Design Overview

Earnings Call Claim Verification System — Design Overview

System Overview

- This is a modular, cloud-scalable system that automates the pipeline: data ingestion, claim extraction, verification, and presentation.
- It leverages large language models (LLMs) with retrieval-augmented reasoning (RAG) for flexible claim interpretation, but grounds their output in actual financial data through deterministic checks. Our architecture uses Python services (FastAPI for web interfaces) and containerization for cloud deployment.
- In practice, the system will fetch earnings transcripts and financial statements, parse them, extract any numeric claims (e.g. "revenue up 15% YoY"), and then compare each claim to the company's reported data. Discrepancies or overly "optimistic" framings will be flagged for review.
- This design draws on recent research and industry practice: analysts manually verify quantitative claims against official releases, and academic benchmarks like Fin-Fact and FINDVER highlight the challenge of automated financial claim verification. I replicate best-practices by combining direct numeric checks with LLM-based reasoning for nuance (e.g. non-GAAP adjustments) to maximize accuracy.

Data Pipeline and Sources

Transcripts — HuggingFace Dataset (Primary)

For rapid development and testing, I use the `Bose345/sp500_earnings_transcripts` dataset on HuggingFace, which contains 33,362 transcripts across 685 S&P 500 companies from 2005–2025. Each record includes `symbol`, `company_name`, `year`, `quarter`, `date`, `content` (full raw text), and `structured_content` (array of `{speaker, text}` objects — already speaker-segmented). This allows preloading transcripts without any API calls during development.[4]

Transcripts — Finnhub API (Secondary)

I plan to use the Finnhub Stock API as our secondary transcript source during scaling. Finnhub provides earnings call transcripts that are speaker-segmented — each utterance is tagged with the speaker name and role (CEO, CFO, Analyst, etc.), which is critical for attributing claims to specific executives. The transcripts include metadata (date, quarter, year, symbol) and structured dialogue segments.[1][2][3]

Financial Data — SEC EDGAR XBRL JSON API + edgartools (Primary)

For structured financial data (revenue, net income, EPS, margins, etc.), I use the SEC EDGAR XBRL API — the official, authoritative, free, unlimited-access source of US public company financials. The SEC provides key endpoints:[5]

- `data.sec.gov/api/xbrl/companyconcept/` — All XBRL disclosures for a single company + concept (e.g., Revenue, NetIncome) as JSON[6][5]
- `data.sec.gov/api/xbrl/companyfacts/` — All facts for a company in one JSON payload
- `data.sec.gov/submissions/` — Filing metadata, CIK lookups

To access this data ergonomically, I use the `edgartools` Python library:[7][8]

- AI-native with built-in MCP server for LLM integration and optimized text extraction
- Multi-period analysis via `MultiFinancials` for comparing quarters side-by-side[8]
- Returns pandas DataFrames for immediate computation
- Parses XBRL natively, handles tag mapping automatically[9]

Financial Data — Finnhub Basic Financials (Secondary)

Finnhub's `company_basic_financials` endpoint provides ~30 quarterly metrics (EPS, margins, ratios, book value, etc.) with 23+ years of history on the free tier. This serves as a quick-lookup cache for common verification targets without needing to parse full XBRL filings.[3]

Target Companies

#	Ticker	Company	Sector
1	AAPL	Apple	Technology
2	MSFT	Microsoft	Technology
3	AMZN	Amazon	E-commerce/Cloud
4	GOOGL	Alphabet	Technology
5	META	Meta Platforms	Social/AI
6	TSLA	Tesla	Auto/Energy
7	JPM	JPMorgan Chase	Banking
8	JNJ	Johnson & Johnson	Healthcare
9	WMT	Walmart	Retail
10	NVDA	NVIDIA	Semiconductors

Claim Extraction Pipeline

I extract quantitative claims from transcript text using a multi-stage pipeline designed for high recall and precision on financial language.

