Smart Order Routing for Digital Securities: Multi-Provider Execution on Lux Network

Zach Kelling

Lux Industries

2025

Abstract

We present the smart order routing (SOR) system for the Lux ATS, a broker-dealer component that routes digital securities orders across 16 execution providers to achieve best execution under SEC Regulation NMS and FINRA Rule 5310. The router is implemented in Go on Hanzo Base and integrates with traditional equity venues (Alpaca, Interactive Brokers), digital securities ATS platforms, and the on-chain Lux DEX VM for native order matching. This work extends the `exchange-sdk` originally developed at Arca (2018) for digital securities trading, adding multi-venue routing, compliance-aware execution, and atomic settlement on Lux EVM. Orders are serialized using the ZAP binary wire protocol for sub-millisecond transmission latency. The router implements a constrained optimization that maximizes execution quality (price improvement, fill rate, speed) subject to compliance constraints (transfer restrictions, jurisdictional rules, accreditation requirements). We formalize the best execution obligation as a multi-objective optimization problem, prove that the greedy allocation algorithm achieves a 2-approximation of the optimal solution, and present benchmarks showing median order-to-fill latency of 12ms for on-chain venues and 45ms for traditional equity providers.

1 Introduction

A registered broker-dealer has a legal obligation to seek the best execution reasonably available for customer orders. Under FINRA Rule 5310, this requires "reasonable diligence to ascertain the best market for the subject security and buy or sell in such market so that the resultant price to the customer is as favorable as possible under prevailing market conditions." For digital securities that trade across multiple venues—on-chain ATS platforms, traditional equity exchanges (for hybrid securities), and bilateral RFQ networks—satisfying this obligation requires a smart order router that evaluates execution quality across all available venues in real time.

1.1 Heritage: Arca exchange-sdk

The Arca `exchange-sdk` (2018) was a trading SDK for digital securities that provided order book management, matching engine integration, and settlement coordination for ArCoin and other Arca Fund securities. The SDK supported a single venue (the Arca internal matching engine) and a single asset class (ERC-1400 security tokens).

The Lux SOR extends this foundation to support:

- 16 execution providers across three venue types (on-chain, ATS, traditional).
- Multiple asset classes (security tokens, tokenized equities, structured products).
- Compliance-constrained routing that respects transfer restrictions per security [2].
- Atomic settlement on Lux EVM for on-chain executions [1].
- ZAP binary wire protocol for sub-millisecond order serialization [4].

1.2 Contributions

1. Formalization of the best execution obligation as a constrained multi-objective optimization problem over venue selection.
2. A greedy allocation algorithm with provable 2-approximation guarantee.
3. Integration architecture for 16 providers spanning on-chain, ATS, and traditional venues.
4. Compliance-aware routing that respects per-security transfer restrictions at the venue selection layer.
5. Sub-millisecond order serialization via ZAP binary protocol.

2 Provider Landscape

2.1 Execution Providers

The SOR integrates with 16 execution providers across three categories:

Table 1: Execution provider registry.

#	Provider	Type	Asset Classes
1	Lux DEX VM	On-chain	Security tokens, tokenized equities
2	Lux ATS (internal)	On-chain	Security tokens
3	Alpaca	Traditional	Listed equities, fractional shares
4	Interactive Brokers	Traditional	Equities, options, futures
5	Tradier	Traditional	Equities, options
6	DriveWealth	Traditional	Fractional equities
7	tZERO ATS	ATS	Digital securities
8	Securitize Markets	ATS	Digital securities
9	INX	ATS	Digital securities
10	OpenFinance	ATS	Digital securities
11	Rialto Markets	ATS	Digital securities
12	MERJ Exchange	ATS	Digital securities, equities
13	Archax	ATS	Digital securities (UK)
14	SDX (SIX Digital)	ATS	Digital securities (CH)
15	Tokeny (T-REX)	ATS	Digital securities (EU)
16	Republic/Carta	ATS	Private securities

2.2 Provider Interface

Each provider implements a common interface:

