☁️ Hạ Tầng & Triển Khai AWS - Tài Liệu Bảo Vệ Luận Văn

Infrastructure & Deployment - AWS Cloud Architecture

📑 Mục Lục Nhanh (Table of Contents)

🚀 BẢN TÓM TẮT "BỎ TÚI" (Executive Summary)

(Phần này dùng để trả lời nhanh trong 30 giây đầu tiên)

- **AWS EC2 Separation (Web vs AI):** Tách biệt 2 EC2 instances: Web Server (Next.js + Spring Boot) và n8n Server (AI Chatbot + Vector DB). Lý do: isolation về resource, security, và scaling độc lập. Web Server xử lý user traffic, n8n Server xử lý AI workloads riêng biệt.
- **Security Group Strategy (Chaining):** Security Groups được chain để tạo defense-in-depth. RDS chỉ accept connections từ Web Server SG, Web Server chỉ accept HTTPS (443) từ Internet và internal traffic từ n8n Server. n8n Server có restricted access, chỉ expose port 443 qua Caddy.
- **Caddy (Auto-SSL):** Sử dụng Caddy làm reverse proxy với automatic HTTPS/SSL certificate từ Let's Encrypt. Không cần manual certificate management, tự động renew. Caddy route traffic: `holidate.site` → Frontend (3000), `api.holidate.site` → Backend (8080), `n8n.holidate.site` → n8n (5678).
- **Docker/ECR (Containerization):** Tất cả services chạy trong Docker containers, images được push lên AWS ECR. Lợi ích: consistency giữa dev/prod, easy rollback, và version control cho images. Mỗi service có Dockerfile riêng và docker-compose để orchestrate.

1. AWS EC2 Separation Strategy (Chiến Lược Tách Biệt EC2)

1.1. Vấn đề & Giải pháp

Vấn đề:

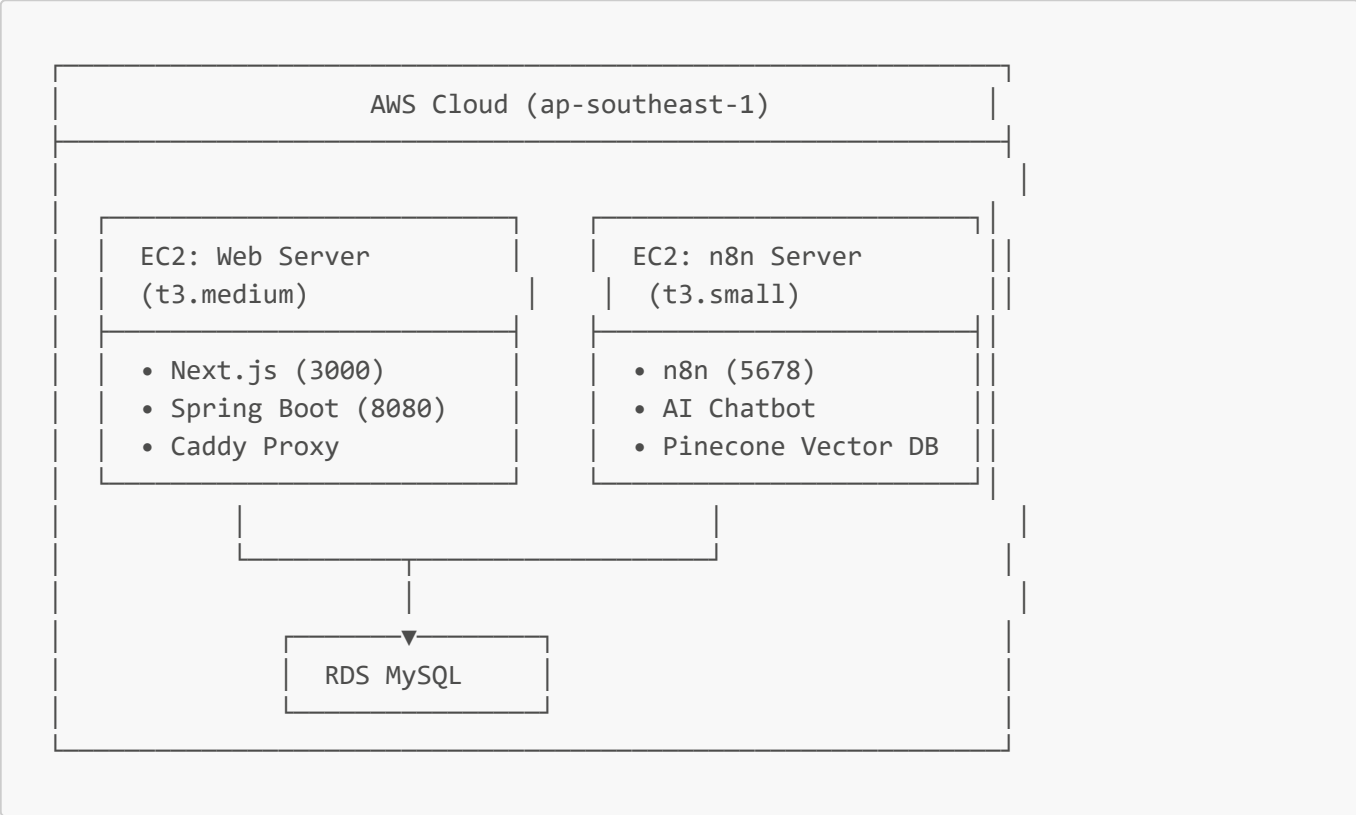
- AI workloads (chatbot, vector embeddings) tiêu tốn nhiều CPU/memory → làm chậm user traffic
- Scaling requirements khác nhau: Web Server scale theo user traffic, AI Server scale theo AI requests
- Security isolation: AI Server access external APIs (Pinecone, OpenAI) → risk exposure cao hơn

