Chapitre 4
Architecture dirigée par les événements (Event-Driven Architecture - EDA)

- Le style d'architecture dirigée
  croissance significative en ter
  ces dernières années.
- Cette adoption croissante s
  capacité à résoudre des proble
  gestion de flux de travail non d
  de systèmes hautement réactifs.
- De plus, l'émergence de nouvelles techniques, d'outils, de frameworks et de services cloud a rendu cette architecture plus accessible et plus facilement exploitable, incitant ainsi de nombreuses équipes à l'adopter pour répondre à des problématiques métier complexes.

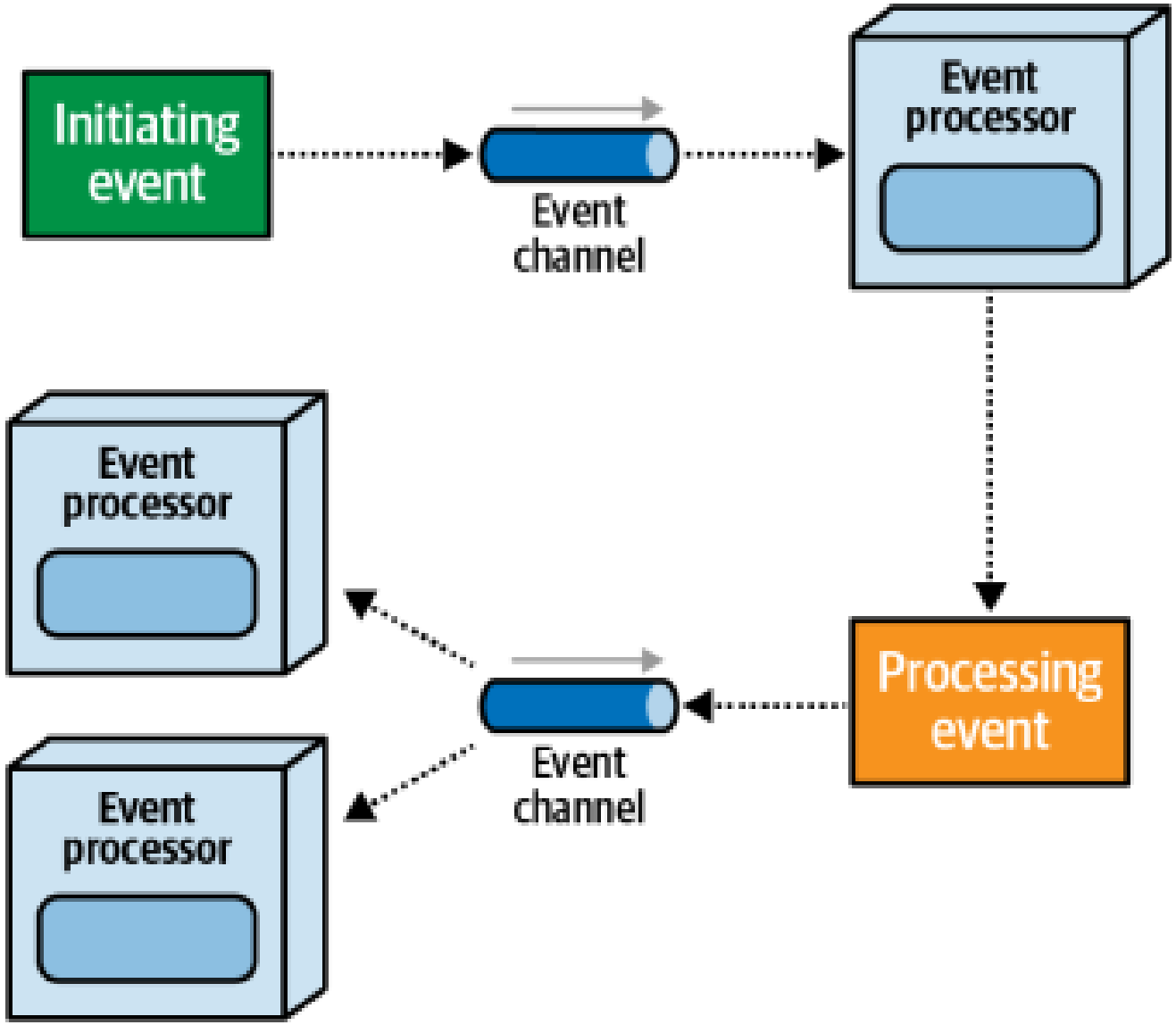
Un **flux de travail non déterministe** est un processus dont :

