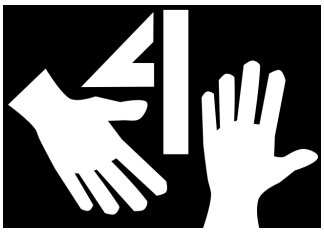


Hands4Hire - platform for connecting handymen with customers

Marta Borek Krzysztof Fijałkowski Tomasz Owienko
Michał Jakomulski Wojciech Sekuła Arkadiusz Niedzielski

Contents

1 Introduction

Connecting Customers with Trusted Service Providers — Simplified

Traditionally, finding a trustworthy handyman or home repair specialist has relied heavily on word-of-mouth recommendations from friends or neighbors. While personal referrals can be valuable, this informal system often makes it difficult for customers to discover available professionals — especially on short notice — and for skilled service providers to consistently find work or expand their client base.

Our platform was designed to address these challenges by offering a centralized, transparent, and reliable digital solution for household services. Whether it's plumbing, electrical repairs, or general handyman tasks, we simplify how Customers connect with Vendors based on availability, location, and service type.

At the same time, Vendors benefit from a dedicated space to showcase their services, manage scheduled visits, communicate with clients, and receive payments — all in one interface.

By streamlining this process, we aim to create a modern, scalable, and accessible ecosystem that benefits both sides of the marketplace.

1.1 Target Audience

This platform is designed for two primary user groups:

- **Customers** seeking reliable, local household service providers for specialized repairs and general handyman work
- **Vendors**, including self-employed professionals or small service companies, who want to expand their client base, manage bookings, and improve service visibility through digital means

Additionally, system administrators form a third side, responsible for managing operations, analytics, and support.

1.2 Competitive landscape

The platform operates within a growing ecosystem of digital solutions that connect individuals with service professionals. Notable competitors in the Polish market include:

- **Fixly.pl** – A popular platform that connects customers with professionals for a variety of services. While Fixly provides broad access it does not offer full transparency in availability and pricing, necessitating back-and-forth communication outside the platform.
- **Fucha** – A mobile-first app targeting gig workers and handymen seeking flexible jobs. Its informal structure and focus on short-term work make it less ideal for customers seeking scheduled and longer-term services.
- **Oferteo.pl** – A marketplace where service providers are matched to customers via quote requests. While it supports a wide range of services the communication is often moved off-platform.
- **OLX** and **Allegro** – Although primarily used for selling products, both platforms include service advertisements. However, they do not offer structured scheduling and availability management.

Our platform differentiates itself by providing:

- Real-time vendor availability and booking through an integrated calendar system.

- Build-in identity verification at the beginning of the visit.
- End-to-end communication and payment capabilities, keeping the full service lifecycle within the platform.

This combination of transparency, control, and trust positions our platform as a modern and comprehensive solution to the traditional word-of-mouth and classified-based service discovery methods.

1.3 Monetization Strategy

At present, the platform generates revenue through a one-time registration fee for all Users, ensuring commitment and basic verification before entering the marketplace.

Planned and evolving monetization strategies to be introduced in future iterations include:

- Advertisement-based revenue, allowing for on-platform advertising placements, targeted by region or service category.
- Featured listings and promotional boosts, giving vendors the opportunity to gain increased visibility through paid placement on the search interface or homepage.
- Future subscription tiers, which will introduce optional paid plans for vendors, unlocking features such as:
  - Advanced analytics and insights into the Vendor company's prosperity
  - Priority customer support
- Transaction-based commissions where the platform will retain a small percentage of each successful service payment. This model will also enable:
  - Enhanced platform guarantees and dispute resolution support
  - Insurance and reliability features, increasing customer trust and reducing risk
- Affiliate partnerships with home improvement brands, insurance providers, or tool suppliers, creating added value and convenience for the Users and new revenue streams.

This multi-layered strategy will enable the platform to upgrade its services while continuously adding more avenues for income.

2 Requirements

This section begins with a set of **Use Cases**, which describe typical interactions and workflows from the perspectives of end-users, highlighting functionalities and characteristics of the service. Core system behaviors identified in this process, such as user authentication and visits management, are detailed in the Functional Requirements. The Non-Functional Requirements cover system attributes such as scalability, performance and security, to be achieved through the general structural criteria of the platform and the tools selected for its development.

2.1 Use cases

2.1.1 Create Profile

Actor: Unregistered User
Preconditions: User is not yet registered or logged in
Basic steps:

1. User opens registration page
2. Authenticates Google account
3. Picks role
   - *Vendor* - service provider

    - *Customer* - service recipient
4. Provides detailed data necessary for the role
5. Pays registration fee
6. System creates profile and redirects to dashboard

Postconditions: Customer profile is stored in the system

2.1.2 Edit Profile

Actor: Registered Customer or Vendor
Preconditions: User is logged in
Basic steps:

1. User navigates to "Edit Profile" from the home page
2. Updates fields (e.g. skills, address, contact info)
3. Submits changes
4. System syncs data and confirms updates

Postconditions: Profile gets updated and reflects latest data

2.1.3 Update Calendar

Actor: Registered Vendor
Preconditions: Vendor is logged in
Basic steps:

1. Vendor opens calendar from profile page
2. Adds/deletes available slots
3. Can change status of a previously *accepted* booking to *canceled*
4. Submits changes
5. System syncs data and confirms updates

Postconditions:

- *Canceled* booking removed from the calendar, still visible in history
- Updated calendar is reflected in searches and Vendor profile
- Customer gets notification about Booking's status change

2.1.4 Search for Vendor

Actor: Registered Customer
Preconditions: Customer is logged in
Basic steps:

1. Customer opens search page
2. Selects filters (e.g. type of service, geographical range, availability, customer rating)
3. Submits search
4. System displays matching Vendors profiles

Postconditions: Filtered list is shown

2.1.5 View Vendor Profile

Actor: Registered Customer or Vendor
Preconditions:

1. User is logged in
2. Vendor Search/Filtering results available

3. **Or For Customer:** Vendor already in Customer's Booking History

Basic steps:

1. Select Vendor from Search List or Booking History
2. View Full Profile
    - Contact Info
    - Skills & Experience
    - Calendar availability
    - User rating and comments from previous jobs

Postconditions: System displays public Vendor's Profile

2.1.6 Book Vendor

Actor: Registered Customer

Preconditions:

1. Customer is logged in
2. Selected Vendor's Profile open

Basic steps:

1. Select time slot from Vendor's Calendar
2. Leave optional service note
3. Submit request

Postconditions:

- Booking request stored in the system with *pending* status
- Request added to Customer's booking history, visible in Customer's dashboard
- Notification sent to Vendor, request visible in Vendor's dashboard

2.1.7 Identity Verification during Visit

Actor: Registered Customer and Registered Vendor

Preconditions:

1. Customer is logged in
2. Vendor is logged in
3. Current time corresponds to predefined range before/during/after the scheduled Booking time (e.g., Booking time slot + 30 minutes before and after the slot)

Basic steps:

1. Customer gets notification about the upcoming visit + verification code (both email and in-app)
2. Vendor selects appropriate Booking from their Booking List
3. Selects "Verify"
4. Enters Customer's code into a form
5. Vendor sends completed form
6. Vendor gets immediate feedback on whether the code matches & option to retry if incorrect

Postconditions:

- Booking gets status update to *verified* in Customer's and Vendor's Booking Histories
- Notification about the Booking verification sent to both parties v

2.1.8 View Customer Profile

Actor: Registered Vendor or Customer

Preconditions:
1. User is logged in
2. Vendor received Booking from a Customer
3. OR User views comments left by other Customers on Vendor's Profile

Basic steps:
1. Select Customer from Booking request, or Comments List
2. View Full Profile
    - Contact Info
    - Booking History
    - Comments and ratings for previous jobs

Postconditions: System displays public Customer Profile

2.1.9 Accept or Decline Booking

Actor: Registered Vendor

Preconditions:
1. Vendor received Booking from a Customer
2. Booking status is *pending*

Basic steps:
1. Vendor received a Booking Request notification (in-app/e-mail)
2. Notification redirection to Booking Request, with eventual authentication step
3. OR Open Booking request directly from dashboard
4. Booking Request shows:
    - Customer who placed the request
    - Date, Time & Duration
    - Service note from the Customer
    - Current status *pending, canceled*
5. If not *canceled* Vendor can Accept or Decline with designated buttons
6. System confirms the status update

Postconditions:
- Booking status changes to *accepted* or *declined*
- If *accepted*, is now visible in the Vendor availability Calendar and will be reflected in the Vendor Searches
- Customer gets a notification about the Booking status change

2.1.10 View & Cancel Booking Status

Actor: Registered Customer

Preconditions:
1. Customer placed at least one Booking

Basic steps:
1. Customer goes to Booking History on home page
2. Selects an entry to open a Booking page
3. OR Notification about Booking status update redirection to Booking Request, with eventual authentication step
4. Booking entry shows:
    - Booked Vendor
    - Date, Time & Duration
    - Service note from the Customer

    - Current status *accepted, declined, pending, canceled*
5. If *accepted* or *pending*, Customer can cancel the request
6. System confirms eventual status update

Postconditions:

- Booking status changes to *canceled*
- If previously *accepted* by Vendor, it is now removed from the Vendor availability Calendar and will be reflected in the Vendor Searches
- Vendor gets a notification about the Booking status change

2.1.11 Contact Between Customer & Vendor

Actor: Registered Customer or Vendor

Preconditions:

1. Booking request placed by Customer
2. OR User Profile viewed
3. OR Previously registered in the chat history

Basic steps:

1. User selects "Message" for a new chat from Recipient's Profile
2. OR an active chat from chats list on the dashboard
3. Enters message in chat interface
4. Sends message
5. Recipient receives notification

Postconditions:

- Notification about a new message sent to the Recipient
- System stores new message and updates the chat history
- If no previous chat history with the Recipient, System adds new chat to the chat history, available from User dashboard

2.1.12 Comment & Rate Past Booking

Actor: Registered Customer

Preconditions:

1. Booking accepted by Vendor
2. Successful identity verification of Booking parties
3. Booking time slot passed

Basic steps:

1. User selects Booking from their Booking History
2. User selects "Rate and Comment"
3. Enters rating within a specified range
4. AND/OR Writes a comment
5. Selects "Publish"

Postconditions:

- Notification about a new rating sent to the Vendor
- System stores new rating and updates the affected Vendor's profile
- Booking marked with *rated* status in Client's Booking History
- Client cannot leave another comment/rating on that Booking

2.2 Functional requirements

2.2.1 User Profile Management

Registration

- Users get registered through their Google account
- Account creation process requires a one time registration fee

User Profile

- Each user owns a profile with editable personal data
- Additionally:
  - *Vendors* profiles contain:
    - Skills and specialization areas
    - User ratings and comments from previous jobs
    - Calendar with their availability
  - *Customers* profiles contain:
    - Booking history
    - Ratings for previous bookings
  - *Administrator* profile holds a privileged access and can:
    - Manage Users
    - View Vendors' system reports

Ratings and Comments

- Customers can leave feedback for the jobs from the Booking History
- Feedback in a form of a star range rating (scale up to 5 or 10) and an optional comment

2.2.2 Calendar

- Vendors manage their availability in a personal calendar, which is visible on their profile
- Customers view Vendors's calendar on their profile and can make booking requests within the free slots

2.2.3 Bookings Management

Vendor Search Functionality for searching and filtering Vendors profiles according to:

- Skills/Type of Service
- Price
- Location or geographical range
- Availability
- Customer rating

Booking placement Through calendars available at Vendor profiles, Customers can place bookings:

- Bookings can be accepted or declined by Vendors
- Bookings can be canceled by both Vendors and Customers, regardless of their status
- Status change of a Booking results in a notification to the second party

Report generation Automatic statistical report generation per Vendor profile. Includes:

- Bookings count
- Work hours
- Earnings
- Types of services

2.2.4 Communicator

- User Chat for communication between Vendors and Customers
- Code for Booking confirmation and identity authentication sent via chat

2.3 Non-functional requirements

- **Architecture** As dictated by the general project requirements:
  - Application is hosted on a public cloud (Google Cloud Provider)
  - Application consists of at least 3 types of microservices
  - The infrastructure build and setup are automated with the use of automation tool, Terraform
  - Database and state are owned by each microservice type separately, with two of them having private cache of the other's data subset
- **Performance & Responsiveness** System should support real-time updates for chat and booking confirmations.
- **Authorization & Security**
  - User identity verification is done with the use of trusted identity provider Google OAuth 2.0
  - Authentication in payment processing is done via Stripe secret keys, securely stored in GCP Secret Manager
- **Maintainability & Observability**
  - The use of automation tool, Terraform, allows for reproducible deployment
  - All microservices emit structured messages to Kafka and the Google Cloud Logging Agent

3 System Architecture

Designed as a scalable, modular, and cloud-native solution, the platform follows a microservices-based architecture hosted on Google Cloud Platform (GCP). The architecture leverages containerization, infrastructure-as-code, and managed cloud services to support scalability, modular development. In the sections that follow, we outline the key components of the infrastructure stack, the design choices, microservices modules and give description of the role each one performs and how it integrates with the broader platform.

