Cloudwalk Software Engineer Technical Assignment

Candidate: Juliano Fischer Naves

February 7, 2022

1 The Problem

Your challenge is to **answer InfinitePay's clients automatically in the chat in a proper way to solve the client problem**.

You have access to systems in the form of APIs in which you can check client's problems and the needed information to solve them.

Check what chats are opened/active, and, for each chat, check what is the client's complaints. Using the APIs and database tables that are available, show us how you would build a software to automatically answer the client in a proper way.

Consider that instead of a SQLite Database (it's attached to this email), it's a PostgreSQL deployed in a GCP Cloud SQL. The APIs you have access were deployed on App Engine.

Remember that the solution needs to be **scalable in case both the number of chats and the number of potential problems a customer may have increase (in which case it should be easy to patch the automatic solution to a frictionless new problem)**.

If you have any doubt, get in touch by sending an email to ‘ana.tamais@cloudwalk.io‘. **Good questions will be counted positively for this hiring process**.

Lastly, **we already solved this problem internally** and we don’t want you to think that you are working for free. If your solution is better than ours, (and you have a cultural fit with us) we will try to hire you.

1.1 APIs you have access to

You have access to 3 APIs: - The first one is a **logistics** API (think of this API as a FEDEX API) - ‘https://logistics-api-dot-active-thunder-329100.rj.r.appspot.com‘. In this API, you will have 2 endpoints:

[**POST**] **/tracking**: This endpoint is useful to **track a delivery**. You give a body as input in the following format:

```
{"id_sale": "123456"}
```

And receives as a response:

```
{"id": "123456",
"status": "On delivery route",
"delivery_forecast": "01/12/2021",
"destination_zip_code": "31160550"}
```

In practice, this endpoint is useful to track the delivery of InfinitePay’s credit card machines.

[**POST**] **/zip_code**: This endpoint is useful to, given your ZIP code, give the complete address as the response. You should give a body as input in the following format:

```
{"zip_code": "31160550"}
```

And receives as a response:

```
{
  "neighborhood": "Palmares",
  "ZIP_code": "31160-550",
  "city": "Belo Horizonte",
  "complement": "",
  "street": "Rua Professor Patrocínio Filho",
  "state": "MG"
}
```

Remember to add a header called ‘authorization‘ with the value ‘teste‘

The second one is a **telecommunications** API (think of this API as an API of a company that sells chips to be used on the credit card machines) - ‘https://telecom-api-dot-active-thunder-329100.rj.r.appspot.com‘. In this API, you will find 1 endpoint:

[**POST**] **/chip_status**: This endpoint is used to check the **chip status**. You give as input the chip id in the following format:

```
{"chip_id": "CHIP37648"}
```

And receives as a response:

```
{
  "id": "CHIP37648",
  "status": "active",
  "description": "OK"
}
```

The possible status are: ‘active‘, ‘inactive‘ and ‘without connection‘. In the description key we have the reason of the chip problem, if the status is ‘inactive‘ or ‘without connection‘.

Remember to add a header called ‘authorization‘ with the value ‘teste‘.

The third API is the **conversations** API. When an InfinitePay client have any problem with one of our solutions, they can get in touch with us using our chat. This API enables that you collect conversations from our chat - ‘https://chats-api-dot-active-thunder-329100.rj.r.appspot.com‘. In this API, we have 2 endpoints:

[**POST**] **/conversations**: This endpoint returns a list of conversation ids, whose conversations are happening in the same time as the request, it means, a list of the active conversation ids.

You don’t need to give a body as input, you will receive as response:

```
{"conversation_ids": [754893, 754894, 754895, 754896, 754897]}
```

[**POST**] **/conversation_info**: This endpoint returns **conversation information** given a conversation_id. You should give a body in this format:

```
{"conversation_id": "754893"}
```

And receives as a response:

```
{
  "conversation_id": "754893",
  "messages": [{
           "message_id": "754893-0",
           "text": "hello, my money hasn't landed in my bank
               account yet. I want to know what happened",
           "created_at": "1640197668"}
```

```
    ],
 "merchant_id": "558392",
 "subject": "bank account receipt problem"
}
```

The field ‘subject‘ is generated by an AI that classifies automatically the conversation subject. The field ‘merchant_id‘ is the shopkeeper id (InfinitePay’s client id).

