Beam + Spanner in Production: Multi-Tenant Identity Resolution and Resilient Ad Platform Activation

BEAM SUMMIT NYC 2026

Jonathan Sudhakar

getangler.ai

SECTION 1 · ANGLER AI INTRODUCTION

About Angler AI

Signal enrichment and prediction for ~75 marketing brands

~2B events/mo in

Web SDK · API · CSV

→

Angler Beam Jobs

Identify + Enrich + Activate

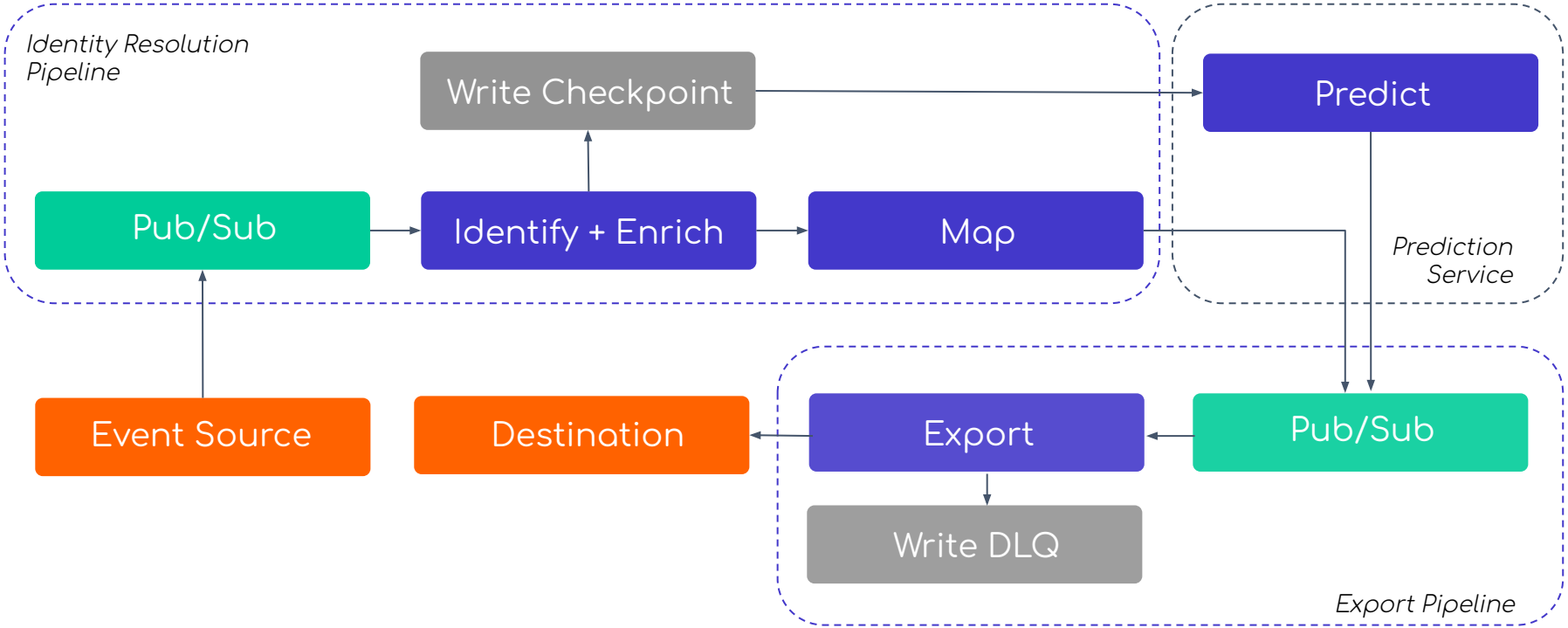
→

~5B events/mo out

2500 events/s peak throughput

- Identifies every visitor with a 360° profile with purchase, demographic, and interest attributes
- Predicts conversion likelihood, LTV, and best offer for each prospect using a deep neural network
- Enriches ad platforms (Meta, Google, TikTok) with these predictions via each platform's conversions API
- Delivers +37% ROAS (return on ad spend) and −26% CAC (cost per acquisition) on average

The pipelines



SECTION 2 · THE LEGACY SYSTEM

One person, a dozen identifiers

Anonymous web visit
`client_id` · `fbclid` · `IP` · `user-agent`

Newsletter signup
`email` - sometimes hashed, sometimes not

Checkout
`order_id` · `phone` · `shipping zip`

Server-side conversion
`order_id` · `hashed_email` - hours later

Login
`client_id` · `email` · `customer_id`

CRM import
`customer_id` · `email`

Identity resolution = connected components over a bipartite graph (entities ↔ identifiers) - computed online, per event, on the hot path

