Agentic Earnings Call Tone Analysis Platform on Google Cloud

Vertex AI + LangGraph + Ontology-Driven Multi-Agent System

Lucas Kim

February 25, 2026

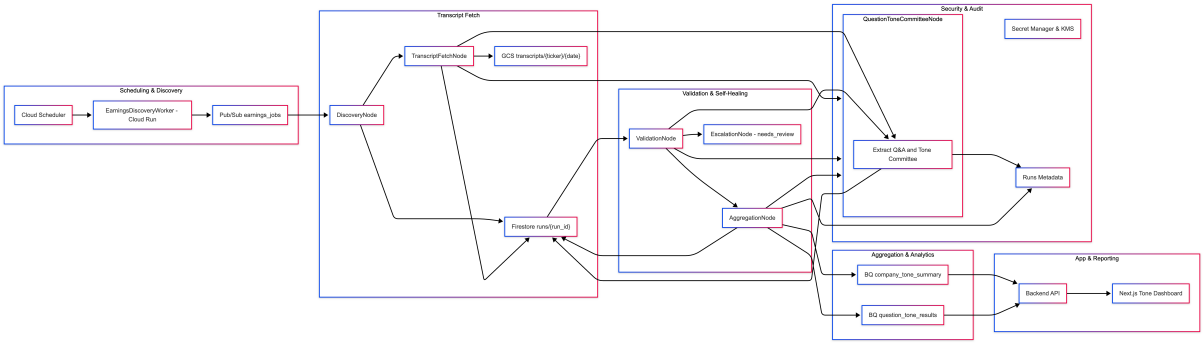

Figure 1: Agentic architecture for earnings call tone analysis.

1 Executive Summary

This proposal describes a production-grade, agentic architecture for an earnings call tone analysis platform targeted at NYSE and NASDAQ listed companies. The system continuously discovers companies that reported earnings in the last week, retrieves their earnings call transcripts, identifies Q&A segments, and computes a robust "analyst tone" metric for each company.

Unlike a traditional ETL pipeline, the proposed design is a *stateful, self-healing multi-agent system*. It uses:

- Google Cloud Run, Pub/Sub, Firestore, and BigQuery for scalable, observable infrastructure,
- Vertex AI with Gemini 1.5 Flash and context caching for cost-efficient LLM inference [1],
- LangGraph-style workflows for reliable agent orchestration and retry logic [4],
- An explicit tone ontology and a three-agent "tone committee" (Praise, Skepticism, Neutral) for explainable classification [5, 6].

The result is a hedge-fund–grade platform: every tone metric is traceable to raw transcripts, model versions, and prompt templates, satisfying both research needs and audit requirements.

2 Functional Requirements

The system is designed to satisfy the challenge requirements:

R1 Track companies that reported earnings in the last week on NYSE/NASDAQ.

R2 Retrieve their earnings call transcripts from a reliable source.

R3 Identify the Q&A section of each earnings call.

R4 Detect whether analyst questions reflect praise/support, skepticism/disappointment, or neutral tone.

R5 Produce structured outputs summarizing last week's calls and a measurable tone metric per company.

R6 Operate continuously: new day, new earnings calls, automatic updates.

Non-functional constraints:

- High concurrency during peak earnings weeks.
- Cost-efficiency via token reuse and context caching [2].
- Production-grade security: KMS, Secret Manager, IAM, audit logging.
- Agentic reliability: explicit state management and self-correction loops.

3 Architecture

3.1 Component Overview

At a high level, the system consists of:

- **Scheduling & Discovery**: Cloud Scheduler and an earnings discovery worker on Cloud Run publish company–date pairs to a Pub/Sub topic.
- **Agent Orchestration**: A LangGraph-style workflow orchestrated as Cloud Run jobs manages state transitions in Firestore ("runs").
- **Transcript Ingestion**: A TranscriptFetch node calls a transcript API provider and persists raw artifacts in Cloud Storage.
- **QuestionToneCommittee Node**: A consolidated node that (i) extracts the Q&A section and question–answer pairs and (ii) runs a three-agent tone committee using Gemini 1.5 Flash with context caching [1].
- **Validation & Self-Healing**: A Validation node checks coverage and consistency and may trigger retries with alternative parsing prompts.
- **Aggregation & Analytics**: Per-question and per-company metrics are written to BigQuery for reporting and research.
- **App & Reporting**: A Next.js frontend and backend API expose dashboards and programmatic access to the metrics.
- **Security & Lineage**: Secret Manager, KMS, Cloud Logging, and Vertex AI Metadata provide key management, audit logs, and full lineage from metrics back to raw transcripts [7].

