WWW.IJRPR.COM

International Journal of Research Publication and Reviews

Journal homepage: www.ijrpr.com ISSN 2582-7421

Design and Development of a Centralised Restaurant Operations Management System for Multi-Platform Order Integration

U.Swathi [1], N.Yashema [2], S. Balaji [3]

[1234] Department of Computer Science and Engineering Kingston Engineering College, Vellore, Tamil Nadu, India
[2]AP/CSE, [3]HOD/CSE
swathiumasankar30@gmail.com , yash.engineering@kingston.ac.in , balajicse.engineering@kingston.ac.in

A B S T R A C T :

Restaurants that receive orders through multiple delivery platforms and in-store channels may need to monitor several systems separately. This manuscript presents a unified dashboard concept for viewing order notifications and coordinating order review, kitchen preparation, and operational summaries. The browser prototype provides distinct Orders, Kitchen, and Reports views, a demonstration notification inbox for sample platform-labelled orders, manual entry of walk-in orders, order acceptance and preparation-status controls, profile editing, and local report summaries. It is implemented with HTML, CSS, and client-side JavaScript; browser local storage preserves demonstration records. The notification inbox does not receive live Swiggy or Zomato events. Authorised platform integration, a production database, secure authentication, and field evaluation remain future work. The contribution is a documented workflow and functional interface prototype, not evidence of measured operational improvement. Five diagrams distinguish the current prototype from a proposed future integration architecture and data model.

Keywords: Restaurant Management System, Cloud Kitchen, Multi-Platform Order Integration, Kitchen Management, Order Tracking, Payment Management.

Introduction

Restaurants receive orders through delivery platforms and in-store channels. When staff monitor each channel separately, notifications, acceptance decisions, preparation updates, and completion information can become fragmented. A consolidated operational view can help staff identify an order's source and follow its progress across order handling and kitchen preparation. Research on delivery-platform strategies and cloud-kitchen operations provides context for this environment, but does not evaluate the dashboard workflow presented here (Mayya & Li, 2025; Rout et al., 2024).

This project specifies a restaurant-facing dashboard with separate Orders, Kitchen, and Reports views. The browser prototype demonstrates order handling through sample platform-labelled records and manually entered walk-in orders. Its notification inbox illustrates how an alert could appear, but it does not receive live Swiggy or Zomato events. The manuscript describes the implemented interface and its limitations; it does not claim measured operational improvement or report a formal user study.

This manuscript contributes: (1) a requirements and workflow model derived from the supplied project concept; (2) a browser-based artifact with distinct Orders, Kitchen, and Reports views; (3) architecture, workflow, use-case, and data-model diagrams; and (4) a clear separation between implemented features and proposed future integration. The separation is important because the current version contains no server-side connector or empirical evidence of operational benefit.

Problem Statement

The problem addressed is the difficulty of monitoring and coordinating orders that arrive through multiple platforms and in-store channels. When each platform is checked separately, staff may lack a single view of new order notifications, acceptance, kitchen preparation, and completion. The proposed system consolidates these activities in one dashboard. Delivery-platform notifications would enter through authorised integrations, while staff would enter walk-in orders manually.

The supplied project document describes a broader cloud-based system, including automatic order aggregation, payment tracking, analytics, and user management. The present prototype has a narrower scope: it demonstrates the browser interface and local order flow. The objectives for the artifact are therefore:

- specify a common order representation that retains channel, customer, item, timing, status, and amount fields;
- provide manual creation for walk-in orders only;
- separate order review from kitchen preparation in the navigation and information hierarchy;

- compute source and status summaries from local demonstration records; and
- document the backend, security, platform access, and evaluation work required before deployment.

Literature Review

3.1 Wang (2024) authored a thesis titled "A Web-Based Restaurant Order Management System." The topic is relevant to digitising restaurant order workflows. The source is a thesis rather than a journal article; this review treats it as contextual prior work and does not attribute an unverified technology stack or measured performance outcome to it.

3.2 Rout et al. (2024) used a game-theoretic model to examine how cloud-kitchen co-location, restaurant location decisions, delivery capacity, and delivery times affect restaurants, delivery platforms, and customers. Their analysis identifies conditions under which co-location can benefit these stake-holders. The study examines an operating model rather than software-based order aggregation, so it provides context but does not evaluate the present prototype.

