Architecture événementielle

Infrastructure de découplage temporel pour systèmes agentiques

AUTEUR	Jérôme Vetillard
PUBLICATION	27 mai 2026
MAJ	29 mai 2026
SOURCE	Twingital Institute
RAISE	Architecture composite · Auditabilité distribuée
PLANCHES	PL.01 · PL.02

VALIDITY DOMAIN

Architectures agentiques en production où la coordination est multi-composants, asynchrone, et auditable. Hors cadre : agents simples mono-composant synchrones, dont l'EDA serait disproportionnée.

SOMMAIRE

Architecture événementielle

RESUME

L'EDA n'est pas une option d'optimisation pour systèmes agentiques régulés, c'est le paradigme structurant. Examine quatre distinctions binaires (synchrone vs asynchrone, commande vs événement, orchestration vs chorégraphie, état comme source vs événement comme source de vérité), passe en revue les brokers (Kafka, NATS JetStream, Pulsar, Redis Streams, RabbitMQ, EventBridge), et déploie les patterns essentiels (event sourcing, CQRS, Saga, Outbox, ECST, UNS). Pose la thèse que l'EDA est la première condition de l'auditabilité distribuée.

01 L’EDA n’est pas une option d’optimisation

Le débat reste régulièrement posé en termes d’arbitrage. Faut-il déployer un système agentique en architecture synchrone classique, plus simple à concevoir et à dépanner, ou faut-il introduire une infrastructure événementielle, plus complexe mais plus performante à l’échelle ? Cette manière de poser la question est inadéquate dès lors que le système opère en environnement régulé.

En environnement régulé, l’architecture événementielle (event-driven architecture, EDA) n’est pas une option d’optimisation. Elle est le paradigme structurant qui permet à l’architecture composite signable, exposée au cours 8 de la présente série, de tenir son théorème fondamental. Sans EDA, les propriétés de signabilité, de rejouabilité, et d’auditabilité distribuée que la régulation exige deviennent atteignables seulement dans des architectures mono-composants synchrones triviales. Toute architecture agentique qui mobilise plus d’un acteur, qui coordonne des étapes dans le temps, ou qui mémorise un état longitudinal sort de la trivialité et exige une infrastructure événementielle.

> L’EDA n’est pas une option d’optimisation pour systèmes agentiques régulés, c’est le paradigme structurant.

Cette assertion n’est pas une préférence d’architecte. Elle découle de quatre contraintes que la régulation pose en environnement santé, finance, défense, ou tout autre domaine où l’imputabilité d’un acte est exigée par le droit. La première contrainte est l’imputabilité d’un événement à un acteur identifié. La deuxième contrainte est la rejouabilité d’une séquence d’événements pour audit. La troisième contrainte est la résilience aux pannes partielles, qui dans une architecture synchrone produit une rupture de chaîne et donc une perte d’auditabilité. La quatrième contrainte est la non-répudiation, c’est-à-dire l’impossibilité technique pour un acteur d’effacer ou de modifier rétroactivement un événement qu’il a produit.

Ces quatre contraintes sont satisfaites nativement par une architecture événementielle correctement conçue, et difficilement atteignables sans elle. Le présent cours expose les quatre distinctions binaires qui fondent l’EDA, cartographie les brokers disponibles en 2026, présente les cinq patterns essentiels qui structurent les implémentations, et articule le tout à la doctrine d’auditabilité distribuée.

02 Quatre distinctions binaires

L'EDA repose sur quatre distinctions binaires qui doivent être assumées explicitement dès la conception. Toute architecture qui mélange leurs deux côtés crée des ambiguïtés structurelles, et ces ambiguïtés sont précisément les zones où l'auditabilité s'effondre devant un auditeur sérieux.

Synchrone et asynchrone

Premier axe. Un appel est synchrone lorsque l'appelant suspend son exécution jusqu'à recevoir une réponse de l'appelé. Il est asynchrone lorsque l'appelant continue son exécution sans attendre, et que la réponse, lorsqu'elle vient, est traitée à part. Cette distinction est élémentaire mais ses conséquences architecturales sont profondes.

Le synchrone simplifie le raisonnement. La séquence d'opérations est lisible de haut en bas dans le code source, la propagation des erreurs suit la pile d'appels, le débogage est direct. Cette simplicité a un prix. Tout composant en attente est un composant immobilisé. Si l'appelé est lent, l'appelant l'est aussi. Si l'appelé tombe, l'appelant échoue. La latence et la disponibilité se composent négativement le long de la chaîne d'appels.

L'asynchrone élimine ces deux pénalités, au prix d'une complexité accrue de coordination. L'appelant et l'appelé ne sont plus couplés temporellement. Une panne de l'appelé n'arrête pas l'appelant. Une latence importante de l'appelé n'oblige pas l'appelant à attendre. En contrepartie, l'asynchrone exige une infrastructure dédiée pour transporter les messages entre acteurs, et un modèle mental nouveau pour raisonner sur des séquences qui ne sont plus localement ordonnées dans le temps.

Dans un système agentique en production, certains appels peuvent rester synchrones (validation immédiate d'une saisie utilisateur, par exemple), mais tout ce qui dépasse une centaine de millisecondes de latence prévisible doit être traité en asynchrone. La règle pratique : si l'appelé peut être lent, indisponible, ou simplement non urgent, l'asynchrone est la voie sûre.

Commande et événement

Deuxième axe, plus subtil et plus important. Un message peut transporter soit une commande, soit un événement. La distinction sémantique fonde la suite.

Une commande est l'expression d'une intention. Elle est adressée à un destinataire unique, identifié, qui a la responsabilité de l'exécuter ou de la refuser. Sa formulation est à l'impératif (créer une promotion, archiver un cas, fermer un compte). Le destinataire peut rejeter la commande pour raison de capacité, de compétence, de domaine, de gouvernance, ou de port. Une commande refusée est journalisée mais elle ne produit pas d'effet observable au-delà du refus.

Un événement est l'expression d'un fait accompli. Il est diffusé à tous les consommateurs intéressés, sans destinataire unique. Sa formulation est au passé composé (promotion créée, cas archivé, compte fermé). Aucun consommateur ne peut rejeter un événement, parce qu'il décrit un état du monde qui a déjà eu lieu. Le consommateur peut choisir d'en tenir compte ou de l'ignorer, mais il ne peut pas le contester.