3.2 Architecture Diagram Walkthrough

Figure 1 shows the end-to-end agentic architecture. In this section we describe, in detail, the responsibility and behavior of each subsystem and how data and control flow between them.

1. Scheduling & Discovery (S)

- **Cloud Scheduler (A)**
  Cloud Scheduler triggers the earnings discovery workflow at a fixed cadence (typically once per day, but configurable). It is the only time-based entrypoint to the system: everything downstream is event-driven. The schedule can be aligned with U.S. market close or with transcript provider SLAs.
- **EarningsDiscoveryWorker – Cloud Run (B)**
  The *EarningsDiscoveryWorker* is a stateless Cloud Run service that performs the following steps:
  1. Query an earnings calendar API restricted to NYSE and NASDAQ tickers over a lookback window (e.g., the last 7 days).
  2. Normalize each earnings event to a canonical job payload: `{ticker, call_date, exchange, source_metadata}`.

3. De-duplicate against recent jobs (to avoid reprocessing the same call) by checking a small cache table or Firestore collection.
4. Publish one message per distinct call to the **Pub/Sub `earnings_jobs` topic (C)**.

This component is purely I/O bound and scales horizontally: more earnings events simply translate to more Pub/Sub messages.

- **Pub/Sub `earnings_jobs` (C)**
  The `earnings_jobs` topic is the buffer between the external world and the agentic system. Each message represents a unit of work: "process the earnings call for ticker T on date D". Pub/Sub provides:
  - decoupling between discovery and processing,
  - automatic backpressure during peak earnings weeks,
  - at-least-once delivery semantics, which the downstream workflow handles idempotently.

2. Run Initialization and State (D, E)

- **DiscoveryNode (D)**
  Each Pub/Sub message triggers a LangGraph "run" starting at the *DiscoveryNode*. This node is responsible for:
  - creating a unique `run_id` for the call (e.g., a UUID),
  - initializing the *AgentState* with `ticker`, `call_date`, and default fields (`retry_count = 0`, `status = "pending"`),
  - persisting this initial state into Firestore.

  At this point, the workflow has a durable handle on the work it is about to perform.
- **Firestore `runs/{run_id}` (E)**
  Firestore acts as the *hot state store* for the agent. For each earnings call, we store a single document:

```
runs/{run_id} = {
  ticker: "...",
  call_date: "...",
  transcript_uri: null,
  qa_pairs: [],
  analysis_results: [],
  retry_count: 0,
  status: "pending",
  error_reason: null,
  company_metrics: {}
}
```

  All subsequent nodes (TranscriptFetch, QuestionToneCommittee, Validation, Aggregation, Escalation) read and update this document. This design externalizes state from any particular container instance and makes the workflow robust to Cloud Run restarts or scaling events.

3. Transcript Fetch (T1)

- **TranscriptFetchNode (F)**
  The *TranscriptFetchNode* is a Cloud Run service (or LangGraph node) that:
  1. Reads `run_id`, `ticker`, and `call_date` from Firestore.
  2. Calls the configured transcript provider API (e.g., FMP, Tradefeeds, or API Ninjas) to fetch the full earnings call transcript for that date.
  3. Normalizes the response into a canonical structure:
     - raw text or HTML,
     - per-utterance speaker metadata when available (management vs analyst vs operator),

- call-level meta (time, language, etc.).

4. Writes the normalized artifact to Cloud Storage (G) under a deterministic path: `gs://.../transcripts/{ticker}/{call_date}.json`.
5. Updates the Firestore state (E) with:
   - `transcript_uri`,
   - `status = "running"`,
   - any provider metadata useful for later debugging.