- Maintenance: Update AI components không nên ảnh hưởng production web services

Giải pháp:

- **2 EC2 Instances riêng biệt:**
  - **EC2 Web Server:** Next.js (3000) + Spring Boot (8080)
  - **EC2 n8n Server:** n8n (5678) + AI Chatbot + Pinecone integration
- **Resource Isolation:** Mỗi instance có CPU/memory riêng → không compete resources
- **Independent Scaling:** Scale từng server độc lập dựa trên traffic patterns
- **Network Isolation:** Security Groups riêng biệt → giảm attack surface

1.2. Kiến trúc (Architecture)



Traffic Flow:

- `holidate.site` → Web Server → Caddy → Next.js (3000)
- `api.holidate.site` → Web Server → Caddy → Spring Boot (8080)
- `n8n.holidate.site` → n8n Server → Caddy → n8n (5678)
- Backend → RDS MySQL, Backend → S3, n8n → S3

Mẹo Bảo Vệ (Defense Tip)

> "Hệ thống tách biệt 2 EC2 instances để đảm bảo resource isolation và independent scaling. Web Server xử lý user traffic (Next.js + Spring Boot), trong khi n8n Server xử lý AI workloads (chatbot, vector embeddings) riêng biệt. Lý do chính: AI workloads có thể tiêu tốn nhiều CPU/memory, nếu chạy chung sẽ làm chậm user experience khi mua hàng. Bằng cách tách biệt, chúng em có thể scale từng server độc lập và update AI components mà không ảnh hưởng đến production web services."

