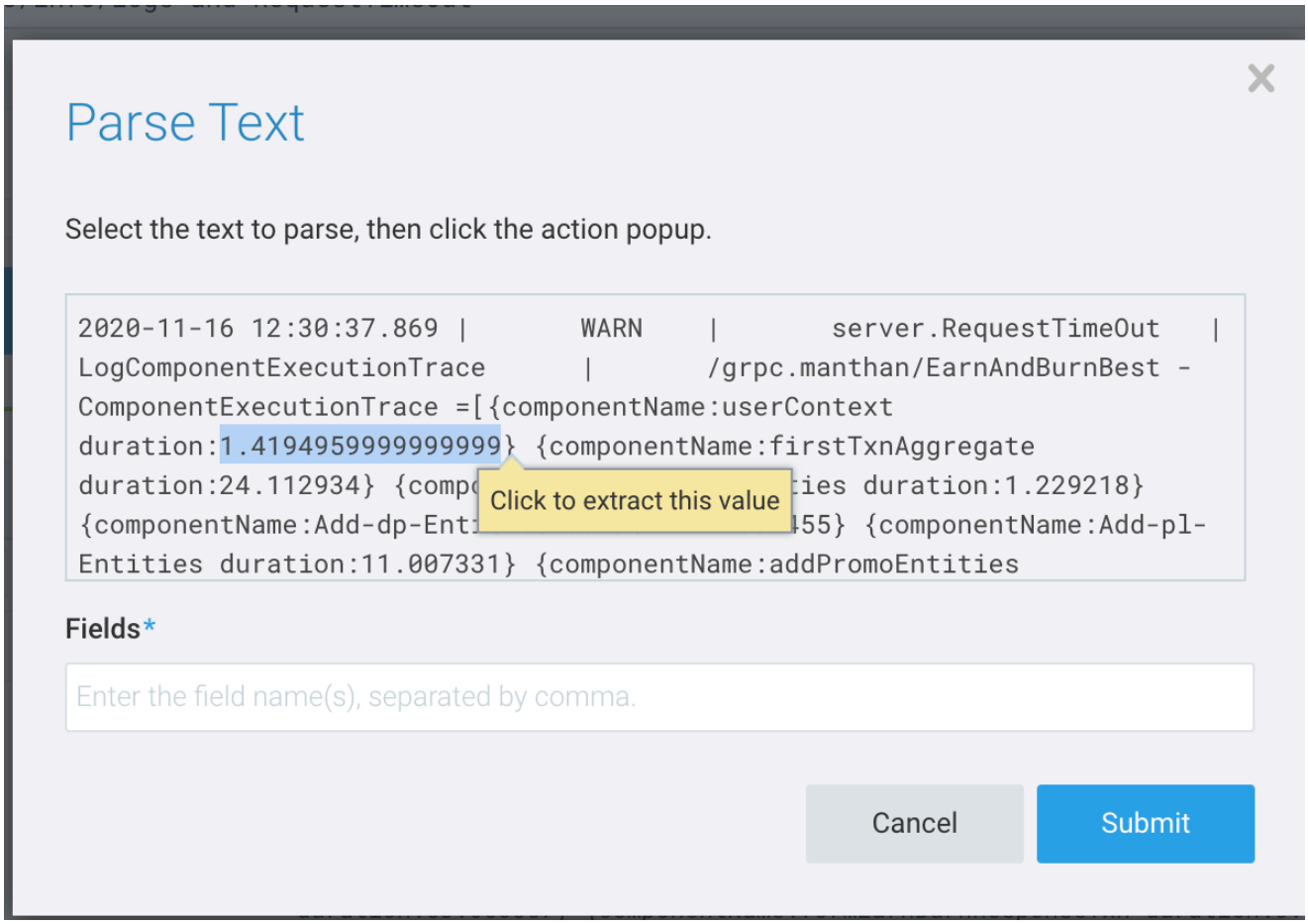
We prefer approach-2 as this reduces the integration of multiple applications with Sumologic Rest Client.



