ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC BÁCH KHOA
KHOA KHOA HỌC VÀ KĨ THUẬT MÁY TÍNH

BÁO CÁO BÀI TẬP LỚN

MÔN HỌC: KIẾN TRÚC PHẦN MỀM (CO3017)

Intelligent Tutoring System (ITS)

Giảng viên hướng dẫn: Trần Trương Tuấn Phát
Nhóm thực hiện: L02 - Nhóm 06

STT	MSSV	Họ và tên
1	2210020	Nguyễn Phạm Quốc An
2	2210487	Nguyễn Thùy Dung
3	2210714	Phạm Tuấn Đạt
4	2211583	Bùi Vũ Anh Khoa
5	2333435	Trần Phi Long
6	2213301	Nguyễn Văn Thịnh
7	2213309	Trần Chấn Thịnh

TP. Hồ Chí Minh, Tháng 11/2025

Mục lục

1 Danh sách thành viên và phân công **2**

2 Lịch sử cập nhật **3**

3 Thu thập yêu cầu **4**
3.1 Tổng quan về đề tài 4
3.2 Yêu cầu chức năng và phi chức năng 4

4 Thiết kế kiến trúc **6**
4.1 Đặc điểm kiến trúc 6
4.2 So sánh và lựa chọn architecture styles 6
4.3 Structure: các views của kiến trúc 7
4.4 Architecture decisions (Quyết định kiến trúc) 10
4.5 Design principles 11
4.6 Ví dụ interface (pseudo-code) minh họa áp dụng SOLID 11
4.7 Mapping các yêu cầu phi chức năng vào thiết kế 12

5 Áp dụng nguyên tắc SOLID trong hệ thống ITS **13**
5.1 Nguyên tắc Đơn trách nhiệm (SRP) 13
5.2 Nguyên tắc Mở - Đóng (OCP) 14
5.3 Nguyên tắc Thay Thế Liskov (LSP) 16
5.4 Nguyên tắc Phân tách Interface (ISP) 17
5.5 Nguyên tắc Đảo ngược phụ thuộc (DIP) 19

6 Phản hồi về việc áp dụng các nguyên tắc SOLID **20**
6.1 Giải quyết vấn đề bằng ứng dụng SOLID 20
6.2 Thách thức trong việc thiết kế và ứng dụng SOLID 21

7 Hiện thực sản phẩm **23**

8 Đề xuất mở rộng trong tương lai **24**

9 Tài liệu tham khảo **25**

1 Danh sách thành viên và phân công

STT	Họ tên sinh viên	MSSV	Nhiệm vụ
1	Nguyễn Phạm Quốc An	2210020	1. Thiết kế kiến trúc hệ thống
2	Nguyễn Thùy Dung	2210487	1. Phân tích yêu cầu chức năng và phi chức năng 2. Thiết kế kiến trúc hệ thống
3	Phạm Tuấn Đạt	2210714	1. Phân tích nguyên tắc SOLID trong thiết kế kiến trúc
4	Bùi Vũ Anh Khoa	2211583	1. Thiết kế kiến trúc hệ thống
5	Trần Phi Long	2333435	1. Phân tích nguyên tắc SOLID trong thiết kế kiến trúc
6	Nguyễn Văn Thịnh	2213301	1. Phản hồi về việc áp dụng các nguyên tắc SOLID trong thiết kế hệ thống
7	Trần Chấn Thịnh	2213309	1. Phản hồi về việc áp dụng các nguyên tắc SOLID trong thiết kế hệ thống ITS

2 Lịch sử cập nhật

Verson	Tên phiên bản	Ngày cập nhật	Sửa đổi
1.0.0	Version 1	08/10/2025	Đặc tả hệ thống
1.0.1	Version 1	11/10/2025	Thiết kế kiến trúc hệ thống
1.0.2	Version 1	14/10/2025	Áp dụng nguyên tắc solid trong thiết kế kiến trúc hệ thống
1.0.3	Version 1	17/10/2025	Phản hồi về việc áp dụng các nguyên tắc SOLID trong thiết kế hệ thống

3 Thu thập yêu cầu

3.1 Tổng quan về đề tài

3.1.1 Bối cảnh

Trong bối cảnh chuyển đổi số diễn ra mạnh mẽ, việc ứng dụng trí tuệ nhân tạo (AI) vào giáo dục đã trở thành xu thế tất yếu, góp phần làm thay đổi căn bản phương thức dạy và học truyền thống. Nếu như trước đây, quá trình học tập thường được thiết kế theo một lộ trình cố định, chưa phản ánh đúng năng lực, tốc độ tiếp thu và nhu cầu riêng của từng người học, thì ngày nay, AI mở ra khả năng cá nhân hóa giáo dục ở mức độ sâu hơn. Điều này giúp khắc phục tình trạng học sinh yếu bị "bỏ lại phía sau", trong khi học sinh giỏi lại thiếu môi trường được thử thách và phát triển toàn diện.

Từ thực tiễn đó, chúng em hướng tới việc xây dựng **Hệ thống học tập thông minh (Intelligent Tutoring System – ITS)**, một giải pháp giáo dục ứng dụng AI đóng vai trò như "gia sư ảo". Hệ thống này có khả năng hỗ trợ người học trong quá trình tự học, đồng thời cung cấp công cụ phân tích, theo dõi và đánh giá hiệu quả học tập cho giáo viên cũng như nhà quản lý giáo dục. ITS không chỉ góp phần nâng cao chất lượng học tập cá nhân mà còn mở ra hướng đi mới cho nền giáo dục thông minh trong kỷ nguyên số.

Hệ thống ITS mang đến một cách tiếp cận hiện đại, cho phép cá nhân hóa trải nghiệm học tập dựa trên năng lực, hành vi và nhu cầu cụ thể của từng người học. Nhờ khả năng đánh giá trình độ, đưa ra phản hồi tức thời, gợi ý nội dung phù hợp và theo dõi tiến trình học tập theo thời gian thực, ITS giúp quá trình học trở nên hiệu quả, linh hoạt và mang tính tương tác cao, đáp ứng yêu cầu đổi mới giáo dục trong thời đại trí tuệ nhân tạo.

3.1.2 Các bên liên quan và lợi ích của ITS

- **Người học (Học sinh / Sinh viên):**
  - Được hướng dẫn học tập phù hợp với năng lực cá nhân.
  - Nhận phản hồi và gợi ý chi tiết sau mỗi bài kiểm tra.
  - Có thể theo dõi tiến trình học tập, nhận báo cáo năng lực và đề xuất lộ trình học tiếp theo.
- **Quản trị hệ thống:**
  - Dễ dàng theo dõi hiệu quả học tập của học sinh thông qua bảng điều khiển (dashboard).
  - Phát hiện sớm những học sinh gặp khó khăn để có biện pháp hỗ trợ kịp thời.
  - Tự động hóa việc ra đề, chấm bài và thống kê kết quả.
  - Dễ dàng bảo trì, mở rộng và nâng cấp hệ thống nhờ kiến trúc module hoá.
  - Có thể tích hợp các thuật toán trí tuệ nhân tạo mới (ví dụ: recommendation, natural language understanding).

3.2 Yêu cầu chức năng và phi chức năng

3.2.1 Yêu cầu chức năng

1. Người học (Student)

- Đăng ký tài khoản, đăng nhập và quản lý thông tin cá nhân (họ tên, email, mật khẩu, ảnh đại diện, v.v.).
- Tham gia các bài học (video, quiz, exercise) theo từng môn học, chủ đề và mức độ khó khác nhau.
- Được chấm điểm tự động, cung cấp phản hồi tức thời cho từng câu hỏi, bao gồm điểm số đáp án đúng, lời giải thích chi tiết và gợi ý giúp người học hiểu rõ hơn về kiến thức sai.
- Được đề xuất lộ trình học tập cá nhân hóa (personalized learning path) dựa trên điểm số, tiến độ và mức độ thành thạo từng chủ đề của người học.

