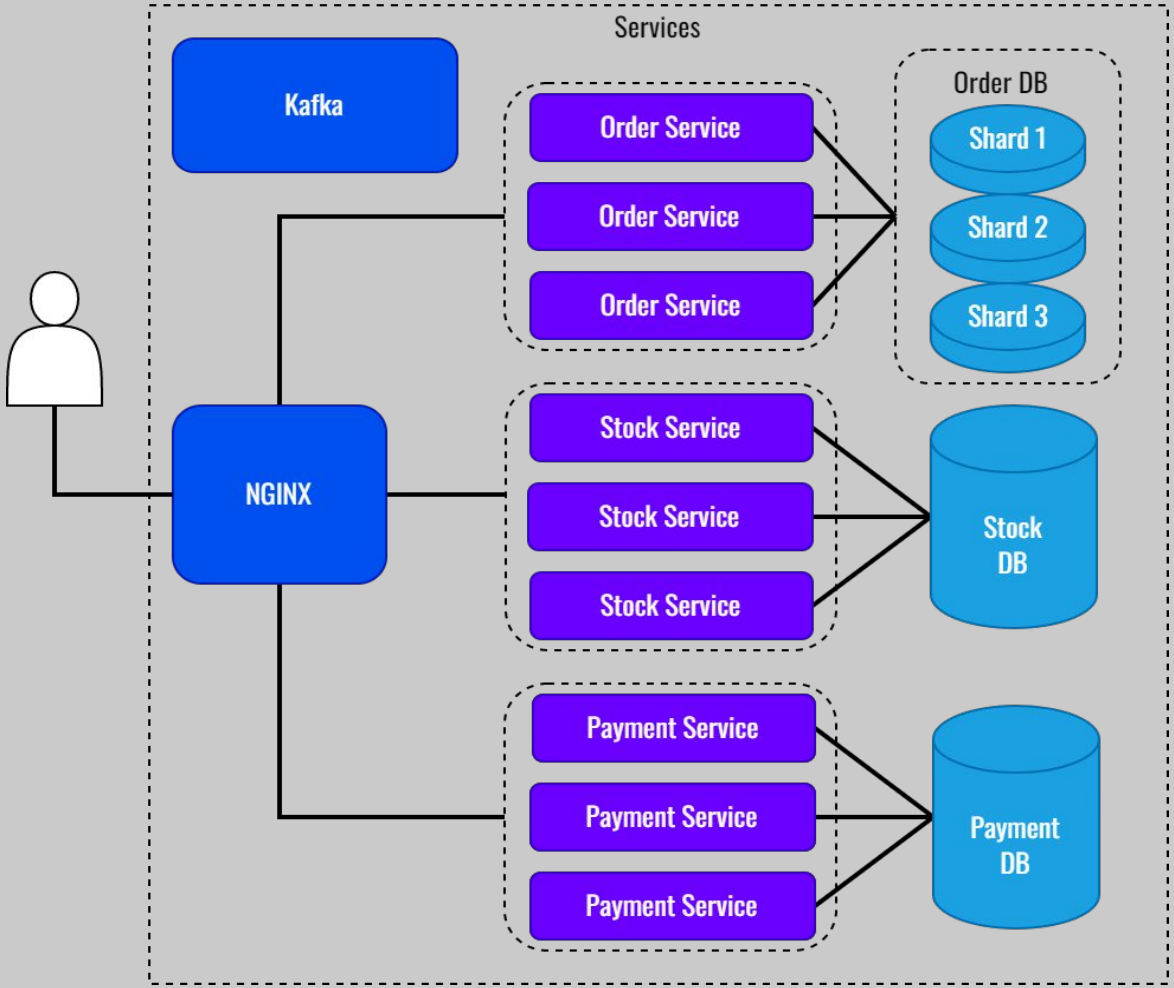
Web-Scale Data Management Project

...

