Implementation Notes - Resource Usage Tracker

Motivation

We need to create a reliable way to collect and store data for billing purposes. Also, provide the user with some statistics about their usage and credits left.

Data

Currently, we are interested in :

- Service start time
- Service end time
- CPU/Memory limits
- AWS Instance type

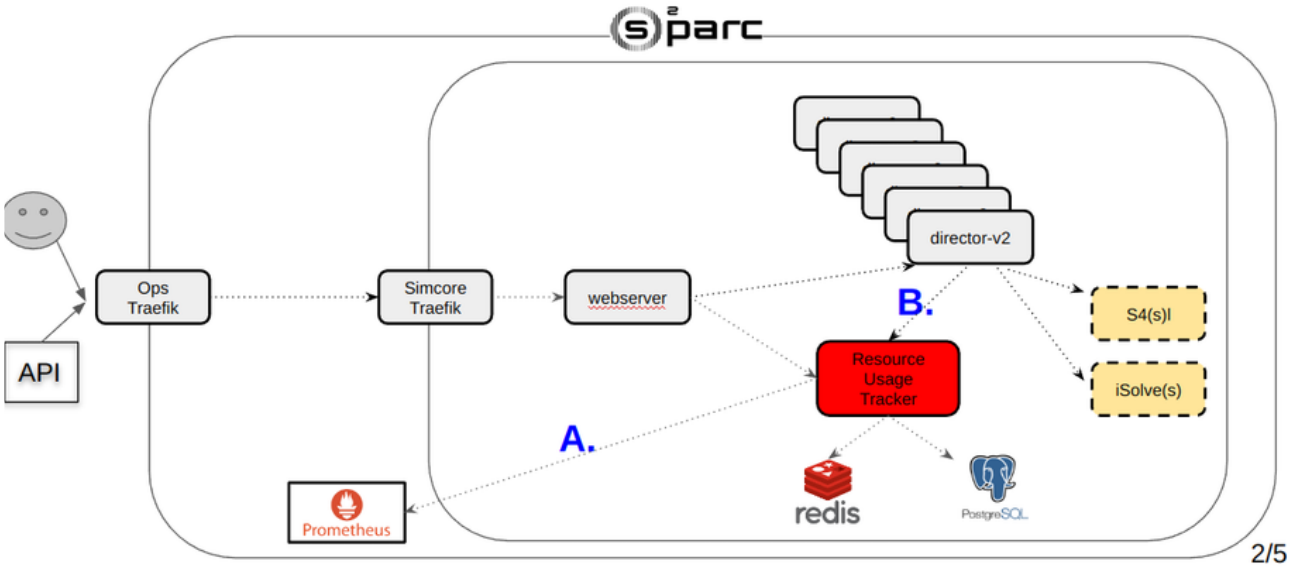
potentially also *Storage* and *Egress* should be discussed (I didn't comment on them in this implementation notes).

We have 2 data sources:

- **(A) Prometheus** (cAdvisor)
  - *PROS*: as it is on the container level, it works equally for all services (legacy/dynamic/computational). It is information from a lower level that tells us the real use of containers in the system.
  - *CONS*: ex. hanging sidecars (container might stay running, even though the user stopped the service). Or the container started to run, but the user for whatever reason didn't have access to the service.
- **(B) Application layer** (Director-v2, Director-v0)
  - *PROS*: here we have the closest information to the user when they started/stopped the service

- *CONS*: the start or stop request might be lost

In the following image, we can see a new resource usage tracker service in Osparc microservice architecture:



Prometheus (cAdvisor)

- lower level information about running containers
- we can get additional useful metrics that Prometheus provides us and potentially can be used for billing purposes (ex. `container_network_transmit_bytes_total`). But currently, we are mainly focused on how long the containers were running.
- we will run 2 separate Prometheus on different nodes, which means that the data are stored in two locations in case of loss. (preventive measure to improve robustness)

Two options for how to interact with Prometheus:

1. query all services periodically and store them somewhere in Database (ex. Postgres)