> Une commande peut être refusée. Un événement, jamais.

Cette distinction est centrale pour la conception d'un système agentique signable. Les commandes sont les instruments par lesquels les acteurs (humains ou systèmes) exercent leur autorité. Les événements sont les inscriptions factuelles de ce qui a été décidé. Le port de promotion exposé en cours 8 sépare les deux mondes. La proposition issue de l'activity est une commande candidate à exécution. Le franchissement du port produit un événement signé. C'est cet événement, et lui seul, qui devient la trace primordiale dans l'audit.

Orchestration et chorégraphie

Troisième axe. Lorsque plusieurs composants doivent coordonner leurs actions pour réaliser un processus métier (par exemple, qualifier un dispositif médical, instruire un dossier, conduire une consultation augmentée), deux topologies de coordination sont possibles.

L'orchestration confie la coordination à un composant central qui pilote les autres. Le composant orchestrateur connaît la séquence des étapes, invoque chaque participant dans l'ordre voulu, gère les erreurs, et garantit la cohérence globale. Cette topologie est centralisée. Elle est plus facile à comprendre, à observer, et à modifier, parce que la logique de coordination vit dans un seul endroit. Elle est plus fragile à la panne, parce que la défaillance de l'orchestrateur arrête tout le processus.

La chorégraphie confie la coordination à chaque participant. Aucun composant n'a la vue d'ensemble. Chaque acteur réagit à des événements émis par d'autres et émet ses propres événements, sans connaître l'ensemble du processus. La logique métier est distribuée. Cette topologie est résiliente parce qu'aucun composant unique n'arrête tout. Elle est plus difficile à observer, parce que la séquence du processus n'est pas écrite dans un seul endroit mais émerge des interactions entre acteurs.

Dans un système agentique en environnement régulé, les deux topologies coexistent. L'orchestration est utilisée pour les processus métier critiques où la séquence d'étapes est codifiée par la régulation (signature électronique d'un acte clinique, par exemple) et où la traçabilité d'un orchestrateur unique facilite l'audit. La chorégraphie est utilisée pour les processus d'arrière-plan où la résilience prime sur la séquence (mise à jour de cohortes longitudinales, par exemple). La règle pratique : quand la régulation exige un signataire identifié pour la séquence, orchestrer. Quand la régulation tolère une coordination distribuée, chorégraphier.

État et événement comme source de vérité

Quatrième axe, le plus profond architecturalement. Toute architecture stocke des données. La question est de savoir lesquelles constituent la source de vérité, c'est-à-dire la version autoritaire à partir de laquelle toutes les autres vues sont dérivées.

Dans une architecture classique, l'état est la source de vérité. Une base de données relationnelle ou documentaire contient les entités du système (utilisateurs, dossiers, transactions), et chaque écriture met à jour l'état courant. Les anciennes valeurs sont écrasées. L'histoire est perdue, sauf si l'application implémente explicitement des journaux d'audit séparés.

Dans une architecture événementielle correctement déployée, l'événement est la source de vérité. La séquence des événements qui se sont produits dans le système est stockée de manière immuable, en append-only. L'état courant n'est plus la source primordiale. Il est dérivé de la séquence d'événements par projection (replay). Si l'on perd l'état courant, on peut le reconstruire en rejouant les événements. Si l'on perd les événements, on ne peut rien reconstruire.

> L'état dérive des événements. Pas l'inverse.

Cette inversion a une conséquence directe sur l'auditabilité. Dans le modèle classique, l'auditeur lit l'état courant et doit faire confiance aux journaux applicatifs pour reconstituer l'histoire. Ces journaux peuvent être incomplets, illisibles, ou falsifiés. Dans le modèle événementiel, l'auditeur lit la séquence d'événements signés, dont l'intégrité est garantie cryptographiquement. L'état courant n'a plus de statut autoritaire, parce qu'il peut toujours être recalculé.

C'est cette inversion qui rend possible la non-répudiation des actes en environnement régulé. Un événement signé et inscrit dans la séquence ne peut être ni effacé ni modifié sans laisser une trace de la modification elle-même comme nouvel événement. La séquence devient une chaîne d'inscription qui résiste à la révision rétroactive.

03 Cartographie des brokers 2026

Une fois ces quatre distinctions assumées, l'EDA exige une infrastructure technique pour transporter les événements entre composants, et pour les stocker de manière durable. Cette infrastructure est portée par un broker, terme générique qui désigne le composant intermédiaire entre les producteurs et les consommateurs d'événements. Le paysage des brokers est mature en 2026 mais hétérogène, et le choix du broker est une décision structurante qu'il faut prendre à la conception, pas en cours de route.

Apache Kafka

Kafka est la référence historique de l'EDA depuis 2011, projet initial de LinkedIn devenu Apache Software Foundation. Sa proposition de valeur est une combinaison de débit élevé (centaines de milliers de messages par seconde par broker), de rétention long terme (les messages restent disponibles aussi longtemps que la politique de rétention le prévoit, pratiquement indéfiniment), et de garanties d'ordre dans les partitions.

L'architecture Kafka est construite autour de topics partitionnés. Chaque topic est divisé en partitions ordonnées, et les consommateurs lisent les partitions à leur rythme via un offset. Cette architecture donne à Kafka sa propriété distinctive : un consommateur peut rejouer la totalité d'un topic depuis le début, ce qui en fait un système de stockage événementiel autant qu'un broker de messages. Cette propriété est précisément ce qu'exige l'event sourcing présenté plus loin.

Kafka a longtemps souffert de la complexité opérationnelle de son orchestrateur Zookeeper. Le KIP-500 a introduit KRaft (Kafka Raft) qui élimine la dépendance à Zookeeper en internalisant le consensus distribué. Kafka 4.0, sorti en février 2025, a finalisé cette migration et déprécié Zookeeper définitivement. La complexité opérationnelle reste élevée mais a significativement diminué.

Pour un système agentique en environnement régulé, Kafka est la valeur sûre. Sa maturité, sa communauté, et son écosystème (Connect pour intégration, Streams pour traitement, Schema Registry pour gouvernance des schémas) en font le broker par défaut des déploiements industriels exigeants. Son inconvénient principal reste son coût opérationnel pour les déploiements modestes, où une infrastructure plus légère serait suffisante.

