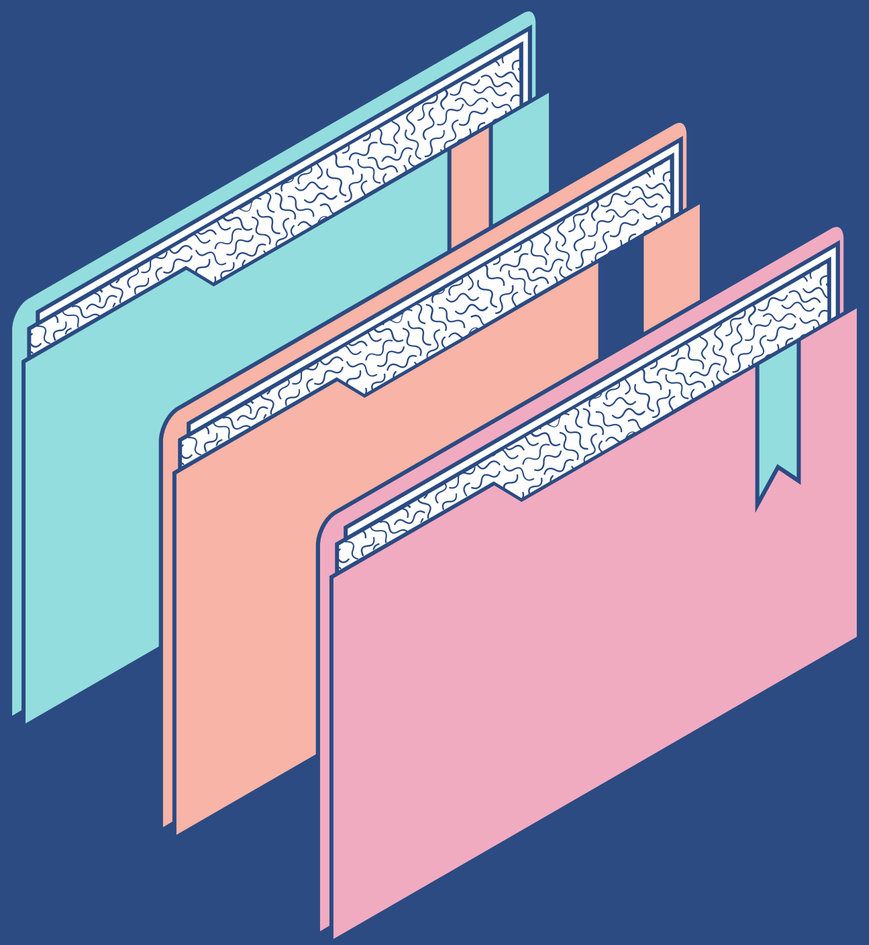
# PeerPrep

CS3219 Software Engineering Principles and Patterns

G35



# Agenda

- SDLC
- Requirements
- Architecture
- Individual Services
- Demo
- Design Considerations
- CD
- QnA

# Software Development Life Cycle

1 — 2 — 3

**STEP**

**Software development process**

Agile model with weekly sprints.

**STEP**

**User stories, FRs, NFRs**

Prioritized and grouped.

**STEP**

**Development**

Implement based on services.
Peer review.

## Functional Requirements

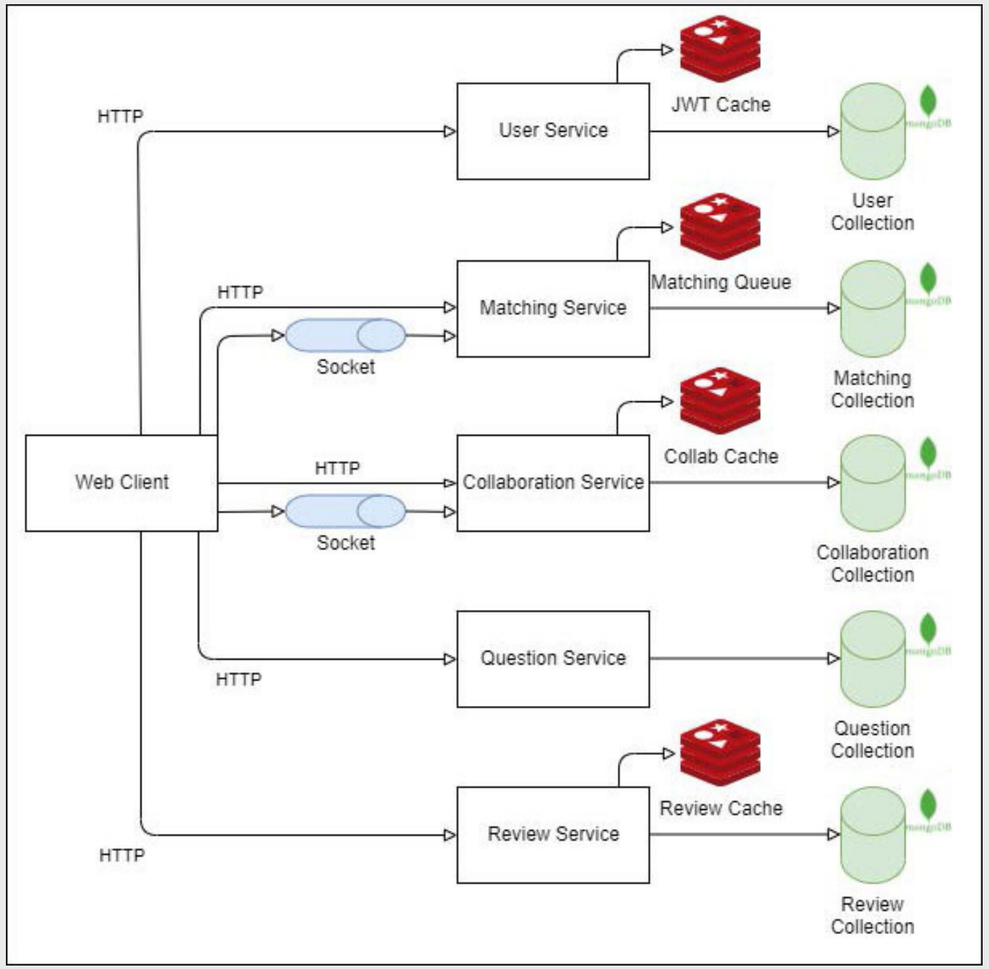
- PeerPrep should allow users to manage their accounts.
- PeerPrep should allow users to find matches based on difficulty.
- PeerPrep should allow users to collaborate.
- PeerPrep should allow users to work on a question.
- PeerPrep should allow users to rate their peers.