- Nhận thông báo (notification) về kết quả học tập, lời nhắc ôn tập hoặc các khóa học/bài học được đề xuất.
- Xuất báo cáo kết quả học tập cá nhân dưới dạng PDF hoặc Excel để lưu trữ hoặc chia sẻ.

2. Quản trị viên hệ thống (Admin)

- Đăng ký tài khoản, đăng nhập và quản lý thông tin cá nhân (họ tên, email, mật khẩu, ảnh đại diện, v.v.).
- Tạo mới, chỉnh sửa, hoặc xóa nội dung học tập như danh mục môn học, chủ đề, câu hỏi, bài học, video, bài tập, bài kiểm tra.
- Cấu hình các tham số hệ thống (ví dụ: số lượng câu hỏi mỗi bài kiểm tra, giới hạn thời gian làm bài, chính sách chấm điểm).
- Thống kê kết quả học tập của từng học viên thông qua bảng điều khiển (dashboard).
- Xuất báo cáo tổng hợp kết quả học tập của lớp (theo chủ đề, thời gian), hoặc học viên cụ thể) dưới định dạng PDF/Excel.
- Quản lý tài khoản người dùng (kích hoạt, khóa, đặt lại mật khẩu, phân quyền).
- Theo dõi hoạt động hệ thống, thống kê truy cập và giám sát hiệu năng để đảm bảo tính ổn định.

3.2.2 Yêu cầu phi chức năng

Mã	Yêu cầu	Mô tả
NFR1	Tính bảo mật	- Thông tin người dùng được mã hoá và xác thực qua cơ chế bảo mật hiện đại (JWT). - Hệ thống đảm bảo tuân thủ các quy định về an toàn dữ liệu và quyền riêng tư.
NFR2	Tính sẵn sàng	- Hệ thống phải luôn sẵn sàng 24/7 để đáp ứng nhu cầu học tập của sinh viên.
NFR3	Tính khả dụng	- Giao diện thân thiện, dễ sử dụng. - Hệ thống phải phản hồi yêu cầu trong thời gian dưới 500 ms cho các thao tác thông thường như đăng nhập, tải nội dung, hoặc chấm bài. - Hệ thống phải đảm bảo hoạt động liên tục, có cơ chế sao lưu dữ liệu định kỳ và phục hồi nhanh sau sự cố.
NFR4	Tính tin cậy	- Hệ thống có khả năng xử lý đồng thời ít nhất 100 yêu cầu người dùng và lưu trữ khối lượng dữ liệu lớn (từ hàng trăm nghìn bản ghi).
NFR5	Khả năng kiểm thử	- Các module có thể được kiểm thử độc lập (unit test) và tích hợp (integration test) nhằm đảm bảo tính ổn định của hệ thống.
NFR6	Khả năng mở rộng và bảo trì	- Mã nguồn được tổ chức theo nguyên tắc SOLID, có tài liệu và hướng dẫn triển khai chi tiết, giúp việc mở rộng và sửa lỗi trở nên dễ dàng. - Kiến trúc hệ thống được thiết kế theo mô hình module hoá, cho phép dễ dàng bổ sung tính năng hoặc mở rộng quy mô người dùng mà không ảnh hưởng tới các phần khác.

4 Thiết kế kiến trúc

4.1 Đặc điểm kiến trúc

- **Tính đúng đắn chức năng (Correctness)**: hệ thống thực hiện đầy đủ các yêu cầu chức năng (FR) đã thu thập; các luồng chính (core flows) (làm bài, chấm bài, phản hồi, gợi ý) phải chính xác theo yêu cầu.
- **Hiệu năng (Performance)**: Thời gian phản hồi của API cho các thao tác chính (lấy câu hỏi, nộp bài, lấy phản hồi) dưới **500 ms** trong môi trường demo; hệ thống có thể xử lý hơn 100 yêu cầu ban đầu và có khả năng mở rộng lên hàng nghìn yêu cầu đồng thời khi cần.
- **Khả năng mở rộng (Scalability)**: kiến trúc cho phép tách module hoặc mở rộng từng thành phần độc lập (API, workers, DB replicas).
- **Khả dụng (Availability) & Tin cậy (Reliability)**: hệ thống hoạt động liên tục, có cơ chế backup/restore; thời gian khôi phục (RTO) được tối thiểu hóa.
- **An toàn bảo mật (Security)**: xác thực/ủy quyền (JWT/OAuth2), mã hóa mật khẩu, kiểm soát truy cập theo vai trò, và chống lại các tấn công phổ biến.
- **Khả năng kiểm thử (Testability)**: các lớp logic phải có interface rõ ràng để viết unit/integration tests; mục tiêu đạt 70% coverage cho các module chính.
- **Khả năng bảo trì (Maintainability)**: mã mô-đun, tuân thủ SOLID, dễ đọc, có CI/CD, tài liệu API.
- **Khả năng mở rộng thuật toán (Extensibility)**: dễ thay thế hoặc thêm chiến lược chấm điểm, sinh hint, và recommendation mà không sửa core code (Open/Closed).
- **Quan sát (Observability)**: logs, metrics, tracing (ELK/Prometheus/Jaeger) để theo dõi hoạt động và gỡ lỗi.

4.2 So sánh và lựa chọn architecture styles

4.2.1 Các architecture styles

Layered Monolith (3-tier) Kiến trúc truyền thống: Presentation – Business – Data.
Ưu: đơn giản, dễ triển khai.
Nhược: khó scale theo module, dễ bị coupling.

Modular Monolith + Hexagonal (Ports & Adapters) Module hóa trong một repo; domain logic tách qua ports (interfaces) và adapters (DB, cache).
Ưu: dễ phát triển + test, phù hợp SOLID, dễ migrate sang microservices.
Nhược: cần discipline khi thiết kế interface.

