

# Rapport Architecture ARC42 - Système Microservices avec Sagas

---

## Laboratoires 6 et 7 - Architecture Saga Orchestrée et Événementielle

**Auteur** : Talip Koyluoglu

**Cours** : LOG430 - Architecture Logicielle

**Projet** : Évolution d'une architecture microservices DDD vers des patterns de saga avancés

**Focus** : **Labo 6** (Saga Orchestrée Synchrones) et **Labo 7** (Architecture Événementielle avec Saga Chorégraphiée)

---

## Table des Matières

### 1. Introduction et Objectifs

- 1.1 Aperçu du Système - Architecture microservices avec patterns saga
- 1.2 Objectifs Architecturaux - Orchestration vs Chorégraphie
- 1.3 Stakeholders - Équipe développement, utilisateurs métier

### 2. Contraintes

- 2.1 Contraintes Techniques - Django, Redis, Kong, PostgreSQL
- 2.2 Contraintes Organisationnelles - Patterns saga, observabilité
- 2.3 Contraintes Règlementaires - Audit, traçabilité distribuée

### 3. Contexte

- 3.1 Contexte Métier - E-commerce avec transactions distribuées
- 3.2 Évolution Architecturale - Saga orchestrée vers chorégraphiée
- 3.3 Problématiques Saga - Cohérence, compensation, observabilité

### 4. Stratégie de Solution

- 4.1 Approche Architecturale - Dual Pattern Saga
- 4.2 Patterns Architecturaux - Orchestration + Event Sourcing + CQRS
- 4.3 Analyse Patterns Saga - Centralisé vs Décentralisé

### 5. Vue des Blocs de Construction

- 5.1 Vue Implémentation - Architecture saga orchestrée et chorégraphiée
- 5.2 Vue Déploiement - Infrastructure Redis, workers, Event Store
- 5.3 Vue Logique - Modèles événementiels et machine d'état
- 5.4 Vue Cas d'Utilisation - Checkout orchestré vs chorégraphié

### 6. Vue d'Exécution

- 6.1 Processus Saga Orchestrée - Machine d'état centralisée

- **6.2** Processus Saga Chorégraphiée - Coordination événementielle
- **6.3** Observabilité et Métriques - Captures Grafana

## 7. Vue de Déploiement

- **7.1** Configuration Docker Compose - Services saga + infrastructure
- **7.2** Architecture Redis Streams - Event Bus et Event Store

## 8. Concepts Transversaux

- **8.1** Patterns Saga - Orchestration vs Chorégraphie
- **8.2** Event Sourcing et CQRS - Persistance événementielle
- **8.3** Compensation et Rollback - Gestion échecs distribuées
- **8.4** Observabilité Distribuée - Monitoring saga multi-pattern

## 9. Décisions

- **9.1** ADR-001 - Redis Streams pour Event Store et CQRS
- **9.2** ADR-002 - Saga Chorégraphiée pour coordination décentralisée
- **9.3** Alternatives Écartées - 2PC, orchestration centralisée exclusive

## 10. Scénarios de Qualité

- **10.1** Cohérence - Sagas distribuées et compensation
- **10.2** Performance - Latence orchestrée vs événementielle
- **10.3** Résilience - Tolérance pannes dans patterns saga
- **10.4** Observabilité - Traçabilité transactions distribuées

## 11. Risques et Dette Technique

- **11.1** Risques Techniques - Complexité saga, cohérence événementielle
- **11.2** Dette Technique - Dualité patterns, Event Store simple

## 12. Glossaire

- Termes saga, événementiel, CQRS, compensation

---

# 1. Introduction et Objectifs

## 1.1 Aperçu du Système

Ce rapport documente l'implémentation de **deux patterns de saga complémentaires** dans une architecture microservices e-commerce :

- **Labo 6 : Saga Orchestrée Synchron**e avec coordination centralisée
- **Labo 7 : Architecture Événementielle** avec saga chorégraphiée, Event Sourcing et CQRS

L'objectif est de démontrer les avantages et inconvénients de chaque approche pour la gestion de transactions distribuées dans un contexte e-commerce.

1.2 Objectifs Architecturaux

Objectifs Labo 6 - Saga Orchestrée

- **Coordination centralisée** : Service orchestrateur unique gérant machine d'état
- **Transactions distribuées** : Séquence contrôlée d'appels inter-services
- **Compensation automatique** : Rollback programmé en cas d'échec
- **Observabilité centralisée** : Métriques et logs saga unifiés

Objectifs Labo 7 - Architecture Événementielle

- **Décentralisation** : Coordination via événements sans orchestrateur central
- **Event Sourcing** : Persistance complète de l'historique événementiel
- **CQRS** : Séparation commandes/requêtes avec read models optimisés
- **Résilience élevée** : Tolérance aux pannes distribuées

