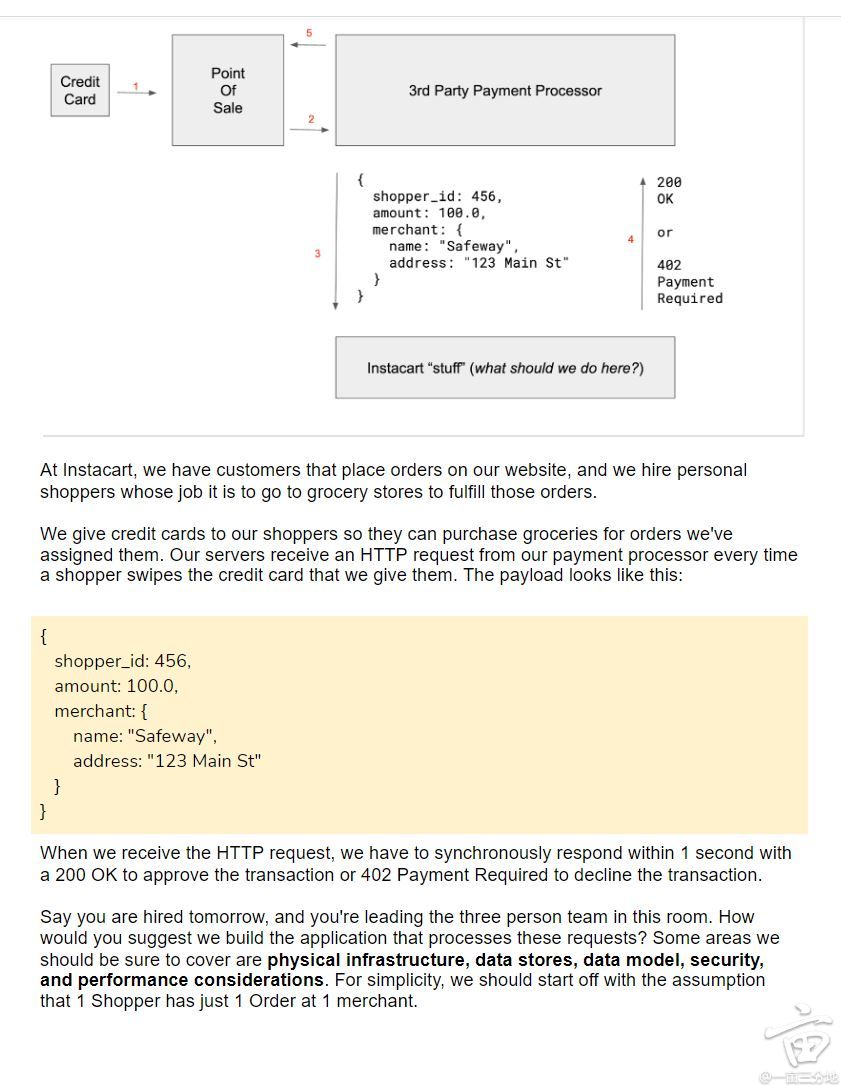
**Design Payment Service**



**ZH：题目里给的 payload 没有交易 ID。我会先问清楚：“PSP 会不会在请求里带上一个唯一的交易编号？如果没有，我们是不是可以用组合键，比如 (shopper\_id, amount, merchant, timestamp) 来生成一个近似唯一的 key？” 然后我会解释：在真实系统里 PSP 一般都会带 transaction id，我会把它存成唯一键来避免重复处理。**

**EN: The given payload doesn’t include a transaction ID. I’d clarify with the interviewer: *“Does the PSP provide a unique transaction identifier in the request? If not, can we construct a surrogate key, say (shopper\_id, amount, merchant, timestamp)?”* In real-world systems, PSPs usually send a transaction id, and I’d store it as a unique key to guarantee we don’t double-process retries.**

**题目核心 (核心场景和考察点)**

Instacart 给每个 shopper 一张信用卡。shoppers 去超市结账时，支付处理器会向 Instacart 的 **Payment Verification API** 发起 HTTP 请求，包含：