Stage 1 — GLiNER Entity Pre-Filter

The first pass uses GLiNER, a zero-shot Named Entity Recognition model that identifies custom entity types without any training data. GLiNER is a compact transformer encoder (<500M parameters) that runs on CPU, produces deterministic outputs with confidence scores, and outperforms ChatGPT on zero-shot NER benchmarks. I define custom financial entity types:[10][11][12][13]

- `REVENUE_CLAIM`, `INCOME_CLAIM`, `EPS_CLAIM`, `MARGIN_CLAIM`
- `GROWTH_PERCENTAGE`, `ABSOLUTE_VALUE`, `COMPARISON_CLAIM`
- `NON_GAAP_INDICATOR`, `PERIOD_REFERENCE`, `HEDGING_LANGUAGE`

```
from gliner import GLiNER
model = GLiNER.from_pretrained("urchade/gliner_medium-v2.1")
entities = model.predict_entities(text, labels=financial_entity_types)
```

This filters out ~70% of transcript content (qualitative discussion, greetings, analyst questions about strategy, etc.) and passes only sentences containing financial entities to the LLM extraction stage. This saves 60-70% of LLM token costs.[14]

Stage 2 — LLM-Based Claim Extraction (Claimify-Inspired)

For sentences flagged by GLiNER, I use an LLM-based extraction pipeline inspired by Microsoft's Claimify methodology (ACL 2025), which achieves 87.9% accuracy — the highest of any claim extraction method tested. The pipeline operates in three sub-stages:[15][16][17]

Selection: The LLM identifies which sentences contain verifiable quantitative content vs. unverifiable opinions or forward-looking statements. Forward-looking claims ("I expect", "guidance", "outlook") are tagged but excluded from fact-checking.

Disambiguation: For sentences with ambiguity, the LLM resolves it using surrounding context. If ambiguity cannot be resolved, the claim is marked as ambiguous rather than guessed at.[15]

Decomposition: Each disambiguated sentence is decomposed into standalone, verifiable claims with preserved context. The LLM returns structured JSON:

```
{
  "metric": "revenue",
  "claim_type": "percentage_growth",
  "stated_value": 15.0,
  "unit": "percent",
  "period": "YoY",
  "is_gaap": true,
  "is_forward_looking": false,
  "hedging_language": false,
  "raw_text": "Revenue increased 15% year over year to $94.8 billion",
  "speaker": "CEO"
}
```

Stage 3 — Normalization & Context Enrichment

Extracted claims are deduplicated, normalized to a canonical schema, and enriched:

- **Metric normalization**: Map aliases ("top line" → "revenue", "bottom line" → "net_income", "earnings per share" → "eps")
- **Period normalization**: "year over year" → "YoY", "sequentially" → "QoQ"
- **GAAP detection**: Keywords like "adjusted", "non-GAAP", "excluding", "pro forma" → `is_gaap=False`
- **Hedging detection**: Words like "approximately", "roughly", "about" → widens verification tolerance

- **Context capture**: 2 sentences before and after the claim for verification reasoning

Each claim is stored as: `(id, ticker, quarter, year, speaker, metric, value, unit, period, gaap_flag, raw_text, extraction_method, confidence, context)`.

Hybrid RAG Retrieval Pipeline

Financial claim verification has unique retrieval challenges: exact numbers matter, temporal precision is critical (Q2 2025 vs Q2 2024), and metric specificity is essential ("Revenue" vs "Net Revenue" vs "Adjusted Revenue"). I use a multi-stage retrieval pipeline.

Stage 1 — Deterministic Database Lookup

Before any vector search, I attempt a direct database query against our structured financial data in PostgreSQL. Since our financial data is stored in structured form (via edgartools/SEC EDGAR), many claims can be verified purely through SQL/DataFrame lookups:

```
current = financial_db.query(ticker="AAPL", metric="Revenue", quarter="Q2", year=2025)
prior = financial_db.query(ticker="AAPL", metric="Revenue", quarter="Q2", year=2024)
actual_growth = (current - prior) / prior * 100
```

This can handle ~60-70% of straightforward claims without any LLM or vector search.

