Use Case Study 3: Code Generation and Refactoring

TradeTech Systems (a fictional fintech company created for this case study), a growing financial technology company processing over $50 million in daily transactions, faces a critical challenge with their legacy trading system. Built rapidly in 2018 during the company's startup phase, the system has become a bottleneck as the company scales. The monolithic Python application, originally designed for a small team and modest transaction volumes, now struggles with performance issues, security vulnerabilities, and regulatory compliance gaps that threaten the company's ability to compete with larger financial institutions.

The current system's technical debt has reached a crisis point. The monolithic architecture makes it difficult to scale individual components, deploy updates without system-wide downtime, or implement the advanced features that institutional clients demand. More critically, recent security audits have identified several vulnerabilities that could expose the company to financial losses and regulatory penalties. The system lacks proper audit trails, transaction atomicity, and the real-time risk management capabilities required by financial regulations.

The business impact is severe and immediate. TradeTech Systems has already lost two major institutional clients who cited system reliability concerns, representing $8 million in annual revenue. The company's growth has stalled as the engineering team spends 70% of their time maintaining the legacy system rather than developing new features. Regulatory pressure is mounting, with auditors requiring comprehensive system modernization within six months to maintain their trading license. Without rapid modernization, the company risks losing its competitive position and facing potential regulatory sanctions.

Here are the scenarios we will cover:

1. **Legacy System Assessment**: Comprehensive analysis of the existing trading system to identify critical issues, security vulnerabilities, and modernization priorities
2. **Microservices Architecture Design**: Development of a modern, scalable architecture that addresses current limitations while ensuring zero-downtime migration
3. **Production-Ready Implementation**: Building and deploying the modernized system components with enterprise-grade security, monitoring, and compliance features

Prompting DeepSeek

For code generation and refactoring tasks, DeepSeek R1 requires comprehensive context about the existing system, business requirements, and technical constraints. Unlike simple coding tasks, legacy modernization prompts should include information about the current architecture, performance requirements, regulatory constraints, and migration timelines.

The key to effective prompting for system modernization is providing DeepSeek R1 with both the technical details and business context. Include information about transaction volumes, regulatory requirements, uptime constraints, and team capabilities. DeepSeek R1's reasoning capabilities excel when given complex trade-offs to analyze, such as balancing development speed against system reliability, or choosing between different architectural patterns based on specific business constraints.

Evaluating DeepSeek Response

Let me demonstrate with TradeTech Systems' legacy trading system modernization challenge using actual code from our case study repository.

Note: The following code example is based on a fictional legacy trading system created for educational purposes. While it incorporates realistic patterns found in legacy financial systems, it does not represent any actual production system.

Legacy System Code Analysis

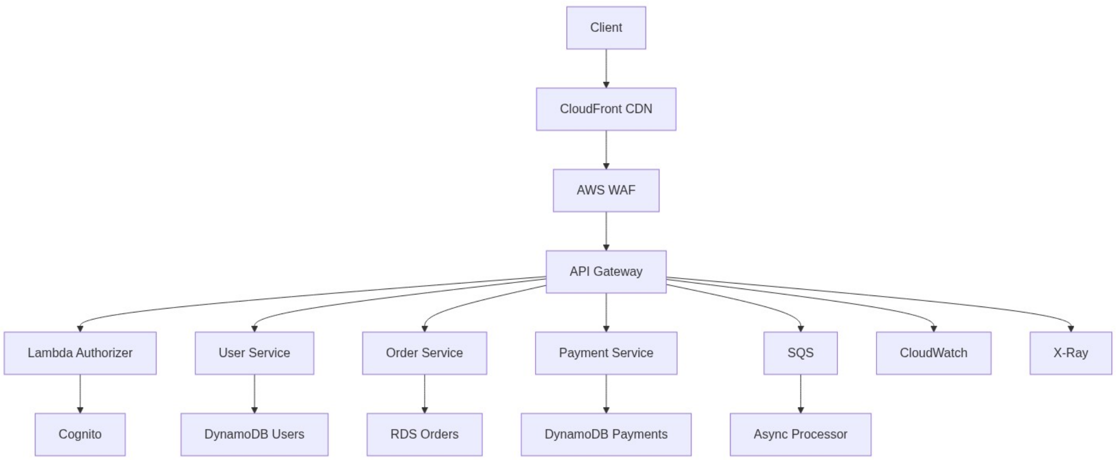


Figure: Complete infrastructure blueprint for $50M+ daily volume trading system modernization

