Design and Implementation of a Regional Culinary Community Platform with AI-Assisted Question Answering: A Case Study of Xunwei Zhanjiang

YiZhi Cai

School of International Education, Neusoft Institute Guangdong, Foshan, Guangdong, 528225, China

Keywords

Regional culinary culture;
Community platform;
Web application;
Conversational AI;
System design

ABSTRACT

Digital platforms for regional culinary culture need to connect information about local dishes and places with community participation and accessible exploration services. This study presents the design and implementation of Xunwei Zhanjiang, a community-oriented web platform that integrates regional food discovery, user interaction, and AI-assisted culinary enquiry. Built with Next.js, React, Drizzle ORM, and MySQL, the platform organizes culinary information through district-based displays and structured food cards, alongside community features for sharing posts, commenting, bookmarking, and following users. A server-side integration with Dify enables natural-language enquiries about local cuisine and suggested food itineraries. The system incorporates standardized API responses, input validation, cookie-based authentication, and database uniqueness constraints to support consistent interactions and prevent duplicate social relationships. Preliminary functional checks confirmed successful operation of selected content retrieval endpoints, user login and profile retrieval, access restrictions, and the integrated AI enquiry interface. These observations support the feasibility of bringing structured culinary content, community interaction, and conversational assistance together within a regional web application. However, they do not establish improvements in recommendation quality, cultural engagement, or performance at scale. Further work will address implementation gaps and evaluate response quality, system performance, and usability through systematic testing and user studies.

CONTACT:

[1]Author: 17322905680@qq.com

*Corresponding Author: 17322905680@qq.com

Note: Authors designated with "†" are recognised as co-first authors.

DOI.10.64549/jaai-ii.v1i2.124

This work is licensed under the CC BY 4.0 HTTPS://CREATIVECOMMONS.ORG/LICENSES/BY/4.0/

1. Introduction

People exploring a region's cuisine often need more than a restaurant rating or an isolated answer to a question. They may want to understand what makes a local dish distinctive, discover the culinary characteristics of different districts, read other people's experiences, and ask questions before deciding what to try. These activities are closely connected, yet users may need to move between merchant listings, community posts, and conversational tools to complete them. For a regional culinary platform, the practical challenge is to bring these activities together in a clear and accessible environment.

Review-oriented services, exemplified by Dianping, place considerable emphasis on merchant discovery and consumption decisions through user reviews, ratings, and rankings. Such services also incorporate AI functions, so the distinction cannot simply be reduced to review platforms versus AI platforms. Instead, this study focuses on a different organizing principle: presenting regional cuisine as a combination of food knowledge, geographical context, and community experience. A merchant-centered interface does not necessarily make the origins, preparation methods, seasonal characteristics, and cultural meanings of local dishes the primary route through which users explore a region. Similarly, a standalone conversational interface can answer questions but does not, by itself, provide an organized collection of culinary content or a persistent community in which users publish experiences, bookmark posts, and follow other contributors.

These considerations motivate Xunwei Zhanjiang, a regional culinary community platform centered on the food culture of Zhanjiang, China. The platform brings together three activities: exploring structured culinary information, sharing food-related experiences, and asking natural-language questions. Food cards present information about local dishes, while district-based browsing provides geographical context. Community functions support posts, comments, bookmarks, and following relationships. An AI-assisted question-answering interface provides an additional way for users to enquire about local cuisine. The design objective is to make these complementary functions available within one application.

The integration is implemented using Next.js App Router, with React components providing the user interface and server-side Route Handlers handling application requests. MySQL stores culinary and community records, and Drizzle ORM supports database access. Dify provides the external conversational service through a dedicated server-side endpoint. Database-backed content operations and AI requests follow separate processing paths. Accordingly, the platform integrates these functions at the application level; it does not establish that AI responses are grounded in the platform's database or that the system performs automated route optimization.

This paper makes three contributions. First, it presents the design of a regional culinary platform that combines structured food information, community participation, and AI-assisted enquiry. Second, it describes the implementation of the application architecture, persistent data relationships, and controlled access to the external conversational service. Third, it reports preliminary functional verification and identifies the implementation limitations observed during testing. The contribution is an implemented application and an account of its design, rather than a new recommendation algorithm or demonstrated improvement in user engagement.

The remainder of this paper is organized as follows. Section 2 reviews related work on digital culinary information, community content, and conversational systems. Section 3 presents the system design and overall architecture. Section 4 describes the implementation of the principal modules. Section 5 reports the evaluation results and limitations. Section 6 concludes the paper.

2. Related Work

2.1 Culinary Culture and Web-Based Information Presentation

Research on culinary tourism provides a basis for deciding what regional food platforms should represent. Horng and Tsai (2010) examined government tourism websites across six Asian destinations, focusing on how they presented local foods, restaurants, recipes, culinary tours, and dining cultures. Their analysis treats culinary information as part of a destination's cultural representation, extending beyond descriptions of individual dishes. This perspective informs the present study's decision to organize food content through geographical context, preparation methods, and cultural descriptions. However, their investigation primarily concerns destination promotion through official websites, whereas Xunwei Zhanjiang also supports contributions from individual users. (Horng and Tsai, 2010)

For the present platform, structured food cards provide a consistent format for presenting culinary information, while district-based browsing situates that information within Zhanjiang. These are application design choices informed by earlier research on culinary presentation. Whether they improve users' understanding of local food culture requires empirical evaluation.

