ChatGPT

Project Obvix Lake – AI-Powered Support Ticket Resolution & Knowledge Expansion System

General Architecture & System Overview

Project **Obvix Lake** is an advanced conversational AI architecture designed to transform customer support from a reactive, ticket-by-ticket process into a **continuous learning system**. It builds upon the prior Project Hyperion's design (which used a Finite-State Machine for dialogue management) and extends it with new components that enable the system to **learn from every support ticket**. The core idea is a **closed-loop knowledge expansion framework**: the system captures unresolved issues, mines resolution patterns from human-handled tickets, automatically grows the knowledge base with vetted solutions, and identifies emerging support trends for proactive action. In essence, every support interaction becomes an opportunity to make the overall system smarter.

Traditional support systems treat each ticket in isolation, but Obvix Lake treats the entire ticket ecosystem as **training data**. When an incoming issue exceeds the scope of the current knowledge base, the system intelligently routes it to a human agent **while** capturing that event for analysis. Every successful human resolution is semantically extracted and validated by experts before being integrated back into the knowledge base, creating a compounding knowledge advantage with each ticket resolved. The architecture leverages **Retrieval-Augmented Generation (RAG)** for semantic search of solutions, **unsupervised clustering** for trend detection, and **human-in-the-loop** validation for quality assurance. This orchestration yields a support platform that continuously improves over time, turning support operations from a cost center into a self-improving knowledge engine.

Key Challenges Addressed: Obvix Lake is purpose-built to tackle several long-standing pain points in customer support:

- **Knowledge Gaps:** Static knowledge bases quickly become outdated and fail to address new issues without constant manual updates.
- **Repeated Tickets:** The same problems recur frequently, forcing agents to repeatedly solve issues that were solved before.
- **Lost Resolutions:** Each time a human agent resolves a ticket, that valuable solution often remains uncodified and is not added to the knowledge base.
- **Trend Blindness:** It's hard to recognize emerging issue trends (e.g. a new bug affecting many customers) until they reach a crisis point, because data from individual tickets isn't analyzed in aggregate.
- **High Manual Overhead:** Keeping documentation and FAQs up to date requires dedicated staff and lags behind the actual issues customers face.
- **Siloed Historical Data:** Past support tickets (e.g. in systems like GLPI) contain solutions that never make it to the public knowledge base, resulting in duplicated effort and missed opportunities to deflect tickets.

Market Opportunity: The customer service software market is increasingly adopting AI automation, but most solutions still **separate ticket resolution from knowledge management**. Organizations need a system that can simultaneously resolve immediate customer issues *and* continuously improve

the support knowledge infrastructure. Obvix Lake meets this need by tightly integrating the real-time ticket handling with ongoing learning and content creation.

Architecture Overview: Obvix Lake retains the core conversation management foundation from Project Hyperion and adds several new intelligent modules to enable this closed-loop learning:

- *Maintained Core Modules (from Hyperion):*
  Customer Data Platform – unified customer profiles (CRM data, interaction history, etc.) to provide context.
  FSM Conversation Engine – a finite-state machine that orchestrates multi-turn conversations through predefined states (ensuring goal-directed dialogues).
  LLM Orchestration Layer – integrates a Large Language Model (LLM) to generate context-aware responses while maintaining coherence and following the conversation flow.

- *New Obvix Lake Modules:*
  Ticket Router & Classifier: Intelligently classifies incoming tickets and makes real-time routing decisions. It uses a multi-dimensional classifier (e.g. urgency, category, issue type) and a semantic similarity engine. High-confidence issues are auto-resolved via the knowledge base (self-service), while low-confidence ones are routed to human agents for intervention.
  Ticket System Integration & Resolution Extractor: Connects to the internal ticketing system (e.g. GLPI) to pull recently closed tickets and **extract structured resolutions**. This module uses NLP to parse the problem and solution from each resolved ticket, identifies key technical entities, and packages the information for reuse.
  Knowledge Base Expansion Pipeline: Automatically generates and vets new knowledge articles from the extracted resolutions. This pipeline applies templates and an LLM to draft articles, checks for duplicates, and then routes them through a multi-tier approval workflow (team lead and subject matter experts) before final integration into the knowledge base.
  Clustering & Trend Analytics Engine: Continuously analyzes unresolved and escalated tickets to detect emerging issues and support trends. It performs unsupervised clustering of ticket descriptions to find novel problem groupings and uses time-series analysis to flag when certain issue types are spiking. This allows the team to **proactively address new issues** (e.g. by creating a knowledge article or bug fix) before they affect a wide customer base.
  Feedback Loop Controller: Monitors performance metrics and closes the learning loop. It collects feedback from agents and customers (e.g. article helpfulness ratings, satisfaction scores), tracks usage analytics, and periodically triggers model retraining and knowledge base curation. This ensures the system keeps getting better – updating outdated content, improving classification accuracy, and aligning with evolving user needs.

These modules together enable a **closed-loop workflow** for support ticket handling and learning. At a high level, the process works as follows:

1. **Ticket Triage & Auto-Resolution:** When a customer submits a support ticket (in natural language), the **Ticket Router** module classifies the issue and searches the knowledge base for similar problems. If a matching solution is found with high confidence, the system provides an instant answer to the customer (self-service resolution), resolving the ticket without human intervention.
2. **Human Escalation:** If no good match is found (low confidence), the ticket is routed to a human support agent. The agent works with the customer and resolves the issue manually as in a traditional workflow.
3. **Capture Resolution:** Once the agent closes the ticket (e.g. in the GLPI system), the **Resolution Extractor** module kicks in. It automatically retrieves the ticket and parses out the key details: the

problem description, the steps the agent took to fix it, any important error codes or product names mentioned, and the final resolution notes. It creates a structured **"resolution object"** representing what was done to solve the issue.

4. **Generate Knowledge Article:** The new resolution object is fed into the **Knowledge Base Expansion Pipeline**. Here, a draft knowledge base article is generated (using a template populated with the resolution steps and enhanced with contextual details via an LLM). Metadata like relevant product keywords or tags are added.
5. **Approval Workflow:** Before becoming official, the drafted article goes through a **multi-tier human review**. First, a team lead or content reviewer checks it for clarity, correctness, and duplicates. Then, a domain expert (SME) may review technical accuracy. Only if it passes quality checks at each tier is the article approved for publishing (if issues are found, it's sent back for revision or rejected).
6. **Knowledge Integration:** Once approved, the new article is added to the **knowledge base**. It is indexed in the semantic search system (the content is broken into chunks and embedded as vectors for fast retrieval). The next time a similar issue arises, the system will be able to find this solution. The link between the original ticket and the new KB article can also be recorded (and even fed back into the ticketing system, so that if the issue ever reoccurs agents are pointed to the solution). The classifier models can be retrained on the expanded knowledge if needed.
7. **Trend Analysis & Proactive Alerts:** In the background, the **analytics engine** clusters any tickets that remain unresolved or were recently escalated. If it detects a *new cluster* of similar issues that aren't yet in the knowledge base (for example, a spike in support tickets about a specific error after a product update), the system will flag this as an **emerging trend**. It can alert support leaders or product teams about the trend, prompting proactive creation of a new knowledge article or a fix in the product. This way, the knowledge base coverage keeps pace with new issues before they become widespread.

