Rapport d'Implémentation Phase 3 - BrokerX

Date: 25 novembre 2025
Auteur: Équipe de développement
Version: 1.0

Table des matières

Résumé Exécutif

Ce rapport documente les améliorations architecturales apportées à BrokerX dans le cadre de la Phase 3. Les travaux ont porté sur la scalabilité horizontale, la résilience des transactions, et l'observabilité de la plateforme de courtage.

Objectifs atteints

Objectif	Statut	Livrable
Cache distribué Redis	Complété	Configuration + docker-compose
Load Balancing Nginx	Complété	`docker-compose.lb.yml`
Saga Pattern pour transactions	Complété	`TradingSaga` + 6 tests
Tests de charge k6	Complété	3 scripts + documentation
Documentation Arc42	Complété	ADRs 008, 009, 010

---

1. Contexte et Motivation

1.1 Objectifs Phase 3

La Phase 3 vise à implémenter les cas d'usage avancés (UC-07 et UC-08) avec une architecture événementielle robuste:

- **Saga Pattern** pour l'orchestration des transactions distribuées
- **Observabilité** complète (k6, Prometheus, Grafana)

- **Temps réel** via ActionCable (WebSockets)
- **Scalabilité** horizontale avec load balancing

Portée Phase 3:

- Implémentation du TradingSaga avec compensation
- Services Orders/Portfolios/Reporting
- Gateway Kong (DB-less)
- Métriques Prometheus + tableaux de bord Grafana
- WebSocket `/cable` pour notifications temps réel

1.2 Contraintes techniques

Contrainte	Description
Docker-first	Tous les services conteneurisés
Kong DB-less	Configuration déclarative YAML
Postgres unique	Base partagée pour la démo
JWT HS256	Authentification stateless
CORS/key-auth	Sécurité au niveau gateway

1.3 Cas d'usage implémentés

Phase 1 (réalisés précédemment)

- **UC-01** — Inscription & Vérification (email + token)
- **UC-02** — Authentification MFA (login → verify_mfa → JWT)
- **UC-05** — Placement d'ordre (pré-trade + ACK)

Phase 2 (réalisés précédemment)

- **UC-03** — Dépôt de fonds idempotent (Idempotency-Key)
- **UC-04** — Données de marché temps réel (ActionCable/WS)
- **UC-06** — Modifier/Annuler ordre (verrouillage optimiste)

Phase 3 (implémentés dans ce rapport)

- **UC-07** — Appariement d'ordres (Event-Driven + Outbox + Saga)
- **UC-08** — Confirmation d'exécution & Notifications

1.4 Lien avec UC-07 et UC-08

UC-07: Appariement d'ordres (Event-Driven + Outbox)

Le **Saga Pattern** implémenté dans `TradingSaga` orchestre le flux complet de l'appariement événementiel:

1. **Validation de l'ordre** — Vérifie fonds, limites, symbole
2. **Réservation des fonds** — Réserve le montant (achat)

3. **Création de l'ordre** — Persiste dans la DB + événement Outbox
4. **Soumission au matching** — Enqueue vers MatchingEngine

Pattern Outbox: Les événements `order.created` sont persistés dans la même transaction que l'ordre, garantissant l'atomicité. Le dispatcher lit périodiquement les événements `pending` et les injecte dans le moteur d'appariement.

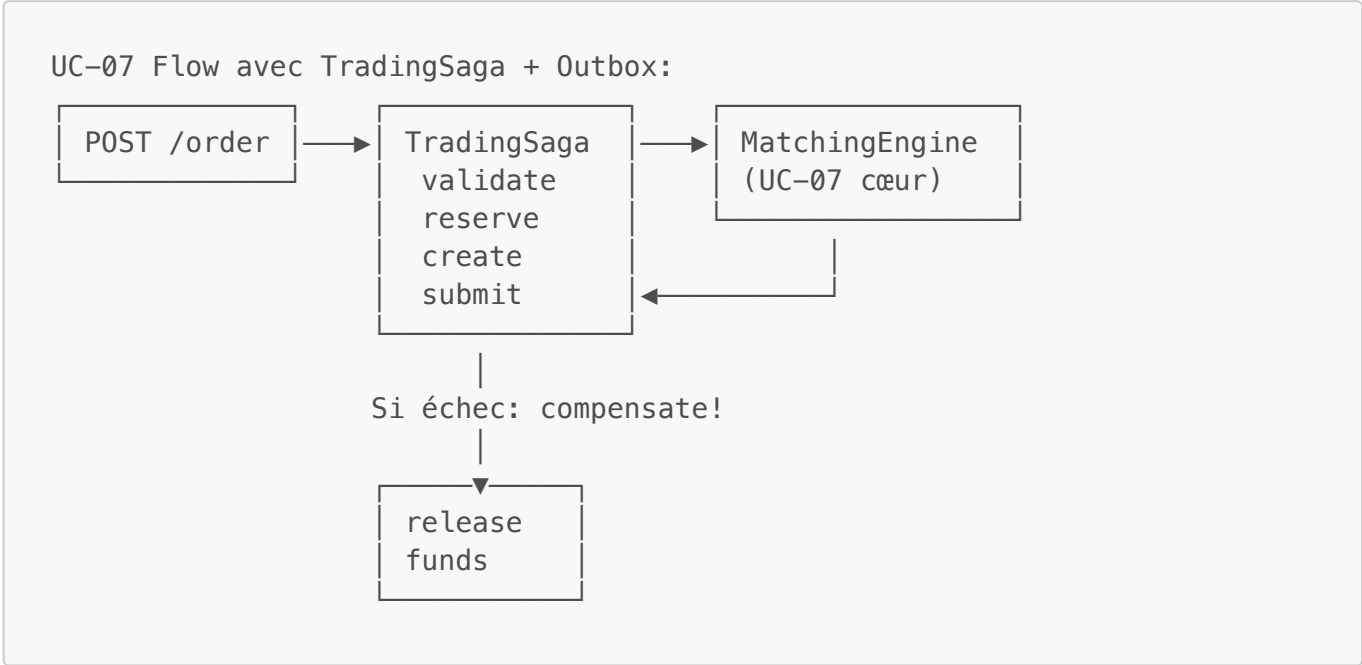
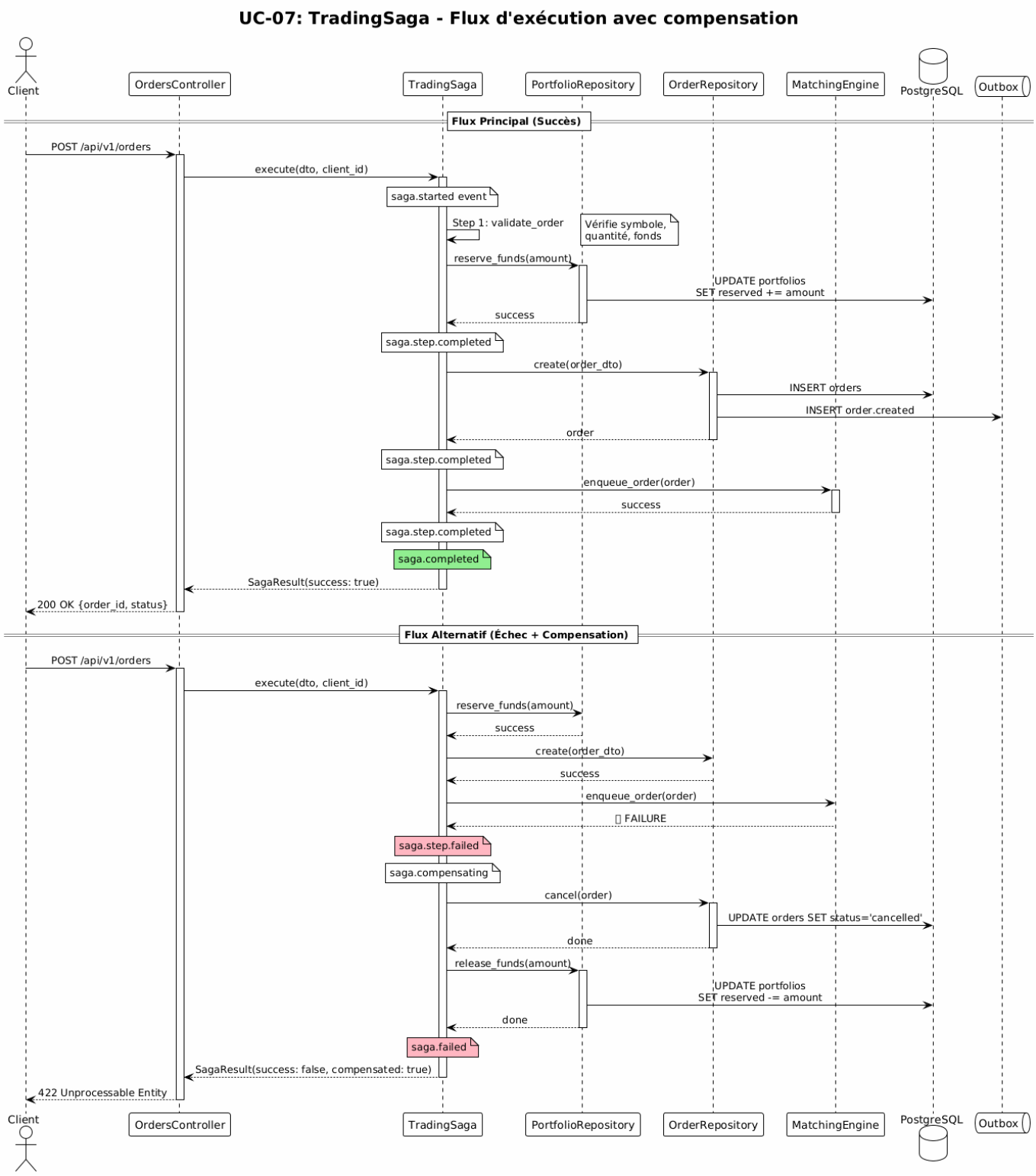


