Project Voice2SIEM: Turning Customer Support Audio Into Real-Time Security Events

by Dinis Cruz and ChatGPT Deep Research, 2025/10/02

Problem and Architecture Overview

Phone-based **social engineering and vishing (voice phishing)** attacks are on the rise, targeting customer support and help desk agents. Attackers impersonate customers or executives to manipulate agents into divulging sensitive information or performing unauthorized actions (password resets, fund transfers, etc.). With AI-driven voice cloning, fraudsters can even mimic a victim's voice to fool biometric checks or **bypass security questions by social engineering the agent**[1]. The business impact is severe -- breached accounts, financial loss, compliance violations -- and **detecting these threats during a call is extremely challenging**. Human agents may miss subtle cues of manipulation, especially if the caller is persuasive or the attack blends into normal support workflows.

The technical challenge is to **analyze voice conversations on-the-fly for signs of phishing or manipulation**. This means turning an audio stream into structured data (transcripts, sentiment, detected intents) that security systems can evaluate. A straightforward approach is to treat each support call as an input to a pipeline of detection stages. The pipeline would transcribe the audio, analyze linguistic content and vocal tone for *urgency or stress*, extract semantic information (e.g. what requests are being made, what account or device is discussed), and then apply logic to decide if the call is suspicious. If a threat is likely, the system should generate a **security event** (e.g. alert in the Security Information and Event Management system, SIEM) so the incident can be investigated or action can be taken (like warning the agent or supervisor).

Real-time vs "fast enough" detection: Importantly, this pipeline does **not** need to operate in true real-time (i.e. within milliseconds). Unlike a network firewall, we don't necessarily have to stop the conversation mid-sentence. The goal is to be *fast enough* -- ideally analyzing the call **within seconds to a minute** -- so that if a high-risk social engineering attempt is detected, we can intervene *before the attacker's goal is achieved*. For example, if the caller is trying to convince the agent to reset a password or reveal an OTP (one-time passcode), detecting the threat just in time could prompt a supervisor to intervene or require additional verification. In essence, as long as the system flags the call **before a fraudulent transaction completes**, it's effectively real-time for security purposes.

Architectural Workflow

At a high level, **Project Voice2SIEM** proposes a pipeline with multiple stages, each transforming the raw audio into more refined signals (see **Figure 1**). By the end, the system has enough understanding of the conversation to decide if it warrants a security alert. The stages include:

1. **Audio Input Capture:** The customer support call (which may be telephone audio or VoIP) is recorded or streamed into the system. This could be a real-time audio feed from the telephony system or a recording saved after the call.
2. **Speech-to-Text Transcription:** The raw audio is converted to text using Automatic Speech Recognition (ASR). This produces a transcript of the call with timestamps and speaker identification (who said what). Accurate transcription is crucial since all further analysis relies on the textual content.
3. **Urgency, Tone, and Emotion Detection:** Beyond the words spoken, the system analyzes *how* things are said. It looks at indicators of stress or emotion in the audio (e.g. elevated volume or pitch, rapid speech suggesting urgency, or pauses suggesting hesitation). In the transcript, this can be supplemented by sentiment analysis -- e.g. detecting if the **customer is angry, anxious, or insistent**[2]. A manipulative caller might sound either unusually pushy or feign distress; these vocal features provide early clues.
4. **Semantic Extraction of Content:** In parallel with tone analysis, the transcript is parsed for **topics, intents, and requests** being made. For example, the system extracts what the caller is asking for ("reset my password", "update my email", "check a charge on my account"), any mention of sensitive information (account numbers, one-time passcodes), and entities like names or organizations. Natural Language Processing (NLP) techniques identify key entities and the overall intent of the call. The system might also detect conversation acts (caller provided authentication info, agent verified identity, caller requested an exception to policy, etc.).
5. **Conversation Graph Generation:** All the extracted data -- semantic facts, entities, the dialog sequence, emotional indicators -- are aggregated into a **graph representation** of the conversation. In this graph, nodes might represent the participants (customer, agent), the **utterances** (individual statements or questions), and important entities (like an account or ticket number). Edges can illustrate the flow of the conversation ("agent asks for verification after customer request") or relationships ("caller identity provided matches account name"). The graph format makes it easier to see the overall structure of the interaction and to apply pattern-matching for known attack scenarios.
6. **Scoring and Decision for SIEM Escalation:** Finally, a decision engine evaluates the graph and all collected signals to produce a **threat score**. This could be a simple rules engine or a machine-learning model trained on past call data. It considers factors like: *Did the caller exhibit high stress or urgency while requesting a sensitive action? Were there anomalies in authentication (e.g. multiple failed attempts, or the caller bypassing questions)? Does the sequence of events match known fraud playbooks (such as the "CEO impersonation" scam)?* If the score exceeds a threshold or certain red flags are present, the system generates a **security event** that is sent to the SIEM. This event would contain the pertinent details (timestamp, call

