iJRASET
International Journal For Research in Applied Science and Engineering Technology

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Volume: 14 **Issue:** II **Month of publication:** February 2026

DOI: https://doi.org/10.22214/ijraset.2026.77332

www.ijraset.com

Call: 08813907089 | E-mail ID: ijraset@gmail.com

International Journal for Research in Applied Science & Engineering Technology (IJRASET)
ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538
Volume 14 Issue II Feb 2026- Available at www.ijraset.com

An Intelligent and Scalable: AI Based QuickServe Service Marketplace for Real-Time Urban Service Management

Anushka Shimpi[1], Pratik Sawant[2], Tushar Shinde[3], Omkar Solanke[4]
Department of Computer Science and Engineering, Faculty of Science and Technology, JSPM University Pune Maharashtra India

Abstract: ***On-demand service marketplaces have become an integral part of modern urban life by enabling users to access professional services in a convenient and time-efficient manner. However, existing platforms often suffer from inefficient provider selection, limited transparency during service execution, delayed responses, and insufficient real-time coordination. These limitations reduce user satisfaction and erode trust in service platforms. This paper presents AI-Based QuickServe, a comprehensive intelligent service marketplace that integrates artificial intelligence, geolocation services, and real-time communication technologies to address these challenges. The proposed system includes AI-assisted request understanding using image analysis, a multi-criteria provider matching algorithm combining distance, availability, reliability, and timeliness, and real-time tracking using WebSockets within a secure, scalable architecture. Extensive functional evaluation with realistic workloads demonstrates improved matching quality, reduced response latency, and enhanced transparency while maintaining stable performance under concurrent usage. The system provides a robust foundation for next-generation urban service ecosystems.***

Index Terms: Service marketplace, Intelligent matching, Real-time tracking, Location-based services, WebSockets, FastAPI, React, JWT, KYC.

I. INTRODUCTION

Digital platforms have fundamentally changed how users discover and consume home services such as plumbing, electrical repair, AC maintenance, cleaning, and appliance servicing. On-demand marketplaces promise convenience and speed by matching customers with nearby professionals through web or mobile interfaces. Despite their popularity, real-world deployments reveal recurring pain points: delayed responses, last-minute cancellations, poor visibility into provider progress, and limited trust due to unverified professionals.

Many existing platforms rely on simplistic, distance-only matching, or manual assignments. They treat each request as a static entry and do not leverage the full context of the service, such as urgency, complexity, or historical behavior of providers. Furthermore, once a provider is assigned, customers often lose visibility in job progress, leading to uncertainty and dissatisfaction. From the provider's perspective, poor allocation and lack of coordination limit their ability to plan routes, manage workload, and maximize earnings. Recent advances in artificial intelligence (AI), cloud computing, and real-time communication enable more intelligent, adaptive, and transparent service systems. AI models can interpret images and text to automatically infer service categories and likely issues. Geolocation technologies can support proximity-aware matching, and WebSockets can deliver live status updates and location tracking to all stakeholders.

In this context, we propose AI-Based QuickServe, an intelligent service marketplace platform that automates the service lifecycle from request creation to completion. QuickServe connects customers and verified service providers in real time, using an AI-assisted request flow, a multi-criteria matching algorithm, and a role-aware, secure architecture with three distinct personas: customer, provider, and admin.

II. RELATED WORK AND BACKGROUND

Service marketplaces have been widely studied in the contexts of ride-hailing, logistics, and task allocation. Traditional systems primarily focus on geographic proximity and user ratings. While this approach provides basic functionality, it neglects dynamic factors such as provider availability, current workload, and route feasibility. Many platforms implement static filters and simple sorting but lack the richer intelligence required in complex urban environments.

Recommendation systems and optimization algorithms have been applied to job matching in recruitment and gig economy platforms. Machine learning techniques can learn preferences and improve matching accuracy using historical data. However, in many existing service platforms, requests are treated as simple text forms, and there is limited automation in understanding the actual problem type or urgency. This leads to mismatches and suboptimal assignments.

In terms of transparency, prior systems often stop assigning a provider and displaying an approximate arrival time. Continuous real-time tracking and bidirectional notifications are less common, especially in small to medium-scale marketplaces. Without strong verification mechanisms and live coordination, trust issues, cancellations, and disputes become frequent.

On the technical side, microservice architectures built with RESTful APIs, token-based authentication, and WebSockets are now standard for scalable real-time applications. Frontends commonly use single-page applications (SPAs) with frameworks like React, while backends leverage modern frameworks such as FastAPI and Node.js. Role-based access control (RBAC) is used to implement secure multi-role systems.