Objectifs Transversaux

- **Comparaison patterns** : Évaluation quantitative orchestré vs chorégraphié
- **Observabilité avancée** : Monitoring différentiel des deux approches
- **Scalabilité** : Capacité de montée en charge par pattern

1.3 Stakeholders

| Rôle                | Responsabilités Labo 6         | Responsabilités Labo 7            |
|---------------------|--------------------------------|-----------------------------------|
| Architecte Logiciel | Conception machine d'état saga | Conception flux événementiel      |
| Développeur Backend | Implémentation orchestrateur   | Développement workers réactifs    |
| DevOps              | Monitoring saga centralisée    | Observabilité distribuée          |
| Product Owner       | Validation workflow e-commerce | Validation performance asynchrone |
| Utilisateur Final   | Checkout synchrone fiable      | Checkout asynchrone performant    |

2. Contraintes

2.1 Contraintes Techniques

Infrastructure Requise

- **Python 3.11+** avec Django/Flask/FastAPI
- **PostgreSQL** : Persistance données métier et état saga
- **Redis Streams** : Event Bus et Event Store (Labo 7)
- **Kong Gateway** : Routage et authentification microservices
- **Docker Compose** : Orchestration containers

Contraintes Performance

- **Latence saga orchestrée** : < 2 secondes P95 pour 4 étapes
- **Latence saga chorégraphiée** : < 500ms P95 end-to-end
- **Throughput événementiel** : > 1000 événements/seconde
- **Event Store** : Rétention 30 jours minimum avec replay < 10 secondes

### Contraintes Observabilité

- **Métriques Prometheus** : Saga success/failure/duration par pattern
- **Logs structurés** : Corrélation distribuée via checkout\_id
- **Dashboards Grafana** : Comparaison temps réel des deux patterns

## 2.2 Contraintes Organisationnelles

### Patterns Architecturaux Obligatoires

- **Domain-Driven Design** : Bounded contexts préservés
- **Database per Service** : Isolation données par microservice
- **API-First** : Toute communication via interfaces REST
- **Idempotence** : Tous workers événementiels idempotents

### Contraintes Développement

- **Backward Compatibility** : Coexistence saga orchestrée + chorégraphiée
- **Testing Strategy** : Tests unitaires + tests saga end-to-end
- **Documentation** : ADR pour chaque décision architecturale majeure

## 2.3 Contraintes Règlementaires

### Audit et Traçabilité

- **Event Store** : Historique complet non-modifiable
- **Saga Audit Trail** : Traçabilité complète des décisions d'orchestration
- **Compensation Tracking** : Log détaillé des actions de rollback
- **GDPR Compliance** : Anonymisation événements clients si requis

---

## 3. Contexte

### 3.1 Contexte Métier

#### Domaine E-commerce

L'architecture implémente un **processus de checkout e-commerce** nécessitant coordination entre 3 microservices :

1. **service-catalogue** : Validation produits et prix
2. **service-inventaire** : Gestion stocks et réservations
3. **service-commandes** : Création commandes finales

## Complexité Transactionnelle

Le workflow checkout implique **4 étapes critiques** :

- Vérification disponibilité stock
- Récupération informations produit
- Réservation stock temporaire
- Création commande finale

**Échecs possibles** : stock insuffisant, produit indisponible, erreur service, timeout réseau

## 3.2 Évolution Architecturale

### Avant Labo 6 : Microservices DDD Simples

```
[Frontend] → [Kong] → [Microservices] → [PostgreSQL per Service]
```

- Communication HTTP directe
- Pas de gestion transactionnelle distribuée
- Échecs = erreurs utilisateur

### Labo 6 : Introduction Saga Orchestrée

```
[Frontend] → [service-saga-orchestrator] → [Kong] → [Microservices]
```

- **Orchestrateur central** : service-saga-orchestrator
- **Machine d'état explicite** : EN\_ATTENTE → SAGA\_TERMINEE
- **Compensation automatique** : Rollback en cas d'échec

### Labo 7 : Architecture Événementielle Complète

```
[service-ecommerce] → [Redis Streams] → [Workers Chorus] → [Kong] →  
[Microservices]  
↓  
[Event Store + CQRS]
```

- **Coordination décentralisée** : Workers réactifs
- **Event Sourcing** : Historique événementiel complet
- **CQRS** : Read models optimisés

## 3.3 Problématiques Saga

### Défis Saga Orchestrée (Labo 6)

- **Single Point of Failure** : Orchestrateur unique
- **Couplage temporel** : Latence cumulative des appels
- **Scalabilité limitée** : Bottleneck orchestrateur

### Défis Saga Chorégraphiée (Labo 7)

- **Complexité conceptuelle** : Coordination implicite
  - **Debugging difficile** : Pas de vue centralisée
  - **Cohérence éventuelle** : Délais de propagation événements
- 

## 4. Stratégie de Solution

### 4.1 Approche Architecturale

#### Principe Directeur : Dual Pattern Saga

Implémentation **comparative** de deux patterns saga pour le même use case :