Listing 1: Provider interface.

```
type Provider interface {
    // Identification
    ID() ProviderID
    Name() string
    Type() VenueType // OnChain, ATS, Traditional

    // Market data
    Quote(ctx context.Context, req QuoteRequest) (*Quote, error)
    OrderBook(ctx context.Context, security SecurityID) (*Book, error)

    // Execution
    Submit(ctx context.Context, order Order) (*Execution, error)
    Cancel(ctx context.Context, orderID OrderID) error
    Status(ctx context.Context, orderID OrderID) (*OrderStatus, error)

    // Capabilities
    Supports(security SecurityID) bool
    MaxOrderSize(security SecurityID) uint64
    MinOrderSize(security SecurityID) uint64
    Jurisdictions() []Jurisdiction
    SettlementTime() time.Duration

    // Health
    Ping(ctx context.Context) error
    Latency() time.Duration
}
```

3 Best Execution Model

3.1 Execution Quality Metrics

We define execution quality as a vector of five metrics:

Definition 1 (Execution Quality)**.** *For an order o with quantity q and limit price p_{lim}, the execution quality at venue v is the vector:*

$$\mathbf{Q}(o, v) = (\pi(o, v),\ \phi(o, v),\ \lambda(o, v),\ \gamma(o, v),\ \sigma(o, v))$$

where:

- $\pi(o, v)$*: Price improvement (basis points better than NBBO or last trade).*
- $\phi(o, v)$*: Expected fill rate ($\in [0, 1]$, fraction of order filled).*
- $\lambda(o, v)$*: Execution latency (milliseconds from submission to fill).*
- $\gamma(o, v)$*: Market impact (basis points of adverse price movement).*
- $\sigma(o, v)$*: Settlement risk (probability of settlement failure).*

3.2 Objective Function

The SOR maximizes a weighted execution quality score subject to compliance constraints:

Definition 2 (Best Execution Optimization)**.** *Given an order o, a set of venues V, and an allocation $\mathbf{a} = (a_1, \ldots, a_{|V|})$ where a_v is the quantity allocated to venue v:*

$$\max_{\mathbf{a}} \sum_{v \in V} a_v \cdot (w_\pi \pi(o, v) + w_\phi \phi(o, v) - w_\lambda \lambda(o, v) - w_\gamma \gamma(o, v) - w_\sigma \sigma(o, v))$$

subject to:

$$\sum_{v \in V} a_v = q \qquad \textit{(fill entire order)} \tag{1}$$

$$0 \leq a_v \leq \min(q,\ \textit{depth}(v)) \qquad \textit{(venue capacity)} \tag{2}$$

$$\textit{compliant}(o, v) = \textit{true} \qquad \textit{(transfer restrictions)} \tag{3}$$

$$a_v = 0 \textit{ if } \neg \textit{compliant}(o, v) \qquad \textit{(compliance enforcement)} \tag{4}$$

The compliance predicate $\text{compliant}(o, v)$ verifies that:

- The security's transfer restrictions permit trading on venue v [2].
- Both counterparties satisfy jurisdictional requirements for venue v.
- The venue supports the security's token standard.

3.3 Greedy Allocation Algorithm

Algorithm 1 Greedy order allocation across venues.

Require: Order o with quantity q, venue set V, compliance predicate C
Ensure: Allocation $\mathbf{a}$ maximizing weighted execution quality

$V' \leftarrow \{v \in V : C(o, v) \wedge v.\text{Supports}(o.\text{security})\}$
Compute score$(v) \leftarrow w_\pi \pi(o, v) + w_\phi \phi(o, v) - w_\lambda \lambda(o, v) - w_\gamma \gamma(o, v) - w_\sigma \sigma(o, v)$ for all $v \in V'$
Sort V' by score(v) descending
remaining $\leftarrow q$
for $v \in V'$ in sorted order **do**
    alloc $\leftarrow \min(\text{remaining}, \text{depth}(v))$
    $a_v \leftarrow$ alloc
    remaining $\leftarrow$ remaining $-$ alloc
    if remaining $= 0$ **then**
        break
    end if
end for
if remaining > 0 **then**
    Route remainder to Lux DEX VM as limit order (on-book)
end if
return a

Theorem 1 (Approximation Guarantee)**.** *The greedy allocation (Algorithm 1) achieves a 2-approximation of the optimal weighted execution quality score.*

Proof. This is an instance of the bounded knapsack problem with uniform item values within each venue. The greedy-by-density algorithm for the fractional knapsack achieves the optimal fractional solution. Since our allocations are integral (shares are discrete), the greedy integral solution is within one item (one venue's allocation) of the fractional optimum. For $|V'| \geq 2$ venues, the worst case occurs when the greedy algorithm fills entirely from the second-best venue. The score ratio between the best and second-best venue is bounded by 2 in the worst case (since all scores are positive after compliance filtering). Therefore, the greedy integral solution achieves at least $1/2$ of the optimal score. □

4 On-Chain Execution: Lux D-Chain VM

4.1 The D-Chain Matching VM

On-chain order matching runs on the standalone Lux *D-Chain VM*, a purpose-built `block.ChainVM` described in detail in the Lux Lightspeed DEX paper [5]. The D-Chain is the source-of-truth product chain for the DEX; the SOR submits orders to it exactly as it submits to any other execution venue. For the SOR, the D-Chain VM provides:

- On-chain order book with price-time priority matching.
- Atomic settlement: trade execution and token transfer in a single transaction.
- Compliance-integrated matching: orders are only matched if both parties pass the security's transfer restriction checks.
- Sub-second finality via Lux consensus, with crash-restart durability.

4.2 Matcher-at-Verify, Commit-at-Accept

The D-Chain VM is not a generic state machine that replays matching inside a contract; matching *is* the block-verification rule. The matcher runs in `Block.Verify` against a `versiondb`

overlay, and the resulting fills are committed to the chain's `zapdb` store at `Accept`. Determinism — the property the SOR relies on for reproducible best-execution accounting — comes from three sources:

- The proposer relays each order to the matcher *once*, at block-build time, and carries the resulting fills in the block bytes; every validator re-derives the same fills from the same ordered inputs.
- Trade timestamps are block-derived, not wall-clock; replaying a block on any node yields byte-identical fills.
- Trade identifiers are derived deterministically from the content-addressed order set, so the same matched pair carries the same ID on every node.

Because fills land in `zapdb` at `Accept`, a validator that crashes mid-block and restarts re-loads committed state and resumes without divergence: *Verify-matches* → *zapdb-commits* → *restart-survives*. The on-chain leg of the SOR therefore inherits the same exactly-once settlement guarantee the router needs for its position reconciliation, with no off-chain replay layer.

4.3 D-Chain VM Order Submission

Listing 2: D-Chain VM order submission via SOR. The order is issued as a transaction to the D-Chain; matching and settlement occur in block verification, and `AwaitTx` returns once the fill is committed to `zapdb`.

```
type DEXProvider struct {
    client *luxclient.Client
    dexVM  ids.ID
}

func (d *DEXProvider) Submit(
    ctx context.Context, order Order,
) (*Execution, error) {
    tx := &dex.PlaceOrderTx{
        Security:  order.SecurityID,
        Side:      order.Side,
        OrderType: order.Type,
        Price:     order.Price,
        Quantity:  order.Quantity,
        Partition: order.Partition,
        Expiry:    order.Expiry,
    }

    txID, err := d.client.IssueTx(ctx, d.dexVM, tx)
    if err != nil {
        return nil, fmt.Errorf("dex submit: %w", err)
    }

    // Wait for confirmation (sub-second on Lux)
    result, err := d.client.AwaitTx(ctx, txID)
    if err != nil {
        return nil, fmt.Errorf("dex confirm: %w", err)
    }

    return &Execution{
        OrderID:   order.ID,
        Venue:     d.ID(),
        TxID:      txID,
        Price:     result.Price,
        Quantity:  result.Filled,
        Timestamp: result.Timestamp,
    }, nil
}
```

5 Traditional Venue Integration

5.1 Alpaca OmniSub

Alpaca serves as the primary traditional equity execution provider. The integration uses the Alpaca Broker API for account management and order routing:

Listing 3: Alpaca provider implementation.

```go
type AlpacaProvider struct {
    client *alpaca.Client
    mu     sync.RWMutex
}

func (a *AlpacaProvider) Submit(
    ctx context.Context, order Order,
) (*Execution, error) {
    req := alpaca.PlaceOrderRequest{
        Symbol:      order.Symbol,
        Qty:         decimal.NewFromInt(int64(order.Quantity)),
        Side:        mapSide(order.Side),
        Type:        mapOrderType(order.Type),
        TimeInForce: mapTIF(order.TimeInForce),
    }

    if order.Type == Limit {
        req.LimitPrice = &order.Price
    }

    alpacaOrder, err := a.client.PlaceOrder(req)
    if err != nil {
        return nil, fmt.Errorf("alpaca submit: %w", err)
    }

    return &Execution{
        OrderID:    order.ID,
        Venue:      a.ID(),
        ExternalID: alpacaOrder.ID,
        Status:     mapStatus(alpacaOrder.Status),
    }, nil
}

func (a *AlpacaProvider) SettlementTime() time.Duration {
    return 24 * time.Hour // T+1 for equities
}
```

5.2 Settlement Reconciliation

For orders executed on traditional venues, the SOR reconciles settlement between the off-chain execution and on-chain records:

1. Traditional venue executes the order (T+0 execution).
2. SOR records execution in the `executions` collection.
3. Traditional venue settles (T+1 for equities).
4. SOR mints/burns tokenized wrapper on Lux EVM to reflect settlement.
5. On-chain state reconciled with off-chain position.

6 ZAP Wire Protocol

6.1 Binary Serialization

Orders transmitted between the SOR and execution providers use the ZAP binary wire protocol [4] for minimal serialization overhead:

Table 2: ZAP order message format.

Field	Type	Bytes	Description
magic	uint16	2	Protocol identifier (0x5A41)
version	uint8	1	Protocol version
msg_type	uint8	1	Message type (order, cancel, exec, ...)
flags	uint16	2	Bit flags (side, TIF, compliance)
security_id	bytes32	32	Security identifier
order_id	bytes16	16	Order identifier
price	uint64	8	Price in basis points
quantity	uint64	8	Quantity in base units
timestamp	uint64	8	Nanosecond timestamp
partition	bytes32	32	Partition identifier
checksum	uint32	4	CRC-32C checksum
Total		**114**	

At 114 bytes per order message, ZAP achieves serialization latency under 1 microsecond on modern hardware, compared to 50–200 microseconds for JSON-based protocols.

6.2 Latency Comparison

Table 3: Serialization latency comparison.

Protocol	Message Size	Serialize	Deserialize
ZAP (binary)	114 B	0.8 μs	0.6 μs
Protocol Buffers	180 B	3.2 μs	2.8 μs
FlatBuffers	192 B	1.5 μs	0.4 μs
JSON	420 B	52 μs	38 μs
FIX (text)	350 B	28 μs	22 μs

7 Compliance-Aware Routing

7.1 Pre-Route Compliance Check

Before allocating order quantity to a venue, the SOR checks compliance at three levels:

1. **Security level**: Does the security's transfer restriction policy permit trading on this venue? (e.g., a Reg D security may only trade on US-registered ATS platforms.)
2. **Investor level**: Does the investor's accreditation, KYC status, and jurisdiction permit trading on this venue?
3. **Venue level**: Is the venue currently accepting orders for this security? (e.g., a venue may pause trading for a corporate action.)

Listing 4: Pre-route compliance check.

```
func (r *Router) checkCompliance(
    order Order, venue Provider,
) error {
    // Security-level check
    token := r.registry.GetSecurity(order.SecurityID)
    restrictions := token.TransferRestrictions()

    if !restrictions.AllowsVenue(venue.ID()) {
        return fmt.Errorf("security %s: venue %s not permitted",
            order.SecurityID, venue.ID())
```

```
    }

    // Investor-level check
    investor := r.registry.GetInvestor(order.InvestorID)
    did := investor.DID()

    for _, jur := range venue.Jurisdictions() {
        if did.Jurisdiction() == jur {
            goto jurisdictionOK
        }
    }
    return fmt.Errorf("investor jurisdiction %s not permitted on venue %s",
        did.Jurisdiction(), venue.ID())
jurisdictionOK:

    if restrictions.RequiresAccreditation() {
        if !did.HasValidCredential("accreditation") {
            return fmt.Errorf("investor %s: accreditation expired", order.InvestorID)
        }
    }

    // Venue-level check
    if !venue.Supports(order.SecurityID) {
        return fmt.Errorf("venue %s does not support security %s",
            venue.ID(), order.SecurityID)
    }

    return nil
}
```

7.2 Cross-Venue Settlement

When an order is split across multiple venues, settlement occurs independently at each venue. The SOR tracks settlement status and reconciles:

- **On-chain venues** (Lux DEX VM, Lux ATS internal): Atomic T+0 settlement on Lux EVM.
- **External ATS platforms**: Settlement per the venue's protocol (typically T+0 to T+2).
- **Traditional venues**: Settlement via DTCC (T+1 for equities).

The ATS maintains a settlement ledger that tracks the net position across all venues and reconciles on-chain state with off-chain settlements via the `position_reconcile` cron job [1].

8 Order Matching Priority

8.1 Priority Rules

The SOR applies the following priority hierarchy for venue selection:

1. **Price improvement**: Venues offering better prices are preferred.
2. **Internalization**: If the Lux ATS internal book can fill the order at NBBO or better, the order is internalized (matched against resting orders on the internal book).
3. **Settlement speed**: On-chain venues (T+0) are preferred over traditional venues (T+1) at equal price.
4. **Fill rate**: Venues with higher expected fill rates are preferred.
5. **Latency**: Among equal-quality venues, lower-latency venues are preferred.
6. **Cost**: Trading fees are factored into the effective price.

8.2 Internalization Policy

Internalization (matching orders against the internal book rather than routing to external venues) is permitted when:

- The internal book offers price improvement over the best external quote.

- The order can be fully filled from the internal book.
- Both parties pass compliance checks for the security [2].

When internalization provides no price improvement, orders are routed to the venue with the best composite score per the optimization in Section 3.

9 Market Data Aggregation

9.1 Consolidated Quote

The SOR maintains a consolidated quote across all venues for each security:

Listing 5: Consolidated quote aggregation.

```
type ConsolidatedQuote struct {
    SecurityID SecurityID
    NBBO       NBBO
    Venues     []VenueQuote
    Timestamp  time.Time
}

type NBBO struct {
    BidPrice    decimal.Decimal
    BidSize     uint64
    BidVenue    ProviderID
    AskPrice    decimal.Decimal
    AskSize     uint64
    AskVenue    ProviderID
}

func (r *Router) consolidateQuotes(
    secID SecurityID,
) (*ConsolidatedQuote, error) {
    var quotes []VenueQuote
    var mu sync.Mutex
    var wg sync.WaitGroup

    for _, p := range r.providers {
        if !p.Supports(secID) {
            continue
        }
        wg.Add(1)
        go func(prov Provider) {
            defer wg.Done()
            q, err := prov.Quote(context.Background(),
                QuoteRequest{Security: secID})
            if err != nil {
                return // skip unavailable venues
            }
            mu.Lock()
            quotes = append(quotes, VenueQuote{
                Provider: prov.ID(), Quote: *q,
            })
            mu.Unlock()
        }(p)
    }
    wg.Wait()

    nbbo := computeNBBO(quotes)
    return &ConsolidatedQuote{
        SecurityID: secID,
        NBBO:       nbbo,
        Venues:     quotes,
        Timestamp:  time.Now().UTC(),
    }, nil
}
```

The market data infrastructure descends from Arca's `miners/binance-recorder`, which recorded and replayed exchange data feeds for market analysis. The Lux SOR extends this to aggregate real-time quotes from all 16 providers.

10 Performance

Table 4: SOR performance benchmarks.

Metric	Value
Quote aggregation (16 providers)	8 ms median
Compliance check (per venue)	0.3 ms
Allocation computation	0.1 ms
ZAP serialization	0.8 μs
On-chain submission (Lux DEX VM)	12 ms median
Traditional submission (Alpaca)	45 ms median
End-to-end route decision	9 ms median
Throughput (orders/sec)	50,000

11 Regulatory Compliance

11.1 FINRA Rule 5310: Best Execution

The SOR satisfies FINRA Rule 5310 by:

1. Evaluating execution quality across all reasonably available venues.
2. Documenting the routing decision and execution quality metrics for each order.
3. Periodically reviewing venue quality statistics and adjusting routing weights.
4. Disclosing order routing practices per Rule 606 (formerly Rule 11Ac1-6).

11.2 Regulation NMS: Order Protection

For NMS-eligible securities (listed equities accessed via Alpaca or Interactive Brokers), the SOR complies with Regulation NMS Rule 611 (Order Protection Rule) by:

- Never executing at a price inferior to a protected quote displayed by another venue.
- Routing through to the displaying venue when a better protected quote exists.
- Maintaining a consolidated quote that reflects all protected quotes.

11.3 Audit Trail

Every routing decision is logged with:

- All venue quotes considered.
- Compliance check results per venue.
- Score computation for each eligible venue.
- Final allocation decision.
- Execution result and fill quality.

These records are stored in the `audit_logs` collection and exported to CAT (Consolidated Audit Trail) per SEC Rule 613 [1].

12 Conclusion

The Lux SOR extends the Arca `exchange-sdk` from a single-venue trading SDK to a 16-provider smart order router that satisfies best execution obligations for digital securities. The system routes across on-chain (Lux DEX VM), ATS (digital securities platforms), and traditional (equity exchanges) venues with compliance-aware allocation, ZAP binary serialization, and atomic settlement on Lux EVM.

The SOR operates as a component of the non-custodial ATS [1], routing orders for securities implemented under the Lux Security Token Standard [2], with cap table state maintained by the blockchain transfer agent [3].

References

[1] Z. Kelling, "Non-Custodial Blockchain ATS: Architecture of a Regulatory-Compliant Securities Exchange," Lux Partners, 2025.

[2] Z. Kelling, "Lux Security Token Standard: ERC-1400 on Post-Quantum EVM with MPC Custody," Lux Partners, 2025.

[3] Z. Kelling, "Blockchain Transfer Agent: On-Chain Cap Table with MPC-Custodied Registry," Lux Partners, 2025.

[4] Z. Kelling, "ZAP: Zero-Allocation Binary Wire Protocol for Consensus Transport," Lux Partners, 2023.

[5] Z. Kelling, "Lux Lightspeed DEX: On-Chain Order Book with Sub-Second Finality," Lux Partners, 2023.

[6] Z. Kelling, V. Seesahai, "M-Chain: Decentralized Multi-Party Computation Custody with Quantum-Safe Threshold Signatures," Lux Partners, 2023.

[7] Financial Industry Regulatory Authority, "FINRA Rule 5310: Best Execution and Interpositioning," FINRA Manual.

[8] Securities and Exchange Commission, "Regulation NMS," Release No. 34-51808, 2005.

[9] Securities and Exchange Commission, "Disclosure of Order Handling Information," Rule 606, Securities Exchange Act of 1934.

[10] Financial Industry Regulatory Authority, "Consolidated Audit Trail (CAT)," FINRA Rule 6800 Series.