Wise Mobile Engineering Lead/Manager — Full Interview Prep

Based on Reddit Intel + Candidate Pack Analysis

ROUND 1: Introductory Call (30 mins) — Recruiter

Status: Friendly, no technical questions. Already prepped. ✅

ROUND 2: Pair Programming (60 mins) — Two Engineers, HackerRank

What Reddit Tells Us

The question format is **inconsistent** — candidates report getting different types:

Type	Example	Frequency
Practical/Functional	Adding functionality to existing code, handling HTTP requests	Most common
LeetCode-style	Finding best combinations of ints that sum to a threshold	Sometimes
Cache design	LRU cache with TTL (currency conversion context)	Reported by friends of candidates

Key takeaway: Prepare for ALL three types. Don't assume it'll be pure LC or pure practical.

Type 1: Practical/Functional Questions (MOST LIKELY)

These involve working with **existing code** and adding features. Given Wise is a fintech mobile company, expect scenarios around:

HTTP Request Handling

Since one candidate was caught off-guard by this, prep thoroughly:

```swift
```

```swift
// Pattern 1: Basic API call with error handling
func fetchExchangeRate(from: String, to: String) async throws -> ExchangeRate {
    let url = URL(string: "https://api.wise.com/v1/rates?source=\(from)&target=\(to)")!
    let (data, response) = try await URLSession.shared.data(from: url)

    guard let httpResponse = response as? HTTPURLResponse,
          (200...299).contains(httpResponse.statusCode) else {
        throw APIError.invalidResponse
    }

    return try JSONDecoder().decode(ExchangeRate.self, from: data)
}

// Pattern 2: Retry logic with exponential backoff
func fetchWithRetry<T: Decodable>(url: URL, maxRetries: Int = 3) async throws -> T {
    var lastError: Error?
    for attempt in 0..<maxRetries {
        do {
            let (data, _) = try await URLSession.shared.data(from: url)
            return try JSONDecoder().decode(T.self, from: data)
        } catch {
            lastError = error
            let delay = pow(2.0, Double(attempt)) // 1s, 2s, 4s
            try await Task.sleep(nanoseconds: UInt64(delay * 1_000_000_000))
        }
    }
    throw lastError!
}

// Pattern 3: Concurrent requests (fetch multiple rates simultaneously)
func fetchMultipleRates(pairs: [(String, String)]) async throws -> [ExchangeRate] {
    try await withThrowingTaskGroup(of: ExchangeRate.self) { group in
        for (from, to) in pairs {
            group.addTask {
                try await self.fetchExchangeRate(from: from, to: to)
            }
        }
        var rates: [ExchangeRate] = []
        for try await rate in group {
            rates.append(rate)
        }
        return rates
    }
}
```

Adding Features to Existing Code

They'll give you a codebase and ask you to extend it. Practice:

```swift
```

```
// Example: Given a basic TransferService, add validation + fee calculation

// EXISTING CODE (they provide this)
class TransferService {
    func createTransfer(amount: Decimal, sourceCurrency: String, targetCurrency: String) -> Transfer {
        return Transfer(amount: amount, source: sourceCurrency, target: targetCurrency)
    }
}

// WHAT THEY MIGHT ASK YOU TO ADD:
// 1. Input validation (min/max amounts, valid currency codes)
// 2. Fee calculation based on payment method
// 3. Rate expiry handling (rates are only valid for X seconds)
// 4. Error handling for edge cases

class ImprovedTransferService {
    private let rateService: RateService
    private let feeCalculator: FeeCalculator

    func createTransfer(amount: Decimal,
                        sourceCurrency: String,
                        targetCurrency: String,
                        paymentMethod: PaymentMethod) throws -> Transfer {
        // Validation
        guard amount > 0 else { throw TransferError.invalidAmount }
        guard SupportedCurrencies.contains(sourceCurrency) else { throw TransferError.unsupportedCurrency }
        guard amount <= maxLimit(for: sourceCurrency) else { throw TransferError.exceedsLimit }

        // Get rate (with expiry check)
        let rate = try rateService.getRate(from: sourceCurrency, to: targetCurrency)
        guard !rate.isExpired else { throw TransferError.rateExpired }

        // Calculate fee
        let fee = feeCalculator.calculate(amount: amount, method: paymentMethod)
        let convertedAmount = (amount - fee) * rate.value

        return Transfer(
            amount: amount,
            fee: fee,
            convertedAmount: convertedAmount,
            rate: rate,
            source: sourceCurrency,
            target: targetCurrency
        )
```

```
    }
}
```