ID, transcript highlights, reason for alert) so security analysts or automated responders can take action.

Figure 1: Conceptual Voice2SIEM pipeline converting a support call into a security alert. The process begins with audio input from a customer-agent conversation. The audio is transcribed to text, then analyzed for emotional tone (e.g. urgency, anger) and semantic content (topics, intents, requests). These insights feed into a conversation graph representing the flow of the call. Finally, a scoring logic decides if the graph indicates a likely social engineering attack, triggering a SIEM event.

This architecture acknowledges that **no single signal is conclusive**. A caller might be angry for legitimate reasons, or mention sensitive info as part of normal troubleshooting. It's the **combination of signals and the context** that reveals a social engineering attempt. For example, an attacker might start very friendly (low anger, positive sentiment) then suddenly create urgency ("I need this done *immediately* or I'll get you fired!") -- a stark sentiment change coupled with a policy exception request. The conversation graph for such a call would show an abnormal sequence (from small talk to threats) that differs from genuine customer calls. By breaking down the audio and analyzing these aspects step by step, the system can catch what a human agent alone might miss.

Solution Using Commodity Cloud Services (AWS Reference Implementation)

Designing and implementing the above pipeline from scratch can be complex. Fortunately, modern cloud providers offer building blocks that can be assembled to create this voice-to-SIEM analysis system. To demonstrate, we outline a reference solution using **Amazon Web Services (AWS)** serverless components (note that similar services exist on Azure and Google Cloud, so the approach is portable). The emphasis is on using **managed, pay-per-use services** to achieve this inexpensively at scale. We don't need to maintain servers or deep ML expertise -- we can leverage cloud AI APIs that are readily available[3].

Key AWS components in the pipeline:

- **Audio Ingestion (S3 + Lambda):** A call recording (for post-call analysis) or a live audio stream is fed into the pipeline. In AWS, one simple method is to use Amazon S3 as an ingestion point: the call system saves the audio file (e.g. MP3/WAV) to an S3 bucket at the end of the call. This event triggers an AWS Lambda function (via S3 event notification) to start processing. For live calls, AWS Kinesis Streams or Amazon Chime SDK can stream audio in real-time, but for our "fast enough" approach a short post-call delay is acceptable. The Lambda retrieves the audio from S3 and initiates transcription.
- **Speech-to-Text with Amazon Transcribe:** The Lambda function uses **Amazon Transcribe** to convert the audio to text. Amazon Transcribe can operate in batch mode (transcribing a file from S3 asynchronously) or streaming mode (transcribing a live stream in real-time). In our reference design, the Lambda could call the Transcribe API to start a transcription job on the audio file. The result will be a transcript file (JSON or TXT) -- possibly stored back in S3 or

returned to the Lambda after completion. Transcribe can provide word-by-word timestamps and distinguish between speakers (important for identifying "who said what" in the dialogue).

