Chapter 10 Design a Notification System

What is a notification system?

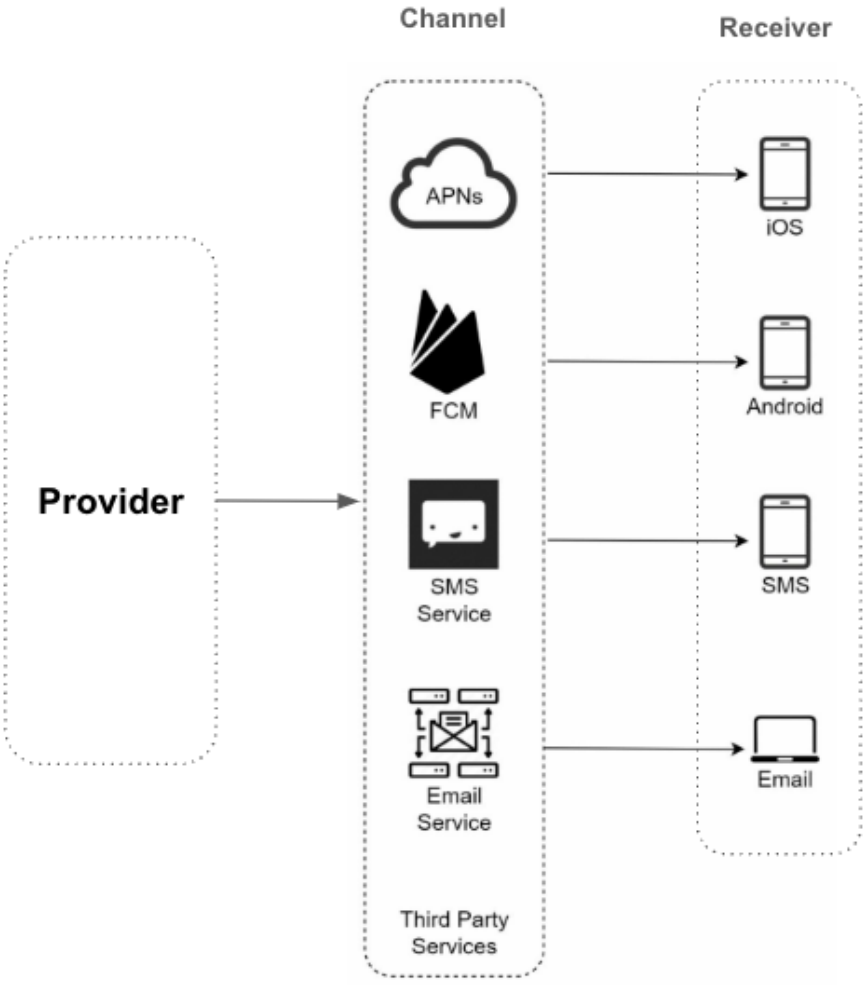
Definition

- A notification alerts a user with important information like breaking news, product updates, events, offerings, etc.
- Wiki: A notification system is a combination of software and hardware that ***provides a means of delivering a message to a set of recipients***.
- GPT: A **notification system** is a framework or service designed to send real-time alerts or messages to users or systems. These notifications typically provide important updates, warnings, or reminders based on specific events or triggers.

