Taerae Technical Report: An Embedded Graph Runtime for Dart and Flutter

Taerae Contributors

February 2026

Abstract

This report documents the current implementation of `taerae`, an embedded graph runtime for Dart and Flutter. The codebase provides: (i) an in-memory directed graph engine with immutable node/edge models and indexed lookup paths, (ii) a file-backed durability layer using append-only newline-delimited JSON (NDJSON) logs and periodic snapshots, (iii) GraphRAG extension interfaces with a baseline in-memory cosine-similarity index, and (iv) Flutter adapters for state management and visualization. The report is evidence-constrained to repository artifacts at the time of writing: source code, tests, benchmark harnesses, and project metadata. We include reproducible benchmark and coverage artifacts generated in this workspace, and explicitly separate measured results from broader production claims. In this report, "GraphRAG-ready" means interface-level composability and baseline implementations, not a production-validated retrieval quality claim.

1 Introduction

`taerae` targets application-embedded graph workloads where developers need graph modeling and traversal without a separate graph server process [18]. The project is organized as two packages: `taerae_core` (pure Dart runtime) [19] and `flutter_taerae` (Flutter integration and re-exports) [17]. Package manifests specify Dart SDK constraint `3.11.x or newer` and Flutter SDK `>=3.3.0`, aligning with current Dart/Flutter toolchains [2, 5].

The system explicitly combines graph execution and local durability. The durability layer uses write-ahead style mutation logging plus snapshots, a standard approach for recovery-oriented storage engines [14]. In parallel, the code exposes retrieval-augmented interfaces (embedder, vector index, chunker, filter, reranker) so graph state can be connected to retrieval workflows related to RAG-style pipelines [11].

This report focuses on implemented behavior. It does not claim distributed graph database properties, ANN index performance, or benchmark throughput numbers that are not present in checked-in artifacts.

2 Related Work and Positioning

Taerae's durability path (operation log + checkpointed snapshots) follows established recovery patterns used in database systems [14]. The implementation target is different from server-grade graph databases: Taerae is embedded and application-local, whereas many graph systems prioritize remote query serving, distributed storage, and cluster operations.

For retrieval-augmented pipelines, Taerae is positioned after foundational RAG work [11] and subsequent studies on retrieval faithfulness, retrieval strategy, and evaluation frameworks [9, 6, 16, 4]. These works motivate why retrieval quality, ranking, and evidence grounding should be treated as first-class concerns when integrating language models with external memory.

Graph-centric RAG methods further inform Taerae's API direction. Recent systems construct or exploit graph structure explicitly during retrieval and reasoning, including GraphRAG [3], graph-text retrieval for question answering [10], memory-oriented graph retrieval [8], lightweight graph retrieval pipelines [7], path-constrained retrieval [1], structure-aware retrieval [12], and graph-oriented retrieval orchestration [20].

Adjacent graph+LLM literature explores symbolic-structural reasoning and integration patterns beyond classic vector retrieval [13, 15]. Compared with these lines, Taerae currently contributes an embedded runtime substrate (graph operations, persistence, and pluggable retrieval interfaces) rather than a task-specific SOTA reasoning model or a fixed production GraphRAG recipe.

All newly added related-work citations in this section are sourced from arXiv and include explicit arXiv identifiers in the bibliography.

3 System Design

3.1 Repository-Level Architecture

The codebase is layered:

1. **Core runtime (`taerae_core`)**: graph models, in-memory graph engine, persistence, and GraphRAG interfaces.

2. **Flutter package (`flutter_taerae`)**: `ChangeNotifier`-based controller, graph visualization widget, and platform-channel wrapper.

3. **Examples (`examples/`)**: domain scenarios (city commute, delivery ops, social recommendation, personal-notes GraphRAG) plus minimal usage samples.

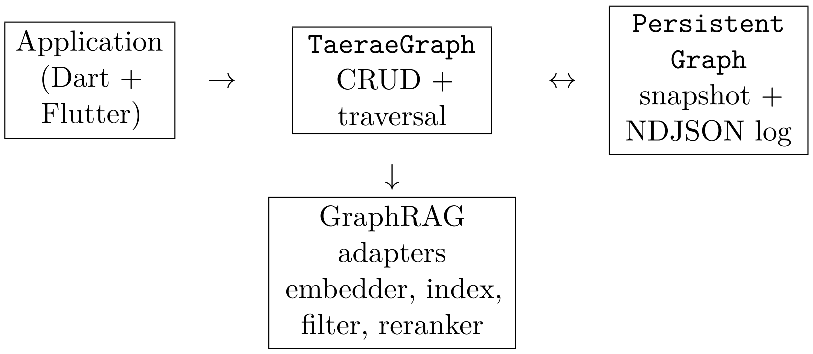
Application (Dart + Flutter)
TaeraeGraph CRUD + traversal
Persistent Graph snapshot + NDJSON log
GraphRAG adapters embedder, index, filter, reranker

Figure 1: High-level runtime data flow in the current architecture.