3.3 Švancár et al. (2024) described a Cloud Kitchen decision-support platform that combines the Unified Planning Framework (UPF) with the Vehicle Routing Problem with Time Windows (VRPTW) to allocate orders to delivery vehicles and plan routes. Using historical data, the study reports fewer deliveries arriving more than 10 minutes late. Its focus is delivery routing; the present prototype focuses on order-source visibility and kitchen preparation status.

3.4 Galabi et al. (2023) described an IoT-based restaurant system using QR-code ordering and a mobile robot for table delivery. The system addresses in-restaurant order and service automation. The present prototype instead focuses on a staff dashboard for platform-labelled orders and walk-in orders; it does not demonstrate live Swiggy or Zomato aggregation.

3.5 Kalyan and Stoney (2023) described a cloud-integrated customer food-ordering application built with Django and REST APIs. The article presents scalability and communication as design aims; the abstract does not report a comparative evaluation. The present prototype focuses on restaurant-side order notifications and kitchen coordination rather than customer-facing ordering.

4. Research Gap and Contribution

The reviewed studies address restaurant-service automation, cloud-kitchen operating decisions, or delivery-platform economics. This project focuses on a different design question: how to present delivery-platform and in-store order flows in one restaurant-facing prototype while providing distinct screens for order review and kitchen preparation. The claimed gap is bounded to this artifact and its intended users; it is not a claim that commercial point-of-sale or order-aggregation products do not exist.

The contribution is a transparent specification of this workflow and a working front-end artifact for discussion and future evaluation. The prototype provides a basis for checking whether the required source labels, kitchen ticket fields, and view separation are understandable before integrating platform services. It does not establish novelty over commercial products or demonstrate performance improvement.

5. Requirements and Proposed System

Table 1. Functional requirements and implementation status

5.1 Functional requirements

ID
R1
R2
R3
R4
R5
R6
R7
R8

5.2 Quality requirements and constraints

In addition to the functional requirements, the intended system needs a clear source label, readable status distinctions, responsive access on restaurant screens, and predictable behavior during busy service periods. These are design requirements, not measured quality results. The front-end prototype provides responsive CSS and visible source/status labels. It does not yet address multi-user concurrency, network failure, security, auditability, or durable storage.

5.3 User roles and use cases

The original project document includes owner/manager and kitchen-staff actors. The prototype provides separate information views but currently uses a single demonstration profile; role-based access is not enforced. The intended role model is summarized in Figure 1.

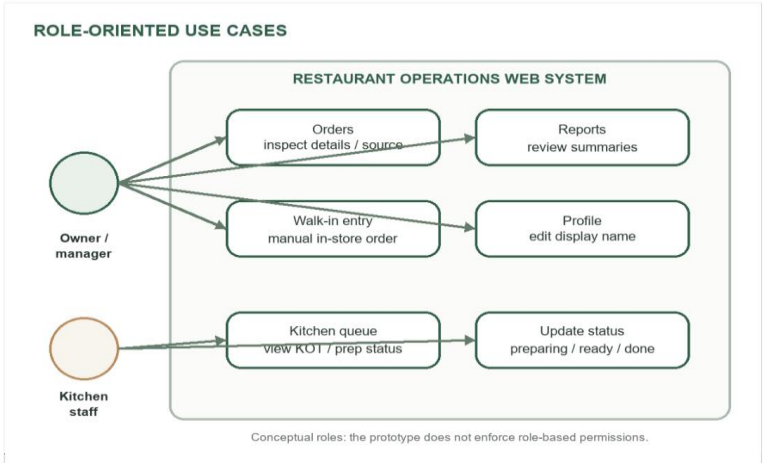


Fig. 1. Conceptual use cases for restaurant management and kitchen staff. Role permissions are not enforced in the current prototype.

5.4 Current prototype architecture

The current artifact runs in a web browser. Sample platform-labelled records and manually entered walk-in records are handled by client-side JavaScript. The main dashboard prominently displays a unified demonstration inbox of pending platform-labelled orders. A simulation control can add another illustrative alert for a walkthrough; it does not connect to an external platform. The Orders, Kitchen, and Reports views read from browser local storage. Figure 2 summarizes this implemented browser architecture. It supports a local walkthrough but has no live platform connector, shared server, durable database, or access control.

