SỐNG SÓT VÀ THĂNG TIẾN TRONG THỜI ĐẠI AI

Cẩm nang toàn diện về kỹ năng "khó bị thay thế" cho developer

---

> **Về tài liệu này**
> Đây là một cuốn sách hoàn chỉnh, tập trung vào **lý thuyết, nguyên lý hoạt động, và kiến thức nền tảng**.
> Mục tiêu: sau khi đọc xong, bạn sẽ hiểu sâu về các mảng kỹ năng khó bị AI thay thế và có lộ trình rõ ràng để phát triển.
> **Cách sử dụng:**
> - Đọc tuần tự từ đầu để xây nền tảng
> - Hoặc nhảy tới phần bạn cần củng cố
> - Mỗi chương có phần "Kiến thức cốt lõi" + "Ví dụ thực tế" + "Câu hỏi tự kiểm tra"

MỤC LỤC

PHẦN I: BỨC TRANH TOÀN CẢNH

Chương 1: AI đang thay đổi thế giới việc làm như thế nào?

1.1. Con số không nói dối

Theo báo cáo *Future of Jobs Report 2025* của Diễn đàn Kinh tế Thế giới (WEF), đến năm 2030:

- **92 triệu việc làm** hiện tại sẽ biến mất do tự động hóa
- **170 triệu việc làm mới** được tạo ra
- **Tăng ròng 78 triệu việc làm** - nhưng với **yêu cầu kỹ năng hoàn toàn khác**[cite:31][cite:34]

Điểm quan trọng không phải là "mất hay thêm bao nhiêu job", mà là **bản chất công việc thay đổi triệt để**. 22% việc làm toàn cầu sẽ trải qua "tái cấu trúc sâu" - nghĩa là cách làm việc, công cụ dùng, kỹ năng cần có đều thay đổi.[cite:31]

1.2. Developer: "AI-native workforce" đầu tiên

Một nghiên cứu của WEF năm 2026 gọi lập trình viên là "lực lượng lao động AI-native đầu tiên" - nhóm nghề đầu tiên sống chung với AI sâu nhất.[cite:25]

Khảo sát với hơn 1.600 developer ở 63 quốc gia cho thấy:

- **37%** nói AI **mở rộng** cơ hội nghề nghiệp của họ (không phải thu hẹp)
- **65%** tin vai trò của họ sẽ được "định nghĩa lại" trong 1-2 năm tới
- Hướng thay đổi: bớt code lặp lại → nhiều hơn thiết kế kiến trúc, tích hợp hệ thống, làm việc cross-functional[cite:25]

1.3. McKinsey: AI tăng năng suất developer, nhưng không đều

Nghiên cứu của McKinsey về GenAI cho developer cho thấy:[cite:20] [cite:38]

Với task đơn giản, lặp lại:

- Viết code CRUD mới: nhanh hơn gần **2x**
- Viết test cơ bản: nhanh hơn **1.5-1.8x**
- Refactor code theo pattern có sẵn: nhanh hơn **1.7x**
- Viết documentation: nhanh hơn **2.5x**

Với task phức tạp:

- Làm việc với framework lạ: lợi ích giảm xuống còn **1.1-1.2x**
- Thiết kế kiến trúc: gần như **không có lợi ích** (đôi khi còn chậm hơn vì phải verify output)
- Debug hệ thống phức tạp: **có thể chậm hơn** nếu dev thiếu kinh nghiệm

Developer junior (< 1 năm kinh nghiệm):

- Với task phức tạp, đôi khi **chậm hơn** khi dùng AI
- Lý do: không biết đánh giá output, không biết đặt câu hỏi đúng, thiếu nền tảng để verify[cite:20][cite:35]

1.4. Bài học rút ra

> **AI không thay thế developer.**
> **AI thay thế developer không có nền tảng, không hiểu hệ thống, chỉ biết gõ code theo ticket.**

Công thức thành công:
Developer có nền tảng vững + Dùng AI như công cụ = Năng suất gấp đôi
Developer thiếu nền tảng + Dùng AI = Có thể còn chậm hơn

1.5. Các ngành nghề bị ảnh hưởng mạnh

Theo WEF và các báo cáo khác:[cite:31][cite:34]

Nghề dễ bị thay thế:

- Nhập liệu văn phòng
- Xử lý hồ sơ đơn giản
- Customer support tuyến đầu (câu hỏi đơn giản, theo script)
- Một số loại kế toán viên (ghi chép, đối chiếu cơ bản)
- Junior content writer (viết theo template)

Nghề tăng trưởng mạnh:

- AI/ML engineer, data scientist
- Cloud engineer, DevOps/SRE
- Cybersecurity specialist

- Product manager hiểu kỹ thuật
- System architect, platform engineer[cite:31][cite:19]

1.6. Điều gì làm nên sự khác biệt?

Phân tích từ nhiều báo cáo cho thấy các công việc **khó bị thay thế** có đặc điểm:[cite:31][cite:25][cite:20]

1. **Cần xét đoán trong bối cảnh mơ hồ**
    - Ví dụ: quyết định kiến trúc cho hệ thống mới khi có nhiều constraint mâu thuẫn
2. **Cần hiểu sâu domain và con người**
    - Ví dụ: thiết kế UX cho nhóm người dùng đặc thù, hiểu pain point thật
3. **Cần chịu trách nhiệm cho quyết định**
    - Ví dụ: quyết định security model, chấp nhận trade-off giữa bảo mật và trải nghiệm
4. **Cần phối hợp nhiều người, nhiều team**
    - Ví dụ: dẫn dắt migration hệ thống lớn, điều phối giữa dev, ops, product, legal
5. **Cần sáng tạo và đề xuất giải pháp mới**
    - Ví dụ: tìm cách optimize cost 70% bằng cách thay đổi kiến trúc

Câu hỏi tự kiểm tra

1. Trong công việc hiện tại của bạn, bao nhiêu % thời gian dành cho:
    - Code lặp lại, theo pattern có sẵn? (dễ bị AI ăn)
    - Thiết kế, quyết định kiến trúc? (khó bị thay thế)
    - Làm việc với stakeholder, hiểu yêu cầu? (khó bị thay thế)
2. Nếu AI có thể viết 70% code bạn đang viết, bạn sẽ làm gì với 70% thời gian đó?
3. Liệu sau 3 năm, công việc hiện tại của bạn có thể được mô tả trong một prompt đủ rõ để AI làm được không?

Chương 2: Nguyên tắc "khó bị thay thế"

2.1. Mô hình 3 tầng: Task - Role - Career

Để hiểu rõ AI tác động thế nào, cần phân tách 3 tầng:

Tầng 1: Task (Tác vụ)

- Một việc cụ thể: viết hàm tính tổng, sửa bug CSS, viết test cho API
- AI tác động mạnh nhất ở tầng này
- **Chiến lược:** Giảm thời gian ở task lặp lại, tăng thời gian ở task phức tạp

Tầng 2: Role (Vai trò)

- Vai trò: Backend Developer, DevOps Engineer, Security Engineer, Product Engineer
- Vai trò thay đổi chậm hơn: ví dụ Backend Dev bớt viết CRUD, nhiều thời gian hơn cho integration, performance, security
- **Chiến lược:** Mở rộng vai trò sang các mảng liền kề (backend → backend + DevOps, backend + security)

Tầng 3: Career (Sự nghiệp)

- Hướng đi dài hạn: Tech Lead, Architect, Principal Engineer, Engineering Manager, Founder
- Thay đổi chậm nhất, phụ thuộc vào tích lũy kỹ năng và kinh nghiệm
- **Chiến lược:** Xây dựng T-shape: sâu một mảng + rộng nhiều mảng liền kề

2.2. Phân loại task: Dễ bị thay thế vs Khó bị thay thế

Task loại A: Dễ bị AI thay thế (hoặc đã bị)

Đặc điểm:

- Lặp lại, theo pattern rõ ràng
- Input/output rõ ràng, ít phụ thuộc context
- Ít yêu cầu xét đoán
- Có thể kiểm tra tự động (test pass/fail)

Ví dụ:

- Viết CRUD API theo schema có sẵn
- Convert design Figma thành HTML/CSS cơ bản
- Viết boilerplate code (model, migration, test fixtures)
- Generate documentation từ code comments
- Viết test unit cho hàm đơn giản

AI đã làm tốt những việc này. Nếu 80% công việc của bạn là loại này, **đây là tín hiệu cảnh báo đỏ**.

Task loại B: Khó bị thay thế trong 3-5 năm

Đặc điểm:

- Cần hiểu bối cảnh hệ thống hiện tại
- Cần trade-off giữa nhiều yếu tố mâu thuẫn
- Cần xét đoán dựa trên kinh nghiệm
- Cần làm việc với người khác để clarify requirement

Ví dụ:

- Refactor module cũ mà vẫn giữ backward compatibility với 10 service khác
- Optimize query phức tạp dựa trên schema và data distribution thực tế
- Thiết kế API mới phải phù hợp với 5 client khác nhau (web, mobile, partner)
- Debug performance issue liên quan đến nhiều layer (app, DB, network, cache)
- Quyết định có nên migrate từ SQL sang NoSQL cho một use case cụ thể

Task loại C: Rất khó bị thay thế (10+ năm)

Đặc điểm:

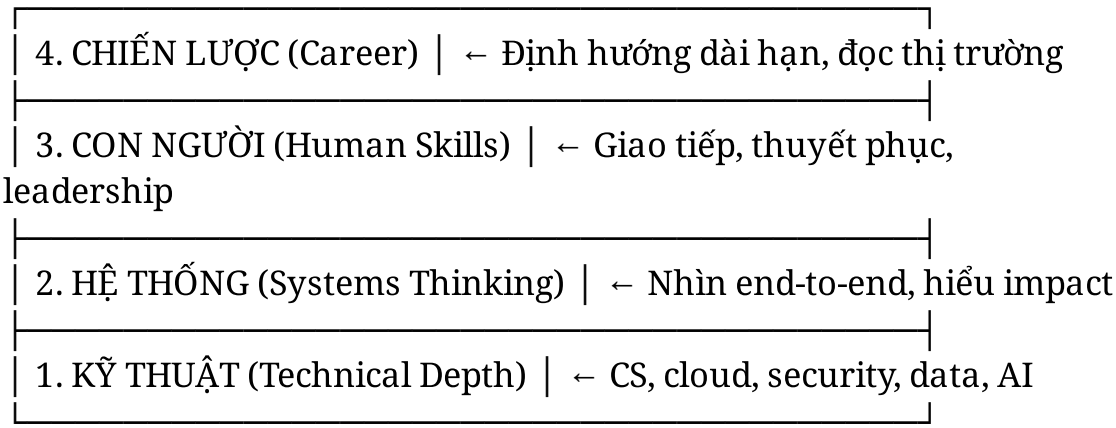
- Cần hiểu sâu domain business
- Cần chịu trách nhiệm cho quyết định có rủi ro cao
- Cần phối hợp nhiều bên liên quan (tech, product, legal, finance)
- Không có "đúng sai" rõ ràng, chỉ có "trade-off hợp lý"

Ví dụ:

- Thiết kế kiến trúc cho hệ thống mới: monolith hay microservices? Async hay sync?
- Quyết định security model: ưu tiên bảo mật hay UX? Where to draw the line?
- Xây dựng error budget và SLO/SLA cho hệ thống
- Thiết kế data model cho hệ thống sẽ scale lên 100x trong 2 năm
- Quyết định tech stack mới khi migrate từ hệ thống legacy

2.3. Mô hình 4 lớp kỹ năng

Kỹ năng "khó bị thay thế" có thể phân thành 4 lớp chồng lên nhau:



Lớp 1: Kỹ thuật (Technical Depth)

- Nền tảng CS: DSA, network, OS, DB
- System design: thiết kế hệ thống có khả năng mở rộng
- Cloud: AWS/GCP/Azure, Kubernetes, Terraform
- Security: OWASP, OAuth, IAM, DevSecOps
- Data & AI: pipeline, LLM integration, RAG

Lớp 2: Tư duy hệ thống (Systems Thinking)

- Nhìn được luồng end-to-end: user → frontend → backend → DB → infra → monitoring
- Hiểu được impact của thay đổi: "Nếu tôi thay đổi X, Y và Z sẽ bị ảnh hưởng như thế nào?"
- Dự đoán được bottleneck và single point of failure

- Biết đặt câu hỏi đúng: "Constraint thật sự là gì? Mục tiêu cuối cùng là gì?"

Lớp 3: Kỹ năng con người (Human Skills)

- Giao tiếp: giải thích kỹ thuật cho non-tech, viết tech spec rõ ràng
- Thương lượng và thuyết phục: convince team chọn giải pháp X thay vì Y
- Làm việc nhóm: code review xây dựng, pair programming hiệu quả
- Mentoring: giúp junior grow, tạo văn hóa học tập
- Leadership: dẫn dắt technical direction ngay cả khi không phải manager

Lớp 4: Tư duy chiến lược (Strategic Thinking)

- Career: chọn mảng chuyên sâu (T-shape: sâu một mảng, rộng nhiều mảng)
- Product sense: hiểu user, hiểu business, biết đề xuất solution tốt hơn PM đưa ra
- Market awareness: đọc trend, biết mảng nào đang hot, kỹ năng nào sẽ cần trong 3-5 năm
- Risk management: đánh giá rủi ro kỹ thuật, business, legal của mỗi quyết định

2.4. Nguyên lý "T-shaped developer"

```
  | Sâu 1 mảng (Specialist)
  |
  | Backend + Cloud + DevOps
  |       (ví dụ)
  |
```

———————┼———————————————————————
Rộng nhiều mảng (Generalist)
Frontend | Security | Data | AI | Product

Chiều dọc (Depth): Chuyên sâu 1-2 mảng

- Ví dụ: Backend + Cloud/DevOps

- Ví dụ: Frontend + Performance Optimization
- Ví dụ: Security + Cloud Security

Chiều ngang (Breadth): Biết đủ để làm việc với các mảng khác

- Biết đủ frontend để thiết kế API tốt
- Biết đủ security để không viết code dễ bị tấn công
- Biết đủ data để thiết kế schema hợp lý
- Biết đủ product để challenge requirement không hợp lý

Sai lầm phổ biến:

- ✕ "Jack of all trades, master of none" - biết rộng nhưng không sâu → dễ bị thay thế
- ✕ Chỉ sâu, không rộng → khó làm việc cross-functional, khó lên senior+

Lộ trình đúng:

1. Năm 1-2: Làm sâu một mảng (ví dụ: backend thuần)
2. Năm 2-4: Mở rộng sang mảng liền kề (backend + cloud/DevOps)
3. Năm 4+: Tiếp tục làm sâu + mở rộng thêm (+ security, + data/AI, + product sense)

2.5. Công thức "khó bị thay thế"

Giá trị của bạn =
(Kỹ thuật sâu) × (Tư duy hệ thống) × (Kỹ năng con người) × (Tư duy chiến lược)

Chú ý: là **nhân**, không phải **cộng**.

- Nếu một trong bốn = 0 → giá trị = 0
- Nếu cả bốn đều mạnh → hiệu ứng nhân lên (synergy)

Ví dụ thực tế:

Developer A:

- Kỹ thuật: 7/10 (giỏi backend, biết cloud)
- Hệ thống: 3/10 (chỉ nghĩ trong phạm vi service của mình)
- Con người: 5/10 (giao tiếp được nhưng không thuyết phục)
- Chiến lược: 2/10 (không biết mình muốn đi đâu)

- **Kết quả:** 7 × 3 × 5 × 2 = 210 → Giá trị trung bình

Developer B:

- Kỹ thuật: 7/10 (giỏi backend, biết cloud)
- Hệ thống: 7/10 (hiểu end-to-end, biết impact)
- Con người: 7/10 (giao tiếp tốt, thuyết phục được)
- Chiến lược: 6/10 (có career plan rõ ràng)
- **Kết quả:** 7 × 7 × 7 × 6 = 2058 → Giá trị **gấp 10 lần** Developer A

Câu hỏi tự kiểm tra

1. Trong 4 lớp kỹ năng, bạn đánh giá bản thân ở mức nào (1-10)?
    - Kỹ thuật: __/10
    - Tư duy hệ thống: __/10
    - Kỹ năng con người: __/10
    - Tư duy chiến lược: __/10
2. Lớp nào là điểm yếu nhất của bạn? (thường là "tư duy hệ thống" hoặc "kỹ năng con người")
3. Bạn có đang theo đuổi mô hình T-shaped không? Nếu có, "chiều sâu" của bạn là mảng nào?

PHẦN II: NỀN TẢNG KỸ THUẬT SÂU

Chương 3: Cấu trúc dữ liệu & thuật toán

3.1. Tại sao DSA vẫn quan trọng thời AI?

Nhiều người nghĩ: "AI viết code giỏi rồi, học DSA làm gì?"

Sự thật:

- AI viết code tốt khi bài toán được mô tả rõ ràng
- Nhưng **bạn** phải biết đặt câu hỏi đúng: "Nên dùng hash table hay binary search tree?"
- Nếu không hiểu DSA, bạn sẽ:
    - Không biết output của AI có đúng không
    - Không biết solution có optimal không

- Không fix được bug liên quan đến time/space complexity

Ví dụ thực tế:

Bạn hỏi AI: "Viết hàm tìm top 10 sản phẩm bán chạy nhất trong 1 triệu sản phẩm"

AI có thể trả về:

1. Load tất cả vào array → sort → lấy 10 đầu
   Time: O(n log n), Space: O(n)
2. Dùng min-heap size 10 → traverse qua tất cả
   Time: O(n log 10), Space: O(10)

Nếu bạn không hiểu DSA: bạn sẽ không biết solution 2 tốt hơn về memory và chỉ chậm hơn một chút.

3.2. Tư duy DSA: Không phải học thuộc, mà là hiểu trade-off

DSA không phải để "làm bài LeetCode", mà để **hiểu trade-off giữa time và space**.

Big O Notation: Ngôn ngữ chung

- O(1): Constant - không phụ thuộc kích thước input
- O(log n): Logarithmic - mỗi bước cắt bỏ một nửa (binary search, balanced tree)
- O(n): Linear - phải xem qua tất cả phần tử một lần
- O(n log n): Log-linear - best possible cho comparison sort
- $O(n^2)$: Quadratic - nested loop, tránh với n lớn
- $O(2^n)$: Exponential - tránh bằng mọi giá (trừ bài toán NP-hard)

Ví dụ thực tế: Tìm kiếm user

Scenario: Hệ thống có 10 triệu user, cần tìm user theo email

Solution 1: Array không sort

- Tìm kiếm: O(n) - worst case 10 triệu lần so sánh
- Với 1000 request/s → 10 tỷ comparison/s → CPU chết

Solution 2: Array đã sort + Binary search

- Tìm kiếm: O(log n) - worst case 24 lần so sánh ($\log_2 10{,}000{,}000 \approx 24$)
- Với 1000 request/s → 24,000 comparison/s → OK

Solution 3: Hash table

- Tìm kiếm: O(1) average - 1 lần so sánh
- Với 1000 request/s → 1,000 comparison/s → Tuyệt vời
- Trade-off: Tốn thêm memory cho hash table

3.3. Cấu trúc dữ liệu cốt lõi

3.3.1. Array & String

Đặc điểm:

- Access: O(1) theo index
- Insert/delete ở cuối: O(1)
- Insert/delete ở giữa: O(n) - phải shift các phần tử phía sau

Kỹ thuật quan trọng:

Two pointers:
Bài toán: Kiểm tra palindrome
Giải pháp: 1 pointer từ đầu, 1 pointer từ cuối, so sánh và di chuyển vào giữa
Time: O(n), Space: O(1)

Sliding window:
Bài toán: Tìm subarray dài nhất có tổng ≤ K
Giải pháp: Maintain một window, expand khi có thể, shrink khi tổng > K
Time: O(n), Space: O(1)

Ứng dụng thực tế:

- Rate limiting: sliding window để đếm request trong N giây
- Log analysis: tìm pattern trong stream log
- String matching: tìm substring hiệu quả

3.3.2. Hash Table (Map/Dictionary)

Nguyên lý hoạt động:

1. Hash function: key → hash code (integer)
2. Hash code % table_size → index trong array
3. Xử lý collision: chaining (linked list) hoặc open addressing

Tại sao O(1) average, không phải O(1) worst case?