## Non Functional Requirements

- Availability: PeerPrep should deal with faults and have high uptime.
- Persistency: PeerPrep should save changes to minimize data loss.
- Reliability: PeerPrep should recover from unexpected inputs and not crash.

# Architecture Diagram

DigitalOcean




## Microservices deployed on DigitalOcean

- User Service
- Matching Service
- Collaboration Service
- Question Service
- Review Service

## Interaction between frontend and microservices

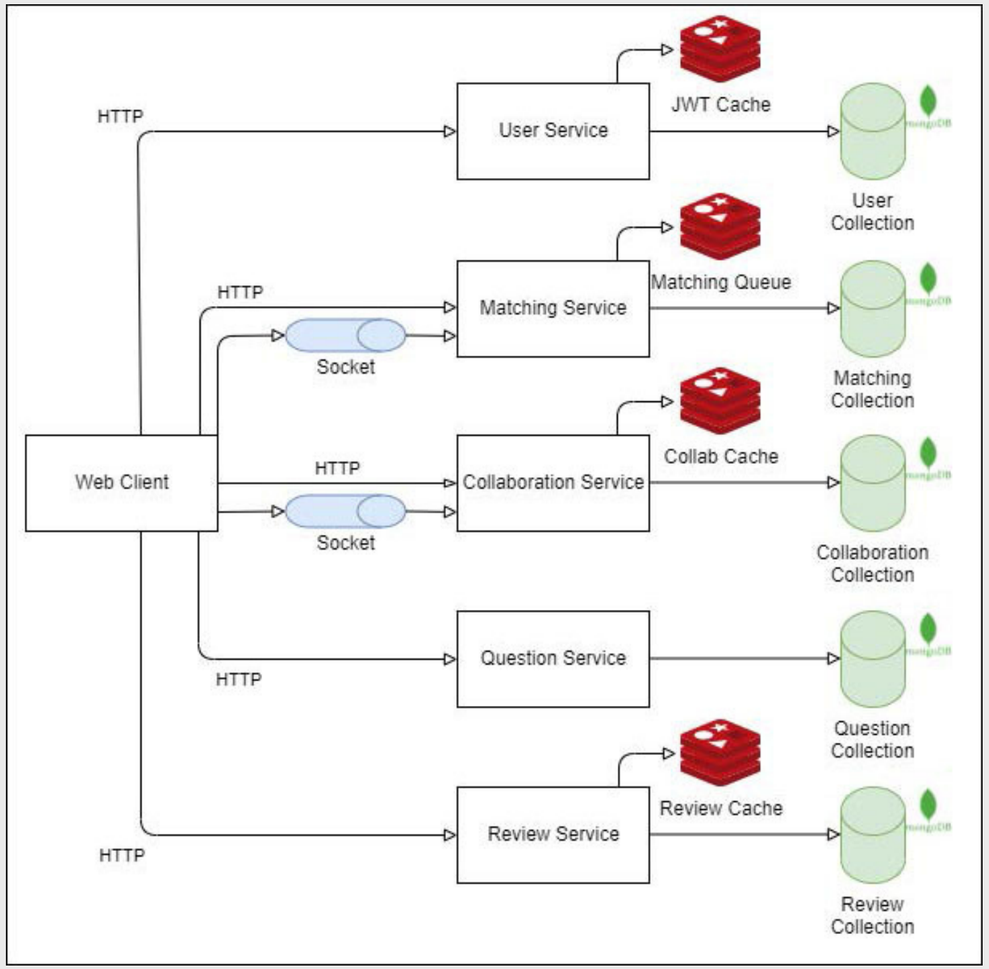
- HTTP calls between frontend and microservice
- HTTP calls from one microservice to another

## Redis

- Caching (for User, Collaboration and Review Services)
- Matching queue (for Matching Service)

# Architecture Diagram

DigitalOcean

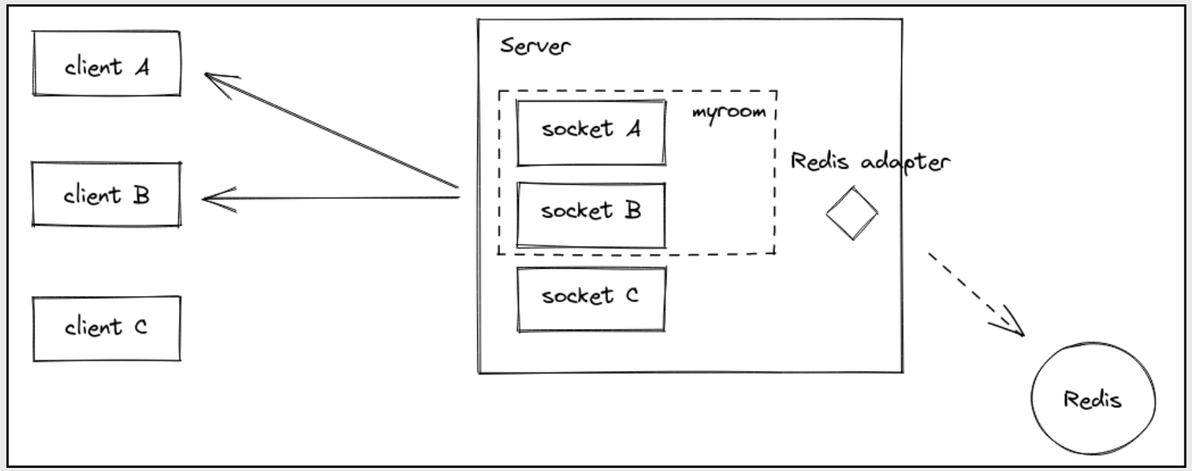



## Database

- Each microservice has its own MongoDB collection

## Socket

- Communication between users via socket.io events (emit, listen, broadcast) for matching and collaboration
- Two users that are matched will join a room



# User Service

# User Service

- Responsible for user authentication of the system
- CRUD operations of a user account
  - **Create**
    - Sign up for an account
  - **Retrieve**
    - Log in to an account
    - Retrieve the logged-in user
  - **Update**
    - Change password on the profile page
  - **Delete**
    - Delete an account on the profile page
    - Cleanup matching and collaboration data of the deleted account via HTTP endpoint for inter-microservice calls
- User account stored in MongoDB