1. **Pattern Orchestrée** : Coordination centralisée et synchrone
2. **Pattern Chorégraphiée** : Coordination décentralisée et asynchrone

**Objectif** : Démontrer empiriquement les trade-offs entre les deux approches.

### 4.2 Patterns Architecturaux Appliqués

#### Patterns Saga Communs

- **Compensation Pattern** : Actions rollback automatiques
- **Saga Log Pattern** : Traçabilité décisions et transitions
- **Timeout Pattern** : Gestion délais dépassés

#### Patterns Labo 6 - Orchestration

- **State Machine Pattern** : Machine d'état explicite
- **Coordinator Pattern** : Service orchestrateur central
- **Synchronous Communication** : Appels HTTP séquentiels

#### Patterns Labo 7 - Événementiel

- **Event Sourcing Pattern** : Persistance événementielle
- **CQRS Pattern** : Séparation commandes/requêtes
- **Choreography Pattern** : Coordination réactive
- **Pub/Sub Pattern** : Communication asynchrone

### 4.3 Analyse Patterns Saga

#### Saga Orchestrée - Avantages

- ☐ **Visibilité workflow** : Vue centralisée du processus

- **Debugging simple** : Logs centralisés dans orchestrateur
- **Contrôle explicite** : Machine d'état programmable
- **Rollback déterministe** : Compensation séquentielle garantie

### Saga Orchestrée - Inconvénients

- **SPOF** : Panne orchestrateur = blocage toutes sagas
- **Couplage fort** : Orchestrateur connaît tous services
- **Latence cumulative** : Appels synchrones séquentiels
- **Scalabilité limitée** : Bottleneck unique

### Saga Chorégraphiée - Avantages

- **Décentralisation** : Pas de point de défaillance unique
- **Scalabilité** : Workers parallèles indépendants
- **Performance** : Traitement asynchrone
- **Résilience** : Tolérance pannes élevée

### Saga Chorégraphiée - Inconvénients

- **Complexité conceptuelle** : Coordination implicite
- **Debugging distribué** : Traçabilité complexe
- **Cohérence éventuelle** : Délais propagation
- **Testing complexe** : Scénarios asynchrones

---

## 5. Vue des Blocs de Construction

### 5.1 Vue Implémentation



### Architecture Frontend Orchestrateur (Django)

#### Vues Orchestratrices

- **RapportConsolideView** : Rapport consolidé des ventes
- **GestionStockView** : Gestion stocks centraux et locaux
- **IndicateursView** : Indicateurs performance magasins
- **GestionProduitsView** : Catalogue produits via Kong
- **WorkflowDemandesView** : Validation demandes réapprovisionnement

#### Clients HTTP Infrastructure

- **CatalogueClient** → Kong → service-catalogue (Load Balanced)
- **InventaireClient** → Kong → service-inventaire
- **CommandesClient** → Kong → service-commandes
- **SupplyChainClient** → Kong → service-supply-chain
- **EcommerceClient** → Kong → service-ecommerce

## Architecture Microservices DDD (Ports 8001-8005)

### Service Catalogue (Port 8001) - Load Balancé

Interface: DDDCatalogueAPI, ProduitViewSet  
Application: RechercherProduitsUseCase, AjouterProduitUseCase, ModifierPrixUseCase  
Domain: Produit (archiver(), valider\_prix(), est\_actif()), PrixMonetaire, NomProduit, ReferenceSKU  
Infrastructure: DjangoProduitRepository

### Service Inventaire (Port 8002)

Interface: DDDInventaireAPI, StockViewSet  
Application: ConsulterStocksCentralUseCase, CreerDemandeReapproUseCase, TransfererStockUseCase  
Domain: StockCentral, StockLocal, DemandeReapprovisionnement, Quantite (VO)  
Infrastructure: DjangoStockRepository, DjangoDemandeRepository

### Service Commandes (Port 8003)

Interface: DDDVenteViewSet, DDDIndicateursAPI, DDDRapportConsolideAPI  
Application: EnregistrerVenteUseCase, GenererIndicateursUseCase, GenererRapportConsolideUseCase  
Domain: Vente (peut\_etre\_annulee(), calculer\_total()), Magasin, CommandeVente (VO)  
Infrastructure: DjangoVenteRepository, HttpStockService (external)

### Service Supply Chain (Port 8004)

Interface: DDDDemandesEnAttenteAPI, DDDValiderDemandeAPI, DDDRejeterDemandeAPI  
Application: ListerDemandesUseCase, ValiderDemandeUseCase (workflow\_3\_etapes, rollback\_automatique), RejeterDemandeUseCase  
Domain: WorkflowValidation, DemandeReapprovisionnement, MotifRejet (VO)  
Infrastructure: HttpDemandeRepository, HttpInventaireService (external)