- Average: hash tốt → phân bố đều → O(1)
- Worst: hash tệ → tất cả vào 1 bucket → O(n)
- Production: hash function tốt + load factor < 0.75 → gần như luôn O(1)

Ứng dụng thực tế:

- Cache: Redis, Memcached
- Database index (một phần)
- Session storage
- Deduplication: đếm unique items

Ví dụ: Hệ thống cache
Requirement: Cache user data, get theo user_id
Solution: Hash table với key = user_id

user_cache = {
"user_123": { name: "Alice", email: "..." },
"user_456": { name: "Bob", email: "..." }
}

get_user(user_id) → O(1)

3.3.3. Stack & Queue

Stack (LIFO - Last In First Out):

- Push/pop: O(1)
- Ứng dụng: function call stack, undo/redo, expression evaluation

Queue (FIFO - First In First Out):

- Enqueue/dequeue: O(1)

- Ứng dụng: message queue, BFS, task scheduling

Ví dụ thực tế: Background job processing
Job queue:

- User upload ảnh → enqueue job "resize_image"
- Worker consume từ queue → process → dequeue
- Guarantee: job được process theo thứ tự submit

Implement: Redis List (LPUSH + BRPOP)

3.3.4. Tree & Binary Search Tree

Binary Search Tree (BST):
Tính chất:

- Left subtree: tất cả node < root
- Right subtree: tất cả node > root

Tìm kiếm: O(log n) nếu balanced, O(n) nếu skewed
Insert/delete: O(log n) nếu balanced

Balanced BST: AVL, Red-Black Tree (dùng trong std::map, TreeMap)

Ứng dụng:

- Database index (B-tree là biến thể)
- Auto-complete (Trie là biến thể)
- Range query: tìm tất cả phần tử trong [A, B]

Ví dụ: Hệ thống order book (exchange)
Yêu cầu:

- Insert order theo giá: O(log n)
- Lấy best bid/ask: O(1)
- Match order: O(log n)

Solution: 2 balanced BST (hoặc 2 heap)

- Buy orders: max heap (hoặc BST sorted descending)
- Sell orders: min heap (hoặc BST sorted ascending)

3.3.5. Heap (Priority Queue)

Nguyên lý:
Min heap: parent ≤ children (root = minimum)
Max heap: parent ≥ children (root = maximum)

Insert: O(log n) - thêm vào cuối, bubble up
Extract min/max: O(log n) - lấy root, swap với cuối, bubble down
Peek min/max: O(1) - chỉ xem root

Ứng dụng thực tế:

1. Top K problem:
Bài toán: Tìm top 10 sản phẩm đắt nhất trong 1 triệu sản phẩm

Solution với min heap size 10:

- Traverse qua tất cả sản phẩm
- Nếu heap < 10 phần tử: insert
- Nếu heap = 10 phần tử và sản phẩm hiện tại > min: remove min, insert sản phẩm
- Cuối cùng: heap chứa top 10

Time: O(n log 10) = O(n)
Space: O(10) = O(1)

2. Merge K sorted lists:
Bài toán: Merge K sorted lists thành 1 sorted list

Solution:

- Min heap chứa phần tử đầu tiên của mỗi list
- Extract min → add vào result → add phần tử kế tiếp từ list đó vào heap
- Repeat

Time: O(N log K) với N = tổng số phần tử, K = số lists

3. Scheduling:
Task scheduler: priority queue theo deadline

- Task có deadline gần nhất được ưu tiên cao nhất

3.3.6. Graph

Biểu diễn:
Adjacency List: node → list of neighbors
{
"A": ["B", "C"],
"B": ["A", "D"],
"C": ["A", "D"],
"D": ["B", "C"]
}

Ưu: tiết kiệm memory với graph sparse
Nhược: check edge tồn tại = O(degree)

Adjacency Matrix: 2D array
[
[0, 1, 1, 0], // A → B, C
[1, 0, 0, 1], // B → A, D
[1, 0, 0, 1], // C → A, D
[0, 1, 1, 0] // D → B, C
]

Ưu: check edge = O(1)
Nhược: tốn memory $O(V^2)$

Traversal:

BFS (Breadth-First Search):

- Dùng queue
- Thăm theo "tầng": tất cả neighbor của A trước, rồi mới đến neighbor của neighbor
- Ứng dụng: shortest path (unweighted), level-order traversal

DFS (Depth-First Search):

- Dùng stack (hoặc recursion)
- Đi sâu vào một nhánh trước khi backtrack
- Ứng dụng: detect cycle, topological sort, connected components

Ví dụ thực tế: Social network
Bài toán: Tìm tất cả người trong vòng 2 kết nối (friend of friend)

Solution: BFS từ user A, depth = 2

- Level 0: A
- Level 1: friends của A
- Level 2: friends của friends của A

Time: O(V + E) với V = số user trong 2 hop, E = số connections

3.4. Thuật toán cốt lõi

3.4.1. Binary Search

Nguyên tắc:
Điều kiện: Array đã sort
Ý tưởng: Mỗi bước loại bỏ một nửa không thỏa mãn

Pseudocode:
left = 0, right = n - 1
while left <= right:
mid = (left + right) / 2
if arr[mid] == target: return mid
if arr[mid] < target: left = mid + 1
else: right = mid - 1
return not_found

Biến thể: Binary search trên answer
Bài toán: Tìm x nhỏ nhất thỏa mãn điều kiện P(x)

Ví dụ: Tìm số server tối thiểu cần để xử lý N request trong T giây

- P(x) = "x server có thể xử lý N request trong T giây"
- Binary search trên x từ 1 đến N

3.4.2. Sorting

Quick Sort:
Ý tưởng: Divide & conquer

- Chọn pivot
- Partition: đưa phần tử < pivot về trái, > pivot về phải
- Recursively sort left và right

Time: O(n log n) average, $O(n^2)$ worst (nếu pivot luôn là min/max)
Space: O(log n) stack

Merge Sort:
Ý tưởng: Divide & conquer

- Chia array thành 2 nửa
- Recursively sort 2 nửa
- Merge 2 sorted arrays

Time: O(n log n) guaranteed
Space: O(n) temporary array

Khi nào dùng cái nào?

- Quick sort: in-place, cache-friendly, trung bình nhanh hơn → default choice
- Merge sort: stable sort (giữ thứ tự tương đối), guaranteed O(n log n) → khi cần stable
- Heap sort: in-place, O(n log n) guaranteed, không stable → hiếm dùng
- Counting sort / Radix sort: O(n) khi range giá trị nhỏ → special cases

3.4.3. Dynamic Programming

Ý tưởng cốt lõi:
Biến bài toán lớn thành các bài toán con
Lưu kết quả bài toán con để tránh tính lại (memoization)

Ví dụ: Fibonacci
Naive recursion: $O(2^n)$
fib(n) = fib(n-1) + fib(n-2)
→ tính lại fib(n-2) rất nhiều lần

DP: O(n)
memo = {}
fib(n):
if n in memo: return memo[n]
if n <= 1: return n
memo[n] = fib(n-1) + fib(n-2)
return memo[n]

Ứng dụng thực tế:

- Longest common subsequence (diff algorithm)
- Edit distance (spell checker)
- Knapsack (resource allocation)
- Path finding với weighted graph (Dijkstra, A*)

3.5. Làm thế nào để học DSA hiệu quả?

Không nên:

- ✕ Học thuộc 500 bài LeetCode
- ✕ Làm random problems không có hệ thống
- ✕ Chỉ học lý thuyết không code

Nên:

- ✓ Học theo pattern: Two pointers, Sliding window, BFS/DFS, DP...
- ✓ Làm 5-10 bài mỗi pattern để hiểu sâu
- ✓ Áp dụng vào bài toán thực tế trong công việc
- ✓ Dùng AI để giải thích, nhưng **tự code lại**

Lộ trình gợi ý:
Week 1-2: Array & String (two pointers, sliding window)
Week 3-4: Hash Table & Stack/Queue
Week 5-6: Binary Search & Sorting
Week 7-8: Tree & BST
Week 9-10: Heap & Priority Queue
Week 11-12: Graph (BFS/DFS)
Week 13-14: DP basics

Câu hỏi tự kiểm tra

1. Cho một hệ thống cache với 1 triệu keys. Bạn chọn cấu trúc dữ liệu gì và tại sao?
2. Bạn cần tìm top 100 products bán chạy nhất từ 10 triệu products. Thuật toán gì? Time complexity?
3. Trong code review, bạn thấy đồng nghiệp dùng nested loop $O(n^2)$ cho một list 10,000 phần tử. Bạn có suggest optimize không? Optimize như thế nào?

Chương 4: Kiến trúc máy tính & Hệ điều hành

4.1. Tại sao dev cần hiểu computer architecture?

Nhiều người nghĩ:
"Mình viết code high-level (Python, JS, Java), hiểu assembly hay CPU làm gì?"

Sự thật:

- Hiểu CPU, memory, disk giúp bạn **optimize performance**
- Hiểu process, thread giúp bạn **thiết kế concurrency đúng**
- Hiểu I/O giúp bạn **chọn giải pháp phù hợp** (sync vs async, blocking vs non-blocking)

Ví dụ thực tế:
Bài toán: API response chậm

Developer không hiểu hardware:
"Code của mình đúng rồi, không biết sao chậm"

Developer hiểu hardware:

- Check: CPU spike → vòng lặp không tối ưu
- Check: Memory spike → memory leak
- Check: Disk I/O high → query DB nhiều, thiếu index
- Check: Network I/O high → gọi external API đồng bộ

4.2. CPU & Memory: Hierarchy

Memory Hierarchy (từ nhanh đến chậm):
CPU Register: ~1 cycle (< 1ns)
↓
L1 Cache: ~4 cycles (~1ns)
↓
L2 Cache: ~12 cycles (~3ns)
↓
L3 Cache: ~40 cycles (~10ns)
↓
RAM: ~200 cycles (~100ns)

↓
SSD: ~100,000 cycles (~100µs) ← 1000x chậm hơn RAM
↓
HDD: ~10,000,000 cycles (~10ms) ← 100,000x chậm hơn RAM

Bài học:

- Mỗi lần access HDD = 100,000 lần access RAM
- → Database query (disk I/O) tốn kém → cần cache
- → Đọc file lớn (100MB) từ disk = chậm → cần streaming, không load hết vào memory

Locality principle:
Temporal locality: data vừa dùng sẽ được dùng lại sớm
→ Cache recent data

Spatial locality: data gần nhau trong memory thường được dùng cùng lúc
→ Array access nhanh hơn linked list (cache line)

Ví dụ code:
Bad: traverse linked list 1 triệu node
→ mỗi node ở địa chỉ memory khác nhau
→ cache miss liên tục
→ chậm

Good: traverse array 1 triệu phần tử
→ data liên tiếp trong memory
→ CPU prefetch, cache hit cao
→ nhanh hơn nhiều

4.3. Process vs Thread

Process:

- Một instance của chương trình đang chạy
- Có memory space riêng (heap, stack, code, data)
- Process A không thể access memory của process B (isolation)
- Giao tiếp giữa processes: IPC (Inter-Process Communication) - pipe, socket, shared memory

Thread:

- Một "luồng thực thi" trong process
- Threads trong cùng process share memory (heap, code, data)
- Mỗi thread có stack riêng
- Giao tiếp: đơn giản hơn (shared memory) nhưng dễ race condition

Ví dụ: Web server xử lý request

Model 1: Multi-process (Apache MPM prefork)

- Mỗi request = 1 process riêng
- Ưu: isolation tốt, một request crash không ảnh hưởng others
- Nhược: tốn memory (mỗi process ~10MB), context switch chậm
- Phù hợp: ít request đồng thời, yêu cầu isolation cao

Model 2: Multi-thread (Apache MPM worker)

- Mỗi request = 1 thread trong thread pool
- Ưu: nhẹ hơn process, context switch nhanh hơn
- Nhược: thread crash có thể kill cả process
- Phù hợp: nhiều request đồng thời, ít blocking I/O

Model 3: Event-driven (Node.js, Nginx)

- Single thread + event loop
- Mỗi I/O operation = non-blocking, callback khi done
- Ưu: rất nhẹ, handle 10k+ connections với 1 thread
- Nhược: CPU-intensive task sẽ block event loop
- Phù hợp: I/O-intensive, ít CPU-intensive

4.4. Concurrency & Parallelism

Concurrency:
Nhiều task tiến triển "đồng thời" (interleaved)
Có thể chỉ có 1 CPU core, switch giữa các tasks

Parallelism:
Nhiều task chạy "cùng lúc thật sự" (simultaneously)
Cần nhiều CPU cores

Ví dụ:
1 CPU core, 2 tasks:

- Concurrency: Task A chạy 10ms → context switch → Task B chạy 10ms → switch back
- Illusion of parallelism, nhưng thực tế là serial

2 CPU cores, 2 tasks:

- True parallelism: Task A trên core 1, Task B trên core 2, chạy đúng cùng lúc

Implication cho developer:

CPU-bound task (tính toán nặng)

for i in range(1_000_000):
result = complex_calculation(i)

Multi-threading không giúp gì do GIL (Python)

→ Cần multi-processing để có true parallelism

I/O-bound task (đợi network, disk)

for url in urls:
data = fetch_url(url) # blocking I/O

Multi-threading hoặc async đều OK

Vì đa số thời gian là đợi I/O, không phải CPU

4.5. Synchronization: Lock, Mutex, Semaphore

Race condition:
2 threads cùng modify shared variable:

Thread 1: count = count + 1
Thread 2: count = count + 1

Nếu không sync:

- count ban đầu = 0
- Thread 1 read: 0, compute: 1 (chưa write)
- Thread 2 read: 0, compute: 1 (chưa write)
- Thread 1 write: 1
- Thread 2 write: 1
- count cuối = 1 (sai! phải là 2)

Mutex (Mutual Exclusion):
Lock bảo vệ critical section

mutex.lock()
count = count + 1 # only 1 thread at a time
mutex.unlock()

Hoặc trong code:
with lock:
count += 1

Deadlock:
Thread 1: lock A → (waiting lock B)
Thread 2: lock B → (waiting lock A)
→ cả 2 chờ mãi mãi

Giải pháp:

- Luôn lock theo thứ tự cố định (A trước, B sau)
- Timeout
- Avoid nested locks

Semaphore:
Cho phép N threads access cùng lúc

Ví dụ: connection pool size = 10

- Semaphore với count = 10
- Thread muốn connection: acquire (count--)
- Release connection: release (count++)
- Khi count = 0, thread phải đợi

4.6. I/O Models

Blocking I/O:
read(socket) ← thread bị block ở đây cho đến khi có data
→ thread không làm gì được khác

Ưu: đơn giản
Nhược: 1 thread chỉ handle 1 connection

Non-blocking I/O:
result = read(socket)
if result == WOULD_BLOCK:
# chưa có data, làm việc khác
else:
# có data, xử lý

Ưu: 1 thread check nhiều connections
Nhược: phải poll liên tục (tốn CPU)

Asynchronous I/O (Event-driven):
register_callback(socket, on_data_ready)
→ khi có data, OS gọi callback

Ưu: hiệu quả nhất, không poll, không block
Nhược: code phức tạp hơn (callback hell, async/await)

Ví dụ: HTTP request

Blocking:
response = requests.get(url) # block here
process(response)

Async:
async def fetch():
response = await aiohttp.get(url) # không block
process(response)

Trong async, khi `await`, thread có thể chạy task khác.

4.7. Virtual Memory & Paging

Vấn đề:
RAM = 8GB
Tổng memory processes cần = 20GB
→ Không đủ RAM

Giải pháp: Virtual memory

- Mỗi process nghĩ nó có 4GB RAM riêng (virtual address space)
- OS map virtual address → physical RAM hoặc disk (swap)
- Khi access virtual address không có trong RAM → page fault → load từ disk

Page & Page table:
Memory chia thành pages (thường 4KB)
Page table: virtual page → physical frame (hoặc disk)

Ví dụ:
Virtual address 0x1000 → Physical address 0x5000 (trong RAM)
Virtual address 0x2000 → Disk (swap)

Implication cho developer:

Ứng dụng dùng quá nhiều memory → OS swap ra disk → chậm
→ "Memory đầy" không phải crash ngay, mà là chậm lại đáng kể

Giải pháp:

- Monitor memory usage
- Set memory limit cho app
- Cache eviction strategy (LRU)

4.8. Context Switch

Context switch là gì?
CPU đang chạy Process A → switch sang Process B

Phải lưu state của A:

- CPU registers
- Program counter
- Stack pointer

Load state của B từ memory
Overhead: ~1-10µs

Nếu switch quá nhiều → tốn thời gian cho switching, ít thời gian cho actual work

Ví dụ thực tế:
Server với 1000 threads, 4 CPU cores
→ 1000/4 = 250 threads per core
→ context switch liên tục
→ throughput thấp

Better: thread pool size = số cores × 2
→ ít context switch hơn

Câu hỏi tự kiểm tra

1. API của bạn chậm. Bạn check thấy:
    - CPU: 10%
    - Memory: 30%
    - Disk I/O: 90%

   Nguyên nhân có thể là gì? Làm thế nào để fix?
2. Bạn design một service xử lý upload ảnh (I/O intensive). Chọn model nào?
    - Multi-process?
    - Multi-thread?
    - Event-driven single thread?

   Tại sao?
3. Code của bạn có race condition. 2 threads đều modify `counter`. Làm sao để fix?

Chương 5: Mạng máy tính & Giao thức web

5.1. Tại sao developer cần hiểu networking?

Nhiều dev nghĩ:
"Gọi API thôi mà, hiểu TCP/IP làm gì?"

Sự thật:

- Hiểu TCP/UDP giúp bạn **chọn protocol phù hợp** (REST vs WebSocket vs gRPC)
- Hiểu HTTP giúp bạn **thiết kế API tốt** và **debug performance issues**
- Hiểu TLS/HTTPS giúp bạn **implement security đúng**
- Hiểu DNS, load balancer giúp bạn **thiết kế hệ thống có khả năng mở rộng**

5.2. OSI Model & TCP/IP Model (tóm tắt)

OSI 7 layers (lý thuyết):
7. Application (HTTP, FTP, SMTP)
6. Presentation (encoding, encryption)
5. Session (connection management)
4. Transport (TCP, UDP)
3. Network (IP, routing)
2. Data Link (Ethernet, Wi-Fi)

1. Physical (cable, radio)

TCP/IP 4 layers (thực tế):
4. Application (HTTP, DNS, SSH)
3. Transport (TCP, UDP)
2. Internet (IP, ICMP)

1. Network Access (Ethernet, Wi-Fi)

Điều quan trọng: Mỗi layer giải quyết một vấn đề riêng, không phụ thuộc layer khác.

5.3. IP Address & Routing

IPv4:
32-bit address: 192.168.1.1
Format: 4 octets, mỗi octet 0-255
Tổng: $2^{32} \approx 4$ billion addresses (đã cạn kiệt)

IPv6:
128-bit address: 2001:0db8:85a3:0000:0000:8a2e:0370:7334
Tổng: 2^{128} addresses (nhiều vô kể)

Private vs Public IP:
Private IP (LAN):

- 192.168.x.x
- 10.x.x.x
- 172.16.x.x - 172.31.x.x
  Không route được trên Internet

Public IP:

- Assigned bởi ISP
- Route được trên Internet
- NAT (Network Address Translation): nhiều private IP share 1 public IP

Routing:
Packet từ A → B qua nhiều routers:
A → Router 1 → Router 2 → ... → Router N → B

Mỗi router check routing table:

- Destination IP → next hop

Traceroute: tool để xem path

5.4. TCP vs UDP

TCP (Transmission Control Protocol):
Đặc điểm:

- Connection-oriented: 3-way handshake trước khi gửi data
- Reliable: guarantee delivery, đúng thứ tự, không duplicate

- Flow control & Congestion control
- Overhead: header 20 bytes, handshake, ACK

Cơ chế hoạt động:

1. Handshake:
   Client → Server: SYN
   Server → Client: SYN-ACK
   Client → Server: ACK
2. Data transfer:
   Sender gửi packet với sequence number
   Receiver gửi ACK
   Nếu không nhận ACK → retransmit
3. Close:
   FIN → ACK → FIN → ACK

UDP (User Datagram Protocol):
Đặc điểm:

- Connectionless: không handshake
- Unreliable: không guarantee delivery, có thể mất packet, sai thứ tự
- No flow control
- Low overhead: header 8 bytes

Khi nào dùng:

- Real-time: video call, gaming (mất vài frame OK)
- DNS query (nếu fail, retry)
- Streaming (mất vài packets không ảnh hưởng nhiều)

Ví dụ:
Video call (Zoom):

- Dùng UDP
- Vì: nếu mất 1 frame video, không cần retransmit (frame đó đã out of date)
- Low latency > reliability

File download:

- Dùng TCP

- Vì: cần 100% data, đúng thứ tự
- Reliability > latency

5.5. HTTP/1.1, HTTP/2, HTTP/3

HTTP/1.1:
Đặc điểm:

- Text protocol (human-readable)
- 1 TCP connection = 1 request at a time (serial)
- Head-of-line blocking: request 1 chậm → request 2 phải đợi

Giải pháp: HTTP/1.1 pipelining (ít browser support) hoặc mở nhiều connections (6-8 connections/domain)

HTTP/2:
Đặc điểm:

- Binary protocol (compact, hiệu quả hơn)
- Multiplexing: nhiều requests/responses trên 1 TCP connection
- Stream priority
- Server push
- Header compression (HPACK)

Ưu điểm:

- Giải quyết head-of-line blocking ở application layer
- Giảm latency: ít TCP connections hơn → ít handshakes

Nhược điểm:

- Vẫn bị head-of-line blocking ở TCP layer (1 packet loss → block cả stream)

HTTP/3:
Đặc điểm:

- Dùng QUIC (UDP-based) thay vì TCP
- Multiplexing ở transport layer (stream độc lập)
- Giải quyết head-of-line blocking hoàn toàn
- Faster handshake: 0-RTT (Round-Trip Time)

Tình trạng: Đang được adopt rộng rãi (Google, Facebook, Cloudflare)

Ví dụ thực tế:
Website load 100 resources (HTML, CSS, JS, images)

HTTP/1.1:

- Mở 6 connections
- Mỗi connection serial → 100/6 ≈ 17 round trips/connection
- Nếu 1 request chậm → block cả connection đó

HTTP/2:

- 1 connection
- Multiplex 100 requests
- Parallel loading
- Nếu có packet loss → TCP head-of-line blocking

HTTP/3:

- 1 QUIC connection
- Multiplex 100 streams
- Stream độc lập → packet loss chỉ ảnh hưởng 1 stream

5.6. HTTPS & TLS

HTTP vs HTTPS:
HTTP:

- Plaintext
- Dễ bị nghe trộm (man-in-the-middle)
- Không verify server identity

HTTPS = HTTP + TLS/SSL:

- Encrypted
- Integrity (không bị sửa đổi giữa đường)
- Authentication (verify server = đúng server)

TLS Handshake (simplified):

1. Client → Server: ClientHello (supported ciphers, TLS version)
2. Server → Client: ServerHello (chosen cipher) + Certificate (public key)
3. Client verify certificate:

    - Check signature từ CA (Certificate Authority)
    - Check domain match
4. Client generate session key, encrypt bằng server public key → gửi cho server
5. Server decrypt bằng private key → có session key
6. Bắt đầu encrypted communication với session key (symmetric encryption)

Lưu ý: TLS 1.3 tối ưu thành 1-RTT hoặc 0-RTT

Certificate:
Certificate chứa:

- Domain name
- Public key
- Issuer (CA: Let's Encrypt, DigiCert, ...)
- Expiry date
- Signature của CA

Browser có list các CA tin cậy (root certificates)
→ verify chain: site cert ← intermediate CA ← root CA

Ví dụ thực tế:
Tại sao cần HTTPS cho mọi site (không chỉ e-commerce)?