Data Parsing / Transformation

Common in fintech interviews — processing transaction data:

```swift
// Parse and group transactions by currency
func groupTransactionsByCurrency(_ transactions: [Transaction]) -> [String: [Transaction]] {
    Dictionary(grouping: transactions, by: { $0.currency })
}

// Calculate running balance from transactions
func runningBalance(from transactions: [Transaction]) -> [(Transaction, Decimal)] {
    var balance: Decimal = 0
    return transactions.map { transaction in
        balance += transaction.type == .credit ? transaction.amount : -transaction.amount
        return (transaction, balance)
    }
}
```

Type 2: LeetCode-Style Questions

Combination Sum (reported by Reddit user)

"Finding best combinations of ints that add up to a given threshold"

```swift
```

```swift
// Classic Combination Sum — find all combinations that sum to target
func combinationSum(_ candidates: [Int], _ target: Int) -> [[Int]] {
    var result: [[Int]] = []
    var current: [Int] = []

    func backtrack(_ start: Int, _ remaining: Int) {
        if remaining == 0 {
            result.append(current)
            return
        }
        if remaining < 0 { return }

        for i in start..<candidates.count {
            current.append(candidates[i])
            backtrack(i, remaining - candidates[i]) // same element can be reused
            current.removeLast()
        }
    }

    backtrack(0, target)
    return result
}

// Variation: Find the BEST combination (fewest elements / closest to target)
func bestCombination(_ candidates: [Int], _ target: Int) -> [Int] {
    var bestResult: [Int] = []
    var current: [Int] = []

    func backtrack(_ start: Int, _ remaining: Int) {
        if remaining == 0 {
            if bestResult.isEmpty || current.count < bestResult.count {
                bestResult = current
            }
            return
        }
        if remaining < 0 { return }

        for i in start..<candidates.count {
            current.append(candidates[i])
            backtrack(i + 1, remaining - candidates[i])
            current.removeLast()
        }
    }

    backtrack(0, target)
```

```
    return bestResult
}
```

Two Sum / Subset Sum Variations (common follow-ups)

swift

```swift
// Two Sum
func twoSum(_ nums: [Int], _ target: Int) -> [Int] {
    var map: [Int: Int] = [:]
    for (i, num) in nums.enumerated() {
        if let j = map[target - num] { return [j, i] }
        map[num] = i
    }
    return []
}

// Partition Equal Subset Sum (DP)
func canPartition(_ nums: [Int]) -> Bool {
    let total = nums.reduce(0, +)
    if total % 2 != 0 { return false }
    let target = total / 2
    var dp = Set<Int>([0])
    for num in nums {
        dp = dp.union(dp.map { $0 + num })
    }
    return dp.contains(target)
}
```

Type 3: LRU Cache with TTL (Currency Conversion Context)

This is VERY likely given Wise's domain. Exchange rates expire — caching with TTL is a real problem they solve.

swift

```
// LRU Cache — basic implementation
class LRUCache<Key: Hashable, Value> {
    private let capacity: Int
    private var cache: [Key: Node<Key, Value>] = [:]
    private let head = Node<Key, Value>() // dummy head
    private let tail = Node<Key, Value>() // dummy tail

    class Node<K, V> {
        var key: K?
        var value: V?
        var prev: Node<K, V>?
        var next: Node<K, V>?
        init(key: K? = nil, value: V? = nil) {
            self.key = key
            self.value = value
        }
    }

    init(capacity: Int) {
        self.capacity = capacity
        head.next = tail
        tail.prev = head
    }

    func get(_ key: Key) -> Value? {
        guard let node = cache[key] else { return nil }
        moveToHead(node)
        return node.value
    }

    func put(_ key: Key, _ value: Value) {
        if let node = cache[key] {
            node.value = value
            moveToHead(node)
        } else {
            let newNode = Node(key: key, value: value)
            cache[key] = newNode
            addToHead(newNode)
            if cache.count > capacity {
                if let lru = removeTail() {
                    cache.removeValue(forKey: lru.key!)
                }
            }
        }
    }
```

```
    private func addToHead(_ node: Node<Key, Value>) {
        node.prev = head
        node.next = head.next
        head.next?.prev = node
        head.next = node
    }

    private func removeNode(_ node: Node<Key, Value>) {
        node.prev?.next = node.next
        node.next?.prev = node.prev
    }

    private func moveToHead(_ node: Node<Key, Value>) {
        removeNode(node)
        addToHead(node)
    }

    private func removeTail() -> Node<Key, Value>? {
        let lru = tail.prev
        if lru === head { return nil }
        removeNode(lru!)
        return lru
    }
}

// LRU Cache WITH TTL — the Wise-specific version
class TTLCache<Key: Hashable, Value> {
    private let cache: LRUCache<Key, (value: Value, expiry: Date)>
    private let ttl: TimeInterval

    init(capacity: Int, ttlSeconds: TimeInterval) {
        self.cache = LRUCache(capacity: capacity)
        self.ttl = ttlSeconds
    }

    func get(_ key: Key) -> Value? {
        guard let entry = cache.get(key) else { return nil }
        if Date() > entry.expiry {
            // Expired — treat as cache miss
            return nil
        }
        return entry.value
    }

    func put(_ key: Key, _ value: Value) {
        let expiry = Date().addingTimeInterval(ttl)
        cache.put(key, (value: value, expiry: expiry))
```

```
    }
}

// USAGE — Currency Rate Cache (30 second TTL, like Wise's real rates)
let rateCache = TTLCache<String, Decimal>(capacity: 100, ttlSeconds: 30)

func getExchangeRate(from: String, to: String) async throws -> Decimal {
    let key = "\(from)-\(to)"

    // Check cache first
    if let cached = rateCache.get(key) {
        return cached
    }

    // Cache miss — fetch from API
    let rate = try await fetchRateFromAPI(from: from, to: to)
    rateCache.put(key, rate)
    return rate
}
```

Pair Programming Tips (from the candidate pack)

1. **Think out loud** — They want to hear your thought process, not just see code
2. **Ask clarifying questions FIRST** — "What's the expected input size?" "Should I handle edge cases for invalid currencies?" "Is thread safety a concern?"
3. **Build iteratively** — Start with a brute force solution, then optimize
4. **Respond to hints** — The interviewers will guide you. Don't ignore their input.
5. **Don't panic if it's unfamiliar** — One candidate "winged it" and still passed

Practice on HackerRank

The interview uses HackerRank's collaborative coding platform. Familiarize yourself with:

- The editor (syntax highlighting, auto-complete behavior)
- Running code in-browser
- Swift/language selection

ROUND 3: Technical/System Design Interview (60 mins) — Two Engineers

What Reddit Tells Us

> "The pattern was **contention and scaling writes/reads** with a **financial twist**. They led most of it. I didn't justify my trade-offs and got confused."

> From the candidate pack: "They'll provide details on a hypothetically existing feature in a mobile app, ask how you'd expect it to be built, then propose improvements."

What They're Looking For (from candidate pack):

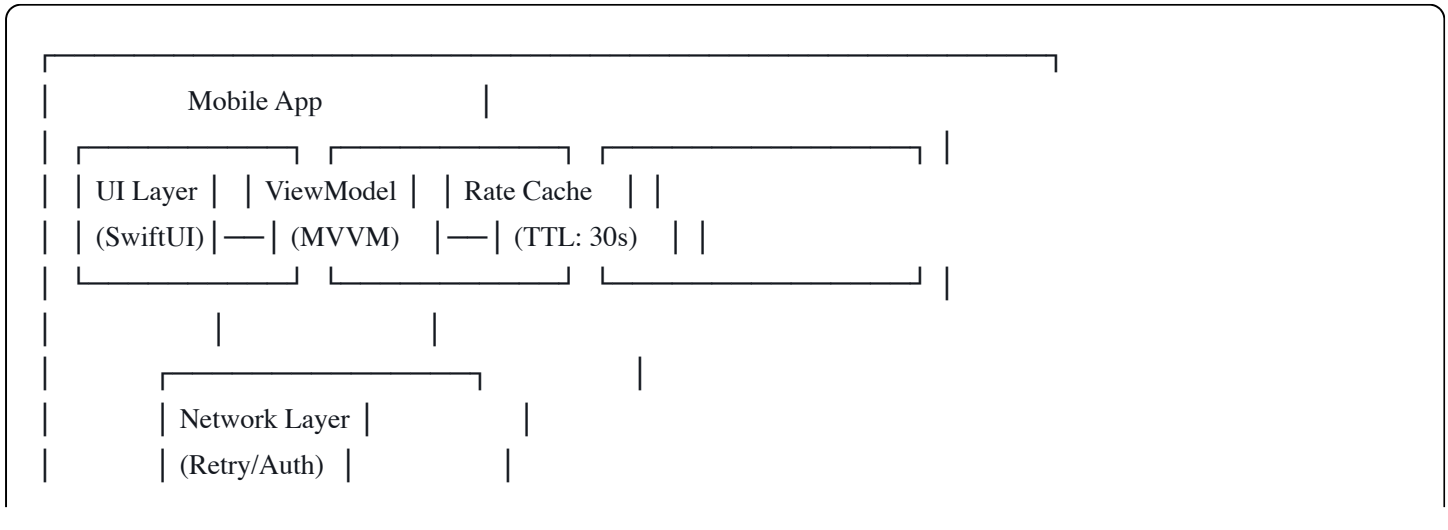
- How you break it down into discrete units of work
- What libraries you'd use
- What questions you'd ask about underlying implementation
- Concerns for backend, product, design
- Best practices, design patterns, scalable solutions

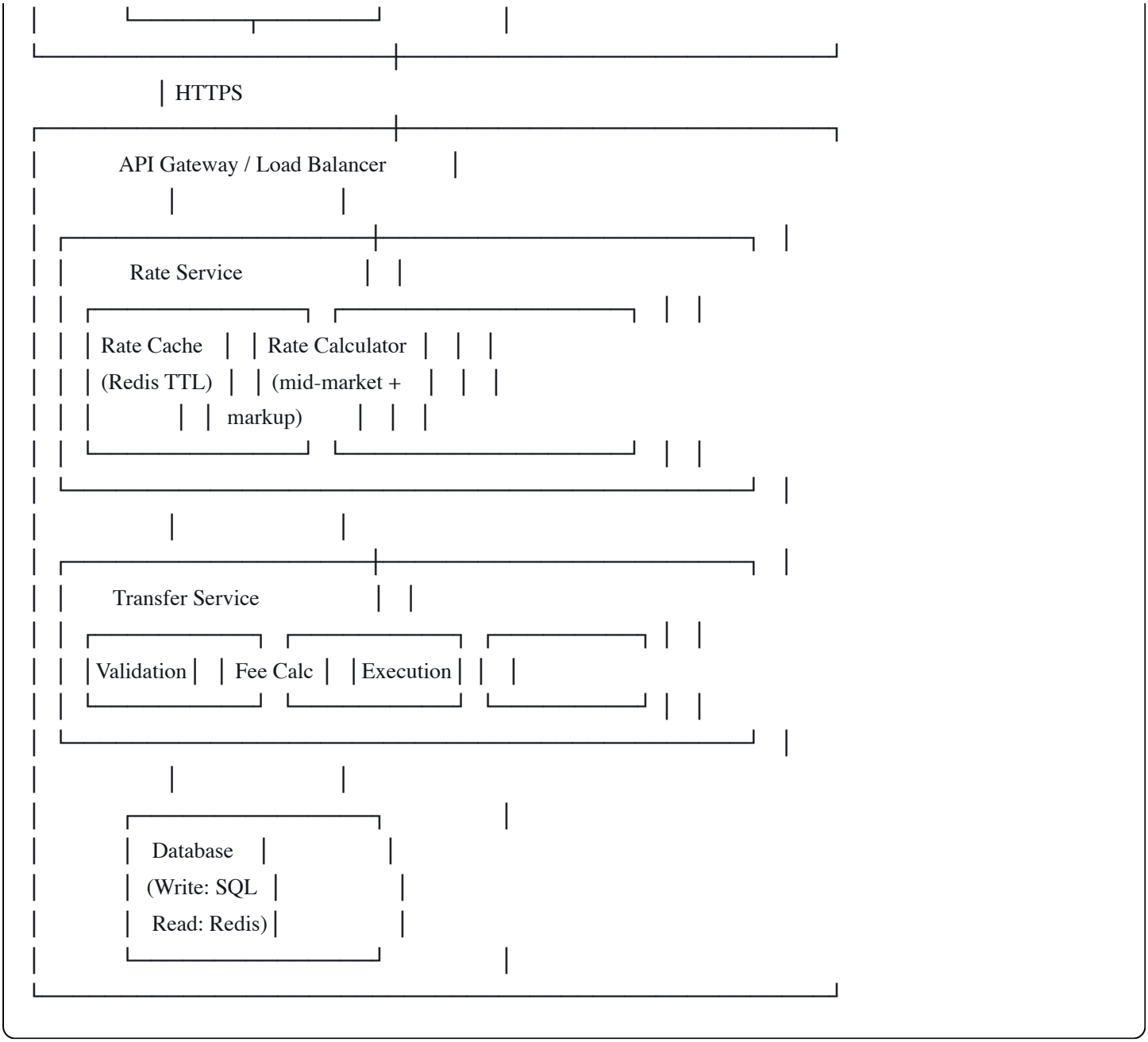
Likely Scenario: Currency Conversion / Transfer Feature

Step 1: Clarifying Questions (ALWAYS START HERE)

> "Before I dive in, let me ask a few questions:
>
> - What's the expected user volume? Thousands or millions of concurrent users?
> - What's the acceptable latency for showing a rate to the user?
> - How long is a quoted rate valid? (Wise uses ~30 seconds in production)
> - Do we need to handle offline scenarios?
> - Is this iOS only, or cross-platform?
> - What's the current architecture — monolith or microservices backend?"

Step 2: High-Level Architecture





Step 3: Contention & Scaling (THE KEY TOPIC)

The financial twist: You can't lose money due to race conditions.

Problem: Rate Contention

- Thousands of users request a rate at the same time
- Rates change every few seconds
- User sees rate → takes 30 seconds to confirm → rate may have changed
- If you honor the old rate, you lose money. If you don't, user experience suffers.

Solution: Rate Locking with Guaranteed Quote

1. User requests quote → Server returns:

```
{
  "quoteId": "uuid-123",
  "rate": 1.2345,
  "expiresAt": "2026-02-16T10:01:30Z",  // 30-second window
  "sourceAmount": 1000,
  "targetAmount": 1234.50,
  "fee": 4.50
}
```

2. User confirms within window → Use guaranteed rate
3. User confirms after window → Fetch new rate, show updated quote

Trade-off to articulate:

- Short TTL (30s) = Less financial risk, worse UX (user has to re-confirm)
- Long TTL (5 min) = Better UX, higher financial risk if rates move
- Wise chose ~30 seconds — this balances risk and conversion rate

Problem: Scaling Writes (Transfer Creation)

- Millions of transfers created daily
- Each transfer must be atomic (debit source, credit target)
- Can't double-charge or lose money

Solution: Event-Driven with Idempotency

Mobile App → API (with idempotency key) → Message Queue → Transfer Processor

- Idempotency key: Client generates UUID, includes in every request
  If network fails and client retries, same UUID = same transfer (no duplicates)

- Write path:
  1. Validate quote (check not expired)
  2. Write transfer to DB with status=CREATED
  3. Publish to message queue
  4. Processor picks up → executes payment → updates status
  5. Push notification to mobile when complete

- Read path (much simpler):
  1. Mobile polls or uses WebSocket for status updates
  2. Read from read replica / cache for transfer history
  3. Heavy reads (history, statements) served from read replicas

Trade-off to articulate:

- Synchronous processing = Simpler, but blocks if payment network is slow
- Async with queue = More complex, but resilient and scalable
- Wise almost certainly uses async — they process £36B/quarter

Problem: Scaling Reads (Transfer History / Dashboard)

- Users check their balance and history frequently
- Read:Write ratio is probably 100:1 or higher

Solution: CQRS (Command Query Responsibility Segregation)

Writes → Primary DB (strong consistency for financial transactions)
Reads → Read Replicas / Redis Cache (eventual consistency OK for history)

Mobile caches:
- Balance: Cache locally, refresh on app foreground + after any transfer
- History: Paginated, cached locally, pull-to-refresh
- Rates: TTL cache (30 seconds)

Step 4: Mobile-Specific Architecture

swift

```
// MVVM + Clean Architecture for the Transfer Feature

// Domain Layer
struct Quote {
    let id: String
    let rate: Decimal
    let sourceAmount: Decimal
    let targetAmount: Decimal
    let fee: Decimal
    let expiresAt: Date

    var isExpired: Bool { Date() > expiresAt }
}

// Repository (abstracts data source)
protocol TransferRepository {
    func getQuote(source: String, target: String, amount: Decimal) async throws -> Quote
    func createTransfer(quoteId: String) async throws -> Transfer
    func getTransferHistory(page: Int) async throws -> [Transfer]
}

// ViewModel
@MainActor
class TransferViewModel: ObservableObject {
    @Published var quote: Quote?
    @Published var isLoading = false
    @Published var error: TransferError?
    @Published var countdownSeconds: Int = 30

    private let repository: TransferRepository
    private var countdownTimer: Timer?

    func fetchQuote(source: String, target: String, amount: Decimal) async {
        isLoading = true
        do {
            quote = try await repository.getQuote(source: source, target: target, amount: amount)
            startCountdown()
        } catch {
            self.error = .quoteFetchFailed
        }
        isLoading = false
    }

    func confirmTransfer() async {
        guard let quote = quote, !quote.isExpired else {
            self.error = .quoteExpired
```

```
            return
        }
        // Create transfer with idempotency
        do {
            let transfer = try await repository.createTransfer(quoteId: quote.id)
            // Navigate to success screen
        } catch {
            self.error = .transferFailed
        }
    }

    private func startCountdown() {
        countdownSeconds = 30
        countdownTimer = Timer.scheduledTimer(withTimeInterval: 1, repeats: true) { _ in
            self.countdownSeconds -= 1
            if self.countdownSeconds <= 0 {
                self.countdownTimer?.invalidate()
                self.quote = nil // Force re-fetch
            }
        }
    }
}
```

Step 5: Questions YOU Should Ask Them

- "What's the current caching strategy for rates on the mobile side?"
- "How do you handle rate changes mid-transfer on the client?"
- "Is the backend event-driven or synchronous for transfer processing?"
- "How do you handle offline scenarios — queuing transfers?"

Common Design Patterns to Know

Pattern	When to Use	Wise Context
MVVM	UI architecture	Standard for iOS at Wise
Repository	Data abstraction	Cache vs. network decisions
CQRS	Read/write separation	Transfer history vs. creation
Event Sourcing	Audit trail	Financial transactions need full history
Circuit Breaker	External service failure	Payment network goes down

Pattern	When to Use	Wise Context
Idempotency	Prevent duplicates	Network retry → don't double-charge
Saga Pattern	Distributed transactions	Multi-currency transfers

---

How to Structure Your Answers

Use this framework (the Reddit candidate failed by not justifying trade-offs):

1. CLARIFY — Ask questions about constraints
2. HIGH LEVEL — Draw the architecture (mobile + backend)
3. DEEP DIVE — Focus on the hard part (contention/scaling)
4. TRADE-OFFS — "I chose X over Y because..."
   ↑ THIS IS WHERE THE REDDIT CANDIDATE FAILED
5. MOBILE SPECIFICS — Architecture patterns, caching, offline
6. EXTENSIONS — "If we had more time, I'd also consider..."

Trade-off Template (memorize this):

> "I'd go with **[Choice A]** over **[Choice B]** because **[reason]**. The downside is **[limitation]**, but in Wise's context where **[specific constraint]**, I think that trade-off is acceptable. If the requirements changed to **[scenario]**, I'd reconsider."

ROUND 4: Product Interview (60 mins) — Product Manager

What Reddit Tells Us

> "Questions around giving examples of how I have prior experience defining KPIs for improving customer experience. To me it's a product side question."

The Reddit candidate's mistake: They dismissed it as "product stuff that devs don't do." Wise WANTS engineers who think about product. This is a cultural value — "Customers > team > ego."

Customer Perspective Questions

"Why do you think customers need Wise?"

> "Traditional banks charge 3-5% on international transfers through hidden exchange rate markups. I've experienced this personally — and at PayPal, I work on the BFX system where I see how currency conversion fees work behind the scenes. Most customers don't even realize they're being charged.
> Wise solves this with radical transparency — showing the mid-market rate and a clear fee upfront. For someone in India sending money to family abroad, or a freelancer in Singapore billing clients in USD, that

> transparency and lower cost make a real difference.
> What might prevent them? Trust is the big one — moving money is emotional. People need to trust that their money won't disappear. Also, for larger amounts, some customers might feel safer with a traditional bank even if it costs more."

"What types of customers does Wise serve?"

> "I see several personas:
>
> - **Expat workers** sending money home (e.g., workers in Singapore sending to India, Philippines)
> - **Freelancers/remote workers** receiving payments in foreign currencies
> - **Small businesses** paying international suppliers or receiving from global customers
> - **Travelers** who need multi-currency spending via the Wise card
> - **Digital nomads** managing money across multiple countries
> - **Enterprise/Platform customers** (via Wise Platform) — banks and fintechs embedding Wise's infrastructure
>
> The one area I think has room for growth is the **investment/savings** use case — Wise Assets is already expanding into this, letting customers earn returns on held balances. That bridges the gap between 'transfer tool' and 'financial home.'"

Metrics / KPI Questions

"What metrics do you value?"

> "I think about metrics in layers:
> **Customer-facing metrics (most important):**
>
> - Transfer completion rate — what % of users who start a transfer actually complete it?
> - Time to deliver — how fast does money arrive?
> - Customer effort score — how many steps/taps to complete a transfer?
> - App crash-free rate — at PayPal we target 99.8%
>
> **Engineering health metrics:**
>
> - App startup time and screen load times (p50, p95, p99)
> - API response times
> - Error rates by endpoint
> - Build and deployment frequency
>
> **Business metrics I'd track as an engineering leader:**
>
> - Feature adoption rate for new mobile features
> - Funnel drop-off — where exactly are users abandoning?
> - Mobile vs. web conversion rates

> At PayPal, I tracked checkout completion rates and identified that a 30% reduction in checkout time directly improved conversion. That's the kind of connection between engineering metrics and business outcomes I'd bring to Wise."

"How do you prioritize between new features and technical debt?"

> "I use a framework I've developed over time:
> **For tech debt, I ask:** Is this debt actively hurting customers (crashes, slow performance), blocking new feature delivery (slowing dev velocity), or creating risk (security, compliance)? If yes to any, it competes directly with features.
> **My approach at PayPal:** I advocated for dedicating ~20% of each sprint to tech debt. I'd quantify the cost — for example, 'This legacy module causes 3 bugs per sprint, each taking 2 days to fix. That's 6 days of lost feature work per sprint. Refactoring it takes 10 days but saves us 6 days every sprint going forward.'
> **For prioritization between features:** I think about impact (how many customers affected × how much value) vs. effort (engineering days). I like using a simple 2×2: high impact/low effort → do first; high impact/high effort → plan carefully; low impact → question whether to do at all.
> The key is that engineering leads should be in the room when these decisions are made, not just receiving requirements. At Wise, with the empowered squad model, I imagine engineers have even more ownership over this prioritization."

"How would you know if a product you've built was successful?"

> "I'd define success criteria BEFORE building, not after. Here's how I'd approach it:
> **Example — If we're building a new 'Scheduled Transfers' feature:**
> Before building:
> - Success metric: 15% of repeat transfer users adopt scheduled transfers within 3 months
> - Secondary: Reduces average time-to-transfer for repeat routes by 50%
> - Guard metric: No increase in failed transfers or support tickets
>
> After launch:
> - Week 1: Is the feature working? (Error rates, crash-free rate)
> - Month 1: Are users discovering and trying it? (Adoption rate)
> - Month 3: Are users retaining? (Repeat usage, not just trial)
> - Month 6: Is it moving the business needle? (Transfer volume impact)
>
> I'd also look at qualitative signals — App Store reviews mentioning the feature, support ticket themes, and user feedback."

STAR Format Examples (from YOUR experience)

Example 1: Defining KPIs that improved customer experience

> **Situation:** At PayPal, the Express Checkout flow on iOS had a high drop-off rate, but nobody was measuring exactly where users were abandoning.
> **Task:** I took ownership of defining mobile-specific KPIs to identify and fix the drop-off points.

> **Action:** I worked with the product team to instrument the checkout funnel with detailed analytics — tracking time spent on each screen, tap-to-response latency, error rates per step, and where users hit "back" vs. closing the app. I identified that the payment method selection screen was causing a 15% drop-off due to slow loading of saved cards. I optimized the local caching strategy to pre-fetch payment methods on app launch.
> **Result:** Checkout completion rate improved, and checkout time reduced by 30%. This directly impacted transaction volume. I also established these KPIs as a standard dashboard that the team now uses for every release.

Example 2: Customer-first technical decision

> **Situation:** On the BFX project, we were ramping up currency conversion in new countries (Italy, Saudi Arabia, Lithuania). The product team wanted to launch all countries simultaneously.
> **Task:** As the engineer owning the ramp, I needed to balance speed of launch with risk to customers.
> **Action:** I advocated for a gradual country-by-country rollout instead. I built monitoring dashboards to track conversion success rates, FX spread accuracy, and error rates per country. After each country went live, we'd validate for a week before proceeding.
> **Result:** We caught a BIN configuration issue in Italy that would have affected card transactions in Saudi Arabia if we'd launched simultaneously. The phased approach protected customers from payment failures, and we still completed the full ramp within the original timeline.

Example 3: Cross-team collaboration for customer impact

> **Situation:** The RUNE payment feature required coordination between the iOS team in Chennai, the FRED team in San Jose, and the QA team testing in multiple environments.
> **Task:** After a bug was filed where the Send/Request page displayed abnormally after login, I needed to diagnose and coordinate a fix across three teams in different time zones.
> **Action:** I analyzed the issue — an experiment flag was returning nil during fresh login but working correctly on app relaunch. I documented the root cause clearly, tagged the relevant experiment team, and provided the exact reproduction steps with app build versions. I also created a knowledge base entry so similar issues could be diagnosed faster in the future.
> **Result:** The bug was identified and routed to the correct team within 24 hours instead of the typical 3-5 day ping-pong between teams. More importantly, I established a debugging playbook for experiment-related display issues.

Motivation Questions

"Why Wise over other fintech companies?"

> "I could go to Grab, which has financial services, or stay at PayPal. But Wise's single-minded focus on making international money movement cheaper and faster is what draws me. PayPal does a thousand things. Wise does one thing and does it obsessively well.
> Also, Wise's transparency is unique — publicly sharing fee structures, publishing the tech stack on Medium, the Product Career Map with visible compensation bands. That level of openness is rare and signals a culture I'd thrive in."

"What would make you NOT want to work at Wise?"

> "If the engineering team didn't have real ownership over product decisions — if engineers were just ticket-takers implementing specs from product managers. But from everything I've researched, Wise's squad model gives engineers genuine ownership, which is exactly what I want.
> I'd also be concerned if the Singapore team felt like a satellite office rather than a real engineering hub. But the recent office expansion and doubling of the team since 2022 tells me Singapore is a priority, not an afterthought."

ROUND 5: Final Interview (60 mins) — Two Engineering Leads, In-Person

Focus: Culture Fit, Values, Leadership

Leadership Questions to Prepare

"How do you handle a low performer on your team?"

> "First, I make sure expectations are clear — sometimes low performance is actually unclear expectations. I'd have a direct 1:1 conversation focused on specifics, not generalizations. 'In the last sprint, these three tasks slipped. Help me understand what happened.' I'd create a clear improvement plan with measurable goals and check in weekly. If after genuine support they're still not meeting the bar, I'd have an honest conversation about whether this is the right role for them. At Wise, the 'no drama, good karma' value resonates here — be kind but direct."

"How do you handle disagreements with product managers?"

> "Data first. If I disagree with a product direction, I bring evidence — user metrics, technical feasibility analysis, examples from other companies. I don't say 'I think this is wrong,' I say 'The data suggests X, which makes me concerned about Y.' If after discussion we still disagree, I disagree and commit. Blocking progress over ego goes against everything Wise stands for."

"How do you build a high-performing mobile team?"

> "Three pillars: ownership, growth, and trust. I give engineers end-to-end ownership of features, not just 'implement this ticket.' I invest in their growth through code reviews that teach, not just gatekeep, and by giving stretch opportunities. And I build trust by being transparent about company/team direction and by shielding the team from unnecessary organizational noise — which ties directly to 'no drama, good karma.'"

Questions to Ask Engineering Leads

1. "What's the biggest technical challenge the mobile team is facing right now?"
2. "How do mobile engineers interact with the product team day-to-day?"
3. "What does the career path from Engineering Lead to Engineering Manager look like at Wise?"
4. "What's one thing you wish someone had told you before joining Wise?"
5. "How does the team handle on-call and incident response for mobile?"

COMPENSATION NOTES

Reddit insight: "I did get the offer but RSUs were a joke so I joined a startup!"

Be prepared for:

- Wise RSUs are based on the London Stock Exchange listing (WISE.L)
- Check current stock price before negotiating
- Ask about vesting schedule (typically 4 years with 1-year cliff)
- Base salary in Singapore for this level: likely SGD 12,000-18,000/month
- Total compensation = Base + RSUs + Benefits
- Wise is known for lower base but meaningful RSU grants
- Ask: "Can you share the compensation band for this level on the Product Career Map?"

Negotiation tip: If the RSU grant seems low, ask about refresh grants and the performance review cycle. Sometimes the initial grant is modest but annual refreshes are generous.

GLASSDOOR PREP

The Reddit user mentioned seeing the pair programming question on Glassdoor. Before your interview:

1. Search "Wise interview questions" on Glassdoor
2. Filter for "Engineering" and "Singapore"
3. Look for recent (last 6 months) pair programming questions
4. Practice any code questions you find there

OVERALL STRATEGY

Round	Your Advantage	Your Risk	Mitigation
Recruiter	FX domain expertise	Notice period/location	Emphasize readiness to relocate
Pair Programming	Strong iOS skills	Unexpected question format	Prep all 3 types (practical, LC, cache)
System Design	Real fintech architecture experience	Not justifying trade-offs	Practice the trade-off template
Product	BFX project = real KPI experience	Dismissing it as "product work"	Embrace product thinking with examples

Round	Your Advantage	Your Risk	Mitigation
Final	Cross-cultural leadership	May probe why short stints	Frame positively, show commitment