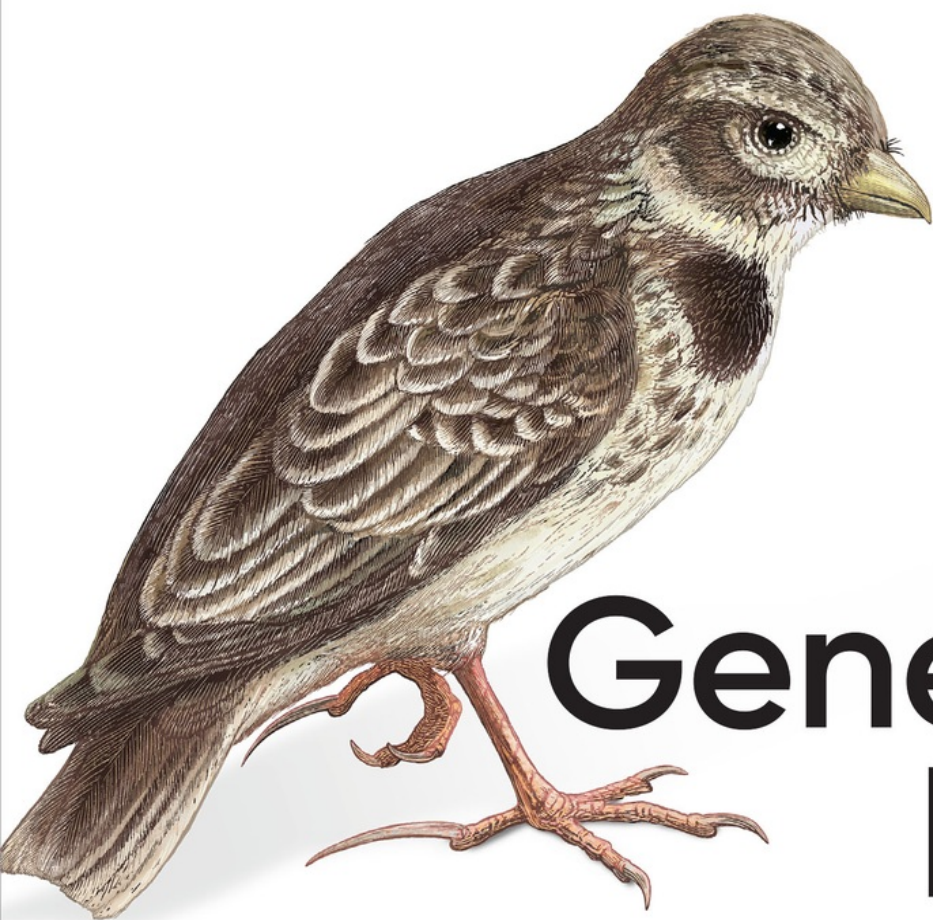


O'REILLY®



Generative Engine Optimization with Python

Data-Driven Methods for
LLM Retrieval and Citation

**Early
Release**

RAW &
UNEDITED

Andreas Voniatis

Generative Engine Optimization with Python

Data-Driven Methods for LLM Retrieval and
Citation

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

Andreas Voniatis

O'REILLY®

Generative Engine Optimization with Python

by Andreas Voniatis

Copyright © 2027 Andreas Voniatis, Inc. All rights reserved.

Published by O'Reilly Media, Inc., 141 Stony Circle, Suite 195, Santa Rosa, CA 95401.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<https://oreilly.com>). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com.

Acquisitions Editor: Aaron Black

Development Editor: Sara Hunter

Production Editor: Christopher Faucher

Cover Designer: Susan Brown

Cover Illustrator: José Marzan Jr.

Interior Designer: David Futato

Interior Illustrator: Kate Dullea

April 2027: First Edition

Revision History for the Early Release

- 2026-07-30: First release
- 2026-09-04: Second release

See <https://oreilly.com/catalog/errata.csp?isbn=9798341674585> for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Generative Engine Optimization with Python*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

The views expressed in this work are those of the author and do not represent the publisher's views. While the publisher and the author have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

979-8-341-67454-7

[LSI]

Brief Table of Contents (*Not Yet Final*)

Preface (unavailable)

Chapter 1: The Shift from SEO to GEO (available)

Chapter 2: Topic Research (unavailable)

Chapter 3: Content for AI (unavailable)

Chapter 4: Technical Optimization for AI Platforms (unavailable)

Chapter 5: Building AI-Visible Authority (unavailable)

Chapter 6: Measuring GEO Performance (available)

Chapter 7: Investigating and Future-Proofing GEO (unavailable)

Chapter 1. Introducing GEO

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 1st chapter of the final book. Please note that the GitHub repo will be made active later on.

If you’d like to be actively involved in reviewing and commenting on this draft, please reach out to the editor at shunter@oreilly.com.

This chapter will teach you what Generative Engine Optimization (GEO) actually is, how it relates to both search engines and SEO, and why I treat it as its own discipline rather than a rebrand of search optimization. You’ll learn how to:

- Define GEO
- Contrast and compare AI platforms with search engines
- Contrast and compare GEO with SEO
- Clarify GEO, AEO, AIO, and AI SEO
- Preview the five principles behind AI search visibility

Defining Generative Engine Optimization (GEO)

In March 2025, the New York Magazine declared “SEO is dead, say Hello to GEO!”. Which begs the question, what is GEO?

If you asked AI what Generative Engine Optimization (GEO) is, AI would define it as the practice of structuring content so that AI platforms such as ChatGPT, Perplexity, Google, Claude and similar systems select the content as source material when they generate an answer.

However, GEO is much more than structuring content. As this chapter and later chapters in this book will show, structuring content is only part of GEO.

I would define GEO as “the practice of optimizing content to be visible in AI search results”.

While this definition is broad and potentially vague, we must accommodate the different components that help content become visible in AI search results so that the definition remains relevant and not to be restricted to just a single aspect of GEO such as the structuring of content.

It’s also worth noting that AI search became mainstream in November 2022 when ChatGPT was first launched. It’s highly likely that best practices will evolve in the next decade and therefore a broad definition will still allow for these evolving practices.

AI Search and Search Engines

Before we discuss the differences between GEO and Search Engine Optimization (SEO), it’s worth appreciating the similarities and differences between AI search platforms and search engines.

In terms of similarities, these search technologies are:

- **Text driven:** Working from a database of internet content that is text based
- **Crawl then retrieve:** Extracting content from URLs, then evaluating before indexing

- **Validating by Internet:** Relying on third party signals such as backlinks to help score competing content against each other
- **Rewarding participation:** In exchange for content, in return publishers are rewarded by way of brand attribution and traffic
- **Adversarial:** Because of the rewards for participation, platforms have built defenses to avoid publishers being able to game the search system
- **Solving Information Retrieval (IR):** The capabilities are different but they still have the same objective of producing answers in response to search queries albeit in different forms.
- **Encoding constrained:** Facing the constraints of encoding a web-scale corpus into something queryable which costs real compute and energy.

Then there's the differences:

- **Lists vs Summaries:** Search engines mainly produce a list of web links while AI produces a summaries
- **Keyword vs Meaning:** Despite Google's advances in improving semantic understanding search engines are still anchored to keywords while AI is primarily driven by semantic meaning. As a result, AI can process and derive a lot more contextual information from the input to produce a more bespoke answer
- **Backlinks vs Mentions:** Backlinks are what helped Google to become the leading search engine. AI relies on mentions regardless of whether a mention is hyperlinked or not.
- **Clicks vs Display:** Search engine users need to click through the list of web results to get the answer, AI simply displays the answer in the interface without the optional need to click through
- **Service costs:** While both search engines and AI face the constraints of encoding a web-scale corpus, most AI researchers

argue including HuggingFace that the cost per search query is significantly higher for AI

- **Deterministic vs Probabilistic:** The search engine results for a given query are generally fixed with little variation for logged in user preferences, location. By contrast, AI is generative by nature which means that no answer will be identical for the same given query at any time
- **Rankings vs Mentions and Cited:** To complicate matters further, search engines core results are a ranked list of web pages. AI has two sets of results being mentioned in answers and being cited as a source URL.

Similarities and Differences with SEO

The term GEO has been and continues to be a source of controversy among the SEO industry community. SEOs will argue that there's no such thing as GEO and that AI search is simply an extension of SEO.

It's worth noting that AI relies heavily on search engine results, such that when a user of AI prompts a search, AI will often perform real time searches on search engines to help ground its response. From that point of view it could be argued that AI is an extension of SEO.

SEO practitioners will also go further to describe GEOs as opportunists looking to spin a new business market with unsustainable tactics designed to exploit the fear of businesses not wishing to miss out on the opportunities offered by AI search.

I will agree that SEO and GEO share the similarity in terms of solving Information Retrieval (IR). Such that both disciplines approach the IR problem for their respective technologies, SEO for search engines and GEO for AI.

Although AI is significant leap in IR capabilities compared to search engines, the components of experience optimization are shared such as:

- **Technical:** The ease of making content discoverable and retrievable for search technologies. Content that's indexable for search engines is usually retrievable to AI.
- **Users:** The impact of content used to train AI or gets included in AI search results
- **Internet:** How third parties on the internet rate the content. Both AI and traditional search engines rely on backlinks and brand searches to different extents in this regard.

However, there are also quite a few differences between SEO and GEO:

- **Persona Prompts not keywords:** SEO will conduct keyword research for campaign performance tracking and content topic selection. GEOs will research prompts based on fan out queries (real time searches conducted by AI). Keywords don't tell you anything about the persona behind the search. For example, the keyword "womens jeans" could cater for both designer or cheap jeans. GEO prompt research can address this by data mining the internet to infer personas so that the prompts match the search intent of the target audience
- **Information Gain (IG) thresholds:** Information Gain measures how much a variable reduces uncertainty about an outcome. From Information Retrieval (IR) perspective, that means content that adds value to a search platform to help it know more and therefore deliver more accurate answers in response to a query, adds IG. Because AI's capabilities, and costs, for solving IR are much higher than search engines, GEO has a much harder job than SEO to create content that delivers IG
- **Sentiment matters:** In search engines it's enough to be ranked but in AI results, it's insufficient. That's because the sentiment of your person or organization matters greatly so that you're not mentioned in an answer to a negative prompt such as "worst companies for

...”. GEOs have the additional challenge of adding positive brand sentiment when campaigning for their clients

- **Linear Regression vs Logistic:** Search engine results are paginated to 10 pages to show approximately 100 ranked results which is continuous data and therefore conducive to linear regression modelling. AI results are limited to a handful of ranked positions and so a person or organization is either in the AI results or they’re not making AI results modelling a logistic regression problem

Extract Transform Load (ETL)

SEO practitioners’ could still argue that the above differences are just additional and new challenges for SEOs to overcome as part of its own professional evolution. However there’s a wider context to be mindful of when we consider the natural roles of search engines, AI models and agents and that concept is Extract Transform Load (ETL).

ETL is the process of moving data from a source into a form you can actually analyze, done in three stages:

1. **Extract:** Pulling raw data from a data source which could be from an API, a database, log files, scraped pages. This stage usually comprises messy, inconsistent raw data
2. **Transform:** From its raw form, the data is then cleaned, reshaped, validated, and enriched before being modelled for the purpose of finding summarized insights
3. **Load:** The transformed data is then placed into its destination which could be a dashboard, another database or a data warehouse.

While ETL is usually reserved for information systems such as business intelligence data warehouses and dashboards, there’s a parallel that can be drawn for online search.

Search engines have successfully served to be the extraction layer while AI search's main operations, and strengths, have been in the transform layer. After all the underlying technology of AI today are Large Language Models (LLMs). This is further underlined by AI relying on search engines for search results when performing real time searches in response to their users' queries.

Agents or Agentic search would therefore fit the load layer and has its own discipline known as Agent Oriented Optimization (AOO), which deserves a book in its own right to cover technicalities such as Model Context Protocols (MCPs) to help agents interact with websites.

Give the differences between traditional and AI search platforms including how they fit within the overall ETL of online information retrieval, it's clear to me that GEO is a discipline in its own right. SEO professionals could absolutely transition to GEO as I have, however the differences must be respected in order to be visible in AI search. Beware of the SEO professional that trivializes AI as just an extension of SEO.

Is SEO Dead?