3.2 Data and State Model

The core data model consists of immutable `TaeraeNode` and `TaeraeEdge`. Constructor and JSON parsing paths enforce non-empty IDs and JSON shape validation. Property values are recursively frozen to prevent accidental mutation after construction.

The mutable execution object is `TaeraeGraph`, which maintains:

1. node and edge maps keyed by ID,

2. outgoing/incoming adjacency maps by node ID,

3. label index (`label -> node IDs`),

4. exact property index (`property key -> hashed value -> node IDs`).

This indexing design supports direct CRUD with synchronous query APIs such as `nodeById`, `nodesByLabel`, and `nodesWhereProperty`. It also supports neighborhood and path queries via `neighbors` and `shortestPathBfs`.

3.3 Persistence and Recovery Layout

`TaeraePersistentGraph` stores data in two files under an application directory:

1. snapshot file (default `graph.snapshot.json`),
2. append-only operation log (default `graph.log.ndjson`).

Startup reads snapshot first, then replays the operation log into memory. Checkpointing writes a full snapshot and truncates the log.

4 Core Algorithms

4.1 Graph Mutation and Index Maintenance

`upsertNode` and `upsertEdge` follow update-or-insert semantics. When updating:

1. old index entries are removed,
2. new immutable objects are inserted,
3. affected indexes are rebuilt.

Removing a node also removes all incident edges and cleans adjacency/index state.

4.2 Traversal and Path Search

`shortestPathBfs` performs directed BFS from `startId` over outgoing edges and reconstructs the first shortest path to `endId`. Optional `edgeType` filtering constrains traversal. If start/end nodes are missing or unreachable, the method returns `null`.

4.3 Property Equality Semantics

Property index keys are wrapped in an internal value-key object that performs deep equality and deep hashing for nested `List`/`Map` values. This allows exact matching of structured property values (for example, nested map equality) while retaining hash-map lookup behavior.

4.4 GraphRAG Retrieval Pipeline

`TaeraeGraphRag` composes three required components (`graph`, `embedder`, `vectorIndex`) and optional components (`chunker`, `filter`, `reranker`). The retrieval flow is:

1. embed query text,
2. retrieve up to `topK * 4` vector hits,
3. map chunk IDs back to node IDs and keep each node's best score,
4. apply optional metadata filter,
5. expand neighborhood context by hop-limited BFS over graph adjacency,
6. apply optional reranker.

4.5 Vector Similarity Baseline

The default index (`TaeraeInMemoryVectorIndex`) stores one vector per ID and computes cosine similarity:

$$\mathrm{sim}(q, v) = \frac{q \cdot v}{\|q\|_2 \|v\|_2}. \tag{1}$$

Search performs exact scoring over all indexed vectors and sorts descending by score (with ID tie-break), so complexity is linear in indexed vectors for scoring and superlinear in sort cost.

4.6 Complexity Summary

5 Durability & Recovery

5.1 Threat Model and Non-Goals

The implemented durability path is best interpreted under a single-process, local-file threat model:

1. process crash or abrupt termination during append/checkpoint,
2. malformed or truncated trailing log line due to interrupted write,
3. local restart with replay from last valid snapshot and log suffix.

Current code and tests do *not* claim guarantees for concurrent multi-writer access, distributed consensus, or Byzantine storage behavior.

Operation	Asymptotic complexity in the current implementation
`upsertNode`	$O(L+P)$ index updates (L: labels, P: properties)
`upsertEdge`	$O(1)$ average map/set operations plus endpoint checks
`removeNode`	$O(L+P+d_{in}+d_{out})$ including incident-edge cleanup
`nodesByLabel`	$O(k)$ for matched nodes
`nodesWhereProperty`	$O(k)$ for matched nodes
`shortestPathBfs`	$O(\|V\|+\|E\|)$ worst-case traversal
`vector search`	$O(Nd)+O(N\log N)$ for exact cosine scoring and full sorting

Table 1: Algorithmic profile derived from source-level implementation.