- **Tone and Sentiment Analysis with Comprehend:** Once the transcript is ready (within seconds for an average call), another Lambda step (or the same Lambda if orchestrated sequentially) analyzes the text. **Amazon Comprehend**, a natural language AI service, can detect sentiment (positive, negative, neutral, mixed) of text and extract key phrases. Comprehend's sentiment analysis helps determine if the caller was angry, frustrated, or urgent during the call. This can be done at the overall call level and even per sentence to see if sentiment shifted over time. Additionally, Comprehend can perform entity recognition -- identifying names, dates, organizations mentioned -- which could flag if the caller mentioned things like a specific bank, a password, an account number, etc. These become pieces of metadata attached to the call record.
- **Intent and Keyword Extraction:** While Comprehend covers basic NLP, AWS offers other tools for deeper insight. Amazon Transcribe itself has a feature called **Call Analytics** that can directly identify **call categories and issues**, like spotting if certain phrases (e.g. "not happy", "speak to manager") occurred[4]. Alternatively, one could use **Amazon Lex**, a conversational AI service, to parse the transcript or even actively listen during the call to identify intents. For example, Lex could be configured with intents such as "PasswordResetIntent", "VerifyIdentityIntent", "AccountUnlockIntent" etc., and the transcript (or live audio via Lex integration) would reveal if the caller's requests match any of these. The combination of Comprehend and Lex can thus provide a structured view of *what the caller was trying to achieve*. AWS's own blog notes that using services like Transcribe with Comprehend and Lex enables capturing both **insights and intents from conversations**[3].
- **Orchestration and Data Flow:** All the above steps can be orchestrated with AWS Lambda functions passing data through Amazon S3 or in-memory. A simple approach is a **pipeline of Lambdas** triggered in sequence: audio file lands in S3 -> triggers Transcription Lambda -> writes transcript to S3 -> triggers Analysis Lambda -> which calls Comprehend/Lex -> outputs findings to a results database. AWS Step Functions (a serverless workflow service) could also manage this multi-step pipeline with error handling and retries built-in. Each service in the pipeline is pay-per-use: you pay only for the seconds of Lambda execution, the seconds of transcription, and the Comprehend API calls used, making it cost-efficient.
- **Graph and Pattern Analysis:** In an AWS-only implementation, we might not explicitly build a graph database for the conversation (since that introduces a stateful component). However, we can simulate the graph analysis through structured data and search indices. For example, after analysis we could construct a JSON object that represents the conversation structure (speakers, sequence of intents, sentiment timeline, entities mentioned). This JSON could be indexed in **Amazon OpenSearch** (the AWS-hosted Elasticsearch service) to enable complex queries and visualization. OpenSearch can serve as a lightweight SIEM database where all call transcripts and alert scores are stored. Analysts could search this index for patterns (like all calls where a password reset was requested and caller sentiment was angry). If needed, Amazon Neptune (a managed graph DB) could be used to store the conversation graph and run graph queries -- but that adds complexity, so our reference keeps it simple with JSON data and OpenSearch.

- **Scoring and SIEM Alerting:** The final step is deciding if a particular call is malicious. This logic can run in a Lambda function once all analysis data is available. The Lambda might use a set of rules (e.g., IF caller_sentiment = "angry" AND requested_action = "password_reset" AND auth_failed = true THEN high_risk) to compute a risk score. More advanced, it could use a machine learning model (perhaps SageMaker or even a Comprehend custom classifier trained on examples of fraudulent vs. normal calls). But clear, explainable rules are a good starting point. If the call is deemed suspicious, the system creates a **security event record**. In AWS, this could be an entry sent to **Amazon EventBridge**, which can route the event to various targets: an SNS notification to Security Engineers, a ticket in an incident management system, or even automatically calling an AWS Lambda to disable the customer's account until verified. Alternatively, the Lambda could index an "alert" document into the OpenSearch index with a field like `alert=true` so it shows up in SIEM dashboards. The key is that a tangible alert or log is generated and integrated with whatever SIEM or logging solution the company uses (could be Splunk, Elastic, Datadog, etc., via webhooks or connectors).
- **Modularity and Cloud Portability:** All components here are loosely coupled. Audio files and transcripts reside in S3 (or could be any object store), triggers are event-driven, and each analysis piece is a replaceable module. For instance, if one wanted to switch to Google Cloud, you could use Cloud Storage instead of S3, Google's Speech-to-Text instead of Transcribe, and Cloud NLP for sentiment/intent instead of Comprehend/Lex. The pipeline concept remains the same. Because it's serverless, scaling to thousands of calls is simply a matter of AWS handling more Lambda invocations and more parallel transcribe jobs -- no infrastructure bottlenecks. Cost-wise, using these services means you pay per call-minute processed, which for sporadic or moderate call volumes is extremely cost-effective compared to hiring a team of human monitors.