If the API call fails or the transcript is missing, the node marks the run as `"failed"` and sets `error_reason`, which can be surfaced in internal dashboards.

- **GCS `transcripts/{ticker}/{date}` (G)**
  Cloud Storage holds the raw, immutable transcript artifacts. This is the ground truth for any re-processing:
  - The parsing and tone classification logic can evolve over time, but the original transcript remains unchanged.
  - Downstream audio or multimodal research can reuse this corpus.

4. QuestionToneCommitteeNode (Q)

- **Extract Q&A and Tone Committee (H)**
  This is the core analysis node and the main consumer of Gemini 1.5 Flash. It performs *two tasks in a single logical step*:
  1. **Q&A Extraction**
     - Load the transcript from GCS using `transcript_uri`.
     - Use a combination of:
       * deterministic rules (speaker labels, operator phrases like "we will now open the line for questions"), and
       * LLM-based parsing

       to identify the Q&A section and segment it into question–answer pairs.
     - Each pair records: `question_text`, `analyst_name`, `analyst_firm` (if available), `answer_text`, and any structural metadata.
  2. **Tone Committee Inference**
     - Load the *Tone Ontology* and committee description (PraiseAgent, SkepticAgent, NeutralAgent, AggregatorAgent) into the prompt.
     - Use Vertex AI context caching to store the ontology, rubric, and examples once, and reuse them across calls. The per-call input focuses on the extracted Q&A pairs, which yields significant token savings [1, 2, 3].
     - For each question, simulate the three agents and the aggregator in a *single* Gemini call, producing:
       * `praise_score`, `skeptic_score`, `neutrality_score`,
       * `final_label` $\in$ {PraiseSupport, SkepticismDisappointment, Neutral},
       * scalar `tone_score` and a `disagreement` flag,
       * textual rationales and evidence spans.
     - Update the Firestore state with:
       * `qa_pairs`: structured question–answer objects,
       * `analysis_results`: one record per question with all scores and labels.

  By merging extraction and tone classification into a single node, we avoid repeated LLM passes over the same context and reduce both latency and cost.

5. Validation & Self-Healing (V)

- **ValidationNode (I)**
  The Validation node reads the updated Firestore state and validates the analysis from several angles:

- **Coverage**: Did we extract a reasonable number of questions for a typical call in this sector?
- **Structural consistency**: Are analysts correctly identified as questioners, and management as answerers?
- **Committee disagreement**: Is the fraction of `disagreement = true` questions below a configured threshold?

Based on these checks:

- If all criteria pass, the node transitions to the Aggregation node (J).
- If quality is insufficient and `retry_count < N`, the node increments `retry_count` and sends control back to the QuestionToneCommittee node (H) with a modified strategy (e.g., more aggressive few-shot examples, different segmentation rules).
- If validation fails repeatedly, the node routes the run to Escalation (K) and sets `status = "needs_review"`.

- **EscalationNode – needs_review (K)**
  Runs that are structurally unusual (e.g., non-standard transcript format, very high committee disagreement) end up here. This node:
  - marks the Firestore document as `"needs_review"`,
  - optionally publishes a message to an internal analyst review queue or writes to a dedicated BigQuery table for manual triage.

  This ensures that dubious data does not silently pollute production metrics.

6. Aggregation & Analytics (A2)

- **AggregationNode (J)**
  Once validation passes, the Aggregation node converts per-question results into per-company metrics:
  - For each run (ticker, call_date), compute:
    - `support_ratio` = #questions labeled PraiseSupport / total,
    - `skeptic_ratio` = #questions labeled SkepticismDisappointment / total,
    - `neutral_ratio`,
    - `tone_index` (e.g., support_ratio − skeptic_ratio),
    - `num_questions`,
    - `high_disagreement_ratio`.
  - Write one row per question into **BQ `question_tone_results`** (M), including:
    - question text, label, scores, disagreement flag,
    - model name and version, prompt template version,
    - context cache identifier, transcript URI, run ID.
  - Write one row per call into **BQ `company_tone_summary`** (L) with the aggregated metrics and the run ID.
  - Update the Firestore state (E) with `company_metrics` and set `status = "done"`.
- **BQ `company_tone_summary` (L)**
  This table is the main entrypoint for the dashboard: one record per company per earnings call, with precomputed ratios and indices suitable for time-series visualisation and cross-sectional screening.
- **BQ `question_tone_results` (M)**
  This detailed table supports:
  - forensic analysis ("why did the system think this call was skeptical?"),
  - research use cases (linking tone to returns or surprise),
  - training data for future models (fine-tuning or calibration).