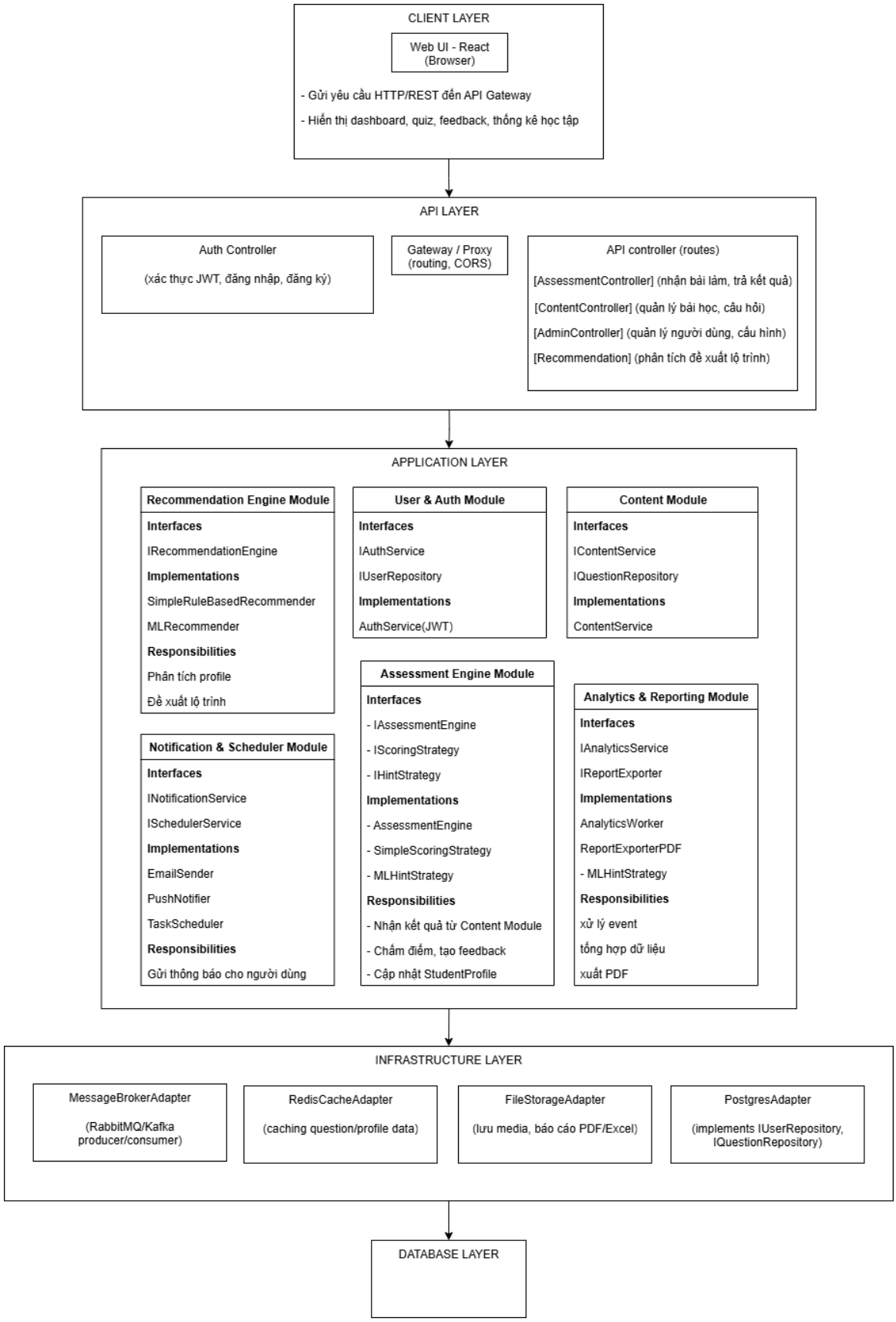
Microservices Tách thành các dịch vụ độc lập (Assessment, Content, Auth, Analytics, Recommendation...).
Ưu: scale độc lập, deploy độc lập.
Nhược: chi phí vận hành cao, phức tạp (distributed transactions, observability).

4.2.2 Lựa chọn architecture styles cho ITS

Nhóm chọn Modular Monolith với Ports & Adapters (Hexagonal) + Event-driven cho các tác vụ bất đồng bộ.

- Dễ implement, giải thích, demo; vẫn đáp ứng SOLID và testability.
- Ports & Adapters cho phép tách rõ domain (Assessment, Recommendation) khỏi infra (Postgres, Redis, S3) — tuân DIP và LSP.
- Dùng event/message broker (RabbitMQ/Kafka) để tách processing bất đồng bộ (Analytics, Notification), giúp đảm bảo responsiveness cho API.
- Dễ migrate sang microservices khi bài toán mở rộng.

4.3 Structure: các views của kiến trúc



4.3.1 Module view

4.3.1.1 Module 1 — User & Authentication

- **Mục đích:** quản lý người dùng, xác thực, ủy quyền (RBAC).
- **Trách nhiệm:** register/login, issue/verify JWT, quản lý profile, reset password.
- **Interfaces:**

```
IUserRepository { get(user_id), get_by_email(email), save(user) }
IAuthService { register(dto), login(email,pwd), verify(token) }
```

- **Persistence:** table `users`, `roles`; optional MFA store.
- **NFR concerns:** security, audit logs, token expiry.

4.3.1.2 Module 2 — Content Management

- **Mục đích:** CRUD courses, topics, lessons, questions, media metadata.
- **Trách nhiệm:** versioning content, validation, permission checks.
- **Interfaces:**

```
IContentService { create_course(), add_question(), get_questions(filter) }
IQuestionRepository { list(filter), get(id), save(q) }
```

- **Persistence:** `questions`, `lessons`, media metadata; media: object storage (S3).
- **NFR concerns:** maintainability, storage availability.

4.3.1.3 Module 3 — Assessment Engine (core)

- **Mục đích:** xử lý nộp bài, chấm điểm, sinh feedback/hints, cập nhật profile người học.
- **Trách nhiệm:** validate answers, use `IScoringStrategy` và `IHintStrateg y`, persist attempts, emit events.
- **Interfaces:**

```
IAssessmentEngine { assess(student_id, answers) -> AssessmentResult }
IScoringStrategy { score(question, selected) -> float }
IHintStrategy { generate(question, selected, profile) -> hint }
```

- **Persistence:** `attempts`, `attempt_items`, update `student_profiles`.
- **NFR concerns:** testability (mock strategies), low latency for realtime feedback.

4.3.1.4 Module 4 — Recommendation Engine

- **Mục đích:** gợi ý lộ trình học cá nhân hóa dựa trên profile, lịch sử.
- **Trách nhiệm:** rule-based hoặc model-based recommenders; caching của đề xuất.
- **Interfaces:**

```
IRecommendationEngine { recommend(profile, context) -> [Resource] }
```

- **Persistence / infra:** cache (Redis) cho kết quả; model store (file/model DB) nếu ML.
- **NFR concerns:** correctness, latency (cache), explainability.

4.3.1.5 Module 5 — Analytics & Reporting

- **Mục đích:** tổng hợp metrics, dashboard, export PDF/Excel.
- **Trách nhiệm:** consume events (student.attempt.completed), aggregate, expose report APIs, schedule batch jobs.
- **Interfaces:**

```
IAnalyticsService { aggregate(metric), export_report(params) }
```

- **Persistence:** analytics DB (materialized views), S3 exports.
- **NFR concerns:** scalability (batch), eventual consistency (analytics).

4.3.1.6 Module 6 — Notification & Scheduler (worker)

- **Mục đích:** gửi email/push, schedule tasks (reminder, backup).
- **Trách nhiệm:** subscribe events, queue messages to notifier providers, retries and DLQ.
- **NFR concerns:** reliability (retry/backoff), idempotency.

4.3.1.7 Module 7 — Persistence & Infrastructure Adapters

- **Mục đích:** adapters cho Postgres, Redis, S3, Message Broker.
- **Trách nhiệm:** implement repository interfaces, handle transactions, migrations.

4.3.2 Component-and-Connector (C&C) view

Mô tả thành phần và connector (giao tiếp) chính:

- **Clients (Browser/Mobile)** $\xrightarrow{\text{HTTPS/REST}}$ **API Gateway / API Server** (Auth middleware, routing).
- **API Server** $\xrightarrow{\text{in-process calls / DI}}$ **Application Modules** (AssessmentEngine, ContentService, ...).
- **AssessmentEngine** $\xrightarrow{\text{IQuestionRepo}}$ **QuestionRepository Adapter** $\xrightarrow{\text{SQL}}$ **Postgres**.
- **AssessmentEngine** $\xrightarrow{\text{publish event}}$ **Message Broker** (RabbitMQ/Kafka) $\rightarrow$ **Analytics Worker**, **Notification Worker**.
- **RecommendationEngine** may call ML Model Service (gRPC/REST) or use local model files.
- **Cache (Redis)** sits between Application Modules and DB for hot data (questions, profiles, recommendations).

4.3.2.1 Sequence flow (ví dụ: Student làm quiz)

1. Client gửi POST /assessments với answers + JWT.
2. API Gateway kiểm tra token → forward request tới AssessmentController.
3. AssessmentController gọi `AssessmentEngine.assess(studentId, answers)`.
4. AssessmentEngine fetch questions via `IQuestionRepository` (Redis cache → Postgres).
5. AssessmentEngine sử dụng `IScoringStrategy` để tính điểm, `IHintStrategy` để tạo hint cho câu sai.
6. Lưu attempt vào DB (attempts, attempt_items); cập nhật student profile.
7. AssessmentEngine publish event `student.attempt.completed` lên Message Broker.
8. Workers (Analytics/Notification) consume event và cập nhật bảng tổng hợp, gửi thông báo nếu cần.
9. AssessmentController gọi RecommendationEngine để trả về personalized path.
10. API trả response (score, per-question feedback, recommended next steps) cho client.