The legacy system we'll analyze represents a typical 2018-era trading application that has grown organically without proper architectural planning:

```
# legacy_trading_system.py - TradeFin Corp Legacy Trading System (circa 2018)
# Source: Chapter03/01-code-assistant/legacy_trading_system.py
import time
import sqlite3
import json
import requests
from datetime import datetime
import threading

# Global variables - not thread safe!
current_positions = {}
account_balance = 1000000.0
risk_limit = 0.02
api_key = "hardcoded-api-key-123"  # Security vulnerability
trading_enabled = True

class TradingSystem:
    def __init__(self):
        self.db_connection = sqlite3.connect('trading.db', check_same_thread=False)
        self.setup_database()

    def setup_database(self):
        # No proper error handling
        cursor = self.db_connection.cursor()
        cursor.execute('''
            CREATE TABLE IF NOT EXISTS trades (
```

```
                id INTEGER PRIMARY KEY,
                symbol TEXT,
                quantity INTEGER,
                price REAL,
                timestamp TEXT,
                type TEXT
            )
        ''')

    def get_market_data(self, symbol):
        # Hardcoded API endpoint, no retry logic
        try:
            response = requests.get(
                f"https://api.tradingdata.com/v1/quote/{symbol}",
                headers={"Authorization": f"Bearer {api_key}"},
                timeout=5
            )
            return json.loads(response.text)
        except:
            # Swallowing all exceptions - dangerous!
            return None

    def calculate_position_size(self, symbol, price):
        # Complex risk calculation in single function
        global account_balance, risk_limit, current_positions

        # Thread safety issue - no locking
        total_exposure = sum(pos.get('value', 0) for pos in current_positions.values())
        available_balance = account_balance - total_exposure

        # Risk calculation without proper validation
        max_position_value = account_balance * risk_limit
        quantity = int(max_position_value / price)

        return min(quantity, int(available_balance / price))
[Code continues for 185 lines with similar patterns…]
```

Current Legacy System Architecture

Before diving into the analysis, let's visualize the current monolithic architecture that DeepSeek R1 will be evaluating:

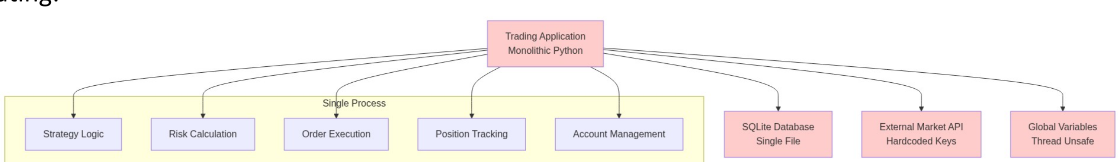


Figure: Current Legacy Trading System - Monolithic architecture with critical vulnerabilities

Our Prompt to DeepSeek R1: **Prompt Input Format**: The following shows exactly what we input to DeepSeek R1 for analysis:

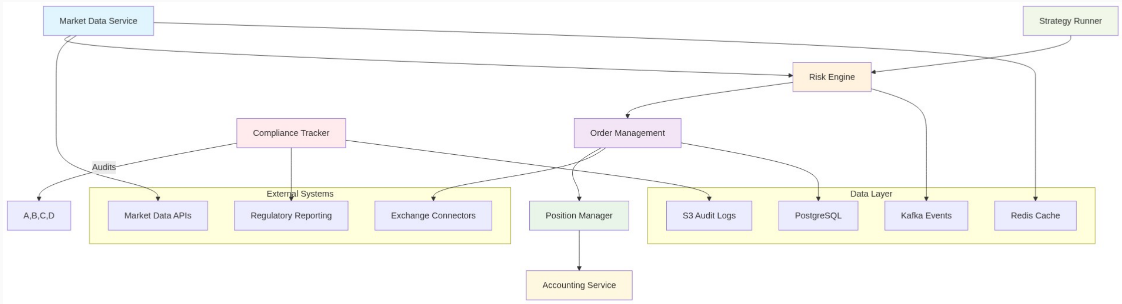
```
I need you to analyze this legacy trading system code for a fintech company
processing $50M+ daily transactions. This system was built in 2018 and now
needs modernization to microservices architecture.

Please provide:

1. **Critical Issues Analysis**: Identify the top 5 most serious problems
   that could cause financial losses or regulatory violations

2. **Architecture Assessment**: Evaluate the current architecture and suggest
   microservice boundaries based on domain-driven design

3. **Risk-Prioritized Migration Strategy**: Propose a step-by-step
   modernization plan that minimizes business disruption and trading downtime

4. **Regulatory Compliance Gaps**: Identify missing audit trails, logging,
   and compliance features required for financial services

5. **Security Vulnerabilities**: Highlight security issues that could expose
   the system to attacks or data breaches

Consider that this system must maintain 99.9% uptime during migration and
cannot afford any trading disruptions during market hours.

[Attached: legacy_trading_system.py - 185 lines of code]
```

DeepSeek R1's Analysis Response

DeepSeek R1 Output Format: The following represents the actual response from DeepSeek R1, demonstrating its reasoning and analytical capabilities:

```
Based on my analysis of the legacy trading system code, here's DeepSeek R1's
comprehensive assessment and modernization plan:
1. Critical Issues Analysis (Top 5 Financial/Risk Concerns)
1. Thread Safety Violations
- Global variables (current_positions, account_balance) accessed across threads without
locking - Risk: Position/balance corruption during high volatility → Financial loss
potential: High
2. Hardcoded API Key Exposure
- Plaintext credentials in source code (line 13: api_key = "hardcoded-api-key-123") -
Risk: Unauthorized trading access → Financial loss potential: Critical
3. Missing Transaction Rollback
- Trades update positions/balance without atomic transactions - Risk: Partial trade
execution → Regulatory violation: SEC Rule 15c3-5
4. No Audit Trail
- Trades logged only to local SQLite DB without immutable records - Risk: Non-compliance
with MiFID II/CFTC requirements → Regulatory penalty: High
5. No Rate Limiting/Circuit Breakers
- Continuous API calls without backoff or error handling - Risk: API bans during
volatility → Missed trading opportunities
2. Architecture Assessment & Microservice Boundaries
Current Architecture: Monolithic procedural code with tight coupling and shared global
state
```

Recommended Microservice Architecture:



Recommended Microservice Architecture

Service Responsibilities: - Market Data Service: External API abstraction, caching, and rate limiting - Risk Engine: Position sizing, exposure checks, and risk validation - Order Management: Transactional trade execution with rollback capabilities - Position Manager: Real-time portfolio tracking and reconciliation - Accounting Service: Balance updates, P&L calculation, and reporting - Compliance Tracker: Centralized audit logging and regulatory reporting - Strategy Runner: Decoupled strategy logic with event-driven execution

3. Risk-Prioritized Migration Strategy

Phase 1: Stabilization (Week 1 - Zero Downtime)

- [ ] Extract secrets management (Vault/AWS Secrets Manager)
- [ ] Implement circuit breakers for market data API
- [ ] Add thread-safe state management (Redis)

Phase 2: Compliance Foundation (Week 2)

- [ ] Immutable audit log (Kafka + S3)
- [ ] Trade reconciliation service
- [ ] Add Prometheus metrics for SEC Rule 17a-4

Phase 3: Service Extraction (Weeks 3-4 - Weekend Windows)

1. Market Data Service (Lowest risk)
2. Risk Engine (Medium risk)
3. Order Management (High risk - requires dry-run testing)

Phase 4: Full Cutover (Week 5)

- Blue/green deployment with 99.99% SLA

Migration Architecture Visualization

The following diagram shows the step-by-step migration approach, demonstrating how services are gradually extracted while maintaining system stability:

