GCP Pub/Sub Best Practices

Deep Dive: Designing Topics and Subscriptions in GCP Pub/Sub

Designing an efficient **topic and subscription** architecture is critical for ensuring **scalability, reliability, and maintainability** in a GCP Pub/Sub system. Here's how to approach this effectively:

1️⃣ Topic Design Strategies

✅ 1.1 One Topic per Event Type

- A topic should represent a single type of event.
- This avoids mixing unrelated messages in a single queue.
- Example:
  - `` `user-signups` ``
  - `` `order-placed` ``
  - `` `payment-processed` ``

✅ 1.2 Separate Topics for High and Low Priority Messages

- If some messages are critical (e.g., fraud detection alerts), they should have a dedicated topic.
- Example:
  - `` `critical-alerts` ``
  - `` `logs-info` ``
  - `` `audit-events` ``

✅ 1.3 Multi-Tenant Considerations

- If supporting multiple customers, consider:
  - **Single topic for all tenants** (simpler but needs filtering).
  - **One topic per tenant** (better isolation, but more complex).
  - **Partitioning by region** for compliance needs.

✅ 1.4 Use Schema Validation

- Use **Pub/Sub Schemas** to enforce JSON/Avro message formats.
- Prevents malformed messages from breaking consumers.

2️⃣ Subscription Design Strategies

A subscription connects a topic to an application consuming messages. The design of subscriptions impacts how messages are processed.

✅ 2.1 Subscription Type: Pull vs. Push

Subscription Type	When to Use	Notes
Pull	High-volume workloads	More control over processing speed, retries, and parallelism.
Push	Real-time processing	Messages are automatically delivered to an HTTP endpoint (e.g., Cloud Run, Cloud Functions).

✅ 2.2 Multiple Subscriptions per Topic

- Multiple consumers can process messages independently.
- Each subscription gets its own copy of the message.
- Use cases:
  - **Audit Logging Subscription** (logs all messages for later analysis).
  - **Real-time Processing Subscription** (e.g., triggers a function).
  - **Analytics Subscription** (sends messages to BigQuery).

Example:

```pgsql
Topic: order-placed
  ├── Subscription 1: send-email-service
  ├── Subscription 2: update-inventory-service
  ├── Subscription 3: analytics-database
```

✅ 2.3 Handling Message Ordering

- By default, Pub/Sub does **not guarantee ordering**.
- If order matters, use:
  - **Ordering Keys** (ensures sequential processing of related messages).
  - **Single Subscriber per Ordering Key** (prevents parallel processing issues).

✅ 2.4 Dead Letter Queues (DLQ)

- Helps catch messages that fail multiple delivery attempts.
- Configure a **Dead Letter Topic** to store undelivered messages.
- Useful for debugging failed events.

```yaml
Topic: payment-events
  ├── Subscription: payment-processor (with DLQ)
├── Dead Letter Topic: payment-failures
```

✅ 2.5 Message Retention & Acknowledgment

- **Default retention**: 7 days (can be extended to 31 days).
- **Auto Acknowledge vs. Manual Acknowledge**
  - Auto: Good for quick processing.
  - Manual: Ensures processing before acknowledging.

✅ 2.6 Fan-out Pattern (One-to-Many Communication)

- Multiple services need to react to the same event.
- Each service gets its own subscription.

```yaml
Topic: user-signups
├── Subscription: welcome-email
├── Subscription: marketing-segmentation
├── Subscription: fraud-detection
```

3️⃣ Advanced Design Patterns

📌 Pattern 1: Filtering Messages by Subscription

- If different consumers only need specific messages, use **Pub/Sub Message Filtering** instead of separate topics.
- Example:

  ```json
  {
    "event_type": "purchase",
    "amount": 100
  }
  ```

- Subscriber 1: Only receives **"purchase"** events.
- Subscriber 2: Only receives events where **`` `amount > 50` ``**.

```yaml
filter: "attributes.event_type = 'purchase' AND attributes.amount > 50"
```

📌 Pattern 2: Routing with Pub/Sub + Cloud Functions

- Route messages dynamically to different systems.
- Example:

- Cloud Function inspects messages and forwards them to different topics.

```vbnet
Topic: transactions
 ├── Cloud Function:
      ├── If "fraud": Send to fraud-alerts topic.
      ├── If "refund": Send to refund-processing topic.
```

📌 Pattern 3: Streaming Analytics Pipeline

- Use Pub/Sub as an event source for **Dataflow**.
- Process and store messages in **BigQuery** for real-time analytics.

```yaml
Topic: sensor-data
 ├── Subscription: stream-processor
     ├── Google Dataflow → BigQuery
```

4️⃣ Monitoring & Security

✅ 4.1 Monitoring & Alerts

- Use **Cloud Logging & Cloud Monitoring** to track message backlog.
- Set alerts for:
  - High unacknowledged message count.
  - Subscription errors.

✅ 4.2 IAM & Access Control

- Restrict who can publish and subscribe.
- Use **IAM roles** like:
  - **`roles/pubsub.publisher`**
  - **`roles/pubsub.subscriber`**
  - **`roles/pubsub.viewer`** (read-only access).

✅ 4.3 VPC Service Controls

- Prevent data exfiltration by securing **Pub/Sub with VPC-SC**.
- Ensures messages stay within the private network.

Key Takeaways

✔️ **One topic per event type** – Keep topics focused on specific events.
✔️ **Use multiple subscriptions** – Allows different services to process messages independently.

✔️ **Ordering Keys** – Use them when message order matters.
✔️ **Dead Letter Topics (DLT)** – Catch undelivered messages for debugging.
✔️ **Filtering** – Reduce message overhead using **subscription filters**.
✔️ **Secure with IAM** – Restrict who can publish and subscribe.
✔️ **Monitor & Scale** – Use **Cloud Monitoring** to prevent message backlog issues.

Would you like a hands-on Terraform or Python example for setting up topics and subscriptions? 🚀