Enrich = a chain of point lookups inside a DoFn

- Serverless · zero ops · single-digit-ms reads
- Document model matched the lookups of the day
- Same rules across all tenants
- But: 4 sequential RPCs per event, and single-hop matching only

enrich_v1.py

```
class Enricher(beam.DoFn):
    def process(self, event):
        e = self.enrich_by_session(event)   # collection-group query
        e = self.enrich_by_identity(e)      # Firestore doc lookups
        e = self.enrich_by_customer(e)      # Firestore doc lookup
        e = self.enrich_by_order(e)         # Firestore doc lookup
        yield e                             # worker-local LRU caches
```

SECTION 2 · THE LEGACY SYSTEM

What scale did to it: two failure modes, one cost curve

1 · Latency-shaped cost - Beam

- 4+ sequential RPCs/event → DoFn threads **park on I/O**
- Parked threads ⇒ low throughput/worker ⇒ **the autoscaler buys more workers to wait**

2 · No graph semantics - Data

- No server-side joins → transitive identity ⇒ materialize **cluster docs**
- Every merge rewrites the doc: **O(cluster size) write amplification** · 1 MiB/doc cap · sustained-write hotspots

At ~3× the cost of what replaced it and still couldn't answer "who is transitively connected to this email?"

Why can't we just use Beam state?

- Beam state is **per key**. A cluster *spans* keys (email ↔ phone ↔ cookie)
- Merges happen **across keys** — no cross-key reads, no transactions in state
- The graph must live in a store that answers **transitive connectivity online**
- It has to outlive the job - pipelines drain, update, and redeploy constantly; identity can't reset or rebuild on every deploy, and streaming, backfill, and maintenance all read the same graph. Beam state belongs to exactly one job; the identity graph is a durable system-of-record.

BEAM OWNS

compute · parallelism · retries · batching

THE STORE OWNS

connectivity · consistency

SECTION 3 · SPANNER

Four properties. Most systems get two.

1 · Online
single-digit-ms point reads, on the event hot path

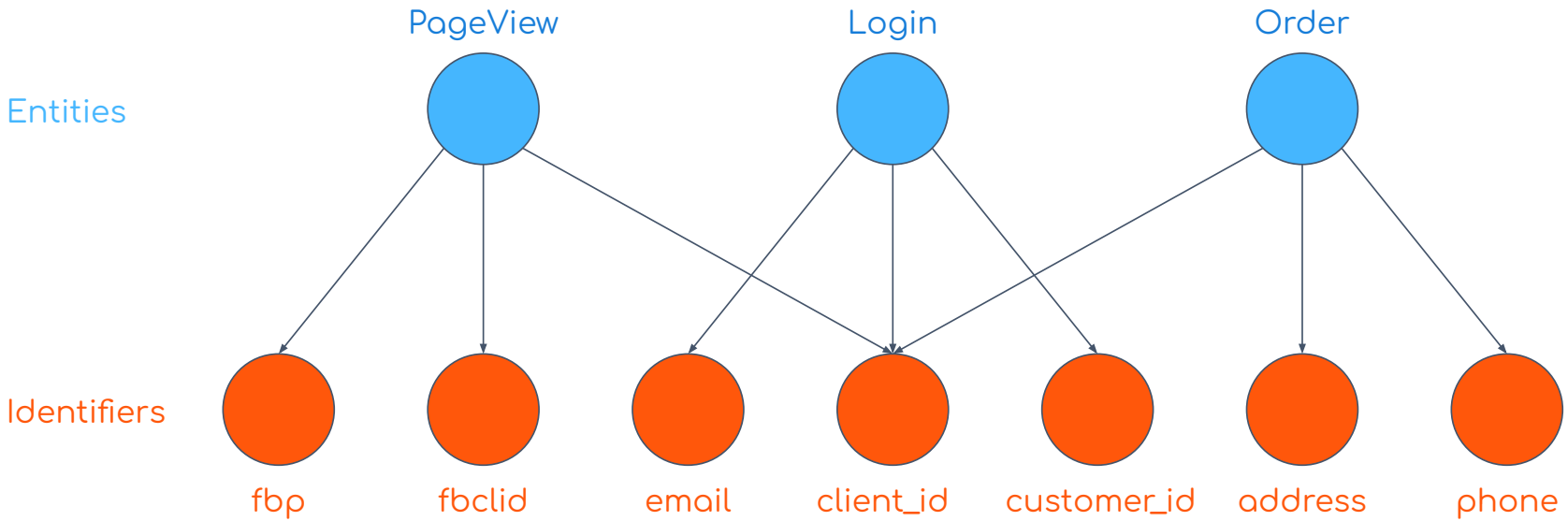
2 · Strongly consistent
merges are concurrent cross-row updates; last-write-wins corrupts graphs

3 · Horizontal write scale
>10000 reads/writes per second

4 · Multi-tenant + lifecycle
>75 tenants in one schema · TTL & PII deletion as a feature

	online	consistent	write scale	tenancy / TTL
Firestore	yes	partial	hotspots	partial
Bigtable	yes	single-row	yes	partial
DynamoDB	yes	limited txns	yes	partial
MySQL / Postgres	yes	yes	vertical	manual
BigQuery	no	yes	yes	yes
Neo4j / Neptune	yes	yes	partial	one-big-graph
Spanner	yes	yes	yes	interleaving + row TTL

The graph model: entities ↔ identifiers



SECTION 3 · SPANNER

The schema: an edge log plus a disposable cluster index

SOURCE OF TRUTH — EDGE LOG

`entities` — event / order / customer · TTL via row deletion policy

`identifiers` — distinct id_type + id_value

`entity_identifiers` — the edges, ~10/event

→ rebuilt incrementally · min-wins merges

MATERIALIZED READ PATH — DISPOSABLE

`identifier_cache_map` — identifier → cluster_id

`cluster_cache_profiles` — cluster → ranked JSON

- **No users table.** The person *is* the connected component
- Clusters = **incrementally materialized union-find**, never authoritative
- Edges expire (TTL) · PII gets deleted · rules change ⇒ clusters must be **recomputable**

SECTION 3 · SPANNER

Bundles, retries, idempotency: the contract every write obeys

Beam streaming = at-least-once bundles. Design for the retry, not the happy path.

A failure anywhere replays the whole bundle

- Every Spanner write is INSERT OR UPDATE, keyed on **deterministic IDs**
- Effectively-once is achieved by **idempotency**, not exactly-once execution

Budget mutations, not rows

- Spanner caps **~80k mutations/commit** = rows × columns × index entries
- Buffer per bundle, flush by **estimated mutation count**, not row count

Min-wins merge: concurrent transactions both pick the lexicographically-min cluster ID - order-free convergence under concurrent merges.

Deterministic failures: when retry is the wrong tool

THE PRODUCTION POSTURE, IN PRIORITY ORDER

1. **Validate at extraction** - 512-char identifier cap; a 4 KB URL is not an identity signal (a hashed twin survives for matching)
2. **Classify errors in the DoFn** - non-retryable ⇒ strip / DLQ the element, never re-raise
3. **Alert on watermark age**, not just error rate - a retry loop is polite and quiet

- Malformed input (oversized identifier, corrupt value) fails the write **every time**
- Fail → retry → identical fail - and **every element sharing the bundle is trapped behind it**
- The watermark stalls while **error metrics stay quiet**

The read path: three phases, one invariant

1 · PROBE

identifier → cluster → cached ranked profile. Fresh hit = done.

Steady-state: the vast majority of events end here

2 · TRAVERSE

On miss: 2-hop join on a **stale read-only snapshot (10 s)** - zero locks at any staleness.

3 · BOOKKEEP

Pick / merge cluster, write cache. **Failure here is non-fatal** - the event is already enriched.

Reads never lock. Writes never read.

Weighted conflict resolution: ~25 scored rules/tenant rank what the cluster "knows"; the event is back-filled from the winner per identifier type.

The lock-free invariant: designing transactions for 300 parallel workers

THE TEXTBOOK DESIGN LOSES

- Traversing **inside** the r/w txn takes shared locks on the same index ranges concurrent writers insert into - hot identifiers are hot on both paths
- At 300 workers: writers **convoy** behind readers → slow bundles → backlog → **autoscaler adds workers → MORE lock pressure**

The recovery mechanism amplifies the problem.

PRODUCTION: CONTENTION STRUCTURALLY IMPOSSIBLE

- Traversals on **lock-free read-only snapshots** - zero locks at any staleness
- Writes = millisecond, cache-only, **one mutation set** - not per-row DML
- Merges = **min-wins** - commutative, order-free, retry-safe

A path that cannot take locks needs no lock tuning. Ever.

