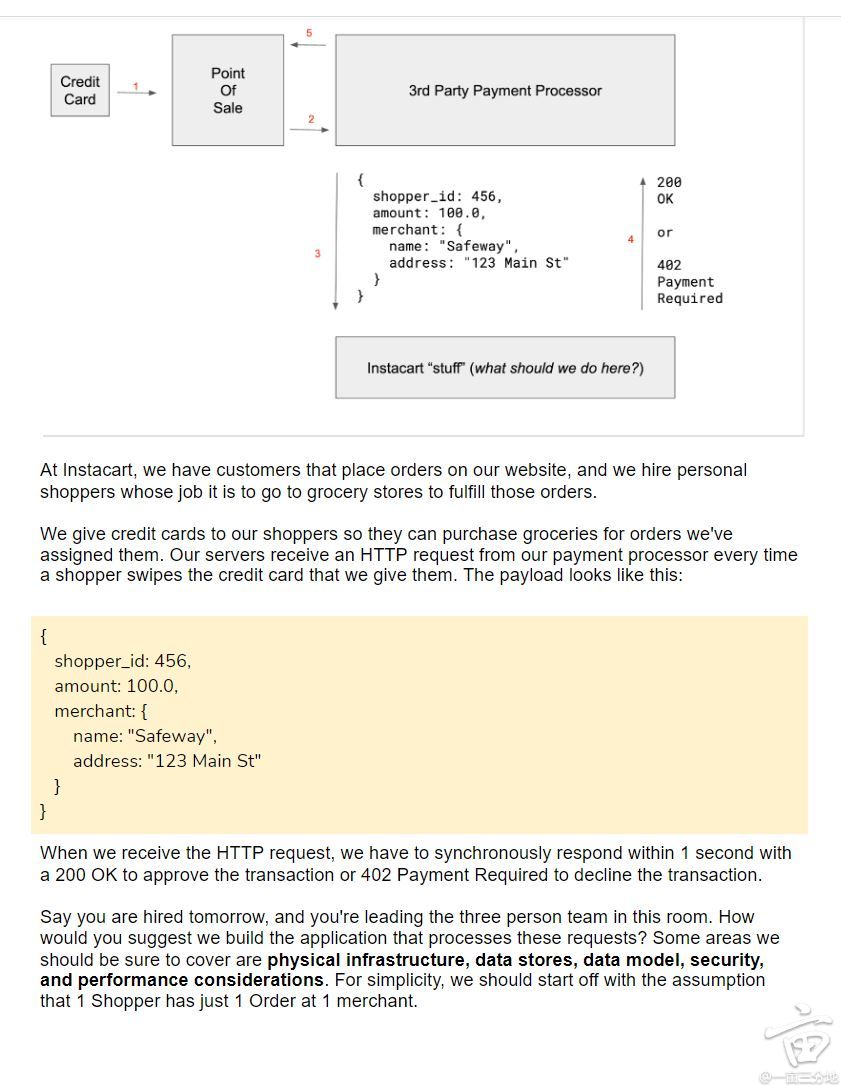
**Design Payment Service**



**题目理解 & 重点**

* **场景**：Instacart 给 shopper 一张信用卡。每次刷卡，**第三方支付处理器**会向我们的 **Payment Verification API** 发一个 HTTP 请求（shopper\_id, amount, merchant{ name,address } ...）。
* **SLA**：**必须在 1s 内**同步返回：200 OK（批准）或 402 Payment Required（拒绝）。
* **V1 简化假设**：1 个 shopper 同时只服务 1 个 order，且只在 1 家 merchant 购物。
* **考察重点**：API 设计、数据模型、数据存储、**安全**、**性能**、**幂等**、**一致性/事务**、**扩展性（DB 与服务）**、**稳定性/容错**、以及**误差/风险判断**。

**1) 题目重述（Problem Recap）**

* **输入 / Input**（来自支付处理器 PSP）：
* { "psp\_txn\_id": "...", "shopper\_id": 123, "amount": 300.00,
* "currency": "USD", "merchant": {"name":"Target","address":"123 Main St"},
* "timestamp":"..." }
* **SLA**：同步 **≤ 1s** 返回 200 APPROVED 或 402 DECLINED。
* **V1 假设**：一个 shopper 同时只有一个订单，且只在一个商户付款。