QuickServe builds on these foundations and addresses specific gaps by:

1) Combining AI-assisted request understanding, geolocation, and multi-criteria matching in a unified pipeline.
2) Enforce strong verification through KYC (Know Your Customer/Provider) workflows.
3) Providing continuous real-time tracking and event-driven notifications throughout the request lifecycle.
4) Exposing a complete admin ecosystem for oversight, analytics, and configurable behavior.

III. PROBLEM DEFINITION AND MOTIVATION

Despite the widespread presence of on-demand service platforms, several core challenges remain unresolved:

1) Limited provider matching: Many systems match solely on geographic proximity or simple ratings, ignoring provider availability, current workload, response behavior, and verification status.
2) Weak request understanding: Manual text descriptions often lack structured information about the problem, causing mismatches or unnecessary back-and-forth communication.
3) Lack of transparency: Customers frequently cannot see where the provider is, when they arrive, or what the current job status is, reducing trust and satisfaction.
4) Operational inefficiency: Without intelligent automation, manual coordination becomes a bottleneck as request volume grows, especially in dense urban settings.
5) Inadequate admin control: Platform owners require tools to monitor KYC, manage service categories, tune matching parameters, and analyze performance trends.

IV. PROPOSED SYSTEM

A. *System Overview*

AI-Based QuickServe is a full-stack service marketplace that automates the complete lifecycle of urban home services. The platform provides three separate experiences:

1) Customer: create requests, optionally upload images for AI analysis, view ranked nearby providers, confirm bookings, track providers in real time, and rate completed services.
2) Provider: register and complete KYC, set up service categories and pricing, go online/offline, receive and accept/reject jobs, navigate using live maps, and review earnings.
3) Admin: review and approve or reject KYC submissions, manage providers and service types, monitor all requests, adjust platform settings (e.g., matching radius), and view analytics.

The system emphasizes end-to-end automation: from AI-assisted request capture to automatic provider assignment, live tracking, status transitions, notifications, and rating flows.

B. *Architecture*

The system adopts a layered architecture:

1) Frontend: React 19 + Vite, Tailwind CSS, React Router v6.
2) Backend: FastAPI (Python), SQL Alchemy ORM, PostgreSQL.
3) Cache: Redis for session tokens and frequently accessed data (e.g., active provider locations).
4) Real-time: WebSockets for notifications and live tracking.

5) External services:
Groq Vision API for image-based request understanding.
Cloudinary for image and document storage.
OpenCage for geocoding and reverse geocoding.
MapmyIndia / Leaflet for map visualization and route hints.
Data flows from the frontend to the backend via REST APIs and WebSockets. The backend interacts with the database and external APIs and pushes updates back to subscribed clients.

C. System Workflow

The typical workflow is:

1) Service request creation:
The customer selects or confirms a service type.
Optionally uploads an image of the issue (e.g., broken pipe, damaged plug).
Text and image are analyzed to auto-suggest a service category and problem description, which the user can edit.

2) Intelligent provider matching:
The backend identifies eligible providers based on service category, location, KYC status, and online availability.
It computes a multi-criteria score for each provider using the matching algorithm described in Section 5.
The top provider is assigned automatically or presented for customer confirmation.

3) Tracking and coordination:
After assignment, WebSockets are used to broadcast status updates and location changes.
The customer views real-time provider movement on a map, including estimated time of arrival.
The provider sees job details, customer addresses, and navigation tips.

4) Completion and feedback:
The provider marks the job as completed, and the system updates the request status.
The customer confirms completion and can rate and review the provider.
The rating updates the provider's profile and influences future matching.

5) Admin oversight:
Admins can monitor KYC queues, active and historical requests, suspicious patterns, and platform metrics from a dedicated dashboard.
Matching radius, service categories, and other parameters are configurable via settings.