1. Privacy: ISP không thấy được user đang xem gì
2. Integrity: không bị inject ads/malware giữa đường
3. SEO: Google boost HTTPS sites
4. Browser: Chrome/Firefox cảnh báo HTTP = "Not Secure"
5. Modern APIs: Service Worker, Geolocation, ... chỉ work trên HTTPS

5.7. DNS (Domain Name System)

Vai trò:
Chuyển domain name → IP address
example.com → 93.184.216.34

DNS Resolution flow:

1. User nhập: example.com
2. Browser check cache → không có

3. OS check cache → không có
4. Query DNS Resolver (ISP)
5. Resolver check cache → không có
6. Resolver query:
   Root DNS → .com TLD DNS → example.com authoritative DNS
7. Authoritative DNS trả về: 93.184.216.34
8. Resolver cache result (TTL) và trả về cho OS
9. OS cache và trả về cho browser
10. Browser cache và connect tới 93.184.216.34

TTL (Time To Live):
Record có TTL: 3600 seconds (1 hour)
→ Resolver cache 1 hour
→ Sau 1 hour, phải query lại

Implication:

- Thay đổi IP → có thể mất đến 24-48 hours để propagate hết (do cache ở nhiều layer)
- Trước khi migrate, giảm TTL xuống (ví dụ: 300 seconds)

DNS Record types:
A record: domain → IPv4
AAAA record: domain → IPv6
CNAME: alias → canonical name
www.example.com → example.com
MX: mail server
TXT: arbitrary text (SPF, DKIM, verification, ...)

5.8. Load Balancer

Vai trò:
Phân phối traffic tới nhiều servers
User → Load Balancer → Server 1 / Server 2 / Server 3

Algorithms:
Round Robin:

- Request 1 → Server 1
- Request 2 → Server 2
- Request 3 → Server 3

- Request 4 → Server 1
- ...

Least Connections:

- Gửi tới server đang có ít connections nhất

IP Hash:

- Hash IP của client → chọn server
- Same client → same server (session affinity)

Weighted Round Robin:

- Server 1 (powerful): weight 3
- Server 2 (weak): weight 1
- 3 requests → Server 1, 1 request → Server 2

Layer 4 vs Layer 7 load balancing:
Layer 4 (Transport layer):

- Balance based on IP + port
- Không nhìn vào HTTP headers
- Nhanh (low latency)
- Ví dụ: AWS NLB (Network Load Balancer)

Layer 7 (Application layer):

- Balance based on HTTP headers, URL path, cookies, ...
- Có thể route khác nhau: /api/* → backend servers, /static/* → CDN
- Chậm hơn nhưng linh hoạt hơn
- Ví dụ: AWS ALB (Application Load Balancer), Nginx, HAProxy

Health check:
Load balancer định kỳ ping servers:
GET /health → 200 OK → healthy
GET /health → timeout/500 → unhealthy → stop routing traffic

Graceful shutdown:

- Server trước khi shutdown: trả về 503 cho health check
- Load balancer stop gửi request mới

- Server đợi existing requests hoàn thành
- Shutdown

5.9. REST, WebSocket, gRPC

REST (Representational State Transfer):
Đặc điểm:

- HTTP-based
- Stateless
- Resources identified by URL
- CRUD = GET/POST/PUT/DELETE

Ưu:

- Đơn giản, phổ biến
- Cache-friendly (HTTP cache)
- Human-readable (JSON)

Nhược:

- Overhead cho real-time (polling hoặc long-polling)
- Không có schema enforcement (phải document carefully)

WebSocket:
Đặc điểm:

- Full-duplex: server và client đều có thể gửi message bất kỳ lúc nào
- Persistent connection
- Low latency
- Text hoặc binary

Use cases:

- Chat, notification real-time
- Collaborative editing (Google Docs)
- Gaming
- Live dashboard

Nhược:

- Khó scale (maintain connection state)

- Không có built-in authentication/authorization (phải tự implement)

gRPC:
Đặc điểm:

- RPC framework by Google
- Dùng Protocol Buffers (binary, compact)
- HTTP/2 multiplexing
- Strongly typed (schema trong .proto file)
- Support streaming (client stream, server stream, bidirectional)

Ưu:

- Hiệu quả (binary, compression)
- Schema enforcement
- Code generation (client + server từ .proto)
- Streaming built-in

Nhược:

- Không human-readable
- Cần tooling (protoc compiler)
- Ít browser support (cần proxy như Envoy)

Use cases:

- Microservices internal communication
- High-throughput data pipeline

Khi nào dùng cái nào?
REST:

- Public API
- CRUD simple
- Browser-based client

WebSocket:

- Real-time bidirectional
- Low latency quan trọng
- Push from server

gRPC:

- Microservices internal
- High performance
- Strongly typed schema

5.10. API Design Best Practices

RESTful API design:

1. URL = Noun, HTTP method = Verb
   ✅ GET /users
   ✅ POST /users
   ✅ GET /users/123
   ✅ PUT /users/123
   ✅ DELETE /users/123
   ❌ GET /getUsers
   ❌ POST /createUser
2. Use plurals
   ✅ /users
   ❌ /user
3. Nested resources
   ✅ GET /users/123/orders
   ❌ GET /orders?userId=123 (nếu order luôn thuộc về user)
4. Versioning
   /v1/users
   /v2/users
5. Pagination
   GET /users?page=2&limit=20
   hoặc cursor-based: GET /users?cursor=abc123&limit=20
6. Filtering, sorting
   GET /users?role=admin&sort=created_at&order=desc
7. Response format
   {
   "data": [...],
   "meta": {
   "total": 100,
   "page": 1,
   "limit": 20

```
}
}
```

8. Error response

```
{
"error": {
"code": "INVALID_INPUT",
"message": "Email is required",
"field": "email"
}
}
```

Status codes:
2xx: Success

- 200 OK: GET successful
- 201 Created: POST created new resource
- 204 No Content: DELETE successful

4xx: Client error

- 400 Bad Request: invalid input
- 401 Unauthorized: not authenticated
- 403 Forbidden: authenticated nhưng không có permission
- 404 Not Found: resource không tồn tại
- 429 Too Many Requests: rate limit

5xx: Server error

- 500 Internal Server Error: bug trong server
- 503 Service Unavailable: server overload hoặc maintenance

Câu hỏi tự kiểm tra

1. Bạn thiết kế một hệ thống chat real-time. Chọn protocol nào?
    - REST (polling)?
    - WebSocket?
    - gRPC?

    Tại sao?
2. API của bạn có endpoint GET /users trả về 10,000 users (50MB JSON). Làm thế nào để optimize?

3. Bạn deploy app mới, thay đổi IP từ 1.2.3.4 → 5.6.7.8. DNS TTL = 3600. Một số users vẫn connect tới IP cũ. Tại sao? Làm sao để giảm downtime khi migrate?

Chương 6: Cơ sở dữ liệu

6.1. Tại sao hiểu database quan trọng?

Nhiều dev nghĩ:
"Dùng ORM là đủ, cần gì hiểu SQL hay index?"

Sự thật:

- ORM sinh ra query không tối ưu → performance issue
- Không hiểu index → query chậm
- Không hiểu transaction → data inconsistency
- Không hiểu isolation level → race condition, lost update

Ví dụ thực tế:
Query: "Lấy 10 orders mới nhất của user_id=123"

ORM naive:
SELECT * FROM orders WHERE user_id = 123 ORDER BY created_at DESC LIMIT 10

Nếu không có index trên (user_id, created_at) → full table scan

→ Full table scan 1 triệu rows → chậm

Với index:
CREATE INDEX idx_user_orders ON orders(user_id, created_at DESC);
→ Index seek → chỉ scan 10 rows → nhanh

6.2. Relational Database: SQL & ACID

Relational model:
Data được tổ chức thành tables (relations)

- Rows = records/tuples
- Columns = attributes
- Schema = structure + constraints

Ví dụ:
Table: users

id	name	email	created_at
1	Alice	alice@example.com	2024-01-01
2	Bob	bob@example.com	2024-01-02

Relationship:
One-to-many: 1 user có nhiều orders
Many-to-many: nhiều students có nhiều courses (qua junction table)

ACID Properties:

Atomicity (Nguyên tử):
Transaction = tất cả hoặc không có gì
Không có "một nửa transaction"

Ví dụ: Chuyển tiền A → B
BEGIN TRANSACTION;
UPDATE accounts SET balance = balance - 100 WHERE id = A;
UPDATE accounts SET balance = balance + 100 WHERE id = B;
COMMIT;

Nếu statement 2 fail → rollback statement 1 → balance không thay đổi

Consistency (Nhất quán):
Database luôn ở trạng thái hợp lệ (satisfy constraints)

Ví dụ: constraint balance >= 0
→ Transaction làm balance < 0 sẽ bị reject

Isolation (Độc lập):
Concurrent transactions không ảnh hưởng lẫn nhau
"Như thể" chúng chạy tuần tự

Ví dụ:
Transaction 1: đọc balance của A
Transaction 2: update balance của A

Transaction 1: đọc balance của A lần nữa
→ Kết quả phụ thuộc vào isolation level

Durability (Bền vững):
Khi COMMIT thành công, dữ liệu được lưu vĩnh viễn
Dù server crash ngay sau đó, data không mất

Cơ chế: Write-Ahead Logging (WAL)

- Mọi thay đổi ghi vào log trước
- Commit = flush log to disk
- Crash recovery = replay log

6.3. Index: Làm thế nào database tìm kiếm nhanh?

Vấn đề:
Table có 10 triệu rows
Query: SELECT * FROM users WHERE email = 'alice@example.com'

Không có index:
→ Scan 10 triệu rows (full table scan)
→ O(n) time

Có index:
→ Index seek
→ O(log n) time với B-tree index

B-Tree Index:

Cấu trúc:

- Tree cân bằng
- Mỗi node chứa nhiều keys (không phải binary)
- Leaf nodes chứa pointers tới actual rows
- Internal nodes dùng để navigate

Ví dụ B-tree với keys = emails:

```
        [m]
    /           \
```

```
   [d, h]              [r, v]
  /   |   \           /  |  \
```

[a,b] [e,f] [i,k] [n,o] [s,t] [w,z]

Tìm 'alice@example.com':

- Start từ root: 'a' < 'm' → đi trái
- 'a' < 'd' → đi trái
- Tìm trong leaf [a, b]
- Found → return pointer → fetch row

Depth = O(log n) → với 10M rows, depth ≈ 4-5 levels
→ 4-5 disk reads thay vì 10M

Composite Index:
Index trên nhiều columns

CREATE INDEX idx_user_location ON users(country, city, created_at);

Thứ tự quan trọng:
Query: WHERE country = 'US' AND city = 'NYC'
→ Dùng được index

Query: WHERE city = 'NYC'
→ KHÔNG dùng được index (vì country là prefix)

Rule: Index sử dụng được từ trái sang phải

Covering Index:
Index chứa tất cả columns cần thiết

Query: SELECT name, email FROM users WHERE email = 'alice@example.com'

Index: (email)
→ Tìm row qua index → fetch name từ table

Covering index: (email, name)
→ Tìm row qua index → lấy luôn name từ index (không cần fetch table)
→ Nhanh hơn

Trade-off của Index:
Ưu:

- Query nhanh hơn nhiều (O(log n) vs O(n))

Nhược:

- Insert/update/delete chậm hơn (phải update index)
- Tốn disk space
- Quá nhiều index → query planner khó chọn index tốt nhất

Rule of thumb:

- Index trên columns hay dùng trong WHERE, JOIN, ORDER BY
- Đừng index mọi thứ

6.4. Transaction Isolation Levels

Vấn đề:
2 transactions đồng thời access cùng data → có thể có anomalies

Anomalies:

1. Dirty Read:
T1 update nhưng chưa commit
T2 đọc data đã update
T1 rollback
→ T2 đọc data "dơ" (không tồn tại)

2. Non-Repeatable Read:
T1 đọc row
T2 update row và commit
T1 đọc row lần nữa → khác kết quả lần 1
→ "Không lặp lại được"

3. Phantom Read:
T1 đọc tất cả rows thỏa mãn điều kiện (ví dụ: age > 20)
T2 insert row mới thỏa mãn điều kiện và commit
T1 đọc lại → thấy row mới (phantom)

Isolation Levels:

Read Uncommitted (thấp nhất):

- Cho phép dirty read
- Hiếm dùng

Read Committed (mặc định của nhiều DB):

- Không cho phép dirty read
- Cho phép non-repeatable read và phantom read

Cơ chế: Mỗi statement đọc snapshot tại thời điểm nó chạy

Repeatable Read:

- Không cho phép dirty read và non-repeatable read
- Vẫn có thể có phantom read (MySQL InnoDB thực ra không có phantom do MVCC)

Cơ chế: Transaction đọc snapshot tại thời điểm nó bắt đầu

Serializable (cao nhất):

- Không có anomaly nào
- Transactions như thể chạy tuần tự

Cơ chế: Lock ranges, có thể dùng predicate locks

Trade-off:

- Isolation cao hơn = concurrency thấp hơn = throughput giảm
- Chọn level phù hợp với yêu cầu business

Ví dụ thực tế:

Use case 1: Bank transfer
Yêu cầu: Không được mất tiền, không được double-spend
→ Cần SERIALIZABLE hoặc REPEATABLE READ với explicit locking

BEGIN TRANSACTION;
SELECT balance FROM accounts WHERE id = A FOR UPDATE; -- lock row
-- check balance >= amount
UPDATE accounts SET balance = balance - amount WHERE id = A;
UPDATE accounts SET balance = balance + amount WHERE id = B;
COMMIT;

Use case 2: View count
Yêu cầu: Tăng view count mỗi lần user xem post
→ Dirty read OK, non-repeatable read OK
→ Dùng READ COMMITTED hoặc thậm chí READ UNCOMMITTED

UPDATE posts SET view_count = view_count + 1 WHERE id = 123;

6.5. Query Optimization

Explain Plan:
Mọi SQL DB có EXPLAIN để show execution plan

EXPLAIN SELECT * FROM orders WHERE user_id = 123 ORDER BY created_at DESC LIMIT 10;

Output:

- Seq Scan on orders (cost=0..1000 rows=10000) ✖ Chậm
  hoặc
- Index Scan using idx_user_orders (cost=0..50 rows=10) ✓ Nhanh

Metrics quan trọng:

- Scan type: Seq Scan (full scan) vs Index Scan
- Rows: số rows ước tính phải xem
- Cost: ước tính chi phí

Common issues:

1. Missing index:
SELECT * FROM orders WHERE user_id = 123;
→ Full table scan

Fix: CREATE INDEX idx_user_id ON orders(user_id);

2. Function trên indexed column:
SELECT * FROM users WHERE LOWER(email) = 'alice@example.com';
→ Không dùng được index trên email

Fix: Lưu email lowercase, hoặc tạo functional index

3. OR trong WHERE:
SELECT * FROM products WHERE category = 'electronics' OR price <

100;
→ Khó optimize

Fix: Tách thành 2 queries và UNION (nếu có index riêng)

4. SELECT * thay vì SELECT columns cần:
SELECT * FROM users WHERE id = 123;
→ Fetch tất cả columns (có thể nhiều data)

Fix: SELECT id, name, email FROM users WHERE id = 123;
→ Có thể dùng covering index

5. N+1 query problem:
Code:
users = query("SELECT * FROM users LIMIT 10")
for user in users:
orders = query("SELECT * FROM orders WHERE user_id = ?", user.id)

→ 1 query users + 10 queries orders = 11 queries

Fix: JOIN hoặc IN clause
query("SELECT * FROM orders WHERE user_id IN (1,2,3,...,10)")
→ 2 queries total

6.6. Normalization & Denormalization

Normalization:
Tách data thành nhiều tables để giảm redundancy

Ví dụ:
Unnormalized:

order_id	user_name	user_email	product_name	price
1	Alice	alice@...	Laptop	1000
2	Alice	alice@...	Mouse	20

Problem: user_name và user_email lặp lại (redundancy)

Normalized:
users:

| id | name | email |

orders:
| id | user_id | product_id |

products:
| id | name | price |

Ưu:

- Không redundancy → update 1 chỗ (ví dụ: email thay đổi)
- Consistency: không có "Alice" ở order 1 và "Alice Smith" ở order 2

Nhược:

- Query phức tạp hơn (cần JOIN)
- Performance: JOIN tốn kém

Denormalization:
Cố ý thêm redundancy để tăng performance

Ví dụ: Cache user_name trong orders table
| order_id | user_id | user_name | product_id |

Trade-off:

- Query nhanh (không cần JOIN)
- Update phức tạp (phải update nhiều chỗ)

Khi nào denormalize?

- Read-heavy workload (1000 reads : 1 write)
- Data ít thay đổi (ví dụ: user_name ít đổi)
- Performance quan trọng hơn consistency

6.7. SQL vs NoSQL

SQL (Relational):
Ví dụ: PostgreSQL, MySQL, SQL Server

Đặc điểm:

- Schema cố định (table, columns, types)
- ACID transactions

- JOIN, foreign keys
- Vertical scaling (scale up)

Use cases:

- Data có structure rõ ràng, ít thay đổi schema
- Cần transactions (banking, e-commerce)
- Complex queries, reporting, analytics

NoSQL:
Nhiều loại:

1. Document DB (MongoDB, Couchbase):
Data = JSON documents

```
{
"_id": "123",
"name": "Alice",
"orders": [
{ "product": "Laptop", "price": 1000 },
{ "product": "Mouse", "price": 20 }
]
}
```

Ưu:

- Schema flexible (mỗi document có thể khác nhau)
- Denormalized → ít JOIN
- Horizontal scaling

Nhược:

- Không có transactions multi-document (trước MongoDB 4.0)
- Phức tạp khi cần JOIN hoặc aggregation

Use cases:

- Content management, user profiles
- Rapid prototyping (schema thay đổi nhanh)
- Catalog, product data

2. Key-Value Store (Redis, DynamoDB):
Data = key → value (string, list, hash, set)

Ưu:

- Cực nhanh (in-memory hoặc SSD)
- Simple API (GET/SET/DELETE)
- Scale tốt

Nhược:

- Không có complex queries
- Không có transactions multi-key (trừ Redis Cluster có giới hạn)

Use cases:

- Cache
- Session storage
- Rate limiting, counters
- Real-time leaderboard

3. Column-family (Cassandra, HBase):
Data tổ chức theo columns, không phải rows

Ưu:

- Rất scale (petabyte-scale)
- Write throughput cao
- Time-series data

Use cases:

- Time-series (metrics, logs)
- Event sourcing
- IoT data

4. Graph DB (Neo4j, Amazon Neptune):
Data = nodes + relationships

Ưu:

- Query relationship hiệu quả (traversal)
- Natural representation cho social network, knowledge graph

Use cases:

- Social networks (friend of friend)
- Recommendation engine
- Fraud detection (network analysis)

Chọn SQL hay NoSQL?

Dùng SQL khi:

- Cần ACID transactions
- Schema ổn định, quan hệ phức tạp
- Cần complex queries (JOIN, aggregation, window functions)

Dùng NoSQL khi:

- Cần scale horizontal massive (TB-PB data)
- Schema flexible, thay đổi thường xuyên
- Simple queries (key-value lookup, aggregation đơn giản)
- Write-heavy workload

Thực tế:
Nhiều hệ thống dùng cả hai (polyglot persistence)

- SQL cho transactional data (orders, payments)
- NoSQL cho session cache (Redis)
- NoSQL cho analytics (Cassandra, ClickHouse)

6.8. Scaling Database

Vertical Scaling (Scale Up):
Tăng CPU, RAM, disk của DB server

Ưu:

- Đơn giản (không thay đổi code)
- Không có phức tạp distributed

Nhược:

- Có giới hạn (max 96 cores, 1TB RAM, ...)
- Expensive
- Single point of failure

Horizontal Scaling (Scale Out):

1. Read Replicas:
1 primary (write) + nhiều replicas (read)

Primary → Replica 1
→ Replica 2
→ Replica 3

Write: vào primary
Read: từ replicas (load balanced)

Ưu:

- Scale read capacity
- High availability (nếu primary down, promote replica)

Nhược:

- Replication lag (replica có thể chậm hơn primary vài ms-giây)
- Write vẫn bottleneck tại primary

Use case:

- Read-heavy workload (90% read, 10% write)

2. Sharding (Partitioning):
Chia data thành nhiều shards (partitions)

Ví dụ: 10M users

- Shard 1: user_id 0-999,999
- Shard 2: user_id 1M-1,999,999
- ...
- Shard 10: user_id 9M-9,999,999

Sharding strategy:

- Range-based: user_id ranges (dễ hotspot)
- Hash-based: hash(user_id) % num_shards (đều hơn)
- Directory-based: lookup table

Ưu:

- Scale write (mỗi shard nhận subset của writes)

- Scale storage (mỗi shard chỉ lưu subset của data)

Nhược:

- Phức tạp:
  - Cross-shard queries (JOIN giữa shards) rất khó
  - Rebalancing khi thêm/bớt shards
  - Transaction cross-shard phức tạp

Use case:

- Write-heavy workload
- Data quá lớn cho 1 server

Câu hỏi tự kiểm tra

1. API của bạn có query chậm. Bạn chạy EXPLAIN và thấy "Seq Scan on users (cost=0..10000)". Điều này có nghĩa gì? Làm thế nào để fix?
2. Hệ thống e-commerce cần:
   - Chuyển tiền giữa 2 accounts (ACID critical)
   - Lưu session của 10M users (fast access)
   - Store product catalog (flexible schema)

   Bạn chọn database gì cho từng use case?
3. Hệ thống của bạn có 90% read, 10% write. Users phàn nàn DB slow. Strategy nào để scale?