Throughout this cycle, the **Feedback Loop** subsystem continuously measures how well the system is performing. It tracks metrics like what percentage of tickets are being solved via self-service, how fast resolutions are, customer satisfaction scores, etc., and uses this data to improve the system. For example, if certain knowledge articles are rarely used or getting poor feedback, they can be revised or removed; if the AI classifier starts making mistakes because of evolving terminology, it can be retrained with the latest data.

Expected Outcomes: By treating every support interaction as a learning opportunity, Obvix Lake aims to deliver significant improvements in support efficiency and quality. Key target metrics include:

- **Knowledge Base Coverage:** Increase the proportion of tickets resolved by the knowledge base (without human agent involvement) from ~45% to ~72% within the first year.
- **Resolution Time:** Reduce average ticket resolution time from about 24 hours to under 6 hours, thanks to instant answers for known issues and faster human resolutions aided by relevant suggestions.
- **Customer Satisfaction:** Improve customer satisfaction (CSAT) scores from roughly 3.8/5.0 to ~4.4/5.0 by providing quicker, more consistent solutions and proactive support.
- **Agent Productivity:** Enable support agents to handle more tickets per day (e.g. ~15/day, up from ~8/day) by offloading repetitive queries to the AI and equipping agents with readily available solutions for complex cases.
- **Knowledge Growth Efficiency:** Achieve a high ratio of auto-generated knowledge to manual documentation – for example, on the order of 4:1, meaning four new articles are programmatically extracted from real cases for every one article written manually. This indicates the system's ability to continuously expand its knowledge with minimal manual effort.

By closing the loop between ticket resolution and knowledge base evolution, Obvix Lake **turns support operations into a self-improving system**. Over time, this results in faster resolutions, lower support costs, happier customers, and a living knowledge repository that captures the organization's collective support wisdom.

Technical Implementation & Components

This section provides a detailed view of Obvix Lake's implementation, breaking down each component and how they work together. The current prototype has been developed as a Flask-based application (as a proof-of-concept) that implements many of these core components: it syncs persona-specific knowledge bases from Google Drive, updates embeddings in real-time, performs intent classification (tone and intent detection), uses a finite-state machine for conversation flow, and routes chats to the appropriate persona. Below, we describe each major part of the system and illustrate how the architecture has been realized.

Core System Components

Ticket Router & Classifier

This module is responsible for front-line **ticket triage**. It accepts an incoming customer issue (typically a text description of the problem) and decides how to route it. Internally, it has two main sub-components: an **Intent Classification model** and a **Semantic Similarity search engine**.

- **Intent Classification:** The system uses a fine-tuned BERT-based model to classify the ticket's intent and attributes. This NLP model (fine-tuned on IT support data) analyzes the text to determine things like the issue category (e.g. login issue, network outage, billing question), its urgency or sentiment (is the customer angry or is it a normal query?), and other tags. This classification helps in logging the ticket with the correct metadata and in choosing how to respond.
- **Knowledge Base Lookup:** In parallel, the system encodes the incoming ticket description into a vector embedding and searches for similar known solutions. A Sentence-Transformer model converts the text into a high-dimensional embedding vector. The system then queries a **vector index** (using FAISS or a vector database) to find the most semantically similar articles or past tickets. If one or more relevant knowledge base articles are found, a similarity score is computed (e.g. cosine similarity).

Using the classification results and similarity scores, the **Routing Decision Engine** applies business rules to route the ticket: if a solution in the knowledge base is a very close match (above a confidence threshold), the system will automatically respond to the user with that solution (thus resolving the ticket via self-service). In this case, the ticket can often be closed immediately with the answer provided. If no good match is found (or the confidence is below threshold), the ticket is marked for human handling – it gets assigned to a support agent. The classification (issue type, severity) helps ensure it goes to the right team or priority queue. Importantly, even when routed to an agent, the system can still assist by presenting the agent with the top similar articles for reference. The **technology stack** for this module includes Python (PyTorch + HuggingFace Transformers for BERT), OpenAI or Sentence-Transformer models for embeddings, and FAISS (Facebook AI Similarity Search) or an equivalent vector search engine for quick nearest-neighbor lookup.

GLPI Integration & Resolution Extractor

Once a ticket is resolved by a human agent, Obvix Lake shifts to **learning mode** – capturing that resolution so the AI can use it next time. This is handled by the GLPI integration module. **GLPI** (Gestionnaire Libre de Parc Informatique) is an open-source IT Service Management (ITSM) and helpdesk system [1] that the organization uses to track support tickets. Obvix Lake connects to GLPI via its REST API (using OAuth2 authentication) to fetch data on recently closed tickets.

Automated Ticket Sync: On a scheduled interval (e.g. every 6 hours), the system pulls all tickets that have been marked "closed" or "solved" since the last sync. There's also support for real-time updates through webhooks – if a ticket is closed, GLPI can send a notification to Obvix Lake to trigger an immediate fetch of that ticket's details. This combination of periodic sync and event-driven updates ensures no resolved ticket is missed. Robust error handling (with exponential backoff retries, etc.) is in place to handle API failures, and any problematic cases can go to a "dead letter" queue for manual review if needed.

Resolution Parsing: For each retrieved ticket, the **Resolution Extractor** processes the ticket data (especially the resolution notes entered by the agent). It performs several NLP steps to structure this information:

1. **Segment the resolution text:** It separates the **problem statement** (what the user asked or what issue occurred) from the **solution steps** (what the agent did or instructed the user to do). Often this is done by looking at how the agent's notes are formatted or by using an LLM to identify the steps versus the initial issue description.
2. **Extract technical entities:** Using Named Entity Recognition (NER) and custom regex rules, it pulls out key technical terms – e.g. product names, error codes, software versions, configuration parameters – that appear in the problem or solution. These become part of the structured data (they are useful as metadata and for searching similar issues).
3. **Classify resolution type:** It tags the general nature of the solution – for example, was this a **troubleshooting** issue (resolved by a series of diagnostic steps), a **configuration fix**, a **software bug** that required a patch, a **usage question** answered with information, etc. This categorization can help in reporting and in routing future issues (e.g. if something is classified as a known bug, the system might route similar new tickets directly to a specific team).
4. **Gather related references:** If the agent's notes or the ticket include any documentation links or knowledge base article references, those are extracted as well. (For instance, maybe the agent followed a company runbook or cited a manual – the system will note those links.)
5. **Compute resolution confidence/quality:** The system attaches a confidence score or quality score to the extracted resolution. This could be based on various ticket metadata – for example, the customer's post-resolution satisfaction rating, the agent's seniority or expertise level, whether the problem stayed resolved (no reopen), how long it took to resolve, etc. A high score suggests the solution is likely a good candidate for the knowledge base, whereas a low score might mean the issue was a one-off or the solution might not be generally applicable.
6. **Structure the resolution object:** All the above information is compiled into a structured JSON-like object, e.g.:

```
{
  "ticket_id": "GLPI-5432",
  "issue_title": "VPN connection timeout after firmware upgrade",
  "issue_description": "After upgrading the router firmware, all client VPN
connections time out.",
  "resolution_steps": [
```

```
    {"step": 1, "action": "Checked VPN client version and compatibility with
new firmware."},
    {"step": 2, "action": "Reset VPN configuration cache on the server."},
    {"step": 3, "action": "Rebooted the router to apply changes."}
  ],
  "technical_entities": ["VPN client v2.1", "Router firmware v8.3", "timeout
error code 504"],
  "resolution_type": "troubleshooting",
  "supporting_links": [],
  "resolution_confidence": 0.87,
  "agent_expertise": "senior",
  "customer_satisfaction": 0.92
}
```

In this example (illustrative data), the system captured what the issue was and the key steps the agent took to fix it, along with contextual info (entities, ratings). This **structured resolution** is now ready to enter the Knowledge Base Expansion Pipeline.

Knowledge Base Expansion Pipeline

This module takes the structured resolutions from the extractor and turns them into polished, published knowledge base articles. It consists of several stages that correspond to the content creation workflow, augmented by AI:

- **Draft Generation:** The pipeline's Article Generation Engine uses a combination of templates and an LLM to create an initial draft article from the resolution. A template provides a consistent format (for example: *Problem*, *Environment*, *Solution Steps*, *Additional Notes* sections), and the LLM (e.g. GPT-4) is used to enhance the text – rewriting the technician's resolution notes into a more user-friendly explanation, adding context where needed, and ensuring clarity. The draft includes metadata fields such as keywords, related issue tags, estimated difficulty, etc., which are derived from the technical entities and classifications done earlier. (For instance, if the issue was a firmware bug, the draft might include a tag for the product line and firmware version.)
- **Deduplication Check:** Once a draft article is prepared, the system performs a semantic **duplicate detection** to avoid knowledge base bloat. The draft's content is embedded into a vector using the same method as other articles, and compared against the existing knowledge base vectors for similarity. If the new article is very similar to an existing article (above a similarity threshold, e.g. cosine similarity > 0.85), it might indicate this issue was actually already covered by an existing solution. In that case, instead of publishing a redundant article, the system flags it for human review – possibly to merge the information with the existing article or to update that article with new details. If no close match is found, the draft moves forward.

- **Multi-Tier Approval Workflow:** Publishing new support knowledge requires quality control. Obvix Lake implements a **multi-tier approval process** to ensure each article meets standards. The typical workflow is:

- **Tier 1 – Team Lead Review:** The draft article is first reviewed by a support team lead or content moderator. The system can aid this by running automatic quality checks (ensuring the article has proper formatting, no placeholder text, no sensitive data, etc.). The team lead verifies that the content is clear, accurate at a high level, and not duplicating existing articles. They can approve it to proceed, or send it back for edits.

- **Tier 2 – SME Review:** For technical or complex articles, a Subject Matter Expert (SME) in that domain reviews the article in depth. They check technical accuracy, completeness, and that the solution would indeed resolve the issue. This step catches any mistakes the AI or the first reviewer might have missed. The SME can approve, request revisions, or reject the article if it's not salvageable.
- **Tier 3 – Optional Stakeholder/Customer Review:** In some cases (e.g. if an article pertains to a critical product issue or will be widely visible), a final review by a product manager, documentation team, or even a limited customer beta can be done. This is optional and used for high-impact knowledge items.

The workflow is managed by a **custom approval engine**, which notifies the relevant reviewers (usually via email or an internal dashboard) and tracks the status of each article through the stages (Pending, Approved, Rejected, etc.). If an article is rejected or needs changes, it is routed back to the content editing stage (with feedback). Approved articles proceed to publishing. This human-in-the-loop layer ensures the knowledge base maintains high quality and trustworthiness.

- **Publish & Index:** In the final integration phase, the approved article is assigned a unique ID and saved into the **Knowledge Base repository**. The content is indexed both in a traditional manner (for keyword search in the help center, etc.) and in the vector index for semantic search. The system also updates references: for example, it may add a link in the GLPI ticket that this issue is now answered by KB article X (creating a backward link so if someone looks at that ticket in the future, they see the knowledge article). Version control is applied if this updates an existing article. The vector embedding of the article (and its chunks) is stored in the vector database so that the **next time a similar ticket comes in, the Ticket Router can retrieve this article** and potentially solve the ticket automatically.

Throughout this pipeline, various technologies are used: **LangChain** (a framework for orchestrating LLM calls and workflows) helps with integrating GPT-4 for content generation and summarization, a vector database like **Weaviate** (or Pinecone) stores the embeddings and supports similarity queries, and the approval workflow logic may be a custom Python module or even integrated into a tool like Jira or an internal dashboard for tracking approvals.

Clustering & Trend Analytics Engine

Not all issues can be solved immediately, and some patterns only emerge when looking at many tickets together. The Clustering & Analytics engine addresses this by performing **batch analysis** on the pool of unresolved or recently resolved tickets, to glean higher-level insights.

- **Unsupervised Clustering:** The engine periodically takes a set of tickets (for example, all tickets from the past week that were escalated to humans or that took unusually long to resolve) and computes embeddings for their descriptions. It then applies multiple clustering algorithms to these vectors to group similar issues. It uses **K-Means** clustering as a primary method (with the number of clusters *k* determined via an elbow method or silhouette score to find a good fit). Additionally, it may use **Hierarchical Agglomerative Clustering** to visualize how clusters relate (this can suggest an issue taxonomy or sub-clusters), and **DBSCAN** (Density-Based Spatial Clustering) to detect outliers or small clusters of very similar issues that might be overlooked by K-Means. By using these algorithms in tandem, the system captures both broad groupings and niche problem signals.
- **Cluster Analysis & Summarization:** For each cluster found, the system generates a **summary description**. It can pick representative tickets (e.g. the one closest to the cluster centroid) or use an LLM to summarize the common theme of the cluster ("e.g. a number of users cannot reset their password via the mobile app"). It calculates metrics like cluster size (how many tickets), the

time range (are they all in the last 2 days?), and any common entities (maybe all involve a certain product or error code). Each cluster is given a tentative label (like "Password Reset – Mobile App Issue") to make it easily understandable to humans. The engine also computes a **severity or priority score** for each cluster based on factors such as the number of tickets in it, the average customer pain (maybe via satisfaction scores or urgency levels of those tickets), and whether the cluster is growing.

- **Trend Detection:** Using historical data, the engine performs time-series analysis on cluster frequencies. It employs models like **Facebook Prophet** or other forecasting techniques to predict expected ticket volumes for known issue categories. If the actual volume of a cluster exceeds the forecast by a significant margin (e.g. more than 2 standard deviations), that cluster is flagged as an **anomaly** – potentially an emerging issue that is trending upward unexpectedly. Similarly, it can detect cyclical patterns (for example, certain issues happen every Monday, or after every product release). Each cluster gets a **trend direction** classification: growing, stable, or declining. An example rule: if a cluster's size has increased by >30% week-over-week and is forecasted to continue growing, mark it as an emerging trend that needs attention.
- **Insights & Visualization:** The results of this analysis feed into dashboards and reports. The system can output concrete **insights** such as: "Top 5 Emerging Issues This Week" (a list of cluster labels that grew the most), "Highest Impact Unresolved Issue Areas" (clusters with large size and high customer impact that are not yet fully addressed), or "Knowledge Gap Analysis" (clusters for which no existing knowledge base articles match – indicating where new content is needed most). These insights are presented via visualizations like cluster heatmaps, trend line charts, or impact matrices. Such analytics are invaluable for support managers and product teams – they provide a data-driven early warning system for problems in the field.

Under the hood, this component uses Python libraries like **scikit-learn** for clustering algorithms, possibly **UMAP/t-SNE** for dimensionality reduction to visualize clusters, and libraries like **Prophet** for time-series forecasting. Visualization might be handled outside the core system (e.g. via an analytics dashboard using Plotly or Grafana), but the engine provides the processed data.

Feedback Loop Controller

The Feedback Loop controller ensures that the system continues to refine itself using real-world outcomes. It focuses on four main areas: metric tracking, feedback ingestion, model retraining, and knowledge maintenance.

- **Metric Tracking:** The system automatically gathers key **performance metrics** for the support operation. Examples include: **Knowledge Base coverage** (what percent of incoming tickets were answered by the AI vs needed an agent), **Average resolution time** (overall and by category), **Customer satisfaction scores** from surveys, **Article usage** (how often each knowledge article is being viewed or used in answers), and **Agent escalation rate** (are agents seeing fewer repetitive tickets over time?). These metrics are logged and trended over time. They feed into weekly/monthly reports and also back into the AI improvement process.
- **Agent Feedback Loop:** The platform solicits feedback from support agents on the AI-provided knowledge. For instance, if an agent ends up handling a ticket, and the system had suggested an article or provided an answer that the agent found unhelpful, the agent can flag it. Agents might rate how useful a suggested article was, or whether the AI classification was correct. Low-rated articles are automatically flagged for review in the knowledge base – perhaps the content was inaccurate or not easily applicable, so it might need revision or removal. This helps maintain KB quality. Additionally, the feedback can be used as training data; for example, if agents consistently mark that certain suggestions were wrong for a ticket, the system can learn to not

surface those suggestions for similar tickets in the future (adjusting similarity thresholds or adding that scenario as a negative example for the classifier).

- **Customer Feedback Loop:** After a ticket is resolved (whether by AI or human), customers might be prompted with a short survey ("Did this answer your question? How satisfied are you?"). The system collects these **satisfaction scores and comments**. If a customer indicates that the AI-provided answer was not helpful, that is recorded and those cases can be reviewed to improve the knowledge base or the answer formatting. There's also **implicit feedback**: for example, if the system provided an answer and the customer ends up reopening the ticket or asking a follow-up, it implies the answer might not have fully solved the issue – that data is captured as well. All this helps identify weaknesses in knowledge articles or where the AI misunderstood the query.
- **Model Retraining:** To keep the AI models from getting stale or drifting, Obvix Lake sets up a schedule to **retrain and fine-tune models** with fresh data. For example, the intent classification model (BERT) can be retrained every week or month on a growing dataset of labeled tickets (including new tickets that were resolved, using their final categorization and any corrections made by agents as labels). The clustering and semantic models might also be updated as new vocabulary enters the support domain. The system can use A/B testing to validate model improvements – e.g. deploy a new classifier to a subset of queries and ensure it performs better (higher accuracy or lower false-positive rate) before full rollout. Model performance metrics like precision/recall are tracked against a baseline to know when retraining is needed.
- **Knowledge Base Maintenance:** Over time, some knowledge articles may become obsolete (e.g. referring to old product versions or issues that have been fixed permanently). The system monitors KB **usage and effectiveness** – if an article hasn't been used in a long time and corresponds to an old product, it might be a candidate for archiving. Similarly, if an article consistently fails to help (low feedback scores), it might be rewritten or removed. There is a versioning mechanism so that when products change, updated solutions can replace old ones while keeping an edit history. Periodic audits can be triggered for articles that haven't been reviewed in a while, to ensure compliance with any new policies or product changes.

In summary, the Feedback Loop controller ties together the metrics and human inputs to **continuously refine** both the knowledge content and the AI models, ensuring sustained performance gains and relevance of the system.

End-to-End Data Flow

The interactions between all the components can be visualized in the end-to-end data flow diagram below. It shows how a ticket progresses through the system, how knowledge is updated, and how feedback loops around:

```
CUSTOMER TICKET
        ↓
[ Ticket Router & Classifier ] → **Confidence Score**
        ↓
    HIGH CONFIDENCE (above threshold)            LOW CONFIDENCE (below
threshold)
        ↓                                                ↓
[ Self-Service Response ]                      [ Route to Human Agent ]
        ↓                                                ↓
  🟢 **Customer Issue Resolved**                    **Agent Resolves
Ticket**
```

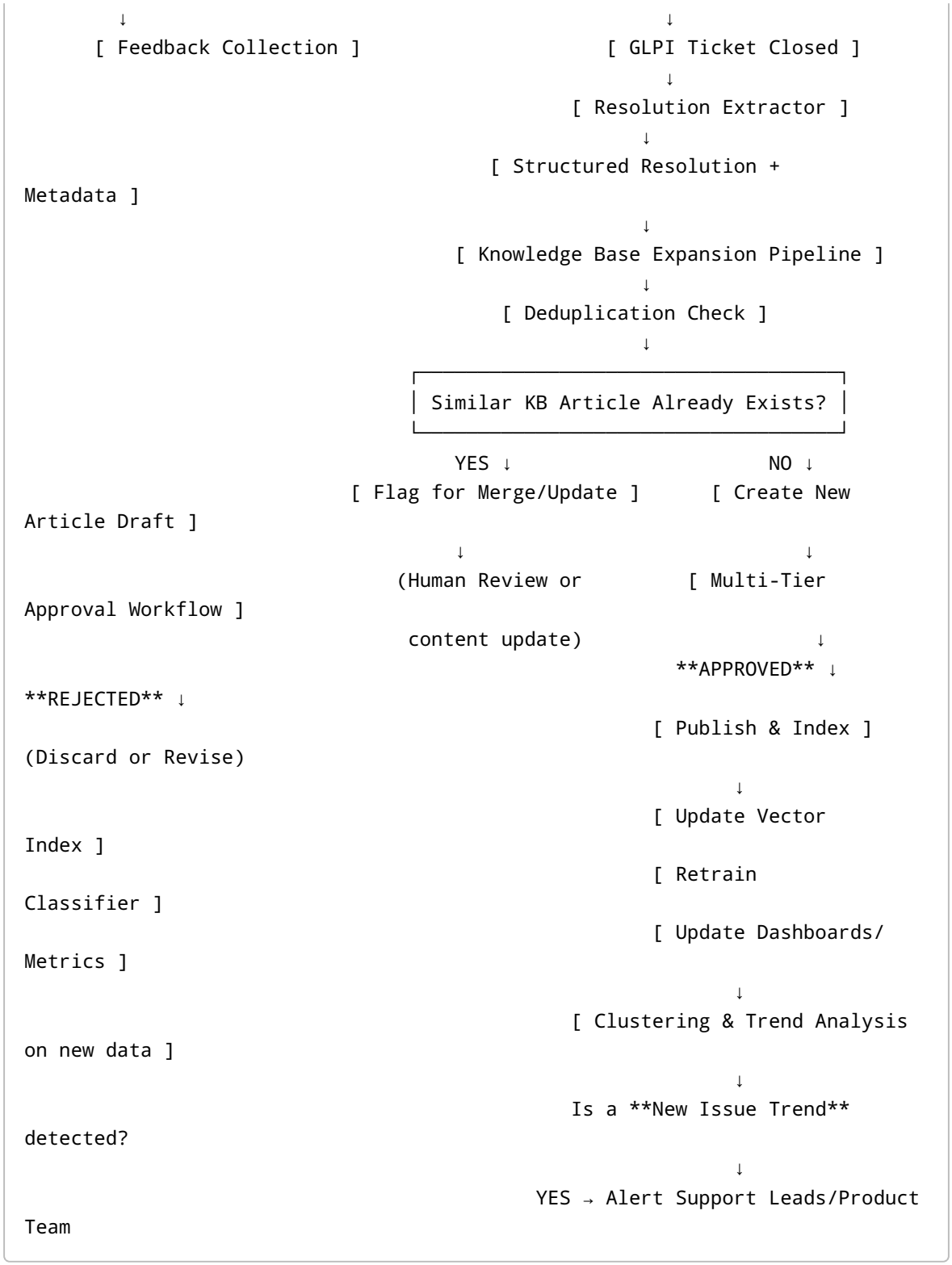

Diagram: High-level data flow of Obvix Lake. Tickets are either auto-resolved via knowledge base or handled by agents. Agent resolutions feed into the knowledge pipeline (draft → review → publish). The knowledge base and models are updated continuously, and trend analysis identifies emerging issues for proactive action.

In this flow, the left path shows an ideal scenario where the system knew the answer (self-service), and the right path shows the learning loop when an agent had to resolve it. Notice that after publishing a new article, the system immediately updates the **vector index** (so the new knowledge is searchable)

and can retrain the classifier if the new data changes the classification boundaries. The continuous clustering ensures even if an issue isn't fully solved, it's being watched and will trigger alerts. Authentication, error handling, and sync processes (not shown in detail above) ensure that the data flow is secure and reliable.

Architecture & Technology Stack

Obvix Lake is implemented as a collection of microservices and background workers orchestrated together. The architecture is modular, with clear separation of concerns across different layers:

- **Frontend Layer:** The user-facing components can include a Support Portal or Chatbot interface (for customers to ask questions and get answers), an Agent Dashboard (where support agents see tickets and AI suggestions), and an Admin Dashboard (for knowledge managers to oversee content). For example, a web UI built in React or Vue might serve the support portal and agent interface, and a Next.js or similar framework for any customer self-service site integration. These frontends communicate with the backend via APIs.
- **API Layer:** A set of RESTful (or GraphQL) APIs forms the integration point for frontends and external systems. These could be implemented with FastAPI (Python) or Express/Koa (Node.js) for high performance. Key services include: a **Ticket Router API** (to submit tickets or queries and get answers or routing info), a **Resolution Extraction API** (for triggering the GLPI data fetch and extraction, if not fully automated), a **Knowledge Base Management API** (to query articles, perhaps to add/edit articles manually if needed), an **Analytics API** (to fetch metrics and trends for dashboards), and an **Approval Workflow API** (to manage the review process for new articles). These APIs enforce authentication and orchestrate calls to the underlying services.
- **Service Layer:** This is the heart of the system where the business logic and AI live. It includes:
- **Classification Service:** (Python) The service hosting the intent classification model (BERT). It exposes an interface to classify incoming text and returns labels/confidence.
- **Embedding & Search Service:** (could be Python or Node.js) Responsible for generating text embeddings (using OpenAI's API or local models) and performing similarity search. It interfaces with the vector database (or FAISS index). This service is used by both the Ticket Router (to find similar articles) and by the pipeline (deduplication, etc.).
- **Resolution Extraction Service:** (Python) Contains the logic to connect to GLPI's API and run the resolution parsing algorithms (NER, etc.). This might run as a scheduled job or on-demand when triggered by a webhook.
- **Knowledge Base Expansion Service:** Manages the article generation and approval workflow. It invokes the LLM for drafting, checks the vector DB for duplicates, and interacts with human reviewers (perhaps via email or an internal web interface) to move the article through approval stages.
- **Clustering/Analytics Service:** (Python) Runs the clustering and trend detection jobs, perhaps as a batch process (e.g., a job that runs every night to analyze recent tickets). It stores results in a database that the Analytics API can serve to dashboards.
- **Feedback Loop Service:** (Python) Monitors incoming feedback data, triggers model retraining jobs, and performs knowledge maintenance tasks (like flagging old articles). This could be implemented as a set of cron jobs or an automation script.
- **Integration Layer:** These are connectors to external systems and resources:
- **GLPI Connector:** (Python module or microservice) Handles the OAuth2 authentication and API calls to the GLPI ticketing system for pulling tickets and posting any updates (for example, creating knowledge base reference links in GLPI, if desired).
- **CRM/External Connectors:** If needed, connectors to CRM systems or other data sources (e.g. if we want to enrich customer context or feed resolution info back to a CRM). Possibly implemented via webhooks or integration platforms (like Zapier or custom scripts).

- **Notification Service:** (e.g. using SendGrid or AWS SES) to send out emails for things like approval requests or alerting about emerging trends.
- **External API Gateway:** If exposing some functionality to third-parties or other internal systems, an API gateway could manage and throttle requests.
- **Data Layer:** The storage components used by the system:
- **Transactional Database:** A relational database (e.g. PostgreSQL) for core data like tickets, users, article metadata, and workflow states. The schema includes tables for Tickets, Resolutions, Knowledge_Articles, Clusters, Approvals, etc., capturing all information as described earlier.
- **Vector Store:** A specialized vector database (such as Weaviate or Pinecone) holds the embedding vectors for articles (and possibly for past tickets). This enables semantic similarity queries. Weaviate, for instance, provides a hybrid search (combining vector similarity with symbolic filters) and a GraphQL API for querying, which can be very useful for rich queries (e.g. "find articles about VPN issues with firmware in category networking"). If a fully managed vector DB is not used in the prototype, an in-memory FAISS index or even storing vectors in a MongoDB collection can be done for simplicity, though those are less scalable.
- **Cache:** A Redis cache might be used for storing intermediate results or frequently accessed data (for example, caching classification results or popular queries) to improve response times.
- **Document Storage:** A MongoDB or cloud storage (S3) might store unstructured data like raw ticket logs, conversation transcripts, or file attachments from tickets. In our case, the knowledge base articles themselves (the full content) might be stored in MongoDB as well for quick retrieval, while only their embeddings are in the vector store.
- **ML/AI Models:** The system integrates various models and external AI APIs:
- **LLM:** An OpenAI GPT-4 (or GPT-3.5 Turbo) model is used for generating answers, drafting knowledge articles, and performing advanced NLP tasks (like summarization or complex classifications). This is accessed via OpenAI's API. In future or on-prem deployments, this could be an Anthropic Claude or a local LLM if needed.
- **Embedding Model:** We use OpenAI's latest embedding model, `text-embedding-3-large`, to convert text into high-dimensional vectors [2]. This model produces 3072-dimensional embeddings and offers state-of-the-art semantic understanding for retrieval tasks. It allows the system to understand semantic similarity between customer questions and knowledge articles with high fidelity.
- **Classifier Model:** A fine-tuned BERT or similar transformer model (potentially hosted on HuggingFace or Azure ML) is used for intent and tone classification of incoming queries. Fine-tuning was done on thousands of historical support tickets to specialize it in IT support categories.
- **Clustering & Forecasting:** Algorithms like K-Means (from scikit-learn) and Prophet (for time-series) are part of the analytical toolkit. These are not learned models in the same way, but they require parameter tuning (e.g. optimal number of clusters).
- **Sentiment/Intent Models:** In addition to the main classifier, lightweight models or prompt-based classifiers are used for things like emotional tone detection or "buying intent" classification (useful if the system is also used for sales inquiries – more on that in persona section). These could be simple logistic regression classifiers or just handled by prompting the LLM to categorize tone/intent.
- **Infrastructure & DevOps:** The system is designed to be cloud-native and scalable. All components are containerized with **Docker**, and deployment is managed via **Kubernetes** (allowing scaling out services, rolling updates, etc.). Continuous Integration/Continuous Deployment (CI/CD) pipelines (e.g. GitHub Actions) are set up for automated testing and deployment of changes. Monitoring and observability are achieved through tools like **Prometheus** (collecting metrics on API response times, queue lengths, etc.) and **Grafana** for dashboards. Logging is centralized using an ELK stack (Elasticsearch, Logstash, Kibana) or cloud logging service, enabling developers to trace issues across microservices. Alerts (via Datadog or

NewRelic APM) are configured to notify if something goes wrong (like the GLPI sync fails, or a spike in errors).

Tech Stack Summary: In summary, the implementation uses a **Python-heavy backend** (Flask/FastAPI for APIs, PyTorch for ML, scikit-learn for analytics), complemented by specialized stores (Postgres, MongoDB, Weaviate vector DB) and integrates third-party AI services (OpenAI for GPT-4 and embeddings). The choice of Weaviate for vector search was motivated by its hybrid search capability and ease of use, but the design is abstract enough to swap in any vector database. The LLM is accessed via API to offload heavy language tasks, ensuring the solution remains at the cutting edge of AI capabilities without requiring on-premise model hosting. Containerization and cloud deployment ensure that the system can scale to handle enterprise workloads and can be maintained efficiently.

Persona-Specific Knowledge Base & Google Drive Sync

While the core system was described in the context of support tickets, Obvix Lake's architecture is built to handle **multiple personas or departments**, each with their own knowledge domain. In the current implementation, a **Google Drive** folder is used as a simple content repository for knowledge bases, with separate sub-folders for each persona. The use of Google Drive allows non-technical users to easily add or update documents (in formats like Google Docs, PDFs, etc.), and the system will automatically ingest those into its knowledge index.

Persona Knowledge Folders: Inside the main Google Drive (connected via a Google Drive API service account and credentials in a `client.json`), we maintain dedicated subfolders for each persona/agent we support. For example:

- `ol_customer_service` – Knowledge base for customer service inquiries (e.g. account issues, billing FAQs).
- `ol_technical_and_diagnostics` – Technical support knowledge (troubleshooting guides, diagnostic procedures).
- `ol_sales` – Information for sales agents (product info, pricing, upsell strategies).
- `ol_ops_and_logistics` – Content for operations/logistics (shipping procedures, inventory management FAQs).
- `ol_manufacturing_and_qa` – Knowledge base for manufacturing or QA-related queries (production processes, quality checklists).

Each of these folders essentially represents an **isolated knowledge base** for that persona. Documents placed in, say, the `ol_sales` folder will only be used by the Sales virtual agent and not by the Technical support agent, and vice versa. This segregation enforces context isolation – ensuring that each AI persona only cites information relevant to its domain and speaks in its domain's context.

Automated Drive Sync: A background **Flask syncer service** is responsible for keeping the system's knowledge indices in sync with the Google Drive content. It runs periodically (or can be triggered on demand) and performs the following:

- Connect to Google Drive using the service account credentials and list the contents of each persona's folder.
- Recursively traverse any subdirectories and gather all files (documents). The traversal ensures that if the knowledge documents are organized in sub-folders (e.g. "Product Manuals" under sales, or "Network Issues" under technical), those are all picked up.

- For each file, check if it's a new file or an updated version of an existing one (the syncer can keep track of file IDs and last modified timestamps). New or changed files will be processed; unchanged files can be skipped to save time.
- Download the file's content. Depending on the file type, different extraction is done: Google Docs can be downloaded as text or HTML, PDFs can be parsed with a PDF reader, etc. The result is plain text content of the document.
- Split the content into smaller **chunks** suitable for embedding (for example, splitting by paragraph or every few hundred tokens, ensuring not to exceed model token limits). For each chunk, generate a semantic embedding vector using OpenAI's `text-embedding-3-large` model. This produces a 3072-dimension numeric vector representation for the text, capturing its meaning in vector space.
- Store the chunks and embeddings in the vector database (or in the prototype, in a MongoDB collection if a full vector DB is not available). Each stored record at least includes: the persona identifier, the chunk text, the embedding vector, and a reference to the source document (so that we can retrieve the full context or provide a link to the original doc if needed). Optionally, metadata like the chunk's position in the document or the document title can also be stored.

A simplified pseudo-code excerpt for the sync logic is as follows:

```
def sync_persona_folder(persona_name, folder_id):
    files = drive_api.list_files(folder_id)
    for file in files:
        if file.is_folder:
            sync_persona_folder(persona_name, file.id)   # Recurse into
subfolders
        else:
            content = download_file_as_text(file.id)
            chunks = split_text_into_chunks(content, max_tokens=500)
            for chunk_text in chunks:
                embedding = openai.Embedding.create(input=chunk_text,
model="text-embedding-3-large")["data"][0]["embedding"]
                vector_db.upsert({
                    "persona": persona_name,
                    "text": chunk_text,
                    "source_doc": file.name,
                    "embedding": embedding
                })
```

Code snippet: Pseudocode for Google Drive sync. It iterates through each persona folder, downloads documents, splits them, and calls OpenAI's embedding API to store/update the semantic vectors in the database.

This process runs for each persona (e.g. `persona_name="ol_customer_service"`, etc.). After a sync, the system's vector index is up-to-date with the latest contents of the Google Drive knowledge bases. If, for example, the sales team updates a pricing document, within the next sync cycle the Sales agent's knowledge will include that update.

Persona Profiles: In addition to pure knowledge documents, each persona folder can also include a special **profile document** that defines the persona's characteristics – such as the tone of voice, introductory script, or any specific guidelines that agent should follow. For instance,

`ol_customer_service` might have a file describing that this persona should be friendly, apologetic for inconveniences, and follow support empathy best practices; `ol_sales` might have a profile emphasizing a proactive, enthusiastic tone and up-selling when appropriate. The sync process can detect such profile documents (e.g. by a naming convention like `persona_profile.txt`) and load them as part of the persona's configuration rather than as searchable knowledge. These profiles are later used to prime the LLM prompts so that each persona's responses have the right style and context.

Isolated Embeddings & Query Routing: Crucially, the system **enforces isolation** between persona knowledge bases at query time. This means when the Sales agent is asked a question, the retrieval step will only search within the embeddings tagged as `persona = "ol_sales"`. Each persona's vector data is either stored in a separate index or partitioned by a persona field so queries can be filtered. This prevents, say, a technical support answer from leaking into a sales conversation. It also allows each persona's knowledge base to evolve independently.

At runtime, when a request comes in, the API specifies which persona is being engaged. For example, an endpoint might be called as `/chat/ol_sales` with the user's query. The backend will route this to the appropriate knowledge base. A simplified example of how a chat query is handled for a given persona:

```
@app.post("/chat/{persona}")
def chat_with_persona(persona: str, user_message: str):
    # Retrieve top relevant knowledge chunks for this persona
    results = vector_db.query(persona=persona, query_text=user_message,
top_k=5)
    retrieved_content = "\n".join([res["text"] for res in results])
    # Load persona profile/instructions
    profile = persona_profiles.get(persona, "")
    # Craft the prompt for the LLM with persona profile + retrieved content
    prompt = f"{profile}\nUser query: {user_message}\nRelevant info:
\n{retrieved_content}\nAgent answer:"
    # Generate answer using the LLM (with tone/intent considerations, see
below)
    answer = llm.generate(prompt)
    return {"answer": answer}
```

Code snippet: Pseudocode for handling a chat request for a specific persona.

In this pseudocode, `vector_db.query` ensures only content for the specified persona is searched, and the persona's profile is prepended to guide the LLM's response. The system can thus dynamically route conversations to the correct persona's knowledge and style based on either an API parameter or an initial classification. For instance, if a unified chatbot front-end is used, a classifier could examine the user's question and decide it's a sales question vs. a support question, then internally forward it to `/chat/ol_sales` or `/chat/ol_customer_service` as appropriate – achieving **dynamic persona routing**. In our architecture, the clear demarcation of knowledge by persona and the routing logic ensure that each AI agent behaves as a specialist in its domain, with no cross-contamination of information.

Conversational AI & Multi-Stage Dialogue Flow (LLM Integration & FSM)

At the core of each persona agent's interaction with users is a combination of **LLM-driven conversation** and structured dialog management via a Finite-State Machine (FSM). This ensures that while the agent benefits from the creativity and understanding of a large language model, it still follows a coherent, goal-oriented flow appropriate for the conversation type (be it troubleshooting, sales inquiry, etc.).

LLM Integration for Understanding and Generation: We integrate an LLM (like OpenAI's GPT-4) to handle the natural language understanding and response generation tasks. The LLM is used in several ways:

- **Intent & Tone Classification:** Before formulating a response, the system may use the LLM (or a smaller classification model) to analyze the user's message for emotional tone and intent. For example, it classifies if the user is frustrated/angry (requiring a very apologetic tone), or if the user's query indicates a potential sales opportunity (buying intent), or if there is a specific urgency. These classifications influence how the agent should respond. In practice, this can be done by prompting the LLM with the user message and asking it to output a structured analysis (e.g. *"Tone: Angry; Intent: Complaint about billing; No obvious sales opportunity;"*). This could also be accomplished with separate fine-tuned models for sentiment and intent, but leveraging the LLM keeps the architecture simpler.
- **Contextual Retrieval-Augmented Generation:** As shown in the persona routing snippet, when generating an answer, we augment the LLM's prompt with relevant retrieved knowledge. The prompt typically includes the persona's profile/instructions, the top knowledge base excerpts (if any were found for the query), and the user's question. For example, a prompt for the technical support persona might be: *"You are a helpful technical support agent. [Profile instructions] ... The user asks: '<user question>'. Here are some relevant knowledge base snippets: ... <snippets> ... Using this information, provide a clear resolution."* The LLM then produces a response that hopefully uses the provided facts to answer the question. Because the knowledge base info is included, the LLM's answer will be grounded in the company's actual data (that's the RAG approach).
- **Multi-turn Dialogue Memory:** The LLM can also be fed a summary or the last few turns of the conversation to maintain context. For instance, if the conversation state is in "troubleshooting step 3," the prompt will include that context so the LLM knows what has been done so far. This can be managed via the FSM (discussed next) by summarizing the state or including state variables in the prompt.

Finite-State Machine for Dialog Flow: While the LLM is very powerful at understanding and generating text, we use an **FSM Engine** to enforce a logical progression in certain conversations, especially for support troubleshooting. The FSM defines states and transitions that represent the typical flow of an interaction. For example, a simple support FSM might have states like: `Greet -> Collect Issue Details -> Offer Solution -> Confirm Resolution -> Close or Escalate`. Each state can have certain actions (like scripts or checks) and transitions based on user input or system conditions.

In Obvix Lake, the FSM works in tandem with the LLM: the FSM controls *when* to ask the next question or *what* kind of response to give next, while the LLM handles *how* that question or answer is phrased naturally.

- **Example (Technical Support):** The conversation might start in a `Greeting` state where the agent says hello and asks how it can help. Once the user describes an issue, the FSM transitions to `Diagnostics` state. In this state, the system might have a series of checks (the FSM can use

the knowledge of common issues to decide on next steps). It could prompt the user for specific details (using the LLM to word the question). The user's answer might either satisfy a condition (e.g. problem identified) or require further probing. The FSM ensures all necessary info (like error codes, device type) are collected – it will keep looping in `Diagnostics` questions until certain slots are filled. When enough info is gathered, the FSM transitions to `Solution Proposal` state. Here, it uses the knowledge base to form a solution (with the LLM formulating the answer). After giving a solution, it goes to `Confirmation` state to ask if the issue is resolved. If yes, transition to `Closing` (and maybe collect feedback); if no, maybe loop back to an earlier state or escalate to a human.

- **Example (Sales):** For a sales persona, the FSM might be simpler: `Greet -> Qualify Lead -> Pitch Product -> Handle Objections -> Close (or Follow-up)`. The FSM would guide the conversation to ensure the agent asks qualifying questions (budget, requirements) before pitching, and attempts a closing or call-to-action. The LLM, enriched with the sales knowledge base (product details, etc.), generates the content of the pitch or answers to objections, but the FSM ensures the agent doesn't, for instance, forget to ask for the sale or gather contact info.

The FSM engine can be implemented via a state transition table or using a library. Our prototype uses a lightweight approach where each persona has a defined script/flow configuration. The conversation manager keeps track of the current state and transitions based on either keywords in the user's reply or flags set by the LLM's analysis of the reply. For example, if the user says "Yes, that solved my problem," the LLM might output an intent like `user_confirms_resolution`, which triggers the FSM to transition to the closing state.

Emotion and CTA (Call-to-Action) Handling: We also integrate specialized analysis for things like **CTA scoring** – especially in sales conversations, the system evaluates how close the interaction is to achieving a call-to-action (purchase, sign-up). The FSM could adjust strategy if the CTA score is low (e.g. user not convinced) by looping the agent to provide more info or a better offer. Conversely, if the LLM detects high buying intent (user is ready to purchase), the FSM can fast-track to closing the sale or handing off to a human closer. Emotional tone detection ensures that if a user is upset (say in a support context), the agent FSM might insert an apology state or escalate faster.

In practice, the **LLM is called multiple times** during a single complex interaction: one call might be to classify the user's last message (intent, sentiment), another to generate the next agent utterance given the state and retrieved knowledge. We make use of OpenAI's ChatCompletion with function calling for some of this – for instance, defining a "classify_message" function schema that the model can return, containing fields like tone, intent, next_state_recommendation. The FSM logic then consumes that and decides the transition.

Overall, this combination of an FSM-driven framework with LLM intelligence gives us the best of both worlds: structured conversations that meet business requirements and unpredictable scenarios handled with AI flexibility. The current Flask prototype includes a basic implementation of this: for example, a simple state machine for troubleshooting that interacts with the LLM. Even at this prototype stage, the multi-stage FSM and integrated LLM have demonstrated the viability of guided yet natural dialogues.

Security & Privacy Considerations

A production deployment of Obvix Lake must adhere to strict security and privacy standards, given it deals with potentially sensitive customer data and internal knowledge. The architecture incorporates several measures:

- **Authentication & Access Control:** All APIs are secured – using OAuth2 with JWT tokens for internal services and API keys or SSO for integration points. Role-based access control ensures that, for example, only authorized staff can publish knowledge articles or view certain analytics. Rate limiting is in place on public-facing endpoints to prevent abuse (e.g. brute-force or DoS attacks on the chatbot API). CORS policies are locked down so that only approved domains can embed the support widgets.
- **Encryption:** Data is encrypted both **in transit** and **at rest**. TLS 1.3 is enforced for all client-server communications (the APIs, the web interface). For data at rest, the databases use encryption (for instance, PostgreSQL with TDE or cloud-provided encryption, and AES-256 encryption for any file storage like S3 buckets). This protects ticket and customer data in case of any unauthorized access to the storage.
- **PII Handling:** The system performs automatic **PII detection and masking** on content that gets added to the knowledge base. Before a knowledge article is published, it's scanned for personal data (like customer names, emails, phone numbers) using regex patterns and ML models. Any such details are either removed or anonymized. Likewise, if a user query contains personal info, the system avoids logging it or ensures the logs are scrubbed. This is crucial for privacy laws like GDPR. In fact, Obvix Lake supports a "Right to be Forgotten" mechanism: if a customer requests deletion of their data, any tickets or chat logs associated with them can be located and purged from the system after a retention period (the architecture flags data with customer IDs to facilitate this).
- **Compliance:** The design is aligned with common compliance frameworks. For example, audit logs are kept for all important actions (publishing an article, updating a model, who approved what and when). This supports **SOC 2** Type II requirements (security and change management). If deployed in domains like healthcare or finance, additional measures can be taken to comply with **HIPAA** or **PCI-DSS** (e.g. not storing any credit card info, and using specialized redaction for health info). Our use of human-in-the-loop review also allows ensuring that no sensitive or disallowed content is present in AI-generated text (as humans will catch anything inappropriate before it goes live).
- **Securing Integrations:** The integration with GLPI and Google Drive uses dedicated service accounts with scoped permissions. API credentials (tokens, keys) are stored securely (e.g. in an environment vault or Kubernetes secret) and not hard-coded. The system also implements **exponential backoff and circuit-breaking** for external API calls to avoid overwhelming them and to handle any token refresh flows gracefully.
- **Monitoring & Alerts:** From a security operations perspective, the system's logs and metrics are monitored for anomalies. For instance, if there are many failed login attempts to the admin dashboard or a sudden spike in API usage out of normal patterns, alerts will be triggered. Integration failures (like inability to fetch GLPI tickets due to auth issues) similarly raise alerts so that they can be addressed promptly, ensuring data consistency.

In summary, Obvix Lake's architecture is designed with enterprise security in mind, combining technical safeguards (encryption, access control, logging) with procedural safeguards (approval workflows, audits) to protect data and ensure trustworthiness of the AI's output.

Implementation Team & Resources

Implementing a system of this scope requires a multi-disciplinary team, as well as cloud infrastructure to run the services and AI workloads. Below is an overview of the projected resource requirements:

- **Team Roles:** A team of around **6–7 full-time equivalents (FTE)** was estimated. Key roles include:
- **Machine Learning Engineer (1 FTE):** Focused on developing and fine-tuning the NLP models (classification, embeddings, clustering). Also responsible for model deployment and monitoring.
- **Backend Engineers (2 FTE):** Responsible for implementing the microservices (APIs, business logic, database integration) and ensuring the different modules (extractor, pipeline, etc.) work together.
- **DevOps Engineer (1 FTE):** In charge of the cloud infrastructure, CI/CD pipeline, container orchestration (K8s), and overall system reliability/monitoring.
- **QA Engineer (1 FTE):** Dedicated to testing the system – writing unit/integration tests, performing end-to-end testing of the workflows (including the AI components in a sandbox environment), and ensuring quality before releases.
- **Product Manager (0.5 FTE):** A half-time allocation for a PM to manage requirements, interface with the support teams (the end users of this system), and prioritize features.
- **Solutions Architect (0.2 FTE):** A part-time architect to provide guidance on system design choices, review security and scalability plans, and ensure the architecture meets the business needs and can integrate with existing IT.
- **Infrastructure & Budget:** The system runs on cloud infrastructure (AWS/GCP/Azure). Major cost centers include: cloud compute instances for running the microservices and the vector database, OpenAI API usage costs, and monitoring/tooling services. Based on initial estimates:
- Cloud VMs/Containers for the various services and databases were roughly **₹60,000 per month** (with redundancy and scaling headroom).
- The managed vector database (Weaviate Cloud or similar) and other storage costs came to about **₹15,000 per month**.
- OpenAI API usage (embedding and GPT-4 calls) was projected around **₹30,000 per month** initially, depending on volume of tickets and queries (this could vary with usage; the embedding model calls are cheaper per call, whereas GPT-4 is relatively expensive per message, so the budget assumes moderate usage and would scale with load).
- Monitoring/Logging services and other tooling (like a subscription to Datadog, plus the ELK stack maintenance, etc.) around **₹20,000 per month**.
- Development tools and licenses (e.g. GitHub Enterprise, any paid libraries) a smaller amount, about **₹8,000 per month**.

Summing these, the operational infrastructure cost is roughly **₹1.33 lakh per month**. Over 12 months that's ~₹16 lakhs. The bulk of the expense is in personnel: ~₹65 lakhs/year for the team. We also set aside a training/documentation budget (~₹2 lakhs) and a contingency (~10% of total) for unexpected costs or overages (~₹8+ lakhs). The **total project budget for the first year** comes to on the order of **₹90+ lakhs** (approximately \$120k), which covers development and deployment. This investment is justified by the anticipated reduction in support labor costs (estimated ~35% reduction by automation) and improved customer retention (faster, better support reduces churn by a few percentage points, which can translate to significant revenue protection).

Conclusion

Project Obvix Lake represents a **paradigm shift** in how support operations are handled – moving from a purely reactive model to a proactive, continuously learning model. By orchestrating advanced AI components and human expertise in a closed-loop system, it ensures that:

1. **Every unresolved ticket becomes training data** for improving the AI. The system learns from each failure to answer and comes back stronger with new knowledge.
2. **Every agent's expertise is captured as institutional knowledge.** Solutions that traditionally lived only in an individual's memory or in scattered ticket notes are now systematically collected and turned into reusable knowledge articles.
3. **Emerging issues are detected and addressed early.** The analytics engine spots patterns in support queries before they explode into major incidents, allowing the business to fix product problems or publish FAQs proactively.
4. **The knowledge base expands and updates itself automatically**, guided by human feedback to ensure quality. This greatly reduces the manual effort of documentation while keeping content accurate and up-to-date.

In the end, Obvix Lake delivers a self-reinforcing cycle: the more it's used, the better it gets. Faster resolutions and smarter self-service lead to happier customers and lower support loads; lower loads free up experts to create higher-value content and improvements. This **compounding effect** is the core promise of Obvix Lake. The successful prototype implementation has demonstrated the viability of the approach – with agents being able to get relevant suggestions and the system autonomously generating useful knowledge from real tickets. Moving forward, with disciplined execution and iterative refinement, Obvix Lake stands to achieve its targets (e.g. ~72% automated resolution, 6-hour average resolution time, 4.4/5 satisfaction) and provide a sustainable competitive advantage in customer experience. It's not just about solving tickets faster – it's about **building an ever-growing lake of knowledge** that continuously empowers both customers and support teams.

---

[1] GitHub - glpi-project/glpi: GLPI is a Free Asset and IT Management Software package, Data center management, ITIL Service Desk, licenses tracking and software auditing.
https://github.com/glpi-project/glpi

[2] New embedding models and API updates | OpenAI
https://openai.com/index/new-embedding-models-and-api-updates/