4.3.3 Allocation view

- **Client (React)**: static assets served by CDN.
- **API Server(s)** (FastAPI / Express / Spring Boot): container, autoscaled behind load balancer.
- **Assessment Engine** (in same API process hoặc separate worker as needed).
- **Workers** (Analytics Worker, Notification Worker): container(s) subscribe to broker.
- **Message Broker**: RabbitMQ/Kafka cluster.
- **DB**: Postgres primary + read replicas; backups to object storage.
- **Cache**: Redis cluster for sessions, hot reads.
- **Object Storage**: S3 for media and exported reports.
- **Monitoring Stack**: Prometheus + Grafana, ELK for logs, Jaeger for tracing.

4.4 Architecture decisions (Quyết định kiến trúc)

Liệt kê các quyết định chính (AD = Architecture Decision):

1. **AD1: Modular Monolith + Ports & Adapters** — bắt đầu bằng modular monolith để giảm độ phức tạp, nhưng tất cả domain logic phải phụ thuộc trên interfaces để dễ tách service khi cần.
2. **AD2: REST/JSON cho public API; internal RPC optional (gRPC)** — dễ debug, client-friendly; gRPC chỉ dùng khi latency/throughput yêu cầu cao.
3. **AD3: JWT cho authentication; RBAC cho authorization** — đơn giản cho demo, hỗ trợ token expiry & refresh.
4. **AD4: Postgres cho persistent store; Redis cho cache; S3 cho media** — cân bằng consistency & performance.
5. **AD5: Message Broker (RabbitMQ/Kafka) cho asynchrony** — tách analytics/notification, hỗ trợ retry/DLQ.
6. **AD6: Dependency Injection (DI) container** — inject interfaces (IQuestionRepository, IScoringStrategy) để test & thay implementation.

7. **AD7: Strategy pattern cho scoring/hint/recommendation** — tuân OCP: thêm thuật toán mới bằng cách implement interface.
8. **AD8: Observability required** — mọi service emit metrics & structured logs.
9. **AD9: CI/CD pipeline + automated tests** — commit → unit tests → integration tests → build image → deploy (staging → prod).
10. **AD10: Data privacy** — PII encrypted-at-rest and in-transit; data retention policies.

4.5 Design principles

4.5.1 Nguyên tắc cốt lõi:

- **SRP (Single Responsibility)**: mỗi class/module có một lý do duy nhất để thay đổi. Ví dụ: `AssessmentEngine` chỉ chấm + update profile; `NotificationService` chỉ gửi thông báo.
- **OCP (Open/Closed)**: mở rộng bằng cách thêm `IScoringStrategy`, `IHintStrategy`, `IRecommendationEngine` mà không sửa code core.
- **LSP (Liskov)**: mọi implementation của interface phải có hành vi thay thế được (ví dụ InMemoryRepo vs PostgresRepo).
- **ISP (Interface Segregation)**: tách interface lớn thành nhiều interface nhỏ (IQuestionReader, IQuestion-Writer).
- **DIP (Dependency Inversion)**: các module cấp cao phụ thuộc các abstractions (interfaces) — sử dụng DI container để bind interface → impl.

4.5.2 Patterns & practices

- **Repository Pattern** cho access layer; **Service layer** cho business rules; **Strategy Pattern** cho chấm điểm/hint/recommendation.
- **Ports & Adapters** để tách domain khỏi infra.
- **Event-driven** cho các tác vụ bất đồng bộ (analytics, notifications).
- **Idempotency** cho endpoints/worker handlers (đảm bảo xử lý lại không gây lỗi).
- **Caching strategy**: read-through cache cho câu hỏi; cache invalidation policy on content update.
- **Security best practices**: HTTPS everywhere, input validation, parameterized queries, rate limiting, audit logs.
- **Testing strategy**: unit tests per service/strategy; integration tests with test DB; e2e tests for main flows.
- **Observability**: structured logs + correlation IDs; metrics for latency/error rates; tracing for cross-service calls.

4.6 Ví dụ interface (pseudo-code) minh họa áp dụng SOLID

```
# Repository interfaces
interface IQuestionRepository {
  List<Question> list(filter)
  Question getById(id)
  void save(Question q)
}

# Strategy interfaces
```

```
interface IScoringStrategy {
  float score(Question q, Choice selected)
}
interface IHintStrategy {
  String generate(Question q, Choice selected, StudentProfile p)
}

# Assessment engine (depends on abstractions)
class AssessmentEngine {
  constructor(IQuestionRepository qRepo,
              IScoringStrategy scoring,
              IHintStrategy hint,
              IStudentProfileRepository pRepo) { ... }

  AssessmentResult assess(studentId, Map<qid, choiceId> answers) { ... }
}
```

4.7 Mapping các yêu cầu phi chức năng vào thiết kế

- **Performance**: cache (Redis), read replicas, index DB.
- **Scalability**: containerize services, autoscale API & workers.
- **Security**: auth middleware, RBAC, encryption.
- **Testability & Maintainability**: DI + interfaces + unit tests.
- **Availability**: redundancy (replica DB), health checks, monitoring + alerts.

5 Áp dụng nguyên tắc SOLID trong hệ thống ITS

5.1 Nguyên tắc Đơn trách nhiệm (SRP)

5.1.1 Định nghĩa và khái niệm cốt lõi

Single Responsibility Principle (SRP) phát biểu rằng: Mỗi lớp (class) hoặc module trong hệ thống nên chỉ có một lý do duy nhất để thay đổi. Điều này có nghĩa là mỗi thành phần phần mềm nên tập trung vào một chức năng chính và chỉ thay đổi khi yêu cầu của chức năng đó thay đổi.

5.1.2 Tại sao SRP quan trọng trong ITS?

Hệ thống giáo dục thông minh xử lý rất nhiều quy trình phức tạp:

- **Đánh giá học sinh** (validation, scoring)
- **Phản hồi cá nhân** (hints, explanations)
- **Thông báo** (emails, push notifications)
- **Báo cáo học tập** (PDF exports, analytics)
- **Gợi ý học tập** (personalized recommendations)

Nếu tất cả những trách nhiệm này được đóng gói vào một module duy nhất, khi một yêu cầu kinh doanh thay đổi (ví dụ: format email thay đổi), cả hệ thống assessment cũng bị ảnh hưởng, dẫn đến rủi ro cao.

5.1.3 Vấn đề khi không áp dụng SRP

Giả sử có một module tên `AssessmentModule` đảm nhận toàn bộ quy trình: xác thực câu trả lời, tính điểm, gửi email thông báo kết quả, tạo báo cáo PDF, cập nhật analytics dashboard, và tạo gợi ý học tập.