Authors:
Alex Jeleniewski
Andrei Stefan
Paula Iacoban
Stef Kaptein
Tiago Gomes

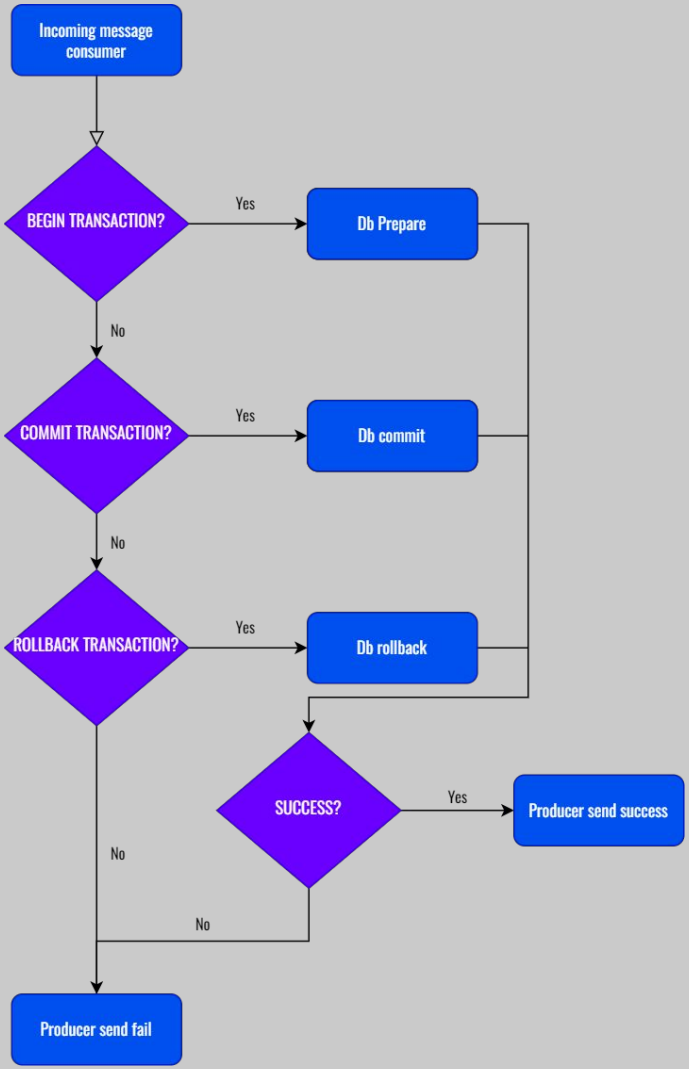
Design Architecture

- PostgreSQL
- Python Flask
- Kafka
- NGINX



Transaction Execution - 2PC

- Order service acts as coordinator -> Produce
- Stock and Payment consume instructions
- 3 instructions:
  - Prepare
  - Commit
  - Rollback
- 3 topics:
  - 1 for stock instructions
  - 1 for payment instructions
  - 1 for results

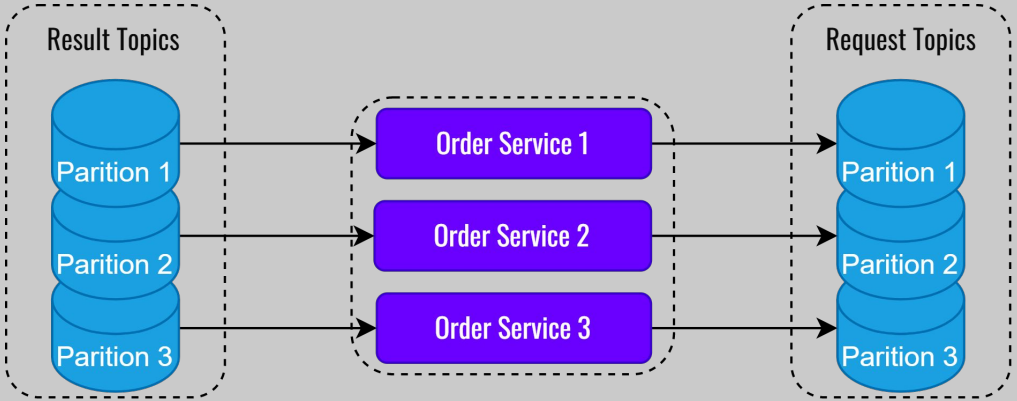


Transactional Design and Consistency Guarantees

- No illegal database state allowed
- 2 Phase Commit
- ACID Transactions
- Timeout detection

Scalability Methods

- Sharding of the Order database
- Multiple instances of all services
  - Actions can be performed by an arbitrary instance
  - Using partitions in Kafka
  - Example order service:



Fault-Tolerance Methods

- Two-Phase Commit
- Kafka’s native capabilities
- Shards
- Multiple servers/service