5.2 Operation Logging

Every persistent mutation is serialized as a typed operation (`upsert_node`, `remove_node`, `upsert_edge`, `remove_edge`, `clear`) and appended as one NDJSON line. The log reader parses line-by-line and can optionally tolerate a malformed trailing line if the file does not end with a line terminator (crash-truncated append case).

5.3 Durability Policy Controls

The persistence layer exposes four knobs:

1. `logFlushPolicy`: `immediate`, `everyNOperations`, `onCheckpoint`,
2. `flushEveryNOperations`: batch size for periodic flush,
3. `writeAtomicityPolicy`: `writeAhead` vs `inMemoryFirst`,
4. `atomicSnapshotWrite`: temp-file + rename snapshot commit.

These settings define the write-latency vs crash-window trade-off in local storage behavior.

5.4 Checkpoint and Close Semantics

`checkpoint()` forces log flush, writes snapshot, truncates log, and resets pending counters. `close(checkpointOnClose: false)` skips snapshot compaction but still flushes the log, enabling later replay on reopen.

5.5 Evidence from Tests

Persistence tests cover:

1. replay correctness and snapshot round-trip,
2. auto-checkpoint log compaction behavior,
3. strict vs tolerant truncated-log recovery,
4. durability option validation and closed-state mutation rejection.

This test set exercises nominal and defensive branches in durability and recovery code paths.

6 Flutter Integration

6.1 Controller Layer

`TaeraeGraphController` wraps `TaeraeGraph` with `ChangeNotifier`. Mutation methods invalidate cached node/edge lists and notify listeners. Query methods delegate directly to core graph APIs. Import/export is provided as both map and string JSON interfaces.

6.2 Visualization Layer

`TaeraeGraphView` renders graph state via:

1. `AnimatedBuilder` subscribed to controller updates,
2. deterministic circular layout fallback (or custom layout callback),
3. edge painting with arrowheads and labels,
4. node/edge hit-testing callbacks,
5. optional pan/zoom through `InteractiveViewer`.

6.3 Platform Plugin Surface

The plugin wrapper currently exposes one method-channel API for platform-version reporting. Android, iOS, macOS, Linux, and Web stubs report platform versions. No graph operations are delegated through native channels at this stage.

7 Evaluation

7.1 Evaluation Goals and Setup

This evaluation targets three questions: (i) throughput and latency of core graph operations, (ii) scaling behavior as graph size increases, and (iii) test/coverage evidence for implemented code paths. All measurements were generated locally during manuscript preparation from repository scripts and tests.

Primary benchmark environment (from run metadata):

1. macOS 15.7.4 (ARM64), 10 logical CPU cores,
2. Dart SDK 3.11.0,
3. benchmark driver: `paper_benchmark.dart` in the core benchmark directory.

7.2 Benchmark Methodology

The extended benchmark run used:

```
dart run benchmark/paper_benchmark.dart
-presets=generic,social,delivery,notes_rag
-sizes=1000,5000,10000 -warmup-runs=1 -repeat=3
-lookup-queries=10000 -path-queries=500
```

and produced artifacts under the `arxiv_report_20260222_full` benchmark results directory.

For each metric, throughput is computed as:

$$\text{ops/s} = \frac{N_{\text{ops}}}{T_{\text{sec}}}, \tag{2}$$

and average per-operation latency is:

$$\mu s/\text{op} = \frac{10^6}{\text{ops/s}}. \tag{3}$$

Run-to-run variability is summarized with the coefficient of variation:

$$\mathrm{CV}(\%) = 100 \cdot \frac{\sigma}{\mu}, \tag{4}$$

where μ and σ are the sample mean and sample standard deviation across measured runs.

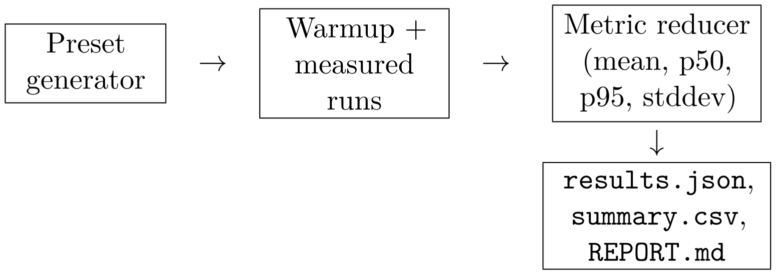
Preset generator
Warmup + measured runs
Metric reducer (mean, p50, p95, stddev)
results.json, summary.csv, REPORT.md

Figure 2: Benchmark data pipeline used for reproducible artifact generation.

