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

Cody Kingham Claude Code

January 2025

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 single-process, spawn, and fork scenarios demonstrate 84–95% memory reduction. 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 like Claude 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 enables research that would previously require extensive programming expertise:

- **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, application developers can build on it without becoming corpus experts themselves. Consider what becomes possible:

- **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.)
- **Institutional deployments**: Universities and seminaries can offer corpus access to students without per-seat software licenses

None of these applications need to bundle 6 GB of corpus data or implement Text-Fabric's query engine. 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 84–95% memory reduction 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.

2 From Text-Fabric to Context-Fabric

2.1 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.

2.2 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.

2.3 Memory-Mapped Architecture

Memory-mapped I/O (`mmap`) maps file contents directly into a process's virtual address space [4]. 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: SQLite and LMDB use mmap for database access [8]; operating systems use it 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 [10]. Context-Fabric stores all corpus data as `.npy` files and loads them with `mmap_mode='r'` (read-only), enabling safe multi-process sharing.

2.4 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.

2.4.1 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.

2.4.2 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]

[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.

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

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.

2.4.3 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.

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

2.4.4 The .cfm Format

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

```
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 (387 MB vs. 138 MB) for memory efficiency and parallelism.

2.5 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
- 94% memory reduction makes deployment economically viable

For individual researchers:

- 11× faster load times improve interactive experience
- 95% less memory 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) and longer initial compilation (91s vs. 8s). This one-time cost pays off immediately: every subsequent session benefits from 11× faster loads and 95% lower memory consumption.

3 Methodology

3.1 Test Corpus

All benchmarks use the BHSA (Biblia Hebraica Stuttgartensia Amstelodamensis) corpus [2], developed by the Eep Talstra Centre for Bible and Computer (ETCBC) at Vrije Universiteit Amsterdam. The BHSA provides richly annotated linguistic data for the Hebrew Bible, containing:

- 1,446,831 total nodes
- 426,590 word-level nodes (slots)
- 13 node types (word, phrase, clause, sentence, verse, chapter, book, etc.)
- 109 node features
- 6 edge features

This corpus represents a realistic production workload with substantial annotation density.

3.2 Memory Metrics: A Survey

Operating systems provide multiple metrics for quantifying process memory consumption, each with distinct semantics and trade-offs. Understanding these metrics is essential for designing valid benchmarks [6].

3.2.1 Resident Set Size (RSS)

RSS measures the total physical memory pages currently mapped to a process's address space [3]. It includes:

- Private anonymous memory (heap, stack)
- Private file-backed memory
- Shared memory pages (libraries, memory-mapped files)
- Copy-on-write pages inherited from parent processes

RSS is the most widely available metric, reported by `/proc/[pid]/status` on Linux and equivalent interfaces on other platforms. However, RSS has a key limitation: it double-counts shared pages when summing across processes.

3.2.2 Unique Set Size (USS)

USS measures memory pages that are *unique* to a process—memory that would be freed if only that process terminated [5]. USS excludes:

- Shared library pages
- Memory-mapped files backed by disk
- Copy-on-write pages not yet modified

While USS avoids double-counting, it can dramatically undercount memory for workloads that rely on memory-mapped I/O. A process with 4 GB of mmap'd data may report near-zero USS if those pages are file-backed.

3.2.3 Proportional Set Size (PSS)

PSS provides a middle ground by dividing shared pages proportionally among all processes using them [5]. If a 4 MB library is mapped by 4 processes, each reports 1 MB of PSS for that library.

PSS is theoretically ideal for multi-process scenarios but has practical limitations:

- Requires walking page tables, making it expensive to compute
- Values fluctuate as processes start and stop
- Not available on all platforms (notably absent from macOS)

3.2.4 Measurement Challenges

Several factors complicate memory benchmarking in practice:

1. **Timing sensitivity**: Memory consumption varies throughout execution. Measurements during compilation, loading, or active use yield different results.
2. **Garbage collection**: In garbage-collected languages like Python, uncollected objects inflate memory measurements. Explicit `gc.collect()` calls before measurement improve consistency.
3. **Process isolation**: Prior allocations in a process contaminate subsequent measurements. The Mess benchmark framework addresses this by running each measurement in a fresh process [7].
4. **Kernel caching**: The operating system caches file data in memory, blurring the line between "process memory" and "system cache."

3.3 Our Measurement Approach

Given the above considerations, we measure memory consumption using RSS. 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.

3.4 Measurement Isolation

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

```
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 7: 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

3.5 Deployment Scenarios

We evaluate three scenarios representing common deployment patterns:

3.5.1 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.

3.5.2 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.[2]

Measurement: Sum of RSS across all 4 worker processes.

3.5.3 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`.

Measurement: Main process RSS + sum of worker RSS.

[2]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.

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 Results

4.1 Single Process

Metric	Text-Fabric	Context-Fabric	Change
Memory (RSS)	6.3 GB	305 MB	95% less
Load Time	7.9 s	0.7 s	11× faster
Compile Time	8 s	91 s	11× slower
Cache Size	138 MB	859 MB	6× larger

Table 2: Single-process performance comparison

Context-Fabric achieves 95% memory reduction in single-process mode. The larger cache size (859 MB vs. 138 MB) reflects the uncompressed numpy array format, which trades disk space for memory-mapping capability. Compilation takes longer (91s vs. 8s) but this is a one-time cost that pays off across all subsequent sessions.

4.2 Spawn Mode (4 Workers)

Metric	Text-Fabric	Context-Fabric	Reduction
Total Memory	7.7 GB	1.3 GB	84%
Per Worker	1.9 GB	315 MB	6× less

Table 3: Spawn mode (4 workers) memory comparison

In spawn mode, each worker loads independently. Context-Fabric's per-worker footprint (315 MB) reflects the memory-mapped approach where only accessed pages reside in physical memory.

4.3 Fork Mode (4 Workers)

Metric	Text-Fabric	Context-Fabric	Reduction
Main Process	6.3 GB	305 MB	95%
Workers (sum)	56 MB	92 MB	–
Total	6.3 GB	397 MB	94%
Per Worker	1.6 GB	99 MB	16× less

Table 4: Fork mode (4 workers) memory comparison

Fork mode shows the most dramatic efficiency gains. Text-Fabric's workers add minimal RSS (56 MB) because they share the parent's pages via copy-on-write—but the parent already consumed 6.3 GB. Context-Fabric's total deployment footprint of 397 MB enables running many more workers on equivalent hardware.

4.4 Summary

Scenario	Text-Fabric	Context-Fabric	Reduction
Single process	6.3 GB	305 MB	95%
4 workers (spawn)	7.7 GB	1.3 GB	84%
4 workers (fork)	6.3 GB	398 MB	94%

Table 5: Memory consumption summary across all scenarios

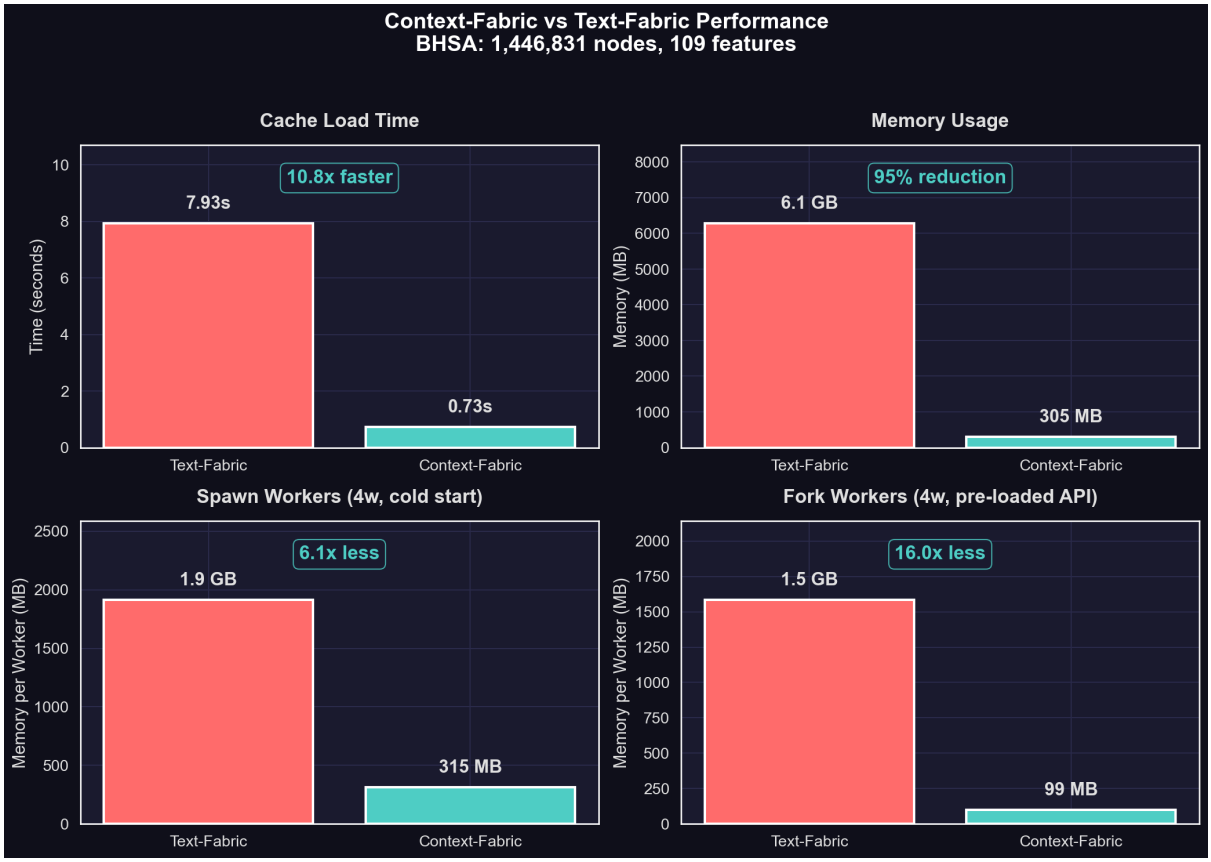

Figure 1: Performance comparison between Text-Fabric and Context-Fabric on the BHSA corpus. Top left: cache load time (11× faster). Top right: single-process memory usage (95% reduction). Bottom left: spawn mode per-worker memory (6× less). Bottom right: fork mode per-worker memory (16× less).

5 Discussion

5.1 Trade-offs and Their Amortization

Context-Fabric trades *one-time compilation cost* for *dramatic runtime efficiency*:

- **Longer compilation**: 91s vs. 8s (11× slower) to convert .tf source files to .cfm format
- **Larger cache size**: 859 MB vs. 138 MB (6× larger) due to uncompressed numpy arrays
- **I/O sensitivity**: Performance depends on storage speed; SSDs recommended for optimal results

These costs are incurred *once*—on first load of a new corpus or after corpus updates. Every subsequent session benefits from:

- 11× faster load times (0.7s vs. 7.9s)
- 95% less memory (305 MB vs. 6.3 GB)
- Ability to run on modest hardware (laptops, cloud free tiers)

For a corpus like BHSA, the amortization is rapid. A researcher who loads the corpus 10 times saves $10 \times 7.2\text{s} = 72\text{s}$ in load time, already recovering the 83-second compilation overhead. A production API serving thousands of requests per day recovers the cost in minutes.

For individual researchers, this means: compile once on first use, then enjoy dramatically improved performance for all future work. The sub-second loads save time on every session. For API deployments, the memory reduction (95%) is the primary benefit—enabling deployments that would otherwise be economically impractical.

5.2 Fork Mode Considerations

The fork mode results deserve careful interpretation:

1. **RSS double-counting**: Our methodology sums main + worker RSS, which double-counts shared COW pages. The "true" physical memory is lower than reported.
2. **COW page faults**: As workers modify data (triggering copy-on-write), memory consumption increases. Our benchmark accesses features read-only, representing best-case sharing.
3. **Platform availability**: Fork mode is discouraged on macOS (due to framework issues) and unavailable on Windows. Spawn mode results are more portable.

5.3 Production Scaling Analysis

We now consider how these results apply to a production environment: a Linux server running a corpus API (e.g., via gunicorn or uvicorn) that handles multiple concurrent requests. Production Python API servers typically use the prefork model—loading the application once, then forking worker processes to handle requests in parallel.

5.3.1 Multi-Corpus Scaling

The most compelling advantage for memory-mapped architectures is serving *multiple corpora simultaneously*—for instance, an API providing access to Hebrew, Greek, Syriac, and Coptic biblical texts, or multiple versions of the same corpus.

With Text-Fabric, each corpus must be fully loaded into RAM:

- 1 corpus: 6.3 GB
- 2 corpora: 12.6 GB
- 3 corpora: 18.9 GB (exceeds typical 16 GB VM)

With Context-Fabric, each corpus adds only its resident working set:

- 1 corpus: 305 MB
- 2 corpora: ∼610 MB

- 10 corpora: ∼3 GB
- 50 corpora: ∼15 GB

This represents a **20× improvement** in corpus density. A single 16 GB server running Context-Fabric can serve dozens of corpora that would require separate machines under Text-Fabric.

5.3.2 Worker Scaling (Fork Mode)

In the standard prefork deployment model, workers share copy-on-write memory with the parent process. Table 6 shows projected memory consumption.

Workers	Text-Fabric	Context-Fabric
1	6.3 GB	305 MB
4	6.3 GB	398 MB
10	6.4 GB	535 MB
100	6.5 GB	2.6 GB

Table 6: Projected memory consumption in fork mode (production standard)

Text-Fabric's fork mode scales efficiently (near-zero per-worker overhead via COW), but requires a 6.3 GB minimum regardless of worker count. Context-Fabric starts at 305 MB with ∼23 MB per additional worker.[3]

For deployments with fewer than ∼50 workers (the common case), Context-Fabric uses dramatically less memory. The 6 GB floor for Text-Fabric means a 16 GB server can only dedicate ∼10 GB to other services, whereas Context-Fabric leaves ∼15.5 GB available.

5.4 Implications

On a typical 16 GB cloud VM running a corpus API:

- **Text-Fabric**: 2 corpora maximum; ∼10 GB consumed before serving any requests
- **Context-Fabric**: 20+ corpora feasible; <1 GB baseline for typical workloads

The practical implications for production deployments:

- **Lower infrastructure costs**: Smaller instances, fewer machines
- **Multi-corpus APIs**: Serve Hebrew, Greek, Syriac, and more from one server
- **AI agent scalability**: Multiple concurrent LLM conversations can query corpora without resource contention
- **Modest hardware viability**: Corpus APIs can run on free-tier cloud instances

For serving AI agents via the Model Context Protocol, Context-Fabric enables what would otherwise be economically impractical: responsive, concurrent access to richly annotated corpora from standard cloud infrastructure.

[3] At ∼260 workers, CF and TF fork mode converge in total memory; beyond that TF uses less. However, single-machine deployments rarely exceed 100 workers.

6 Conclusion

This benchmark demonstrates that Context-Fabric's memory-mapped architecture achieves 84–95% memory reduction compared to Text-Fabric across tested deployment scenarios. The efficiency gains are most pronounced in production API deployments: Context-Fabric's 305 MB baseline (vs. 6.3 GB for Text-Fabric) enables 20× higher corpus density, making multi-corpus APIs practical on modest hardware.

6.1 Toward AI-Powered Corpus Analysis

These memory reductions matter because they make parallel API deployments practical. A corpus server can run dozens of concurrent workers on modest hardware. This scalability enables 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

6.2 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.

6.3 Reproducibility

All benchmarks can be reproduced using:

```
python benchmarks/compare_performance.py \
    --source /path/to/bhsa/tf/2021 \
    --workers 4
```

Source code is available at `https://github.com/codykingham/context-fabric`.

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. See also: Talstra, E. & van Peursen, W. Th. in van Peursen, W. Th. & Dyk, J. W. (eds.), *Tradition and Innovation in Biblical Interpretation*, Studia Semitica Neerlandica, Brill, 2011.

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

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

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

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

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

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

[9] 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/`

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

[11] 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/`