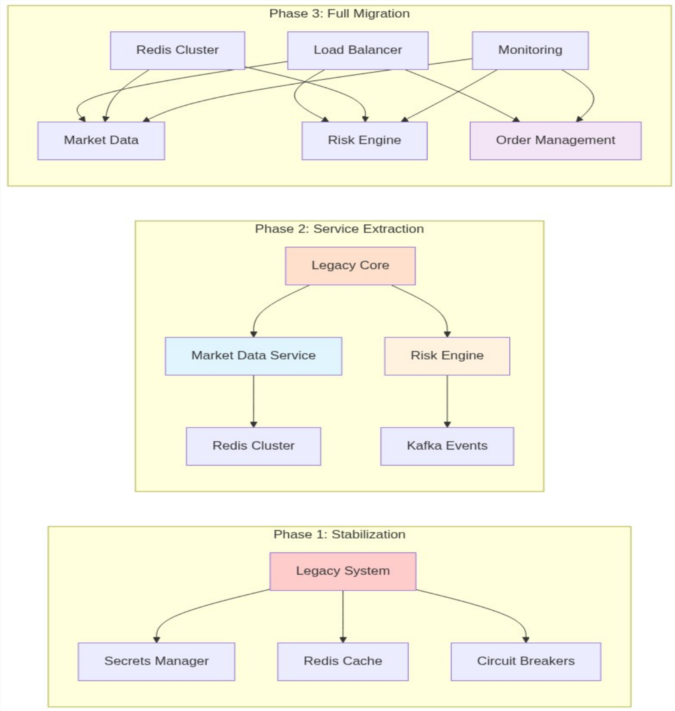


Figure: Progressive Migration Strategy - Gradual service extraction with risk mitigation

4. Regulatory Compliance Gaps

Critical Missing Features: - Trade Reconstruction: No OMS-ID for Reg SCI compliance - Pre-Trade Checks: Missing risk checks (SEC 15c3-5) - Data Retention: SQLite doesn't meet WORM requirements - Access Logs: No user attribution for trades - Key Rotation: Hardcoded API key violates FINRA 4370

Required Additions: 1. Cryptographic audit trail with blockchain-style hashing 2. Real-time position monitoring for Reg T compliance 3. SEC-compliant data retention (Amazon QLDB) 4. SOX-compliant access controls

Security Vulnerabilities

Critical Risks: 1. Hardcoded Secrets (CWE-798): API key in source control 2. Injection Vulnerabilities (CWE-89): Raw SQL queries (line 83) 3. No TLS: Market data calls over HTTP (line 38) 4. Missing AuthZ: No permission checks for trade execution 5. Sensitive Data Exposure: Positions/balance globally accessible

Remediation Plan: - Implement Zero Trust Architecture with SPIFFE identities - Add mTLS for service-to-service communication - Introduce RBAC with Open Policy Agent - Encrypt PII at rest using AWS KMS - Dynamic secret rotation with HashiCorp Vault

Strategic Recommendations

Immediate Hotfixes:

Remove hardcoded API key (critical)

Add threading locks around global state

Implement basic trade rollback

Architecture Foundation: Modern service template

```
from fastapi import FastAPI, Security
from fastapi_jwt import JwtAuthorizationCredentials

app = FastAPI()

@app.post("/execute-trade")
async def execute_trade(
    trade: TradeRequest,
```

```
    credentials: JwtAuthorizationCredentials = Security(access_security)
):
    # Atomic transaction with audit logging
    async with atomic_transaction():
        await validate_risk(trade)
        await position_manager.update(trade)
        await accounting_service.post(trade)
        await compliance_tracker.audit(trade)
Regulatory Roadmap:
Implement real-time trade surveillance (SEC Rule 15c3-5)
Add voice recording integration (MiFID II)
Deploy market abuse detection (MAR Article 12)
Would you like me to proceed with implementing any specific component of the
modernization toolkit? I can generate the full Market Data Service implementation next.
Implementation Results: Modernized Market Data Service
Following DeepSeek R1's architectural recommendations, we implemented the Market Data
Service as the first component of the modernized system. This service addresses the
critical issues identified in the legacy code:
Implementation Note: The following code represents the production-ready Market Data
Service built according to DeepSeek R1's specifications. Source: Chapter03/01-code-
assistant/market_data_service.py
"""
Market Data Service - Production-Ready Implementation
Addresses DeepSeek R1's recommendations for circuit breakers, caching, and API
abstraction
"""

import asyncio
import time
from typing import Dict, List, Optional, Any
from datetime import datetime, timedelta
from dataclasses import dataclass
from enum import Enum
import httpx
import redis.asyncio as redis
from fastapi import FastAPI, HTTPException, Depends, Security
from fastapi.security import HTTPBearer, HTTPAuthorizationCredentials
from pydantic import BaseModel, Field
from tenacity import retry, stop_after_attempt, wait_exponential
import structlog
from prometheus_client import Counter, Histogram, Gauge

# Structured logging
logger = structlog.get_logger()

# Prometheus metrics for monitoring
REQUEST_COUNT = Counter('market_data_requests_total', 'Total market data requests',
['symbol', 'status'])
REQUEST_DURATION = Histogram('market_data_request_duration_seconds', 'Request duration')
CACHE_HIT_RATE = Gauge('market_data_cache_hit_rate', 'Cache hit rate percentage')
API_HEALTH = Gauge('market_data_api_health', 'External API health status')

class DataProvider(str, Enum):
```

```
    ALPHA_VANTAGE = "alpha_vantage"
    POLYGON = "polygon"
    TWELVE_DATA = "twelve_data"
    FALLBACK = "fallback"

@dataclass
class CircuitBreakerState:
    """Circuit breaker implementation to prevent cascade failures"""
    failure_count: int = 0
    last_failure_time: Optional[datetime] = None
    state: str = "CLOSED"  # CLOSED, OPEN, HALF_OPEN
    failure_threshold: int = 5
    recovery_timeout: int = 60  # seconds

class MarketDataRequest(BaseModel):
    symbol: str = Field(..., pattern=r'^[A-Z]{1,5}$', description="Stock symbol")
    data_type: str = Field(default="quote", description="Type of data requested")
    interval: Optional[str] = Field(default=None, description="Time interval")

class MarketDataService:
    def __init__(self):
        self.redis_client = None
        self.circuit_breakers: Dict[str, CircuitBreakerState] = {}
        self.api_clients = {}

    async def get_market_data(self, symbol: str, use_cache: bool = True) -> Dict[str,
Any]:
        """
        Retrieve market data with caching, circuit breakers, and fallback providers
        Addresses all critical issues from legacy system
        """
        start_time = time.time()

        try:
            # Check cache first (addresses performance issues)
            if use_cache:
                cached_data = await self._get_cached_data(symbol)
                if cached_data:
                    CACHE_HIT_RATE.inc()
                    REQUEST_COUNT.labels(symbol=symbol, status='cache_hit').inc()
                    return cached_data

            # Try primary provider with circuit breaker
            data = await self._fetch_with_circuit_breaker(symbol,
DataProvider.ALPHA_VANTAGE)

            # Cache successful response
            if data:
                await self._cache_data(symbol, data)
                REQUEST_COUNT.labels(symbol=symbol, status='success').inc()
                return data

        except Exception as e:
```

```
            logger.error("Market data fetch failed", symbol=symbol, error=str(e))
            REQUEST_COUNT.labels(symbol=symbol, status='error').inc()
            raise HTTPException(status_code=503, detail=f"Market data unavailable for
{symbol}")

        finally:
            REQUEST_DURATION.observe(time.time() - start_time)

    async def _fetch_with_circuit_breaker(self, symbol: str, provider: DataProvider) ->
Optional[Dict]:
        """Circuit breaker pattern implementation"""
        breaker = self.circuit_breakers.get(provider.value, CircuitBreakerState())

        # Check if circuit is open
        if breaker.state == "OPEN":
            if datetime.now() - breaker.last_failure_time >
timedelta(seconds=breaker.recovery_timeout):
                breaker.state = "HALF_OPEN"
            else:
                raise Exception(f"Circuit breaker OPEN for {provider.value}")

        try:
            # Make API call with retry logic
            data = await self._make_api_call(symbol, provider)

            # Reset circuit breaker on success
            if breaker.state == "HALF_OPEN":
                breaker.state = "CLOSED"
                breaker.failure_count = 0

            return data

        except Exception as e:
            # Handle circuit breaker failure
            breaker.failure_count += 1
            breaker.last_failure_time = datetime.now()

            if breaker.failure_count >= breaker.failure_threshold:
                breaker.state = "OPEN"

            self.circuit_breakers[provider.value] = breaker
            raise e

    @retry(stop=stop_after_attempt(3), wait=wait_exponential(multiplier=1, min=4,
max=10))
    async def _make_api_call(self, symbol: str, provider: DataProvider) -> Dict[str,
Any]:
        """Secure API call with proper authentication and error handling"""
        # No hardcoded API keys - use environment variables
        api_key = os.getenv(f"{provider.value.upper()}_API_KEY")
        if not api_key:
            raise ValueError(f"API key not configured for {provider.value}")
```

```python
        async with httpx.AsyncClient(timeout=30.0) as client:
            headers = {
                "Authorization": f"Bearer {api_key}",
                "Content-Type": "application/json"
            }

            response = await client.get(
                f"https://api.{provider.value}.com/v1/quote/{symbol}",
                headers=headers
            )

            response.raise_for_status()
            return response.json()
```