---

2. Security Group Strategy & Network Isolation (Chiến Lược Bảo Mật Mạng)

2.1. Vấn đề & Giải pháp

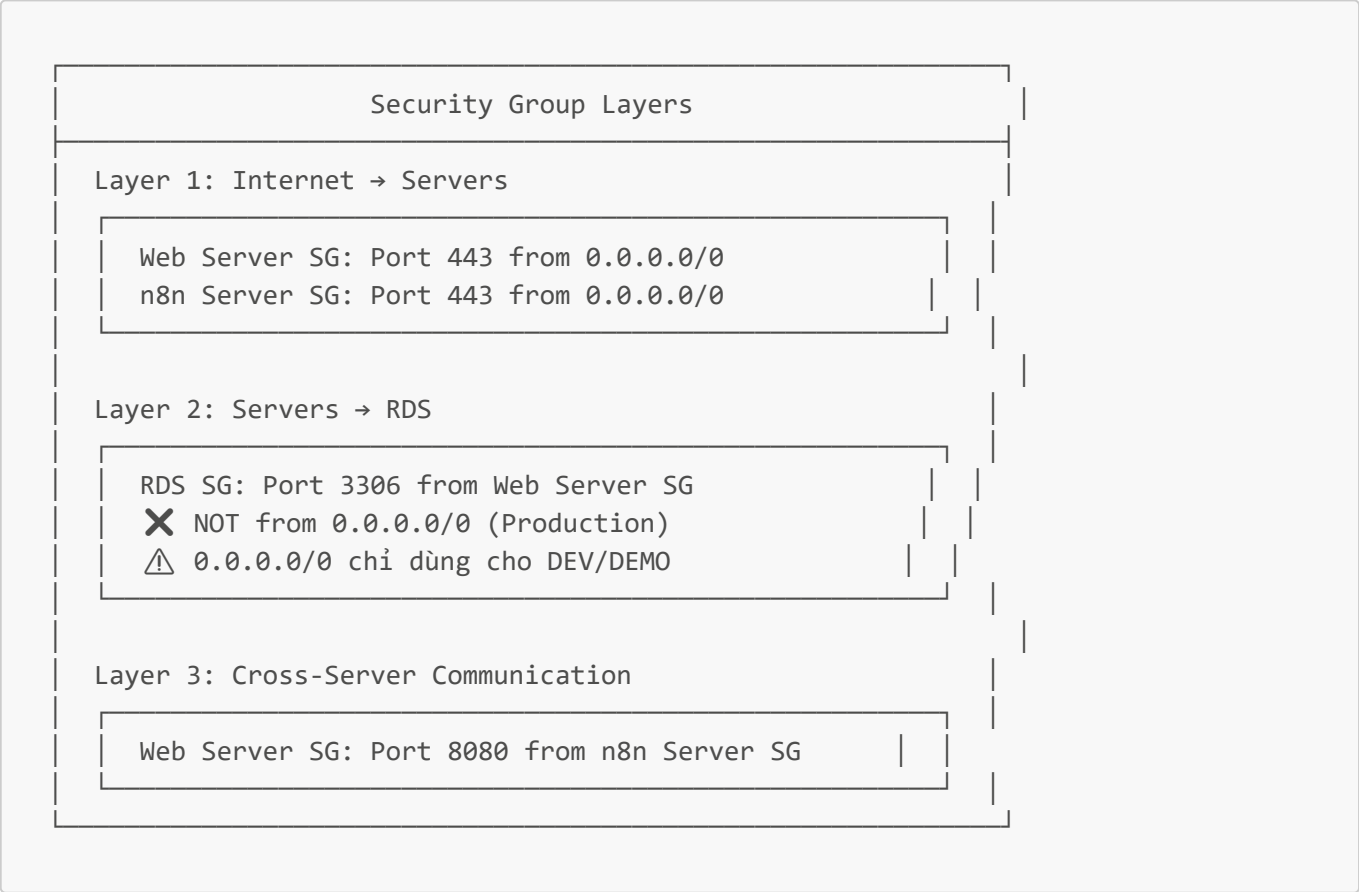
Vấn đề:

- RDS MySQL cần được bảo vệ khỏi public internet access
- Web Server và n8n Server cần communicate nhưng vẫn isolated
- Cần defense-in-depth: nhiều lớp bảo vệ thay vì chỉ 1 lớp
- Production security: principle of least privilege

Giải pháp:

- **Security Group Chaining (3-layer model):**
  - **Layer 1:** Internet → Web/n8n (chỉ HTTPS 443)
  - **Layer 2:** Web/n8n → RDS (RDS chỉ accept từ Web Server SG)
  - **Layer 3:** n8n → Web (internal API calls)
- **Source Group Chaining:** Dùng Security Group IDs thay vì IP addresses → dynamic và scalable
- **Port Restrictions:** Chỉ mở ports cần thiết (443, 3306, 8080, 3000, 5678)

2.2. Security Group Architecture



2.3. Defense Scenario: "Tại sao RDS mở 0.0.0.0/0?"

Câu hỏi thường gặp:

> "Tôi thấy RDS Security Group có rule cho phép 0.0.0.0/0, đây có phải là lỗ hổng bảo mật không?"

Trả lời (Defense Script):

> "Cảm ơn thầy/cô đã hỏi câu này. Đây là một điểm quan trọng về security architecture.
>
> **Trong môi trường DEV/DEMO hiện tại:**
>
> - RDS có thể được config với `0.0.0.0/0` để **debug nhanh** và **demo thuận tiện**
> - Cho phép connect từ bất kỳ đâu (local machine, CI/CD) mà không cần VPN
>
> **Tuy nhiên, Architecture Design chuẩn của hệ thống là:**
>
> - **Source Group Chaining:** RDS Security Group chỉ accept connections từ **Web Server Security Group ID**
> - **Không dùng IP addresses:** Vì IP có thể thay đổi khi restart instance
> - **Defense-in-depth:** Kết hợp với VPC, Subnet isolation, và Network ACLs
>
> **Production deployment sẽ:**
>
> 1. Remove rule `0.0.0.0/0` từ RDS Security Group
> 2. Add rule: Port 3306, Source = Web Server Security Group ID
> 3. Optional: Thêm VPC peering hoặc VPN cho admin access
>
> **Lý do thiết kế này:**
>
> - **Principle of Least Privilege:** RDS chỉ cần accessible từ Web Server, không cần từ Internet
> - **Attack Surface Reduction:** Giảm risk của SQL injection từ external sources
> - **Compliance:** Tuân thủ security best practices (OWASP, AWS Well-Architected Framework)
>
> Trong luận văn, chúng em document cả 2 scenarios: DEV config (0.0.0.0/0) và PROD design (SG chaining) để thể hiện understanding về security trade-offs."