Bounding external I/O: client timeouts bound attempts, not calls

- A client timeout= bounds **one RPC attempt** - streaming clients resume transparently, and **each resume gets a fresh deadline** . Total wait: unbounded
- A wedged stream doesn't error. It *waits* . Bundle pinned, watermark stalled, **every error metric reading zero**
- The only true bound is **wall-clock, enforced from outside the client**

wall_clock_bound.py

```
ex = ThreadPoolExecutor(1)    # NEVER `with` - exit joins,
try:                          # nullifying your timeout
    rows = ex.submit(run_query).result(timeout=30)
finally:
    ex.shutdown(wait=False, cancel_futures=True)
    # leak the thread; free the bundle
```

Late data: windows are for aggregates, graphs are for structure

Server-side conversions arrive hours after their browser events

THE AGGREGATION ANSWER

Windows + allowed lateness + retractions

Wrong tool for identity - a late conversion isn't a late number to revise

THE GRAPH ANSWER

Append the edge. Connectivity updates by construction.

- A late order (email, phone) links the email-cluster the moment it lands
- The next event for *either* identifier sees the merged person
- No allowed-lateness horizon: merges are min-wins, so **arrival order doesn't matter**

Beam moves data with correct semantics; the graph makes lateness a non-event.

Composite match keys: weak signals, combined and hashed

A per-tenant scored waterfall of ~25 rules. Lower score = stronger = wins ties.

deterministic PII	1.0	name + zip	FIRSTNAME · LASTNAME · ZIP
	2.0	email + zip	EMAIL · ZIP
first-party IDs	4.0	email	EMAIL
	5.0	phone	PHONE
	6.2	client id	CLIENTID
probabilistic device / geo	7.0	device fp	UA · ZIP · CONN · IP
	25.0	ip only	IP

- Composite = `sha256(sorted("type=value|…"))` - emitted only when **every component is present**
- Each plaintext rule has a **hashed twin** (`HEMAIL…`, score x.1) — server-side PII matches without re-hashing
- **Composite-only types** (IP, UA, geo, city, state) are never standalone identifiers
- `max_match_priority` = per-tenant cutoff: drop everything weaker than score N

`EMAIL=example@getangler.ai | ZIP=10001` → `sha256` → `a3f9…`

one identifier node — two people with the same email+zip collide here by construction

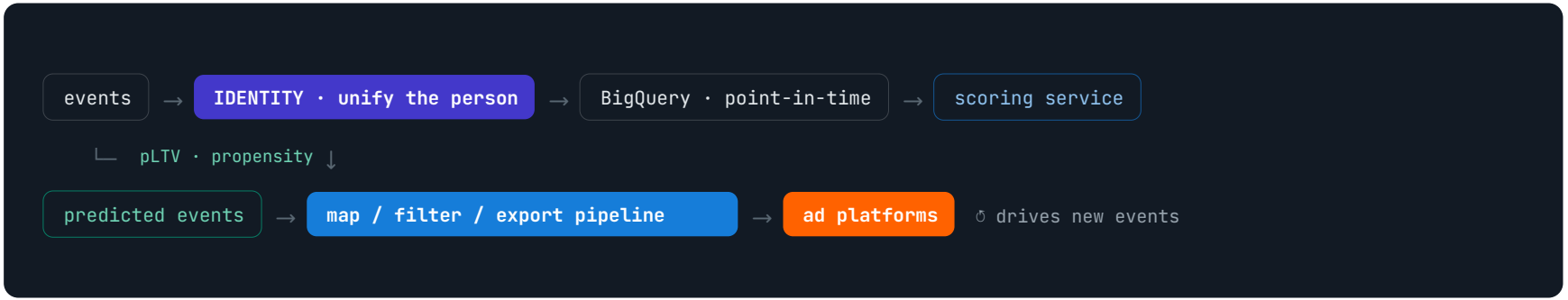
Tenant isolation in Spanner

```
CREATE TABLE identifiers (
ws_id STRING(36) NOT NULL, -- tenant leads every PK
id_type STRING(128) NOT NULL,
id_value STRING(512) NOT NULL,
) PRIMARY KEY(ws_id, id_type, id_value),
INTERLEAVE IN PARENT workspaces
ON DELETE CASCADE;
```

- `ws_id` leads every PK → a tenant's rows are **one contiguous key range**
- Interleaving co-locates the graph under `workspaces`; **CASCADE** makes offboarding one delete
- Per-tenant knobs are **columns**: `max_match_priority`, `super_node_threshold`
- Sharing is an explicit allow-list (`allowed_ws_ids`) - cross-tenant leakage is **impossible by construction**

SECTION 3 · SPANNER

The payoff: identity-keyed ML / pLTV



- Score the **person**, not the fragment - a model on un-resolved events trains on ghosts
- **pLTV → conversion value** for value-based bidding
- Predictions ride back through the same pipeline - `predictions.<key>` is addressable in filters & value mappings
- **propensity → audience gate** — closes the loop, no new code

