

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



## ĐỀ CƯƠNG LUẬN VĂN

---

Nền tảng quản lý trao đổi Email và mở rộng  
đa kênh ứng dụng AI

---

GVHD: ThS. Võ Thanh Hùng

SV thực hiện: Ngô Nguyễn Quốc Thịnh 1814168

TP Hồ Chí Minh, Tháng 12 Năm 2025



## Mục lục

<b>1 Tổng quan</b>	<b>2</b>
1.1 Đặt vấn đề . . . . .	2
1.2 Nghiên cứu các giải pháp hiện có . . . . .	3
1.3 Xác định "lỗ hổng" thị trường và đề xuất giải pháp của báo cáo . . . . .	5
1.4 Mục tiêu và phạm vi báo cáo . . . . .	6
<b>2 Phân tích và Đặc tả Yêu cầu</b>	<b>8</b>
2.1 Yêu cầu chức năng (Functional Requirements) . . . . .	8
2.2 Yêu cầu phi chức năng (Non-Functional Requirements) . . . . .	9
2.3 Quyết định công nghệ và Chiến lược kiến trúc . . . . .	10
<b>3 Cơ sở công nghệ và lựa chọn Tech stack</b>	<b>12</b>
3.1 Frontend stack: Next.js (React) và TypeScript . . . . .	12
3.2 Backend orchestration: Next.js API Routes và mô hình microservice . . . . .	12
3.3 Cơ sở dữ liệu: NoSQL và MongoDB cho dữ liệu Email bán cấu trúc . . . . .	14
3.4 Cache/Queue: Redis cho PoC và định hướng mở rộng . . . . .	14
3.5 Triển khai PoC: Docker Compose và khả năng chuyển tiếp . . . . .	15
3.6 Bối cảnh AI trong các hệ thống giao tiếp . . . . .	16
3.7 Phân loại giải pháp AI: Local vs. Cloud . . . . .	16
3.8 Các phương pháp tiếp cận LLM . . . . .	19
3.9 AI Agent và khả năng mở rộng trong tương lai . . . . .	21
3.10 Hạn chế hiện tại của LLM trong bối cảnh doanh nghiệp . . . . .	23
<b>4 Thiết kế Hệ thống</b>	<b>25</b>
4.1 Kiến trúc tổng thể hệ thống . . . . .	25
4.2 Thiết kế luồng tương tác theo yêu cầu chức năng . . . . .	27
4.3 Thiết kế Cơ sở dữ liệu (Database Schema) . . . . .	36
<b>5 Hiện thực và Thử nghiệm PoC</b>	<b>40</b>
5.1 Môi trường triển khai và cách chạy PoC . . . . .	40
5.2 Tổng kết các chức năng đã hiện thực . . . . .	41
<b>6 Tổng kết</b>	<b>42</b>
6.1 Tổng quan giai đoạn 1 (14 tuần) . . . . .	42
6.2 Kế hoạch ban đầu giai đoạn 1 (Planned) . . . . .	42
6.3 Kết quả thực tế giai đoạn 1 (Actual) và khó khăn . . . . .	43
6.4 Kế hoạch giai đoạn 2 (14 tuần) . . . . .	44
6.5 Rủi ro giai đoạn 2 (Risks) . . . . .	44



## 1 Tổng quan

Cuộc cách mạng công nghiệp lần thứ tư, cùng với làn sóng chuyển đổi số mạnh mẽ trong thập niên vừa qua, đã làm thay đổi sâu sắc cách thức con người và tổ chức trao đổi thông tin. Trong môi trường doanh nghiệp hiện đại, Email vẫn giữ vai trò là kênh giao tiếp chính thức, dùng cho những trao đổi mang tính lưu trữ, pháp lý hoặc điều phối quy trình. Song song đó, các nền tảng nhắn tin tức thời (Instant Messaging - IM) như Zalo, WhatsApp, Telegram hay Viber ngày càng được ưa chuộng nhờ tính tức thời, thân thiện và phù hợp với tương tác hàng ngày.

Sự tồn tại đồng thời của hai nhóm kênh này đã hình thành nên một hệ sinh thái giao tiếp đa kênh đa dạng nhưng cũng đầy phức tạp. Nếu như Email thường được sử dụng để khởi tạo mối quan hệ (gửi hồ sơ ứng tuyển, báo giá, hợp đồng...), thì các trao đổi tiếp theo, mang tính thương lượng, xác nhận nhanh hay chăm sóc khách hàng, lại dễ dàng "trôi" sang các ứng dụng IM. Về phía người dùng cuối, điều này mang lại trải nghiệm linh hoạt và tiện lợi. Tuy nhiên, dưới góc nhìn quản lý và vận hành, sự đa dạng hóa kênh giao tiếp lại dẫn đến một hệ quả đáng lo ngại: **sự phân mảnh nghiêm trọng của dữ liệu liên lạc**.

Thay vì có một bản ghi đầy đủ về lịch sử trao đổi với từng đối tác, khách hàng hay ứng viên, dữ liệu thực tế bị chia cắt thành nhiều mảnh rời rạc: một phần nằm trong hộp thư Email, một phần trong các đoạn chat Zalo, một phần khác lại nằm trong nhóm Telegram nội bộ. Trạng thái này khiến cho việc truy xuất lại bối cảnh, kiểm tra lại cam kết hoặc tổng hợp báo cáo trở nên tốn thời gian, phụ thuộc vào trí nhớ của cá nhân và dễ phát sinh sai sót. Trong bối cảnh doanh nghiệp ngày càng coi dữ liệu là tài sản chiến lược, tình trạng phân mảnh đó không chỉ là bất tiện kỹ thuật mà còn là rủi ro về mặt vận hành và ra quyết định.

Báo cáo này tập trung nghiên cứu và đề xuất một hướng tiếp cận nhằm giải quyết bài toán nêu trên. Mục tiêu tổng quát là xây dựng cơ sở lý luận, phân tích thực trạng và đề xuất mô hình kiến trúc cho một **Nền tảng Quản lý Trao đổi Tập trung** (Centralized Communication Management Platform). Nền tảng được thiết kế với Email là thành phần trung tâm (Email-centric), đóng vai trò "xương sống" của luồng giao tiếp, đồng thời sở hữu kiến trúc module linh hoạt, cho phép tích hợp tùy chọn các kênh giao tiếp khác sau. Cách tiếp cận này kỳ vọng vừa khắc phục bài toán phân mảnh dữ liệu, vừa giữ được tính đơn giản và khả năng triển khai thực tế cho các đội nhóm vừa và nhỏ.

### 1.1 Đặt vấn đề

Trong các quy trình nghiệp vụ mang tính chuyên nghiệp, chẳng hạn như tuyển dụng nhân sự, quản lý quan hệ khách hàng (Customer Relationship Management - CRM) hay hỗ trợ kỹ thuật (Support), Email thường được sử dụng như kênh giao tiếp khởi đầu. Các trao đổi thông qua Email mang tính trang trọng, có cấu trúc và dễ lưu trữ, tạo nền tảng cho việc hình thành hồ sơ và quy trình nội bộ. Tuy nhiên, để đáp ứng yêu cầu phản hồi nhanh, thuận tiện và gần gũi hơn, không ít cuộc trao đổi sau đó được "dịch chuyển" dần sang các ứng dụng IM như Zalo, WhatsApp hay Telegram.

Sự chuyển dịch không đồng nhất này diễn ra một cách tự nhiên theo thói quen của người dùng, chứ không theo một thiết kế hệ thống nào. Theo thời gian, một quy trình tưởng chừng đơn giản như tiếp nhận và xử lý yêu cầu của khách hàng có thể trải dài trên nhiều kênh: từ email ban đầu, tới các tin nhắn trao đổi ngắn, rồi quay lại email để chốt thỏa thuận cuối cùng. Đối với từng cá nhân, việc "nhảy" qua lại giữa các ứng dụng này có thể không phải vấn đề lớn. Nhưng đối với tổ chức, đặc biệt là khi có nhiều nhân sự cùng tham gia xử lý, các bất cập sau sẽ dần bộc lộ rõ:

- **Sự phân mảnh dữ liệu (Data Fragmentation):** Đây là thách thức cốt lõi và mang tính nền tảng. Lịch sử trao đổi với một đối tượng cụ thể (ứng viên, khách hàng, đối tác)



không còn tồn tại tại một nơi duy nhất, mà bị phân tán trên nhiều ứng dụng khác nhau (Email, Zalo, Telegram, nhóm chat nội bộ...). Khi cần nhìn lại toàn cảnh, người dùng buộc phải tìm kiếm thủ công từ nhiều nguồn, dễ bỏ sót chi tiết quan trọng và gần như không thể tự động hóa.

- **Sự đứt gãy ngữ cảnh (Contextual Discontinuity):** Mỗi lần một nhân sự mới được giao tiếp quản một vấn đề đang dang dở, hoặc đơn giản chỉ là quay lại xử lý một cuộc trao đổi sau một khoảng thời gian, họ phải lần lượt mở từng ứng dụng, tìm kiếm theo từ khóa, đổi chiếu thời gian để dựng lại bức tranh bối cảnh. Quá trình này không chỉ mất thời gian mà còn tạo cảm giác "không liền mạch" trong trải nghiệm làm việc, làm giảm khả năng đưa ra quyết định chính xác và kịp thời.
- **Suy giảm hiệu suất và gia tăng rủi ro nghiệp vụ:** Việc liên tục chuyển đổi giữa các ứng dụng khác nhau trong suốt ca làm việc gây ra hiện tượng "context switching" liên tục, được chứng minh là có tác động tiêu cực đến năng suất. Bên cạnh đó, các thông tin quan trọng (ví dụ: yêu cầu nâng mức dịch vụ, phản hồi tiêu cực của khách hàng) nếu chỉ xuất hiện trong một đoạn chat riêng lẻ rất dễ bị bỏ sót, dẫn tới những rủi ro về uy tín và chất lượng dịch vụ.
- **Hiện tượng quá tải thông tin (Information Overload):** Ngay cả trong trường hợp toàn bộ dữ liệu được tập hợp lại, khối lượng email và tin nhắn khổng lồ trong một lịch sử trao đổi kéo dài nhiều tháng vẫn là một gánh nặng nhận thức. Người phụ trách khó có thể đọc lại tất cả nội dung mỗi lần cần nắm bắt bối cảnh. Do đó, nảy sinh nhu cầu về một cơ chế thông minh có khả năng tự động "tiêu hóa" lượng thông tin thô này, chắt lọc và trích xuất các ý chính, mốc quan trọng và vấn đề trọng tâm.

Dõi chiếu với thực tế làm việc tại nhiều doanh nghiệp dịch vụ vừa và nhỏ, có thể nhận thấy mô hình đa kênh hiện tại phần lớn phát triển theo hướng *tự phát* hơn là *được thiết kế có chủ đích*. Từ những phân tích nêu trên, việc nghiên cứu một giải pháp hợp nhất, thông minh, đặt Email ở vị trí trung tâm để tái cấu trúc cách tổ chức lưu trữ và khai thác dữ liệu liên lạc là nhu cầu mang tính cấp thiết, không chỉ về mặt tiện ích mà còn về mặt chiến lược dữ liệu của tổ chức.

## 1.2 Nghiên cứu các giải pháp hiện có

Để khẳng định tính mới và xác định đúng "khoảng trống" mà báo cáo hướng tới, việc khảo sát bức tranh giải pháp hiện có trên thị trường là cần thiết. Qua quá trình tham khảo tài liệu, trải nghiệm dùng thử và phân tích tính năng, các giải pháp quản lý liên lạc và chăm sóc khách hàng hiện nay có thể được quy nạp thành ba nhóm chính, mỗi nhóm thể hiện một triết lý thiết kế khác nhau với các điểm mạnh và hạn chế riêng.

### Nhóm 1: Các ứng dụng tập trung vào Email (Email-centric)

Nhóm đầu tiên là các ứng dụng đặt trọng tâm vào việc "nâng cấp" trải nghiệm làm việc với Email truyền thống. Thay vì bổ sung thật nhiều module, các giải pháp này tập trung vào việc làm cho việc xử lý hộp thư trở nên nhanh, gọn, thông minh và dễ chịu hơn đối với người dùng chuyên nghiệp.

- **Các đại diện tiêu biểu:** Superhuman, Front, Missive.
- **Phân tích ưu điểm:** Các ứng dụng này thường cung cấp trải nghiệm người dùng (User Experience - UX) được tối ưu rất kỹ: giao diện tối giản nhưng giàu thông tin, hỗ trợ phím



tắt gần như cho mọi thao tác, cho phép xử lý số lượng lớn email trong thời gian ngắn. Một số sản phẩm như Front, Missive còn bổ sung tính năng cộng tác nội bộ (shared inbox, internal comments), giúp nhiều thành viên cùng phối hợp trên một địa chỉ email chung. Trong những năm gần đây, nhiều giải pháp trong nhóm này cũng bắt đầu tích hợp các tính năng AI ở mức độ cơ bản như tóm tắt email, gợi ý câu trả lời, kiểm tra lại tông giọng (tone) trước khi gửi.

- **Phân tích hạn chế:** Dù mạnh về trải nghiệm với Email, phần lớn các giải pháp này vẫn coi Email là trung tâm duy nhất. Việc tích hợp với các kênh IM nếu có thường chỉ dùng ở mức độ thông báo (notifications) hoặc các kết nối cơ bản, không được thiết kế như một phần của kiến trúc dữ liệu xuyên suốt. Do đó, bài toán hợp nhất lịch sử trao đổi đa kênh chưa được giải quyết triệt để: người dùng vẫn phải chạy song song nhiều ứng dụng nếu muốn bao quát đầy đủ các kênh tương tác với khách hàng.

### Nhóm 2: Các nền tảng Quản lý Quan hệ Khách hàng (CRM) đa kênh

Nhóm thứ hai là các nền tảng CRM (Customer Relationship Management) đa kênh, thường được cung cấp theo mô hình phần mềm dịch vụ (Software-as-a-Service - SaaS). Đây là những hệ thống lớn, được thiết kế để bao phủ gần như toàn bộ vòng đời tương tác giữa doanh nghiệp và khách hàng.

- **Các đại diện tiêu biểu:** Zendesk, HubSpot Service Hub, Intercom.
- **Phân tích ưu điểm:** Điểm mạnh rõ rệt của các nền tảng này là khả năng hợp nhất dữ liệu đa kênh: email, live chat, mạng xã hội, SMS, thậm chí cả cuộc gọi điện thoại đều có thể được ghi nhận và liên kết với một hồ sơ khách hàng duy nhất. Ngoài ra, chúng thường đi kèm với bộ công cụ quản lý ticket, workflow tự động, báo cáo và dashboard phong phú, giúp nhà quản lý có góc nhìn 360 độ về mỗi khách hàng và toàn bộ hoạt động chăm sóc.
- **Phân tích hạn chế:** Dổi lại, các hệ thống này thường khá cồng kềnh. Việc triển khai đòi hỏi thời gian khảo sát, cấu hình, tích hợp và đào tạo nội bộ đáng kể. Mô hình định giá thường dựa trên số lượng người dùng (per-seat) nên chi phí có thể trở nên rất cao đối với các doanh nghiệp nhỏ. Dối với những đội ngũ chỉ có nhu cầu quản lý email và một vài kênh IM đơn giản, "gói tính năng" đồ sộ của các nền tảng CRM đa kênh dễ trở nên vượt quá nhu cầu thực tế nhưng vẫn phải chi trả đầy đủ.

### Nhóm 3: Các giải pháp mã nguồn mở (Open-source)

Nhóm thứ ba là các giải pháp mã nguồn mở, cho phép tổ chức tự triển khai (self-host) trên hạ tầng của riêng mình. Chúng thường được cộng đồng phát triển theo hướng linh hoạt, dễ mở rộng và không bị khóa chặt vào mô hình kinh doanh của một nhà cung cấp duy nhất.

- **Các đại diện tiêu biểu:** Chatwoot, Rocket.Chat.
- **Phân tích ưu điểm:** Giá trị nổi bật nhất của nhóm này là *quyền riêng tư và quyền kiểm soát dữ liệu* (data sovereignty). Bằng việc tự triển khai trên máy chủ nội bộ hoặc hạ tầng cloud riêng, tổ chức có thể kiểm soát hoàn toàn nơi dữ liệu được lưu trữ, cách sao lưu, mã hóa và truy cập. Bên cạnh đó, mã nguồn mở cho phép mức độ tùy biến cao: từ giao diện người dùng tới workflow nghiệp vụ, tích hợp thêm các adapter mới hoặc gắn AI vào những điểm chạm cụ thể.



- **Phân tích hạn chế:** Đổi lại cho sự tự do đó là yêu cầu về năng lực kỹ thuật nội bộ: đội ngũ phải có kiến thức về quản trị hệ thống, bảo mật, cập nhật phiên bản và xử lý sự cố. Ngoài ra, dù nhiều dự án open-source đã hỗ trợ đa kênh tương đối tốt, trải nghiệm làm việc với Email (như một ứng dụng chuyên biệt) vẫn chưa đạt mức tinh chỉnh và mượt mà như các sản phẩm ở Nhóm 1, vốn được tối ưu chỉ cho một bài toán duy nhất.