- *PROS*: We store needed data in a relational database which is then quickly accessible. We have database backups and availability guaranteed by AWS.
- *CONS*: PromQL doesn't support pagination, so we have to create the logic of what we query ourselves so we don't get too much data back (otherwise we might overload/kill Prometheus). Also, querying and storing data for all running services on the platform can also be a challenging task to make it robust.

```
SELECT * FROM "resource_tracker_container" LIMIT 50 OFFSET 750
```

Edit

Modify	container_id	user_id	product_name	service_settings_reservation_additional_info	container_cpu_usage_seconds_total	prometheus_created	prometheus_
	/docker/34e597118a2e0adf8d68b81d71c1315130e0c9e9a6148a57bcf061fcb2d2e06d	5	osparc	{}	16.491862043	2023-07-11 23:12:00+00	2023-07-11 23:
	/docker/072b06de0f6317b8b8f8894eb29e859716079e39cc79e09aacac079f3d4170a3	731	osparc	{}	16.634978219000004	2023-07-12 06:50:00+00	2023-07-12 06:
	/docker/ebd16b557da77b9ffbf1cd7e8208c622a41cb44401a9724d07572ae75e794c2e	723	osparc	{}	13.370155355999996	2023-07-12 08:42:00+00	2023-07-12 08:
	/docker/5ecfc6731c0b9c993bffe50cfe721e14f678836e805568cf7e2232324ab618a2	729	osparc	{}	16.868894998	2023-07-12 10:46:00+00	2023-07-12 10:
	/docker/a1f2a7e29b31940adb87892c0e86c92bd9a8c688e24e1ab6aaf7795668eb04c3	5	osparc	{}	16.606454266999997	2023-07-12 07:11:00+00	2023-07-12 07:
	/docker/e64d3e1511106ab8b67f36617f5892b134691c5a147297c417e26f901e04f3da	5	osparc	{}	6.308706546999999	2023-07-12 09:14:00+00	2023-07-12 09:
	/docker/28c74d6ea5dd749f7bfebe05824d307ff15379790dacccc03a4142ce7cbc04bc	732	osparc	{}	13.519360465	2023-07-12 10:47:00+00	2023-07-12 10:
	/docker/577a8717d075b0a368b7077d2d436e0c8bd96ae020e988af363469a8818a19a4	5	osparc	{}	6.238505134999999	2023-07-12 07:11:00+00	2023-07-12 07:
	/docker/b6c848a3cc16f15f4e5cb17f11c4bafa41043b143e7b3ea74eb6c9af2e3d4dc0	5	osparc	{}	16.772351669	2023-07-12 09:14:00+00	2023-07-12 09:
	/docker/a5840fb8faea0a75a4699bcfd0e6d0f9361339cc6757020ea9dcee17d6fa729f	731	osparc	{}	4.749657099999999	2023-07-12 10:46:00+00	2023-07-12 10:
	/docker/71a7e6ac1c490128336c57f90a9da4555d84dcb6d946bc5658562fe3455533de	5	osparc	{}	6.271038485	2023-07-12 11:11:00+00	2023-07-12 11:
	/docker/7b76c32b55ce010d96d464271578fc9af051f72d86116b83b9248c6de7e6901c	714	osparc	{}	16.911634884	2023-07-11 17:37:00+00	2023-07-11 17:
	/docker/ed9664e91d2b3ba0f828afaad56378b94d88eb454c8d3d491c9630343329fe28	5	osparc	{}	16.568879121	2023-07-12 11:11:00+00	2023-07-12 11:

2. query only for specific dynamic sidecar/user services in case of need (this will be a complementary data source to the Application layer data source)

- *PROS*: Much fewer requests on Prometheus, much fewer data stored in Postgres, and can be queried only for specific services in case of need (therefore should not be a problem with pagination).

- *CONS*: Data are not stored in our managed database, but we rely on Prometheus data which lives in the docker volume. (Maybe we can use AWS-managed Prometheus?)

Application layer (Director-v2, Director-v0)

As director-v2 is responsible for starting/stopping user services, we should store this information somewhere. Probably we need to deal with each of the following services differently:

- legacy dynamic services (director-v0?)
- dynamic services (director-v2)
- computation services (director-v2 or comp_runs DB table)

Pseudo Code of Resource Usage Tracker

Here is a pseudocode of how we can deal with tracking of running user services and updating user credits/tokens left.

```
sessions: list = [ ]

# When resource tracker gets information that some service starts we add a new “session” to the list of sessions
sessions.add("s4l-id-123")

# When resource tracker gets information that the user stopped the service we remove the session
sessions.remove("s4l-id-123")
update_credits("s4l-id-123") # This will trigger function that will compute how much credits were used and updat

# As we want sometimes to update credits of long running services, we need to trigger regulary compute_credits f
# periodically sheduled task (ex. each 1 hour)
for session in sessions:
  update_credits(session)
```

Start/Stop events

There are 2 options for how to work with service start/stop events

1. Publish events from director-v2 to the resource usage tracker
   a. *PROS*: Fewer changes to director-v2
   b. *CONS*: The event can be lost on the way.
2. Create dyn_runs table in director-v2 (similarly to comp_runs for computational services)
   a. *PROS*: We store the history of all running user services with their current state (as starting/stopping user services is the responsibility of director-v2 he can save the start event reliably through transaction).
   b. *CONS*: New table in DB which will grow over time (MD: we really don’t need to worry about this).

1. Publish events from director-v2 to the resource usage tracker

The following table shows all situations when the user starts/stops service and whether the event is received by the Resource usage tracker service.

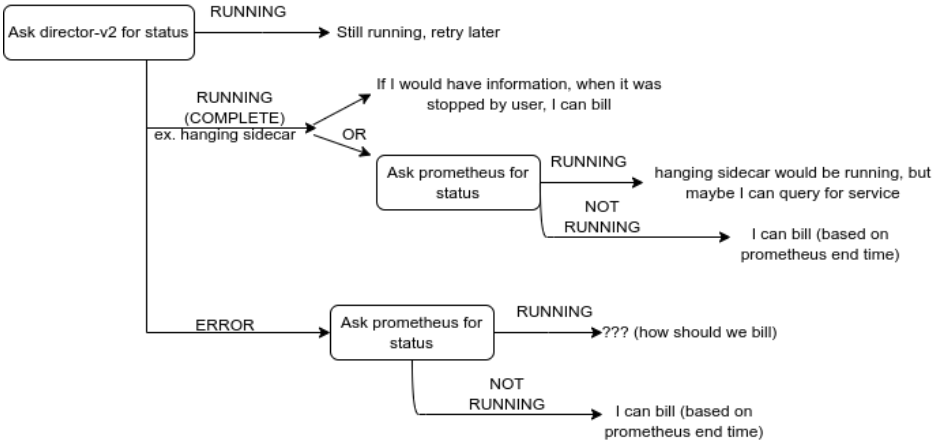
Situation	Start	Stop	Comment
A	✅	✅	This is good, we are able to reliably bill.
B	✅	❌	
C	❌	✅	
D	❌	❌	

(A) situation:

- this is good, we are able to reliably bill (I guess we even do not need to double-check with the Prometheus data source)

(B) situation:

- I get the information that the user started a service, therefore I can add a new session to the sessions list.
- The problem is how to determine whether the service is still running, or some problem occurred. Also, how should we bill in this situation?
- We can periodically ask director v2 for a status (as currently, director-v2 has only information about the currently running service, he is able to provide us running status, but most probably we will get an error if the service is not running anymore)



(C) & (D) situation:

The problem here is that we didn't receive the event about a user starting a service. We need to get the information from somewhere else (when we get it, then the problem is reduced to situations A & B). These are some options that come to my mind:

1. Scans periodically Prometheus for all services and when it finds a service that the resource usage tracker does not have info about, it will add it to the sessions list.
   a. *PROS*:
   b. *CONS*: We need to query Prometheus for all of the containers regularly which might be challenging.
2. Ask periodically director-v2 for a list of currently running user services, when the resource usage tracker finds out that there was a service that he didn't know about, it will add it to the sessions list.
   a. *PROS*:
   b. *CONS*: director-v2 can provide us only with the current state of the resources, he doesn't store the history.
3. Director-v2 will periodically stream events to the resource-usage-tracker about running services.
   a. *PROS*:
   b. *CONS*: A lot of data will be streamed periodically, and some of them might be potentially lost on the way.

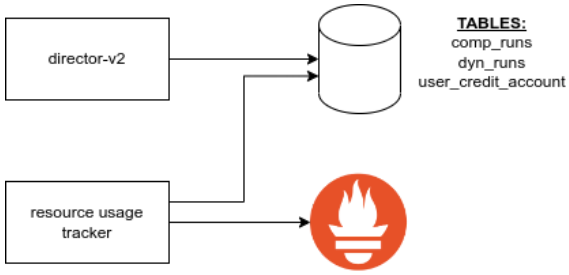
2. Create dyn_runs table in director-v2 (similarly to comp_runs for computational services)

This is a different approach, where we will introduce dyn_runs table in the director-v2. By this, we can ensure in a transaction, that when we start a user service the “run_id“ is stored in the database. Basically, each time somebody runs a dynamic service, a new row in the dyn_runs table will be created. Secondly, director-v2 will update the status of the service in the table.

Basically, we will reduce the previous table just for the first two situations:

Situation	Start	Stop	Comment
A	✅	✅	This is good, I am able to reliably bill.
B	✅	❌	

Resource usage tracker “sessions“ list will be obtained by the comp_runs & dyn_runs tables (or we can create one combined table, the DB model example can be found in the *database model section* below). Regarding updating user credits, the resource usage tracker will only deal with the (A) and (B) situations.



Regular flow in resource usage tracker (PSEUDO CODE):

```
# Get all not billed services from DB
sessions = get_not_billed_services()

for session in sessions:
  # Check in DB, whether director-v2 updated status to COMPLETE
  if session['status'] = "COMPLETE":
    # I can bill
    update_credits(session)
    mark_as_billed(session)
  else:
    # Either it is still running, or director-v2 might have problem and didnt update status correctly
    # Therefore we will double check with prometheus
    prometheus_status = check_user_service_prometheus_status(session)
    if prometheus_status = "RUNNING":
      # User service is still running, I will update credits
      update_credits(session)
    else:
      # So probably was problem in director-v2
      # I can bill based on Prometheus last running timestamp + update status to complete
      update_status(session, "COMPLETE")
      update_credits(session)
      mark_as_billed(session)


```

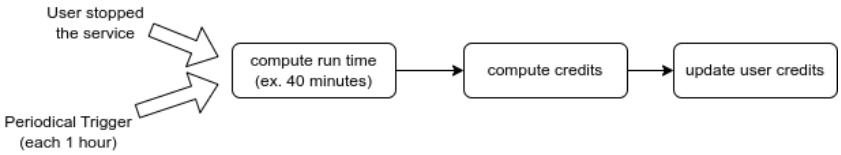
- *PROS*:
  - We have a good source of truth when the user starts a service.
  - If resource-usage-tracker was down we can still compute services that have run because we have a history in the dyn_runs table.
  - We need to deal with much less corner cases.
  - We can provide this run_id to the container label, so we can easily combine it with the second data source.
  - It might also help us during debugging of strange situations
- *CONS*:
  - some work on director-v2

Questions/Notes:

- *Optimization*: The resource usage tracker can listen to director-v2 stop service events, which would trigger the update_credits() right away in case of a user stopping the service.
- Maybe director-v2 can import the data into a “new database“, the same holds for computational services (@Sylvain maybe we do not need to refactor the comp_runs table but we can create a new one in the same structure for both comp & dyn services)

Credits/Tokens

This is a diagram of what needs to happen when the user stopped the service. (Also this needs to be triggered periodically so we can update the user account for long-running services - this will have some delay similarity to AWS Cost Explorer)



1. We need to compute how long the service was running
2. We need to compute how many credits/tokens the service consumed
   a. ex. service with 4 CPUs, small aws instance type run for 40 minutes:

AWS instance pricing mapping table

instance	credits	product
small	5	sim4life
medium	10	sim4life
big	15	sim4life

CPU hours mapping table

CPUh	credits	product
1	4	sim4life

Credit used:

0.66 * (5 + 4 * 4) = 13.86

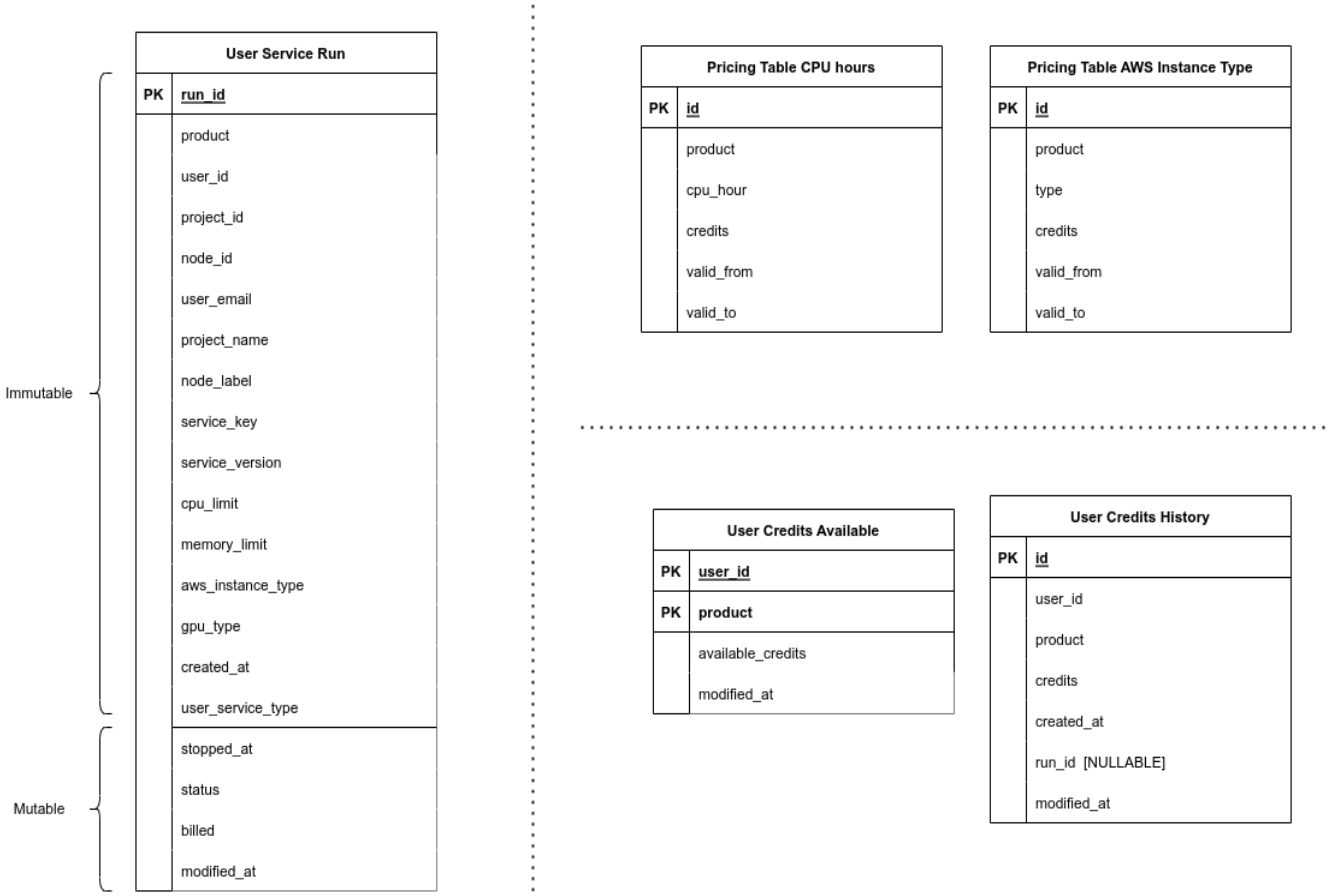
3. We need to update the user credits table
   a. ex. 100-13.86 = only 86.14 credits left

Questions/Notes:

- How we will deal with situations when the user is reaching 0 credits?
  - should we probably ask before each start of the service whether users have enough buffer to start the service? how we will estimate what is enough?
  - Based on Niels's input we should inform the user when they have only a small amount of credits left.
  - What happens when the user reaches 0 credits? Are we able to stop the service? Will they have access to the data of the service?