7. App & Reporting (UI)

- **Backend API (N)**

A backend service (FastAPI or Node.js on Cloud Run) provides authenticated endpoints for:
  - fetching company-level tone metrics for arbitrary time windows,
  - drilling down into question-level records,
  - exporting data slices for offline research.

  The API reads from the BigQuery tables (L, M) and enforces any access control or rate limits required for production use.
- **Next.js Tone Dashboard (O)**
  The dashboard is a Next.js application that:
  - visualizes weekly tone distributions across tickers,
  - highlights companies with unusually high skepticism or disagreement,
  - allows users to click into a call and inspect individual analyst questions and their tone classification.

  Because aggregation is done ahead of time, the dashboard can serve institutional users with low latency, even when querying large universes.

8. Security & Audit (Sec)

- **Secret Manager & KMS (P)**
  All sensitive credentials (transcript APIs, database passwords, Gemini keys) are stored in Secret Manager and encrypted with Cloud KMS. Only the specific Cloud Run services that require a given secret have IAM permission to access it.
- **Cloud Logging (Q)**
  Each major node (F, H, I, J) logs structured events:
  - run ID, node name, timestamps, durations,
  - counts of extracted questions, counts of each tone label,
  - validation outcomes and retry decisions.

  These logs support both operational monitoring (SLOs, error rates) and compliance audits.
- **Runs Metadata (R)**
  In addition to per-question fields in BigQuery, we maintain a *runs metadata* store (either Vertex AI Metadata or a dedicated BigQuery table) that tracks:
  - model name and version,
  - prompt template version,
  - context cache ID used,
  - configuration flags (e.g., which extraction strategy was active),
  - creation time and completion time.

  This enables full lineage: any company-level tone metric can be traced back to the exact transcript, model, and prompt configuration that produced it, which is essential for regulated financial environments.

4 Tone Ontology

A key design choice is to treat tone not as a flat label, but as an explicit ontology. We define the following top-level concept:

4.1 Categories

ToneCategory is an enumeration with three values, aligned with the challenge:

- **PraiseSupport**: Questions that express support, confidence, or positive reinforcement regarding management's execution, strategy, or guidance.
- **SkepticismDisappointment**: Questions that express doubt, disappointment, or challenge around sustainability of performance, risk exposure, or credibility of disclosures.

- **Neutral**: Questions that are primarily information-seeking or clarificatory, with minimal directional emotional content.

4.2 Attributes and Cues

Each ToneCategory is enriched with:

- Canonical lexical cues and phrasings.
- Question intent patterns: clarification vs. challenge vs. endorsement.
- Example Q&A snippets from historical transcripts (for few-shot prompting).
- A list of "ritual politeness" phrases that must be ignored (e.g., "thanks for taking my question", "congrats on the quarter").

We maintain this ontology as a versioned artifact (e.g., `tone_ontology_v1.yaml`) loaded into the LLM prompt and cached via Vertex AI context caching to avoid repeated token costs across calls [1, 2].

5 Multi-Agent Tone Committee

5.1 Agent Roles

All agents share the same tone ontology (Section 4), but each adopts a different perspective:

PraiseAgent (Support Analyst)

- Persona: A supportive sell-side analyst focusing on positive signals.
- Responsibility: Estimate how strongly a question reflects PraiseSupport tone, ignoring neutral or skeptical elements.
- Output per question:
  - `praise_score` $\in [0, 1]$,
  - evidence text spans,
  - a short natural-language rationale.

SkepticAgent (Bearish Risk Analyst)

- Persona: An analyst specializing in downside risk, accounting quality, and guidance credibility.
- Responsibility: Estimate the degree of SkepticismDisappointment.
- Output per question:
  - `skeptic_score` $\in [0, 1]$,
  - list of risk vectors (e.g., demand, margin, regulation, execution),
  - rationale.

