**Translation section**

API key not valid

**Page 1**

Re-evaluation by factoring in Azure and  
Martin Fowler's guidelines  
Here’s a point-by-point re-evaluation of your draft (“MicrOSINT”) against  
Microsoft’s Azure architecture styles and Fowler’s guidance—plus the specific  
changes I think you should make.  
1) How your layers map to Azure styles  
Your Layer 1 — Infrastructure & Persistent Storage → N-tier “shell” + Big  
Data foundation.  
This layer is the right place to host the “governance shell” (front end/API/data) that  
Azure labels N-tier. But because you plan to store raw HTML/PDF/media and  
progressively refined structures, you also need a lakehouse (bronze/silver/gold)  
as the durable backbone for historical analytics, joins, lineage, and reproducibility  
—i.e., Azure’s Big Data architecture style. Microsoft Learn+2Microsoft Learn+2  
Your Layer 2 — NLP & Translation → Big Data + Big Compute adjunct.  
The microservices are correct; make them consumers of stream/queue events,  
persisting normalized outputs to the lake’s silver layer. For heavy jobs (OCR on  
large PDFs, batch translation, embedding generation), treat this as Big Compute  
(burstable, batch, fault-tolerant) rather than always-on services. Microsoft Learn  
Your Layer 3 — Data Ingestion & Source Integration → Event-driven ingress.  
This layer is textbook Event-Driven architecture: producers (scrapers, APIs, file  
drops) → event channels → consumers (normalizers, enrichers). Use an event  
mesh (Event Hubs/Event Grid/Kafka) plus Service Bus topics for business  
events and routing. Microsoft Learn  
Your Layer 4 — Agent Engine → Event-driven + Web-Queue-Worker.  
Agents should not be directly invoked by the UI or by crawlers. Instead, they  
should: (a) subscribe to events; (b) pick up queued work items for long-running  
flows. That is exactly Azure’s Web-Queue-Worker style: a thin web/API tier, a  
queue, and workers that process with retries/idempotency. Add queue-based  
load leveling to ride out news spikes. Microsoft Learn+1  
Your Layers 5–6 — Command Console & UI / User Interface → N-tier façade.  
Keep them as the access/governance shell (auth, RBAC, throttling, API

**Page 2**

Management). They should never orchestrate long jobs—just enqueue or publish  
events and return immediately. Microsoft Learn  
2) Fowler lens: where to tighten the design  
3) Concrete changes I recommend  
A. Insert an explicit event mesh and split streams by purpose.  
B. Wrap all long-running work in Web-Queue-Worker.  
The UI/API acknowledges quickly, enqueues a job, and returns a job ID. Workers  
(Functions, Container Apps, or AKS) pick up jobs with queue-based load  
leveling, retries, idempotency keys, and poison queues. This is the most impactful  
reliability upgrade you can make. Microsoft Learn+1  
C. Formalize the lakehouse and enforce schema contracts.  
Adopt bronze (raw), silver (normalized/enriched), gold (curated  
entities/events/relationships). Enforce JSON schema on every event and queue  
payload; reject/repair non-conforming outputs from agents. This is Azure’s Big  
Data best practice and will drastically reduce downstream fragility. Microsoft  
Learn+1  
Bounded contexts. Your draft already implies them. Make them explicit:  
Collection (ingest), Enrichment (NLP/translation), Analysis (agents, scoring,  
entity/event modeling), Delivery (search, reports, alerts). Each context owns its  
model and contracts. martinfowler.com+1  
Selective CQRS. Use CQRS only at the Delivery edge: write-heavy pipelines  
flow to the lake/OLTP; read models are projections (search indices,  
denormalized views) optimized for UI/API. Don’t sprinkle CQRS through  
ingestion. (Fowler cautions about complexity creep.) martinfowler.com  
Evolution (Strangler Fig). For any tightly-coupled n8n or direct-call paths,  
place a façade in front, route new functionality via events/queues, and retire  
old pieces incrementally. martinfowler.com  
Ingress streams (high-throughput): Event Hubs/Kafka for raw items.  
Business events (routing, retries, dead-letter): Service Bus topics/queues  
between bounded contexts (e.g., document.normalized, entity.extracted,  
report.published).  
This cleanly implements the Event-Driven style, keeps components  
decoupled, and supports fan-out/fan-in processing. Microsoft Learn

**Page 3**

D. Treat heavy ML as Big Compute jobs.  
Run embedding refresh, similarity joins, clustering, and geospatial aggregation as  
bursty, fault-tolerant jobs (Databricks/AKS/Azure ML with autoscale). Don’t leave  
GPUs hot unless needed. Microsoft Learn  
E. Keep the N-tier façade thin.  
Use API Management + Web/App Service for the Delivery context (search,  
dashboards, alerts). All orchestration goes to events/queues, not controllers. This  
aligns with Azure’s guidance for layered apps. Microsoft Learn  
F. Add observability and contracts at the seams.  
At every boundary (topic/queue), log: schema version, producer, correlation ID,  
attempt count, and provenance. This gives you traceability and reliable replay  
(useful for audits and “why did we assert X?”).  
4) Mapping your components to Azure  
choices (decision points)  
5) What to keep from your draft (good calls)  
Event channel: Event Hubs (firehose) vs Kafka (self-managed) for raw  
ingress; Service Bus for business events (sessions, dead-lettering, rules).  
Microsoft Learn  
Workers: Azure Functions for lightweight stateless steps; Container Apps or  
AKS for stateful, GPU, or long-running agent pods. Web-Queue-Worker  
pattern is the umbrella. Microsoft Learn  
Lakehouse: ADLS Gen2 + Parquet/Delta; transformations in  
Databricks/Synapse/Fabric. Bronze/Silver/Gold with enforced schemas.  
Microsoft Learn+1  
Read models: Azure AI Search indexes from gold; optional Postgres read  
replicas for dashboards.  
Governance: Purview (catalog/lineage), Key Vault (secrets), API Management  
(throttling/RBAC), standard Well-Architected guardrails. Microsoft Learn  
Modularity as a first principle (agents as composable micro-apps).  
Message-based agent comms (you already propose Kafka/RabbitMQ).  
Formalize this as your event mesh.  
UI agents and a console as thin clients over the API layer.

**Page 4**

6) What to tighten (summary)  
0) Executive summary  
Build the platform as four bounded contexts—Collection, Enrichment, Analysis,  
Delivery—stitched together by an event mesh and queues. Land everything in a  
governed lakehouse (bronze/silver/gold). Run slow/variable work behind web-  
queue-worker jobs. Use CQRS only at the Delivery edge (APIs/UI). Treat heavy  
ML/LLM workloads as big compute bursts. Enforce function calling +  
structured output contracts at every boundary. Migrate incrementally via a  
Strangler façade around any legacy direct-call flows.  
1) Domain model (bounded contexts)  
A) Collection (Ingestion)  
Responsibility: Acquire external data reliably.  
Inputs: Webhooks, scrapers, APIs, file drops.  
Outputs: Raw items as immutable events into an ingress stream; originals  
stored in lake bronze.  
Notes: Rate limits, politeness (robots.txt), source metadata capture, basic  
validation. No business logic.  
B) Enrichment (NLP/translation/normalization)  
Security is fractal (zero-trust per module) — keep it and wire it to RBAC at  
the API and queue/topic levels.  
Decouple everything with events + queues (no direct calls across layers).  
Enforce schema contracts at every boundary (reject/repair non-conformant  
payloads).  
Make heavy NLP/LLM/OCR queue-driven and Big Compute when  
appropriate.  
Adopt a lakehouse bronze/silver/gold lifecycle.  
Use CQRS only where read/write concerns really diverge (Delivery).  
Evolve via Strangler façade around any legacy direct-call or n8n paths.

**Page 5**

Responsibility: Turn raw content into normalized, language-agnostic records.  
Inputs: “item.received” events from Collection.  
Outputs: Clean text, metadata, language, OCR text, translation, detected entities  
→ lake silver; “item.normalized”, “entity.extracted” events.  
Notes: Stateless microservices where possible; long jobs (OCR, large PDFs,  
video ASR, translation batches) run via queues.  
C) Analysis (Agents & knowledge)  
Responsibility: Reasoning, correlation, deduplication, entity linking, reliability  
scoring, clustering, geospatial joins, alert thresholds.  
Inputs: “entity.extracted”, “fact.detected”, “similarity.ready”.  
Outputs: Curated events/entities/relations → lake gold; “alert.triggered”,  
“report.drafted”.  
Notes: Agents are tool-using via function calling; contracts enforce structured  
JSON I/O.  
D) Delivery (APIs, Search, UI)  
Responsibility: Present read-optimized views; manage tasks/alerts; produce  
reports.  
Inputs: Gold projections.  
Outputs: API responses, UI dashboards, exports.  
Notes: CQRS: writes flow through upstream services; reads served from search  
indices and denormalized projections.  
2) Interaction style (macro patterns)  
Event-driven backbone: A high-throughput ingress stream for raw items;  
business event topics between contexts (normalized, extracted, curated,  
alerted).  
Web-queue-worker for long jobs: UI/API acknowledges quickly, enqueues  
work; idempotent workers process with retries, back-off, and DLQs.  
N-tier façade: A thin API/UI shell for auth, RBAC, throttling; never orchestrates  
long work.  
Big data/lakehouse: Bronze (raw), Silver (normalized), Gold (curated);  
Parquet/Delta; schema enforcement and lineage.

**Page 6**

3) Data lifecycle (lakehouse)  
Bronze (immutable): Original HTML/PDF/media + fetch metadata.  
Silver (normalized): Clean text, language, OCR/translation outputs, extracted  
entities with confidence, timestamps normalized to ISO-8601, geocodes.  
Gold (curated): Deduplicated Events(who/what/when/where),  
Entities(person/org/place/equipment), Relations(actor-event, entity-entity),  
Source reliability scores, Provenance links.  
Retention: Bronze long-term (cold), Silver mid-term, Gold hot. Version everything;  
no in-place mutation.  
4) Contracts & schemas (function calling +  
structured output)  
All cross-boundary messages use JSON with versioned schema. Example  
essentials:  
4.1 item.received (Collection → Enrichment)  
json  
CopyEdit  
{ "schema":"item.received@v1", "item\_id":"uuid", "source":  
{"type":"web","url":"...","collection\_rule":"rule-17"}, "content\_ref":  
{"lake\_uri":"bronze://..."}, "observed\_at":"2025-08-09T07:10:00Z",  
"language\_guess": "fa", "trace\_id":"uuid" }  
4.2 item.normalized (Enrichment → Analysis)  
json  
CopyEdit  
{ "schema":"item.normalized@v1", "item\_id":"uuid", "text":"...",  
"language":"fa", "translations":[{"lang":"en","text\_ref":"silver://..."}],  
Big compute (elastic): Embeddings, clustering, geospatial, model fitting run  
as bursty jobs with autoscale.

**Page 7**

"media":{"ocr":true,"ocr\_conf":0.86}, "entities":  
[{"type":"Person","value":"...","confidence":0.87}], "geo":[{"lat":...,  
"lon":..., "method":"gazetteer"}], "trace\_id":"uuid" }  
4.3 event.curated (Analysis → Delivery)  
json  
CopyEdit  
{ "schema":"event.curated@v1", "event\_id":"uuid",  
"type":"Airstrike|Protest|Sanction|Deployment", "time":"2025-08-  
08T21:35:00Z", "place":{"name":"...","lat":...,"lon":...}, "actors":  
["org:...","person:..."], "supporting\_docs":  
[{"item\_id":"uuid","evidence\_span":[[120,180]]}],  
"dedup\_cluster\_id":"uuid", "reliability":0.78, "provenance":  
{"algorithm":"v3.2","inputs":["item\_id1","item\_id2"]}, "trace\_id":"uuid" }  
Validation: Any invalid payload is rejected with a contract error; producers must  
retry/repair. Schema versions are evolutionary (additive fields; deprecate with  
grace periods).  
5) Compute & storage components (tech-  
agnostic with Azure mapping)  
Capability  
Tech-agnostic choice  
Azure mapping (later)  
Event ingress   
(firehose)  
Kafka/Redpanda  
Event Hubs  
Business events  
Pub/Sub with DLQ  
Service Bus Topics/Queues  
Long jobs  
Worker fleet + queues  
Container   
Apps/AKS/Functions + SB  
Lake storage  
S3-compatible +   
Parquet/Delta  
ADLS Gen2 + Delta  
Batch   
transforms  
Spark/SQL engines  
Databricks/Synapse/Fabric  
Search  
Lucene-based/Vec search  
Azure AI Search  
OLTP metadata  
Postgres  
Azure Database for   
PostgreSQL

**Page 8**

Capability  
Tech-agnostic choice  
Azure mapping (later)  
Vector memory  
Qdrant/FAISS/Pinecone  
Azure Cosmos DB (vector) or   
3rd-party  
Secrets  
Vault  
Key Vault  
Identity  
OIDC  
Entra ID  
Observability  
OpenTelemetry +   
ELK/Prom/Tempo  
Azure Monitor/Log   
Analytics/Grafana  
Keep your service interfaces cloud-neutral; only adapters speak provider SDKs.  
6) Agent engine (design rules)  
7) Delivery (CQRS + APIs/UI)  
Plan-Act-Observe loop with function calling to tools: crawl, translate, OCR,  
extract, score, geocode, dedup, index, alert.  
Stateless-by-default; explicit memory via vector store keyed by trace\_id  
and case\_id .  
Cost & latency envelopes per agent (max tokens, max runtime, priority  
class).  
Safety rails: allow-listed tools; redaction of PII; rate-limit per source;  
provenance logging of every call with inputs/outputs hashes.  
Failure policy: idempotency keys, retries with jitter, circuit breakers; partial  
results are acceptable with caveats.  
Queries: Read from search indices and gold projections (materialized  
views for dashboards, timelines, maps).  
Commands: Create tasks, acknowledge alerts, attach analyst notes;  
commands enqueue events to upstream services—no direct writes to gold.  
Exports: PDF/HTML briefs created by workers (queue), then streamed to  
users.  
Access control: Role- and case-scoped permissions; audit every read/write;  
redact by role.

**Page 9**

8) Reliability, scaling, and cost  
9) Security & governance  
10) Observability & SLOs  
Load leveling: All heavy work passes through queues with concurrency caps.  
Autoscaling: Scale workers on queue depth and age; scale stream  
consumers on lag.  
Back-pressure: If gold build lags, shed non-critical jobs (e.g., low-priority  
translations).  
Cold vs hot paths: Alerts, priority sources go hot (fast lanes). Bulk crawling,  
historical backfills go cold (batch).  
Caching: Edge caches for common queries; content-addressable storage  
(hash-based) to avoid re-processing dupes.  
Zero-trust per service: Each component gets the minimum permissions.  
Secretless runtime: Prefer managed identities; else short-lived tokens from  
Vault.  
Provenance: Every curated event links to evidence items + algorithm  
versions; keep reproducibility logs.  
Compliance: Respect robots.txt; configurable source allowlists/denylists;  
encryption in transit/at rest; signed artifacts for builds.  
Tracing: Propagate trace\_id end-to-end; correlate logs, metrics, spans.  
Golden signals: event lag, queue age, job success rate, schema-compliance  
rate, cost per item, time-to-alert, search p95 latency.  
SLO examples:  
Time-to-ingest (ingress → bronze): p95 ≤ 30s  
Time-to-normalize (bronze → silver): p95 ≤ 2m  
Time-to-alert (silver → alert): p95 ≤ 3m  
Search API p95 ≤ 500ms; availability ≥ 99.9%

**Page 10**

11) Migration plan (Strangler)  
12) Two canonical flows (so you can test  
early)  
12.1 Real-time alert flow  
12.2 Weekly report flow  
13) What to build first (vertical slice)  
1. Front a façade in front of any n8n/direct pipelines.  
2. New sources publish to the ingress stream; new processors read from  
topics/queues.  
3. Gradually rewrite legacy processors as stateless workers; retire old endpoints.  
4. Turn on schema enforcement at edges; repair producers until pass rates  
stabilize.  
5. Move storage into bronze/silver/gold; backfill using batch jobs.  
1. Collection publishes item.received .  
2. Enrichment normalizes → item.normalized .  
3. Analysis agent extracts facts, links entities, scores reliability; if threshold met  
→ alert.triggered .  
4. Delivery sends notification; UI shows alert with evidence links.  
1. Scheduler enqueues “compose.weekly.report(case=XYZ)”.  
2. Worker queries gold + search for week window, clusters events, drafts  
narrative with citations.  
3. Human review; upon approve, worker renders PDF/HTML and emits  
report.published .  
Ingress: one source (RSS or webhook) → item.received .  
Enrichment: language detect + translation → item.normalized .  
Analysis: simple rule (keyword + geo) → alert.triggered .  
Delivery: minimal search index + alert view.

**Page 11**

This gives you an end-to-end line you can demo, measure, and harden—then  
scale horizontally with more sources, more enrichers, and smarter agents.  
Practical Guidelines  
Lakehouse: bronze/silver/gold folders + basic schemas.  
Observability: trace IDs, queue depth dashboards.