💡 Mẹo Bảo Vệ (Defense Tip)

> "Hệ thống sử dụng Security Group Chaining với 3 lớp bảo vệ: (1) Internet chỉ access được HTTPS (443) vào Web/n8n Servers, (2) RDS chỉ accept connections từ Web Server Security Group (không phải từ Internet), và (3) Cross-server communication giữa n8n và Web Server được restrict bằng Security Group IDs. Trong DEV/DEMO, RDS có thể mở 0.0.0.0/0 để debug nhanh, nhưng Architecture Design chuẩn là dùng Source Group Chaining (RDS chỉ nhận traffic từ Web Server SG) để đảm bảo defense-in-depth và tuân thủ principle of least privilege."

---

3. Caddy Reverse Proxy & Auto-SSL (Reverse Proxy & Tự Động SSL)

3.1. Vấn đề & Giải pháp

Vấn đề:

- Cần HTTPS/SSL certificates cho 3 domains: `holidate.site`, `api.holidate.site`, `n8n.holidate.site`
- Manual certificate management (Let's Encrypt) phức tạp: renew mỗi 90 ngày, risk của expired certificates
- Nginx config phức tạp, dễ sai syntax, khó maintain

- Cần reverse proxy để route traffic: domain → internal ports (3000, 8080, 5678)

Giải pháp:

- **Caddy Server:** Automatic HTTPS với Let's Encrypt, zero-config SSL
- **Auto-renewal:** Caddy tự động renew certificates trước khi expire (30 days before)
- **Simple Config:** Caddyfile syntax đơn giản hơn Nginx, dễ đọc và maintain
- **Memory Safety:** Caddy viết bằng Go → memory-safe, ít bugs hơn C-based Nginx
- **Built-in Features:** Compression, HTTP/2, automatic redirect HTTP → HTTPS

3.2. Luồng xử lý (Logic Flow)

```
User Request → Caddy (Port 443)
    │
    ├─ holidate.site → Next.js Container (3000)
    ├─ api.holidate.site → Spring Boot Container (8080)
    └─ n8n.holidate.site → n8n Container (5678)

SSL Certificate Management:
    ├─ First Request: Auto-request from Let's Encrypt
    ├─ Validate domain ownership (DNS challenge)
    └─ Auto-renew before expiration (30 days before)
```

3.3. Key Configuration Snippet

```
# Web Server Caddyfile
holidate.site {
    reverse_proxy localhost:3000
    encode gzip
}

api.holidate.site {
    reverse_proxy localhost:8080 {
        header_up X-Forwarded-For {remote_host}
        header_up X-Forwarded-Proto {scheme}
    }
    encode gzip
}
```

Why Caddy vs Nginx:

- **Nginx:** Cần config SSL manually, certbot cron jobs, risk của expired certs
- **Caddy:** Zero-config, auto-renewal, built-in Let's Encrypt integration

Mẹo Bảo Vệ (Defense Tip)

> "Hệ thống sử dụng Caddy làm reverse proxy thay vì Nginx vì 3 lý do chính: (1) **Automatic HTTPS:** Caddy tự động request và renew Let's Encrypt certificates, không cần manual management hoặc

> certbot cron jobs, giảm risk của expired certificates. (2) **Simple Configuration:** Caddyfile syntax đơn giản và dễ đọc hơn Nginx config, giảm lỗi config. (3) **Memory Safety:** Caddy viết bằng Go (memory-safe) so với Nginx (C-based), ít bugs và security vulnerabilities hơn. Caddy route 3 domains: `holidate.site` → Frontend (3000), `api.holidate.site` → Backend (8080), `n8n.holidate.site` → n8n (5678), tất cả đều có HTTPS tự động."