So while the New York Times contends that "SEO is Dead", I would highly disagree because of:

- **Drama:** News outlets need to create controversy to create clicks
- **Retrieval-Augmented Generation (RAG):** RAG is the architecture that lets AI answer questions in real time by conducting online searches on the fly which relies on search results. SEO can't be dead if search engine results give you the opportunity to be part of AI's search results when RAG is used
- **Search engine dominance:** Google still makes up over 80% of all online search traffic in most markets so SEO is still required

Even if AI is on the rise, and search traffic has mostly disappeared, SEO will still be required to ensure content continues to be discoverable and

indexable.

GEO vs AEO vs AIO vs AI SEO

With GEO's distinction from SEO established, the other point of contention is what should optimizing for AI search be called. There's several names for the seemingly same discipline:

- Generative Engine Optimization (GEO)
- Answer Engine Optimization (AEO)
- Artificial Intelligence Optimization (AIO)
- Artificial Intelligence Search Engine Optimization (AI SEO)

To my mind, GEO, AIO and AI SEO are describing the same discipline. Answer Engine Optimization (AEO) by contrast is much narrower in scope as it refers to only one of the 2 components of AI search results which is the generated answer i.e. being mentioned.

Therefore I prefer the broader definition as it's much easier to become the answer and secure brand mentions if your content is cited rather than not cited, as a source helping to generate the answer.

Given the New York Times article chose to use the term GEO, I'm happy to go with GEO for both its cultural adoption and broad scope.

Principles behind AI search visibility

To be visible in AI search your website must fulfil the requirements of:

- **Persona Driven Topics:** Your topics needs to be driven by conversations your audiences are already having and will have
- **Information Gain:** The topics your create content must be data driven enough to add value to the internet and provide information gain

- **Retrievability:** Once the content is published, above the basic SEO best practices, the content needs to be structured and marked up in a manner that maximizes LLM retrievability
- **Sentiment:** Your brand and preferably your content needs to be received well such that it elicits positive reviews from community forums such as Reddit
- **Authority:** Because of RAG, your content will need to be capable of earning backlinks from high traffic websites so that they perform strongly in the search engine results

The following chapters in this book will teach you how to achieve the above in a data driven way using python.

Summary

This chapter defined Generative Engine Optimization (GEO) and established it as a discipline distinct from SEO, not a rebrand of it.

To draw that distinction, we:

1. Compared AI platforms against search engines: shared traits (text-driven, crawl-then-retrieve, backlink validation, compute constraints) versus key divergences (summaries vs. lists, meaning vs. keywords, mentions vs. backlinks, cost, probabilistic output)
2. Compared GEO against SEO directly — persona-driven prompts over keywords, Information Gain as a content threshold, sentiment as a visibility risk, and AI visibility as a logistic rather than linear problem
3. Placed search engines, AI platforms, and agents inside a single ETL framework — extraction, transform, and load — the strongest argument for treating GEO as its own discipline

In the next chapter, we start building the Python pipeline that selects audience driven topics for AI visibility.

Chapter 2. Measuring GEO Performance

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 6th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you’d like to be actively involved in reviewing and commenting on this draft, please reach out to the editor at shunter@oreilly.com.

This chapter will teach you how to build Python analytical pipelines that track and interpret brand visibility across AI search platforms for your dashboards. You’ll learn how to:

- Extract GEO raw performance data across multiple platforms
- Analyze performance metrics aggregated by search intent, platform and overall AI search

It’s near impossible to know how your website shows up in AI search results for your target audiences. This chapter outlines the data pipelines to help you build the infrastructure to track Generative Engine Optimization (GEO) performance to power your dashboards and reporting systems.

You can’t reduce AI search visibility to a single metric because it hides the many moving parts that make up the overall score. Tracking AI search visibility successfully requires that you analyze/track(?) two sets of results:

- **Answer:** generated in response to user prompts forming the main content of the AI search platform's response
- **Cited sources:** web page results AI search relies upon to summarize when generating the answers. Cited sources is an additional dimension to traditional search engine results which is mainly a list of website page links.

Within the generated answer itself, there are two attributes also worth tracking:

- **Position:** The rank of your organization within a list when your target audience are prompting for best or recommended organizations in your industry
- **Sentiment:** How AI speaks about your organization

While rank position has been the mainstay of performance tracking in search engines, sentiment is new and incredibly important. For example, if a user prompted:

“Which are companies should I avoid for Oracle Netsuite consulting?”

Having your organization as part of the answer would be highly undesirable.

The importance of GEO performance measurement

Because the feedback loop between GEO and AI search results can be quite slow, noisy and distributed across multiple AI platforms, having a measurement framework is critical to managing expectations and optimizing campaigns. Especially as AI platforms tend to behave differently from one another. For example, you'll see in this chapter's pipeline how ChatGPT doesn't cite many source website pages when compared with Gemini.

The performance measurement framework will help the GEO see how the activities are moving (or not) the metrics on mentions, citations, positions and sentiment. It also allows the GEO to falsify the evolving best practices that are being constantly published online.

Coding the performance measurement framework into a pipeline will automate the data extraction and analysis, ready for reports exporting to dashboard, spreadsheets or otherwise.

Data Pipelines for Performance Measurement

The pipeline starts by querying the APIs of the main AI search platforms. For this walkthrough, I've chosen to include the following: , :

- ChatGPT
- Gemini
- Perplexity
- Claude

You may choose to query other T AI search platforms, such as xAI. Whatever you choose, you'll need to have an API key which is usually paid under subscription.

From the platforms, you'll extract search responses for mentioned, citations, position, and sentiment metrics.

You'll then aggregate the metrics by search intent, platform and AI overall to get a sense of where the website is under and over performing. Because AI search is generative by nature (i.e. made up), the results are not identical for each prompt so you'll need to make multiple API requests for each prompt to obtain samples that help you determine the rates for each metric.

Data Extraction from AI search APIs

Each AI search platform API has python packages which will need to be installed before importing. As a reminder, you'll need a paid subscription to do this.

```
import os
import anthropic
from google import genai as google_genai
from openai import OpenAI
```

Once imported the API keys are set and instantiated:

```
OPENAI_API_KEY = "sk-proj-..."
GEMINI_API_KEY = "..."
ANTHROPIC_API_KEY = "sk-ant-..."
PERPLEXITY_API_KEY = "pplx-..."

openai_client = OpenAI(api_key=OPENAI_API_KEY)
anthropic_client = anthropic.Anthropic(api_key=ANTHROPIC_API_KEY)
genai.configure(api_key=GEMINI_API_KEY)
```

Further imports are made to help manipulate and process the data:

```
import os
import re
import time
import glob
from datetime import date
from urllib.parse import urlparse

import pandas as pd
from transformers import pipeline as hf_pipeline
```

Pipeline input constants are set to prevent magic strings being scattered throughout the notebook starting with the identity.

Identity

The identity section sets the parameters for the website, its target region and the region prompt. The run date will be used for export file naming.

```

YOUR_DOMAIN    = "randgroup.com"
TARGET_REGION  = "United States"
REGION_HINT    = f"You are answering for a user based in the
{TARGET_REGION}."
RUN_DATE       = date.today().isoformat()

```

Pipeline parameters

For this walkthrough, I'm using 10 samples by way of API requests to be taken per prompt. If the API responses returned faster, then I recommend taking higher sample sizes up to 100.

The sleep seconds (SLEEP_SECS) is a per-platform pause between API calls to help with rate limiting, cost control and avoid triggering abuse detection even if the requests are within the rate limits.

Sleep second values will vary according to the platform based on previous API usage and the platform's rate limits.

```

RUNS_PER_QUESTION = 10    #
SLEEP_SECS = {
    "openai": 1.0,
    "gemini": 2.0,
    "claude": 0.5,
    "perplexity": 0.5,
}
TOP_N_COMPETITORS = 10

```

Reserved for the visualisation pipeline

```

BRAND_COLOURS = {"primary": "#069492", "secondary": "#29265b",
"accent": "#e1ff00"}

```

Brand map

The DOMAIN_BRAND_MAP dictionary acts as a lookup table mapping the competitor domains to their brand names used in the AI API response as AI models typically don't print domain names in response text.

The client brand names CLIENT_ALIASES is set by using get on the DOMAIN_BRAND_MAP

```

DOMAIN_BRAND_MAP = {
    "buildxact.com": ["buildxact"],
    "bluecollar.cloud": ["bluecollar", "blue collar"],
    "getjobber.com": ["jobber"],
    "plative.com": ["plative"],
    "accrete.com": ["accrete"],
    "appficiency.com": ["appficiency"],
    "randgroup.com": ["rand group", "randgroup"],
}

CLIENT_ALIASES = DOMAIN_BRAND_MAP.get(YOUR_DOMAIN, ["randgroup",
"rand group"])

```

Target Questions and Competitive Queries

The prompts are set in a list assigned to TARGET_QUESTIONS and COMPETITIVE_QUERIES which are categorized for analysis later. TARGET_QUESTIONS covers questions that are typically asked by the target audience during their research phase.

```

TARGET_QUESTIONS = [
    # Fragmented data pain points
    "why is my construction business data spread across so many systems",
    "how do I get a single view of job costs across my construction trades business",
    "how do construction trades businesses eliminate spreadsheet dependency",
    "what causes data silos in construction trades companies",
    "how do I connect field operations data with my back office in construction",
    "why do my project managers and accountants see different numbers for the same job",
    "how do construction companies get real time visibility into job profitability",
    "what is the best way to consolidate financial and operational data in construction",
    "how do I stop losing money because of delayed billing in construction trades",
    "how do construction trades CEOs get accurate cash flow forecasting",
    # NetSuite solutions for construction trades
    "is NetSuite good for construction trades businesses",
    "best ERP for construction trades companies",
    "how does NetSuite handle job costing for construction",

```

```

    "can NetSuite manage subcontractor payments for construction
    trades",
    "how long does a NetSuite implementation take for a
    construction company",
    "NetSuite vs Sage for construction trades",
    "how do construction trades businesses customise NetSuite for
    field operations",
    "what does a NetSuite implementation cost for a mid-size
    construction company",
    "how do I know if my construction business is ready for
    NetSuite",
]

```

COMPETITIVE_QUERIES are commercial where audiences search for a partner to help them resolve their problem.

```

COMPETITIVE_QUERIES = [
    "best NetSuite consultants for construction trades",
    "top ERP implementation partners for construction companies",
    "best NetSuite implementation partners USA construction",
]

```

Both sets of questions are merged into a single list.

```

ALL_QUESTIONS = TARGET_QUESTIONS + COMPETITIVE_QUERIES

```

Below are a set of printouts to

```

CLUSTER_MAP = {
    **{q: "problem" for q in TARGET_QUESTIONS[:10]},
    **{q: "solutions" for q in TARGET_QUESTIONS[10:]},
    **{q: "brands" for q in COMPETITIVE_QUERIES},
}

```

Below are a set of printouts to verify our inputs before running the API calls:

```

print(f"Total API calls      : {len(ALL_QUESTIONS) *
    RUNS_PER_QUESTION * 4:,} (4 platforms × 10 sessions ×
    {len(ALL_QUESTIONS)} prompts)")
print(f"Client domain        : {YOUR_DOMAIN}")

```

```

print(f"Target region      : {TARGET_REGION}")
print(f"Target questions   : {len(TARGET_QUESTIONS)}")
print(f"Competitive queries: {len(COMPETITIVE_QUERIES)}")
print(f"Runs per question    : {RUNS_PER_QUESTION}")
print(f"Run date              : {RUN_DATE}")
print(f"Total API calls       : {len(ALL_QUESTIONS) *
RUNS_PER_QUESTION * 4:,} (4 platforms × 10 sessions ×
{len(ALL_QUESTIONS)} prompts)")

```

Output:

```

Total API calls      : 880 (4 platforms × 10 sessions × 22
prompts)
Client domain        : randgroup.com
Target region        : United States
Target questions     : 19
Competitive queries: 3
Runs per question    : 10
Run date             : 2026-06-11
Total API calls      : 880 (4 platforms × 10 sessions × 22
prompts)

```

The output summarizes the number of API calls being made to extract visibility performance data broken down by the number of questions, sessions and platforms.