Data Pipeline Example (AWS): Below is a summary of how data flows in this serverless design:

1. **Ingestion:** Agent software or telephony records the call audio and uploads `call123.mp3` to S3.
2. **Transcription Trigger:** S3 event kicks off Lambda "TranscribeCall". It calls Amazon Transcribe to transcribe the audio file (language can be auto-detected if needed). Transcribe outputs `call123-transcript.json` to an S3 `transcripts/` folder.
3. **Analysis Trigger:** The upload of the transcript JSON triggers Lambda "AnalyzeCall". This function loads the transcript (could also get it from the Transcribe API result) and calls Amazon Comprehend for sentiment and entities. It may also invoke Amazon Lex (or a custom intent classifier) with the transcript to get recognized intents (e.g. `intent: ResetPassword`). The function compiles an analysis result JSON with fields like `sentiment_trend`, `key_phrases`, `detected_intent`, `caller_tone`, `auth_attempts`, etc. It saves this to S3 or sends it directly to the next step.
4. **Scoring & Alert:** A final Lambda "ScoreCall" (triggered by the presence of analysis results) evaluates all inputs. It might say: *sentiment went from neutral to highly negative when agent asked security question*, *caller requested high-risk action*, *caller provided account info after failing initial verification*. These factors are tallied into a risk score (say 85/100). If above threshold (e.g. 80), the Lambda sends an event to EventBridge with details (`alert: true, risk:85,`

`call_id:123, reason:"high urgency password reset"`). EventBridge forwards it to the security notification topic and also logs it into OpenSearch. If the score is low, the call record might just be logged in OpenSearch with `alert: false, risk:10` for learning/tracking.

5. **SIEM Integration:** In our case, Amazon OpenSearch acts as a simple SIEM data store where all calls and alerts reside. Security teams can query and visualize this (e.g. see trends, or get an alert feed). If the organization has an existing SIEM (Splunk, QRadar, etc.), the EventBridge rule could instead call a webhook or use a connector Lambda to forward the event to that system in real-time.

Through this AWS solution, we achieve a functional **voice-to-SIEM pipeline using off-the-shelf services**. We leveraged **Transcribe for speech-to-text**, **Comprehend for NLP insights**, and simple logic in Lambda for the decision -- demonstrating that even complex-sounding capabilities like emotion detection or intent recognition are accessible via APIs[3]. All data (audio, text, results) is centralized in S3/OpenSearch which provides an audit trail. Moreover, the **serverless architecture** means we can handle bursts of calls or scale down to zero when no calls are happening, paying only for actual usage. This cloud reference implementation proves the concept: organizations can start detecting social engineering in calls *today*, without waiting for a specialized vendor product, by composing existing cloud services.

Open Source and Semantic Graph-Based Implementation

While cloud services are convenient, some organizations prefer open-source, self-hosted solutions -- for flexibility, **transparency, and avoiding vendor lock-in**. Furthermore, to push the envelope, we can design a system that not only detects threats but does so in a **fully explainable, graph-driven manner**, aligning with cutting-edge semantic analysis techniques. In this section, we present an open-source blueprint using Dinis Cruz's stack of tools and frameworks, which are geared towards building **type-safe, graph-centric, and auditable AI systems**. The goal is to show how the Voice2SIEM pipeline can be constructed with open technologies, yielding a high degree of control and introspection into how decisions are made.

The open-source solution will use the following key components:

- **Cache Service (FastAPI-based)** -- for structured storage of audio, transcriptions, metadata, and event outputs.
- **MGraph-DB (Memory Graph Database)** -- for building and querying the semantic graph of each conversation.
- **Type_Safe Schema System** -- to define all data models (call records, transcript segments, analysis results, alerts) with strict types, ensuring consistency and traceability across the pipeline.
- **MyFeeds.ai LETS Pipeline** -- an architectural pattern (Load-Extract-Transform-Save) that guides the data flow in deterministic, debuggable steps.
- **Persona Modeling and Scoring** -- a layer that infers the "persona" or likely intent of the caller (legitimate customer vs. potential fraudster) and scores threat likelihood based on how the

conversation aligns with known patterns.