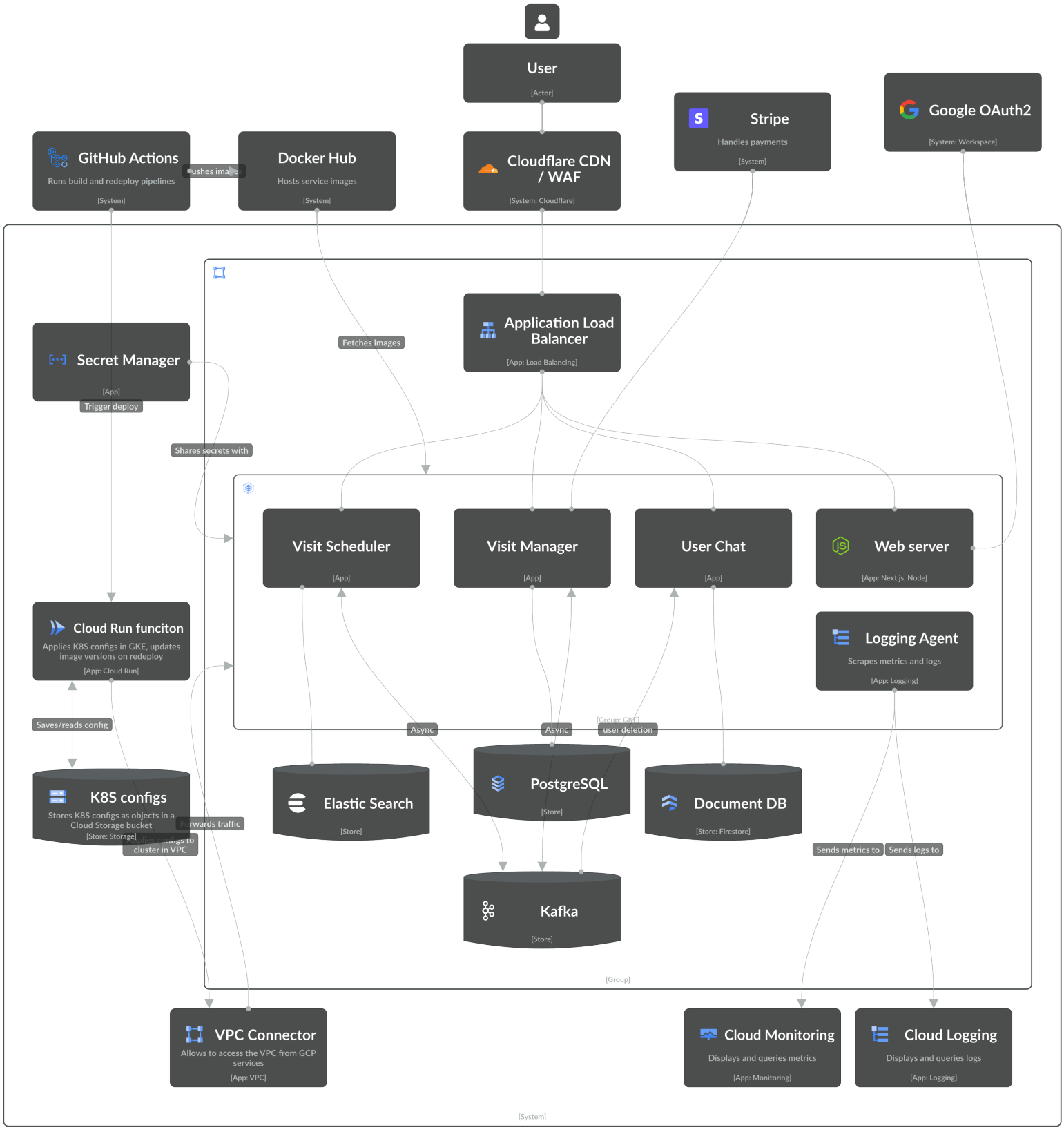


Figure 1: Architecture Diagram created with the C4 philosophy in IcePanel.

3.1 Infrastructure Stack

3.1.1 GCP

Google Cloud Platform was selected as the cloud infrastructure for this project due to a combination of strategic, financial, and technical advantages:

- **Market Leadership** - As one of the top three global cloud providers, GCP offers enterprise-grade reliability, security, and scalability.
- **Cost-effectiveness** - Especially important from the student perspective, GCP provides a generous free tier for students, including credits and access to its core services.
- **Strong containerization** - Google originated Kubernetes and continues to lead in its evolution. The project uses Google Kubernetes Engine (GKE), making GCP a natural choice due to its native support and optimized performance for container orchestration.

In addition to these motives, GCP provides an integrated suite of services that simplify core infrastructure concerns and are utilized in our platform, as showcased in Figure 1.

Cloud Run is used to run small deployment scripts and one-time jobs like updating Kubernetes configs, scaling down to zero when idle.

VPC (Virtual Private Cloud) securely isolates and connects all internal components like GKE and Cloud Run, through granular network control and firewall rules. All services—including GKE nodes and Cloud Run functions—communicate via the private VPC network.

VPC Connector lets serverless services like Cloud Run access the VPC without exposing private services to the internet.

Secrets Management securely stores sensitive data like API keys and database credentials, which are accessed at runtime using IAM-controlled permissions.

Traffic Management and Load Balancing GCP's Application Load Balancer (ALB) manages incoming HTTP/S traffic, distributing requests to services running on GKE or Cloud Run. It handles SSL termination, performs health checks, and supports features like request throttling and routing via Kubernetes Ingress. The built-in GCP firewall rules operate at the network level, restricting traffic based on IP ranges and protocols to ensure secure access between components. While Cloudflare provides application-level protection and performance optimization at the edge, its role is covered more thoroughly in 4.1.

Observability Stack

- **Cloud Logging** collects logs from all services.
- **Cloud Monitoring** exposes application and infrastructure metrics.
- Custom **Logging Agents** are deployed alongside services to aggregate and tag logs for traceability.

3.2 Visit Scheduler Module

The role of this microservice is to handle Vendor discovery and timeslot availability logic. It is built with the use of FastAPI backend, picked for its performance and modern features and

Elasticsearch Search Engine , which is applied to dynamic filtering of Vendor profiles based on user-defined criteria, such as:

- Skills/Type of Service, which include:
- Price
- Location or geographical range
- Date&Time Availability
- Customer rating

Its purpose in the ecosystem is to serve as the matchmaking logic between service providers and clients by processing search queries and exposing available booking slots in real-time. It communicates closely with the Visit Manager microservice for validating time slots and availability before booking. While Elasticsearch is the only core data component in the stack not offered as a managed service by GCP, its powerful full-text search and filtering capabilities made the operational overhead worthwhile for the use case.

3.3 Visit Manager Module

The Visit Manager acts as the platform's central component for handling user data, visit records, and system-wide coordination. It utilizes a PostgreSQL database which schema is showcased in Figure 2. It exposes REST APIs for core actions such as visit creation, modification, and deletion, as well as user registration and lifecycle management; the core routing endpoints are listed in the section below. The service also plays a significant role of integration with the Stripe payment service.