Stage 2 — Hybrid Sparse-Dense Retrieval

For claims that reference non-standard metrics, segment data, or require context from filing text, I use hybrid retrieval over PostgreSQL with pgvector:[18][19]

Sparse Component — SPLADE-v3: Generates learned sparse vectors that outperform BM25 through term expansion. pgvector 0.7+ natively supports sparse vectors via the `sparsevec` type, so SPLADE embeddings are stored and queried directly in PostgreSQL. Critical for financial data because exact terms ("EBITDA", "free cash flow", "Q2 2025") need lexical matching, while SPLADE also expands to related terms.[20][21][22]

```
from fastembed import SparseTextEmbedding
sparse_model = SparseTextEmbedding("Qdrant/SPLADE_PP_en_v1")
```

Dense Component — BGE-M3 / bge-large-en-v1.5: Captures semantic similarity (e.g., "top line grew" → "revenue increased"). Stored as pgvector `vector` columns.[23]

Fusion — Reciprocal Rank Fusion (RRF): Combines ranked lists from sparse and dense retrieval without requiring score normalization, implemented in pure SQL. Formula: `RRF_score(d) = Σ 1/(k + rank_i(d))` where k=60.[24][19][18]

Hybrid retrieval achieves nDCG@10 of 0.741 vs 0.685 for dense-only.[24]

Stage 3 — Cross-Encoder Reranking

The reranking stage optimizes for precision using `cross-encoder/ms-marco-MiniLM-L12-v2`. Cross-encoders jointly encode query+document in a single transformer pass, enabling full cross-attention between tokens — this captures nuanced relevance like distinguishing "Q2 2025 revenue" from "Q2 2024 revenue" in the same document. Reranks top-100 candidates from Stage 2, returns top-10. [25][26][24]

Financial-Domain Optimizations

Metadata-Based Pre-Filtering: Following FinRAG research, documents are indexed with rich metadata (`ticker` , `quarter` , `year` , `metric_type` , `source_type` , `is_gaap`). When verifying an AAPL Q2 2025 revenue claim, I first filter via SQL WHERE clause to only AAPL + Q2 2025 rows, then run hybrid search within that subset.[27]

Structural Chunking: Financial documents are chunked by structural elements rather than fixed token windows — one chunk per line item per quarter for income statements, one chunk per speaker segment for transcripts. Each chunk retains full metadata headers.

Unified Database — PostgreSQL + pgvector

All data lives in a single PostgreSQL instance with pgvector extension:[28][18]

- Structured financial data (revenue, EPS, income — the actual numbers I verify against)
- Vector embeddings (dense `vector` columns + sparse `sparsevec` columns for SPLADE)
- Claim results and verdicts
- Transcript text and metadata

This eliminates a separate vector database service. At our scale (~20K-50K vectors for 10 companies × 4 quarters), pgvector handles this trivially with full ACID transactional consistency.[29][30]

Pipeline Architecture

```
CLAIM INPUT
    │
    ▼
┌──────────────────────────────┐
│  Stage 1: Deterministic DB   │  ← Direct SQL lookup
│  Lookup (structured data)    │  ← Handles ~60-70% of claims
└──────────────┬───────────────┘
               │ (if not resolved)
               ▼
┌──────────────────────────────┐
│  Stage 2: Hybrid Retrieval   │
│  ┌───────────┬────────────┐  │
│  │ SPLADE-v3 │  BGE-M3    │  │  ← Sparse + Dense in parallel
│  │ (sparse)  │  (dense)   │  │  ← All in PostgreSQL + pgvector
│  └─────┬─────┴─────┬──────┘  │
│        └─────┬─────┘         │
│     Reciprocal Rank Fusion   │  ← RRF in pure SQL
│        (top-100)             │
└──────────────┬───────────────┘
               │
               ▼
┌──────────────────────────────┐
│  Stage 3: Cross-Encoder      │
│  Reranking                  │  ← ms-marco-MiniLM-L12-v2
│  (top-100 → top-10)        │
└─────────────────────────────┘
              │
              │
              ▼
┌─────────────────────────────┐
│  LLM Verification Engine    │  ← Chain-of-thought reasoning
└─────────────────────────────┘
```

Verification Engine

The verification step is the system's centerpiece. For each extracted claim, the system retrieves the relevant official data and checks it. For example, if the claim is "Revenue grew 15% YoY in Q2", I fetch Q2 revenue for this year and last year and compute the actual growth. If the numbers match (within an acceptable tolerance), the claim is verified; if not, it is flagged as a discrepancy.

