AGENT-ORIENTED ARCHITECTURE FOR CITY INFRASTRUCTURE ANALYSIS

Kosovan V.
Assistant lecturer in the Department of Mathematical Modelling, Yuriy Fedkovych Chernivtsi National University Chernivtsi, Ukraine https://orcid.org/0009-0001-8628-3130
Melnyk H.
Lecturer in the Department of Applied Mathematics and Informational Technologies, Yuriy Fedkovych Chernivtsi National University Chernivtsi, Ukraine https://orcid.org/0000-0001-5386-5903
Melnyk V.
Assistant lecturer in the Department of Mathematical Modelling, Yuriy Fedkovych Chernivtsi National University Chernivtsi, Ukraine https://orcid.org/0009-0005-2880-6449
Melnyk O.
Student, Yuriy Fedkovych Chernivtsi National University Chernivtsi, Ukraine
https://doi.org/10.5281/zenodo.21105842

Abstract

This paper presents the design and implementation of GuideLoc, an agent-oriented advisory service for city infrastructure analysis. The system enables users to discover urban locations, including restaurants, pharmacies, parks, and other points of interest, through a conversational chat interface. The proposed architecture employs a multi-agent model consisting of an orchestrator agent responsible for intent recognition and delegation, and a specialized location agent responsible for querying and ranking urban objects. The server-side is built on Python and FastAPI, while the mobile client is developed using React Native and TypeScript. Persistent storage is managed via SQLite with SQLAlchemy, and intelligent response generation is powered by the OpenAI API. Real-time response streaming is implemented through Server-Sent Events, and user authentication relies on JSON Web Tokens. The system accounts for contextual factors such as weather conditions, user budget, route planning, and personal preferences when forming recommendations. Experimental evaluation confirms the feasibility of the approach and demonstrates the extensibility of the architecture for incorporating additional domain-specific agents.

Keywords: agent-oriented architecture, city infrastructure, advisory system, multi-agent systems, natural language processing

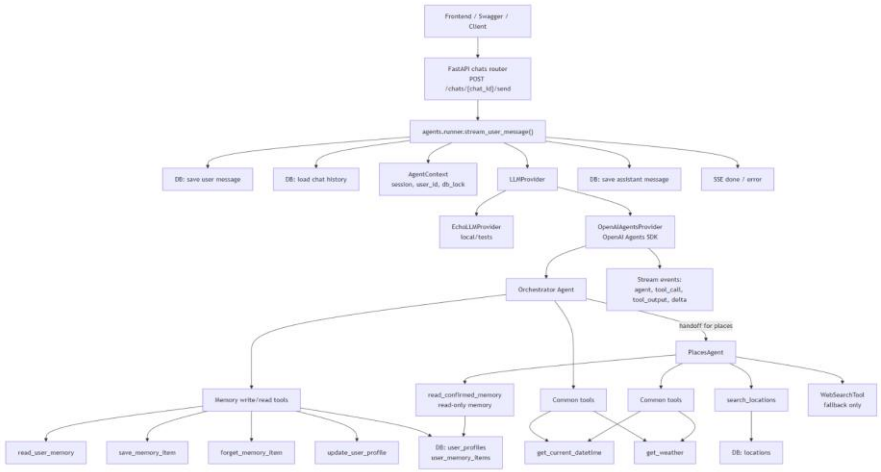
The rapid growth of urban environments and the increasing complexity of city infrastructure have created a significant demand for intelligent systems capable of assisting residents, tourists, and newcomers in navigating urban spaces. Traditional approaches to information retrieval — relying on maps, review platforms, and search engines in isolation — require considerable effort and do not provide cohesive, context-aware recommendations. A user wishing to find a suitable café within a given budget and reach it on foot in rainy weather would typically need to consult several separate services.

Agent-oriented computing offers a paradigm well suited to this challenge. By decomposing the advisory task into autonomous, cooperating agents, each responsible for a well-defined domain, the system can handle complex, multi-dimensional user requests in a unified conversational interface. This approach has been widely studied in the context of service composition, decision support, and intelligent assistants, and its application to city infrastructure analysis represents a natural and practically relevant extension.

Urban advisory systems have been approached from several directions in prior research. Recommendation systems based on collaborative filtering and content-based methods have demonstrated effectiveness in contexts such as restaurant selection and tourism planning. However, these approaches typically lack the capacity to handle compound queries involving multiple constraints simultaneously. The integration of large language models into advisory workflows has opened new possibilities, particularly in enabling natural language understanding as a first-class feature of the system rather than a preprocessing step.

Multi-agent systems have been applied to smart city contexts in domains such as traffic management, energy distribution, and emergency response coordination. Their application to citizen-facing advisory services, however, has received comparatively less attention. The design of an orchestrator-agent pattern — in which a central agent interprets intent and delegates subtasks to domain specialists — has precedents in enterprise workflow automation and intelligent virtual assistant research. GuideLoc draws on this pattern and adapts it to the requirements of a mobile, conversational urban advisory service.

The use of large language model APIs as a backend for agent reasoning has become increasingly common following advances in instruction-following models. The OpenAI API, used in this system, enables the orchestrator agent to perform intent classification, slot filling, and response generation in a unified manner, reducing the architectural complexity that would otherwise arise from maintaining separate natural language processing components.



The GuideLoc system is organized into three primary layers: a mobile client, a server-side API layer, and an agent subsystem. The mobile client is implemented in React Native with TypeScript, providing a cross-platform interface for both Android and iOS. The client renders a conversational chat interface, manages user authentication state, and maintains a local history of past sessions accessible through a side menu. Real-time response display is achieved through a streaming protocol that renders agent responses incrementally as they are generated, rather than presenting a complete reply only after processing concludes.

The server-side is built on Python using the FastAPI framework, which supports asynchronous request handling and integrates naturally with the Pydantic library for data validation and schema definition. Each user message received by the server is authenticated via JSON Web Token verification before being routed to the agent subsystem. The server exposes an internal API for the mobile client as well as integration points for the OpenAI API, the Weather API, and the Google Maps API.

Data persistence is managed through SQLite, accessed via the SQLAlchemy object-relational mapper. The database stores user accounts, session histories, individual messages, user preference profiles, and the local catalogue of urban objects. Each urban object record includes the name, category, address, geographic coordinates, operating hours, and a textual description used by the location agent during query matching. Schema evolution is handled through Alembic migration scripts, ensuring that the database structure can be updated incrementally as the system is extended.

The agent subsystem consists of two agents in the current implementation. The orchestrator agent receives the user's message along with the conversation history and is responsible for determining the intent of the request, deciding whether external information such as weather data or route information is required, and delegating the retrieval task to the location agent. The orchestrator communicates with the OpenAI API and is guided by a structured prompting system that defines its role, the available tools, and the expected format of its outputs.

The location agent is responsible for querying the local database of urban objects, applying filters based on category, budget, operating hours, and geographic proximity, and returning a ranked list of candidate locations to the orchestrator. The orchestrator then incorporates the retrieved candidates, any contextual information gathered from external APIs, and the user's stated preferences into a final response that is streamed back to the client.

The prompting architecture of the system is a central design element. Each agent is initialized with a system prompt that describes its responsibilities, the format of inputs it should expect, and the structure of the outputs it should produce. This design allows the behavior of individual agents to be modified or refined without altering the surrounding code, and it provides a natural extension point for adding new agents: a new system prompt paired with a corresponding handler is sufficient to introduce a new domain-specific capability into the orchestrator's delegation logic.

The system was evaluated through functional testing of each major component and end-to-end testing of representative user query scenarios. Test cases included queries for food establishments within a specified budget, pharmacy searches with operating hour constraints, requests for outdoor walking locations accompanied by weather-conditioned clothing advice, and multi-step queries combining location discovery with route generation.

In all tested scenarios, the orchestrator agent correctly identified the primary intent and activated the appropriate combination of downstream services. The location agent successfully filtered the local database and returned ranked results consistent with the stated constraints. Weather-conditioned recommendations correctly reflected current conditions retrieved from the external API. Route generation requests produced valid links to the Google Maps navigation interface.

Response latency was observed to vary primarily with the complexity of the OpenAI API call required to generate the final response text. Simple location queries with no external API dependencies completed within two to three seconds, while compound queries involving weather retrieval and multi-step reasoning extended

to five to eight seconds. The streaming delivery mechanism mitigated the subjective impact of this latency by presenting partial responses to the user as they became available.

The architecture demonstrated a clear separation of concerns between the orchestrator, the location agent, and the external service integrations. This separation was confirmed to support extensibility: a prototype weather-advisory agent focused solely on clothing recommendations was added during development without requiring changes to the orchestrator's core routing logic. The result supports the claim that the agent-oriented design is well suited to incremental expansion of the system's capabilities.

References:

1. Eisenman, M., Felber, P. (2010). React Native: Building Mobile Apps with JavaScript. O'Reilly Media.
2. Expo Documentation. https://docs.expo.dev/
3. Bierman, G., Abadi, M., Torgersen, M. (2014). Understanding TypeScript. In: European Conference on Object-Oriented Programming. Springer, Berlin, pp. 257–281.
4. Van Rossum, G., Drake, F. (2009). Python 3 Reference Manual. CreateSpace.
5. Ramírez, S. (2021). FastAPI Documentation. https://fastapi.tiangolo.com/
6. Hipp, R. D. (2020). SQLite: A Self-Contained, Serverless, Zero-Configuration, Transactional SQL Database Engine. https://sqlite.org/
7. Bayer, M. (2012). SQLAlchemy. In: Brown A., Wilson G. (eds) The Architecture of Open Source Applications. Vol. 2.
8. Alembic Documentation. https://alembic.sqlalchemy.org/
9. Pydantic Documentation. https://docs.pydantic.dev/
10. OpenAI Platform Documentation. https://platform.openai.com/docs/
11. Jones, M., Bradley, J., Sakimura, N. (2015). JSON Web Token (JWT). RFC 7519, IETF.
12. Fette, I., Melnikov, A. (2011). The WebSocket Protocol. RFC 6455, IETF.
13. Hailuo AI Platform. https://hailuoai.video/
14. GitHub Documentation. https://docs.github.com/