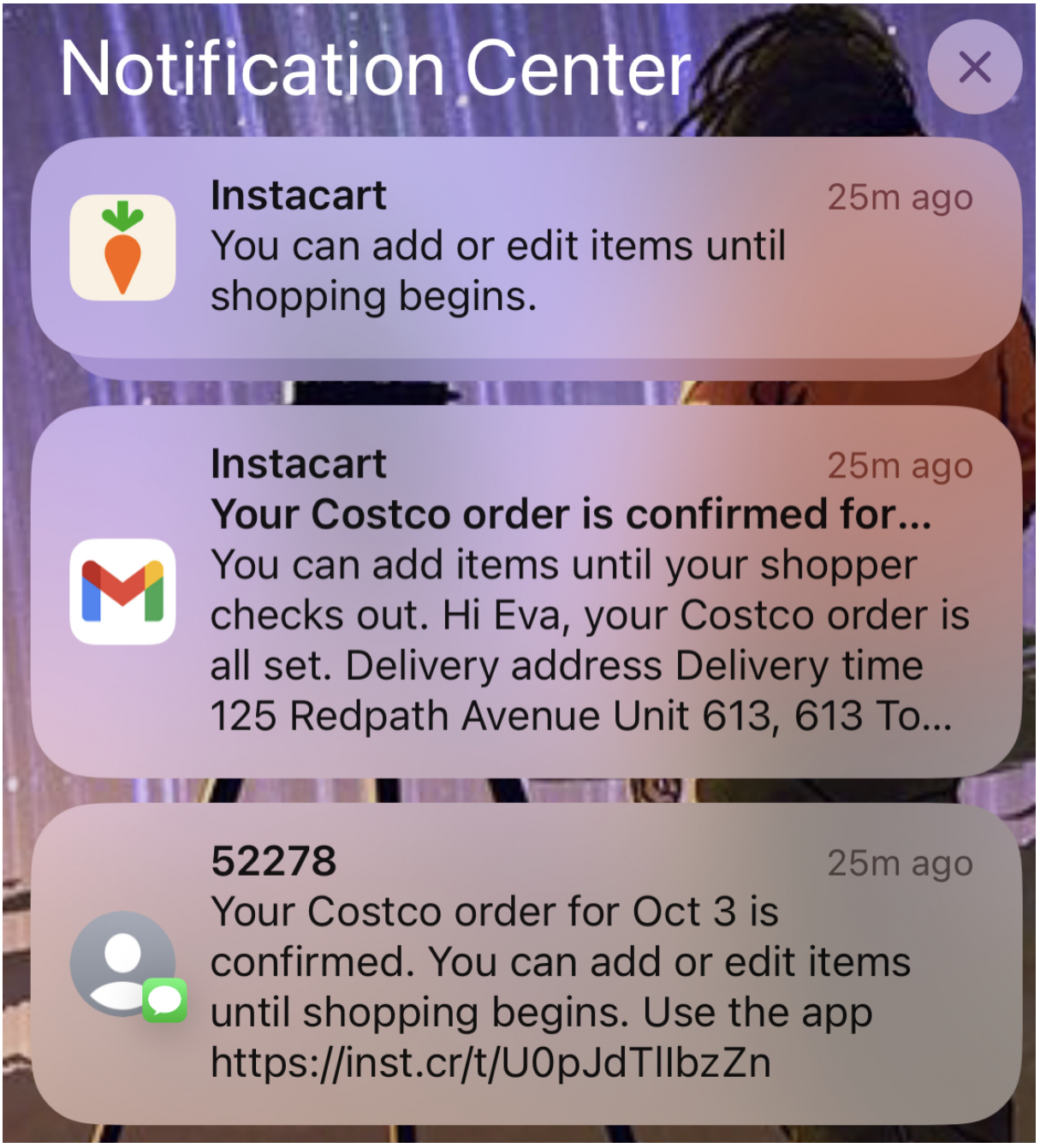
Format

- Mobile push notification
- SMS message
- Email

Components of a Notification System



*Note: **Firebase Cloud Messaging** (**FCM**), formerly known as Google Cloud Messaging (GCM), is a cross-platform cloud service for messages and notifications for Android, iOS, and web applications. On October 21, 2014, Firebase announced it had been acquired by Google for an undisclosed amount.- - Wiki



A notification system generally consists of the following components:

1. **Event Source**:
    - This is where the trigger or event occurs, which initiates the notification. It could be user activity, system updates, or third-party integrations.
    - Example: A user receives a message, a new software update is available, or a bank transaction is processed.
2. **Notification Generator**:

- This component captures events from the event source and determines if a notification needs to be generated.
- Example: It decides whether to send a message alert, a system error, or an order status update.

3. **Notification Channel**:
    - The notification system can use various channels to deliver messages to users. Common channels include:
        - **Email** notifications
        - **Push** notifications (on mobile devices or web browsers)
        - **SMS** or text messages
        - **In-app** notifications (within a web or mobile app)
        - **Desktop** notifications
        - **Webhook** notifications (for sending messages to other systems)
4. **Delivery Mechanism**:
    - This is the infrastructure that ensures the message reaches the user. It might involve third-party services, APIs, or messaging queues.
    - Example: Firebase Cloud Messaging (FCM) for push notifications, Twilio for SMS notifications, or Mailgun for email delivery.
5. **Notification Manager**:
    - The core logic behind the system. It manages:
        - **Delivery retries** in case of failures.
        - **Scheduling** notifications (sending them at specific times or intervals).
        - **User preferences**, which allow users to choose which notifications they want to receive or which channels to use.
        - **Prioritization** of notifications (e.g., high-priority alerts vs. regular updates).
6. **Tracking and Analytics**:
    - This component collects data on how notifications are delivered and how users interact with them.

- Metrics include **delivery success rates**, **user engagement** (such as open or click rates), and **response times**.
- Example: How many users opened the notification, how many ignored it, and if any actions were taken based on it.

Characteristics of a Good Notification System

1. **Scalability**:
   - The system should be able to handle large volumes of notifications, especially during peak times, without delays or failures.
2. **Reliability**:
   - Ensures that notifications are delivered consistently and on time, even if the system experiences failures or downtime.
3. **Personalization**:
   - Users should have control over the types of notifications they receive, and the content should be personalized based on user preferences or behavior.
4. **Real-Time**:
   - Notifications should be delivered promptly when an event occurs, especially for time-sensitive alerts such as fraud detection or emergency notifications.
5. **Multi-Channel Support**:
   - Users should be able to receive notifications through different channels depending on their preferences or urgency (e.g., mobile push, SMS, email, etc.).
6. **Tracking and Analytics**:
   - The system should track whether notifications are delivered successfully, whether they are opened, and how users interact with them, providing feedback to improve the system.

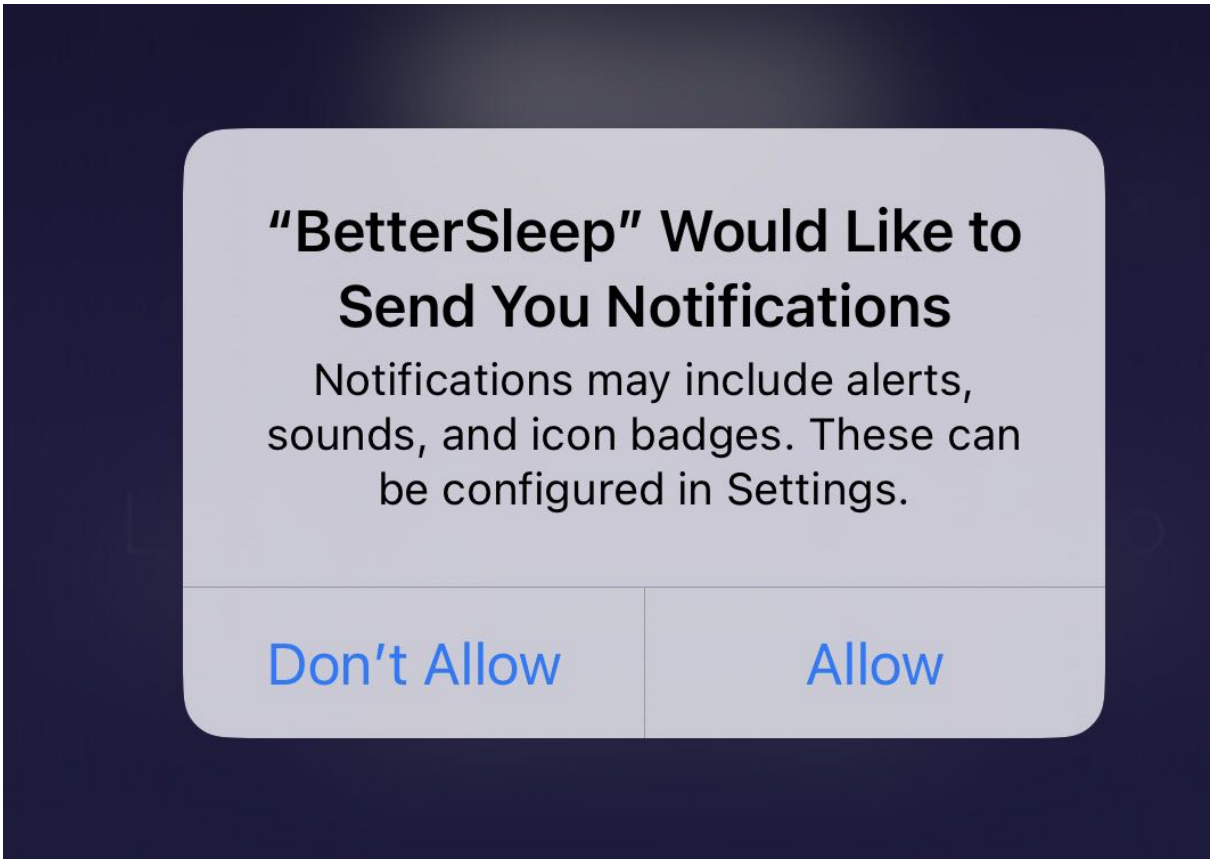
Why we need a notification system?