Deterministic Verification

For straightforward claims mapping to known metrics, I compute the exact value programmatically:

- **Growth claims (YoY/QoQ)**: `actual = (current - prior) / abs(prior) × 100`
- **Absolute value claims**: Direct comparison
- **EPS claims**: Compare to diluted EPS with penny-level precision

Tolerance logic:

- Claims with hedging language ("approximately", "roughly"): ±2% for percentages, ±5% for dollar amounts
- Precise claims: ±0.5% for percentages, ±1% for dollar amounts

LLM Chain-of-Thought Verification

For nuanced claims (non-GAAP metrics, segment data, multi-step calculations), the LLM receives the claim + retrieved context and reasons step-by-step:

1. **IDENTIFY**: Map the claimed metric to official data fields
2. **RETRIEVE**: Find exact numbers from official data for all relevant periods
3. **COMPUTE**: Calculate the actual value with explicit math
4. **COMPARE**: Compare claimed vs. actual, state the difference
5. **TOLERANCE**: Apply appropriate tolerance based on hedging language
6. **MISLEADING CHECK**: Evaluate framing even if technically true
7. **VERDICT**: Classify the claim

This mirrors the FINDVER approach of "context-grounded claim verification". The LLM outputs an explainable verdict so auditors can see the reasoning.

Misleading Framing Detection

The LLM is specifically prompted to detect:

- **Cherry-picking**: Highlighting a positive metric while hiding a negative one (e.g., "Revenue up 10%!" while Net Income dropped 50%)
- **Non-GAAP vs GAAP divergence**: When "Adjusted EBITDA" is touted but actual Net Income is negative
- **Period-shopping**: Using YoY when QoQ looks bad (or vice versa)
- **Base-effect manipulation**: Huge percentage growth off a tiny base
- **Omission of context**: Revenue growth from acquisitions presented as organic growth

Verdict Categories

Each claim receives one of:

- **VERIFIED**: Claim matches official data within tolerance
- **APPROXIMATELY_TRUE**: Claim is close but slightly off, consistent with rounding
- **FALSE**: Claim materially misrepresents the actual data
- **MISLEADING**: Claim may be technically true but framed in a misleading way
- **UNVERIFIABLE**: Insufficient data to verify (with explanation)

Quarter-to-Quarter Discrepancy (Bonus)

For every claim, I also compute the alternative time comparison. If a CEO claims high YoY growth, I also compute QoQ. If QoQ was flat or down, I flag this contextual discrepancy — showing if a figure jumps unusually between successive quarters.

LLM Model Architecture

The system uses a tiered model architecture with automatic fallback, accessible through a unified OpenAI-compatible API interface.

Tier 1: Default Cloud — deepseek-v3.1:671b-cloud via ollama

I use the deepseek-v3.1:671b-cloud model via ollama as the default model for claim verification. It is a large language model that is trained on a massive dataset of text and code, and it is capable of performing a wide range of tasks, including claim verification.

Tier 2: Premium Cloud — Claude Opus 4.1 / GPT-5

For users who want to use their own models for maximum accuracy: I plan to add a feature for them to use their own API keys and run the verification engine.

Plug-and-Play Switching

All models are accessed through LiteLLM for a unified interface.

Edge Cases and Quality

To be robust, the system handles many edge cases:

- **Missing or garbled data**: If an API lacks data for a quarter or the transcript lacks numeric detail, I skip or flag that claim.
- **Non-numeric language**: Phrases like "double-digit growth" are interpreted as ranges (10–99%). If numeric data can't pin it down exactly, I mark it as "ambiguous" and present both ends of the range.
- **Currency conversions**: If the transcript cites one currency and the financial report uses another, I convert using exchange rates from that date.
- **One-time items**: Executives often exclude unusual costs ("adjusted"), so if a claim uses non-GAAP terms, I identify the related GAAP line and compare accordingly. If the LLM identifies "non-GAAP" wording, I retrieve both GAAP and non-GAAP figures.
- **Composite statements**: e.g. "Revenue in Asia grew 20%". I handle this by retrieving segment data where available. If segment data isn't available, I note that limitation.

Handling these intricacies is informed by financial fact-checking research. Fin-Fact notes that financial claims often require understanding domain-specific terms and context. Our system annotates the reasoning (via the LLM's chain-of-thought) so auditors can see why a claim was marked false or misleading, echoing the "explainable" emphasis in FINDVER and FinChain.

Architecture & Scalability

I build the system in Python with a clear module structure. Key components (transcript ingestion, data lookup, claim extraction, verification, and UI) are decoupled.

This microservices approach ensures each part can scale independently.

The transcript-fetcher can run as a scheduled job;
The claim extraction and document_chunker can run independently;

Our environment is managed by `uv` and a proper `pyproject.toml`, ensuring reproducibility. I containerize the app with Docker and deploy to the cloud. Each service runs in its own container with a load balancer in front.

Technology Stack

Component	Technology
Database	PostgreSQL + pgvector (unified: structured data + vectors)
Sparse Encoder	SPLADE-v3 via FastEmbed → stored as pgvector `sparsevec`
Dense Encoder	BGE-M3 / bge-large-en-v1.5 → stored as pgvector `vector`
Fusion	Reciprocal Rank Fusion (k=60) in pure SQL
Reranker	cross-encoder/ms-marco-MiniLM-L12-v2
Entity Pre-Filter	GLiNER (urchade/gliner_medium-v2.1)
Claim Extraction	LLM (Claimify-style pipeline)
Verification LLM	Tiered: Groq/Claude/GPT/Ollama
Structured Data	edgartools + SEC EDGAR XBRL
Web Framework	FastAPI
UI	Streamlit

Component	Technology
Package Manager	uv
Deployment	Streamlit Cloud for UI and fastAPI backend + NeonDB for PostgreSQL + pgvector

---

Claude Skill (Bonus)

As a lightweight bonus, I include a `SKILL.md` file in the repository that wraps our FastAPI API as a Claude Skill. This allows anyone with Claude Pro/Team to query the system conversationally (e.g., "Verify AAPL's revenue claims from Q2 2025"). It requires zero changes to the core system — just a single markdown file describing the skill's instructions and API endpoints.[42][43]

References

1. Exploring the finnhub.io API | IBKR Quant - The free tier is quite generous and offers more than enough for doing proof of concept work and test...
2. Earnings Call Transcripts API - Finnhub - Free stock API for realtime market data, global company fundamentals, economic data, and a...
3. Exploring the finnhub.io API - Robot Wealth - One such newcomer is finnhub.io. Its offering includes stock, bond, crpto, and FX historical price d...
4. Bose345/sp500_earnings_transcripts · Datasets at Hugging Face - This comprehensive dataset contains earnings call transcripts for S&P 500 companies and US large-cap...
5. EDGAR Application Programming Interfaces (APIs) - SEC.gov - The company-concept API returns all the XBRL disclosures from a single company (CIK) and concept (a ...
6. Downloading Company Financial Data Using the SEC EDGAR API - I downloaded, organized, and queried the JSON file that contains the CIK, the company name, and the...
7. edgartools - Navigate Edgar filings with ease - EdgarTools is a Python library for downloading and analyzing SEC EDGAR filings. Extract 10-K, 10-Q, ...
8. Extract Financial Statements - EdgarTools - Python Library for ... - Extract Financial Statements ... Learn how to extract and work with financial statements from SEC fi...
9. edgartools - Python library for SEC EDGAR data : r/quant - Reddit - Pulls financials directly from XBRL (income statements, balance sheets, cash flows). Accesses any SE...
10. The rise of small language models for information extraction - Despite their small size, GLiNER models support impressive zero-shot entity recognition capability. ...
11. GLiNER2: An Efficient Multi-Task Information Extraction System with ... - GLiREL (Boylan et al., 2025) extended the approach to relation extraction, while GLiClass (Knowledga...
12. urchade/GLiNER: Generalist and Lightweight Model for ... - GitHub - GLiNER is a framework for training and deploying Named Entity Recognition (NER) models that can iden...
13. GLiNER: Generalist Model for Named Entity Recognition Using ... - GLiNER is an open-source NER model using a bidirectional transformer encoder. It allows for parallel...
14. Extract any entity from text with GLiNER | Towards Data Science - At the time of publication, GLiNER outperforms both ChatGPT and LLM optimized zero-shot NER datasets...