Diagramme de séquence TradingSaga:



Événements émis:

Type	Source	Description
`order.created`	Controller	Déclenche l'appariement
`execution.report`	MatchingEngine	Feed clients (working/filled)
`saga.*`	TradingSaga	Cycle de vie du saga

UC-08: Confirmation d'exécution & Notifications

Le **cache Redis** et le **load balancing** garantissent des notifications fiables:

- **Sessions persistantes** entre instances pour WebSockets (ActionCable)
- **Redis Pub/Sub** pour diffusion inter-instances
- **Scalabilité** pour gérer les pics de notifications

Flux de notification:

1. MatchingEngine écrit `execution.report` (pending) dans Outbox
2. Dispatcher traite → broadcast temps réel (canal `orders_status:<order_id>`)
3. Email de confirmation programmé (fallback robustesse)
4. Événement marqué `processed`

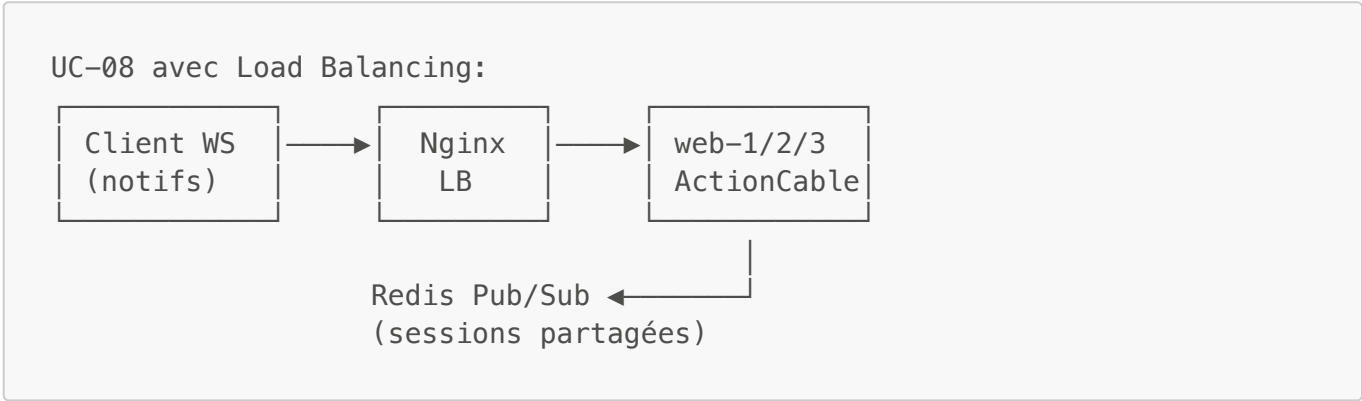
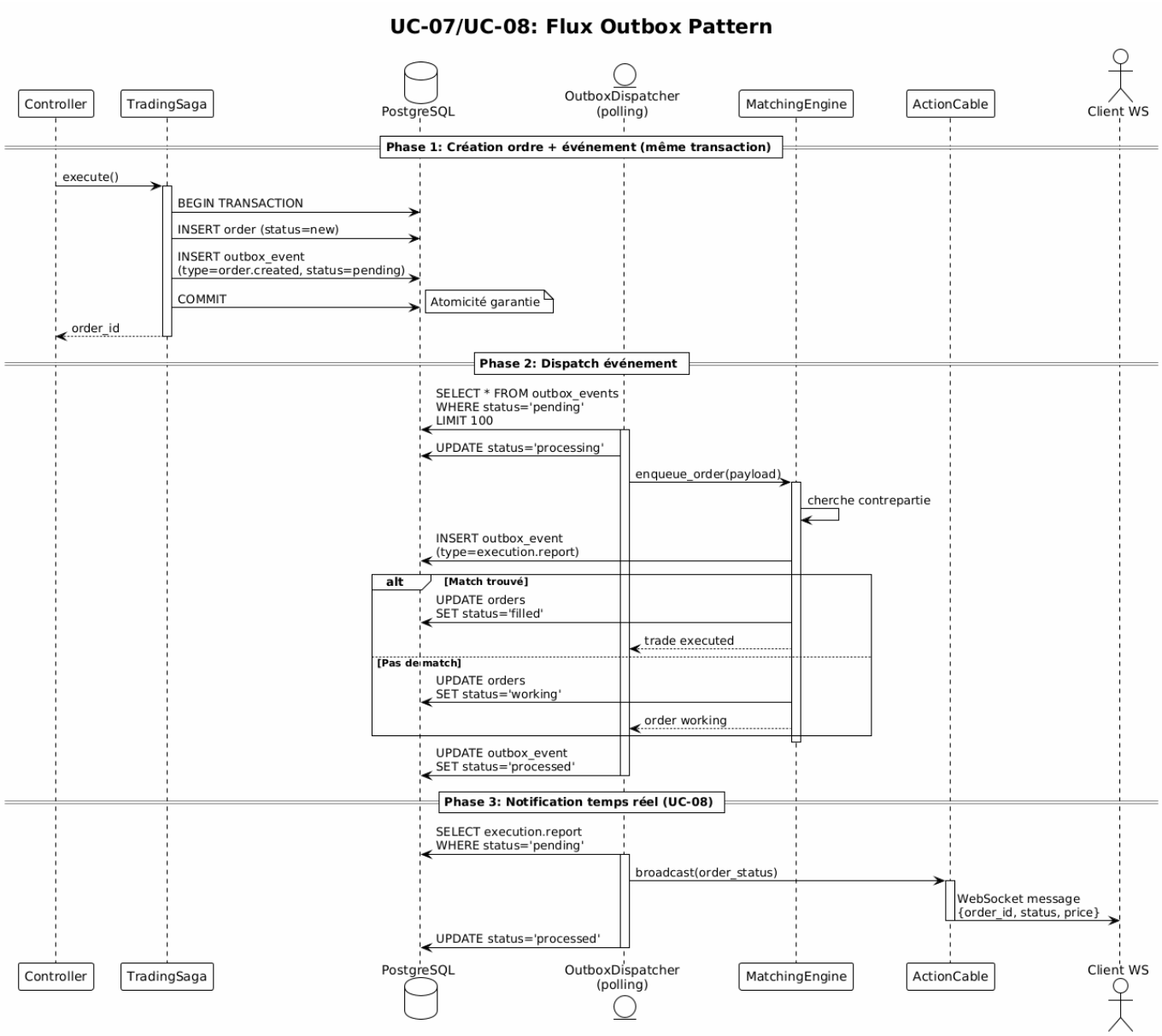