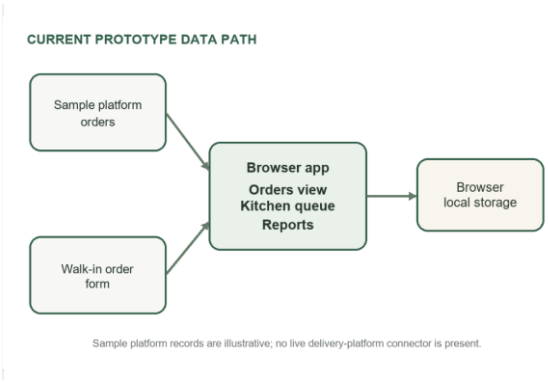


Fig. 2. Current prototype data path. Platform-labelled records are samples; only walk-in entry is manually created in the interface.

5.5 Proposed integration architecture

The target architecture in Figure 3 is a design proposal, not the implementation in the browser prototype. A secure integration service would receive platform events through authorized interfaces, validate and normalize the payloads, deduplicate repeated events, and write normalized orders to a server-side database. Walk-in records would enter through the staff interface. Role-aware web views would consume the shared data store. Credentials and platform secrets would remain on the server and would not be embedded in the browser.

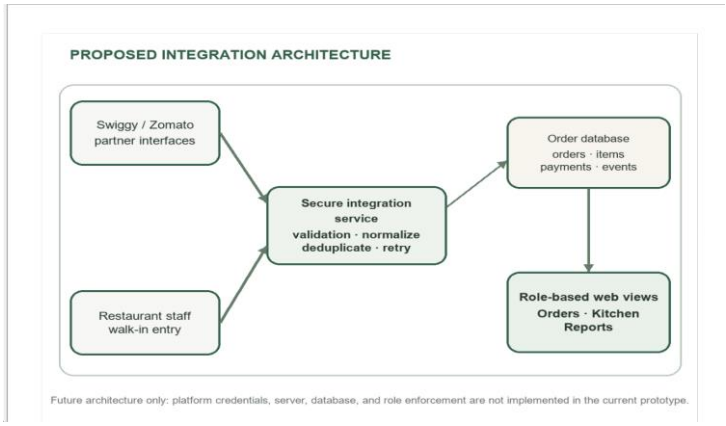


Fig. 3. Proposed future architecture for authorized platform integration and shared order storage. Components inside the boundary are not yet implemented.

6. Methodology

The development process follows a design-science sequence informed by Hevner et al. (2004) and Peffers et al. (2007): (1) identify the operational problem from the project concept; (2) define artifact requirements; (3) design the source-to-order-to-kitchen workflow; (4) implement a browser prototype; and (5) demonstrate the interface through a functional walkthrough. Requirement discovery did not include restaurant interviews, observation, or a survey. The walkthrough was not a formal usability study.

The process can be described in four practical work packages. Problem framing converted the original proposal into an operator scenario: a staff member must distinguish a delivery-platform order from a walk-in entry and then hand preparation work to the kitchen. Artifact design assigned customer-facing details to the Orders view and preparation fields to the Kitchen view. Implementation created a responsive front end and sample records. Demonstration consisted of navigating these views, adding a walk-in record, advancing its kitchen status, and inspecting the resulting report summaries. This sequence documents the artifact; it does not establish that the requirements are complete or that the design is preferred by users.

The implementation uses HTML for page structure, CSS for responsive styling, and JavaScript for view switching, form handling, status updates, filtering, report aggregation, and profile display. Local storage retains the demo order collection and profile fields in the browser. The sample set includes delivery-platform-labelled records and walk-in records. No third-party API, server framework, relational database, payment gateway, or credential service is configured.

6.1 Client-side structure and state

The browser artifact is organized as a sign-in page and an operations dashboard. The sign-in interface is defined in login.html and login.css, with form handling in login.js. The dashboard structure is in index.html, its visual rules are in styles.css, and its order and view behavior is in app.js. This separation keeps page markup, presentation, and interaction logic in distinct files while leaving the application as a static client-side prototype. The application does not use a server-rendered framework or a third-party integration library.

The order collection is represented as an array of JavaScript objects. The six seed records illustrate three source labels—Swiggy, Zomato, and Walk-in—and include fields for an internal order identifier, customer, contact number, item summary, source, elapsed-minute value, status, total, payment method and state, address or table, special instructions, and order type. These fields make the views comparable across sources, but they are demonstration records and do not originate from platform feeds or a restaurant database.