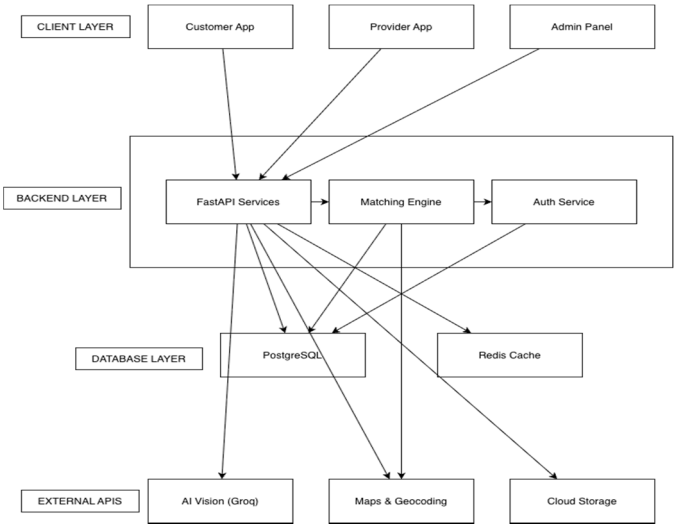


Figure_1_QuickServe_Architecture.jpg

V. INTELLIGENT MATCHING ALGORITHM

A. Design Goals

The provider matching algorithm is central to QuickServe. It is designed to:
Select relevant providers who can perform the requested service,
Prefer providers who are closer to the customer to minimize travel time,
Incorporate provider quality, availability, and timeliness,
Remain fast and scalable for many concurrent requests.

B. Request and Provider Representation

An incoming service request r is represented as:
$r = (u, c, lu, \tau, \kappa)$
where:
u is the requesting customer,
c is the effective service category (detected from user input and AI analysis),
lu is the customer location (latitude, longitude),
τ is the creation timestamp,
κ includes attributes like urgency or budget.
Each provider p is represented as:
$p = (id, Cp, lp, vp, ap, Rp, Tp)$

where:
Cp is the set of service categories the provider supports,
lp is the provider's current location,
vp is a verification flag (KYC approved or not),
ap encodes current availability (online, workload),
Rp is a reliability score based on ratings, completed jobs, and cancellations,
Tp is a timeliness score capturing response and arrival behavior.

C. Algorithm Steps

The Intelligent Provider Matching Algorithm operates in three main phases: filtering, distance computation, and scoring.
Step 1: Category and constraint filtering
Derive the effective category c from the request using AI-based analysis and user input.
Build an initial candidate set:
$Pc = \{p \text{ in } P \mid c \text{ in } Cp \text{ and } vp = true \text{ and } ap = available\}$

Step 2: Distance-based filtering
For each p in Pc, compute the geographic distance d(p) between lu and lp using the Haversine formula or a geospatial query.
Retain only providers within the maximum configured radius r_max:
$Pd = \{p \text{ in } Pc \mid d(p) \leq r\_max\}$

Step 3: Feature computation
For each p in Pd, derive normalized features:
Distance feature Dp (higher is better for closer providers):
$Dp = 1 - \min(1, d(p) / r\_max)$
Reliability feature Rp:
Derived from average rating, completion rate, dispute rate, and can be computed as a normalized weighted combination.
Availability feature Ap:
Encodes provider load and willingness, including active job count and acceptance ratio.
Timeliness feature Tp:
Based on average response time to new jobs and on-time arrival rate.

Step 4: Weighted scoring
Compute an overall score Sp for each candidate:
Sp = wd * Dp + wr * Rp + wa * Ap + wt * Tp
where wd, wr, wa, wt are non-negative weights summing to These weights can be tuned per deployment to emphasize different priorities.

Step 5: Ranking and selection
Sort Pd in descending order of Sp.
Select the provider with the highest score as the primary assignment.
Optionally, retain the top-k providers as backup candidates in case of cancellation or non-response.

D. Distance Computation

The geographic distance between the customer and provider is computed using the Haversine formula.
Let lu = (φu, λu) and lp = (φp, λp) in radians;
the distance d(p) is:
$d(p) = 2R * \arcsin(\sqrt{\sin^2((\varphi p - \varphi u)/2) + \cos(\varphi u) * \cos(\varphi p) * \sin^2((\lambda p - \lambda u)/2)})$
where R is the Earth's mean radius.
This distance is then normalized for use in Dp.

E. Complexity and Scalability

Let N be the total number of providers and M as the number of candidates after initial filtering.
Filtering: O(N) in the worst case.
Distance computation: O(M).
Feature computation: O(M).
Sorting by score: O (M log M).
Overall time complexity per request is O (N + M log M), with space complexity O(M). Because M is typically much smaller than N due to early filtering, this design scales well for many requests and providers. Further, geospatial indexes and caching can reduce effective complexity.

VI. IMPLEMENTATION AND SECURITY

A. Backend Implementation

The backend is implemented using FastAPI and SQL Alchemy. Key modules include:
Routers for:
Authentication and authorization,
Customer operations (requests, ratings, payments),
Provider operations (jobs, KYC, earnings, locations),
Admin operations (KYC review, analytics, settings),
Health checks.