2.2 User-Generated Content and Community Participation

User-generated content provides another perspective on tourism information services. Xiang and Gretzel (2010) investigated the representation of social media in travel-related search results through simulated searches involving predefined keywords and nine United States destinations. They found that social media constituted a substantial part of the retrieved results, demonstrating its visibility within the online travel information environment. Their findings establish the relevance of user-contributed information to travel discovery, although they do not show that such content is necessarily accurate or more useful than professionally produced material. (Xiang and Gretzel, 2010)

This distinction is relevant to regional culinary platforms. Structured descriptions can provide consistent information fields, while posts and comments accommodate personal experiences. Xunwei Zhanjiang places these content forms within the same application and provides bookmarking and following functions to support continued access to content and contributors. The resulting design supports participation without treating individual posts as verified culinary knowledge. Content quality and moderation remain separate requirements from the technical ability to publish and interact.

2.3 Conversational Assistance and Knowledge Grounding

Conversational recommendation research examines how users can express preferences, ask questions, and provide feedback during information discovery. Jannach et al. (2021) review approaches according to their supported interactions, underlying knowledge, and technical methods. Their survey provides a useful conceptual basis for introducing natural-language interaction alongside browsing and search. In the present platform, users can ask about dishes or request suggested food itineraries through a Dify-integrated interface. Its current implementation supports individual enquiries rather than a complete conversational recommendation process with persistent preference tracking. (Jannach et al., 2021)

Knowledge grounding is a related technical consideration. Lewis et al. (2020) introduced retrieval-augmented generation, combining a generative model with an external retrieval mechanism for knowledge-intensive language tasks. This approach offers a relevant direction for future integration of documented local culinary information. However, connecting a website to a language-model service does not itself establish that retrieval-augmented generation has been implemented. Accordingly, this study describes the verified Dify connection as AI-assisted enquiry and makes no claim that responses are grounded in the platform's food database. (Lewis et al., 2020)

Recent studies clarify the design choices involved in extending a conversational interface with retrieval. Self-RAG introduces adaptive retrieval and self-reflection to assess retrieved passages and generated responses (Asai et al., 2024). Wang et al. (2024) examine combinations of RAG components and the balance between

response quality and processing efficiency. LightRAG incorporates graph structures and dual-level retrieval to represent entities and relationships (Guo et al., 2025). These approaches provide alternatives for organizing documented dishes, locations, and cultural associations, but none was implemented or evaluated in the present platform.

The use of retrieved information also requires careful context design. Liu et al. (2024) show that language-model performance can vary with the position of relevant information in a long input context. For a regional culinary application, this finding motivates evaluating passage selection and ordering rather than assuming that supplying more content necessarily produces a better answer.

Evaluation must also distinguish successful operation from useful assistance. Jannach (2023) emphasizes that conversational recommender evaluation involves both component performance and users' perceptions of the overall system. This distinction informs the evaluation scope of the present study: functional checks can establish whether requests are processed successfully, while answer relevance, factual correctness, and user satisfaction require separate assessment. (Jannach, 2023)

RAGAs provides complementary evaluation methods that distinguish retrieval relevance, answer faithfulness, and generation quality (Es et al., 2024). These dimensions would become relevant if culinary-document retrieval were added, but no RAGAs scores are reported here. The present evidence concerns successful service communication and selected application workflows; it does not measure whether generated answers are supported by retrieved sources.

Deployment research also separates the behavior of a web application from the efficiency of model inference. Sarathi-Serve addresses the throughput-latency tradeoff through chunked prompt processing and inference scheduling (Agrawal et al., 2024). Xunwei Zhanjiang delegates inference to an external service and does not implement these scheduling techniques. Nevertheless, this work reinforces the need to assess latency under defined workloads rather than treating a single successful API request as evidence of scalable performance.

2.4 Positioning of the Present Study

The reviewed literature informs three complementary aspects of the platform: the presentation of culinary culture, participation through user-generated content, and natural-language assistance for information discovery. Xunwei Zhanjiang brings these aspects together in a region-specific implementation. Its contribution lies in documenting their integration through a shared web interface, persistent community data, and server-side service coordination. The study examines the feasibility and limitations of this implementation; improvements in cultural engagement or recommendation effectiveness remain questions for subsequent evaluation.

3. System Design and Architecture

3.1 Design Objectives and Overall Architecture

Xunwei Zhanjiang is designed to support three connected activities: exploring regional culinary information, sharing food-related experiences, and asking natural-language questions about local cuisine. These activities impose different requirements on the system. Food discovery requires consistent content organization and accessible browsing interfaces. Community participation requires persistent user identities and reliable relationships between users and content. AI-assisted enquiry requires controlled communication with an external service.