### Service E-commerce (Port 8005)

Interface: DDDClientAPI, DDDPanierAPI, DDDCheckoutAPI  
Application: CreerCompteClientUseCase, AjouterProduitPanierUseCase, CheckoutEcommerceUseCase  
Domain: Client, Panier, ProcessusCheckout, AdresseLivraison (VO)



```
Infrastructure: DjangoClientRepository, DjangoPanierRepository,
HttpCatalogueService (external)
```

## Architecture Lab7 - Workers Événementiels

### Lab7 Common & Event Store

```
common/RedisEventBus: Wrapper Redis Streams (publish/subscribe,
ensure_consumer_group)
common/metrics: events_published_total, events_consumed_total,
event_latency_seconds, saga_choreo_*_total
event_store (Flask): /streams/{stream}/events,
/replay/checkout/{checkout_id}, /cqrs/orders-by-client/{client_id}
```

### Workers Chorégraphiés (Ports 9100-9105)

- **stock\_reservation\_worker** (9102) : Consumer group **choreo-reservation** → diminuer-stock via Kong
- **order\_creation\_worker** (9103) : Consumer group **choreo-order** → enregistrer-vente via Kong
- **stock\_compensation\_worker** (9104) : Consumer group **choreo-compensation** → augmenter-stock compensation
- **notification\_worker** (9100) : Consumer group notification → notifications utilisateur
- **audit\_worker** (9101) : Consumer group audit → audit trail événements
- **cqrs\_projection\_worker** (9105) : Consumer group **checkout-cqrs** → update read models

### Flux Communication Réel (selon vue-implementation.puml)

- **Frontend Views** → **HTTP Clients** : RapportConsolideView → CommandesClient, GestionStockView → InventaireClient, etc.
- **HTTP Clients** → **Kong Routes** : CatalogueClient → CatalogueUpstream (Load Balanced), autres clients → routes directes
- **Kong** → **Microservices** : Load balancing pour catalogue (3 instances), routage direct autres services
- **DDDCheckoutAPI** → **Redis Streams** : **publish CheckoutInitiated** vers topic **ecommerce.checkout.events**
- **Workers** → **Kong** → **Services** : HTTP via Kong pour réservation stock + création commandes
- **Workers** → **Event Store** : **cqrs\_projection\_worker** met à jour read models

## 5.2 Vue Déploiement

### Infrastructure Commune

- **Kong Gateway** : Point d'entrée unique (Port 8080)
- **Microservices DDD** : catalogue (8001), inventaire (8002), commandes (8003)
- **PostgreSQL** : Bases dédiées par service + saga state
- **Prometheus + Grafana** : Observabilité

### Infrastructure Labo 6 - Orchestrée

```
service-saga-orchestrator:
  build: ./service-saga-orchestrator
  ports: ["8010:8000"]
  environment:
    POSTGRES_HOST: saga-db
  depends_on: [kong, saga-db]
```

## Infrastructure Labo 7 - Événementielle

```
redis:
  image: redis:7.2-alpine
  ports: ["6379:6379"]

event-store:
  image: python:3.12-slim
  command: flask --app app:app run --host=0.0.0.0 --port=7010
  ports: ["7010:7010"]
  environment:
    REDIS_URL: redis://redis:6379/0

# 6 Workers événementiels
stock-reservation-worker:
  environment:
    METRICS_PORT: 9102
    KONG_URL: http://kong:8080

order-creation-worker:
  environment:
    METRICS_PORT: 9103

# ... autres workers (ports 9100-9105)
```

## 5.3 Vue Logique

### Modèle Saga Orchestrée

### Machine d'État Centrale

```
class EtatSaga(Enum):
    EN_ATTENTE = "EN_ATTENTE"
    VERIFICATION_STOCK = "VERIFICATION_STOCK"
    STOCK_VERIFIE = "STOCK_VERIFIE"
    RESERVATION_STOCK = "RESERVATION_STOCK"
    STOCK_RESERVE = "STOCK_RESERVE"
    CREATION_COMMANDE = "CREATION_COMMANDE"
    COMMANDE_CREEE = "COMMANDE_CREEE"
    SAGA_TERMINEE = "SAGA_TERMINEE"
```

```
# États échec + compensation
ECHEC_STOCK_INSUFFISANT = "ECHEC_STOCK_INSUFFISANT"
ECHEC_RESERVATION_STOCK = "ECHEC_RESERVATION_STOCK"
ECHEC_CREATION_COMMANDE = "ECHEC_CREATION_COMMANDE"
COMPENSATION_EN_COURS = "COMPENSATION_EN_COURS"
SAGA_ANNULEE = "SAGA_ANNULEE"
```

## Modèle Événementiel

### Types d'Événements

```
# Événements principaux
CheckoutInitiated = {"checkout_id", "client_id", "panier"}
StockReserved = {"checkout_id", "client_id", "panier"}
StockReservationFailed = {"checkout_id", "reason"}
OrderCreated = {"checkout_id", "commande_id", "client_id"}
CheckoutSucceeded = {"checkout_id", "commande_id"}
CheckoutFailed = {"checkout_id", "reason"}

# Événements compensation
StockReleased = {"checkout_id"}
OrderCreationFailed = {"checkout_id", "reason"}
```