— Helper Functions

Query corpus

This section will define functions that convert a raw API content response into data structures with extracted URLs, domains, mentions using the brand map defined earlier, and position.

```

def extract_urls(text: str) -> list[str]:
    return re.findall(r'https?:\/\/[^\s\)\]\,\\\'<>]+' , text)

def normalise_domain(url: str) -> str:
    try:
        parsed = urlparse(url if url.startswith("http") else
"https://" + url)

```

```

        return parsed.netloc.lower().removeprefix("www.")
    except Exception:
        return ""

def get_cited_domains(text: str) -> list[str]:
    return list({d for u in extract_urls(text) if (d :=
normalise_domain(u))})

def detect_mentions(text: str, domain_brand_map: dict) ->
dict[str, bool]:
    text_lower = text.lower()
    return {
        domain: any(
            re.search(rf'\b{re.escape(alias)}\b', text_lower)
            for alias in aliases
        )
        for domain, aliases in domain_brand_map.items()
    }

def position_score(text: str, aliases: list[str]) -> float:

    text_lower = text.lower()
    hits = [
        m.start()
        for alias in aliases
        if (m := re.search(rf'\b{re.escape(alias)}\b',
text_lower))
    ]
    return round(1.0 - min(hits) / max(len(text), 1), 4) if hits
else 0.0

def classify_format(text: str) -> str:
    if re.search(r'\\|s*[-:]s*\\|', text) or text.count("|") > 4:
        return "table"
    if re.search(r'^s*[-•*]s+', text, re.MULTILINE):
        return "list"
    return "paragraph"

```

Sentiment Classifier (HuggingFace)

The next cell is our sentiment classifier which will extract the sentiment from the raw API content response by loading a pretrained sentiment model

from HuggingFace to classify the sentences that contain a brand mention.

I used HuggingFace as opposed to hard-coded term lists (positive/negative word banks) which are brittle and miss negation, irony, and domain-specific phrasing. The HuggingFace is a fine-tuned transformer that will handle these correctly without manual curation.

HuggingFace uses the distilbert-base-uncased-finetuned-sst-2-english model which is lightweight, doesn't require a GPU and is highly suited to short sentence structures.

The classifier runs once per response record and operates only on the sentences that contain a brand alias rather than the full response text to help keep input length manageable and focused on the brand mention context rather than the entire answer.

```
sentiment_classifier = hf_pipeline(
    "sentiment-analysis",
    model="distilbert-base-uncased-finetuned-sst-2-english",
    truncation=True,
    max_length=512,
)

def extract_mention_context(text: str, aliases: list[str]) -> str:
    """Return the sentences containing any brand alias,
    concatenated."""
    sentences = re.split(r'[.!?]', text)
    relevant = [
        s.strip() for s in sentences
        if any(re.search(rf'\b{re.escape(a)}\b', s.lower()) for a
in aliases)
    ]
    return " ".join(relevant)

def classify_sentiment(text: str, aliases: list[str]) -> float:
    context = extract_mention_context(text, aliases)
    if not context:
        return 0.0
    try:
        result = sentiment_classifier(context)[0]
        label = result["label"].lower()
        score = result["score"]
```

```

        scaled = (score - 0.5) * 2
        return round(scaled if label == "positive" else -scaled,
4)
    except Exception:
        return 0.0

print("Sentiment classifier loaded:",
      sentiment_classifier.model.name_or_path)

```

Output:

```

Loading weights: 100%
104/104 [00:00<00:00, 7976.15it/s]
Sentiment classifier loaded: distilbert-base-uncased-finetuned-
sst-2-english

```

The output confirms the model is loaded successfully. The function will score sentiment along a scale from -1 (negative) to 1 (positive). The confidence score from the classifier is used to scale the result, so a weakly positive result returns less than 1.0.

AI Platform API Functions

Query functions are defined per platform (OpenAI, Gemini, Claude, Perplexity). Each accepts a full conversation history and returns the next response plus the updated history, a question string and a conversation history list, returning response text, cited domains, cited URLs, and the updated history. The shared utility functions `trim_history` and `with_retries` are defined and used by all four API Functions.

Each adapter appends the user question to history, calls the platform API with `trim_history(history)` as the payload, extracts response text and cited the sources using platform-specific extraction logic before appending the assistant response to history. Then it returns a dict with `response_text`, `cited_domains`, `cited_urls`, `cited_pairs` where applicable, and `history`. On exception, the user message is popped from history before returning an empty result so history remains consistent for subsequent questions.

The API keys for each platform are set below.

```
openai_client      = OpenAI(api_key=OPENAI_API_KEY)
anthropic_client   =
anthropic.Anthropic(api_key=ANTHROPIC_API_KEY)
google_client      = google_genai.Client(api_key=GEMINI_API_KEY)
perplexity_client  = OpenAI(api_key=PERPLEXITY_API_KEY,
base_url="https://api.perplexity.ai")
```

Cap the history at 6 user/assistant pairs to prevent token costs and latency growing across 23 prompts.

```
MAX_HISTORY_TURNS = 6
```

Returns the last `MAX_HISTORY_TURNS * 2` messages, bounding the API payload regardless of session length.

```
def trim_history(history: list[dict]) -> list[dict]:
    return history[-(MAX_HISTORY_TURNS * 2):]
```

Retries a function up to 3 times with exponential backoff on 429 rate limit errors.

```
def with_retries(fn, retries: int = 3, backoff: int = 30):
    for attempt in range(retries):
        try:
            return fn()
        except Exception as e:
            is_rate_limit = "429" in str(e) or "rate" in
str(e).lower()
            if is_rate_limit and attempt < retries - 1:
                wait = backoff * (attempt + 1)
                print(f"\n rate limited - retrying in {wait}s")
                time.sleep(wait)
            else:
                raise
```

Reconstructs history as a system prompt addition, calls gpt-4o with `web_search_preview`, extracts inline URLs as best-effort citations.

```

def query_openai(question: str, history: list[dict]) -> dict:
    try:
        history.append({"role": "user", "content": question})
        history_context = "\n\n".join(
            f"{'User' if m['role'] == 'user' else 'Assistant'}: {m['content']}"
            for m in trim_history(history[:-1])
        )
        system = REGION_HINT
        if history_context:
            system += f"\n\nPrior conversation context:\n{history_context}"

        resp = with_retries(lambda:
openai_client.responses.create(
            model="gpt-4o",
            tools=[{"type": "web_search_preview"}],
            instructions=system,
            input=question,
        ))
        text = resp.output_text or ""
        history.append({"role": "assistant", "content": text})
        extracted_urls = extract_urls(text)
        domains = list({normalise_domain(u) for u in
extracted_urls if u})
        return {"response_text": text, "cited_domains": domains,
"cited_urls": extracted_urls, "history": history}
    except Exception as e:
        print(f"\n [OpenAI] {e}")
        history.pop()
        return {"response_text": "", "cited_domains": [],
"cited_urls": [], "history": history}

```

Converts history to Gemini's Content/Part schema, calls gemini-2.5-flash with Google Search grounding, extracts domains from grounding_metadata.grounding_chunks.

```

def query_gemini(question: str, history: list[dict]) -> dict:
    try:
        history.append({"role": "user", "content": question})
        contents = [
            google_genai.types.Content(
                role="user" if m["role"] == "user" else "model",
                parts=
[google_genai.types.Part(text=m["content"])],
            )

```

```

        for m in trim_history(history)
    ]
    resp = with_retries(lambda:
google_client.models.generate_content(
    model="gemini-2.5-flash",
    contents=contents,
    config=google_genai.types.GenerateContentConfig(
        system_instruction=REGION_HINT,
        tools=[google_genai.types.Tool(

google_search=google_genai.types.GoogleSearch()
    )],
    ),
    ))
    text = resp.text or ""
    cited = []
    if resp.candidates and
resp.candidates[0].grounding_metadata:
        chunks =
resp.candidates[0].grounding_metadata.grounding_chunks or []
        for chunk in chunks:
            if chunk.web and chunk.web.title:
                cited.append({"url": chunk.web.uri or "",
"domain": chunk.web.title})

        domains = list({c["domain"] for c in cited if
c["domain"]})
        urls = list({c["url"] for c in cited if c["url"]})
        history.append({"role": "assistant", "content": text})
        return {
            "response_text": text,
            "cited_domains": domains,
            "cited_urls": urls,
            "cited_pairs": cited,
            "history": history,
        }
    except Exception as e:
        print(f"\n [Gemini] {e}")
        history.pop()
        return {"response_text": "", "cited_domains": [],
"cited_urls": [], "history": history}

```

Nudges web search on every prompt, calls claude-sonnet-4 with `web_search_20250305`, extracts URLs from `web_search_tool_result` blocks.

```

def query_claude(question: str, history: list[dict]) -> dict:
    try:
        augmented_question = question + " Please search the web
        for current information."
        history.append({"role": "user", "content":
        augmented_question})
        resp = with_retries(lambda:
        anthropic_client.messages.create(
            model="claude-sonnet-4-20250514",
            max_tokens=1024,
            tools=[{"type": "web_search_20250305", "name":
            "web_search"}],
            system=REGION_HINT,
            messages=trim_history(history),
        ))

        text = " ".join(
            b.text for b in resp.content
            if getattr(b, "type", None) == "text"
        )

        cited = []
        for block in resp.content:
            if getattr(block, "type", None) ==
            "web_search_tool_result":
                for result in getattr(block, "content", []):
                    url = getattr(result, "url", None)
                    if url:
                        cited.append({"url": url, "domain":
                        normalise_domain(url)})

        domains = list({c["domain"] for c in cited if
        c["domain"]})
        urls = list({c["url"] for c in cited if c["url"]})

        history.append({"role": "assistant", "content": text})
        return {"response_text": text, "cited_domains": domains,
        "cited_urls": urls, "history": history}
    except Exception as e:
        print(f"\n [Claude] {e}")
        history.pop()
        return {"response_text": "", "cited_domains": [],
        "cited_urls": [], "history": history}

```

Calls sonar-pro via the OpenAI-compatible endpoint, extracts URLs from resp.citations which is Perplexity's citation field.

```

def query_perplexity(question: str, history: list[dict]) -> dict:
    try:
        history.append({"role": "user", "content": question})
        resp = with_retries(lambda:
perplexity_client.chat.completions.create(
            model="sonar-pro",
            messages=[{"role": "system", "content": REGION_HINT}]
+ trim_history(history),
            timeout=30,
        ))
        text = resp.choices[0].message.content or ""
        citations = getattr(resp, "citations", []) or []
        urls = list({u for u in citations if u})
        domains = list({normalise_domain(u) for u in urls if
u})
        history.append({"role": "assistant", "content": text})
        return {"response_text": text, "cited_domains": domains,
"cited_urls": urls, "history": history}
    except Exception as e:
        print(f"\n [Perplexity] {e}")
        history.pop()
        return {"response_text": "", "cited_domains": [],
"cited_urls": [], "history": history}

PLATFORM_ADAPTERS = {
    "openai": query_openai,
    "gemini": query_gemini,
    "claude": query_claude,
    "perplexity": query_perplexity,
}

```

```

print("Platform adapters ready:", list(PLATFORM_ADAPTERS.keys()))

```

Output:

```

Platform adapters ready: ['openai', 'gemini', 'claude',
'perplexity']

```

After these steps, the platform functions are loaded. The API calls will be made in 2 ways to extract performance data:

1. **Multi:** which runs all questions into the same session passing the history from the first prompt into the next to simulate the user experience of asking all prompts as part of an ongoing conversation
2. **Single:** runs each question as a single session independently of each other.

This is as opposed to the cold isolated queries based on Sundar Pichai's comments at the Google I/O 2026 event. Search is moving from individual queries to ongoing conversations. AI search is increasingly persona-driven and context-accumulating.

Passing chat history under the Multi mode replicates the experience most users actually have, making visibility measurements more representative of real-world AI search behaviour.

As a minimum for AI search performance, I recommend running both multi or single modes multiple times per week.

Multi: extract performance data

Let's walk through how to extra performance data under the Multi mode. First you'll run `RUNS_PER_QUESTION` full conversation sessions per platform where each session asks all `TARGET_QUESTIONS` then `COMPETITIVE_QUERIES` in sequence with history accumulated across the full thread.

The competitive queries asked at the end of an established conversation context will attempt to replicate how a real user researching would naturally arrive at vendor comparison questions after first exploring their pain points.

`ALL_QUESTIONS = TARGET_QUESTIONS + COMPETITIVE_QUERIES` defines the thread order such that a fresh history list is initialised per session, the response and citation records are tagged with question type (i.e. target vs competitive) so they can be split later on.