**2) 高层架构（High-Level Architecture）**

Processor → [ALB] → Payment Verification Service (stateless pods)

│

L1 in-proc cache (short TTL)

│ miss

Redis (L2)

│

PostgreSQL (primary + replicas)

│

Async Events (Kafka/SNS) → Risk/Recon/BI

* **无状态服务**水平扩展；**L1/L2 缓存**保证亚毫秒～毫秒级读取；DB 保证强一致与审计；**异步事件**用于风控学习、对账与报表。

**3) API 设计（对 PSP）**

**Endpoint**: POST /v1/payments/authorize  
**Headers**: Idempotency-Key（或使用 psp\_txn\_id），X-Signature (HMAC)，mTLS

**Request** 同上；**Responses**：

// 200

{ "decision":"APPROVED", "approved\_amount":300.00,

"order\_id":"ord\_123", "psp\_txn\_id":"..." }

// 402

{ "decision":"DECLINED", "reason\_code":"AMOUNT\_EXCEEDS\_LIMIT|MERCHANT\_NOT\_ALLOWED|DUPLICATE|TIME\_WINDOW|SYSTEM\_UNAVAILABLE" }

**Idempotency**：psp\_txn\_id 或 Idempotency-Key 在服务端**唯一**，重复调用直接返回首次决策。

**4) 数据模型（Data Model）**

**PostgreSQL（强一致 + 审计）**

* orders：order\_id(PK), shopper\_id, allowed\_merchants(标准化/指纹),  
  expected\_total, tolerance\_rules(jsonb), time\_window,  
  spent\_so\_far, partial\_allowed, status, timestamps
* payment\_attempts：psp\_txn\_id(UNIQUE), order\_id, shopper\_id,  
  amount, merchant\_fp, decision, reason, raw\_payload(jsonb), timestamps
* shoppers / merchants：基础档案、标准化信息

**Redis（L2 缓存，写通 / write-through）**

* Key: active\_order:{shopper\_id}
* Value:
* { "order\_id":"ord\_123", "allowed\_merchant\_fp\_set":["..."],
* "expected\_total":300.00, "tolerance":{"grocery":0.15,"default":0.05},
* "time\_window":{"start":"...","end":"..."},
* "spent\_so\_far":120.00, "partial\_allowed":true }

**5) 决策逻辑（Hot Path Decision Rules）**

**ZH**：

1. 验签 & 鉴权：mTLS + HMAC + IP allowlist。
2. 读取 active\_order:{shopper\_id}（L1→L2，miss 才回 DB）。
3. **幂等检查**：payment\_attempts.psp\_txn\_id 唯一；已存在则返回原决策。
4. 校验规则：
   * 时间窗合法；
   * 商户在 allowlist（名称+地址标准化/指纹，必要时地理围栏）；
   * 金额：spent\_so\_far + amount <= expected\_total \* (1 + tolerance)；生鲜/称重品可更高容忍；
   * 若允许分笔，累计不能超上限。
5. **原子更新**：
   * 方案 A（简单可靠）：Postgres 短事务：SELECT ... FOR UPDATE 锁订单 → 校验 → INSERT payment\_attempts（幂等唯一）→ UPDATE orders.spent\_so\_far → 提交。
   * 方案 B（更快）：Redis **Lua** 脚本原子加和与校验；随后异步落库（或事务消息）。
6. 返回决策，并异步发事件给风控/对账/BI。

**EN**: Same steps—auth, read snapshot, idempotency check, rule checks (time window, merchant allowlist, amount within tolerance, cumulative cap), atomic update (DB txn or Redis Lua), respond, publish events.