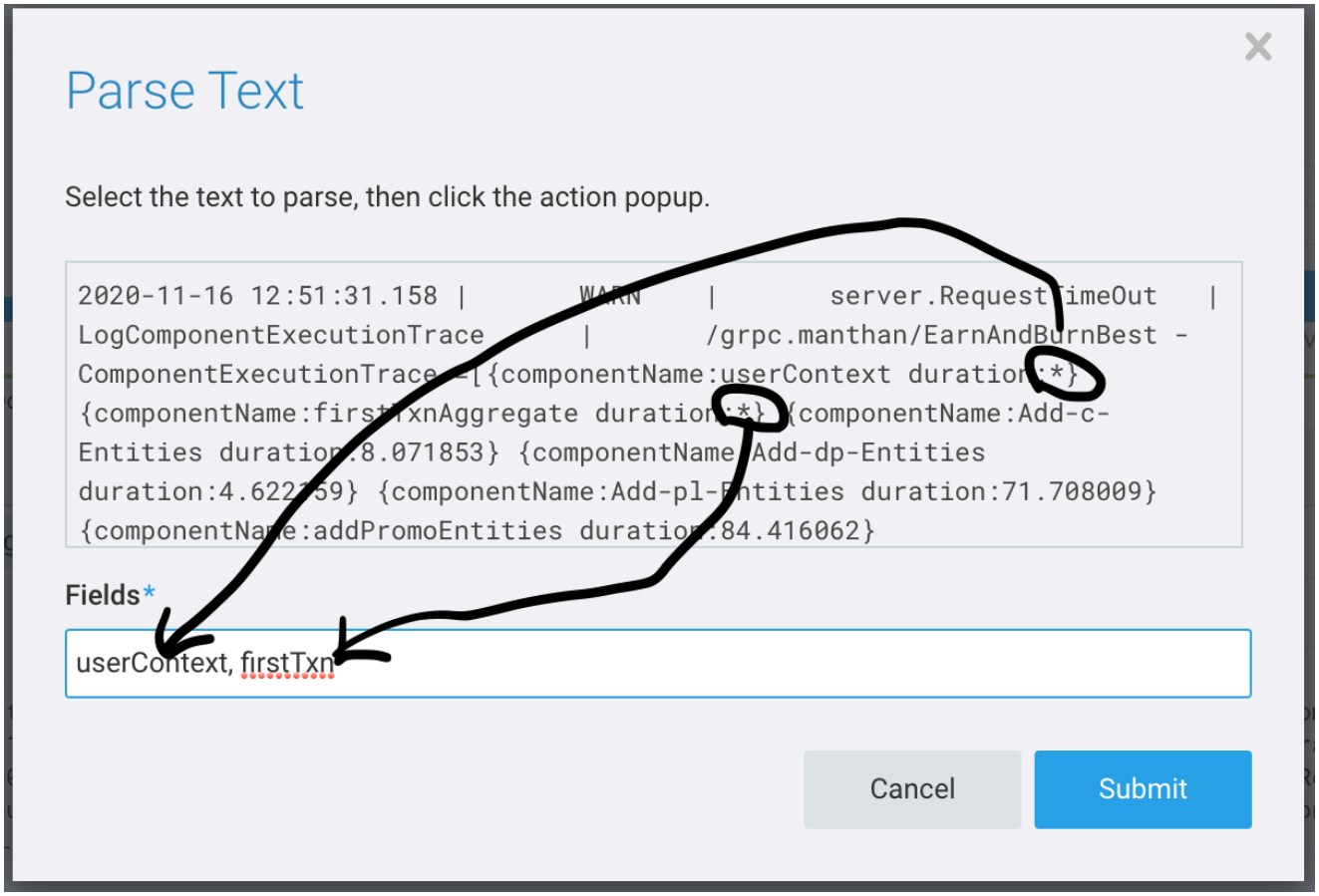
Open the sumologic dashboard, right-click on the log and select **Parse Selected Text.** Once clicked it opens up a dialog box.