Le Schema Registry, en particulier, est une composante critique pour l'EDA en environnement régulé. Il maintient un dépôt versionné des schémas d'événements (typiquement en Avro, Protobuf, ou JSON Schema) avec des règles de com-

patibilité (backward, forward, full) qui empêchent un producteur de publier un événement dont le schéma casserait les consommateurs. Cette discipline de gouvernance des contrats événementiels est l'équivalent technique de la gouvernance des contrats d'API en architecture synchrone, et elle est encore plus exigeante parce qu'un événement publié ne peut pas être révoqué.

Les politiques de rétention Kafka méritent également attention. La rétention par temps (par exemple sept ans pour les événements médicaux conformément aux exigences MDR) est configurable par topic. La rétention par taille (par exemple 100 GB par partition) permet d'éviter les explosions de stockage. La compaction (Kafka cleanup policy compact) conserve uniquement la dernière valeur pour chaque clé, ce qui est utile pour les topics qui représentent un état courant plutôt qu'un journal historique. Pour les topics d'audit auditables sur sept ans, on combine généralement rétention par temps avec archivage external (S3 versionné, Glacier) via Kafka Connect.

L'infrastructure souveraine Kafka en Europe est un sujet stratégique pour les programmes santé. Plusieurs offres existent. OVH propose Kafka managé certifié HDS. Scaleway propose Kafka managé en région française. Confluent Cloud, l'offre managée des créateurs de Kafka, est disponible en région européenne mais reste opérée par une entité américaine, ce qui pose des questions de souveraineté pour certains programmes sensibles. Pour les programmes très sensibles, le déploiement on-premises ou sur cloud souverain reste préférable, au prix d'une charge opérationnelle plus élevée.

NATS JetStream

NATS, projet open-source porté par Synadia, occupe le pôle opposé de Kafka sur l'axe légèreté. NATS Core est un broker pub-sub minimaliste à faible latence, optimisé pour la diffusion de messages éphémères. NATS JetStream, introduit dans NATS 2.2, ajoute la persistance et la rétention au cœur NATS.

NATS JetStream est conçu pour la simplicité opérationnelle. Un cluster NATS de trois nœuds tient dans quelques centaines de mégaoctets de RAM, démarre en quelques secondes, et n'exige pas d'orchestrateur externe. Sa configuration est déclarative, sa gestion est simple. NATS supporte nativement les patterns request-reply, pub-sub, et queueing, ainsi que la rétention durable via JetStream.

NATS gagne du terrain dans les déploiements edge et les architectures distribuées géographiquement. Sa capacité à fédérer plusieurs clusters via NATS Leaf Nodes en fait un candidat sérieux pour les architectures multi-sites, par exemple un programme territorial comme PREDICARE où des nœuds locaux doivent fonctionner en autonomie partielle puis se synchroniser avec un nœud central.

Sa principale faiblesse face à Kafka est l'écosystème, plus modeste. Pour un système agentique qui s'appuiera massivement sur des intégrations tierces (Connect, Schema Registry, Streams), NATS demande plus de développement spécifique. Pour un système plus contenu et où la simplicité opérationnelle prime, NATS est souvent préférable.

Le cas d'usage territorial mérite mention. Un programme comme PREDICARE, qui vise à instrumenter la médecine prédictive sur un territoire avec une fédération de cabinets, hôpitaux, EHPAD, et structures sociales, présente des contraintes architecturales spécifiques. Chaque nœud local doit pouvoir fonctionner en autonomie partielle pendant des périodes d'isolement réseau (panne d'opérateur télécom, congestion en heure de pointe rurale). Les événements produits localement doivent être stockés sur place et synchronisés avec le nœud central dès le rétablissement réseau. La latence en mode dégradé doit rester opérationnelle pour les soins urgents. L'architecture NATS Leaf Node répond précisément à ces contraintes. Un cluster NATS local de petite taille tourne sur chaque site, conserve les événements localement dans son JetStream, et se synchronise au nœud central via un leaf node lien quand le réseau est disponible. Les consommateurs locaux ne subissent jamais la latence du nœud central. Le nœud central reçoit les événements dès qu'ils peuvent être transmis. La structure de fédération est explicite, et chaque leaf node peut être qualifié indépendamment.

Apache Pulsar

Pulsar, initialement développé par Yahoo et donné à Apache en 2018, occupe une position intermédiaire entre Kafka et NATS. Son architecture sépare le service de broker (qui gère les producteurs et consommateurs) du service de stockage (qui assure la persistance via BookKeeper). Cette séparation permet de faire évoluer indépendamment les deux dimensions, et offre une élasticité opérationnelle supérieure à celle de Kafka.

Pulsar supporte nativement les topics persistants et non persistants, le multi-tenancy strict, les fonctions serverless intégrées (Pulsar Functions), et les bridges vers d'autres brokers. Il est particulièrement adapté aux architectures multi-tenant où plusieurs équipes partagent une infrastructure événementielle commune avec des isolations strictes.

Pulsar souffre d'une adoption plus faible que Kafka en Europe, et son écosystème open-source est plus modeste. Pour un programme isolé, Pulsar exige plus de travail interne d'expertise. Pour un programme multi-tenant à grande échelle, son architecture séparée peut justifier l'investissement.

Redis Streams

Redis Streams, introduit dans Redis 5.0 en 2018, transforme Redis (initialement cache et store clé-valeur) en broker événementiel léger. Les streams Redis sont des structures de données append-only, accessibles via les commandes XADD, XREAD, et XLEN, avec consumer groups pour partitionner le traitement.

Redis Streams est idéal pour les architectures où Redis est déjà déployé et où ajouter Kafka serait disproportionné. Sa latence est très faible (millisecondes), sa simplicité opérationnelle est élevée, et son intégration avec les autres structures Redis est immédiate. Sa principale faiblesse est l'absence de rétention long terme native. Les streams Redis sont conçus pour des fenêtres temporelles modestes (quelques heures à quelques jours), pas pour archivage indéfini.

Pour un système agentique en environnement régulé, Redis Streams peut servir d'infrastructure événementielle de premier niveau pour les flux opérationnels, avec une archive à plus long terme (Kafka, S3 versionné) en aval pour la rétention auditable.