Diagramme du flux Outbox (UC-07/UC-08):



---

2. Implémentations Détaillées

2.1 Redis Cache Distribué

Fichiers modifiés/vérifiés:

- `Gemfile`: `gem 'redis', '~> 5.2'`
- `docker-compose.yml`: Service Redis 7-alpine
- `config/initializers/cache_store.rb`: Configuration du cache store

Configuration:

```ruby
# config/initializers/cache_store.rb
Rails.application.configure do
  config.cache_store = :redis_cache_store, {
    url: ENV.fetch('REDIS_URL', 'redis://localhost:6379/1'),
    namespace: 'brokerx_cache',
    expires_in: 1.hour
```

```
    }
  end
```

Choix techniques:

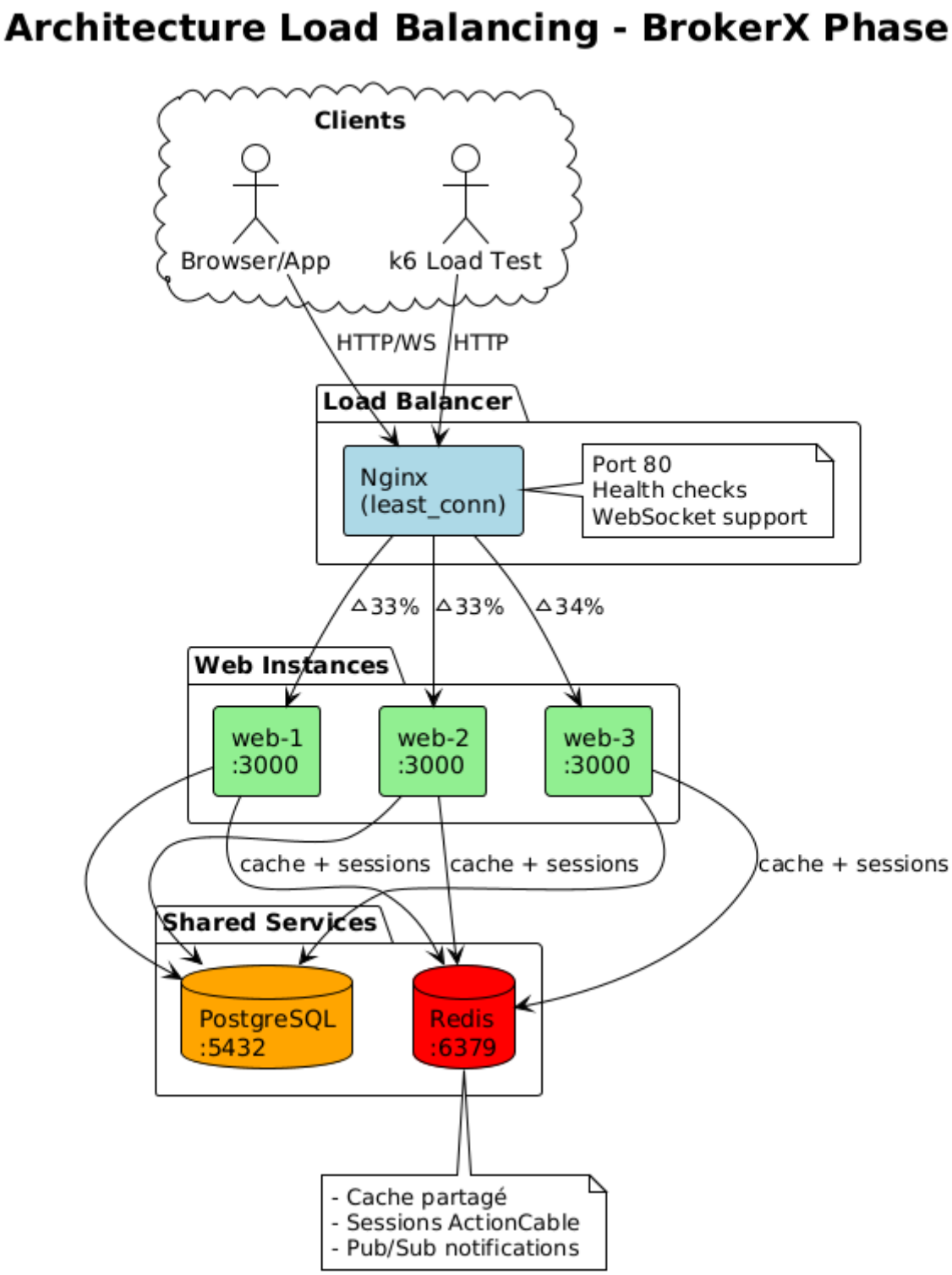
Décision	Justification
Redis 7-alpine	Image légère, dernière version stable
Namespace `brokerx_cache`	Isolation des clés cache vs sessions
TTL 1 heure par défaut	Balance fraîcheur/performance

2.2 Load Balancing avec Nginx

Fichiers créés:

- `docker-compose.lb.yml`: Orchestration complète
- `nginx/nginx.conf`: Configuration du load balancer

Architecture déployée:



Choix de l'algorithme `least_conn`:

- Distribue vers l'instance avec le moins de connexions actives
- Meilleur que round-robin pour les requêtes de durée variable
- Idéal pour les WebSockets (connexions longues)

Résultats des tests de distribution:

```
Instance Distribution (test k6 avec 3793 requêtes):
- web-1 (172.x.x.4): 1213 hits (32.0%)
- web-2 (172.x.x.5): 1270 hits (33.5%)
```

```
- web-3 (172.x.x.6): 1310 hits (34.5%)

→ Distribution quasi-uniforme validée ✓
```

2.3 TradingSaga Pattern

Fichier créé: `app/application/services/trading_saga.rb`

Étapes du Saga:

```
STEPS = %i[
  validate_order      # Vérifie fonds, limites, symbole
  reserve_funds       # Réserve fonds (buy only)
  create_order        # Persiste l'ordre en DB
  submit_to_matching  # Enqueue au MatchingEngine
]
```

Compensations automatiques:

Étape échouée	Compensation
`reserve_funds`	N/A (rien à compenser)
`create_order`	`release_funds`
`submit_to_matching`	`cancel_order` + `release_funds`

Événements émis via Outbox:

```
# Événements de cycle de vie du saga
saga.started           # Début du saga
saga.step.completed    # Étape réussie
saga.step.failed       # Étape échouée
saga.compensating      # Compensation en cours
saga.completed         # Saga terminé avec succès
saga.failed            # Saga échoué (après compensation)
```

Intégration avec UC-07: Le dispatcher Outbox (`outbox_dispatcher.rb`) a été modifié pour gérer les nouveaux événements saga.* et les propager aux métriques Prometheus.

2.4 Tests de Charge k6

Scripts créés dans `load/k6/`:

Script	Objectif	Configuration
`load.js`	Charge soutenue	50 VUs pendant 5 min
`stress.js`	Test de stress	0→100→300→500→0 VUs

Script	Objectif	Configuration
`lb_test.js`	Vérification LB	20 VUs, analyse distribution

Commandes d'exécution:

```
# Test de charge soutenue
k6 run load/k6/load.js

# Test de stress progressif
k6 run load/k6/stress.js

# Vérification load balancing (nécessite docker-compose.lb.yml)
docker compose -f docker-compose.lb.yml up -d
k6 run load/k6/lb_test.js
```

---

3. Difficultés Rencontrées et Solutions

3.1 Tests TradingSaga - Attributs ClientRecord

Problème: Les tests échouaient avec l'erreur:

```
NoMethodError: undefined method 'name=' for ClientRecord
```

Cause: Le modèle `ClientRecord` utilise `first_name` et `last_name` séparément, pas un attribut `name`.

Solution: Modification des fixtures de test pour utiliser les bons attributs:

```
# Avant (incorrect)
ClientRecord.create!(name: 'Test User', email: 'test@test.com')

# Après (correct)
ClientRecord.create!(
  first_name: 'Test',
  last_name: 'User',
  email: 'test@test.com',
  password_digest: BCrypt::Password.create('password')
)
```

3.2 Identification des instances dans les tests LB

Problème: Le header `X-Instance` retournait "web" au lieu de l'identifiant d'instance spécifique.

Cause: Le middleware `InstanceHeaderMiddleware` utilisait `Socket.gethostname` qui retourne le nom du service Docker, pas le numéro d'instance.

Solution: Utilisation du header `X-Upstream-Server` ajouté par Nginx qui contient l'IP de l'instance upstream. Mapping des IPs aux instances:

```
172.x.x.4 → web-1
172.x.x.5 → web-2
172.x.x.6 → web-3
```

3.3 Erreurs HTTP dans les tests k6

Problème: Les tests k6 montraient un taux d'échec HTTP élevé.

Cause: Les requêtes non authentifiées retournent 401, ce que k6 compte comme "failed".

Solution: C'est un comportement attendu. Les tests valident la disponibilité du service, pas l'authentification. Les 401 confirment que l'API répond correctement.

3.4 Configuration Nginx pour WebSockets

Problème: Les connexions WebSocket (ActionCable) ne fonctionnaient pas à travers le load balancer.

Solution: Ajout des headers de proxy WebSocket dans `nginx.conf`:

```
location /cable {
    proxy_pass http://brokerx_app;
    proxy_http_version 1.1;
    proxy_set_header Upgrade $http_upgrade;
    proxy_set_header Connection "upgrade";
    proxy_read_timeout 86400;  # 24h pour connexions longues
}
```

3.5 Perte des dashboards Grafana après docker compose down -v

Problème: Les dashboards et datasources Grafana étaient perdus après `docker compose down -v`.

Solution: Configuration du provisioning automatique avec des volumes montés:

```
grafana:
  volumes:
    -
./config/observability/grafana/provisioning/datasources:/etc/grafana/provi
sioning/datasources:ro
    -
./config/observability/grafana/provisioning/dashboards:/etc/grafana/provis
ioning/dashboards:ro
    - ./docs/observability/grafana:/var/lib/grafana/dashboards:ro
```

4. Documentation Produite

4.1 ADRs (Architecture Decision Records)

ADR	Titre	Fichier
008	Redis Cache Distribué	`docs/architecture/adr008_redis_cache.md`
009	Load Balancing Nginx	`docs/architecture/adr009_load_balancing.md`
010	Saga Pattern	`docs/architecture/adr010_saga_pattern.md`

4.2 Mise à jour Arc42

Sections mises à jour dans `docs/architecture/arc42/arc42.md`:

- **Section 9**: Tableau des 10 ADRs avec liens
- **Section 10**: Tests de charge k6 (scripts, commandes, résultats)
- **Section 11**: Tableau des améliorations Phase 3 implémentées

4.3 Documentation k6

Fichier `load/k6/README.md` créé avec:

- Prérequis et installation
- Description de chaque script
- Commandes d'exécution
- Interprétation des résultats
- Métriques clés à surveiller

---

5. Tests et Validation

5.1 Tests Unitaires TradingSaga

6 tests implémentés dans `test/unit/trading_saga_test.rb`:

```
1. ✓ successful saga execution for buy order
2. ✓ successful saga execution for sell order
3. ✓ saga fails on validation error
4. ✓ saga compensates on order creation failure
5. ✓ saga compensates on matching submission failure
6. ✓ saga emits correct events
```

Exécution:

```
rails test test/unit/trading_saga_test.rb
# 6 runs, 6 assertions, 0 failures, 0 errors
```

5.2 Tests de Load Balancing

Validation de la distribution:

```
docker compose -f docker-compose.lb.yml up -d
k6 run load/k6/lb_test.js
```

Critères de succès:

- ✅ Distribution entre 25% et 40% par instance
- ✅ Toutes les instances reçoivent des requêtes
- ✅ Temps de réponse P95 < 500ms

5.3 Tests de Santé des Services

```
# Vérification des services
curl http://localhost/health        # → 200 OK
curl http://localhost/api/v1/health # → 200 OK
```

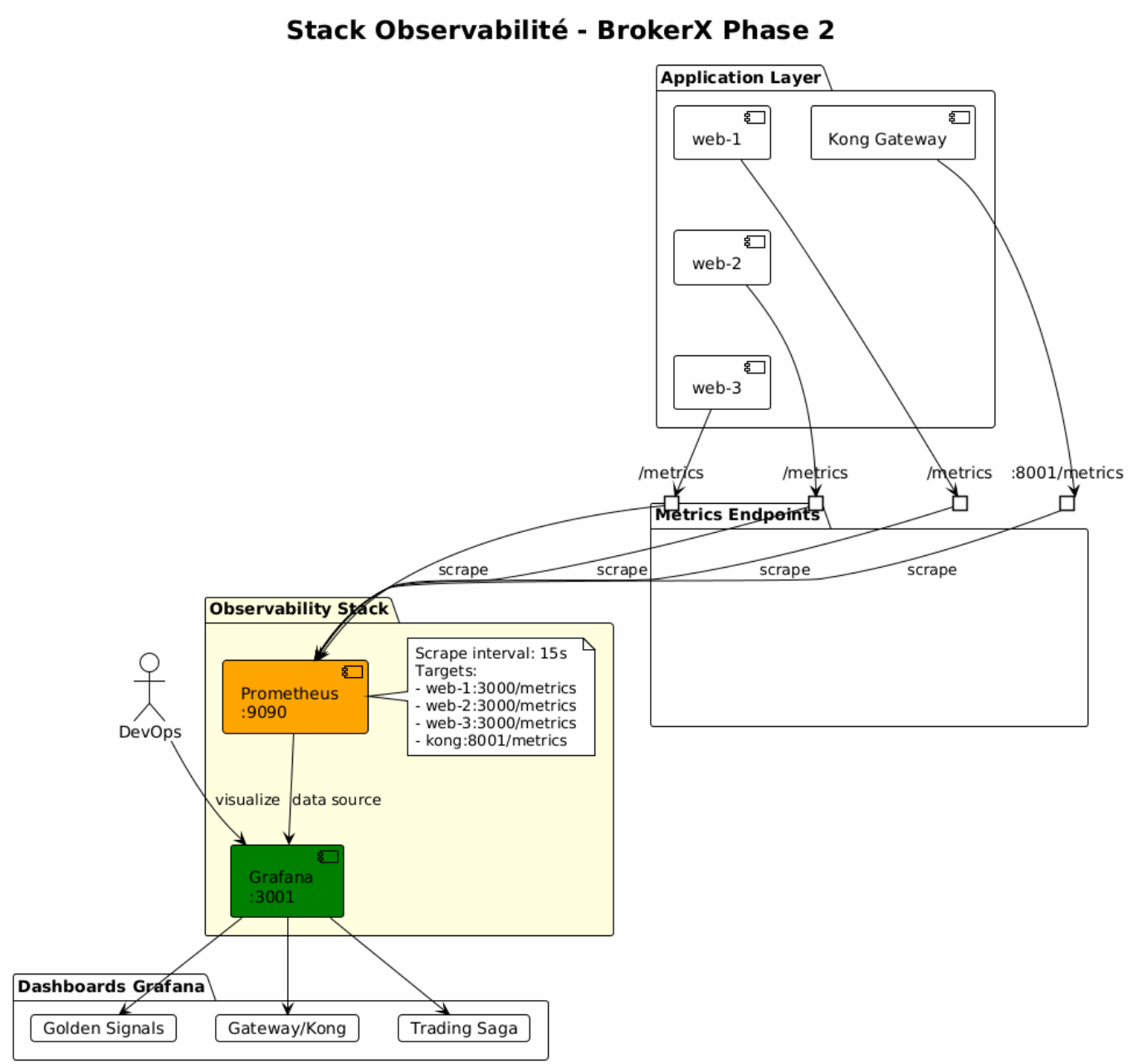
5.4 Résultats des tests k6 via Gateway

```
█ THRESHOLDS

    http_req_duration
    ✓ 'p(95)<600' p(95)=35.23ms

    http_req_failed
    ✓ 'rate<0.05' rate=0.00%

█ TOTAL RESULTS

    checks_total.......: 6760   56.24585/s
    checks_succeeded...: 66.56% 4500 out of 6760

    ✓ portfolio 200/401
    ✓ has X-Instance

    HTTP
    http_req_duration..: avg=16.47ms min=2.62ms med=13.94ms max=143.32ms
p(95)=35.23ms
    http_req_failed....: 0.00%  0 out of 4510
    http_reqs..........: 4510   37.524968/s
```

