

Philip Quan

cs6650/assignment-03

<https://github.com/thephilipquan/cs6650/tree/main/assignment-03>

design.v1

In summary, my design flow and architecture is similar to assignment-02. My client spams requests to the server which interacts with an RDS, and returns a response.

Refer to the following for changes...

client

- Client request count has dropped from 1000 -> 100.
- **Removed** `getAlbum(albumId)`.
- **Added** `postReaction(albumId)`.

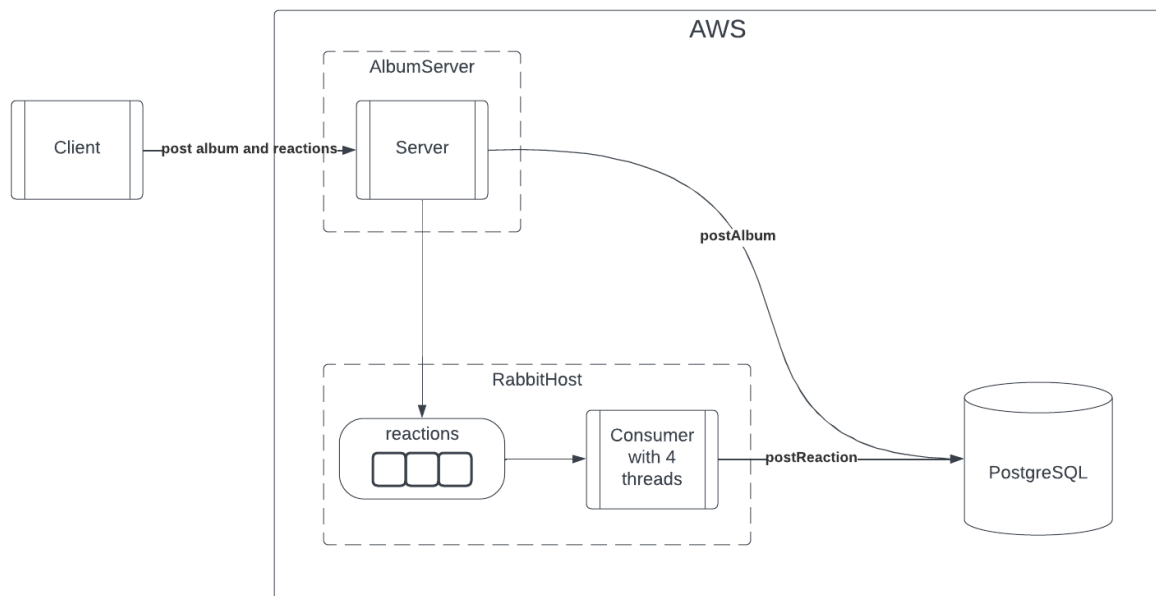
server

- **Added** `/reviews/*` endpoint.
- `/reviews/*` POST handler publishes to RabbitMQ on another EC2 instance.

rabbithost (new)

- Created a queue called `reactions`.
- Runs `consumer.jar` that spawns `x` threads to consume from `reactions` and `postReaction` to the RDS instance.

Below is the architecture as described.



design.v1.results

My throughput for assignment-02 assignment was around **2400/s** (see [assignment-02 report](#)).

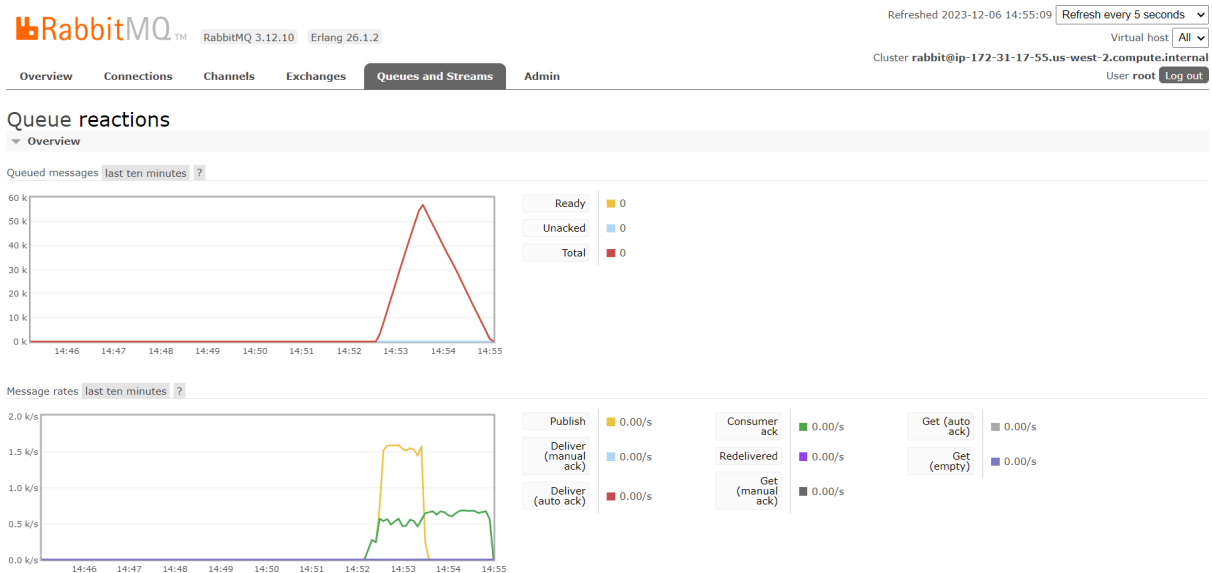
	walltime	throughput
groupSize=10	28.155 s	1563 r/s
groupSize=20	46.765 s	1796 r/s
groupSize=30	62.932 s	1970 r/s

s = seconds

r/s = requests per second

You can **view the file output for more details** at <https://github.com/thephilipquan/cs6650/tree/main/assignment-03/client/report>.

design.v1.queue.results



As you can see, the consumption rate was about $\frac{1}{3}$ of the publish rate. Naturally, I decided to increase the threads to something like **20**, but there was no change in the performance.

