Building Real-Time AI Fraud Prevention Systems

for Gig Platforms in Rust

Prabhakar Singh, Senior Software Engineer at Meta

Agenda

The Gig Economy Fraud Challenge

Unique challenges, millisecond decision requirements, and why traditional solutions fall short

Why Rust for Fraud Prevention

Memory safety, performance, and fearless concurrency advantages for security-critical systems

Multi-Layered Architecture

Building with Tokio, Candle ML, Petgraph and implementing stream processing, zero-copy deserialization

Real-World Results & Future Applications

Case studies from food delivery and ride-sharing platforms, plus emerging applications

The Gig Economy Fraud Challenge

Millisecond Decision Windows

Fraud prevention systems must approve/reject transactions within 50-100ms to maintain seamless user experience

Decentralized Operations

Widely distributed workforce and customers create complex geographic fraud patterns

Instant Payment Cycles

Immediate cashouts mean potential fraud losses cannot be recovered post-transaction

AI Processing Requirements

ML models require high-throughput processing of thousands of transactions per minute

Legacy systems built with Python, Java, or Node.js often struggle with these requirements, creating painful tradeoffs between speed and accuracy.

Common Fraud Patterns in Gig Platforms

- **Account Takeover:** Legitimate accounts compromised through credential stuffing
- **Synthetic Identity:** Fabricated identities created by combining real and fake information
- **Worker-Customer Collusion:** Coordinated fraud between delivery drivers and customers
- **GPS Spoofing:** Falsifying location data to manipulate ride/delivery assignments
- **Payment Method Fraud:** Using stolen payment methods for immediate cashouts

These patterns evolve rapidly, requiring systems that can adapt quickly to new fraud vectors while maintaining high performance.

Why Rust for Fraud Prevention?

Memory Safety Without Garbage Collection

Eliminates entire classes of bugs (buffer overflows, use-after-free) that can compromise fraud systems without runtime penalty