- l'ordre des étapes n'est pas strictement prédéfini,
- les transitions dépendent d'événements externes imprévisibles,
- plusieurs chemins d'exécution sont possibles selon le contexte.

- Contrairement aux architectures traditionnelles (ex. workflows séquentiels ou orchestrés rigides), l'EDA repose sur des **événements déclencheurs** plutôt que sur un enchaînement prédéfini d'actions.
- **Principe de fonctionnement**
  - Chaque composant **écoute des événements** (event consumers).
  - Lorsqu'un événement survient, il déclenche une action locale.
  - Cette action peut produire de **nouveaux événements**, propageant ainsi le flux.
- On parle souvent de **chorégraphie** plutôt que d'orchestration :
  - **Chorégraphie** : chaque service agit de manière autonome en réaction aux événements.
  - **Orchestration** : un composant central contrôle explicitement le flux.

- Exemple: Dans un système e-commerce :
  - Commande créée → déclenche paiement, préparation, notification.
  - Paiement refusé → annule la commande.
  - Stock insuffisant → déclenche réapprovisionnement ou annulation.
- L'ordre réel dépend des événements reçus, donc il est **non déterministe**.

Topologie


Initiating event
Event channel
Event processor
Event processor
Processing event
Event channel
Event processor

Topologie: le processeur d'évènements

- Un processeur d'événements, souvent désigné aujourd'hui comme un **service**, constitue l'unité fondamentale de déploiement dans une architecture pilotée par les événements.
- Son niveau de granularité peut varier d'une fonction simple, dédiée à une tâche spécifique (par exemple la validation d'une commande), à un processus métier complexe, tel que l'exécution ou le règlement d'une transaction financière.
- Les processeurs d'événements sont capables à la fois de **générer des événements asynchrones** et de **réagir aux événements asynchrones produits par d'autres composants du système**.
  - Dans la plupart des architectures, un processeur d'événements assure simultanément ces deux rôles.

Topologie: l'évènement initiateur

- Un événement initiateur provient généralement d'une source externe au système et déclenche l'exécution d'un processus ou d'un flux de traitement asynchrone.
- Parmi les exemples d'événements initiateurs figurent le passage d'une commande, l'achat d'actions sur un marché financier, la participation à une enchère en ligne ou encore la déclaration d'un sinistre auprès d'une compagnie d'assurance.
- Dans la plupart des cas, cet événement est pris en charge par un service unique qui initie ensuite une chaîne d'événements destinée à assurer son traitement. Néanmoins, un même événement initiateur peut également être consommé par plusieurs services.
- Par exemple, dans le cas d'une enchère en ligne, l'événement correspondant à la soumission d'une enchère peut être traité simultanément par un service d'enregistrement des enchères et par un service de suivi des enchères.

Topologie: l’évènement de traitement

- Un événement de traitement, également appelé événement dérivé, est produit lorsqu’un service subit une modification d’état et diffuse cette information au reste du système.
- La relation entre un événement initiateur et les événements de traitement est généralement de type un-à-plusieurs : un événement initiateur unique peut engendrer plusieurs événements de traitement internes.
- Par exemple, dans le cadre d’un processus de gestion de commande, l’événement initiateur « passer une commande » peut conduire à la génération d’événements tels que « commande enregistrée », « paiement effectué » ou encore « mise à jour du stock ».
- Il convient également de noter que les événements initiateurs sont généralement exprimés sous une forme verbe-nom, tandis que les événements de traitement sont plutôt formulés sous une forme nom-verbe.