---

4. Docker & ECR Containerization (Containerization & Quản Lý Images)

4.1. Vấn đề & Giải pháp

Vấn đề:

- **Environment Inconsistency:** Code chạy trên dev machine nhưng fail trên production (missing dependencies, version conflicts)
- **Deployment Complexity:** Manual deployment dễ sai, khó rollback khi có lỗi
- **Version Control:** Khó track version nào đang chạy trên production
- **Resource Isolation:** Multiple services trên cùng server có thể conflict (port, dependencies)

Giải pháp:

- **Docker Containers:** Mỗi service chạy trong isolated container với dependencies riêng
- **AWS ECR (Elastic Container Registry):** Centralized image repository, version tagging
- **Multi-stage Dockerfiles:** Optimize image size, separate build và runtime
- **Docker Compose:** Orchestrate multiple containers, easy start/stop/restart
- **Easy Rollback:** Pull previous image version từ ECR và restart container

4.2. Containerization Workflow

```
Development → Dockerfile (Multi-stage) → Build Image
    │
    ├─ Tag: holidate-backend:v1.0.4
    ├─ Push to AWS ECR
    └─ Production: docker-compose pull & up
```

Multi-stage Build Concept:

- **Stage 1 (Build):** Gradle/Node build environment → Generate artifacts
- **Stage 2 (Runtime):** Lightweight JRE/Alpine → Copy artifacts only
- **Result:** Smaller images, faster deployment, better security (non-root user)

4.3. Key Configuration Snippets

Multi-stage Dockerfile (Spring Boot):

```
# Stage 1: Build
FROM gradle:8.5-jdk21 AS build
WORKDIR /app
```

```
COPY . .
RUN ./gradlew build -x test --no-daemon

# Stage 2: Runtime (lightweight)
FROM eclipse-temurin:21-jre-alpine
WORKDIR /app
COPY --from=build /app/app.jar app.jar
USER spring  # Non-root user
EXPOSE 8080
ENTRYPOINT ["java", "-jar", "app.jar"]
```

Docker Compose với ECR:

```
services:
  backend:
    image: 387056640966.dkr.ecr.ap-southeast-1.amazonaws.com/holidate-backend:v1.0.4
    ports:
      - "8080:8080"
    deploy:
      resources:
        limits:
          cpus: '2.0'
          memory: 1G
```

ECR Versioning:

- Images được tag: `v1.0.4`, `v1.0.3`, `v1.0.2` → Easy rollback
- Production: `docker-compose pull` → `docker-compose up -d`

Mẹo Bảo Vệ (Defense Tip)

> "Hệ thống sử dụng Docker containers và AWS ECR để đảm bảo deployment consistency và easy rollback. Mỗi service (Backend, Frontend, n8n) có multi-stage Dockerfile riêng để optimize image size và security (non-root user). Images được push lên AWS ECR với version tags (v1.0.4, v1.0.3, ...), cho phép track versions và rollback nhanh khi có lỗi. Docker Compose orchestrate các containers với resource limits và health checks. Lợi ích: (1) Environment consistency giữa dev và prod, (2) Easy rollback bằng cách pull previous image version, (3) Version control cho production deployments, (4) Resource isolation giữa các services."

---

5. FAQ - Trả Lời Phản Biện (Frequently Asked Questions)

5.1. "Tại sao dùng EC2 mà không dùng Serverless (Lambda) hoặc Kubernetes?"

Câu hỏi:

> "Với AWS, tại sao không dùng Lambda (Serverless) hoặc EKS/Kubernetes để hiện đại hơn? EC2 có vẻ 'cổ điển'."

Trả lời:

> "Cảm ơn thầy/cô đã hỏi câu này. Đây là một câu hỏi về architecture trade-offs rất quan trọng.
>
> **Lý do chọn EC2 thay vì Serverless (Lambda):**
>
> 1. **Cost Efficiency cho Long-Running Services:**
>   - Lambda tính phí theo request và execution time (100ms increments)
>   - Spring Boot application có startup time ~10-20 giây → Lambda không phù hợp
>   - Next.js SSR cần persistent connection → Lambda không support WebSocket tốt
>   - **EC2 t3.medium:** ~$30/tháng cho 24/7 uptime → rẻ hơn Lambda nếu traffic ổn định
> 2. **Foundation Knowledge & Learning Curve:**
>   - Luận văn tập trung vào **business logic** (booking, search, payment) chứ không phải infrastructure complexity
>   - EC2 cho phép hiểu rõ hơn về: OS-level, networking, security groups, Docker
>   - Serverless có nhiều abstraction layers → khó debug và understand root cause
>   - **Pedagogical value:** Sinh viên cần hiểu infrastructure từ bottom-up trước khi dùng managed services
> 3. **Stateful Services:**
>   - n8n automation cần persistent state (workflows, credentials)
>   - AI chatbot cần maintain conversation context
>   - Lambda là stateless → cần external storage (DynamoDB, S3) → tăng complexity
>
> **Lý do không dùng Kubernetes (EKS):**
>
> 1. **Over-engineering cho quy mô hiện tại:**
>   - Hệ thống chỉ có 3 services (Frontend, Backend, n8n) → không cần orchestration phức tạp
>   - EKS có overhead: control plane cost (~$73/tháng), learning curve cao
>   - Docker Compose đủ cho single-server deployment
> 2. **Cost:**
>   - EKS: Control plane + Worker nodes → ~$100+/tháng
>   - EC2 + Docker Compose: ~$30/tháng
>   - **ROI:** Kubernetes chỉ justify khi có 10+ services hoặc multi-region
> 3. **Future Scalability:**
>   - Khi cần scale, có thể migrate lên EKS hoặc ECS (managed container service)
>   - Architecture hiện tại (Docker + ECR) đã compatible với Kubernetes
>   - **Migration path:** Chỉ cần thay docker-compose → Kubernetes manifests
>
> **Kết luận:** EC2 là lựa chọn hợp lý cho luận văn vì: (1) Cost-effective, (2) Educational value cao, (3) Đủ cho quy mô hiện tại, (4) Dễ migrate lên K8s/Serverless sau này. Chúng em document rõ trade-offs và future migration path trong luận văn."

5.2. "Nếu Web Server chết thì sao? Có Single Point of Failure (SPOF) không?"

Câu hỏi:

> "Hệ thống chỉ có 1 EC2 Web Server, nếu server này down thì toàn bộ hệ thống sập. Đây có phải là lỗ hổng architecture không?"

Trả lời:

> "Cảm ơn thầy/cô đã chỉ ra điểm này. Đây là một nhận xét rất đúng về high availability.
>
> **Thừa nhận SPOF:**
>
> Vâng, hiện tại hệ thống có **Single Point of Failure (SPOF)** ở Web Server EC2. Nếu server này down (hardware failure, AWS outage, hoặc misconfiguration), toàn bộ user-facing services sẽ không accessible.
>
> **Lý do thiết kế hiện tại:**
>
> 1. **Scope của Luận Văn:**
>   - Luận văn tập trung vào **business logic** (booking system, search algorithm, payment integration)
>   - High Availability (HA) là **operational concern**, không phải core research question
>   - **Trade-off:** Ưu tiên implement đầy đủ features thay vì optimize cho 99.99% uptime
> 2. **Cost Constraint:**
>   - Multi-AZ deployment cần ít nhất 2 EC2 instances + ALB (Application Load Balancer)
>   - Cost: ~$60-80/tháng (2x EC2 + ALB) vs ~$30/tháng (1x EC2)
>   - **Budget constraint** của luận văn không cho phép
> 3. **Proof of Concept Stage:**
>   - Hệ thống đang ở giai đoạn **MVP (Minimum Viable Product)**
>   - Chưa có production traffic thực tế → chưa cần HA ngay
>
> **Future Architecture (Đã Document trong Luận Văn):**
>
> Khi scale lên production, chúng em sẽ implement:
>
> - **Auto Scaling Group (ASG):** Min: 1, Desired: 2, Max: 4 instances
> - **Application Load Balancer (ALB):** Health checks, automatic failover
> - **Multi-AZ RDS:** Primary DB ở AZ-1, Standby ở AZ-2, automatic failover < 60s
> - **Route 53 Health Checks:** DNS failover nếu toàn bộ region down
>
> **Kết luận:** Chúng em thừa nhận SPOF hiện tại và đã document rõ **future architecture** với Auto Scaling + ALB trong luận văn. Đây là **pragmatic approach**: implement HA khi cần thiết (production traffic), không over-engineer ở giai đoạn MVP."

5.3. "Pinecone nằm ở đâu trong sơ đồ? Tại sao vẽ trong EC2?"

Câu hỏi:

> "Trong deployment diagram, Pinecone Vector DB được vẽ bên trong EC2 n8n Server. Nhưng Pinecone là SaaS chạy trên cloud của họ, không phải trên EC2 của mình. Đây có phải là lỗi trong diagram không?"

