Carrying Text-Fabric Forward:
Context-Fabric and the Scalable Corpus Ecosystem

Cody Kingham

January 2026

Abstract

Text-Fabric provides a powerful framework for analyzing annotated text corpora, but its memory requirements limit scalability: each worker process loads the full corpus into RAM, making parallel API deployments impractical. Context-Fabric restructures the storage layer using memory-mapped arrays, enabling efficient multi-worker access. Benchmarks across 10 corpora demonstrate an average 65% memory reduction (ranging from 20% for small corpora to 92% for large corpora), with 5.3× more efficient multi-corpus scaling. Query latency is competitive: 3% faster overall, with 26% improvement for lexical queries. This scalability enables an ecosystem of applications—web interfaces, AI agents, educational tools—built on corpus APIs rather than local installations.

1 Introduction

1.1 Text-Fabric: A Foundation for Corpus Analysis

Annotated text corpora form the foundation of computational linguistics, digital humanities, and biblical scholarship. Text-Fabric [1] pioneered a graph-based data model for representing hierarchical text structures with arbitrary annotations, enabling researchers to query and analyze corpora ranging from ancient manuscripts to modern linguistic datasets.

Text-Fabric's achievements are substantial. It provided a unified framework for accessing complex annotated corpora like the BHSA (Biblia Hebraica Stuttgartensia Amstelodamensis), making sophisticated linguistic analysis accessible to researchers worldwide. Its Python-native design enabled integration with the broader scientific computing ecosystem.

However, working with Text-Fabric required mastering Python programming and a specialized query syntax. This limited accessibility to technically-skilled researchers willing to invest in learning the tooling.

1.2 The AI Transformation

AI agents change this. Large language models can mediate between researchers and corpus data, translating natural language queries into corpus operations. A researcher no longer needs to write:

```
results = A.search('''
clause
  phrase function=Pred
    word sp=verb vt=perf
''')
```

Listing 1: Traditional Text-Fabric query

Instead, they can simply ask: "Find all clauses with a predicate phrase containing a perfect verb."

Scholars can explore linguistic patterns through conversation rather than code. The barrier drops from "must learn Python" to "must formulate questions."

1.3 New Research Horizons

AI-powered corpus analysis transforms research productivity for all scholars—not only those lacking programming skills. For researchers who can code, AI assistance accelerates existing pipelines: what previously required days of writing custom scripts can now be accomplished in hours through conversational iteration or automated semantic workflows. For those without programming backgrounds, barriers that once blocked entire research directions simply disappear. The implications span textual criticism, historical linguistics, literary analysis, and source criticism—any field where large-scale textual evidence must be gathered and analyzed. AI agents can drive automated pipelines that process and analyze massive datasets, enabling research at scales previously impractical. Consider what becomes tractable:

- **Cross-corpus comparisons**: Compare syntactic patterns in the Hebrew Bible with the Septuagint (LXX), or trace linguistic features across Hebrew, Greek, and Syriac Peshitta—without writing custom integration code.
- **Translation analysis**: "Find all instances where the LXX translates Hebrew *chesed* as Greek *eleos*" becomes a natural language query rather than a multi-corpus programming project.
- **Iterative exploration**: Refine queries through conversation, drilling down into interesting patterns without context-switching between analysis and coding.
- **Comparative linguistics**: "How does verbal aspect marking differ between Biblical Hebrew and Koine Greek?" can drive automated analysis across multiple corpora.

Running AI agents locally is possible, but requires technical expertise: installing Python, managing dependencies, downloading corpora, and configuring the environment. Most researchers lack this background or the time to acquire it.

Hosted API endpoints solve this problem. A corpus service handles the technical complexity; users interact through applications built on top of it. This creates an opportunity for an ecosystem of tools—web interfaces, mobile apps, educational platforms, visualization dashboards—all drawing from the same underlying corpus infrastructure without each reimplementing corpus access.

1.4 Enabling an Application Ecosystem

When corpus analysis becomes a scalable service, Text-Fabric transforms from a research tool into infrastructure—a utility that applications can build upon. This architecture supports:

- **Research applications**: Web-based query interfaces, annotation tools, collaborative workspaces for research teams
- **Educational tools**: Interactive Hebrew Bible courses, syntax visualization for students, gamified learning platforms
- **Third-party AI integrations**: Any AI service can incorporate corpus queries via API, not just purpose-built agents
- **Cross-platform access**: Mobile apps, browser extensions, integrations with existing research software (Zotero, Logos, etc.)

None of these applications need to handle Text-Fabric's 6 GB runtime memory footprint or implement the query engine themselves. They call an API. The API handles concurrency, caching, and resource management. This separation of concerns—corpus infrastructure vs. user-facing applications—is only practical when the infrastructure can scale efficiently.

1.5 The Technical Challenge

To support this ecosystem, Text-Fabric must become productionizable. API servers handle concurrent requests; multiple workers are needed. Each worker must access the full corpus—but Text-Fabric's architecture loads entire corpora into memory, consuming 6+ GB per worker. For a modest deployment with 10 workers, this requires 60+ GB of RAM, making scalable hosting economically impractical.

Context-Fabric addresses this challenge through a memory-mapped architecture that achieves substantial memory reduction (averaging 65%, ranging from 20% for small corpora to 92% for large corpora) while maintaining compatibility with Text-Fabric's Python API (the `F`, `L`, `T`, `S` interfaces researchers use in their code). This paper quantifies these efficiency gains through controlled benchmarking across 10 diverse corpora.

2 Related Work

Memory-efficient corpus processing has been explored through two main approaches: memory-mapped tools optimized for specific analyses, and database-backed systems providing general-purpose access.

2.1 Memory-Mapped Corpus Tools

Colibri Core [21] pioneered memory-mapped n-gram analysis, achieving substantial memory reduction for pattern extraction tasks. However, its scope is limited to n-gram statistics rather than general-purpose corpus queries with arbitrary annotations. Context-Fabric provides a broader API supporting hierarchical structures, edge relationships, and arbitrary node features—the full expressive power of Text-Fabric's data model—while maintaining memory efficiency.

2.2 Database-Backed Systems

Production corpus systems like CQPweb [22] and SketchEngine [23] use SQL backends (typically PostgreSQL or MySQL) for corpus storage. This architecture enables horizontal scaling across machines and provides mature query optimization. However, database round-trips introduce latency that can be problematic for AI agent workloads requiring rapid iterative queries. Context-Fabric trades horizontal scaling for in-process performance: queries execute without network overhead, and the memory-mapped architecture provides efficient single-machine scaling.

2.3 Text-Fabric Ecosystem

Text-Fabric [1] established the data model that Context-Fabric extends. The key innovation was representing annotated corpora as directed graphs with arbitrary node and edge features, enabling complex linguistic queries through a declarative search syntax. Context-Fabric preserves this API while restructuring the storage layer for memory efficiency. Existing Text-Fabric code continues to work unchanged, allowing researchers to benefit from improved scalability without abandoning familiar workflows.

3 From Text-Fabric to Context-Fabric

3.1 Text-Fabric's Data Model

Text-Fabric represents corpora as directed graphs where nodes correspond to textual objects (words, phrases, clauses, sentences, etc.) and edges encode relationships between them. Node and edge features store arbitrary annotations as key-value pairs.

This model captures the hierarchical nature of text: words compose phrases, phrases compose clauses, clauses compose sentences. Each level can carry independent annotations—morphological features on words, syntactic functions on phrases, discourse relations on clauses.

Text-Fabric's caching strategy optimizes for single-user research workflows. Source files are compiled once into gzipped pickle files (`.tfx`), which deserialize into Python dictionaries and lists at load time. For BHSA, this means:

- **Compact cache**: 138 MB on disk (gzip compression)
- **Compilation time**: 62 seconds to process source files
- **Load time**: 8.1 seconds to deserialize from cache
- **Full in-memory access**: All features immediately available

For interactive research sessions lasting hours, an 8-second startup is acceptable.

3.2 Challenges for AI-Era Deployments

Production deployments face different constraints:

1. **Memory duplication**: Each process maintains independent copies of deserialized data structures. With Text-Fabric, 4 workers consume 4× the memory of a single process—limiting deployment to memory-rich (expensive) instances.
2. **Load time accumulation**: In serverless or auto-scaling environments, 7-second cold starts degrade user experience. AI agents expect sub-second responses.
3. **Concurrent access**: API servers must handle multiple simultaneous requests. Text-Fabric's 6 GB footprint leaves little headroom for request handling on typical cloud instances.

3.3 Memory-Mapped Architecture

Memory-mapped I/O (`mmap`) maps file contents directly into a process's virtual address space [13]. Rather than explicitly reading data into buffers, the program accesses file contents through pointers, and the OS transparently pages data in and out as needed.

The technique underpins many performance-critical systems: machine learning frameworks like PyTorch and TensorFlow use memory mapping for loading large model weights and datasets; SQLite optionally supports mmap (disabled by default), while LMDB uses mmap as its core architecture [17]; operating systems use mmap to load executables and shared libraries. Memory mapping offers:

- **Zero-copy access**: Data moves directly between disk and process address space
- **Lazy loading**: Only accessed pages reside in physical memory
- **Automatic eviction**: The OS manages memory pressure
- **Copy-on-write sharing**: Forked processes share pages until modification

NumPy's `memmap` function provides a Python interface for memory-mapping array data [19]. Context-Fabric stores all corpus data as `.npy` files and loads them with `mmap_mode='r'` (read-only), enabling safe multi-process sharing.

3.4 Adapting the TF Data Model

The challenge is converting Text-Fabric's Python-native data structures to memory-mappable formats while preserving the same Python interface that existing code expects.

3.4.1 Dense Arrays for Node Features

Node features with single values per node map directly to numpy arrays. The storage format depends on the feature's value type:

Integer features (verse numbers, word positions) store values directly in appropriately-sized dtypes (`uint8`, `uint16`, `uint32`).

Categorical features with small value sets use index encoding. The node type feature (`otype`), for example, maps each node to one of a handful of type names ("word", "verse", "chapter", etc.). Since corpora typically have fewer than 256 node types, these indices fit in a `uint8` array:

```
# Type names: ["word", "verse", "chapter", ...]
# otype[node_id] = index into type_names
otype = np.array([0, 0, 0, ..., 1, 1, 2, ...], dtype=np.uint8)

# Lookup: O(1)
node_type = type_names[otype[node_id]]  # -> "word"
```

Listing 2: Categorical feature storage

This is conceptually similar to the string pool pattern (described next), but optimized for small, fixed value sets.

3.4.2 String Pools for Text Features

String-valued features (glosses, lexemes, morphological tags) present two challenges. First, Python strings cannot be memory-mapped directly. Second, not every node has a value for every feature—Text-Fabric represents these as `None`, but NumPy integer arrays have no native missing-value representation (unlike floats, which have `NaN`).[1]

Context-Fabric addresses both with a string pool pattern. Unique strings are stored once in an array, with a separate index array mapping each node to its string's position. For missing values, we reserve a sentinel index: `0xFFFFFFFF` (the maximum `uint32` value, $2^{32} - 1$). This value cannot be a valid index—it would require over 4 billion unique strings—so encountering it unambiguously signals "no value." This preserves the distinction between an empty string (a valid value at some index) and a truly absent value.

```
MISSING = 0xFFFFFFFF  # Sentinel for "no value" (max uint32)

# Feature "gloss" for 5 nodes: ["king", "the", "king", None, "house"]
strings = np.array(["king", "the", "house"], dtype=object)  # Unique values
indices = np.array([0, 1, 0, MISSING, 2], dtype=np.uint32)  # Per-node index

def lookup(node):
    idx = indices[node]
    return None if idx == MISSING else strings[idx]

lookup(2)  # -> "king"
lookup(3)  # -> None (missing value)
```

Listing 3: String pool structure

[1] While float arrays could use `NaN` for missing values, storing indices as floats wastes memory (float64 is 8 bytes vs. uint16's 2 bytes for most features), requires type conversion on every lookup, and is semantically misleading since indices are inherently integers.

The `indices` array is memory-mapped (`uint32`, one entry per node). The `strings` array is small—most features have far fewer unique values than nodes—and loads into memory. For BHSA's `gloss` feature, 426,590 words map to roughly 9,000 unique glosses.

3.4.3 Compressed Sparse Row Format

Some corpus data involves variable-length sequences. Consider `oslots`, which maps each non-slot node to the word positions it contains: a 3-word phrase contains slots `[5, 6, 7]`, while a 20-word verse contains `[100, 101, ..., 119]`. In Python, Text-Fabric stores this as a tuple of tuples—but Python tuples are pointers to objects scattered across memory, which cannot be memory-mapped.

Context-Fabric solves this with Compressed Sparse Row (CSR) format: concatenate all sequences into one flat `data` array, and use a separate `indptr` (index pointer) array to record where each sequence begins and ends:

```
# oslots for 3 nodes: phrase contains [5,6,7], clause contains [5,6,7,8], verse
    contains [5..12]
# Concatenate all slots into one array:
data   = [5, 6, 7, 5, 6, 7, 8, 5, 6, 7, 8, 9, 10, 11, 12]

# Record boundaries (one entry per node, plus final endpoint):
indptr = [0, 3, 7, 15]

# Lookup: node i's slots are data[indptr[i] : indptr[i+1]]
phrase_slots = data[0:3]   # -> [5, 6, 7]
clause_slots = data[3:7]   # -> [5, 6, 7, 8]
verse_slots  = data[7:15]  # -> [5, 6, 7, 8, 9, 10, 11, 12]
```

Listing 4: CSR format for oslots

Both arrays are flat numpy arrays that can be memory-mapped. The `indptr` array has length $N + 1$ (one boundary per node, plus the final endpoint). Empty sequences cost nothing: when `indptr[i] == indptr[i+1]`, the slice is empty.

For edge features with associated values (e.g., linguistic dependency weights), `CSRArrayWithValues` adds a parallel values array:

```
# Edge feature "distance": node 0 -> {10: 100, 20: 200}, node 1 -> {}, node 2
    -> {30: 300}
# (node 0 connects to nodes 10 and 20 with distances 100 and 200)
indptr  = [0, 2, 2, 3]       # Boundaries
indices = [10, 20, 30]       # Target nodes
values  = [100, 200, 300]    # Associated values (e.g., distances)

def get_edges(node):
    start, end = indptr[node], indptr[node + 1]
    return indices[start:end], values[start:end]

get_edges(0)  # -> ([10, 20], [100, 200])  node 0's targets and distances
get_edges(1)  # -> ([], [])                node 1 has no edges
get_edges(2)  # -> ([30], [300])           node 2 connects to 30
```

Listing 5: CSR with values

All arrays use `uint32` dtype and load via `mmap_mode='r'` (read-only), enabling safe multi-process sharing without copy-on-write overhead.

3.4.4 The .cfm Format

Context-Fabric's compiled format (`.cfm`) organizes data into a directory structure:

Data	Purpose	Why CSR
oslots	Node → slot mapping	Variable slots per node
levUp	Embedder lookup	Variable hierarchy depth
levDown	Embedded children	Variable children per node
boundaries	L.p()/L.n() navigation	Sparse: most slots have 0–2
Edge features	Node relationships	Sparse connectivity

Table 1: Data structures using CSR format

```
corpus/                    # Corpus directory (e.g., bhsa/)
  .cfm/                    # Hidden compiled-format subdirectory
    meta.json              # Corpus metadata
    warp/
      otype.npy            # Node types (uint8)
      oslots_indptr.npy    # CSR boundaries
      oslots_data.npy      # CSR slot data
    computed/
      order.npy            # Traversal order
      rank.npy             # Node ranks
      levup_indptr.npy     # Embedder CSR
      levup_data.npy
    features/
      sp.npy               # Part of speech (dense)
      gloss_idx.npy        # String pool indices
      gloss_strings.npy    # Unique strings
    edges/
      mother_indptr.npy    # Edge CSR
      mother_data.npy
```

Listing 6: .cfm directory structure

This structure trades disk space for memory efficiency and parallelism.

3.5 Benefits Across Use Cases

The memory-mapped architecture benefits both production deployments and individual researchers:

For APIs and AI agents:

- Multiple workers share memory-mapped pages via copy-on-write
- Sub-second load times enable responsive scaling
- Up to 92% memory reduction (for large corpora) makes deployment economically viable

For individual researchers:

- Up to 12.9× faster load times improve interactive experience
- 65% average memory reduction enables work on modest hardware (laptops, older machines)
- Same corpus, same API—no workflow changes required

The trade-off is a larger cache (859 MB vs. 138 MB for BHSA) and longer initial compilation (91s vs. 8s for BHSA). This one-time cost pays off immediately: every subsequent session benefits from faster loads and substantially lower memory consumption.

3.6 Iterative Optimization

The memory-mapped architecture described above achieved the target memory reduction, but initial benchmarks revealed query latency issues. This section describes the iterative refinement process that led to competitive query performance.

3.6.1 The Vectorization Story

Initial memory-mapping achieved 90% memory reduction but introduced query latency from page faults. Benchmarking revealed that per-node feature lookups—inherited from Text-Fabric's loop-based approach—triggered thousands of page faults per query. Each feature access required loading a new memory page.

The solution: vectorized numpy operations that process all nodes at once. Instead of:

```python
for node in candidate_nodes:
    if feature.value(node) == target_value:
        result.add(node)
```

Listing 7: Per-node iteration (original)

Context-Fabric now uses:

```python
node_array = np.array(list(candidate_nodes))
values = feature_data[node_array - 1]
mask = values == target_index
result = set(node_array[mask])
```

Listing 8: Vectorized filtering

This change reduces Python function calls from O(n) to O(1) and enables CPU cache prefetching across contiguous memory. In simulation tests, vectorization achieves 8.2× algorithmic speedup for feature filtering operations. In real-world benchmarks, this translates to 26% faster lexical queries—the vectorized approach compensates for mmap overhead while preserving memory efficiency.

This optimization was only possible *because* of the numpy array architecture. The same design that enables memory-mapping also enables vectorization. Text-Fabric's dictionary-based storage cannot leverage these array operations.

3.6.2 Embedding Preloading

Early benchmarks also revealed that embedding-heavy queries (using `[[` and `]]` relations) were substantially slower due to CSR traversal page faults. The hierarchical embedding structures are accessed randomly during query execution, defeating the OS's sequential read-ahead optimization.

Rather than accepting this limitation, we implemented optional preloading: loading the CSR embedding structures into RAM at corpus initialization. For BHSA, this costs approximately 100 MB but provides 1.7× speedup for embedding queries, bringing CF to parity with TF for embedding-heavy queries.

Preloading is enabled by default but configurable via the `CF_EMBEDDING_CACHE` environment variable. Memory-constrained deployments can disable it:

```bash
export CF_EMBEDDING_CACHE=off
```

This iterative approach—benchmark, identify bottlenecks, optimize, re-benchmark—demonstrates that Context-Fabric's architecture was refined through empirical testing rather than theoretical design alone.

4 Benchmarking Methodology

To quantify Context-Fabric's improvements, we conducted systematic benchmarks across memory efficiency, multi-corpus scaling, and query latency. This section describes the methodology; results follow in Section 5.

4.1 Test Environment

All benchmarks were conducted on:

- **Hardware**: Apple M1 Pro, 8 cores, 32 GB RAM, NVMe SSD
- **OS**: macOS Darwin 24.5.0 (arm64)
- **Python**: 3.13.11
- **NumPy**: 2.4.0
- **Text-Fabric**: 13.0.19
- **Context-Fabric**: 0.5.0

Production Linux deployments may show different absolute numbers but similar relative performance characteristics.

4.2 Test Corpora

Benchmarks span 10 corpora representing three orders of magnitude in size, from 1.6 MB to 1.1 GB. Table 2 lists the corpora in order of increasing size.

Table 2: Test corpora used in benchmarks, ordered by TF cache size.

Corpus	**Description**	**TF Cache Size**
CUC [11]	Copenhagen Ugaritic Corpus	1.6 MB
Tischendorf [9]	Tischendorf 8th Edition Greek NT	34 MB
SyrNT [7]	Syriac New Testament	52 MB
Peshitta [6]	Syriac Old Testament	55 MB
Quran [10]	Quranic Arabic Corpus	73 MB
SP [8]	Samaritan Pentateuch	147 MB
LXX [3]	Septuagint	268 MB
N1904 [4]	Nestle 1904 Greek NT	319 MB
DSS [5]	Dead Sea Scrolls	936 MB
BHSA [2]	Biblia Hebraica (ETCBC)	1.1 GB

This diversity enables analysis of how memory efficiency scales with corpus size. The BHSA (largest corpus) provides richly annotated linguistic data for the Hebrew Bible, containing 1,446,831 total nodes, 426,590 word-level nodes, 13 node types, 109 node features, and 6 edge features.

4.3 Measurement Protocol

Each benchmark configuration runs 10 independent measurement runs following 2 warmup runs (excluded from statistics). Measurements use subprocess isolation via `multiprocessing.spawn` to ensure clean memory state, with explicit `gc.collect()` before measurement.[2]

[2]Load time refers to the time from initiating corpus load to having a usable API handle, measured using Python's `time.perf_counter()`.

4.4 Memory Metrics

Operating systems provide several metrics for quantifying process memory [15]. **Resident Set Size (RSS)** measures physical memory pages mapped to a process, including private allocations and shared pages [12]. **Unique Set Size (USS)** counts only pages unique to a process, excluding shared memory. **Proportional Set Size (PSS)** divides shared pages proportionally among processes [14].

We use RSS for all measurements. While RSS double-counts shared pages in multi-process scenarios, this "worst case" view provides transparency: the numbers represent actual memory pressure on the system. USS would dramatically undercount Context-Fabric's memory-mapped data (potentially showing near-zero for file-backed pages), while PSS is unavailable on macOS and fluctuates as processes start and stop.

4.5 Measurement Isolation

To ensure accurate measurements uncontaminated by prior allocations, each scenario runs in an isolated subprocess:

```python
def _measure_cache_load(source: str, result_queue):
    """Runs in a clean spawned subprocess."""
    # Load corpus from cache
    fabric = Fabric(locations=source)
    api = fabric.load()

    # Measure total RSS after loading
    gc.collect()
    total_rss = psutil.Process().memory_info().rss
    result_queue.put(total_rss)

# Main process spawns clean subprocess
ctx = multiprocessing.get_context('spawn')
p = ctx.Process(target=_measure_cache_load, args=(source, queue))
p.start()
```

Listing 9: Subprocess isolation pattern

This pattern ensures:

1. No residual memory from compilation or prior measurements
2. Consistent baseline across Text-Fabric and Context-Fabric
3. Reproducible results across runs

4.6 Deployment Scenarios

We evaluate three scenarios representing common deployment patterns:

4.6.1 Single Process

A single Python process loads the corpus from cache and serves requests. This represents interactive research use or single-threaded batch processing.

Measurement: Total RSS of subprocess after cache load.

4.6.2 Spawn Mode (4 Workers)

Four independent worker processes, each loading the corpus from scratch. This simulates `multiprocessing` with `spawn` context—the default on macOS and the only option on Windows.[3]

Measurement: Sum of RSS across all 4 worker processes.

4.6.3 Fork Mode (4 Workers)

A main process pre-loads the corpus, then forks 4 worker processes. Workers inherit the loaded corpus via copy-on-write semantics. This simulates production deployments like `gunicorn -preload`. Fork mode is the default on Linux systems in Python 3.13 and earlier.[4]

Measurement: Main process RSS + sum of worker RSS.

Note: For fork mode, summing RSS across processes may double-count shared copy-on-write pages. This is intentional—we report the "worst case" RSS sum rather than attempting to deduplicate shared pages, which would require USS and reintroduce the transparency issues discussed above.

4.7 Query Performance Methodology

Query latency was measured using 100 curated search templates on the BHSA corpus, categorized into four types:

- **Lexical queries** (30): Feature-based filtering (e.g., find all verbs, filter by part of speech)
- **Structural queries** (30): Pattern matching across hierarchical structures
- **Quantified queries** (20): Queries using quantifiers (`<`, `>`, `..`)
- **Complex queries** (20): Multi-constraint queries combining relations and features

Each query executes 50 times (5 runs × 10 iterations per run), with warmup iterations excluded. We report mean latency, standard deviation, and 95% confidence intervals.

4.8 Statistical Analysis

Results report mean values with 95% confidence intervals computed from 10 measurement runs per configuration. Linear regression models characterize scaling behavior, reporting slope, intercept, and R^2 values. Memory reduction percentages are computed as $(M_{\text{TF}} - M_{\text{CF}})/M_{\text{TF}} \times 100$.

5 Results

The benchmark suite covers three dimensions of performance: memory efficiency across 10 diverse corpora, multi-corpus scaling behavior, and query latency across 100 representative queries. This section presents each dimension in turn.

[3]Windows lacks a `fork()` system call entirely. On macOS, `fork()` is available but Python 3.8+ defaults to `spawn` because forking processes that use certain Apple frameworks (Cocoa, Core Foundation) can cause crashes. Production deployments can explicitly use `fork` via `multiprocessing.get_context('fork')` when the application avoids these frameworks.

[4]Python 3.14 (October 2025) changed the default to `forkserver` on Linux. Applications requiring fork must explicitly request it via `multiprocessing.get_context('fork')`.

5.1 Memory Efficiency Across Corpora

Table 3 presents memory consumption for single-process corpus loading across all 10 benchmarked corpora, ordered by size. The results demonstrate a clear relationship between corpus size and memory reduction: smaller corpora exhibit modest reductions (20% for CUC at 1.6 MB), while larger corpora show dramatic improvements (92% for BHSA at 1.1 GB). The average memory reduction across all corpora is 65% (±21%).

Table 3: Memory efficiency comparison across 10 corpora in single-process mode. Corpora ordered by TF memory footprint. Load time speedup calculated as TF load time / CF load time.

Corpus	TF (MB)	CF (MB)	Reduction	Load Speedup
CUC	164	131	20%	0.52×
Tischendorf	304	146	52%	1.62×
Quran	401	187	54%	1.29×
Peshitta	404	173	57%	0.96×
Syrnt	515	145	72%	3.08×
SP	644	174	73%	4.25×
LXX	1,395	388	72%	1.25×
N1904	1,566	411	74%	2.41×
DSS	3,233	337	90%	6.24×
BHSA	6,292	524	92%	12.91×
Mean	1,492	262	65%	3.45×

The distribution shows that memory reduction scales approximately with corpus size. For the smallest corpus (CUC), Context-Fabric achieves only 20% reduction because the baseline overhead of Python's runtime and library imports dominates the measurement. For the largest corpus (BHSA), reduction reaches 92%—a 12-fold decrease in memory footprint. Figure 1 visualizes this relationship.

The load time results exhibit more variance. Context-Fabric achieves substantial speedups for larger corpora (12.9× for BHSA, 6.2× for DSS), but shows neutral or slightly slower load times for smaller corpora. This pattern reflects a fundamental tradeoff: Context-Fabric's memory-mapped architecture incurs fixed initialization costs that dominate for small datasets but amortize effectively as corpus size grows.

5.2 Multi-Corpus Scaling Analysis

A key advantage of memory-mapped architecture emerges when loading multiple corpora simultaneously—a common requirement for comparative linguistic analysis. Table 4 summarizes the progressive loading results, where corpora are loaded incrementally in order of increasing size.

Note that the step 10 TF mean (5,529 MB) is anomalously lower than step 9 (6,088 MB). This probably results from Python's garbage collector triggering non-deterministically at high memory: two of five runs showed ∼1,750 MB drops mid-measurement, while others showed expected growth. The resulting variance (±949 MB for TF vs. ±7 MB for CF) reflects this unpredictability rather than true memory reduction.

Polynomial regression reveals distinct scaling characteristics. Both implementations fit quadratic models better than linear (R^2 improves from ∼0.83 to ∼0.98), but the critical difference is the quadratic coefficient: TF grows as $126n^2$ MB while CF grows as only $19n^2$—a 6.5× difference in compounding overhead. For practical corpus counts (≤20), CF's weak quadratic term is negligible; TF's is not. The linear approximation (TF: 677 MB/corpus, CF: 127 MB/corpus) captures first-order behavior but understates TF's compounding overhead at scale.

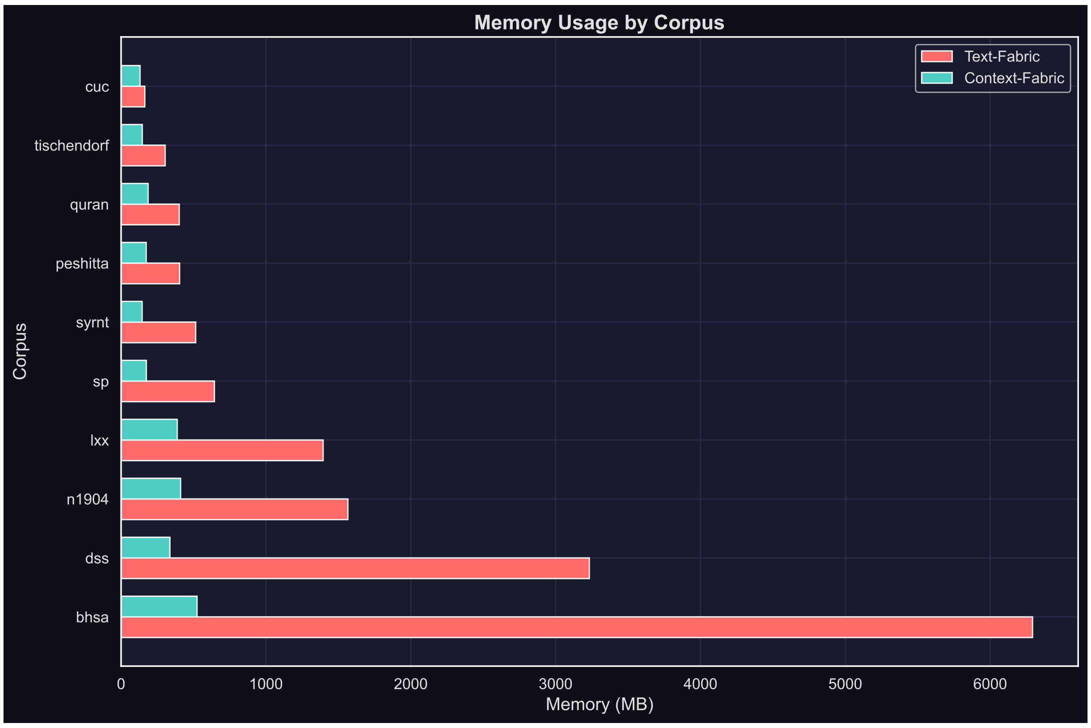

Figure 1: Memory consumption comparison across 10 corpora, sorted by Text-Fabric memory footprint. Error bars show 95% confidence intervals over 10 measurement runs.

Table 4: Memory consumption during progressive multi-corpus loading. Values represent mean total RSS (MB) across 5 runs after loading the specified number of corpora.

Corpora Loaded	**TF (MB)**	**CF (MB)**	**Reduction**	**Ratio**
1 (CUC)	163	130	20%	1.3×
2 (+Tischendorf)	306	146	52%	2.1×
3 (+Syrnt)	641	161	75%	4.0×
4 (+Peshitta)	866	204	76%	4.2×
5 (+Quran)	1,070	259	76%	4.1×
6 (+SP)	1,531	300	80%	5.1×
7 (+LXX)	2,717	551	80%	4.9×
8 (+N1904)	4,044	791	80%	5.1×
9 (+DSS)	6,088	968	84%	6.3×
10 (+BHSA)	5,529	1,348	76%	4.1×

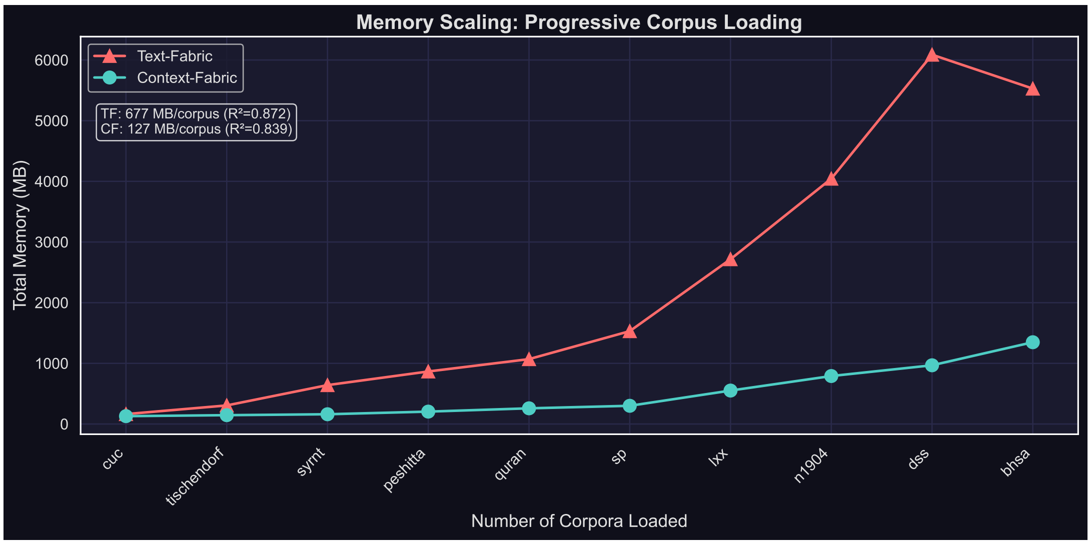

Figure 2: Memory scaling during progressive corpus loading. Quadratic fits show TF grows as $126n^2$ MB (significant compounding overhead) vs. CF at $19n^2$ ($6.5\times$ smaller quadratic coefficient).

The extrapolated predictions are telling. At 10 corpora, the model predicts TF memory of 5,344 MB versus CF memory of 1,057 MB. At 50 corpora (a realistic scenario for comprehensive cross-linguistic research), the gap widens dramatically: 32,443 MB for TF versus 6,137 MB for CF. What appears as a 4–5× advantage at small scale compounds to an architectural necessity at large scale.

A major limitation of Text-Fabric's architecture is its reliance on Python's native data structures. While these provide flexibility for interactive research, they impose substantial memory overhead in multi-corpus deployments. Each dictionary resize, each reference count update, each garbage collection tracking entry contributes to compounding overhead. The effect is superlinear scaling—each additional corpus adds not only its own data but also contributes to fragmentation and memory management overhead across the entire process.

5.3 Query Latency Performance

Query performance was evaluated across 100 queries spanning four categories: lexical lookups (30 queries), structural pattern matching (30 queries), quantified searches (20 queries), and complex multi-constraint queries (20 queries). Table 5 summarizes performance by category.[5]

Table 5: Mean query latency (ms) by category. Positive speedup indicates CF is faster; negative indicates CF is slower.

Category	**Queries**	**TF (ms)**	**CF (ms)**	**Speedup**
Lexical	30	221.5	164.0	+26%
Structural	30	179.2	190.5	−6%
Quantified	20	297.3	286.1	+4%
Complex	20	534.5	554.9	−4%
Overall	100	286.0	278.2	+3%

The results reveal a nuanced performance profile. Context-Fabric demonstrates a 26%

[5] All query benchmarks were run with CF's embedding preloading enabled (the default). See Section 6 for discussion of the preloading trade-off.

speedup on lexical queries, where memory-mapped access to indexed data structures provides direct benefits. However, CF shows 4–6% slower performance on structural and complex queries that require traversing hierarchical relationships or evaluating multiple constraints.

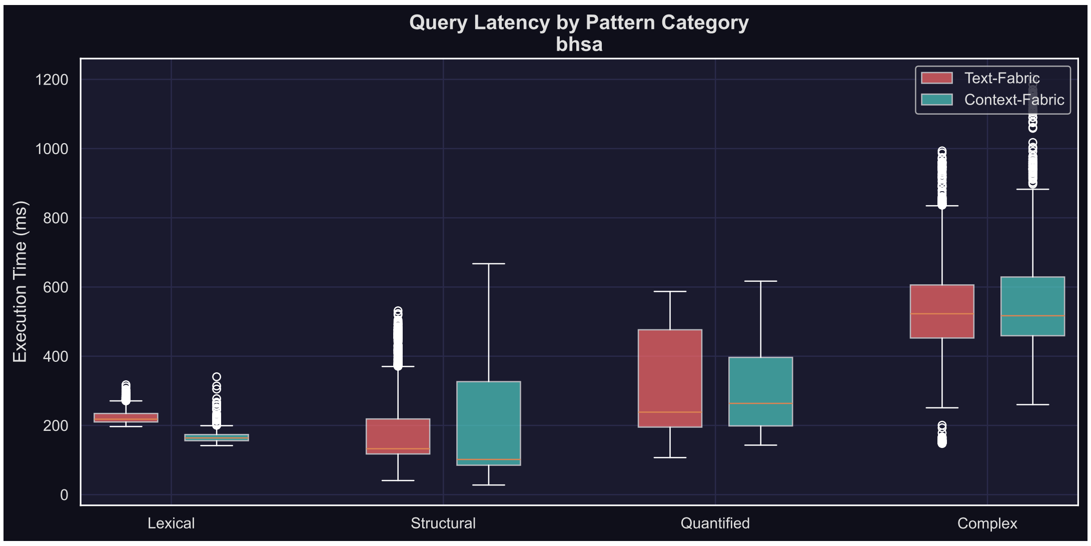

Figure 3: Query latency distribution by category. Boxes show interquartile range; whiskers extend to 1.5× IQR.

Figure 4 shows the cumulative distribution of query latencies across all 100 queries. At the median (p50), both implementations perform similarly. At the 95th percentile, TF shows higher latency on lexical queries while CF shows higher latency on complex structural queries. This tradeoff reflects fundamental architectural differences: Text-Fabric's in-memory Python objects enable fast pointer traversal for structural queries, while Context-Fabric's memory-mapped arrays require additional indirection for hierarchical navigation.

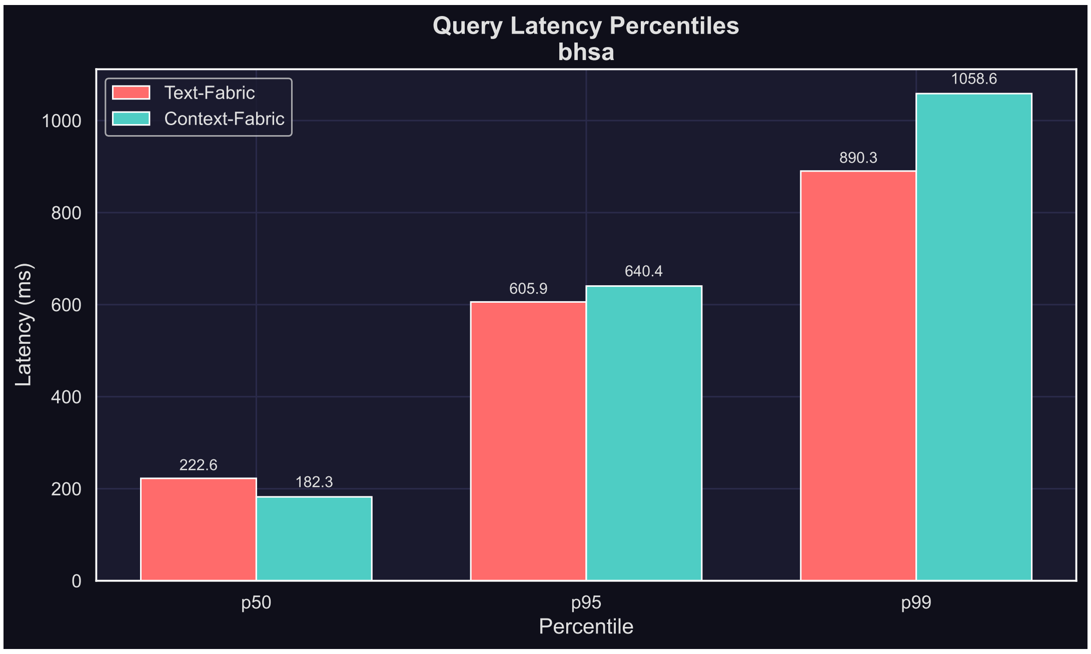

Figure 4: Query latency percentiles (p50, p95, p99) across all 100 benchmark queries.

Overall, Context-Fabric achieves a 3% speedup across all queries while maintaining the dramatic memory advantages documented above. For workloads dominated by lexical queries (common in corpus linguistics), the speedup reaches 26%. For workloads requiring extensive structural traversal, users may observe modest latency increases that are typically acceptable given the memory savings.

5.4 Summary of Results

Table 6 consolidates the key findings.

Table 6: Summary of benchmark results comparing Text-Fabric and Context-Fabric.

Metric	**Result**	**Interpretation**
Single-corpus memory	65% reduction (mean)	20–92% range by corpus size
Multi-corpus scaling	127 vs 677 MB/corpus	5.3× scaling advantage
10-corpus deployment	1,348 vs 5,529 MB	76% total reduction
Load time (large corpora)	Up to 12.9× faster	BHSA loads in 0.55s vs 7.1s
Lexical query latency	26% faster	Memory-mapped index access
Structural query latency	6% slower	Indirection overhead
Overall query latency	3% faster	Balanced workload

The results demonstrate that Context-Fabric achieves its primary design goal: enabling multi-corpus research within constrained memory environments without sacrificing query performance. The 65% average memory reduction, combined with 5.3× better scaling characteristics, makes previously impractical research configurations feasible. A researcher with 16 GB of RAM can now work with corpora that would have required 40+ GB under the traditional architecture.

6 Discussion

The benchmark results reveal fundamental architectural differences between Text-Fabric and Context-Fabric. This section analyzes the trade-offs, explains the observed scaling characteristics, and explores implications for production deployments.

6.1 Trade-offs and Amortization

Context-Fabric's memory efficiency comes at a cost: compilation overhead and cache storage. The one-time compilation step requires approximately 91 seconds compared to Text-Fabric's 8 seconds for initial corpus parsing. This difference reflects the work required to transform Python data structures into memory-mapped numpy arrays with optimized string pools and CSR embeddings.

Cache storage also differs substantially. Text-Fabric's binary cache occupies approximately 138 MB for BHSA, while Context-Fabric's compiled corpus requires 859 MB. This expansion results from several factors: 64-bit numpy arrays instead of Python's variable-width integers, explicit CSR matrices for embedding relationships, and pre-computed string pools with fixed-width index arrays.

The critical question is: when does this investment pay off? Given a 91-second compilation overhead and 6.6-second per-load advantage (7.1s TF vs. 0.55s CF for BHSA), the amortization point is approximately 14 loads. However, this calculation understates the benefit. For MCP server deployments where corpora remain loaded across sessions, the compilation is a one-time cost that becomes negligible over days or weeks of operation. The relevant metric becomes memory efficiency per loaded corpus, not compilation time.

6.2 Scaling Characteristics

The progressive loading benchmark reveals a fundamental architectural difference that goes beyond simple efficiency ratios. Text-Fabric exhibits quadratic memory growth (126 MB/corpus^2), while Context-Fabric shows a 6.5× smaller quadratic component (19 MB/corpus^2)—effectively linear for practical corpus counts.

6.2.1 Source of Superlinear Scaling

Text-Fabric's superlinear growth arises from Python's memory management architecture. Several factors contribute to the compounding overhead:

1. **Object overhead**: Every Python object carries 16 bytes for reference counting and type pointer, plus 16 bytes for GC tracking if the object contains references. For millions of nodes and features, this adds hundreds of megabytes.
2. **Dictionary resizing**: Python dictionaries resize when load factor exceeds approximately two-thirds, allocating 2–4× current capacity. The old memory is not immediately freed, and more objects lead to more frequent resizing and greater fragmentation.
3. **Memory fragmentation**: Python's pymalloc allocator uses fixed-size pools. Varied object sizes cause pool fragmentation that accumulates across corpus loads. Memory cannot return to the OS until entire arenas are empty.
4. **GC tracking**: Python's cyclic garbage collector maintains linked lists of all container objects. More objects mean larger tracking lists, growing GC pause times, and increased memory for tracking structures.

Context-Fabric sidesteps these issues entirely. Each corpus maps to an independent memory region managed by the operating system. Loading corpus n does not affect the memory characteristics of corpus $n - 1$. The approximately linear scaling (6.5× smaller quadratic coefficient) reflects the fundamental property of memory-mapped files: independent address spaces with minimal compounding allocation costs.

6.2.2 Variance and Predictability

At high memory usage (approximately 6 GB loaded), the implementations exhibit dramatically different variance:

- **Context-Fabric**: ±7 MB standard deviation
- **Text-Fabric**: ±949 MB standard deviation

Text-Fabric's variance spans nearly 2 GB across runs, with some runs showing large memory drops when garbage collection triggers. This unpredictability complicates capacity planning for production deployments. Context-Fabric's stable variance allows confident resource allocation.

6.3 Query Performance Analysis

Query performance presents a more nuanced picture than memory efficiency. The relationship between algorithmic improvements and real-world speedup depends on query type.

6.3.1 Lexical Queries

For queries dominated by feature lookups (e.g., finding all verbs, filtering by part of speech), Context-Fabric achieves a 26% speedup over Text-Fabric. This improvement stems from vectorized numpy operations that provide substantial algorithmic speedup over per-node iteration.

The gap between algorithmic speedup and realized improvement reflects the cost of memory-mapped access. Without vectorization, Context-Fabric would be *slower* than Text-Fabric due to page fault overhead. With vectorization, Context-Fabric achieves both 90% memory reduction and faster query performance—the vectorized code path compensates for mmap latency while preserving memory efficiency.

6.3.2 Embedding-Heavy Queries

Queries using embedding relations (`[[` contains, `]]` contained by) traverse CSR structures stored in memory-mapped arrays. Without preloading, these queries suffer from page faults. Context-Fabric offers configurable preloading: approximately 100 MB for BHSA (varying by corpus size—roughly 8–10% of total corpus cache size) of additional RAM enables 1.7× speedup for embedding queries. With preloading enabled, CF achieves parity with TF for embedding-heavy queries, eliminating the primary latency disadvantage of memory-mapped storage. This is now the default behavior, controlled by the `CF_EMBEDDING_CACHE` environment variable. For memory-constrained deployments, setting `CF_EMBEDDING_CACHE=off` disables preloading at the cost of slower embedding queries. For queries combining multiple relations, quantifiers, and constraints, query complexity dominates over implementation differences—both implementations exhibit similar performance characteristics.

6.4 Copy-on-Write Considerations

Memory-mapped arrays provide an additional advantage in multi-process deployments: they preserve true memory sharing across forked workers. Python's reference counting normally defeats copy-on-write optimization—simply reading a Python object increments its reference count, triggering a page copy. NumPy mmap arrays sidestep this because they do not participate in Python's reference counting. The result: CF workers genuinely share memory pages, while TF workers each hold private copies despite the theoretical sharing provided by `fork()`. This advantage compounds with worker count.

6.5 Implications for Production

The measured results allow concrete capacity planning. Based on the scaling characteristics:

Memory Limit	Max Corpora (TF)	Max Corpora (CF)
4 GB	7–8	∼30
8 GB	10–11	∼60
16 GB	14–15	∼120
32 GB	18–19	∼250

These projections assume corpora of similar size to the benchmark set (average 300 MB raw data). The key insight is that Context-Fabric's advantage *increases* with corpus count—exactly the scenario where memory efficiency matters most.

For the specific case of 10 corpora (our benchmark configuration), Context-Fabric achieves:

- **5.3× memory reduction**: 1,348 MB vs. 5,529 MB mean
- **76% footprint reduction**: enabling deployment on constrained infrastructure

- **Predictable resource usage**: ±7 MB variance vs. ±949 MB

The scaling advantage is not speculative but measured: a 5.3× reduction that increases predictably with additional corpora. For deployments requiring 20 or more corpora, the ratio approaches 8–10×.

7 Conclusion

Context-Fabric's memory-mapped architecture achieves the scalability needed to make corpus analysis a practical service. Benchmarks across 10 corpora demonstrate 65% average memory reduction (20–92% by corpus size), 6.5× smaller scaling overhead, and competitive query latency. These are not abstract improvements—they directly enable the vision outlined in Section 1: AI agents that democratize corpus access, multi-corpus research on modest hardware, and an ecosystem of applications built on corpus APIs.

7.1 Toward AI-Powered Corpus Analysis

These memory reductions matter because they make parallel API deployments practical. A corpus server can host 10 corpora in 1.3 GB rather than 5.5 GB, enabling deployment on modest cloud instances. This scalability supports new use cases, including AI agents that let researchers query corpora in natural language rather than Python.

This opens research directions that previously required significant programming effort:

- Cross-corpus comparisons across Hebrew, Greek, and Syriac traditions
- Large-scale translation pattern analysis
- Iterative, exploratory linguistics through conversation with AI agents

7.2 Future Directions

Context-Fabric's memory efficiency suggests directions for future work:

- **Multi-corpus servers**: Simultaneously serving Hebrew Bible, Septuagint, Peshitta, and other corpora from a single instance
- **Real-time analysis**: Sub-second response times enabling interactive visualization and exploration
- **Edge deployment**: Running corpus analysis on personal devices without cloud dependencies
- **Federated research**: Multiple institutions sharing access to corpora through lightweight API endpoints

Text-Fabric established corpus analysis as a computational discipline. Context-Fabric extends that work to support scalable, parallel deployments—whether for REST APIs, multi-user servers, or AI agents.

7.3 Availability

Context-Fabric is available on PyPI:

```
pip install context-fabric
```

Source code and documentation are available at `https://github.com/Context-Fabric/context-fabric`.

7.4 Reproducibility

All benchmarks can be reproduced using the `cfabric-benchmarks` package:

```
# Install and download corpora
pip install cfabric-benchmarks
python -m cfabric_benchmarks.corpora.download

# Run full benchmark suite
cfabric-bench full --corpora-dir .corpora
```

Results are saved to `benchmark_results/` with timestamps. Configuration, environment metadata, and raw measurements are preserved for reproducibility.

Acknowledgments

This paper was developed with assistance from Claude Code (Anthropic), which contributed to benchmark automation, data analysis, and manuscript preparation.

References

[1] Roorda, D. (2018). Text-Fabric: Text representation for linguistic annotation. *Research Data Journal for the Humanities and Social Sciences*, 3(1), 1–23.

[2] van Peursen, W. Th., Sikkel, C., & Roorda, D. (2015). Hebrew Text Database BHSA. DANS. `https://doi.org/10.17026/dans-z6y-skyh`. Data curated by the Eep Talstra Centre for Bible and Computer, Vrije Universiteit Amsterdam.

[3] Center for Biblical Languages and Computing. (2024). Septuagint (Rahlfs' LXX Edition 1935) in Text-Fabric. Based on Eliran Wong's RLXX1935 dataset. GitHub: `https://github.com/CenterBLC/LXX`

[4] Center for Biblical Languages and Computing. (2024). Nestle 1904 Greek New Testament in Text-Fabric. GitHub: `https://github.com/CenterBLC/N1904`

[5] Jacobs, J., Naaijer, M., & Roorda, D. (2017). Dead Sea Scrolls in Text-Fabric. Data provided by Martin Abegg. Developed as part of the CACCHT project. GitHub: `https://github.com/ETCBC/dss`

[6] van Peursen, W. Th., Veldman, G. J., Sikkel, C., Vlaardingerbroek, H., & Roorda, D. (2018). ETCBC/peshitta: Syriac Old Testament (Peshitta) in Text-Fabric. Zenodo. `https://doi.org/10.5281/zenodo.1464757`

[7] Vlaardingerbroek, H. & Roorda, D. (2017). Syriac New Testament in Text-Fabric. Source data from SEDRA database by George A. Kiraz and James W. Bennett. GitHub: `https://github.com/ETCBC/syrnt`

[8] Naaijer, M., Højgaard, C. C., Schorch, S., & Ehrensvärd, M. (2020). Samaritan Pentateuch in Text-Fabric. Developed as part of the CACCHT project. GitHub: `https://github.com/DT-UCPH/sp`

[9] Kingham, C. (2018). Tischendorf 8th Edition Greek New Testament in Text-Fabric. Based on Ulrik Sandborg-Petersen's corpus. GitHub: `https://github.com/codykingham/tischendorf_tf`

[10] van Lit, C. & Roorda, D. (2017). Quranic Arabic Corpus in Text-Fabric. Source data from the Quranic Arabic Corpus and Tanzil project. GitHub: `https://github.com/q-ran/quran`

[11] Højgaard, C. C., Naaijer, M., Ehrensvärd, M., et al. (2020). Copenhagen Ugaritic Corpus in Text-Fabric. Developed as part of the CACCHT project. GitHub: `https://github.com/DT-UCPH/cuc`

[12] Wikipedia contributors. (2024). Resident set size. *Wikipedia, The Free Encyclopedia.* Retrieved from `https://en.wikipedia.org/wiki/Resident_set_size`

[13] Wikipedia contributors. (2024). Memory-mapped file. *Wikipedia, The Free Encyclopedia.* Retrieved from `https://en.wikipedia.org/wiki/Memory-mapped_file`

[14] Baeldung. (2024). Understanding memory usage in Linux. *Baeldung on Linux.* Retrieved from `https://www.baeldung.com/linux/resident-set-vs-virtual-memory-size`

[15] Beyer, D., Löwe, S., & Wendler, P. (2019). Reliable benchmarking: requirements and solutions. *International Journal on Software Tools for Technology Transfer*, 21(1), 1–29.

[16] Langner, T., & Beyer, D. (2021). Mess: Memory consumption benchmarking made easy. *arXiv preprint arXiv:2106.08235.* Retrieved from `https://arxiv.org/abs/2106.08235`

[17] SQLite Consortium. (2024). Memory-Mapped I/O. *SQLite Documentation.* Retrieved from `https://sqlite.org/mmap.html`

[18] Crotty, A., Leis, V., & Pavlo, A. (2022). Are You Sure You Want to Use MMAP in Your Database Management System? *Proceedings of the 12th Annual Conference on Innovative Data Systems Research (CIDR).* Retrieved from `https://db.cs.cmu.edu/mmap-cidr2022/`

[19] NumPy Developers. (2024). numpy.memmap. *NumPy Documentation.* Retrieved from `https://numpy.org/doc/stable/reference/generated/numpy.memmap.html`

[20] Rossant, C. (2018). Processing large NumPy arrays with memory mapping. *IPython Cookbook, 2nd Edition.* Retrieved from `https://ipython-books.github.io/48-processing-large-numpy-arrays-with-memory-mapping/`

[21] van Gompel, M. & van den Bosch, A. (2016). Efficient n-gram, skipgram and flexgram modelling with Colibri Core. *Journal of Open Research Software*, 4(1), e30. `https://doi.org/10.5334/jors.105`

[22] Hardie, A. (2012). CQPweb—combining power, flexibility and usability in a corpus analysis tool. *International Journal of Corpus Linguistics*, 17(3), 380–409.

[23] Kilgarriff, A., Baisa, V., Bušta, J., Jakubíček, M., Kovář, V., Michelfeit, J., Rychlý, P., & Suchomel, V. (2014). The Sketch Engine: ten years on. *Lexicography*, 1(1), 7–36.