**6) 幂等与并发（Idempotency & Concurrency）**

* psp\_txn\_id（或 Idempotency-Key）**UNIQUE**，防止**重复扣款**。
* 并发多笔：对 orders 行加行级锁，或用 Redis Lua 保证**读判加**原子性。
* 同订单不同金额的重复请求：首个成功写入的决定为准；后续按幂等或“超过上限”拒绝。

**7) 缓存与数据库（Cache & DB）**

* **Why cache?** 1 秒 SLA 下，热路径避免多表查询；把“当前订单核验快照”放在缓存里。
* **写策略**：订单指派/变更 → **写通**到 Redis；授权成功 → **原子增加** spent\_so\_far。
* **回源**：缓存失效仅回源一次；失败则拒绝并告警。
* **DB 索引**：
  + payment\_attempts(psp\_txn\_id UNIQUE)
  + orders(shopper\_id, status) 覆盖索引；orders(order\_id) PK
  + 大表分区：payment\_attempts 按月分区并归档到对象存储

**8) 安全与合规（Security & Compliance）**

* **mTLS** + **HMAC-SHA256** 签名，防重放（时间窗 + nonce）。
* 只接收 **card token**，不存 PAN；敏感字段 **KMS 加密**；日志脱敏；审计不可篡改（WORM）。
* RBAC/ABAC、WAF、速率限制、IP allowlist。
* PCI DSS、GDPR/CCPA 数据保留与删除策略。

**9) 稳定性与扩展（Reliability & Scale）**

* **可用性**：多 AZ，无状态 pods + 自动扩缩；Redis 高可用/分片；Postgres 主从复制。
* **超时策略**：处理预算 ~300–600ms；超时**默认拒绝**（可配置）。
* **降级**：缓存 miss→快速回源；DB 慢→熔断拒绝；事件异步落本地队列重放。
* **扩展路径**：
  + 数据库：读写分离、按 shopper\_id/region 分片，或上 Aurora/Spanner。
  + 服务：按 QPS 横向扩容；把风控复杂计算下沉到异步流。