---

6. Métriques et Observabilité

6.1 Architecture Observabilité

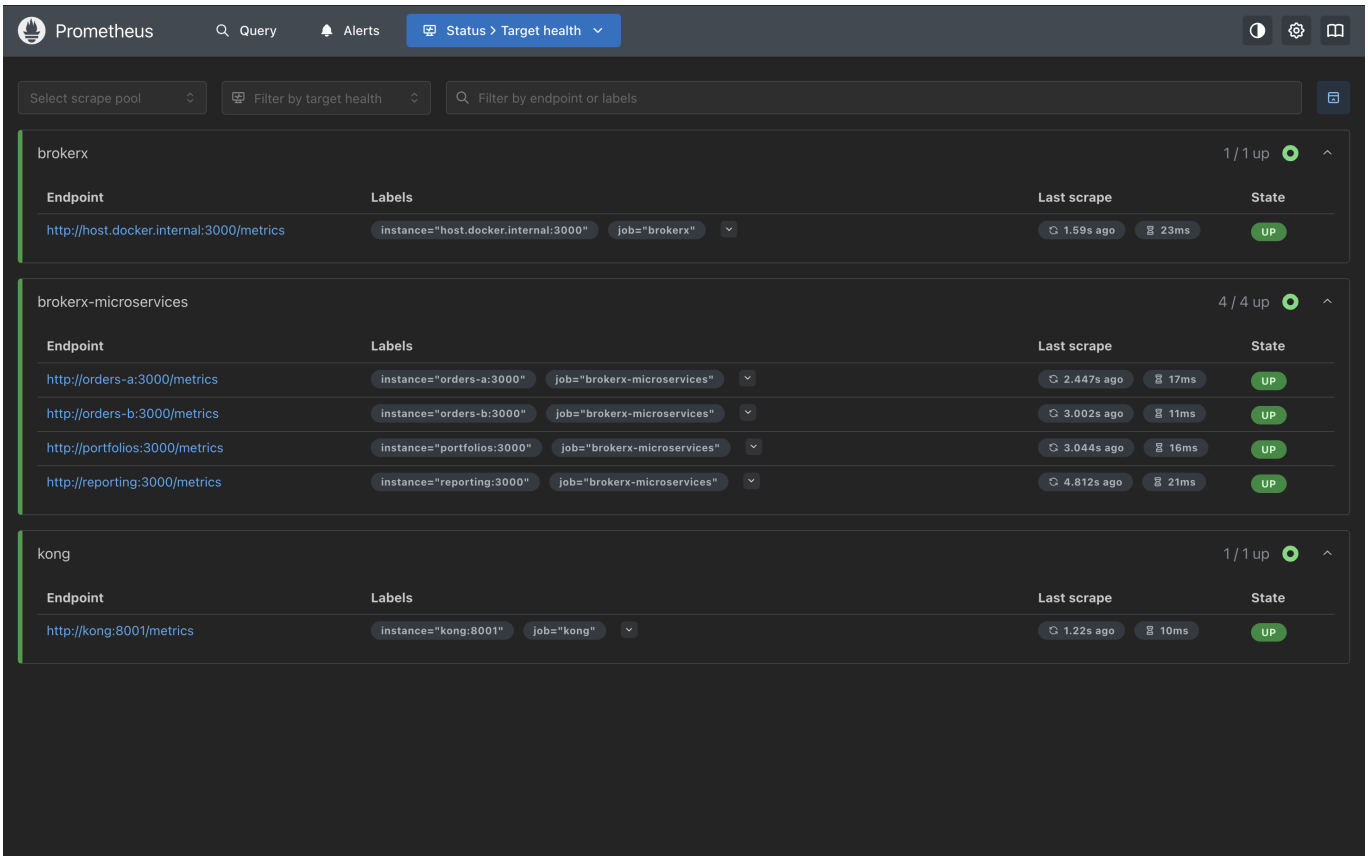


6.2 Nouvelles Métriques Prometheus

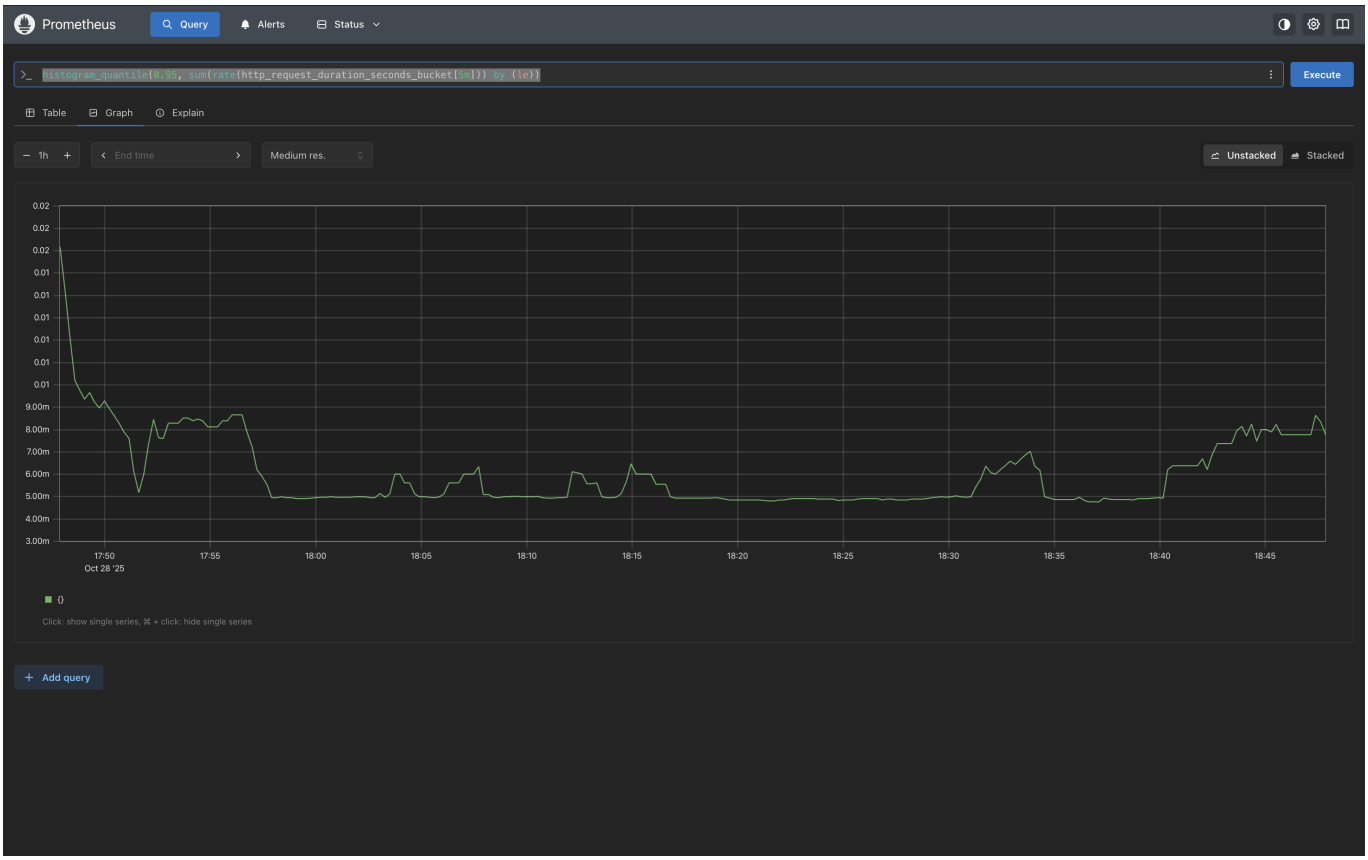
```
# Métriques TradingSaga
saga_started_total
saga_completed_total{status="success|failed"}
saga_compensations_total
saga_steps_total{step="...", status="completed|failed"}

# Métriques existantes enrichies
outbox_events_total{type="saga.*", status}
```

6.3 Captures Prometheus

Targets Prometheus (toutes les cibles UP):



Expression Latence P95:



6.4 Dashboards Grafana

Dashboard Golden Signals

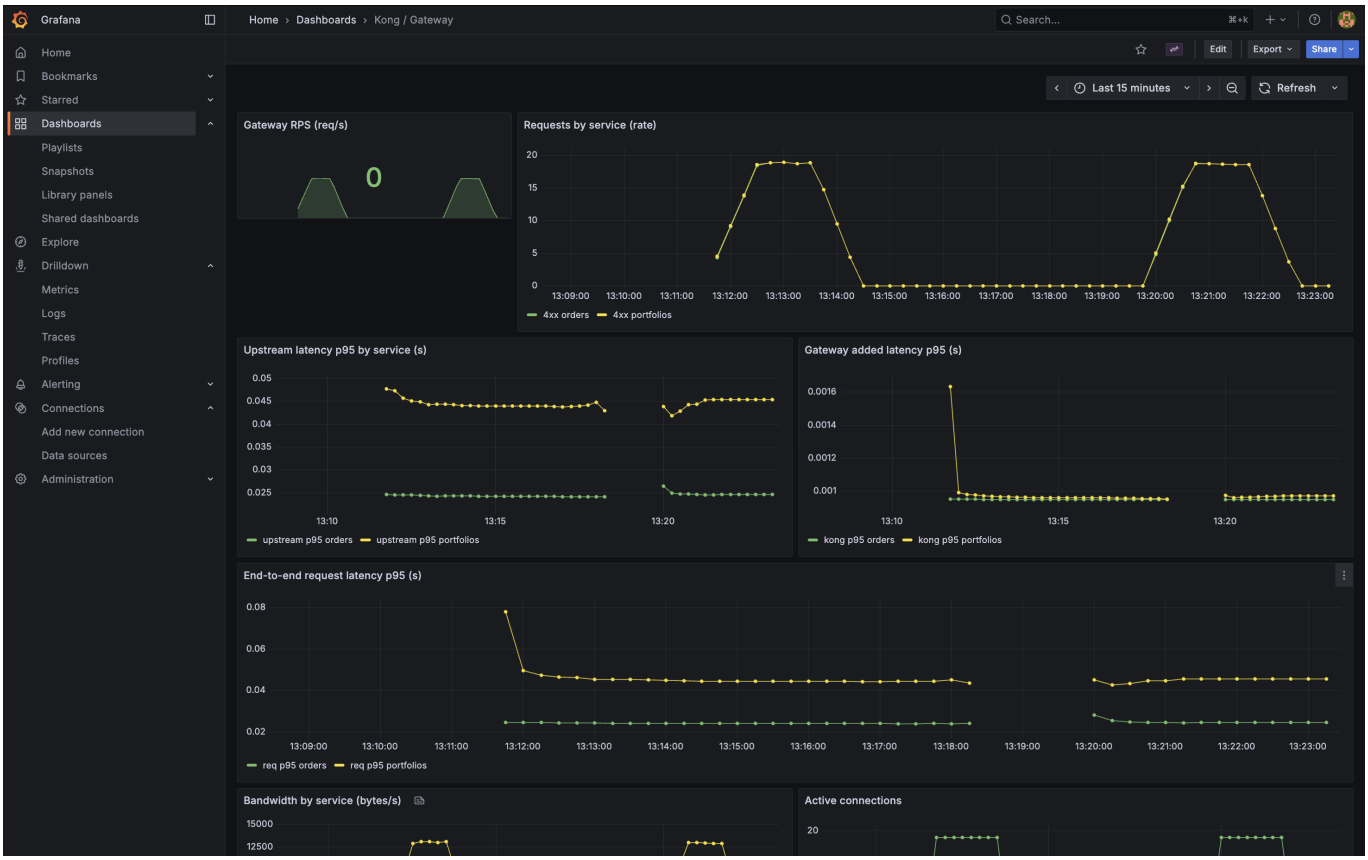
Vue d'ensemble des 4 Golden Signals (Latence, Trafic, Erreurs, Saturation):

Métriques observées:

- **RPS (req/s)**: ~2.93 requêtes par seconde
- **HTTP latency p95**: ~6-9ms (excellent)
- **HTTP by status**: Majorité 2xx (vert), quelques 4xx (orange) pour auth
- **ActionCable connections**: Connexions WebSocket actives

Dashboard Kong Gateway

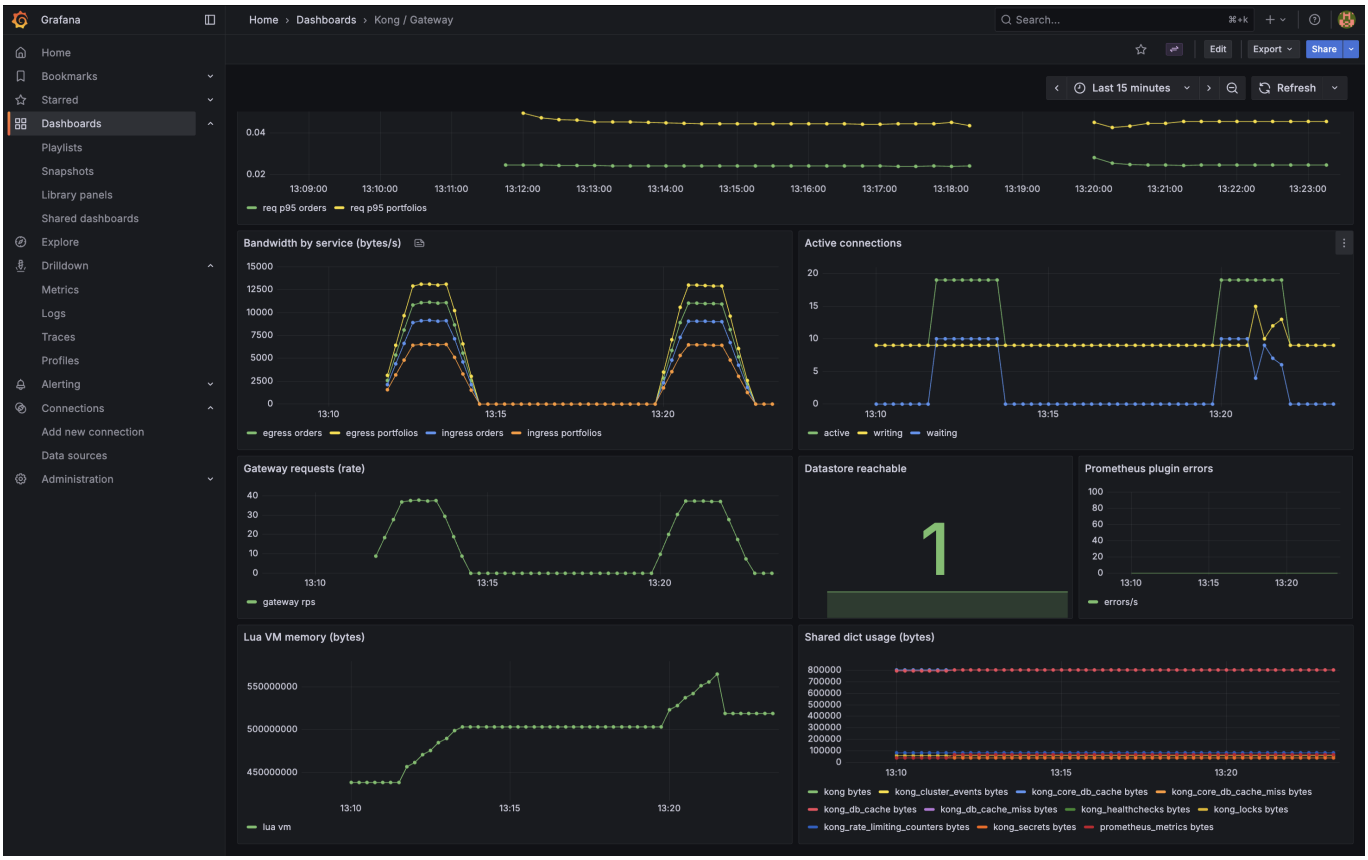
Vue globale du Gateway:



Métriques détaillées:

- **Gateway RPS**: Trafic à travers Kong
- **Requests by service**: Distribution entre orders et portfolios
- **Upstream latency p95**: ~40-50ms pour orders, ~25ms pour portfolios
- **Gateway added latency**: Overhead du gateway (~1-1.5ms)

Vue complète avec ressources système:



Métriques système:

- **Bandwidth by service**: ~10-15 KB/s egress/ingress
- **Active connections**: ~10-20 connexions actives
- **Gateway requests (rate)**: ~30-40 req/s pendant les tests
- **Datastore reachable**: (indicateur vert = 1)
- **Lua VM memory**: ~450-550 MB utilisés
- **Shared dict usage**: Cache Kong stable

6.5 Dashboards Recommandés

Dashboard	Métriques clés
Golden Signals	Latence p95, RPS, erreurs HTTP, ActionCable
Kong Gateway	Distribution services, overhead gateway, bandwidth
Trading Saga	Taux succès/échec, compensations, durée
Outbox	Événements pending/processed/failed

7. Déploiement

7.1 Commandes de déploiement

```
# Environnement complet (Gateway + Observabilité)
docker compose -f docker-compose.yml \
  -f docker-compose.gateway.yml \
```

```
  -f docker-compose.observability.yml up -d

# Environnement load balancé (prod-like)
docker compose -f docker-compose.lb.yml up -d

# Vérification
docker compose ps
```

7.2 Configuration requise

Variable	Description	Défaut
`REDIS_URL`	URL Redis	`redis://redis:6379/1`
`INSTANCE_ID`	ID de l'instance	hostname
`DATABASE_URL`	URL PostgreSQL	configuré dans compose

7.3 URLs des services

Service	URL	Credentials
Grafana	http://localhost:3001	admin/admin
Prometheus	http://localhost:9090	-
Kong Gateway	http://localhost:8080	API Key: `brokerx-key-123`
Kong Admin	http://localhost:8001	-
Web App	http://localhost:3000	-

---

8. Recommandations Futures

8.1 Améliorations à court terme

- [ ] Ajouter retry automatique pour les sagas échouées
- [ ] Implémenter les fills partiels (UC-07)
- [ ] Dashboard Grafana dédié aux sagas

8.2 Améliorations à moyen terme

- [ ] Migration vers message broker externe (Kafka/RabbitMQ)
- [ ] Auto-scaling basé sur les métriques
- [ ] Circuit breaker pour le MatchingEngine

8.3 Améliorations à long terme

- [ ] Découpage en microservices (TradingService, NotificationService)
- [ ] Event sourcing complet
- [ ] CQRS pour les lectures

---

9. Conclusion

Les implémentations Phase 3 apportent des améliorations significatives à BrokerX:

1. **Scalabilité**: 3 instances web derrière Nginx avec distribution équilibrée (~33% chacune)
2. **Résilience**: Saga Pattern garantit la cohérence des transactions avec compensation automatique
3. **Performance**: Cache Redis partagé entre instances, latence p95 < 40ms
4. **Observabilité**: Dashboards Grafana complets (Golden Signals + Kong Gateway)
5. **Traçabilité**: ADRs documentant chaque décision architecturale

Ces fondations permettent d'envisager sereinement l'évolution vers une architecture microservices tout en maintenant la cohérence métier requise pour une plateforme de courtage.

Annexes

A. Arborescence des fichiers créés/modifiés

```
brokerx/
├── docker-compose.lb.yml          # CRÉÉ - Orchestration LB
├── nginx/
│   └── nginx.conf                  # CRÉÉ - Config Nginx
├── app/
│   ├── application/services/
│   │   ├── trading_saga.rb         # CRÉÉ - Saga orchestrator
│   │   └── outbox_dispatcher.rb    # MODIFIÉ - Support saga.*
│   └── middleware/
│       └── instance_header_middleware.rb  # MODIFIÉ - INSTANCE_ID
├── config/observability/grafana/
│   └── provisioning/               # CRÉÉ - Auto-config Grafana
│       ├── datasources/
│       │   └── datasources.yml
│       └── dashboards/
│           └── dashboards.yml
├── test/unit/
│   └── trading_saga_test.rb        # CRÉÉ - 6 tests
├── load/k6/
│   ├── load.js                     # CRÉÉ - Test charge
│   ├── stress.js                   # CRÉÉ - Test stress
│   ├── lb_test.js                  # CRÉÉ - Test LB
│   └── README.md                   # CRÉÉ - Documentation
└── docs/
    ├── architecture/
    │   ├── adr008_redis_cache.md   # CRÉÉ
    │   ├── adr009_load_balancing.md # CRÉÉ
    │   ├── adr010_saga_pattern.md  # CRÉÉ
    │   └── arc42/arc42.md          # MODIFIÉ - Sections 9, 10, 11
    └── phase3/
        ├── rapport.md              # CE RAPPORT
        ├── puml/                   # Diagrammes PlantUML
        └── screenshots/            # Captures Grafana/Prometheus
```

B. Diagrammes PlantUML

Les diagrammes suivants sont disponibles dans `docs/phase3/puml/`:

Fichier	Description
`trading_saga_sequence.puml`	Séquence complète du TradingSaga avec compensation
`load_balancing_architecture.puml`	Architecture Nginx + 3 instances web
`outbox_event_flow.puml`	Flux Outbox Pattern (UC-07/UC-08)
`observability_stack.puml`	Stack Prometheus + Grafana

Régénérer les images PNG:

```
cd docs/phase3/puml
for f in *.puml; do
  docker run --rm -v "$(pwd)":/data plantuml/plantuml -tpng "/data/$f"
done
```

C. Références

- Arc42 Template
- Saga Pattern
- k6 Documentation
- Nginx Load Balancing
- PlantUML
- Grafana Provisioning