Let's discuss each in the context of Voice2SIEM:

Cache Service for Data Ingestion and Storage: The Cache Service is a lightweight, fast key-value store accessible via API (built with FastAPI and OSBot-FastAPI extensions). We use it as the glue between pipeline stages. For example, when a call audio is received, it can be stored as a binary blob in the Cache (under some unique key like `call/123/audio`). When the transcription step runs (as a microservice or job), it pulls the audio from Cache, produces a transcript object, and saves that back to the Cache (e.g. under `call/123/transcript`). Similarly, analysis stages store their outputs (tone analysis, intent extraction results, etc.) in the Cache with versioning. The Cache Service essentially provides a **central state repository** so that each stage of the pipeline is decoupled (they communicate by reading/writing data via the Cache API). Because it's backed by a fast datastore and supports JSON natively, it's ideal for persisting the conversation data at each step. This also means every intermediate artifact is saved -- enabling **replay, auditing, and debugging**. If an alert is raised on a call, we have the full chain of data (audio -> transcript -> graph -> score) stored for forensic analysis or model improvement.

Transcription and Analysis (Open-Source Tools): For transcription, one could use an open-source ASR engine. A leading choice is **OpenAI Whisper** (which has a high-accuracy model that can run on-premise GPUs or even on CPU for smaller models). There are also others like Kaldi or Vosk. In our blueprint, we'll assume using Whisper for speech-to-text, integrated into the pipeline as a service (perhaps a container that the Cache Service can call, or an offline batch process that writes results to Cache). For text analysis (sentiment, intent), we can leverage open-source NLP libraries or models: for sentiment, models from HuggingFace (e.g. a RoBERTa sentiment classifier) can be used; for keyword/entity extraction, spaCy or transformers can identify entities; for intent, one might train a simple classifier or use an LLM with prompts. The **persona modeling** can even utilize a **large language model** in a controlled way: for instance, use an LLM to parse the transcript and fill in a structured **"CallAnalysis" JSON schema** with fields like `suspected_attack_vector`, `caller_persona`, `important_entities`. By defining the schema and validating it (with Type_Safe classes), we ensure the LLM's output is structured and can be parsed deterministically (a technique proven in Dinis's MyFeeds.ai project, where LLMs populate predefined JSON schemas)[5]. Each of these analysis steps writes its JSON results to the Cache. This approach means even if we use advanced AI (LLM) for analysis, we **capture its reasoning in data form**, rather than a black-box judgment.

MGraph-DB for Semantic Graph Construction: Once the transcript and initial analyses are done, we consolidate the information into a **semantic graph** using MGraph-DB. MGraph-DB is an in-memory graph database optimized for JSON and Python usage[6]. We create nodes for key elements of the call: e.g. a node for the *Caller*, a node for the *Agent*, nodes for each *Utterance* (with properties like timestamp, sentiment, speaker), nodes for important *Entities* (like *Account* or *PIN Code* if they were mentioned), and perhaps nodes for *Intents* or *Requests* (like *ResetPasswordAction*). Then we add edges to connect these: *Caller* --(speaks)→ *Utterance1*; *Utterance1* --(intent)→ *ResetPasswordAction*; *Utterance2* --(sentiment)→ *Angry*; *Caller* --(provides)→ *AccountNumberEntity*, etc. The exact schema of the graph can evolve, but the idea is to create a rich representation that can be traversed to answer questions like "Did the caller provide credentials?" or "How did the agent respond after the

caller got angry?". MGraph-DB, being type-safe and in-memory, allows us to quickly build and query this graph within a Python service. Its **type-safe nature** ensures we only create valid node/edge types as defined in our schema (catching mistakes early)[7][8]. We can also serialize this graph to JSON for storage or debugging, since MGraph-DB supports JSON persistence.

Type_Safe Schemas and Data Models: All data entities in this pipeline are defined as classes using the **Type_Safe** system (from OSBot). For example, we might define `class Transcript(Type_Safe): ...` with fields for call_id, full_text, segments, etc., or `class Utterance(Type_Safe): speaker, text, timestamp, sentiment`. By using Type_Safe, we get runtime-checked, self-documenting data structures that can seamlessly convert to/from JSON[8]. This means when we pass data between services (or even within the graph DB), we do so in a structured manner. It also aids transparency: each piece of data can be logged or inspected with confidence in its format. The Type_Safe schema definitions essentially act as the **contract** for each pipeline stage's input/output. Moreover, this system helps with auditability: for instance, a `SecurityAlert` class might require certain fields (e.g. reason, score, timestamp, evidence graph reference), ensuring no alert is created without sufficient data. Sharing these schema classes between the pipeline components (the Cache service, the analysis code, the graph builder, etc.) guarantees consistency -- similar to how the client-server model in OSBot shares schemas to have a single source of truth[8].