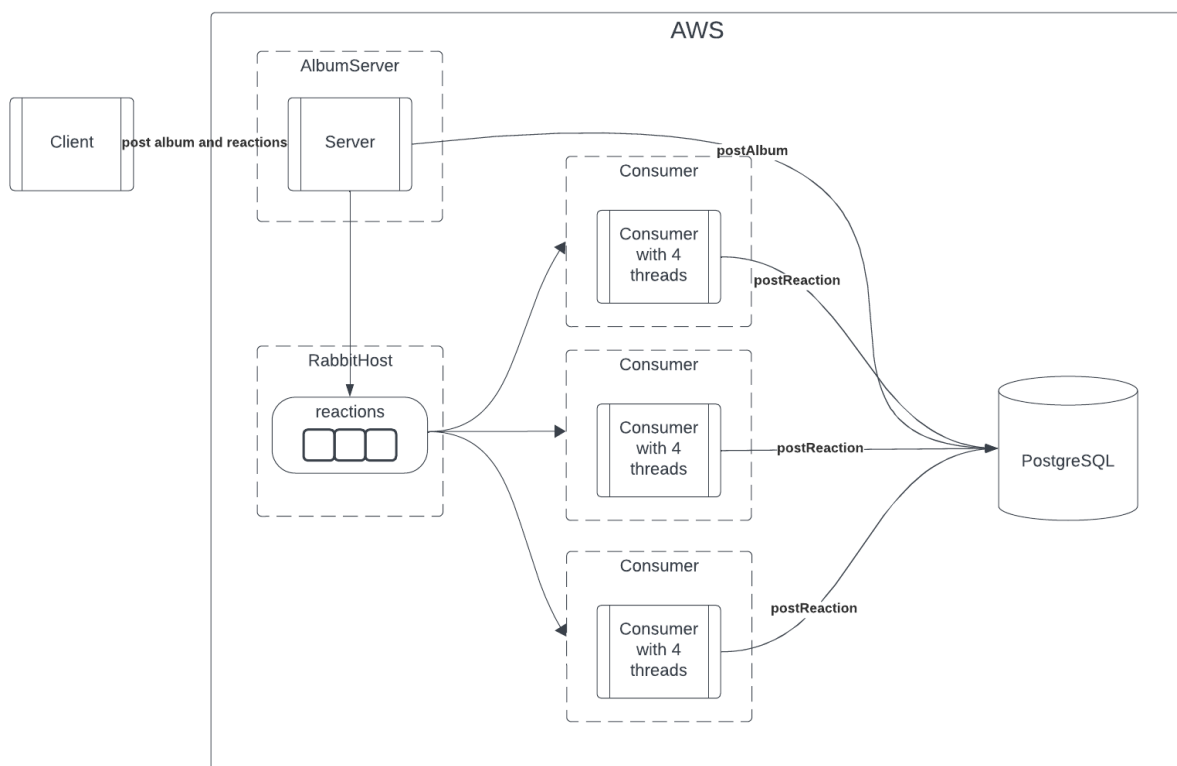
That's how I ended up with **design.v2**.

design.v2

I didn't know if the RabbitMQ service or the consumer threads were more taxing, and I didn't want to spend a lot of time testing every adjustment to find the perfect configuration.

Instead, I created 3 EC2 instances that each spawned 4 threads to consume from the queue that was hosted on another instance.

Refer below.



And with no other changes to the code, I received the results on the next page.

design.v2.results

The client side results are nearly unchanged, as it should be since design.v2 changes target the queue consumption rate.

	walltime	throughput
threadCount = 10	27.56 s	1597 r/s
threadCount = 20	46.55 s	1805 r/s
threadCount = 30	64.876 s	1911 r/s

s = seconds

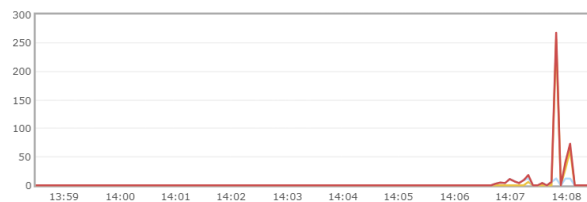
r/s = requests per second

design.v2.queue.results

Queue reactions

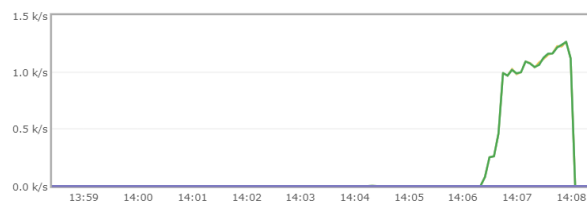
Overview

Queued messages last ten minutes ?



Ready 0
Unacked 0
Total 0

Message rates last ten minutes ?



Publish 0.00/s
Consumer ack 0.00/s
Get (auto ack) 0.00/s
Deliver (manual ack) 0.00/s
Redelivered 0.00/s
Get (empty) 0.00/s
Deliver (auto ack) 0.00/s
Get (manual ack) 0.00/s

As you can, the total queued messages caps out at just under **300**, a drastic decrease from [design.v1.queue.results](#) **60,000**.

But more importantly, **the publish and consume rate is nearly the same** throughout the lifetime of the program.

And to top it off (I don't have proof), the consumers finished just as the client finished.