Services for:
Matching, including distance calculation and scoring,
Request lifecycle management and state transitions,
Location tracking with provider location history,
KYC workflow (document upload, review, approval/rejection),
Admin report and analytics generation.
FastAPI's dependency injection is used to manage database sessions and user authentication.

B. Frontend Implementation

The frontend is a React SPA using:
React 19 with Vite for fast development and bundling,

Tailwind CSS for styling,
React Router v6 for route definitions and protected routes by role,
Axios for HTTP requests,
React Context for global state management (auth, notifications),
Leaflet and MapmyIndia for map rendering and route overlays.
Separate dashboards are defined for customers, providers, and admins, each with tailored navigation and components.

C. Real-Time Communication

WebSockets are used for:
Request-level channels to broadcast status changes and provider location updates;
Provider location channels to stream GPS updates at regular intervals to subscribed clients;
User notification channels delivering alerts about assignments, arrivals, completions, payments, and ratings.
This enables near real-time feedback and improved coordination between customers and providers.

D. Security and Privacy

Security is a first-class design goal:
Authentication:

- JWT tokens (HS256) are issued on login.
- Access tokens have short lifetimes; refresh tokens extend sessions.
- Passwords are stored as salted hashes using secure algorithms (e.g., bcrypt).

Authorization:

- Role-based access control (RBAC) restricts endpoints to roles (customer, provider, admin).
- Backend dependencies enforce role checks on each request.

Data protection:

- All communication is expected over HTTPS.
- Sensitive data (secrets, API keys) resides only in environment variables.
- Logs exclude confidential information such as passwords or full tokens.

KYC and verification:

- Providers must upload identity and address documents stored in Cloudinary.
- Admins verify documents before providers become eligible for jobs.

VII. EVALUATION AND RESULTS

A. Evaluation Setup

The system was evaluated under realistic conditions with:
Multiple customers creating service requests across different categories (AC repair, electrical, plumbing, etc.),
Providers located at various simulated coordinates, toggling online/offline and accepting or rejecting jobs,
Admin users reviewing KYC requests and adjusting matching radius.
Functional tests were performed via API calls (Swagger UI and automated test scripts) and through the full frontend.

B. Matching Quality and Response Time

Experiments demonstrated that:
The multi-criteria matching algorithm successfully prioritized providers who were both close and reliable, avoiding low-rated or distant providers when better options existed.
Changing the weights allowed the platform to emphasize speed (distance and timeliness) or quality (reliability and verification), demonstrating flexibility.
Average time from request creation to provider assignment was low, due to efficient filtering and scoring.

C. Transparency and User Experience

Real-time tracking significantly improved transparency:
Customers could see providers move toward them, along with an estimated time of arrival, which reduced uncertainty.

ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538
Volume 14 Issue II Feb 2026- Available at www.ijraset.com

Providers benefited from clear job states and navigation contexts.
Admin users had visibility over ongoing and historical requests, KYC queues, and platform activity.

D. Performance and Scalability

Under concurrent simulated workloads:
The system maintained stable response times for matching and request handling.
Caching and efficient database indexes improved performance for frequent queries (e.g., finding active providers in a radius).
WebSocket connections scaled to multiple parallel sessions without significant degradation at tested loads.

VIII. APPLICATIONS, LIMITATIONS, AND FUTURE SCOPE

A. Applications

AI-Based QuickServe can be applied to:
Home services (current focus),
Emergency maintenance (e.g., urgent plumbing or electrical faults),
Professional freelancing platforms where location matters,
Smart city service coordination, integrating municipal and private service providers.

B. Limitations

Current limitations include:
Dependence on external APIs (Groq, Cloudinary, OpenCage, mapping APIs), which introduces latency and external failure modes.
Location accuracy dependent on user device GPS and network conditions.
Manual weight tuning for the matching algorithm; no automated learning from historical outcomes yet.