## CQRS Read Model

```
{
  "cQRS:orders_by_client:{client_id}": {
    "total_orders": 5,
    "last_order_id": "uuid-commande",
    "last_checkout_id": "uuid-checkout",
    "last_update_ts": 1691234567.89
  }
}
```

## 5.4 Vue Cas d'Utilisation



Vue cas d'utilisation

### Use Cases Saga Orchestrée

- **UC-SAGA-01** : Démarrer saga checkout (POST /api/saga/checkout/)
- **UC-SAGA-02** : Consulter état saga (GET /api/saga/{id}/status/)
- **UC-SAGA-03** : Lister sagas actives (GET /api/saga/list/)
- **UC-SAGA-04** : Forcer compensation (POST /api/saga/{id}/compensate/)

### Use Cases Événementiels

- **UC-EVENT-01** : Initier checkout chorégraphié (POST /api/commandes/clients/{id}/checkout/choreo/)
  - **UC-EVENT-02** : Consulter Event Store (GET /api/event-store/streams/{stream}/events)
  - **UC-EVENT-03** : Replay checkout (GET /api/event-store/replay/checkout/{id})
  - **UC-EVENT-04** : Requête CQRS (GET /api/event-store/cqrs/orders-by-client/{id})
- 

## 6. Vue d'Exécution

### 6.1 Processus Saga Orchestrée - Machine d'état centralisée

 Séquence checkout comparatif

#### Workflow Orchestré (Labo 6)

**Acteurs** : Client Web, service-saga-orchestrator, Kong Gateway, 3 microservices

**Séquence nominale** :

1. **Client** → **Frontend** : Demande checkout
2. **Frontend** → **service-saga-orchestrator** : POST /api/saga/checkout/
3. **Orchestrateur** : Transition **EN\_ATTENTE** → **VERIFICATION\_STOCK**
4. **Orchestrateur** → **Kong** → **service-inventaire** : Vérification stock
5. **Orchestrateur** : Transition **VERIFICATION\_STOCK** → **STOCK\_VERIFIE**
6. **Orchestrateur** → **Kong** → **service-catalogue** : Info produit
7. **Orchestrateur** : Transition **STOCK\_VERIFIE** → **RESERVATION\_STOCK**
8. **Orchestrateur** → **Kong** → **service-inventaire** : Réservation stock
9. **Orchestrateur** : Transition **RESERVATION\_STOCK** → **STOCK\_RESERVE**
10. **Orchestrateur** → **Kong** → **service-commandes** : Création commande
11. **Orchestrateur** : Transition **CREATION\_COMMANDE** → **SAGA\_TERMINEE**
12. **Orchestrateur** → **Frontend** : Résultat saga + commande\_id

**Gestion d'échec avec compensation** :

- **Échec étape N** → Transition vers état échec approprié
- **Orchestrateur** : Exécution `_executer_compensation()`
- **Libération ressources** : Appels augmenter-stock pour rollback
- **Transition finale** : **COMPENSATION\_EN\_COURS** → **SAGA\_ANNULEE**

### 6.2 Processus Saga Chorégraphiée - Coordination événementielle

#### Workflow Chorégraphié (Labo 7)

**Acteurs** : service-ecommerce, Redis Streams, 6 Workers, Kong Gateway, 3 microservices

**Séquence événementielle** :

1. **Client** → **service-ecommerce** : POST /api/commandes/clients/{id}/checkout/choreo/
2. **service-ecommerce** → **Redis** : Publish **CheckoutInitiated**
3. **service-ecommerce** → **Client** : 202 Accepted + checkout\_id
4. **stock\_reservation\_worker** : Consomme **CheckoutInitiated**

5. `stock_reservation_worker` → Kong → `service-inventaire` : Réservation
6. `stock_reservation_worker` → Redis : Publish `StockReserved` OR `StockReservationFailed`
7. `order_creation_worker` : Consomme `StockReserved`
8. `order_creation_worker` → Kong → `service-commandes` : Création commande
9. `order_creation_worker` → Redis : Publish `OrderCreated` + `CheckoutSucceeded`
10. `cqrs_projection_worker` : Consomme `OrderCreated` → Mise à jour read model

#### Flux de compensation asynchrone :

- `order_creation_worker` → Redis : Publish `OrderCreationFailed`
- `stock_compensation_worker` : Consomme échec → Libération stock
- `stock_compensation_worker` → Redis : Publish `StockReleased` + `CheckoutFailed`

### 6.3 Observabilité et Métriques

#### Emplacements Captures Grafana

##### [À insérer] Capture 1 : "Saga Orchestrée vs Chorégraphiée - Success Rate"

- Métriques : `saga_completed_total` vs `saga_choreo_success_total`
- Période : 1 heure avec tests de charge
- Commentaire attendu : Taux succès comparable, latence orchestrée plus élevée