{

shopper\_id: 123,

amount: 300,

merchant: {

name: "Target",

address: "123 Main St"

}

}

系统必须 **在 1 秒内返回**：

* + 200 OK → 批准交易
  + 402 Payment Required → 拒绝交易

**限制假设**：  
一个 shopper 同时只服务一个订单，且只在一个商家购物（简化）。

* **考察重点**：
  + API 设计（输入、输出、幂等性、错误处理）
  + 数据模型 & 存储选择（SQL/NoSQL，索引，事务一致性）
  + 性能 (1 秒 SLA，低延迟，QPS/TPS scaling)
  + 安全性 (mTLS, OIDC/JWT, 数据加密)
  + 可扩展性 (水平扩展，缓存策略，分布式架构)
  + 稳定性/容错 (failover，重试，rollback，ACID vs eventual consistency)

**常见 Follow-up 问题分类**

**1. API & Transaction Handling**

* 如何设计 API (幂等性、防止重复扣款，Unique key)？
* 如果同一个 shopper 刷了两次卡，如何保证只扣一次？
* 如果同时收到两个 request（金额不同），怎么决定？
* 如何处理 transaction rollback / error handling？

**2. 数据模型 & 存储**

* DB 用 SQL 还是 NoSQL？为什么？
* 如何设计 table schema（订单、shoppers、merchant、transaction logs）？
* 如果 DB 被多个 service 共享，如何保证性能？
* 如何 scale database？
* 如何利用 index 优化查询？

**3. 性能 & 可扩展性**

* 如何在 1 秒 SLA 内完成验证？
* 如何加 cache？存什么？cache 写策略（write-through, write-around, write-back）？
* 如何做负载均衡 (load balancing)？
* QPS/TPS 提升方案？

**4. 安全性**

* 如何保证请求来自可信的 PSP（第三方支付处理器）？
* 使用什么认证方式 (mTLS, JWT/OIDC)？
* 如何防止数据泄露、篡改？

**5. 稳定性 & 容错**

* 如果 DB crash/断电，如何保证交易数据安全？（日志、binlog、replication）
* 如何做 failover？
* 如何做 backup & recovery？

**6. 业务逻辑 & 风控**

* 金额不完全匹配（Instacart 价格 vs 店内实际价格）：
  + 正常误差（重量、时价波动） vs 异常（欺诈）如何判断？
  + 如何利用历史数据做 risk evaluation？
* 如果 shopper 在 **不合作的商家** 刷卡怎么办？
* 如果订单需要多次刷卡（太重分开买，库存不足等），如何处理？
* 如果 shopper 买的数量不同于订单要求（少买、多买），怎么处理？

**小结**

这道题表面上是 **设计 Payment Verification API**，但实际上是 **小型系统设计**，覆盖：

* **API design**
* **DB & schema**
* **cache & scaling**
* **security**
* **transaction correctness**
* **fraud detection**