On startup, the dashboard attempts to read the saved order array from browser local storage. If no usable saved array exists, or parsing fails, it loads a copy of the seed records. Existing records are normalized with default values for missing fields, and the legacy source label Direct is displayed as Walk-in. When staff add a walk-in record or advance a kitchen status, the updated array is saved to local storage and all dashboard summaries are recalculated. This produces persistence within one browser profile; it does not provide synchronization between devices or users.

6.2 Interaction flow and state transitions

The sign-in form uses the browser's built-in form validation to require a name, email, and password field. The script saves the display name and email to local storage, sets a demo session flag in session storage, and redirects to the dashboard. The password is not verified against an account service or stored by the application. The dashboard does not enforce the session flag as an access-control check. The sign-in flow is therefore a visual demonstration and must not be treated as authentication.

A walk-in order is created through a form that requires a customer name, item description, and positive whole-number total; phone and special instructions are optional. The form assigns the next internal order identifier, labels the source Walk-in and type Dine-in, and marks the order Accepted because the staff member has entered it into the system. The form does not create a delivery-platform order. Platform-labelled seed records appear in the demonstration notification inbox and remain examples until an authorised integration is developed.

A platform-labelled sample order can be accepted from the demonstration notification inbox. The kitchen status action then advances an accepted record through In progress, Ready, and Completed; a walk-in record is accepted when staff create it. Each transition updates the same order object, saves the new state locally, and refreshes the order list, kitchen queue, metrics, and reports. The prototype does not maintain a separate event history, record who changed a status, or persist a server timestamp for each transition. Consequently, it cannot support audit trails or reliable preparation-time analysis in its current form.

6.3 View behavior and displayed metrics

The Orders view supports text search over order identifiers, customer names, item summaries, and source labels, as well as status filters. Selecting an order opens a detail dialog containing customer contact, source, order type, items, displayed elapsed time, status, address or table, payment information, instructions, and total. The list is limited to six visible rows in the current renderer. Dynamic values are escaped before being placed in HTML, and form inputs are checked for required values; these measures do not replace server-side validation or access control.

The Kitchen view selects records in Accepted, In progress, or Ready states and displays up to four active tickets. Each ticket shows a KOT-style identifier, source, order type, items, status, and an action for the next state. A default preparation target of 18 minutes is used when a record has no target value. The displayed elapsed and remaining values are computed from the demonstration minutesAgo field; the progress bar and delayed-order warning are illustrative estimates, not clock-driven measurements. An Accepted ticket begins at zero progress and a Ready ticket is shown at full progress.

The Reports view computes order count, total of stored order amounts, completed-order count, source-channel counts and amounts, and counts by status from the current in-browser array. The total is not a reconciled payment ledger, and the report does not include historical date ranges, refunds, taxes, discounts, exports, or a verification step. Report values are therefore useful for demonstrating aggregation logic, but they should not be used for accounting or business decisions.

The workflow begins with a source record. In the intended production flow, delivery-platform notifications would arrive through an authorised connector; in the current prototype, sample platform records appear in a labelled demonstration inbox. Walk-in orders are entered by restaurant staff. After review and acceptance, records are represented in a common order format, displayed in the Orders view, advanced through kitchen preparation states, and included in report summaries. Figure 4 distinguishes the future platform event from the current demonstration and manual walk-in entry.

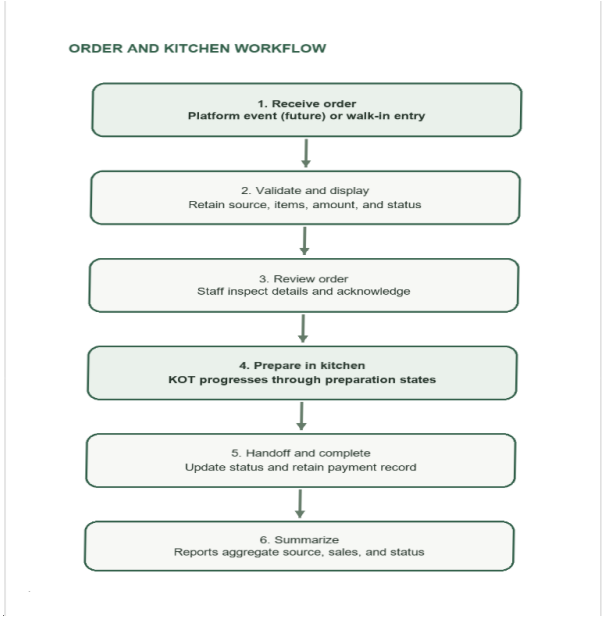


Fig. 4. Intended order lifecycle. Platform event reception is a future integration; the prototype simulates platform records.