The `MODE_PREFIX` in this cell functions as the export naming constant.

```
MODE_PREFIX = "multi"
```

The `ThreadPoolExecutor` runs one thread per platform in parallel to save time while threading. `Lock` prevents concurrent threads from corrupting the shared `citation_rows` and `response_rows` lists when writing simultaneously. The progress tracks each platform's session and prompt position for the live display while the cell runs and `start_time` anchors the elapsed time calculation.

```
from concurrent.futures import ThreadPoolExecutor, as_completed
from IPython.display import clear_output
import threading

citation_rows = []
response_rows = []
lock          = threading.Lock()
start_time    = time.time()

progress = {
    p: {"session": 0, "prompt": 0, "status": "waiting",
        "elapsed": 0, "start": None}
    for p in PLATFORM_ADAPTERS
}
```

The `print_progress` function clears the output of Jupyter cells and reprints a single live status block showing total elapsed time to avoid printing multiple verbose lines with each platform's current session, prompt position, per-platform elapsed time, and running/waiting/done status.

```
def print_progress():
    now = time.time()
    clear_output(wait=True)
    total_elapsed = int(now - start_time)
    mins, secs = divmod(total_elapsed, 60)
    print(f"  total elapsed {mins:02d}:{secs:02d}\n")
    for p, s in progress.items():
        if s["start"] is not None:
            p_elapsed = int(now - s["start"])
```

```

        p_mins, p_secs = divmod(p_elapsed, 60)
        elapsed_str = f"{p_mins:02d}:{p_secs:02d}"
    else:
        elapsed_str = "--:--"
    print(
        f"  [{p:10}]  "
        f"session {s['session']:2}/{RUNS_PER_QUESTION}  "
        f"prompt {s['prompt']:2}/{len(ALL_QUESTIONS)}  "
        f"{elapsed_str}  "
        f"{s['status']}"
    )

```

The `run_platform` function executes all sessions for one platform sequentially so that each session initialises a fresh conversation history, walks through every question in `ALL_QUESTIONS` passing accumulated history to the adapter, tags each response as either target or competitive. Then it builds response and citation row dicts, and appends them to the shared lists under the thread lock before sleeping.

```

def run_platform(platform: str, adapter) -> None:
    """Run all sessions for a single platform sequentially."""
    progress[platform]["status"] = "running"
    progress[platform]["start"] = time.time()

    for session_idx in range(RUNS_PER_QUESTION):
        history = []

        for q_idx, question in enumerate(ALL_QUESTIONS, 1):
            result = adapter(question, history)
            text = result["response_text"]
            history = result["history"]

            progress[platform]["session"] = session_idx + 1
            progress[platform]["prompt"] = q_idx
            print_progress()

            q_type = "target" if question in TARGET_QUESTIONS
            else "competitive"

            r_row = {
                "question": question,
                "question_type": q_type,
                "session": session_idx,
                "platform": platform,
            }

```

```

        "response_text": text,
        "run_date":      RUN_DATE,
        "cluster":       CLUSTER_MAP.get(question,
"unknown"),
    }
    pairs = result.get("cited_pairs") or [
        {"domain": normalise_domain(u), "url": u}
        for u in result.get("cited_urls", [])
    ]

    c_rows = [
        {
            "question":      question,
            "question_type": q_type,
            "session":       session_idx,
            "platform":      platform,
            "domain":        p["domain"],
            "url":           p["url"],
            "run_date":      RUN_DATE,
        }
        for p in pairs
    ]

    with lock:
        response_rows.append(r_row)
        citation_rows.extend(c_rows)

    time.sleep(SLEEP_SECS[platform])

    progress[platform]["status"] = "done"

```

The `ThreadPoolExecutor` block launches the `run_platform` thread for each platform simultaneously, before waiting for all of them to complete, printing any platform-level exceptions that may arise without aborting the other threads.

```

with ThreadPoolExecutor(max_workers=len(PLATFORM_ADAPTERS)) as
executor:
    futures = {
        executor.submit(run_platform, platform, adapter):
platform
        for platform, adapter in PLATFORM_ADAPTERS.items()
    }
    for future in as_completed(futures):
        platform = futures[future]

```

```

        if future.exception():
            print(f"\n [{platform}] failed:
{future.exception()}")

```

While collecting data via the API calls, the code below prints the final progress state and record counts, converts the raw row lists to DataFrames, and persists both to mode-prefixed parquet files keyed by run date for later cells to load without having to re-run the API calls.

```

print_progress()
print(f"\nCollection complete.")
print(f"  Response records : {len(response_rows):,}")
print(f"  Citation records  : {len(citation_rows):,}")

responses_df = pd.DataFrame(response_rows)
citations_df = pd.DataFrame(citation_rows)

responses_df.to_parquet(f"
{MODE_PREFIX}_responses_{RUN_DATE}.parquet", index=False)
citations_df.to_parquet(f"
{MODE_PREFIX}_citations_{RUN_DATE}.parquet",  index=False)

```

Output:

```

total elapsed 158:25

[openai    ] session 10/10 prompt 22/22 158:25 done
[gemini     ] session 10/10 prompt 22/22 158:25 done
[claude     ] session 10/10 prompt 22/22 158:25 done
[perplexity] session 10/10 prompt 22/22 158:25 done

Collection complete.
  Response records : 880
  Citation records : 4,776

```

All of the API requests were run successfully taking over 2 hours to run.

Single: extract performance data

This time run RUNS_PER_QUESTION are independent single-question sessions per platform where each question is asked in isolation with no

prior conversation history.

The single-question mode is how commercial 3rd party AI search software tools like Profound and Peec measures raw entity authority showing how visible the brand is when a query arrives cold with no conversational context.

This will be contrasted with multi-question mode to reveal whether brand visibility is driven by entity authority alone or by conversational priming from related prior questions in the same session.

This data collection call runs with history is reset to [] before every question rather than accumulating across the session. Otherwise all other collection logic is identical to the previous multi cell above and exports are prefixed single_ to distinguish from multi_ exports.

```
from concurrent.futures import ThreadPoolExecutor, as_completed
from IPython.display import clear_output
import threading
```

Tags all exports from this cell with single_ to distinguish them from multi-question mode outputs.

```
MODE_PREFIX = "single"

citation_rows_single = []
response_rows_single = []
lock_single          = threading.Lock()
start_time_single    = time.time()
progress_single = {
    p: {"session": 0, "prompt": 0, "status": "waiting", "start":
None}
    for p in PLATFORM_ADAPTERS
}
```

Same as print_progress in Multi but shows total independent calls per platform (230) rather than session/prompt counts. This reflects that single mode has no conversational sessions.

```

def print_progress_single():
    now = time.time()
    clear_output(wait=True)
    total_elapsed = int(now - start_time_single)
    mins, secs = divmod(total_elapsed, 60)
    total_calls = len(ALL_QUESTIONS) * RUNS_PER_QUESTION
    print(f" [single] total elapsed {mins:02d}:{secs:02d} |
{total_calls} independent calls per platform\n")
    for p, s in progress_single.items():
        if s["start"] is not None:
            p_elapsed = int(now - s["start"])
            p_mins, p_secs = divmod(p_elapsed, 60)
            elapsed_str = f"{p_mins:02d}:{p_secs:02d}"
        else:
            elapsed_str = "--:--"
        calls_done = (s["session"]) * len(ALL_QUESTIONS) +
s["prompt"]
        print(
            f" [{p:10}] "
            f"call {calls_done:3}/{total_calls} "
            f"(run {s['session']:2}/{RUNS_PER_QUESTION} q
{s['prompt']:2}/{len(ALL_QUESTIONS)}) "
            f"{elapsed_str} "
            f"{s['status']}")
    )

```

This is identical to `run_platform` except history resets to `[]` before every question rather than accumulating across the session, isolating each query as a cold independent call.

```

def run_platform_single(platform: str, adapter) -> None:
    progress_single[platform]["status"] = "running"
    progress_single[platform]["start"] = time.time()

    for session_idx in range(RUNS_PER_QUESTION):

        for q_idx, question in enumerate(ALL_QUESTIONS, 1):
            history = []

            result = adapter(question, history)
            text = result["response_text"]

            progress_single[platform]["session"] = session_idx +
1
            progress_single[platform]["prompt"] = q_idx
            print_progress_single()

```

```

        q_type = "target" if question in TARGET_QUESTIONS
    else "competitive"

    r_row = {
        "question":      question,
        "question_type": q_type,
        "session":       session_idx,
        "platform":      platform,
        "response_text": text,
        "run_date":      RUN_DATE,
        "cluster":       CLUSTER_MAP.get(question,
"unknown"),
        "mode":          MODE_PREFIX,
    }

    pairs = result.get("cited_pairs") or [
        {"domain": normalise_domain(u), "url": u}
        for u in result.get("cited_urls", [])
    ]

    c_rows = [
        {
            "question":      question,
            "question_type": q_type,
            "session":       session_idx,
            "platform":      platform,
            "domain":        p["domain"],
            "url":           p["url"],
            "run_date":      RUN_DATE,
            "mode":          MODE_PREFIX,
        }
        for p in pairs
    ]

    with lock_single:
        response_rows_single.append(r_row)
        citation_rows_single.extend(c_rows)

    time.sleep(SLEEP_SECS[platform])

    progress_single[platform]["status"] = "done"

```

Launches all four platforms in parallel using `run_platform_single`, collecting exceptions without aborting other threads.

```

with ThreadPoolExecutor(max_workers=len(PLATFORM_ADAPTERS)) as
executor:
    futures = {
        executor.submit(run_platform_single, platform, adapter):
platform
        for platform, adapter in PLATFORM_ADAPTERS.items()
    }
    for future in as_completed(futures):
        platform = futures[future]
        if future.exception():
            print(f"\n [{platform}] failed:
{future.exception()}")

```

Prints final call counts, converts row lists to responses_single_df and citations_single_df, and persists both as single_ prefixed parquet files.

```

print_progress_single()
print(f"\nCollection complete [{MODE_PREFIX}].")
print(f"  Response records : {len(response_rows_single):,}")
print(f"  Citation records : {len(citation_rows_single):,}")

responses_single_df = pd.DataFrame(response_rows_single)
citations_single_df = pd.DataFrame(citation_rows_single)

responses_single_df.to_parquet(f"
{MODE_PREFIX}_responses_{RUN_DATE}.parquet", index=False)
citations_single_df.to_parquet(f"
{MODE_PREFIX}_citations_{RUN_DATE}.parquet", index=False)

```

Output

```

[single] total elapsed 39:12 | 220 independent calls per
platform

```

```

[openai    ] call 220/220 (run 10/10 q 22/22) 38:45 done
[gemini     ] call 220/220 (run 10/10 q 22/22) 39:12 done
[claude     ] call 220/220 (run 10/10 q 22/22) 38:58 done
[perplexity] call 220/220 (run 10/10 q 22/22) 37:21 done

```

```

Collection complete [single].
  Response records : 880
  Citation records : 11,161

```

The single data collection mode took under 1 hour to run which is about 4 times faster than multi the conversation history adds latency i.e. the cold short prompt calls are much faster to process.

Performance Analysis and Visualizations

With the data collected, we can now calculate mention, citation, position, and sentiment metrics.

This cell builds a `cited_lookup` dict keyed on (question, session, platform) for citation checks, then applies a single `extract_metrics` function across the `responses_df`. The sentiment runs via the HuggingFace classifier loaded earlier.

The data data collection modes of multi and single are toggled to drive all cells downstream.

```
ANALYSIS_MODE = "multi"    # or "single"
```

Routes the correct mode's data into shared names so extraction logic runs unchanged for both modes.

```
responses_df = multi_responses_df if ANALYSIS_MODE == "multi"
else single_responses_df
citations_df = multi_citations_df if ANALYSIS_MODE == "multi"
else single_citations_df
```

Pre-builds a (question, session, platform) where domain set dict for efficient citation checks per row.

```
cited_lookup = (
    citations_df
    .groupby(["question", "session", "platform"])["domain"]
    .apply(set)
    .to_dict()
)
```

All domains in DOMAIN_BRAND_MAP except the client, used for competitive displacement scoring.

```
COMPETITOR_DOMAINS = [
    domain for domain in DOMAIN_BRAND_MAP.keys()
    if domain != YOUR_DOMAIN
]
```

Computes all metrics for a single response row, returning them as a named Series.

```
def extract_metrics(row: pd.Series) -> pd.Series:
    text      = row["response_text"]
    key = (row["question"], row["session"], row["platform"])
    cited     = cited_lookup.get(key, set())

    if not text:
        return pd.Series({
            "mention":      False,
            "citation":      False,
            "position_score": 0.0,
            "sentiment":     0.0,
            "response_format": "unknown",
            "response_length": 0,
            "comp_citations": 0,
            "comp_mentions": 0,
        })

    mentions    = detect_mentions(text, DOMAIN_BRAND_MAP)
    mention     = mentions.get(YOUR_DOMAIN, False)
    citation    = YOUR_DOMAIN in cited
    pos         = position_score(text, CLIENT_ALIASES)
    sentiment   = classify_sentiment(text, CLIENT_ALIASES)
    fmt         = classify_format(text)

    comp_citations =
len(cited.intersection(set(COMPETITOR_DOMAINS)))
    comp_mentions = sum(1 for d in COMPETITOR_DOMAINS if
mentions.get(d, False))

    return pd.Series({
        "mention":      mention,
        "citation":      citation,
        "position_score": pos,
        "sentiment":     sentiment,
```

```

        "response_format": fmt,
        "response_length": len(text),
        "comp_citations": comp_citations,
        "comp_mentions": comp_mentions,
    })

metrics_df = pd.concat(
    [responses_df, responses_df.apply(extract_metrics, axis=1)],
    axis=1,
)

```

Coerces sentiment to float, replacing any residual strings with 0.0.

```

metrics_df["sentiment"] = pd.to_numeric(metrics_df["sentiment"],
errors="coerce").fillna(0.0)

```

Persists extracted metrics to a mode-prefixed parquet file.

```

metrics_df.to_parquet(f"
{ANALYSIS_MODE}_metrics_{RUN_DATE}.parquet", index=False)

```

Assigns metrics_df to the correctly named global so both modes coexist in memory.

```

if ANALYSIS_MODE == "multi":
    multi_metrics_df = metrics_df.copy()
else:
    single_metrics_df = metrics_df.copy()

```

Confirms record count and shows a five-row sample for sanity checking.

```

print(f"Metrics extracted [{ANALYSIS_MODE}]: {len(metrics_df):,}
records")
print(metrics_df[[
    "question", "platform", "session",
    "mention", "citation", "position_score", "sentiment",
]].head(5).to_string(index=False))

```

Outputs for both modes:

Metrics extracted [multi]: 880 records

question	platform	session	mention	citation	position_score
why is my construction business data spread across so many systems	openai		0	False	False
0.0	0.0				
why is my construction business data spread across so many systems	perplexity		0	False	False
0.0	0.0				
why is my construction business data spread across so many systems	gemini		0	False	False
0.0	0.0				
how do I get a single view of job costs across my construction trades business	openai		0	False	False
0.0	0.0				
how do I get a single view of job costs across my construction trades business	perplexity		0	False	False
0.0	0.0				

Metrics extracted [single]: 880 records

question	platform	session	mention	citation	position_score
best ERP for construction trades companies	perplexity		0	False	False
0.0					0.0
how does NetSuite handle job costing for construction	openai		1	False	False
0.0	0.0				
why is my construction business data spread across so many systems	openai		0	False	False
0.0					0.0
can NetSuite manage subcontractor payments for construction trades	openai		1	False	False
0.0					0.0
why is my construction business data spread across so many systems	gemini		0	False	False
0.0					0.0

Both modes extracted cleanly at 880 records each. The all-zero sample rows are expected as these are generic problem-cluster queries where no specific vendor would be mentioned, with the visibility signal concentrated in the brands and solutions clusters.

With the metrics calculated, you can now aggregate these at different levels for analysis and visualization.

Query-Level

Aggregate raw metrics to query for each platform with one row per (cluster, question, platform) combination as query-level is the most granular reportable unit. This will show which specific questions the brand appears on and which it doesn't and the ground truth for all higher-level aggregations.

This is achieved by using groupby on (cluster, question, platform) calculating the mean across sessions to give rates, min/max sentiment for the range of framing.

```
ANALYSIS_MODE = "multi"    # or "single"
metrics_df     = multi_metrics_df if ANALYSIS_MODE == "multi" else
single_metrics_df
```

Aggregates metrics_df to one row per (cluster, question, platform), computing rates and distributions across all sessions for that combination.

```
query_summary_df = (
    metrics_df
    .groupby(["cluster", "question", "platform"])
    .agg(
        mention_rate      = ("mention",      "mean"),
        citation_rate      = ("citation",      "mean"),
        mean_position      = ("position_score", "mean"),
        mean_sentiment      = ("sentiment",    "mean"),
        min_sentiment      = ("sentiment",    "min"),
        max_sentiment      = ("sentiment",    "max"),
        format_list        = ("response_format", lambda s: (s ==
"list").mean()),
        format_table       = ("response_format", lambda s: (s ==
"table").mean()),
        format_paragraph   = ("response_format", lambda s: (s ==
"paragraph").mean()),
        mean_comp_mentions = ("comp_mentions", "mean"),
        mean_comp_citations = ("comp_citations", "mean"),
        mean_length        = ("response_length", "mean"),
```

```

        session_count      = ("session",      "count"),
    )
    .round(4)
    .reset_index()
)

if ANALYSIS_MODE == "multi":
    multi_query_summary_df = query_summary_df.copy()
else:
    single_query_summary_df = query_summary_df.copy()

query_summary_df.to_parquet(f"
{ANALYSIS_MODE}_query_summary_{RUN_DATE}.parquet", index=False)
query_summary_df.to_csv(f"
{ANALYSIS_MODE}_query_summary_{RUN_DATE}.csv",
index=False)

print(f"Query-level summary [{ANALYSIS_MODE}]:
{len(query_summary_df):,} rows")
print(query_summary_df[[
    "question", "platform", "mention_rate", "citation_rate",
    "mean_position", "mean_sentiment",
]].head(1).to_string(index=False))

```

Output

Query-level summary [multi]: 88 rows

question	platform	mention_rate	citation_rate	mean_position	mean_sentiment
how do I connect field operations data with my back office in construction	claude	0.0	0.0	0.0	0.0
how do I connect field operations data with my back office in construction	gemini	0.0	0.0	0.0	0.0
how do I connect field operations data with my back office in construction	openai	0.0	0.0	0.0	0.0
how do I connect field operations data with my back office in construction	perplexity	0.0	0.0	0.0	0.0

Query-level summary [single]: 88 rows

question

platform	mention_rate	citation_rate	mean_position	mean_sentiment
	best NetSuite consultants for construction trades			
claude	0.0	0.0	0.0000	0.0000
	best NetSuite consultants for construction trades			
gemini	0.0	0.0	0.0000	0.0000
	best NetSuite consultants for construction trades			
openai	0.3	0.6	0.0903	0.1966
	best NetSuite consultants for construction trades			
perplexity	0.0	0.2	0.0000	0.0000

The single mode sample shows OpenAI returning 30% mention and 60% citation on “best NetSuite consultants” while the same query scores zero on Claude, Gemini and Perplexity.

The heatmap grid to visualize query visibility, which you’ll see later in this section, is plotted for each platform scores for each metric,

to compare multi-question session mode against single-question mode.

The heatmap encodes sparsity where zero cells are white, non-zero cells show colour intensity proportional to the metric value, making the few visibility signals immediately visible.

Ordering by cluster then ascending metric value within cluster highlights the highest-signal questions at the bottom of each cluster group, with horizontal dividers making cluster boundaries more explicit.

Loads Seaborn for the heatmap function and colour palette generation.

```
import seaborn as sns
```

four aggregated column names and their display labels used consistently across all plot cells.

```
METRICS = ["mention_rate", "citation_rate",
            "mean_position", "mean_sentiment"]
```

```
METRIC_LABELS = ["Mention Rate", "Citation Rate", "Mean
Position", "Mean Sentiment"]
```

Explicit platform order controlling column sequence in all heatmaps and charts.

```
PLATFORMS      = ["claude", "gemini", "openai", "perplexity"]
```

Explicit cluster order controlling row sequence: problem → solutions → brands — reflecting ascending commercial search intent.

```
CLUSTERS        = ["problem", "solutions", "brands"]
CLUSTER_ORDER   = ["problem", "solutions", "brands"]
```

Maps mode names to brand colours so multi and single are visually consistent across all plot cells.

```
COLOURS         = {
    "multi":  BRAND_COLOURS["primary"],      # #069492
    "single": BRAND_COLOURS["secondary"],    # #29265b
}
```

Draws a heatmap with dynamic font colour — white text on dark cells above 60% of vmax, dark text on light cells.

```
def annotated_heatmap(ax, pivot, cmap, vmax, fmt=".2f",
**heatmap_kwargs):
    sns.heatmap(
        pivot,
        ax=ax,
        cmap=cmap,
        annot=False,
        linewidths=0.4,
        linecolor="#e0e0e0",
        cbar=True,
        vmin=0,
        vmax=vmax,
        **heatmap_kwargs,
    )
    threshold = vmax * 0.6
    for i in range(pivot.shape[0]):
```

```

        for j in range(pivot.shape[1]):
            val = pivot.iloc[i, j]
            colour = "white" if val >= threshold else "#333333"
            ax.text(
                j + 0.5, i + 0.5,
                f"{val:{fmt}}",
                ha="center", va="center",
                fontsize=7,
                color=colour,
                fontweight="bold" if val >= threshold else
"normal",
            )

```

Creates a grid of subplots with one row per metric and two columns for multi vs single.

```

fig, axes = plt.subplots(
    nrows=len(METRICS),
    ncols=2,
    figsize=(20, 28),
    constrained_layout=True,
)

```

Sets the chart title including client domain and run date for traceability.

```

fig.suptitle(
    f"Query × Platform Visibility Heatmap | {YOUR_DOMAIN} | {RUN_DATE}",
    fontsize=13, fontweight="bold", y=1.01,
)

```

Iterates over every metric and mode combination, building and rendering one heatmap per subplot.

```

for row_idx, (metric, metric_label) in enumerate(zip(METRICS,
METRIC_LABELS)):
    for col_idx, (mode, summary_df) in enumerate([
        ("multi", multi_query_summary_df),
        ("single", single_query_summary_df),
    ]):
        ax = axes[row_idx][col_idx]

        question_means = (

```

```

summary_df
    .groupby(["cluster", "question"])[metric]
    .mean()
    .reset_index()
    .rename(columns={metric: "mean_val"})
)

ordered_questions = []
for cluster in CLUSTER_ORDER:
    cluster_qs = (
        question_means[question_means["cluster"] ==
cluster]
        .sort_values("mean_val", ascending=True)
["question"]
        .tolist()
    )
    ordered_questions.extend(cluster_qs)

pivot = (
    summary_df
    .pivot_table(index="question", columns="platform",
values=metric, aggfunc="mean")
    .reindex(index=ordered_questions, columns=PLATFORMS)
    .fillna(0.0)
)

cluster_lookup = (
    summary_df[["question", "cluster"]]
    .drop_duplicates()
    .set_index("question")["cluster"]
    .to_dict()
)

yticklabels = [
    f"[{cluster_lookup.get(q, '?')[:3].upper()}]"
{q[:55]}"]
    for q in ordered_questions
]

vmax = pivot.values.max() if pivot.values.max() > 0 else
1.0

annotated_heatmap(
    ax=ax,
    pivot=pivot,
    cmap=sns.light_palette(
        COLOURS["multi"] if mode == "multi" else
COLOURS["single"],
        as_cmap=True,

```

```

        ),
        vmax=vmax,
        fmt=".2f",
        yticklabels=yticklabels,
    )

    current_pos = 0
    for cluster in CLUSTER_ORDER:
        cluster_count = sum(
            1 for q in ordered_questions
            if cluster_lookup.get(q) == cluster
        )
        current_pos += cluster_count
        if current_pos < len(ordered_questions):
            ax.axhline(current_pos, color="#333333",
linewidth=1.5, zorder=5)

        ax.set_title(f"{metric_label} [{mode}]", fontsize=9,
fontWeight="bold")
        ax.set_xlabel("")
        ax.set_ylabel("")
        ax.tick_params(axis="x", labelsize=8, rotation=30)
        ax.tick_params(axis="y", labelsize=7, rotation=0)

```

Saves the figure to a date-stamped PNG, renders it inline, and confirms the file path (figure 6-1).

```

plt.savefig(f"query_heatmap_{RUN_DATE}.png", dpi=150,
bbox_inches="tight")
plt.show()
print(f"Saved query_heatmap_{RUN_DATE}.png")

```


Figure 2-1. Saved query_heatmap_2026-06-11.png

The graphic shows the problem and solutions search intents are white across all platforms and both modes, with zero visibility.

