LLM Assisted Customer Support Ticket Triage using OOP Design

Jayanth Vunnam
Department of Computer Science
University of Colorado Boulder
Boulder, CO, USA
jayanth.vunnam@colorado.edu

Saiteja Poluka
Department of Computer Science
University of Colorado Boulder
Boulder, CO, USA
saiteja.poluka@colorado.edu

Abstract

This project proposes an object-oriented framework for automating support ticket triage by combining multiple routing strategies within a unified architecture. A router applies interchangeable strategies like keyword/regex rules, embedding-based semantic matching, and an LLM classifier that returns structured JSON and merges their scores with confidence-weighted voting. Input flows through a preprocessing chain (language detection, PII masking, normalization). The LLM sits behind an adapter so providers can be swapped without touching domain code. Once a department is chosen, a factory instantiates a department-specific responder agent (e.g., Billing, Tech Support) to draft a first reply. The design emphasizes object-oriented principles encapsulation, interface segregation, and dependency injection—to keep the system modular, testable, and easy to evolve (Truss and Boehm 2024).

Introduction

Enterprise service desks today handle thousands of diverse tickets every day ranging from billing disputes and subscription issues to device configurations and access problems—often written in a mix of informal language, multiple languages, and containing sensitive personal information. When tickets land in the wrong queue, resolution slows to a crawl, SLAs slip, hand-offs multiply, and operational costs skyrocket, with misclassification costing \$100-\$500 per incident. Traditional rule-based routers excel at well-known cases but quickly break down as terminology and phrasing evolve, leading to 15-25% misclassification rates (Zencoder AI 2024). On the other hand, pure LLM-based approaches generalize well but introduce new challenges around latency, expense, vendor lock-in, and overall reliability in production environments (Zhao et al. 2025).

Three core engineering tensions drive our design. First, there's the trade-off between accuracy and latency/cost: richer models yield better classification but at the price of speed and expense, while lightweight rules are fast and cheap yet fragile. Second, evolvability versus coupling forces us to avoid hard-coded logic and provider-specific SDKs that make it difficult to update models or switch vendors without introducing regressions. Third, reliability at scale demands safeguards against tail latency, upstream rate limits, transient failures, and outages—without which even sporadic delays can become full-blown operational incidents.

To address these, we adopt a pattern-driven approach. A Strategy pattern encapsulates three interchangeable routing methods high-precision rules, semantic similarity via embedding-based schema-validated LLM classification—allowing us to choose the right tool for each ticket. A Chain of Responsibility pipeline handles language detection, PII scrubbing, and text normalization before any external calls, improving both security and cache efficiency. We introduce an Adapter to hide the specifics of LLM providers, keeping the core system vendor-agnostic, and layer in a Proxy/Decorator for caching, rate limiting, retries, and circuit breaking to control tail-latency and manage costs. Finally, a lightweight Factory produces department-specific responder stubs (for Billing, Technical Support, etc.) that can generate initial draft replies via simple FAQ retrieval, demonstrating end-to-end functionality without the need for full-blown agent automation (subhromitra 2021).

Background and Related Work

Automated ticket triage has largely been approached as a text classification problem using traditional NLP and machine learning pipelines. Practical systems such as Automatic Ticket Classification Using NLP and Machine Learning (sukhijapiyush 2020), Automated Helpdesk Ticket Classifier (subhromitra 2021), Support-Ticket-Classification-using-NLP, and the Intelligent Support Ticket Routing System by all follow a similar pattern: preprocess ticket text, extract features, and train supervised models to predict categories or priorities. Academic and institutional work, including Automated Ticket Classification for Information Technology Helpdesks using Machine Learning (ResearchGate 2024).

More recent work shifts towards AI-augmented and LLM-based triage. Industry articles such as Analytics

Vidhya's overview of automated ticket triage (Analytics Vidhya 2023). Instruction-tuned large language models have been applied to closely related tasks such as automated bug triaging [Kiashemshaki et al.(2025)], suggesting that LLMs can effectively map free-form technical text to structured labels. However, these works primarily focus on model choice and empirical performance rather than on the underlying software architecture. In contrast, the present project contributes an object-oriented design that treats rule-based classifiers, embedding-based similarity models, and LLM-based classifiers as interchangeable routing strategies behind stable interfaces, aiming to provide a reusable and maintainable architectural template for enterprise ticket triage systems.

Modern customer support teams rely on ticketing systems to capture issues from email, web forms, and in-app channels, but a surprising amount of delay and cost still comes from simply getting tickets to the right team. Many organizations started with manual triage or simple rule-based routing (e.g., keyword and regex filters that send "refund" tickets to Billing and "password reset" tickets to Accounts), but these approaches are brittle, hard to maintain, and struggle with informal language, multilingual users, and evolving product terminology. As ticket volumes grew, teams introduced machine learning and NLP pipelines to classify tickets automatically, yet these often exist as isolated models bolted onto larger systems with ad-hoc glue code. In parallel, the rise of large language models and semantic embeddings has made it possible to perform more flexible, context-aware ticket classification, but integrating these capabilities into production helpdesk platforms raises new challenges around reliability, cost control, privacy, and maintainability. This project is situated in that space: it focuses on building a robust, object-oriented ticket triage service that can combine rules, embeddings, and LLMs under a clean, extensible architecture rather than treating triage as a one-off model.

Problem Statement

Enterprise customer support operations struggle with the high volume and diversity of incoming tickets ranging from billing inquiries and subscription errors to technical configurations and access issues often expressed in informal, multilingual language and containing sensitive information. When tickets are misrouted to the wrong department, customers experience longer wait times, service level agreements are violated, and support teams incur increased operational costs and inefficiencies (Rochester Institute of Technology 2025).

Existing solutions fall short in key ways. Rule-based routing systems offer speed and predictable cost but fail to adapt as product vocabularies and user phrasing evolve, leading to brittle performance and growing misclassification rates. Conversely, purely LLM-driven classifiers generalize more effectively but introduce unacceptable latency, volatility in API costs, vendor lock-in risks, and reliability concerns in production environments. This gap between precision and practicality leaves enterprises without a balanced, maintainable, and cost-effective approach to automated ticket routing.

Objectives

Develop a Modular Classification Framework

Design and implement a Strategy-based architecture that encapsulates multiple routing algorithms—rule-based, semantic similarity (embedding k-NN), and LLM-driven classification—within a common interface. This objective ensures seamless extensibility and the ability to compare algorithm performance under identical conditions (sukhijapiyush 2020).

Implement a Robust Preprocessing Pipeline

Construct a Chain of Responsibility pipeline to perform language detection, PII scrubbing, and text normalization before classification. The goal is to standardize diverse ticket inputs, protect sensitive data, and improve cache effectiveness across routing strategies (Analytics Vidhya 2023).

Ensure Vendor-Agnostic AI Integration

Create an Adapter layer abstracting interactions with different LLM providers (e.g., OpenAI, Anthropic, local models), enabling transparent switching of AI services without modifying core logic. This supports enterprise requirements for compliance, cost negotiation, and risk management (ResearchGate 2024).

Optimize Performance and Reliability

Introduce a Proxy/Decorator layer to manage caching, rate limiting, retry logic, and circuit breaking for all external API calls. This objective focuses on controlling tail latency, reducing costs, and maintaining high availability under production-like load conditions (Pia AI 2024).

Facilitate Comprehensive Evaluation

Establish an evaluation framework with automated A/B testing, ablation studies, and detailed logging of per-ticket decisions, confidence scores, latencies (e.g., $p95$), cache hit rates, and cost per thousand tickets. This ensures data-driven insights into the trade-offs between accuracy, speed, and expense (Juno445 2022).

Demonstrate End-to-End Feasibility

Build a lightweight Factory-based responder component that generates draft replies using FAQ retrieval for different departments (Billing, Technical Support, Sales). This objective showcases the system's ability to deliver actionable outputs within a bounded scope and term (subhromitra 2021).

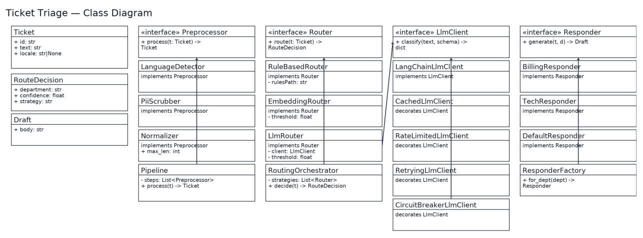

Figure 1: High-level class diagram of the ticket triage system. (Replace with your actual class diagram figure.)

Methodology

System Architecture Overview

The system is constructed as a modular, object-oriented pipeline that cleanly separates concerns and allows independent evolution of each component. Incoming tickets flow through three major stages—preprocessing, routing, and response generation—each implemented as interchangeable modules behind well-defined interfaces. An orchestration layer coordinates these stages, invoking the preprocessing chain, then passing normalized ticket data to the routing subsystem, and finally delegating to the factory-generated responder stubs (Incorporating Large Language Models into Production Systems, 2024).

System Class Diagram The class diagram illustrates the core abstractions and demonstrates design patterns that ensure modularity and maintainability. The `Ticket`, `RouteDecision`, and `Draft` classes form the domain model, while the `Preprocessor` and `Router` interfaces establish contracts for pluggable components. Concrete implementations demonstrate the Strategy pattern (`RuleBasedRouter`, `EmbeddingRouter`) and Chain of Responsibility pattern (`LanguageDetector`, `PiiScrubber`, `Normalizer`), following Gang of Four design principles (Gamma et al., 1994).

System Sequence Diagram The sequence diagram traces a ticket's journey through the system. A client initiates a `POST /triage` request, triggering the pipeline's preprocessing chain, which normalizes and validates the ticket. The routing orchestrator then invokes strategies in sequence (RuleBasedRouter → EmbeddingRouter → LLMRouter), with each strategy potentially returning a `RouteDecision`. Once a confident routing decision is made, the `ResponderFactory` instantiates the appropriate department responder, which generates a draft reply. Finally, the API returns a JSON response containing the department, confidence score, routing strategy used, and draft response body.

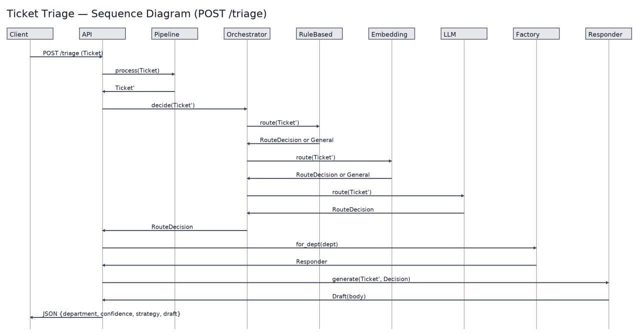

Figure 2: Sequence diagram showing the triage request flow from API call to responder draft. (Replace with your actual sequence diagram.)

Listing 1: Preprocessing pipeline in Python.

```python
from typing import List
from triage.core.models import Ticket
from .base import Preprocessor

class Pipeline:
    def __init__(self, stages: List[
        Preprocessor]):
        self.stages = stages

    async def process(self, t: Ticket)
        -> Ticket:
        for s in self.stages:
            t = await s.process(t)
        return t
```

Component Descriptions

At a high level, the system decomposes into four main responsibilities:

- Representation of domain entities such as tickets, routing decisions, and draft responses.
- Preprocessing of incoming text (language detection, PII scrubbing, normalization).
- Routing: selecting a target department and confidence based on one or more strategies.
- Response generation: producing an initial reply template tailored to the assigned department.

Preprocessing Chain The preprocessing chain implements a Chain of Responsibility pattern where each handler performs a single preprocessing task: language detection, PII scrubbing, and text normalization (tokenization, stemming, stop-word removal). This approach follows established NLP best practices for preparing text data for classification and matches pipelines reported in prior IT service desk ticket classifiers (KushalPatil 2022).

As shown in Listing 1 and 2, 3 and 4, the preprocessing pipeline applies each stage in order to the incoming ticket.

Routing Strategies Routing strategies are encapsulated under a `Router` interface using the Strategy pattern:

Listing 2: Language detector preprocessor.

```python
# triage/preprocess/language.py
from triage.core.models import Ticket
from .base import Preprocessor


class LanguageDetector(Preprocessor):
    async def process(self, t: Ticket)
        -> Ticket:
        # Stubbed language detection (
            could integrate fasttext/
            langdetect)
        locale = t.locale or "en"
        return Ticket(id=t.id, text=t.
            text, locale=locale, meta=t.
            meta)
```

- **RuleRouter:** keyword and regex rules for known intents, similar to traditional rule-based baselines in ITSM systems.
- **EmbeddingRouter:** sentence embeddings for semantic matching against labeled exemplars, leveraging dense vector similarity to handle paraphrases and varied phrasing in 5.
- **LLMRouter:** schema-validated LLM classification with confidence scores, where the LLM outputs structured JSON describing the predicted department and a confidence estimate.

This combination allows the system to use rules for high-precision known patterns, fall back to embedding similarity for more flexible matching, and escalate difficult cases to the LLM for richer reasoning.

LLM Adapter and Proxy Layer The LLM integration is split into an Adapter and a Proxy/Decorator layer.

Adapter: exposes a uniform `LlmClient` interface. Concrete adapters wrap OpenAI, Anthropic, or local models, translating between domain objects and provider APIs, and allowing the rest of the codebase to remain vendor-agnostic (Incorporating Large Language Models into Production Systems, 2024).

Proxy/Decorator: wraps any `LlmClient` to add caching, rate limiting, retry logic, and circuit breaking. This layer controls latency and API spending without altering core client logic, aligning with best practices for AI performance and reliability management (Nebius, 2024).

Responder Factory The `ResponderFactory` uses the Factory pattern to instantiate department-specific responder stubs (e.g., `BillingResponder`, `TechResponder`). Each stub implements a `Responder` interface that invokes a simple FAQ retrieval service to generate minimal draft replies for demonstration. Similar lightweight advisor components have been used in prior ticket-advisor systems to provide first-response suggestions without taking full automation risk (Feng et al., "TaDaa").

These pattern choices are guided by classical OO principles:

Listing 3: PII scrubbing preprocessor.

```python
# triage/preprocess/pii.py
import re

from triage.core.models import Ticket
from .base import Preprocessor


EMAIL = re.compile(r"[A-Za-z0-9._%+-]+@[
    A-Za-z0-9.-]+\.[A-Za-z]{2,}")
PHONE = re.compile(r"\+?\d[\d\- ]{7,}\d"
    )
ORDER = re.compile(r"(?:order|ord|#)\s
    *[\w\-]{5,}", re.I)


def _redact(s: str) -> str:
    s = EMAIL.sub("<EMAIL>", s)
    s = PHONE.sub("<PHONE>", s)
    s = ORDER.sub("<ORDER_ID>", s)
    return s


class PiiScrubber(Preprocessor):
    async def process(self, t: Ticket)
        -> Ticket:
        return Ticket(
            id=t.id,
            text=_redact(t.text),
            locale=t.locale,
            meta=t.meta,
        )
```

- **Single Responsibility Principle (SRP)**: handlers, routers, clients, and responders each focus on one role.
- **Open/Closed Principle (OCP)**: new routing strategies, preprocessing steps, and responders can be introduced via new classes implementing existing interfaces.
- **Liskov Substitution Principle (LSP)**: all routers can be treated interchangeably by the orchestrator; all LLM clients can be treated uniformly by decorators and routers.
- **Dependency Inversion Principle (DIP)**: high-level components (API layer, orchestrator) depend on abstractions (Router, LlmClient, Responder), not concrete implementations, which are wired in at composition time.

Evaluation Harness A lightweight evaluation harness supports A/B testing and ablation studies:

- A test runner toggles between routing configurations (e.g., rules-only vs. rules + embeddings + LLM) and logs per-ticket metrics.
- Ablation controls disable individual components (e.g., preprocessing, cache, specific strategies) to measure their impact on accuracy, $p95$ latency, cache hit rate, and cost per 1,000 tickets.

These metrics mirror those recommended for AI system

Listing 4: Text normalization preprocessor.

```python
# triage/preprocess/normalize.py
import unicodedata
import re

from triage.core.models import Ticket
from .base import Preprocessor


class Normalizer(Preprocessor):
    def __init__(self, max_chars: int =
        512):
        self.max_chars = max_chars

    async def process(self, t: Ticket)
        -> Ticket:
        text = unicodedata.normalize("
            NFC", t.text)
        text = re.sub(r"\s+", " ", text)
            .strip()
        text = text[: self.max_chars]
        return Ticket(
            id=t.id,
            text=text,
            locale=t.locale,
            meta=t.meta,
        )
```

evaluation in production environments (Nebius, 2024; Zhao et al., 2025).

Testing and Evaluation

Robust testing and rigorous evaluation are essential to validate correctness, performance, and production readiness of the Python-based ticket-routing system. We organize testing into three layers: unit testing, integration testing, and system-level evaluation, using Python-native tools and frameworks.

Unit Testing

Objective: verify that each Python module behaves correctly in isolation across expected and edge-case inputs.

Tools and setup:

- `pytest` for test execution.
- `unittest.mock` for mocking external dependencies (LLM APIs, Redis cache, FAQ service).
- `coverage.py` to track code coverage, with a target of at least 90% for core modules.

Integration Testing

Objective: ensure that the preprocessing chain, routing strategies, adapter, proxy, and responder factory work together seamlessly.

Tools and setup:

- `pytest` fixtures to load anonymized ticket samples.

Scenarios:

Listing 5: Embedding-based router using LangChain FAISS + HF embeddings.

```python
class EmbeddingRouterLC:
    """LangChain FAISS + HF embeddings.
        Sync, but fast enough for API."""
    async def route(self, t: Ticket) ->
        RouteDecision:
        results = self.store.
            similarity_search_with_relevance_scores
            (
            t.text, k=self.k
        )
        if not results:
            return RouteDecision("
                General", 0.0, "embed")

        top_doc, top_score = max(results
            , key=lambda pair: pair[1])
        dept = top_doc.metadata.get("
            dept", "General")
        conf = float(min(1.0, max(0.0,
            top_score)))

        if conf >= self.threshold:
            return RouteDecision(
                department=dept,
                confidence=conf,
                strategy="embed",
            )
        return RouteDecision("General",
            conf, "embed")
```

- *End-to-end pipeline:* for each sample ticket, run `pipeline.process(ticket)` and assert the returned department and draft reply format.
- *Rate limit and circuit breaker:* configure the mock LLM API to return HTTP 429 for N consecutive calls and verify that the proxy opens the circuit and falls back gracefully.

System-Level Evaluation

System-level tests simulate realistic workloads to measure throughput, tail latency, and cost. We replay historical or synthetic ticket streams through different configurations and record:

- routing accuracy against labeled ground truth;
- latency percentiles (p50, p90, p95);
- cache hit rate for LLM calls;
- estimated cost per 1,000 tickets under assumed API pricing.

These experiments are designed to compare rules-only, hybrid, and LLM-only variants and to surface trade-offs between accuracy, latency, and cost.

Expected Outcomes

Modular Routing Framework

A fully functional Python-based library implementing a Strategy-pattern router with three interchangeable strategies—rule-based, embedding k-NN, and LLM-driven classification—exposed through a consistent `Router` interface. This framework will demonstrate seamless integration and switching of classification methods with minimal code changes.

Robust Preprocessing Pipeline

A Chain of Responsibility preprocessing module capable of accurately detecting ticket language, scrubbing sensitive PII data, and normalizing text across multiple languages and formats. The module will include unit tests confirming high accuracy for language detection and correct redaction of all PII categories in benchmark datasets.

Vendor-Agnostic AI Integration Layer

An Adapter-based abstraction for interacting with multiple LLM providers, combined with a Proxy/Decorator layer that provides caching, rate limiting, retry logic, and circuit breaking. Performance benchmarks are expected to show a substantial cache hit rate and a significant reduction in external API calls under typical workloads, translating to significant cost savings (Nebius, 2024).

Evaluation Results and Metrics Dashboard

A comprehensive evaluation report and interactive dashboard showcasing:

- latency percentiles (p50, p90, p95) aiming for p95 $\leq$ 500 ms;
- cost per 1,000 tickets under simulated API billing, targeting a 40–60% reduction compared to a baseline LLM-only approach;
- comparative accuracy across routing configurations.

Documentation and Test Suite

Comprehensive code documentation, including API references and architectural diagrams, and a full `pytest` test suite with at least 90% code coverage for core modules. The codebase will demonstrate core OO design principles: encapsulation, loose coupling, and polymorphism.

Foundation for Open-Source Expansion

The design and documentation will allow future contributors to develop new plugins easily (e.g., additional routers, custom preprocessors, new responder types), promoting collaborative expansion of the library.

Conclusion

This project delivers a comprehensive, production-minded architecture for automated support ticket classification and routing that harmonizes cutting-edge AI capabilities with proven software engineering principles. By leveraging the Strategy pattern for interchangeable routing methods, the Chain of Responsibility for robust preprocessing, an Adapter/Proxy layer for vendor-agnostic and performance-optimized LLM integration, and a Factory for department-specific responder generation, the system addresses critical enterprise requirements: accuracy, maintainability, cost control, and reliability. The rigorous testing and evaluation methodology—including unit tests, integration tests, A/B comparisons, and ablation studies—will yield actionable insights into the trade-offs between model complexity, latency, and operational expenses, guiding optimal pipeline configurations for diverse organizational contexts Zhao et al. (2025); Kiashemshaki et al. (2025).

Beyond its immediate application to support desks, this work establishes a reusable architectural template for other decision-driven workflows such as content moderation, lead routing, or incident triage. The open-source Python library, complete with documentation, test suites, and evaluation dashboards, will empower enterprises to integrate AI-enhanced decision systems without sacrificing software quality or incurring unsustainable costs. Ultimately, this research bridges the gap between theoretical AI advancements and their practical deployment in complex, high-stakes production environments, advancing the state of the art in enterprise software engineering.

References

Analytics Vidhya. 2023. Enhancing Customer Support Efficiency Through Automated Ticket Triage. https://www.analyticsvidhya.com/blog/2023/11/enhancing-customer-support-efficiency-through-automated-ticket-triage/. Online article, accessed 2025-12-03.

Juno445. 2022. Ticket-classification-model: Intelligent Support Ticket Routing System. https://github.com/Juno445/ticket-classification-model. GitHub repository, accessed 2025-12-03.

Kiashemshaki, S.; et al. 2025. Automated Bug Triaging Using Instruction-Tuned Large Language Models. *arXiv preprint*. Preprint.

KushalPatil. 2022. Support-Ticket-Classification-using-NLP: Machine Learning for ITSM Tickets. https://github.com/KushalPatil/Support-Ticket-Classification-using-NLP. GitHub repository, accessed 2025-12-03.

Pia AI. 2024. How AI Transforms Ticket Triage: Efficiency Beyond Human Speed. https://pia.ai/blog/how-ai-transforms-ticket-triage-efficiency-beyond-human-speed/. Online article, accessed 2025-12-03.

ResearchGate. 2024. Automated Ticket Classification for Information Technology Helpdesks Using Machine Learning. https://www.researchgate.net/publication/382280579_Automated_Ticket_Classification_for_Information_Technology_Helpdesks_using_Machine_Learning. Online article / project, accessed 2025-12-03.

Rochester Institute of Technology. 2025. Automated Prioritization and Routing of IT Support Tickets Using Machine Learning. https://repository.rit.edu/theses/12020/. Online thesis/project page, accessed 2025-12-03.

subhromitra. 2021. Automated Helpdesk Ticket Classifier: Classify Category, Urgency, and Impact. https://github.com/subhromitra/Automated-helpdesk-ticket-classifier. GitHub repository, accessed 2025-12-03.

sukhijapiyush. 2020. Automatic Ticket Classification Using NLP and Machine Learning. https://github.com/sukhijapiyush/NLP-Case-Study---Automatic-Ticket-Classification. GitHub repository, accessed 2025-12-03.

Truss, M.; and Boehm, S. 2024. AI-based Classification of Customer Support Tickets: State of the Art and Implementation with AutoML. Technical report.

Zencoder AI. 2024. Traditional IT Helpdesk Systems Struggle with Manual Routing, Leading to 15–25% Misclassification Rates. https://zencoder.ai/blog/software-design-patterns-with-ai-future-trends. Online article, accessed 2025-12-03.

Zhao, P.; et al. 2025. Triage in Software Engineering: A Systematic Review of Research and Practice. *arXiv preprint*. Preprint.

Github Link: https://github.com/Jayanth710/triage