Alert users about key information or events.

Background: How does push notification work?

When we install an app on your phone, the registration typically happens during the app's initialization process.

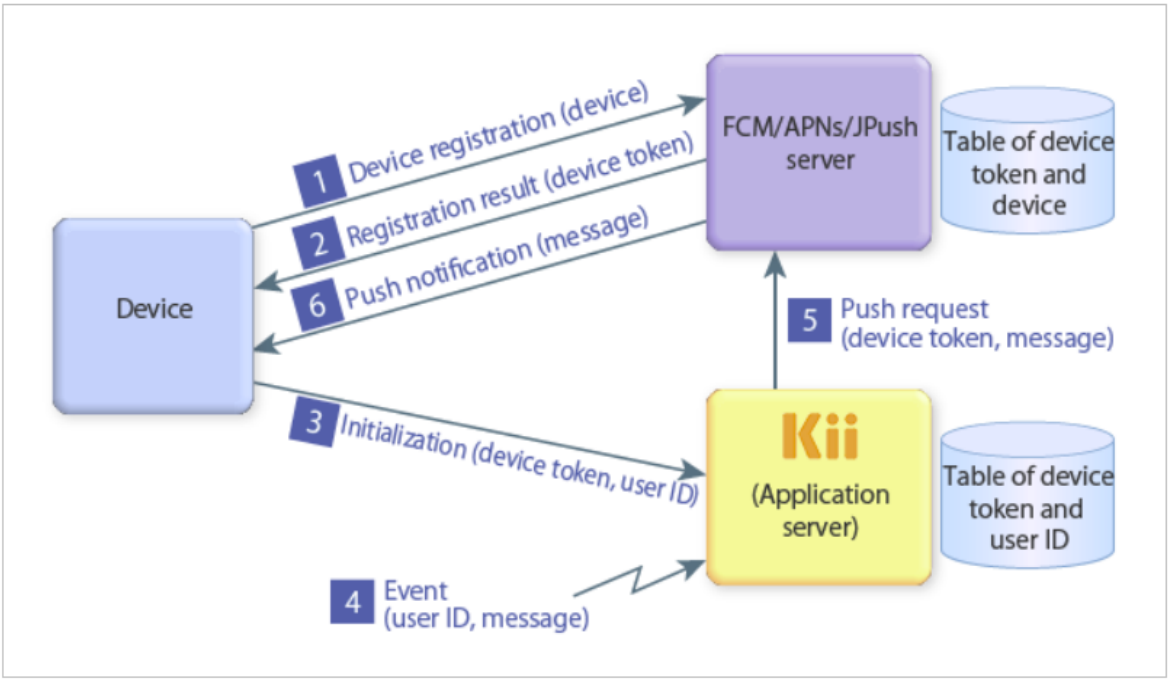
- **iOS (APNs)**: Registration with APNs occurs when the user first grants permission for notifications after installing the app. The device token is then received and sent to the app's backend server.
- **Android (FCM)**: Registration with FCM typically happens automatically when the app is first initialized (launched), and the registration token is received and sent to the app's backend.



1. A device sends a request to the FCM or APNs server to register the device.
2. The FCM or APNs server registers the device and issues a device token.
3. The device calls the push initialization API of the Kii Cloud SDK when a user logs in. The API sends the device token and the user ID to Kii Cloud. These are stored in Kii Cloud as a pair.
4. An event occurs on Kii Cloud. Using the stored pair of the user ID and device token, Kii Cloud identifies the recipient user's device to send a push notification.

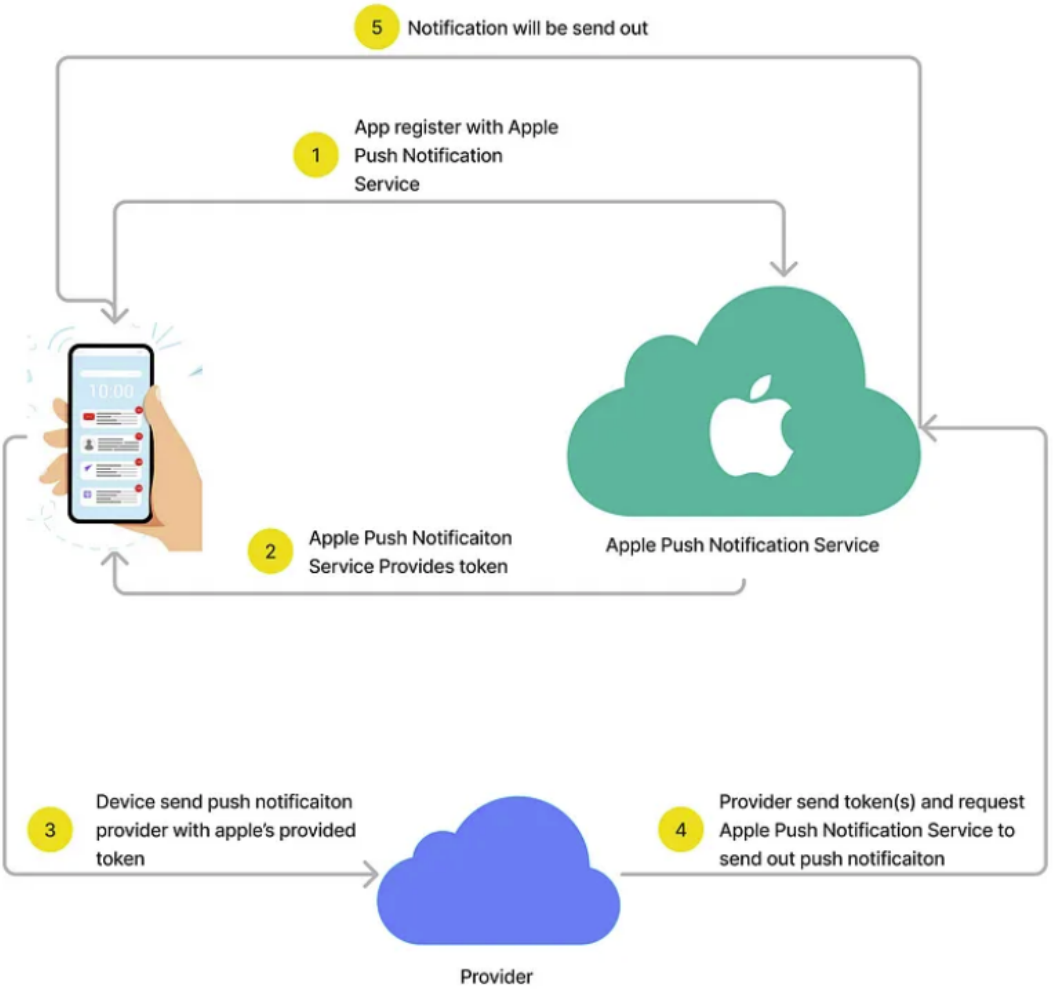
5. Kii Cloud requests the FCM or APNs server to send a push notification to the target device token.
6. The FCM or APNs server sends a push message to the device. The device processes the received push message.

- ISO/Android



https://docs.kii.com/en/guides/cloudsdk/rest/managing-push-notification/push-overview/structure/

- IOS



https://achsuthan.medium.com/how-does-ios-push-notification-work-bcedc1bcf37b

How to design a notification system?

Step 1 - Understand the problem and establish design scope

Suggested Flow of Asking Questions:

1. **Start with the High-Level Use Case**:
    - Understand the general purpose of the system and the main problem it is solving.
2. **Ask About Users and Channels**:
    - Determine who the users are and how they expect to receive notifications.

3. **Get Into the Technical Details**:
    - Ask about scalability, event triggers, delivery mechanisms, and failure handling.
4. **Address User Preferences and Security**:
    - Understand how users will control notifications and any potential security concerns.
5. **End with Performance and Monitoring**:
    - Cover performance expectations, monitoring, analytics, and potential cost constraints.

Candidate: What types of notifications does the system support?
Interviewer: Push notification, SMS message, and email.

2. Channels

Candidate: Is it a real-time system?
Interviewer: Let us say it is a soft real-time system. We want a user to receive notifications as soon as possible. However, if the system is under a high workload, a slight delay is acceptable.

5. Performance

Candidate: What are the supported devices?
Interviewer: iOS devices, android devices, and laptop/desktop.

1. Users

Candidate: What triggers notifications?
Interviewer: Notifications can be triggered by client applications. They can also be scheduled on the server-side.

3. Technical details: event triggers

Candidate: Will users be able to opt-out?
Interviewer: Yes, users who choose to opt-out will no longer receive notifications.

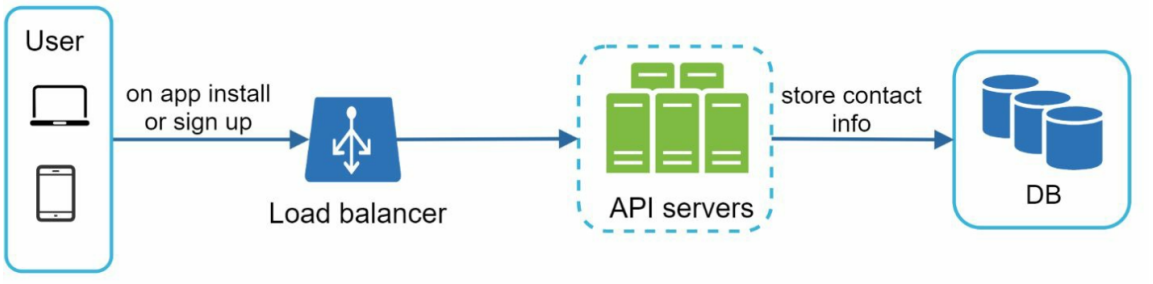
6. User preference

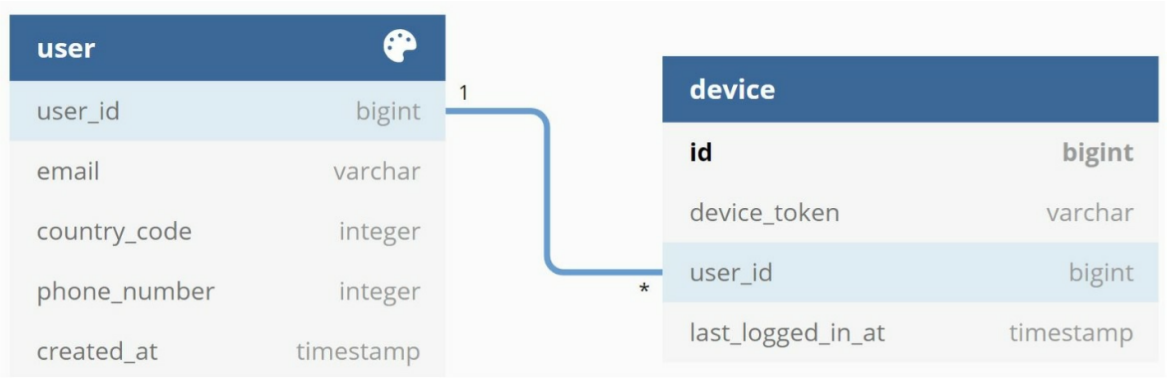
Candidate: How many notifications are sent out each day?
Interviewer: 10 million mobile push notifications, 1 million SMS messages, and 5 million emails.

4. Technical details: scalability

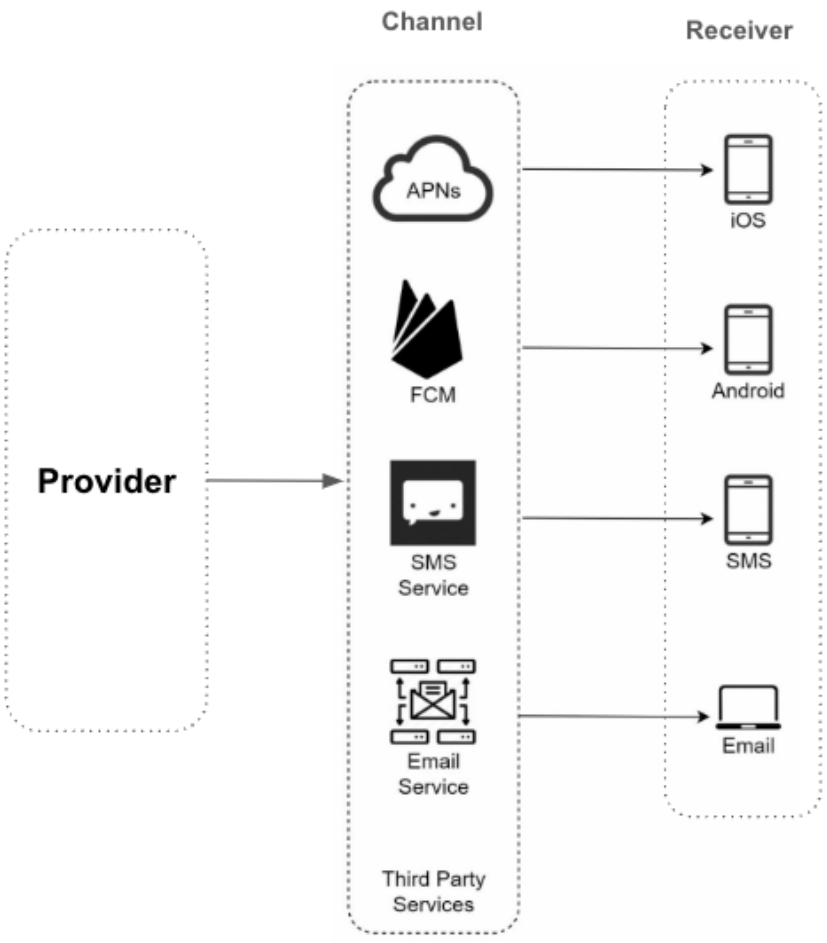
Step 2 - Propose high-level design and get buy-in

- Contact info gathering

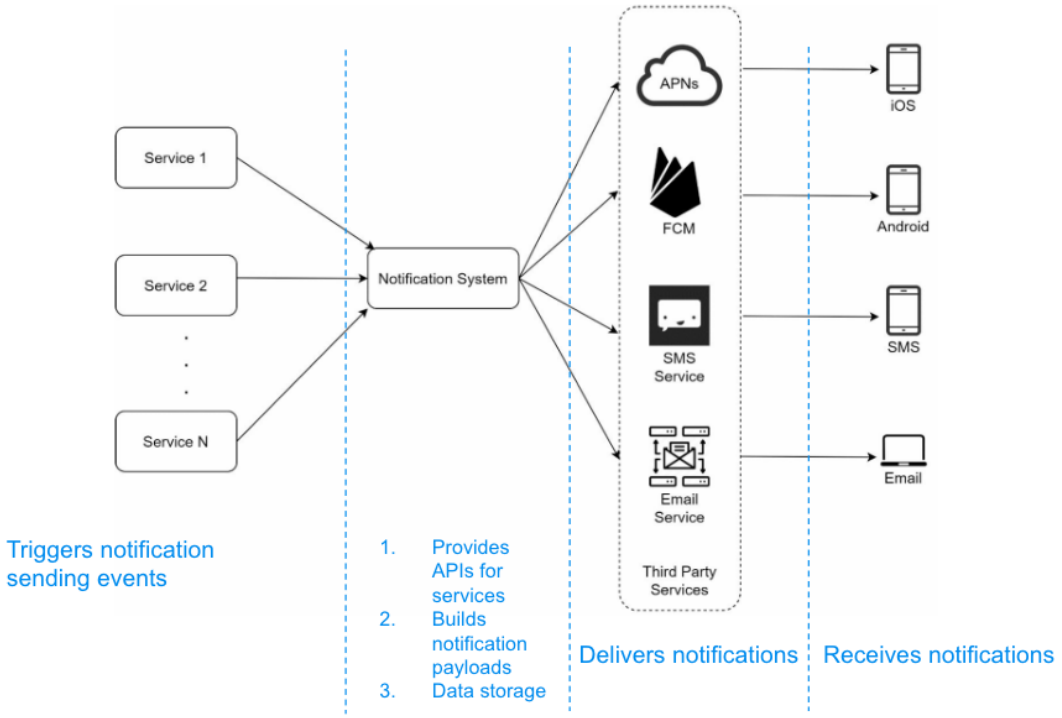




- Notification sending/receiving
  - Initial Design



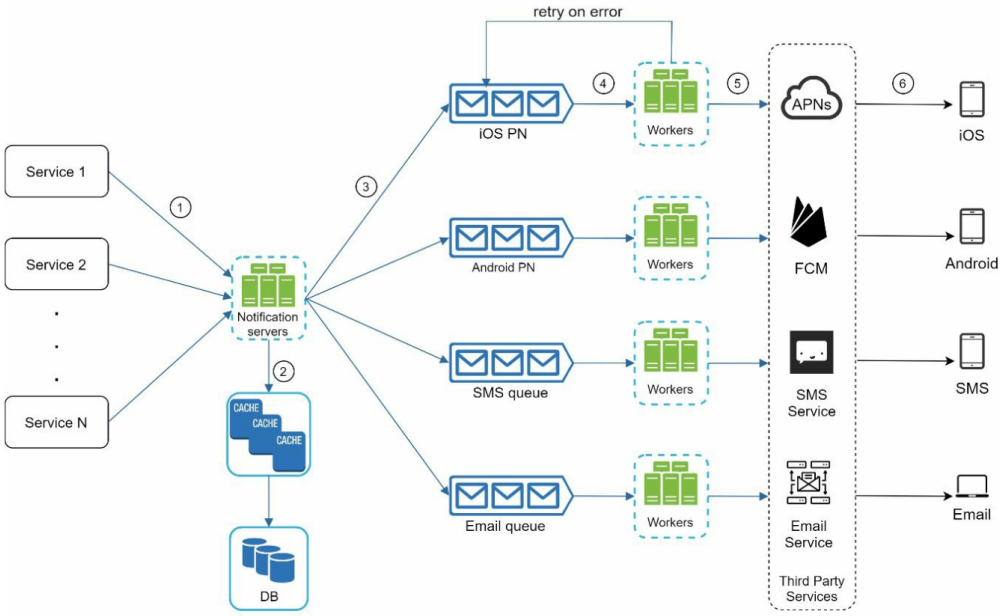
*Note: **Firebase Cloud Messaging** (**FCM**), formerly known as Google Cloud Messaging (GCM), is a cross-platform cloud service for messages and notifications for Android, iOS, and web applications. On October 21, 2014, Firebase announced it had been acquired by Google for an undisclosed amount.- - Wiki



Problem:

- Only one notification server → Single point of failure
- Hard to scale?
- Performance bottleneck: system overload, especially during peak hours

- Improved Design



*Note: PN stands for push notification

Added components:

- More notification servers
  - Provides APIs for services
  - Basic verifications(email, phone number, user ID...)
  - Query cache or database to fetch data needed for rendering a notification
  - Send data to message queue for parallel processing

  Benefits:
  - Avoid single point of failure
  - Support automatic horizontal scaling and avoid system overload
- Cache: User info, device info, notification templates
- DB: data about user, notification, settings, etc.
- Message queue to decouple system components
  - remove dependencies between components
  - serve as buffers

  Benefits:
  - avoid impacting other notification types if one third-party failed

    - More efficient processing and sending notifications
- Workers
    - Pull notification events from message queues and send to third-party services

Example to explain how every component works together to send a notification:

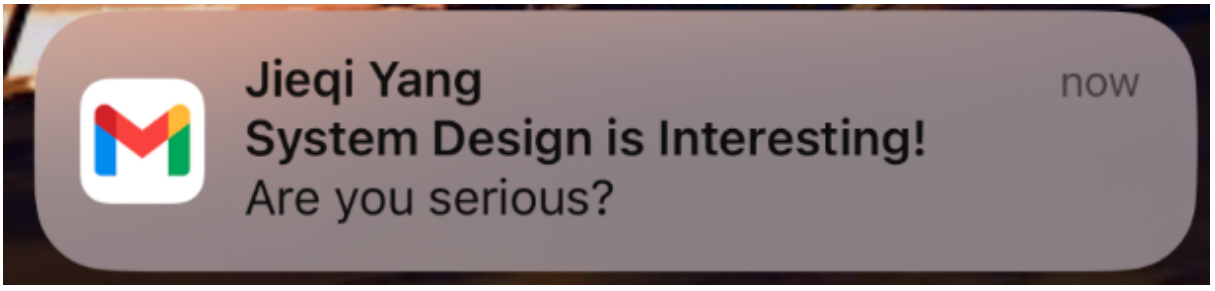
1. Call APIs provided by notification servers and send a notification

```
POST https://api.example.com/v/email/send

RequestBody:
{
    "to":[
        {
            "user_id": 12345
        }
    ],
    "from":{
        "email": "yang.jieq@......."
    },
    "subject": "System Design is Interesting!",
    "content": [
        {
            "type": "text/plain",
            "value": "Are you serious?"
        }
    ]
}
```

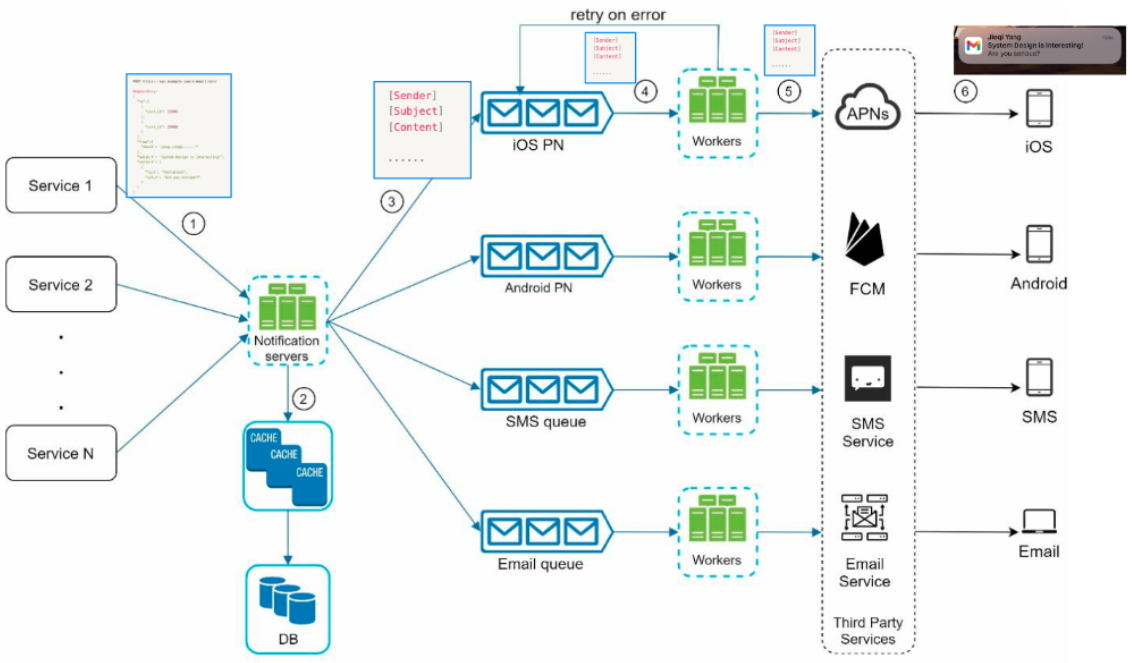
2.

- Notification servers fetch and valid metadata(userID, emails, device token, etc.)
- Build the notification payload

```
[Sender]
[Subject]
[Content]

......
```



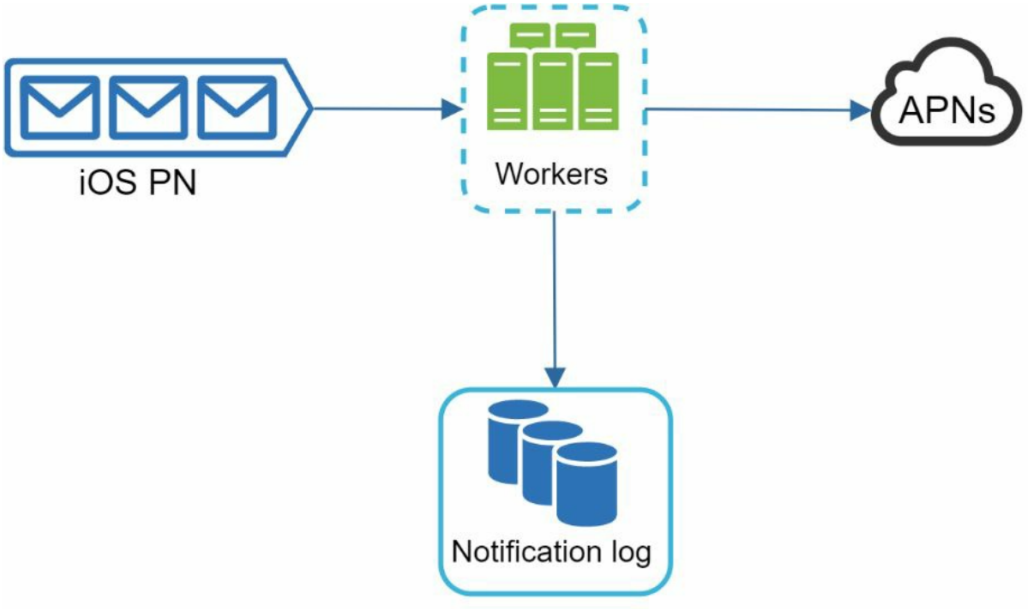
3-6:



Step 3 - Design deep dive

- Reliability
  - To prevent data loss

- Persist notification data in databases
- Implement retry mechanism

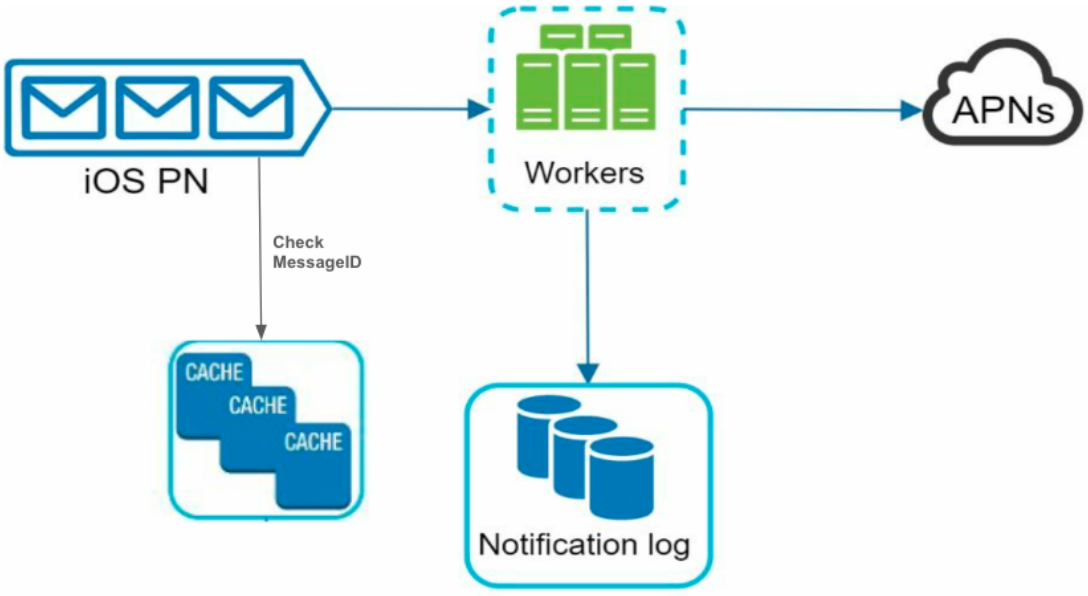


- Implement dedupe mechanism to reduce duplication occurrence

  This is commonly used in **message queues** like **RabbitMQ** or **Kafka**. The system can store **message IDs** in a cache (e.g., Redis) and check each incoming message against the cache to prevent duplicates from being processed.

```
POST https://api.example.com/v/email/send

RequestBody:
{
    "MessageID": "550e8400-e29b-41d4-a716-446655440000", // UUID
    "to":[
        {
            "user_id": 12345
        }
    ],
    "from":{
        "email": "yang.jieq@......."
    },
```

```
    "subject": "System Design is Interesting!",
    "content": [
        {
            "type": "text/plain",
            "value": "Are you serious?"
        }
    ]
}
```

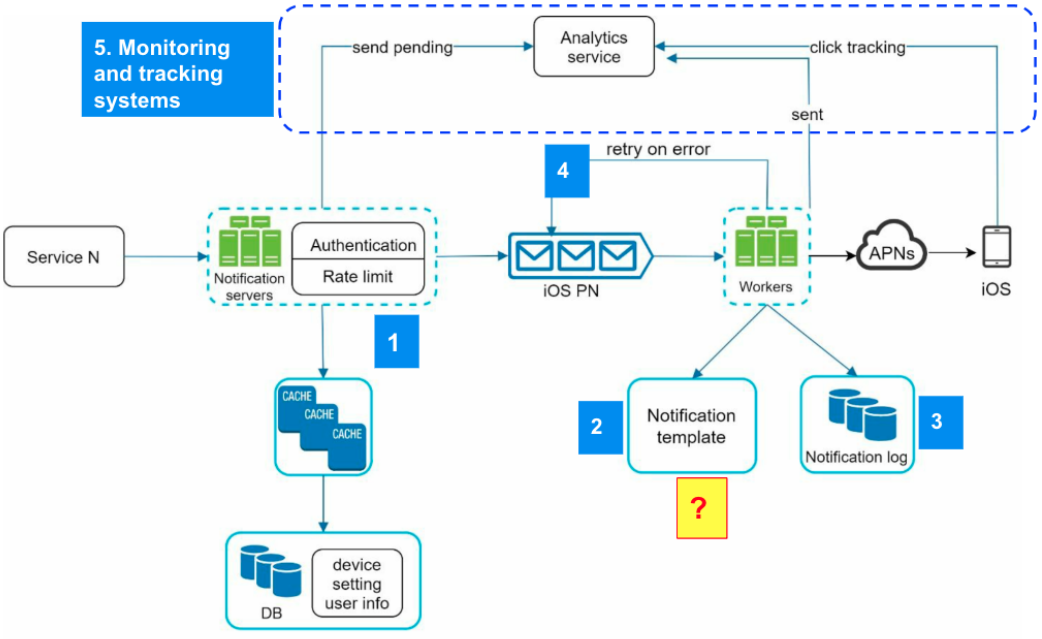


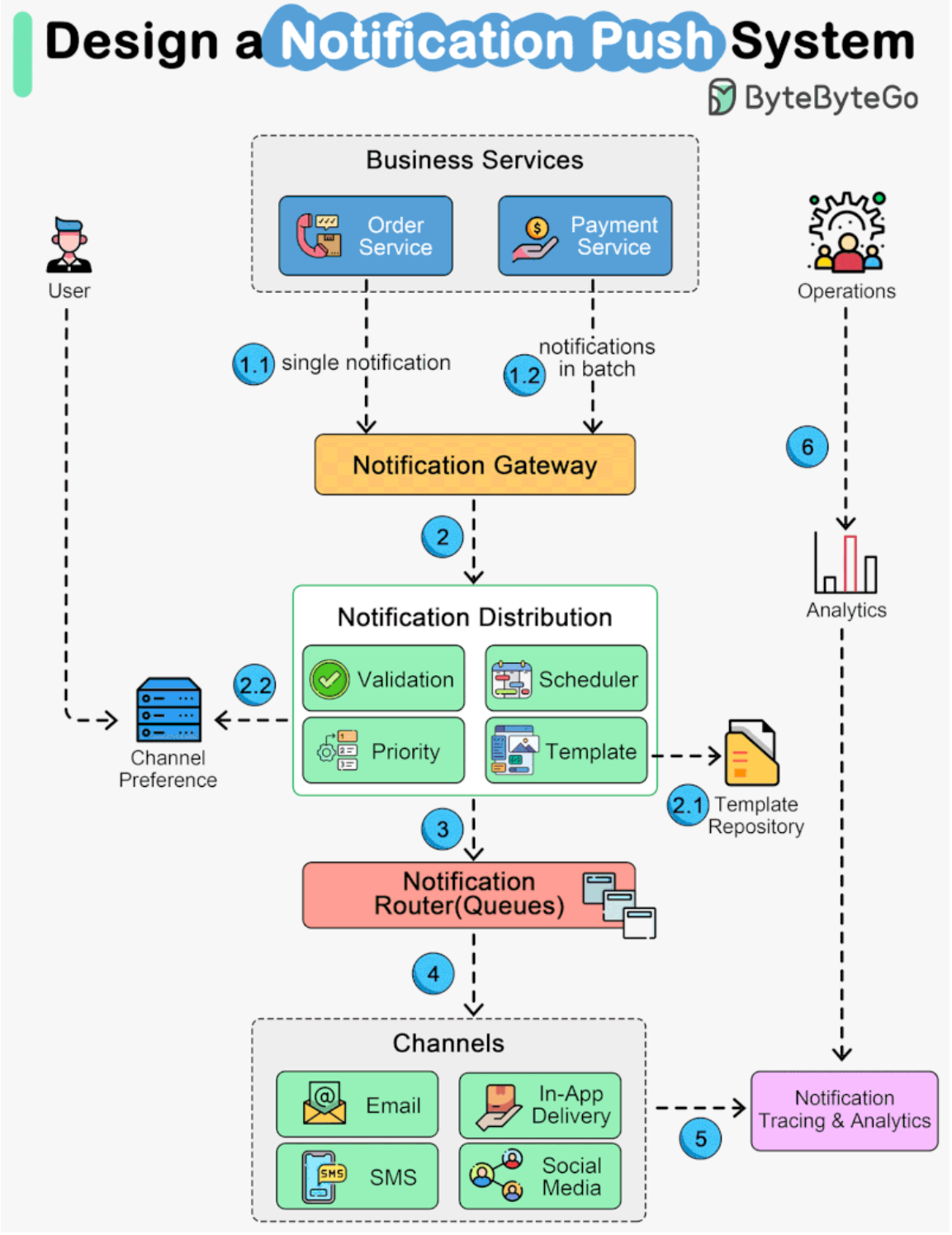
- Additional components and considerations
  - Notification template for maintaining a consistent format, reducing the margin error, and saving time
  - Notification setting: give users fine control over notification settings, such as opt-out
    - Notification setting table stored in DB, check opted-in status before sending notifications

```
user_id bigInt
channel varchar // push notification, email or SMS
```

```
opt_in boolean // opt-in to receive notification
```

- Rate limiting
  - Avoid overwhelming users
- Retry mechanism: When a third-party failed to send a notification, the notification will be added into the message queue again for retrying.
- Security in push notifications: Only verified clients are allowed to send notifications with APIs provided by notification servers
- Monitor queued notifications: If there are too many queued notification, the workers will process very slow. To avoid delay in notification delivery, more workers could be added.
- Event tracking: metrics, such as open rate, click rate, ...are useful in business analysis





https://www.linkedin.com/posts/alexxubyte_systemdesign-coding-interviewtips-activity-7150527952118575105-Y5Bu/

Step 4 - Wrap up