C. Future Work

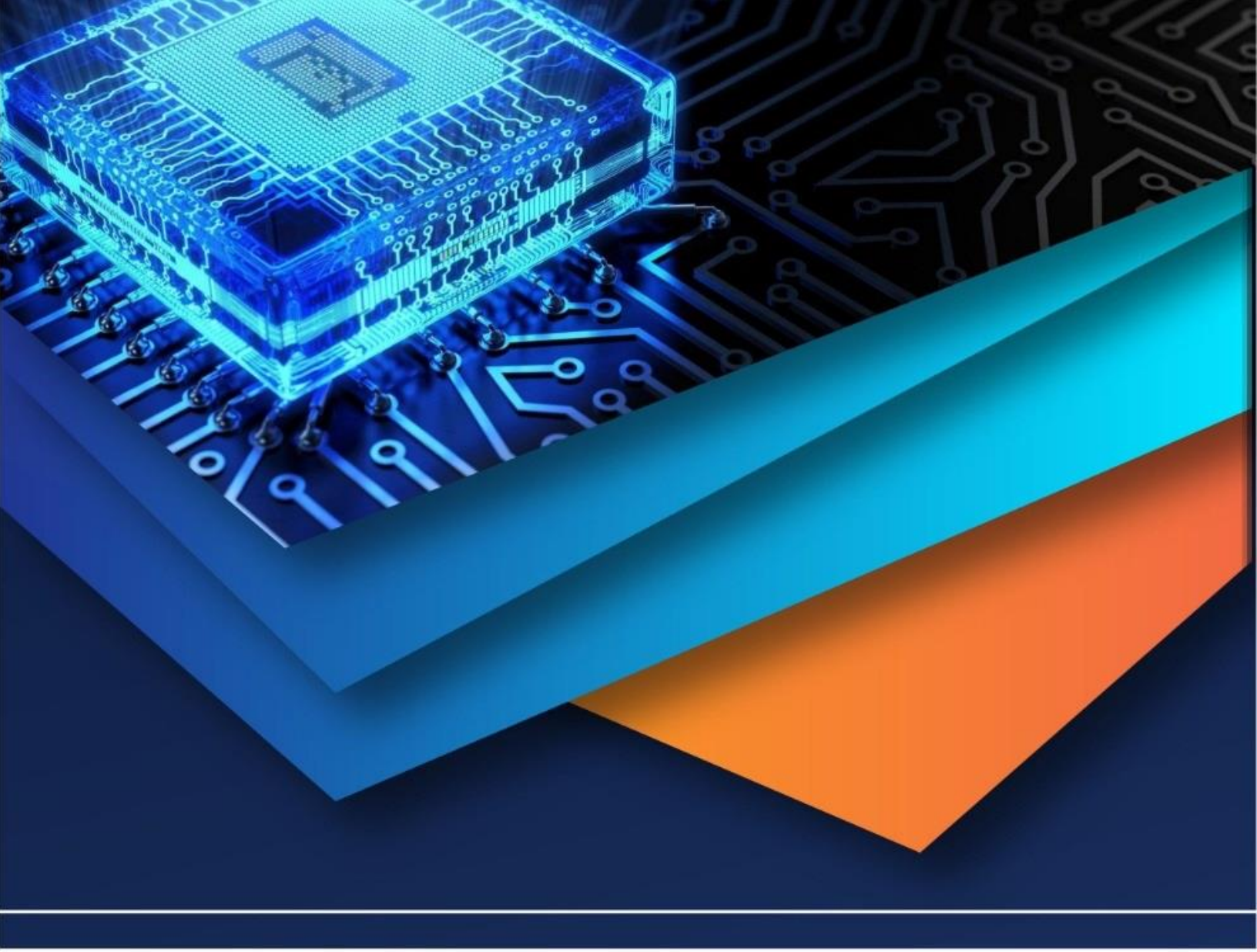
Future improvements may include:
AI-driven personalization of provider suggestions based on user history and preferences.
Predictive analytics for demand forecasting and dynamic pricing.
Voice-based or conversational request interfaces.
Multilingual support and localization for broader deployment.
Automatic learning of matching weights and features from historical success metrics (e.g., completion rate, satisfaction scores).

IX. CONCLUSION

This paper presented AI-Based QuickServe, a comprehensive intelligent service marketplace for real-time urban service management. By combining AI-assisted request understanding, a multi-criteria matching algorithm, geolocation, and real-time WebSocket communication within a secure, scalable architecture, the system addresses key limitations of existing platforms, including inefficient provider selection, limited transparency, and poor coordination. Functional evaluation shows that QuickServe improves matching quality, reduces response latency, and enhances user trust. The modular architecture and configurable matching logic make it suitable as a foundation for future, city-scale intelligent service ecosystems.

X. ACKNOWLEDGMENT

The authors would like to thank the faculty and technical staff of JSPM University for their guidance and support throughout the development of this project, and acknowledge the open-source communities behind FastAPI, React, PostgreSQL, Redis, and associated tools.



10.22214/IJRASET

45.98

IMPACT FACTOR: 7.129

IMPACT FACTOR: 7.429

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY