Benchmark Technologique : Stack Optimal Bot Polymarket

Date : Janvier 2025 **Objectif :** Identifier les technologies les plus performantes pour un bot de trading Polymarket avec interface web de monitoring

---

1. Analyse du Code Existant (polymarket-arbitrage-bot)

1.1 Structure du Projet

Module	Lignes	Fonction
`client.py`	768	API REST (CLOB + Relayer gasless)
`websocket_client.py`	691	Streaming temps réel orderbook
`bot.py`	645	Interface trading principale
`market_manager.py`	512	Orchestration découverte marchés
`base_strategy.py`	459	Framework stratégies trading
`config.py`	446	Gestion configuration YAML/env
`signer.py`	348	Signatures EIP-712
`position_manager.py`	332	Tracking positions & P&L
`crypto.py`	285	Encryption clés (PBKDF2 + Fernet)
Total	**6,292**	Codebase Python complète

1.2 Librairies Python Actuelles

```
web3>=6.0.0              # Ethereum/Polygon interactions
eth-account>=0.5.0       # EIP-712 signing, account management
cryptography>=41.0.0     # PBKDF2 + Fernet encryption
websockets>=12.0         # WebSocket client async
requests>=2.28.0         # HTTP client REST API
pyyaml>=6.0              # Configuration YAML
python-dotenv>=1.0.0     # Variables environnement
```

1.3 Fonctionnalités Implémentées

- **WebSocket Streaming** : Orderbook, trades, prix temps réel
- **Gasless Trading** : Builder Program (Polygon)
- **Signatures EIP-712** : Ordres Gnosis Safe
- **Stratégie Flash Crash** : Détection drops de probabilité
- **Position Tracking** : Suivi P&L en temps réel
- **Auto-reconnect** : Exponential backoff WebSocket
- **Encryption** : Clés privées sécurisées (PBKDF2)

1.4 APIs Polymarket Utilisées

REST API (CLOB)

```
Base URL: https://clob.polymarket.com
- GET  /data/orders         # Liste ordres utilisateur
- GET  /order-book/{token}  # Orderbook d'un marché
- POST /order               # Placer un ordre
- DELETE /cancel-all        # Annuler tous les ordres
```

WebSocket

```
URL: wss://ws-subscriptions-clob.polymarket.com/ws/market
Messages: book, price_change, last_trade_price, tick_size_change
```

Relayer API (Gasless)

```
Base URL: https://relayer-v2.polymarket.com
- POST /order               # Ordre sans gas
```

1.5 Points de Performance Critiques

Composant	Latence Actuelle	Cible Optimale
Détection WebSocket	5-40 ms	< 10 ms
Message parsing	250-500 µs	< 20 µs
Order signing	< 1 ms	OK
HTTP round-trip	100-500 ms	Incompressible (réseau)

---

2. Benchmark Langages - Trading Engine

2.1 Performance WebSocket

Langage	Librairie	Throughput	Latence p99	GC Pauses
Rust	tokio-tungstenite	Excellent	3-10 ms	Aucun
C++	uWebSockets	Excellent	3-10 ms	Aucun
Go	gorilla/websocket	Très bon	10-20 ms	< 10 ms
Node.js	uWebSockets.js	Très bon	5-15 ms	Variable
Python	websockets	Moyen	20-50 ms	Variable

Observation clé : Node.js avec uWebSockets peut surpasser Rust/Go sur le throughput pur grâce à son modèle async optimisé, mais Rust offre une latence plus prévisible sans GC.

2.2 Performance High-Frequency Trading (HFT)

Langage	Latence/message	CPU Efficiency	Use Case Idéal
C++	6-12 µs	Maximale	Execution engines institutionnels
Rust	6-12 µs	Maximale	Order books, systèmes critiques
Go	~1 ms	Très bonne	Infrastructure cloud, services
Python	250-500 µs	Faible	Backtesting, recherche, prototypage

Constat : Rust offre **20-40x moins de latence** que Python pour le parsing de messages de marché.

2.3 Écosystème Web3/Ethereum

Langage	Librairie Principale	Maturité	Performance	Notes
Rust	Alloy (ex ethers-rs)	Production	Excellente	10x plus rapide ABI encoding, standard 2025
Go	go-ethereum (Geth)	Production	Très bonne	Client Ethereum de référence
Python	web3.py	Production	Moyenne	Flexible mais lent
TypeScript	ethers.js / viem	Production	Bonne	Excellent DX, écosystème riche

Important : ethers-rs est **deprecated**. Migrer vers **Alloy** qui offre :

- 60% de gain sur les opérations U256
- 10x plus rapide sur l'ABI encoding statique
- Support no_std pour embedded

---

3. Benchmark Frameworks - Dashboard Web

3.1 Frontend Frameworks (Real-time)

Framework	Bundle Size	Perf Real-time	Écosystème Charts	Learning Curve
Solid.js	~7 KB	Excellente	Moyen	Moyenne
Svelte 5	~3 KB	Excellente	Bon	Faible
React 19	~40 KB	Très bonne	Excellent	Moyenne
Vue 3	~30 KB	Très bonne	Très bon	Faible

Highlight : Svelte 5 génère des bundles **3x plus petits** que React et rend **60% plus vite** en conditions réelles.

3.2 Backend API Frameworks

Framework	Langage	Requests/sec	Latence p99	Async
Actix-web	Rust	~400,000	< 1 ms	Natif

Axum	Rust	~380,000	< 1 ms	Natif (tokio)
Fiber	Go	~300,000	~1 ms	Goroutines
Gin	Go	~280,000	~1 ms	Goroutines
FastAPI	Python	~70,000	~5 ms	asyncio

Constat : Axum (Rust) est **5x plus rapide** que FastAPI (Python) en conditions de charge.

3.3 Desktop App (Optionnel)

Framework	Langage Backend	Mémoire	Startup	Bundle Size
Tauri	Rust	30-40 MB	< 500 ms	2-3 MB
Electron	Node.js	100+ MB	1-2 s	80-120 MB

Avantage Tauri : 50% moins de mémoire, 3x plus rapide au démarrage, meilleure sécurité native.

---

4. Benchmark Bases de Données

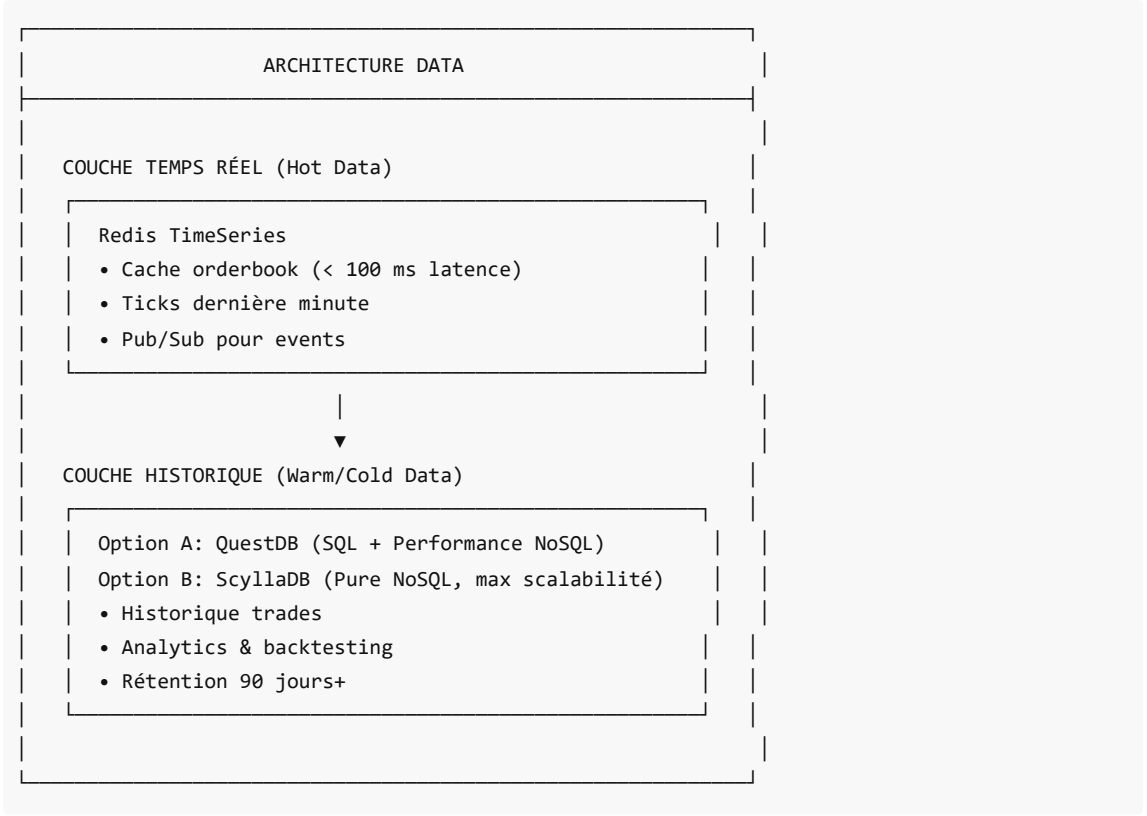
4.1 Time-Series Databases

Database	Ingestion	Query Performance	Langage Query	Type
QuestDB	4.3M rows/s	270% > TimescaleDB	SQL	Column-oriented
ScyllaDB	3M+ rows/s	Excellente	CQL	Wide-column NoSQL
TimescaleDB	~500K rows/s	Bonne (complex queries)	SQL	PostgreSQL extension
InfluxDB	~1M rows/s	Moyenne	Flux	Time-series natif
Redis TS	Millions/s	Ultra-rapide	Redis commands	In-memory