NeutralAgent (Fact-Gathering Analyst)

- Persona: A quant/strategist seeking information with minimal directional view.
- Responsibility: Assess Neutral tone, especially when a question is purely clarificatory.
- Output:
  - `neutrality_score` $\in [0, 1]$,
  - rationale.

5.2 AggregatorAgent

A meta-agent aggregates the three perspectives into a final label and score.

For each question q:

$$
\begin{aligned}
s_p(q) &= \text{praise\_score}, \\
s_s(q) &= \text{skeptic\_score}, \\
s_n(q) &= \text{neutrality\_score}.
\end{aligned}
$$

We define:

$$
\begin{aligned}
s_{\max}(q) &= \max\{s_p(q), s_s(q), s_n(q)\}, \\
\text{label}(q) &= \begin{cases} \text{Neutral}, & \text{if } s_{\max}(q) < \tau, \\ \arg\max\{s_p(q), s_s(q), s_n(q)\}, & \text{otherwise}, \end{cases}
\end{aligned}
$$

where $\tau \in [0, 1]$ is a neutrality threshold. A scalar tone index can be defined as

$$
\text{tone\_score}(q) = s_p(q) - s_s(q),
$$

which maps Praise-dominant questions to positive values and Skeptic-dominant questions to negative values.

We also define a disagreement flag:

$$
\text{disagreement}(q) = \begin{cases} 1, & \text{if } \max\{s_p, s_s, s_n\} - \min\{s_p, s_s, s_n\} > \delta, \\ 0, & \text{otherwise}, \end{cases}
$$

to identify questions where agents strongly disagree, which can be routed for manual review.

5.3 Execution Pattern

For cost and latency reasons, we implement the committee within a *single* Gemini 1.5 Flash call:

- The system prompt describes the three roles, the ontology, and the aggregation rules.
- The model receives a batch of Q&A pairs and outputs a JSON object of the form:

```
{
  "questions": [
    {
      "id": "...",
      "praise": {...},
      "skeptic": {...},
      "neutral": {...},
      "final": {
        "label": "SkepticismDisappointment",
        "tone_score": -0.72,
        "disagreement": false
      }
    },
    ...
  ]
}
```

The ontology, examples, and rubric are cached via Vertex AI context caching to reduce repeated input tokens for each call [1, 2].

6 State Model and LangGraph Workflow

6.1 Agent State

We model the workflow state as an explicit typed dictionary:

```
from typing import TypedDict, List, Dict, Optional

class AgentState(TypedDict):
   run_id: str
   ticker: str
   call_date: str
   transcript_uri: str

   qa_pairs: List[Dict] # extracted Q&A pairs
   analysis_results: List[Dict] # per-question scores and labels
   retry_count: int

   status: str # "pending" | "running" | "retrying" |
                          # "failed" | "needs_review" | "done"
   error_reason: Optional[str]

   company_metrics: Dict # aggregated metrics
```

This state is persisted in Firestore under `runs/{run_id}`, providing a single source of truth for the workflow.

6.2 Workflow Nodes

Using a LangGraph-style workflow [4], we define the following nodes:

DiscoveryNode

- Reads a job from Pub/Sub (ticker, call date).
- Initializes `AgentState` in Firestore (status = "pending").

TranscriptFetchNode

- Calls the transcript provider API for the given ticker and date.
- Stores the raw transcript in GCS, sets `transcript_uri` in state.
- On failure, sets status to "failed" and records `error_reason`.

QuestionToneCommitteeNode

- Loads the transcript from GCS.
- Extracts the Q&A section and question–answer pairs (using a mix of deterministic rules and LLM parsing).
- Invokes Gemini 1.5 Flash with the cached financial tone rubric to run the three-agent committee on all questions in a single call.
- Writes `qa_pairs` and `analysis_results` back into state.

ValidationNode

- Validates basic quality criteria:
  - Minimum number of questions,
  - Reasonable mapping of speakers (analysts vs management),
  - Fraction of high-disagreement questions below a threshold.
- If validation fails and `retry_count < N`, it increments `retry_count` and returns control to the QuestionToneCommittee node with an alternative prompt strategy.
- If validation fails repeatedly, it sets status to "needs_review" and routes to Escalation.

