Cons

- Maintaining data consistency between the database and cache is hard. We need to think carefully about how this inconsistency affects user experience.

Data consistency among services

In a traditional monolithic architecture [11], a shared relational database is used to ensure data consistency. In our microservice design, we chose a hybrid approach by having Reservation Service handle both reservation and inventory APIs so that the inventory and reservation database tables are stored in the same relational database. As explained in the "Concurrency issues" section on page 206, this arrangement allows us to leverage the ACID properties of the relational database to elegantly handle many concurrency issues that arise during the reservation flow.

However, if your interviewer is a microservice purist, they might challenge this hybrid approach. In their mind, for a microservice architecture, each microservice has its own databases as shown on the right in Figure 7.17.

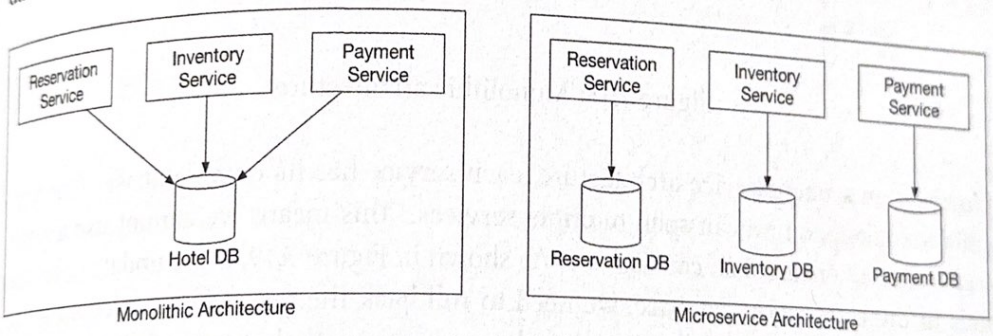


Figure 7.17: Monolithic vs microservice

This pure design introduces many data consistency issues. Since this is the first time we cover microservices, let's explain how and why it happens. To make it easier to understand, only two services are used in this discussion. In the real world, there could be hundreds of microservices within a company. In a monolithic architecture, as shown in Figure 7.18, different operations can be wrapped in a single transaction to ensure ACID properties.

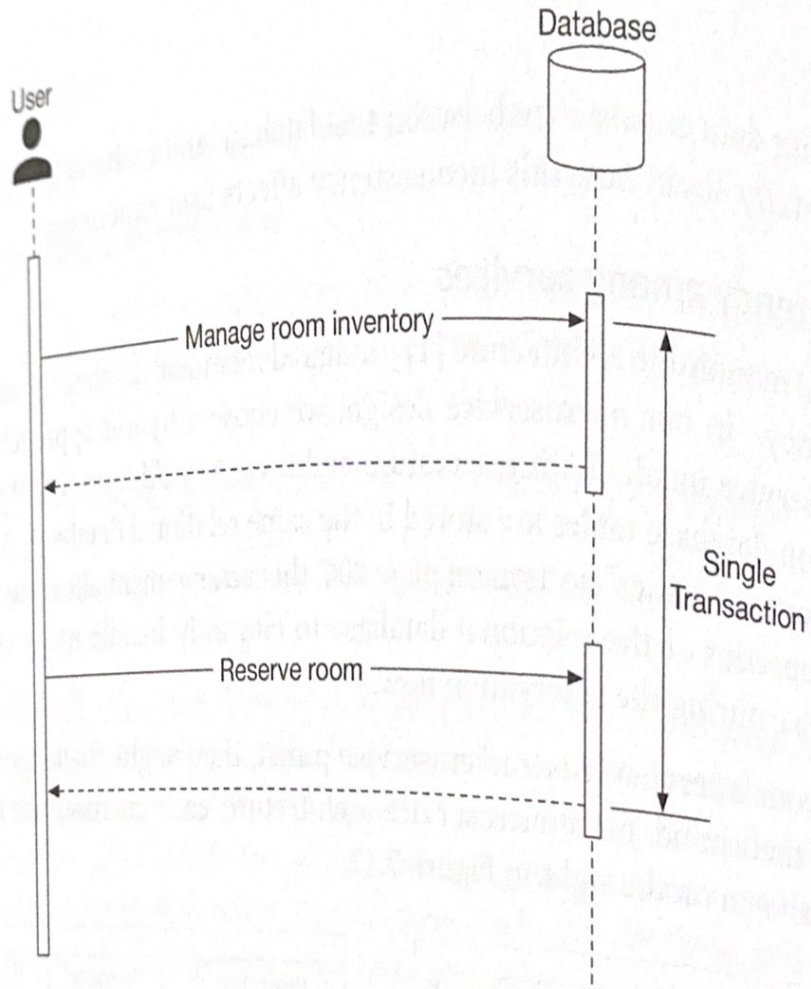
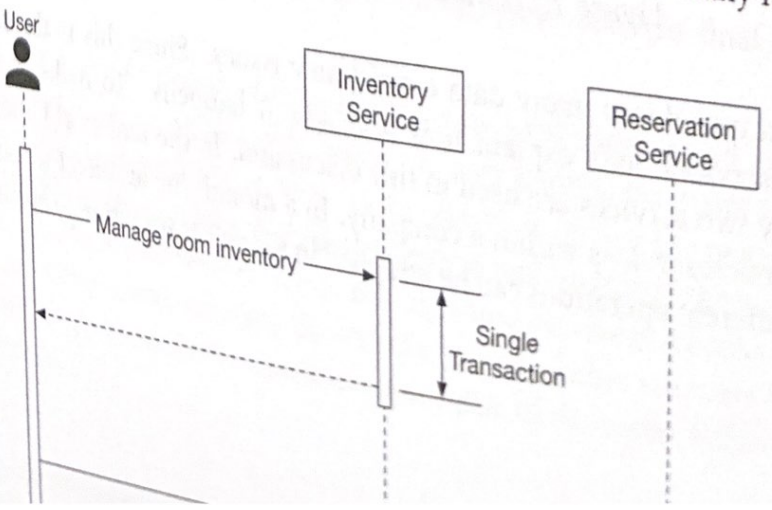
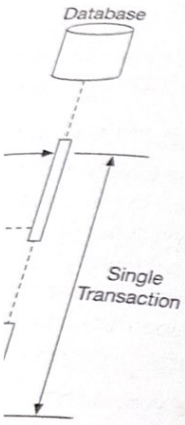


