Technology & Architecture Decisions

Recommended Choices (Easier Path):

1. **Mock Implementation** - More control, faster development, easier debugging
2. **Market Order** - Simplest logic (immediate execution), no price monitoring needed
3. **Architecture**: Layered architecture with clear separation of concerns

Why These Choices:

- **Mock over Real Devnet**: Eliminates blockchain network unpredictability, SDK version conflicts, and wallet management complexity
- **Market Order over Limit/Sniper**: No continuous price monitoring, no event listeners, straightforward execution flow
- **Extension Strategy**: The engine architecture supports adding Limit (add price monitor service) and Sniper (add blockchain event listener) by plugging into the same routing and execution pipeline

---

Tech Stack Installation Order

Phase 0: Environment Setup

1. Node.js 18+ and pnpm
2. Docker Desktop (for Redis & PostgreSQL)
3. VS Code with extensions (ESLint, Prettier, TypeScript)

Phase 1: Core Dependencies

1. Initialize TypeScript project with strict config
2. Fastify + WebSocket plugin
3. PostgreSQL + Prisma ORM
4. Redis + BullMQ
5. Zod for validation
6. Winston for logging

Phase 2: Development Dependencies

1. Jest + Supertest for testing
2. ts-node-dev for hot reload
3. ESLint + Prettier configs

Project Architecture

```
src/
├── config/           # Environment configs
├── types/            # TypeScript interfaces & types
├── services/         # Business logic layer
│   ├── dex/          # DEX routing logic
│   ├── orders/       # Order processing
│   ├── queue/        # BullMQ job handlers
│   └── websocket/    # WebSocket manager
├── controllers/      # HTTP request handlers
├── routes/           # API route definitions
├── repositories/     # Database access layer
├── utils/            # Helpers & utilities
├── validators/       # Request validation schemas
└── app.ts            # Application entry point
```

Design Principles:

1. **Single Responsibility**: Each service handles one concern
2. **Dependency Injection**: Easy testing and mocking
3. **Repository Pattern**: Database abstraction
4. **Factory Pattern**: For creating orders and WebSocket connections
5. **Strategy Pattern**: For DEX routing decisions

Implementation Steps

Step 1: Project Foundation (Day 1 - 2 hours)

- [ ] Initialize Git repository with proper .gitignore
- [ ] Setup TypeScript config with strict mode
- [ ] Create directory structure
- [ ] Setup Docker Compose for Redis + PostgreSQL
- [ ] Configure environment variables (.env.example)
- [ ] Setup ESLint + Prettier

Step 2: Database Layer (Day 1 - 2 hours)

- [ ] Design database schema (Orders, OrderStatus, ExecutionDetails)
- [ ] Setup Prisma with PostgreSQL
- [ ] Create migrations
- [ ] Build repository interfaces and implementations
- [ ] Add Redis client for active order caching

Step 3: Core Services - DEX Router (Day 1-2 - 3 hours)

- [ ] Create DEX interface (Raydium, Meteora)
- [ ] Implement MockDexRouter with realistic delays
- [ ] Add price comparison logic (2-5% variance)
- [ ] Implement best route selection
- [ ] Add logging for routing decisions
- [ ] Unit tests for routing logic

Step 4: Queue System (Day 2 - 2 hours)

- [ ] Setup BullMQ with Redis
- [ ] Create order processing queue
- [ ] Implement worker with concurrency limit (10)
- [ ] Add exponential backoff retry (max 3)
- [ ] Rate limiting (100 orders/minute)
- [ ] Add queue event listeners for monitoring

Step 5: Order Service (Day 2 - 3 hours)

- [ ] Create Order entity and DTOs
- [ ] Implement Market Order execution logic
- [ ] Add order validation (Zod schemas)
- [ ] Integrate DEX router
- [ ] Add transaction simulation
- [ ] Error handling with detailed messages
- [ ] Unit tests for order processing

Step 6: WebSocket Manager (Day 3 - 3 hours)

- [ ] Setup Fastify WebSocket plugin
- [ ] Create WebSocket connection manager
- [ ] Implement status update broadcaster
- [ ] Add connection lifecycle handling
- [ ] Room-based broadcasting (orderId rooms)
- [ ] Heartbeat/ping-pong for connection health
- [ ] Integration tests for WebSocket flow

Step 7: HTTP API (Day 3 - 2 hours)

- [ ] Create POST /api/orders/execute endpoint
- [ ] Implement HTTP → WebSocket upgrade
- [ ] Add request validation middleware
- [ ] Error handling middleware
- [ ] Rate limiting middleware
- [ ] Health check endpoint

Step 8: Integration & Flow (Day 4 - 3 hours)

- [ ] Wire all services together
- [ ] Implement complete order lifecycle:
  - pending → routing → building → submitted → confirmed
- [ ] Add comprehensive logging
- [ ] Test concurrent order processing
- [ ] Test error scenarios and retries
- [ ] Performance testing (100 orders/min)

Step 9: Testing Suite (Day 4 - 3 hours)

- [ ] Unit tests for each service (70% coverage minimum)
- [ ] Integration tests for order flow
- [ ] WebSocket lifecycle tests
- [ ] Queue behavior tests
- [ ] Mock different failure scenarios
- [ ] Create Postman/Insomnia collection

Step 10: Documentation (Day 5 - 2 hours)

- [ ] Write comprehensive README
- [ ] Document API endpoints
- [ ] Add architecture diagrams
- [ ] Setup instructions
- [ ] Design decisions explanation
- [ ] Extension strategy for Limit/Sniper orders

Step 11: Deployment (Day 5 - 2 hours)

- [ ] Choose hosting (Render/Railway/Fly.io)
- [ ] Setup production environment variables
- [ ] Deploy PostgreSQL (Supabase/Neon)
- [ ] Deploy Redis (Upstash/Redis Cloud free tier)
- [ ] Deploy application
- [ ] Test deployed endpoints

Step 12: Demo Video (Day 5 - 1 hour)

- [ ] Prepare demo script
- [ ] Submit 3-5 orders simultaneously
- [ ] Show WebSocket status updates
- [ ] Display DEX routing logs
- [ ] Demonstrate queue processing
- [ ] Record and upload to YouTube

Key Implementation Details

Mock DEX Router Strategy:

```
 - Base price calculation with controlled variance
- Raydium: 0.3% fee, -2% to +2% price variance
- Meteora: 0.2% fee, -3% to +3% price variance
- Simulate 200-300ms network delay per quote
- Simulate 2-3s execution time
- Generate realistic mock transaction hashes
```

WebSocket Status Flow:

```
1. Client POST /api/orders/execute
2. Server validates, creates order, returns orderId
3. Connection upgrades to WebSocket
4. Client joins room: `order:${orderId}`
5. Worker emits status updates to room
6. Client receives real-time updates
```

Queue Processing Strategy:

```
- Concurrency: 10 workers
- Rate limit: 100 jobs/min (600ms delay between jobs)
- Retry: 3 attempts with exponential backoff (1s, 2s, 4s)
- Failed jobs stored in DB with error reason
```

Error Handling Categories:

```
1. Validation errors → 400 response
2. Routing failures → Retry with backoff
3. Execution failures → Mark as failed + persist reason
4. Network errors → Retry mechanism
5. System errors → Log + alert
```

Testing Strategy

Unit Tests (≥10):

1. DEX price comparison logic
2. Order validation
3. Queue retry mechanism
4. WebSocket room management
5. Mock execution simulation
6. Error handling paths
7. Rate limiting
8. Repository methods
9. Price calculation
10. Transaction hash generation

Integration Tests:

1. Complete order flow (pending → confirmed)
2. Concurrent order processing
3. WebSocket lifecycle
4. Queue failure scenarios
5. API endpoint workflows

Timeline Estimate

- **Total Development**: 4-5 days
- **Testing & Documentation**: 1 day
- **Deployment & Video**: 0.5 day
- **Buffer**: 0.5 day

Total: 6-7 days for complete, polished delivery

This plan prioritizes clean architecture, testability, and demonstrating solid engineering practices over complex blockchain integration.