The platform adopts an integrated full-stack architecture built on Next.js App Router. React components provide the user interface, while Route Handlers expose backend endpoints within the same application. The system is logically divided into presentation, application service, and data access layers. MySQL stores culinary and community data, and Drizzle ORM supports schema definitions and database operations. Dify operates as

an external conversational service accessed through a dedicated server-side endpoint. Figure 1 illustrates the overall architecture of the Xunwei Zhanjiang platform.

This organization separates responsibilities without requiring independently deployed frontend and backend applications. Client components manage user input and interface state, backend endpoints validate requests and coordinate business operations, and the data access layer handles persistent records. The external AI connection follows a separate request path from database-backed content retrieval.

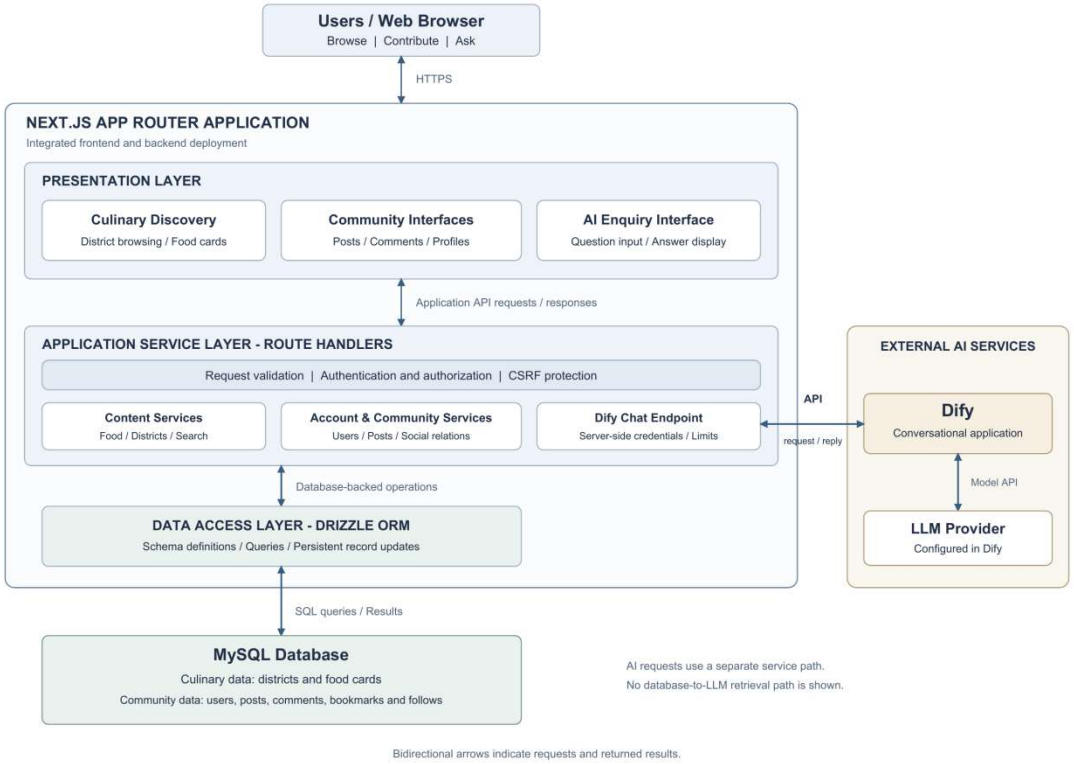


Figure 1. Overall architecture of the Xunwei Zhanjiang platform.

3.2 Functional Organization and User Roles

The presentation layer groups the platform's functions into culinary discovery, community participation, personal account management, and administrative management.

Culinary discovery includes district-based displays, food cards, and ranked food listings. Food cards organize information such as origin, seasonality, preparation methods, taste characteristics, and cultural descriptions. District displays provide geographical context, while ranked listings offer an additional browsing entry point.

Community functions include post publication, keyword search, comments, bookmarks, and user following. Personal pages provide access to a user's profile and associated posts, comments, and bookmarks. The conversational interface provides a complementary entry point through which users can ask questions in natural language.

The system distinguishes unauthenticated visitors, authenticated users, and a super-administrator role. Public endpoints support content browsing, while participation functions require authentication. Administrative endpoints apply an additional database-backed role check. The implemented administrative scope includes user

listing and the removal of posts and comments through content status changes. This scope does not imply a complete pre-publication moderation workflow.

3.3 Data Model and Relationship Integrity

The relational data model separates account information, community content, and curated culinary information. The principal community entities are users, posts, comments, bookmarks, and following relationships. Each post identifies its author, each comment identifies its author and associated post, and each bookmark connects a user to a post. Following records represent directed relationships between two users.

Composite uniqueness constraints on bookmark and following records prevent duplicate user–post and follower–followee pairs. Backend checks verify the existence of relevant entities before selected write operations. These checks complement database constraints by returning application-level errors when a requested target is unavailable. The toggle behavior of bookmarking and following remains state-dependent; uniqueness constraints prevent duplicate records but do not make repeated toggle requests universally idempotent.