SECTION 4 · EXPORTS

The egress problem: 8 platforms, 8 failure dialects

~5B	8	>1000	2500 events/s
events/month out	ad platforms	destination configs	Peak Throughput

Per-platform physics: batch caps 500–2,000 · payload caps 1–6 MB · distinct rate limits · distinct retry semantics · **distinct failure dialects:**

- Extended 5xx outages, unacknowledged on any status page
- HTTP **200 OK** wrapping {"success": false}
- Batch-size limits that change without notice

A connector is a translator: every dialect compiles to three words: retryable · not-retryable · slow-down

SECTION 4 · EXPORTS

Fan out export to destination

1 · MapAndFilter

DNF filter: does this event qualify? · event-name map: `"purchase"` → per-platform vocab

→

2 · AttributeMapper

config-driven derivation · `STRIP_FIRST($client_id,6)` · `predictions.pltv`

→

3 · Connector.transform

platform schema · PII hashing · `action_source` · then GroupIntoBatches

enriched event

→

Meta

`Purchase UserData · CustomData + action_source`

Google

`conversion gclid · value currency_code`

TikTok

`CompletePayment context · properties ttclid`

GA4

`purchase client_id · items[] params`

- **Canonical internally, reshaped per destination** - a new destination is config, a new platform is a connector subclass
- No matching config = **expected silent drop**, not an error

SECTION 4 · EXPORTS

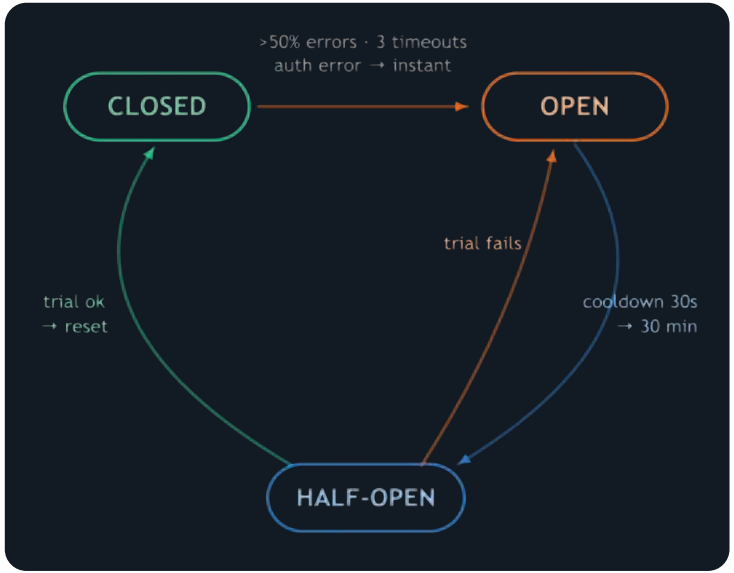
Adaptive batching: GroupIntoBatches by (tenant, destination)

- Batch size = **platform physics** (Meta 1000 · TikTok 500 · Google Ads 2000) - legal by construction; byte-cap re-chunk for 1–6 MB payload limits
- **Per-key batching = isolation**: a slow platform queues against *its own* key
- State-backed: half-built batches **survive bundle retries and worker churn**
- Adaptive **to the destination** - deliberately *not* dynamic backpressure sizing

export_batching.py

```
events | beam.WindowInto(...)    # plumbing window for state/timers
    | beam.GroupIntoBatches(
        batch_size=DEST_BATCH_CAP[platform],
        max_buffering_duration=...)    # latency escape hatch
# keyed by (workspace_id, destination_id)
```

Circuit breakers: a platform outage must not become a pipeline outage



- OPEN = **fail-fast to DLQ**: zero upstream calls, **no parked worker threads**
- Per (tenant, destination), worker-local - workers converge on "platform X is down" in **~100 calls**
- Extended outage → that platform DLQs cleanly, **the other 7 run unaffected**, full replay on recovery

Without breakers: timeouts park DoFn threads → worker pool starves → the watermark stalls for every destination.

SECTION 4 · EXPORTS

DLQ + replay: the loop closes through the pipeline

A DLQ you can replay is a feature. A DLQ you can't is a graveyard.

Structured dead letters

Payload + destination + error class + attempt count + breaker context - **queryable in BigQuery**, not a log line

Automated replay

Batch Beam job feeds dead letters back through the **same connector transforms** - same batching, rate limits, breakers

Poison batches

Binary-search isolation - split the failing batch, retry halves, converge on the toxic element(s), quarantine, deliver the rest

Recovery

Platform recovers → replay drains the backlog → **audit trail end to end**