LETS Pipeline Orchestration: We adopt the **LETS (Load-Extract-Transform-Save)** architecture to manage the pipeline execution in clear steps[9][10]. Here's how it maps to Voice2SIEM: - **Load:** Bring in the raw data (audio file) and save it unmodified. In practice, when a call audio arrives, we "load" it by storing it in the Cache (this is analogous to saving the raw RSS feed in MyFeeds.ai's pipeline[11]). This ensures the exact original audio is preserved. - **Extract:** Derive initial structured data from the raw input. This would be the transcription step (audio -> text) and perhaps parsing the text into structured dialogues. The transcript (with speaker turns) is saved as a JSON in the Cache. We might also extract other low-level info, like a timeline of who spoke when, or a list of detected keywords. The key is this stage is about structuring the raw audio into data we can work with (similar to how MyFeeds extracted article JSON from raw RSS)[12]. - **Transform:** Perform higher-level transformations to enrich the data with semantics and insights. In our case, several sub-stages of Transform happen: sentiment analysis, intent detection, building the conversation graph, and scoring the risk. Each of these can be its own Transform step that takes the output of Extract or a previous transform and produces a new artifact. For example, "Transform 1" might take the Transcript and produce a **SentimentTimeline** object (list of utterances with sentiment tags). "Transform 2" might take Transcript + SentimentTimeline and produce the **ConversationGraph** (using MGraph-DB). "Transform 3" might take the ConversationGraph and produce a **ThreatHypothesis** (which encapsulates the potential attack vectors identified, e.g. "caller impersonating CEO scenario"). Each of these transforms saves its output to the Cache (and could be an API call or microservice in the implementation). By chaining multiple fine-grained transformations, we make the system easier to debug and extend[5]. Crucially, we **persist after each transform**, so intermediate data is always available for inspection. If the final decision seems wrong, we can go back to see, for example, what the conversation graph looked like or what the sentiment analysis found, to pinpoint which stage misinterpreted the data. - **Save:** In LETS, every stage saves its output, but finally we also **save the final results to their destination**. In this context,

the ultimate "save" is to log the security event (if any) and related data in a permanent store. The Cache service could serve in this capacity (it might append the alert to an "alerts" collection in its storage). Or we might output it to a SIEM system or even just a JSON file repository. The Save step here emphasizes versioning and traceability: we label the final outputs with version IDs and timestamps. If six months later we want to audit why a certain call was flagged, we can retrieve the exact data versions that led to it. This strong provenance tracking is a core benefit of the LETS approach[13] -- every decision can be traced back through the pipeline with records of each intermediate state.

Using LETS in this pipeline means the solution is **fully reproducible and debuggable**. If an error is found in our analysis logic, we can re-run that step on the saved inputs to test a fix. It also means we can gradually improve each stage (swap out a model, fine-tune a rule) and have confidence how it impacts the end result, since we can replay past calls through the pipeline offline if needed.

Persona Modeling and Threat Scoring: One of the powerful ideas we introduce is modeling the personas involved in the call -- especially the caller, who could be genuine or malicious. Over time, the system can build profiles of legitimate customer behavior versus known attacker tactics. For instance, a legitimate customer might answer verification questions slowly but correctly, exhibit frustration *after* repeated problems, but will comply with security checks. In contrast, an attacker persona might **either** act overly authoritative ("I'm in a huge hurry, I'm the CTO, just reset my password now") or overly distressed ("I've been locked out and this is an emergency, please help, I can't answer all these questions!"). By creating a library of these **persona archetypes** (which could be informed by real fraud cases), the system can compare a live call's features to the personas. The **persona modeling system** (inspired by MyFeeds.ai's approach of creating persona interest graphs[12]) could represent each archetype as a set of expected behaviors or markers. For example, an *"Impersonator Executive" persona* might have markers like: high authority tone, expresses urgency, drops names of company executives, attempts to bypass normal process. A *"Distraught User" persona* might: sound panicked, mention personal crises, push the agent to bend rules out of sympathy. These can be encoded in a data structure (even as a graph or just a profile object).

During the Transform stage, we take the conversation data and try to **match it against these persona profiles**. This could be done with rules or possibly an ML model that classifies the call into one of several personas. If a strong match to a malicious persona is found, that heavily influences the threat score. It's not a binary thing -- we might see partial matches to multiple personas -- so the scoring system can weigh various factors. For example: *caller matched 80% of "Impersonator Executive" traits + caller requested password reset (a high-risk action) + call came from an unusual phone number (metadata anomaly)* -- combine these to yield a, say, 95/100 risk score, clearly over the threshold.