Figure 7.18: Monolithic architecture

However, in a microservice architecture, each service has its own database. One logically atomic operation can span multiple services. This means we cannot use a single transaction to ensure data consistency. As shown in Figure 7.19, if the update operation fails in the reservation database, we need to roll back the reserved room count in the inventory database. Generally, there is only one happy path, but many failure cases that could cause data inconsistency.





database. One logi-
cannot use a single
e update operation
oom count in the
failure cases that

niques. If you want to read the details, please refer to the reference materials.

- Two-phase commit (2PC) [12]. 2PC is a database protocol used to guarantee atomic transaction commit across multiple nodes, i.e., either all nodes succeeded or all nodes failed. Because 2PC is a blocking protocol, a single node failure blocks the progress until the node has recovered. It's not performant.
- Saga. A Saga is a sequence of local transactions. Each transaction updates and publishes a message to trigger the next transaction step. If a step fails, the saga executes compensating transactions to undo the changes that were made by preceding transactions [13]. 2PC works as a single commit to perform ACID transactions while Saga consists of multiple steps and relies on eventual consistency.

It is worth noting that addressing data inconsistency between microservices requires some complicated mechanisms that greatly increase the complexity of the overall design. It is up to you as an architect to decide if the added complexity is worth it. For this problem, we decided that it was not worth it and so went with the more pragmatic approach of storing reservation and inventory data under the same relational database.