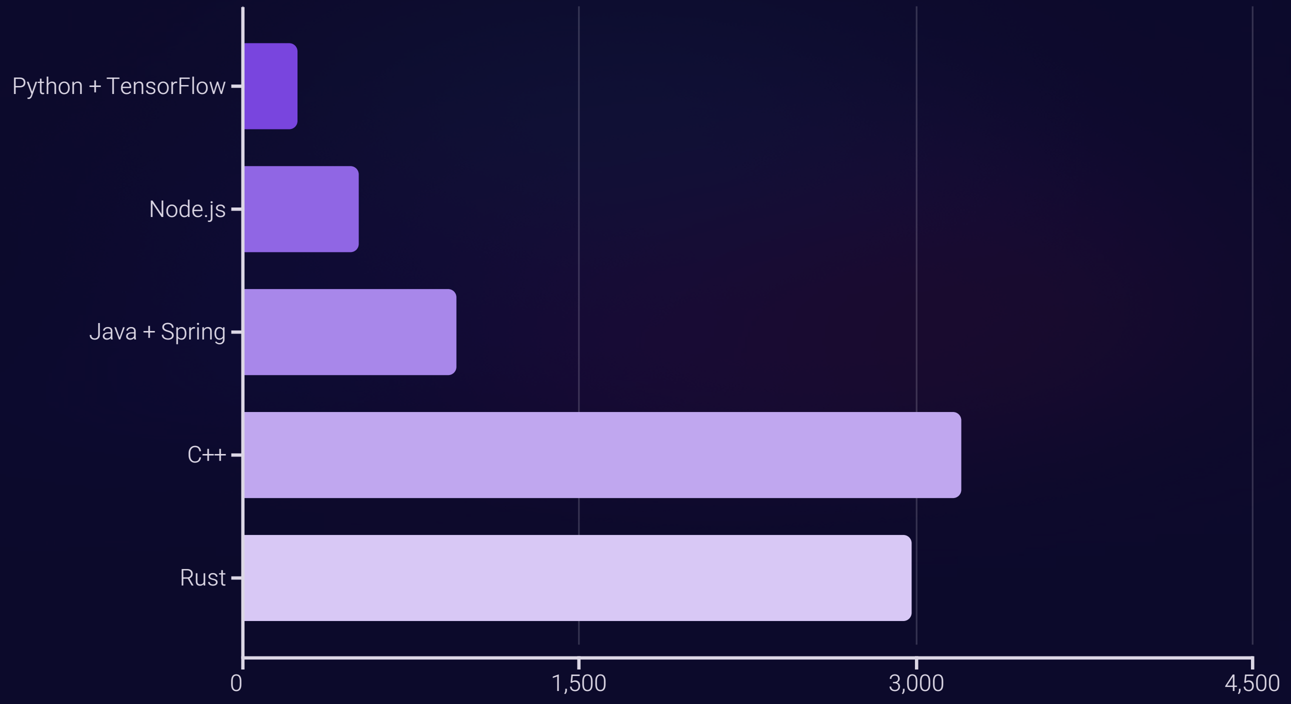
C++-Level Performance

Zero-cost abstractions and LLVM-powered optimizations deliver processing speeds rivaling C++ with development velocity closer to Python

Fearless Concurrency

Ownership and borrowing model prevents data races in concurrent processing, critical for high-throughput fraud detection

Real-World Results: Performance Comparison



Key Metrics from Production

- Rust achieves 93% of C++'s raw performance
- 6.2x faster than Java implementation
- 12.4x faster than Python with TensorFlow
- Consistent p99 latency under 35ms
- 90% reduction in CPU utilization compared to Python

Data from benchmark tests on a major food delivery platform processing 50,000+ orders per hour.

Multi-Layered Rust Architecture for Fraud Prevention

1. **Real-Time Stream Processing with Tokio**

   Asynchronous processing of transaction streams using Tokio and Futures, drastically reducing detection latency

2. **Rule-Based Detection Layer**

   High-speed pattern matching using Rust's match expressions and custom DSLs

3. **Statistical Anomaly Detection**

   Time-series analysis with statistical models implemented in pure Rust

4. **ML Classification with Candle**

   Fast ML inference using Candle for deep learning models with CUDA acceleration

5. **Graph Analysis with Petgraph**

   Network analysis for detecting coordinated fraud rings

Rust-Powered Innovations

Async Stream Processing

Tokio and Futures enable processing thousands of transactions per second with minimal latency

```
// Transaction processing with Tokio
let mut stream = StreamExt::throttle(
    transactions, Duration::from_millis(1)
);

while let Some(tx) = stream.next().await
{
    let verdict = detect_fraud(tx).await?;
    decisions.push(verdict).await?;
}
```

Zero-Copy Deserialization

Serde implementations with zero-copy parsing for high-throughput JSON/Protobuf data

```
// Zero-copy deserialization with Serde
#[derive(Deserialize)]
struct Transaction<'a> {
    #[serde(borrow)]
    user_id: &'a str,
    amount: f64,
    #[serde(borrow)]
    device_id: &'a str,
}
```

WASM-Compiled Modules

Secure, portable fraud detection logic that runs consistently across diverse environments

ML Inference with Candle

Why Candle for ML in Fraud Detection?

- Pure Rust ML framework - memory safety throughout the stack
- CUDA and Metal acceleration for GPU inference
- Supports ONNX model imports from PyTorch/TensorFlow
- No Python dependencies in production
- Tight integration with Rust's type system

Graph Analysis for Fraud Networks

- **Detecting Coordinated Fraud with Petgraph**

  Rust's Petgraph crate enables efficient graph algorithms for finding fraud networks:

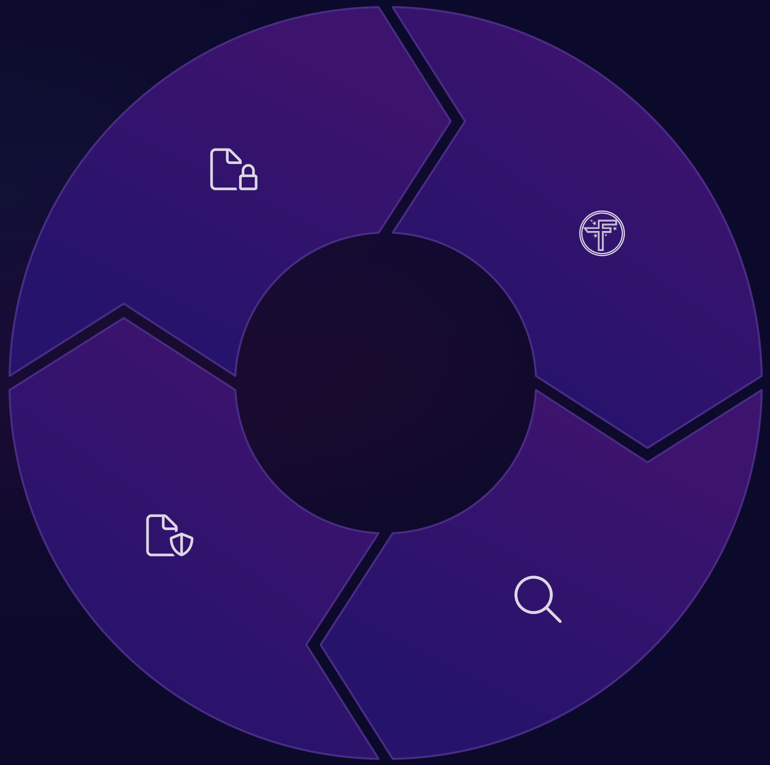
  - Connected component analysis reveals collusion between workers and customers
  - Centrality measures identify key nodes in fraud networks
  - Graph embeddings detect structural similarities in transaction patterns
  - Temporal graph analysis tracks evolving fraud patterns over time

- **Real-World Example: Driver-Customer Collusion**

  A food delivery platform detected a sophisticated fraud ring involving 32 delivery drivers and 18 customers placing fake orders. Graph analysis revealed the pattern when standard ML models missed it.

```
// Find suspicious components in the graph
let components = petgraph::algo::kosaraju_scc(&graph);
for component in components {
 if component.len() > 5 {
 analyze_potential_fraud_ring(component);
 }
}
```

Privacy-Preserving Fraud Detection



Homomorphic Encryption

Rust implementations of FHE libraries allow computation on encrypted data without decryption

Federated Learning

Distributed ML training across devices/regions without centralizing sensitive data

Secure Enclaves

Rust code running in TEEs provides hardware-level isolation for sensitive fraud logic

Differential Privacy

Adding calibrated noise to protect individual privacy while preserving statistical utility

Rust's strong type system and zero-cost abstractions make it possible to implement these advanced privacy techniques without prohibitive performance penalties.

Case Study: Major Ride-Sharing Platform

Challenge

Platform was losing $3.2M monthly to sophisticated fraud schemes including GPS spoofing and synthetic identity creation.

Rust Solution

Replaced Python-based system with Rust microservices using:

- Tokio for async HTTP API and stream processing
- Custom Rust implementation of device fingerprinting
- WASM modules for distributed fraud rules
- Candle for ML model inference

Results

87% reduction in fraud losses, 65% decrease in false positives, and 99.98% uptime over 12 months.

Emerging Applications in Rust Fraud Prevention

1. Behavioral Biometrics (Now)

Rust processing of touch dynamics, typing patterns, and device motion signals to authenticate users continuously without explicit verification steps

2. Explainable AI Pipelines (6-12 months)

Rust implementations of SHAP, LIME, and custom explainability tools using Candle and Burn ML frameworks to provide human-interpretable fraud detection decisions

3. Advanced FHE (12-24 months)

Practical homomorphic encryption implementations in Rust providing order-of-magnitude performance improvements, enabling private computation on sensitive fraud signals

4. GPGPU Acceleration (24+ months)

Native Rust GPU programming for fraud detection using emerging frameworks like Rust-GPU and Candle's expanded CUDA capabilities

Key Takeaways

Performance Matters

Rust delivers near-C++ performance with development velocity approaching higher-level languages, critical for millisecond-scale fraud decisions

Safety Is Non-Negotiable

Memory safety guarantees and concurrency without data races eliminate entire classes of vulnerabilities in security-critical systems

Ecosystem Is Maturing

Crates like Tokio, Candle, and Petgraph now provide production-ready foundations for sophisticated fraud prevention systems