PHẦN III: THIẾT KẾ HỆ THỐNG & KIẾN TRÚC

Chương 7: Nguyên tắc thiết kế phần mềm

7.1. Tại sao design principles quan trọng?

Reality check:

- Code mới viết = dễ hiểu
- Code sau 6 tháng = khó hiểu
- Code sau 2 năm + 5 developers = nightmare

Mục tiêu của good design:

- Dễ đọc, dễ hiểu
- Dễ thay đổi (change is constant)
- Dễ test
- Tái sử dụng được

7.2. SOLID Principles

S - Single Responsibility Principle (SRP):
Một class chỉ nên có một lý do để thay đổi

✕ **Bad:** God class làm mọi thứ
class UserService:
def create_user(self, data):
# validate
# save to DB
# send email
# log
# update cache

✓ **Good:** Mỗi class một trách nhiệm
class UserRepository:
def save(self, user): ...

class EmailService:
def send_welcome_email(self, user): ...

class UserService:
def **init**(self, repo, email_service):
self.repo = repo
self.email_service = email_service

```
def create_user(self, data):
    user = User(data)
    self.repo.save(user)
    self.email_service.send_welcome_email(user)
```

O - Open/Closed Principle:
Open for extension, closed for modification

✕ **Bad:** Phải sửa code khi thêm payment method mới
class PaymentService:
def process(self, method, amount):
if method == 'credit_card':
# credit card logic
elif method == 'paypal':
# paypal logic
# Thêm method mới → phải sửa function này

✓ **Good:** Extend bằng cách thêm class mới
class PaymentProcessor(ABC):
@abstractmethod
def process(self, amount): ...

class CreditCardProcessor(PaymentProcessor):
def process(self, amount): ...

class PayPalProcessor(PaymentProcessor):
def process(self, amount): ...

Thêm method mới → chỉ cần thêm class mới

L - Liskov Substitution Principle:
Subclass phải có thể thay thế base class mà không làm hỏng logic

✕ **Bad:** Override method nhưng thay đổi behavior không tương thích
class Bird:
def fly(self): ...

class Penguin(Bird):
def fly(self):
raise Exception("Penguins can't fly!") # Breaks LSP

✓ **Good:** Redesign hierarchy
class Bird: ...
class FlyingBird(Bird):
def fly(self): ...

class Penguin(Bird):
def swim(self): ...

I - Interface Segregation Principle:
Client không nên phụ thuộc vào interface mà nó không dùng

✕ **Bad:** Fat interface
class Worker(ABC):
@abstractmethod
def work(self): ...
@abstractmethod
def eat(self): ...

class Robot(Worker): # Robot không cần eat()
def work(self): ...
def eat(self):
pass # Forced to implement unused method

✓ **Good:** Tách thành multiple interfaces
class Workable(ABC):
@abstractmethod
def work(self): ...

class Eatable(ABC):
@abstractmethod
def eat(self): ...

class Human(Workable, Eatable):
def work(self): ...
def eat(self): ...

class Robot(Workable):
def work(self): ...

D - Dependency Inversion Principle:
High-level modules không nên depend vào low-level modules. Cả hai nên depend vào abstractions.

✕ **Bad:** UserService depend trực tiếp vào PostgresRepository
class PostgresUserRepository:
def save(self, user): ...

class UserService:
def **init**(self):
self.repo = PostgresUserRepository() # Tight coupling

✅ **Good:** Depend vào interface
class UserRepository(ABC):
@abstractmethod
def save(self, user): ...

class PostgresUserRepository(UserRepository):
def save(self, user): ...

class UserService:
def **init**(self, repo: UserRepository):
self.repo = repo # Depend on abstraction

7.3. DRY, KISS, YAGNI

DRY (Don't Repeat Yourself):
Mỗi piece of knowledge chỉ nên có 1 representation trong hệ thống

❌ **Bad:** Duplicate validation logic

Controller 1

def create_user(data):
if not data.email or '@' not in data.email:
raise ValueError("Invalid email")
...

Controller 2

def update_user(data):
if not data.email or '@' not in data.email:
raise ValueError("Invalid email")
...

✅ **Good:** Extract to reusable function
def validate_email(email):

if not email or '@' not in email:
raise ValueError("Invalid email")

def create_user(data):
validate_email(data.email)
...

KISS (Keep It Simple, Stupid):
Chọn giải pháp đơn giản nhất solve được vấn đề

❌ **Bad:** Over-engineering

Singleton factory builder visitor pattern cho... một config file

class ConfigSingletonFactoryBuilder:
_instance = None
...

✅ **Good:** Simple solution

Just use a global dict hoặc simple class

config = load_json("config.json")

YAGNI (You Aren't Gonna Need It):
Đừng implement tính năng mà bạn "nghĩ" sẽ cần trong tương lai

❌ **Bad:** Premature optimization

Implement caching system phức tạp cho feature chưa có users nào dùng

class MultiLevelCacheWithEvictionPolicy:
...

✅ **Good:** Implement khi thật sự cần

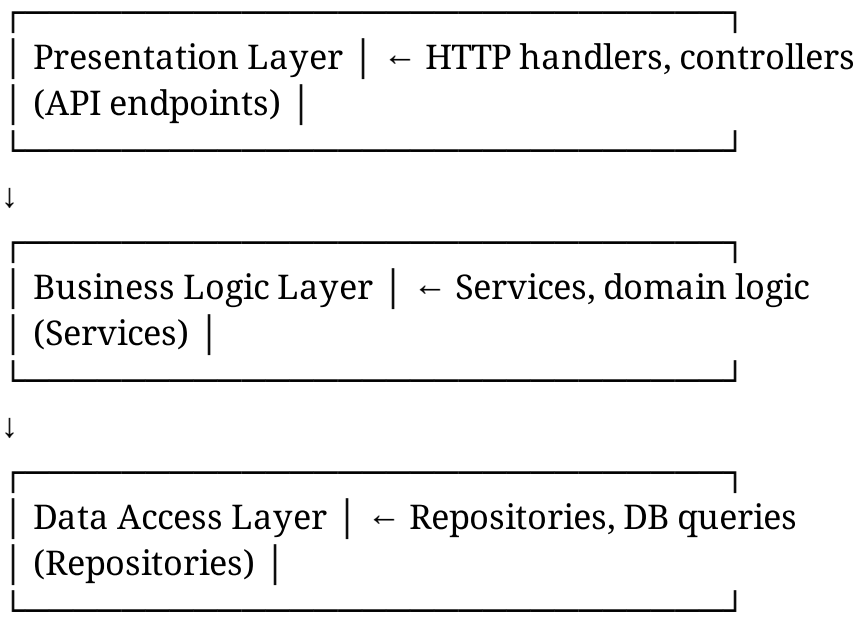
Start simple, add cache sau khi có performance issue thật

def get_user(id):
return db.query("SELECT * FROM users WHERE id = ?", id)

7.4. Separation of Concerns & Layered Architecture

Ý tưởng:
Tách code thành layers, mỗi layer một trách nhiệm

Typical 3-tier architecture:



Ví dụ cụ thể:

Presentation Layer

@app.post("/users")
def create_user_endpoint(data: UserCreateRequest):
user = user_service.create_user(data)
return UserResponse(user)

Business Logic Layer

class UserService:
def **init**(self, user_repo, email_service):
self.user_repo = user_repo
self.email_service = email_service

```
def create_user(self, data):
    # Validation
    if self.user_repo.exists_by_email(data.email):
        raise ValueError("Email already exists")

    # Business logic
    user = User(
        name=data.name,
        email=data.email,
        created_at=datetime.now()
    )

    # Persist
    self.user_repo.save(user)

    # Side effects
    self.email_service.send_welcome_email(user)

    return user
```

Data Access Layer

class UserRepository:
def save(self, user):
db.execute("INSERT INTO users (...) VALUES (...)", user)

```
def exists_by_email(self, email):
    return db.exists("SELECT 1 FROM users WHERE email = ?", email)
```

Lợi ích:

- Test dễ: mock layer bên dưới
- Thay đổi implementation dễ: ví dụ đổi DB mà không đụng business logic
- Đọc code dễ: biết nên tìm logic ở đâu

7.5. Dependency Injection

Vấn đề:
Class tự tạo dependencies → hard to test, tight coupling

class UserService:
def **init**(self):
self.repo = PostgresUserRepository() # Hard-coded
self.email = SendGridEmailService() # Hard-coded

Giải pháp: Inject dependencies

class UserService:
def **init**(self, repo: UserRepository, email: EmailService):
self.repo = repo
self.email = email

Production

service = UserService(
repo=PostgresUserRepository(),
email=SendGridEmailService()
)

Testing

service = UserService(
repo=MockUserRepository(),
email=MockEmailService()
)

DI Container (optional):
Frameworks như Spring (Java), NestJS (TS) có DI container tự động inject

@Injectable()
class UserService {
constructor(
private repo: UserRepository,
private email: EmailService
) {} // Container tự inject
}

Câu hỏi tự kiểm tra

1. Trong code của bạn, có class nào vi phạm SRP (làm quá nhiều việc)? Làm thế nào để refactor?
2. Bạn cần thêm một payment method mới (Bitcoin). Code hiện tại có follow Open/Closed Principle không? Nếu không, làm sao để refactor?
3. Service của bạn đang hard-code `PostgresRepository()` trong constructor. Làm sao để apply Dependency Injection?

Chương 8: Design Patterns

8.1. Tại sao học design patterns?

Không phải để:

- ✕ Học thuộc 23 patterns của Gang of Four
- ✕ Fit patterns vào mọi vấn đề
- ✕ "Show off" bằng cách dùng pattern phức tạp

Mà để:

- ✅ Có vocabulary chung khi discuss với team: "Dùng Strategy pattern cho payment processing"
- ✅ Nhận diện problems đã có proven solutions
- ✅ Hiểu code của người khác (nhiều libraries/frameworks dùng patterns)

8.2. Creational Patterns

Factory Pattern:
Tạo object mà không expose creation logic

Use case: Tạo different types of notifications

class Notification(ABC):
@abstractmethod
def send(self, message): ...

class EmailNotification(Notification):
def send(self, message):
print(f"Email: {message}")

class SMSNotification(Notification):
def send(self, message):
print(f"SMS: {message}")

class PushNotification(Notification):
def send(self, message):
print(f"Push: {message}")

Factory

class NotificationFactory:
@staticmethod
def create(type: str) -> Notification:
if type == 'email':
return EmailNotification()
elif type == 'sms':
return SMSNotification()
elif type == 'push':
return PushNotification()

else:
raise ValueError(f"Unknown type: {type}")

Usage

notification = NotificationFactory.create('email')
notification.send("Hello!")

Builder Pattern:
Xây dựng complex object step by step

Use case: Tạo HTTP request với nhiều optional parameters

class HttpRequest:
def **init**(self):
self.method = None
self.url = None
self.headers = {}
self.body = None
self.timeout = 30

class HttpRequestBuilder:
def **init**(self):
self.request = HttpRequest()

```
def method(self, method):
    self.request.method = method
    return self

def url(self, url):
    self.request.url = url
    return self

def header(self, key, value):
    self.request.headers[key] = value
    return self

def body(self, body):
    self.request.body = body
```

```
    return self

def timeout(self, timeout):
    self.request.timeout = timeout
    return self

def build(self):
    return self.request
```

Usage (fluent interface)

request = (HttpRequestBuilder()
.method('POST')
.url('https://api.example.com/users')
.header('Content-Type', 'application/json')
.header('Authorization', 'Bearer token123')
.body({'name': 'Alice'})
.timeout(60)
.build())

Singleton Pattern:
Ensure chỉ có 1 instance của class

Use case: Database connection pool, logger, config

class DatabasePool:
_instance = None

```
def __new__(cls):
    if cls._instance is None:
        cls._instance = super().__new__(cls)
        cls._instance.connections = []  # Initialize once
    return cls._instance
```

Usage

pool1 = DatabasePool()
pool2 = DatabasePool()
assert pool1 is pool2 # Same instance

Lưu ý: Singleton có thể là anti-pattern (global state, hard to test). Consider alternatives: dependency injection với scope singleton.

8.3. Structural Patterns

Adapter Pattern:
Chuyển đổi interface của class thành interface khác

Use case: Integrate với third-party library có interface khác

Third-party library (không thể thay đổi)

class LegacyEmailService:
def send_message(self, to, subject, content):
print(f"Sending to {to}: {subject} - {content}")

Our interface

class EmailService(ABC):
@abstractmethod
def send(self, email: Email): ...

Adapter

class LegacyEmailAdapter(EmailService):
def **init**(self):
self.legacy_service = LegacyEmailService()

```
def send(self, email: Email):
    self.legacy_service.send_message(
        to=email.recipient,
        subject=email.subject,
        content=email.body
    )
```

Usage

service: EmailService = LegacyEmailAdapter()
service.send(Email(recipient='alice@example.com', subject='Hi', body='...'))

Decorator Pattern:
Add behavior to object dynamically

Use case: Add logging, caching, retry logic

Base component

class DataSource(ABC):
@abstractmethod
def read(self): ...

class DatabaseDataSource(DataSource):
def read(self):
return "data from database"

Decorators

class CachedDataSource(DataSource):
def **init**(self, wrapped: DataSource):
self.wrapped = wrapped
self.cache = None

```
def read(self):
    if self.cache is None:
        print("Cache miss")
        self.cache = self.wrapped.read()
    else:
        print("Cache hit")
    return self.cache
```

class LoggedDataSource(DataSource):
def **init**(self, wrapped: DataSource):
self.wrapped = wrapped

```
def read(self):
    print("Reading data...")
    data = self.wrapped.read()
    print(f"Read {len(data)} bytes")
    return data
```

Usage (stack decorators)

data_source = LoggedDataSource(
CachedDataSource(
DatabaseDataSource()
)
)

data_source.read() # Logs + cache miss + DB read
data_source.read() # Logs + cache hit

Facade Pattern:
Provide simplified interface to complex subsystem

Use case: Simplify complex library hoặc multiple services

Complex subsystem

class PaymentGateway:
def authorize(self, card, amount): ...
def capture(self, auth_id): ...
def refund(self, transaction_id): ...

class InventoryService:
def reserve(self, product_id, quantity): ...
def release(self, reservation_id): ...

class ShippingService:
def calculate_cost(self, address): ...
def create_shipment(self, order): ...

Facade

class OrderFacade:
def **init**(self):
self.payment = PaymentGateway()
self.inventory = InventoryService()
self.shipping = ShippingService()

```
def place_order(self, order_data):
    # Hide complexity
    reservation = self.inventory.reserve(
        order_data.product_id,
        order_data.quantity
    )

    auth = self.payment.authorize(
        order_data.card,
        order_data.amount
    )

    try:
        self.payment.capture(auth.id)
```

```
        shipment = self.shipping.create_shipment(order_data)
        return Order(reservation, auth, shipment)
    except Exception:
        self.inventory.release(reservation.id)
        raise
```

Usage (simple interface)

facade = OrderFacade()
order = facade.place_order(order_data) # All complexity hidden

8.4. Behavioral Patterns

Strategy Pattern:
Define family of algorithms, encapsulate each one, make them interchangeable

Use case: Different payment methods, sorting algorithms, compression algorithms

Strategy interface

class PaymentStrategy(ABC):
@abstractmethod
def pay(self, amount): ...

Concrete strategies

class CreditCardStrategy(PaymentStrategy):
def **init**(self, card_number):
self.card_number = card_number

```
def pay(self, amount):
    print(f"Paying ${amount} with credit card {self.card_number}")
```

class PayPalStrategy(PaymentStrategy):
def **init**(self, email):

self.email = email

```
def pay(self, amount):
    print(f"Paying ${amount} via PayPal {self.email}")
```

class CryptoStrategy(PaymentStrategy):
def **init**(self, wallet_address):
self.wallet_address = wallet_address

```
def pay(self, amount):
    print(f"Paying ${amount} to crypto wallet {self.wallet_address}")
```

Context

class ShoppingCart:
def **init**(self):
self.items = []
self.payment_strategy = None

```
def set_payment_strategy(self, strategy: PaymentStrategy):
    self.payment_strategy = strategy

def checkout(self):
    total = sum(item.price for item in self.items)
    self.payment_strategy.pay(total)
```

Usage

cart = ShoppingCart()
cart.add_item(Item("Laptop", 1000))

User chọn payment method

cart.set_payment_strategy(CreditCardStrategy("1234-5678"))
cart.checkout()

Hoặc

cart.set_payment_strategy(PayPalStrategy("alice@example.com"))
cart.checkout()

Observer Pattern (Pub/Sub):
Define one-to-many dependency: khi object thay đổi state, tất cả dependents được notify

Use case: Event system, real-time updates, message broker

Subject

class EventManager:
def **init**(self):
self.listeners = {}

```
def subscribe(self, event_type, listener):
    if event_type not in self.listeners:
        self.listeners[event_type] = []
    self.listeners[event_type].append(listener)

def notify(self, event_type, data):
    if event_type in self.listeners:
        for listener in self.listeners[event_type]:
            listener.update(data)
```

Observers

class EmailListener:
def update(self, data):
print(f"Sending email for event: {data}")

class LoggerListener:
def update(self, data):
print(f"Logging event: {data}")

class AnalyticsListener:
def update(self, data):
print(f"Tracking event: {data}")

Usage

event_manager = EventManager()
event_manager.subscribe('user_created', EmailListener())
event_manager.subscribe('user_created', LoggerListener())
event_manager.subscribe('user_created', AnalyticsListener())

When event happens

event_manager.notify('user_created', {'user_id': 123, 'email': 'alice@example.com'})

→ Email sent

→ Event logged

→ Analytics tracked

Template Method Pattern:
Define skeleton of algorithm, let subclasses override specific steps

Use case: Data processing pipeline, authentication flow

class DataProcessor(ABC):
# Template method
def process(self, data):
raw = self.extract(data)
validated = self.validate(raw)
transformed = self.transform(validated)
self.load(transformed)

```
@abstractmethod
def extract(self, data): ...

@abstractmethod
def validate(self, data): ...

@abstractmethod
def transform(self, data): ...

@abstractmethod
def load(self, data): ...
```

Concrete implementations

class CSVProcessor(DataProcessor):
def extract(self, data):
print("Extracting from CSV")
return data

```
def validate(self, data):
    print("Validating CSV data")
    return data

def transform(self, data):
    print("Transforming CSV to JSON")
    return data
```

```
    def load(self, data):
        print("Loading to database")
```

class XMLProcessor(DataProcessor):
def extract(self, data):
print("Extracting from XML")
return data

```
# ... similar structure, different implementation
```

Usage

processor = CSVProcessor()
processor.process(data) # Follow same flow, different implementation

8.5. Khi nào dùng pattern?

Signals để dùng pattern:

Factory: Khi có nhiều subclasses và cần choose dynamically
Builder: Khi object có nhiều optional parameters (>5)
Adapter: Khi integrate third-party library incompatible interface
Decorator: Khi cần add behavior dynamically without modifying class
Strategy: Khi có nhiều algorithms interchangeable
Observer: Khi cần event-driven, multiple subscribers
Template Method: Khi có algorithm với skeleton chung, steps khác nhau

Anti-patterns to avoid:

- ✕ Áp pattern vào mọi thứ (over-engineering)
- ✕ Dùng pattern vì "cool" chứ không solve real problem
- ✕ Tạo abstraction khi chưa có >2 implementations (YAGNI)

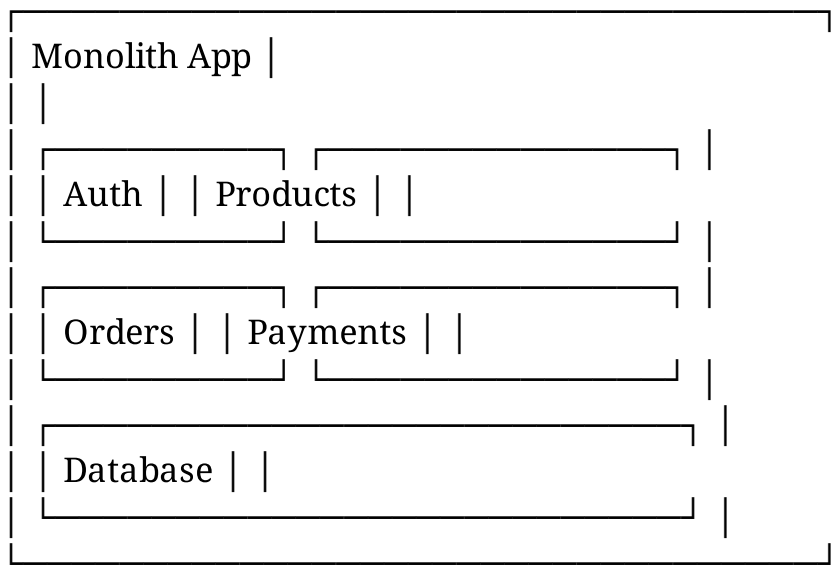
Câu hỏi tự kiểm tra

1. Hệ thống của bạn có 5 loại notifications (email, SMS, push, Slack, webhook). Code hiện tại là một đống if-else. Pattern nào phù hợp để refactor?
2. Bạn cần add logging và caching cho một service mà không thay đổi code hiện tại. Pattern nào?
3. Bạn có 3 data sources (DB, API, file) và cần process với flow giống nhau nhưng implementation khác nhau. Pattern nào?