### 1.3 Xác định "lỗ hổng" thị trường và đề xuất giải pháp của báo cáo

Từ việc phân tích ba nhóm giải pháp nêu trên, có thể nhận diện tương đối rõ một "lỗ hổng" trong không gian giải pháp hiện tại:

- Người dùng, đặc biệt là các đội nhóm vừa và nhỏ, mong muốn một công cụ có **trải nghiệm làm việc với Email xuất sắc** (nhanh, mượt, hỗ trợ thao tác chuyên nghiệp) như các ứng dụng ở Nhóm 1.
- Đồng thời, công cụ đó cần duy trì **khả năng mở rộng đa kênh linh hoạt** như Nhóm 2, cho phép từng bước đưa thêm dữ liệu từ các kênh IM vào cùng một bức tranh tổng thể mà không bị **cồng kềnh và đắt đỏ** như các hệ thống CRM toàn diện.
- Cuối cùng, giải pháp phải cho phép tổ chức **giữ quyền kiểm soát dữ liệu** của riêng mình và **tích hợp AI ở mức sâu**, vượt ra khỏi những tính năng bề mặt như tóm tắt một email đơn lẻ, hướng tới khả năng hiểu ngữ cảnh, trích xuất vấn đề, gợi ý hành động tiếp theo.

Báo cáo này đề xuất một hướng tiếp cận nhằm giải quyết bài toán đó thông qua việc **thiết kế và hiện thực một hệ thống Chứng minh Khái niệm (Proof of Concept - PoC)** với các đặc điểm kiến trúc cốt lõi sau:

1. **Kiến trúc lấy Email làm lõi (Email-centric Core):** Hệ thống được thiết kế với giả định rằng Email luôn là kênh bắt buộc và đóng vai trò xương sống trong giao tiếp nghiệp vụ. Thay vì xem Email là "một trong nhiều kênh" như nhiều hệ thống CRM, báo cáo lựa chọn đi sâu giải quyết thật tốt bài toán Email trước, sau đó mới mở rộng ra các kênh khác.
2. **Thiết kế module và microservice linh hoạt:** Nền tảng được tổ chức theo kiến trúc monorepo nhưng bên trong là các dịch vụ độc lập (backend, frontend, AI-service) giao tiếp qua API rõ ràng. Cách tổ chức này vừa giúp quá trình phát triển, nâng cấp từng phần diễn ra linh hoạt, vừa chuẩn bị sẵn "điểm mốc" để về sau có thể bổ sung các adapter cho các kênh IM như Zalo, Telegram mà không ảnh hưởng đến lõi.
3. **Tích hợp AI như một dịch vụ độc lập (AI as a Service):** Thay vì nhúng trực tiếp các lời gọi AI vào backend, báo cáo tách riêng một dịch vụ AI (phát triển bằng FastAPI) hoạt động như một microservice độc lập. Cách tiếp cận này mang lại hai lợi ích: (1) cho phép hoán đổi linh hoạt giữa các nhà cung cấp AI (OpenAI, mô hình nội bộ, HuggingFace, v.v.) mà không phải sửa logic nghiệp vụ; (2) tạo điều kiện để AI thực thi các tác vụ phức tạp như tóm tắt thread dài, gợi ý phản hồi, thậm chí mở rộng thành AI agent trong tương lai theo mô hình bất đồng bộ (async), không chặn luồng xử lý chính.
4. **Khả năng tự triển khai (Self-hostable):** Toàn bộ hệ thống được đóng gói bằng Docker Compose, cho phép người dùng tự triển khai trên hạ tầng của mình một cách tương đối đơn giản. Điều này giúp cân bằng giữa ưu điểm trải nghiệm của Nhóm 1, năng lực hợp nhất dữ liệu của Nhóm 2 và quyền kiểm soát dữ liệu của Nhóm 3.



PoC được xây dựng trong khuôn khổ báo cáo sẽ tập trung hiện thực hóa các trụ cột kiến trúc trên, với ba đường hướng chính: (1) Đồng bộ và hợp nhất Email vào một giao diện timeline xuyên suốt; (2) Tự động tóm tắt và trích xuất vấn đề từ các luồng email dài để giảm "quá tải thông tin"; và (3) Đặt nền móng kiến trúc, cả ở mức dữ liệu lẫn dịch vụ, để sẵn sàng mở rộng đa kênh trong các giai đoạn phát triển tiếp theo.

## 1.4 Mục tiêu và phạm vi báo cáo

### Mục tiêu

Trên nền tảng bối cảnh và lỗ hổng thị trường đã phân tích, báo cáo đặt ra các mục tiêu kỹ thuật cụ thể sau đây:

- Phân tích và thiết kế kiến trúc của một hệ thống quản lý giao tiếp tập trung, lấy Email làm lõi, theo mô hình microservice linh hoạt, phù hợp với bối cảnh sử dụng thực tế của các đội ngũ vừa và nhỏ.
- Hiện thực một PoC với các module chính: Backend (Next.js), Frontend (Next.js/React) và AI Service (FastAPI), thể hiện rõ cách các thành phần tương tác với nhau trong một kiến trúc monorepo.
- Xây dựng cơ chế đồng bộ hóa email với Gmail và hợp nhất dữ liệu liên lạc vào cơ sở dữ liệu MongoDB dưới dạng các thực thể Contact, Conversation và Timeline Event.
- Phát triển chức năng AI (sử dụng OpenAI API làm giải pháp ban đầu) để tóm tắt luồng email và gợi ý phản hồi theo ngữ cảnh, minh họa khả năng ứng dụng AI theo hướng hỗ trợ ra quyết định.
- Thủ nghiệm và đánh giá PoC thông qua các kịch bản dữ liệu giả lập, tập trung vào trải nghiệm người dùng, độ ổn định của kiến trúc và tiềm năng mở rộng trong các giai đoạn tiếp theo.

### Phạm vi của PoC

Do giới hạn về thời gian và nguồn lực trong khuôn khổ một báo cáo tốt nghiệp, PoC được xây dựng với phạm vi rõ ràng, tập trung vào những thành phần cốt lõi nhất nhưng vẫn phản ánh đúng tinh thần kiến trúc đề xuất:

- **Trong phạm vi (In-scope):**

- Hệ thống chỉ tập trung vào tích hợp và xử lý kênh **Email**, chưa triển khai kết nối thực tế với các kênh IM khác.
- Chức năng AI sử dụng API của bên thứ ba (OpenAI) để xử lý tóm tắt và gợi ý phản hồi, chưa triển khai mô hình tự huấn luyện.
- Giao diện người dùng (Frontend) tập trung vào việc hiển thị timeline trao đổi, chi tiết từng email và các thông tin tóm tắt/dề xuất từ AI.
- Thủ nghiệm trên dữ liệu giả lập (synthesized data) hoặc dữ liệu thử nghiệm, nhằm đánh giá kiến trúc và trải nghiệm, không nhằm mục tiêu triển khai sản phẩm thương mại hoàn chỉnh.

- **Ngoài phạm vi (Out-of-scope):**



- Chưa tích hợp trực tiếp với các API của kênh nhắn tin (Zalo, WhatsApp, Telegram...).  
Báo cáo chỉ dừng ở mức thiết kế và chuẩn bị kiến trúc sẵn sàng cho việc này.
- Chưa triển khai các mô hình AI local (như Ollama, mô hình self-host) do hạn chế về tài nguyên.
- Chưa phát triển các tính năng nâng cao như AI agent đa bước, hệ thống phân quyền và xác thực người dùng (authentication/authorization) ở mức sản phẩm.



## 2 Phân tích và Đặc tả Yêu cầu

Dựa trên mục tiêu xây dựng một nền tảng quản lý giao tiếp tập trung với Email là lõi (Email-centric), chương này trình bày bức tranh tổng thể về **yêu cầu hệ thống** dưới góc nhìn vừa kỹ thuật vừa nghiệp vụ. Trong phạm vi PoC (Proof of Concept), hệ thống được giả định vận hành theo mô hình *single-tenant* (một người dùng hoặc một tài khoản tổ chức), ưu tiên kiểm chứng tính khả thi của kiến trúc và năng lực xử lý dữ liệu kết hợp AI, hơn là giải quyết các bài toán multi-tenant ở quy mô lớn.

Để đảm bảo tính cỗ định, chương này tổ chức lại yêu cầu thành hai nhóm chính: **yêu cầu chức năng** và **yêu cầu phi chức năng**, thay vì tách quá nhiều tiểu mục nhỏ. Trong từng nhóm, các yêu cầu sẽ được mô tả theo góc nhìn "module lõi" của kiến trúc (Email, Contact, Timeline, AI, Da kenh và Hạ tầng).

### 2.1 Yêu cầu chức năng (Functional Requirements)

Nhóm yêu cầu chức năng mô tả *hệ thống cần làm gì* để hỗ trợ người dùng quản lý email và thông tin liên lạc một cách tập trung, có ngữ cảnh và được tăng cường bởi AI.

#### (1) Lõi quản lý Email và đồng bộ hai chiều

Email là kênh giao tiếp bắt buộc và là nguồn dữ liệu chính của hệ thống.

- FR-01 – Đồng bộ Email gần thời gian thực (near real-time sync):** Hệ thống cần tích hợp với Gmail API và sử dụng cơ chế *Push Notification/Webhook* thay vì *polling* thủ công. Khi có email mới tới hộp thư gốc, một sự kiện sẽ được đẩy tới backend, từ đó ghi nhận vào cơ sở dữ liệu và hiển thị lên giao diện Timeline trong khoảng thời gian mục tiêu dưới 5 giây. Cơ chế này không chỉ giúp giảm độ trễ mà còn tiết kiệm tài nguyên mạng và chi phí API.
- FR-02 – Soạn thảo và gửi Email trực tiếp:** Người dùng có thể soạn email ngay trong giao diện hệ thống với hỗ trợ *Rich Text* (in đậm, in nghiêng, bullet, trích dẫn) và đính kèm tệp. Email gửi đi từ nền tảng phải đồng bộ ngược lại vào Gmail để đảm bảo lịch sử ở hai phía luôn nhất quán, tránh tình trạng "mỗi nơi một mảnh".
- FR-03 – Quản lý trạng thái Email và phản ánh lại máy chủ:** Hệ thống cho phép đánh dấu *đã đọc/chưa đọc, lưu trữ* (archive) hoặc gắn nhãn (labels/tags) đối với từng email. Các thay đổi này được đồng bộ ngược (two-way sync) về Gmail, giúp người dùng có thể sử dụng luân phiên cả giao diện gốc lẫn nền tảng PoC mà không bị lệch trạng thái.

#### (2) Dòng thời gian giao tiếp (Communication Timeline)

Timeline là nơi thể hiện giá trị "Email-centric nhưng có ngữ cảnh".

- FR-04 – Hiển thị Inbox và Timeline theo Thread:** Hệ thống cần cung cấp giao diện Inbox hiển thị danh sách các thread email, sắp xếp theo thời gian của tin nhắn gần nhất. Mỗi thread hiển thị thông tin tóm tắt (snippet, người gửi, thời gian). Khi người dùng chọn một thread, hệ thống hiển thị toàn bộ lịch sử trao đổi theo dạng timeline, kết hợp với kết quả phân tích AI (nếu có). Đây là tính năng cốt lõi giúp người dùng có cái nhìn tổng quan về từng cuộc hội thoại.



- **FR-05 – Cập nhật giao diện tức thời (real-time UI update):** Khi backend nhận Webhook từ Gmail hoặc khi một tác vụ AI hoàn tất tóm tắt, giao diện Timeline phải được cập nhật tự động qua WebSocket hoặc cơ chế push tương đương, giúp người dùng luôn thấy trạng thái mới nhất mà không phải F5 thủ công.

### (3) Chức năng AI: tóm tắt, gợi ý phản hồi và hợp nhất hồ sơ

Module AI vận hành như một *AI microservice* độc lập, giao tiếp với backend qua HTTP/REST.

- **FR-06 – Tự động khởi tạo, làm giàu và hợp nhất hồ sơ Contact (AI-assisted Contact Management):** Khi xuất hiện một địa chỉ email mới trong luồng dữ liệu, backend ghi nhận thông tin kỹ thuật cơ bản (email, thời gian, header), còn AI service chạy nền để suy luận thêm metadata: tên hiển thị, ngôn ngữ thường sử dụng, domain liên quan tới tổ chức nào. Ngoài ra, dựa trên lịch sử giao tiếp và các đặc trưng như domain, chữ ký, cách xưng hô, AI service có thể đề xuất gộp nhiều địa chỉ email khác nhau vào cùng một Contact duy nhất. Người dùng xem xét và chấp nhận/từ chối gợi ý, sau đó hệ thống thực hiện hợp nhất nền để quy về một hồ sơ.
- **FR-07 – Tóm tắt luồng trao đổi (Thread Summarization):** Đối với các chuỗi email dài (thread), hệ thống theo yêu cầu người dùng gửi nội dung thread tới AI Service để sinh ra một bản tóm tắt ngắn gọn (summary) kèm theo danh sách "Key Issues" và "Action Required". Bản tóm tắt này được lưu lại trong DB, gắn với thread tương ứng và hiển thị trực tiếp trên giao diện Timeline để người dùng nắm nhanh "bức tranh tổng thể".
- **FR-08 – Gợi ý phản hồi thông minh (Smart Reply Suggestion):** Khi người dùng mở một email gần nhất trong thread, AI Service dựa trên nội dung email hiện tại cộng với ngữ cảnh lịch sử để đề xuất 2-3 phương án trả lời (reply candidates). Các gợi ý này ở dạng có thể chỉnh sửa lại trước khi gửi, đảm bảo vai trò AI là hỗ trợ (augmented intelligence) chứ không thay thế hoàn toàn con người.

### (4) Mở rộng đa kênh và kiến trúc adapter

Mặc dù PoC chỉ tích hợp Email, kiến trúc cần sẵn sàng cho việc bổ sung các kênh IM.

- **FR-09 – Định nghĩa interface chuẩn cho Message/Conversation:** Hệ thống phải xây dựng các interface trừu tượng (ví dụ: Message, Conversation) độc lập với nguồn kinh. Điều này cho phép về sau có thể thêm các adapter cho Zalo, Telegram, WhatsApp mà không phải viết lại logic lõi (core business logic). PoC có thể minh họa cấu trúc này bằng các stub hoặc module giả lập.

## 2.2 Yêu cầu phi chức năng (Non-Functional Requirements)

Yêu cầu phi chức năng mô tả *hệ thống cần vận hành như thế nào* về mặt hiệu năng, bảo mật và khả năng mở rộng, để đảm bảo PoC không chỉ chạy được mà còn phản ánh đúng định hướng kiến trúc cho tương lai.

### (1) Hiệu năng và trải nghiệm người dùng

- **Độ trễ đồng bộ Email:** Thời gian từ khi email tới hộp thư gốc đến khi xuất hiện trên Timeline được đặt mục tiêu dưới 5 giây trong điều kiện mạng ổn định, nhờ cơ chế Webhook và xử lý bất đồng bộ ở backend.



- **Xử lý AI bất đồng bộ:** Các tác vụ AI như tóm tắt và gợi ý phản hồi có thể kéo dài từ vài giây đến hàng chục giây. Do đó, toàn bộ pipeline AI phải được thực hiện theo mô hình async, sử dụng hàng đợi (Redis Queue) để không chặn luồng xử lý chính. Người dùng vẫn nhìn thấy email mới gần như ngay lập tức, trong khi kết quả AI được cập nhật bổ sung sau.

### (2) Bảo mật và quản lý cấu hình

- **Quản lý thông tin nhạy cảm qua môi trường:** Các thông tin như API key của nhà cung cấp LLM (Google Gemini, OpenAI), thông tin OAuth Client của Gmail, chuỗi kết nối (connection string) tới MongoDB và Redis phải được cấu hình qua biến môi trường (.env), không được ghi cứng trong mã nguồn. Điều này vừa tuân thủ thực hành bảo mật tốt, vừa giúp PoC dễ dàng triển khai trên nhiều môi trường khác nhau.
- **Phân tách quyền truy cập (ở mức PoC):** Mặc dù PoC không đi sâu vào bài toán authentication/authorization phức tạp, kiến trúc phải sẵn sàng để gắn thêm lớp xác thực (ví dụ: OAuth2, JWT) ở các lớp API mà không cần thay đổi cấu trúc tổng thể.

### (3) Khả năng mở rộng và triển khai

- **Containerization toàn bộ thành phần:** Frontend (Next.js), Backend (Next.js API Routes), AI Service (FastAPI), Database (MongoDB) và Redis đều được đóng gói thành các container độc lập, được điều phối bởi Docker Compose. Cách tiếp cận này giúp mô phỏng khá sát môi trường triển khai thực tế và tạo tiền đề để sau này chuyển sang các nền tảng container orchestration (như Kubernetes) nếu cần.
- **Mở rộng theo chiều ngang (horizontal scalability) ở mức kiến trúc:** Dù PoC không bắt buộc chạy ở quy mô lớn, các lựa chọn công nghệ (Node.js event-driven, FastAPI async, Redis Queue) đều hướng tới khả năng nhân bản (scale-out) trong tương lai: có thể tăng số replica của Backend hoặc AI Service mà không phải thay đổi code nhiều.