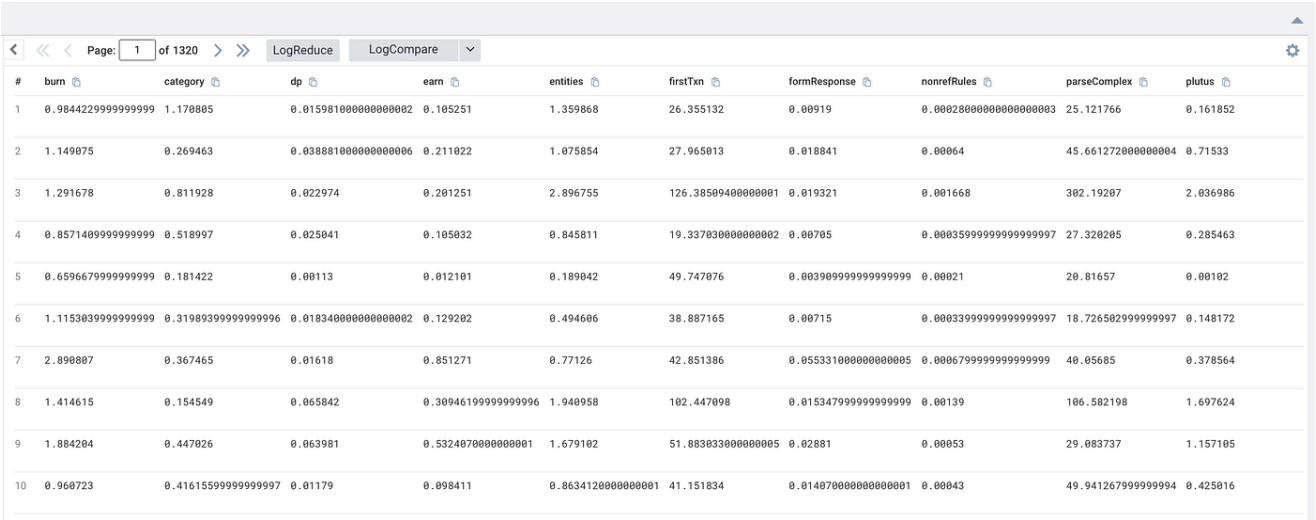
Now click on component execution time and you'll get a text like below.



Say I want to extract this value as a user context.