Curated culinary information is maintained separately from community posts. Food cards store structured descriptions, while regional display records and associated carousel content support district-based exploration. Separating these content types allows the application to present both organized reference material and individual experiences without assuming that user posts have undergone factual verification.

The schema also defines upload assets, temporary upload claims, and references linking uploaded files to application entities. These structures are intended to support file reuse and lifecycle management. However, the inspected local database did not yet contain these tables, so the upload subsystem is treated as an implementation gap rather than a validated component.

3.4 API Coordination and Authentication

Backend endpoints use a shared response structure containing success status, a response code, a message, and optional data and metadata. Paginated responses place pagination information within the metadata object. This convention gives client components a consistent basis for interpreting results and displaying errors.

Shared validation utilities check request bodies, identifiers, pagination parameters, and selected field constraints. Authentication uses signed JSON Web Tokens stored in HttpOnly cookies. State-changing endpoints additionally perform cross-site request forgery checks before processing protected operations. Password handling supports BCrypt hashes and the migration of legacy plaintext passwords following successful verification.

These mechanisms provide a common foundation for request processing, although their presence alone does not establish comprehensive system security. In particular, consistent enforcement of account status across protected business endpoints remains an area for further improvement.

3.5 Integration with the Conversational Service

AI enquiries follow the path from the browser to the platform's /API/Dify/Chat endpoint and then to the configured Dify application. The backend checks the request's authentication and CSRF token, validates the question, and applies an in-memory request limit before contacting the external service. The Dify application key is read from server-side configuration.

The current implementation sends the question, an empty application-input object, and a user identifier derived from the authenticated account. It uses a blocking response mode with a 15-second timeout. The returned answer is then included in the platform's standard API response.

This integration establishes a clear service boundary. The website does not currently forward its food cards or community posts as retrieval context, and it does not pass a conversation identifier between requests. Consequently, the implemented capability is single-request conversational assistance. Persistent dialogue,

database-grounded answers, and geographic route optimization would require additional components and validation.

4. Key Technical Implementations

4.1 Standardized Request Processing and API Responses

The backend uses shared utilities to organize request validation, authentication, response construction, and error handling. This arrangement provides a consistent processing pattern across endpoints while allowing each endpoint to retain its own business logic. For protected state-changing operations, the typical sequence consists of CSRF verification, authentication, request parsing, field validation, business-condition checks, database operations, and response construction. Invalid requests are rejected before the corresponding business operation is performed.

A common response envelope contains the fields success, code, message, data, and meta, as applicable to the response. The success flag indicates the application-level outcome, the code identifies the result category, and the message provides a description suitable for client-side presentation. Business results are placed in the data field, while supplementary information, including pagination details, is placed in metadata. HTTP status codes accompany this envelope to distinguish successful requests, invalid input, authentication failures, unavailable resources, and server-side errors.

Shared validation functions handle recurring requirements such as non-empty strings, positive integer identifiers, email formats, permitted category values, and pagination bounds. Route-specific checks supplement these utilities when an operation has additional constraints. For example, the post publication endpoint restricts categories and locations to supported values and verifies that the image collection is an array of strings. This division centralizes common validation rules while keeping domain-specific requirements close to their associated operations.

Consistent responses also simplify client-side processing. Components can interpret the same success indicator and message fields across different endpoints instead of maintaining separate response conventions. However, a standardized envelope does not eliminate the need to distinguish recoverable input errors from service failures. Detailed exceptions remain relevant to server-side diagnosis, while client responses use controlled messages.

4.2 Authentication and Protection of State-Changing Requests

The login process separates account lookup, password verification, and session creation. After validating the request, the backend retrieves the account associated with the submitted email address. A shared password utility verifies the supplied password against the stored representation. BCrypt is used for hashed passwords. For legacy accounts containing plaintext passwords, successful verification triggers a replacement with a BCrypt hash. This migration mechanism allows existing accounts to move to the newer storage format during normal login.

Once verification succeeds, the backend checks whether the account is disabled and signs a JSON Web Token containing the user identifier. The token is stored in an HttpOnly cookie. Subsequent protected requests are processed by a shared authentication helper that reads the cookie, verifies the token, and exposes the authenticated user identifier to the endpoint. Business operations consequently obtain the acting user’s identity from the verified session rather than accepting it directly from request data.

State-changing requests additionally require a CSRF token. The implementation generates a token containing a timestamp, a random nonce, and a signature derived from the configured secret. The client obtains this token through a dedicated endpoint and sends it in a request header. The server verifies that the header and

cookie values match and that the signature and validity period are acceptable. A shared client-side request helper coordinates this process for protected operations.

Administrative requests apply a further authorization step. The backend retrieves the current account role and status from the database before permitting administrative access. This avoids relying solely on role information embedded in an earlier token. Nevertheless, authentication and account-state enforcement remain distinct concerns: the general authentication helper verifies token validity, and ordinary business endpoints do not yet consistently enforce disabled-account status.

4.3 Community Content Publication and Retrieval

Post publication combines content validation with checks on the author and any uploaded images. The endpoint validates the title, description, category, location, and image collection, then verifies that the authenticated account still exists. It generates a short excerpt from the description for use in content listings.

Where uploaded images are included, the backend checks the user's current upload claims before creating the post. These claims associate uploaded resources with an authenticated user and an intended purpose. The post and its image references are subsequently written within a database transaction. This transaction groups the related database changes so that a failure does not leave a successfully committed post with only part of its intended reference records.

For content retrieval, the community listing endpoint joins posts with author information and related interaction records. Keyword search uses partial matching against post titles and descriptions. Active content is selected through status conditions, and results are ordered by creation time.

Aggregating several one-to-many relationships in the same query requires particular care. A post with multiple comments and bookmarks can produce repeated combinations of rows after joins. The implementation uses distinct comment and bookmark identifiers when calculating counts, avoiding inflation caused by these combinations. This is a query-correctness mechanism; it should not be interpreted as a guarantee of efficient execution at arbitrary data scales. The current general post listing also lacks bounded pagination, which remains an important consideration as the dataset grows.

4.4 Bookmarking, Following, and Concurrent Requests

Bookmarking and following are implemented as relationship operations between existing entities. A bookmark connects a user to a post, while a following record connects a follower to another user. Before changing a relationship, the backend checks authentication, validates the target identifier, and verifies the relevant resource. Following additionally rejects attempts to follow the same account.

The endpoints use toggle semantics. If the relationship exists, the operation removes it; otherwise, the operation attempts to create it. Composite uniqueness constraints on the user–post and follower–followee pairs provide database-level protection against duplicate records.

The implementation also handles a specific insertion race. Two requests may both observe that a relationship is absent and then attempt to insert it. If one succeeds and the other encounters a duplicate-key error, the latter is interpreted as an already-established relationship rather than an unexpected server failure. This preserves the uniqueness of the relationship and returns a meaningful state to the client.

The distinction between duplicate prevention and idempotency is important. Repeating a toggle after the first request has completed can reverse its effect. Therefore, these endpoints provide duplicate-record protection and handling of concurrent insertions, but they do not guarantee that every repeated request has the same effect. Explicit "add" and "remove" operations would provide clearer semantics for future retry-sensitive clients.

4.5 Image Processing and Upload Resource Management

The upload implementation authenticates the request before reading its multipart body. It then checks the uploaded file and processes the image using file-type detection and image decoding. Supported formats are

JPEG, PNG, and WebP. The processing code imposes a 5 MB input limit and rejects images whose width or height exceeds 4096 pixels.

Image processing depends on the intended use. Avatar images are resized to a square format and encoded as WebP. Post images are resized within a maximum bounding dimension while avoiding unnecessary enlargement. The processed image is then used to calculate a SHA-256 content hash, from which a managed storage path is constructed.

The resource model separates physical assets, temporary upload claims, and persistent references. An asset represents a processed file, a claim records a user's temporary right to attach that asset, and a reference connects the asset to a post or profile. This separation allows identical processed files to share an asset while retaining user-specific attachment checks. It also provides the basis for identifying resources that are no longer referenced.

Database transactions do not automatically include filesystem writes. Consequently, file creation and database updates have different failure boundaries, and resource cleanup remains necessary to handle unreferenced files. Although these mechanisms are present in the code, missing local dependencies and upload-management tables prevented complete runtime verification during the inspection. Their operational behavior must therefore be validated after environment alignment.

4.6 Server-Side Integration with Dify

The conversational interface sends questions to an internal endpoint rather than calling Dify directly from the browser. The server reads the application key from environment configuration, validates the question, and associates the request with the authenticated user. Questions are limited to 500 characters, and an in-memory rate limiter permits up to 20 requests per user within a 60-second window.

The backend sends a blocking request to Dify and applies a 15-second timeout. Successful answers are returned through the platform's standard response structure. Upstream failures and request timeouts are converted into controlled service-error responses. This design centralizes credentials and external-service handling within the backend.

Two limitations affect the current user experience. First, the interface maintains message history locally, but the backend does not forward a conversation identifier or prior messages. Visible history therefore does not imply retained model context. Second, the returned answer is displayed without separating reasoning markup from the final response. Model output containing such markup can consequently appear directly in the conversation view.

The rate limiter also has a deployment boundary: its counters reside in application memory. They are neither shared across multiple instances nor preserved across restarts. A distributed deployment would require coordinated storage if the same limits were to apply consistently across instances.

System Prompt Configuration. The application backend forwards the user's question to Dify without embedding a system prompt in the API request. Assistant instructions are therefore managed separately from the Next.js request-handling logic through the Dify application configuration. This separation allows the conversational role and response style to be maintained independently of authentication, input validation, and external service communication. The local implementation does not include a versioned copy of the deployed prompt, so its exact wording is not reproduced here. Furthermore, prompt-based instructions alone do not establish that responses are grounded in the platform's culinary database.

4.7 Content Status and Administrative Operations

Posts and comments use status fields to distinguish active content from content removed by users or administrators. Public retrieval applies status conditions to exclude removed content from relevant views.

Administrative deletion changes the status rather than physically removing the record, preserving the underlying entry for subsequent inspection.

This mechanism separates content visibility from physical storage and provides a basic basis for administrative intervention. It does not, by itself, constitute a complete moderation system. Review queues, restoration procedures, reporting workflows, and comprehensive administrative audit trails would require additional implementation. The next section evaluates the implemented behavior and distinguishes verified results from these remaining development needs.

5. Results and Discussion

5.1 Evaluation Scope and Procedure

The evaluation examined whether the local implementation supported its principal information-access and authenticated interaction workflows. It combined HTTP-level functional checks, inspection of the database and installed dependencies, static analysis, and execution of the project's automated tests and production build. These activities addressed operational feasibility and implementation consistency. They did not constitute a controlled assessment of user experience or recommendation effectiveness.

The inspected application ran in a Windows development environment with a local MySQL database and an external Dify application. The installed Next.js version was 16.2.2, although the project manifest specified version 16.2.7. This discrepancy was recorded because differences between declared and installed dependencies affect the reproducibility of an evaluation.

At the time of inspection, the local database contained six user records, eleven posts, twelve comments, six bookmarks, one following relationship, twelve food cards, and nine regional display records. These records provided sufficient material for checking selected retrieval and account workflows. They were development data rather than a recruited participant sample, and their size does not support conclusions about large-scale community activity.

The evaluation used existing records to inspect selected read operations and access restrictions. It did not systematically exercise every state-changing workflow, device configuration, or failure condition. Consequently, the reported results concern the operations explicitly checked rather than complete acceptance of the application.

5.2 Functional and Security Verification

Verification was organized into two groups. Table 1 summarizes interface and API checks for account access and content operations. Table 2 covers access control and selected edge cases. Results distinguish runtime verification from code inspection: an implemented safeguard does not, by itself, demonstrate that the corresponding test has passed.

Table 1. Interface/API Verification

Test case	Operation and expected result	Verification result
Account login	Submit valid credentials; establish an authenticated session and allow access to the user profile.	Verified after applying the existing database migration.
Culinary content retrieval	Request district information and food cards; return the corresponding records successfully.	Selected endpoints verified.

Test case	Operation and expected result	Verification result
Post retrieval	Request the post list and an existing post's details; return the corresponding content.	Selected endpoints verified.
Personal content retrieval	Request the authenticated user's posts, comments, and bookmarks; return the corresponding records.	Selected endpoints verified.
Post creation	Submit valid post content as an authenticated user; persist the post and make it retrievable.	Implemented; end-to-end verification pending.
Post update and deletion	Modify or delete an owned post; confirm the resulting database state and API response.	End-to-end verification pending.
AI enquiry	Submit a question through the authenticated chat endpoint; receive a response from Dify.	A sampled request returned HTTP 200 and a response. Answer quality was not evaluated.

Table 2. Security and Edge Cases

Test case	Operation and expected result	Verification result
Unauthenticated administrative access	Request an administrative endpoint without a valid session; reject access.	Verified: HTTP 401 returned.
Insufficient administrative privileges	Request an administrative endpoint using an ordinary user account; reject access.	Verified: HTTP 403 returned.
Cross-user content modification	Attempt to modify or delete another user's content; reject the operation and preserve the record.	Runtime verification pending.
Invalid CSRF token	Submit a protected state-changing request with a missing or invalid CSRF token; reject the request.	Protection identified in code; negative-request testing pending.
Concurrent follow requests	Send simultaneous follow requests for the same user pair; confirm that no duplicate relationship records are created.	A unique constraint and duplicate-insert handling were identified; concurrent execution testing pending.
Concurrent bookmark requests	Send simultaneous bookmark requests for the same user–post pair; confirm that no duplicate relationship records are created.	A unique constraint and duplicate-insert handling were identified; concurrent execution testing pending.
AI input and service limits	Submit an oversized question, exceed the request limit, or simulate an upstream timeout; return a controlled error.	Input limits, rate limiting, and timeout handling were identified in code; boundary testing pending.

The verified checks demonstrate that selected account, content retrieval, and AI communication paths operated in the inspected environment. Administrative access checks also confirmed rejection of unauthenticated and insufficiently privileged requests. However, complete CRUD coverage, cross-user

modification checks, and concurrent relationship operations remain unverified. In particular, database uniqueness constraints support duplicate prevention, but do not establish the behavior of simultaneous toggle requests without execution-based testing.

5.3 Database and Dependency Consistency

The inspection identified a mismatch between the application schema and the local database. The login endpoint attempted to retrieve user-role and account-status fields that were absent from the existing user table. The resulting database exception caused the application to return an internal-server-error response before password verification could complete.

The project's existing incremental migration was applied to add the relevant user fields, post and comment status fields, and associated indexes. Record counts for users, posts, and comments remained unchanged after this operation. Subsequent login and profile checks succeeded, supporting the diagnosis that the immediate failure resulted from schema inconsistency rather than invalid account credentials.

Further inspection found that the local database did not contain the three tables required by the upload-management subsystem: upload assets, upload claims, and upload references. The installed dependency environment also lacked resolvable copies of sharp and file-type, although both were declared in the project manifest. These findings explain why the availability of basic browsing and login did not imply that image-dependent workflows were ready for use.

The observed inconsistencies highlight a reproducibility requirement for the system. A working copy of the source code must be accompanied by a matching dependency installation and a database migration state. Verifying only that the development server starts is insufficient because individual routes may depend on components that are not exercised during initial startup.

5.4 Static Analysis, Automated Tests, and Build Results

Static analysis and automated execution produced different outcomes, reflecting the scope of each check. Type generation and TypeScript checking completed successfully. ESLint reported no errors and nineteen warnings, including image-usage recommendations, unused variables, and React Hook dependency warnings. These results indicate that the configured checks completed, but they do not demonstrate complete correctness of the JavaScript business logic.

The automated test runner reported fifty-two passes and nine failures across sixty-one reported test entries. This summary requires qualification: two test files failed to load because of a missing dependency, so the output does not represent sixty-one fully executed business-level assertions. Six schema-export tests failed when a subprocess could not resolve the pnpm command on Windows. A further failure arose from a path assertion that expected forward slashes while the implementation returned a Windows-style path.

The production build failed because the upload-processing module could not resolve sharp and file-type. Table 3 presents the results without treating checks of different scope as interchangeable.

Table 3. Engineering verification results in the inspected environment.

Check	Result	Main observation
Type generation and TypeScript checking	Passed	Configured type-checking process completed
ESLint	0 errors; 19 warnings	Maintainability and interface-related warnings remained
Automated tests	52 passed; 9 failed	Included file-load failures and Windows compatibility failures
Production build	Failed	Upload-processing dependencies could not be resolved
Dependency consistency	Incomplete	Installed packages did not fully match the manifest
Database consistency	Partially aligned	Authentication-related fields were corrected; upload tables remained absent

The failures should be interpreted according to their causes. A test-process launch failure does not demonstrate that the underlying database schema is incorrect, and a platform-specific path assertion does not necessarily indicate faulty file handling. Nevertheless, these failures prevent the test suite from providing complete evidence for the inspected environment. Likewise, successful static analysis cannot compensate for a failed production build.

5.5 Verification of AI Service Integration

AI integration was examined at both the configuration and request-processing levels. The initial local configuration used environment-variable names that did not match those read by the backend. Correcting these names enabled the server to access the configured Dify application key. An application-information request then succeeded, confirming that the key was accepted for the application at the time of testing.

A subsequent conversational request returned an error indicating that the previously configured model was unavailable. After the model configuration was updated, a short culinary question submitted through the website's authenticated backend returned a successful answer. The measured interval for that individual request was approximately 2.7 seconds.

This observation demonstrates successful execution of the tested request, not a performance benchmark. The measurement did not include repeated trials, a controlled network configuration, varied question lengths, or concurrent users. It therefore cannot establish typical latency, tail latency, or service reliability.

The response also contained reasoning markup that was passed through to the conversation interface. Although the service call succeeded, the displayed output required further processing to present only the intended user-facing content. In addition, inspection confirmed that the integration did not forward a conversation identifier or previous messages. Follow-up questions relying on earlier context were therefore outside the verified capability of the website integration.

The distinction between connectivity and answer quality is particularly important. A successful HTTP response confirms neither the factual accuracy of local food information nor the practicality of a suggested itinerary. These properties require a question set, explicit assessment criteria, and independent review.

5.6 Discussion of System Feasibility

The results provide preliminary evidence that structured culinary information, authenticated community queries, and an external conversational service can operate within the same regional web application. Selected request paths successfully crossed the presentation, application, and data-access boundaries defined in the architecture. This supports the technical feasibility of the integration approach at the scale and under the conditions examined.

The inspection also showed that functional availability was uneven across subsystems. Browsing, login, and AI enquiry could operate while image processing and production packaging remained incomplete. This matters when interpreting a prototype demonstration: visible success in one workflow should not be generalized to the entire platform.

The authentication failure illustrates the practical importance of coordinating code and database changes. The application contained the expected account logic, but the database could not satisfy the query generated from the current schema. Applying the corresponding incremental change restored the tested workflow without changing the number of existing user or content records. The finding supports treating migration state as part of the evaluated system configuration.

The AI observations similarly show that an external-service dependency introduces several distinct failure points. Local configuration, application-key validity, model availability, request handling, and response presentation can fail independently. Separating these stages made it possible to identify configuration and model problems without attributing every failure to an invalid API key.

The study's present contribution is therefore an implementation case with documented operational evidence and limitations. The findings do not establish increased cultural participation, improved food-discovery efficiency, or superiority over existing platforms. Those outcomes remain empirical questions.

5.7 Limitations and Further Evaluation

The evaluation has four principal limitations. First, the dataset was small and development-oriented. Second, no systematic user study was conducted. Third, automated verification was incomplete because of dependency and cross-platform execution problems. Fourth, the AI check involved a limited request example rather than a representative evaluation set.

Further evaluation should begin with environment alignment and successful execution of the complete test and build processes. End-to-end checks should then cover registration, login, image upload, post publication, commenting, bookmarking, following, and relevant administrative actions. Tests should include both successful operations and expected rejection paths.

AI evaluation should use a documented set of culinary questions covering dish descriptions, regional distinctions, dietary preferences, and itinerary requests. Human assessment can examine factual correctness, relevance, clarity, and unsupported claims. Context-dependent follow-up questions should be evaluated separately if persistent conversation support is introduced. Because user input is forwarded to an external language-model service, AI safety also requires explicit evaluation, including resistance to prompt injection and the effectiveness of mitigation measures; studies of indirect injection in tool-integrated agents further motivate testing retrieved content if RAG is introduced (Zhan et al., 2024), although neither these attacks nor their mitigations were evaluated in this study.

Performance evaluation should record the deployment configuration, dataset size, concurrent workload, response-time distribution, and error rate. Database-backed endpoints and external AI requests should be

reported separately because they depend on different resources. A subsequent user study could measure task completion, errors, perceived usefulness, and satisfaction. Any claim of improvement should be tied to a clearly defined comparison condition and reported observations.

6. Conclusion

This paper presented the design and implementation of Xunwei Zhanjiang, a regional culinary community platform integrating structured food information, community interaction, and AI-assisted question answering. Built with Next.js, MySQL, Drizzle ORM, and Dify, the platform supports culinary exploration, experience sharing, and conversational enquiry within a single application. Preliminary functional checks support the technical feasibility of this integration within the tested scope.

The next steps are to improve the testing environment and resolve outstanding dependency and build issues, introduce retrieval-augmented generation (RAG) to ground AI responses in curated regional culinary information, and conduct user studies to assess usability, response usefulness, and support for local food discovery.

References

Agrawal, A., Kedia, N., Panwar, A., Mohan, J., Kwatra, N., Gulavani, B., Tumanov, A. and Ramjee, R., 2024. Taming throughput-latency tradeoff in LLM inference with Sarathi-Serve. In: 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24), pp.117–134. Available at: https://www.usenix.org/conference/osdi24/presentation/agrawal.

Asai, A., Wu, Z., Wang, Y., Sil, A. and Hajishirzi, H., 2024. Self-RAG: Learning to retrieve, generate, and critique through self-reflection. In: The Twelfth International Conference on Learning Representations. Available at: https://openreview.net/forum?id=hSyW5go0v8.

Es, S., James, J., Espinosa Anke, L. and Schockaert, S., 2024. RAGAs: Automated evaluation of retrieval augmented generation. In: Proceedings of the 18th Conference of the European Chapter of the Association for Computational Linguistics: System Demonstrations, pp.150–158. doi: 10.18653/v1/2024.eacl-demo.16.

Guo, Z., Xia, L., Yu, Y., Ao, T. and Huang, C., 2025. LightRAG: Simple and fast retrieval-augmented generation. In: Findings of the Association for Computational Linguistics: EMNLP 2025, pp.10746–10761. doi: 10.18653/v1/2025.findings-emnlp.568.

Horng, J.S. and Tsai, C.T., 2010. Government websites for promoting East Asian culinary tourism: A cross-national analysis. Tourism Management, 31(1), pp.74–85. doi: 10.1016/j.tourman.2009.01.009.

Jannach, D., 2023. Evaluating conversational recommender systems: A landscape of research. Artificial Intelligence Review, 56, pp.2365–2400. doi: 10.1007/s10462-022-10229-x.

Jannach, D., Manzoor, A., Cai, W. and Chen, L., 2021. A survey on conversational recommender systems. ACM Computing Surveys, 54(5), Art. no. 105, pp.1–36. doi: 10.1145/3453154.

Lewis, P. et al., 2020. Retrieval-augmented generation for knowledge-intensive NLP tasks. In: Advances in Neural Information Processing Systems, 33, pp.9459–9474.

Liu, N.F., Lin, K., Hewitt, J., Paranjape, A., Bevilacqua, M., Petroni, F. and Liang, P., 2024. Lost in the middle: How language models use long contexts. Transactions of the Association for Computational Linguistics, 12, pp.157–173. doi: 10.1162/tacl_a_00638.

Wang, X. et al., 2024. Searching for best practices in retrieval-augmented generation. In: Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing, pp.17716–17736. doi: 10.18653/v1/2024.emnlp-main.981.

Xiang, Z. and Gretzel, U., 2010. Role of social media in online travel information search. Tourism Management, 31(2), pp.179–188. doi: 10.1016/j.tourman.2009.02.016.

Zhan, Q., Liang, Z., Ying, Z. and Kang, D., 2024. InjecAgent: Benchmarking indirect prompt injections in tool-integrated large language model agents. In: Findings of the Association for Computational Linguistics: ACL 2024, pp.10471–10506. doi: 10.18653/v1/2024.findings-acl.624.