**10) 面试常见追问（Quick Answers）**

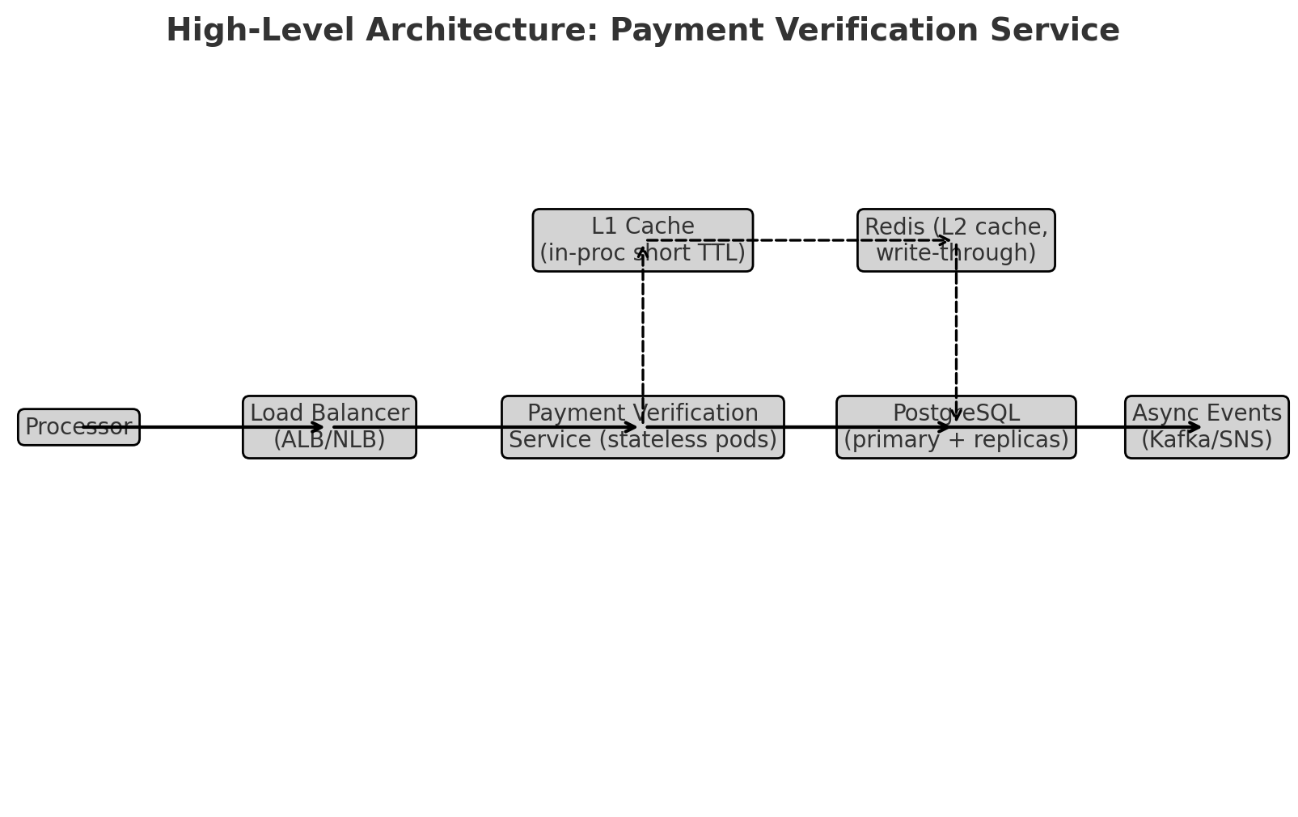
**Q1 商户/金额不一致怎么判？**

* 价格时滞 & 称重误差 → **按品类容忍度**（如生鲜 10–15%，默认 2–5%）；超过阈值拒绝并告警。  
  **Q2 非合作商户？**
* 名称+地址标准化做**指纹**匹配，辅以**地理围栏**；不匹配拒绝。  
  **Q3 分多次结账？**
* partial\_allowed=true，累计不可超上限；每笔都走原子校验。  
  **Q4 刷两次只扣一次？**
* psp\_txn\_id/Idempotency-Key 全局唯一 + 行级锁/原子脚本 → 幂等。  
  **Q5 为什么用写通缓存、只存“最新一单”？**
* 命中率高、逻辑简单、避免缓存穿透与复杂回收；核验只关心“当前订单”。  
  **Q6 如何 scale DB？**
* 分区 + 归档、读写分离、按用户或区域分片；热路径靠缓存，DB 走短事务与唯一索引。  
  **Q7 宕机/断电如何保证落库？**
* DB 事务；或 Redis 原子 + 事务消息/Outbox 模式保证“先决策、后持久化”一致性；失败重放。  
  **Q8 指标与监控？**
* P50/P95/P99 延迟、命中率、拒绝原因分布、幂等命中率、DB/Redis 错误率、事件积压、每商户风控命中。

**11) 你可以背的 1 分钟总结（Closing Pitch）**

**ZH**：  
我们把核验拆成“快路径 + 审计持久化”。快路径用缓存中的“当前订单快照”做商户与金额校验，并用 DB 行锁或 Redis Lua **原子**更新已花费，配合 psp\_txn\_id **幂等**保证“刷两次只扣一次”。数据在 Postgres 做强一致与审计，Redis 做 L2 缓存写通。全链路 mTLS/HMAC，超时默认拒绝。服务无状态水平扩容，DB 分区/分片可扩展，异步事件驱动风控学习与对账，能稳定满足 **1 秒 SLA**。

**EN**:  
We split verification into a fast path plus durable audit. The fast path reads an “active order snapshot” from cache, validates merchant and amount, and **atomically** updates spend (DB row lock or Redis Lua). With a **unique psp\_txn\_id**, retries are idempotent—“swipe twice, charge once.” PostgreSQL provides strong consistency and audit; Redis is a write-through L2 cache. End-to-end mTLS/HMAC, decline on timeout. Stateless services scale out; DB scales via partitioning/sharding; events feed risk and reconciliation. This reliably meets the **1s SLA**.



* **左边**：Processor（支付处理器） → 通过 **Load Balancer** 打到无状态的 Payment Verification Service。
* **上方虚线链路**：Service 先查 **L1 内存缓存**，miss 就去 **Redis L2 缓存**，再 miss 才回源 **PostgreSQL**。
* **右边**：验证决策后，结果写入 **PostgreSQL** 并异步发事件（Kafka/SNS）做风控、对账、BI。

**ZH 总结**：主线请求只走 Processor → Service → 缓存/DB，保证 ≤1s；副线（事件）异步处理。  
**EN Summary**: Hot path is Processor → Service → Cache/DB (≤1s SLA); side path is async events for risk, reconciliation, analytics.

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 idempotency 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.