7. System Design

7.1 Orders view

The Orders view is designed for front-of-house or owner review. The list shows an order identifier, customer, item description, source, elapsed time, status, and total. Selecting an order opens its detail panel, which can show contact information, delivery address or table, payment method and status, and special instructions. Manual order creation is restricted to walk-in orders so that platform orders are not represented as staff-entered records.

7.2 Kitchen View

The Kitchen view emphasizes preparation rather than customer/payment review. Each kitchen ticket shows a KOT-style identifier, source, order type, item description, preparation progress, elapsed or estimated time, and a status control. The demonstration state sequence is Accepted, In progress, Ready, and Completed; platform-labelled samples first appear as Received in the notification inbox. The delayed-order highlight and visual timer are illustrative: they use demo record fields and are not tied to a server clock or a validated kitchen start event.

7.3 Reports View

The Reports view calculates current totals, source-channel counts and sales, and order-status counts from the browser's stored records. These are descriptive summaries of the local demo dataset. The prototype does not produce audited sales reports, date-range comparisons, export files, or multi-location analytics.

7.4 Sign-in and profile

The sign-in screen asks for a display name, email, and password. In the current prototype, the name is passed into the dashboard welcome heading, sidebar profile, and avatar initials. Profile edits are saved locally. The form is not a real authentication system: it does not validate accounts, hash passwords, establish server sessions, or enforce roles. The demo must not be used to protect real customer or business data.

7.5 Module boundaries

The project proposal separates order aggregation, order management, kitchen workflow, payment management, analytics, and user management. In the implemented prototype, these modules are represented at the interface level rather than as independently deployed services. Order aggregation is represented by sample source-labelled records; order management by list, search, filters, and details; kitchen workflow by tickets and status controls; analytics by browser-side summaries; and user management by a demonstration profile. Payment information is displayed as a field, but no payment gateway or settlement reconciliation is implemented. This module distinction helps prevent a screen element from being mistaken for a completed external service.

8. Data Model and Order processing

The original proposal describes user, order, order-item, kitchen-status, payment, and report data. Figure 5 presents a normalized conceptual schema for a future backend. The browser prototype stores denormalized JavaScript objects in local storage and does not implement relational keys or constraints. Separating order items and kitchen events in a future database would allow multiple items and status changes to be retained without overwriting the order record.

Table 2. Candidate entities and fields for a future backend

Entity
User
Order
Order item
Kitchen event
Payment record

The candidate fields in Table 2 are a design proposal, not a database already created by the current app. A production implementation should define required and optional fields against the authorized platform contract. For example, an external platform order identifier is needed to recognize repeated webhook events, while the internal identifier allows the restaurant interface to remain consistent across channels. Each status change should be timestamped if the system is later used to calculate preparation or service-time metrics.

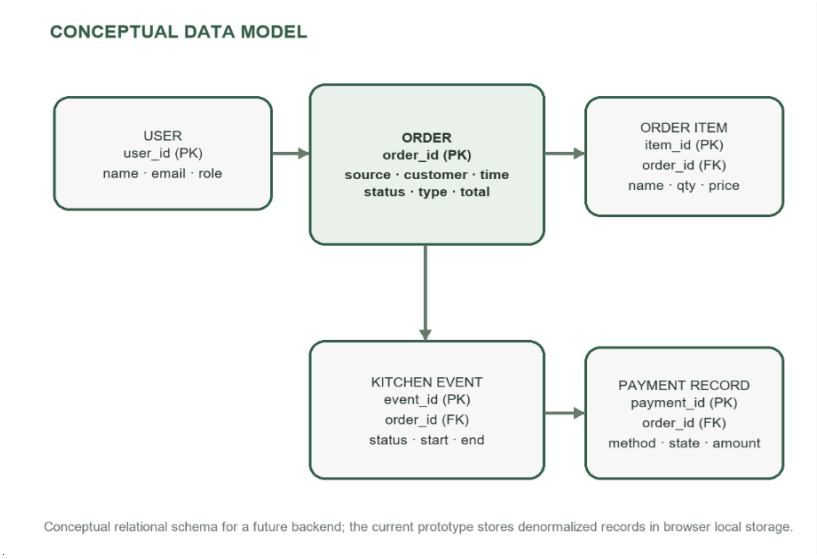


Fig. 5. Conceptual relational model for future server-side persistence; not implemented in the browser prototype.

The following algorithm describes the intended order lifecycle. Platform event validation and persistent writes are marked as future work; current platform-labelled records are loaded as demonstration data.

1. Receive an order record from an authorized platform connector, or create a walk-in record through the staff form.
2. Assign or retain a unique internal order identifier and record the source channel.
3. Validate required fields: source, items, amount, order time, and initial status.
4. Display the record in the Orders view and expose the relevant kitchen ticket.
5. Update preparation status as kitchen staff advance the ticket; preserve the resulting state for the order.
6. Mark the order complete and include the record in source, sales, and status summaries.

9. Prototype Demonstration and Implementation Status

The implemented artifact contains the main screens required for a prototype walkthrough: sign-in, overview, Orders, Kitchen, Reports, order details, walk-in creation, and profile editing. The navigation switches between separate Orders, Kitchen, and Reports views. Orders and Kitchen present different information priorities: the order screen supports customer/order review, while kitchen tickets emphasize KOT, source, preparation progress, and status actions. Reports recalculate from the browser's current demo order records.

Table 3 summarizes artifact coverage. The status column describes what a reviewer can inspect in the current local application and distinguishes it from the future service dependencies. The walkthrough uses no production account, restaurant database, or delivery-platform credential.

Table 3. Demonstrable prototype behavior and implementation boundaries

Area
Sign-in/profile
Notifications
Orders
Kitchen
Reports and storage

These results are implementation observations, not measured user or business outcomes. The prototype has not been tested with live restaurant staff, platform credentials, or production transactions. No sample size, task timing, error rate, satisfaction score, or comparative baseline is available. The current artifact therefore supports a design walkthrough and requirements discussion only.

9.1 Proposed Evaluation Plan

The evaluation is guided by two questions: RQ1: Can intended restaurant users complete key order-review and kitchen-status tasks with the prototype? RQ2: What usability difficulties do they encounter while completing those tasks? These questions remain unanswered because no participant evaluation has been conducted.

A future moderated usability walkthrough can involve restaurant owners and kitchen staff using synthetic orders. Representative tasks may include reviewing a platform-labelled sample notification, opening order details, entering a walk-in order, advancing a kitchen ticket, and interpreting a report summary. Participant recruitment should begin only after the institution’s required review and consent process.

The evaluation can record task completion, time on task, navigation or status-update errors, and participant feedback. Any report should state the number and roles of participants, task protocol, session conditions, and limitations. This is a proposed protocol only; no participant data or results are reported here.

10. Discussion

Separating the Orders and Kitchen views makes the intended division of work explicit while preserving the order identifier and source as common reference points. This can be evaluated with staff to determine whether the different information priorities help them locate an order and act on it. The current implementation cannot establish that benefit because role-specific use has not been observed in a restaurant setting.

The integration boundary is the main technical dependency. A browser page cannot infer or receive Swiggy or Zomato orders solely because it is open; an authorized partner interface or integration service must provide order events. The sample records illustrate the desired dashboard appearance but do not demonstrate API access. A backend also becomes necessary for cross-device synchronization, secure authentication, auditability, event deduplication, and recovery when a connection fails.

The conceptual database design should be reconciled with the actual platform data contract before implementation. Source identifiers, item modifiers, taxes, discounts, cancellation states, partial refunds, and pickup/delivery events may vary by integration. These fields should not be assumed until the authorized integration documentation is available.