而 **follow-up 会逐步深入**：从 API → DB → 缓存 → 安全 → 容错 → 风控。

**Payment Verification Service 设计方案 (Simplified Bilingual Version)**

**1) API 设计 (API Design)**

**API Design (精简版)**

**核验交易 / Authorize Payment**

**URL:  
POST /v1/payments/authorize**

**Request (请求):**

**{**

**"shopper\_id": 123,**

**"amount": 300.00,**

**"currency": "USD",**

**"merchant": { "name": "Target", "address": "123 Main St" },**

**"psp\_txn\_id": "abc-123"**

**}**

**Response (成功 - 批准 / 200 OK - APPROVED):**

**{**

**"decision": "APPROVED",**

**"auth\_id": "A1B2C3",**

**"reason": "OK",**

**"order\_id": 987,**

**"remaining\_cap": 25.50,**

**"decided\_at": "2025-09-04T21:30:12Z"**

**}**

**Response (成功 - 拒绝 / 402 Payment Required - DECLINED):**

**{**

**"decision": "DECLINED",**

**"reason": "AMOUNT\_OUT\_OF\_RANGE",**

**"order\_id": 987,**

**"decided\_at": "2025-09-04T21:30:12Z"**

**}**

**Response (错误示例 / Error - 400 Bad Request):**

**{**

**"error": {**

**"code": "INVALID\_REQUEST",**

**"message": "amount field is missing"**

**}**

**}**

**2) 数据模型和存储 (Data Model & Storage)**

**表 1：orders（订单快照） / Table 1: orders (Order Snapshot)**

**CREATE TABLE orders (**

**id BIGSERIAL PRIMARY KEY,**

**shopper\_id BIGINT NOT NULL,**

**merchant\_id BIGINT NOT NULL,**

**expected\_total NUMERIC(12,2) NOT NULL, -- 期望总额 / expected total**

**tol\_abs NUMERIC(12,2) NOT NULL DEFAULT 0, -- 绝对误差 / absolute tolerance**

**tol\_pct NUMERIC(5,4) NOT NULL DEFAULT 0, -- 百分比误差(0.05=5%) / percent tolerance**

**authorized\_total NUMERIC(12,2) NOT NULL DEFAULT 0, -- 已批准累计 / spent so far**

**status TEXT NOT NULL DEFAULT 'IN\_PROGRESS', -- IN\_PROGRESS/COMPLETED/CANCELLED**

**updated\_at TIMESTAMP NOT NULL DEFAULT NOW(),**

**CONSTRAINT chk\_expected\_nonneg CHECK (expected\_total >= 0),**

**CONSTRAINT chk\_tolerances\_nonneg CHECK (tol\_abs >= 0 AND tol\_pct >= 0),**

**CONSTRAINT chk\_authorized\_nonneg CHECK (authorized\_total >= 0),**

**CONSTRAINT chk\_status\_valid CHECK (status IN ('IN\_PROGRESS','COMPLETED','CANCELLED'))**

**);**

**-- 一个 shopper 只有一张进行中订单（题目简化假设） / One active order per shopper (simplified assumption)**

**CREATE UNIQUE INDEX ux\_orders\_active\_shopper**

**ON orders (shopper\_id)**

**WHERE status = 'IN\_PROGRESS';**

**-- 按商家查询 / Merchant lookups**

**CREATE INDEX ix\_orders\_merchant ON orders (merchant\_id);**

**用途（ZH）：存放核验所需的订单信息：指定商家、期望总额、误差容忍（绝对/百分比）、已批准累计金额、订单状态。通过部分唯一索引保证“一个 shopper 只有一张进行中订单”。  
Purpose (EN): Holds verification essentials: assigned merchant, expected total, tolerances (absolute/percent), accumulated approved amount, and status. A partial unique index enforces “one active order per shopper.”**

**一句话总结（ZH）：orders 就是“预算 + 容忍 + 已用额度 + 状态”的订单快照。  
One-liner (EN): orders is the snapshot of “budget + tolerance + spent so far + status.”**