RabbitMQ

RabbitMQ implémente le protocole AMQP 0.9.1 et reste l'un des brokers de messages les plus déployés en entreprise. Sa proposition de valeur historique est la fiabilité et la sophistication des routing patterns (exchanges directs, topic, fanout, headers). Le pattern de queueing classique avec acknowledgements et messages durables y est natif et éprouvé.

Sur le terrain de l'EDA strictement événementielle, RabbitMQ a perdu du terrain face à Kafka et NATS. Sa architecture est moins adaptée à la rétention long terme et au replay. Il reste excellent pour le queueing transactionnel et pour les architectures hybrides qui mélangent commandes et événements.

L'introduction des quorum queues (RabbitMQ 3.8) a renforcé sa fiabilité en consensus distribué, ce qui en fait un candidat solide pour les déploiements où la durabilité des messages prime sur le débit.

AWS EventBridge

EventBridge (anciennement CloudWatch Events) est le service événementiel managé d'AWS. Sa proposition de valeur est l'intégration native avec l'écosystème AWS et la disponibilité immédiate sans gestion d'infrastructure. EventBridge supporte les schemas via le Schema Registry, le routing par règles déclaratives, et l'archive pour replay.

Pour un déploiement entièrement AWS, EventBridge réduit le temps de mise en production. Pour un déploiement souverain européen, ou pour un déploiement hybride multi-cloud, EventBridge crée un verrouillage opérateur qui peut être problématique. Le service est aussi limité en débit comparé à Kafka, et son modèle de facturation à l'événement peut devenir coûteux à grande échelle.

EventBridge mérite considération pour les déploiements AWS contenus et pour les intégrations avec des SaaS tiers via les partner event sources. Pour les architectures de souveraineté infrastructure, il est généralement écarté au profit de Kafka ou NATS déployés sur infrastructure souveraine.

04 Six patterns essentiels

Au-delà du choix de broker, l'EDA mobilise six patterns d'implémentation qui structurent la manière dont les événements sont produits, consommés, et reflétés dans l'état du système. Ces patterns sont aujourd'hui canoniques, documentés par des auteurs comme Greg Young, Chris Richardson, Martin Fowler, et Walker Reynolds.

Event sourcing

L'event sourcing, théorisé par Greg Young à partir de 2010, est le pattern qui réalise l'inversion entre état et événement comme source de vérité. Au lieu de stocker l'état courant des entités du domaine et de le mettre à jour, on stocke la séquence complète des événements qui se sont produits, et l'état courant est reconstruit par projection.

Concrètement, pour une entité comme un compte client, on ne stocke pas le solde courant. On stocke la liste ordonnée des événements (ouverture du compte, dépôt, retrait, fermeture). Le solde courant est calculé en parcourant la liste et en appliquant chaque événement. Cette approche a quatre conséquences majeures.

Première conséquence, l'audit est gratuit. L'historique complet de l'entité est la donnée elle-même, pas un sous-produit d'un journal applicatif. La régulation EU AI Act article 17 et 21 CFR Part 11 qui exigent la traçabilité complète des décisions sont nativement satisfaites.

Deuxième conséquence, la rétroactivité est possible. On peut recalculer l'état courant avec une nouvelle règle métier appliquée à des événements historiques, par exemple pour corriger une erreur de calcul rétroactivement ou pour appliquer un nouveau modèle d'inférence à des cas passés sans repartir des données brutes.

Troisième conséquence, la complexité de schéma augmente. Les événements ont leur propre structure typée, et la projection vers l'état courant doit être maintenue cohérente avec l'évolution des types d'événements. La migration de schéma événementiel est un sujet à part entière (upcasting, downcasting, versioning), bien documenté mais à anticiper.

Quatrième conséquence, le stockage croît linéairement avec l'usage. Une entité très active accumule beaucoup d'événements. Les techniques de snapshot (sauvegarde périodique de l'état projeté pour éviter le replay complet à chaque lecture) compensent cette croissance, au prix d'une complexité d'architecture supplémentaire.

L'event sourcing est particulièrement adapté aux systèmes agentiques où la rejouabilité est exigée par la régulation. Pour ToxTwin V3.0, par exemple, la séquence des prédictions de toxicité moléculaire émises par le modèle est stockée comme séquence d'événements signés, et l'état courant (cohorte de molécules qualifiées) est dérivé par projection. Une modification rétroactive du modèle peut être appliquée en rejouant la séquence sur la même cohorte de holdout figée, ce qui mesure l'écart de promotion au sens de l'Article VI RAISE.

Trois techniques pratiques structurent l'usage opérationnel de l'event sourcing.

La première technique est le snapshot. Pour éviter de rejouer toute la séquence d'événements à chaque lecture d'une entité, on stocke périodiquement un snapshot de l'état projeté avec sa version (l'index du dernier événement appliqué). À la lecture, on charge le snapshot le plus récent et on rejoue uniquement les événements postérieurs. Cette technique réduit drastiquement le coût de lecture sans compromettre la propriété de rejouabilité, parce que les snapshots restent dérivables à tout moment depuis les événements.

La deuxième technique est le versioning de schéma événementiel. Les types d'événements évoluent dans le temps. Un événement `PromotionCreated` v1 peut acquérir un champ supplémentaire en v2. Trois stratégies permettent de gérer cette évolution. L'upcasting transforme à la lecture un événement v1 en v2 (en remplissant les nouveaux champs par défaut). Le double-write produit pendant une période transitoire les deux versions de l'événement pour permettre la migration progressive des consommateurs. Le multi-version stockage conserve toutes les versions et fait que chaque consommateur choisit la version qu'il comprend. Le choix dépend du degré de couplage acceptable et de la durée de la migration.

La troisième technique est l'archivage des événements anciens. La régulation peut exiger une rétention longue (sept ans pour les actes médicaux en France), mais l'opération courante n'a généralement besoin que des événements récents. On déplace les événements anciens vers un stockage froid (S3 Glacier, archive WORM) tout en conservant la possibilité de les rejouer en cas d'audit. Cette discipline réduit significativement les coûts de stockage chaud sans compromettre l'auditabilité juridique.