10.1 Threats to validity

The current manuscript describes implementation behavior, so its validity is limited by the type of evidence available. Construct validity is limited because interface elements such as a preparation progress bar and report totals do not measure operational efficiency or financial accuracy. Internal validity is limited because there is no controlled comparison with the restaurant's existing workflow, and the prototype's displayed minutes are not derived from event timestamps. External validity is limited because the artifact has not been evaluated across multiple restaurants, operating periods, staff roles, or live platform connections. Reliability is also constrained by browser-local storage, the absence of a server-side event log, and the lack of multi-user concurrency handling. These limits preclude causal claims about reduced order time, fewer errors, or improved customer satisfaction.

11. Limitation and Future Work

The prototype has five primary limitations. First, platform-labelled orders and notifications are illustrative sample records and are not received in real time. Second, data and profile settings are stored only in the local browser. Third, the sign-in flow does not provide authentication or authorisation. Fourth, kitchen delay indicators use demonstration values rather than verified preparation timestamps. Fifth, no empirical evaluation has been conducted. Reports are interface summaries rather than verified financial records. These limitations restrict the current artifact to demonstration and design discussion.

The next engineering phase should verify partner access, define normalized order and event schemas, implement server-side integration with retries and idempotency, store records in a secure database, and introduce role-based authentication. Consent and privacy handling should be specified before collecting real customer or operational data.

After authorized platform integration is available, a limited restaurant pilot could compare equivalent service periods using order-receipt-to-kitchen time, missed or duplicated orders, status-update latency, and task errors. The analysis should state the sample size, observation period, exclusions, and uncertainty. This pilot has not been conducted.

12. Conclusion

This manuscript documents a unified dashboard prototype for monitoring multi-platform order notifications and coordinating restaurant operations. It describes the problem, requirements, workflow, current browser implementation, proposed integration architecture, conceptual data model, and role-oriented views. The prototype presents platform-labelled samples in a demonstration notification inbox, supports manual walk-in entry, and provides separate Orders, Kitchen, and Reports screens. Live platform integration, secure backend services, and field evaluation remain future work. The manuscript therefore describes a design and implementation artifact without claiming performance outcomes that have not been measured.

Statement and Declarations

Ethics and consent: This manuscript reports a software artifact and no study involving human participants or real customer data. Ethics approval and participant consent were therefore not applicable to the work reported here. Any future user study or pilot must follow the institution's review and consent requirements before data collection.

Data availability: No empirical dataset was generated. The records shown in the prototype are illustrative demonstration data.

Generative AI disclosure: OpenAI Codex was used to assist with drafting and editing this manuscript. It was not used to collect data or generate empirical results. The author is responsible for the accuracy and integrity of the final submitted version.

REFERENCES

1. Galabi, A. V., Vaze, A. N., Anusha, R., Chandana, S., & Sattigeri, S. K. (2023). Smart Restaurant Management System. International Journal of Engineering Research & Technology (IJERT), 11(5), ICEI 2023.https://www.ijert.org/smart-restaurant-management-system
2. Hevner, A. R., March, S. T., Park, J., & Ram, S. (2004). Design science in information systems research. MIS Quarterly, 28(1), 75–105. https://aisel.aisnet.org/misq/vol28/iss1/6/
3. Kalyan, H., & Stoney, G. (2023). Cloud-integrated online food ordering application using Django and REST API. International Journal of Data Science and IoT Management System, 2(4), 19–21. https://ijdim.com/journal/index.php/ijdim/article/view/85
4. Mayya, R., & Li, Z. (2025). Growing platforms by adding complementors without a contract. Information Systems Research, 36(3), 1670–1690. https://pubsonline.informs.org/doi/10.1287/isre.2023.0237
5. Peffers, K., Tuunanen, T., Rothenberger, M. A., & Chatterjee, S. (2007). A design science research methodology for information systems research. Journal of Management Information Systems, 24(3), 45–77. https://www.jmis-web.org/articles/765
6. Rout, A., Dawande, M., & Janakiraman, G. (2024). Cloud-kitchens: Value creation through co-location. Production and Operations Management, 33(2), 512–529. https://journals.sagepub.com/doi/10.1177/10591478231224950
7. Švancár, S., Chrpa, L., Dvořák, F., & Balyo, T. (2024). Cloud Kitchen: Using planning-based composite AI to optimize food delivery processes. arXiv:2402.10725.https://arxiv.org/abs/2402.10725
8. Wang, X. (2024). A web-based restaurant order management system [Thesis]. LUT University. https://lutpub.lut.fi/handle/10024/167478