Chương 9: Kiến trúc ứng dụng

9.1. Monolith vs Microservices vs Event-Driven

Monolith:
Toàn bộ application trong 1 codebase, deploy cùng nhau



Ưu điểm:

- Đơn giản: develop, test, deploy dễ
- Performance: function calls thay vì network calls
- Transaction: dễ dàng ACID transactions
- Debugging: 1 process, dễ trace

Nhược điểm:

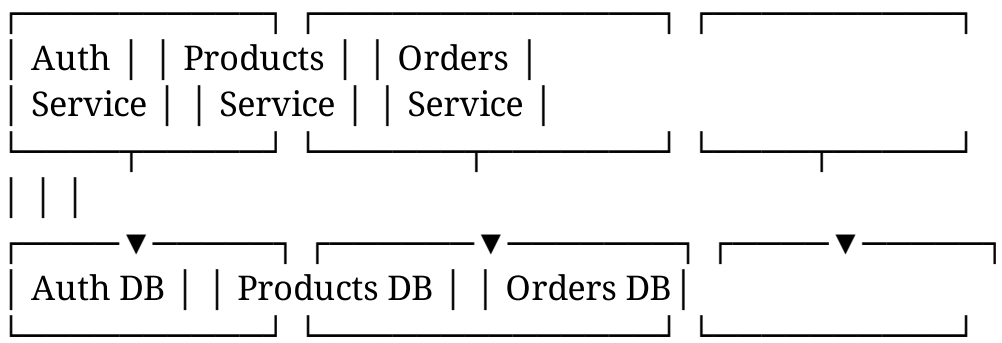
- Scaling: phải scale toàn bộ app (không thể scale riêng 1 module)

- Team size: khó nhiều team cùng làm (merge conflicts, tight coupling)
- Deployment: 1 change nhỏ phải deploy lại toàn bộ
- Technology: stuck với 1 tech stack
- Reliability: 1 module crash → toàn bộ app crash

Khi nào dùng Monolith:

- Startup giai đoạn đầu (speed to market)
- Team nhỏ (< 10 devs)
- Domain đơn giản, không cần scale khác nhau giữa các modules
- MVP, prototype

Microservices:
Chia application thành nhiều services độc lập, mỗi service một trách nhiệm



Ưu điểm:

- Independent scaling: scale service theo nhu cầu
- Independent deployment: deploy service mà không ảnh hưởng others
- Technology freedom: service A dùng Python, service B dùng Go
- Team autonomy: mỗi team own 1-2 services
- Fault isolation: service A crash không làm service B chết

Nhược điểm:

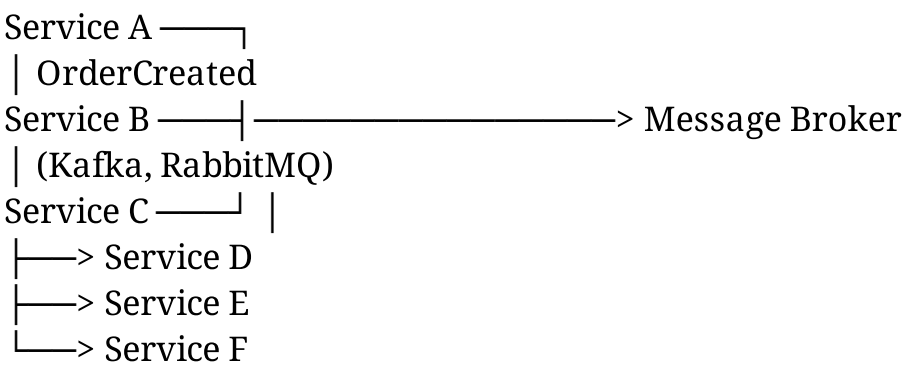
- Complexity: distributed system (network, latency, partial failure)
- Data consistency: không có distributed transactions dễ
- Testing: integration testing phức tạp
- Operational overhead: nhiều services → nhiều logs, metrics, deployments

- Network overhead: function call → HTTP call

Khi nào dùng Microservices:

- Scale lớn (different services need different scaling)
- Team lớn (> 20 devs), muốn autonomy
- Domain phức tạp, tách rõ bounded contexts
- Cần deploy frequently và independently

Event-Driven Architecture:
Services giao tiếp qua events (messages) thay vì direct calls



Ưu điểm:

- Decoupling: services không biết về nhau
- Scalability: consumers có thể scale independently
- Reliability: asynchronous, không sợ service down
- Auditability: event log = source of truth

Nhược điểm:

- Eventual consistency: không có immediate consistency
- Debugging: khó trace flow qua nhiều services
- Message ordering: có thể nhận events không đúng thứ tự
- Duplicate messages: cần idempotency

Khi nào dùng Event-Driven:

- Use cases async (không cần response ngay)
- Cần decoupling cao
- Cần audit trail (event sourcing)
- Multiple consumers cho 1 event

9.2. Ví dụ thực tế: E-commerce System

Yêu cầu:

- User authentication
- Product catalog
- Shopping cart
- Order processing
- Payment
- Inventory management
- Notification

Approach 1: Monolith

Single codebase

class UserController:
def register(self, data): ...
def login(self, data): ...

class ProductController:
def list_products(self): ...
def get_product(self, id): ...

class OrderController:
def create_order(self, data):
# All in one transaction
order = Order(data)
inventory.reserve(order.items)
payment.charge(order.total)
notification.send_email(order.user)
return order

Approach 2: Microservices

Services:

1. **Auth Service:** user authentication/authorization
2. **Product Service:** product catalog, search
3. **Cart Service:** shopping cart
4. **Order Service:** order processing

5. **Payment Service:** payment processing
6. **Inventory Service:** stock management
7. **Notification Service:** email, SMS

Flow: Create order (synchronous)

1. Client → Order Service: POST /orders
2. Order Service → Inventory Service: POST /reservations
3. Order Service → Payment Service: POST /charges
4. Order Service → Notification Service: POST /emails
5. Order Service → Client: 200 OK

Vấn đề: Nếu step 3 (Payment) fail → phải rollback step 2 (Inventory) → Distributed transaction phức tạp

Approach 3: Event-Driven Microservices

Flow: Create order (asynchronous)

1. Client → Order Service: POST /orders
2. Order Service:
    - Save order với status = PENDING
    - Publish event: OrderCreated
    - Return 202 Accepted
3. Inventory Service subscribe OrderCreated:
    - Reserve stock
    - Publish event: StockReserved
4. Payment Service subscribe StockReserved:
    - Charge payment
    - Publish event: PaymentCompleted
5. Order Service subscribe PaymentCompleted:
    - Update order status = CONFIRMED
    - Publish event: OrderConfirmed
6. Notification Service subscribe OrderConfirmed:
    - Send email

Ưu điểm:

- Services hoàn toàn độc lập
- Dễ add new consumers (Analytics Service subscribe tất cả events)
- Có audit trail (replay events)

Nhược điểm:

- Client không biết khi nào order xong (cần polling hoặc WebSocket)
- Phức tạp hơn: error handling, compensation logic

9.3. Saga Pattern: Distributed Transaction

Vấn đề:
Microservices không có distributed ACID transaction
→ Làm sao ensure consistency?

Giải pháp: Saga pattern

Saga = sequence of local transactions, mỗi transaction publish event

Nếu step N fail → execute compensating transactions cho steps 1..N-1

Ví dụ: Order saga

Happy path:

1. Order Service: Create order (status = PENDING)
2. Inventory Service: Reserve stock
3. Payment Service: Charge
4. Order Service: Confirm order (status = CONFIRMED)

Failure path: Payment fails

1. Order Service: Create order (PENDING)
2. Inventory Service: Reserve stock ✓
3. Payment Service: Charge ✖ FAIL
4. Compensate: Inventory Service: Release stock
5. Compensate: Order Service: Cancel order (status = CANCELLED)

Implementation:

Order Service

def create_order(data):
order = Order(data, status='PENDING')
db.save(order)

```
event_bus.publish('OrderCreated', {
    'order_id': order.id,
    'items': order.items,
    'user_id': order.user_id,
    'total': order.total
})

return order
```

Event handlers

@subscribe('StockReserved')
def on_stock_reserved(event):
order = db.get_order(event.order_id)
order.status = 'STOCK_RESERVED'
db.save(order)

@subscribe('PaymentCompleted')
def on_payment_completed(event):
order = db.get_order(event.order_id)
order.status = 'CONFIRMED'
db.save(order)

@subscribe('PaymentFailed')
def on_payment_failed(event):
order = db.get_order(event.order_id)
order.status = 'CANCELLED'
db.save(order)

```
# Trigger compensation
event_bus.publish('OrderCancelled', {
    'order_id': order.id
})
```

Inventory Service

@subscribe('OrderCancelled')
def on_order_cancelled(event):
reservation = db.get_reservation(event.order_id)
release_stock(reservation)

Lưu ý:

- Idempotency: mỗi event handler phải idempotent (xử lý event nhiều lần = 1 lần)
- Timeout: nếu không nhận event sau X phút → trigger compensation
- Monitoring: track saga state để debug

Câu hỏi tự kiểm tra

1. Startup của bạn có 3 devs, build MVP. Chọn kiến trúc nào? Tại sao?
2. Hệ thống hiện tại là monolith, có vấn đề: module Orders cần scale 10x nhưng module Products OK. Chiến lược migration?
3. Trong event-driven architecture, làm sao ensure payment chỉ charge 1 lần dù nhận event OrderCreated nhiều lần (duplicate messages)?

Chương 10: System Design

10.1. Tại sao System Design là kỹ năng "khó bị thay thế"?

AI có thể:

- Generate code cho component cụ thể
- Suggest patterns cho bài toán đã biết

AI khó làm:

- Thiết kế hệ thống end-to-end với nhiều constraints mâu thuẫn
- Trade-off giữa consistency, availability, partition tolerance (CAP theorem)
- Quyết định kiến trúc dựa trên business constraints, budget, team size

Lý do:

- System design cần:
  - Hiểu requirements không rõ ràng (clarifying questions)
  - Xét đoán dựa trên kinh nghiệm (best practices vs trade-offs)
  - Cân nhắc nhiều dimensions (technical, business, operational)
  - Chịu trách nhiệm cho quyết định (AI không chịu trách nhiệm)

10.2. Framework để approach System Design

Bước 1: Requirements clarification (5-10 phút)

Functional requirements:

- Core features gì?
- Use cases chính?

Non-functional requirements:

- Scale: bao nhiêu users, requests/s, data size?
- Performance: latency requirements?
- Availability: uptime target (99.9%? 99.99%)?
- Consistency: strong consistency hay eventual consistency OK?

Bước 2: Estimation (5 phút)

Back-of-the-envelope calculations:

- Traffic: QPS (Queries Per Second)
- Storage: data size
- Bandwidth: network

Bước 3: High-level design (10-15 phút)

Draw boxes:

- Client
- Load Balancer
- App Servers
- Database
- Cache
- CDN (nếu có static assets)

Bước 4: Deep dive (15-20 phút)

Pick 2-3 components để dig deeper:

- Database schema
- API design
- Caching strategy
- Scaling strategy

Bước 5: Discussion (còn lại)

- Bottlenecks?
- Single points of failure?
- Monitoring & alerting?
- Deployment strategy?

10.3. Ví dụ: Design URL Shortener (như bit.ly)

Step 1: Requirements

Functional:

- Shorten URL: long URL → short URL
- Redirect: short URL → long URL
- Custom alias (optional)
- Analytics (optional)

Non-functional:

- Scale: 100M URLs created/month, 10B redirects/month
- Latency: redirect < 100ms
- Availability: 99.99%
- Data retention: forever (hoặc expire sau X năm)

Step 2: Estimation

Write QPS:

- 100M URLs/month = 100M / (30 * 24 * 3600) ≈ 40 URLs/s
- Peak: 40 * 10 = 400 URLs/s

Read QPS:

- 10B redirects/month ≈ 4000 redirects/s
- Peak: 40,000 redirects/s

Read:Write ratio = 100:1 (read-heavy)

Storage:

- Mỗi URL: long URL (200 bytes) + short code (7 bytes) + metadata (100 bytes) ≈ 300 bytes
- 100M URLs/month * 12 months * 5 years = 6B URLs
- 6B * 300 bytes ≈ 1.8 TB

Step 3: High-level design

Client
↓
Load Balancer
↓
App Servers (stateless)
↓
Cache (Redis) ← 80% hits
↓
Database (SQL hoặc NoSQL)

API:

- POST /shorten
  - Body: { "long_url": "https://example.com/very/long/path" }
  - Response: { "short_url": "https://short.ly/abc123" }
- GET /{short_code}
  - Response: 302 redirect to long URL

Step 4: Deep dive

4.1. Short code generation

Yêu cầu:

- Unique
- Short (6-7 characters)
- Random (không dự đoán được)

Option 1: Hash (MD5, SHA256)

- Hash long URL → lấy first 7 characters
- Problem: collision (2 URLs khác nhau → cùng 7 chars)
- Giải pháp: check collision, nếu có thì append counter và hash lại

Option 2: Base62 encoding

- Generate unique ID (auto-increment hoặc distributed ID generator)
- Encode ID thành base62: [a-zA-Z0-9]
- 62^7 ≈ 3.5 trillion unique codes

Chọn option 2:

- Đơn giản hơn
- Guarantee unique
- Predictable length

4.2. Distributed ID generation

Vấn đề: Nhiều app servers, làm sao generate unique ID?

Option A: Database auto-increment

- Problem: single point of failure, bottleneck

Option B: UUID

- Problem: 128-bit (quá dài)

Option C: Twitter Snowflake

- 64-bit ID = timestamp (41 bits) + machine ID (10 bits) + sequence (12 bits)
- Guarantee unique, sortable by time

- 4096 IDs/ms per machine

4.3. Database schema

Table: urls

Column	Type	Index
id	bigint	PK
short_code	varchar(7)	Unique index
long_url	text	
user_id	bigint	Index
created_at	timestamp	Index
expire_at	timestamp	Index

4.4. Caching

Cache short_code → long_url

- TTL: 24 hours (hoặc không expire vì URLs ít thay đổi)
- Eviction: LRU
- Cache hit rate target: 80%+

Cache size estimation:

- 10B requests/month, 80% hit = 8B hits
- Cache top 20% popular URLs
- 6B URLs * 20% = 1.2B URLs
- 1.2B * 300 bytes ≈ 360 GB
- Distribute across 10 Redis instances → 36 GB each (OK)

4.5. Database scaling

Strategy: Read replicas + Sharding

Read replicas:

- 1 primary (write) + 5 replicas (read)
- Read load: 40K QPS / 5 = 8K QPS per replica (OK)

Sharding (khi data > TB):

- Shard key: hash(short_code) % num_shards
- Range-based: short_code [a-m] → shard 1, [n-z] → shard 2 (có thể hotspot)
- Hash-based: uniform distribution

4.6. Rate limiting

Prevent abuse:

- Limit: 10 URLs/hour per IP
- Implement: Redis với key = IP, counter, TTL = 1 hour

Step 5: Bottlenecks & improvements

Bottleneck:

- Database write: 400 QPS (peak) → 1 primary OK
- Database read: 40K QPS → cache + replicas handle

Single points of failure:

- Load balancer: 2 LBs với health check
- Database primary: automatic failover với replicas
- Redis: Redis Cluster với replication

Monitoring:

- Metrics: QPS, latency P50/P95/P99, error rate, cache hit rate
- Alerts: latency > 200ms, error rate > 1%, cache hit < 70%

10.4. CAP Theorem

CAP = Consistency + Availability + Partition Tolerance

Theorem: Distributed system chỉ có thể guarantee tối đa 2/3

Consistency:
Mọi read nhận latest write (hoặc error)

Availability:
Mọi request nhận response (có thể không phải latest)

Partition Tolerance:
Hệ thống tiếp tục hoạt động dù có network partition

Trade-off:

CA (Consistency + Availability):

- Không partition tolerance → không thể scale distributed
- Ví dụ: single-node database (PostgreSQL, MySQL single instance)

CP (Consistency + Partition Tolerance):

- Khi có partition → sacrifice availability
- Ví dụ: MongoDB, HBase, Zookeeper
- Use case: Financial transactions, inventory

AP (Availability + Partition Tolerance):

- Khi có partition → sacrifice consistency (eventual consistency)
- Ví dụ: Cassandra, DynamoDB, Riak
- Use case: Social media feeds, product catalog

Trong thực tế:

- Hầu hết systems chọn AP hoặc CP tùy use case
- Nhiều systems có tunable consistency (ví dụ: Cassandra)

10.5. Patterns cho Scalability

1. Horizontal scaling:
Add more machines (scale out)

2. Vertical scaling:
Add more CPU/RAM to existing machine (scale up)
→ Có limit, expensive

3. Caching:
Store frequently accessed data in memory

- Types: client-side, CDN, server-side (Redis), database query cache

4. Load balancing:
Distribute traffic across multiple servers

- Algorithms: round robin, least connections, IP hash

5. Database replication:
Read replicas cho read-heavy workloads

6. Database sharding:
Partition data across multiple databases

7. Asynchronous processing:
Use message queues for non-critical tasks

8. CDN:
Cache static assets gần users (images, CSS, JS)

9. Microservices:
Split monolith → independent services

10. Auto-scaling:
Automatically add/remove instances based on metrics

Câu hỏi tự kiểm tra

1. Design một hệ thống chat như Slack/Discord với:
   - 10M daily active users
   - Real-time messaging
   - Message history
   - File upload

   Bạn sẽ thiết kế như thế nào?
2. Hệ thống URL shortener ở trên có bottleneck gì khi scale lên 100x (10B URLs/month → 1 trillion)?
3. Bạn phải chọn giữa strong consistency và high availability cho một social media feed. Bạn chọn gì? Tại sao?

PHẦN IV: CLOUD, DEVOPS & SRE

Chương 11: Tư duy Cloud-Native

11.1. Tại sao Cloud Engineer/DevOps/SRE là nghề "khó bị thay thế"?

Theo báo cáo WEF và LinkedIn, cloud-related roles nằm top fastest-growing tech jobs:[cite:18][cite:21][cite:27]

- Cloud Engineer
- DevOps Engineer
- Site Reliability Engineer (SRE)
- Platform Engineer

Lý do khó tự động hóa:

- Cần hiểu hạ tầng, networking, security end-to-end
- Quyết định phức tạp: cost vs performance vs reliability trade-offs
- Incident response: cần xét đoán trong tình huống mơ hồ, áp lực cao
- Vận hành production: chịu trách nhiệm cho uptime, data integrity

AI có thể:

- Generate Terraform/CloudFormation code
- Suggest best practices

AI khó làm:

- Design hạ tầng cho hệ thống cụ thể với constraints cụ thể
- Debug production incident với log phức tạp, missing information
- Optimize cost khi có hàng trăm services, hàng nghìn resources
- Make judgment call: "Nên rollback hay fix forward?"

