Software Engineer Technical Assessment
Rule Engine Architecture & Implementation Challenge

This assessment focuses on **system architecture and coding quality**. A post-submission discussion will dive deeper into your implementation decisions, trade-offs, and how you'd extend the system.

Assessment Philosophy

We evaluate:

- Problem decomposition and systems thinking over rote coding
- Effective use of AI tools (encouraged and expected)
- Production-ready thinking (error handling, edge cases, testability)
- Code that clearly communicates intent and architecture

Note: Use any AI assistants. We'll discuss your AI usage during the post-submission discussion. Use it as much as you need but make sure you're able to explain your usage.

Part 1: The Scenario

Business Context

You're joining a fintech company providing real-time compliance calculations for e-commerce. The core system ("Compliance Engine") determines rule applicability and calculates fees/adjustments.

Current pain points:

- Monolithic architecture where it is difficult to add new rule types
- Poor separation between validation, evaluation, and calculation
- Limited observability during execution
- No graceful degradation when dependencies fail
- Testing is difficult due to tight coupling

Mission: Rebuild the core rule engine with a greenfield approach.

System Requirements

The Compliance Engine must:

- Accept transaction payloads (items, amounts, customer info, destination)
- Execute "gates" (validation checks) -any failure stops processing
- Determine applicability based on thresholds (e.g., merchant volume)
- Check exemptions (customer-level and item-level)
- Calculate fees by combining rate layers (state + county + city + category)
- Return detailed breakdown for debugging/auditing

Example: Fee Calculation

For a $100 SOFTWARE item shipped to Los Angeles, CA:

- State (CA): 6% → $6.00
- County (LA County): 0.25% → $0.25
- City (Los Angeles): 2.25% → $2.25
- Category modifier (SOFTWARE): 1% → $1.00
- **Total fee: $9.50 (9.5% effective rate)**

Expected Response Structure

The API must enable debugging and auditing. See example in Appendix A.

Part 2: Core Challenge

Task 1: Architecture Design

Deliverable: 1–2 pages architecture proposal with diagram

Address these key areas:

- Component Design: How to separate validation, applicability, exemptions, and calculation?
- Gate Pattern: Sequential checkpoints where any failure short-circuits
- Extensibility: Add new rule types without modifying existing code
- Observability: Ensure visibility for debugging/auditing
- Error Handling: Graceful degradation when external dependencies fail

Task 2: Implementation

Deliverable: Working rule engine implementation with tests

Language: Your choice (Python/TypeScript preferred, but use what showcases your strengths)

Requirements

Input format (simplified - see Appendix B for full structure):

- Transaction ID, merchant ID, customer ID
- Destination: country, state, city
- Items: id, category, amount
- Total amount and currency

Processing pipeline:

- Address Validation Gate: Verify destination is supported
- Applicability Gate: Check merchant presence requirements
- Exemption Check: Customer (WHOLESALE) and item (FOOD in some states)
- Fee Calculation: Apply state + county + city + category modifier rates

Output requirements:

- Gate pass/fail status with reasons
- Per-item fee breakdown (show each rate layer)
- Total fees calculated
- Audit trail of decisions

What We're Looking For

- Clean separation of concerns (separate validation, calculation, and orchestration)
- Proper error handling (not just happy path)
- Unit tests demonstrating the different paths
- Code clarity - it should read like documentation
- Testability - can you easily test components in isolation?

Task 3: Resilience & Edge Cases

Extend your implementation to handle:

- External Service Failure: Address validation service unavailable - how does system respond?
- Precision Handling: 47-line items must have consistent rounding (sum of item fees = total)
- Invalid Input: Missing required fields or malformed data

Deliverable: Tests demonstrating these scenarios (code changes optional if architecture already handles them)

Submission Guidelines

What to Submit

- Architecture document (PDF/Markdown with diagram)
- Code repository (GitHub link or zip) with README explaining how to run

README Must Include

- How to run the code (dependencies, setup, execution)
- How to run tests
- Any assumptions or simplifications made
- What you'd improve with more time

Post-Submission Discussion

Critical: This discussion is a mandatory part of the assessment. You must demonstrate deep understanding of the code you submit.

Expectation: You should be able to explain any part of your submission in real-time, walk through your code line-by-line if asked, and discuss implementation details without preparation. We're evaluating whether you truly understand what you've built, regardless of how much AI assistance you used.

We'll explore:

- Code walkthrough: "Explain how this function works" (expect to do this on the spot)
- Architecture decisions: "Why did you structure it this way?"
- Trade-off analysis: "What alternatives did you consider?"
- Real-time problem solving: "How would you add multi-currency support?"
- Edge case reasoning: "What happens if...?"
- Production considerations: "How would you monitor this in production?"
- AI collaboration approach: "How did you use AI? Where did you validate its suggestions?"

This is your opportunity to demonstrate mastery of your solution, explain choices we might misunderstand, discuss alternative approaches you considered, and show deeper engineering thinking beyond what code alone reveals.

Final Notes

- Perfect execution matters less than clear thinking and communication
- We expect some rough edges in the timeframe - focus on demonstrating your approach
- The discussion session is where we truly evaluate your engineering judgment
- Use AI heavily - it's a tool we want you proficient with - but ensure you understand everything it generates
- Ask questions if anything is unclear

Appendix A: Example Response Structure

Expected API response format:

{ "transactionId": "txn_123", "status": "CALCULATED", "gates": [{ "name": "ADDRESS_VALIDATION", "passed": true, "message": "Valid US address" }, { "name": "APPLICABILITY", "passed": true, "message": "Merchant above $100K threshold in CA" }, { "name": "EXEMPTION_CHECK", "passed": true, "appliedExemptions": [] }], "calculation": { "items": [{ "itemId": "item_1", "amount": 100.00, "category": "SOFTWARE", "fees": { "stateRate": { "jurisdiction": "CA", "rate": 0.06, "amount": 6.00 }, "countyRate": { "jurisdiction": "Los Angeles County", "rate": 0.0025, "amount": 0.25 }, "cityRate": { "jurisdiction": "Los Angeles", "rate": 0.0225, "amount": 2.25 }, "categoryModifier": { "category": "SOFTWARE", "rate": 0.01, "amount": 1.00 } }, "totalFee": 9.50 }], "totalFees": 9.50, "effectiveRate": 0.095 }, "auditTrail": ["Address validated via cache", "Merchant volume: $2.3M in CA (threshold: $100K)", "No exemptions applied", "Applied CA state rate: 6%", "Applied LA County rate: 0.25%", "Applied LA City rate: 2.25%", "Applied SOFTWARE category modifier: 1%"] }

Appendix B: Input Transaction Structure

Expected input format for transactions:

{ "transactionId": "txn_123", "merchantId": "merchant_456", "customerId": "customer_789", "destination": { "country": "US", "state": "CA", "city": "Los Angeles" }, "items": [{ "id": "item_1", "category": "SOFTWARE", "amount": 100.00 }, { "id": "item_2", "category": "PHYSICAL_GOODS", "amount": 50.00 }], "totalAmount": 150.00, "currency": "USD" }

Test Data Assumptions

For your implementation, you can hardcode these simplified rules:

Supported Jurisdictions:

1. US states: CA, NY, TX
2. CA cities: Los Angeles, San Francisco, San Diego
3. NY cities: New York City, Buffalo

Merchant Thresholds (simplified):

1. merchant_456: $2.3M in CA (above $100K threshold)
2. merchant_789: $50K in NY (below $100K threshold)

Exemptions:

1. Customer type WHOLESALE: fully exempt
2. FOOD category: exempt in CA, not exempt in NY

Rate Structure:

1. CA: 6% state, 0.25% county (if LA County), 2.25% city (if LA)
2. NY: 4% state, 0.5% county, 1% city (if NYC)
3. Category modifiers: SOFTWARE +1%, PHYSICAL_GOODS +0%

Note: These are simplified rules for testing. Real systems would load from databases/APIs.