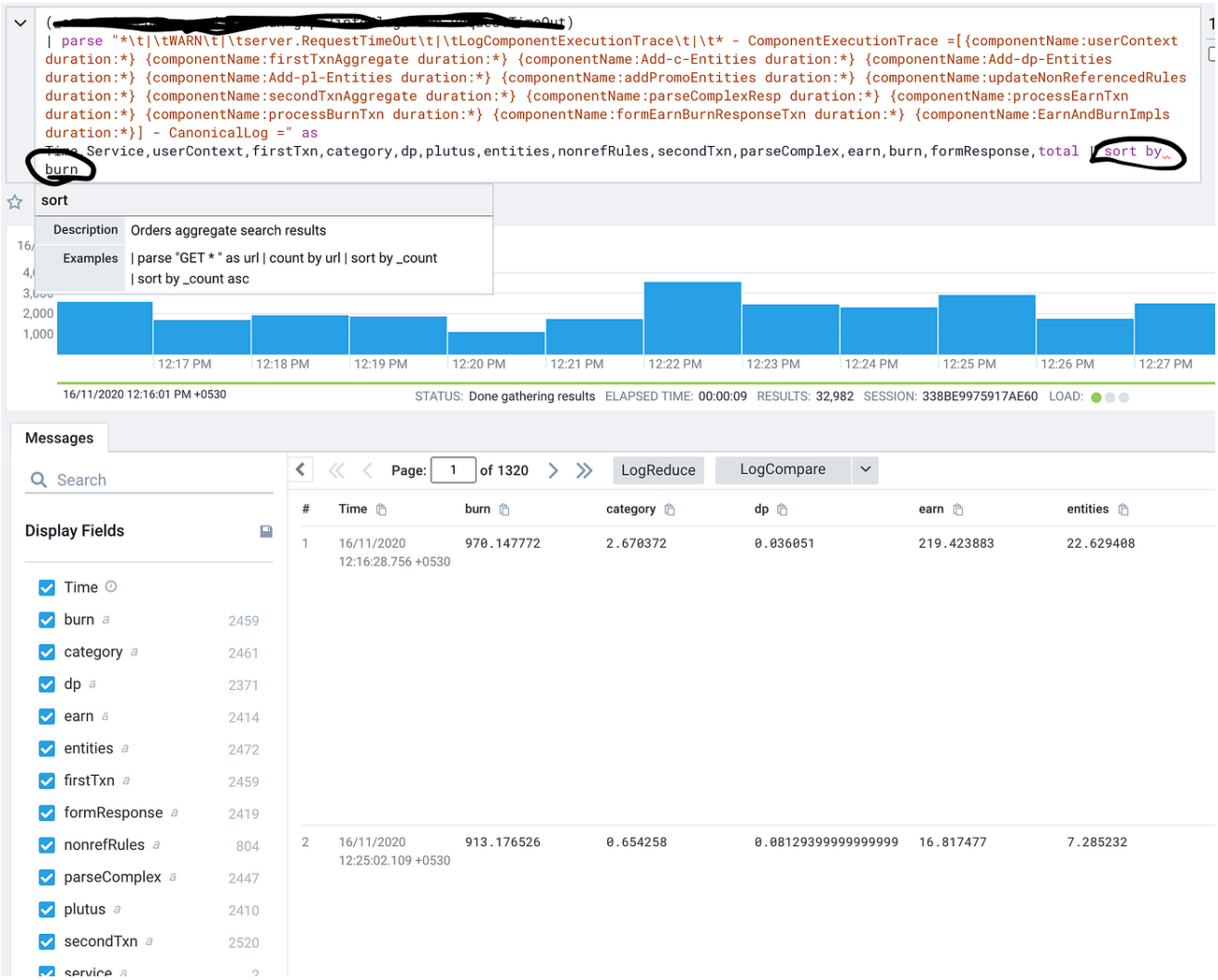
And the final result goes this way.



Page: 1 of 1320 LogReduce LogCompare

#	burn	category	dp	earn	entities	firstTxn	formResponse	nonrefRules	parseComplex	plutus
1	0.9844229999999999	1.170805	0.015981000000000002	0.105251	1.359868	26.355132	0.00919	0.0002800000000000003	25.121766	0.161852
2	1.149075	0.269463	0.038881000000000006	0.211022	1.075854	27.965013	0.018841	0.00064	45.661272000000004	0.71533
3	1.291678	0.811928	0.022974	0.201251	2.896755	126.38509400000001	0.019321	0.001668	302.19207	2.036986
4	0.8571409999999999	0.518997	0.025041	0.105032	0.845811	19.337030000000002	0.00705	0.00035999999999999997	27.320205	0.285463
5	0.6596679999999999	0.181422	0.00113	0.012101	0.189042	49.747076	0.003909999999999999	0.00021	20.81657	0.00102
6	1.1153039999999999	0.31989399999999996	0.018340000000000002	0.129202	0.494606	38.887165	0.00715	0.00033999999999999997	18.726502999999997	0.148172
7	2.890807	0.367465	0.01618	0.851271	0.77126	42.851386	0.055331000000000005	0.0006799999999999999	40.05685	0.378564
8	1.414615	0.154549	0.065842	0.30946199999999996	1.940958	102.447098	0.015347999999999999	0.00139	106.582198	1.697624
9	1.884204	0.447026	0.063981	0.5324070000000001	1.679102	51.883033000000005	0.02881	0.00053	29.083737	1.157105
10	0.960723	0.41615599999999997	0.01179	0.098411	0.8634120000000001	41.151834	0.014070000000000001	0.00043	49.941267999999994	0.425016

And now, if I need to check what's the max time burn component is taking.



With this complex log parsing, not only we can achieve sort functionality but also many aggregate functionalities where we can calculate 95 % request processing time, 99% processing time, etc.

Hope I covered what I've intended to. In case of any questions or queries do post them in the comments section and I'll reply back as soon as possible.