AggregationNode

- Aggregates per-question results into company-level metrics, e.g.:
  - support ratio,
  - skeptic ratio,
  - neutral ratio,
  - tone index,
  - high-disagreement ratio.
- Writes:
  - per-question rows to `BQ question_tone_results`,
  - per-company summary to `BQ company_tone_summary`.
- Updates Firestore state to status = "done".

EscalationNode

- Marks runs that require human review (e.g., atypical transcript structure, persistent disagreement).
- Optionally pushes a message into an internal review queue.

7 Data Storage and Access Patterns

7.1 Firestore: Hot State

Firestore holds the agent state documents:

- Fast reads/writes for concurrent runs (thousands of earnings calls).
- Natural fit for LangGraph-style state transitions where multiple nodes update the same logical run.

7.2 BigQuery: Analytics and Research

BigQuery stores:

- Per-question results with full metadata: tone scores, labels, disagreement flags, model/prompt/context identifiers.
- Per-company per-call summary metrics for dashboards and time series analysis.

Firestore → BigQuery integration patterns are well-supported and scalable [7].

7.3 Cloud Storage

GCS holds:

- Raw transcript artifacts (PDF, HTML, JSON) per ticker and date.
- Optionally, normalized Q&A-only JSON for downstream audio or multimodal experiments.

8 Security, Privacy, and Lineage

8.1 Secrets and Access Control

- All external API keys (transcript providers), database credentials, and Gemini API keys are stored in Secret Manager.
- Access is controlled via IAM with least-privilege principles.
- At-rest encryption is managed via Cloud KMS.

8.2 Audit Logging

- All node invocations (TranscriptFetch, QuestionToneCommittee, Validation, Aggregation) log to Cloud Logging.
- Logs include run IDs, node names, timing, and summary statistics (but never raw PII).

8.3 Model Lineage

We record the following fields for each question-level record in BigQuery:

- `model_name`, `model_version`,
- `prompt_template_version`,
- `context_cache_id`,
- `transcript_uri`,
- `run_id`.

This enables full lineage from any company-level tone metric back to the raw transcript and the exact model configuration that produced it, which is crucial for model risk and audit in regulated financial environments.

9 Scalability and Cost Optimization

9.1 High-Concurrency Earnings Weeks

- Pub/Sub provides backpressure and decouples discovery from processing.
- Cloud Run services scale horizontally with concurrent runs and are stateless; state is externalized to Firestore and BigQuery.

9.2 Token and Latency Optimization

- Parsing and tone classification are consolidated into the QuestionToneCommittee node, avoiding duplicate LLM passes over the same context.
- Vertex AI context caching is used to store the financial tone rubric and examples once, and reuse them across calls, which can reduce input token cost by up to 4× according to Google Cloud documentation [1, 2, 3].

References

[1] Google Cloud, "Context caching overview — Generative AI on Vertex AI", 2026. https://docs.cloud.google.com/vertex-ai/generative-ai/docs/context-cache/context-cache-overview.

[2] Google Cloud Blog, "Vertex AI context caching", 2025. https://cloud.google.com/blog/products/ai-machine-learning/vertex-ai-context-caching.

[3] Google Cloud, "Use caching to make your LLM input up to 4 times cheaper. Vertex AI Context Caching with Gemini", 2024. https://www.youtube.com/watch?v=eAUIanQMJJg.

[4] LangChain, "LangGraph: Agent Orchestration Framework for Reliable AI Agents", 2024. https://www.langchain.com/langgraph.

[5] E. Cambria et al., "OntoSenticNet: A Commonsense Ontology for Sentiment Analysis", 2018. https://sentic.net/ontosenticnet.pdf.

[6] Z. Author et al., "SentiMM: A Multimodal Multi-Agent Framework for Sentiment Analysis", 2023. https://arxiv.org/pdf/2508.18108.pdf.

[7] Google Firebase, "Integrate with BigQuery — Firestore", 2026. https://firebase.google.com/docs/firestore/solutions/bigquery.