7.3 Benchmark Results

Table 2 summarizes key metrics at 5k nodes per preset (three measured runs each).

Preset	upsertNode	upsertEdge	nodeById	path query
generic	255,749	1,117,305	9,047,390	1,269
social	567,632	1,479,535	6,522,060	674
delivery	442,191	1,143,105	6,746,734	2,858
notes_rag	376,718	1,165,578	6,012,800	1,692

Table 2: Mean throughput (ops/s) at 5k nodes. Path query denotes each preset's shortest-path metric.

Scaling behavior differs by operation class (Table 3). In this workload harness, `nodeById` throughput increases with larger graph sizes, while shortest-path throughput decreases as expected with deeper traversal frontiers and larger reachable subgraphs.

Variability is generally low for point lookups and higher for neighbor-heavy workloads. At 5k nodes, generic shortest-path BFS has CV = 0.46%. In contrast, notes_rag and social outgoing-neighborhood metrics show higher variance, with CV values of 18.76% and 15.77%, respectively.

Preset	nodeById@1k	nodeById@10k	ratio	shortest ratio
generic	3,725,981	14,079,242	3.78×	0.09×
social	2,060,239	11,274,345	5.47×	0.08×
delivery	2,056,398	11,339,977	5.51×	0.09×
notes_rag	2,053,841	11,060,564	5.39×	0.09×

Table 3: Scaling summary. Ratio = nodeById(10k)/nodeById(1k). Shortest ratio = shortest-path(10k)/shortest-path(1k).

7.4 Test Suite and Coverage Evidence

From `taerae_core/test` and `flutter_taerae/test`, this workspace currently includes:

1. 58 core `test(...)` cases,
2. 11 Flutter `test(...)` cases,
3. 3 Flutter `testWidgets(...)` cases.

Total: 72 Dart/Flutter test cases executed in this analysis run.

Line coverage is reported with `lcov` as:

$$\text{Coverage}(\%) = 100 \cdot \frac{LH}{LF}, \tag{5}$$

where LH is covered lines and LF is instrumented lines.

Package	LH	LF	Coverage
taerae_core	791	791	100.00%
flutter_taerae	281	297	94.61%
combined	1,072	1,088	98.53%

Table 4: Line coverage from local `lcov.info` artifacts generated in this run.

Coverage gaps are concentrated in the Flutter visualization layer. `taerae_graph_view.dart` is 183/199 lines (91.96%), while other measured Flutter library files are 100%. Remaining uncovered branches are mainly rendering and interaction edge paths rather than core graph correctness logic.

7.5 Artifact Inventory and Reproducibility

The evaluation in this report references two concrete output directories:

1. `arxiv_report_20260222` (earlier small run),
2. `arxiv_report_20260222_full` (extended run with 4 presets, 3 sizes, 3 repeats).

Each directory includes `results.json`, `summary.csv`, and `REPORT.md`. Rerunning with the same CLI arguments and seed policy reproduces the same measurement procedure and artifact schema, though absolute throughput values may vary across machines and thermal states.

8 Limitations

1. **Single-machine benchmarking**: despite multiple presets and graph sizes, all measurements come from one local machine and should not be interpreted as cross-platform performance guarantees.
2. **Exact vector search only**: `TaeraeInMemoryVectorIndex` performs full scans and full sort; no ANN backend or persisted vector index is provided.
3. **No end-task GraphRAG quality study**: current evidence shows interface and pipeline execution behavior, not downstream task quality metrics.
4. **Single-process semantics**: there is no explicit transaction, lock manager, or multi-writer coordination API.
5. **Web persistence gap**: persistence relies on `dart:io` and is not available in Flutter Web runtime paths.
6. **Minimal plugin channel**: native plugin surface is limited to platform-version smoke checks.
7. **CI path mismatch**: workflow jobs still reference `taerae_flutter` in places, while the actual directory is `flutter_taerae`; this can affect default CI evidence collection.

9 Future Work

1. Add automated benchmark publishing (store measured benchmark artifacts per commit/tag).
2. Introduce pluggable ANN and persisted vector backends for larger retrieval workloads.
3. Add transactional APIs and explicit concurrency semantics for multi-isolate or multi-client mutation scenarios.
4. Provide web-compatible persistence adapters for browser deployments.
5. Expand Flutter/native integration beyond platform-version checks (for example, storage and instrumentation hooks).
6. Add fault-injection and crash-recovery integration tests that exercise OS-level interruption points.