15. Claimify: Extracting high-quality claims from language model outputs - A common strategy for fact-checking LLM-generated texts – especially complex, highly detailed output...
16. Towards Effective Extraction and Evaluation of Factual Claims - I propose a framework for evaluating claim extraction and introduce Claimify, an LLM-based method t...
17. [PDF] Towards Effective Extraction and Evaluation of Factual Claims - Claimify achieved the highest accuracy (87.9%) and macro F1 score (83.7%), fol- lowed by DnD (accura...
18. Building Hybrid Search for RAG: Combining pgvector and Full-Text ... - Reciprocal Rank Fusion. Here's where the magic happens. RRF merges two ranked lists without needing ...
19. Hybrid search with PostgreSQL and pgvector - Jonathan Katz - Hybrid search is the act of combining an alternative searching method with a vector similarity searc...
20. Comparing SPLADE Sparse Vectors with BM25 - This article discusses two of the most popular algorithms for transforming text into sparse embeddin...
21. [PDF] Reranking with Compressed Document Representation - arXiv - First, a late interaction architecture has been proposed for cross-encoder-based rerankers, but it c...
22. Similarity Search with SPLADE Inside Postgres - ParadeDB - With the release of v0.7.0 , sparse vector support was added to pgvector . As a result, I have depr...
23. [PDF] ClimateCheck Task: Fact-Checking Social Media Claims - A clear pattern from subtask I is the use of hybrid pipelines, combining sparse retrievers (e.g., BM...
24. Graph-Augmented Hybrid Retrieval and Multi-Stage Re-ranking - This paper addresses critical limitations in modern Retrieval-Augmented Generation (RAG) systems, na...
25. How to Build Cross-Encoder Re-Ranking - OneUptime - Learn to implement cross-encoder re-ranking with pairwise scoring, batch optimization, and model fin...
26. Cross-Encoder Reranking - Emergent Mind - Cross-Encoder Reranking. Updated 5 October 2025. Cross-encoder reranking is a neural retrieval techn...
27. [PDF] FinRAG: A Retrieval-Based Financial Analyst - Stanford University - FinRAG aims to enhance retrieval-augmented generation (RAG) methods for fi- nancial data analysis, w...
28. Hybrid Search in PostgreSQL: The Missing Manual - We'll walk through building a complete solution that combines lexical precision with semantic unders...
29. Debunking 6 common pgvector myths - Nile Postgres - Myth 6: pg_vector does not support BM25 (and other sparse vectors). There are two types of vector em...
30. Production Vector Databases - Dataquest - Performance Patterns Are Clear: pgvector delivers 2.5ms baseline (fastest), Qdrant 52ms (moderate wi...
31. Best Free Open Source LLM APIs in 2026 - Hypereal AI - Widest selection of open-source models. If you want to test different models (Llama, Qwen, Mistral, ...
32. GroqCloud™ Makes DeepSeek R1 Distill Llama 70B Available - Groq believes DeepSeek-R1-Distill-Llama-70b is the right balance of performance versus price, where ...
33. Best LLMs for Financial Analysis: A Guide for BFSIs - Neurons Lab - Choosing LLMs for finance? Learn which models fit banking and FS use cases, reduce hallucinations, m...
34. LLM Leaderboard: Which LLMs are Best for Which Tasks? - Stack AI - Models like GPT-5, Claude Opus, and Gemini 2.5 Pro hold a clear advantage in reasoning-heavy financi...
35. Democratizing Financial Advice with DeepSeek-R1: A Case Study in ... - Ollama is an open-source tool that enables the local running of large language models, providing gre...
36. Ultimate Guide - The Best Open Source LLM for Finance in 2026 - Our top three picks for finance in 2026 are DeepSeek-R1, Qwen3-235B-A22B, and QwQ-32B. Each of these...
37. Deploy a FastAPI App | Railway Guides - This guide covers how to deploy a FastAPI app on Railway in four ways: One-click deploy from a templ...

38. Best Free Hosting Platforms For Backend Projects in 2025 - Railway auto-detects your backend language and build commands. If your app requires a database, add ...
39. Cerebrium articles | 5 Top Free Hosting Platforms for Python Apps - In this article, we'll share an honest, detailed comparison of your hosting options for CPU-intensiv...
40. Railway vs Render (2026): Which cloud platform fits your workflow ... - Does Railway have a free tier? Not in the traditional sense. It gives you $5 in usage credits once. ...
41. Streamlit free tier max users - Community Cloud - The free tier isn't limited by the number of public users per se, just by the amount of resources th...
42. [PDF] The Complete Guide to Building Skills for Claude | Anthropic - We've found that the most effective skill creators iterate on a single challenging task until Claude...
43. What are Claude Skills? Modular Workflows & MCP Integration - Vertu - Claude Skills are reusable instruction packages that bridge the gap between AI capabilities and spec...