L'event sourcing comporte aussi des anti-patterns connus à éviter. Le premier anti-pattern est de tomber dans le couplage temporel entre événements (un consommateur qui dépend de l'ordre exact d'arrivée d'événements de plusieurs producteurs, sans utiliser de timestamps logiques ou de versions d'agrégat). Le deuxième anti-pattern est de stocker des événements trop riches qui transportent toute la base de données à chaque modification, ce qui annule les bénéfices de découplage. Le troisième anti-pattern est de mélanger événements métier et événements techniques dans le même topic, ce qui rend la gouvernance impossible. La règle pratique : un topic par concept métier, des événements de granularité métier (pas technique), et une gouvernance stricte des schémas.

CQRS

CQRS (Command Query Responsibility Segregation), pattern formalisé par Greg Young et Udi Dahan à partir de 2010, sépare strictement les opérations qui modifient l'état du système (commands) de celles qui le lisent (queries). Les deux côtés peuvent avoir des modèles différents, des stockages différents, et des contraintes de cohérence différentes.

Cette séparation est complémentaire de l'event sourcing mais en est distincte. On peut faire de l'event sourcing sans CQRS (en projetant un seul modèle pour les lectures et les écritures). On peut faire du CQRS sans event sourcing (en utilisant deux bases relationnelles distinctes pour les commands et les queries). Mais le couplage entre les deux patterns est tel qu'ils sont souvent déployés ensemble.

Le principal avantage du CQRS est l'optimisation indépendante des deux côtés. Le côté command peut viser la cohérence stricte et la sécurité (validation, autorisation, signature). Le côté query peut viser la performance et la flexibilité (vues dénormalisées, indexes spécialisés, caches). En contrepartie, la cohérence entre les deux côtés est éventuelle. Une commande validée n'est pas immédiatement visible dans les queries, ce qui exige de raisonner sur la latence de propagation.

Pour un système agentique signable, CQRS isole le port de promotion (côté command, exigeant et lent) du tableau de bord opérationnel (côté query, rapide et tolérant à la latence). Cette séparation rend les deux côtés plus simples à raisonner.

Concrètement, le côté command d'un système comme ToxTwin V3.0 traite les commandes de qualification (création d'une prédiction, validation par le signataire, inscription dans la cohorte de production). Chaque commande passe par des validations métier strictes (cohérence des données moléculaires d'entrée, droits du signataire, état autorisé de la cohorte cible), produit zéro ou un événe-

ment signé, et s'inscrit dans l'audit. Le côté query expose les vues opérationnelles utiles (tableau de bord des prédictions du jour, distribution des écarts de promotion, suivi du taux de refus par catégorie). Ces vues sont construites par projection asynchrone à partir des événements émis côté command, sont mises à jour avec une latence de quelques secondes à quelques minutes, et tolèrent cette latence parce qu'elles servent à du pilotage et non à des décisions individuelles. La séparation des deux côtés permet de scaler chacun selon ses propres exigences. Le côté query peut servir mille fois plus de lectures que d'écritures sans charger le côté command. Le côté command peut garantir une cohérence stricte sans payer le coût de l'optimisation des lectures.

Saga

Le pattern Saga, théorisé par Hector Garcia-Molina et Kenneth Salem en 1987 et largement diffusé par Chris Richardson dans le contexte des microservices, traite des transactions distribuées longues. Une saga est une séquence de transactions locales chacune effectuée par un service différent, avec des transactions de compensation qui annulent les transactions précédentes en cas d'échec d'une étape ultérieure.

Le pattern Saga est essentiel dans les architectures agentiques où un processus métier traverse plusieurs composants. Par exemple, qualifier un acte médical augmenté par IA peut impliquer une vérification d'identité, une consultation du dossier patient, une inférence du modèle, une validation par un signataire, et l'inscription dans le dossier patient. Chaque étape est une transaction locale. Si la validation par le signataire échoue, les étapes antérieures doivent être compensées (libération de la lecture du dossier, annulation de l'inférence, libération de la réservation).

Deux variantes de saga coexistent. La saga orchestrée centralise la logique de compensation dans un orchestrateur unique, plus simple à raisonner. La saga chorégraphiée distribue la compensation à chaque acteur, plus résiliente mais plus difficile à comprendre. Le choix dépend des mêmes critères que le choix entre orchestration et chorégraphie exposé plus haut.

Un cas pratique illustre la mécanique. Considérons une plateforme de qualification d'acte clinique augmenté par IA, semblable à un module de PREDICARE. Le processus métier comporte cinq étapes. Première étape, vérification d'identité du clinicien et du patient via un service IAM (Identity and Access Management). Deuxième étape, réservation d'un slot de calcul GPU pour l'inférence du modèle. Troisième étape, lecture du dossier patient et préparation du contexte d'infé-

rence. Quatrième étape, exécution de l'inférence et production d'une proposition. Cinquième étape, présentation au signataire humain pour validation et inscription dans le dossier patient.

Si la cinquième étape échoue (refus du signataire, indisponibilité réseau, timeout), une saga compense les quatre étapes précédentes. La réservation GPU est libérée, le contexte d'inférence est purgé du cache temporaire, la lecture du dossier patient est journalisée comme consultation sans suite, et l'identification IAM est marquée comme close. Chaque compensation est elle-même un événement signé inscrit dans la séquence, ce qui maintient la cohérence de l'audit trail. Aucun état intermédiaire ne reste orphelin dans le système.

La distinction subtile entre saga orchestrée et chorégraphiée se manifeste ici dans la localisation de la logique de compensation. En saga orchestrée, un workflow durable central pilote les cinq étapes et émet les commandes de compensation au moment où il détecte un échec. En saga chorégraphiée, chaque service réagit aux événements émis par les autres et déclenche sa propre compensation lorsqu'il observe un événement d'annulation downstream. Pour un programme régulé qui exige un signataire identifié pour la séquence globale (orchestrateur identifié dans le dossier de qualification), l'orchestration est généralement préférable. Pour un système distribué multi-acteurs sans signataire unique de la séquence (interactions inter-services à grande échelle), la chorégraphie est plus naturelle.

Une nuance importante. La compensation n'est pas un rollback transactionnel ACID. Elle est une opération métier qui annule l'effet observable des opérations précédentes, mais sans capacité à effacer l'inscription de ces opérations dans l'audit trail. Une consultation de dossier patient compensée reste inscrite comme consultation. Une réservation GPU libérée reste inscrite comme réservation puis libération. Cette discipline est essentielle pour l'auditabilité : chaque action et chaque compensation sont des faits qui ont eu lieu, et qui restent visibles à l'audit.