Database model

Here is a proposal for the data model, so we can start some discussions:



1. User Service Run table

- In this proposal, we have one table and whenever we run a user service (computational or dynamic one) we will create a unique **run_id** and populate it with all needed information. Most of the columns are immutable and we store them so we can have their history (even when the user deletes the project/node or even when the user is deleted, the immutable part of this table stays untouched).
- The mutable part of this table is modified when the user service is either stopped or analyzed by regular triggers for long-running services.

NOTE: run_id can be passed as a container label, then it can be easily queried in Prometheus if needed.

2. Pricing tables

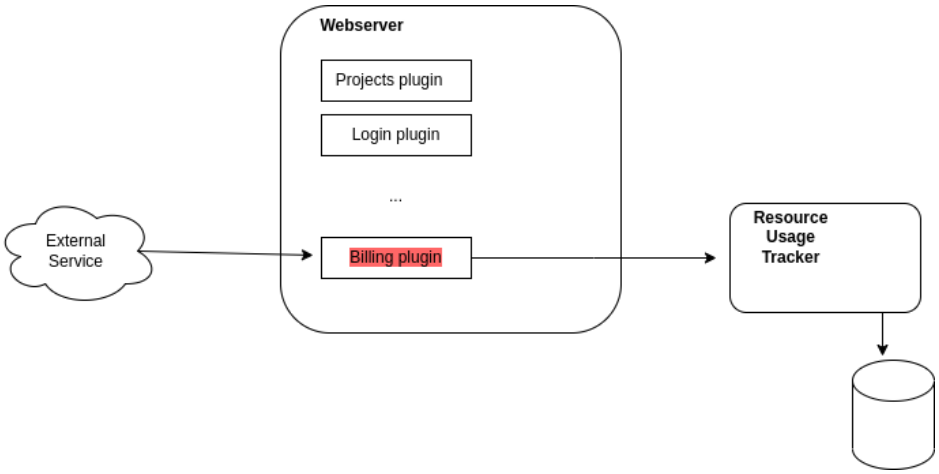
- We need to have mapping tables and also their validity.
- We can provide an internal endpoint to update the price.
- Regarding AWS instances, in the beginning, we will want to have three options: small, medium, and big machines.

3. User Credits tables

- Here are probably more options on how to design this, this proposal takes into consideration two tables (One is to have the whole history, also good when we need to debug something, the second one is to have quick access to the user's current amount of available credits):
  - **User Credits History** should be the whole history of adding or using credits, meaning when the user adds 1000 credits it should occur there, but also when we compute some finished user service it should occur there with a negative sign (that is the reason why run_id is there for reference). When we SUM everything we get the current amount of available credits.
  - **User Credits Available** should be the aggregated always up-to-date table with currently available credits. The reason for this is that we do not need to compute always his whole history to get this number, but rather compute it incrementally.

External service integration

We will need to integrate with 3rd party service that will provide us with the number of credits, which the user purchases, which need to be stored in the User Credits History table.



Conclusion

We discussed the pros and cons and decided we will go with the “DIRECTOR-v2 → RabbitMQ → Resource Usage Tracker” approach, meaning the new table with run_id will be created in the resource-usage-tracker service which will listen to the events published by director-v2 and dy-sidecars.

Other notes:

- It is sufficient to have a general Credit Mapping Table that can be used for different use cases
- Yet another alternative to where the resource usage tracker can get the data from was briefly discussed. We can always look just for container labels. Potencionaly they can be updated during runtime. → Still we decided we will go with the RabbitMQ approach.

Next Steps: