DAT – Document d'Architecture Technique du projet Amazoff

I. Vision et Objectifs de l'Architecture

L'architecture Amazoff a pour objectif principal de garantir la performance, la sécurité, la scalabilité et la maintenabilité de la plateforme e-commerce. La vision est de bâtir une architecture modulaire, évolutive et résiliente, inspirée des grands modèles microservices (à la manière de Netflix), afin que chaque fonctionnalité reste indépendante, robuste et simple à faire évoluer.

Concrètement, la plateforme doit être capable :

- D'assurer une haute disponibilité et de limiter les interruptions, même en cas de panne d'un service,
- De continuer à servir les utilisateurs lors des pics de trafic,
- De rester compréhensible et maintenable par une petite équipe,
- De garantir la fiabilité des opérations e-commerce critiques (ajout au panier, paiement, validation des commandes),
- De répondre à des exigences de sécurité minimales (authentification, protection des données sensibles, chiffrement),
- Et de permettre des mises à jour rapides et sûres grâce à l'automatisation des déploiements.

En résumé, Amazoff vise à offrir une expérience utilisateur fluide (temps de réponse de l'ordre de quelques centaines de millisecondes), tout en gardant une architecture solide, évolutive et prête pour le cloud.

II. Approche Modulaire et Évolutive

1. Architecture Microservices

Chaque domaine métier est isolé dans un microservice indépendant, disposant de sa propre base de données et de son environnement d'exécution. Parmi les principaux services :

- Articles / Catalogue (Python/Flask ou FastAPI) : gestion et consultation des produits,
- Panier (Python/Flask ou FastAPI) : suivi et mise à jour des paniers utilisateurs,
- Commandes (Python/Flask ou FastAPI) : validation et historique,
- Avis (Python/Flask ou FastAPI) : notation et commentaires,

- Utilisateurs (TypeScript/Node.js) : authentification et profils,
- Magasin (C++/Crow) : hautes performances sur des fonctionnalités critiques,
- Notifications (Node.js) : gestion des confirmations et alertes,
- Paiement (Node.js) : validation et suivi des transactions.

Ce découpage permet :

- Une résilience renforcée (une panne n'interrompt pas l'ensemble du site),
- Des déploiements indépendants (un service peut être mis à jour sans impacter les autres),
- Une réutilisation possible dans d'autres projets.

2. API-First et API Gateway

Toutes les communications passent par des API REST standardisées. Une API Gateway joue le rôle de point d'entrée unique :

- Elle route les requêtes vers le bon service,
- Gère l'authentification, le rate limiting et le logging,
- Met en cache certaines réponses pour améliorer la rapidité.

Ainsi, l'API Gateway masque la complexité interne et facilite l'intégration avec des services externes (ex. Stripe, PayPal).

3. Choix Technologiques

- Python (Flask/FastAPI) : pour la rapidité de prototypage,
- Node.js : pour la gestion des flux asynchrones (notifications temps réel, WebSockets),
- C++/Crow : pour les traitements intensifs en ressources (redimensionnement d'images, calculs de recommandations, chiffrage),
- Bases de données hybrides : MariaDB pour la fiabilité transactionnelle, MongoDB pour la flexibilité des données et la scalabilité horizontale.

III. Stratégie Cloud-First

1. Conteneurisation & Orchestration

Tous les services sont packagés dans des conteneurs Docker, chacun ayant son Dockerfile. Un fichier docker-compose.yml orchestre l'ensemble (microservices + bases de données) en une seule commande.

Cette approche garantit :

- La portabilité (reproductible sur n'importe quelle machine),
- La scalabilité (ajout d'instances en fonction de la charge),
- La haute disponibilité (services isolés et relançables indépendamment).

2. Infrastructure as Code (IaC)

L'infrastructure (réseaux, bases, services) est décrite dans des fichiers versionnés et automatisés via **scripts Bash et Makefile**. La prochaine étape consiste à intégrer un pipeline **CI/CD** (GitHub Actions, Terraform, etc.) pour déployer directement sur un cloud provider (AWS, GCP, Azure).

IV. Sécurité et Résilience Intégrées

- Sécurité by Design :
    - Authentification (standard JWT),
    - Mots de passe hachés et salés (crypto),
    - Chiffrement des données sensibles,
    - Gestion des rôles et permissions.

- Résilience :
    - Réplication des bases de données,
    - Redémarrage indépendant de chaque service en cas de panne.

- Gestion proactive des risques :
    - endpoints /health pour vérifier l'état des services.

V. Optimisation des Performances

- Scalabilité :
    - Verticale (ajout de ressources CPU/RAM sur MariaDB),
    - Horizontale (sharding MongoDB pour absorber plus de trafic).
- Caching & CDN : intégration future de Redis pour accélérer les requêtes fréquentes et d'un CDN pour les fichiers statiques.
- Monitoring : supervision en temps réel grâce à des métriques (MariaDB, MongoDB) et possibilité d'évoluer vers Prometheus + Grafana.

VI. Automatisations et DevOps

- CI/CD : automatisation du build, des tests et du déploiement,
- Tests unitaires et d'intégration pour chaque microservice,
- Configuration centralisée via .env,
- Scripts d'initialisation des bases de données.

Des tests manuels (via Postman) ont été menés pour vérifier les routes, simuler des erreurs et tester les intégrations entre services.

VII. Gouvernance et Conformité

- Respect des réglementations (RGPD),
- Gestion stricte des accès et permissions,
- Chiffrement des données sensibles,
- Traçabilité des actions via logs centralisés,
- Audits réguliers pour garantir la conformité.

VIII. Innovation et Agilité

- Stack multi-langages (Node.js, Python, C++) pour répondre aux besoins métier,
- Prototypage rapide grâce à des environnements isolés,
- Possibilité de tester de nouveaux services sans impacter la production,
- Intégration future d'API de paiement réelles (Stripe, PayPal).

IX. Livrables et Documentation

- Documents : DAT, README global, README spécifiques par microservice,
- Documentation technique : orchestration, configuration, scripts d'initialisation,
- Code source : organisé par microservice et frontend,
- Outils et scripts : fichiers Docker, Makefile, .env.exemples.

X. Bilan et Perspectives

Le projet Amazoff démontre qu'une architecture microservices, modulaire et cloud-ready peut être mise en place même par une petite équipe. Les principaux défis rencontrés concernaient :

- La complexité du C++ (Crow + BSON avec mongocxx),
- Le typage faible de Node.js nécessitant plus de validations,
- La synchronisation entre services Docker au démarrage.

Pour l'avenir, les priorités sont :

- Ajouter des tests automatisés (Jest, GoogleTest),
- Renforcer l'observabilité (Prometheus, Grafana),
- Intégrer des solutions de paiement réalistes (Stripe, PayPal),
- Améliorer la sécurité avancée (audits, chiffrement end-to-end).