The signal is entirely concentrated in the brands cluster at the bottom, where multi-mode shows Claude leading the citation rate (0.50) and mean position (0.54).

The single-mode reveals a different pattern with OpenAI dominating on citation rate (0.60) and mention rate (0.30) on the top competitive query. So the prompt mode and cold-query retrieval highlight the brand on different platforms.

Search Intent

Aggregate query-level summaries to cluster for each AI platform as search intent scores reveal visibility patterns by whether the brand appears on problem queries, solution queries or brand queries which is the primary diagnostic unit for GEO strategy.

We then group by cluster and platform to produce a cluster-level summary, adding a coverage metric that shows the proportion of questions in each search intent cluster where the brand was mentioned at least once across any session.

The code then selects the appropriate metrics DataFrame depending on whether the analysis ran in conversational or single-question mode.

```
ANALYSIS_MODE      = "multi"    # or "single" multi
query_summary_df = multi_query_summary_df if ANALYSIS_MODE ==
"multi" else single_query_summary_df
```

Computes the proportion of questions per cluster where the brand was mentioned at least once across any session, measuring breadth of visibility rather than depth.

```
coverage_df = (
    metrics_df
```

```

        .groupby(["cluster", "question"])["mention"]
        .any()
        .reset_index()
        .groupby("cluster")["mention"]
        .mean()
        .rename("coverage")
        .round(4)
        .reset_index()
    )

```

Aggregates query_summary_df to one row per (cluster, platform), joining coverage to give a complete cluster-level visibility profile.

```

cluster_summary_df = (
    query_summary_df
    .groupby(["cluster", "platform"])
    .agg(
        mention_rate      = ("mention_rate",      "mean"),
        citation_rate      = ("citation_rate",      "mean"),
        mean_position      = ("mean_position",      "mean"),
        mean_sentiment      = ("mean_sentiment",      "mean"),
        mean_comp_mentions = ("mean_comp_mentions", "mean"),
        mean_comp_citations = ("mean_comp_citations", "mean"),
        question_count      = ("question",          "nunique"),
    )
    .round(4)
    .reset_index()
    .merge(coverage_df, on="cluster", how="left")
)

if ANALYSIS_MODE == "multi":
    multi_cluster_summary_df = cluster_summary_df.copy()
else:
    single_cluster_summary_df = cluster_summary_df.copy()

```

Persists the cluster summary in binary columnar format for efficient loading in a comparison later.

```

cluster_summary_df.to_parquet(f"
{ANALYSIS_MODE}_cluster_summary_{RUN_DATE}.parquet", index=False)
cluster_summary_df.to_csv(f"
{ANALYSIS_MODE}_cluster_summary_{RUN_DATE}.csv",
index=False)

```

Exports a human-readable version for sharing or manual inspection outside the notebook.

```
print(f"Cluster-level summary [{ANALYSIS_MODE}]:")
print(cluster_summary_df.to_string(index=False))
```

Output

```
Cluster-level summary [multi]:
  cluster platform mention_rate citation_rate mean_position
mean_sentiment mean_comp_mentions mean_comp_citations
question_count coverage
  brands      claude      0.2667      0.3000      0.1814
0.2646              0.4333      0.1667              3
1.0
  brands      gemini      0.1333      0.1000      0.0704
0.1328              0.4000      0.1667              3
1.0
  brands      openai      0.0000      0.0000      0.0000
0.0000              0.0000      0.0000              3
1.0
  brands perplexity      0.0000      0.0000      0.0000
0.0000              0.0000      0.0000              3
1.0
  problem      claude      0.0000      0.0000      0.0000
0.0000              0.0000      0.0000              10
0.0
  problem      gemini      0.0000      0.0000      0.0000
0.0000              0.0000      0.0000              10
0.0
  problem      openai      0.0000      0.0000      0.0000
0.0000              0.0000      0.0000              10
0.0
  problem perplexity      0.0000      0.0000      0.0000
0.0000              0.0000      0.0000              10
0.0
solutions      claude      0.0000      0.0222      0.0000
0.0000              0.0667      0.3333              9
0.0
solutions      gemini      0.0000      0.0222      0.0000
0.0000              0.0778      0.1333              9
0.0
solutions      openai      0.0000      0.0000      0.0000
0.0000              0.0000      0.0000              9
0.0
```

solutions perplexity	0.0000	0.0000	0.0000
0.0000	0.0000	0.0000	9
0.0			

Cluster-level summary [single]:

cluster	platform	mention_rate	citation_rate	mean_position
mean_sentiment	mean_comp_mentions	mean_comp_citations		
question_count	coverage			
brands	claude	0.2000	0.3000	0.1020
0.1516	0.7333		0.7667	3
1.0				
brands	gemini	0.2000	0.2333	0.0709
0.1993	0.7667		0.7000	3
1.0				
brands	openai	0.1333	0.2667	0.0466
0.0988	0.3333		0.4000	3
1.0				
brands perplexity		0.0333	0.1000	0.0258
0.0330	0.0000		0.0000	3
1.0				
problem	claude	0.0000	0.0000	0.0000
0.0000	0.0000		0.0000	10
0.0				
problem	gemini	0.0000	0.0000	0.0000
0.0000	0.0300		0.0100	10
0.0				
problem	openai	0.0000	0.0000	0.0000
0.0000	0.0000		0.0000	10
0.0				
problem perplexity		0.0000	0.0000	0.0000
0.0000	0.0000		0.0000	10
0.0				
solutions	claude	0.0000	0.0222	0.0000
0.0000	0.1000		0.6111	9
0.0				
solutions	gemini	0.0000	0.1222	0.0000
0.0000	0.1444		0.7333	9
0.0				
solutions	openai	0.0000	0.0000	0.0000
0.0000	0.0000		0.0000	9
0.0				
solutions perplexity		0.0000	0.1222	0.0000
0.0000	0.2111		0.6444	9
0.0				

The single mode surfaces the “client” Rand Group on OpenAI (13.3% mention, 26.7% citation) and Perplexity (3.3% mention, 10% citation) where multi mode shows zero.

This confirms that the single mode used by 3rd party tools likely flatters the visibility of the brands they track on their platforms where substantially higher competitor citation rates (mean_comp_citations 0.77 on Claude vs 0.17 in multi), shows cold queries trigger more competitive retrieval overall and the brand is competing in a more crowded single-query results set.

```
fig, axes = plt.subplots(
    nrows=len(METRICS),
    ncols=2,
    figsize=(14, 16),
    constrained_layout=True,
)
fig.suptitle(
    f"Cluster × Platform Visibility Heatmap | {YOUR_DOMAIN} | {RUN_DATE}",
    fontsize=13, fontweight="bold", y=1.01,
)

for row_idx, (metric, metric_label) in enumerate(zip(METRICS,
METRIC_LABELS)):
    for col_idx, (mode, summary_df) in enumerate([
        ("multi", multi_cluster_summary_df),
        ("single", single_cluster_summary_df),
    ]):
        ax = axes[row_idx][col_idx]

        pivot = (
            summary_df
            .pivot_table(index="cluster", columns="platform",
values=metric, aggfunc="mean")
            .reindex(index=CLUSTER_ORDER, columns=PLATFORMS)
            .fillna(0.0)
        )

        vmax = pivot.values.max() if pivot.values.max() > 0 else
1.0

        annotated_heatmap(
            ax=ax,
            pivot=pivot,
```

```

        cmap=sns.light_palette(
            COLOURS["multi"] if mode == "multi" else
COLOURS["single"],
            as_cmap=True,
        ),
        vmax=vmax,
        fmt=".3f",
    )

    # horizontal dividers between clusters
    for i in range(1, len(CLUSTER_ORDER)):
        ax.axhline(i, color="#333333", linewidth=1.0,
zorder=5)

        ax.set_title(f"{metric_label} [{mode}]", fontsize=9,
fontweight="bold")
        ax.set_xlabel("")
        ax.set_ylabel("")
        ax.tick_params(axis="x", labelsiz=8, rotation=30)
        ax.tick_params(axis="y", labelsiz=9, rotation=0)

plt.savefig(f"cluster_heatmap_{RUN_DATE}.png", dpi=150,
bbox_inches="tight")
plt.show()
print(f"Saved cluster_heatmap_{RUN_DATE}.png")

```

Cluster × Platform Visibility Heatmap | randgroup.com | 2026-06-11

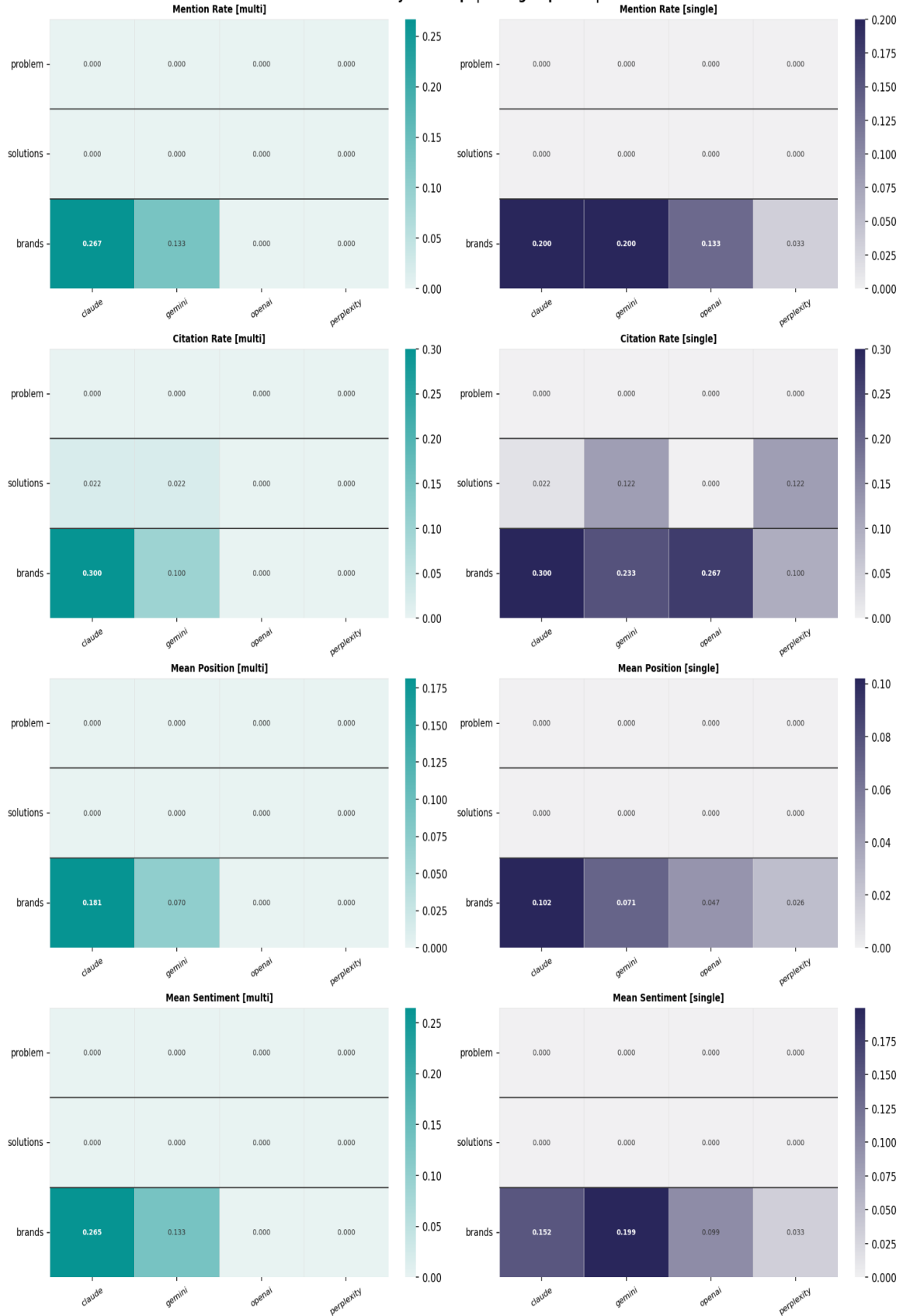


Figure 2-2. Saved cluster_heatmap_2026-06-11.png

The heatmap helpfully shows multi mode visibility is concentrated mainly in Claude (with a bit in Gemini) in the brands cluster, while single mode distributes signal across all four platforms with OpenAI and Gemini emerging as significant citation sources (0.267 and 0.233) that are entirely invisible in conversational context.

The solutions cluster shows a similar pattern with Gemini and Perplexity citation rates jumping from 0.022 to 0.122 in single mode, suggesting these platforms actively retrieve Rand Group content on NetSuite queries but conversational priming suppresses that retrieval.

AI Search Platform

We then aggregate metrics to platform level across all clusters and questions. This produces a single score per platform showing how consistently each AI engine surfaces the brand which is the primary input for deciding which platforms to prioritise in a GEO programme.

The code then selects the appropriate metrics DataFrame depending on whether the analysis ran in conversational or single-question mode as before.

```
ANALYSIS_MODE = "multi"    # or "single"
metrics_df     = multi_metrics_df if ANALYSIS_MODE == "multi" else
single_metrics_df
```

Aggregates metrics_df to one row per platform across all clusters and questions, giving the top-line visibility score for each AI system.

```
platform_summary_df = (
    metrics_df
    .groupby("platform")
    .agg(
        mention_rate      = ("mention",      "mean"),
        citation_rate      = ("citation",     "mean"),
        mean_position      = ("position_score", "mean"),
        mean_sentiment     = ("sentiment",    "mean"),
```

```

        mean_comp_mentions = ("comp_mentions", "mean"),
        mean_comp_citations = ("comp_citations", "mean"),
        response_count      = ("session",       "count"),
    )
    .round(4)
    .reset_index()
)

if ANALYSIS_MODE == "multi":
    multi_platform_summary_df = platform_summary_df.copy()
else:
    single_platform_summary_df = platform_summary_df.copy()

platform_summary_df.to_parquet(f"
{ANALYSIS_MODE}_platform_summary_{RUN_DATE}.parquet",
index=False)
platform_summary_df.to_csv(f"
{ANALYSIS_MODE}_platform_summary_{RUN_DATE}.csv",
index=False)

print(f"Platform-level summary [{ANALYSIS_MODE}]:")
print(platform_summary_df.to_string(index=False))

```

Output

```

Platform-level summary [multi]:
  platform mention_rate citation_rate mean_position
mean_sentiment mean_comp_mentions mean_comp_citations
response_count
  claude          0.0364          0.0500          0.0247
0.0361          0.0864          0.1591          220
  gemini          0.0182          0.0227          0.0096
0.0181          0.0864          0.0773          220
  openai          0.0000          0.0000          0.0000
0.0000          0.0000          0.0000          220
perplexity        0.0000          0.0000          0.0000
0.0000          0.0000          0.0000          220

```

```

Platform-level summary [single]:
  platform mention_rate citation_rate mean_position
mean_sentiment mean_comp_mentions mean_comp_citations
response_count
  claude          0.0273          0.0500          0.0139
0.0207          0.1409          0.3545          220
  gemini          0.0273          0.0818          0.0097

```

0.0272	0.1773	0.4000	220
openai	0.0182	0.0364	0.0064
0.0135	0.0455	0.0545	220
perplexity	0.0045	0.0636	0.0035
0.0045	0.0864	0.2636	220

OpenAI and Perplexity, invisible in multi mode, show measurable visibility in isolation (OpenAI 1.82% mention, 3.64% citation; Perplexity 0.45% mention, 6.36% citation).

Gemini's citation rate triples from 2.27% to 8.18% in single mode and the sudden rise in mean_comp_citations across all platforms in single mode confirms cold queries trigger substantially more competitive retrieval, meaning the brand is being cited alongside more competitors per response than in conversational sessions.

```
fig, axes = plt.subplots(
    nrows=len(METRICS),
    ncols=2,
    figsize=(14, 16),
    constrained_layout=True,
)
fig.suptitle(
    f"Cluster × Platform Visibility Heatmap | {RUN_DATE}",
    fontsize=13, fontweight="bold", y=1.01,
)

for row_idx, (metric, metric_label) in enumerate(zip(METRICS,
METRIC_LABELS)):
    for col_idx, (mode, summary_df) in enumerate([
        ("multi", multi_cluster_summary_df),
        ("single", single_cluster_summary_df),
    ]):
        ax = axes[row_idx][col_idx]

        pivot = (
            summary_df
            .pivot_table(index="cluster", columns="platform",
values=metric, aggfunc="mean")
            .reindex(index=CLUSTER_ORDER, columns=PLATFORMS)
            .fillna(0.0)
        )

        vmax = pivot.values.max() if pivot.values.max() > 0 else
```

1.0

```
annotated_heatmap(  
    ax=ax,  
    pivot=pivot,  
    cmap=sns.light_palette(  
        COLOURS["multi"] if mode == "multi" else  
COLOURS["single"],  
        as_cmap=True,  
    ),  
    vmax=vmax,  
    fmt=".3f",  
)  
  
# horizontal dividers between clusters  
for i in range(1, len(CLUSTER_ORDER)):  
    ax.axhline(i, color="#333333", linewidth=1.0,  
zorder=5)  
  
    ax.set_title(f"{metric_label} [{mode}]", fontsize=9,  
fontweight="bold")  
    ax.set_xlabel("")  
    ax.set_ylabel("")  
    ax.tick_params(axis="x", labelsiz=8, rotation=30)  
    ax.tick_params(axis="y", labelsiz=9, rotation=0)  
  
plt.savefig(f"cluster_heatmap_{RUN_DATE}.png", dpi=150,  
bbox_inches="tight")  
plt.show()  
print(f"Saved cluster_heatmap_{RUN_DATE}.png")
```

Output

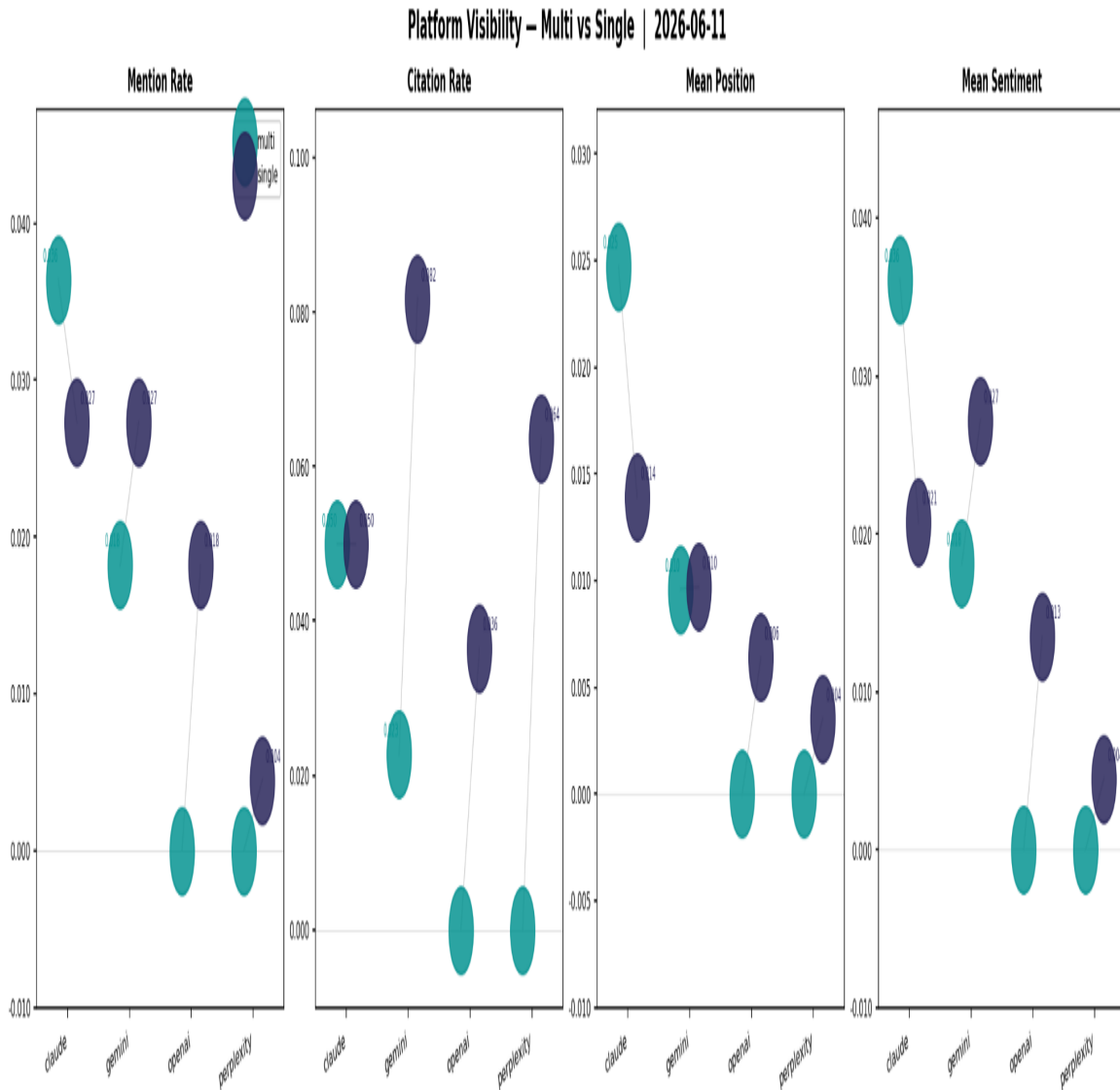


Figure 2-3. Platform visibility by performance metric and prompt mode

The teal (multi) sits at zero for OpenAI and Perplexity across all four metrics while navy (single) rises above zero on both, and the connector lines between modes on Claude and Gemini show multi outperforming single on mention rate and mean position while single dominates on citation rate, particularly Gemini’s 0.082 citation rate in single mode against 0.050 in multi.

Overall AI Visibility

Now let's compute the overall AI visibility summary across all platforms, clusters and questions including central tendency, dispersion and range statistics for the four core visibility metrics.

The cluster and platform summaries show where visibility exists but the overall summary is the top-line longitudinal KPI with a single comparable number per run date that tracks whether GEO interventions are delivering performance. The dispersion statistics reveal whether visibility is concentrated on a few queries or broad across the prompts.

```
ANALYSIS_MODE = "multi"    # or "single"
metrics_df     = multi_metrics_df if ANALYSIS_MODE == "multi" else
single_metrics_df

RAW_METRICS      = ["mention", "citation", "position_score",
"sentiment"]
RAW_METRIC_LABELS = ["Mention Rate", "Citation Rate", "Mean
Position", "Mean Sentiment"]
```

Computes mean, std, min, median, max, quartiles and zero rate for each raw metric across all 880 records, characterising the full distribution of AI visibility scores.

```
ai_visibility_distribution_df = (
    metrics_df[RAW_METRICS]
    .astype({
        "mention": float,
        "citation": float,
    })
    .agg([
        "mean",
        "std",
        "min",
        "median",
        "max",
        lambda x: x.quantile(0.25),
        lambda x: x.quantile(0.75),
        lambda x: (x == 0).mean(),
    ])
    .round(4)
)
```

Replaces the default lambda repr index labels with readable stat names:
mean, std, min, median, max, q25, q75, zero_rate.

```
ai_visibility_distribution_df.index = [  
    "mean", "std", "min", "median", "max", "q25", "q75",  
    "zero_rate"  
]
```

Cast boolean columns to float for numeric operations

```
mention_s = metrics_df["mention"].astype(float)  
citation_s = metrics_df["citation"].astype(float)  
position_s = metrics_df["position_score"]  
sentiment_s = metrics_df["sentiment"]
```