11.2. Cloud Fundamentals

Core services (AWS examples):

Compute:

- EC2: Virtual machines

- Lambda: Serverless functions
- ECS/EKS: Containers
- Fargate: Managed containers

Storage:

- S3: Object storage (files, images, backups)
- EBS: Block storage (disk for EC2)
- EFS: File storage (shared filesystem)

Database:

- RDS: Managed SQL (PostgreSQL, MySQL)
- DynamoDB: NoSQL key-value
- Aurora: High-performance MySQL/PostgreSQL
- ElastiCache: Redis/Memcached

Networking:

- VPC: Virtual network
- Subnet: Network segments (public/private)
- Security Group: Firewall rules
- Internet Gateway: Connect VPC to internet
- NAT Gateway: Outbound internet for private subnet
- Load Balancer: Distribute traffic (ALB/NLB)

IAM (Identity and Access Management):

- Users, Groups, Roles
- Policies: who can do what on which resources

11.3. Ví dụ: Deploy web app lên AWS

Architecture:

Internet
↓
Route 53 (DNS)
↓
CloudFront (CDN) ← S3 (static assets)
↓
Application Load Balancer

↓
EC2 Auto Scaling Group (app servers)
↓
RDS (database)

Step-by-step:

1. Setup VPC
VPC: 10.0.0.0/16
├─ Public Subnet 1: 10.0.1.0/24 (AZ us-east-1a)
├─ Public Subnet 2: 10.0.2.0/24 (AZ us-east-1b)
├─ Private Subnet 1: 10.0.3.0/24 (AZ us-east-1a)
└─ Private Subnet 2: 10.0.4.0/24 (AZ us-east-1b)

Public subnet: có Internet Gateway → access internet
Private subnet: có NAT Gateway → outbound only

2. Security Groups

ALB Security Group:
Inbound: 443 (HTTPS) from 0.0.0.0/0
Outbound: All

App Security Group:
Inbound: 8080 from ALB Security Group
Outbound: All

DB Security Group:
Inbound: 5432 from App Security Group
Outbound: None

3. Launch RDS (PostgreSQL)

- Multi-AZ: primary ở AZ-1a, standby ở AZ-1b
- Private subnets (không expose ra internet)
- Automated backups: daily, retention 7 days
- Encryption at rest

4. Launch EC2 với Auto Scaling

Launch Template:

- AMI: Amazon Linux 2

- Instance type: t3.medium
- User data script:
  #!/bin/bash
  yum update -y
  yum install -y docker
  systemctl start docker
  docker run -d -p 8080:8080
  -e DATABASE_URL=postgresql://...
  myapp:latest

Auto Scaling Group:

- Min: 2, Desired: 2, Max: 10
- Target tracking: CPU > 70% → scale out
- Health check: ELB health check

5. Application Load Balancer

- Listener: HTTPS:443
- Target group: EC2 instances port 8080
- Health check: GET /health → 200 OK

6. CloudFront + S3 cho static assets

- S3 bucket: myapp-static
- Upload: CSS, JS, images
- CloudFront distribution:
  - Origin: S3 bucket
  - Cache policy: 24 hours
  - HTTPS only

7. Route 53

- Domain: myapp.com
- A record: myapp.com → ALB
- CNAME: static.myapp.com → CloudFront

Result:

- User → CloudFront (static) hoặc ALB (API)
- ALB → EC2 instances (round robin)
- EC2 → RDS

- Auto scaling: traffic tăng → add instances
- High availability: multi-AZ

11.4. Infrastructure as Code (IaC)

Vấn đề:

- Click qua Console → không reproducible
- Manual → error-prone
- Không có version control

Giải pháp: IaC

- Terraform
- CloudFormation (AWS)
- Pulumi
- CDK (Cloud Development Kit)

Ví dụ Terraform:

VPC

resource "aws_vpc" "main" {
cidr_block = "10.0.0.0/16"

tags = {
Name = "main-vpc"
}
}

Public Subnet

resource "aws_subnet" "public" {
count = 2
vpc_id = aws_vpc.main.id
cidr_block = "10.0.${count.index + 1}.0/24"
availability_zone =
data.aws_availability_zones.available.names[count.index]

```
tags = {
Name = "public-subnet-${count.index + 1}"
}
}
```

Security Group

```
resource "aws_security_group" "app" {
vpc_id = aws_vpc.main.id

ingress {
from_port = 8080
to_port = 8080
protocol = "tcp"
security_groups = [aws_security_group.alb.id]
}

egress {
from_port = 0
to_port = 0
protocol = "-1"
cidr_blocks = ["0.0.0.0/0"]
}
}
```

RDS

```
resource "aws_db_instance" "main" {
identifier = "myapp-db"
engine = "postgres"
engine_version = "14.6"
instance_class = "db.t3.medium"

allocated_storage = 100
max_allocated_storage = 1000 # Auto-scaling storage

db_name = "myapp"
username = "admin"
password = var.db_password # From variable
```

multi_az = true
publicly_accessible = false
vpc_security_group_ids = [aws_security_group.db.id]
db_subnet_group_name = aws_db_subnet_group.main.name

backup_retention_period = 7
backup_window = "03:00-04:00"
maintenance_window = "sun:04:00-sun:05:00"

enabled_cloudwatch_logs_exports = ["postgresql"]

tags = {
Name = "myapp-db"
}
}

Auto Scaling Group

resource "aws_autoscaling_group" "app" {
name = "app-asg"
vpc_zone_identifier = aws_subnet.private[*].id
target_group_arns = [aws_lb_target_group.app.arn]
health_check_type = "ELB"

min_size = 2
max_size = 10
desired_capacity = 2

launch_template {
id = aws_launch_template.app.id
version = "$Latest"
}

tag {
key = "Name"
value = "app-instance"
propagate_at_launch = true
}
}

Workflow:

1. Write code (Terraform)
2. `terraform plan` → show changes
3. Review
4. `terraform apply` → execute
5. Commit to Git
6. CI/CD auto-apply on merge to main

Lợi ích:

- Version control: rollback dễ dàng
- Reproducible: recreate infrastructure từ code
- Documentation: code = documentation
- Collaboration: code review infrastructure changes
- Automation: CI/CD apply changes

11.5. Cost Optimization

Cloud cost có thể explode nếu không cẩn thận

Common wastes:

- EC2 instances idle (dev/test không tắt)
- Over-provisioned instances (t3.xlarge khi t3.small đủ)
- Data transfer (giữa AZs, ra internet)
- Unused EBS volumes, snapshots
- Unused Elastic IPs

Strategies:

1. Right-sizing:
Monitor actual usage (CPU, memory, network) → downsize

Ví dụ: EC2 instance dùng 10% CPU → switch từ t3.large → t3.medium = save 50%

2. Reserved Instances / Savings Plans:
Commit 1-3 years → discount 30-70%

Use case: baseline workload (always-on production)

3. Spot Instances:
Bid for unused capacity → discount 70-90%

Use case: batch jobs, non-critical workloads, stateless apps

4. Auto-scaling:
Scale down khi không cần (ví dụ: dev environment tắt đi ban đêm)

5. S3 lifecycle policies:

- Transition to S3 Infrequent Access sau 30 days
- Transition to Glacier sau 90 days
- Delete sau 365 days

6. Delete unused resources:

- EBS volumes không attach
- Old snapshots
- Elastic IPs không dùng
- Load balancers không có targets

7. Use AWS Cost Explorer & Budgets:

- Set alerts: "Nếu monthly cost > $1000 → email"
- Analyze: service nào tốn nhất?

Ví dụ thực tế:

Company X:

- Ban đầu: $50K/month
- Audit:
  - 30% EC2 idle → schedule stop/start = save $5K
  - 20% over-provisioned → right-size = save $3K
  - Unused resources (old EBS, snapshots) = save $2K
  - Reserved Instances cho baseline = save $10K
- Sau optimize: $30K/month → save 40%

Câu hỏi tự kiểm tra

1. Bạn cần deploy một web app lên cloud. App có:
   - Web server (stateless)
   - Database (PostgreSQL)
   - File uploads (images)

   Bạn chọn AWS services nào? Vẽ architecture.

2. Team của bạn manage infrastructure bằng Console (click). Có vấn đề gì? Làm thế nào để improve?
3. Cloud bill tháng này tăng đột ngột từ $5K → $15K. Bạn sẽ investigate thế nào?

Chương 12: Container & Kubernetes

12.1. Tại sao Container?

Vấn đề truyền thống:

Dev machine: Python 3.9, Ubuntu 20.04
→ "Works on my machine"

Production server: Python 3.7, CentOS 7
→ Broken

Dependency hell:

- Library A requires lib X v1.0
- Library B requires lib X v2.0
- Conflict

Giải pháp: Container

Container = package app + dependencies + runtime vào 1 unit

Container Image:
├─ App code
├─ Python 3.9
├─ Libraries (requirements.txt)
├─ OS libraries
└─ Config

Run ở đâu cũng giống nhau: dev laptop, CI/CD, production

12.2. Docker Basics

Dockerfile:

Base image

FROM python:3.9-slim

Set working directory

WORKDIR /app

Copy requirements

COPY requirements.txt .

Install dependencies

RUN pip install --no-cache-dir -r requirements.txt

Copy application code

COPY . .

Expose port

EXPOSE 8080

Run app

CMD ["python", "app.py"]

Build & Run:

Build image

docker build -t myapp:v1.0 .

Run container

docker run -d -p 8080:8080
-e DATABASE_URL=postgresql://...
--name myapp
myapp:v1.0

View logs

docker logs myapp

Execute command in container

docker exec -it myapp bash

Docker Compose (local development):

version: '3.8'

services:
app:
build: .
ports:
- "8080:8080"
environment:
DATABASE_URL: postgresql://db:5432/myapp
depends_on:
- db
- redis

db:
image: postgres:14
environment:
POSTGRES_DB: myapp

POSTGRES_USER: user
POSTGRES_PASSWORD: password
volumes:
- db-data:/var/lib/postgresql/data

redis:
image: redis:7-alpine
ports:
- "6379:6379"

volumes:
db-data:

Start all services

docker-compose up -d

Stop all

docker-compose down

12.3. Kubernetes (K8s) Overview

Vấn đề:

- Có 100 containers
- Cần deploy, scale, health check, restart khi crash
- Cần load balancing
- Cần rolling update (deploy mới mà không downtime)

Kubernetes = Container orchestration platform

Core concepts:

Pod:
Smallest unit, 1+ containers (thường 1)

apiVersion: v1
kind: Pod
metadata:

name: myapp-pod
spec:
containers:

- name: myapp
  image: myapp:v1.0
  ports:
    - containerPort: 8080

Deployment:
Manage replicas của Pods

apiVersion: apps/v1
kind: Deployment
metadata:
name: myapp
spec:
replicas: 3 # Chạy 3 pods
selector:
matchLabels:
app: myapp
template:
metadata:
labels:
app: myapp
spec:
containers:
- name: myapp
image: myapp:v1.0
ports:
- containerPort: 8080
env:
- name: DATABASE_URL
valueFrom:
secretKeyRef:
name: db-secret
key: url
resources:
requests:
memory: "256Mi"

cpu: "500m"
limits:
memory: "512Mi"
cpu: "1000m"
livenessProbe:
httpGet:
path: /health
port: 8080
initialDelaySeconds: 30
periodSeconds: 10
readinessProbe:
httpGet:
path: /ready
port: 8080
initialDelaySeconds: 10
periodSeconds: 5

Service:
Expose Pods (load balancing)

apiVersion: v1
kind: Service
metadata:
name: myapp-service
spec:
selector:
app: myapp
ports:

- protocol: TCP
  port: 80
  targetPort: 8080
  type: LoadBalancer # hoặc ClusterIP, NodePort

ConfigMap:
Store config (non-sensitive)

apiVersion: v1
kind: ConfigMap
metadata:
name: app-config

data:
LOG_LEVEL: "info"
FEATURE_FLAG_X: "true"

Secret:
Store sensitive data (passwords, API keys)

apiVersion: v1
kind: Secret
metadata:
name: db-secret
type: Opaque
data:
url: cG9zdGdyZXNxbDovL... # base64 encoded

Ingress:
HTTP routing

apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
name: myapp-ingress
annotations:
cert-manager.io/cluster-issuer: "letsencrypt-prod"
spec:
tls:

- hosts:
  - myapp.com
    secretName: myapp-tls
    rules:
- host: myapp.com
  http:
  paths:
  - path: /
    pathType: Prefix
    backend:
    service:
    name: myapp-service
    port:
    number: 80

12.4. Kubernetes Scaling

Horizontal Pod Autoscaler (HPA):

apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
name: myapp-hpa
spec:
scaleTargetRef:
apiVersion: apps/v1
kind: Deployment
name: myapp
minReplicas: 2
maxReplicas: 10
metrics:

- type: Resource
  resource:
  name: cpu
  target:
  type: Utilization
  averageUtilization: 70

CPU > 70% → scale out
CPU < 70% → scale in (với cooldown)

Cluster Autoscaler:
Tự động add/remove nodes (EC2 instances) khi cần

12.5. Best Practices

1. Multi-stage Docker builds (giảm image size):

Build stage

FROM node:18 AS builder
WORKDIR /app
COPY package*.json ./
RUN npm ci

```
COPY . .
RUN npm run build

# Production stage

FROM node:18-alpine
WORKDIR /app
COPY --from=builder /app/dist ./dist
COPY package*.json ./
RUN npm ci --only=production
CMD ["node", "dist/index.js"]
```

2. Health checks:

- Liveness probe: container còn sống không? (nếu fail → restart)
- Readiness probe: container sẵn sàng nhận traffic không? (nếu fail → remove khỏi load balancer)

3. Resource limits:

- Requests: minimum cần
- Limits: maximum cho phép
- Nếu exceed limits → container bị kill (OOMKilled)

4. Don't run as root:

```
RUN addgroup -S appgroup && adduser -S appuser -G appgroup
USER appuser
```

5. Secrets management:

- Không hard-code trong image
- Dùng Kubernetes Secrets hoặc external (AWS Secrets Manager, HashiCorp Vault)

6. Logging:

- Log to stdout/stderr (Kubernetes collect automatically)
- Structured logging (JSON)

7. Rolling updates:

- Deploy version mới dần dần (1 pod at a time)
- Nếu health check fail → rollback tự động

spec:
strategy:
type: RollingUpdate
rollingUpdate:
maxSurge: 1
maxUnavailable: 0

Câu hỏi tự kiểm tra

1. Viết Dockerfile cho một Python Flask app với requirements.txt. Làm sao để image size nhỏ nhất?
2. Bạn deploy app lên Kubernetes với 3 replicas. Traffic tăng đột ngột, CPU tăng lên 90%. Làm thế nào để tự động scale?
3. Pod của bạn bị crash loop: start → crash → restart → crash. Làm sao để debug?

(Tiếp tục với các chương còn lại...)ss-shard queries khó (ví dụ: JOIN users từ shard 1 với orders từ shard 2)

- Transactions cross-shard khó (hoặc không support)
- Resharding (thay đổi số shards) = migration lớn

Use case:

- Very large dataset không fit trong 1 server
- Write-heavy và không thể scale đủ bằng replicas

3. Caching:
Thêm cache layer (Redis, Memcached) trước DB

App → Cache → DB

Flow:

1. App query cache
2. Cache hit → return
3. Cache miss → query DB → store in cache → return

Ưu:

- Giảm load DB đáng kể (cache hit rate 90%+ = 90% requests không tới DB)
- Latency thấp (in-memory)

Nhược:

- Cache invalidation phức tạp
- Stale data nếu không invalidate đúng

Cache strategies:

- Cache-aside: app control cache
- Write-through: write cache + DB cùng lúc
- Write-behind: write cache trước, async write DB sau

Câu hỏi tự kiểm tra

1. Bạn có query chậm:
   SELECT * FROM orders WHERE user_id = 123 AND status = 'pending' ORDER BY created_at DESC LIMIT 10;
   Nên tạo index như thế nào?
2. Hệ thống của bạn có 1 triệu users, 100 triệu orders. Hiện tại 1 DB server. Làm thế nào để scale?
3. Bạn cache user profile trong Redis (TTL = 3600s). User update profile → làm sao để cache không stale?

PHẦN III: THIẾT KẾ HỆ THỐNG & KIẾN TRÚC

Chương 7: System Design - Thiết kế hệ thống phân tán

7.1. Tại sao System Design là kỹ năng "khó bị thay thế"?

AI có thể:

- Generate code cho một feature
- Viết test
- Refactor theo pattern

AI **khó** làm:

- Quyết định kiến trúc tổng thể của hệ thống
- Trade-off giữa consistency, availability, latency, cost
- Thiết kế cho scale (1K users vs 10M users)
- Xử lý failure scenarios (network partition, server crash)

Lý do: System design cần:

- Hiểu requirements mơ hồ → đặt câu hỏi clarifying
- Kinh nghiệm về các patterns và anti-patterns
- Xét đoán (judgment) chứ không phải thuật toán
- Trade-off awareness: không có "best solution", chỉ có "suitable solution"

7.2. CAP Theorem

Statement:
Distributed system chỉ có thể đảm bảo 2 trong 3:

- **C**onsistency: Mọi read nhận data mới nhất
- **A**vailability: Mọi request đều nhận response (không error)
- **P**artition tolerance: Hệ thống tiếp tục hoạt động dù có network partition

Giải thích:

Network partition (P) là thực tế không tránh khỏi trong distributed system
→ Phải chọn: C hay A khi có partition

CP system:
Chọn Consistency, hy sinh Availability
Khi có partition → một phần hệ thống không respond (để đảm bảo consistency)

Ví dụ: Traditional SQL với synchronous replication

- Primary down → system không accept writes (để không inconsistent)

AP system:
Chọn Availability, hy sinh Consistency

Khi có partition → vẫn respond, nhưng data có thể stale hoặc inconsistent

Ví dụ: Cassandra, DynamoDB

- Mỗi node nhận writes độc lập
- Eventually consistent: sau một khoảng thời gian, data sẽ consistent

CA system:
Chỉ tồn tại trong single-node system (không distributed)

Thực tế:
Không phải black-and-white

- Có thể CP cho một phần data, AP cho phần khác
- Có thể tunable: DynamoDB cho phép chọn consistency level per request

7.3. Consistency Models

Strong Consistency:
Sau write thành công, mọi read ngay lập tức thấy data mới

Cơ chế:

- Synchronous replication: write phải replicate tới tất cả nodes trước khi ack
- Hoặc: quorum read/write

Trade-off:

- Ưu: Đơn giản cho developer
- Nhược: Latency cao, availability thấp

Eventual Consistency:
Sau write, có một khoảng thời gian mà reads có thể thấy data cũ
Sau đó (eventually), tất cả replicas converge về data mới

Cơ chế:

- Asynchronous replication
- Conflict resolution (last-write-wins, vector clock, ...)

Trade-off:

- Ưu: Latency thấp, availability cao
- Nhược: Phức tạp cho developer (phải handle stale data)

Ví dụ thực tế:

Facebook timeline:

- Post mới → replicate async
- User ở Singapore có thể thấy post sau user ở US vài giây
- OK vì eventual consistency chấp nhận được

Bank balance:

- Cần strong consistency
- Read balance phải là accurate
- Eventual consistency = sai số dư → vấn đề lớn

7.4. Distributed Transactions

Vấn đề:
Transaction span nhiều services/databases

Ví dụ: E-commerce order

1. Order service: tạo order
2. Payment service: charge card
3. Inventory service: reserve items
4. Shipping service: tạo shipment

Nếu step 3 fail → phải rollback step 1, 2

Giải pháp:

1. Two-Phase Commit (2PC):
Phase 1: Prepare

- Coordinator gửi "prepare" tới tất cả participants
- Participants vote: "yes" (ready to commit) hoặc "no" (abort)

Phase 2: Commit/Abort

- Nếu tất cả vote "yes": coordinator gửi "commit"

- Nếu có 1 vote "no": coordinator gửi "abort"

Trade-off:

- Ưu: Strong consistency
- Nhược:
  - Blocking: nếu coordinator crash giữa 2 phases → participants bị block
  - Latency cao
  - Availability thấp

Thực tế: ít dùng trong microservices, chủ yếu trong DB clusters

2. Saga Pattern:
Sequence of local transactions
Mỗi transaction có compensating transaction (để rollback)

Ví dụ:

1. Create order → compensating: cancel order
2. Charge payment → compensating: refund
3. Reserve inventory → compensating: release inventory

Nếu step 3 fail:

- Execute compensating for step 2 (refund)
- Execute compensating for step 1 (cancel order)

Implementation:

- Choreography: mỗi service emit event, service khác listen và react
- Orchestration: một orchestrator điều phối

Trade-off:

- Ưu: Không blocking, high availability
- Nhược:
  - Eventual consistency
  - Phức tạp: phải implement compensating transactions
  - Có thể có failure ở compensating transaction

Thực tế: Phổ biến trong microservices

3. Try-Confirm/Cancel (TCC):
Biến thể của 2PC

Mỗi operation có 3 phases:

- Try: reserve resources (ví dụ: reserve $100 từ account)
- Confirm: commit (deduct $100)
- Cancel: rollback (release reservation)

7.5. Caching Strategies

Where to cache:

- Client-side: browser cache, mobile app cache
- CDN: static assets (images, JS, CSS)
- Application cache: Redis, Memcached
- Database cache: query result cache

Cache invalidation strategies:

1. Time-to-Live (TTL):
Cache expire sau X seconds

Ưu: Đơn giản
Nhược: Stale data trong TTL period

2. Write-through:
Write cache + DB cùng lúc

Ưu: Cache luôn fresh
Nhược: Latency cao (phải write 2 chỗ)

3. Write-behind (Write-back):
Write cache trước, async write DB sau

Ưu: Latency thấp
Nhược: Risk data loss nếu cache crash trước khi persist

4. Cache-aside:
App check cache → miss → query DB → populate cache

Ưu: Flexible, app control invalidation
Nhược: Có thể stale nếu không invalidate đúng

Eviction policies:
Khi cache đầy, evict gì?