# User Service

## Security Implementations

- Hashing of password with bcrypt library
- Use of cookies to store authenticated user's JSON Web Token (JWT)
- JWT expiration
  - Session is valid for 15 mins, after which the JWT expires
  - Any actions performed after the JWT expires will redirect user to the log in page

# User Service

## Security Implementations (continued)

- Multiple sessions on different devices/browsers not allowed
  - Check with Redis to ensure that JWT is valid
  - Invalid JWT will log the user out of the session

## Design Considerations

- Redis is used as a cache for JWT

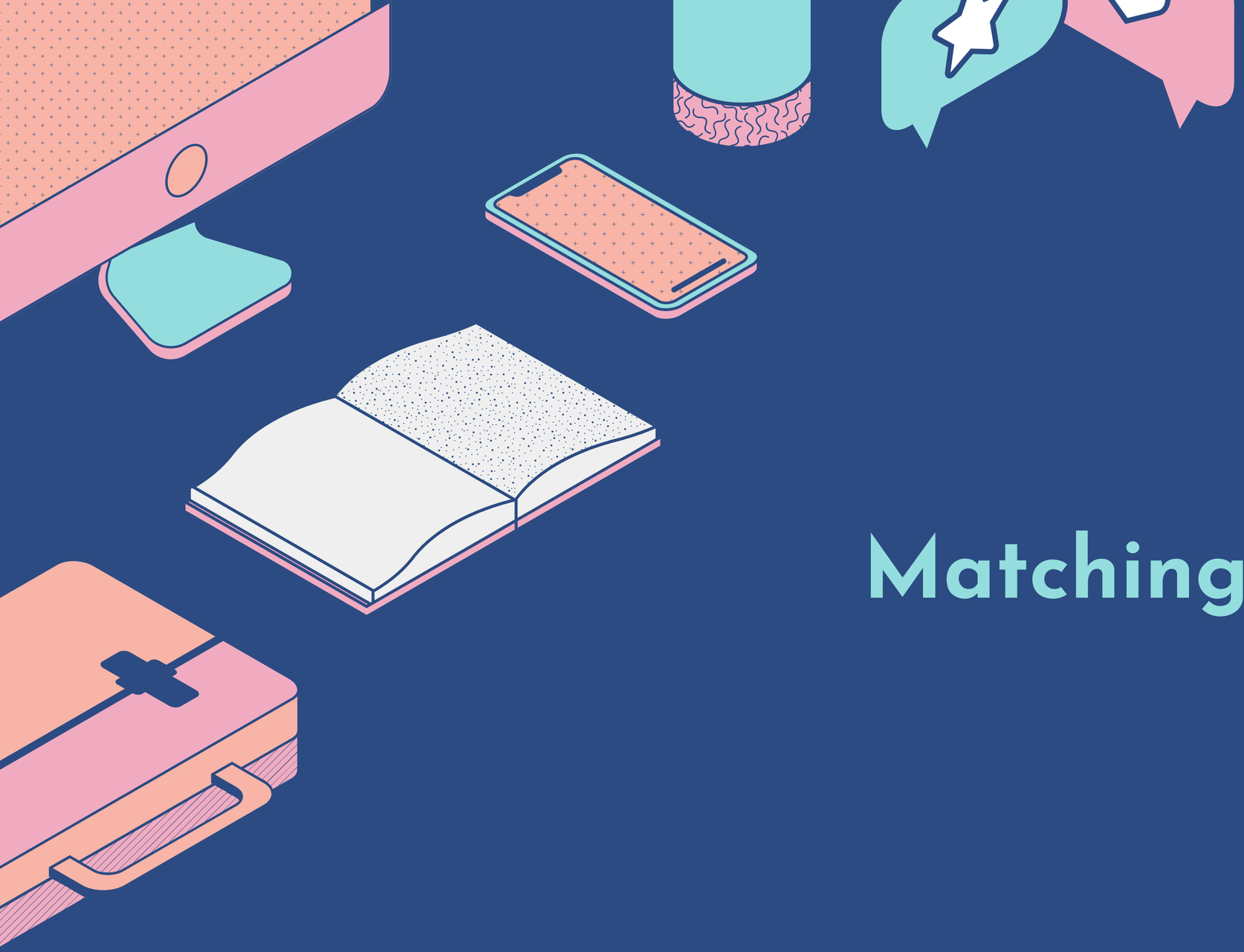
# Log in

Username

Password

CREATE AN ACCOUNT LOG IN

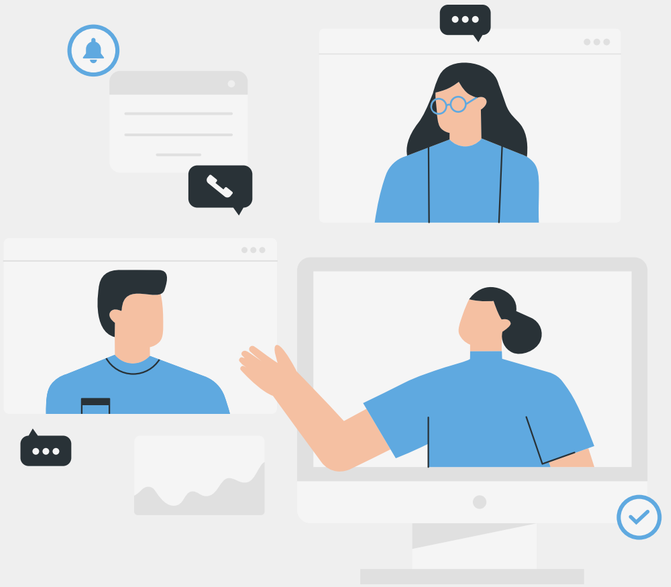
You have been logged out of your session. You may be logged in on another device or your session may have expired.



