**Implementation Guide: Minimalistic Crypto Betting System**

This guide explains how Python implementation demonstrates the key architectural patterns from the article [linkedin article link]. I have created a simplified crypto betting platform that shows event-driven architecture and high throughput patterns in action.

**How to Run the Demo**

1. Download/pull the repository
2. Look for the file named crypto\_betting\_system.py
3. Make sure you have Python 3.7+ installed
4. Run with: python crypto\_betting\_system.py

The demo will:

1. Initialize the system components
2. Get a user's account details
3. Check current cryptocurrency odds
4. Place a bet on Bitcoin
5. Process the bet asynchronously
6. Simulate price updates
7. Show bet settlement and balance updates

**Key Architectural Patterns Demonstrated**

**1. Event-Driven Architecture**

# In EventBus class

def publish(self, event\_type: str, payload: dict) -> str:

"""Publish an event to the bus"""

event\_id = str(uuid.uuid4())

event = Event(

event\_id=event\_id,

event\_type=event\_type,

timestamp=datetime.now(),

payload=payload

)

# Persist event first (event sourcing pattern)

self.\_db.write\_event(event)

# Then process asynchronously

asyncio.create\_task(self.\_process\_event(event))

return event\_id

This demonstrates:

* Asynchronous communication between services
* Event persistence for durability
* Publish/subscribe pattern

**2. Fast Path/Slow Path Separation**

# In BettingService class

async def place\_bet(self, user\_id: str, asset: CryptoAsset, direction: BetDirection, amount: float):

# Fast validations first

user = await self.\_user\_service.get\_user(user\_id)

if not user:

return {"success": False, "error": "User not found"}

# ...validation logic...

# Save bet to database

self.\_db.write\_bet(bet)

# Publish event for async processing

self.\_event\_bus.publish("bet\_placed", {...})

# Return fast response

return {

"success": True,

"bet\_id": bet\_id,

"message": "Bet accepted, processing payment"

}

This demonstrates:

* Quick validations in the request path
* Immediate response to the client
* Complex processing moved to asynchronous flow

**3. Read/Write Separation**

# In Database class

def write\_user(self, user: User) -> None:

"""Write to primary storage"""

self.\_users[user.user\_id] = user

# Simulate eventual consistency

asyncio.create\_task(self.\_update\_read\_replica\_user(user))

# Read operations (read replicas)

def read\_user(self, user\_id: str) -> Optional[User]:

"""Read from replica"""

return self.\_read\_users.get(user\_id)

This demonstrates:

* Separate paths for reads and writes
* Eventual consistency model
* Optimized read performance

**4. Caching Strategy**

# In UserService class

async def get\_user(self, user\_id: str) -> Optional[User]:

"""Get user with cache first strategy"""

# Try cache first (fast path)

cache\_key = f"user:{user\_id}"

cached\_user = await self.\_cache.get(cache\_key)

if cached\_user:

return cached\_user

# Cache miss, get from DB (slow path)

user = self.\_db.read\_user(user\_id)

if user:

# Update cache for next time

await self.\_cache.set(cache\_key, user)

return user

This demonstrates:

* Cache-first strategy
* Cache invalidation on writes
* TTL-based caching for volatile data

**5. Service Decomposition**

The system is decomposed into specialized services:

* UserService: Manages user accounts and balances
* OddsService: Handles odds calculation and pricing
* BettingService: Core betting operations
* SettlementService: Asynchronous bet settlement
* MarketDataService: Simulates real-time price updates

**6. API Gateway Pattern**

# In APIGateway class

async def handle\_request(self, endpoint: str, method: str, data: dict) -> Dict[str, any]:

"""Handle incoming API requests"""

# Rate limiting

client\_id = data.get("client\_id", "anonymous")

if not self.\_check\_rate\_limit(client\_id):

return {"error": "Rate limit exceeded", "status\_code": 429}

# Route to appropriate handler

if endpoint == "/api/bet" and method == "POST":

return await self.\_handle\_place\_bet(data)

# ...other endpoints...

This demonstrates:

* Centralized request handling
* Rate limiting
* Routing to appropriate services

**Scale-Related Features**

While simplified, this implementation demonstrates several key patterns for high throughput systems:

1. **Non-blocking operations**:
   * All service methods use async/await
   * No synchronous blocking between services
2. **Resilience patterns**:
   * Event persistence for recovery
   * Optimistic concurrency control
   * Circuit breaker pattern (simulated)
3. **Eventual consistency**:
   * Services operate independently
   * State propagates asynchronously
   * Changes are eventually visible system-wide

**How This Relates to the Article**

This implementation illustrates the key points from the article:

1. **Event-driven communication**: Services communicate through events, not direct calls
2. **Microservices architecture**: Each service has a clear responsibility
3. **Fast path/slow path separation**: Critical operations return quickly
4. **Caching strategy**: Performance optimization via caching
5. **Read/write separation**: Optimized paths for different operations

The code demonstrates why event-driven architecture is superior for high throughput systems - it enables:

* Independent scaling of services
* Fault isolation
* Asynchronous processing
* Better resource utilization

**Extending the Implementation**

This demo could be extended to handle greater scale by:

1. Replacing in-memory components with real databases and message brokers
2. Adding proper error handling and retry logic
3. Implementing more sophisticated odds calculation
4. Adding monitoring and observability
5. Implementing proper authentication and security

Even in this simplified form, it shows the core architectural patterns that make high throughput systems scalable and resilient.