Outbox

Le pattern Outbox, formalisé par Chris Richardson, résout un problème pratique récurrent. Lorsqu'un composant doit à la fois mettre à jour son état local (dans une base de données) et publier un événement (dans un broker), il y a un risque d'incohérence si l'une des deux opérations réussit et l'autre échoue. Si la mise à jour réussit mais que la publication échoue, l'état local et le monde extérieur sont désynchronisés. Si la publication réussit mais que la mise à jour échoue, on a publié un événement qui ne reflète pas la réalité.

L'Outbox résout cette incohérence en regroupant les deux opérations dans une seule transaction locale. Au lieu de publier directement l'événement dans le broker, le composant écrit l'événement dans une table outbox de sa propre base de données, dans la même transaction que la mise à jour de l'état. Un processus distinct (souvent appelé relais ou poller) lit la table outbox et publie effectivement les événements dans le broker.

L'avantage est double. La cohérence est garantie par la transaction locale. La publication dans le broker peut échouer transitoirement sans risque d'incohérence, parce que les événements restent dans l'outbox jusqu'à publication effective. Le coût est une légère latence supplémentaire et un peu de mécanique de relais.

Le pattern Outbox est particulièrement précieux pour les architectures qui exigent une cohérence forte entre l'état persistant et la diffusion événementielle, ce qui est typiquement le cas en environnement régulé.

Exemple concret. Un module clinique qui inscrit la décision d'un signataire dans le dossier patient doit à la fois mettre à jour la base de données du dossier (état) et émettre l'événement signé correspondant dans le broker (audit trail). Sans Outbox, une panne réseau entre ces deux opérations peut produire trois pathologies : inscription au dossier sans événement dans l'audit (perte de traçabilité), événement dans l'audit sans inscription au dossier (audit incohérent), ou double inscription si une retry est mal gérée. Avec Outbox, la mise à jour du dossier et l'écriture de l'événement dans la table outbox du dossier sont commitées dans une seule transaction PostgreSQL. Un poller distinct lit la table outbox et publie les événements dans Kafka. Si Kafka est temporairement indisponible, les événements restent dans l'outbox et seront publiés au retour. Si la publication réussit, l'événement est marqué comme publié dans l'outbox et peut être purgé après une période de rétention. Cette mécanique garantit que ce qui est inscrit au dossier patient finit toujours dans l'audit trail, sans perte ni doublon, indépendamment des défaillances de l'infrastructure de messagerie.

ECST (Event-Carried State Transfer)

Le pattern ECST (Event-Carried State Transfer), nommé par Martin Fowler dans son catalogue sur l'EDA et popularisé par des praticiens comme Adam Warski, traite du contenu des événements lui-même. La question est de savoir si un événement doit être minimal (juste l'identifiant de l'entité et le type de changement, par exemple `PatientUpdated{id: 42}`) ou enrichi (l'identifiant plus l'ensemble des données pertinentes, par exemple `PatientUpdated{id: 42, name: "Dupont", birthdate: "1972-03-15", ...}`).

L'événement minimal force les consommateurs à interroger le producteur pour obtenir les détails, ce qui réintroduit un couplage temporel. L'événement enrichi (ECST) transporte l'état utile avec lui, ce qui permet aux consommateurs de fonctionner sans interroger le producteur, et donc de rester découplés même si le producteur est indisponible.

L'ECST a un coût en bande passante et en duplication de données, mais il renforce l'autonomie des consommateurs et la résilience globale du système. Pour un système agentique distribué, l'ECST est souvent le bon choix, à condition de gouverner soigneusement la taille des événements et les questions de cohérence éventuelle entre la copie portée par l'événement et la donnée d'origine chez le producteur.

UNS (Unified Namespace)

L'UNS (Unified Namespace), pattern popularisé par Walker Reynolds dans le mouvement Industry 4.0 à partir de 2020, prolonge et systématise l'ECST en y ajoutant trois disciplines structurelles. Le concept lui-même n'est pas nouveau, il s'appuie sur la norme ANSI/ISA-95 (devenue IEC/ISO 62264) qui hiérarchise les niveaux d'intégration entre ERP, MES, et systèmes de contrôle industriels. Reynolds en a fait le pattern de référence pour les architectures industrielles distribuées, et son adoption progresse rapidement au-delà du périmètre manufacturier initial.

Première discipline, la hiérarchie de noms structurée. Toutes les entités du système ont une adresse canonique dans un namespace arborescent unique. La hiérarchie ISA-95 classique organise en cinq niveaux (Enterprise, Site, Area, Cell, Equipment), mais le pattern est broker-agnostique et la hiérarchie est adaptable au domaine. Pour un système agentique en santé territoriale comme PREDICARE, la hiérarchie pourrait être Programme, Territoire, Site, Modalité, Sujet, et chaque entité instanciée aurait sa coordonnée précise dans cette structure.

Deuxième discipline, le principe de single source of truth. Pour chaque donnée du système, exactement un nœud du namespace est l'autorité productrice. Les autres consommateurs s'abonnent à ce nœud, ils ne dupliquent pas son contenu dans leur propre namespace. Cette discipline élimine les divergences silencieuses entre vues concurrentes de la même donnée, qui sont l'une des sources d'incohérence les plus fréquentes en architecture distribuée naïve.

Troisième discipline, l'état courant porté par le namespace lui-même. Le broker maintient le dernier message connu sur chaque topic, ce qui permet à un nouveau consommateur d'obtenir immédiatement l'état courant sans interroger le

producteur. En MQTT, c'est la fonction des retained messages combinée à la spécification Sparkplug B (Cirrus Link Solutions, adoptée par Eclipse Foundation) qui standardise les payloads industriels. En Kafka, c'est la compaction (cleanup.policy=compact) qui conserve la dernière valeur par clé. En NATS, c'est la fonction Key-Value de JetStream.

> Le namespace est l'architecture rendue adressable.