Topologie: le canal d’évènements

- Le canal d’événements constitue l’infrastructure de messagerie permettant de transporter les événements au sein d’une architecture pilotée par les événements.
- Il s’agit d’un artefact de communication, tel qu’une file d’attente ou un topic, servant à stocker les événements générés et à les transmettre aux services chargés de les traiter.
- Dans la majorité des cas, les événements initiateurs transitent par un canal de communication de type point à point, généralement implémenté au moyen de files de messages ou de services de messagerie.
- En revanche, les événements de traitement sont le plus souvent diffusés via des mécanismes de type publication–abonnement, reposant sur l’utilisation de topics ou de services de notification.

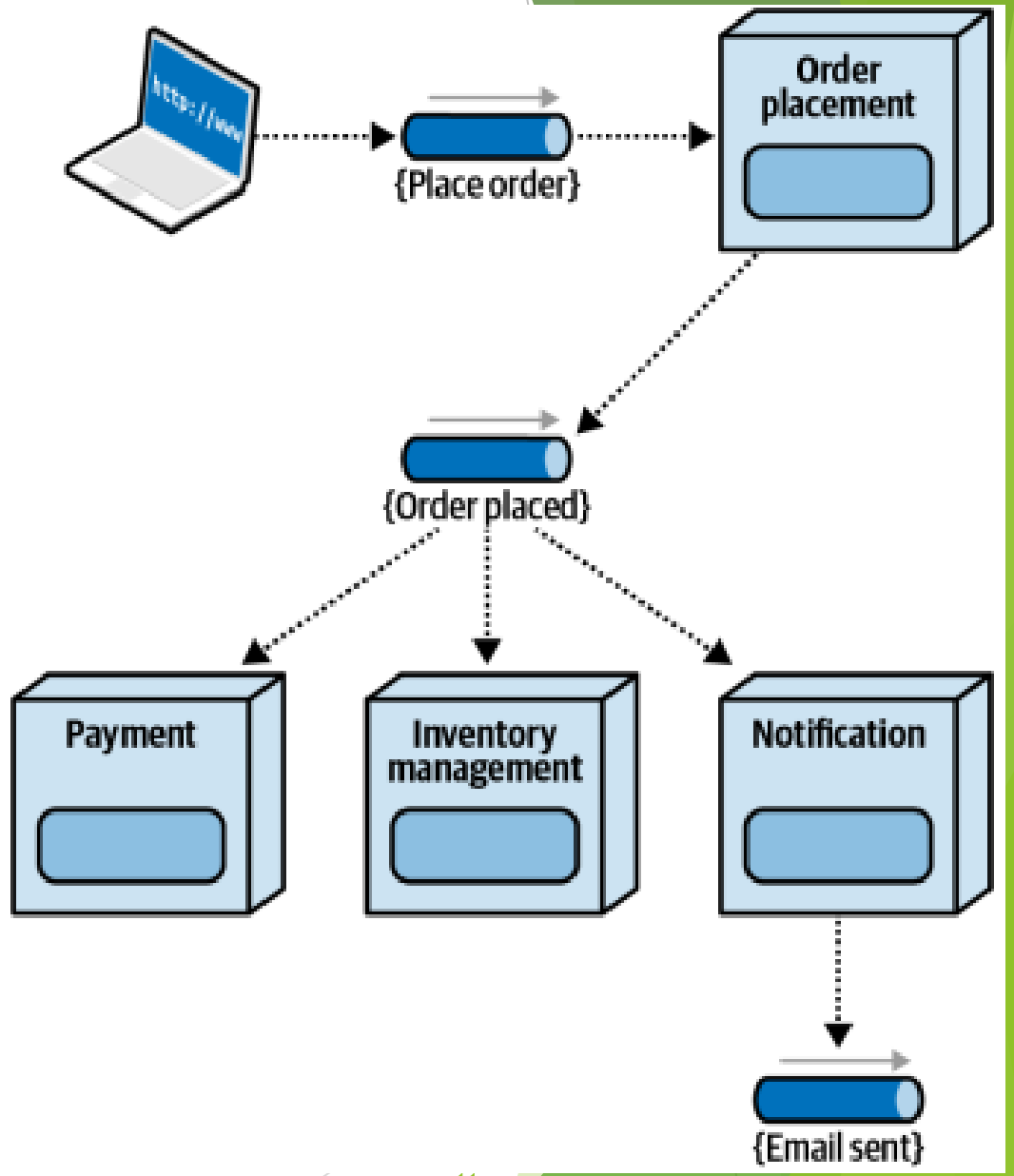
Un topic fonctionne comme un tableau d'affichage virtuel :