Step 4 - Wrap Up

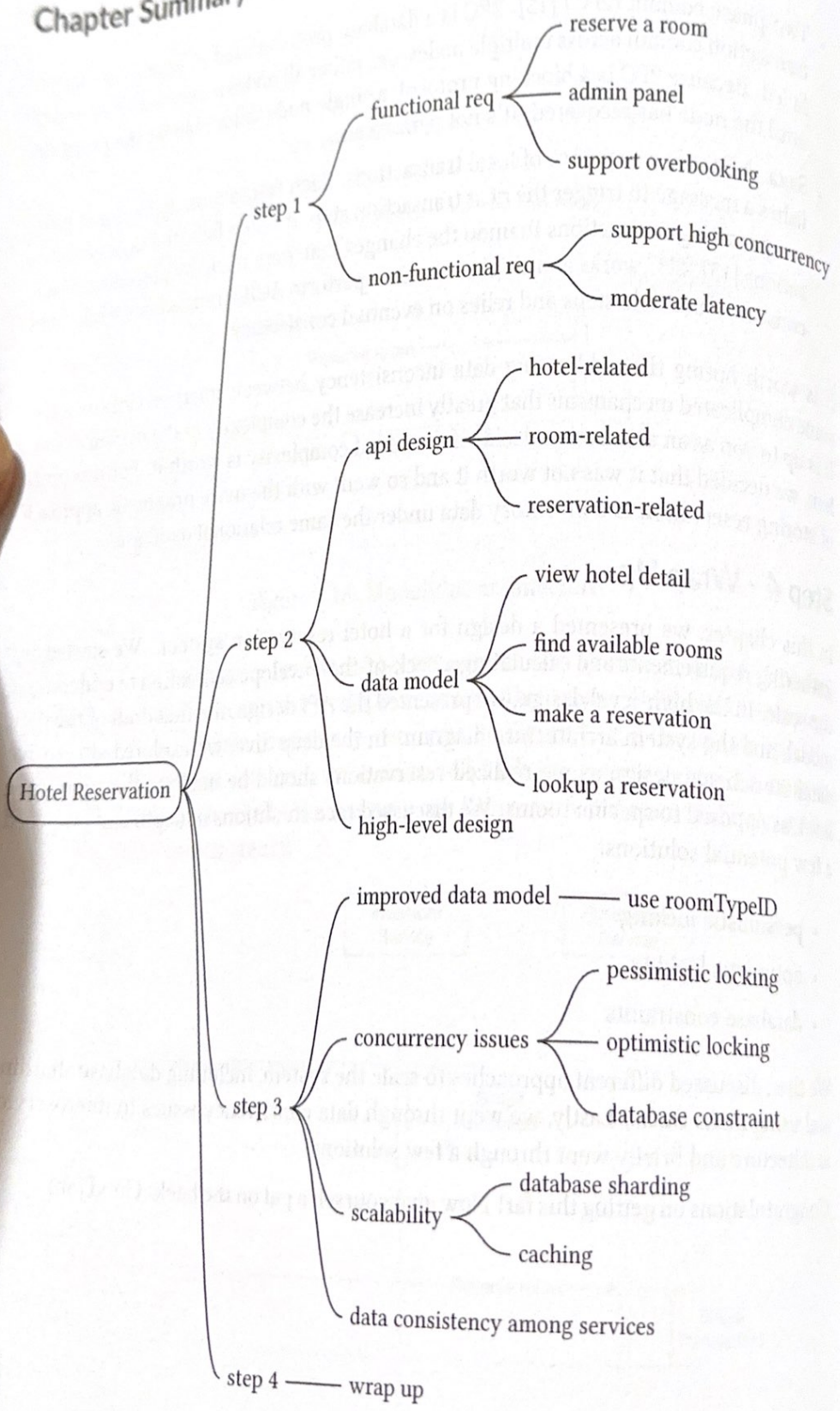
In this chapter, we presented a design for a hotel reservation system. We started by gathering requirements and calculating a back-of-the-envelope estimation to understand the scale. In the high-level design, we presented the API design, the first draft of the data model, and the system architecture diagram. In the deep dive, we explored alternative database schema design as we realized reservations should be made at the room type level, as opposed to specific rooms. We discussed race conditions in depth and proposed a few potential solutions:

- pessimistic locking
- optimistic locking
- database constraints

We then discussed different approaches to scale the system, including database sharding and using Redis cache. Lastly, we went through data consistency issues in microservice architecture and briefly went through a few solutions.

Congratulations on getting this far! Now give yourself a pat on the back. Good job!

Chapter Summary



Reference Material

[1] What Are The Benefits of Microservices Architecture? https://www.appdynamics.com/topics/benefits-of-microservices.

[2] Microservices. https://en.wikipedia.org/wiki/Microservices.

[3] gRPC. https://www.grpc.io/docs/what-is-grpc/introduction/.

[4] Booking.com iOS app.

[5] Serializability. https://en.wikipedia.org/wiki/Serializability.

[6] Optimistic and pessimistic record locking. https://ibm.co/3Eb293O.

[7] Optimistic concurrency control. https://en.wikipedia.org/wiki/Optimistic_concurrency_control.

[8] Change data capture. https://docs.oracle.com/cd/B10500_01/server.920/a96520/cdc.htm.

[9] Debizium. https://debezium.io/.

[10] Redis sink. https://bit.ly/3r3AEUD.

[11] Monolithic Architecture. https://microservices.io/patterns/monolithic.html.

[12] Two-phase commit protocol. https://en.wikipedia.org/wiki/Two-phase_commit_protocol.

[13] Saga. https://microservices.io/patterns/data/saga.html.