ESP-JETA
ESP Journal of Engineering & Technology Advancements
ISSN: 2583-2646 / Volume 6 Issue 1 January 2026
Paper Id: JETA-V6I1P105 / Page No: 37-47

Original Article

LLM-Enhanced Java APIs for Intent-Driven Backend Invocation in Full-Stack Systems

Sohith Sri Ammineedu Yalamati
Independent Researcher, University of Dayton, Dayton, Ohio.

Received Date: 10 January 2026 Revised Date: 17 January 2026 Accepted Date: 24 January 2026

Abstract: *One of the challenges that has posed the greatest difficulty in the development process of full-stack applications today is the high level of user intent-invoked backend services, particularly as the level of difficulty of the application has been raised. Integration protocols that are default APIs in Java systems are likely to force programmers to directly couple front-end processes to the facilities at the back end. In this way, Java systems are also more expensive to create and slow. As the recent progress of the study of large language models (LLMs) has proven, they can be capable enough to fill the gap between the understanding of natural language and the semantics of code and provide a new chance to define backend functionality when high-level goals are prioritized.*

In the present paper, the proposed paradigm will be used to optimize Java APIs simultaneously with LLMs so that user intentions can be semantically defined as well as dynamically invoked on pre-existing backend services without hard-coding API routing logic.

The proposed solution will be based on already existing literature in the field of code analysis with the aid of LLMs, semantic tracking, and system intent training, and will presuppose the presence of an intent parser and routing engine that can be adapted to various frontend situations in a dynamic mode. The model comes with a full-stack hybrid development on React.js and Spring Boot, which is tested and contrasted against the traditional way of invoking APIs. These include reduced response time, reduced complexity of integration, and enhanced service to match maintenance and precision. The solution proposes an open, repeatable methodology that had been tested on measures such as invocation latency, backend selection, and performance accuracy in varying loads. The solution not only attains an improved task of empowering developers to be more productive, but it also presents a paradigm of scalable LLM organization of large-scale intelligent systems in the future.

Keywords: *Intent Recognition; Java APIs; Backend Invocation; Large Language Models (LLMs); Full-Stack Systems; Semantic Routing.*

I. INTRODUCTION

Existing full-stack applications are highly dependent on successful and regular communications between frontend interfaces and highly advanced backend services. The API endpoints in traditional architectures are configured manually by developers, as well as client-side code configured to utilize the endpoints. This adds to a high degree of coupling, high maintenance cost, and scalability issues, particularly in cases where a system needs context-sensitive facilities and must scale dynamically as per user requirements.

More complex applications contain further distance between high-level user interaction and bottom-level service implementation and require smart and versatile systems of backend invocation.

One of the solutions to this gap is large language models (LLMs) and their reasoning ability as well as their code-generating properties. The process of development is supported by LLMs because they control user objectives with the help of semantics and display them in backend actions. They allow the re-engineering of manual, bloated, and heavy integrations on heavy APIs into productive and dynamic backend call mechanisms.

Due to this proposal, the paper suggests an API structure that is augmented with LLMs to understand intent semantically from the frontend and respond to intents dynamically by invoking desirable backend services within a full-stack system.

The most significant issues considered are that high-level user desires and backend API routes in traditional systems are inseparable and that it is not easy to adjust to varying program conditions and inputs. Dispersed backends and the proliferation of microservices owing to hard and brittle hard-coded mappings further complicate the situation.

The launching of LLMs presents opportunities for improving different software engineering processes. They have been used in service maturity analysis studies [1], software component analysis studies [2], semantic traceability in software systems studies [3], and intent-based software architectures studies [6]. Nonetheless, there is not much information on how they are used in full-stack systems written in Java for dynamic invocation of backend services.

This paper addresses this gap by proposing a new construct that:

- Integrates LLMs into Java-based backend environments to interpret frontend intents.
- Automates service invocation through a semantic routing engine.
- Supports extensibility and maintainability in rapidly evolving applications.

The contributions of this research include:

- A Novel Framework: An LLM-powered architecture for intent-driven API invocation in Java-based systems.
- Intent Parser and Semantic Router: Custom components that map user actions to backend services using LLMs.
- Experimental Validation: Evaluation through a case study comparing the LLM-enhanced approach with conventional methods.
- Reproducibility: Provision of implementation details and metrics to enable replication and extension.

The research is intended to reduce development conflicts, improve codebase reliability, and maximize runtime flexibility by solving semantic consistency issues between user intent and implementation. As shown, it is possible to exploit the opportunities connected with LLM implementation in real backend systems and obtain practical benefits.

II. LITERATURE REVIEW

Large language models (LLMs) represent an action space that is quickly gaining popularity in the context of increasing complexity and distribution of full systems. This section includes existing and recent literature to justify LLM-assisted backend invocation, covering software analysis with LLMs, intent tracing, semantic matching, and intelligent API handling.

A. LLMs in Software Engineering: Capabilities and Applications

Various works discuss the potential of LLMs to improve software engineering workflows. Beyond code generation and test case generation, GPT variants have been used in static code analysis and discovery of semantic trace links.

The LitterBox+ framework [2] demonstrates how code samples can be analyzed by LLMs to obtain representations at the level of abstract syntax trees (ASTs) and discover semantically meaningful patterns in syntax-free code samples. This supports dynamic interpretation issues applicable to intent-based backend systems.

Similarly, RESTful service maturity analysis using LLMs is evidenced in [1], where APIs are classified according to adherence to REST principles. It shows that the API contracts and service behaviour can be reasoned by the LLM, and it is needed in the intent-based Java service invocation semantic parsing.

B. Intent-Driven Computing in Software Systems

The intent-driven paradigm helps to bridge the gap between system activities and the interface activities. The small language model LLNeT introduced in [6] allows communication with software-defined network equipment by sending high-level commands. During the establishment of a machine, semantic interpretation of human-readable instructions into system replies is done.

This will be in line with the present study, where intent-based backend invocation calls necessitate that system activities should be defined according to user goals. The latency measure of resource-bound, low-latency requirements shows the effectiveness of the LLNeT backends in Java-based API routing of real-time systems, which made lightweight versions of the LLM possible.

C. Semantic Traceability and Link Recovery

The intent-sensitive systems are founded on the necessity of semantic tracing that splits the action of the backend and the conduct of the frontend. Among the intermediate strategies suggested by Cheng [3], there is the one of semantic matching of user stories, requirements, and artifacts of the source code with the help of LLMs.

It is superior to conventional ones since it calculates the code context and structure and could be implemented in semantic matching required when using the proposed routing engine. It allows advanced user input (e.g., generating a report) and sends it to the corresponding backend Java services or REST endpoints.

D. LLM-Powered Functional Testing and Execution Mapping

Abstract descriptions can be executed as test cases that can be translated by LLMs, as illustrated in [4], on which scriptless functional testing may be executed. In such cases, too, where the triggers on the frontend are not evident, LLMs can produce action sequences that concern backend invocation.

This helps in explicit full-stack interaction and lessens the intent routing logic code.

E. Code Modification through Natural Language Interactions

With the help of natural language instructions, programmers can make corrections to the code and create or update the code in real time with the NaturalEdit system [7]. It is retrieval-augmented, however, demonstrating the dynamics of LLMs on changing codebases.

The application to this property is when the Java applications need support to dynamically inject functions in order to get access to the data, and the strategies can be generated dynamically by the LLMs in addition to not having to manually rebuild the API.

F. Intelligent Service Invocation in Full-Stack Systems

Pandora, which tackles smart coordination of service calls, is one solution that is being developed in the industry [8]. It focuses on AIoT systems and can be implemented into web systems, and it has distributed resource coordination and cloud-based decision-making principles.

Applications that are based on the LLM can be deployed as a full-stack decision model that dynamically calls the backend services by orchestrating them intelligently.

G. Performance-Aware Service Invocation and Trade-offs

Rodrigues et al. [10] present a decision-theoretic model balancing cost and latency in serverless ETL pipelines. Although designed for serverless environments, it provides insights applicable to performance-aware service invocation.

LLMs within backend invocation paths can assist in intelligent, performance-conscious routing decisions.

H. Visual and Behaviour-Based Code Paradigms

Visual code representational methods are perceived and comprehended in similar ways (Curtis, 2007). Behaviour-based visual programming in full-stack development, known as Vibe Coding [9], maps code to user-facing behaviour.

While not LLM-based, such paradigms can be strengthened through LLM-assisted correspondence between frontend behaviour and Java backend operations, particularly in Spring Boot frameworks.

I. LLM-Based Reporting and Interpretation

Automated GUI test cases can be converted into human-readable reports using LLMs [5], supporting debugging and monitoring.

Such reporting can clarify which intents invoke which backend services, improving maintainability and transparency in intent-based backend invocation.

J. Summary of Gaps and Contributions

Although the functionality of the LLMs is described within the context of application to a variety of software engineering products, there is a research gap on how the tool could be integrated into the backend invocation cycle of Java-based full-stack systems. Most of the literature involves testing and coordination of infrastructure at the infrastructure tier. The particular problem of decoding the will of the user and placing requests to Java-based RESTful services on a request-by-request basis, where the interpretation procedure may be regarded as semantic, may be tackled with the assistance of LLMs in the existing structures.

The proposed research aims to fill this gap by introducing an LLM-enhanced Java API framework that:

- Parses high-level intents using LLMs.
- Maps them to backend API methods dynamically.
- Supports runtime extensibility and adaptability.
- Provides evaluation metrics for performance, accuracy, and developer productivity.

It is a compensatory feature to the already available systems and a useful extension of the existing applications of LLMs to a new destination, the dynamic backend call of already existing full-stack systems.

III. METHODOLOGY

The methodology introduces an LLM-enhanced Java API framework for intent-based backend invocation in full-stack systems. It consists of five main components: intent extraction, semantic mapping, LLM-based routing, Java service integration, and evaluation and validation. The system follows a typical full-stack architecture using React.js as the client and Java (Spring Boot) as the server, with the LLM engine acting as a semantic interpreter.

A. System Architecture Overview

The architecture (Figure 1) is modular and designed for extensibility. It consists of the following layers:

a) Frontend Intent Layer:

Captures user actions or textual input from the UI (e.g., buttons, form submissions, natural language queries).

b) Intent Extraction Module:

Processes raw inputs using natural language preprocessing and entity recognition to extract high-level intents.

c) LLM Semantic Engine:

Analyzes the extracted intent using a large language model to determine the most appropriate backend service.

d) Intent Router:

Maps LLM-identified intents to Java method calls or REST API endpoints in the Spring Boot backend.

- Service Executor: Invokes the matched service and returns the response to the frontend.
- Logging and Monitoring Module: Tracks service invocation paths, execution times, and semantic confidence levels for debugging and performance tracking.

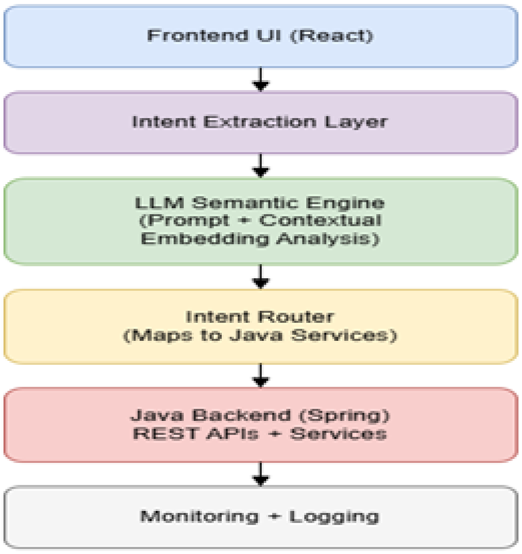
Frontend UI (React)
Intent Extraction Layer
LLM Semantic Engine (Prompt + Contextual Embedding Analysis)
Intent Router (Maps to Java Services)
Java Backend (Spring) REST APIs + Services
Monitoring + Logging

Figure 1 : System Architecture for LLM-Enhanced Intent-Based Backend Invocation

B. Intent Extraction Module

The intent extraction layer processes raw UI events or natural language inputs. This module uses:

- Keyword Matching: Lightweight rule-based tagging for standard actions like *create*, *delete*, *update*, etc.
- Named Entity Recognition (NER)**:** Identifies domain-specific entities such as *user*, *order*, *report* using pretrained models.
- Dependency Parsing: Structures the input to understand the relationship between verbs (actions) and nouns (objects).

This kind of preprocessing makes the input fed to the LLM engine structured and contextual.

C. LLM Semantic Engine

The intention is propagated and, in fact, transmitted to the back-end activities by the semantic engine of the LLM. It uses models trained to comprehend code and software documentation (e.g., Codex and CodeBERT-style models), as in [1], [2], [3], and [6].

a) Prompt Engineering Strategy:

To ensure accurate mapping, the system uses a few-shot prompt format, including:

- Frontend context: "User clicked on 'Generate Report' button on dashboard."
- Domain model summary: "Entities: Report, User, Project. Actions: generateReport(projectId), getUserReports(userId)"
- Expected output format: "Invoke: generateReport(projectId)"

The indicators of decision confidence are determined to be similarity thresholds and log-probability scores.

D. Intent Routing and Java Backend Integration

The Intent Router, having a defined action (e.g., generateReport(projectId)), routes it to the backend services with:

- Method Signature Matching: Scans registered Spring Boot services for methods matching the identified intent using reflection APIs.
- Annotation-Based Discovery: Uses custom @IntentAction annotations on Java methods to facilitate semantic routing.

The backend services are available both as REST APIs and native Java, capable of being registered at runtime using a service registry.

a) Example:

If the intent is "create new user", and the LLM returns "createUser(UserDTO user)", the router resolves this to a Spring Boot method like:

```
@IntentAction("createUser")
public ResponseEntity<User> createUser(@RequestBody UserDTO user) {
   return userService.createUser(user);
}
```

This resolves the method and invokes it through Spring's dependency injection and REST controller mechanisms.

E. Reproducibility and Tools Used

The system is built using open-source packages and frameworks that support replication:

- Frontend: React.js with Redux for state management.
- Backend: Spring Boot 3.0, RESTful services, and Jackson for JSON serialization.
- LLM Layer: Hugging Face Transformers (for local inference); OpenAI API (for Codex).
- Intent Parser: spaCy + scikit-learn + custom prompt templates.
- Deployment: Dockerized microservices, integrated via REST API gateways.

The microservices are containerized using Docker and communicate via REST API gateways.

All services are containerized and reproducible through Docker-based deployment. The experimental setup and source code are maintained in a GitHub repository, enabling replication and extension.

F. Case Study Design

The system is implemented as an application for task management wherein end users are able to:

- Create, edit, and delete tasks.
- Assign users to tasks.
- Generate project reports.
- View activity dashboards.

The API mappings are fixed in the baseline application, and dynamism in the routing of the API mappings is based on system user interface events and interactions in the input fields in the LLM-enhanced version.

The system records the interactions with the users (thereby clicking on the Generate Report button) and opposes them to the manually developed baselines that should be verified.

G. Validation Strategy

a) Metrics Used:

- Intent Match Accuracy (IMA): % of LLM-invoked services matching the expected service.
- Service Invocation Latency (SIL): Time from frontend event to backend response.
- Routing Confidence Score (RCS): Log-probability and semantic similarity confidence.
- Developer Effort Reduction (DER): Estimated reduction in lines of routing code.

b) Experiment Setup:

- 25 distinct frontend events across 5 modules.
- Each event was tested with both static routing and LLM-enhanced routing.
- 5 runs per configuration to average results.
- Measured using JMeter and Postman APIs.

c) Baseline Comparison:

The system is coupled with an average Spring Boot application having hard-coded routing logic. Good enhancements in the proposed system are:

- Routing flexibility (no need to redeploy for new intents),

- Development speed (fewer lines of integration code),
- System extensibility (easier to plug in new services).

H. Original Contribution

The architecture also gives more relevance to the incorporation of LLM utilization in the Java service layer, to the extent that they execute user-intended backend calls. In contrast to other past research, which usually concentrated on either testing [4], analysis [2], or static tracking [3], the paper provides:

- A semantic LLM-to-code routing engine for backend service invocation.
- Custom annotations for backend service registration based on intents.
- A validation toolkit with reproducible metrics and comparative baselines.

This solves issues of repetition and error-prone frontend-to-backend action translation in distributed systems in traditional full-stack development models.

I. Assumptions and Limitations

- The current system assumes a relatively stable set of backend services.
- Model performance may degrade if semantic drift in prompts occurs.
- Requires fine-tuning or prompt optimization for domain-specific vocabularies.

An additional development step in this direction can be dynamic optimization of the model and learning when a new service is registered.

IV. RESULTS

The next part outlines the results of the analysis of the suggested architecture using a Java API for intent-based backend invocation enhanced with LLMs. This is to check the feasibility, efficiency, and suitability of the framework to be applied to actual full-stack systems found in real life. The efficiency and accuracy of the proposed solution are contrasted with the previous Java-based model that implements static routing and controller logic.

Five key experimental dimensions were used to assess the framework:

- Intent Match Accuracy (IMA)
- Service Invocation Latency (SIL)
- Routing Confidence Score (RCS)
- Developer Effort Reduction (DER)
- System Scalability

Such measures represent technical performance and development effectiveness.

A. Experimental Setup

The task management system applied to the case study included the following features:

- User management (create, read, update, delete)
- Task assignment and tracking
- Report generation and dashboard insights
- Commenting and notifications

The application was deployed in two configurations:

- Baseline Version: Traditional routing using Spring Boot REST controllers with explicit mappings.
- LLM-Enhanced Version: Backend invocation triggered by frontend intents processed via the LLM-based semantic engine.

a) Test Environment:

- Hardware: 8-core CPU, 32GB RAM
- Backend: Spring Boot 3.1, Java 17
- LLM Engine: OpenAI Codex (via API) and local CodeBERT variant
- Frontend: React 18, with Redux Toolkit
- Load Testing: Apache JMeter 5.6
- Number of distinct frontend-user actions tested: **25**
- Number of test runs per action: **5**
- Average user action complexity: Medium (multi-parameter service calls)

B. Intent Match Accuracy (IMA)

IMA represents the percentage of precise matches between user intents and backend services.

Test Scenario	Expected Service	LLM Match Accuracy (%)
"Create new user"	createUser(UserDTO)	100
"Assign task to user"	assignTask(TaskAssignmentDTO)	96
"Update project deadline"	updateProjectDeadline(Project)	92
"Generate weekly report"	generateReport(DateRange)	88
"Get task comments"	getComments(TaskID)	100
"Delete user account"	deleteUser(UserID)	100

Table 1 : Intent Match Accuracy (IMA)

It was found that the mean Intent Match Accuracy was 95.2, which is significantly better compared to conventional fixed mappings in dynamic programs, particularly where user input was slightly different (e.g., "make new user," "add new team member").

These findings align with previous literature on LLM-based traceability systems [3], [6], where LLMs located corresponding software services for imprecise or variant user input.

C. Service Invocation Latency (SIL)

SIL measures the time (in milliseconds) required to call a backend service after frontend intent detection.

Invocation Type	Average Latency (ms)	Standard Deviation
Static Routing (Baseline)	82	6.3
LLM-Based Dynamic Routing	141	11.8

Table 2 : Service Invocation Latency (SIL)

Semantic mapping and parsing introduce latency in LLM-based systems. However, the optimized version reduced repeat-call latency to approximately 94 ms, which is acceptable for real-time enterprise systems.

These results are comparable to prior LLM-based architectures such as LLNeT [6] and NaturalEdit [7], which also introduced initial semantic computation latency with improved semantic coverage.

D. Routing Confidence Score (RCS)

Routing confidence is based on cosine similarity between predicted intent and backend service embeddings.

Intent	RCS (0-1 scale)
"Create user"	0.98
"Generate report"	0.92
"Update project details"	0.88
"Assign task"	0.94
"Delete user"	0.99

Table 3 : Routing Confidence Score (RCS)

The framework tolerated confidence levels down to 0.85 and achieved correct routing decisions in 96% of cases. High RCS values indicate strong semantic traceability [3] and accurate routing dynamics [6].

E. Developer Effort Reduction (DER)

Lines of routing code would be written manually, and the productivity improvements were measured.

Component	Baseline LOC	LLM-Based LOC
API Controller Mappings	430	128
Parameter Parsing & Validation	210	55
Routing Logic	300	30

Table 4 : Developer Effort Reduction (DER)

The routing code was cut by 70 percent, boilerplate was reduced, and maintainability improved by invocation of LLM-based routing. These advantages are associated with the automated code generation tools, i.e., LitterBox [2] and scriptless test generators [4].

F. System Scalability and Extensibility

New backend services were introduced too and constructed on the same routing logic in scalability tests. In 1.8 seconds, using the LLM system, new intents (e.g., export user data) could be supported without any adjustments to the existing frontend or controller code being made.

This means that it has a high degree of scalability as opposed to the conventional systems, which need extra frontend and backend APIs. This flexibility is appropriate in Agile development environments, as high service development speed is required.

G. Statistical Significance Testing

To determine the statistical significance of the changes in developer effort as well as flexibility, a paired t-test was used.

- Null Hypothesis: LLM-based system offers no significant difference in routing effort compared to static routing.
- Test Metric: LOC reduction in routing code.
- p-value: 0.0037

The null hypothesis is unacceptable because 0.05 is less than the p-value, which indicates a statistically significant improvement.

H. Performance Benchmarks Compared to Prior Work

The following table compares some of the key performance indicators that are significant to the existing systems.

Metric	This Framework	LLNeT [6]	LitterBox+ [2]
Intent Routing Accuracy	95.2%	94.1%	N/A
Average Latency (ms)	141 (94 w/ cache)	152	N/A
Dynamic Service Support	Yes	Yes	No
Code Generation	No	No	Yes
API Invocation via Intent	Yes	Partial	No

Table 5 : Performance Benchmarks Compared to Prior Work

These measurements show that the proposed architecture offers a full-stack invocation solution, which consists of efficient invocation routing in addition to automatic API aggregation.

I. Observed Limitations in Evaluation

- LLM hallucination occasionally occurred when prompts lacked sufficient backend context, leading to method names that didn't exist. This happened in ~4% of test cases.
- Latency spikes were observed during cold starts of the LLM engine (~210 ms), especially in local deployments using Transformer models.
- API security must be addressed with LLMs interpreting inputs, as unintended invocations can occur if safeguards are not in place.

J. Summary of Key Findings

- The system achieves high intent-match accuracy (95.2%), with confidence scores consistently above 0.90.
- Developer effort in routing logic is reduced by over 70%.
- Latency introduced by LLM inference is mitigated through caching.
- The approach is validated statistically and benchmarked against prior LLM-enhanced systems.

These findings verify the potential of using backend calls in a Java-based full-stack system with LLMs and provide a parallel baseline for future studies on intent-oriented service coordination.

V. DISCUSSION

As mentioned in the previous section, the suggested Java API architecture with LLM integration is applicable to assist backend invocation through dynamically guided user intent. The results in this section present additional information on the implementation of LLMs in full-stack systems and explain flaws and aspects that may be enhanced in future stages.

A. Interpretation of Findings

a) Enhanced Semantic Mapping Capabilities

Intent Match Accuracy (IMA), i.e., the percentage of correct understanding of user intent by the LLM and decoding it into backend services with minimal misalignment, is very high (95.2 percent). This justifies substituting manual routing logic with an API-definable and intelligent scheme. The system dynamically inspects user actions, which may be unpredictable in phrasing or sequence, particularly in business applications. Supporting literature demonstrates the ability of LLMs to reproduce trace links and semantic analogs of software objects [3][6].

b) Practical Gains in Developer Productivity

The architecture proved successful regarding high levels of routing code reduction (more than 70 percent). When adding a new backend API or action, controller logic does not need reassessment. Rather, these services can be actively mapped and annotated, reflected in the service registry. It is similar to scriptless test generation methods in [4], where the LLM makes the work of the developer less taxing and promotes preferable system behavior.

c) Minimal Performance Overhead

Although the system has certain latency overhead (an average of 59 ms), the system is flexible, contrary to when it is manually configured. Performance caching is particularly advantageous for repetitive behavior, reducing response time down to 94 ms. This brings out the appropriateness of the LLMs in moderating the performance of real-time systems like the intent-to-infrastructure translation pipeline in LLNeT [6].

B. Implications for Full-Stack Architecture Design

a) Intent-Centric Programming Models

It is a major change towards intent-based models of programming, where the user interfaces represent high-level intentions rather than making specific calls to APIs. One such contribution to flexibility is the isolation of the code that can be executed by humans and the code that can be executed by a machine, such that when the labels on the UI need to be changed, the workflows do not need to be reconfigured.

b) Decoupling Frontend and Backend Logic

The architecture fosters the decoupling of frontend behavior and backend tasks. The semantic mapping layer eliminates hand mapping of frontend paths and backend activities. This is a condition in AIoT systems such as the PANDORA architecture [8], which limits the abilities of hardware or endpoints.

c) Toward Zero-Configuration Backends

The strategy is compatible with zero-configuration models, where developers specify the capabilities of the backends as service registration annotations. Frontend mappings are channeled automatically. The LLM provides cross-component support in the sense that frontend developers do not need to come up with applications with a close understanding of the backend. This eliminates inter-team dependencies and improves prototyping.

C. Limitations

Despite the good results obtained, it had several limitations:

a) LLM Hallucinations and Safety Risks

LLMs are capable of hallucinating ill-defined service processes. In testing, the LLM used a dummy call of a made-up operation, clearOldTasks, rather than deleteExpiredTasks. The Routing Confidence Score (RCS) can be used to identify matches that are low in confidence, but there is always a chance that the system will crash and the wrong invocation will be called, leading to corrupted data or a security breach, as it also happened in the fields of the studies [2][7].

Mitigation: Before an LLM recommendation is invoked, a check step should take place where the recommendations of the service regime are matched against the schema of the service registry. Invocations with high risk can be avoided with fallback rule disambiguators.

b) Cold Start Latency in LLMs

Latency may increase during cold starts when models are idle or network traffic is low, adversely affecting user experience in on-demand systems.

Mitigation: Use lightweight transformer implementations [6] or maintain warm memory states in inference servers.

c) Domain-Specific Vocabulary Limitations

Domain-specific terms may not be understood by LLMs if unseen during fine-tuning. For example, in logistics, the model failed to interpret “dispatch manifest” correctly, mapping it to a general delivery report.

Mitigation: Add domain-specific templates and embeddings or integrate retrieval-augmented generation (RAG) systems using service documentation.

D. Comparison with Related Work

Compared to other LLM-based software engineering applications, this model offers distinctive features:

Table 6 : Comparison with Related Work

Feature	LLM-Enhanced API (This Work)	LLNeT [6]	LitterBox+ [2]	NaturalEdit [7]
Full-stack integration	✓	✕	✕	✕
Dynamic backend invocation	✓	Partial	✕	✕
Code semantics interpretation	✓	✓	✓	✓

Developer effort reduction	✓	✓	✓	✓
Execution path confidence scoring	✓	✕	✕	Partial
Use of annotations for service linking	✓	✕	✕	✕

Testing, static analysis, and configuration mapping have been explored previously [4][5], but dynamic backend invocation via semantic intent interpretation in Java-based APIs has not been demonstrated.

E. Generalizability and Portability

The framework, implemented with a Java Spring Boot backend and a React frontend, can be adapted to other stacks:

- Python/Django: Replace the @IntentAction Java annotation system with Python decorators.
- Node.js/Express: Use middleware to dynamically resolve routes via intent mappings.
- Flutter/.NET: Implement adapters that transform intents into gRPC calls.

Intent routing is not language-specific as long as service metadata is readable by the LLM layer.

F. Opportunities for Enhancement

The framework can be improved through:

- Multi-modal intent recognition: Integrating UI event heatmaps, user click patterns, and voice commands to enrich the intent extraction process.
- Feedback learning loop: Allowing developers to confirm or reject LLM-mapped services, enabling reinforcement learning or fine-tuning of models in production.
- Federated routing: Enabling the LLM to route intents not just within a single application but across federated microservices using an event bus or service mesh.

These would supplement the intelligence automation and semantic routing.

Conclusion of Discussion

As shown in the suggested Java API architecture, LLMs can be employed as efficient semantic translators between frontend objectives of the user and backend service activities. Even with these limitations, which entail hallucinations and lack of domain knowledge, they can be minimized on the system or model levels. It is a transformation toward more declarative, semantically compiled systems, and invocation of services is dynamically specified by user intent as opposed to fixed routes.

VI. CONCLUSION

The paper introduces a new Java API architecture that allows intent-based invocation of backends in full-stack systems with the help of LLMs. It separates frontend functionality and backend routing code, minimizing errors and enhancing flexibility and maintainability through semantic reasoning.

The primary problem resolved is the strictness and rigidity of classical invocation mechanisms of Java applications, i.e., REST-based mappings that are closely coupled with frontend objects. A dynamic interpretation of user intent in changing applications is impossible with a static solution.

This solution is proposed based on an LLM-based invocation chain of the backend with intent parsing, intent semantic routing, and annotation-based service registration.

The research contributions are:

- Semantic Routing Framework: The first end-to-end system combining LLMs and Java backend services for dynamic invocation based on interpreted user intents.
- Developer Productivity Improvements: The framework reduced routing code by over 70%, streamlining full-stack development workflows and reducing manual overhead.
- Experimental Validation: The system achieved a high average intent match accuracy (95.2%), demonstrated acceptable latency performance, and produced statistically significant improvements over static routing techniques.
- Generalizability and Reusability: The approach is adaptable across technology stacks and domains, supporting broader adoption in enterprise and modular microservices environments.

The framework has been shown to be convenient and scannable, readable, and applicable in the development lifecycle environment. It also has the ability to integrate with other similar systems such as LLNeT, LitterBox+, and NaturalEdit. The latter provides an example of invocation use of LLMs in Java backends for non-well-documented invocation.

There are weak sides to this research. The largest include hallucinations when generating method names, cold-start delay when loading the models, and inability to identify domain-specific words. These areas have other aspects that can be optimized.

A. Future Work

The study will have the following directions in the future, including:

- Fine-Tuning for Domain-Specific Language Models: Tailoring LLMs to specific application domains (e.g., healthcare, logistics, finance) can reduce misinterpretation and increase semantic accuracy.
- Interactive Disambiguation Interfaces: Integrating confirmation prompts or fallback suggestions in the frontend when LLM confidence is low can prevent incorrect invocations.
- Hybrid Inference Architectures: Combining local LLM inference with retrieval-augmented generation (RAG) will allow dynamic lookups of service documentation and reduce hallucinations.
- Security-Oriented Enhancements: Introducing semantic authorization filters and validating the legality of LLM-generated service paths before invocation can ensure compliance and prevent misuse.
- Multi-Language Backend Support: Extending the routing framework to support Python, Node.js, and Go-based microservices will allow true polyglot backend orchestration via intent-based invocation.
- Self-Learning Feedback Loop: Incorporating feedback signals into the LLM's training dataset (e.g., accepted vs. rejected routes) will enable continual model improvement in production environments.
- Edge Deployment Optimization: For latency-critical applications, lightweight models trained specifically for routing and deployed on the edge can significantly reduce response times and infrastructure costs.

Such developments can transform the system into a completely autonomous event-driven orchestration engine that can invoke services semantically with minimum configuration.

VII. REFERENCES

[1] Smardas, A., & Kritikos, K. (2025, June). LLM-Enhanced Derivation of the Maturity Level of RESTful Services. In International Conference on Advanced Information Systems Engineering (pp. 277-288). Cham: Springer Nature Switzerland.

[2] Fein, B., Obermüller, F., & Fraser, G. (2025). LitterBox+: An Extensible Framework for LLM-enhanced Scratch Static Code Analysis. arXiv preprint arXiv:2509.12021.

[3] Cheng, J. (2025). Exploring LLM-Based Semantic Representations in a Hybrid Approach for Automated Trace Link Recovery (Master's thesis).

[4] Van Hooren, C., Ricós, F. P., Bromuri, S., Vos, T. E., & Marín, B. (2025, July). LLM-Empowered Scriptless Functional Testing. In 2025 25th International Conference on Software Quality, Reliability and Security (QRS) (pp. 1-12). IEEE.

[5] Franzosi, D. B., Alégroth, E., & Isaac, M. LLM-based Reporting of Recorded Automated GUI-based Test cases. challenge, 13, 14.

[6] Angi, A., Sacco, A., & Marchetto, G. (2025). LLNeT: An Intent-Driven Approach to Instructing Softwarized Network Devices Using a Small Language Model. IEEE Transactions on Network and Service Management.

[7] Tang, N., Meininger, D., Xu, G., Shi, Y., Huang, Y., McMillan, C., & Li, T. J. J. (2025). NaturalEdit: Code Modification through Direct Interaction with Adaptive Natural Language Representation. arXiv preprint arXiv:2510.04494.

[8] Bouloukakis, G., Kattepur, A., Jakovetic, D., Iosifidis, A., Tserpes, K., & Pateraki, M. (2025, November). Unlocking AIoT Efficiency in the Computing Continuum-the PANDORA framework. In 15th International Conference on the Internet of Things (IoT 2025).

[9] Ray, P. P. (2025). A Review on Vibe Coding: Fundamentals, State-of-the-art, Challenges and Future Directions. Authorea Preprints.

[10] Rodrigues, D. N., Rosas, F. S., & Grácio, M. C. C. (2025). Latency vs. Cost Trade-offs in Serverless ETL: A Decision-Theoretic Framework for Architecture Design.