- Un producteur (*publisher*) envoie un message ou un événement au topic.
- Plusieurs consommateurs (*subscribers*) peuvent s'abonner au topic et recevoir automatiquement les messages publiés dessus.

Contrairement à une file d'attente (queue) où un message est consommé par un seul destinataire, un topic permet de diffuser le même message à plusieurs services en même temps.

Exemple d'architecture EDA: commande d'un livre

- Considérons l'exemple illustré dans la Figure ci-contre, où un client souhaite commander un exemplaire d'un livre.
- Lorsque le service de placement de commande déclenche l'événement ***Order Placed***, il n'a aucune connaissance des services susceptibles de réagir à cet événement. Cette situation illustre le faible couplage ainsi que le caractère non déterministe propre aux architectures dirigées par les événements, dans lesquelles les producteurs d'événements ne connaissent pas explicitement les consommateurs de ces événements.
- Trois services distincts répondent à l'événement ***Order Placed*** : le service de paiement, le service de gestion des stocks et le service de notification. Chacun de ces services exécute les opérations métier qui lui sont propres et diffuse ensuite des événements de traitement afin de tenir informé le reste du système des modifications ou actions effectuées.



- **Evénement initiateur (Initiating Event) : Place Order**

  Le client passe une commande, ce qui constitue l'événement initiateur Place Order. Cet événement provient de l'extérieur du système et déclenche un flux de traitement asynchrone.

- **Processeur d'événements (Event Processor / Service) : Order Placement Service**

  L'événement initiateur est reçu par le service de placement de commande (Order Placement Service), qui se charge de passer la commande du livre. Le service exécute sa logique métier et génère ensuite un événement de traitement (Processing Event) : Order Placed, afin d'informer le reste du système de l'action réalisée.

- **Événements de traitement et services consommateurs**

  Plusieurs services réagissent à l'événement Order Placed :

  - Payment Service : effectue le paiement et déclenche un événement de traitement Payment Applied.
  - Inventory Management Service : met à jour le stock et déclenche un événement de traitement Inventory Updated.
  - Notification Service : envoie une notification au client et déclenche un événement de traitement Notified Customer.

  Chaque service exécute sa fonction métier propre et informe le système de ses actions via de nouveaux événements de traitement.