10 Conclusion

`taerae` already implements a coherent embedded graph stack: indexed in-memory graph operations, configurable local durability with replayable logs and snapshots, GraphRAG extension points, and Flutter-first controller/view integration. Current evidence includes benchmark artifacts across four workload presets and three graph scales, plus high line coverage in core runtime paths. The remaining gap to publication-grade systems evidence is broader multi-machine benchmarking, richer GraphRAG end-task evaluation, and continuously published performance and fault-injection traces across releases.

References

[1] Weijie Chen et al. Pathrag: Pruning graph-based retrieval augmented generation with relational paths. *arXiv preprint*, 2025. arXiv:2502.14902.

[2] Dart Team. Dart documentation. `https://dart.dev/`, 2026. Accessed: 2026-02-22.

[3] Darren Edge, Ha Trinh, Newman Cheng, Joshua Bradley, Alex Chao, Apurva Mody, Steven Truitt, and Jon Larson. From local to global:

A graphrag approach to query-focused summarization. *arXiv preprint*, 2024. arXiv:2404.16130.

[4] Shahul Es, Jithin James, Luis Espinosa-Anke, and Steven Schockaert. Ragas: Automated evaluation of retrieval augmented generation. *arXiv preprint*, 2023. arXiv:2309.15217.

[5] Flutter Team. Flutter documentation. `https://docs.flutter.dev/`, 2026. Accessed: 2026-02-22.

[6] Yunfan Gao et al. Retrieval-augmented generation for large language models: A survey. *arXiv preprint*, 2023. arXiv:2312.10997.

[7] Zirui Guo et al. Lightrag: Simple and fast retrieval-augmented generation. *arXiv preprint*, 2024. arXiv:2410.05779.

[8] Bernal Jimenez Gutierrez, Yiheng Shu, Yu Gu, Michihiro Yasunaga, and Yu Su. Hipporag: Neurobiologically inspired long-term memory for large language models. *arXiv preprint*, 2024. arXiv:2405.14831.

[9] Hangfeng He, Hongming Zhang, and Dan Roth. Rethinking with retrieval: Faithful large language model inference. *arXiv preprint*, 2023. arXiv:2301.00303.

[10] Zhitao He et al. G-retriever: Retrieval-augmented generation for textual graph understanding and question answering. *arXiv preprint*, 2024. arXiv:2402.07630.

[11] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive nlp tasks. In *Advances in Neural Information Processing Systems*, volume 33, pages 9459–9474, 2020.

[12] Zhuo Li, Yikai Wang, Xi Yang, Qiang Wang, Fang Chao, Yikang Li, Hanghang Wang, Peng Zhang, Senzhang Wang, and Xifeng Yan. Structrag: Boosting large language model reasoning by inference-time hybrid information structurization. *arXiv preprint*, 2024. arXiv:2410.08815.

[13] Linhao Luo, Xin Sun, Yufei Wang, Fei Wang, Hao Zhao, Wei Chen, and Kun Zhang. Reasoning on graphs: Faithful and interpretable large language model reasoning. *arXiv preprint*, 2023. arXiv:2310.01061.

[14] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems*, 17(1):94–162, 1992.

[15] Shirui Pan, Linhao Luo, Yufei Wang, Chenyu Chen, Jiarui Wang, and Xue Wu. Integrating graphs with large language models: Methods and prospects. *arXiv preprint*, 2023. arXiv:2310.05499.

[16] Parth Sarthi, Salman Abdullah, Arnav Tuli, Shubh Khanna, Micah Goldblum, Alan Hanjalic, Radu Soricut, and Animesh Rawat. Raptor: Recursive abstractive processing for tree-organized retrieval. *arXiv preprint*, 2024. arXiv:2401.18059.

[17] taerae contributors. flutter_taerae package. `https://github.com/JAICHANGPARK/taerae/tree/main/packages/flutter_taerae`, 2026. Accessed: 2026-02-22.

[18] taerae contributors. taerae: Embedded graph runtime for dart and flutter. `https://github.com/JAICHANGPARK/taerae`, 2026. Accessed: 2026-02-22.

[19] taerae contributors. taerae_core package. `https://github.com/JAICHANGPARK/taerae/tree/main/packages/taerae_core`, 2026. Accessed: 2026-02-22.

[20] Haozhen Zhang, Tao Feng, and Jiaxuan You. Gor: Unified graph-oriented retrieval for finding supporting evidence for claims. *arXiv preprint*, 2024. arXiv:2410.11001.