Trả lời:

> "Cảm ơn thầy/cô đã phát hiện điểm này. Đây là một câu hỏi rất hay về diagram semantics.
>
> **Giải thích về Diagram Representation:**
>
> Pinecone **KHÔNG** chạy trên EC2 của chúng em. Pinecone là **managed SaaS service** chạy trên infrastructure của Pinecone Inc.
>
> **Lý do vẽ trong EC2 trong diagram:**
>
> 1. **Logical View vs Physical View:**
>   - Diagram hiện tại là **Logical Architecture Diagram** (mô tả components và relationships)
>   - **KHÔNG phải** Physical Deployment Diagram (mô tả actual servers và locations)
>   - Pinecone được vẽ trong EC2 để thể hiện **logical relationship**: n8n Server **sử dụng** Pinecone như một component
> 2. **Clarity & Readability:**
>   - Nếu vẽ Pinecone bên ngoài AWS Cloud box → diagram phức tạp hơn
>   - Mục đích diagram: thể hiện **data flow** và **component interactions**
>   - Vẽ trong EC2 giúp reader hiểu: "AI components (Chatbot + Pinecone) chạy trên n8n Server"
> 3. **Convention trong Software Architecture:**
>   - Trong UML/Architecture diagrams, external services thường được vẽ như **logical components** trong system boundary
>   - Ví dụ: Google OAuth, VNPay Payment Gateway cũng có thể vẽ trong system boundary dù là external services
>
> **Clarification trong Luận Văn:**
>
> Trong luận văn, chúng em đã clarify:
>
> - **Section Architecture:** "Pinecone là external SaaS service, không deploy trên EC2"
> - **Section Integration:** "n8n Server gọi Pinecone API qua HTTPS (external network)"
> - **Diagram Note:** "Diagram thể hiện logical view, Pinecone là external service"
>
> **Kết luận:** Diagram hiện tại là **Logical Architecture Diagram**, không phải Physical. Pinecone được vẽ trong EC2 để thể hiện logical relationship và giữ diagram đơn giản. Trong luận văn, chúng em đã clarify rõ Pinecone là external SaaS service."

5.4. "Lưu DB Password và API Keys ở đâu? Có hardcode không?"

Câu hỏi:

> "Trong code có thấy database password, VNPay API keys, OpenAI API keys. Các thông tin nhạy cảm này được lưu ở đâu? Có hardcode trong source code không?"

Trả lời:

> "Cảm ơn thầy/cô đã hỏi câu này. Đây là một vấn đề bảo mật rất quan trọng.
>
> **Tuyệt đối không hardcode:**
>
> Hệ thống **KHÔNG** hardcode bất kỳ secret nào trong source code. Tất cả passwords, API keys, và sensitive configuration được quản lý thông qua environment variables.
>
> **Giải pháp Secret Management:**
>
> 1. **Development Environment:**
>   - Sử dụng file `.env` trong project root
>   - File `.env` được thêm vào `.gitignore` → **KHÔNG** được commit lên Git
>   - Format: `DB_PASSWORD=xxx`, `VNPAY_SECRET_KEY=yyy`, `OPENAI_API_KEY=zzz`
> 2. **Production Deployment:**
>   - **Docker Compose Environment Variables:** Secrets được inject vào container thông qua `docker-compose.yml` environment section
>   - **AWS Systems Manager Parameter Store:** Trong môi trường production, secrets được lưu trong AWS SSM Parameter Store (encrypted)
>   - Application đọc từ environment variables, không đọc từ file
> 3. **Git Security:**
>   - Source code có thể public (GitHub) → không có vấn đề
>   - Secrets nằm trong `.env` (không commit) → **Code lộ nhưng key không lộ**
>   - `.gitignore` đảm bảo `.env` files không bao giờ được track
>
> **Best Practices:**
>
> - **12-Factor App:** Configuration qua environment variables (không hardcode)
> - **Separation of Concerns:** Code và secrets tách biệt hoàn toàn
> - **Rotation:** Có thể rotate secrets mà không cần thay đổi code
>
> **Kết luận:** Hệ thống tuân thủ security best practices: không hardcode secrets, sử dụng environment variables, và trong production dùng AWS SSM Parameter Store để quản lý secrets một cách an toàn."

5.5. "Dữ liệu Booking rất quan trọng, nếu DB lỗi thì sao?"

Câu hỏi:

> "Dữ liệu booking (đơn hàng, thanh toán) là rất quan trọng. Nếu RDS MySQL bị lỗi (hardware failure, corruption, hoặc accidental deletion) thì sao? Có backup không?"