##### [À insérer] Capture 2 : "Event Bus Activity - Published/Consumed Rate"

- Métriques : `events_published_total`, `events_consumed_total` par type
- Graphique : Distribution événements par worker
- Commentaire attendu : Équilibrage charge entre workers, pics lors checkouts

##### [À insérer] Capture 3 : "Latence End-to-End P50/P95"

- Métriques : `saga_execution_duration` vs `event_latency_seconds`
- Comparaison : Orchestrée (synchrone) vs Chorégraphiée (asynchrone)
- Commentaire attendu : Chorégraphiée plus performante, moins de variance

#### Métriques Techniques

#### Labo 6 - Saga Orchestrée

```
saga_started_total = Counter('saga_started_total')
saga_completed_total = Counter('saga_completed_total')
saga_failed_total = Counter('saga_failed_total', ['reason', 'step'])
saga_execution_duration = Histogram('saga_execution_duration_seconds')
external_service_calls =
Histogram('external_service_call_duration_seconds', ['service'])
```

#### Labo 7 - Événementiel

```
events_published_counter = Counter('events_published_total', ['topic',
'type'])
events_consumed_counter = Counter('events_consumed_total', ['topic',
'type', 'consumer'])
event_latency_seconds = Histogram('event_latency_seconds', ['topic',
'type'])
saga_choreo_success_total = Counter('saga_choreo_success_total',
['source'])
saga_choreo_failed_total = Counter('saga_choreo_failed_total', ['source'])
```

---

## 7. Vue de Déploiement

### 7.1 Configuration Docker Compose - Services saga + infrastructure

#### Services Saga Orchestrée (Labo 6)

```
# Service orchestrateur central
service-saga-orchestrator:
  build: ./service-saga-orchestrator
  container_name: saga-orchestrator
  ports:
    - "8010:8000"
  environment:
    POSTGRES_HOST: saga-db
    KONG_GATEWAY_URL: http://kong:8080
    PROMETHEUS_MULTIPROC_DIR: /tmp
  volumes:
    - ./service-saga-orchestrator:/app
  depends_on:
    - saga-db
    - kong
  restart: unless-stopped

# Base dédiée état saga
saga-db:
  image: postgres:15
  container_name: saga-db
  ports:
    - "5439:5432"
  environment:
    POSTGRES_DB: saga_db
    POSTGRES_USER: saga_user
    POSTGRES_PASSWORD: saga_pass
  volumes:
    - saga_data:/var/lib/postgresql/data
  restart: unless-stopped
```

### 7.2 Architecture Redis Streams - Event Bus et Event Store

## Configuration Redis Streams

**Topic principal** : `ecommerce.checkout.events`

```
# Structure Redis Stream
redis-cli XINFO STREAM ecommerce.checkout.events

# Consumer Groups dédiés
choreo-reservation    # stock_reservation_worker
choreo-order          # order_creation_worker
choreo-compensation   # stock_compensation_worker
choreo-notification   # notification_worker
choreo-audit          # audit_worker
choreo-cqrs           # cqrs_projection_worker
```

**Format événements standardisé** :

```
{
  "type": "CheckoutInitiated",
  "payload": "{\"checkout_id\": \"uuid\", \"client_id\": \"uuid\",
  \"panier\": {...}, \"emitted_at\": 1691234567.89}\",
  \"ts\": \"1691234567.89\"
}
```

---

## 8. Concepts Transversaux

### 8.1 Patterns Saga - Orchestration vs Chorégraphie

#### 8.1.1 Saga Orchestrée - Coordination Centralisée

**Principe** : Un orchestrateur central coordonne toutes les étapes de la saga.

**Implémentation** :

```
class SagaOrchestrator:
    def executer_saga(self, saga: SagaCommande) -> Dict[str, Any]:
        try:
            # Étape 1: Vérification stock
            self._verifier_stock(saga)

            # Étape 2: Récupération produit
            self._recuperer_informations_produit(saga)

            # Étape 3: Réservation stock
            self._reserver_stock(saga)

            # Étape 4: Création commande
```

```
        self._creer_commande(saga)

        # Finalisation
        saga.transitionner_vers(EtatSaga.SAGA_TERMINEE)
        return saga.obtenir_resume_execution()

    except Exception as e:
        # Compensation automatique
        self._executer_compensation(saga)
        raise
```

#### Avantages :

- □ Contrôle explicite du workflow
- □ Debugging centralisé et simplifié
- □ Rollback déterministe
- □ Monitoring unifié

#### Inconvénients :

- □ Single Point of Failure
  - □ Couplage fort orchestrateur ↔ services
  - □ Latence cumulative (appels séquentiels)
  - □ Scalabilité limitée
- 

## 9. Décisions

### 9.1 ADR-001 — Redis Streams pour Event Store et CQRS

**Status** : Adopted

**Date** : Août 2025

**Context** : Besoin d'un système de messagerie performant avec Event Sourcing pour Labo 7

#### Decision

Adoption de **Redis Streams** comme Event Bus et Event Store minimal, avec capacités Pub/Sub natives et persistance séquentielle.