Pour un système agentique en environnement régulé, l'UNS apporte une valeur architecturale spécifique. Le namespace devient l'épine dorsale de l'audit en temps réel. Chaque modèle déployé, chaque cohorte de référence, chaque signataire habilité, chaque workflow durable, chaque activity encapsulée, possède une adresse canonique dans le namespace, et son état courant est lisible directement par tout composant autorisé. L'auditeur n'a pas besoin de remonter une séquence d'événements pour connaître l'état présent : il interroge le namespace. La séquence événementielle reste disponible pour l'audit historique, le namespace donne l'audit instantané.

L'UNS articule naturellement les architectures industrielles (capteurs IIoT, jumeaux numériques d'équipements, MES) et les architectures agentiques en santé régulée. Cette continuité fait de l'UNS le pattern de référence pour les programmes hybrides où systèmes physiques instrumentés et systèmes agentiques en environnement régulé doivent coopérer dans un même cadre architectural. Les déploiements industriels typiques utilisent HiveMQ ou Mosquitto comme broker MQTT central. Les déploiements agentiques peuvent utiliser Kafka avec compaction ou NATS avec JetStream Key-Value. Dans les deux cas, c'est la discipline du namespace qui structure l'architecture, pas le broker spécifique.

05 L’EDA comme première condition de l’auditabilité distribuée

Les quatre distinctions binaires et les cinq patterns convergent vers une thèse architecturale unique. Dans un système agentique en environnement régulé, l’EDA est la première condition de l’auditabilité distribuée, parce qu’elle réalise techniquement les quatre propriétés que la régulation exige : imputabilité, rejouabilité, résilience, non-répudiation.

L’imputabilité est portée par la signature des événements au moment où ils franchissent le port de promotion. Chaque événement est inscrit avec l’identité cryptographique de son émetteur ou de son signataire humain, ce qui rend l’attribution non contestable.

La rejouabilité est portée par le stockage append-only des événements et par les techniques de projection. Un auditeur peut rejouer une séquence d’événements pour reconstituer un état historique, vérifier qu’une décision a bien été prise selon les règles applicables à l’instant t, et comparer le comportement du modèle entre deux versions.

La résilience est portée par le découplage temporel entre acteurs. Un acteur indisponible n’arrête pas la chaîne, parce que les événements sont stockés par le broker et seront consommés dès que l’acteur revient. La défaillance d’un composant ne propage pas à toute la chaîne, contrairement aux architectures synchrones.

La non-répudiation est portée par l’immutabilité du stockage événementiel et par la signature des événements. Un acteur ne peut ni effacer ni modifier un événement qu’il a émis sans laisser une trace de la modification elle-même comme nouvel événement.

> Ce qui n’est pas un événement signé n’a pas eu lieu.

Cette compression formule résume la doctrine. Elle articule directement la thèse du cours 8 (un acte n’existe juridiquement que s’il est signable) à la matière du présent cours (la signabilité est portée techniquement par l’événement signé inscrit dans le journal d’audit immuable).

06 Articulation à l’architecture composite signable

L’EDA n’est pas une couche posée par-dessus une architecture quelconque. Elle est l’épine dorsale de la couche déterministe de l’architecture composite signable. La couche non déterministe (LLM, modèles d’inférence, propositions issues des activities) produit ses sorties dans le cadre d’une activity encapsulée. La couche déterministe (workflow durable, port de promotion, journal d’audit, refusal taxonomy) est entièrement portée par des événements signés transitant par le broker.

Cette articulation explique pourquoi le choix du broker n’est pas un détail d’infrastructure mais une décision architecturale fondamentale. Un broker qui ne supporte pas la rétention long terme (NATS Core sans JetStream) ne permet pas la rejouabilité. Un broker qui ne supporte pas l’ordre strict des messages (certains brokers pub-sub naïfs) ne permet pas la reconstitution séquentielle des décisions. Un broker qui ne supporte pas le multi-tenancy strict (Kafka sans configuration spécifique, EventBridge sans isolation) ne permet pas la séparation des audit trails entre clients ou entre programmes.

Le broker, dans une architecture composite signable, n’est pas un middleware de messagerie. Il est l’infrastructure d’inscription doctrinale. Son choix doit être fait à la conception, en fonction des exigences réglementaires applicables, et il est très coûteux à changer en cours de route. Pour un système agentique en santé régulée européenne, le Twingital Institute recommande Kafka en infrastructure souveraine HDS, avec NATS JetStream comme option pour les déploiements territoriaux distribués comme PREDICARE, où la fédération multi-sites est plus structurante que la rétention long terme centralisée.

L’EDA s’articule également à la doctrine de mémoire agentique exposée en cours 3. La mémoire long terme d’un système agentique est alimentée par projection à partir des événements signés du journal d’audit. C’est cette discipline de projection (et non l’écriture directe par l’agent dans la base mémoire) qui rend la mémoire elle-même auditable et conforme aux exigences RGPD article 17 sur le droit à l’oubli, parce qu’effacer une donnée personnelle revient à inscrire un événement d’effacement signé et à re-projeter la mémoire.

07 L’EDA et la conformité régulatoire 2026

L’EDA ne sert pas la conformité par accident. Elle la sert parce qu’elle satisfait quatre exigences réglementaires spécifiques qui sont aujourd’hui posées dans plusieurs cadres convergents.

L’EU AI Act article 17 sur le système de management des risques exige un processus continu et itératif de gestion des risques sur l’ensemble du cycle de vie du système d’IA. Cette continuité suppose la capacité de tracer chaque décision du système dans le temps, de mesurer l’évolution des indicateurs de risque, et de remonter la chaîne causale d’une décision problématique vers ses entrées. L’EDA réalise cette continuité techniquement. Chaque décision est un événement signé inscrit dans une séquence ordonnée. La mesure de l’écart de promotion au sens de l’Article VI RAISE est dérivée directement de cette séquence. La remontée causale d’une décision vers ses entrées est portée par les corrélateurs d’événements (correlation_id, causation_id) qui lient chaque événement à ses prédécesseurs dans la chaîne.

Le 21 CFR Part 11 de la FDA américaine exige que les enregistrements électroniques en environnement régulé satisfassent à six propriétés : authenticité, intégrité, lisibilité, non-répudiation, horodatage fiable, et identité du signataire. L’EDA satisfait ces six propriétés nativement lorsqu’elle est correctement déployée. L’authenticité est portée par la signature cryptographique de chaque événement. L’intégrité est garantie par l’append-only du stockage événementiel. La lisibilité est portée par les schémas versionnés et la rétention durable. La non-répudiation est portée par l’immutabilité du stockage. L’horodatage fiable est porté par les timestamps signés au moment de l’inscription. L’identité du signataire est portée par l’identifiant cryptographique inscrit dans chaque événement signable.