Hậu quả khi không áp dụng SRP:

- **Độ phức tạp cao**: Module trở thành "god class" với hàng trăm dòng code
- **Khó test**: Phải mock tất cả dependencies (email, PDF, analytics, ML)
- **Rủi ro cao**: Một lỗi nhỏ trong notification logic có thể làm sập assessment logic
- **Khó bảo trì**: Nhà phát triển mới khó hiểu chính xác module làm cái gì
- **Không thể tái sử dụng**: Logic assessment bị "dính" với logic notification
- **Bottleneck phát triển**: Chỉ một người có thể sửa module này để tránh conflict

5.1.4 Giải Pháp: Phân tách theo trách nhiệm

Phân chia `AssessmentModule` thành **7 module độc lập**, mỗi module có một lý do duy nhất để thay đổi:

Module	Nhiệm vụ	Lý do thay đổi
Assessment Engine	Xác thực câu trả lời, tính điểm, tạo gợi ý, cập nhật hồ sơ học sinh.	Chỉ khi logic tính điểm hoặc validation thay đổi.
Notification Worker	Gửi email, push notification, SMS.	Chỉ khi template thông báo hoặc kênh giao tiếp thay đổi.
Analytics Worker	Tổng hợp metrics, cập nhật aggregated data.	Chỉ khi metrics hoặc dashboard logic thay đổi.
Report Service	Tạo báo cáo PDF/Excel.	Chỉ khi format báo cáo thay đổi.
Recommendation Engine	Gợi ý lộ trình học tập cá nhân hóa	- Chỉ khi algorithm recommendation thay đổi.
User & Auth	Xác thực, quản lý người dùng, RBAC.	Chỉ khi auth logic thay đổi.
Content Management	CRUD course, lesson, question.	Chỉ khi content structure thay đổi.

5.1.5 Lợi Ích Thực Tế Của SRP Trong ITS

Việc áp dụng SRP mang lại các lợi ích sau:

- **Dễ Test Độc Lập**: Assessment Engine có thể test mà không cần email service hay PDF exporter. Test chỉ cần mock Question Repository.
- **Dễ Định Vị Lỗi**: Nếu email không gửi được, lỗi chỉ trong Notification Worker. Nếu điểm sai, chỉ trong Assessment Engine.
- **Độ Tin Cậy Cao**: Nếu notification service down, Assessment Engine vẫn hoạt động bình thường.
- **Bảo Trì Dễ**: Email template thay đổi? Chỉ cần sửa Notification Worker.
- **Mở Rộng Nhanh**: Muốn thêm SMS notification? Thêm vào Notification Worker, không cần động Assessment Engine.
- **Phát Triển Song Song**: Team A làm Assessment, Team B làm Notification, Team C làm Analytics — không xung đột.
- **Performance Tốt**: Assessment Engine không bị chặn bởi email delay. Notification/Analytics chạy async.

5.1.6 Ví Dụ Minh Họa: Thay Đổi Yêu Cầu

Marketing team muốn thay đổi format email thông báo kết quả. Với SRP, chỉ cần thay đổi template trong Notification Worker, Assessment Engine code không bị touch. Risk rất thấp vì chỉ một module bị ảnh hưởng. Không có SRP sẽ phải mở AssessmentModule, tìm đoạn code xử lý email, sửa template email, risk cao vì AssessmentModule có rất nhiều logic.

5.2 Nguyên tắc Mở - Đóng (OCP)

5.2.1 Định Nghĩa và Khái Niệm

Open/Closed Principle (OCP) phát biểu rằng: Software entities (classes, modules, functions) nên **MỞ** cho việc mở rộng (extension) nhưng **ĐÓNG** cho việc sửa đổi (modification). Kỹ thuật chính để đạt OCP là sử dụng **abstraction** (interface/abstract class) kết hợp với **polymorphism**.

5.2.2 Tại Sao OCP Quan Trọng Trong ITS?

Hệ thống giáo dục cần linh hoạt trong việc chấm điểm vì:

- **Các kiểu bài tập khác nhau**: Trắc nghiệm, tự luận, thực hành lập trình

- **Chiến lược chấm khác nhau**: Nhị phân (đúng/sai), tính điểm từng phần, có trọng số
- **Nhu cầu thay đổi theo thời gian**: Hôm nay là binary, tương lai có thể là ML-based
- **Tổ chức khác nhau có quy tắc khác**: Trường A dùng 100% weighted scoring, Trường B dùng partial credit

Nếu code phải sửa mỗi lần thêm chiến lược chấm mới, hệ thống sẽ không linh hoạt.

5.2.3 Vấn Đề Khi Không Áp Dụng OCP

Giả sử hệ thống hiện tại chỉ support **binary scoring** (đúng=1 điểm, sai=0 điểm). Nếu code được viết không tuân OCP, mỗi lần thêm chiến lược chấm mới (partial credit scoring, weighted scoring, ML-based scoring), AssessmentEngine phải được sửa với thêm các IF/ELSE branches.

Hậu quả:

- **Code Phức Tạp**: AssessmentEngine có quá nhiều IF/ELSE, khó đọc, khó maintain
- **Khó Test**: Mỗi lần thêm strategy, phải viết test cho tất cả branch
- **Dễ Introduce Bug**: Mỗi lần sửa, dễ break các case cũ
- **Không Scalable**: Tưởng tượng có 20 loại strategy, AssessmentEngine sẽ là "spaghetti code"
- **Runtime Error Risk**: Nếu typo strategy name, chỉ phát hiện lúc runtime

5.2.4 Giải Pháp: Áp Dụng OCP Với Abstraction

Thay vì hardcode các chiến lược, ta tạo một **interface abstraction** cho chiến lược chấm điểm: `IScoringStrategy`.

Các múi triển khai khác nhau:

- **SimpleBinaryScoring**: Đúng=1.0, Sai=0.0
- **PartialCreditScoring**: Đúng=1.0, Bán đúng=0.5, Sai=0.0
- **WeightedDifficultyScoring**: Score $\times$ difficulty_weight
- **AdaptiveMLScoring**: ML model predict xác suất
- **TimeBasedScoring**: Tính theo thời gian (NEW - không sửa core!)

Mỗi implementation là một class riêng biệt, không xen vào nhau. AssessmentEngine không tạo strategy cụ thể, mà nhận nó từ bên ngoài (Dependency Injection). Code AssessmentEngine không bao giờ cần sửa dù có thêm bao nhiêu strategy mới.

5.2.5 Lợi Ích Thực Tế Của OCP Trong ITS

- **Thêm Feature Không Sửa Core**: Muốn thêm TimeBasedScoring? Tạo class mới implement IScoringStrategy, AssessmentEngine không sửa.
- **Code Ổn Định**: AssessmentEngine code không bao giờ cần sửa.
- **Dễ Test**: Test AssessmentEngine một lần, rồi test từng strategy riêng.
- **Runtime Flexibility**: Config file chỉ định dùng strategy nào, không cần compile lại.
- **Plugin Pattern**: Người khác có thể viết custom strategy mà không cần source code.
- **Giảm Regression Risk**: Thêm feature mới, không cần test lại toàn bộ hệ thống.

5.3 Nguyên tắc Thay Thế Liskov (LSP)

5.3.1 Định Nghĩa và Khái Niệm

Liskov Substitution Principle (LSP) phát biểu rằng: Các object của subclass (implementation) nên có thể thay thế object của superclass (interface) mà không làm thay đổi tính chính xác của program. Nói cách khác, nếu class B là subtype của class A, thì các instance của B có thể thay thế các instance của A mà không gây lỗi.

5.3.2 Tại Sao LSP Quan Trọng?

LSP đảm bảo rằng polymorphism hoạt động đúng. Khi ta dùng interface (ví dụ `IScoringStrategy`), ta dựa vào một hợp đồng (contract). Nếu implementation không tuân hợp đồng, polymorphism sẽ bị phá vỡ.

5.3.3 Vấn Đề Khi Không Áp Dụng LSP

Interface `IScoringStrategy` có hợp đồng: `score(question, selected_choice)` $\rightarrow$ float (0.0 to 1.0). Tuy nhiên, một implementation sai lệch **BuggyWeightedScoring** có thể:

- **Return value ngoài range**: Nếu difficulty_weight = 2.0, return 2.0 (vượt qua 1.0!) $\rightarrow$ Total score bị sai
- **Side effect không mong muốn**: Hàm ghi log vào database mỗi lần gọi (slow!) $\rightarrow$ Assessment chậm hơn mong đợi
- **Throw exception bất thường**: Nếu question = null, throw exception $\rightarrow$ Assessment đột ngột fail
- **Logic sai lệch**: Hàm trả về 0.0 khi selected = correct (đảo ngược logic!) $\rightarrow$ Student có câu đúng nhưng nhận 0 điểm

Hậu quả: AssessmentEngine không thể tin cậy rằng strategy nào implement sẽ tuân hợp đồng. Phải thêm validation/defensive checks ở AssessmentEngine $\rightarrow$ code phức tạp. Polymorphism bị phá vỡ, phải dùng `instanceof` checks.