Trả lời:

> "Cảm ơn thầy/cô đã hỏi câu này. Đây là một concern rất quan trọng về data protection.
>
> **RDS Automated Backups:**
>
> RDS MySQL được cấu hình với **Automated Backups** (tự động bật mặc định):
>
> 1. **Daily Snapshots:**
>   - RDS tự động chụp snapshot hàng ngày vào thời điểm maintenance window
>   - Snapshots được lưu trong S3 (durable storage)
>   - Retention period: 7 ngày (có thể tăng lên 35 ngày)
> 2. **Transaction Logs (Point-in-Time Recovery):**
>   - RDS lưu transaction logs liên tục
>   - Cho phép **Point-in-Time Recovery (PITR)**: khôi phục về bất kỳ thời điểm nào trong 5 phút trước
>   - Ví dụ: Nếu xóa nhầm data lúc 14:30, có thể restore về 14:25
>
> **Recovery Process:**
>
> Nếu có sự cố:
>
> 1. **Từ Snapshot:** Restore ra RDS instance mới từ snapshot gần nhất → ~10-15 phút
> 2. **Point-in-Time Recovery:** Restore đến thời điểm cụ thể trước khi xảy ra lỗi → ~15-20 phút
> 3. **Update Application:** Point application đến RDS instance mới (chỉ cần update connection string)
>
> **Additional Protection (Future):**
>
> Khi scale lên production, có thể thêm:
>
> - **Multi-AZ Deployment:** Primary DB ở AZ-1, Standby ở AZ-2 → Automatic failover < 60s
> - **Read Replicas:** Tách biệt read traffic, có thể promote thành primary nếu primary fail
> - **Cross-Region Backups:** Replicate snapshots sang region khác để disaster recovery
>
> **Kết luận:** RDS có Automated Backups (daily snapshots) và Transaction Logs (PITR) đảm bảo dữ liệu booking được bảo vệ. Nếu có sự cố, có thể restore ra DB instance mới trong vòng 15 phút. Đây là managed service benefit của AWS RDS - không cần tự quản lý backup scripts."

Phụ Lục (Appendix)

A.1. Caddyfile Configuration Snippet

```
# Web Server Caddyfile
holidate.site {
    reverse_proxy localhost:3000
    encode gzip
}
```

```
api.holidate.site {
    reverse_proxy localhost:8080 {
        header_up X-Forwarded-For {remote_host}
        header_up X-Forwarded-Proto {scheme}
    }
    encode gzip
}
```

A.2. Docker Compose Snippet

```
services:
  backend:
    image: 387056640966.dkr.ecr.ap-southeast-1.amazonaws.com/holidate-
backend:v1.0.4
    ports:
      - "8080:8080"
    deploy:
      resources:
        limits:
          cpus: '2.0'
          memory: 1G
```

A.3. Security Group Rules Table

Security Group	Type	Port	Source	Description
Web Server SG				
	Inbound	HTTPS (443)	0.0.0.0/0	Public HTTPS access
	Inbound	TCP (8080)	n8n-server-sg-id	Internal API calls
	Outbound	All	0.0.0.0/0	Allow all outbound
n8n Server SG				
	Inbound	HTTPS (443)	0.0.0.0/0	Public HTTPS access
	Outbound	All	0.0.0.0/0	Allow all outbound
RDS Security Group				
	Inbound	MySQL (3306)	web-server-sg-id	⚠ PRODUCTION: Only from Web Server SG

Security Group	Type	Port	Source	Description
	Inbound	MySQL (3306)	0.0.0.0/0	⚠ DEV/DEMO: For debug convenience

⚠ **Lưu ý:** Production phải dùng Security Group chaining (RDS chỉ accept từ Web Server SG), không dùng `0.0.0.0/0`.

Tổng Kết

Hệ thống được thiết kế với các đặc điểm chính về Infrastructure & Deployment:

1. **EC2 Separation:** Tách biệt Web Server và n8n Server để resource isolation và independent scaling
2. **Security Group Chaining:** 3-layer defense-in-depth với Source Group chaining (không dùng IP addresses)
3. **Caddy Auto-SSL:** Automatic HTTPS với Let's Encrypt, zero-config certificate management
4. **Docker & ECR:** Containerization với multi-stage builds, version control, và easy rollback
5. **Pragmatic Architecture:** SPOF được thừa nhận, future architecture (Auto Scaling + ALB) đã được document

Tất cả các components đều có health checks, logging, resource limits, và security best practices để đảm bảo reliability và maintainability.

Tài liệu này phục vụ cho mục đích bảo vệ luận văn, cung cấp câu trả lời sẵn cho các câu hỏi phản biện về Infrastructure & Deployment Architecture.