[**POST**] **/send_message**: This endpoint is used to **send a message** in a conversation. You should give a body in this format:

```
{"conversation_id": "754893",
 "message": "test, test"}
```

And receives as a response:

Remember to add a header called ‘authorization‘ with the value ‘teste‘.

Besides the 3 APIs, you also have access to a SQLite Database (it’s attached to this email), in which you will find 3 tables:

sales: This table shows the sales data for credit card machines. This sales are from InfinitePay to its clients. As column, we do have:

- id_sale: InfinitePay’s sales unique id - it’s a sale of credit card machine made by InfinitePay
- merchant_id: shopkeeper unique id (InfinitePay’s client)
- chip_id: credit card machine’s chip unique id
- created_at: datetime from time of purchase of credit card machine
- status: sale status

- description: status details

transactions: This table shows transaction data which went through InfinitePay's credit card machines (for example: if a supermarket, which is a InfinitePay customer, passes a transaction to sell a chocolate to the final customer, this transaction will be in the transactions table)

- transaction_id: unique id of the transaction that passed on an InfinitePay customer's credit card machine.
- merchant_id: shopkeeper unique id who passed that transaction
- created_at: transaction datetime
- value: transaction value

receipt: This table shows value transfer data to an InfinitePay customer's bank account arising from transactions made the day before. (Context: When a market, for example, makes a transaction of R$50,00 in credit, the market will only receive in its bank account the money referring to this sale the following day).

- merchant_id: shopkeeper unique id
- created_at: datetime InfinitePay tried to transfer the money to the merchant's account
- status: bank transfer status
- description: status details
- value: value that should be transferred to the shopkeeper. This value must be equal to the sum of transaction's values from the previous day

2 Implementation

2.1 Architecture

Figure 2.3 shows the proposed architecture to solve the problem. The solution is comprised of 3 main components.

- **dispatcher**: is the program responsible for query the *conversations service* for ongoing conversations and queue these conversations in the message broker.
- **message broker**: the message broker is responsible for fast and reliable communication among the project components. For this proof-of-concept, RabbitMQ was used. The possible advantages of RabbitMQ will be presented later.
- **worker**: is the program responsible for dequeueing messages from the message broker and run suitable solutions for each message.

2.2 Horizontal Scaling

Regarding Horizontal Scaling, it can be achieved in several ways. Following, are some suggestions.

- multithreading: the use of multithreading allows the creation of multiple threads within a process. Thus, we can have several workers running within the same process. However, *pika*, the library used as the interface to RabbitMQ is not thread-safe. Furthermore, Python threads don't run simultaneously due to the way they are implemented (Global Interpreter Lock). The side effect is that multithreading may

not speed up all programs. It are usually most suitable for IO-bound tasks.

- RabbitMQ Multiple Queues: RabbitMQ queues are single-threaded. We can achieve a best performance in multi-core systems with multiple queues (and consumers).

2.3 Vertical Scaling

Vertical Scaling can be achieved by using multiple workers. We can multiple workers in a multi-core system or leverage virtualization capabilities in order to run multiple workers, for instance, using Docker. It also can be achieved by scaling RabbitMQ through clusters.

By using virtualization arises the possibility of using Kubernetes as containers orchestration technology in order to automate dimensioning, implementation and management of containers.