**表 2：transactions（刷卡尝试） / Table 2: transactions (Swipe Attempts)**

**CREATE TABLE transactions (**

**id BIGSERIAL PRIMARY KEY,**

**psp\_txn\_id TEXT NOT NULL, -- 唯一约束 / unique key (retries return same decision)**

**shopper\_id BIGINT NOT NULL,**

**order\_id BIGINT NOT NULL REFERENCES orders(id),**

**merchant\_id BIGINT NOT NULL,**

**amount NUMERIC(12,2) NOT NULL,**

**decision TEXT NOT NULL, -- APPROVED / DECLINED**

**reason TEXT NOT NULL, -- OK / OUT\_OF\_RANGE / WRONG\_MERCHANT / TIMEOUT / ...**

**created\_at TIMESTAMP NOT NULL DEFAULT NOW(),**

**CONSTRAINT ux\_transactions\_psp UNIQUE (psp\_txn\_id),**

**CONSTRAINT chk\_amount\_nonneg CHECK (amount >= 0),**

**CONSTRAINT chk\_decision\_valid CHECK (decision IN ('APPROVED','DECLINED'))**

**);**

**-- 常用查询索引 / Helpful lookups**

**CREATE INDEX ix\_tx\_order\_time ON transactions (order\_id, created\_at DESC);**

**CREATE INDEX ix\_tx\_shopper\_time ON transactions (shopper\_id, created\_at DESC);**

**用途（ZH）：记录每次 PSP 回调的“尝试与决策”，并用 psp\_txn\_id 的 unique key 保证重复请求不会被再次处理；reason 用可枚举值便于看板/告警统计。  
Purpose (EN): Stores each PSP attempt and the decision, with a unique key on psp\_txn\_id to avoid reprocessing duplicates; keep reason enumerable for dashboards/alerts.**

**一句话总结（ZH）：transactions 记录“每次刷卡 + 决策”，靠 psp\_txn\_id 的 unique key 防重复。  
One-liner (EN): transactions logs “every swipe + decision,” and the unique key on psp\_txn\_id prevents duplicates.**

**事务更新与一致性 / Transaction & Consistency**

* **ZH：审批时对该 shopper 的进行中订单行加锁，校验商家与金额容忍，若通过则原子更新 authorized\_total 并插入一条 transactions 记录；若不通过则仅插入 DECLINED 记录。这样可在并发下保证“至多批准一次”的正确性。**
* **EN: During approval, lock the active order row for the shopper, validate merchant and tolerance; if valid, atomically update authorized\_total and insert a row in transactions; otherwise insert a DECLINED row. This ensures correctness under concurrency (at-most-once approval).**

**与缓存的配合 / Interaction with Cache**

* **ZH：orders 的活动快照（商家、预算、容忍、已用额度）放入 Redis；订单变更时采用写穿策略同步更新 Redis，服务侧优先读缓存，未命中再读库。**
* **EN: Place the active orders snapshot (merchant, budget, tolerance, spent) in Redis; use write-through updates on order change; service reads cache first, DB on miss.**

**ZH（中文口语化）：  
我们就设计了两张表。第一张是 orders，它主要存订单的快照，比如哪个商家、预期花多少钱、允许多少误差，以及 shopper 到目前为止已经花了多少，还在不在进行中。第二张是 transactions，用来记录每一次刷卡尝试的金额和结果，而且我在 psp\_txn\_id 上加了唯一约束，这样同一笔交易如果重复过来，就不会被重复处理。整个流程就是：我锁定这个 shopper 的进行中订单，检查额度够不够，如果够就更新订单的已花金额并插入一条交易记录，不够就插一条拒绝记录。这样既保证数据一致性，也能避免重复扣款。**

**EN (English, conversational):  
I keep it really simple with just two tables. The first one is orders, which stores the active order snapshot—things like which merchant it’s for, the expected total, tolerance rules, how much has already been spent, and whether it’s still in progress. The second table is transactions, which logs every swipe attempt with the amount and the decision. I put a unique key on psp\_txn\_id, so if the same transaction comes in again, we won’t double process it. The flow is straightforward: lock the active order row for that shopper, check if there’s enough capacity, update the spent amount and insert a record if it’s valid, or insert a declined record if not. That way we get consistency and no double charging.**

**1) API & Transaction Handling**

**Q1. 如何设计 API（幂等、防重复、Unique key）？**  
**EN:** POST /v1/payments/authorize with psp\_txn\_id (or Unique-Key). I put a **unique key** on it in DB. Same txn twice → I return the same prior decision.  
**ZH：** POST /v1/payments/authorize，请求里带 psp\_txn\_id（或 Unique-Key），数据库上做**唯一约束**。同一交易重复到达 → 直接返回之前的决策。

**Q2. 同一个 shopper 刷两次卡，只扣一次？**  
**EN:** The **unique key** on psp\_txn\_id guarantees **at-most-once** approval. Duplicates hit the same row; I respond with the stored decision.  
**ZH：** 依靠 psp\_txn\_id 的**唯一约束**保证“**至多批准一次**”。重复请求命中同一条记录，我返回已存的结果。

**Q3. 同时收到两个不同金额的请求怎么决定？**  
**EN:** I **lock** the shopper’s active order row (SELECT ... FOR UPDATE), recompute remaining cap/tolerance, approve the first valid one, and the other is declined as out-of-cap or stale.  
**ZH：** 我对该 shopper 的活动订单**加行锁**，重算剩余额度/容忍区间；**先到且合法**的通过，另一个因为超额或状态过期被拒。

**Q4. 事务回滚 / 错误处理？**  
**EN:** Single ACID transaction: validate → decide → update order totals → insert transaction log → commit. On any error, rollback; return a structured decline code.  
**ZH：** 单事务 ACID：校验→决策→更新订单额度→写交易日志→提交。任何错误就回滚；用标准化原因码返回拒绝。

**2) 数据模型 & 存储**

**Q1. SQL 还是 NoSQL？为什么？**  
**EN:** **SQL (PostgreSQL)** for **strong consistency** and easy reconciliation/reporting for money.  
**ZH：** 选 **SQL（PostgreSQL）**，因为**强一致**，且资金相关对账/报表更方便。

**Q2. 表怎么设计？**（简化版）  
**EN:**

* orders(id, shopper\_id, merchant\_id, expected\_total, tolerance\_abs, tolerance\_pct, remaining\_cap, status, updated\_at)
* transactions(id, psp\_txn\_id UNIQUE, shopper\_id, order\_id, amount, decision, reason, created\_at)
* merchants(id, name, normalized\_addr, allowlisted)
* (optional) shopper\_assignments(shopper\_id PK, order\_id, merchant\_id, snapshot\_json, updated\_at)  
  **ZH：**（同上字段，不赘述）

**Q3. 多服务共享 DB，如何保证性能？**  
**EN:** Connection pooling, **read replicas** (for analytics), **Redis cache** for hot reads, and keep write paths short.  
**ZH：** 连接池、**只读副本**（做分析）、**Redis 缓存**热数据，写路径保持精简。

**Q4. 如何扩展数据库？**  
**EN:** Start vertical scale + read replicas; then **partition big tables** (e.g., transactions by time/range) and **shard by shopper\_id** if needed.  
**ZH：** 先纵向扩容 + 读副本；再对大表**分区**（按时间/范围），必要时按 shopper\_id **分片**。

**Q5. 索引怎么用？**  
**EN:**

* transactions(psp\_txn\_id UNIQUE)
* orders(shopper\_id) WHERE status IN ('ASSIGNED','IN\_PROGRESS')
* merchants(name, normalized\_addr)  
  **ZH：** 同上要点：交易唯一键、活动订单按 shopper\_id 条件索引、商家名/地址组合索引。

**3) 性能 & 可扩展性**

**Q1. 1 秒 SLA 怎么做到？**  
**ZH（中文）：  
保证 1 秒 SLA 的关键是缓存优先。先查进程内 L1 缓存，没命中就查 Redis 里的 L2 快照，数据库只在缺失或更新时访问。这样大部分请求都能在几毫秒内返回。我们的目标是 P95 延迟控制在 300ms 以内，并且设置全局 900ms 的硬超时，超时就保守拒绝。**

**EN (English):  
To hit the 1-second SLA, the key is cache-first. We check the in-process L1 cache (map or object in code) first, then fall back to Redis for the L2 snapshot, and hit the database only on cache miss or update. This keeps most requests in the low-millisecond range. Our target is P95 under 300ms, with a global hard cutoff at 900ms—if we exceed that, we fail closed to stay safe.**

**Q2. 怎么加缓存？存什么？写策略？**  
**EN:** Store **shopper’s active order snapshot** (merchant, expected\_total, tolerance, remaining\_cap). Use **write-through** when assignment changes—DB update and Redis update together.  
**ZH：** 缓存**活动订单快照**（商家、期望金额、容忍、剩余额度）。用**写穿（write-through）**：指派或修改订单时同时更新 DB 和 Redis。

**Q3. 负载均衡？**  
**EN:** Stateless pods behind LB (e.g., NLB/ALB); enable health checks and **graceful shutdown**.  
**ZH：** 无状态实例挂在负载均衡后，做健康检查和**优雅下线**。

**Q4. QPS/TPS 怎么提升？**  
**EN:** Scale out stateless service; keep decisions in cache; **async log to Kafka**; DB with partitioned transactions; tune connection pools.  
**ZH：** 横向扩容；决策走缓存；**日志异步进 Kafka**；交易表分区；调优连接池。

**4) 安全性**

**Q1. 确保来自可信 PSP？**  
**EN:** **mTLS** with PSP + **IP allowlist**, and optional **HMAC** signature over body.  
**ZH：** 和 PSP **双向 TLS** + **IP 白名单**，可加 **HMAC** 对请求体签名。

**Q2. 用什么认证（mTLS / JWT/OIDC）？**  
**EN:** External: **mTLS**. Internal: short-lived **JWT/OIDC** between services.  
**ZH：** 外部：**mTLS**；内部：短时 **JWT/OIDC**。

**Q3. 防数据泄露/篡改？**  
**EN:** No PAN storage—**tokenize**; **encryption in transit/at rest**; secrets in Vault/KMS; **audit logs** and least privilege.  
**ZH：** **不存卡号**、**代币化**；传输/静态**加密**；密钥进 Vault/KMS；**审计日志**与最小权限。

**5) 稳定性 & 容错**

**Q1. DB crash/断电如何保证数据安全？**  
**EN:** Rely on **WAL/binlog**, **sync replicas**, and ACID transactions; uncommitted work is rolled back safely.  
**ZH：** 有 **WAL/binlog**、**同步副本**和 ACID 事务；未提交的变更安全回滚。

**Q2. Failover？**  
**EN:** **Multi-AZ** primary with **automatic failover** to standby; apps use connection string with failover; retries are idempotent.  
**ZH：** **多可用区**主库 + **自动切换**；应用端使用支持切换的连接；重试因**唯一约束**而安全。

**Q3. 备份与恢复？**  
**EN:** Daily full + continuous archiving for **PITR**; regular **restore drills**.  
**ZH：** 每日全量 + 连续归档实现**时间点恢复**；定期做**演练**。

**6) 业务逻辑 & 风控**

**Q1. 金额不匹配：正常误差 vs 欺诈？**  
**EN:**

* **Normal**: produce weight, shelf price lag, taxes. Check **tolerance**: abs(amount - expected) <= max(tol\_abs, expected\*tol\_pct).
* **Fraud**: outside tolerance, wrong merchant, odd time/location, rapid swipes.  
  **ZH：**
* **正常**：称重、标价延迟、税费 → 看**容忍区间**：abs(实际-期望) <= max(绝对值, 百分比)。
* **可疑**：超容忍、非指定商家、异常时间/地点、频繁多刷。

**Q2. 历史数据如何做 risk evaluation？**  
**EN:** Log all attempts → Kafka → build features (per-shopper/per-merchant error distribution, time-of-day, velocity) → adjust tolerance or require manual review.  
**ZH：** 全量日志进 Kafka → 抽特征（用户/商家误差分布、时间段、频次）→ 调整阈值或触发人工复核。

**Q3. 在不合作商家刷卡？**  
**EN:** Decline by rule: merchant must match the **assigned merchant allowlist**.  
**ZH：** 规则拒绝：商家必须在**订单指定/白名单**内。

**Q4. 订单需要多次刷卡（分开买/分单）？**  
**EN:** Track remaining\_cap. Each swipe must be ≤ remaining cap and within tolerance; update atomically.  
**ZH：** 跟 remaining\_cap；每次刷卡 ≤ 剩余额度且在容忍内；原子更新。

**Q5. 购买数量与订单不一致（少买/多买）？**  
**EN:** Approve if total stays within tolerance. Later we **reconcile** item-level diffs asynchronously (price/quantity adjustments on the order).  
**ZH：** 只要总额在容忍内就批准；**事后对账**按商品维度异步调整（价格/数量修正）。

**Summary：**

So the way I’d design this payment verification service is to keep the hot path extremely fast and predictable, because we have a one-second SLA. When the payment processor calls our API, the request comes through a load balancer to a stateless verification service. The first thing we do is verify the signature and the Unique key, so if the processor retries we’ll always return the same decision.

Inside the service, we don’t want to hit the database on every request, so we maintain an active order snapshot for each shopper. That snapshot lives in a small in-process cache with a fallback to Redis, and Redis is written through from Postgres whenever the order is created or updated. The snapshot contains the allowed merchant, the expected budget, tolerance rules, and how much has already been spent.

When a swipe comes in, we just check that the merchant matches, the time window is valid, and the new total including this amount is still within the budget plus tolerance. We update the ‘spent so far’ in Redis atomically—either with a DB transaction or a Redis Lua script—and then persist the attempt in Postgres for audit and reconciliation. If anything fails or times out, we simply decline.

Security-wise, we run over mTLS, sign each request with HMAC, never store raw card data, and redact logs. And to scale, the service is stateless, so we can add pods behind the load balancer; Redis and Postgres are replicated and partitioned as needed. Finally, every decision is also published asynchronously to Kafka so our risk engine and reconciliation systems can process more complex logic offline.

This way we guarantee a sub-second, idempotent, and auditable decision process that can scale with traffic.

**Requirement Clarification:**

**1. 功能范围 (Scope)**

* **EN:** “Do we only need to verify merchant and amount for approval, or should we also integrate with external payment gateways/banks?”
* **ZH:** “我们只需要核对商家和金额来决定批准/拒绝吗？还是要负责和外部银行/支付网关的结算交互？”

**2. SLA & 流量 (SLA & Scale)**

* **EN:** “We have a 1-second SLA. Do we know the expected peak QPS/TPS?”
* **ZH:** “SLA 是 1 秒，那预期的峰值流量是多少？”

**3. 商家和订单关系 (Merchant & Order)**

* **EN:** “For simplicity, should I assume one shopper has at most one active order at one merchant?”
* **ZH:** “是不是可以简化假设为：一个 shopper 同时只会有一个订单，且只在一个商家购物？”

**4. 金额误差 (Amount tolerance)**

* **EN:** “Is it expected that the charged amount might differ from the expected order total, e.g. weight-based produce? If so, what tolerance rules should we enforce?”
* **ZH:** “是不是金额和订单金额可能会有偏差，比如称重的水果？如果有，需要我们在服务里定义容忍区间吗？”

**5. 重试 & 唯一约束 (Retries & Unique key)**

* **EN:** “Will the payment processor retry the same transaction if it doesn’t get a response? Should we treat their transaction ID as a unique key so retries return the same decision?”
* **ZH:** “支付处理器如果没收到响应，会不会重试？是不是可以把他们的交易 ID 当作唯一约束，用来保证重复请求返回相同结果？”

**6. 风控 (Risk / Fraud checks)**

* **EN:** “Should this service include basic fraud checks (wrong merchant, too many retries), or just pass/fail on merchant and amount?”
* **ZH:** “这个服务需要做一些简单的风控校验吗，比如非指定商家、多次刷卡过快，还是只判断商家和金额？”

**7. 数据持久化 (Persistence & Audit)**

* **EN:** “Do we need to persist every attempt for audit/reconciliation, or just the final approved transaction?”
* **ZH:** “我们需要把每一次刷卡尝试都存下来做审计/对账吗？还是只存最后成功的那一次就可以？”

**8. 安全 (Security)**

* **EN:** “How does the PSP call us? Should we enforce mTLS or signed requests?”
* **ZH:** “支付处理器调用我们接口时，需要强制双向 TLS 或签名验证吗？”