3.3.1 Routing

- **/register** - submit User's registration details
- **/register_visit** - visit registration, passed from the Visit Scheduler module, which handles Customer's input
- **/client/my_visits** -
- **/vendor/my_visits** - returns a list of visits that a Vendor has scheduled
- **/get_visit_code** - returns generated code for Vendor's identity confirmation during the visit (hash of visit_id for simplicity reasons)
- **/check_visit_code** - returns information about validity of the code submitted in a request by the Customer
- **/add_opinion** - enabled only after the scheduled visit took place, allows Customer to leave a rating (1-5 scale) and review

3.4 User Chat

This microservice facilitates communication between Clients and Vendors. It is built using FastAPI and backed by Firestore. This REST-based approach was chosen over WebSockets for simplicity. Since the chat functionality is not mission-critical and does not require persistent connections or real-time typing indicators, using Firestore, a scalable NoSQL document database that handles real-time updates efficiently was a practical and resource-light solution.

The minimal version of the service attained by the end of the project defines Firestore helpers for storing and retrieving chat messages and exposes two endpoints—*/send* and */messages*—secured with JWT-based authentication to ensure that only authorized users can interact with the chat.

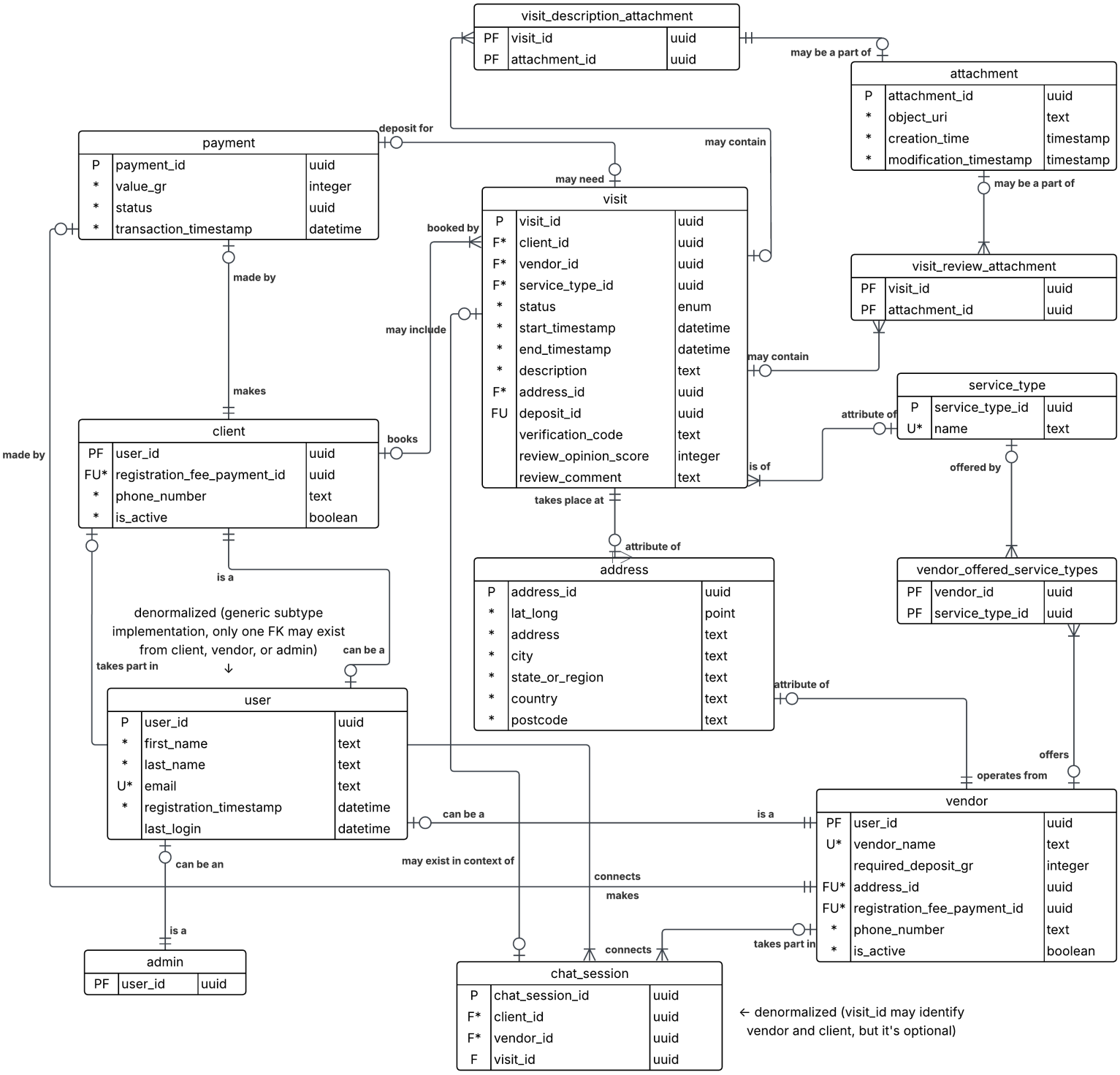
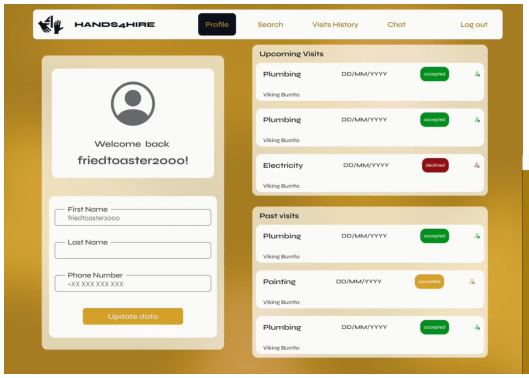


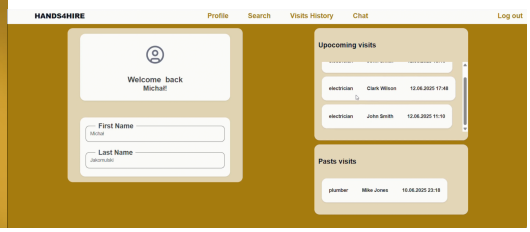
Figure 2: PostgresQL Data Base schema diagram made in LucidChart.

3.5 Web Server

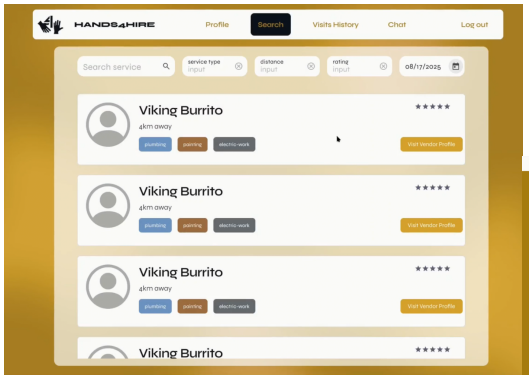
The Web Server microservice serves as the frontend of the platform, built using React and Node.js. It delivers the user interface through which Clients and Vendors interact with the system. The design process was centered around Figma, a collaborative interface design tool widely used for creating and prototyping UI/UX layouts. Core screens were first designed in Figma, enabling rapid iteration and feedback. These designs were then exported and transformed into functional React components using Anima—a Figma plugin that converts design elements into responsive React code—and Figma's built-in Dev Mode, which helps developers access CSS, spacing, and component properties directly. This workflow ensured visual consistency and streamlined the handoff from design to development. A side by side comparison of Figma screens and actual pages are displayed in Figure 3.



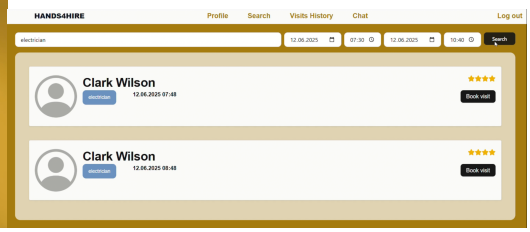
(a) Homepage screen in Figma.



(b) Homepage screen in the App.



(c) Vendor search screen in Figma.



(d) Vendor search screen in the App.

Figure 3: A side-by-side comparison of Figma screens and Web Frontend pages.

3.6 Kafka Service Communication

Our platform primarily uses synchronous REST-based communication between microservices for direct operations. However, to decouple services, improve resilience, and handle asynchronous workflows in scenarios such as booking status change or user lifecycle status, we also integrate Apache Kafka as a messaging backbone. Kafka was chosen due to its strong market position as a high-throughput, fault-tolerant event streaming platform. It integrates well with Google Cloud through Cloud Pub/Sub, making it a robust and scalable choice for inter-service communication within the ecosystem. The design of our minimal integration is

showcased in Figure 4.

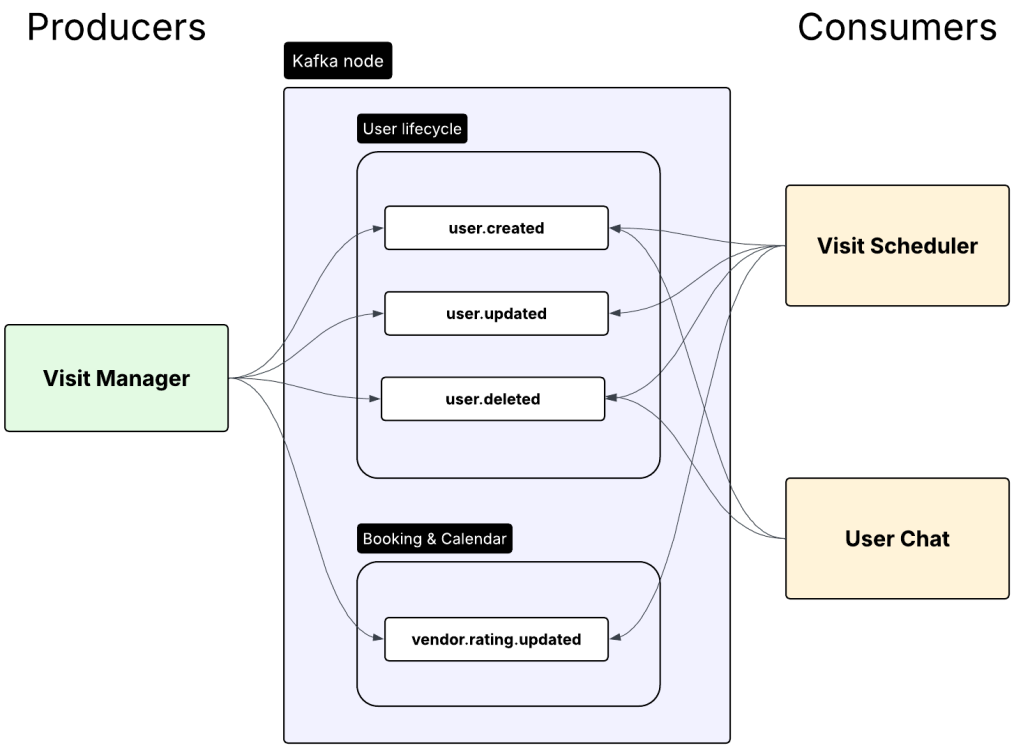


Figure 4: Kafka Topics Diagram

3.7 Stripe integration

The platform integrates Stripe for handling payment processing, specifically using the Stripe Payments simulation environment for development and testing. This integration allows us to model real-world transaction flows without incurring actual charges or needing production credentials during early development stages. We opted for the prebuilt Stripe Checkout solution, which offers a secure and PCI-compliant interface while reducing the need for custom implementation.

All payment-related interactions are handled through secure backend endpoints, which communicate with Stripe's API over HTTPS. Webhooks are also configured in the simulated environment to test various payment outcomes (e.g., success, failure, cancellation). This setup enables the platform to fully validate and observe the payment lifecycle, ensuring a smooth transition to live payments in production when needed.

3.8 Google OAuth2.0 Authentication

For user authentication, the platform integrates Google OAuth 2.0, providing a secure, widely adopted login mechanism. It was chosen for its popularity, ease of use, and seamless integration with other Google Cloud services—ensuring a unified provider for both infrastructure and identity. By relying on Google's ecosystem, we reduce complexity, avoid managing passwords directly, and benefit from built-in security features.

3.9 CI/CD Workflow & Deployment

3.9.1 Source Control (GitHub)

All code is version-controlled in GitHub. Each microservice resides in a separate repository, with independent Dockerfiles and test configurations. The project is available at `https://github.com/25l-ersms`.

3.9.2 Development & Production

To streamline development and reduce cloud costs, our team worked primarily in a local development environment before deploying to production. This environment used **Docker Compose** for service orchestration—chosen over Minikube due to its lower overhead and better support for hot-reloading during rapid development.

To ensure code quality and consistency, the same rules applied both **pre-commit** (via hooks) and in CI pipelines, checking for formatting, linting, and test coverage. GitHub Actions automate this process, running on every push or merge to the main branch.

Reusable GitHub workflows were employed for tasks like pushing Docker images to Docker Hub, ensuring standardization across repositories. **Dependabot** was used for automated package dependency analysis, keeping images secure and up to date.

Upon code push or merge to the main branch, when targeting the production environment, GitHub Actions executes the following pipeline:

1. **Build Docker Images**
2. **Test**: Linting, unit tests, and integration tests are executed.
3. **Push Docker Images** to Docker Hub.
4. **Deploy to GKE** using kubectl and version-controlled Kubernetes manifests.

Kubernetes manifests are defined and managed via Terraform, ensuring infrastructure-as-code consistency and safe rollbacks.

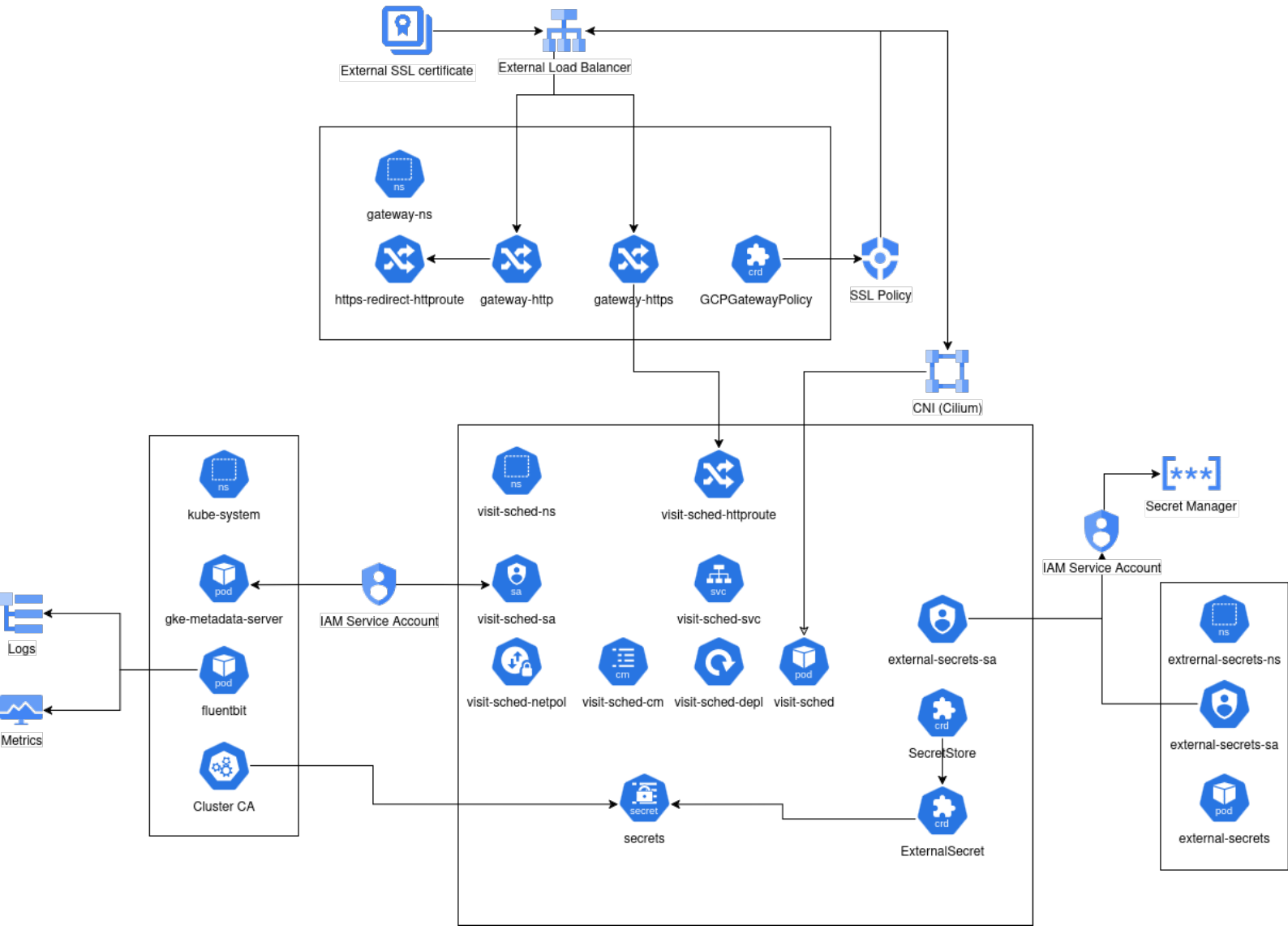


Figure 5: Kubernetes component diagram.

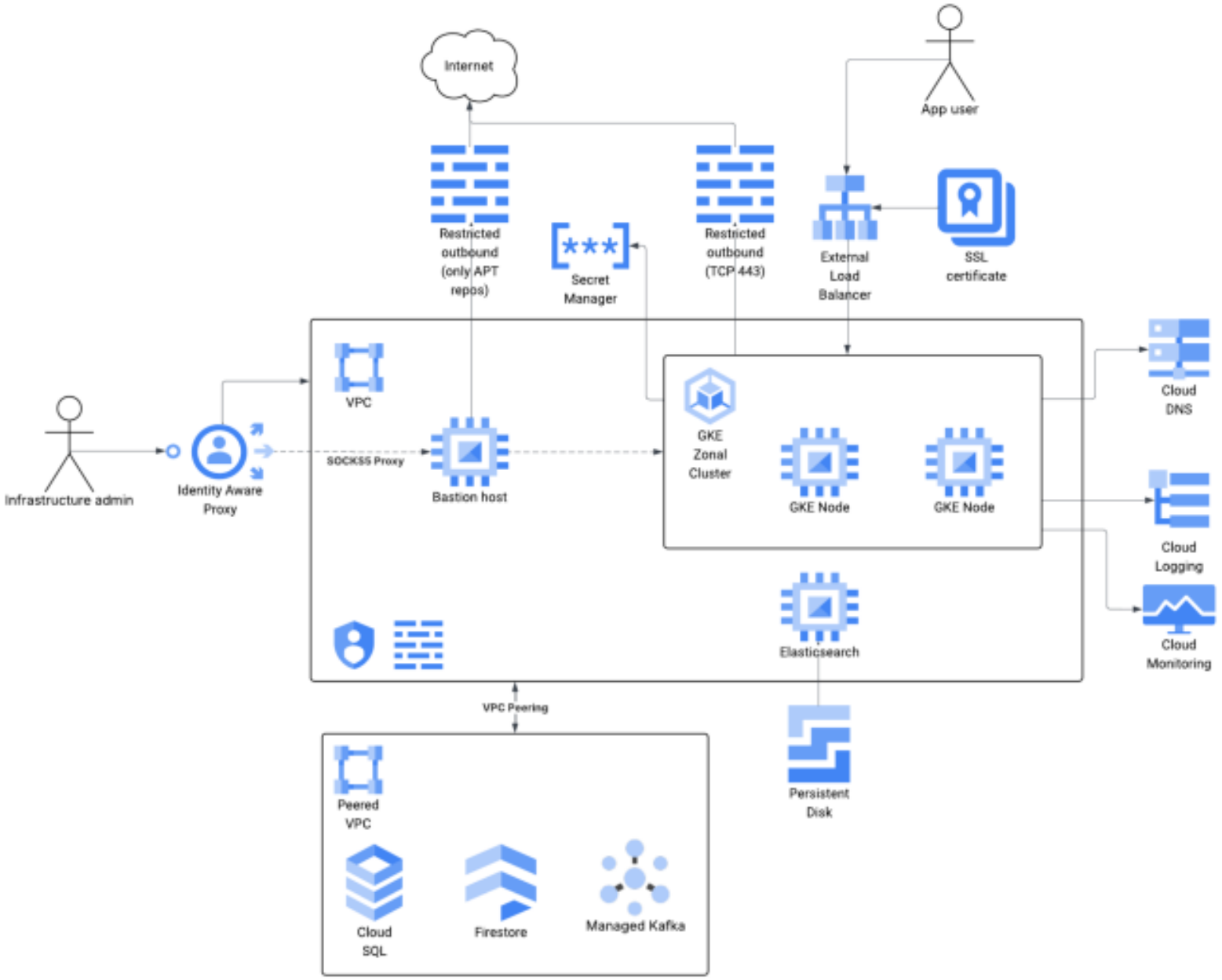


Figure 6: Cloud infrastructure diagram.

4 Security & Threat model

4.1 Implemented security measures

Security was a central consideration in the platform's architecture, with decisions guided by a pragmatic threat model that prioritizes data integrity, access control, and system resilience. Several layers of defense were implemented across infrastructure, networking, and application layers.

4.1.1 Backups and Replication

To minimize data loss in case of infrastructure failure or misconfiguration, we propose a plan for daily backups taken for critical components: the entire GKE cluster, PostgreSQL, Firestore, and Elasticsearch. Both PostgreSQL and Firestore also benefit from Point-In-Time Recovery (PITR), allowing restoration of data up to seven days back. While, for simplicity reasons, compute resources are not replicated, all backups are replicated across zones, ensuring survivability even in the case of a full Google Cloud data center failure. Secrets, managed via GCP Secret Manager, are replicated by default.

4.1.2 Secrets Management

Sensitive credentials, such as API keys and database passwords, are never embedded in code. Instead, they are securely stored in GCP Secret Manager and synced into Kubernetes using the External Secrets Operator. This operator runs with its own IAM permissions and ensures that secrets are namespace-scoped and tightly controlled. This design enforces principle of least privilege across workloads.

4.1.3 Network Security and Firewalls

A defense-in-depth strategy was adopted with two firewall layers:

- **Cloudflare**, serving as a Web Application Firewall (WAF) and Content Delivery Network (CDN), handles edge-layer protection. It blocks DDoS attacks, enforces SSL/TLS encryption, and provides DNS, rate-limiting, and bot protection.
- **GCP's built-in** firewalls and **Application Load Balancer (ALB)** control internal traffic flow and route requests securely into the private GKE cluster.

Administrative access is further restricted via Identity-Aware Proxy (IAP) with IAM-based authentication, and in some cases, a socks proxy is used for access to the GKE cluster. Importantly, no inbound traffic is allowed directly—all requests must go through IAP or Cloudflare.

4.1.4 Data Encryption

All volumes—both ephemeral and persistent—are encrypted at rest, and this includes backups. Traffic in transit is also encrypted, whether it's between services, to the load balancer, or to GCP-managed services (via Google's Application Layer Transport Security, ALTS).

The security strategies outlined above were designed to address and mitigate a wide spectrum of potential risks, implemented as a direct response to the most critical threats identified during the platform's early threat modeling phase.

4.2 STRIDE

Threat modeling process has been conducted with accordance to the **STRIDE** framework. It was chosen for its structured way to analyze threats across various components of a system. Its broad coverage of both technical and procedural risks makes it particularly suitable for cloud-native, microservice-based architectures like ours. By applying STRIDE, we were able to identify vulnerabilities at both the infrastructure and application levels, ensuring that security concerns were proactively integrated into the design and implementation phases of the project. Modeling process's outcome is showcased on the following pages as a single, summary document generated with the use of OWASP Threat Dragon modeling software.

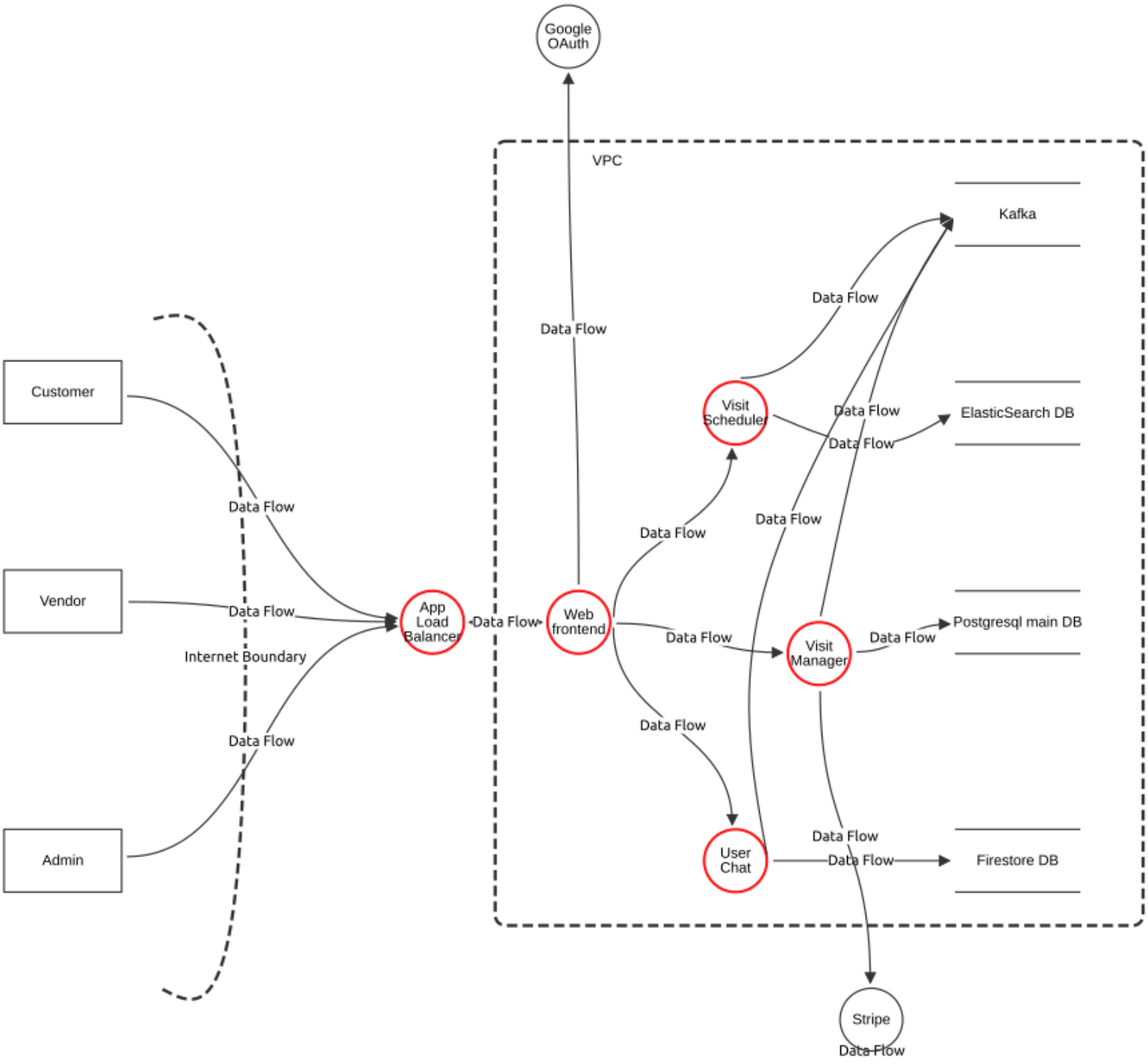


Figure 7: Threat modeling Data Flow Diagram

H4H

Owner:
Reviewer:
Contributors:
Date Generated: Wed May 28 2025

Executive Summary

High level system description

Not provided

Summary

Total Threats	26
Total Mitigated	15
Not Mitigated	11
Open / High Priority	5
Open / Medium Priority	6
Open / Low Priority	0
Open / Unknown Priority	0

DFD-H4H

Visit Scheduler (Process)

Description: FastAPI App

Number	Title	Type	Priority	Status	Score	Description	Mitigations
11	Forged user ID in requests	Spoofing	High	Open		Attackers could manipulate request bodies to act on behalf of another vendor/client and tampering with the issuing of visits that they book/accept and e.g. pay a deposit for.	
12	ElasticSearch query manipulation	Tampering	High	Open		Injection of malicious search filters to bypass constraints and return unauthorized data.	Use of proper ElasticSearch query builders, more resistant to such attempts.
13	Missing audit logs for bookings	Repudiation	Medium	Mitigated		Without proper event logging, users could deny placing a booking or changing their availability.	Proper logging and persistance and data retention.
14	Improper role enforcement	Elevation of privilege	Medium	Mitigated		Clients potentially accessing vendor-only functionality (calendar availability) and vice-versa, Vendors being able to book visits with other Vendors.	User role validation via JWT.
15	Heavy search queries	Denial of service	Medium	Open		Unbounded queries stressing ElasticSearch and slowing down its performance due to heavy processing.	Employ proper pagination within ES.

Visit Manager (Process)

Description: FastAPI app

Number	Title	Type	Priority	Status	Score	Description	Mitigations
23	Payment data manipulation	Tampering	High	Open		Modifying Stripe request data for altering payment amount, recipient etc.	
24	SQL injection	Tampering	High	Mitigated		Direct injection of malicious code.	Use of parametrized queries and ORM.
25	User Personally Identifiable Information leakage	Information disclosure	Medium	Open		Email, phone number, address exposed via poor API restrictions.	Following of GDPR guidelines in handling of data. Selective logging and masking of potentially sensitive information.
26	Stripe webhook forgery	Spoofing	High	Mitigated		Malicious requests resulting in faking of the payment events and status changes.	Validating of Stripe signatures and secrets.

User Chat (Process)

Description: FastAPI app

Number	Title	Type	Priority	Status	Score	Description	Mitigations
16	Impersonation via JWT replay	Spoofing	High	Mitigated		Interception and reusing of JWT tokens resulting in attackers sending messages on behalf of other users.	Short-lived JWT, enforce TLS encryption (HTTPS).

Number	Title	Type	Priority	Status	Score	Description	Mitigations
17	Unauthenticated message sending	Spoofing	Medium	Mitigated		Bots or anonymous users sending spam messages if endpoints lack strict authentication.	JWT authentication enforced on all endpoints.
18	Denial of message sending	Repudiation	Medium	Mitigated		User denying sending a message, difficult to prove message origin.	Proper logging and storing of message metadata, including origin verification.
21	Accidental logging of sensitive data	Information disclosure	Medium	Open		Sensitive message content may unintentionally get logged.	Log redaction, selective storing of only the necessary metadata.
22	Lack of logging for message delivery	Repudiation	Low	Mitigated		Loss of message traceability upon sending.	Implementation of proper message delivery receipts in the logging system.

App Load Balancer (Process)

Description: Nodejs App

Number	Title	Type	Priority	Status	Score	Description	Mitigations
1	JWT forwarding	Spoofing	High	Open		Attacker intercepting or replaying tokens due to improper security controls at ALB level; could result in full account takeover.	TLS encryption enforced upon the communication between the services (HTTPS). Proper token validation, short-lived JWTs.
2	Volumetric attacks	Denial of service	Medium	Mitigated		Large scale traffic floods designed to overwhelm ALB or downstream resources.	GCP Cloudflare employed.
3	Request header mangling	Tampering	Medium	Open		Attackers replacing specific request headers leading to potential bypass of security checks.	Proper header validation and sanitization.
4	Routing misconfiguration	Information disclosure	Medium	Mitigated		Misconfigured routing may expose internal endpoints to the public.	Blocking and restricting access to internal paths.
5	Lack of request attribution	Repudiation	Low	Mitigated		Without proper logging at the level of App Load Balancer it may be difficult to notice and track malicious activitiy.	Accurate monitoring setup at the outer-most layer and alerting on anomalies (Cloud Logging).
6	CORS misconfiguration	Information disclosure	Medium	Open		Without proper restriction, App Load Balancer may allow cross-origin requests, leaking data to potentially malicious sites.	Proper CORS configuration, specific headers and only trusted origins.

Firestore DB (Store)

Description:

Number	Title	Type	Priority	Status	Score	Description	Mitigations
19	Document manipulation	Tampering	High	Mitigated		If security rules are improperly configured, attackers may modify chat history records.	Strict security rules within the DB.
20	Access to unauthorized chat histories	Information disclosure	Medium	Mitigated		Missing access checks may lead to the potentially malicious users seeing conversations that they should not have access to.	Enforce Firestore access control for all documents.

Web frontend (Process)

Description: Nodejs app

Number	Title	Type	Priority	Status	Score	Description	Mitigations
7	Token leakage from localStorage	Spoofing	High	Open		Attackers getting user's locally stored credentials with the use of malicious scripts.	Use of PKCE in OAuth.

Number	Title	Type	Priority	Status	Score	Description	Mitigations
8	Sensitive data leakage from localStorage	Information disclosure	Medium	Mitigated		Data such as chat history, or payment information, if stored locally can be leaked.	Avoid storing sensitive data client-side.
9	Volumetric attacks on backend endpoints	Denial of service	Medium	Open		Risk of repeated spam on the endpoints of the downstream microservices.	Enforcing rate limits per IP address or user.
10	Unauthorized access to dev tools	Elevation of privilege	High	Mitigated		Routes that are not blocked/restricted properly may expose access to administration/development features.	Routing safeguarding, access restriction via backend authorization.