4.2 Comparatif SQL vs NoSQL pour Trading

Critère	SQL (QuestDB/TimescaleDB)	NoSQL (ScyllaDB/Redis)
Requêtes complexes	Excellent (JOINs, agrégations)	Limité
Ingestion haute fréquence	Très bon	Excellent
Scalabilité horizontale	Limitée	Excellente
Flexibilité schéma	Rigide	Très flexible
Analytics	Excellent	Basique
Latence lecture	< 10 ms	< 1 ms

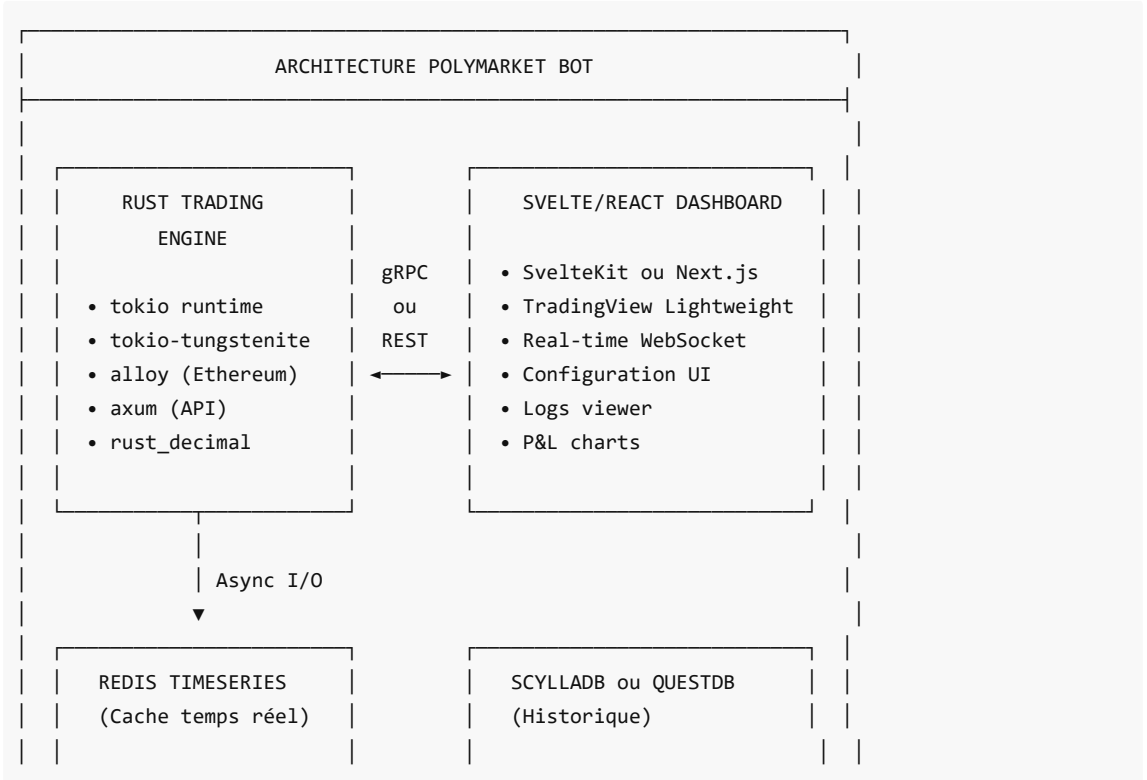
4.3 Recommandation Architecture Data

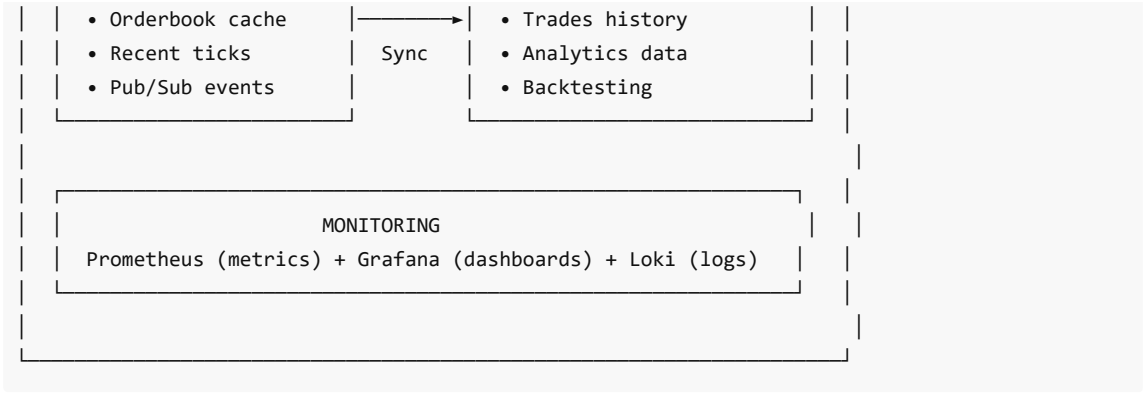


---

5. Recommandation Finale : Stack Optimal

5.1 Architecture Globale





5.2 Stack Détaillé

Trading Engine (Rust)

```toml
# Cargo.toml
[dependencies]
# Runtime async
tokio = { version = "1.35", features = ["full"] }

# WebSocket haute performance
tokio-tungstenite = "0.24"

# Ethereum/Polygon (remplace ethers-rs deprecated)
alloy = "1.0"

# API REST/gRPC
axum = "0.7"
tonic = "0.12"                    # gRPC (optionnel)

# Sérialisation
serde = { version = "1.0", features = ["derive"] }
serde_json = "1.0"

# Précision financière (pas de floating point errors)
rust_decimal = "1.33"

# Base de données
sqlx = { version = "0.8", features = ["runtime-tokio", "postgres"] }
redis = { version = "0.27", features = ["tokio-comp"] }
scylla = "0.14"                   # Si choix ScyllaDB

# Observabilité
tracing = "0.1"
tracing-subscriber = "0.3"
metrics = "0.23"
metrics-exporter-prometheus = "0.15"

# Crypto
sha2 = "0.10"
```

```
hmac = "0.12"

# Utils
anyhow = "1.0"
thiserror = "1.0"
chrono = { version = "0.4", features = ["serde"] }
```

Dashboard Frontend (TypeScript)

```
{
  "dependencies": {
    // Framework (choisir un)
    "@sveltejs/kit": "^2.0",
    // OU "next": "^15.0",

    // Charts trading