- LRU (Least Recently Used): evict item lâu không dùng nhất
- LFU (Least Frequently Used): evict item ít dùng nhất
- FIFO: evict item cũ nhất
- Random

Thực tế: LRU là phổ biến nhất

Cache stampede:
Cache expire → nhiều requests cùng lúc query DB → DB overload

Giải pháp:

- Lock: request đầu tiên query DB, các requests khác đợi
- Probabilistic early expiration: random refresh trước khi expire

7.6. Load Balancing Strategies

Layer 4 vs Layer 7:
(Đã nói ở Chương 5, tóm tắt lại)

Layer 4: IP + port, fast, simple
Layer 7: HTTP headers, URL, cookies, flexible, powerful

Algorithms:

- Round Robin
- Least Connections
- Weighted Round Robin
- IP Hash (sticky sessions)

Health Checks:
Passive: detect failures khi request fail
Active: ping /health định kỳ

Session Affinity:
Vấn đề: User session lưu ở server A, request kế tiếp route tới server B → session mất

Giải pháp:

1. Sticky sessions: load balancer route cùng user → cùng server
   Nhược: Không scale tốt, server down = mất session
2. Session store tập trung: Redis, Memcached
   Ưu: Scale tốt, HA
   Nhược: Latency thêm

7.7. Message Queue & Event-Driven Architecture

Vấn đề:
Service A gọi service B synchronously → latency, coupling cao

Giải pháp:
Message queue: async communication

Producer → Queue → Consumer

Benefits:

- Decoupling: producer không cần biết consumer
- Async: producer không đợi consumer process
- Load leveling: consumer process với rate riêng của nó
- Reliability: message persist trong queue, retry nếu fail

Patterns:

1. Point-to-point:
1 message → 1 consumer

Use case: task queue (email sending, image processing)

2. Pub-Sub:
1 message → nhiều subscribers

Use case: event notification (order created → inventory, shipping, analytics subscribe)

Popular tools:

- RabbitMQ: traditional message queue
- Kafka: high-throughput, log-based
- AWS SQS/SNS: managed service
- Redis Pub/Sub: simple, in-memory

Kafka deep dive:
Đặc biệt: không phải queue truyền thống, là distributed log

Concepts:

- Topic: category of messages
- Partition: shard của topic (parallel processing)
- Offset: position trong partition
- Consumer group: nhiều consumers cùng group share partitions

Ưu:

- Throughput rất cao (millions messages/sec)
- Durability: persist to disk, replication
- Replay: có thể đọc lại message cũ từ offset

Use case:

- Event sourcing
- Stream processing
- Log aggregation

7.8. Microservices vs Monolith

Monolith:
Toàn bộ app trong 1 codebase, 1 deployment

Ưu:

- Đơn giản: dễ develop, debug, deploy
- Transaction dễ: tất cả trong 1 DB
- No network overhead

Nhược:

- Scale: phải scale cả cục (dù chỉ cần scale 1 module)
- Deploy: deploy 1 chỗ → risk ảnh hưởng toàn bộ
- Team: khó chia team theo domain (codebase lớn, conflict nhiều)
- Tech stack: mắc kẹt với 1 tech stack

Microservices:
Chia app thành nhiều services nhỏ, độc lập

Ưu:

- Scale: scale từng service riêng
- Deploy: deploy độc lập, ít risk
- Team: mỗi team own 1-2 services
- Tech stack: mỗi service dùng tech phù hợp

Nhược:

- Complexity: network, distributed transactions, observability
- Overhead: mỗi service cần infra (DB, cache, monitoring)
- Data consistency: không có transaction cross-service dễ

Khi nào chọn gì?

Start with monolith khi:

- Team nhỏ (< 10 người)
- Domain chưa rõ (chưa biết cần chia thế nào)
- Chưa có vấn đề scale

Move to microservices khi:

- Team lớn (20+ người)
- Domain rõ ràng, có thể chia theo bounded context
- Cần scale selective (một phần chịu load cao)
- Cần deploy frequently và independently

"Majestic Monolith":
Monolith tốt với:

- Modular architecture bên trong (DDD, clean architecture)
- Clear boundaries (chuẩn bị cho sau này tách microservices nếu cần)

7.9. Design Patterns for Distributed Systems

1. Circuit Breaker:
Vấn đề: Service B down → Service A keep gọi B → timeout, latency cao

Giải pháp:
States: Closed → Open → Half-open

- Closed: normal, requests pass through
- Failures > threshold → Open: requests fail fast (không gọi B)
- Sau timeout → Half-open: thử vài requests
    - Success → Closed
    - Fail → Open lại

2. Retry với Exponential Backoff:
Transient failure (tạm thời) → retry

Strategy:

- Retry 1: đợi 1s
- Retry 2: đợi 2s
- Retry 3: đợi 4s
- ...

Tránh thundering herd: thêm random jitter

3. Rate Limiting:
Giới hạn số requests trong time window

Algorithms:

- Token bucket
- Leaky bucket
- Fixed window
- Sliding window

4. Bulkhead:
Isolation giữa các resources

Ví dụ: Thread pool riêng cho mỗi dependency

- Thread pool A cho service A (10 threads)
- Thread pool B cho service B (10 threads)
- Service A slow/down → chỉ block pool A, không ảnh hưởng pool B

7.10. Ví dụ System Design: URL Shortener

Requirements:
Functional:

- Shorten URL: long URL → short code
- Redirect: short code → long URL
- Custom short code (optional)
- Analytics (optional)

Non-functional:

- High availability (99.9%+)
- Low latency (< 100ms)
- Scalable (100M URLs, 10K requests/sec)

Design:

1. API:
POST /api/shorten
{
"long_url": "https://example.com/very/long/url",
"custom_code": "abc123" // optional
}

Response:
{
"short_url": "https://short.ly/abc123"
}

GET /{code}
→ 301/302 redirect to long URL

2. Database:
Table: urls

- id (primary key)
- short_code (unique index)
- long_url
- created_at
- expires_at (optional)

3. Short code generation:

Option A: Base62 encoding của auto-increment ID
ID = 12345 → base62 = "dnh"
6 characters base62 = 62^6 ≈ 56 billion URLs

Ưu: Đơn giản, guaranteed unique
Nhược: Predictable (có thể guess)

Option B: Hash (MD5/SHA) + truncate
MD5(long_url + salt) → truncate 6 chars

Ưu: Không predictable
Nhược: Collision có thể xảy ra → phải check và retry

Option C: Random + check unique
Generate random 6 chars → check DB → nếu tồn tại → retry

Ưu: Simple
Nhược: Có thể retry nhiều lần khi đầy

4. Caching:
Redis cache: short_code → long_url
TTL = 24 hours (URLs phổ biến)

Hit rate ≈ 80% (rule 80-20)

5. Scaling:

- DB: read replicas (read-heavy workload)
- Cache: Redis cluster
- App: stateless, scale horizontal
- Load balancer: distribute traffic

6. Analytics (optional):
Event: mỗi redirect → log (timestamp, short_code, IP, user_agent, referer)
Store: Kafka → stream processing → data warehouse (BigQuery, Snowflake)
Query: dashboard với metrics (clicks, geo, devices)

Câu hỏi tự kiểm tra

1. Bạn thiết kế một social media feed. User có 1000 followers. Khi user post, làm thế nào để 1000 followers thấy post trong feed của họ?
    - Push model: write to 1000 feeds ngay khi post?
    - Pull model: mỗi follower query khi mở app?
    - Hybrid?
2. Hệ thống của bạn call external payment API. API đôi khi timeout. Làm thế nào để handle?
3. Design một hệ thống chat có 10M concurrent users. Yêu cầu: realtime, message history, group chat.

PHẦN IV: CLOUD, DEVOPS & SRE

Chương 8: Cloud-Native Thinking

8.1. Tại sao cloud là kỹ năng "future-proof"?

Theo LinkedIn 2026, hơn 70% software job listings yêu cầu kỹ năng cloud.[cite:74][cite:77]

Lý do:

- Majority of workload đang/sẽ chuyển lên cloud (on-prem → cloud migration wave)
- Cost efficiency: pay-as-you-go vs upfront hardware investment
- Speed: provision resources trong minutes vs weeks
- Global reach: deploy globally với vài clicks

AI khó thay thế cloud engineer vì:

- Cần hiểu business context: cost optimization, compliance, security
- Cần xét đoán: trade-off giữa managed service vs self-managed
- Cần kinh nghiệm: troubleshoot production issues
- Cần phối hợp: security, network, devs, product

8.2. Cloud Service Models

IaaS (Infrastructure as a Service):
Provider: compute, storage, network
Bạn: OS, runtime, app

Ví dụ: AWS EC2, Google Compute Engine, Azure VMs

Use case:

- Full control
- Migration từ on-prem (lift-and-shift)

PaaS (Platform as a Service):
Provider: compute, storage, network, OS, runtime
Bạn: app, data

Ví dụ: Heroku, Google App Engine, AWS Elastic Beanstalk

Use case:

- Focus on app development
- Faster deployment

SaaS (Software as a Service):
Provider: everything
Bạn: dùng

Ví dụ: Gmail, Salesforce, Slack

Serverless / FaaS:
Provider: everything kể cả scaling
Bạn: function code

Ví dụ: AWS Lambda, Google Cloud Functions, Azure Functions

Use case:

- Event-driven
- Sporadic workload
- Zero ops

8.3. Core Cloud Concepts

Regions & Availability Zones:
Region: geographic area (ví dụ: us-east-1, eu-west-1)
Availability Zone (AZ): isolated datacenter trong region

Best practice: Deploy across multiple AZs cho high availability

Ví dụ:

- Load balancer trải đều traffic qua 3 AZs
- Database replicas ở 3 AZs khác nhau
- 1 AZ down → still available

Auto Scaling:
Tự động tăng/giảm số instances dựa trên metrics

Scaling policies:

- Target tracking: maintain CPU = 50%
- Step scaling: CPU > 70% → add 2 instances, CPU > 90% → add 5 instances
- Scheduled scaling: scale up trước peak hours

Load Balancing:
(Đã nói ở chương 5)

Application Load Balancer (ALB): Layer 7, HTTP/HTTPS
Network Load Balancer (NLB): Layer 4, TCP/UDP

Virtual Private Cloud (VPC):
Isolated network trong cloud

Components:

- Subnets: public (internet-facing) vs private (internal only)
- Route tables: định tuyến traffic
- Internet Gateway: kết nối VPC với internet
- NAT Gateway: cho private subnet access internet (outbound only)
- Security Groups: firewall cho instances
- Network ACL: firewall cho subnets

8.4. Compute Options

VMs (EC2, Compute Engine):
Full control, flexible

Use case:

- Legacy apps
- Specific OS/kernel requirements
- Long-running processes

Containers (ECS, GKE, AKS):
Lightweight, portable

Use case:

- Microservices
- CI/CD pipelines
- Hybrid cloud (portability across clouds)

Serverless (Lambda, Cloud Functions):
No server management, pay per execution

Use case:

- API backends (với API Gateway)
- Data processing (triggered by S3 upload, DB change)
- Cron jobs
- Event-driven workflows

Managed Kubernetes:
Container orchestration platform

Use case:

- Complex microservices
- Multi-cloud strategy
- Need fine-grained control

8.5. Storage Options

Object Storage (S3, Cloud Storage):
Store files, images, videos

Features:

- Unlimited capacity
- Durability: 99.999999999% (11 nines)
- Versioning, lifecycle policies
- Static website hosting

Use case:

- Backups
- Data lakes
- Media storage
- Static assets (images, videos)

Block Storage (EBS, Persistent Disk):
Attached to VMs, like hard drive

Features:

- Low latency
- Snapshots

Use case:

- Database storage
- Application data

File Storage (EFS, Filestore):
Shared filesystem, multiple instances mount

Use case:

- Shared data giữa containers
- Home directories

8.6. Database Options

Managed Relational (RDS, Cloud SQL):
Provider manage: backups, patching, replication

Ưu:

- Less ops overhead
- Built-in HA, backups
- Easy scaling (read replicas, vertical scaling)

Nhược:

- Cost cao hơn self-managed
- Ít control hơn

Managed NoSQL (DynamoDB, Firestore):
Serverless, auto-scaling

Ưu:

- Zero ops
- Scale to massive workload

Nhược:

- Query limitations
- Learning curve (data modeling khác SQL)

Data Warehouse (BigQuery, Redshift, Snowflake):
Analytics, OLAP

Use case:

- Business intelligence
- Data analytics
- Reporting

8.7. Cost Optimization

Principles:

- Right-sizing: không over-provision
- Reserved instances: commit 1-3 years → discount 30-60%
- Spot instances: bid for unused capacity → discount 70-90% (có thể bị terminate)
- Auto-scaling: scale down khi không cần
- Storage tiering: S3 Standard → S3 IA → S3 Glacier (dữ liệu ít access)

Monitoring:

- CloudWatch (AWS), Cloud Monitoring (GCP), Azure Monitor
- Set budgets & alerts
- Analyze cost reports

Tagging:
Tag resources by team, project, environment
→ Cost allocation
→ Identify waste

Câu hỏi tự kiểm tra

1. Bạn deploy app lên cloud với 1 instance. App critical, cần 99.9% uptime. Làm sao để đạt được?
2. Workload của bạn:
    - 9am-5pm: 1000 requests/sec
    - 5pm-9am: 100 requests/sec

   Làm sao để optimize cost?
3. Bạn có 100TB data cũ (access < 1 lần/năm). Chi phí S3 Standard = $2300/tháng. Làm sao giảm?

Chương 9: DevOps & CI/CD

9.1. DevOps Culture

Traditional:
Dev team → code → "throw over the wall" → Ops team → deploy → maintain

Problems:

- Slow deployment (weeks/months)
- Lack of ownership (dev không quan tâm production)
- Finger-pointing khi có issue

DevOps:
Dev + Ops collaborate
"You build it, you run it" (Amazon motto)

Benefits:

- Faster deployment (days/hours)
- Higher quality (dev có skin in the game)
- Shared responsibility

Key practices:

- Automation (CI/CD, IaC)
- Monitoring & observability
- Continuous improvement (postmortems, retrospectives)
- Blameless culture (focus on system, not individual)

9.2. Continuous Integration (CI)

Goal: Integrate code changes frequently (multiple times/day)

Pipeline:

1. Developer push code → trigger CI
2. Run linting, static analysis
3. Run unit tests
4. Run integration tests
5. Build artifact (Docker image, binary)
6. If all pass → ready for deployment

Benefits:

- Catch bugs early (before merge)
- Reduce integration hell (merge conflicts)
- Always have a deployable artifact

Best practices:

- Fast feedback (< 10 minutes ideal)
- Fail fast (stop on first error)
- Keep builds green (broken build = top priority to fix)

9.3. Continuous Deployment (CD)

Goal: Automatically deploy every change that passes CI

Pipeline:
CI (build + test) → Deploy to staging → Automated tests on staging → Deploy to production

Deployment strategies:

1. Blue-Green:
2 environments: Blue (current prod) và Green (new version)

- Deploy to Green
- Run smoke tests on Green
- Switch traffic: Blue → Green
- Keep Blue for quick rollback

Ưu: Zero downtime, easy rollback
Nhược: 2x resources (cost)

2. Canary:
Gradually shift traffic to new version

- 5% traffic → new version
- Monitor metrics (error rate, latency)
- If OK → 25% → 50% → 100%
- If not OK → rollback

Ưu: Low risk (impact 5% users nếu có bug)
Nhược: Phức tạp hơn

3. Rolling:
Update instances one-by-one

- Instance 1 → new version
- Health check OK → Instance 2
- ...

Ưu: No extra resources
Nhược: Mixed versions cùng lúc (có thể compatibility issue)

9.4. Infrastructure as Code (IaC)

Concept:
Define infrastructure trong code (không phải manual click)

Tools:

- Terraform: multi-cloud, declarative
- AWS CloudFormation: AWS only
- Ansible: configuration management
- Pulumi: code-based (TypeScript, Python)

Example: Terraform
resource "aws_instance" "web" {
ami = "ami-abc123"
instance_type = "t2.micro"
tags = {
Name = "WebServer"
}
}

Benefits:

- Version control: track changes, review, rollback
- Reproducible: spin up identical environments (dev, staging, prod)
- Documentation: code = documentation
- Automation: no manual steps

Best practices:

- Store state remotely (S3, Terraform Cloud)
- Use modules (reusable components)

- Separate environments (dev.tf, prod.tf)

9.5. Configuration Management

Tools:

- Ansible: agentless, YAML playbooks
- Chef: agent-based, Ruby DSL
- Puppet: agent-based, declarative

Use cases:

- Install packages
- Configure services
- Manage users, permissions
- Deploy apps

Example: Ansible playbook

- hosts: webservers
  tasks:
  - name: Install Nginx
    apt: name=nginx state=present
  - name: Start Nginx
    service: name=nginx state=started

9.6. Monitoring & Observability

Metrics:
Numeric time-series data

Examples:

- CPU usage, memory usage
- Request rate, error rate, latency
- Queue depth, cache hit rate

Tools: Prometheus, Datadog, CloudWatch

Logs:
Discrete events

Best practices:

- Structured logging (JSON format)
- Include context (request_id, user_id, trace_id)
- Centralized (ELK stack, CloudWatch Logs, Splunk)

Traces:
Track request flow across services

Tools: Jaeger, Zipkin, AWS X-Ray

Example:
Request ID: abc-123

- API Gateway: 5ms
- Auth Service: 20ms
- Order Service: 100ms
    - Database query: 80ms
    - Payment API: 15ms

→ See bottleneck (DB query 80ms)

Alerting:
Notify khi có issue

Best practices:

- Alert on symptoms (user-facing), not causes
- Example:
  ✅ Alert: "Error rate > 1%"
  ❌ Alert: "CPU > 80%" (chưa chắc ảnh hưởng user)
- Runbook: what to do when alert fires
- On-call rotation: share burden

9.7. Incident Response

Severity levels:

- SEV1/P1: Critical (site down, data loss)
- SEV2/P2: Major (core feature broken)
- SEV3/P3: Minor (non-core feature broken)

Process:

1. Detect: monitoring alert

2. Triage: assess severity
3. Mitigate: restore service (rollback, restart, scale up)
4. Investigate: root cause analysis
5. Resolve: permanent fix
6. Postmortem: learn lessons

Postmortem:
Blameless, focus on system

- Timeline: what happened when
- Root cause: why it happened
- Action items: prevent recurrence

Example action items:

- Add monitoring for X metric
- Add test for Y scenario
- Improve documentation
- Automate Z runbook

Câu hỏi tự kiểm tra

1. Bạn deploy version mới → error rate tăng từ 0.1% → 5%. Bạn đang dùng canary deployment (10% traffic). Làm gì tiếp theo?
2. Team của bạn deploy manual (SSH vào server, pull code, restart). Tốn 2 hours/deploy. Làm sao để improve?
3. Production down lúc 3am. Bạn là on-call engineer. Hành động đầu tiên?

PHẦN V: SECURITY - AN NINH MẠNG

Chương 10: Security Fundamentals

10.1. Tại sao security là kỹ năng "future-proof"?

Cybersecurity workforce demand dự kiến tăng 32% tới 2030, nhanh hơn nhiều so với average các ngành khác.[cite:75][cite:78]

Lý do:

- Cyber attacks ngày càng tinh vi (ransomware, supply chain attacks)
- Regulatory compliance (GDPR, CCPA, HIPAA) ngày càng nghiêm
- Mọi công ty đều digitizing → attack surface tăng
- Cost của breach rất cao (average $4.45M theo IBM 2023)

AI khó thay thế security engineer vì:

- Attackers dùng AI → defenders cũng cần AI + human judgment
- Cần hiểu business context, risk appetite
- Cần creativity (think like attacker)
- Cần decision-making trong incident response

10.2. CIA Triad

Confidentiality (Bảo mật):
Data chỉ được access bởi người có quyền

Threats: data breach, unauthorized access
Controls: encryption, access control, authentication

Integrity (Toàn vẹn):
Data không bị modify trái phép

Threats: data tampering, man-in-the-middle
Controls: hashing, digital signatures, checksums

Availability (Khả dụng):
Data/service available khi cần

Threats: DoS/DDoS, ransomware, hardware failure
Controls: redundancy, backups, rate limiting

10.3. OWASP Top 10 (2021)

1. Broken Access Control:
User có thể access data/function không được phép

Example:

- IDOR (Insecure Direct Object Reference): `/api/users/123` → bạn thay 123 → 124 → xem được user khác
- Missing function level access: user thường gọi được admin API

Fix:

- Server-side access control check
- Principle of least privilege

2. Cryptographic Failures:
Sensitive data không được encrypt đúng cách

Example:

- Password stored plaintext
- HTTP instead of HTTPS
- Weak encryption algorithm (MD5, SHA1)

Fix:

- Encrypt data at rest (AES-256)
- Encrypt data in transit (TLS 1.2+)
- Hash passwords (bcrypt, Argon2)

3. Injection:
Untrusted data được execute như code

Example:

- SQL injection: `SELECT * FROM users WHERE name = ' + userInput + '`
  → `userInput = ' OR '1'='1` → bypass
- Command injection: `os.system("ping " + userInput)`
  → `userInput = 8.8.8.8; rm -rf /` → execute malicious command

Fix:

- Parameterized queries (prepared statements)
- Input validation
- Escaping
- Principle: never trust user input

4. Insecure Design:
Thiết kế hệ thống không an toàn từ đầu

Example:

- Không có rate limiting → brute force
- Không có audit log → không detect breach
- Mọi user đều admin by default

Fix:

- Threat modeling
- Security by design
- Defense in depth (multiple layers)

5. Security Misconfiguration:
Default config, permissions quá rộng

Example:

- Default password không đổi
- Directory listing enabled
- Error messages reveal stack trace, DB structure

Fix:

- Hardening: disable unused features
- Principle of least privilege
- Automated security scanning

6. Vulnerable and Outdated Components:
Dùng libraries có vulnerabilities

Example:

- Log4j vulnerability (Log4Shell) - Dec 2021
- OpenSSL Heartbleed - 2014

Fix:

- Dependency scanning (Snyk, Dependabot)
- Regular updates
- Monitor security advisories

7. Identification and Authentication Failures:
Weak auth mechanism

Example:

- Weak password policy (cho phép "123456")
- Không có MFA
- Session không expire
- Credential stuffing (reuse password từ leaked DB)

Fix:

- Strong password policy
- Multi-factor authentication (MFA)
- Session management (timeout, revoke)
- Rate limiting for login

8. Software and Data Integrity Failures:
Code/data bị modify mà không detect

Example:

- CI/CD pipeline bị compromise → inject malicious code
- Insecure deserialization → RCE

Fix:

- Code signing
- Verify signatures
- Integrity checks (checksums)

9. Security Logging and Monitoring Failures:
Không log đầy đủ → không detect attack

Example:

- Không log login attempts

- Không alert on suspicious activities
- Log không được protect (attacker xóa log)

Fix:

- Log security events (login, access, changes)
- Centralized logging
- Real-time monitoring & alerting
- Log retention policy

10. Server-Side Request Forgery (SSRF):
Attacker trick server thực hiện request

Example:
API: POST /fetch { "url": "https://example.com" }
→ Server fetch URL and return
→ Attacker: { "url": "http://169.254.169.254/latest/meta-data" } (AWS metadata)
→ Server fetch internal URL → leak credentials

Fix:

- Whitelist allowed domains
- Block internal IPs (localhost, 169.254.x.x, 10.x.x.x)
- Network segmentation

10.4. Authentication vs Authorization

Authentication (Xác thực):
"Who are you?"
Verify identity

Methods:

- Password
- OTP (One-Time Password)
- Biometrics (fingerprint, face)
- Security keys (YubiKey)

Authorization (Phân quyền):
"What can you do?"
Verify permissions

Models:

- RBAC (Role-Based Access Control)
  - User có roles (admin, editor, viewer)
  - Roles có permissions (read, write, delete)
- ABAC (Attribute-Based Access Control)
  - Dựa trên attributes (department, location, time)
  - Example: Allow if (user.department == "Finance" AND time >= 9am AND time <= 5pm)

10.5. OAuth 2.0 & OpenID Connect

OAuth 2.0:
Authorization framework
"Allow app X to access your data on service Y without giving password"

Roles:

- Resource Owner: user
- Client: app muốn access data
- Authorization Server: issue token
- Resource Server: API với data

Flow: Authorization Code (most secure):

1. User click "Login with Google"
2. Redirect to Google: /authorize? client_id=...&redirect_uri=...&scope=profile email
3. User login & consent
4. Google redirect back: /callback?code=abc123
5. App exchange code for token: POST /token { code: abc123, client_id, client_secret }
6. Google return access_token + refresh_token
7. App call API với access_token: GET /userinfo (Authorization: Bearer access_token)

OpenID Connect:
Layer on top of OAuth 2.0 cho authentication
→ Return ID token (JWT) chứa user info

10.6. JWT (JSON Web Token)

Structure:
Header.Payload.Signature

Example:
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c

Decoded:
Header: { "alg": "HS256", "typ": "JWT" }
Payload: { "sub": "1234567890", "name": "John Doe", "iat": 1516239022 }
Signature: HMACSHA256(header + payload, secret)

Claims:

- sub: subject (user ID)
- exp: expiration timestamp
- iat: issued at
- iss: issuer
- aud: audience
- Custom: role, permissions, ...