Le règlement MDR (UE) 2017/745 pour les dispositifs médicaux exige la surveillance post-commercialisation (post-market surveillance, PMS) et la production périodique de PSUR (Periodic Safety Update Report) qui documente le comportement du dispositif en production. Pour un dispositif à base d’IA, le PSUR exige des indicateurs de performance dans le temps. L’EDA fournit la matière première de ces indicateurs. La séquence d’événements signés contient toutes les décisions du système, leur résultat, les écarts de promotion, les refus typés. Les indicateurs PSUR sont des projections de cette séquence selon des règles métier explicites.

Le règlement (UE) 2016/679 RGPD article 17 sur le droit à l'oubli pose un problème spécifique à l'event sourcing. Effacer une donnée personnelle ne peut pas se faire en supprimant les événements qui la contiennent, parce que cela viole l'append-only et casse la rejouabilité. La pratique courante est le crypto-shredding. Au moment de l'inscription d'un événement contenant des données personnelles, le contenu sensible est chiffré avec une clé spécifique au sujet (per-subject key). La séquence d'événements stocke le contenu chiffré, et la clé est stockée séparément dans un vault. Lors d'un droit à l'oubli, on n'efface pas les événements, on détruit la clé. Les événements deviennent illisibles pour le sujet concerné mais la séquence reste intacte, ce qui préserve la rejouabilité des autres aspects du système. Le RGPD considère cette pratique comme conforme dès lors que la clé est effectivement détruite et que la donnée chiffrée devient cryptographiquement irrécupérable.

Au-delà de ces quatre cadres principaux, l'EDA satisfait également les exigences d'identification électronique du règlement eIDAS (910/2014) lorsque la signature des événements est portée par un dispositif de création de signature qualifié, et les exigences d'auditabilité des futurs cadres en construction (FRIA de l'article 27 EU AI Act, déclarations de conformité IMDA Singapour, audit trails du CSA Agentic Trust Framework).

L'articulation EDA et conformité régulatoire pose une exigence opérationnelle souvent négligée. L'infrastructure événementielle elle-même (broker, stockage) doit être qualifiée au même niveau que le système agentique qu'elle supporte. Un broker Kafka non qualifié HDS ne peut pas servir d'épine dorsale à un dispositif médical SaMD. Une infrastructure de stockage non auditée selon les exigences de la régulation applicable ne peut pas porter l'audit trail. Cette exigence de qualification de l'infrastructure de support est l'une des raisons pour lesquelles le choix de l'infrastructure souveraine (cloud HDS, déploiement on-premises) est aussi structurant que le choix du broker.

08 Limites assumées

Quatre limites de la doctrine EDA exposée ici doivent être assumées explicitement.

Premièrement, l'EDA introduit une complexité de raisonnement et d'opération qui n'est pas justifiée pour les architectures triviales. Un agent mono-composant qui répond à des requêtes synchrones sans état longitudinal n'a pas besoin d'EDA. Forcer une infrastructure événementielle dans ce cas serait disproportionné. La règle pratique du Twingital Institute : l'EDA s'impose dès que le système coordonne plus de deux composants, ou dès qu'il maintient un état longitudinal sur plus de trente jours, ou dès qu'il est en environnement régulé.

Deuxièmement, le paysage des brokers évolue. Les versions citées dans le présent cours (Kafka 4.0, NATS 2.x, Redis 5.0+) sont datées de mai 2026. Les fonctionnalités et les arbitrages comparatifs peuvent évoluer dans les 18 prochains mois. La cartographie demande à être révisée tous les deux trimestres, en cohérence avec la matrice de décision frameworks exposée au cours 2.

Troisièmement, l'EDA suppose une discipline organisationnelle qui ne se décrète pas. Les équipes habituées au synchrone doivent apprendre à raisonner sur des séquences distribuées, à concevoir des compensations de saga, à gouverner les schémas d'événements, à versionner les contrats événementiels. Cet apprentissage prend du temps et de l'investissement, et son absence est la première cause d'échec des migrations vers l'EDA. Le Twingital Institute recommande une bascule progressive (un domaine métier à la fois) plutôt qu'une réécriture en bloc.

Quatrièmement, l'EDA ne dispense pas de toute architecture synchrone. Certaines opérations restent fondamentalement synchrones (validation immédiate d'une saisie, recherche en base, requête simple). Le bon design articule les deux régimes au lieu de les opposer. L'EDA est la structure pour la coordination longue distance et auditable, la requête synchrone reste appropriée pour les interactions courtes et locales.

Le présent cours expose la doctrine et l'infrastructure. Le cours 5 traite de la durable execution comme réalisation technique de la couche workflow au-dessus des événements. Le cours 6 traite des protocoles d'interopérabilité (MCP, A2A, ACP) comme couche transport des architectures distribuées. Ensemble, ces trois cours constituent l'épine dorsale technique de l'architecture composite signable.

A propos de cette serie

Ce cours appartient a la serie Architectures agentiques 2026 publiee par le Twingital Institute. La serie aborde l'architecture des systemes agentiques en environnement regule a partir du Framework RAISE.

L'ensemble de la serie, les planches d'architecture interactives, et la matrice de decision frameworks sont accessibles sur le site de l'Institut : twingital-ventures.com/fr/cours/ https://twingital-ventures.com/fr/cours/

TITRE	Architecture événementielle
SOUS-TITRE	Infrastructure de découplage temporel pour systèmes agentiques
AUTEUR	Jérôme Vetillard
SOURCE	Twingital Institute
SERIE	Architectures agentiques 2026 · Cours 4 / 8
PUBLICATION	27 mai 2026
MAJ	29 mai 2026
RAISE	Architecture composite · Auditabilité distribuée
PLANCHES	PL.01 · PL.02
MOTS-CLES	EDA · event-driven · Kafka · NATS · event-sourcing · saga · outbox · UNS
URL	https://twingital-ventures.com/fr/cours/2026-05-cours-4-eda-decouplage-temporel-fr/