## 2.3 Quyết định công nghệ và Chiến lược kiến trúc

Dựa trên yêu cầu ở trên, báo cáo đưa ra một số quyết định công nghệ và chiến lược kiến trúc trọng yếu.

### (1) Backend hướng sự kiện với Node.js/Next.js

Backend sử dụng **Next.js API Routes** trong môi trường Node.js. Bài toán là *I/O-bound*: hệ thống phải tiếp nhận webhook từ Gmail, kết nối tới MongoDB/Redis, đồng thời duy trì kênh WebSocket tới nhiều client.

So với Java (Spring Boot) hay Go, Node.js cho tốc độ phát triển PoC nhanh hơn, cấu hình đơn giản, hệ sinh thái npm phong phú. So với các framework Python hướng web truyền thống, mô hình event loop của Node.js phù hợp hơn cho việc xử lý đồng thời nhiều kết nối mạng. Đồng thời, dùng TypeScript ở cả frontend và backend giúp chia sẻ kiểu dữ liệu, giảm lỗi tích hợp.

### (2) MongoDB cho dữ liệu bán cấu trúc

Dữ liệu email là bán cấu trúc (semi-structured), các trường header và metadata đa dạng. **MongoDB** cho phép lưu trữ trực tiếp các document JSON từ Gmail API cùng với kết quả phân tích AI (summary, key\_issues, sentiment,...) mà không cần schema cứng như RDBMS. Điều này



giúp PoC linh hoạt thay đổi mô hình dữ liệu trong giai đoạn khám phá (exploration), đồng thời vẫn đảm bảo hiệu năng ghi/đọc ở mức chấp nhận được.

### (3) AI Microservice với Python/FastAPI

Module AI được tách ra thành **FastAPI service** viết bằng Python để tận dụng hệ sinh thái AI/ML. Việc tách microservice này giúp:

- Tránh làm nghẽn luồng xử lý của backend khi AI phải gọi các API có độ trễ cao (Google Gemini, OpenAI).
- Dễ dàng thay đổi nhà cung cấp AI hoặc chuyển sang mô hình local (HuggingFace, Ollama) trong tương lai mà không ảnh hưởng tới backend.

### (4) Hàng đợi và xử lý bất đồng bộ với Redis

Redis được sử dụng như một **message queue** đơn giản để hiện thực mô hình Producer–Consumer giữa backend và AI service. Khi có email mới, backend chỉ lưu dữ liệu, đẩy một job vào Redis rồi trả về kết quả tối thiểu cho frontend. AI service đọc job từ hàng đợi, xử lý tóm tắt/gợi ý và cập nhật lại DB, sau đó kích hoạt thông báo real-time cho người dùng.

### (5) Cách tiếp cận AI: Prompt Engineering và Contextualization

Trong giai đoạn PoC, hệ thống áp dụng hướng tiếp cận *Prompt Engineering* với output có cấu trúc (structured JSON) thay vì fine-tuning. Các prompt được thiết kế để AI trả về cả nội dung tóm tắt lẫn thông tin có cấu trúc (sentiment, key\_issues, action\_required,...) nhằm phục vụ cho các logic hiển thị hoặc cảnh báo sau này. Đồng thời, AI luôn được cung cấp ngữ cảnh ở cấp độ thread thay vì từng email đơn lẻ, giúp kết quả tóm tắt và gợi ý phản hồi sát với diễn biến thực tế của cuộc trao đổi.



### 3 Cơ sở công nghệ và lựa chọn Tech stack

#### 3.1 Frontend stack: Next.js (React) và TypeScript

Frontend của hệ thống là lớp giao diện tương tác trực tiếp với người dùng, chịu trách nhiệm hiển thị danh sách thread (Inbox) và nội dung trao đổi theo từng thread (Timeline). Trong bối cảnh web app hiện đại, **Next.js** là một framework dựa trên **React** cung cấp cấu trúc routing theo file, hỗ trợ kết hợp nhiều chế độ render (client/server) và tích hợp tốt với hệ sinh thái JavaScript/TypeScript. **TypeScript** là lớp kiểm định (static typing) trên JavaScript, giúp mô tả cấu trúc dữ liệu rõ ràng và giảm lỗi tích hợp khi Frontend trao đổi dữ liệu với Backend. Ngoài ra, **TailwindCSS** được sử dụng như một utility-first CSS framework giúp tăng tốc độ dựng UI và duy trì tính nhất quán về style trong giai đoạn PoC.

Với các tiêu chí của đề tài (đội phát triển nhỏ, cần kiểm chứng nhanh, và khả năng mở rộng về sau), Frontend cần đảm bảo luồng điều hướng rõ ràng (Inbox → Thread detail) và có khả năng mở rộng UI khi bổ sung các chức năng AI (hiển thị summary, gợi ý reply) hoặc đa kênh.

**So sánh các lựa chọn tiếp cận.** Bảng 1 so sánh ba lựa chọn phổ biến cho việc triển khai Frontend web app.

Table 1: So sánh các lựa chọn Frontend cho PoC Email-centric

Tiêu chí	Next.js (React)	Angular
Mục tiêu phù hợp	Web app nhiều màn hình; routing rõ ràng; dễ mở rộng	Enterprise app; quy chuẩn mạnh
Tốc độ phát triển PoC	Cao (routing, cấu trúc dự án sẵn)	Trung bình (boilerplate và learning curve cao hơn)
Hệ sinh thái	Rất rộng (React/Next.js ecosystem)	Rộng nhưng thiên về full framework
Độ phức tạp	Trung bình; phù hợp team nhỏ	Cao; phù hợp team lớn/chuẩn hóa chặt
Khả năng mở rộng	Tốt; dễ chuẩn hóa layout/routing	Tốt nhưng phụ thuộc tổ chức dự án
Nhận xét trong bối cảnh đề tài	Cân bằng tốt giữa tốc độ và cấu trúc	Overkill cho PoC; chi phí học/triển khai cao

**Kết luận lựa chọn.** Báo cáo chọn **Next.js (React)** và **TypeScript** cho Frontend nhằm tối ưu tốc độ phát triển PoC và đảm bảo tính nhất quán kiểu dữ liệu khi giao tiếp với Backend. TypeScript giúp giảm lỗi tích hợp do sai lệch cấu trúc dữ liệu (data shape) giữa các API và UI, đồng thời hỗ trợ refactor an toàn khi hệ thống mở rộng.

#### 3.2 Backend orchestration: Next.js API Routes và mô hình microservice

Backend là lớp điều phối (orchestration layer) của hệ thống: tiếp nhận request từ Frontend, gọi Gmail API để đồng bộ dữ liệu, truy vấn MongoDB/Redis và gọi AI Service khi cần xử lý ngôn ngữ tự nhiên. Đây là workload *I/O-bound* (nhiều kết nối mạng, độ trễ phụ thuộc dịch vụ bên



ngoài), do đó tiêu chí lựa chọn ưu tiên sự đơn giản khi triển khai, khả năng xử lý đồng thời tốt, và tích hợp chặt với Frontend.

Trong PoC, Backend được xây dựng bằng **Next.js API Routes** (Node.js + TypeScript). Bên cạnh đó, phần AI được tách thành **AI microservice** riêng bằng **FastAPI** (Python) để tận dụng hệ sinh thái AI/LLM và cô lập độ trễ của LLM khỏi luồng xử lý chính.

**So sánh các lựa chọn tiếp cận.** (a) **So sánh framework Backend.** Bảng 2 tóm tắt ba lựa chọn phổ biến.

Table 2: So sánh các lựa chọn framework Backend cho PoC

Tiêu chí	Next.js API Routes	NestJS/Express	FastAPI monolith
Mục tiêu phù hợp	PoC nhanh; TypeScript end-to-end	Backend Node.js chuyên biệt; kiến trúc rõ (NestJS)	Backend Python; hợp khi AI là trung tâm
Tốc độ phát triển PoC	Cao (cùng ecosystem với Frontend)	Trung bình-cao (tách repo, setup nhiều hơn)	Cao cho AI; trung bình cho web app tổng thể
Tích hợp với Frontend	Rất tốt (cùng Next.js/TypeScript)	Tốt nhưng cần chuẩn hoá DTO riêng	Cần cầu nối kiểu dữ liệu với TS (OpenAPI/SDK)
I/O-bound	Tốt (Node.js event loop)	Tốt (Node.js)	Tốt (async FastAPI)
Nhận xét trong bối cảnh đề tài	Giảm chi phí phối hợp; phù hợp PoC	Hợp cho hệ backend lớn; có thể nặng nề cho PoC	Hợp nếu chọn Python làm lõi; kém thuận lợi cho TS end-to-end

(b) **So sánh kiến trúc tích hợp AI.** Bảng 3 so sánh Monolith và Microservice tách AI.

Table 3: So sánh hai lựa chọn kiến trúc tích hợp AI

Tiêu chí	Monolith (Backend + AI chung)	Microservice tách AI (FastAPI)
Dộ cô lập độ trễ LLM	Thấp; request người dùng dễ bị chặn	Cao; Backend điều phối, AI xử lý riêng
Khả năng scale	Scale chung; dễ lãng phí tài nguyên	Scale độc lập theo tải AI
Phụ thuộc công nghệ	Thường phải chọn một hệ sinh thái chính	Cho phép kết hợp TS (web) + Python (AI)
Dộ phức tạp vận hành	Thấp hơn ở PoC	Tăng nhẹ (giao tiếp service-to-service)
Nhận xét trong bối cảnh đề tài	Đơn giản nhưng kém linh hoạt về sau	Cân bằng tốt giữa PoC và định hướng mở rộng

**Kết luận lựa chọn.** PoC chọn **Next.js API Routes** làm lớp điều phối (orchestration layer) và tách **FastAPI** thành một **AI microservice**. Cách tiếp cận này vừa tận dụng lợi thế TypeScript ở luồng web app, vừa tận dụng ecosystem Python cho AI, đồng thời phù hợp với định hướng kiến trúc phân tán trình bày ở Chương 4.



### 3.3 Cơ sở dữ liệu: NoSQL và MongoDB cho dữ liệu Email bán cấu trúc

Dữ liệu email có tính bán cấu trúc (semi-structured): phần header và metadata đa dạng; nội dung có thể là plain text hoặc HTML; và hệ thống cần lưu kèm kết quả AI (summary, key issues, action required). Trong hệ thống Email-centric, dữ liệu tự nhiên nhất để truy vấn thường là **Thread** và **Message**, đồng thời schema có thể thay đổi nhanh trong giai đoạn PoC khi hoàn thiện mô hình dữ liệu.

**MongoDB** là một document database lưu trữ dữ liệu dưới dạng document (gần với JSON), phù hợp khi cần lưu trữ linh hoạt, dễ mở rộng trường dữ liệu và thuận tiện khi ingest dữ liệu từ các API như Gmail. Ngược lại, các hệ quản trị quan hệ (RDBMS) như **MySQL**, **PostgreSQL**, **SQL Server** mạnh ở ràng buộc dữ liệu, transaction, và mô hình quan hệ chặt, nhưng có thể tăng độ phức tạp schema khi dữ liệu biến thiên mạnh.

**So sánh các lựa chọn tiếp cận.** Bảng 4 so sánh ba lựa chọn triển khai phổ biến cho tầng dữ liệu.

Table 4: So sánh các lựa chọn cơ sở dữ liệu cho dữ liệu Email

Tiêu chí	MongoDB (Document)	PostgreSQL (RDBMS)	MySQL/SQL Server (RDBMS)
Cấu trúc dữ liệu	Rất tốt (schema linh hoạt)	Tốt nếu thiết kế schema kỹ; cần mapping	Tốt nếu schema ổn định; cần mapping
Tốc độ iterate PoC	Cao (thêm trường nhanh)	Trung bình (migration/DDL)	Trung bình (migration/DDL)
Ràng buộc dữ liệu	Trung bình (ở mức ứng dụng)	Mạnh (constraints, transaction)	Mạnh (constraints, transaction)
Truy vấn quan hệ	Trung bình; join hạn chế	Rất tốt (SQL, join)	Rất tốt (SQL, join)
Nhận xét trong bối cảnh đề tài	Phù hợp ingest Gmail + lưu summary	Hợp nếu ưu tiên reporting/join phức tạp	Hợp cho hệ thống doanh nghiệp truyền thống

**Kết luận lựa chọn.** Báo cáo chọn **MongoDB** để lưu trữ dữ liệu email và metadata do phù hợp với dữ liệu bán cấu trúc, hỗ trợ mở rộng schema nhanh, và thuận lợi cho chiến lược *embed summary* nhưng *reference messages* nhằm tránh giới hạn kích thước document.

### 3.4 Cache/Queue: Redis cho PoC và định hướng mở rộng

Hệ thống cần một thành phần nhẹ để hỗ trợ cache và phối hợp tác vụ bất đồng bộ trong tương lai. **Redis** là một in-memory data store phổ biến, có thể dùng cho cache và một số mô hình hàng đợi đơn giản; đồng thời Redis hỗ trợ Pub/Sub phù hợp cho các cơ chế cập nhật tức thời ở mức thiết kế.

Trong bối cảnh PoC, Redis được ưu tiên vì đơn giản trong triển khai và vận hành (chạy được qua Docker), đủ đáp ứng nhu cầu cache/queue cơ bản. Trong triển khai đầy đủ, tùy mức yêu cầu reliability và throughput, hệ thống có thể cân nhắc message broker chuyên dụng.



**So sánh các lựa chọn tiếp cận.** Bảng 5 so sánh Redis với các lựa chọn message broker phổ biến.

Table 5: So sánh Redis, RabbitMQ và Kafka trong bối cảnh PoC

Tiêu chí	Redis	RabbitMQ	Kafka
Triển khai PoC	Rất tốt (setup nhanh)	Tốt nhưng cần cấu hình nhiều hơn	Kém, cồng kềnh
Message reliability	Trung bình (tuỳ pattern)	Cao (ack/retry/routing)	Cao (log-based, replay)
Vận hành/giám sát	Đơn giản	Trung bình	Cao
Throughput/scalability	Tốt ở mức vừa	Tốt	Rất tốt (streaming lớn)
Nhận xét trong bối cảnh đề tài	Dủ cho PoC; phù hợp cache/PubSub	Hợp cho production queue chuẩn	Hợp cho event streaming quy mô lớn

**Kết luận lựa chọn.** PoC chọn **Redis** như một nền tảng đa dụng cho cache/queue, giúp đơn giản hoá vận hành. Trong triển khai đầy đủ, Redis vẫn hữu ích cho cache và Pub/Sub, trong khi message broker chuyên dụng có thể được cân nhắc khi yêu cầu reliability và throughput tăng.

### 3.5 Triển khai PoC: Docker Compose và khả năng chuyển tiếp

Mục tiêu triển khai PoC là đảm bảo môi trường chạy đồng nhất và tái lập (reproducible) trên nhiều máy, giảm rủi ro sai lệch phiên bản của MongoDB/Redis và cấu hình mạng giữa các service. **Docker** cho phép đóng gói ứng dụng và phụ thuộc thành container; **Docker Compose** cung cấp cách mô tả nhiều container và network bằng cấu hình, phù hợp với hệ thống có nhiều service.

**So sánh các lựa chọn tiếp cận.** Bảng 6 so sánh ba cách triển khai môi trường cho PoC và cho production.

Table 6: So sánh các lựa chọn triển khai môi trường

Tiêu chí	Docker Compose	Chạy local không container	Kubernetes
Tính tái lập	Cao	Thấp-trung bình	Cao
Thiết lập PoC	Nhanh	Nhanh ban đầu, dễ lặp	Chậm; cấu hình nhiều
Vận hành/giám sát	Đơn giản	Phụ thuộc máy người dùng	Phức tạp (production-grade)
Khả năng mở rộng	Trung bình; đủ cho demo	Hạn chế	Rất tốt
Nhận xét trong bối cảnh đề tài	Phù hợp nhất cho PoC	Chỉ hợp demo nhỏ; khó tái lập	Hợp production; overkill cho PoC



**Kết luận lựa chọn.** Báo cáo chọn **Docker Compose** để mô phỏng hạ tầng tối thiểu (MongoDB, Redis) cho PoC. Khi hệ thống mở rộng, kiến trúc hiện tại có thể chuyển tiếp sang các nền tảng orchestration như Kubernetes với chi phí thay đổi thấp do các thành phần đã được container hóa.

### 3.6 Bối cảnh AI trong các hệ thống giao tiếp

Trong Chương 1, báo cáo đã phân tích hai vấn đề cốt lõi trong môi trường giao tiếp đa kênh hiện nay: (i) Sự phân mảnh dữ liệu liên lạc giữa nhiều kênh khác nhau và (ii) Hiện tượng quá tải thông tin khi khối lượng email và tin nhắn tăng nhanh theo thời gian. Dưới góc nhìn kỹ thuật, đây là hai thách thức mà AI, đặc biệt là các mô hình xử lý ngôn ngữ tự nhiên (Natural Language Processing – NLP), tỏ ra phù hợp để hỗ trợ.

Trong khoảng 5 năm trở lại đây, sự xuất hiện của các LLM như GPT-3, GPT-4 (OpenAI), PaLM 2 và Gemini (Google), Claude (Anthropic), LLaMA/Llama 2/Llama 3 (Meta) đã tạo ra bước nhảy vọt về khả năng hiểu và sinh ngôn ngữ tự nhiên. Các mô hình này không chỉ dừng ở việc phân loại hay trích xuất thông tin đơn lẻ, mà có thể tóm tắt văn bản dài, trả lời câu hỏi có ngữ cảnh, gợi ý phản hồi, thậm chí lập kế hoạch hành động theo chuỗi bước. Các nhà cung cấp lớn như OpenAI, Google, Anthropic, Amazon Bedrock đã cung cấp các API thương mại ổn định để khai thác năng lực này trong ứng dụng thực tế.

Đối với một hệ thống Email-centric, AI có thể đóng vai trò ở nhiều phương diện:

- Ở phương diện **hiểu nội dung (Understanding)**, AI phân tích nội dung email và tin nhắn để nhận diện chủ đề chính, yêu cầu công việc, độ ưu tiên, cảm xúc.
- Ở phương diện **tóm tắt (Summarization)**, AI rút gọn các chuỗi trao đổi dài thành vài đoạn ngắn, giúp người dùng nắm bối cảnh nhanh chóng mà không phải đọc lại toàn bộ.
- Ở phương diện **hỗ trợ quyết định (Decision Support)**, AI gợi ý câu trả lời, soạn sẵn bản nháp email, hoặc đề xuất các công việc tiếp theo (gửi nhắc nhở, đặt lịch hẹn, yêu cầu thông tin bổ sung...).
- Ở phương diện **tổ chức lại dữ liệu (Knowledge Organization)**, AI giúp gắn tag, phân cụm, hợp nhất hồ sơ liên lạc (Contact Unification) dựa trên nội dung trao đổi, thay vì chỉ dựa trên địa chỉ email tĩnh.

Tuy nhiên, để khai thác AI một cách hiệu quả và bền vững, kiến trúc hệ thống cần lựa chọn hình thức triển khai phù hợp (Local vs. Cloud), cũng như kỹ thuật tích hợp LLM tương ứng với từng yêu cầu chức năng.

### 3.7 Phân loại giải pháp AI: Local vs. Cloud

#### Tìm hiểu về Local AI và Cloud AI

Các giải pháp AI hiện nay có thể được chia thành hai nhóm chính:

- **Cloud AI:** mô hình AI được vận hành trên hạ tầng đám mây của các nhà cung cấp lớn như OpenAI (Azure/OpenAI Service), Google Cloud (Vertex AI, Gemini API), Amazon Web Services (Amazon Bedrock, Amazon Titan), Anthropic (Claude)<sup>1</sup>. Ứng dụng khách gửi request tới API từ xa (over-the-wire), nhận kết quả thông qua giao thức HTTP(S) hoặc WebSocket.

<sup>1</sup>Xem tài liệu chính thức của OpenAI, Google Cloud, AWS, Anthropic về các dịch vụ LLM API.



- **Local AI / On-premise AI:** mô hình AI được tải về và chạy trên hạ tầng do tổ chức tự quản lý (máy chủ GPU nội bộ, cluster Kubernetes, hoặc thậm chí máy trạm cá nhân). Ví dụ: triển khai các mô hình mã nguồn mở như Llama 3 (Meta), Mistral, Mixtral, Phi-3 (Microsoft) thông qua các framework như Hugging Face Transformers, vLLM, Ollama hoặc các hệ thống suy luận tối ưu hóa.

Về bản chất, cả hai nhóm đều dựa trên các kiến trúc LLM tương tự (transformer-based), nhưng khác nhau ở vị trí *deploy* (đám mây công cộng vs. hạ tầng tự quản lý), mô hình chi phí và cách tiếp cận về bảo mật, tuân thủ quy định (compliance).

### Ưu điểm, nhược điểm và ví dụ tiêu biểu

**Cloud AI.** Các dịch vụ Cloud AI mang lại nhiều lợi ích cho giai đoạn PoC và cả triển khai sản phẩm:

- **Ưu điểm:**

- *Chất lượng mô hình cao:* Các nhà cung cấp như OpenAI, Google, Anthropic liên tục huấn luyện và tinh chỉnh những mô hình mới (ví dụ GPT-4.1, GPT-4o, Gemini 1.5, Claude 3) với chất lượng vượt trội so với nhiều mô hình mã nguồn mở cùng thời điểm theo các benchmark công khai (MMLU, GSM8K, BIG-Bench).
- *Hạ tầng sẵn có và khả năng mở rộng:* Người dùng không phải quản lý cluster GPU, không phải lo về scaling khi số lượng request tăng, vì hạ tầng đã được ẩn sau lớp API.
- *Thời gian triển khai nhanh:* Việc tích hợp chỉ cần gọi API qua HTTP/REST hoặc WebSocket, phù hợp cho các đội nhỏ muốn kiểm chứng nhanh ý tưởng.
- *Dịch vụ đi kèm phong phú:* Nhiều nền tảng cung cấp thêm tính năng như embedding, RAG-as-a-service, function calling, tool calling, monitoring và logging tập trung.

- **Nhược điểm:**

- *Chi phí vận hành theo usage:* Chi phí phụ thuộc vào số token đầu vào/đầu ra, số lần gọi API và model tier. Với khối lượng email lớn hoặc cần tóm tắt lịch sử dài thường xuyên, chi phí có thể tăng nhanh nếu không có chiến lược tối ưu (cache, batch, giới hạn độ dài).
- *Phụ thuộc vào bên thứ ba:* Hệ thống phụ thuộc vào SLA, chính sách giá, chính sách dữ liệu và độ ổn định của nhà cung cấp.
- *Quan ngại về dữ liệu:* Dù các nhà cung cấp lớn thường cam kết không dùng dữ liệu của khách hàng để huấn luyện thêm mà không có opt-in, nhiều tổ chức vẫn có yêu cầu nghiêm ngặt về việc không đưa dữ liệu nhạy cảm (như hợp đồng, hồ sơ ứng viên) ra ngoài hạ tầng nội bộ.

**Local AI / On-premise AI.** Local AI lại mang đến một tập ưu/nhược điểm khác:

- **Ưu điểm:**

- *Kiểm soát dữ liệu tốt hơn:* Tất cả dữ liệu và kết quả suy luận được xử lý trong hạ tầng nội bộ, phù hợp với các yêu cầu tuân thủ như GDPR, yêu cầu của ngành tài chính, y tế hoặc các doanh nghiệp có chính sách bảo mật nghiêm ngặt.



- *Mô hình chi phí linh hoạt:* Sau khi đầu tư hạ tầng (phần cứng, GPU), chi phí suy luân biên (Marginal cost) cho mỗi request có thể thấp hơn so với Cloud AI nếu khối lượng sử dụng lớn và liên tục.
- *Tùy biến sâu:* Tổ chức có thể tinh chỉnh mô hình (Fine-tune), cài thêm thành phần kiểm soát, lọc nội dung, hoặc kết hợp nhiều mô hình theo cách riêng.

- **Nhược điểm:**

- *Yêu cầu hạ tầng và chuyên môn:* Vận hành Local AI đòi hỏi kiến thức về quản trị hệ thống, tối ưu suy luận (Inference optimization), cũng như cập nhật bảo mật cho các thành phần liên quan.
- *Khó bắt kịp độ tiến hóa của mô hình:* Các mô hình mã nguồn mở mới liên tục xuất hiện, nhưng vẫn thường chậm hơn 1–2 thế hệ so với mô hình thương mại hàng đầu về chất lượng tổng thể.
- *Chi phí đầu tư ban đầu cao:* Với doanh nghiệp nhỏ hoặc PoC, việc đầu tư GPU chuyên dụng có thể không kinh tế so với việc dùng API Cloud theo mức sử dụng.

### Bảng so sánh Local AI và Cloud AI

Bảng 7 tóm tắt một số khía cạnh so sánh chính giữa Cloud AI và Local AI trong bối cảnh một nền tảng Email-centric.

Table 7: So sánh Cloud AI và Local AI trong bối cảnh hệ thống Email-centric

Tiêu chí	Cloud AI	Local AI / On-premise
Chi phí ban đầu	Thấp, trả theo mức sử dụng (pay-as-you-go)	Cao, đầu tư hạ tầng (GPU, server) ban đầu
Chi phí dài hạn	Có thể cao nếu khối lượng request lớn, cần tối ưu qua cache và batch-ing	Có thể rẻ hơn nếu tận dụng tối đa hạ tầng đã đầu tư
Dộ trễ (latency)	Phụ thuộc mạng Internet và vị trí data center; thường ổn định nhưng có <i>network overhead</i>	Có thể thấp hơn nếu hạ tầng đặt gần ứng dụng; phụ thuộc tối ưu suy luận
Bảo mật và quyền riêng tư	Dữ liệu đi qua hạ tầng bên thứ ba; phụ thuộc vào cam kết và chính sách của nhà cung cấp	Dữ liệu nằm trong hạ tầng nội bộ; dễ đáp ứng các yêu cầu tuân thủ nghiêm ngặt
Khả năng mở rộng	Rất cao, do hạ tầng được ẩn sau API	Phụ thuộc tài nguyên vật lý; cần cơ chế scale-out thủ công hoặc qua Kubernetes
Cập nhật mô hình	Tự động được hưởng lợi từ các phiên bản model mới	Cần chủ động cập nhật mô hình, kiểm thử lại hệ thống
Mức độ kiểm soát	Ít kiểm soát nội bộ đối với kiến trúc và tham số mô hình	Kiểm soát sâu, có thể tinh chỉnh, thêm guardrail và pipeline riêng

Đối với PoC trong báo cáo này, việc sử dụng Cloud AI (cụ thể là Google Gemini API) là lựa chọn hợp lý để rút ngắn thời gian phát triển và tận dụng mức chi phí linh hoạt cho cá nhân/nhóm nhỏ, trong khi kiến trúc vẫn được thiết kế mở để có thể thay thế hoặc bổ sung Local AI trong tương lai khi nhu cầu bảo mật và tối ưu chi phí tăng cao.



### 3.8 Các phương pháp tiếp cận LLM

#### Phương pháp Prompt Engineering

Prompt Engineering là kỹ thuật thiết kế và điều chỉnh nội dung yêu cầu (prompt) gửi tới LLM nhằm đạt được kết quả mong muốn, ổn định và dễ kiểm soát mà không cần thay đổi tham số bên trong mô hình. Các nhà cung cấp lớn như OpenAI, Google (Gemini), Anthropic đều khuyến nghị coi prompt như một *lớp lập trình* trên mô hình, với các kỹ thuật như chain-of-thought, few-shot learning, role specification<sup>2</sup>.

Trong bối cảnh nền tảng Email-centric, Prompt Engineering có một số vai trò quan trọng:

- Ràng buộc mô hình tạo ra các tóm tắt email ngắn gọn, tập trung vào các key issues, tránh *hallucination*.
- Yêu cầu mô hình trả về kết quả ở dạng JSON có cấu trúc (ví dụ: summary, sentiment, key\_issues, action\_required) để backend dễ xử lý tiếp.
- Hướng mô hình sinh gợi ý phản hồi (smart reply) phù hợp với vai trò và ngữ cảnh giao tiếp (tuyển dụng, chăm sóc khách hàng, hỗ trợ kỹ thuật...).

**Ví dụ prompt tóm tắt chuỗi email.** Dưới đây là một ví dụ rút gọn về prompt dùng cho tác vụ FR-07 (tóm tắt luồng trao đổi) trong hệ thống:

```
1 You are an assistant that summarizes email threads for a recruiter.  
2  
3 Input: a list of emails in chronological order.  
4  
5 Task:  
6 - Produce a concise summary (3–5 sentences) focusing on:  
7   * current status of the conversation  
8   * key decisions and agreements  
9   * open questions or pending actions  
10 - Return JSON with fields: 'summary', 'key_issues', 'action_required'.  
11  
12 Emails:  
13 {{thread_text}}
```

Với cách thiết kế này, hệ thống có thể lấy kết quả JSON, lưu trực tiếp vào MongoDB (cùng với bản ghi email thread) và đẩy một bản tóm tắt ngắn lên giao diện Timeline.

**Ví dụ prompt gợi ý phản hồi (smart reply).** Đối với FR-08 (gợi ý phản hồi), prompt có thể được thiết kế như sau:

```
1 You are an assistant helping a recruiter reply to candidates.  
2  
3 Given the latest email from the candidate and the conversation context,  
4 propose 2–3 short, polite reply options.  
5  
6 Requirements:  
7 - Keep each reply under 150 words.  
8 - Match the tone: professional, friendly, concise.  
9 - Do NOT invent facts (salary, company policy) if they are not present.  
10  
11 Return JSON: { "replies": [ "...", "...", "..."] }.  
12  
13 Context:
```

<sup>2</sup>Xem hướng dẫn Prompt Engineering từ OpenAI, Google Gemini trong tài liệu chính thức của họ.



```
14 {{conversation_context}}
15
16 Latest_email:
17 {{latest_email}}
```

Prompt Engineering có ưu điểm là **không cần dữ liệu huấn luyện riêng và triển khai nhanh**, rất phù hợp với giai đoạn PoC. Tuy nhiên, nhược điểm là kết quả đôi khi thiếu ổn định, phụ thuộc vào model version và khó đảm bảo hành vi tuyệt đối nhất quán trong mọi trường hợp.

### Fine-tuning và LoRA

Fine-tuning là quá trình tiếp tục huấn luyện một mô hình đã được huấn luyện sẵn (pre-trained model) trên một tập dữ liệu hẹp hơn, mang tính miền (domain-specific) hoặc tác vụ (task-specific), nhằm điều chỉnh hành vi của mô hình phù hợp hơn với nhu cầu cụ thể. Các nền tảng như OpenAI, Google Vertex AI / Gemini, AWS Bedrock đều cung cấp dịch vụ fine-tuning cho một số dòng mô hình nhất định.

LoRA (Low-Rank Adaptation) là một kỹ thuật fine-tuning hiệu quả, trong đó chỉ một số ma trận hạng thấp (low-rank matrices) được huấn luyện thêm và *gắn* vào mô hình gốc, giúp giảm đáng kể số tham số cần cập nhật và tài nguyên tính toán so với việc tinh chỉnh toàn bộ mô hình. LoRA phổ biến trong cộng đồng mô hình mã nguồn mở (ví dụ với Llama, Mistral) do tính linh hoạt và tiết kiệm chi phí.

**Quy trình fine-tuning tổng quát.** Quy trình điển hình gồm các bước:

1. Thu thập và làm sạch dữ liệu huấn luyện phù hợp với tác vụ (ví dụ: cặp email thread – summary chuẩn, hoặc email – reply chất lượng cao).
2. Chuyển đổi dữ liệu về định dạng mà nhà cung cấp yêu cầu (JSONL, CSV, v.v.).
3. Cấu hình bài toán fine-tuning (model base, số epoch, learning rate, batch size, tiêu chí dừng...).
4. Chạy quá trình huấn luyện trên hạ tầng cloud hoặc local GPU.
5. Dánh giá mô hình mới trên tập kiểm thử (validation/test set) và so sánh với mô hình gốc.
6. Triển khai (deploy) mô hình đã tinh chỉnh vào pipeline suy luận của hệ thống.

**Khi nào nên dùng fine-tuning / LoRA với hệ thống này.**

- **Nên dùng nếu:**

- Doanh nghiệp có lượng lớn dữ liệu nội bộ chất lượng cao (email được gắn nhãn, các bản tóm tắt chuẩn, reply chuẩn theo quy định), đủ để huấn luyện.
- Cần hành vi rất đặc thù, ví dụ giọng văn thương hiệu, quy tắc nghiệp vụ cố định, hoặc phải tuân thủ các template chặt chẽ.
- Mô hình mặc định dù đã sử dụng Prompt Engineering nhưng vẫn cho kết quả không đủ ổn định.

- **Không nên (hoặc chưa cần) dùng nếu:**

- Hệ thống đang ở giai đoạn PoC, dữ liệu chưa đủ nhiều và chưa được gắn nhãn kỹ.



- Chi phí và độ phức tạp để triển khai, giám sát một mô hình fine-tuned là quá lớn so với lợi ích tăng thêm.
- Prompt Engineering kết hợp với RAG đã cho kết quả đáp ứng yêu cầu.

Dối với nền tảng Email-centric trong báo cáo này, ưu tiên **Prompt Engineering + RAG** trên các mô hình có sẵn, trong khi fine-tuning và LoRA được xem như hướng mở rộng khi hệ thống đã đi vào vận hành thực tế và tích lũy dữ liệu.

### Retrieval-Augmented Generation (RAG)

RAG (Retrieval-Augmented Generation) là kiến trúc kết hợp giữa mô hình sinh ngôn ngữ (LLM) và một hệ thống truy vấn tri thức (retrieval system). Thay vì yêu cầu mô hình “ghi nhớ” tất cả tri thức trong tham số, RAG tách riêng phần tri thức thành một kho lưu trữ (ví dụ: vector database, Elasticsearch, hoặc đơn giản là collection trong MongoDB), sau đó:

1. Nhận input (ví dụ: câu hỏi, yêu cầu tóm tắt, yêu cầu phân tích).
2. Truy vấn (retrieve) các đoạn văn bản liên quan trong kho dữ liệu (email lịch sử, log trao đổi, tài liệu hướng dẫn...).
3. Kết hợp input ban đầu với các đoạn trích liên quan thành một prompt đầy đủ.
4. Gửi prompt này tới LLM để sinh câu trả lời/tóm tắt, đảm bảo kết quả dựa trên tri thức cập nhật và chính xác hơn.

RAG được nhiều công ty lớn (như Microsoft, Meta, Google) khuyến nghị như một chiến lược chính để xây dựng ứng dụng AI doanh nghiệp, vì cho phép cập nhật tri thức liên tục mà không cần fine-tuning mô hình gốc.

Trong hệ thống Email-centric, RAG đặc biệt phù hợp với các bài toán:

- Tóm tắt chuỗi email dài: thay vì đưa toàn bộ lịch sử vào prompt, hệ thống có thể chỉ truy vấn các đoạn email quan trọng (dựa trên thời gian, người gửi, từ khóa) rồi đưa vào LLM.
- Trả lời câu hỏi theo bối cảnh: ví dụ, “Khách hàng A đang ở trạng thái deal nào?” hoặc “Lần trao đổi gần nhất với ứng viên B là gì?” có thể được trả lời bằng cách truy vấn lịch sử giao tiếp tương ứng rồi cho LLM tóm tắt.
- Kết hợp nhiều nguồn: tích hợp email, ghi chú nội bộ, ticket hỗ trợ vào một kho chung để LLM có cái nhìn toàn diện khi sinh câu trả lời.

## 3.9 AI Agent và khả năng mở rộng trong tương lai

### Khái niệm và kiến trúc tổng quát

AI Agent là một mô hình (hoặc tập hợp mô hình) AI không chỉ sinh văn bản đơn lẻ, mà còn có khả năng lập kế hoạch, tương tác với môi trường và sử dụng công cụ để hoàn thành một mục tiêu ở mức cao (high-level goal). Nhiều nghiên cứu và sản phẩm gần đây (OpenAI o1, ReAct, AutoGPT, LangChain Agents, Microsoft AutoGen) đề xuất kiến trúc agent trong đó LLM đóng vai trò bộ não (reasoning engine), còn các “tool” là các API hoặc hàm có thể gọi được (function calling / tool calling).

Một kiến trúc agent điển hình bao gồm các thành phần:

- **Planner:** mô-đun (thường là LLM) có nhiệm vụ phân rã yêu cầu lớn thành chuỗi các bước nhỏ hơn, có thể thực thi được.



- **Tools / Actions:** tập các hành động mà agent có thể thực thi, ví dụ gọi API backend để truy vấn email, tạo contact, gửi nhắc nhở, ghi chú timeline.
- **Memory:** cơ chế lưu trữ trạng thái dài hạn hoặc ngắn hạn (short-term / long-term memory), bao gồm lịch sử trao đổi với người dùng, các hành động đã thực hiện, kết quả trung gian.
- **Environment:** thế giới bên ngoài mà agent tương tác, ở đây là hệ thống Email-centric, cơ sở dữ liệu, Redis, các dịch vụ khác.
- **Feedback Loop:** cơ chế đánh giá kết quả và điều chỉnh kế hoạch nếu cần (tự đánh giá hoặc có sự can thiệp của con người – human-in-the-loop).

Trong bối cảnh doanh nghiệp, các agent thường được thiết kế dưới dạng *co-pilot* cho nhân viên, nghĩa là agent hỗ trợ một phần công việc chứ không tự động hóa hoàn toàn, nhằm đảm bảo yếu tố an toàn, tuân thủ và trải nghiệm người dùng.

### Kịch bản ứng dụng trong nền tảng Email-centric

Dựa trên kiến trúc của hệ thống PoC, một số kịch bản agent tiềm năng có thể bao gồm:

**Agent quản lý inbox (Inbox Management Agent).** Agent này có mục tiêu hỗ trợ người dùng xử lý hộp thư đến một cách hiệu quả hơn:

- Phân loại email theo mức độ ưu tiên (gấp, quan trọng, có thể đọc sau) dựa trên nội dung, lịch sử trao đổi và role của người gửi.
- Gợi ý nhóm email theo “topic” hoặc “deal” để dễ theo dõi.
- Đề xuất hành động tiếp theo (trả lời, chuyển tiếp, tạo task nội bộ, đặt lịch họp) cho từng email hoặc nhóm email.

Lúc này, planner sẽ đọc một batch email trong inbox, quyết định email nào cần gọi tool `summarizeThread`, tool nào gọi `suggestReply`, và khi nào cần ping người dùng để xác nhận.

**Agent follow-up (Follow-up Agent).** Agent follow-up chịu trách nhiệm theo dõi các thread đang treo (pending) và nhắc nhở người dùng gửi follow-up đúng thời điểm:

- Theo dõi những email chưa nhận được phản hồi sau một khoảng thời gian cấu hình.
- Sinh bản nháp email follow-up gợi ý, phù hợp ngữ cảnh (ví dụ nhắc nhẹ nhàng, hỏi thêm thông tin, chốt lịch).
- Đề xuất lịch follow-up dựa trên mức độ ưu tiên của deal hoặc ứng viên.

Agent có thể dùng RAG để nhìn lại toàn bộ bối cảnh thread, đảm bảo follow-up không gây khó chịu (spam) và mang lại giá trị.

### Agent tạo ghi chú và báo cáo (Note-taking)

**Reporting Agent.** Agent này tập trung chuyển đổi các cuộc trao đổi rời rạc thành ghi chú và báo cáo có cấu trúc:

- Tự động tạo “meeting notes” sau các cuộc trao đổi email dài hoặc sau khi người dùng gắn tag “meeting” cho một thread.
- Tổng hợp nhật ký giao tiếp (communication timeline) theo từng contact hoặc account (khách hàng, ứng viên), phục vụ cho việc họp nội bộ hoặc báo cáo quản lý.



### Ưu điểm, hạn chế và thách thức triển khai

Việc đưa AI Agent vào nền tảng Email-centric mang đến nhiều cơ hội nhưng cũng không ít thách thức:

- **Ưu điểm:**

- Tăng mức độ tự động hóa công việc lặp lại (lọc email, nhắc follow-up, tạo ghi chú), giúp người dùng tập trung vào các quyết định mang tính chiến lược.
- Khai thác tốt hơn tri thức ẩn trong lịch sử giao tiếp (implicit knowledge), nhờ khả năng tổng hợp và lập kế hoạch của agent.
- Tạo ra trải nghiệm “trợ lý cá nhân” (personal assistant) gắn liền với inbox của mỗi người.

- **Hạn chế và thách thức:**

- *Dộ tin cậy (reliability)*: Agent có thể đưa ra kế hoạch hoặc hành động không phù hợp nếu prompt, cấu hình tool hoặc dữ liệu không đầy đủ. Cần cơ chế giám sát và giới hạn phạm vi hành động (ví dụ chỉ gọi ý, không tự động gửi email).
- *An toàn và quyền riêng tư*: Agent phải tuân thủ chặt chẽ chính sách truy cập dữ liệu (ai được phép xem email nào, ghi chú nào), đặc biệt khi hệ thống mở rộng sang multi-tenant.
- *Trải nghiệm người dùng (UX)*: Nếu agent đưa ra quá nhiều gợi ý không liên quan hoặc gây phiền hà, người dùng dễ mất niềm tin và tắt chức năng này. Cần thiết kế luồng tương tác mượt mà, cho phép người dùng điều chỉnh mức độ can thiệp.
- *Chi phí vận hành*: Agent thường cần nhiều lượt gọi LLM hơn so với các chức năng đơn lẻ, do phải lập kế hoạch, gọi nhiều tool và đánh giá kết quả. Điều này đặt ra bài toán tối ưu chi phí (caching, batching, chọn model phù hợp cho từng bước).

### 3.10 Hạn chế hiện tại của LLM trong bối cảnh doanh nghiệp

Mặc dù LLM mang lại nhiều cơ hội cho các hệ thống giao tiếp, vẫn tồn tại một số hạn chế cần cân nhắc khi triển khai trong bối cảnh doanh nghiệp:

- **Hiện tượng “hallucination”**: Mô hình đôi khi tạo ra thông tin không chính xác hoặc hoàn toàn bị ra nhưng trông có vẻ hợp lý. Trong môi trường doanh nghiệp, điều này có thể dẫn đến sai sót nghiệp vụ hoặc thông tin sai lệch gửi cho khách hàng nếu không có lớp kiểm tra.
- **Thiếu tính minh bạch (interpretability)**: Việc giải thích tại sao mô hình đưa ra một câu trả lời cụ thể là khó, khiến việc kiểm toán (audit) và chứng minh tuân thủ trở nên phức tạp.
- **Chi phí và hiệu năng**: Các mô hình mạnh như GPT-4.x, Claude 3 thường có chi phí sử dụng và độ trễ cao hơn so với các mô hình nhỏ. Cần có chiến lược chọn model tier phù hợp cho từng tác vụ (ví dụ dùng model nhỏ cho phân loại, model lớn cho tóm tắt phức tạp).
- **Quản lý dữ liệu và tuân thủ**: Doanh nghiệp phải đảm bảo dữ liệu gửi tới mô hình (đều là cloud hay local) tuân thủ các quy định như GDPR, các quy định nội bộ, và không làm rò rỉ thông tin nhạy cảm.



- **Phụ thuộc vào nhà cung cấp:** Trong trường hợp sử dụng Cloud AI, thay đổi về giá, chính sách hoặc sự cố dịch vụ có thể ảnh hưởng trực tiếp đến hệ thống.
- **Nhu cầu kỹ năng mới:** Đội ngũ phát triển cần bổ sung kỹ năng về Prompt Engineering, MLOps, Data Governance để vận hành hệ thống AI một cách bền vững.

## 4 Thiết kế Hệ thống

### 4.1 Kiến trúc tổng thể hệ thống

Sơ đồ kiến trúc hệ thống (Overall Architecture Diagram)

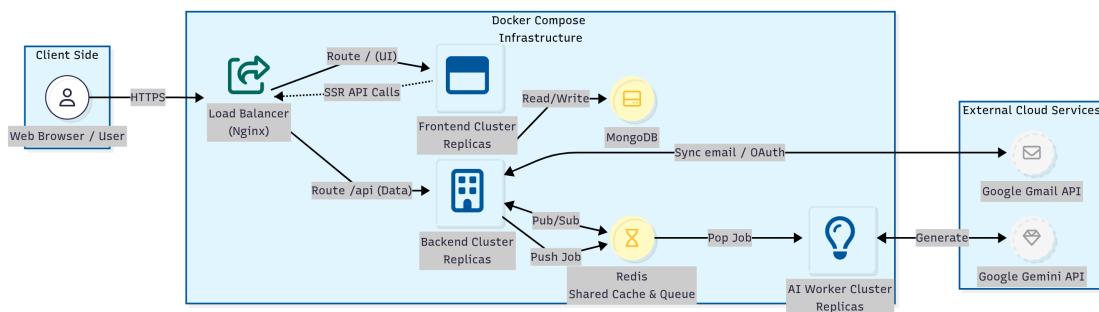


Figure 1: Kiến trúc tổng quan của hệ thống Email-centric

Hình 1 minh họa kiến trúc tổng thể của nền tảng Email-centric ở mức high-level. Hệ thống hoạt động dựa trên mô hình **\*\*Single Entry Point\*\*** (Điểm truy cập duy nhất) thông qua một Load Balancer, điều phối lưu lượng đến các cụm dịch vụ (Service Clusters) phi trạng thái phía sau. Các thành phần chính bao gồm:

- **Load Balancer / Reverse Proxy (Nginx):** Đóng vai trò là cổng giao tiếp duy nhất giữa Client và hạ tầng bên trong. Thành phần này chịu trách nhiệm:
  - Điều hướng (Routing) lưu lượng dựa trên đường dẫn: các yêu cầu trang web (/) được chuyển đến Frontend Cluster, trong khi các yêu cầu dữ liệu (/api) được chuyển đến Backend Cluster.
  - Cân bằng tải (Load Balancing) để phân phối đều request giữa các bản sao (replicas) của dịch vụ, ngăn chặn tình trạng quá tải cục bộ.
- **Frontend Cluster (Next.js):** Bao gồm các bản sao của ứng dụng Next.js chạy song song. Việc container hóa và cluster hóa Frontend đảm bảo khả năng xử lý lượng lớn yêu cầu truy cập đồng thời, đặc biệt là các tác vụ Render phía máy chủ (Server-Side Rendering - SSR). Khi thực hiện SSR, các node Frontend cũng gọi API thông qua Load Balancer để đảm bảo tính nhất quán trong việc định tuyến nội bộ.
- **Backend Cluster (Next.js API):** Là trung tâm xử lý logic nghiệp vụ, được thiết kế hoàn toàn *Stateless* (phi trạng thái). Các node Backend không lưu trữ session người dùng cục bộ mà ủy quyền cho tầng dữ liệu. Thiết kế này cho phép hệ thống tự do thêm/bớt số lượng node Backend tùy theo tải thực tế mà không gây gián đoạn dịch vụ.
- **AI Worker Cluster (FastAPI):** Là tập hợp các microservice worker chuyên trách xử lý các tác vụ AI nặng (tóm tắt thread, gợi ý phản hồi). Các worker này hoạt động theo cơ chế bất đồng bộ (Asynchronous): chúng không nhận request trực tiếp từ User mà "lắng nghe" và xử lý các tác vụ (jobs) từ hàng đợi trong Redis. Cơ chế này giúp tách biệt tải của AI ra khỏi luồng xử lý chính của người dùng.
- **Data & State Layer (Redis + MongoDB):**

- **Redis (Shared Cache & Queue):** Redis lưu trữ Session chung cho toàn bộ các cluster (giải quyết bài toán Stateless), đồng thời hoạt động như một Message Broker để điều phối hàng đợi công việc giữa Backend và AI Service.
- **MongoDB:** Lưu trữ bền vững (Persistent Storage) cho dữ liệu người dùng, email threads và siêu dữ liệu (metadata).

Kiến trúc này cũng thể hiện rõ sự tương tác với các dịch vụ bên ngoài (External Cloud Services) như Gmail API (đồng bộ dữ liệu) và Google Gemini API (xử lý ngôn ngữ tự nhiên), đảm bảo hệ thống lõi chỉ tập trung vào logic nghiệp vụ và điều phối.

Sơ đồ thể hiện rõ đường đi của dữ liệu: từ Gmail API vào Backend, được lưu trong MongoDB, sau đó được Backend cung cấp cho Frontend để hiển thị Inbox/Timeline. Khi người dùng yêu cầu tóm tắt, Backend lấy dữ liệu thread từ MongoDB, gửi sang AI Service gọi Gemini, nhận kết quả tóm tắt và lưu trở lại, rồi trả về cho Frontend.

### Biểu đồ Use Case

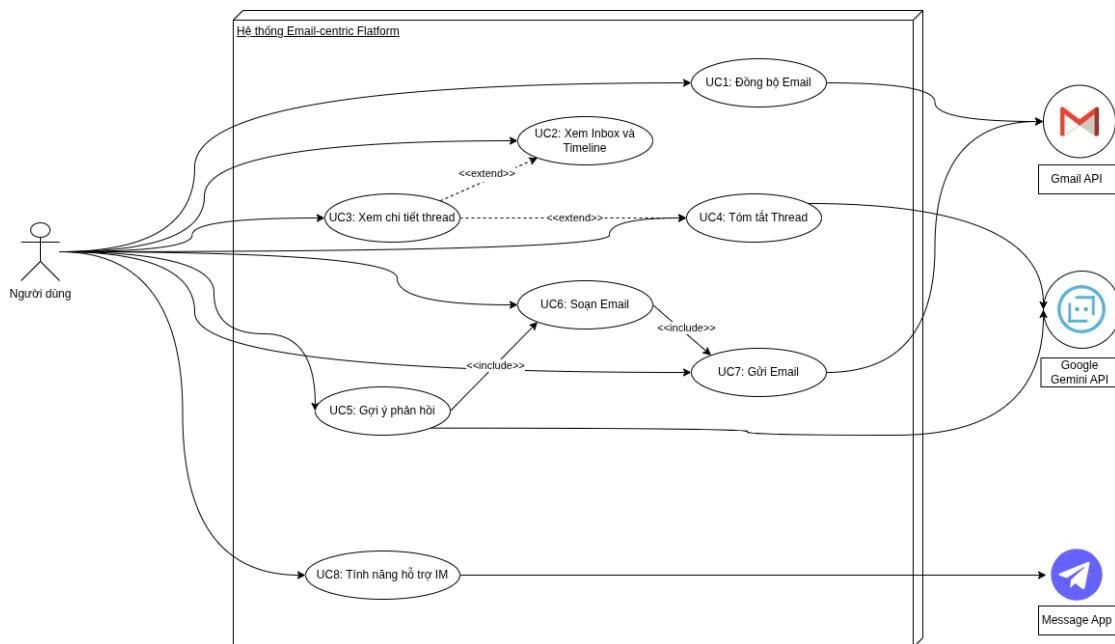


Figure 2: Biểu đồ Use Case tổng thể của hệ thống (bao gồm thiết kế mở rộng đa kênh)

Hình 2 mô tả các use case của hệ thống, làm rõ phạm vi chức năng và sự tương tác giữa người dùng với các hệ thống bên ngoài.

Các tác nhân (Actors) trong hệ thống bao gồm:

- **Người dùng (Primary Actor):** Là người trực tiếp sử dụng hệ thống để quản lý giao tiếp (ví dụ: nhân viên tuyển dụng, nhân viên hỗ trợ).
- **Gmail API (Secondary Actor):** Hệ thống bên ngoài đóng vai trò cung cấp dữ liệu email gốc và thực hiện lệnh gửi email thực tế.



- **Google Gemini API (Secondary Actor):** Hệ thống trí tuệ nhân tạo bên ngoài, cung cấp khả năng xử lý ngôn ngữ tự nhiên cho các tác vụ thông minh.
- **Instant Message App:** Các ứng dụng nhắn tin tức thời như Slack, Microsoft Teams, Facebook Messenger, được tích hợp để mở rộng kênh giao tiếp.

Các Use Case được tổ chức theo luồng nghiệp vụ như sau:

#### 1. Nhóm Quản lý và Xem thông tin (Information Retrieval):

- **UC01 – Đồng bộ Email:** Tương tác với Gmail API để cập nhật dữ liệu mới nhất về hệ thống.
- **UC02 – Xem Inbox & Timeline:** Cho phép người dùng xem danh sách tổng quan các luồng trao đổi.
- **UC03 – Xem chi tiết Thread:** Là chức năng mở rộng (*extend*) từ việc xem Inbox, cho phép đi sâu vào nội dung cụ thể của từng cuộc hội thoại.

#### 2. Nhóm Tác vụ Email (Email Actions):

- **UC06 – Soạn Email:** Cung cấp trình soạn thảo văn bản (Rich Text Editor).
- **UC07 – Gửi Email:** Thực hiện đầy email đi thông qua Gmail API. Trong thiết kế hệ thống, chức năng Soạn thảo bao hàm (*include*) chức năng Gửi như một bước hoàn tất quy trình.

#### 3. Nhóm Tính năng AI (AI-Assisted Features):

- **UC04 – Tóm tắt Thread:** Được kích hoạt từ giao diện xem chi tiết (quan hệ *extend*). Hệ thống gửi ngữ cảnh sang Google Gemini API để tạo bản tóm tắt.
- **UC05 – Gợi ý phản hồi:** AI phân tích ngữ cảnh để đề xuất câu trả lời. Use case này có quan hệ bao hàm (*include*) với UC06, nghĩa là khi chọn một gợi ý phản hồi, hệ thống sẽ tự động chuyển nội dung đó vào trình soạn thảo để người dùng tiếp tục chỉnh sửa.

#### 4. Nhóm Tích hợp Đa kênh (Multi-channel Integration - Design):

- **UC09 – Tính năng hỗ trợ IM:** Thiết kế module kết nối với các nền tảng như Zalo và Microsoft Teams, nhằm gom dữ liệu chat về cùng giao diện Timeline với Email.

### 4.2 Thiết kế luồng tương tác theo yêu cầu chức năng

Các biểu đồ sequence mô tả luồng tương tác chi tiết giữa Frontend, Backend, AI Service và các thành phần hạ tầng cho một số FR trọng tâm.

### FR-01 – Đồng bộ Email

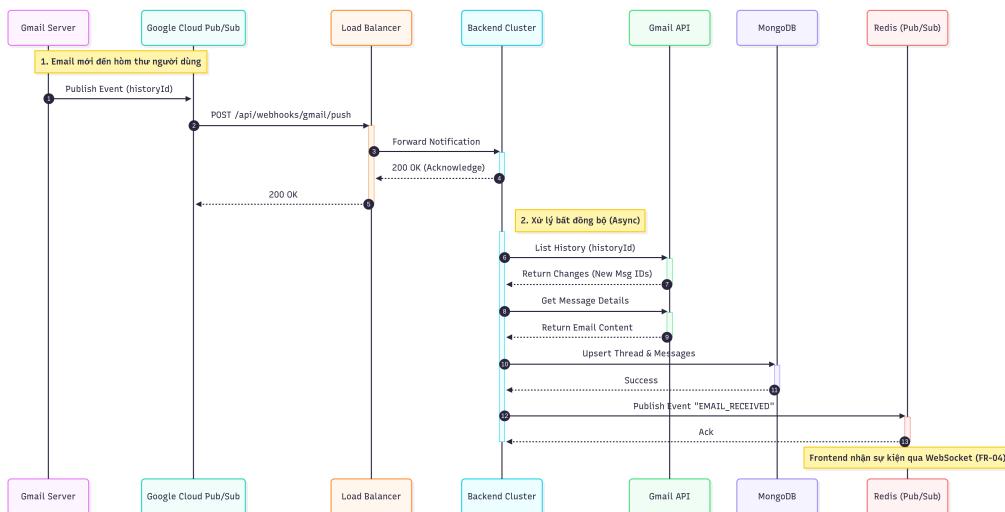


Figure 3: Biểu đồ tuần tự cho chức năng Đồng bộ Email qua Webhook (FR-01)

Để đáp ứng yêu cầu đồng bộ gần thời gian thực (Near Real-time), hệ thống được thiết kế sử dụng cơ chế **Push Notification** thông qua Google Cloud Pub/Sub thay vì kỹ thuật Polling truyền thống. Hình 3 minh họa luồng dữ liệu khi có email mới:

- Trigger Sự kiện:** Khi email mới đến hộp thư Gmail, Google Cloud Pub/Sub tự động gửi một thông báo (Webhook) chứa **historyId** đến điểm cuối công khai của hệ thống thông qua Load Balancer.
- Xác nhận và Xử lý:** Backend nhận thông báo và phản hồi mã 200 OK ngay lập tức để xác nhận đã nhận tin. Sau đó, một tiến trình xử lý ngầm (background process) được kích hoạt.
- Đồng bộ Dữ liệu (Sync):** Backend sử dụng **historyId** để truy vấn **Gmail API**, lấy danh sách các thay đổi (incremental changes) và tải về nội dung email chi tiết. Dữ liệu sau đó được lưu trữ vào **MongoDB**.
- Phát tán Sự kiện (Real-time update):** Sau khi lưu trữ thành công, Backend đẩy một sự kiện **EMAIL\_RECEIVED** vào **Redis Pub/Sub**. Các kết nối WebSocket từ Frontend sẽ lắng nghe sự kiện này để cập nhật giao diện Timeline tức thời (FR-04) mà không cần người dùng tải lại trang.

### FR-02 – Soạn thảo và gửi Email trực tiếp

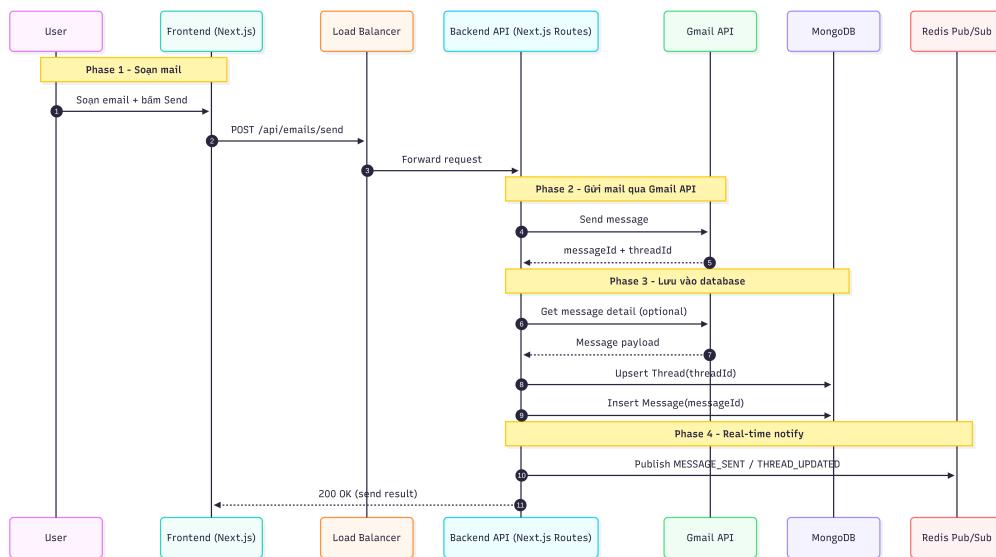


Figure 4: Biểu đồ tuần tự cho chức năng Soạn thảo và gửi Email (FR-02)

FR-02 cho phép người dùng soạn email trực tiếp trên giao diện hệ thống và gửi thông qua Gmail API. Thiết kế nhấn mạnh hai điểm: (i) Gmail vẫn là nguồn thực thi hành động gửi (source of truth cho việc gửi) và (ii) hệ thống lưu lại bản ghi gửi đi để đảm bảo Inbox/Timeline hiển thị nhất quán và có thể truy vấn nhanh.

Luồng xử lý được chia thành các phase chính như sau:

- Phase 1 - Soạn thảo & Gửi (Compose & Send):** Người dùng nhập nội dung và bấm gửi. Frontend gửi request `POST /api/emails/send` đến Backend thông qua Load Balancer.
- Phase 2 - Gửi qua Gmail API:** Backend xác thực phiên, chuyển đổi nội dung sang định dạng Gmail yêu cầu và gọi Gmail API để gửi email. Gmail trả về `messageId` và `threadId` (trường hợp reply sẽ gắn vào thread sẵn có).
- Phase 3 - Đồng bộ và Lưu trữ:** Backend tải chi tiết message vừa gửi (hoặc dùng payload đã có) để chuẩn hoá và lưu vào MongoDB dưới dạng `Message`, đồng thời cập nhật `Thread` (ví dụ `lastMessageDate`, `snippet`) để phục vụ hiển thị Inbox/Timeline.
- Phase 4 - Cập nhật giao diện:** Backend phát sự kiện cập nhật (ví dụ `MESSAGE_SENT` hoặc `THREAD_UPDATED`) qua Redis Pub/Sub để Frontend có thể cập nhật UI tức thời theo cơ chế ở FR-05.

### FR-03 – Quản lý trạng thái Email và phản ánh lại máy chủ

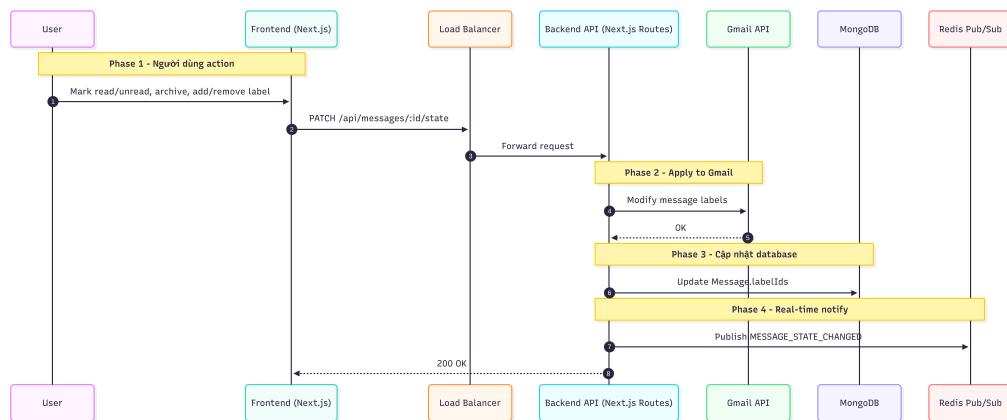


Figure 5: Biểu đồ tuần tự cho chức năng Quản lý trạng thái Email (FR-03)

FR-03 cho phép người dùng thao tác trạng thái email (đã đọc/chưa đọc, archive, labels) trên nền tảng, đồng thời phản ánh (two-way sync) thay đổi này về Gmail để đảm bảo nhất quán giữa hai phía.

Luồng xử lý gồm các phase:

- Phase 1 - Người dùng thao tác trạng thái:** Frontend gửi request (ví dụ PATCH /api/messages/:id/state) kèm danh sách labels cần thêm/bớt.
- Phase 2 - Cập nhật Gmail:** Backend gọi Gmail API (modify labels) để áp dụng thay đổi lên message tương ứng.
- Phase 3 - Cập nhật MongoDB:** Sau khi Gmail xác nhận thành công, Backend cập nhật bản ghi `Message.labelIds` trong MongoDB để UI truy vấn nhanh mà không cần gọi Gmail liên tục.
- Phase 4 - Thông báo real-time:** Backend phát sự kiện (ví dụ `MESSAGE_STATE_CHANGED`) để Frontend cập nhật UI theo FR-05.

### FR-04 – Xem Inbox và Thread Timeline

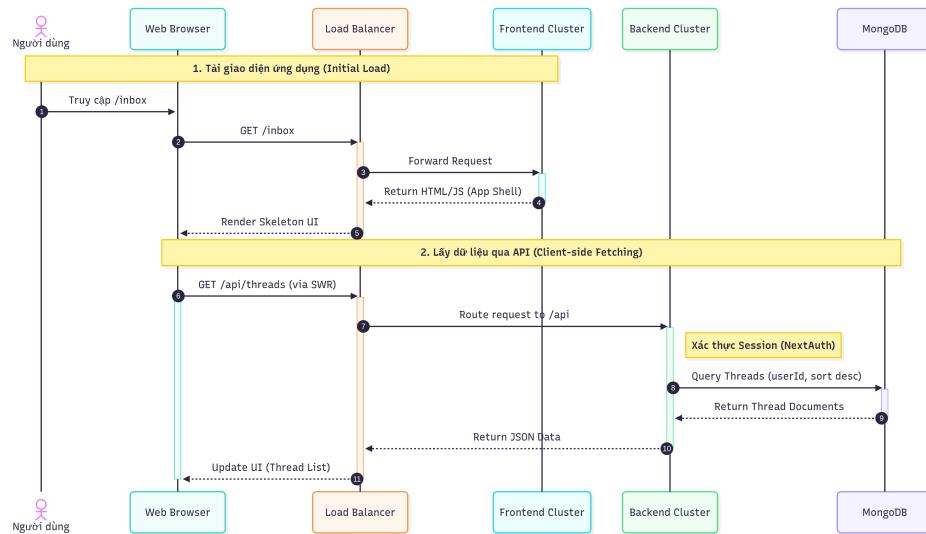


Figure 6: Biểu đồ tuần tự cho chức năng Xem Inbox (FR-04)

Chức năng hiển thị Inbox và Timeline được thực hiện theo mô hình *Client-side Data Fetching*, tận dụng khả năng của Next.js và thư viện SWR để tối ưu trải nghiệm người dùng. Hình 6 mô tả chi tiết hai phase của quá trình này:

- 1. Phase 1 - Tải giao diện (Initial Load):** Khi người dùng truy cập đường dẫn, Load Balancer điều hướng yêu cầu tới *Frontend Cluster*. Frontend trả về khung ứng dụng (App Shell) và các tài nguyên tĩnh (JS/CSS) để trình duyệt hiển thị giao diện sơ khởi (Skeleton UI) ngay lập tức.
- 2. Phase 2 - Lấy dữ liệu (Data Fetching):** Sau khi giao diện tải xong, trình duyệt (Client) tự động gửi yêu cầu API bắt đồng bộ (GET /api/thread) thông qua Load Balancer tới *Backend Cluster*. Backend thực hiện xác thực phiên làm việc, truy vấn dữ liệu từ *MongoDB*, và trả về kết quả dưới dạng JSON để Frontend cập nhật danh sách email đầy đủ.

Thiết kế này giúp giảm tải cho Frontend Server (không phải chờ dữ liệu từ DB mới render HTML) và tăng cảm giác phản hồi nhanh cho người dùng.

### FR-05 – Cập nhật giao diện tức thời (Real-time UI update)

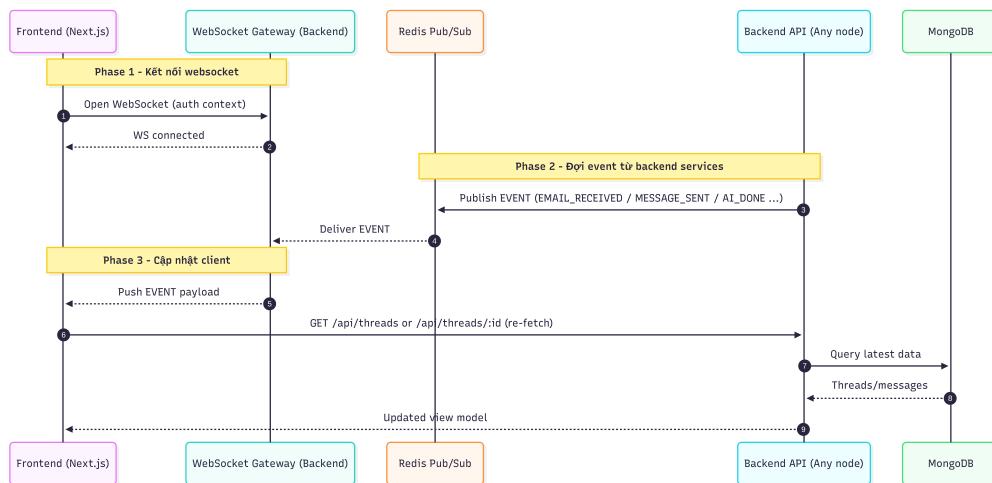


Figure 7: Biểu đồ tuần tự cho cơ chế Cập nhật giao diện tức thời (FR-05)

FR-05 mô tả cơ chế push cập nhật từ Backend tới Frontend nhằm loại bỏ thao tác tải lại trang thủ công. Thiết kế sử dụng WebSocket để giữ kết nối dài hạn với client, kết hợp Redis Pub/Sub để phát tán sự kiện giữa các node Backend trong môi trường cluster.

Các phase chính:

- Phase 1 - Thiết lập kênh real-time:** Frontend mở kết nối WebSocket tới hệ thống. Backend đăng ký (subscribe) theo ngữ cảnh người dùng.
- Phase 2 - Nhận sự kiện nội bộ:** Khi một luồng khác tạo ra cập nhật (ví dụ sync email FR-01, gửi email FR-02, đổi trạng thái FR-03, hoàn tất AI FR-07/FR-08), Backend phát sự kiện vào Redis Pub/Sub.
- Phase 3 - Push xuống client:** Backend nhận sự kiện từ Redis và đẩy thông báo xuống WebSocket. Frontend cập nhật UI và/hoặc re-fetch endpoint liên quan (threads hoặc thread detail) để đồng bộ dữ liệu.

### FR-06 – Tự động tạo & hợp nhất Contact (AI)

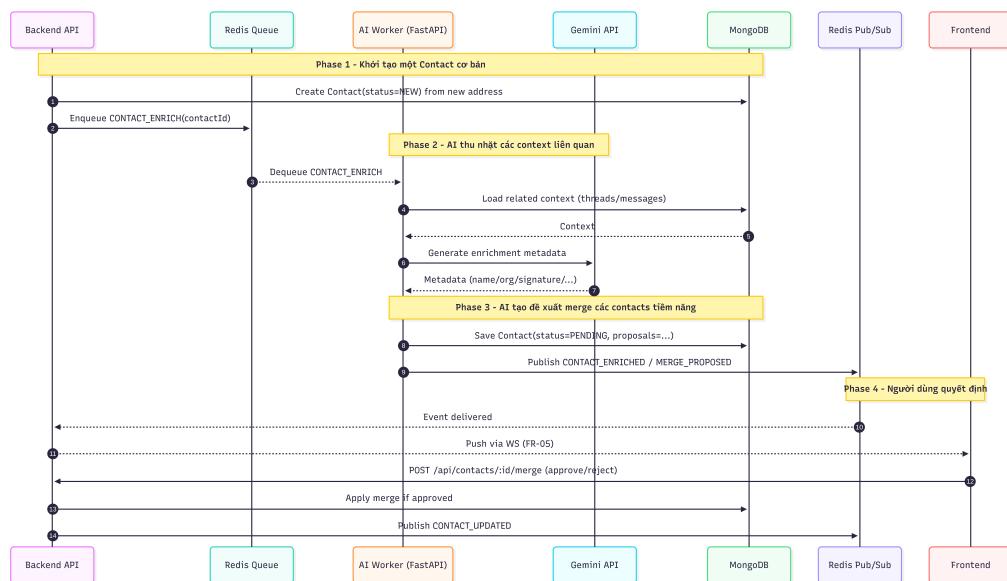


Figure 8: Biểu đồ tuần tự cho chức năng AI hỗ trợ tạo và hợp nhất Contact (FR-06)

FR-06 hướng tới tổ chức lại dữ liệu liên lạc: khi xuất hiện địa chỉ mới, hệ thống tạo Contact cơ bản và dùng AI để làm giàu thông tin, đồng thời đề xuất hợp nhất nhiều định danh về cùng một hồ sơ. Thiết kế áp dụng mô hình *human-in-the-loop*: AI chỉ **đề xuất**, còn quyết định hợp nhất do người dùng xác nhận.

Luồng xử lý gồm các phase:

- Phase 1 - Khởi tạo Contact:** Backend phát hiện địa chỉ mới từ luồng đồng bộ/gửi email và tạo bản ghi Contact ở trạng thái NEW.
- Phase 2 - AI Enrichment:** Backend đẩy một job vào Redis Queue. AI Worker lấy job, truy xuất ngữ cảnh (các email liên quan) rồi gọi Gemini để suy luận metadata (tên hiển thị, tổ chức/domain, ngôn ngữ, chữ ký, v.v.).
- Phase 3 - Propose merge:** AI Worker sinh danh sách ứng viên có thể hợp nhất (merge candidates) kèm lập luận và mức tin cậy, lưu đề xuất ở trạng thái PENDING.
- Phase 4 - Xác nhận người dùng & cập nhật UI:** Hệ thống push thông báo qua FR-05 để Frontend hiển thị gợi ý. Người dùng approve/reject; nếu approve, Backend thực hiện hợp nhất và phát sự kiện cập nhật.

### FR-07 – Tóm tắt Thread

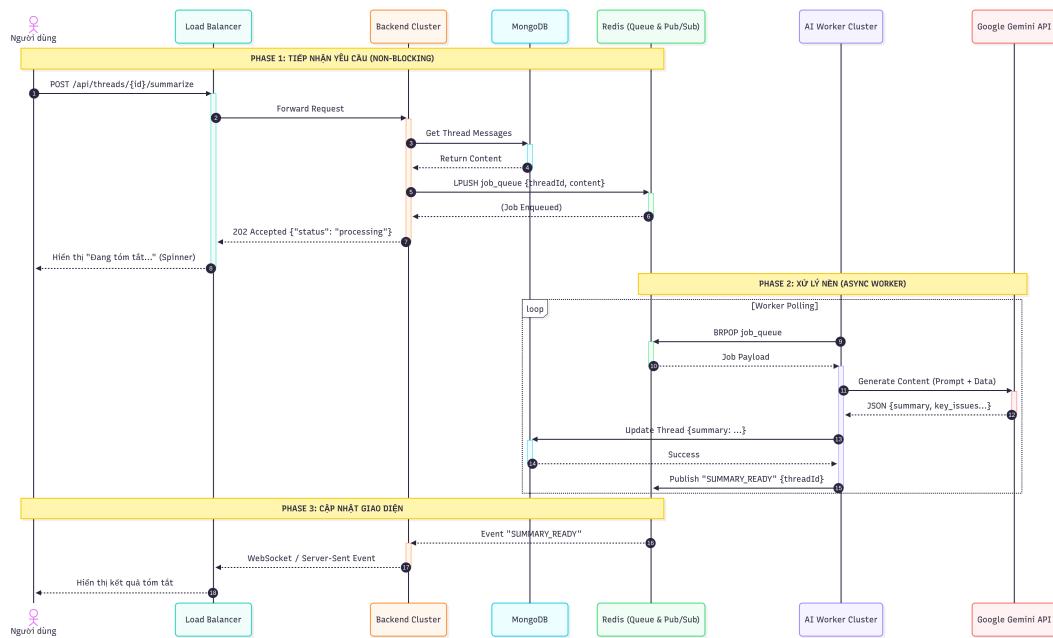


Figure 9: Biểu đồ tuần tự xử lý Tóm tắt AI bất đồng bộ (FR-07)

Việc gọi API của các mô hình ngôn ngữ lớn (LLM) như Google Gemini thường có độ trễ cao (từ vài giây đến hàng chục giây tùy độ dài văn bản) và giới hạn số lượng yêu cầu (Rate Limiting). Nếu xử lý đồng bộ (Synchronous), Backend sẽ bị chiếm dụng tài nguyên kết nối trong thời gian chờ đợi, dễ dẫn đến nghẽn cổ chai.

Do đó, chức năng này được thiết kế theo mô hình **\*\*Bất đồng bộ (Asynchronous)\*\*** sử dụng Redis làm hàng đợi trung gian. Hình 9 mô tả quy trình 3 phase:

- Phase 1 - Tiếp nhận (Dispatch):** Khi nhận yêu cầu từ người dùng, Backend Cluster chỉ thực hiện việc truy xuất nội dung thread từ *MongoDB*, đóng gói thành một bản tin (Job) và đẩy vào hàng đợi *Redis Queue*. Backend lập tức phản hồi mã 202 Accepted cho người dùng. Giao diện Frontend chuyển sang trạng thái "Đang xử lý" (Loading state) mà không bị treo.
- Phase 2 - Xử lý nền (Processing):** Các node trong *AI Worker Cluster* (FastAPI) hoạt động độc lập, liên tục "lắng nghe" hàng đợi. Khi có Job mới, Worker sẽ lấy ra xử lý, gửi ngữ cảnh tới *Google Gemini API* để sinh tóm tắt, sau đó cập nhật trực tiếp kết quả vào *MongoDB*. Thiết kế này cho phép tách biệt hoàn toàn AI khỏi luồng phục vụ người dùng chính.
- Phase 3 - Cập nhật (Notification):** Sau khi cập nhật DB thành công, Worker phát một sự kiện thông báo qua kênh *Redis Pub/Sub*. Backend nhận sự kiện này và đẩy thông báo xuống Frontend (qua *WebSocket* hoặc Frontend tự động kiểm tra lại) để hiển thị kết quả cuối cùng.

### FR-08 – Gọi ý phản hồi thông minh (Smart Reply Suggestion)

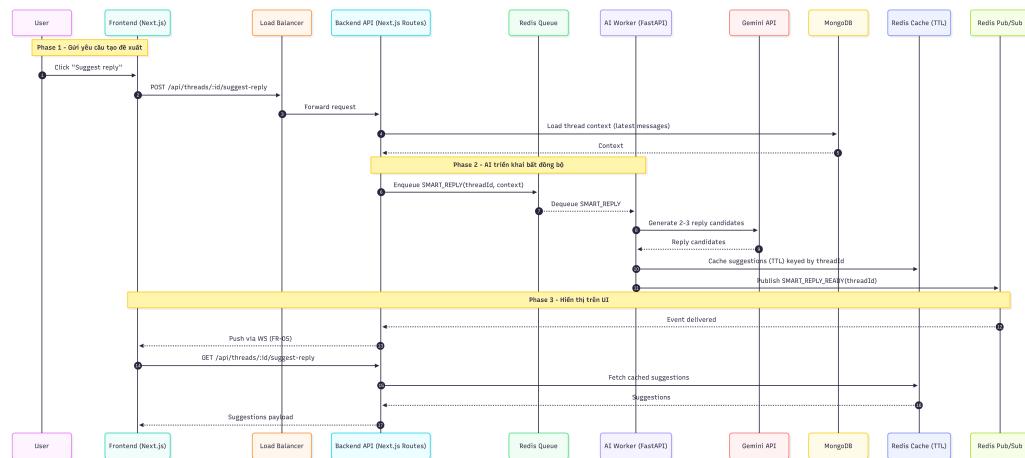


Figure 10: Biểu đồ tuần tự cho chức năng Gọi ý phản hồi thông minh (FR-08)

FR-08 đề xuất 2–3 phương án trả lời dựa trên email mới nhất và ngữ cảnh thread. Tương tự FR-07, luồng xử lý được thiết kế theo hướng bắt đồng bộ để tránh làm nghẽn Backend khi gọi LLM.

Các phase chính:

- Phase 1 - Tiếp nhận yêu cầu:** Frontend gửi request POST `/api/threads/:id/suggest-reply`. Backend truy xuất ngữ cảnh (email mới nhất + lịch sử trao đổi cần thiết) từ MongoDB.
- Phase 2 - Xử lý AI nền:** Backend đóng gói job và đẩy vào Redis Queue. AI Worker lấy job, gọi Gemini để sinh danh sách reply candidates.
- Phase 3 - Trả kết quả và cập nhật UI:** Kết quả được lưu tạm (ví dụ Redis cache theo TTL hoặc lưu bền tùy thiết kế chi tiết), sau đó hệ thống phát sự kiện qua Redis Pub/Sub để Frontend nhận và hiển thị ngay theo cơ chế FR-05.

### 4.3 Thiết kế Cơ sở dữ liệu (Database Schema)

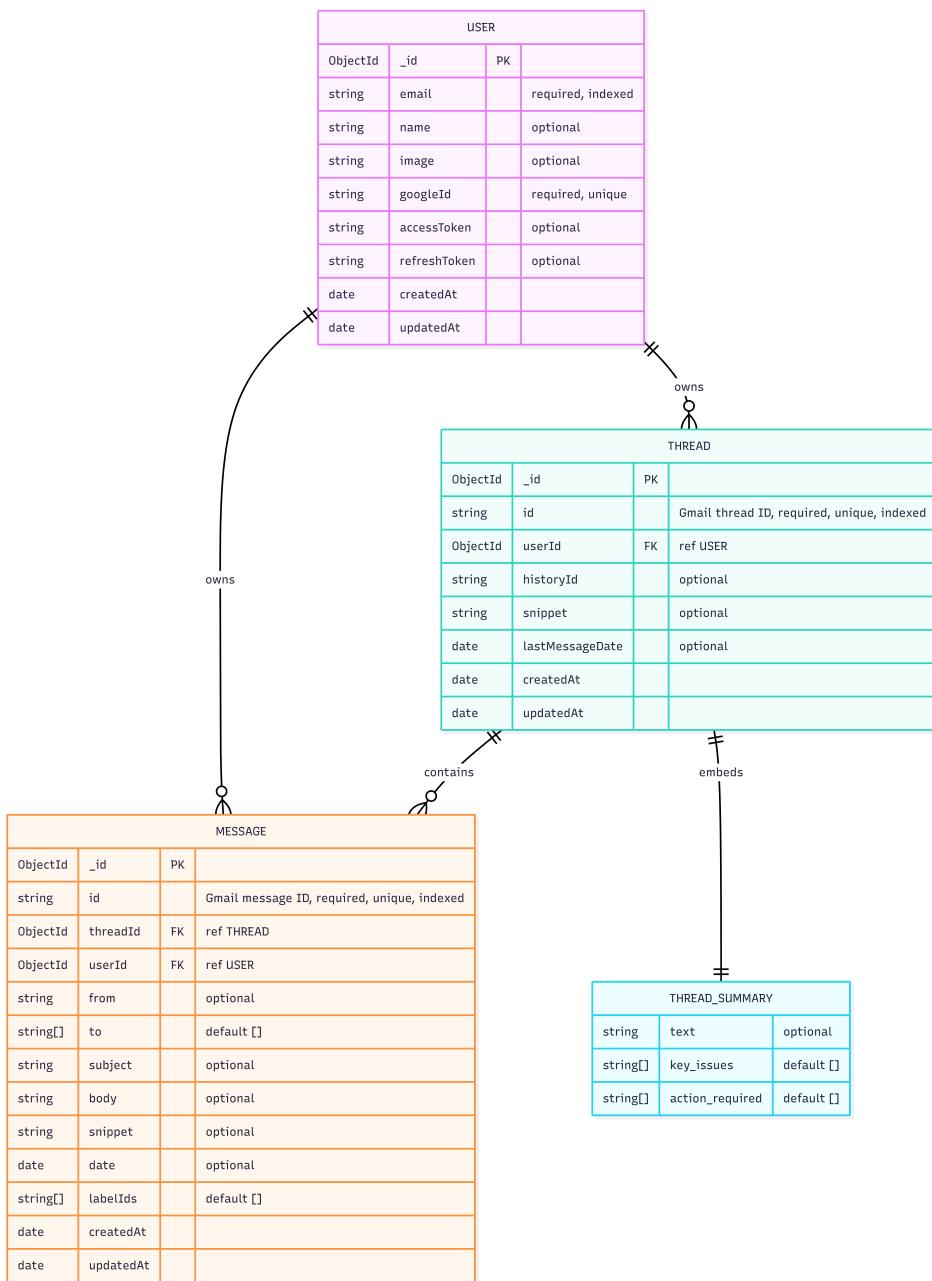


Figure 11: Lược đồ Cơ sở dữ liệu (ERD) với chiến lược Nhúng và Tham chiếu

Hệ thống sử dụng **MongoDB** làm cơ sở dữ liệu chính. Để mô tả cấu trúc lưu trữ bán cấu trúc (NoSQL), lược đồ Quan hệ Thực thể (ERD) trong Hình 11 minh họa các tập thực thể (Collections) và chiến lược liên kết dữ liệu.



Mô hình bao gồm ba thành phần chính:

- User (Collection):** Lưu trữ hồ sơ người dùng và các token xác thực OAuth2. Quan hệ giữa *User* và *Thread* là quan hệ 1-N (Một người dùng sở hữu nhiều luồng email).
- Thread (Collection) và chiến lược Embedding:** *Thread* đại diện cho một luồng hội thoại. Để tối ưu hiệu năng đọc (Read Performance) cho tính năng hiển thị tóm tắt trên Timeline (FR-07), thực thể **Summary** (chứa kết quả AI) được thiết kế theo chiến lược **Embedding (Nhúng)**. Nghĩa là dữ liệu tóm tắt được lưu trực tiếp bên trong document của *Thread* thay vì tách ra bảng riêng, giúp giảm thiểu chi phí truy vấn (No Join).
- Message (Collection) và chiến lược Referencing:** *Message* lưu trữ nội dung chi tiết của từng email. Trái ngược với *Summary*, quan hệ giữa *Thread* và *Message* sử dụng chiến lược **Referencing (Tham chiếu)**. Các *Message* được lưu trong collection riêng biệt và liên kết thông qua khóa ngoại *threadId*. Quyết định này nhằm đảm bảo khả năng mở rộng (Scalability), tránh lỗi tràn bộ nhớ document (16MB limit của MongoDB) đối với các luồng email dài chứa nhiều nội dung HTML hoặc đính kèm.

Table 8: Mô tả field-level cho collection *User*

Field	Type	Ý nghĩa	Liên quan FR
email	String	Email người dùng; dùng để định danh/tracứu nhanh (được đánh index).	FR-01-FR-08
name	String	Tên hiển thị của người dùng (phục vụ UI).	FR-04
image	String	Ảnh đại diện (phục vụ UI).	FR-04
googleId	String	Dịnh danh duy nhất từ Google OAuth (unique), dùng để liên kết phiên đăng nhập và token.	FR-01-FR-08
accessToken	String	Token truy cập Gmail API, cần để gọi các thao tác đồng bộ/gửi/sửa trạng thái email.	FR-01-FR-03
refreshToken	String	Token làm mới để gia hạn accessToken khi hết hạn, đảm bảo hệ thống hoạt động lâu dài.	FR-01-FR-03
createdAt	Date	Thời điểm tạo bản ghi (tự sinh bởi timestamps).	FR-01-FR-08
updatedAt	Date	Thời điểm cập nhật gần nhất (tự sinh bởi timestamps).	FR-01-FR-08



Table 9: Mô tả field-level cho collection *Thread* (bao gồm Summary nhúng)

Field	Type	Ý nghĩa	Liên quan FR
<code>id</code>	String	Gmail thread ID (unique + index). Dùng để đồng bộ và ánh xạ dữ liệu giữa Gmail và hệ thống.	FR-01, FR-04
<code>userId</code>	ObjectId	Tham chiếu về User. Cho phép mỗi người dùng có tập thread riêng.	FR-01, FR-04
<code>historyId</code>	String	Mốc incremental sync từ Gmail (dùng để lấy thay đổi thay vì tải toàn bộ).	FR-01
<code>snippet</code>	String	Đoạn trích ngắn giúp hiển thị nhanh trên Inbox.	FR-04
<code>lastMessageDate</code>	Date	Thời điểm email gần nhất trong thread; dùng để sắp xếp Inbox theo độ mới.	FR-04
<code>summary.text</code>	String	Nội dung tóm tắt do AI sinh; nhúng vào Thread để đọc nhanh trên UI.	FR-07
<code>summary.key_issues</code>	String[]	Danh sách vấn đề chính được trích xuất từ thread (AI).	FR-07
<code>summary.action_required</code>	String[]	Danh sách hành động cần thực hiện (AI).	FR-07
<code>createdAt</code>	Date	Thời điểm tạo bản ghi (tự sinh bởi timestamps).	FR-01-FR-08
<code>updatedAt</code>	Date	Thời điểm cập nhật gần nhất (tự sinh bởi timestamps).	FR-01-FR-08



Table 10: Mô tả field-level cho collection *Message*

Field	Type	Ý nghĩa	Liên quan FR
<b>id</b>	String	Gmail message ID (unique + index). Dùng để tránh trùng lặp khi đồng bộ.	FR-01, FR-04
<b>threadId</b>	ObjectId	Tham chiếu về Thread. Giúp truy vấn danh sách email theo thread.	FR-04
<b>userId</b>	ObjectId	Tham chiếu về User. Giúp phân tách dữ liệu theo người dùng.	FR-01, FR-04
<b>from</b>	String	Địa chỉ người gửi; dùng hiển thị và phục vụ phân tích ngữ cảnh.	FR-04, FR-06
<b>to</b>	String[]	Danh sách người nhận; dùng hiển thị và tái dựng ngữ cảnh hội thoại.	FR-04, FR-06
<b>subject</b>	String	Tiêu đề email; phục vụ hiển thị và định hướng nội dung.	FR-04
<b>body</b>	String	Nội dung email (plain text/HTML tùy cách trích xuất). Là nguồn chính cho AI tóm tắt/gợi ý reply.	FR-07, FR-08
<b>snippet</b>	String	Đoạn trích ngắn từ nội dung; hỗ trợ hiển thị nhanh.	FR-04
<b>date</b>	Date	Thời điểm gửi/nhận email; dùng để sắp xếp timeline theo thời gian.	FR-04
<b>labelIds</b>	String[]	Nhãn Gmail (ví dụ INBOX, UNREAD); là cơ sở để đồng bộ trạng thái đọc/lưu trữ.	FR-03
<b>createdAt</b>	Date	Thời điểm tạo bản ghi (tự sinh bởi timestamps).	FR-01-FR-08
<b>updatedAt</b>	Date	Thời điểm cập nhật gần nhất (tự sinh bởi timestamps).	FR-01-FR-08



## 5 Hiện thực và Thủ nghiệm PoC

### 5.1 Môi trường triển khai và cách chạy PoC

PoC được triển khai dưới dạng monorepo với cấu trúc thư mục chính:

- **apps/frontend**: ứng dụng Next.js phục vụ giao diện người dùng (Inbox, Thread Timeline, AI Summary).
- **apps/backend**: ứng dụng Next.js API đóng vai trò Backend, tích hợp Gmail API, MongoDB và AI Service.
- **apps/ai-service**: dịch vụ FastAPI tách riêng cho các tác vụ AI (tóm tắt thread, gợi ý phản hồi) sử dụng Google Gemini.
- **infra**: cấu hình Docker Compose cho MongoDB, Redis và các dịch vụ liên quan.

#### Công nghệ và phiên bản chính

Trong quá trình hiện thực, báo cáo đã sử dụng các công nghệ sau:

- **Frontend**: Next.js (React + TypeScript), TailwindCSS.
- **Backend**: Next.js API Routes (Node.js, TypeScript), NextAuth cho đăng nhập Google.
- **AI Service**: FastAPI (Python), thư viện Google Generative AI để gọi Google Gemini.
- **Cơ sở dữ liệu**: MongoDB cho lưu trữ bán cấu trúc (User, Thread, Message).
- **Hệ tầng phụ trợ**: Redis cho cache và nền tảng queue/pub-sub trong tương lai; Docker Compose để khởi tạo nhanh môi trường cục bộ.

#### Cấu hình biến môi trường

Toàn bộ các thông tin nhạy cảm (API key, OAuth client secret, connection string) đều được cấu hình thông qua file `.env` riêng cho từng service, thay vì ghi cứng trong mã nguồn:

- **apps/backend/.env**: chứa chuỗi kết nối MongoDB, thông tin OAuth2 của Google, URL tới AI Service.
- **apps/ai-service/.env**: chứa API key và tên model của Google Gemini.

#### Quy trình khởi động PoC

Để phục vụ cho việc thử nghiệm, báo cáo xây dựng một số script npm tại thư mục gốc giúp đơn giản hoá quy trình khởi động. Các bước tổng quát như sau:

1. Cài đặt dependencies Node.js ở cấp độ monorepo:

- `npm install`

2. Chuẩn bị môi trường Python cho AI Service:

- `npm run dev:setup:ai` – tạo virtualenv và cài đặt các thư viện cần thiết theo `requirements.txt`.

3. Khởi động hệ tầng dữ liệu (MongoDB, Redis) qua Docker Compose:



- `npm run dev:db`

4. Chạy đồng thời Backend, Frontend và AI Service trong môi trường phát triển:

- `npm run start:all`

Sau khi các dịch vụ khởi động thành công, có thể kiểm tra nhanh qua các endpoint sức khoẻ (health check):

- Backend: `http://localhost:4000/api/health`.
- AI Service: `http://localhost:5000/`.

Giao diện người dùng (Frontend) được truy cập qua địa chỉ `http://localhost:3000`, nơi người dùng có thể đăng nhập bằng tài khoản Google, bấm nút đồng bộ email, xem Inbox/Thread và yêu cầu tóm tắt nội dung cuộc trao đổi.

## 5.2 Tổng kết các chức năng đã hiện thực

Bảng 11 tóm tắt mức độ hiện thực hóa các yêu cầu chức năng (FR) trong phạm vi PoC, đối chiếu với thiết kế ở các chương trước.

Table 11: Tổng kết mức độ hiện thực các yêu cầu chức năng trong PoC

	Mô tả tóm tắt	Trạng thái trong PoC
FR-01	Đồng bộ Email gần thời gian thực từ Gmail về hệ thống	Đã hiện thực một phần
FR-02	Soạn thảo và gửi Email trực tiếp từ nền tảng	Chưa hiện thực
FR-03	Quản lý trạng thái Email (đã đọc/chưa đọc, archive, labels)	Chưa hiện thực
FR-04	Xem Inbox và Thread Timeline theo từng người dùng	Đã hiện thực
FR-05	Cập nhật giao diện real-time khi có email mới hoặc kết quả AI	Chưa hiện thực
FR-06	Tự động khởi tạo, làm giàu và hợp nhất hồ sơ Contact bằng AI	Chưa hiện thực
FR-07	Tóm tắt luồng trao đổi (Thread Summarization) bằng AI	Đã hiện thực
FR-08	Gợi ý phản hồi thông minh (Smart Reply Suggestion)	Chưa hiện thực
FR-09	Kiến trúc mở rộng đa kênh (Multi-channel Adapter)	Chưa hiện thực

## 6 Tổng kết

Chương này tổng kết toàn bộ kết quả thực hiện trong giai đoạn 1 (14 tuần), đồng thời đề xuất kế hoạch cho giai đoạn 2 (14 tuần) theo định hướng mở rộng đa kênh (multi-channel). Các kế hoạch được trình bày dưới dạng biểu đồ Gantt (PNG sẽ được chèn sau), trong đó mã Mermaid được lưu riêng để thuận tiện export.

### 6.1 Tổng quan giai đoạn 1 (14 tuần)

Trong giai đoạn 1, kỳ vọng ban đầu của đồ án tương đối lớn: vừa phải khảo sát thị trường, nghiên cứu nền tảng AI/LLM và công nghệ web, vừa thiết kế hệ thống chi tiết và triển khai PoC bao phủ nhiều yêu cầu chức năng (FR).

**Kế hoạch ban đầu (Planned).** Trong 14 tuần, dự kiến dành khoảng **30% thời gian cho triển khai PoC**, tương đương **4 tuần**. Phần còn lại dành cho nghiên cứu, thiết kế và viết báo cáo.

**Thực tế (Actual).** Do quản lý thời gian chưa tốt và khối lượng phần lý thuyết/báo cáo lớn hơn dự kiến, thời gian còn lại cho triển khai PoC chỉ khoảng **10%**, tương đương **1 tuần**.

Vì vậy, PoC thực tế chỉ tập trung hoàn thiện một phần các luồng cốt lõi (FR-01/FR-04/FR-07) như đã tổng kết ở Chương 5.

### 6.2 Kế hoạch ban đầu giai đoạn 1 (Planned)

Kế hoạch ban đầu được tổ chức theo 5 nhóm công việc chính: (i) nghiên cứu đề tài và khảo sát giải pháp thị trường, (ii) nghiên cứu công nghệ và các phương pháp tiếp cận AI/LLM, microservices và web app, (iii) thiết kế hệ thống (kiến trúc và database), (iv) triển khai PoC, và (v) viết báo cáo.

Để phù hợp với bối cảnh trang, biểu đồ Gantt được chia thành 2 phần: Tuần 1–7 và Tuần 8–14.

#### Gantt (Planned) – Tuần 1–7



Figure 12: Biểu đồ Gantt giai đoạn 1 (Planned) – Tuần 1–7

#### Gantt (Planned) – Tuần 8–14



Figure 13: Biểu đồ Gantt giai đoạn 1 (Planned) – Tuần 8–14

### 6.3 Kết quả thực tế giai đoạn 1 (Actual) và khó khăn

#### Những gì đã thực hiện

Các kết quả chính trong giai đoạn 1 có thể tổng hợp theo 5 nhóm công việc như sau:

- Nghiên cứu đề tài và khảo sát thị trường:** tổng hợp các hướng tiếp cận Email client/CRM hỗ trợ AI; rút ra các yêu cầu thực tiễn như đồng bộ dữ liệu, phân loại luồng hội thoại, và trợ lý tóm tắt/phản hồi.
- Nghiên cứu công nghệ liên quan:** tổng hợp các phương pháp tiếp cận AI/LLM (Prompt Engineering, fine-tuning), đánh giá lựa chọn microservices, và lựa chọn tech stack web app phù hợp cho PoC.
- Thiết kế hệ thống:** hoàn thiện thiết kế kiến trúc các module chính và thiết kế database (User–Thread–Message, embed Summary), cùng các biểu đồ tuân tự cho các FR trọng yếu.
- Triển khai PoC:** hiện thực các luồng cốt lõi gồm đồng bộ email (FR-01) ở mức PoC, hiển thị Inbox/Thread Timeline (FR-04) và tóm tắt thread bằng AI (FR-07). Các FR còn lại chủ yếu dùng ở mức thiết kế/dề xuất.
- Viết báo cáo:** hoàn thiện phần tổng quan, phân tích yêu cầu, nền tảng công nghệ, thiết kế hệ thống và hiện thực PoC.

#### Khó khăn và nguyên nhân dẫn đến kế hoạch không như dự kiến

Theo Grantt chart của giai đoạn 1, khó khăn chính không nằm ở một điểm cụ thể, mà nằm ở quản lý phạm vi và phân bổ thời gian:

- Kỳ vọng ban đầu quá rộng** so với năng lực thời gian của 14 tuần, trong khi nhiều hạng mục có độ bất định cao (Gmail integration, mô hình dữ liệu, và pipeline AI bất đồng bộ).
- Thời lượng cho phần lý thuyết và viết báo cáo bị kéo dài**, làm thời gian triển khai PoC bị co từ 4 tuần (kế hoạch) xuống còn 1 tuần (thực tế).
- Chi phí tích hợp (integration cost)** giữa nhiều thành phần (Next.js, OAuth, Gmail API, MongoDB, AI Service) khiến triển khai end-to-end cần nhiều buffer hơn dự kiến.

#### Gantt (Actual) – Tuần 1–7



Figure 14: Biểu đồ Gantt giai đoạn 1 (Actual) – Tuần 1–7

### Gantt (Actual) – Tuần 8–14



Figure 15: Biểu đồ Gantt giao đoạn 1 (Actual) – Tuần 8–14

### 6.4 Kế hoạch giao đoạn 2 (14 tuần)

Giao đoạn 2 được lập kế hoạch theo hướng **cân bằng giữa hoàn thiện phạm vi FR và các hạng mục mở rộng/kiểm thử**.

#### Gantt (Giai đoạn 2) – Tuần 1–7 (50%: hoàn tất FR-01..FR-09)

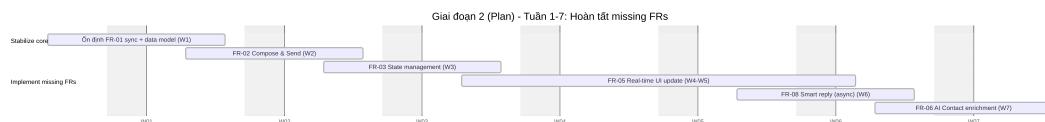


Figure 16: Biểu đồ Gantt giao đoạn 2 (Plan) – Tuần 1–7 (50% hoàn tất FR)

#### Gantt (Giai đoạn 2) – Tuần 8–14 (50%: mở rộng, kiểm thử, tổng kết)

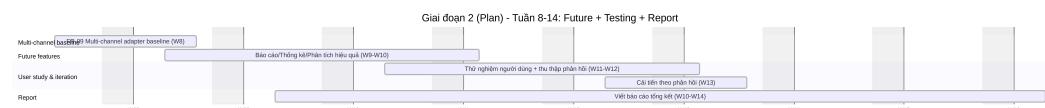


Figure 17: Biểu đồ Gantt giao đoạn 2 (Plan) – Tuần 8–14 (50% future, testing, report)

### 6.5 Rủi ro giao đoạn 2 (Risks)

Các rủi ro chính trong giao đoạn 2 và hướng giảm thiểu bao gồm:

- Rủi ro mở rộng phạm vi do multi-channel (scope creep):** dễ phát sinh yêu cầu tích hợp thêm nhiều kênh khi chưa ổn định lối email. *Giảm thiểu:* khoá phạm vi theo milestone 14 tuần; trong giao đoạn 2 chỉ triển khai FR-09 ở mức adapter baseline, ưu tiên hoàn tất FR-01..FR-08.
- Rủi ro giới hạn Gmail API (quota/rate limit) và lỗi đồng bộ:** có thể ảnh hưởng trải nghiệm nếu sync không ổn định. *Giảm thiểu:* throttling/backoff, lưu historyId đúng cách, bổ sung cơ chế retry và logging.
- Rủi ro chi phí/độ trễ LLM:** các tác vụ FR-06/FR-07/FR-08 có thể tốn thời gian và chi phí token. *Giảm thiểu:* chạy bất đồng bộ (queue), cache kết quả theo TTL, giới hạn context và chuẩn hoá prompt.



- **Rủi ro bảo mật và quyền riêng tư:** dữ liệu email nhạy cảm và token OAuth cần được xử lý chặt chẽ. *Giảm thiểu:* giới hạn log, quản lý secret qua env, phân quyền truy cập dữ liệu theo userId, và cân nhắc chính sách lưu trữ/xoá dữ liệu trong các giai đoạn tiếp theo.