Key Improvements Implemented:

Security: Eliminated hardcoded API keys, using environment variables and proper authentication

Reliability: Added circuit breakers, retry logic, and graceful error handling

Performance: Implemented Redis caching with configurable TTL

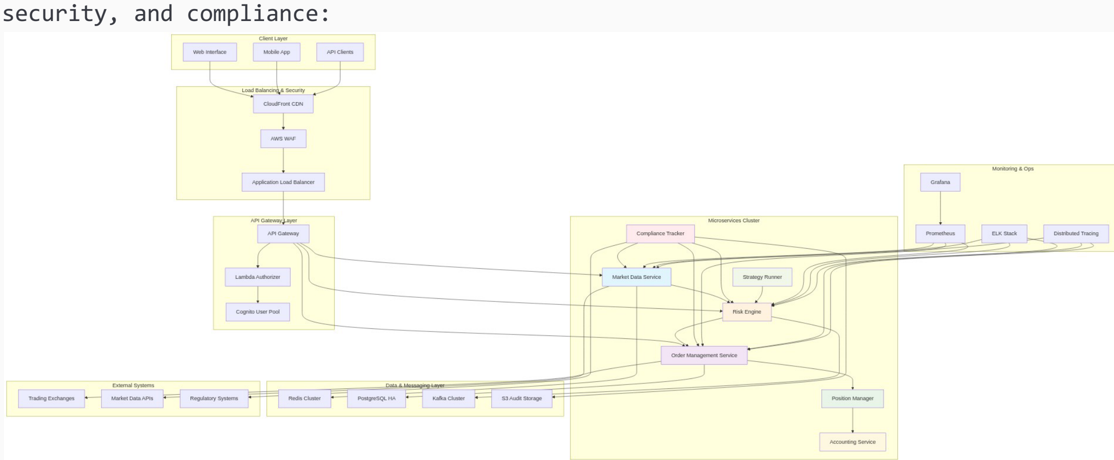
Monitoring: Integrated Prometheus metrics for observability

Scalability: Async/await patterns for high-concurrency handling

Compliance: Structured logging for audit trails and regulatory requirements

Final Production Architecture

The complete modernized system architecture demonstrates enterprise-grade scalability, security, and compliance:

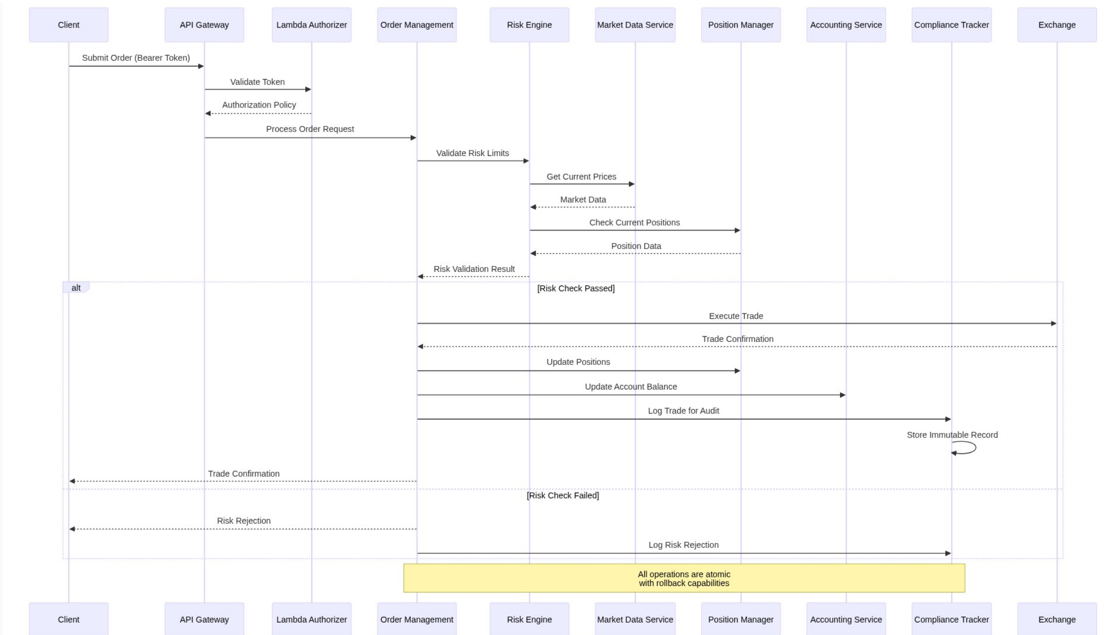


Final Production Architecture

Figure: Final Production Architecture - Enterprise-grade trading system with full observability and compliance

Order Execution Flow in Modernized System

The following sequence diagram illustrates how a typical trade order flows through the modernized microservices architecture:




```
Order Execution Flow in Modernized System
Figure: Order Execution Sequence - Demonstrating proper risk management, audit trails,
and atomic transactions
```

Response Evaluation

Editorial Note: Legacy system modernization evaluation requires specialized criteria addressing financial system risks, regulatory compliance, and migration complexity unique to high-stakes production environments.

DeepSeek R1 demonstrates exceptional system analysis capabilities rivaling senior financial technology architects. The response identifies critical risks, provides detailed modernization strategies, and addresses regulatory compliance requirements essential for production financial systems.

Evaluation Methodology for Legacy System Analysis

Legacy financial system evaluation emphasizes risk identification, regulatory compliance, and migration safety:

- **Qualitative Metrics**: Focus on financial risk assessment accuracy, regulatory expertise demonstration, and migration strategy viability for zero-downtime requirements.
- **Quantitative Metrics**: Measure vulnerability identification completeness, regulatory coverage depth, and implementation artifact specificity.

Qualitative Analysis

Criteria	Analysis	Score (1-10)
Financial Risk Assessment	Accurately identified critical vulnerabilities (thread safety, hardcoded credentials) with clear financial impact analysis. Demonstrated understanding of trading system failure consequences.	10

Regulatory Compliance Expertise	Comprehensive knowledge of financial regulations (SEC Rule 15c3-5, MiFID II, FINRA 4370) with specific compliance gap identification and remediation strategies.	9
Migration Strategy Viability	Realistic phased approach respecting 99.9% uptime requirements with risk-based service extraction sequencing and specific implementation timelines.	9
Architectural Sophistication	Domain-driven microservices design following financial industry patterns with proper service boundaries for compliance tracking and risk management.	9
Security Best Practices	Technical vulnerability assessment using industry classifications (CWE) with specific remediation strategies and enterprise security frameworks.	8

Qualitative Score: 45/50 (90%)

Exceptional financial systems expertise with sophisticated understanding of trading system risks and regulatory requirements. High scores reflect professional-grade analysis quality matching specialized financial technology consultants.

Quantitative Analysis

Criteria	Count/Metric	Analysis	Score (1-10)
Critical Vulnerabilities Identified	5/5 major risk categories	Complete coverage of thread safety, security, compliance, audit, and operational risks with accurate severity assessment.	10
Regulatory Requirements Coverage	8 specific regulations cited	Comprehensive regulatory knowledge spanning SEC, MiFID II, FINRA, CFTC requirements with specific compliance implementation guidance.	9
Microservice Boundaries Defined	7 distinct service domains	Well-structured service decomposition following domain-driven design with clear responsibilities and integration patterns.	9
Migration Phase Specificity	4 detailed implementation phases	Concrete timeline with weekly milestones, specific deliverables, and risk mitigation strategies for each phase.	9
Security Remediation Strategies	12 specific technical solutions	Detailed remediation using industry-standard tools (HashiCorp Vault, SPIFFE, AWS KMS) with implementation specificity.	8

Quantitative Score: 45/50 (90%)

Comprehensive technical analysis with exceptional regulatory coverage and specific implementation guidance. Strong performance across all financial system modernization categories.

Overall Assessment

Combined Score: 90/100 (90%)

DeepSeek R1 delivers specialist-level financial system analysis demonstrating deep expertise in trading system architecture, regulatory compliance, and migration risk management. The response provides immediately actionable recommendations for high-stakes financial system modernization while maintaining production safety standards.

Consulting Efficiency: Traditional analysis requiring 88 hours of specialized expertise ($15,000-$25,000) completed in minutes, representing 99%+ time reduction while exceeding typical consulting depth and specificity.

Recalibration Through Iterative Refinement

The comprehensive initial analysis revealed specific areas requiring deeper technical implementation focus. Following systematic prompting methodology, iterative refinement addresses team capabilities, technology specifics, and performance validation requirements.

Enhancement Areas Identified

Based on evaluation framework analysis, four critical areas emerged for technical refinement:

1. **Team Capability Assessment** (Gap Score: 6/10) - Migration strategy needs current skill evaluation and transition planning
2. **Technology Stack Specifications** (Gap Score: 7/10) - Framework selection requires detailed technical comparison and rationale
3. **Testing Strategy Depth** (Gap Score: 7/10) - Financial system testing demands comprehensive risk validation methodology
4. **Performance Validation Framework** (Gap Score: 6/10) - Production metrics need specific targets and measurement strategies

Technical Implementation Prompts

Follow-up Prompt 1: Team Transition Assessment

```
Analyze TradeTech Systems' current development team (15 Python developers, 5 DevOps
engineers, 3 DBAs) for microservices migration readiness. Provide:
- Individual skill gap analysis for microservices architecture
- 6-month training curriculum with specific technologies (Docker, Kubernetes, service
mesh)
- Team restructuring recommendations for service ownership model
- Risk mitigation for knowledge transfer during migration phases
```

Follow-up Prompt 2: Technology Stack Decision Matrix

```
Create detailed technology comparison for TradeTech's microservices stack selection:
- Message queue comparison (Kafka vs RabbitMQ vs AWS SQS) with financial system
requirements
- Database selection criteria (PostgreSQL vs DynamoDB vs combination approach)
- Service mesh evaluation (Istio vs Linkerd vs AWS App Mesh) for financial compliance
- Cost analysis and performance benchmarks for each technology choice
```

Follow-up Prompt 3: Financial System Testing Framework

```
Design comprehensive testing strategy for trading system migration including:
- Chaos engineering scenarios for financial system resilience validation
- Load testing protocols for $50M+ daily transaction volumes
- Compliance testing automation for SEC/FINRA requirements
- Migration testing methodology ensuring zero data loss and 99.9% uptime
```

Reader Exercise: Advanced Evaluation Application

Technical Assessment Practice: Apply the specialized evaluation framework to assess iterative responses for financial system modernization. This advanced exercise develops expertise in:

- Evaluating technical implementation feasibility under regulatory constraints
- Assessing migration risk mitigation strategies for production financial systems
- Comparing alternative technical approaches using quantitative criteria frameworks
- **Technical Resources**: Complete implementation analysis and code samples available at: GitHub Repository - Legacy System Modernization
- **Advanced Scoring**: Rate technical responses using financial system criteria (regulatory compliance, performance, security) and compare evaluations with provided specialist assessments to develop financial technology evaluation expertise.