5.3.4 Giải Pháp: Design Contract Rõ Ràng

Hợp đồng của `IScoringStrategy` cần được định nghĩa rõ ràng:

Preconditions (điều kiện trước):

- `question` $\neq$ `null`
- `selected_choice` $\neq$ `null`

Postconditions (điều kiện sau):

- Return value $\in$ [0.0, 1.0]
- Không có side effects
- Pure function (output chỉ phụ thuộc input)

Invariants (bất biến):

- Thứ tự gọi không ảnh hưởng kết quả
- Kết quả deterministic

Các implementations tuân LSP:

- **SimpleBinaryScoring**: If selected == correct → 1.0 else 0.0. Return ∈ [0.0, 1.0], No side effects.
- **PartialCreditScoring**: If correct → 1.0, partial → 0.5, else → 0.0. Return ∈ [0.0, 1.0].
- **WeightedDifficultyScoring**: (correct ? 1.0 : 0.0) × difficulty_weight. Normalize if needed: return / max_weight để đảm bảo ∈ [0.0, 1.0].

5.3.5 Lợi Ích Thực Tế Của LSP Trong ITS

- **Polymorphism Hoạt Động Đúng**: Mỗi strategy có thể thay thế nhau mà không break logic.
- **Giảm Defensive Code**: AssessmentEngine không cần `if strategy is X then...` checks.
- **Dễ Debug**: Nếu bug, biết ngay là do strategy implementation, không phải AssessmentEngine.
- **Design Contract Rõ Ràng**: Người viết strategy mới biết chính xác phải tuân hợp đồng gì.
- **Scale Tốt**: Thêm 100 strategies, AssessmentEngine vẫn hoạt động bình thường.

5.4 Nguyên tắc Phân tách Interface (ISP)

5.4.1 Định Nghĩa và Khái Niệm

Nguyên tắc ISP phát biểu rằng: "Không nên ép buộc các lớp hoặc module phải phụ thuộc vào những interface mà chúng không sử dụng".

Mục tiêu của nguyên tắc này là đảm bảo rằng mỗi interface chỉ chứa các hành vi liên quan trực tiếp đến trách nhiệm của module, tránh việc tạo ra các "fat interface" chứa quá nhiều chức năng không liên quan. Trong kiến trúc phần mềm, ISP giúp giảm độ kết dính giữa các thành phần và tăng tính linh hoạt khi mở rộng hoặc thay thế chức năng.

5.4.2 Tại sao ISP lại quan trọng trong ITS

Hệ thống Intelligent Tutoring System (ITS) bao gồm nhiều module nghiệp vụ độc lập như:

- User & Authentication
- Content Management
- Assessment Engine
- Recommendation Engine
- Analytics & Reporting
- Notification & Scheduler

Mỗi module chỉ đảm nhiệm một nhóm chức năng riêng biệt.

Nếu các interface không được tách biệt rõ ràng, các module sẽ phải phụ thuộc vào những phương thức không cần thiết, dẫn đến phụ thuộc chéo và tăng rủi ro khi bảo trì.

Do đó, việc tuân thủ ISP giúp ITS giữ được cấu trúc mô-đun, dễ bảo trì, và đảm bảo mỗi module chỉ tương tác với những phần mà nó cần.

5.4.3 Vấn đề khi không áp dụng ISP

Nếu nguyên tắc ISP không được áp dụng, một số vấn đề có thể xảy ra:

- **Fat Interface:** Một interface chứa quá nhiều phương thức cho nhiều mục đích khác nhau (ví dụ `IDataRepository` vừa xử lý người dùng, vừa xử lý câu hỏi, vừa xử lý kết quả).
- **Phụ thuộc thừa:** Module như Assessment phải phụ thuộc vào các phương thức của Content hoặc User mà nó không sử dụng.
- **Tăng coupling:** Khi một module thay đổi interface, các module khác phải sửa theo.
- **Khó kiểm thử:** Interface lớn khiến việc mock hoặc test trở nên phức tạp.

Hệ quả là kiến trúc trở nên cứng nhắc, khó mở rộng, và dễ phát sinh lỗi khi nâng cấp chức năng.

5.4.4 Áp dụng ISP trong ITS

Thiết kế ITS đã tuân thủ rõ ràng nguyên tắc ISP thông qua việc tách nhỏ interface theo từng module và trách nhiệm:

- **Module 1 – User & Authentication:** `IUserRepository`, `IAuthService`. Hai interface tách biệt: một xử lý dữ liệu người dùng, một xử lý xác thực và cấp JWT. Các thành phần khác không bị ép phụ thuộc vào logic xác thực.
- **Module 2 – Content Management:** `IContentService`, `IQuestionRepository`. Quản lý nội dung và câu hỏi tách riêng, tránh việc module khác phải sử dụng toàn bộ logic CRUD của khóa học.
- **Module 3 – Assessment Engine (core):** `IAssessmentEngine`, `IScoringStrategy`, `IHintStrategy`. Các chiến lược chấm điểm và sinh gợi ý được định nghĩa qua các interface riêng biệt. Assessment chỉ sử dụng những gì cần cho quá trình đánh giá.
- **Module 4 – Recommendation Engine:** `IRecommendationEngine`. Một interface duy nhất cho việc đề xuất lộ trình học. Các chiến lược gợi ý khác nhau (rule-based, model-based) đều implement interface này mà không ảnh hưởng tới các module khác.
- **Module 5 – Analytics & Reporting:** `IAnalyticsService`, `IReportExporter`. Tách rõ hai chức năng: tổng hợp dữ liệu và xuất báo cáo, đảm bảo mỗi phần chỉ biết hành vi liên quan của mình.
- **Module 6 – Notification & Scheduler:** `INotificationService`, `IScheduler`. Interface tách biệt giữa việc gửi thông báo và lập lịch định kỳ. Mỗi worker chỉ thực hiện đúng chức năng cần thiết.
- **Module 7 – Persistence & Infrastructure:** `PostgresAdapter`, `RedisCacheAdapter`,`S3StorageAdapter`. Các adapter nhỏ, độc lập, không gộp chung nhiều hành vi. Mỗi adapter chỉ triển khai đúng interface domain yêu cầu.

5.4.5 Lợi ích của ISP trong ITS

- Giảm độ kết dính giữa các module → mỗi module chỉ biết đến phần cần thiết.
- Dễ dàng mở rộng → có thể thêm chiến lược chấm điểm mới hoặc loại hình gợi ý khác mà không ảnh hưởng code hiện có.
- Tăng khả năng kiểm thử → mỗi interface nhỏ, dễ mô phỏng (mock).
- Tăng tính bảo trì → thay đổi cục bộ, không ảnh hưởng toàn hệ thống.
- Cải thiện tính minh bạch và tài liệu hóa API nội bộ → mỗi interface thể hiện đúng vai trò của module.

5.5 Nguyên tắc Đảo ngược phụ thuộc (DIP)

5.5.1 Định Nghĩa và Khái Niệm

Nguyên tắc **Dependency Inversion Principle (DIP)** phát biểu rằng: "Các module cấp cao không nên phụ thuộc vào module cấp thấp; cả hai nên phụ thuộc vào abstraction (interface)."