    "lightweight-charts": "^4.2",

    // Data fetching & state
    "@tanstack/svelte-query": "^5.0",

    // WebSocket
    "socket.io-client": "^4.7",

    // UI
    "tailwindcss": "^4.0",
    "bits-ui": "^0.21",

    // Tables & data display
    "@tanstack/svelte-table": "^8.0"
  }
}
```

Infrastructure (Docker Compose)

```
version: '3.8'

services:
  trading-engine:
    build: ./rust-engine
    environment:
      - RUST_LOG=info
      - REDIS_URL=redis://redis:6379
      - SCYLLA_HOSTS=scylla:9042
    depends_on:
      - redis
      - scylla
    restart: always

  dashboard:
```

```
    build: ./dashboard
    ports:
      - "3000:3000"
    depends_on:
      - trading-engine

  redis:
    image: redis/redis-stack:latest
    ports:
      - "6379:6379"
    volumes:
      - redis_data:/data

  scylla:
    image: scylladb/scylla:5.4
    ports:
      - "9042:9042"
    volumes:
      - scylla_data:/var/lib/scylla
    command: --smp 2 --memory 2G

  prometheus:
    image: prom/prometheus:v2.50.0
    ports:
      - "9090:9090"
    volumes:
      - ./prometheus.yml:/etc/prometheus/prometheus.yml

  grafana:
    image: grafana/grafana:10.3.0
    ports:
      - "3001:3000"
    volumes:
      - grafana_data:/var/lib/grafana

volumes:
  redis_data:
  scylla_data:
  grafana_data:
```

---

6. Gains Attendus vs Python Actuel

Métrique	Python Actuel	Rust Proposé	Gain
Latence WebSocket	20-50 ms	3-10 ms	**5x**
Message parsing	250-500 µs	6-12 µs	**20-40x**
Memory usage	~200 MB	~50 MB	**4x**
Throughput messages	~10K msg/s	~100K msg/s	**10x**

Startup time	~2 s	~100 ms	**20x**
Binary size	~50 MB (venv)	~5 MB	**10x**

---

7. Roadmap d'Implémentation

Phase 1 : Core Trading Engine (4-6 semaines)