Single-row summary of the run combining mode, date, response count, and all four metrics' mean, std, max and zero rate in one exportable record for longitudinal tracking.

```
ai_visibility_summary_df = pd.DataFrame([  
    "mode": ANALYSIS_MODE,  
    "run_date": RUN_DATE,  
    "response_count": len(metrics_df),  
    "mention_rate": round(float(mention_s.mean()),  
4),  
    "citation_rate": round(float(citation_s.mean()),  
4),  
    "mean_position": round(float(position_s.mean()),  
4),  
    "mean_sentiment": round(float(sentiment_s.mean()),  
4),  
    "mention_std": round(float(mention_s.std()),  
4),  
    "citation_std": round(float(citation_s.std()),  
4),  
    "position_std": round(float(position_s.std()),  
4),  
    "sentiment_std": round(float(sentiment_s.std()),  
4),  
    "mention_max": round(float(mention_s.max()),  
4),  
])
```

```

        "citation_max":      round(float(citation_s.max()),
4),
        "position_max":     round(float(position_s.max()),
4),
        "sentiment_max":    round(float(sentiment_s.max()),
4),
        "mention_zero_rate": round(float((mention_s == 0).mean()),
4),
        "citation_zero_rate": round(float((citation_s == 0).mean()),
4),
        "position_zero_rate": round(float((position_s == 0).mean()),
4),
        "sentiment_zero_rate": round(float((sentiment_s ==
0).mean()), 4),
    })

if ANALYSIS_MODE == "multi":
    multi_ai_visibility_summary_df =
ai_visibility_summary_df.copy()
    multi_ai_visibility_dist_df =
ai_visibility_distribution_df.copy()
else:
    single_ai_visibility_summary_df =
ai_visibility_summary_df.copy()
    single_ai_visibility_dist_df =
ai_visibility_distribution_df.copy()

ai_visibility_summary_df.to_parquet(f"
{ANALYSIS_MODE}_ai_visibility_summary_{RUN_DATE}.parquet",
index=False)
ai_visibility_summary_df.to_csv(f"
{ANALYSIS_MODE}_ai_visibility_summary_{RUN_DATE}.csv",
index=False)
ai_visibility_distribution_df.to_csv(
    f"{ANALYSIS_MODE}_ai_visibility_distribution_{RUN_DATE}.csv"
)

print(f"Overall AI visibility summary [{ANALYSIS_MODE}]:")
print(ai_visibility_summary_df.to_string(index=False))
print(f"\nAI visibility distribution [{ANALYSIS_MODE}]:")
print(ai_visibility_distribution_df.to_string())

```

Output:

```
Overall AI visibility summary [multi]:
mode    run_date    response_count    mention_rate    citation_rate
mean_position    mean_sentiment    mention_std    citation_std
position_std    sentiment_std    mention_max    citation_max
position_max    sentiment_max    mention_zero_rate
citation_zero_rate    position_zero_rate    sentiment_zero_rate
multi 2026-06-11                880                0.0136                0.0182
0.0086                0.0135                0.116                0.1337                0.0762
0.1153                1.0                1.0                0.9546                0.9991
0.9864                0.9818                0.9864
0.9864
```

```
AI visibility distribution [multi]:
          mention    citation    position_score    sentiment
mean      0.0136      0.0182          0.0086      0.0135
std       0.1160      0.1337          0.0762      0.1153
min       0.0000      0.0000          0.0000      0.0000
median    0.0000      0.0000          0.0000      0.0000
max       1.0000      1.0000          0.9546      0.9991
q25       0.0000      0.0000          0.0000      0.0000
q75       0.0000      0.0000          0.0000      0.0000
zero_rate 0.9864      0.9818          0.9864      0.9864
```

98.6% of the mention records are zero showing visibility is extremely sparse. The Max of 1.0 on mention and citation confirms the brand achieves full visibility on at least one (question, session, platform) combination. The Median of 0.0 across all metrics confirms more than half of all responses contain no brand signal

Zero rate on q25 and q75 confirms the brand is absent in at least 75% of all responses

For distribution, the violin and strip plots (figure 6-4) are used across the four core visibility metrics to compare multi-question session mode against single-question mode, with mean, median, max and std annotated directly on the chart.

Violin plot visuals will surface this bimodal structure while strip plots show the exact count of non-zero observations. The violin plot renders distribution shape; scatter with jitter overlays individual query-level data points with the mean as solid horizontal line, median as dashed μ , max and

std annotated per mode per metric. Multi and single compared side by side on the same axis

```
from matplotlib.lines import Line2D

fig, axes = plt.subplots(
    nrows=1,
    ncols=len(METRICS),
    figsize=(18, 6),
    constrained_layout=True,
)
fig.suptitle(
    f"Overall AI Visibility Distribution – Multi vs Single |
{YOUR_DOMAIN} | {RUN_DATE}",
    fontsize=13, fontweight="bold",
)

for col_idx, (metric, metric_label) in enumerate(zip(METRICS,
METRIC_LABELS)):
    ax = axes[col_idx]

    multi_vals = multi_query_summary_df[metric].values
    single_vals = single_query_summary_df[metric].values

    # violin plots
    for pos, vals, colour in [
        (0, multi_vals, COLOURS["multi"]),
        (1, single_vals, COLOURS["single"]),
    ]:
        parts = ax.violinplot(
            vals,
            positions=[pos],
            widths=0.6,
            showmedians=False,
            showextrema=False,
        )
        for pc in parts["bodies"]:
            pc.set_facecolor(colour)
            pc.set_alpha(0.4)

    # strip plots
    np.random.seed(42)
    for pos, vals, colour in [
        (0, multi_vals, COLOURS["multi"]),
        (1, single_vals, COLOURS["single"]),
    ]:
```

```

        jitter = np.random.uniform(-0.08, 0.08, size=len(vals))
        ax.scatter(
            pos + jitter, vals,
            color=colour, alpha=0.6, s=20, zorder=3,
        )

    # mean, median lines and annotations
    for pos, vals, colour in [
        (0, multi_vals, COLOURS["multi"]),
        (1, single_vals, COLOURS["single"]),
    ]:
        mean = np.mean(vals)
        median = np.median(vals)

        ax.hlines(mean, pos - 0.2, pos + 0.2, colors=colour,
            linewidth=2.0, zorder=4)
        ax.hlines(median, pos - 0.2, pos + 0.2, colors=colour,
            linewidth=1.0, zorder=4, linestyle="--")

        ax.annotate(
            f" $\mu$ ={mean:.3f}",
            (pos + 0.22, mean),
            fontsize=6.5, color=colour, va="center",
        )

        ax.annotate(
            f"max={np.max(vals):.3f}",
            (pos, np.max(vals)),
            textcoords="offset points", xytext=(0, 6),
            fontsize=6, color=colour, ha="center",
        )

        ax.annotate(
            f"std={np.std(vals):.3f}",
            (pos, -0.01),
            textcoords="offset points", xytext=(0, -14),
            fontsize=6, color=colour, ha="center",
        )

    ax.set_title(metric_label, fontsize=10, fontweight="bold")
    ax.set_xticks([0, 1])
    ax.set_xticklabels(["multi", "single"], fontsize=9)

    ax.yaxis.set_major_formatter(mticker.FormatStrFormatter("%.2f"))
    ax.tick_params(axis="y", labelsize=8)
    ax.set_xlim(-0.5, 1.5)
    ax.axhline(0, color="#ddddd", linewidth=0.5, zorder=1)

    if col_idx == 0:
        ax.legend(

```

```

        handles=[
            Line2D([0], [0], color="#555555", linewidth=2.0,
label="mean"),
            Line2D([0], [0], color="#555555", linewidth=1.0,
linestyle="--", label="median"),
        ],
        fontsize=7,
        loc="upper right",
    )

plt.savefig(f"overall_distribution_{RUN_DATE}.png", dpi=150,
bbox_inches="tight")
plt.show()
print(f"Saved overall_distribution_{RUN_DATE}.png")

```

Overall AI Visibility Distribution – Multi vs Single | randgroup.com | 2026-06-11

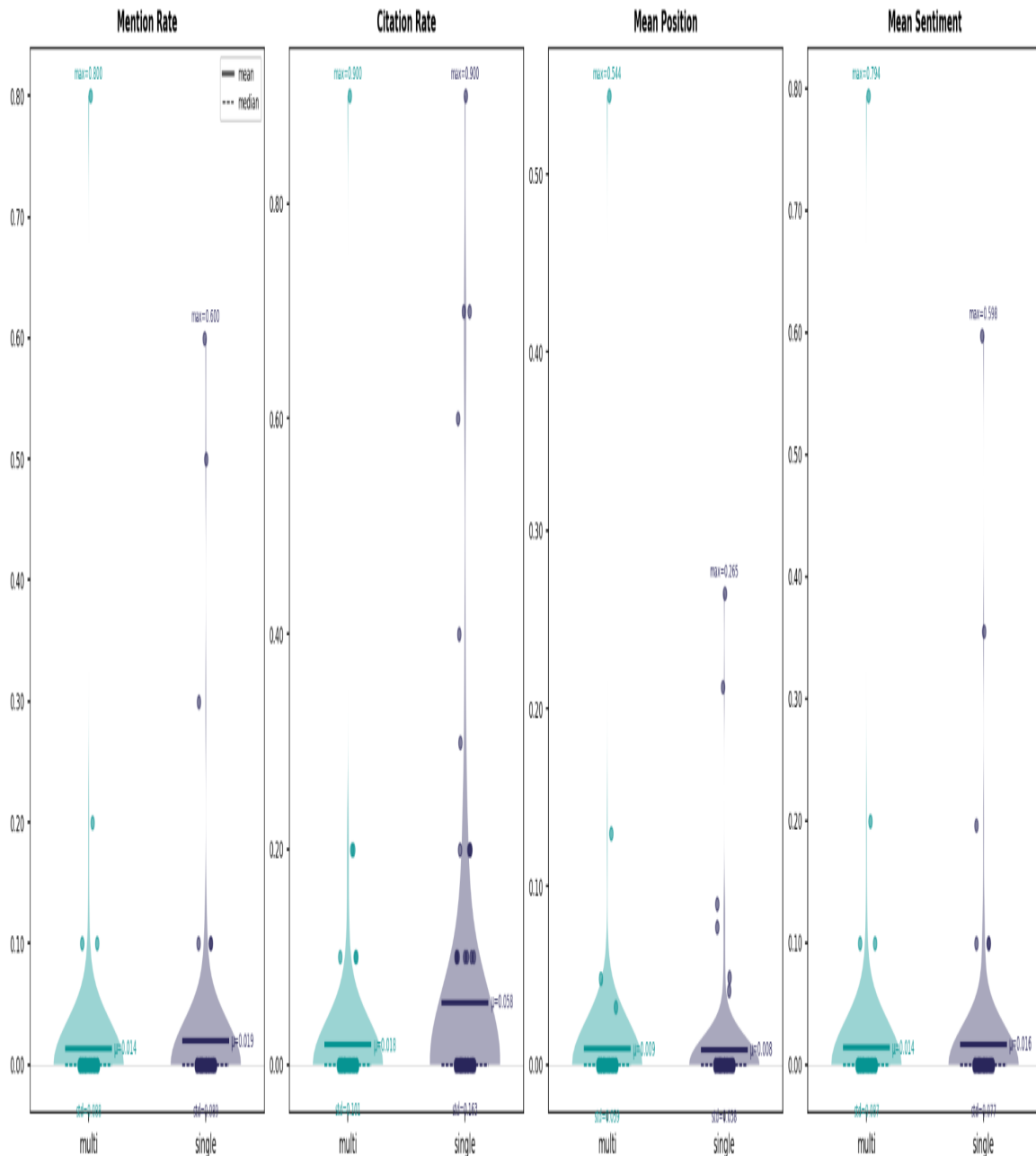


Figure 2-4. Violin plots of AI visibility by performance metric and prompt mode

The zero-massed base as shown in figure 6-4 confirms brand absence across 98% of responses. The single mode's wider citation rate distribution (std 0.163 vs 0.101) shows broader but shallower retrieval, while multi mode's taller mention rate tail reflects visibility concentrated on Claude in conversational mode.

Summary

This chapter covered building a Python pipeline to measure brand visibility across AI search platforms, establishing a repeatable baseline for GEO performance tracking.

To measure AI search visibility we:

- Built platform adapters to query OpenAI, Gemini, Claude and Perplexity across two measurement modes — conversational multi-question sessions and isolated single-question calls
- Extracted six visibility metrics from raw responses: mention rate, citation rate, position score, sentiment, competitive displacement and response format
- Aggregated metrics to query, cluster and platform level, revealing that brand visibility is platform-specific and mode-dependent — conversational context suppresses retrieval on some platforms while amplifying it on others

In the next chapter, we use this measurement infrastructure to attribute GEO impact and quantify the effect of content interventions on AI search visibility.

About the Author

Andreas Voniatis is the founder and CEO of Artios.io, an AI search optimization (GEO) agency that helps B2B companies get recommended by ChatGPT, Claude, and other AI platforms with guaranteed results. A certified accountant turned data scientist, Andreas built his first Large Language Model in 2019, three years before ChatGPT's launch, giving him unique expertise in how AI systems process and recommend information. He is the author of "Data Driven SEO with Python" (Springer, 2022), Data Science For SEO (O'Reilly Media, 2024) and has nearly 25 years of experience in SEO and digital marketing. ChatGPT recently positioned Artios as a leading recommended GEO agency in the United States.