zensar

Domain-Driven Testing Powered by Dynamic RAG

Adaptive retrieval that responds to intent, risk, incidents, and audits

White paper

An RPG Company

Table of Contents

Executive Summary

Domain-Driven Testing (DDT) uses a living, bounded-context corpus — specs, regulations, runbooks, incidents, prior audits — to generate the tests that matter most. Static Retrieval-Augmented Generation (RAG) pipelines retrieve the same way for every prompt, which underserves regulated, safety-critical, and change-heavy workloads. This paper introduces Dynamic RAG, a retrieval layer whose chunking, index choice, depth, freshness, and reranking shift with the intent of the test request, the risk tier of the capability, the incidents and fault patterns on file, and the audits the evidence must satisfy. We describe the architecture, name the five design moves that make a RAG dynamic, and integrate context engineering, functional decomposition, fault-driven generation, and risk-based prioritization as first-class parts of the loop.

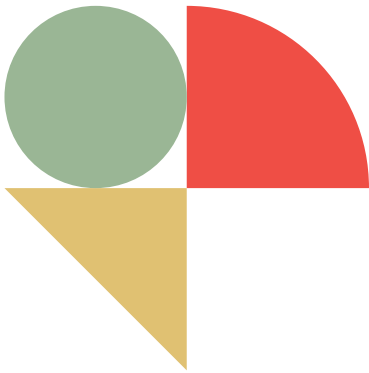
From Static to Dynamic RAG

A static RAG pipeline fixes the retriever upfront: one embedding model, one chunking strategy, one top-k, one rerank policy. It works for question-answers. It fails for testing because a test suite is a portfolio — a few low-risk smoke checks, a long tail of business rules, a small but critical core of regulated flows, and a constantly changing surface of recent incidents. The retriever that fits a login-form test is not the retriever that fits an IFRS 9 expected-credit-loss calculation.

Dynamic RAG keeps the bounded-context corpus and the ubiquitous language of Domain-Driven Design (DDD), but lets the retrieval plan change per request. Five knobs move together: (a) which index is queried (regulatory, code, incidents, audit); (b) the chunking granularity used (clause, function, step, row); (c) the embedding model applied (general vs. domain-tuned); (d) the depth and freshness window (last quarter vs. all time, top-5 vs. top-30 with rerank); and (e) the context assembly policy (ordering, deduplication, conflict resolution).

Why the name matters

Calling the contribution "Vector RAG" undersells it — vector search is table stakes. The differentiator is that the pipeline adapts: intent-aware routing, risk-aware depth, incident-aware freshness, and audit-aware provenance. "Dynamic RAG" names that adaptivity and makes the design reviewable.



Architecture at a Glance

Five layers sit between a capability under test and the generated suite.

1. An ingestion layer normalizes sources into typed chunks with provenance.
2. Multiple indexes hold those chunks — a dense vector index, a sparse BM25 index, and a metadata/graph index — linking regulations to controls and incidents.
3. A router turns the test request into a retrieval plan.
4. A context engineer assembles the retrieved material into a prompt that respects token budgets and priority.
5. A generator produces tests, and an evaluator scores them against golden sets before they land in CI.

Embeddings: Pick the right model for the domain

General-purpose sentence embeddings miss regulatory and financial semantics. For financial services, we recommend a domain-tuned baseline such as FinBERT (Araci 2019) or the continued-pretraining FinBERT of Yang, Uy, and Huang (2020), which are widely cited as strong starting points for financial text [1][2]. For healthcare, ClinicalBERT or BioBERT; for code-heavy retrieval, a code-native embedding. Domain tuning typically improves recall on policy clauses and rule language by a meaningful margin over out-of-the-box models, at no extra inference cost after the one-time fine-tune.

FinBERT as an industry baseline

Start with a domain-tuned BERT for FS corpora (credit policy, AML typologies, IFRS/IAS commentary, product disclosures). Benchmark against a general model on your own golden set — the uplift is real but corpus-dependent. Re-evaluate every 6 – 9 months as open-weight finance models evolve.

Context engineering

Retrieval without context engineering leaks recall into irrelevance. Treat the prompt as a planned artifact with three deliberate moves.

Plan — decide what the test generator needs (rule text, examples, counter-examples, incident patterns, acceptance thresholds) before retrieval fires.

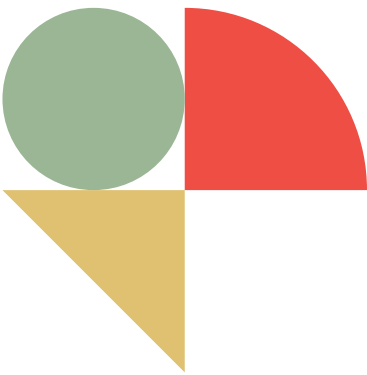
Order — place the highest-authority context first (regulation clause), then the application rule, then historical evidence; models attend more to the head and tail of the window, so contradictory material in the middle gets drowned.

Resolve conflicts — when retrieved clauses disagree (a policy update superseding older guidance), prefer the highest-authority, most-recent, in-scope chunk, and record the dropped chunks with reasons in the prompt metadata so the evaluator can audit the decision.

Dynamic retrieval policies

A policy is a function that maps a request to a retrieval plan. Pseudocode (simplified):

```
def plan(request):
    intent  = classify_intent(request)    # {rule, edge, regression, exploratory}
    risk    = lookup_risk(request.capability)  # R1..R4 tier
    fresh   = incident_window(request.capability)
    plan = RetrievalPlan()
    if intent == "rule":
        plan.indexes = ["regulatory", "policy"]
        plan.chunking = "clause"; plan.k = 10; plan.rerank = "cross_encoder"
    elif intent == "edge":
        plan.indexes = ["regulatory", "incidents", "fault_kb"]
        plan.chunking = "step"; plan.k = 30; plan.rerank = "cross_encoder"
    if risk >= R3:
        plan.embedding = "finbert"  # or clinicalbert, etc.
        plan.provenance = "required"
        plan.k = max(plan.k, 20)
    if fresh < 90:  # days since last incident
        plan.freshness = "last_6_months"
        plan.boost = "incident_linked"
    return plan
```



Functional Decomposition for Test Depth

Business rules, taken whole, generate shallow tests. Decompose them first: Rule > Decision > Condition > Edge. A credit-approval rule decomposes into a small set of decisions (approve, refer, decline); each decision has conditions (DTI band, score band, tenure, product); each condition has edges (boundary values, nulls, currency conversions, effective-date transitions, rounding). The retrieval prompt asks Dynamic RAG for evidence at each level — the rule clause for the decision, the parameter tables for the conditions, and the incident log and fault KB for the edges. The generator then emits a matrix of tests that cover every path to each outcome, not just the happy-path sentence.

Level	What it captures	Retrieval signal
Rule	The statement in policy or regulation	Clause index, authoritative chunk
Decision	The outcomes the rule admits	Decision tables, procedure steps
Condition	The predicates that drive each decision	Parameter tables, schema, code
Edge	Boundaries, nulls, transitions, rounding	Incident log, fault-pattern KB

Failure- and Fault-Driven Test Generation

Most organizations treat their defect history as reporting data. Dynamic RAG treats it as an input to generation. A fault-pattern knowledge base catalogs recurring failure modes — off-by-one in tenor calculations, time-zone bugs at day-boundary posting, null-merchant edge cases in fraud scoring, stale-FX-rate races at rollover. Each pattern has a signature (where it hit, why, which control caught or missed it) and an exemplar test. When the router sees high incident density for a capability or a recent regression on a sibling service, it boosts retrieval from the fault KB and asks the generator to produce tests whose assertions mirror the signatures. The KB grows monotonically; once a class of bug is cataloged, it is tested for wherever that capability or a similar one is deployed.

Fault KB as active driver, not passive log

Treat fault patterns as first-class retrieval citizens. Index them, tag them by capability and control, and let the retrieval policy weight them up when recent-incident density is high. This closes the loop between post-incident reviews and next-sprint coverage.

Risk-Based Prioritization

Exhaustive testing is an illusion; prioritized testing is a practice. We score every test candidate on four axes and compute a weighted priority that decides whether the test is mandatory on every change, nightly, or release-only.

Axis	Signal	Example high-score case
Regulatory exposure	Is evidence required for an audit?	AML screening, IFRS 9 ECL, HIPAA PHI access
Financial impact	Maximum plausible loss if the path fails	Settlement, interest accrual, fee calculation
Customer harm	Safety, privacy, or unfair-treatment risk	Credit decline reason codes, clinical alerts
Change frequency	How often the code or rule changes	Pricing engines, promo logic, tax tables

Weights are set per bounded context. A simple working scheme: $P = 0.35 \cdot Reg + 0.30 \cdot Fin + 0.20 \cdot Customer + 0.15 \cdot Change$, with tiers R1–R4 mapped by $P \geq 0.75$, 0.5, 0.25, <0.25. Tests in R1 gate every change; R2 run nightly with full retrieval depth; R3 run on release; R4 run on demand. The score itself is stored with the test, so auditors see not only that a test ran but why it was deemed critical.

Putting It Together

Dynamic RAG closes the loop between what the business mandates, what engineering built, what broke last quarter, and what the auditor will look for next. A request to test a capability triggers classification (intent, risk), retrieval from the right indexes at the right depth with the right embeddings, context engineering to assemble an ordered, deduplicated, conflict-resolved prompt, a functional decomposition into rule/decision/condition/edge, a fault-pattern boost where incident density warrants it, and finally generation plus evaluation against golden sets gated by risk tier. The same pipeline can be run in shadow mode for two weeks, compared against the existing suite, and rolled in capability by capability.

Where to start

Weeks 1 – 2:
Choose two bounded contexts, stand up clause + code + incidents indexes, wire FinBERT-style embeddings, run the first retrieval plans in shadow.

Weeks 3 – 6:
Add functional decomposition and fault-KB boosts; publish golden sets; wire risk tiers into CI gates.

Weeks 7 – 12:
Expand to the top five capabilities; measure coverage-at-risk, mean-time-to-test-new-rule, and audit-evidence completeness.

References

1. Araci, D. (2019). FinBERT: Financial Sentiment Analysis with Pre-trained Language Models. arXiv:1908.10063.
2. Yang, Y., Uy, M. C. S., & Huang, A. (2020). FinBERT: A Pretrained Language Model for Financial Communications. arXiv:2006.08097.
3. Lewis, P. et al. (2020). Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. NeurIPS.
4. Evans, E. (2003). Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley.
5. Robertson, S. & Zaragoza, H. (2009). The Probabilistic Relevance Framework: BM25 and Beyond. FnTIR.
6. NIST (2023). AI Risk Management Framework (AI RMF 1.0).

Author

Mhahesh Muraleedhara
Head of Quality Intelligence
NA, Zensar

zensar
An RPG Company

At Zensar, we're 'experience-led everything.' We are committed to conceptualizing, designing, engineering, marketing, and managing digital solutions and experiences for over 145+ leading enterprises. Using our 3Es of experience, engineering, and engagement, we harness the power of technology, creativity, and insight to deliver impact.

Part of the $4.8 billion RPG Group, we are headquartered in Pune, India. Our 10,000+ employees work across 30+ locations worldwide, including Milpitas, Seattle, Princeton, Cape Town, London, Zurich, Singapore, and Mexico City.

For more information, please contact: **info@zensar.com | www.zensar.com**