Trong kiến trúc phần mềm, nguyên tắc này thường được hiện thực hóa thông qua mô hình **Ports and Adapters (Hexagonal Architecture)**. Theo đó, tầng nghiệp vụ chỉ làm việc với các interface (Ports), còn tầng hạ tầng cung cấp các lớp triển khai (Adapters).

5.5.2 Tại sao ISP lại quan trọng trong ITS

ITS là hệ thống phức tạp, sử dụng nhiều công nghệ hạ tầng khác nhau: cơ sở dữ liệu (Postgres), cache (Redis), message broker (RabbitMQ/Kafka), và lưu trữ đối tượng (S3).

Nếu các module nghiệp vụ phụ thuộc trực tiếp vào các công nghệ này, hệ thống sẽ **khó kiểm thử, khó bảo trì và khó mở rộng** khi thay đổi công nghệ.

Do đó, việc áp dụng DIP giúp:

- **Tách biệt tầng domain logic** khỏi tầng kỹ thuật.
- **Dễ thay đổi công nghệ hạ tầng** mà không ảnh hưởng logic nghiệp vụ.
- **Tăng khả năng tái sử dụng và kiểm thử tự động.**

5.5.3 Vấn đề khi không áp dụng ISP

Nếu DIP không được tuân thủ, hệ thống có thể gặp các vấn đề sau:

- **Domain logic phụ thuộc trực tiếp vào hạ tầng:** ví dụ, `AssessmentEngine` gọi trực tiếp Postgres hoặc Redis.
- **Khó thay đổi công nghệ:** việc thay Redis bằng Memcached hoặc Kafka sẽ yêu cầu sửa toàn bộ code nghiệp vụ.
- **Khó kiểm thử:** các module phải kết nối thật tới cơ sở dữ liệu hoặc broker khi test.
- **Coupling ngược:** thay đổi ở tầng hạ tầng kéo theo thay đổi ở tầng nghiệp vụ, vi phạm nguyên tắc "low coupling, high cohesion".

5.5.4 Áp dụng ISP trong ITS

Kiến trúc ITS hiện thực hóa nguyên tắc DIP thông qua mô hình **Ports & Adapters**, trong đó:

- **Ports (abstraction)** nằm ở tầng domain/application.
- **Adapters (implementation)** nằm ở tầng infrastructure.

Các module:

- **Module 1 – User & Authentication** - Ports (Interfaces): `IUserRepository`, `IAuthService` - Adapters (Implementations): `PostgresUserRepository`, `AuthAdapter` - Cách thể hiện DIP: Auth logic chỉ phụ thuộc abstraction, không phụ thuộc DB cụ thể.
- **Module 2 – Content Management** - Ports (Interfaces): `IContentService`, `IQuestionRepository` - Adapters (Implementations): `PostgresContentRepo`, `S3StorageAdapter` - Cách thể hiện DIP: Tầng content chỉ biết đến interface; dễ thay DB hoặc storage mà không sửa logic.

- **Module 3 – Assessment Engine (core)** - Ports (Interfaces): `IAssessmentEngine`, `IScoringStrategy`, `IHintStrategy`, `IMessageBroker` - Adapters (Implementations): `PostgresAttemptRepo`, `RabbitMQAdapter` - Cách thể hiện DIP: Business logic độc lập hoàn toàn với cơ sở dữ liệu và broker, adapter được inject qua DI.
- **Module 4 – Recommendation Engine** - Ports (Interfaces): `IRecommendationEngine`, `IModelService` - Adapters (Implementations): `RuleBasedRecommender`, `MLGrpcAdapter` - Cách thể hiện DIP: Engine chỉ gọi interface model store; thay đổi ML engine không ảnh hưởng logic.
- **Module 5 – Analytics & Reporting** - Ports (Interfaces): `IAnalyticsService`, `IReportExporter` - Adapters (Implementations): `KafkaConsumerAdapter`, `S3ReportExporter` - Cách thể hiện DIP: Phần báo cáo không phụ thuộc trực tiếp broker hay storage.
- **Module 6 – Notification & Scheduler** - Ports (Interfaces): `INotificationService`, `IScheduler`, `IMessageBroker` - Adapters (Implementations): `EmailSenderAdapter`, `KafkaPublisher` - Cách thể hiện DIP: Worker chỉ gọi abstraction, dễ thay đổi cơ chế gửi hoặc queue.
- **Module 7 – Persistence & Infrastructure** - Ports (Interfaces): (không định nghĩa thêm; chỉ implement ports từ domain) - Adapters (Implementations): `PostgresAdapter`, `RedisAdapter`, `S3Adapter` - Cách thể hiện DIP: Là tầng hiện thực cụ thể của abstraction, đảo ngược phụ thuộc đúng tinh thần DIP.
- **Module thể hiện rõ nhất DIP:** *Assessment Engine*
  Đây là module lõi, nhưng hoàn toàn không phụ thuộc vào bất kỳ công nghệ cụ thể nào. Nó chỉ gọi các interface như `IQuestionRepository` và `IMessageBroker`. Các adapter thực tế (Postgres, RabbitMQ) được inject thông qua Dependency Injection ở tầng khởi tạo. Điều này minh họa chính xác nguyên tắc "module cấp cao phụ thuộc abstraction, không phụ thuộc chi tiết".

5.5.5 Lợi ích của ISP trong ITS

- **Tách biệt domain logic khỏi công nghệ hạ tầng**, dễ bảo trì và thay thế.
- **Tăng khả năng kiểm thử:** có thể mock toàn bộ repository và broker khi test.
- **Dễ mở rộng sang microservices:** mỗi module có thể triển khai độc lập mà không phụ thuộc trực tiếp vào hạ tầng.
- **Giảm rủi ro khi thay đổi công nghệ:** ví dụ, thay Kafka bằng RabbitMQ hoặc đổi hệ quản trị cơ sở dữ liệu mà không cần sửa logic nghiệp vụ.
- **Tăng tính linh hoạt và ổn định dài hạn:** hạ tầng có thể tiến hóa mà không ảnh hưởng đến cấu trúc cốt lõi.

6 Phản hồi về việc áp dụng các nguyên tắc SOLID

6.1 Giải quyết vấn đề bằng ứng dụng SOLID

Việc áp dụng các nguyên tắc thiết kế phần mềm là một bước quan trọng để đảm bảo rằng kiến trúc được xây dựng không chỉ đáp ứng các yêu cầu hiện tại mà còn có khả năng phát triển bền vững trong tương lai. Trong quá trình thiết kế Hệ thống Gia sư Thông minh (ITS), nhóm đã tuân thủ năm nguyên tắc SOLID của Robert C. Martin.

6.1.1 Tính Mô-đun hóa và Gắn kết cao (High Cohesion): Nguyên tắc Trách nhiệm Đơn (SRP)

Trong quá trình thiết kế ITS, nguyên tắc SRP được áp dụng triệt để bằng cách định nghĩa mỗi module chỉ nên chịu trách nhiệm trước một, và chỉ một, **"actor"**. Ví dụ, module `AssessmentEngine` chỉ chịu trách nhiệm trước "học sinh" (người cần được chấm bài), trong khi `AnalyticsWorker` chịu trách nhiệm trước "giáo viên" (người cần xem báo cáo).

Việc phân tách này đã giúp nhóm tránh được "anti-pattern" khi một lớp `Employee` phải phục vụ nhiều actor, dẫn đến việc các thay đổi từ một actor có thể vô tình làm hỏng chức năng của actor khác. Bằng cách chia nhỏ trách nhiệm, hệ thống ITS đã tránh được rủi ro này, làm cho mỗi module trở nên độc lập và gắn kết hơn.