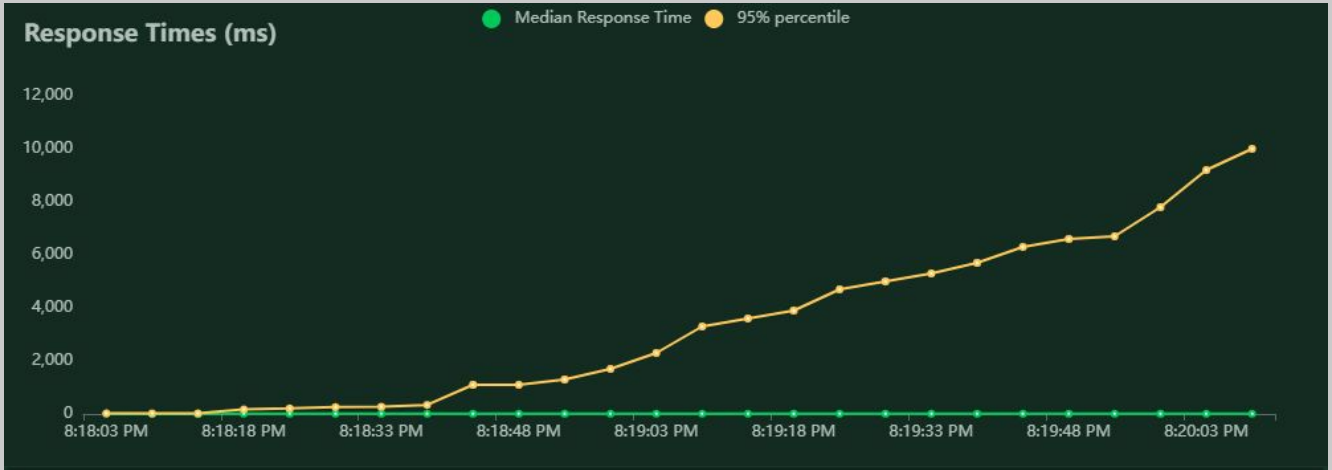
Results & Measurements

- Balances out at around 200 requests per second

Method	Name	# Requests	# Fails	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	RPS	Failures/s
POST	/orders/addItem/[order_id]/[item_id]	3697	0	3514	4	12343	104	27.7	0.0
POST	/orders/checkout/[order_id]	2149	0	3776	8	12375	63	16.1	0.0
POST	/orders/create/[user_id]	2703	0	3370	9	12324	142	20.2	0.0
DELETE	/orders/removeItem/[order_id]/[item_id]	517	0	3294	4	10773	113	3.9	0.0
POST	/payment/add_funds/[user_id]/[amount]	2527	0	7	3	140	87	18.9	0.0
POST	/payment/create_user	2960	0	6	3	96	51	22.1	0.0
POST	/stock/add/[item_id]/[number]	4863	0	7	4	112	65	36.4	0.0
GET	/stock/find/[item_id]	158	0	5	2	34	89	1.2	0.0
POST	/stock/item/create/[price]	4939	0	10	4	127	51	36.9	0.0
POST	/stock/subtract/[item_id]/[number]	158	0	5	4	11	70	1.2	0.0
	Aggregated	**24671**	**0**	**1299**	**2**	**12375**	**78**	**184.5**	**0.0**

Results & Measurements

- Response time keeps increasing with number of users, but just linearly
- Order is the slowest



Method	Name	50%ile (ms)	60%ile (ms)	70%ile (ms)	80%ile (ms)	90%ile (ms)	95%ile (ms)	99%ile (ms)	100%ile (ms)
POST	/orders/addItem/[order_id]/[item_id]	3100	4500	5300	6200	7200	8600	10000	12000
POST	/orders/checkout/[order_id]	3500	4700	5500	6500	7600	9200	11000	12000
POST	/orders/create/[user_id]	2800	4400	5100	6100	6900	7900	10000	12000
DELETE	/orders/removeItem/[order_id]/[item_id]	2600	4300	5200	6100	7300	8300	10000	11000
POST	/payment/add_funds/[user_id]/[amount]	6	7	7	8	10	12	18	140
POST	/payment/create_user	6	7	7	8	9	11	17	97
POST	/stock/add/[item_id]/[number]	6	7	7	8	10	11	20	110
GET	/stock/find/[item_id]	5	5	5	6	9	13	32	35
POST	/stock/item/create/[price]	7	8	10	16	22	27	35	130
POST	/stock/subtract/[item_id]/[number]	5	6	6	7	8	10	11	12
	Aggregated	**8**	**14**	**340**	**2500**	**5500**	**6800**	**9800**	**12000**

Results & Measurements

- Consistent state in the database after each transaction, even if users went up to 1000

Limitations and future work

Limitations

- Tried sharding Payment, but didn't improve performance
- Sharding Stock would need distributed transactions
    - And might not improve performance

Future work

- Dynamic creation of shards (instead of fixed number)
- Dynamic scaling for the number of instances of services
- Sharding Payment and Stock for the cases where the application is under heavy load