Graph-Based Detection of Patterns: Because we have the conversation as a graph in MGraph-DB, we can also directly apply graph algorithms or queries to spot suspicious patterns. For example, we could query for a subgraph where: a *Caller node* is connected to an *Account node* via a *provided credential* edge, but the *Agent node* is connected to a *Verification step* that failed. This pattern (caller provided some info, failed verification, but is still pushing) could be indicative of fraud. Another pattern: the time between *Caller's request* and *Agent's action* is very short because the

caller pressured the agent (could be calculated via timestamps in the graph). Or a sequence pattern: *Caller asks innocuous question -> builds rapport -> then asks for a sensitive action.* In the graph, that might appear as three utterances where the first has neutral intent, second is small talk, third is high-risk intent; the presence of that sequence can be automatically flagged by traversing the graph or by converting the conversation into a sequence of intent labels and running it through a sequence pattern matcher. We can even search across calls: if the same caller (or same voice print, if we did voice analysis) appears in multiple incidents, the graph of their interactions across calls could expose a fraud campaign.

Examples of Detectable Social Engineering Signs: To illustrate, here are a few scenarios and how our system would catch them: - *Sequence Pattern:* An attacker often follows a script. For instance, "Friendly introduction" -> "Problem statement" -> "Urgent request with flattery/threat". A normal call might not have such a scripted progression. Our semantic extraction would label each segment (greeting, verification, request, etc.), and the conversation graph would show an abnormal transition from a casual chat to a critical demand. If our rules know this pattern (perhaps from past examples), the system flags it. For example, *"caller suddenly transitioned from calm to urgent while requesting a policy exception"* could be a rule derived from sequence analysis. - *Stress Indicators:* Suppose the caller's voice analysis shows **elevated stress or anger whenever security protocols are mentioned** (like the agent asking to verify identity causes the caller to raise their voice or heart rate if that could be measured). The sentiment timeline might show spikes of negative sentiment aligned with those moments. This is a red flag -- a genuine user might be annoyed at verification but wouldn't typically become aggressive; an attacker often does when impeded. The system would note *high emotional variance correlated with security steps* as a risk indicator. - *Anomalous Metadata:* Beyond content, contextual metadata can be telling. If the call came in at 3 AM local time from an overseas IP (for VoIP) or the phone number isn't one normally associated with the customer's account, those could be captured in the data model. Perhaps the account's profile says typical call time is daytime and this is highly out of pattern. Or the caller claims to be in one city but the telephony data suggests otherwise. Our pipeline can ingest such metadata at Load time (if available) and attach it to the graph (e.g. a node for "CallOrigin" with attributes). Any anomaly here (especially in combination with suspicious dialog) increases the score.

After all these analyses, the open-source system arrives at a **ThreatLikelihood score** for the call, along with an explanation of why. Thanks to the structured approach, this explanation can be very specific: e.g. *"Alert: 95% likely social engineering. Detected persona: Impersonating Executive. Evidence: Caller insisted on urgent action (ResetPassword) with high anger (sentiment -0.85) after failing verification twice; call origin UK London, but user account holder is in USA; sequence matched known fraud pattern #3."* This kind of rich explanation is what a graph-based and type-safe pipeline can provide. It's **transparent and auditable**, unlike a monolithic "AI black box" solution. Every piece of that explanation links to an artifact in our system (transcript, sentiment value, graph pattern, metadata record). The security team can drill down to confirm each fact (because the data is in the Cache/graph) -- building trust in the system's verdict. In fact, the intermediate outputs themselves can be used for auditor training or improving processes (maybe the company discovers certain verification steps often cause false alarms, and they adjust policy).

Finally, the open-source pipeline would emit the event just like the cloud one -- perhaps writing an entry to the Cache's event store or sending a message to a SIEM connector. The major difference is that with open components, the organization can **own the solution end-to-end**: data stays on their servers, models can be customized, and the logic can be adapted to their unique needs or expanded (for example, integrating a voice biometric check if available, or linking to a database of known fraud caller IDs). All core components (Cache Service, MGraph-DB, OSBot Type_Safe, etc.) are open-source (Apache 2.0 or similar licenses) developed by Dinis Cruz and community, meaning there's no license cost and one can contribute improvements. MGraph-DB's design for high-performance in-memory operation ensures even complex graph queries or large calls can be handled efficiently[6]. The Type_Safe framework makes sure our system's APIs and data are robust and error-checked at development time, reducing runtime surprises[8].