# Matching Service

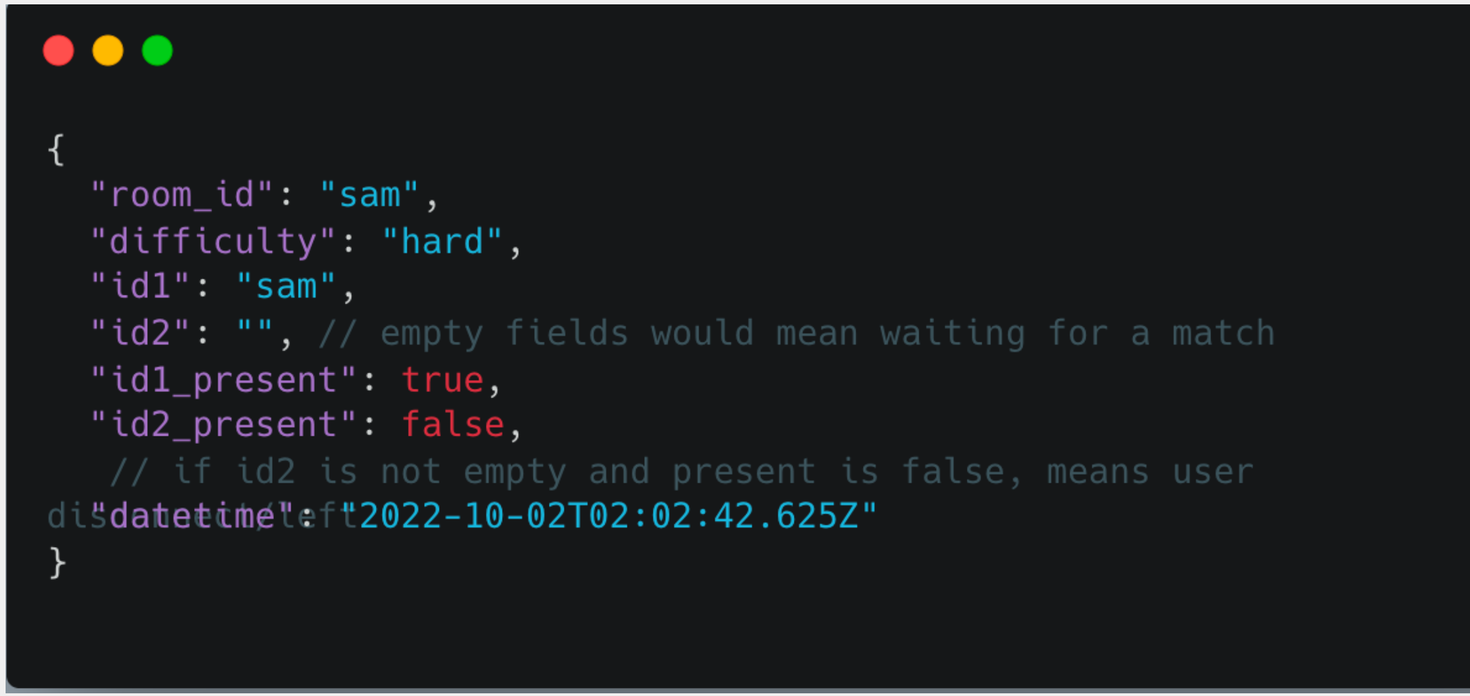
# Matching Service

- Responsible for establishing a connection between 2 users choosing the same difficulty
  - Creation of room
  - Pairing 2 unique users
  - Supports 3 different difficulty
- A successful match will persist data to MongoDB
- Handles different states of searching match



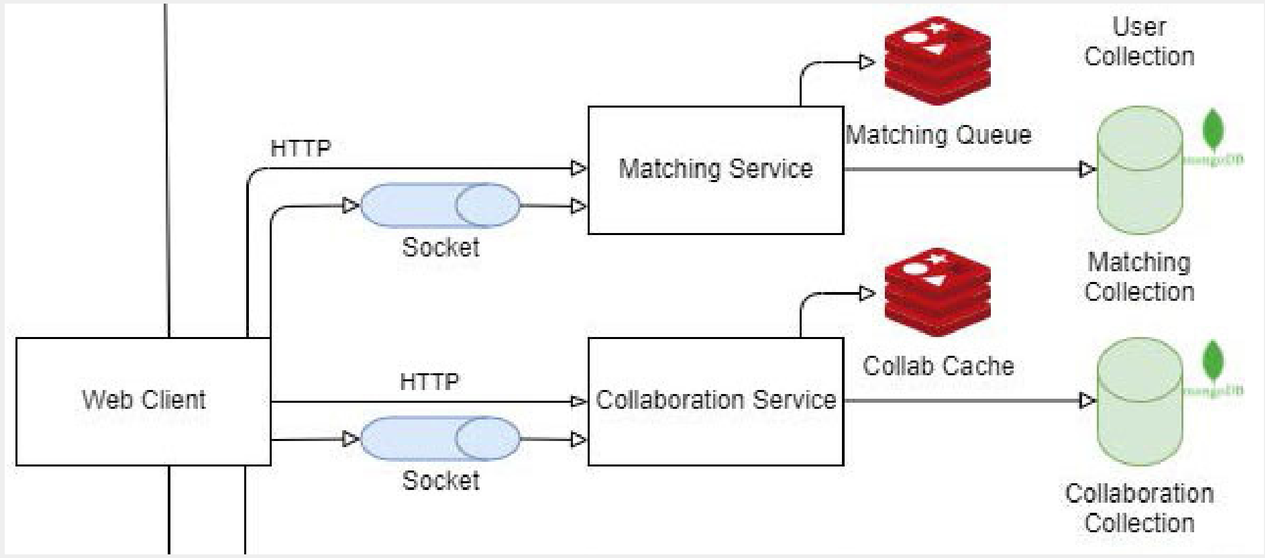
# Matching Service

- Initial design
  - MongoDB for matching
  - Persistence from DB allows for reconnecting
  - Expensive and slow



# Matching Service

- Current design
  - Redis for matching users
  - MongoDB for the persistence of room
  - Socket.io for synchronizing the users to match in real-time

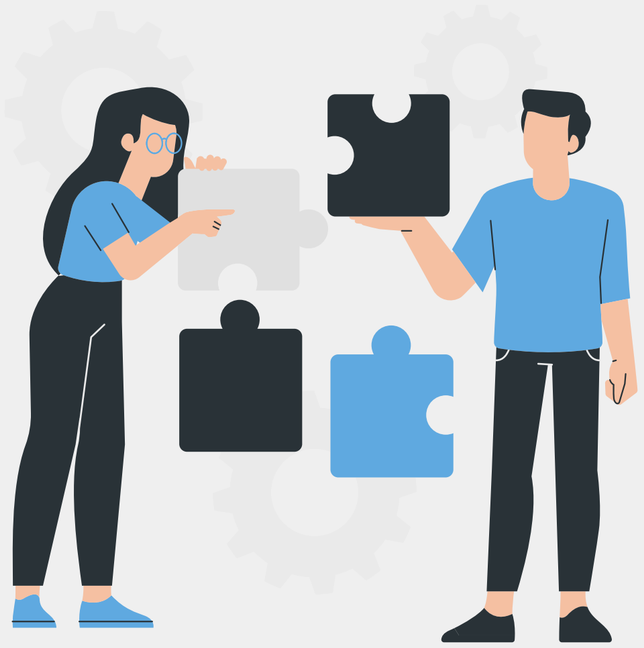


# Matching Service

- Observer Pattern
  - The client subscribes to the topic "found-connection" and is informed where another Client has been found.
- Otherwise, the Client may have to keep pinging the server to check if a match has been found.

```
  matchingSocket.on('found-connection', (username, difficulty, qnsid) =>
{   const room = {
      room_id: username,
      id1: username,
      id2: user.username,
      qnsid: qnsid,
      difficulty: difficulty.toLowerCase(),
      datetime: new Date(),
    }
    setRoom(room)
  })
```

# Collaboration & Question Service



# Collaboration Service

- Responsible for real-time collaboration between 2 users
  - Shared code editing
  - Chat feature
  - Question changes
- Allows persistence of collaboration data
- Handles disconnections & reconnections

# Question Service

- Provides randomly chosen questions based on difficulty
- Also allows the fetching of a specific question
- Each question contains a title, description, and 3 code examples

PeerPrep 📖

Time Remaining: 1 min

## Question

CHANGE QUESTION

**Search Insert Position**

Given a sorted array of distinct integers and a target value, return the index if the target is found. If not, return the index where it would be if it were inserted in order.
You must write an algorithm with `O(log n)` runtime complexity.

**Examples:**
Input: nums = [1,3,5,6], target = 5
Output: 2

Input: nums = [1,3,5,6], target = 2
Output: 1

Input: nums = [1], target = 0
Output: 0

## Chat

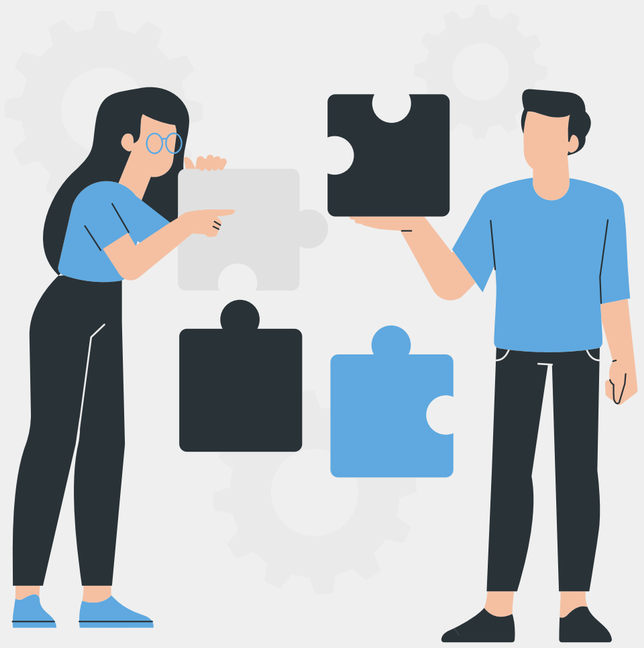
samtest10: Hello!

Type your message...

LEAVE ROOM

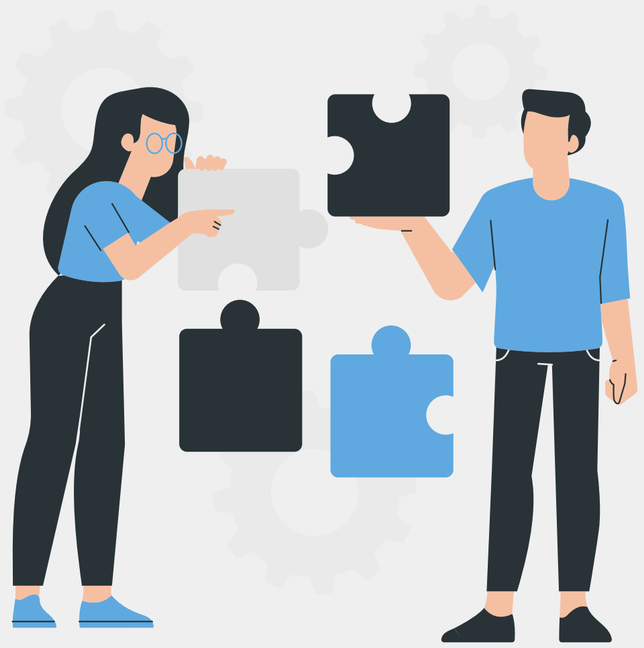
## Code Editor

```
const foo = (xs) => {
  return xs.map(x => x.strip())
}
```



# Collaboration Service

- Implemented as both a Socket.io and Express.js server
- Real-time collaboration through Websockets
  - 2 matched users placed in 1 Socket.io room
- HTTP endpoint for inter-microservice interaction



# Collaboration Service

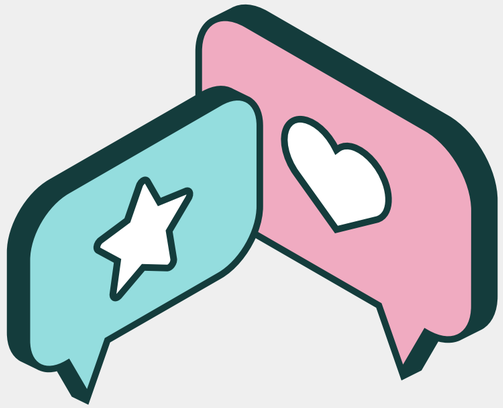
Design considerations:

- Redis is used as a cache
- The cache is flushed to MongoDB for persistence when any user disconnects
- Prevent hitting the DB too frequently

# Review Service

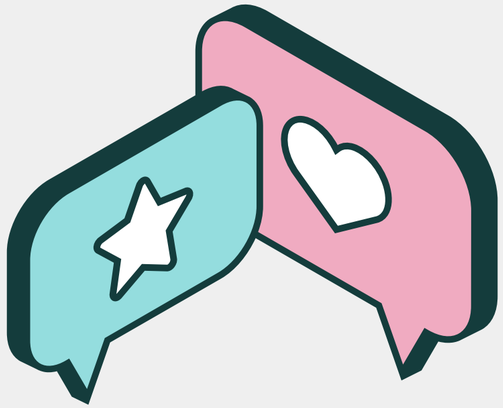
# Review Service

- Allows user to leave a review for his peer
- Review based on 5 qualities
- Allows user to view his ratings



# Review Service

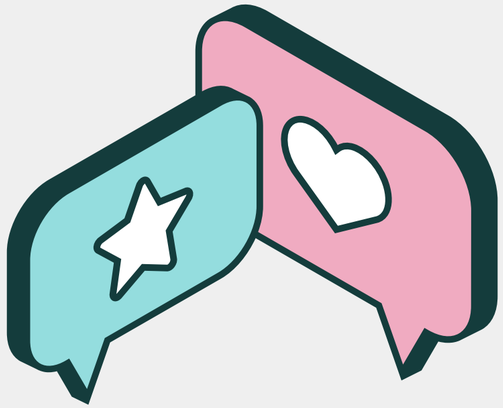
- Implemented with Redis and MongoDB
- Per user session, we query MongoDB at most once
- Subsequent reads from Redis
- New reviews to Redis and MongoDB



# Review Service

Design Considerations:

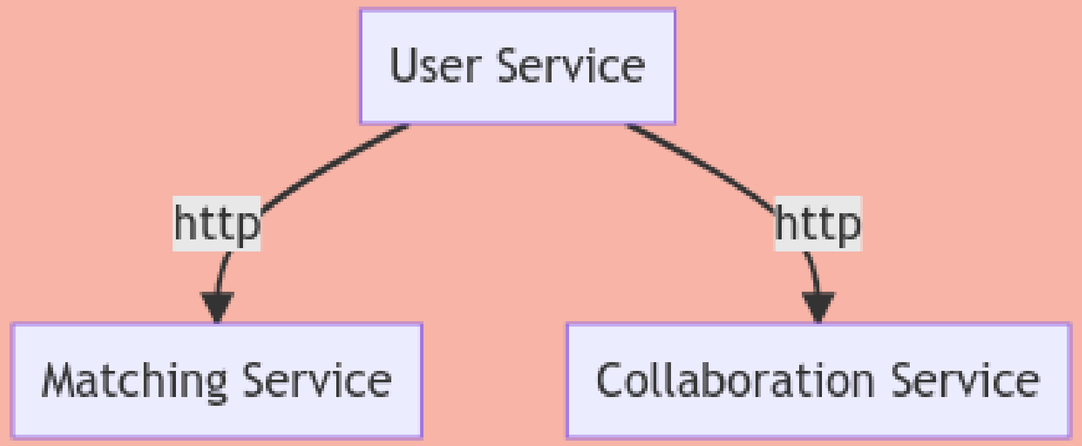
- Implement reviews in User Service
  - Violates SRP
  - Increases Coupling
  - Not extensible



# Other Significant Design Considerations

# Using HTTP Request-Reply Pattern between Microservices

**User deletes his/her account would require User Service to invoke the delete function of matching and collaboration service**

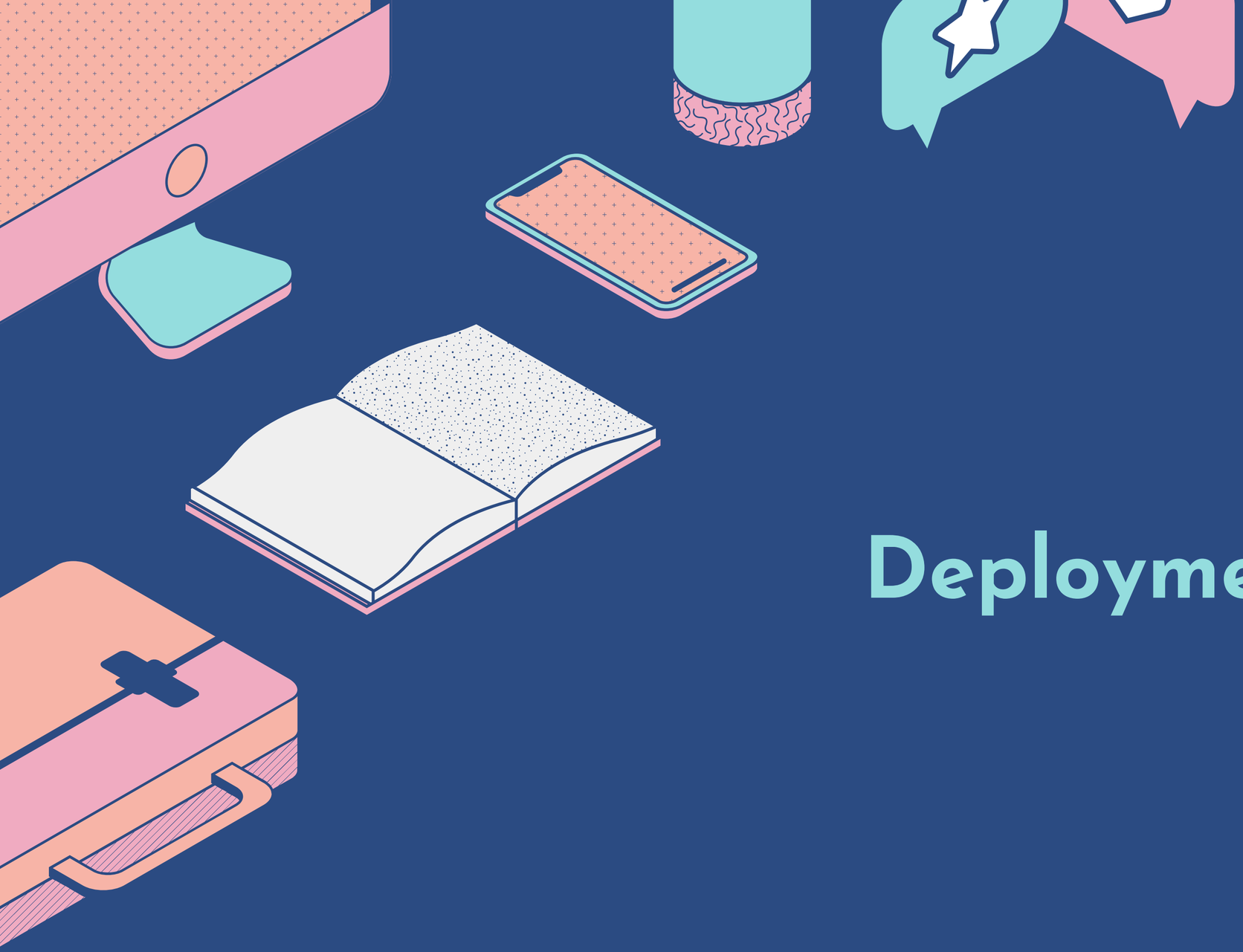



Why do we use HTTP instead of a message-passing solution

- Operations are not time or performance sensitive.
- Infrequent operation for delete
- May result in over-optimizing too early

# Using Redis as a Cache

- Redis is more performant
- Useful turnkey features such as TTL and persistence strategies
- The ease of switching local Redis to a hosted instance is simple unlike in-application memory

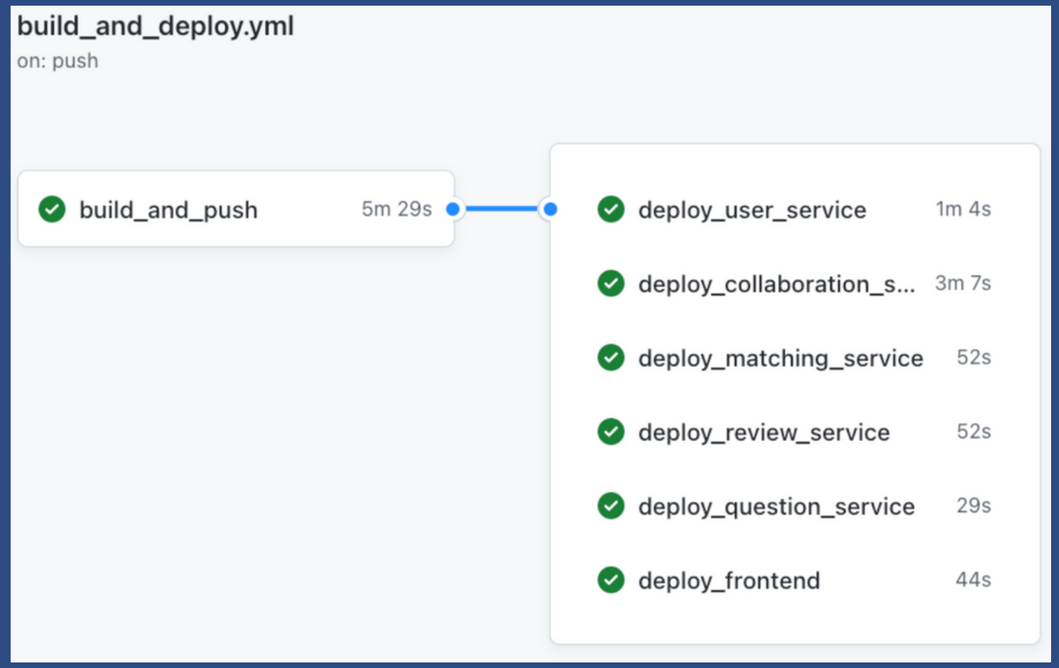
# Deployment

# Deploying PeerPrep

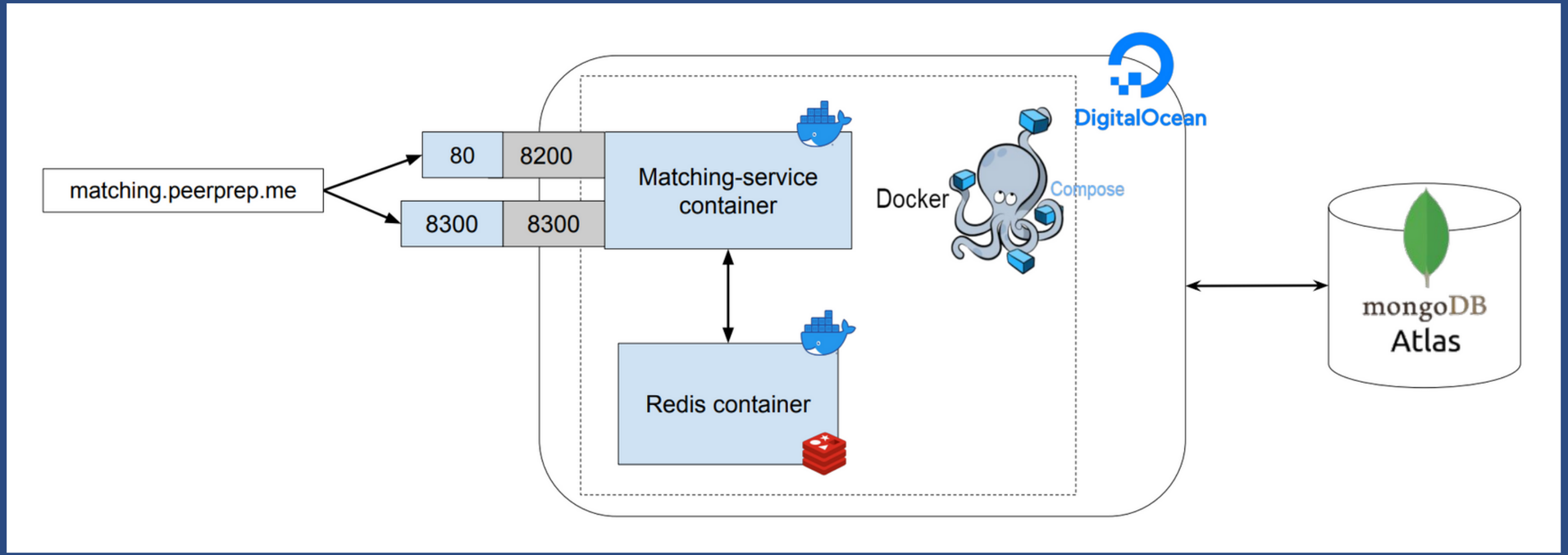
Pipeline runner: GitHub Actions
Cloud provider: DigitalOcean

How we deploy:

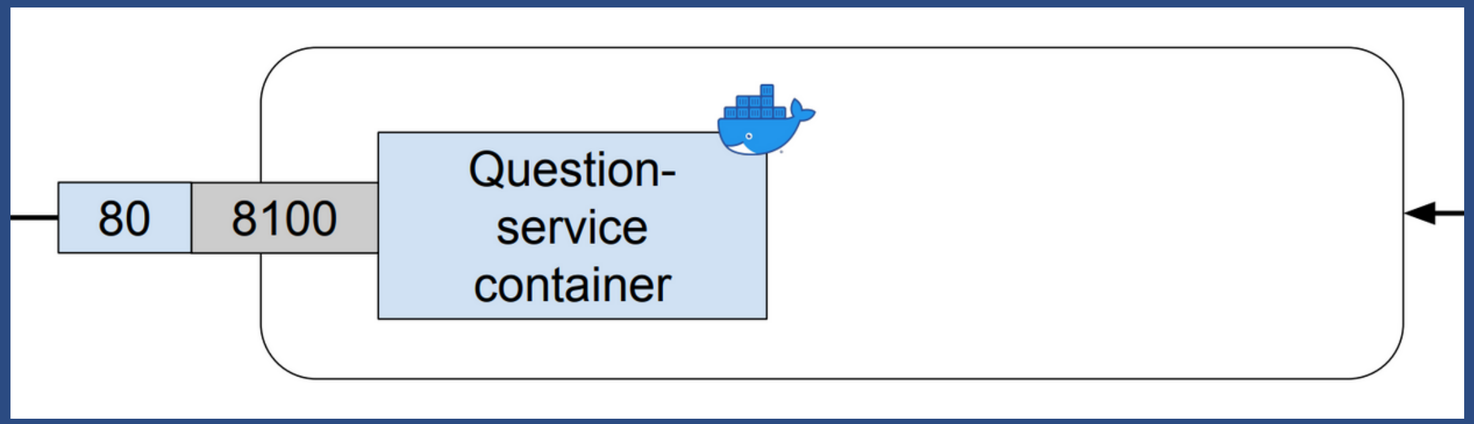
1. Build Docker image & push to DOCR
2. SCP Docker Compose file to Droplet (if any)
3. SSH to Droplet
4. Spin down container/stack
5. Pull new images
6. Spin up new container/stack



# Deploying PeerPrep



# Deploying PeerPrep



# Our Team

Toh Bing Cheng

Back-end

Ryan Cheung

Full Stack

Lau Siaw Sam

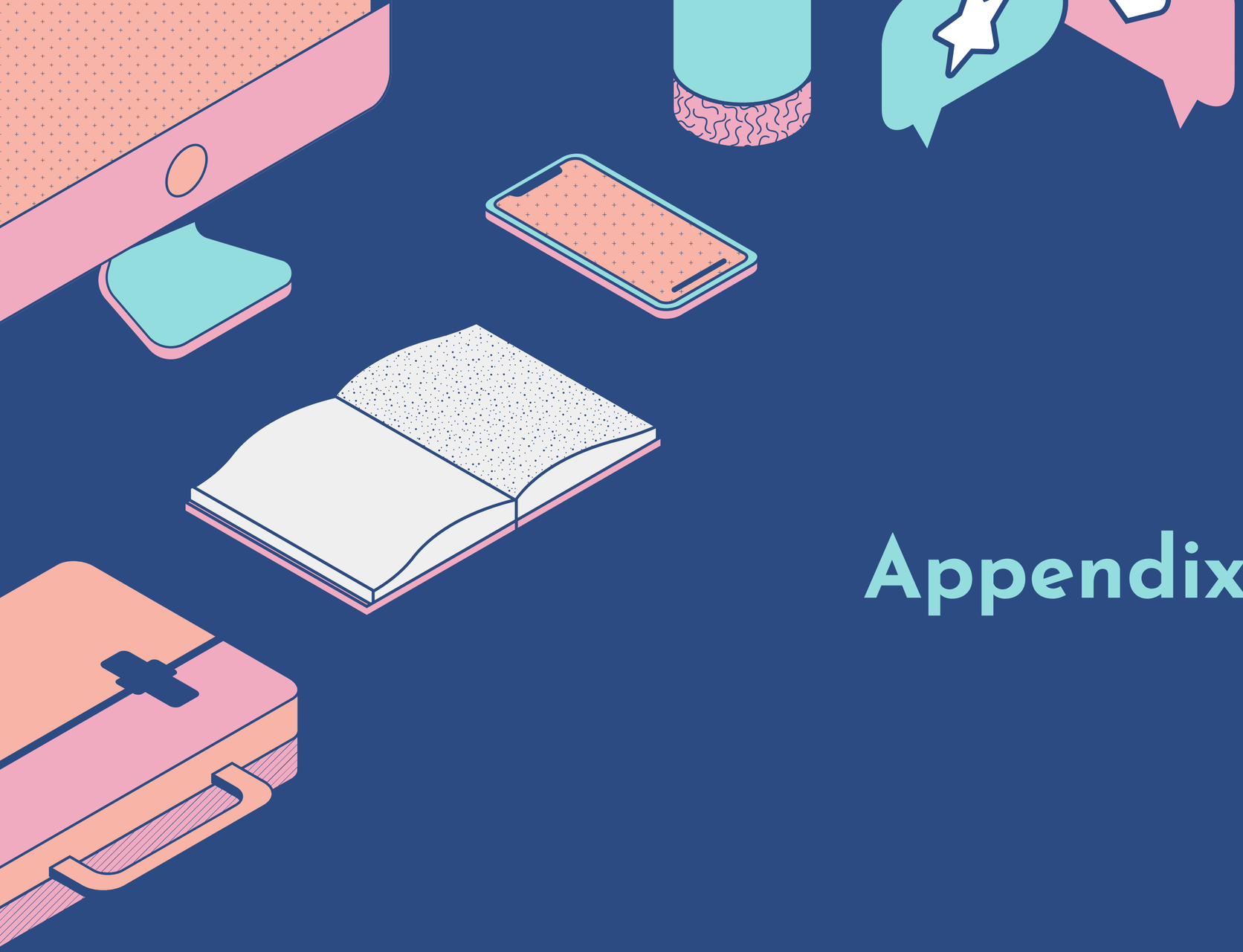
Full Stack

Tay Jun Yang

Front-end

# Do you have any questions?

Thank you for your kind attention!



# Appendix

# Feature Requirements Implemented

We implemented all stated must-haves:

- User service
- Matching service
- Question service
- Collaboration service
- Basic UI
- Deploying locally (native stack)

Nice-to-haves:

- Fancy UI
- Chat feature
- Review service
- Deploying locally (Docker)
- Deployment on DigitalOcean
- Continuous Delivery/Deployment pipeline