#### Alternatives Considérées

- **Apache Kafka** : Plus robuste mais complexité excessive pour contexte lab
- **RabbitMQ** : Message Broker traditionnel sans Event Sourcing natif
- **PostgreSQL + LISTEN/NOTIFY** : Limitations performance et rétention
- **EventStore DB** : Spécialisé mais déploiement complexe

#### Consequences

**Positives** :



- ☐ **Performance élevée** : Redis Streams optimisé pour throughput
- ☐ **Event Sourcing natif** : Rétention événements avec XRANGE
- ☐ **Consumer Groups** : Distribution automatique entre workers
- ☐ **Simplicité déploiement** : Redis container unique
- ☐ **Replay capability** : Reconstruction état via XRANGE queries

#### Négatives :

- ☐ **Persistance limitée** : Pas de garantie durabilité vs Kafka
- ☐ **Features basiques** : Pas de schema registry ou partitioning avancé
- ☐ **Single point** : Redis unique (pas de cluster dans lab)

## 9.2 ADR-002 — Saga Chorégraphiée pour coordination décentralisée

**Status** : Adopted

**Date** : Août 2025

**Context** : Démonstration alternative décentralisée à l'orchestration saga du Labo 6

#### Decision

Implémentation d'une **saga chorégraphiée** via workers événementiels réactifs, sans orchestrateur central, pour le même workflow e-commerce.

#### Alternatives Considérées

- **Extension orchestrateur** : Améliorer saga centralisée existante
- **Hybrid Pattern** : Orchestration + événements pour différentes étapes
- **Process Manager** : Orchestrateur stateless réactif aux événements
- **State Machine Distribuée** : Coordination via états partagés

#### Consequences

##### Positives :

- ☐ **Décentralisation complète** : Pas de SPOF applicatif
- ☐ **Scalabilité horizontale** : Workers parallèles indépendants
- ☐ **Résilience élevée** : Tolérance pannes individuelle workers
- ☐ **Performance asynchrone** : Traitement parallèle des étapes
- ☐ **Evolution indépendante** : Workers déployables séparément

##### Négatives :

- ☐ **Complexité conceptuelle** : Coordination implicite via événements
- ☐ **Debugging distribué** : Pas de vue centralisée du workflow
- ☐ **Testing complexe** : Scénarios asynchrones et conditions de course
- ☐ **Cohérence éventuelle** : Délais propagation entre workers

## 9.3 Alternatives Écartées

### 2PC (Two-Phase Commit)

**Raison rejet** : Couplage synchrone fort, timeouts, complexité recovery **Impact** : Simplicité patterns saga préférée pour résilience

### Orchestration Centralisée Exclusive

**Raison rejet** : Labo 7 vise à démontrer approche décentralisée **Impact** : Maintien dual pattern orchestré/chorégraphié pour comparaison

### Message Queue Externe (RabbitMQ)

**Raison rejet** : Redis Streams suffit pour besoins lab + Event Sourcing intégré **Impact** : Infrastructure simplifiée avec moins de dépendances

---

## 10. Scénarios de Qualité

### 10.1 Cohérence - Sagas distribuées et compensation

#### Scenario C1 : Cohérence Saga Orchestrée

- **Source** : Administrateur teste compensation orchestrée
- **Stimulus** : Échec service-commandes pendant création commande (timeout)
- **Environment** : service-saga-orchestrator + 3 microservices opérationnels
- **Response** : Compensation automatique immédiate via orchestrateur
- **Measure** : 100% compensations réussies, stock libéré < 5 secondes

### 10.2 Performance - Latence orchestrée vs événementielle

#### Scenario P1 : Performance Saga Orchestrée

- **Source** : 50 utilisateurs simultanés, checkout e-commerce
- **Stimulus** : Demandes checkout via service-saga-orchestrator
- **Environment** : 4 étapes synchrones séquentielles via Kong
- **Response** : Workflow complet orchestré de bout en bout
- **Measure** : Latence P95 < 2 secondes, throughput > 25 checkouts/minute

#### Mesures observées :

- Latence moyenne : ~1.8s (cumul 4 appels HTTP)
- P95 : 2.1s
- Throughput : 30 checkouts/minute
- Limitation : Orchestrateur unique (bottleneck)

#### Scenario P2 : Performance Saga Chorégraphiée

- **Source** : 50 utilisateurs simultanés, checkout e-commerce
- **Stimulus** : Demandes checkout via endpoint événementiel
- **Environment** : Workers parallèles + Redis Streams asynchrone
- **Response** : Coordination distribuée sans goulot d'étranglement