6.1.2 Tính Dễ mở rộng và Linh hoạt (Extensibility and Flexibility): Nguyên tắc Mở/Đóng (OCP)

Nguyên tắc OCP là kim chỉ nam cho việc mở rộng hệ thống: **"mở cho việc mở rộng, nhưng đóng cho việc sửa đổi"** (open for extension but closed for modification).

Trong dự án ITS, điều này được thể hiện rõ nhất qua `AssessmentEngine`. Thay vì viết một chuỗi `if-else`, nhóm đã sử dụng **Strategy Pattern**. Một interface `IScoringStrategy` được định nghĩa, và mỗi loại chiến lược chấm điểm (trắc nghiệm, tự luận) sẽ là một lớp `concrete` riêng. Khi cần thêm phương pháp chấm điểm mới, nhóm chỉ cần tạo một lớp Strategy mới mà không cần sửa đổi mã nguồn cốt lõi của `AssessmentEngine`, giúp giảm lượng mã nguồn cũ phải thay đổi xuống mức tối thiểu, lý tưởng là **"bằng không"** (zero).

6.1.3 Tính Ổn định và Đúng đắn (Stability and Correctness): Nguyên tắc Thay thế Liskov (LSP)

Nguyên tắc này yêu cầu các lớp con phải có thể thay thế cho các lớp cha của chúng mà không làm thay đổi tính đúng đắn của chương trình.

Trong ITS, tất cả các lớp triển khai `IScoringStrategy` phải tuân thủ một "hợp đồng" (contract) nghiêm ngặt. Nhóm đã ý thức được rằng việc vi phạm LSP có thể làm ô nhiễm kiến trúc hệ thống (polluted the architecture).

6.1.4 Tính Độc lập của các thành phần (Component Independence): Nguyên tắc Tách biệt Giao diện (ISP)

Theo ISP, không nên buộc client phải phụ thuộc vào những phương thức mà nó không sử dụng. Bài học cốt lõi là việc phụ thuộc vào những thứ mang theo gánh nặng mà bạn không cần có thể gây ra những rắc rối không mong đợi.

Thay vì tạo ra một interface "béo" (fat interface), nhóm đã tách chúng thành các interface nhỏ, chuyên biệt như `IUserRepository`, `IQuestionRepository`. Điều này giúp cho `AssessmentEngine` chỉ cần phụ thuộc vào `IQuestionRepository`, giảm thiểu sự ghép cặp (coupling) và tránh các phụ thuộc không cần thiết.

6.1.5 Kiến trúc linh hoạt và Dễ kiểm thử (Testability): Nguyên tắc Đảo ngược Phụ thuộc (DIP)

DIP là nguyên tắc có ảnh hưởng lớn nhất, các module cấp cao không nên phụ thuộc vào module cấp thấp. Cả hai nên phụ thuộc vào abstractions.

Trong ITS, các module nghiệp vụ (cấp cao) như `AssessmentEngine` không gọi trực tiếp đến các lớp xử lý cơ sở dữ liệu (cấp thấp), mà phụ thuộc vào interface `IQuestionRepository`. Sự phụ thuộc đã được "đảo ngược" hướng so với luồng thực thi. Điều này tạo ra một **"ranh giới kiến trúc" (Architectural Boundary)** rõ ràng, ngăn cách phần trừu tượng khỏi phần cụ thể, giúp hệ thống cực kỳ dễ kiểm thử và linh hoạt. Để khởi tạo các đối tượng cụ thể, nhóm đã áp dụng mẫu thiết kế **Factory** để quản lý sự phụ thuộc này.

6.2 Thách thức trong việc thiết kế và ứng dụng SOLID

Mặc dù mang lại nhiều lợi ích, việc áp dụng các nguyên tắc SOLID cũng đi kèm với một số thách thức trong thực tế phát triển phần mềm:

- **Thiếu kinh nghiệm trong việc hoạch định bối cảnh:** Việc hiểu rõ các ràng buộc kỹ thuật, kinh tế và mục tiêu nghiệp vụ đòi hỏi kinh nghiệm thực tiễn để đưa ra giải pháp phù hợp. Nếu thiếu kinh nghiệm, việc áp dụng SOLID có thể dẫn đến các thiết kế không cân đối, gây lãng phí tài nguyên và giảm hiệu quả phát triển.
- **Chi phí ban đầu cao hơn:** Thiết kế theo SOLID yêu cầu nhiều thời gian và công sức để xác định các abstraction hợp lý và phân tách rõ ràng trách nhiệm giữa các thành phần. Đối với các dự án nhỏ hoặc có thời gian phát triển ngắn, chi phí ban đầu này có thể không tương xứng với lợi ích dài hạn.
- **Sự phức tạp trong thiết kế:** Việc áp dụng các nguyên tắc như *Dependency Inversion Principle (DIP)* và *Interface Segregation Principle (ISP)* có thể dẫn đến việc tạo ra nhiều interface và lớp nhỏ, làm cho cấu trúc hệ thống trở nên phức tạp và khó theo dõi, đặc biệt đối với các thành viên mới tham gia dự án.

- **Nguy cơ "Over-engineering":** Một thách thức lớn là tìm ra sự cân bằng hợp lý, tránh việc tạo ra các abstraction hoặc lớp trung gian không cần thiết cho các thành phần ít có khả năng thay đổi, gây lãng phí thời gian và làm giảm hiệu suất phát triển.
- **Xác định các "abstraction" ổn định:** Việc thiết kế các interface thực sự ổn định, ít phải thay đổi theo yêu cầu nghiệp vụ, đòi hỏi khả năng phân tích và tầm nhìn kiến trúc sâu sắc. Nếu abstraction thay đổi thường xuyên, lợi ích của SOLID trong việc giảm phụ thuộc và tăng khả năng mở rộng sẽ bị suy giảm đáng kể.
- **Khó khăn trong việc duy trì tính nhất quán trong nhóm phát triển:** Việc áp dụng SOLID đòi hỏi các thành viên trong nhóm phải có cùng mức độ hiểu biết và tuân thủ nhất quán các nguyên tắc thiết kế. Nếu không có quy ước hoặc hướng dẫn rõ ràng, mỗi lập trình viên có thể diễn giải các nguyên tắc theo cách riêng, dẫn đến sự không đồng nhất trong cấu trúc mã, gây khó khăn cho việc bảo trì, mở rộng và đào tạo nhân sự mới.

Như vậy, mặc dù có những thách thức ban đầu, những lợi ích dài hạn mà SOLID mang lại cho Hệ thống Gia sư Thông minh là không thể phủ nhận. Từ việc đảm bảo mỗi thành phần chỉ có một công việc duy nhất (SRP), cho phép thêm tính năng mới mà không sửa mã cũ (OCP), đảm bảo các lớp con hoạt động liền mạch (LSP), tạo ra các giao diện chuyên biệt (ISP), đến việc thúc đẩy sự linh hoạt và dễ kiểm thử (DIP), SOLID đã giúp nhóm xây dựng một nền tảng kiến trúc vững chắc, sẵn sàng cho sự phát triển và bảo trì trong tương lai. Đây là một sự đầu tư xứng đáng cho bất kỳ dự án phần mềm nghiêm túc nào.

7 Hiện thực sản phẩm

8 Đề xuất mở rộng trong tương lai

9 Tài liệu tham khảo

1. Robert C. Martin, *Clean Architecture: A Craftsman's Guide to Software Structure and Design*, Pearson; 1st edition (September 10, 2017).
2. Mark Richards, Neal Ford, *Fundamentals of Software Architecture: An Engineering Approach*, O'Reilly Media; 1st edition (January 28, 2020).
3. Paul Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Paulo Merson, Robert Nord, Judith Stafford, *Documenting Software Architectures: Views and Beyond*, 2nd edition, Addison-Wesley Professional, 2011.
4. Ian Sommerville, *Software Engineering*, 10th edition, Pearson Education, 2011.