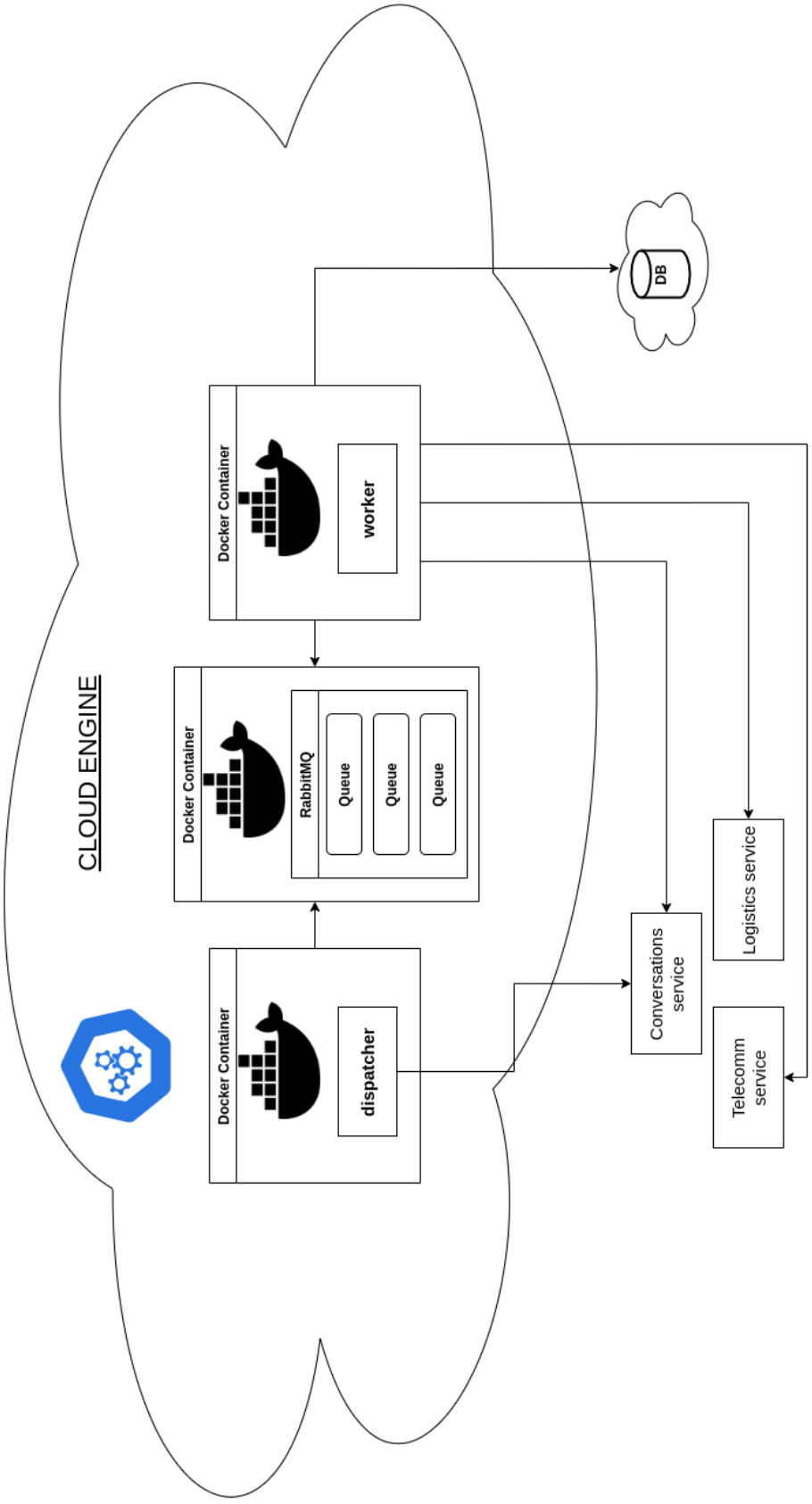

Figure 1: System Architecture

3 Possible Enhancements

Considering the proposed exercise there is a possible bottleneck. There is the need for the dispatcher to check the conversations service for ongoing conversations. How often should the dispatcher check the message service? It is not easy to answer this question. Waiting too long will increase the response time. Waiting too little can result in a *dispatcher* processing bottleneck, especially if we have a lot of conversations queued. In this case, we need to consider that for every conversation the dispatcher will need to check if the conversation was not already queued in RabbitMQ.

Another problem is that if a new message is posted to an ongoing conversation, considering the new messages may result in new AI classification for the conversation or that every new message leads to the need to re-processing the conversation (for a new solution), the dispatcher needs to keep state register, which is a scaling issue.

Thus, as a possible enhancement, the suggestion is the calling of the dispatcher by the conversation service. This way, the service can trigger the dispatcher every time the service itself becomes aware of a new conversation.