Security:

- Signature verify integrity (không bị modify)
- Expiration ngắn (15 minutes) + refresh token
- Store securely (không localStorage cho sensitive data → httpOnly cookie)
- Validate trên server (verify signature, expiration, issuer)

10.7. Encryption

Symmetric encryption:
Cùng key cho encrypt và decrypt

Algorithm: AES (Advanced Encryption Standard)

- AES-128, AES-192, AES-256 (số = key length bits)

Use case:

- Encrypt data at rest (database, files)
- Encrypt data in transit (TLS session key)

Asymmetric encryption:
2 keys: public key (encrypt) và private key (decrypt)

Algorithm: RSA, ECC (Elliptic Curve Cryptography)

Use case:

- TLS handshake (exchange symmetric key)
- Digital signatures
- SSH keys

Hashing:
One-way function: input → hash (không thể reverse)

Algorithm: SHA-256, SHA-3
Password hashing: bcrypt, Argon2 (slow by design → resist brute force)

Use case:

- Password storage (hash password, không lưu plaintext)
- Integrity check (file checksum)
- Digital signatures

Example: Password storage

✖ Bad:
users table: { email, password: "mysecret123" }
→ Breach = attacker có plaintext passwords

✖ Bad:
password: SHA256("mysecret123")
→ Rainbow table attack (precomputed hashes)

✅ Good:
password: bcrypt("mysecret123", salt, cost=12)
→ Salt = random string per user

→ Cost = iterations (12 = 2^12 = 4096 rounds)
→ Slow = hard to brute force

Câu hỏi tự kiểm tra

1. API của bạn: GET /api/orders/{orderId}. Làm sao để prevent IDOR (user A xem được order của user B)?
2. Bạn cần store user passwords. Chọn cách nào?
    - Plaintext?
    - MD5 hash?
    - bcrypt?
      Tại sao?
3. App của bạn dùng JWT với expiration = 7 days. Vấn đề là gì? Làm sao để improve?

PHẦN VI: AI & DATA INTEGRATION

Chương 11: Làm việc với LLM trong Production

11.1. Tại sao dev cần biết integrate AI?

Đến 2026, majority của apps sẽ có AI features.[cite:76][cite:79]

AI không phải "competitor" của dev, mà là "tool" dev phải biết dùng.

Kỹ năng cần:

- Call LLM API (OpenAI, Anthropic, Google, ...)
- Prompt engineering
- Handle LLM limitations (hallucination, context length, latency, cost)
- RAG (Retrieval-Augmented Generation)
- Evaluation & monitoring

11.2. LLM APIs Basics

Providers:

- OpenAI (GPT-4, GPT-3.5)
- Anthropic (Claude)
- Google (Gemini)
- Open-source (Llama, Mistral via Hugging Face, Replicate)

Basic call:
POST https://api.openai.com/v1/chat/completions
{
"model": "gpt-4",
"messages": [
{"role": "system", "content": "You are a helpful assistant."},
{"role": "user", "content": "Explain quantum computing in simple terms."}
],
"temperature": 0.7,
"max_tokens": 500
}

Response:
{
"choices": [
{"message": {"role": "assistant", "content": "Quantum computing is..."}}
],
"usage": {"prompt_tokens": 20, "completion_tokens": 100, "total_tokens": 120}
}

Parameters:

- temperature: 0 = deterministic, 1 = creative (default 0.7)
- max_tokens: limit response length
- top_p: nucleus sampling (alternative to temperature)
- stop: stop sequences

11.3. Prompt Engineering

Principles:

1. Be specific:
✖ "Summarize this."
✓ "Summarize the following article in 3 bullet points, focusing on key findings."

2. Provide context:
✖ "Is this code correct?"
✓ "This Python function calculates Fibonacci numbers. Is there a bug? If yes, explain and fix it."

3. Use examples (few-shot):
Classify sentiment: positive, negative, neutral

Examples:
"I love this product!" → positive
"Terrible experience." → negative
"It's okay." → neutral

Input: "Best purchase ever!"
Output: positive

4. Chain-of-thought:
"Solve this step-by-step:" → improves reasoning

5. System message:
Set behavior: "You are a Python expert. Provide concise, idiomatic solutions."

11.4. RAG (Retrieval-Augmented Generation)

Problem:
LLM knowledge cutoff (ví dụ: GPT-4 trained on data until Oct 2023)
→ Không biết data sau đó
→ Không biết company internal data

Solution: RAG

1. User query

2. Retrieve relevant documents từ knowledge base
3. Include documents trong prompt
4. LLM generate answer based on documents

Architecture:
Knowledge base → Vector DB (embeddings) → Retrieval → LLM

Steps:

1. Indexing:

- Documents → chunking (split thành đoạn ~500 tokens)
- Mỗi chunk → embedding (vector representation)
- Store embeddings trong vector DB (Pinecone, Weaviate, Chroma)

2. Retrieval:

- User query → embedding
- Similarity search trong vector DB (cosine similarity)
- Top K relevant chunks

3. Generation:

- Prompt = system message + retrieved chunks + user query
- LLM generate answer

Example:
User: "What is our company's vacation policy?"

Retrieved chunks:

- Chunk 1: "Employees receive 15 days of paid vacation per year..."
- Chunk 2: "Vacation requests must be submitted 2 weeks in advance..."

Prompt:
"Based on the following policy documents, answer the user's question:

[Chunk 1]
[Chunk 2]

Question: What is our company's vacation policy?"

LLM: "Employees receive 15 days of paid vacation per year. Requests must be submitted at least 2 weeks in advance..."

Benefits:

- Up-to-date info (add new documents without retraining)
- Company-specific data
- Reduce hallucination (grounded in retrieved docs)

Challenges:

- Retrieval quality critical (bad retrieval → bad answer)
- Context length limit (can't include too many chunks)
- Latency (retrieval + generation)

11.5. Evaluation & Monitoring

Metrics:

1. Accuracy:
% of correct answers
→ Need ground truth labels

2. Relevance:
Answer relevant to question?
→ Human eval hoặc LLM-as-judge

3. Hallucination rate:
% of made-up facts
→ Check against source documents

4. Latency:
Time từ request → response
→ P50, P95, P99

5. Cost:
Tokens used × price per token
→ Monitor & optimize

Logging:
Log mọi request/response:

- Timestamp

- User ID (anonymized nếu cần)
- Prompt
- Response
- Tokens used
- Latency
- Model version

→ Debug issues, improve prompts, detect abuse

A/B testing:
Test 2 prompts:

- Prompt A: temperature 0.5
- Prompt B: temperature 0.7
  → Measure quality, cost, latency
  → Pick winner

11.6. Safety & Compliance

Content filtering:

- Input filtering: block inappropriate prompts (toxicity, PII)
- Output filtering: block inappropriate responses

PII (Personally Identifiable Information):

- Don't include PII trong prompts (hoặc anonymize)
- Risk: LLM provider log prompts → data leak

Compliance:

- GDPR: user data handling
- HIPAA: healthcare data (không gửi PHI tới LLM API)
- Contractual: data residency, data processing agreements

Rate limiting & abuse prevention:

- Limit requests per user (prevent spam, abuse)
- Detect malicious prompts (jailbreak attempts)

11.7. Cost Optimization

LLM API costs:

- GPT-4: ~$0.03 per 1K prompt tokens, ~$0.06 per 1K completion tokens
- GPT-3.5: ~$0.001 per 1K tokens (30x cheaper)
- Claude: similar pricing tiers

With 1M requests/month, 500 tokens average:

- GPT-4: ~$45K/month
- GPT-3.5: ~$1.5K/month

Optimization strategies:

1. Use cheaper models when possible:

- Simple tasks (classification, extraction) → GPT-3.5 hoặc Llama
- Complex tasks (reasoning, generation) → GPT-4

2. Prompt optimization:

- Shorter prompts = lower cost
- Remove unnecessary examples, verbose instructions

3. Caching:

- Cache frequent queries
- TTL = hours/days depending on freshness needs

4. Batch processing:

- Batch API (if available) = cheaper but slower

5. Fine-tuning:

- Fine-tune smaller model on specific task → comparable quality, lower cost
- Trade-off: upfront cost + time

Câu hỏi tự kiểm tra

1. Bạn build chatbot customer support. LLM đôi khi "hallucinate" (make up policies). Làm sao để reduce?
2. Bạn dùng GPT-4 cho 1M requests/month, cost = $30K. Quá đắt. Làm sao để optimize?
3. User hỏi: "What is my account balance?" RAG system retrieve 5 chunks, nhưng không chunk nào có balance. LLM vẫn trả lời "Your balance is $1000". Vấn đề là gì? Fix thế nào?

PHẦN VII: PRODUCT & BUSINESS SENSE

Chương 12: Tư duy Product cho Developer

12.1. Tại sao dev cần product sense?

Traditional:
PM viết spec → Dev implement → QA test → Deploy
Dev không cần hiểu "why"

Modern:
Dev tham gia product decisions:

- Challenge requirements không hợp lý
- Đề xuất solutions đơn giản hơn
- Hiểu trade-offs (scope vs time vs quality)

Benefits:

- Better products (dev hiểu user → build đúng thứ)
- Faster execution (ít back-and-forth với PM)
- Career growth (product-minded engineer = highly valued)

12.2. Hiểu User & Pain Points

Framework: Jobs To Be Done (JTBD):
User "hire" product để làm một "job"

Example:

- Job: "I want to stay connected with friends"
- Product: Facebook, Instagram, WhatsApp
- Not: "I want a social media app"

Empathy:
Đặt câu hỏi:

- Who is the user?
- What problem are they trying to solve?
- What is their current workflow?
- Where does it fail?

User research:

- Talk to users (interviews)
- Observe users (usability testing)
- Analyze data (analytics, heatmaps)

12.3. Đo lường & Metrics

Business metrics:

- Revenue, MRR (Monthly Recurring Revenue)
- Customer Acquisition Cost (CAC)
- Lifetime Value (LTV)
- Churn rate

Product metrics:

- DAU/MAU (Daily/Monthly Active Users)
- Retention (% users return after N days)
- Engagement (time spent, features used)
- Conversion (% users complete goal)

Technical metrics:

- Latency (P50, P95, P99)
- Error rate
- Uptime (SLA)

Leading vs lagging indicators:

- Leading: predict future (ví dụ: sign-ups)
- Lagging: measure past (ví dụ: revenue)

12.4. Prioritization

Frameworks:

RICE:

- Reach: # users impacted
- Impact: magnitude of impact (low/medium/high)
- Confidence: certainty of estimates (%)
- Effort: time/resources needed (person-weeks)

Score = (Reach × Impact × Confidence) / Effort

ICE:

- Impact
- Confidence
- Ease

Score = (Impact × Confidence × Ease) / 3

Example:
Feature A: Add dark mode

- Reach: 10K users
- Impact: Medium (2)
- Confidence: 80%
- Effort: 2 weeks

Feature B: Fix critical security bug

- Reach: All users (100K)
- Impact: High (3)
- Confidence: 100%

- Effort: 1 week

RICE A = (10K × 2 × 0.8) / 2 = 8K
RICE B = (100K × 3 × 1.0) / 1 = 300K

→ Prioritize B

12.5. MVP & Iteration

MVP (Minimum Viable Product):
Smallest version có thể ship để validate hypothesis

Not:

- "Minimum" = low quality
- "Viable" = feature-complete

But:

- Minimum scope để learn
- Viable = users can use & provide feedback

Example:
Product: Task management app

MVP v1:

- Create task
- Mark done
- List tasks

NOT in MVP:

- Tags, filters, search
- Collaboration
- Reminders, due dates
- Mobile app

→ Ship v1 → learn → iterate

Iterate:

- Ship → measure → learn → improve → ship
- Continuous cycle

Câu hỏi tự kiểm tra

1. PM yêu cầu: "Build a dashboard with 20 charts showing all metrics."
   Bạn nghĩ gì? Câu hỏi gì để clarify?
2. Feature mới launch: Sign-ups tăng 50%, nhưng retention giảm 20%. Interpret?
3. Bạn có 2 features trong backlog:
   - Feature A: User requested nhiều, effort = 3 weeks
   - Feature B: Không ai request, nhưng improve performance 50%, effort = 1 week

   Prioritize thế nào?

PHẦN VIII: META-SKILLS & LỘ TRÌNH

Chương 13: Kỹ năng học & Cập nhật liên tục

13.1. Học như thế nào để "không lỗi thời"?

Mindset:

- Technology thay đổi nhanh
- Học continuous, không phải one-time
- T-shaped: sâu 1-2 mảng, rộng nhiều mảng

Learning loop:

1. Learn (course, doc, video)
2. Build (project, apply vào work)
3. Teach (blog, talk, mentor)
   → Repeat

Resources:

- Official docs (best source)
- Books (depth)
- Courses (structured)
- Blogs, newsletters (trends)

- Conferences, meetups (network, learn)

13.2. Giao tiếp & Collaboration

Writing:

- Tech specs, RFCs (Request for Comments)
- PR descriptions (why + what + how)
- Documentation (README, API docs, runbooks)

Presenting:

- Explain technical concepts to non-tech
- Demo features
- Postmortem presentations

Code review:

- Constructive feedback
- Praise good code
- Ask questions (không assume)

Meetings:

- Come prepared
- Listen actively
- Contribute ideas
- Follow up with action items

13.3. Lộ trình 12-24 tháng

Months 1-3: Nền tảng

- DSA basics (100-150 problems)
- System design fundamentals (đọc 1-2 books)
- Refactor 1 project cũ

Months 4-6: Cloud & DevOps

- Chọn 1 cloud (AWS/GCP/Azure)
- Deploy 1 app production
- Setup CI/CD pipeline
- Learn Docker, basic Kubernetes

Months 7-9: Observability & Security

- Add logging, metrics, tracing
- OWASP Top 10
- Implement auth/authz properly

Months 10-12: Data & AI

- Build 1 feature with LLM
- Learn RAG basics
- Understand data pipelines

Months 13-18: Chuyên sâu
Chọn 1 track:

- Cloud/DevOps/SRE: Kubernetes advanced, Terraform, observability
- Security: DevSecOps, penetration testing, compliance
- Data/AI: ML pipelines, MLOps, advanced RAG

Months 19-24: Leadership & Product

- Mentoring
- Lead 1 project end-to-end
- Product sense (hiểu metrics, prioritization)
- Start writing/speaking

KẾT LUẬN

AI không giết nghề dev. AI giết cách làm dev kiểu cũ.

Developer kiểu cũ:

- Chỉ biết gõ code theo spec
- Không hiểu hệ thống
- Không hiểu business
- Không học liên tục

Developer "khó bị thay thế":

- Nền tảng kỹ thuật vững (CS, system design, cloud, security, data/AI)

- Tư duy hệ thống (end-to-end, impact analysis)
- Kỹ năng con người (giao tiếp, collaboration, leadership)
- Tư duy chiến lược (career plan, product sense, market awareness)

Con đường:

1. Củng cố nền tảng
2. Chọn T-shape (sâu 1-2 mảng, rộng nhiều mảng)
3. Build projects thực tế (không học chay)
4. Học liên tục (technology thay đổi nhanh)
5. Share & teach (blog, talks, mentoring)

Thời gian:
12-24 tháng dedicated learning → transform từ "có thể bị thay thế" sang "khó bị thay thế"

Hành động tiếp theo:

1. Đánh giá bản thân: 4 lớp kỹ năng (technical, systems, human, strategic) - bạn ở đâu?
2. Chọn 1 mảng để improve trong 3 tháng tới
3. Tìm 1 project thực tế để apply
4. Start today

Remember:

> "The best time to plant a tree was 20 years ago. The second best time is now."

Bắt đầu hành trình của bạn ngay hôm nay.

PHỤ LỤC

A. Tài liệu tham khảo

Books:

- "Designing Data-Intensive Applications" - Martin Kleppmann (system design)
- "System Design Interview" - Alex Xu (interview prep)
- "The Phoenix Project" - Gene Kim (DevOps novel)

- "Site Reliability Engineering" - Google (SRE practices)
- "Clean Architecture" - Robert Martin (software design)

Websites:

- Official cloud docs (AWS, GCP, Azure)
- System Design Primer (GitHub)
- LeetCode, HackerRank (DSA practice)
- OWASP (security)

Courses:

- System Design (Grokking the System Design Interview)
- Cloud (A Cloud Guru, Linux Academy)
- Kubernetes (CKAD, CKA)

B. Checklist tự đánh giá

Nền tảng kỹ thuật:

- [ ] DSA: Giải được medium problems trong 30 phút
- [ ] System design: Thiết kế được hệ thống scalable trong 45 phút
- [ ] Cloud: Deploy được app production
- [ ] DevOps: Setup được CI/CD pipeline
- [ ] Security: Biết OWASP Top 10, implement auth/authz
- [ ] Data/AI: Build được feature với LLM + RAG

Tư duy hệ thống:

- [ ] Hiểu end-to-end flow (user → API → DB → infra)
- [ ] Dự đoán được impact của thay đổi
- [ ] Debug được issue production

Kỹ năng con người:

- [ ] Viết được tech spec rõ ràng
- [ ] Present được technical concept cho non-tech
- [ ] Code review xây dựng
- [ ] Mentoring được junior

Tư duy chiến lược:

- [ ] Có career plan 3-5 năm

- [ ] Hiểu business metrics
- [ ] Contribute product decisions
- [ ] Follow trends, học liên tục

C. Template lộ trình cá nhân hóa

Current state:

- Experience: __ years
- Strong areas: _____
- Weak areas: _____
- Current role: _____

Goal (12 months):

- Target role: _____
- Key skills needed: _____

Action plan:

- Month 1-3: _____ (focus area)
  - Project: _____
  - Learning: _____
- Month 4-6: _____
- Month 7-9: _____
- Month 10-12: _____

Checkpoints:

- Monthly: review progress
- Quarterly: adjust plan
- Yearly: evaluate goal achievement

TÁC GIẢ & NGƯỜI ĐÓNG GÓP

Tài liệu này được biên soạn dựa trên:

- Nghiên cứu từ WEF, McKinsey, LinkedIn về tương lai việc làm
- Best practices từ các tổ chức công nghệ hàng đầu
- Kinh nghiệm thực tế từ cộng đồng developer

Version: 1.0 (February 2026)

License: Creative Commons Attribution 4.0 International

Feedback & Contributions:
Mọi góp ý, bổ sung xin gửi qua [contact method]

Chúc bạn thành công trên hành trình trở thành developer "khó bị thay thế"!