Conclusion and Call to Action

Project Voice2SIEM demonstrates that turning customer support conversations into actionable security intelligence is not only possible -- it's achievable today with open technology and a bit of integration work. By combining voice-to-text, AI-driven analysis, and graph-based correlations, we can shine a light on what has traditionally been a blind spot in security monitoring (the content of phone calls). Importantly, the approach we presented emphasizes **trust, transparency, and auditability**. Every decision the system makes is backed by data artifacts and clear rules, so security teams can trust the alerts and verify why an alert was raised by tracing through the stored intermediate states[13]. This is in stark contrast to opaque "AI magic" solutions; here, we're effectively opening the black box and making it a glass box.

We also made a point to ensure the system respects the practical constraints of real call centers: it doesn't disrupt the live call flow, it works in near real-time, and it produces alerts quickly enough to matter. Whether implemented with cloud serverless components or a fully open-source stack, the blueprint is modular and flexible. Companies can start small -- maybe begin by just transcribing calls and doing sentiment analysis as a pilot -- and gradually build up the full pipeline as confidence grows. Because each piece is replaceable, improvements in AI (say a better speech recognition model or a new NLP technique) can be plugged in without redesigning the whole system.

The authors (Dinis Cruz and the ChatGPT Deep Research team) are releasing this as an open blueprint with **no commercial agenda** -- our intent is purely to advance the state of the art in security monitoring and inspire others to build upon it. All the mentioned open-source tools (Cache Service, MGraph-DB, OSBot Type_Safe, etc.) are available in open repositories, and we invite readers to explore them, contribute, or adapt them to their own projects. We believe this kind of system would be incredibly valuable if deployed broadly: imagine a world where every help desk call or IT support call is quietly analyzed for potential fraud, providing a safety net for human agents who might be socially engineered. Many high-profile breaches could have been mitigated or even prevented if such technology were in place to catch the tell-tale signs of a con artist on the phone.

Call to Action: If you find this project intriguing, consider contributing to its development or trying it out in your environment. You could start by using AWS's AI services to get quick wins on call

analysis, or if you're more experimentally minded, deploy the open-source components and run some recorded calls through it to see what insights surface. Share your findings, build custom persona profiles that fit your industry, and help refine the detection logic. Since this is an open effort, improvements by one can benefit many. Ultimately, securing the "human layer" of support interactions is a shared challenge -- let's collaboratively turn the tide against voice-based social engineering. **Voice2SIEM can be a community-driven shield**, and we welcome you to join us in building it. Together, we can make customer support channels safer through transparency, open tech, and a healthy dose of innovation.

[1] New voice fraud cloning techniques expose a vulnerability of call centers | TSYS

https://www.tsys.com/insights/2024/10/22/new-voice-fraud-cloning-techniques-expose-a-vulnerability-of-call-centers

[2] [4] Amazon Transcribe Call Analytics | Transcripts & Insights | AWS

https://aws.amazon.com/transcribe/call-analytics/

[3] GitHub - aws-samples/amazon-transcribe-live-call-analytics: Amazon Transcribe Live Call Analytics (LCA) Sample Solution

https://github.com/aws-samples/amazon-transcribe-live-call-analytics

[5] [9] [10] [11] [12] [13] lets__load-extract-transform-save__a-deterministic-and-debuggable-data-pipeline_architecture.md

https://github.com/DinisCruz/docs.diniscruz.ai/blob/6e0e278d80d0ea5cdb125f998590a3b2b248021d/docs/2025/05/27/lets__load-extract-transform-save__a-deterministic-and-debuggable-data-pipeline_architecture.md

[6] [7] mgraph-db · PyPI

https://pypi.org/project/mgraph-db/

[8] v0.26.1__developing-fastapi-service-clients.md

https://github.com/owasp-sbot/OSBot-Fast-API/blob/f3bc57390c988d6cce2d9bd5e16d9618beb00868/docs/dev/briefs/v0.26.1__developing-fastapi-service-clients.md