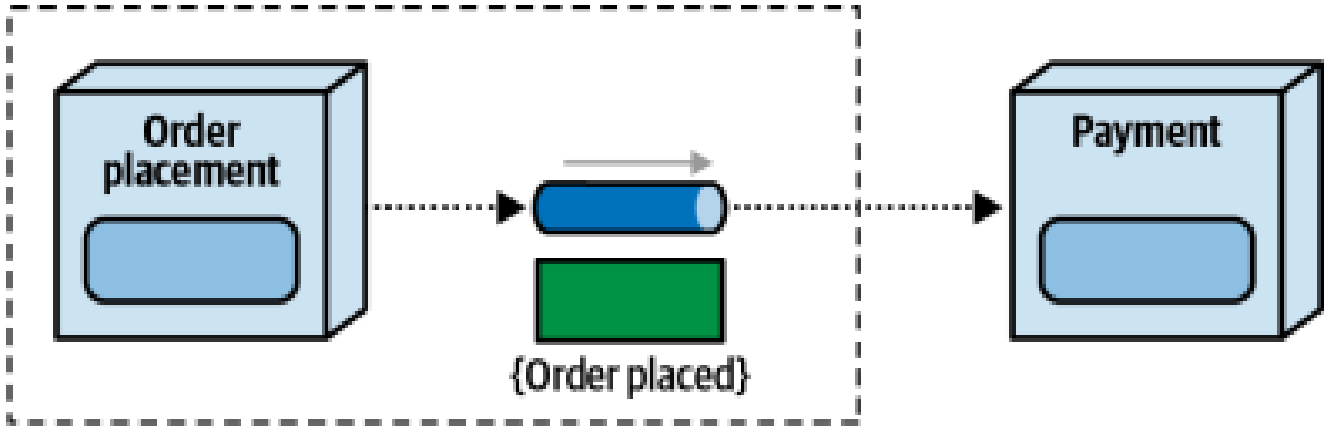
- **Caractère découplé et non déterministe:** Le service de placement de commande **ne sait pas quels services réagiront à l'événement Order Placed**. Cela illustre **le faible couplage et le caractère non déterministe de l'architecture pilotée par les événements**, où **les producteurs d'événements ne connaissent pas explicitement les consommateurs.**
- **Extensibilité et événements "inutilisés"**
  L'événement *Notified Customer*, généré par le service de notification, **n'est pas consommé par d'autres services dans le système actuel**. Toutefois, sa génération **permet d'assurer l'extensibilité de l'architecture**, en fournissant un point d'accroche pour de futurs services, par exemple un service de suivi des notifications. Une **bonne pratique en EDA** consiste donc à **toujours publier les changements d'état des services**, même si aucun autre service n'y réagit immédiatement. Selon la technologie de messagerie utilisée, un événement non consommé **disparaît du topic** ou est **stocké pour un traitement ultérieur**.
- **Canal d'événements (*Event Channel*)**
  Les événements circulent via un **canal de messagerie** :
  - Les **événements initiateurs** transitent généralement par des **canaux point à point (queues)**.
  - Les **événements de traitement** sont souvent diffusés via des **topics (publication-abonnement)** pour permettre à plusieurs services de réagir simultanément.

Event-Driven Vs Message-Driven

- Différence subtile: Les systèmes orientés événements traitent des événements, tandis que les systèmes orientés messages traitent des messages.
- **Différence 1: concerne le contexte de ce qui est transmis au reste du système.**
  - Un événement informe les autres composants d'un changement d'état ou d'une action effectuée.
    - Exemples: des événements incluent des énoncés tels que : « Je viens de passer une commande » ou « Je viens de soumettre une offre pour un article ».
  - Un message constitue une commande ou une requête adressée à un service spécifique.
    - Exemples: de messages incluent : « appliquer un paiement à cette commande », « expédier cet article à cette adresse » ou « fournir l'adresse e-mail du client ».
  - La distinction est ici notable : dans le cas d'un événement, le service qui le déclenche ne sait pas quels services — ni combien — y répondront, tandis qu'un message est généralement destiné à un service unique et identifié (par exemple, le service de paiement).
- **Différence 2: entre un événement et un message réside dans la propriété du canal de communication.**
  - Dans le cas des événements, c'est l'émetteur qui possède le canal d'événements, tandis que pour les messages, c'est le récepteur qui en est propriétaire.

Event-Driven Vs Message-Driven

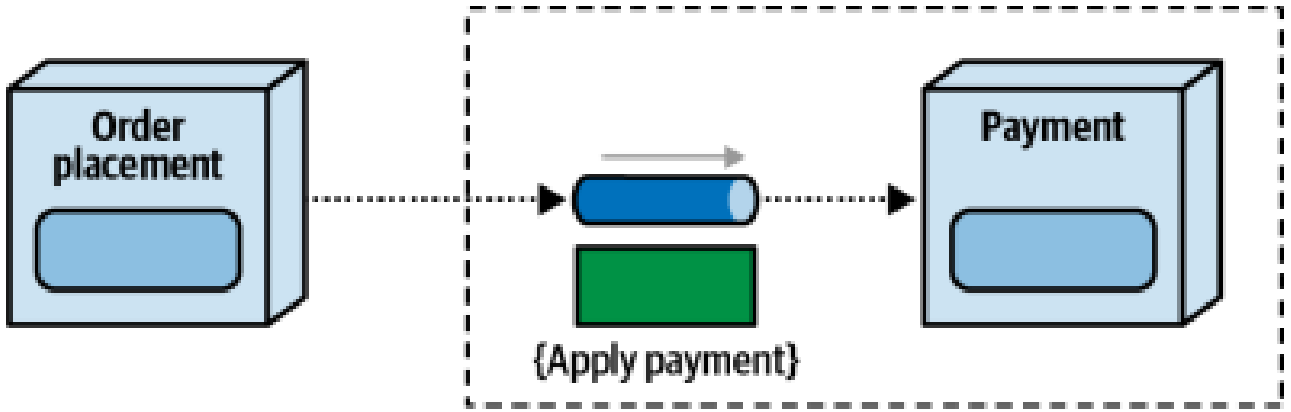
- Cette notion de propriété prend toute son importance lorsque l'on considère le contrat associé à l'événement ou au message.
- Prenons l'exemple illustré à la figure ci-contre, où le service de gestion des commandes (Order Placement) émet un événement « commande passée » (Order Placed), auquel répond le service de paiement (Payment). Dans ce cas, l'émetteur (Order Placement) détient à la fois le canal d'événements et le contrat.



- Toute modification du contrat serait initiée par le service Order Placement, et
- Le service Payment ainsi que tous les autres services réagissant à cet événement devraient se conformer et s'adapter à ces changements.

Event-Driven Vs Message-Driven

- Cependant, dans un système piloté par messages, la situation est exactement inverse : c'est le récepteur qui possède le canal de communication.
- Dans l'exemple, le service de gestion des commandes (*Order Placement*) demande au service de paiement (*Payment*) d'appliquer le paiement sous forme de commande.



- Dans ce cas, le service *Payment* détient à la fois le canal de messages (file d'attente) ainsi que le contrat du message.
- Dans un traitement basé sur les messages, le service *Order Placement* doit se conformer aux modifications du contrat initiées par le service *Payment*.

Event-Driven Vs Message-Driven

- Différence 3: **Le type d'artefact utilisé pour le canal de communication.**
- En général, les systèmes orie… événementiels… messagerie de type *publicatio*… utilisant des *topics* ou des servi… événements.
- En revanche, les systèmes … modèle de communication *po*… (*queues*) ou des services de m…
- Cela ne signifie pas pour autant que les … peuvent pas recourir à une communication point à point, … tains cas, celle-ci est nécessaire pour récupérer des informations spécifiques auprès d'un autre service ou pour contrôler l'ordre ou le moment de survenue des événements au sein du système.

Artefact = composant technique utilisé pour implémenter la communication.
Exemples:
- une file d'attente (queue)
- un topic (publish/subscribe)
- un broker de messages (comme Apache Kafka ou RabbitMQ)
- un service de notification

Rappels

- Un broker de messages est un composant logiciel intermédiaire qui permet à différentes parties d'un système (services, applications, microservices) de communiquer entre elles de manière asynchrone en s'échangeant des messages.
- Un broker agit comme un intermédiaire (middleware) :
  - un producteur (*producer*) envoie un message au broker
  - le broker le stocke temporairement
  - un ou plusieurs consommateurs (*consumers*) récupèrent ce message
- Cela évite que les applications soient directement connectées entre elles.
- Exemple:
  - Apache Kafka
  - RabbitMQ

Rappels

Un broker de messages permet de :

- **Découpler les composants**
  - les services ne se connaissent pas directement
  - ils communiquent via le broker
- **Assurer la communication asynchrone**
  - l'émetteur n'attend pas la réponse immédiate
  - meilleure performance et réactivité
- **Gérer la fiabilité**
  - stockage temporaire (queues, topics)
  - reprise en cas de panne
- **Distribuer les messages**
  - à un ou plusieurs consommateurs
  - équilibrage de charge (load balancing): une technique qui consiste à répartir les requêtes ou les tâches entre plusieurs ressources (serveurs, services, instances) afin d'éviter qu'un seul élément soit surchargé.

- Un topic est un canal logique de communication utilisé dans les systèmes de messagerie (notamment dans Apache Kafka) pour publier et organiser des messages (souvent des événements).
- Un topic est comme une catégorie ou un flux dans lequel les producteurs publient des messages, et auquel les consommateurs s'abonnent.
- Fonctionnement
  - Un producteur envoie un message vers un topic
  - Le broker (ex: Apache Kafka) stocke ce message
  - Plusieurs consommateurs peuvent lire ce message indépendamment

Exemple

- Topic : order-events
- Messages publiés :
  - OrderCreated
  - OrderPaid
  - OrderShipped
- Services abonnés :
  - service facturation
  - service livraison
  - service notification
- Tous reçoivent les événements du même topic.

```
ProducerRecord<String, String> record =
    new ProducerRecord<>("order-events", key, value);
```

- On déclare une variable record de type ProducerRec
- Que signifie ProducerRecord<String, String> ?
- ProducerRecord = objet représentant un message Ka
- <String, String> = types des données :
  - clé (key) → String
  - valeur (value) → String

Donc ici :

- clé = chaîne de caractères
- message = chaîne de caractères (souvent du JSON)

- On crée concrètement le message à envoyer.
- **Paramètres**

"order-events"
nom du **topic**
c'est le canal où le message sera publié

key
identifiant du message
sert à :
choisir la **partition**
garantir l'**ordre des messages liés**

value

```
{
  "eventType": "OrderCreated",
  "orderId": "A100"
}
```

Considérations et analyse

- En raison de sa nature asynchrone et faiblement couplée, l'architecture pilotée par événements se distingue par sa tolérance aux pannes, sa scalabilité et ses hautes performances.
- Elle offre également une excellente extensibilité lors de l'ajout de nouvelles fonctionnalités.
- Cependant, bien que ces caractéristiques soient particulièrement attrayantes, notamment pour les systèmes complexes actuels, il existe de nombreuses raisons de ne pas recourir à une architecture pilotée par événements.

Quand envisager ce style d'architecture?

- L'architecture pilotée par événements constitue le choix privilégié pour les systèmes nécessitant de hautes performances, une forte scalabilité et un niveau élevé de tolérance aux pannes.
- Si la nature de votre traitement métier consiste à réagir à des événements survenant dans et autour du système (plutôt que de répondre à une requête utilisateur), alors ce style d'architecture est pertinent à envisager.
- Écoutez vos parties prenantes métier : utilisent-elles des termes tels que « événement », « déclencheurs » ou « réagir à quelque chose qui se produit » ? Si c'est le cas, il est probable que votre problème métier corresponde à ce style d'architecture.
- Posez-vous également la question suivante : est-ce que je réponds à une requête utilisateur, ou est-ce que je réagis à une action effectuée par l'utilisateur ? Ce sont de très bonnes questions pour déterminer si le problème métier est adapté à ce style d'architecture.

Quand envisager ce style d'architecture?

- L'architecture orientée événements constitue également un choix pertinent lorsque l'on est confronté à des flux de travail complexes et non déterministes, difficiles à modéliser.
- Pendant des décennies, les développeurs ont élaboré des arbres de décision complexes dans le but de prévoir tous les résultats possibles d'un tel flux, échouant toutefois de manière répétée.
- De tels systèmes sont parfois classés sous l'appellation de traitement complexe d'événements (CEP, *Complex Event Processing*), une approche qui est prise en charge de manière native par l'architecture orientée événements.

Quand ne pas envisager ce style ?

- Vous ne devriez pas envisager ce style d'architecture si la majorité de vos traitements repose sur des requêtes.
- Le traitement basé sur les requêtes correspond à la situation classique dans laquelle un utilisateur sollicite des données depuis une base de données ou effectue des opérations CRUD (création, lecture, mise à jour, suppression) sur les entités du système.
- En outre, si l'essentiel de vos traitements requiert une exécution synchrone, obligeant l'utilisateur à attendre l'achèvement du traitement pour une requête donnée, l'architecture orientée événements n'est probablement pas le style d'architecture le plus approprié.

Quand ne pas envisager ce style ?

- Étant donné que tous les traitements sont, en définitive, éventuellement cohérents (*eventual consistency*) dans une architecture orientée événements, ce style d'architecture n'est pas adapté aux problématiques métier exigeant des niveaux élevés de cohérence des données.
- En effet, il existe peu, voire aucune, garantie quant au moment où les traitements seront effectués dans une telle architecture.
- Par conséquent, si vous vous attendez à ce que certaines données soient disponibles à un instant précis, il est préférable de vous orienter vers un autre style d'architecture, tel que l'architecture orientée services, qui contribue davantage à la préservation de la cohérence des données.

Quand ne pas envisager ce style ?

- Une autre raison de renoncer à une architecture orientée événements et d'envisager un style d'architecture différent réside dans le besoin de contrôler le flux de travail ainsi que le moment d'exécution des événements. Ces deux aspects sont particulièrement difficiles à maîtriser dans un contexte de traitement asynchrone des événements.
- À titre d'exemple, considérons la complexité du scénario suivant : les événements A et B doivent être entièrement traités avant de pouvoir déclencher l'événement C ; ensuite, les événements D et E doivent attendre l'achèvement de C, tout en imposant que D commence son traitement avant E. La gestion d'une telle orchestration devient rapidement problématique. Dans ce type de situation, il est préférable de recourir à une architecture orientée services orchestrée ou à des microservices orchestrés, mieux adaptés à la coordination de processus complexes.

Quand ne pas envisager ce style ?

- **La gestion des erreurs** constitue une autre source de complexité qui conduit les équipes à éviter l'architecture orientée événements.
- En effet, en l'absence, dans la plupart des cas, d'un orchestrateur ou d'un contrôleur central du flux de travail, lorsqu'une erreur survient dans un service, il revient à ce service lui-même de tenter de la corriger.
- Par ailleurs, du fait du caractère asynchrone des traitements, d'autres actions ont pu être exécutées entre-temps dans le flux associé à cet événement.

Quand ne pas envisager ce style ?

- Exemple: Supposons qu'un service de traitement des commandes déclenche un événement « commande passée » pour un livre commandé par un client.
- Les services de notification, de paiement et de gestion des stocks réagissent alors simultanément à cet événement.
- Imaginons que les services de notification et de paiement exécutent correctement leurs traitements, tandis que le service de gestion des stocks génère une erreur en raison d'une rupture de stock au moment de la réception de l'événement.
- Comment procéder dans ce cas ?
- Le client a déjà été informé et sa carte bancaire a été débitée, alors même qu'aucun exemplaire du livre n'est disponible pour l'expédition.
- Faut-il annuler le paiement ? Envoyer une nouvelle notification au client ? Suspendre le traitement jusqu'au réapprovisionnement du stock ? Et surtout, quel service doit prendre en charge l'ensemble de cette logique de gestion des erreurs ?
- Ainsi, la gestion des erreurs apparaît comme l'un des aspects les plus complexes de l'architecture orientée événements.

Caractéristiques de l'architecture

Characteristic	Star rating
Partitioning type	Technical
Overall cost	$ $ $
Agility	★★★
Simplicity	★
Scalability	★★★★★
Fault tolerance	★★★★★
Performance	★★★★★
Extensibility	★★★★★