Semaine 1-2 : Setup & WebSocket

- [ ] Setup projet Rust (Cargo workspace)
- [ ] Implémenter WebSocket client Polymarket
- [ ] Gestion reconnexion automatique
- [ ] Parsing orderbook optimisé

Semaine 3-4 : Trading Logic

- [ ] Intégration Alloy (Ethereum/Polygon)
- [ ] Signatures EIP-712
- [ ] API REST client (CLOB + Relayer)
- [ ] Placement/annulation ordres

Semaine 5-6 : Stratégies & Data

- [ ] Framework stratégies (trait-based)
- [ ] Intégration Redis (cache)
- [ ] Intégration ScyllaDB (historique)
- [ ] Flash Crash strategy port

Phase 2 : API & Dashboard (4-6 semaines)

Semaine 7-8 : API Backend

- [ ] API Axum (REST + WebSocket)
- [ ] Endpoints configuration
- [ ] Endpoints monitoring
- [ ] Authentication (JWT ou API keys)

Semaine 9-10 : Dashboard Frontend

- [ ] Setup SvelteKit
- [ ] Intégration TradingView charts
- [ ] Real-time orderbook display
- [ ] Configuration UI

Semaine 11-12 : Logs & Monitoring

- [ ] Logs viewer avec filtres
- [ ] P&L dashboard
- [ ] Alertes (Discord/Telegram)
- [ ] Intégration Prometheus/Grafana

Phase 3 : Production (2-3 semaines)

Semaine 13-14 : Containerization & CI/CD

- [ ] Dockerfiles optimisés (multi-stage)
- [ ] Docker Compose complet
- [ ] GitHub Actions CI/CD
- [ ] Tests automatisés

Semaine 15 : Déploiement

- [ ] Setup VPS (TradingVPS ou équivalent)
- [ ] Monitoring production
- [ ] Documentation opérationnelle
- [ ] Runbooks incidents

---

8. Risques & Mitigations

Risque	Impact	Probabilité	Mitigation
Courbe apprentissage Rust	Moyen	Haute	Documentation, exemples, pair programming
Breaking changes Alloy	Faible	Moyenne	Pin versions, tests de régression
Latence réseau incompressible	Haute	Certaine	VPS proche des serveurs Polymarket
Rate limiting Polymarket	Haute	Moyenne	Backoff exponentiel, monitoring
ScyllaDB complexité opérationnelle	Moyen	Moyenne	Managed service ou QuestDB comme alternative

---

9. Alternatives Considérées

Option B : Go Full-Stack

```
Avantages:
- Courbe apprentissage plus douce que Rust
- Excellent écosystème (Geth, go-ethereum)
- Compilation rapide
- Concurrence native (goroutines)

Inconvénients:
- GC pauses (< 10ms mais présentes)
- Performance légèrement inférieure à Rust
- Moins de garanties mémoire

Verdict: Bon compromis si l'équipe n'est pas à l'aise avec Rust
```

Option C : Node.js/Bun avec TypeScript

```
Avantages:
- Stack unifié frontend/backend
- Excellent écosystème (ethers.js, viem)
- Développement rapide
- Facile à recruter

Inconvénients:
- Latence supérieure pour HFT
- GC imprévisible
- Single-threaded (worker threads complexes)

Verdict: Acceptable pour trading moyenne fréquence
```

---

10. Sources & Références

Performance & Benchmarks

- WebSocket Performance Comparison - Medium (Matt Tomasetti)
- 2024 Fastest Web Servers Benchmark - Medium
- Rust vs Go vs Bun vs Node.js 2024 - DEV Community
- HFT Language Comparison - LinkedIn
- MEV Bot Speed Comparison - Medium (Solid Quant)

Web3 & Ethereum

- Alloy v1.0 Release - Paradigm (Mai 2025)
- Rust vs Go for Blockchain - Medium
- Web3 Programming Languages 2024 - aelf

Frontend & Dashboard

- Svelte 5 vs React 19 vs Vue 4 Benchmarks - JSGuruJobs
- JavaScript Framework Comparison 2025 - Calmops

Backend & API

- FastAPI vs Axum Benchmark - Luke Hsiao
- Rust Web Frameworks Compared - DEV Community

Databases

- QuestDB vs TimescaleDB vs InfluxDB - QuestDB Blog
- TimescaleDB vs QuestDB Benchmarks - QuestDB

Desktop Apps

- Tauri vs Electron Comparison - Hopp Blog

Document généré le : Janvier 2025 **Version :** 1.0