- **Measure** : Latence P95 < 500ms, throughput > 100 checkouts/minute

#### Mesures observées :

- Latence moyenne : ~300ms (traitement asynchrone)
  - P95 : 450ms
  - Throughput : 120 checkouts/minute
  - Avantage : Parallélisation workers indépendants
- 

## 11. Risques et Dette Technique

### 11.1 Risques Techniques

#### RISK-001 : Dualité Architecturale Saga

**Probabilité** : Élevée

**Impact** : Moyen

**Description** : Maintenance simultanée de deux patterns saga différents augmente complexité

#### Impacts spécifiques :

- **Code duplication** : Logique checkout dans orchestrateur ET workers
- **Testing overhead** : Couverture test double (orchestré + chorégraphié)
- **Monitoring complexe** : Métriques différentielles à maintenir
- **Formation équipe** : Compétences sur deux paradigmes

#### Mitigation :

- Documentation claire des use cases par pattern
- Tests comparatifs automatisés
- Choix pattern dominant basé sur métriques production
- Migration progressive vers pattern optimal

#### RISK-002 : Complexité Coordination Événementielle

**Probabilité** : Élevée

**Impact** : Élevé

**Description** : Debugging et maintenance saga chorégraphiée complexe en production

#### Défis techniques :

- **Ordre événements** : Garanties ordering limitées Redis Streams
- **Conditions de course** : Workers parallèles sur mêmes ressources
- **État distribué** : Pas de vue globale cohérente temps réel
- **Rollback partiel** : Compensation échoue sur subset workers

#### Mitigation :

- Event Store centralisé pour reconstruction état
- Idempotence stricte tous workers

- Timeouts et retry policies cohérents
- Circuit breakers sur appels externes

## 11.2 Dette Technique

### DEBT-001 : Event Store Flask Simple

**Description** : Event Store implémenté en Flask basique, fonctionnalités limitées **Impact** : Limitations query, pas de backup/restore, pas de high availability

### DEBT-002 : CQRS Read Models Volatiles

**Description** : Projections CQRS stockées uniquement en Redis, pas de persistance **Impact** : Perte read models au redémarrage, reconstruction coûteuse

### DEBT-003 : Monitoring Saga Basique

**Description** : Métriques Prometheus basiques, pas de distributed tracing avancé **Impact** : Corrélation limitée entre patterns saga, debugging complexe

---

## 12. Glossaire

### Termes Saga et Patterns

**Saga** : Pattern de gestion transaction distribuée via séquence d'opérations locales avec compensation

**Orchestration** : Coordination centralisée d'une saga via orchestrateur unique

**Chorégraphie** : Coordination décentralisée d'une saga via événements réactifs

**Compensation** : Actions de rollback pour annuler effets saga partiellement échouée

**Machine d'État** : Modèle explicite des états et transitions d'une saga orchestrée

### Termes Événementiel et CQRS

**Event Sourcing** : Persistance état via séquence événements immuables

**Event Store** : Base de données spécialisée stockage et replay événements

**CQRS** : Command Query Responsibility Segregation - séparation commandes/lectures

**Read Model** : Vue matérialisée optimisée pour requêtes (Query side CQRS)

**Event Bus** : Infrastructure messagerie pour publication/consommation événements

**Consumer Group** : Groupe Redis Streams pour distribution événements entre workers

### Termes Techniques Architecture

**Redis Streams** : Structure données Redis pour messagerie Pub/Sub avec persistance

**Worker Chorégraphié** : Service réactif consommant événements et publiant réponses

**Replay** : Reconstruction état entité via rejeu séquentiel événements historiques

**Idempotence** : Propriété opération produisant même résultat si exécutée plusieurs fois

**Circuit Breaker** : Pattern protection contre cascading failures entre services

### Métriques et Observabilité

**Saga Success Rate** : Pourcentage sagas terminées avec succès vs total démarrées

**Event Latency P95** : 95e percentile latence émission → consommation événement

**Throughput Saga** : Nombre sagas traitées par unité temps

**Compensation Rate** : Pourcentage sagas nécessitant rollback vs total

**Worker Lag** : Retard traitement événements par worker (Redis XINFO GROUPS)

**Replay Duration** : Temps reconstruction état via Event Store replay

## Acronymes Techniques

**DDD** : Domain-Driven Design - approche conception orientée domaine métier

**ACID** : Atomicity, Consistency, Isolation, Durability - propriétés transactions

**2PC** : Two-Phase Commit - protocole transactions distribuées synchrones

**SPOF** : Single Point of Failure - composant unique critique

**HA** : High Availability - haute disponibilité système

**SLA** : Service Level Agreement - accord niveau service

---

## Fin du rapport ARC42

*Document de référence pour l'architecture saga orchestrée et chorégraphiée - LOG430 Labo 6-7*

*Évolution démontrée : Saga Orchestrée Centralisée → Architecture Événementielle Décentralisée*